Segmenting Large Traces of Inter-Process Communication with a focus on High Performance Computing Systems

Luay Alawneh, Abdelwahab Hamou-Lhadj, Jameleddine Hassine
Department of Software Engineering
Jordan University of Science and Technology, Irbid, Jordan
lmalawneh@just.edu.jo
Software Behaviour Analysis (SBA) Research Lab
Department of Electrical and Computer Engineering
Concordia University, Montréal, QC, Canada
abdelw@ece.concordia.ca
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
jhassine@kfupm.edu.sa

Abstract

The understanding of the interactions among processes of a High Performance Computing (HPC) system can be made easier if trace analysis is used. Traces, however, can be quite large, making it difficult to analyze their content unless some abstraction is provided. This paper presents a novel trace abstraction approach that aims to facilitate the analysis of large execution traces generated from HPC applications. Our approach allows automatic segmentation of large traces into smaller and meaningful clusters that reflect the various execution phases of the traced scenarios. Our approach is based on the application of information theory principles to the analysis of sequences of communication patterns extracted from traces of HPC systems. This work is inspired by recent studies in the field of bioinformatics where several techniques have been proposed to segment DNA sequences into homogeneous sub-domains, where each sub-domain exhibits a certain degree of internal homogeneity. Trace segments can be used in a number of applications such as recovering high-level views of the system behavior and program understanding. We demonstrate the usefulness of our approach by applying it to different traces of hundreds of millions of events, generated from two HPC systems.

Keywords—Dynamic analysis, Trace abstraction and analysis, inter-process communication traces, High performance computing systems, Software maintenance, Program comprehension

1. Introduction

High Performance Computing (HPC) systems are used to solve complex computational problems in a variety of domains including medical image processing, financial trading, bioscience, and data security. These systems run on a large number of processors and can process in parallel quadrillions of operations per second [Newman 14]. HPC has become a popular solution for building powerful applications due to the emergence of multi-core and cloud computing platforms.

Despite the recent advances in the design of HPC systems, there are still challenges related to how to analyze these systems. A typical application involves a large number of processes. Understanding the way these processes interact with each other is a tedious and complex task [Maghraoui 05]. To address this, existing work (e.g., [Noeth 09][Geimer 09]) has been devoted to techniques and tools that enable the analysis of inter-process communication traces. Trace analysis tools support panoply of features, among which the most important one is the ability to extract patterns of inter-process communication from large traces. This way, a software engineer can validate whether or not the system behaves according to predefined communication patterns. Trace patterns can also be seen as a way to build abstractions from large traces, allowing software engineers to examine only patterns of interest, instead of going through the entire trace content (which is practically impossible).

Pattern recognition techniques, however, remain limited as to how much abstraction they provide. This is because of the large size of typical traces (millions of events). There are just too many patterns to analyze. Besides, patterns are often extracted without context. Consider, for instance, an execution trace that is generated from running a machine learning algorithm with large datasets on multiple processors. This trace is expected to contain typical machine learning steps including data preprocessing, training models, validation, testing, etc. A software engineer who wants to understand how a model is trained does not need to see the other parts of the trace. Extracting patterns for the entire trace will still not reveal where the model is trained. In this scenario, it would be useful to know where each of the phases is manifested in the trace. A software engineer can then explore each phase separately. One way to achieve this is to instrument the system in such a way that the program's phases are clearly annotated. The drawback with this approach is that it assumes some knowledge of the system under study.

In this paper, we present a trace abstraction approach that segments a trace of HPC events into smaller and meaningful clusters that represent the execution phases that compose the traced scenario. We define an execution phase as part of a trace where a particular program computation is invoked. That is, an execution phase groups cohesive program elements.

The trace segmentation approach we propose in this paper can be used primarily by software analysts who want to understand the content of a trace with the objective of completing a given maintenance task (such as enhancing an existing feature or detecting the causes of a fault). Our approach can also be used to verify if the behavior of the system complies with the intended behavior during design. We also anticipate that our approach can be useful for analysts who do not maintain the system, but still need to report on the way it behaves. These analysts can benefit from our approach by, for example, reporting on the various phases that are involved in the trace, the number of communication patterns each phase contains, etc. Moreover, we can always infer some statistical data from our approach such as the number of processes involved in a certain pattern, the number of messages exchanged, and so on.

Trace segmentation is a relatively new topic. To our knowledge, there are only a few studies that focus on segmenting inter-process communication traces into execution phases (e.g., [Casas 07][González 13][Chetsa 13]). These studies use performance data to distinguish among the various phases of a program's execution. In our view, these techniques are designed for performance analysis and not for program comprehension. Our approach is designed to allow an analyst to understand how a particular scenario is implemented.

Our trace segmentation approach is inspired by the work of Li et al. [Li 02] in bioinformatics, more particularly the area of DNA processing. The authors introduced a new technique for segmenting DNA sequences into homogeneous sub-domains; each has a certain degree of internal homogeneity (or similarity). DNA segments can be used in a number of applications such as detecting the presence of known genes for medical purposes, identifying new genes and associations with diseases, comparing gene structures of various species, etc. By analogy, we can view a trace of inter-process communication as a large sequence of events, just like a DNA sequence. By segmenting a trace, we mean identifying clusters of events that contribute to the implementation of the same execution phases (sub-domains). This way, browsing a trace would no longer necessitate the examination of low-level trace events, but instead, we can view the trace as a flow of execution phases. Our trace segmentation approach involves two main steps. First, we detect inter-process communication patterns using pattern detection techniques. The second step consists of dividing the sequence of extracted communication patterns into dense homogenous clusters that indicate the presence of execution phases. This is achieved using information theory concepts such as Shannon entropy [Roberts 05] and the Jensen-Shannon Divergence measure [Grosse 02].

We also focus in this paper on Single Program Multiple Data (SPMD) HPC applications. However, our approach should be readily extendible to other inter-process communication models.

The rest of the paper is organized as follows. In Section 2, we present background information on traces from HPC systems, followed by related work. In Section 3, we present our approach and describe the algorithms and techniques used in the detection process. In Section 5, we show the effectiveness of our approach by applying it to traces of hundreds of millions of events, generated from two subject systems. We discuss threats to validity in Section 6 and conclude our work in Section 7.

2. Background and Related Work

2.1. A Brief Overview of HPC

High Performance Computing (HPC) relies on parallel computing in order to solve complex computation-intensive scientific problems. Parallel computing decomposes the problem into several sub-problems that run on various computational units to complete in an acceptable time period. Typically, the computational units need to collaborate in order to accomplish a specific task. Parallel computing utilizes two main programming paradigms, which are the shared memory and distributed memory paradigms. In shared memory, processes collaborate by sharing the same memory space. On the other hand, a distributed memory application consists of many processes running on different distributed processors that interact using the message passing model. These parallel programs may consist of thousands of processes that are coordinating to solve a specific large scale problem. In this paper, we focus on distributed memory applications with a specific interest in programs that use the MPI[1] (Message Passing Interface), a standard for writing parallel applications using message passing, for inter-process communication.

MPI provides point-to-point, collective, and one-sided types of communications. Point-to-point operations support both blocking and non-blocking modes. Point-to-point communication occurs between a pair of MPI processes in the program. The sending process posts a send operation that contains the destination, the data, the data type signature, the tag, and the communicator (a predefined group of MPI processes). The receiving process, on its side, should post a receive operation that matches the incoming message based on its data type signature, the tag value, and the source process (sending process). The receiving process uses the tag value to identify the incoming message. However, a process may post a receive operation that can accept a message originating from any process in the communicator with a wildcard tag value.

MPI collective communication includes a set of operations for exchanging information among the group of processes in a communicator. MPI assumes that the processes in the communicator must perform the same collective operations in the same order. MPI enforces the synchronization among the communicating processes using the *barrier* operation. MPI collective operations are implemented using point-to-point

[1]http://www.mpi-forum.org

operations. However, the collective communication must be in blocking mode only in order to guarantee the synchronization among the communicating processes. Therefore, all processes must post collective operations that exactly match the size and the data type signature of the exchanged data. The new version of MPI (i.e., MPI 3.0[1]) provides non-blocking collective operations. In this study, we work with MPI 2.0. However, we are planning to target MPI 3.0 as part of future work.

An MPI trace consists of events collected from all the running processes in the program. A trace from each process contains user-defined routine call events and MPI operation call events. Inter-process communication traces are those events that correspond to the MPI point-to-point and collective communication operation calls.

2.2. Techniques for detecting communication patterns in traces of HPC systems

The detection of communication patterns in traces of HPC systems has been the focus of many trace analysis research studies in the context of HPC systems. This is because HPC systems tend to follow specific communication patterns throughout their execution. These communication patterns provide a view of the way processes interact with each other. In addition, trace patterns can be used to recover communication topologies.

Preissl et al. [Preissl 08, Preissl 10] presented an approach for detecting communication patterns in MPI traces using compressed suffix trees. Their approach combines both dynamic and static analysis techniques in the detection process. They used MPI seed events to look for areas in the code where communication patterns could occur. The authors demonstrated the usefulness of their approach by showing how the detected communication patterns could be used in improving the overall program performance. Using static analysis is very challenging in the context of parallel systems. Static analysis requires building and storing a static model of the system, which adds complexity to the analysis process. Our communication patterns detection approach depends solely on dynamic analysis and does not require prior knowledge of the source code.

Isaacs et al. [Isaacs 15] presented a new approach to extract the trace’s logical structure by ordering the MPI events based on their happened-before relationships. In their approach, the authors cluster processes and the communication patterns using visualization techniques. Their approach can then be used to identify delayed operations with respect to their peer ones on the other processes. The authors applied their approach on a trace generated from running 64 processes. This approach relies on the visualization of

[1]http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

groups of events to form a global communication pattern, which may work for small number of processes but may be difficult to scale to large configurations.

Knüpfer et al. [Knüpfer 06] presented an approach to remove contiguous patterns from traces of HPC systems based on the compressed complete call graph (cCCG). The cCCG references all call sequences that are equivalent with respect to a call structure and temporal behavior to achieve an improved trace compression scheme. This algorithm does not target the detection of communication patterns. It only detects tandem repeats of events on each process trace separately. Our approach goes two steps further by first detecting communication patterns and then creating smaller segments that reflect the various phases of the execution.

Trahay et al. [Trahay 15] proposed a trace summarization technique to identify points of interest that should be examined first in the trace. The technique uses a variation of the LZW compression technique to detect sequences of repeating events in the trace. The technique generates a new view of the trace as loops and groups of events as opposed to the traditional sequential representation of traces. The approach relies on filtering techniques to eliminate duplicate sequences so as to simplify the localization of points of interests. Our approach differs from this approach as it detects the communication patterns in the entire trace and not only within a given process trace. Furthermore, we identify the computational phases based on the homogeneity of the sub-segments of the list of communication patterns.

Kunz and Seuren [Kunz 97] proposed a communication pattern matching approach that is based on finite state automata. The algorithm determines the longest process pattern in the input communication pattern and finds its occurrences in the trace. The algorithm then starts the construction of the communication pattern by matching the events of the longest pattern with the partner events on the other process traces. This approach is only concerned with finding patterns that match a predefined input pattern, whereas our approach aims to detect all communication patterns in a trace.

Köckerbauer et al. [Köckerbauer 10] used a pattern matching technique to facilitate the debugging of large message passing parallel programs by searching the trace file for predefined communication patterns. The engineer provides the communication pattern description using a custom syntax which is then translated to abstract syntax trees. The generated ASTs are then scaled up to the number of processes in the trace (or a target subset of the processes). Similar to our approach, they run the pattern matching algorithm on each process trace separately. The resulting matching patterns on each process trace are then merged to get the matching communication pattern. In their approach, Köckerbauer et al. look for exact and similar patterns using a hash-based search algorithm. Therefore, the matching communication pattern may be a variation of the user’s specification. Our segmentation process uses a pattern detection algorithm that does not require prior knowledge of the communication patterns used in the program.

Wolf et al. proposed a pattern matching technique for detecting patterns of inefficient behavior based on wait states as part of a HPC performance analysis tool known as KOJAK [Wolf 03][Wolf 07]. They search the trace for known patterns in order to identify inefficient behavior. The tool then classifies the patterns based on the time spent in communication. This work is different from the one presented in this paper since it only looks for patterns of inefficient behavior resulting from processes in long wait states and it does not aim at segmenting the trace into execution phases. Also, the authors assume that knowledge of the communication topology is available in order to display the patterns, which is not always the case. In our study, we detect the communication behavior found in the trace as communication patterns.

2.3. Techniques for segmenting traces into execution phases

The objective of trace segmentation techniques is to divide a trace into coherent segments (that are referred to as execution phases). There exist some studies that target the identification of computational phases in MPI programs. González et al. [González 09] presented a density-based clustering approach to detect the computational phases in SPMD message passing applications. They applied the DBSCAN algorithm to group different CPU bursts, gathered from performance hardware counters provided by modern processors, to identify the different computational phases in the program execution. A CPU burst is observed as a computation region between two consecutive communications. A burst is characterized by the duration and the set of performance counters. However, performance data should be used with thresholds that require a high degree of fine-tuning to obtain accurate computational phases. The authors extended their DBSCAN-based approach by applying Aggregative Cluster Refinement to automate the detection process and overcome the shortcomings of the DBSCAN algorithm [González 12] and [González 13]. The new approach combines clustering with multiple Sequence alignment to refine the quality of the extracted computational structure. Our approach works on execution traces that contain user-defined as well as MPI events and relies on the communication patterns detected in the trace. Our phase identification approach segments the trace based on the homogeneity of the communication patterns in each region.

Aguilar et al. [Aguilar 15] presented an on-line approach to detect the loop nesting structure of MPI applications at runtime using event flow graphs without explicit source code instrumentation. These detected loops are only the ones that contain MPI communication events. The flow graphs provide a compressed representation of MPI traces supported by the iterative structure of MPI parallel applications [Aguilar 14]. They utilize their approach to gather statistical information of the program execution by only collecting a small number of iterations in order to reduce the overhead of the gathered data on the running program. They detect the loop once the program is stable and goes in a long iterative loop (i.e. the Solve phase). Detecting when the program goes in a stable state requires user involvement. An advantage of their approach is that it does not require the whole trace to be collected and that the application structure can still be used to compute the post-mortem statistics. Our approach differs from this work as it needs the whole trace to detect communication patterns and then utilizes the list of patterns to identify the different phases in the program execution.

Casas et al. [Casas 07] proposed an automatic phase detection approach based on signal processing techniques to identify the main phases (initialization, computation, and output) in the traces of MPI applications. The algorithm depends on the iterative behavior of MPI programs in order to identify the different phases in the program. They categorize the phases based on the frequency of their iterative behavior where in the computational phase most of the parallel iterations exist. The authors extended their work to detect the sub-phases in the computational phase [Casas 10]. They used several metrics based on inter-process communications (signals) and CPU computing bursts to mark a computational phase change. Our approach goes one step further as the user is provided with the distinct communication patterns that are forming each phase in the trace.

A similar approach to the one found in [Casas 10] is proposed by Chetsa et al. [Chetsa 13] where the authors presented a phase detection approach based on execution vectors where a vector includes a set of values such as hardware performance counters, network communications, disk I/O values. Their approach detects a new phase when the Manhattan distance between successive vectors exceeds a predefined threshold.

Cornelissen et al. [Cornelissen 09] proposed to visualize the call relations between the functions from different classes and packages into a matrix that potentially shows the emergence of dense groups that can be qualified as trace segments. The authors, however, did not provide an automatic segmentation of the trace data. It is up to the user to interpret the visual rendering of method calls. Reiss et al. **[**Reiss 05] proposed a tool called Jive to visualize the behavior of a Java program using statistical information about the system's behavior in predefined time intervals. The phases are visualized in a user interface. The problem of this approach is that it uses profiling information (not sequences of calls). At the end, the user is only provided with some statistics about each phase.

The techniques that use traces (usually routine call traces) generated from single-process applications include the work of Pirzadeh el al. [Pirzadeh 11a] and [Pirzadeh 11b]. The authors proposed a novel phase detection approach, inspired by the way the human perception system groups lines and dots into shapes and objects. Their approach includes several methods that could automatically group routine-call events into dense elements that formed computational phases. Their work, however, targets traces of routine calls. In the future, we intend to study how their approach can be applied in the context of inter-process communication traces.

2.4. Discussion

Although pattern recognition techniques have been shown to be useful, they can only exhibit how processes interact with others without any context. It is up to the analyst to figure out which parts of the trace implement a specific computation of the traced scenario, the problem addressed in this paper.

Techniques for segmenting inter-process communication traces rely on performance data such as CPU bursts and statistics on a program's execution. In this paper, we focus on traces of call to user-defined functions and MPI operations. These traces can be used to understand how a particular scenario is implemented. We propose to reduce the size of traces by dividing them into smaller and more manageable segments; each contains a set of cohesive trace elements. This way, a software engineer can analyze each segment individually or combine them if need be. A trace analysis tool can implement the trace segmentation process, presented in this paper, to provide the ability for software engineers to explore segments of traces without having to worry about the other parts of the trace. By exploration, we include the common tasks that a software engineer would normally do if presented by the entire trace, such as extracting patterns, searching, viewing properties, etc.

The *novelty* of this work lies in the fact that, to our knowledge, this is the first time that a technique for segmenting traces of inter-process communication based on communication patterns is proposed. The use of DNA processing methods to design an effective trace segmentation approach is also *novel*. We tested our approach on two HPC systems. This research is *significant* because it shows evidence that we can build powerful trace segmentation approaches for traces of inter-process communication.

3. The Trace Segmentation Approach

Figure 1 shows our trace segmentation approach. The first step is to generate the trace of the specified scenario from the target test system. The trace consists of multiple *process traces* ($T_1 \dots T_n$) where each process trace contains both MPI as well as user-defined function events. The next step is to detect communication patterns from the process traces. For this, we improve an algorithm that we presented in [Alawneh 11] (the new algorithm is discussed in more detail in the next subsections). The extracted sequence of communication patterns is then input to the phase identification component. In the phase detection step, we look for changes in communication patterns in the trace and group those that show a certain degree of homogeneity. The result is a binary tree, which we refer to as a segmentation tree, where the root node is the whole input sequence (coarse-grained) and the leaf nodes are the lowest level segments (fine-grained). The final step consists of identifying the beginning and ending of each phase using the segmentation tree. A phase may contain multiple segments. These steps are presented in more detail in the next subsections.

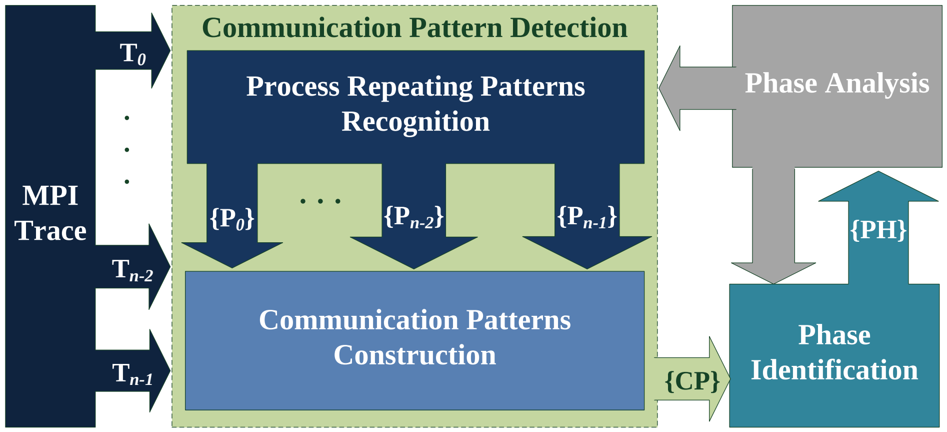


Figure 1. Approach for segmenting traces of inter-process communication

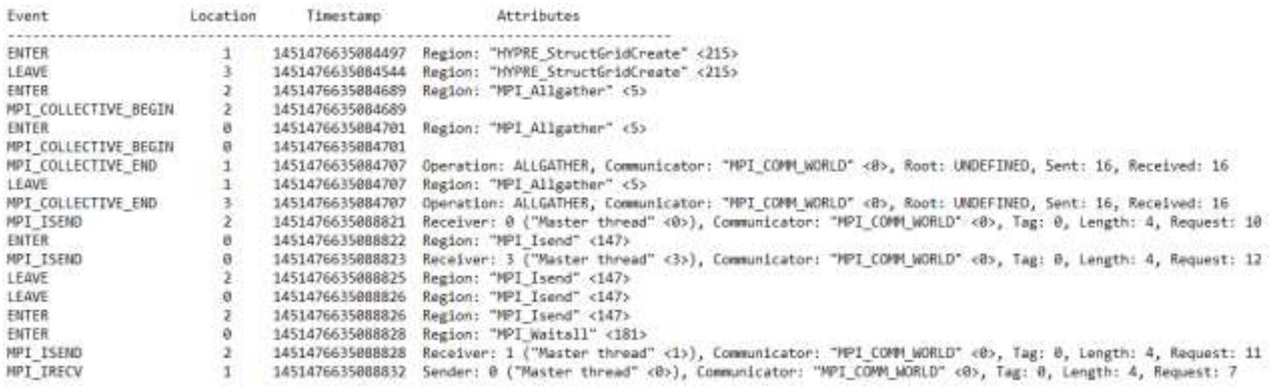
3.1. Trace Generation

The process of generating execution traces is usually done through instrumentation of the code. This consists of inserting probes in the code in places of interest, recompile the code, and run the system. Instrumentation is performed automatically using tools such as VampirTrace[1], TAU[2] and Score-P[3]. Instrumentation can also be at the OS and binary levels using tools such as PIN[4] and LTTng[5].

To collect traces, an analyst needs to exercise the instrumented system by executing the scenario of interest. In this paper, we instrument the entire system. This is because we do not assume that analysts know which parts of the system to analyze. We use the Score-P tool to collect traces in OTF2[6] format. The tool generates the entire trace as a set of files where each file contains a trace for each process. A process trace contains a sequence of events. There are two types of events. The first one consists of calls to user-defined functions and MPI operations (we need to have entry and leave probes for each call). The second type consists of communication events. All events have timestamps associated with them. Figure 2 shows an excerpt of a trace from four processes generated in OTF2 format. In this figure, HYPRE_StructGridCreate is a user-defined function. MPI_Allgather is an MPI operation, which is a collective operation.

[1] http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace
[2] http://www.cs.uoregon.edu/Research/tau
[3] http://www.vi-hps.org/projects/score-p/
[4] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[5] http://www.lttng.org
[6] https://silc.zih.tu-dresden.de/otf2-current/html/

MPI_ISEND and MPI_IRECV are point-to-point operations. The “Location” column shows the process id. The “Attributes” column shows different values that are used by the operation such as a tag in case of point-to-point operations. The number within the angle brackets represents the code (id) of the function.



```
Event                   Location     Timestamp              Attributes
--------------------------------------------------------------------------------
ENTER                      1      1451476635084497  Region: "HYPRE_StructGridCreate" <215>
LEAVE                      3      1451476635084544  Region: "HYPRE_StructGridCreate" <215>
ENTER                      2      1451476635084689  Region: "MPI_Allgather" <5>
MPI_COLLECTIVE_BEGIN       2      1451476635084689
ENTER                      0      1451476635084701  Region: "MPI_Allgather" <5>
MPI_COLLECTIVE_BEGIN       0      1451476635084701
MPI_COLLECTIVE_END         1      1451476635084707  Operation: ALLGATHER, Communicator: "MPI_COMM_WORLD" <0>, Root: UNDEFINED, Sent: 16, Received: 16
LEAVE                      1      1451476635084707  Region: "MPI_Allgather" <5>
MPI_COLLECTIVE_END         3      1451476635084707  Operation: ALLGATHER, Communicator: "MPI_COMM_WORLD" <0>, Root: UNDEFINED, Sent: 16, Received: 16
MPI_ISEND                  2      1451476635088821  Receiver: 0 ("Master thread" <0>), Communicator: "MPI_COMM_WORLD" <0>, Tag: 0, Length: 4, Request: 10
ENTER                      0      1451476635088822  Region: "MPI_Isend" <147>
MPI_ISEND                  0      1451476635088823  Receiver: 3 ("Master thread" <3>), Communicator: "MPI_COMM_WORLD" <0>, Tag: 0, Length: 4, Request: 12
LEAVE                      2      1451476635088825  Region: "MPI_Isend" <147>
LEAVE                      0      1451476635088826  Region: "MPI_Isend" <147>
ENTER                      2      1451476635088826  Region: "MPI_Isend" <147>
ENTER                      0      1451476635088828  Region: "MPI_Waitall" <181>
MPI_ISEND                  2      1451476635088828  Receiver: 1 ("Master thread" <1>), Communicator: "MPI_COMM_WORLD" <0>, Tag: 0, Length: 4, Request: 11
MPI_IRECV                  1      1451476635088832  Sender: 0 ("Master thread" <0>), Communicator: "MPI_COMM_WORLD" <0>, Tag: 0, Length: 4, Request: 7
```

Figure 2. An example of a process trace in OTF2 format

Scientific problems have several input parameters that determine their size and complexity. HPC systems that solve these kinds of problems should be configured by providing the number of processes and the appropriate process topology. The number of processes is constrained by the problem complexity and the hardware capabilities.

3.2. Communication Pattern Detection

As mentioned earlier, our trace segmentation technique takes as input a trace of sequences of communication patterns instead of raw events. To detect patterns, we improve an earlier pattern detection algorithm significantly that we presented in [Alawneh 11]. The algorithm uses a two-step process. First, we detect patterns in each process trace separately. Then we use the patterns from each process trace to construct the final communication patterns. We illustrate these steps in the following sections.

We explain the pattern detection technique through a running example using the sample trace depicted in Figure 3. The trace consists of four processes, where each process trace is represented as a routine-call tree. The processes collaborate in functions F1 to F3 to accomplish a certain task. This example uses MPI_Send and MPI_Recv operations represented using the S and R symbols respectively. Every send operation should have a matching receive operation on the partner process. For example, P3 sends a message to P4 represented as S4 (send to P4) in F1 and P4 posts a receive operation as R3 (receive from P3) in F1. Similarly, P2 sends a message to P1 in F2 and P2 receives this message by posting R2 in F2.

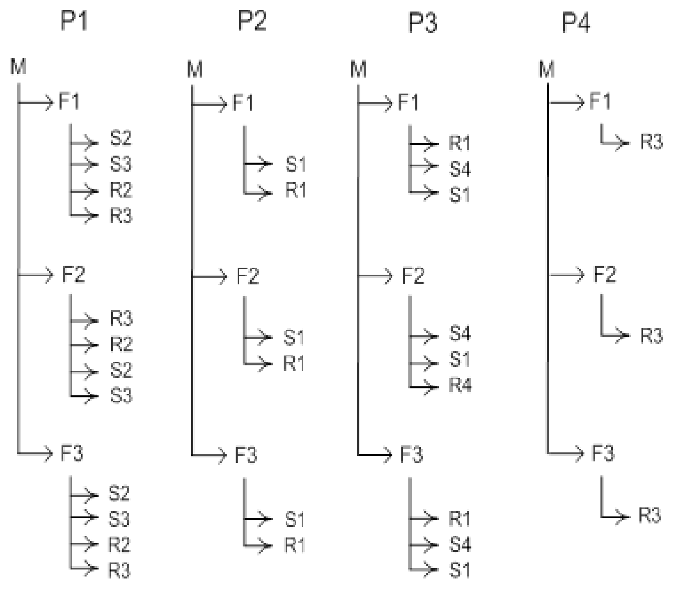


Figure 3. Traces of four Inter-communicating Parallel Processes

Step 1: Detection of per-process patterns

In this step, we iterate on each process trace separately in order to detect the patterns of MPI communication events. A pattern is a maximal repeat that may not be extended to the left and to the right in the sequence that it occurs in. More formally, given a string *S* of length *l*, a maximal repeat in *S* is a tuple *(p1, p2, l)* such that:

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

$$S[p_1 + l] \neq S[p_2 + l] \text{ and } S[p_1 - 1] \neq S[p_2 - 1]$$

Using the concept of n-gram extraction, the algorithm starts by building the list of repeating bi-grams in a process trace along with their positions in the trace. The repeating bi-grams will eventually grow in size, using the algorithm, to be the final detected n-grams. The events of an n-gram (i.e. pattern) may only appear within the same user-defined function call. That is, we do not consider patterns that are formed across functions. In other words, we use functions as a context. This derives from our empirical observation that meaningful patterns are the ones that are defined within specific functions [Alawneh 14]. We found that this limitation was primarily due to the fact that we viewed a trace as a mere stream of MPI communication events without considering where these events appear in the program. We observed that patterns rarely appear outside the boundaries of user-defined functions.

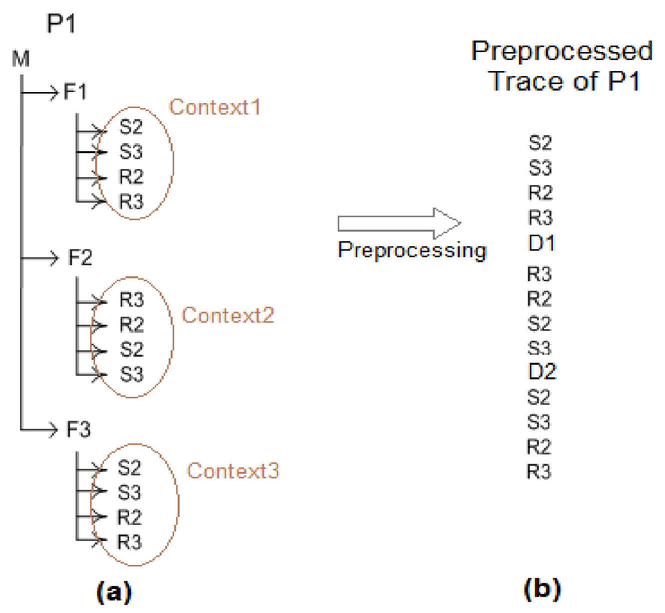


Figure 4. (a) An example of a process trace with contextual information; (b) Delimiters added during the preprocessing of the trace to specify contextual information

Figure 4(a) shows the trace of P1 from Figure 3 with contextual information. This context will be used when looking for trace patterns by treating each MPI call. To make the pattern detection algorithm take the context into account, we simply add delimiters to the trace whenever a context switch occurs (i.e., another function is called). Figure 4(b) shows a preprocessed trace where D1 and D2 (two delimiters are added). The pattern detection algorithm (as we will show later in the paper) does not cross these delimiters when looking for trace patterns.

Figure 5(a) shows a fictive process trace with multiple nesting levels. The corresponding preprocessed trace is shown in Figure 5(b). As we can see, this trace has three contexts due to the call of user-defined function F2 by F1. Delimiters D1 and D2 are added to show the contexts.

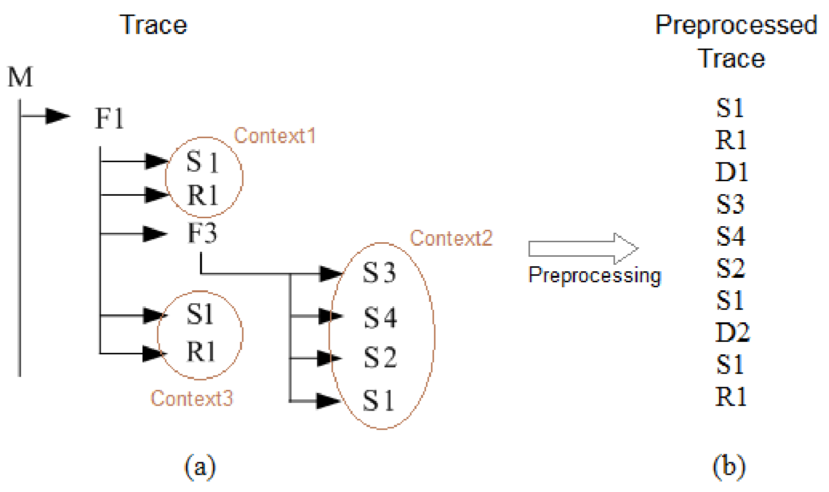


Figure 5. A trace with multiple nesting levels (a) and its corresponding preprocessed trace with delimiters added to show the contexts

We recognize that determining a pattern's context is not that straightforward. However, in most benchmark systems that we have analyzed we observed that by simply considering each function as a unique context, we could enhance the pattern detection process.

Algorithm 1 shows the steps for detecting patterns in a process trace. The algorithm takes each preprocessed process trace as input and returns a set of maximal repeats found in each trace. The first step is the extraction of the repeating bi-grams from each process trace. These bi-grams are used as the start point of patterns for the detection of maximal repeats. Algorithm 1 has two main loops, the outer loop determines the number of passes required to complete the detection process. The *new* variable indicates that a new pattern is detected. Therefore, the stopping criterion for Algorithm 1 is when there are no new patterns added to the list of detected patterns in the last pass. The inner loop reads one event at a time and appends it to the current *repeat* at Line 11. Once a bi-gram is constructed, the algorithm checks if it is in the bi-gram table. If the bi-gram is in the table, then the algorithm reads the next event and appends it to the current *repeat*, otherwise the next bi-gram is read from the trace. A new repeat is added to the pattern list (Line 24). In Line 27, we also add the starting position of *repeat* to the set of starting positions of its corresponding *pattern*, depicted by the collection {positions}. If the previous pattern is not a repeating pattern, then the algorithm clears the current pattern and starts reading from the last position of *previous* (Lines 20 and 21). The number of passes (the outer loop) is directly related to the length of repeats and their frequencies. The more frequent a pattern is, the faster it will be detected.

#	**Maximal Repeats Detection:** trace size = *n* {*patterns*}: list of patterns (initially repeating bi-grams)

```
1   new ← true
2   while (new)
3     new ← false, length ← 0, position ← 0, repeat ← Ø
4     for I ← 0 TO n - 1
5       event = trace [ I ]
6       if event is delimiter then
7         length ← 0, repeat ← Ø → GOTO line 4
8       if repeat is Ø then
9          position ← I
10         repeat ← event
11       else previous ← repeat, repeat || event
12       length ← length + 1
13       if length ≤ 2 then GOTO line 4
14       pattern ← patterns [ repeat ]
15       if pattern is null ∧ length is 2 then
16         I ← I – 1
17         length ← 0, repeat ← Ø → GOTO line 4
18       end if
19       if length is 2 then GOTO line 4
20       if frequency_previous is 1 then
21         I ← I – 2, length ← 0, repeat ← Ø → GOTO line 4
22         if pattern is null then
23           new ← true
24           {patterns} ← pattern
25           length ← 0, repeat ← Ø, I ← I – 1
26         end if
27         {positions}_pattern ← position
28     I: end of for loop
29  N: end of for loop
```

Algorithm 1. Pattern detection algorithm

To illustrate the execution of Algorithm 1, we use the preprocessed trace sample of Figure 4(b). We also add the position of each event to ease the description of the algorithm. S2 appears at position 0, S3 at position 1, etc. The new sequence with the positions is shown in Figure 6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
S2	S3	R2	R3	D1	R3	R2	S2	S3	D2	S2	S3	R2	R3

Figure 6. A simplified representation of preprocessed trace of Figure 4(b).

The maximal repeats detection algorithm starts by extracting the list of repeating bi-grams with their positions and frequency as shown below.

n-gram	**Position**	**Frequency**
S2.S3	0, 7, 10	3
S3.R2	1, 11	2
R2.R3	2, 12	2

The algorithm continues by reading the bi-grams from the input trace. If the bi-gram already exists, it will append the next event to the bi-gram. Similarly, if a 3-gram exists in the patterns list, the algorithm will append the next event, forming a 4-gram. The algorithm takes two passes to detect the longest pattern S2.S3.R2.R3 (4-gram). However, the third pass is needed since the *new* is assigned a *true* value in the second pass when the S3.S4.S2.S1 pattern was detected. The table below shows the pattern list after completing the three passes in Algorithm 1.

n-gram	**Position**	**Frequency**	**Inner Repeat**
S2.S3	0, 7, 10	3	Yes (positions 0 & 10)
S3.R2	1, 11	2	Yes
R2.R3	2, 12	2	Yes
S2.S3.R2	0, 10	2	Yes
S2.S3.R2.R3	0, 10	2	No

The next step is to remove inner patterns. We achieve this by iterating through the list of patterns, resulting from the maximal repeats detection process, and removing the instances of patterns that are part of larger patterns. For example, instances of pattern S2.S3 at positions 0 and 10 are part of pattern S2.S3.R2.R3. Similarly, S3.R2 at positions 1 and 11 are parts of S2.S3.R2.R3. We remove these instances from the list of patterns. The result of removing inner patterns is shown below:

n-gram	**Position**	**Frequency**
S2.S3	7	1
S2.S3.R2.R3	0, 10	2

The final step is to look for single events that were not part of any repeating patterns. These may include point-to-point and collective communications. These events may be part of a communication pattern but could not be detected in Algorithm 1 since they appear separately due to the delimiters that were added in the preprocessing step. These events will be linked to the patterns that are created and will be part of the communication pattern construction algorithm (the next step of the algorithm). The sequence extracted from Figure 4 shows that events R3 and R2 at positions 5 and 6 can be added as new patterns that are occurring only once to the list of detected patterns. The final list of patterns is as follows:

n-gram	**Position**	**Frequency**
S2.S3	7	1
S2.S3.R2.R3	0,10	2
R3	5	1
R2	6	1

In case of single collective communication events, they will also be added as repeats and will be used in the next step to form collective communication patterns.

Step 2: Formation of final communication patterns

To construct the final communication patterns, we simply iterate through the per-process patterns, detected in the previous step, and match the MPI communication events 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 pattern S2.S3.R2.R3 found in Process P1 are R1 in Process P2 (matches S2 in P1), R1 in P3 (matches S3 in P1), S1 in P2 (matches R2 in P1), and finally S1 in P3 which matches R3 in P1. Therefore, the matching patterns for *p1* are S1.R1 on P2, and R1.S4.S1 on P3. We use timestamp information, tag, data type signature, and the communicator in order to identify the matching events. The formation process continues until all MPI communication events are matched. It should be noted that event R3 on Process P4 will be included in the final communication pattern since it matches event S4 (S4 is part of pattern R1.S4.S1) on process P3. The final communication pattern of the fictive trace in Figure 4 is shown in Figure 7.

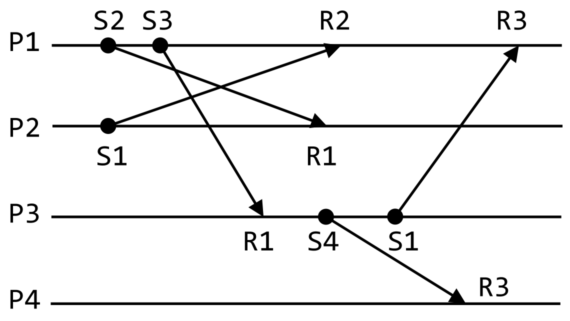


Figure 7. Detected communication pattern

3.3. Detection of trace segments

The proposed trace segmentation approach is based on studies for the analysis of DNA sequences in the field of bioinformatics. Our study for detecting the different computational phases in MPI programs is derived from the algorithm presented by Li et al. [Li 02] to identify the homogeneous sub-domains in a DNA sequence recursively. The segmentation algorithm, a divide-and-conquer algorithm where a problem is subdivided into smaller problems recursively, is proposed by Cormen et al. in [Cormen 90]. It relies on information theory concepts where Shannon entropy [Shannon 48] [Gray 11] and the Jensen-Shannon divergence measures [Grosse 02] are used to guide the segmentation process.

We applied the aforementioned recursive approach to the segmentation of a message passing trace by using the sequence of communication patterns detected in the previous section; where the MPI trace is first abstracted as a sequence of communication patterns (each pattern is represented by a symbol). The segmentation process first measures the degree of heterogeneity of the whole sequence, which results in two segments. For this, Shannon entropy is used [Gray 11] to measure the level of randomness of information in a sequence. A sequence with low entropy (homogeneous) is achieved when all the symbols appear with similar probabilities. On the contrary, sequences with high randomness will have high entropy which means that the data is heterogeneous and could be divided into more homogenous segments. The Shannon entropy *H* of a sequence *S* of length *N* with *k* different symbols is measured using equation 1 [Gray 11].

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

where N_j is the frequency of symbol *j* in *S*.

The first step in segmenting a sequence is to measure its Shannon entropy. Then, we need to locate the position in the sequence where the highest level of heterogeneity occurs. This process is performed based on the following steps in order to select the two new subsequences:

1. For every position *i* in *S*, we calculate the entropy of the left subsequence S_l and the right subsequence S_r from position *i*:

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

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

   where N_j^l is the frequency of symbol *j* in S_l and N_j^r is the frequency of symbol *j* in S_r. Note that the symbol at position *i* is in S_l, and that S_l and S_r may not be empty.

2. The similarity between S_l and S_r is measured using the Jensen-Shannon Divergence (D_{JS}) [Gray 11]. A higher D_{JS} value means more heterogeneity between the two subsequences:

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

From the (N – 2) subsequence pairs, the pair with the highest D_{JS} value contains the new detected phases. The two subsequences may also be segmented further into more subsequences. This segmentation process is applied recursively to the new subsequences until a certain stopping criterion is met.

Li et al. [Li 02] used the model selection framework in order to decide when to stop the segmentation algorithm. There are two models, M_1 and M_2, that we use to determine the split position in a sequence S. M_1 is characterized by the entire sequence S while M_2 is characterized by the two sub-segments S_l and S_r. To further segment S, we need to find a model $M2$ at the border between the underfitting and the overfitting models. The Bayesian Information Criterion (*BIC*) [Akaike 78] is used to select the model that fits the data well with the number of parameters [Li 02] as follows:

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

where L is the maximum likelihood of the model, N is the sequence length, and K represents the number of free parameters in the two models. K is computed as $(k_l + k_r + 1 - k)$ where k_l, k_r, and k refer to the number of distinct parameters in S_l, S_r and S respectively. In order to show the ability of the *BIC* measure to determine the stopping criterion, we need to measure the likelihood for S using:

$$L1(S) = \prod_{j=1}^{k} p_j^{N_j} \quad (6)$$

where p_j is the probability of symbol j to appear in S and is calculated as N_j/N. Consequently, the *log*-likelihood of M_1 is measured using:

$$\log L1(S) = \sum_{j=1}^{k} N_j \log \frac{N_j}{N} \quad (7)$$

From equation 7, it is clear that the *log*-likelihood for S is equal to $(-NH)$ where H is the entropy value for S (refer to equation 1). Additionally, the likelihood for M_2, represented by S_l and S_r, is measured by:

$$L2(S_l, S_r, N_l) = \prod_{j=1}^{k_l} (p_j^l)^{N_j^l} \prod_{j=1}^{k_r} (p_j^r)^{N_j^r} \quad (8)$$

where p^l_j is equal to N^l_j/N and p^r_j is equal to N^r_j/N. Hence, the *log*-likelihood is calculated as:

$$\log L2(S_l, S_r, N_l) = \sum_{j=1}^{k_l} N_j^l \log \frac{N_j^l}{N} + \sum_{j=1}^{k_r} N_j^r \log \frac{N_j^r}{N} \quad (9)$$

Clearly, *log* L2 is equal to $-N_l H_l - N_r H_r$. Thus, the relative increase of the log-likelihood of the two models $\log(L2/L1)$ is equal to $NH - (N_l H_l + N_r H_r)$ which, according to equation 4, is equal to ND_{JS}. Thus, the maximum likelihood is measured at the point with the highest ND_{JS} value. This means that the *BIC* value should be close to zero (i.e. $\Delta BIC < 0$) for segmentation to continue. By substituting $N\hat{D}_{JS}$ ($\hat{D}_{JS}$ is the maximum D_{JS} value) for L in equation 5, we get:

$$2N\hat{D}_{JS} > \log(N)K \quad (10)$$

Thus, the sequence could be further segmented if $\hat{D}_{JS}$ is greater than *log(N)K/2N*. Li et al. [Li 02] defined a new segmentation strength value *s* which is determined by the relative increase of $2ND_{JS}$ from the *BIC* threshold as:

$$s = \frac{2N\hat{D}_{JS} - \log(N)K}{\log(N)K} \qquad (11)$$

A positive segmentation strength value will have the same segmentation effect as when the D_{JS} value is greater than *log(N)K/2N*. Therefore, both measures could be used as the stopping criterion for the recursive segmentation algorithm. In our study, we will use the positive segmentation strength value as the indicator whether to stop or continue the segmentation process. The user can also adjust the value of *s* to be greater than zero which will result in a smaller number of subsequences. On the other hand, specifying a zero threshold value will increase the depth of the segmentation tree which means a larger number of subsequences.

The segmentation algorithm results in a hierarchy of segments, which can be viewed as a binary tree (that we call segmentation tree) where the original sequence forms the root node and the leaves are the detected segments. Each parent node (i.e. segment) in the tree is split into two segments at the point with the maximum D_{JS} value in the parent node. The accuracy of the presented technique is at the cost of its relatively slower computational time since it requires many passes through the data to measure the D_{JS} value for each pair of subsequences in each hierarchy in the tree.

Figure 8 shows a typical segmentation tree (the table shows the values of the various parameters of the recursive algorithm). In this example, S_0 contains the sequence of communication patterns of the entire trace. In the example of Figure 8 no further segmentation is possible.

S	p_s	p_e	*s*	D_{JS}	p_c	*P*
S_0	1	16	2.62	0.91	8	-
S_1	1	8	2.94	0.74	2	S_0
S_2	9	16	4.08	0.95	12	S_0
S_3	1	2	2	0.75	1	S_1
S_4	3	8	-0.66	0.14	4	S_1
S_5	1	1	0	0	1	S_3
S_6	2	2	0	0	2	S_3
S_7	9	12	2.22	0.81	10	S_2
S_8	13	16	1.75	0.69	15	S_2
S_9	9	10	-1	0	10	S_7
S_{10}	11	12	-1	0	12	S_7
S_{11}	13	15	-1	0	15	S_8
S_{12}	16	16	0	0	16	S_8

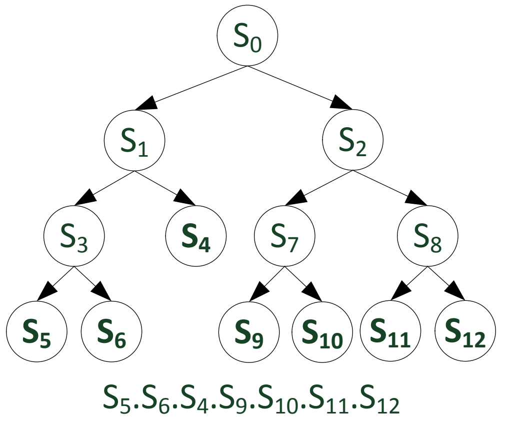


Figure 8. An example of a segmentation tree resulting from applying the algorithm

3.4. Phase Identification and Analysis

Once the tree is constructed, we need to identify the boundaries of each phase. This is because a phase may contain multiple segments. To achieve this, we added a threshold *t* to indicate where to cut the tree. This threshold would provide flexibility to the analyst to vary the level of granularity of the final phases. The threshold *t* is set manually by the analyst and it varies from 0 to the height of the segmentation tree (the depth of the root node). A tool that supports our approach should offer the flexibility to modify the threshold *t*.

On one hand, setting *t* to 0 would result in one large segment (S_0 in the case of Figure 8) that contains all the communication patterns. This would be rarely desirable. On the other hand, setting *t* to the height of the tree would result in fine-grained segments (S_5, S_6, S_4, S_9, S_{10}, S_{11}, S_{12}) and phases. The analyst can vary *t* to decide on the level of granularity.

Once the set of segments is identified, we need to map them to the original execution trace (that contains also the user-defined functions) in order to identify the user-defined functions that invoke the MPI events that form the patterns in the segments. Note that since we are only focusing on SPMD programs, all the processes execute the same functions and follow the same flow. This means that we only need to visit one process trace to be able to identify the user-functions that encompass the segments.

We start by checking the function-call tree to identify all the functions in which the patterns of the first segments are invoked. For example, assume the list of final segments from Figure 8 is: S_5, S_6, S_4, S_9, S_{10}, S_{11}, S_{12}. This means that *t* is set to the height of the tree. We first identify the functions that invoke all the MPI events that form the patterns in S_5. This would result in a call subtree. Assume this subtree is rooted at node *n*. The next step is to check the patterns in S_6 (the next segment right after S_5). If S_6 is also rooted at n1 or any child node of *n* then we put S_5 and S_6 as part of the same execution phase. If this is not the case then S_6 suggests the start of a new phase. This is because a new phase should reflect the fact that some new functions are emerging while the previous ones (belonging to the previous phase) are disappearing in the trace, meaning that new computations are taking place.

Once the phases are identified, we check their validity by referring to the source code or any available documentation. One way to adjust the result is to run the phase detection again by varying the threshold *t*. In practice, the tool that supports our approach should be flexible enough to allow the analyst to modify *t* dynamically. Determining the appropriate value of *t* in advance is a difficult problem because this parameter may vary from one system to another.

4. Evaluation

In this section, we demonstrate the effectiveness of our approach by applying it to 15 different traces generated from the SMG2000[1] industrial HPC system, and two traces from the BT NAS[2] benchmark. We used the Score-P tool to generate the traces from the two target systems. Score-P generates the traces in OTF2 format. Our experiments were performed on a small cluster of 10 nodes (Intel Core i7-3770, 3.4GHz, RAM 12GB), connected using a standard IEEE 802.11 network. We developed our techniques using Java. The question that this study aims to answer is: Can a recursive algorithm be used to effectively segment an inter-process communication trace into execution phases?

4.1. SMG 2000

SMG2000 is an SPMD parallel semi-coarsening multi-grid solver for linear systems arising from finite difference, finite volume, or finite element discretization of the diffusion equation on logically rectangular grids. It performs a large number of non-nearest-neighbor point-to-point communication operations [SMG2000].

The execution of SMG2000 involves three main phases which are Initialization, Setup and Solve [Tiwari 11]. The Initialization phase is responsible for the creation and initialization of the grid objects and has main routines such as HYPRE_StructSMGCreate, and HYPRE_StructMatrixAssemble. The HYPRE_StructSMGSetup routine marks the Setup phase and the HYPRE_StructSMGSolve routine marks the Solve phase. We use this information in the validation of the detected phases.

Table 1 shows the results of running our approach on 12 scenarios from SMG2000 with 1x1x1 input size. The rows of Table 1 are grouped into three categories where each category represents the execution of the system with the same process topology-base. The number of processes range from 16 to 1024. The 16x16x4 scenario generates a very large number of events, over 598 million events.

Table 1 also provides timing information. This time does not include the trace preprocessing step (adding delimiters and keeping only MPI events). The preprocessing time can reach up to one hour for large traces. We see in Table 1 that the communication behavior is directly related to the process topology. For example, when the number of processes is 64, the 3D process topology (4x4x4) scenario shows more complex behavior compared to the same number of processes in the 2D topology (8x8x1). The time it took to detect the process patterns in 4x4x4 was 2,378 milliseconds and in case of 8x8x1 it was only 611 milliseconds. This is related to the number of communications among the processes in each scenario. The same behavior can be seen when comparing the number of process patterns for the other cases that have the

[1]http//www.llnl.gov/asc/purple/benchmarks/limited/smg/
[2]NAS Parallel Benchmarks, http://www.nas.nasa.gov

same number of processes. In situations where the number of processes is larger, but with a fewer number of grids, the communication behavior is less complex. For example, for the two 512-process scenarios, the 16x16x2 topology has a simpler communication behavior when compared to the 8x8x8 topology.

Table 1. SMG2000 Scenarios (problem size: 1x1x1), $\sum ev$: total number of events, $\sum mpi_{ev}$ total number of MPI calls, $\sum MSG$: total number of exchanged messages, $\sum cv_p$: number of collective operations per process, $\sum PP$: total number of distinct process patterns on all processes (excludes single event patterns), PP_d: total process patterns detection execution time, $\sum CP$: number of distinct communication patterns, CP_d: communication patterns construction time, CPL_l: Communication Patterns List Length, PH_d: phase detection time

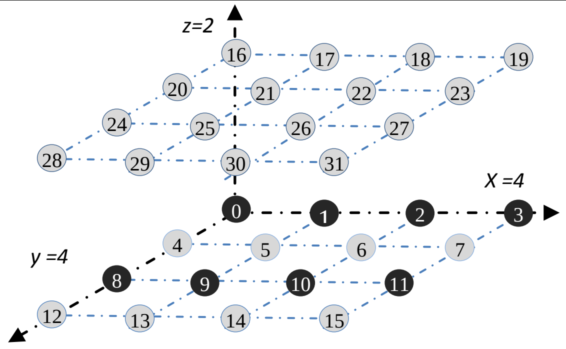
np	***Topology***	$\sum ev$	$\sum mpi_{ev}$	$\sum msg$	$\sum cv_p$	$\sum PP$	PP_d ***(ms)***	$\sum CP$	CPL_l	CP_d ***(ms)***	PH_d ***(ms)***
16	4 x 4 x 1	641,646	13,479	7,132	12	223	127	49	101	14	19
32	4 x 4 x 2	4,027,420	92,275	55,468	13	668	534	145	712	54	124
48	4 x 4 x 3	11,104,714	264,471	170,584	14	1,455	1,866	252	1,626	169	310
64	4 x 4 x 4	14,603,190	347,823	225,756	14	1,906	2,378	318	1,877	184	397
64	8 x 8 x 1	4,241,270	89,739	49,692	12	1,210	611	191	299	91	38
128	8 x 8 x 2	26,342,588	577,771	345,712	13	3,638	2,551	589	1,959	399	459
256	8 x 8 x 4	103,400,644	2,276,539	1,479,184	15	10,035	22,836	1148	5,433	2,217	2,948
512	8 x 8 x 8	309,733,410	6,507,339	4,305,444	16	24,522	129,455	2899	12,644	49,229	13,802
256	16 x 16 x 1	24,394,422	495,163	265,908	12	5,680	1895	712	937	973	180
512	16 x 16 x 2	151,226,556	3,063,087	1,727,100	13	16,550	14,623	2290	6,321	4,813	4,392
768	16 x 16 x 3	418,407,354	8,542,783	5,169,608	14	34,310	117,904	3388	11,473	23,977	13,645
1024	16 x 16 x 4	598,699,396	11,866,079	7,311,836	15	42,088	193,741	3906	15,373	41,801	19,016

Table 2 provides four different scenarios of the SMG2000 4x4x4 topology with varying input problem size. It should be noted that the first entry in Table 2 is the same as the fourth entry in Table 1. It is clear that the problem size has a direct impact on the communication behavior among the processes in the program. It is apparent that problem size (1x1x1) has simpler communication behavior when compared to the other scenarios. For example, the execution of problem size (2x2x2) has 3 times more the number of exchanged point-to-point messages than that of the (1x1x1) case. Furthermore, problem size 3x3x2 has more point-to-point communications (approximately three times larger) when compared to the 8x8x2 process topology in Table 1 with problem size of 1x1x1.

Table 2. SMG2000 Scenarios (Topology: 4x4x4, Varying Input Problem Size)

np	***Problem Size***	$\sum ev$	$\sum mpi_{ev}$	$\sum msg$	$\sum cv_p$	$\sum PP$	PP_d ***(ms)***	$\sum CP$	CPL_l	CP_d ***(ms)***	PH_d ***(ms)***
64	1 x 1 x 1	14,603,190	347,823	225,756	14	1,906	2,378	318	1,877	184	397
64	2 x 2 x 2	45,151,470	1,172,727	767,564	16	3,851	9,678	491	5,736	408	2,377
64	3 x 2 x 2	52,621,296	1,331,775	851,476	16	4,090	11,170	539	6,033	652	2,838
64	3 x 3 x 2	67,778,872	1,722,049	1,100,424	16	5,322	19,969	615	6,694	977	5,494

Figure 9 shows a textual representation of a communication pattern that is repeated 16 times in the 4x4x2 scenario. This example shows that the communication pattern is occurring among the processes in the same grid (black nodes). The pattern shows that only processes P0, P1, P2, P3, P8, P9, P10 and P11 are involved in communication. Process P0 communicates with processes P1, P2, P8, P9, and P10 while processes P9 and P10 communicate with all the processes involved in the pattern.



```
P0 : S1.S2.S8.S9.S10.R1.R2.R8.R9.R10
P1 : S0.S2.S3.S8.S9.S10.S11.R0.R2.R3.R8.R9.R10.R11
P2 : S0.S1.S3.S8.S9.S10.S11.R0.R1.R3.R8.R9.R10.R11
P3 : S1.S2.S9.S10.S11.R1.R2.R9.R10.R11
P8 : S0.S1.S2.S9.S10.R0.R1.R2.R9.R10
P9 : S0.S1.S2.S3.S8.S10.S11.R0.R1.R2.R3.R8.R10.R11
P10: S0.S1.S2.S3.S8.S9.S11.R0.R1.R2.R3.R8.R9.R11
P11: S1.S2.S3.S9.S10.R1.R2.R3.R9.R10
```

Figure 9. Communication patterns in 4x4x2 topology (P0 is Process 0, S1: Send to P1, R1: Recv from P1)

Another interesting pattern that is repeated 20 times in the trace involves all the processes in the two grids where each process communicates with its first and second neighbors in each direction in the two grids. For example, process P1 communicates with processes P2, P4, P5, P6, P8, P9, P10, P16, P17, P18, P20, P21, P22, P24, P25, P26 and process P10 communicates with all the other processes in the two grids. Another pattern that is repeated 10 times in the trace involves only two processes from the two adjacent grids which are processes 10 and 26. All the scenarios have the same repeating collective communication pattern ALLREDUCE.ALLREDUCE. This collective communication occurs three times in the program execution and marks the end of each phase.

S	***p_s***	***p_e***	***L***	***S***	***D_{JS}***	***p_c***	**P**
S_0	1	15,373	15,373	23.68	0.83	5,601	-
S_1	1	5,601	5,601	1.53	0.75	2,814	S_0
S_2	5,602	15,373	9,772	-0.77	0.05	6,257	S_0
S_3	1	2,814	2,814	2.99	0.89	1,081	S_1
S_4	2,815	5,601	2,787	15.77	0.9	4,054	S_1
S_5	1	1,081	1,081	215.04	1.01	540	S_3
S_6	1,082	2,814	1,733	11.49	0.97	1,945	S_3
S_7	2,815	4,054	1,240	22.06	0.96	3,517	S_4
S_8	4,055	5,601	1,547	0.09	0.53	5,008	S_4

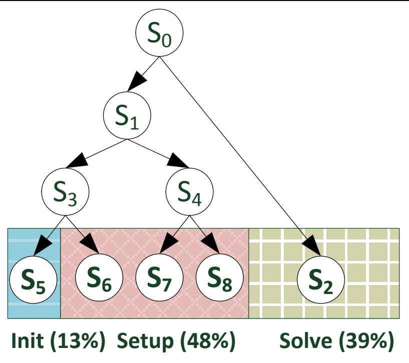
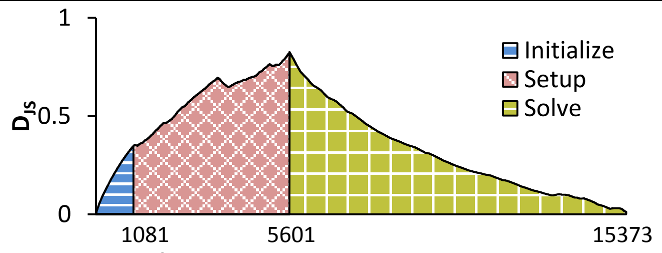


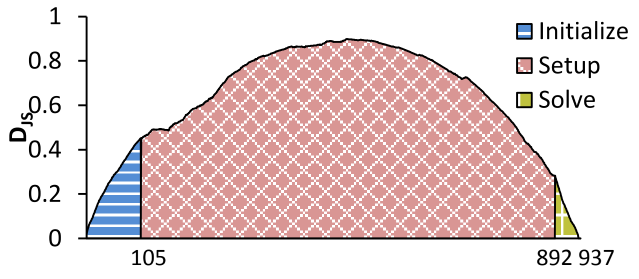
Figure 10. Recursive Segmentation for Communication Patterns Sequence of 16x16x4 Scenario (P_s: start position, P_e: end position, l: length, D_{JS}: Jensen-Shannon Divergence, p_c: cutting position of max divergence, s: Segmentation Strength, P: parent node, t = 3)

Figure 10 shows the segmentation results when applied to the communication pattern sequence generated from the 16x16x4 process topology example in Table 1. The communication pattern list size is 15,373. In this example, we used t= 3 to cut the segmentation tree. By investigating the execution trace, S_1 includes the initialization and setup phases and S_2 represents the Solve phase. S_2 is homogeneous with a negative semgentation strength, which cannot be further segmented using our technique. The pattern at position 5,602 is an ALLREDUCE collective operation which is the first pattern in the Solve phase. The trace has three ALLREDUCE collective patterns. We already explained how a single collective operation could be detected as a communication pattern. Furthermore, the trace has three occurences of the ALLREDUCE.ALLREDUCE pattern. Each occurrence marks the end of a main phase (Initialization, Setup and Solve). The positions of the ALLREDUCE.ALLREDUCE pattern are 1081, 5601, and 15373.

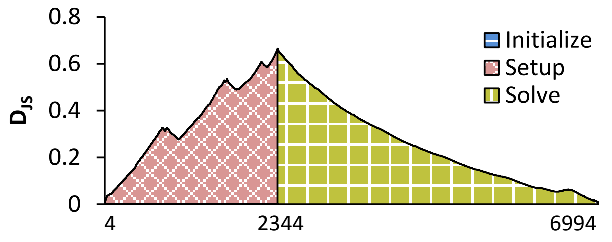
Figure 11 shows the plot of the D_{JS} for the whole pattern sequence for three different scenarios. Figure 11a shows the D_{JS} for 16x16x4 scenario. The graph shows the three identified phases that were detected in Figure 10 where the initialization phase was detected at t=3 as shown in the segmentation tree. The pattern list was first segmented at point 5601 which is exactly at the beginning of the Solve phase. This shows that our approach scales up to larger process numbers. The same behavior was exactly the same for the other 3D process topologies for problem size 1x1x1. Figure 11b shows that when using a 2D topology the communication behavior in the program changed which resulted in a different D_{JS} for the communication pattern list. The Solve phase was detected at t=4 in the segmentation tree. This behavior is consistent for all the 2D topology scenarios. Another interesting result is clear when increasing the problem size in Figure 11c. The initialization phase contains only four communication patterns which are the ALLGATTHER, ALLGATHERV, a communication pattern (call it CP_1) that involves all the processes in the program and finally the ALLREDUCE.ALLREDUCE pattern that marks the end of the phase. The Solve phase starts at position 2345 and was detected from the first segmentation step. It is interesting that CP_1 is also occurring in the Setup phase. In the other topologies, the initialization phase contains short communication patterns that involve a fewer number of processes which may indicate that these patterns together may compose one large communication pattern that involves all the processes in the program which will result in only four patterns in the initialization phase for all the scenarios. However, since our algorithm only looks for repeating patterns (maximal repeats) on each process this communication behavior could not be detected. Moreover, the segmentation algorithm was able to identify the phases much faster in the 1x1x1 problem size scenarios (t=1 for the Solve phase and t=3 for the other two phases). It should be noted that the size of phases in Figure 11 does not represent the execution time of each phase but corresponds to the number of communication patterns in each phase.



(a) D_{JS} for 16x16x4 Topology (1x1x1 Problem Size)



(b) D_{JS} for 16x16x1 Topology (1x1x1 Problem Size)



(c) D_{JS} for 4x4x4 Topology (3x2x2 Problem Size)

Figure 11. Jensen-Shannon Divergence for S_0

We mapped the detected segments to the original trace and located the user-defined routines that were called at the beginning of each phase. We used the detailed SMG2000 description available at [21] to validate the correctness of the detected phases. In the following, we conclude the outcome of our analysis of SMG2000 for the 16x16x4 topology example.

Initialization Phase: Segment S5 in Figure 10 represents the initialization phase. This phase contains a total of 1081 communication pattern instances. The HYPRE_StructGridAssemble sub-phase which includes the ALLGATHER and ALLGATHERV collectives was detected at depth 11 in the segmentation tree. Moreover, the HYPRE_StructMatrixAssemble sub-phase which corresponds to the communications from 4 to 1081 in the pattern sequence was also detected at depth 11 in the tree.

Setup Phase: This phase starts at position 1082 and ends at position 5601 in the patterns sequence. It spans the three segments $S_6.S_7.S_8$ shown in Figure 10 which are included in the call to the HYPRE_StructSMGSetup routine.

Solve Phase: This phase starts at point 5602 and ends at point 15373 in the patterns sequence and only occurs in S_2 segment as shown in Figure 10. The detected phase starts at the *Enter* event of the HYPRE_StructSMGSolve routine and ends at its *Exit* event. S_2 is a very homogenous phase due to its iterative behavior. Therefore, S_2 could not be further segmented to discover its sub-phases. This behavior is consistent in all the 15 presented scenarios. Our approach, which is based on the segmentation of heterogeneous sequences, fails in identifying fine-grained phases in such cases.

4.2. NAS – Block Tridiagonal

The second system in this evaluation is the Block Tridiagonal (BT) benchmark which is part of the NAS PB[1] suite. It uses an implicit algorithm to solve the 3-D compressible Navier-Stokes equations.

Table 3 shows the two scenarios involved in our study. The first one is generated from a class W problem scenario and the second one is from a class B problem scenario. The number of processes is 16 and 32 respectively.

Table 3. NAS BT Scenarios

np	***Class***	$\sum ev$	$\sum mpi$	$\sum msg$	$\sum cv_p$	$\sum PP$	PP_d *(ms)*	$\sum CP$	CPL_l	CP_d *(ms)*	PH_d *(ms)*
16	W	6,500,800	38,912	25,728	14	32	630	4	211	54	89
32	B	1,033,480	48,560	32,160	14	64	954	4	211	101	89

Figure 12 shows the process topology (right) and the detected patterns. Our approach detected one global point-to-point communication pattern PT_1 that is repeated 201 times in the trace. PT_1 is represented textually to simplify reading. The other communication patterns are collective operations. The BCAST communications are performed at the initialization while the REDUCE collectives are performed at the finalization phase.

The NAS BT benchmark consists of the following three main phases [Geisler 99]:

- Initialization: sets all the initial values.
- Solve:
    - Copy Faces: exchanges boundary values between neighboring processes.
    - X Solve: solves the problem in the x-dimension.
    - Y Solve: solves the problem in the y-dimension.
    - Z Solve: solves the problem in the z-dimension.
    - Add: performs a matrix update (no communications).
- Finalization: verifies the solution integrity, cleans up data, and prints the final results.

[1] https://www.nas.nasa.gov/publications/npb.html

```
PT1 (201):
  P0: S1.R1.S2.R2.S8.R8.S9.R9
  P1: S0.R0.S3.R3.S4.R4.S11.R11
  P2: S3.R3.S0.R0.S5.R5.S12.R12
  P3: S2.R2.S1.R1.S7.R7.S6.R6
  P4: S6.R6.S1.R1.S14.R14.S8.R8
  P5: S7.R7.S2.R2.S13.R13.S9.R9
  P6: S4.R4.S3.R3.S12.R12.S10.R10
  P7: S5.R5.S3.R3.S11.R11.S10.R10
  P8: S12.R12.S15.R15.S0.R0.S4.R4
  P9: S11.R11.S15.R15.S0.R0.S5.R5
  P10:S14.R14.S13.R13.S6.R6.S7.R7
  P11:S9.R9.S14.R14.S1.R1.S7.R7
  P12:S13.R13.S8.R8.S6.R6.S2.R2
  P13:S12.R12.S15.R15.S10.R10.S5.R5
  P14:S15.R15.S10.R10.S11.R11.S4.R4
  P15:S14.R14.S13.R13.S9.R9.S8.R8

PT2 (5): BCAST, PT3 (3): REDUCE
PT4 (2): BCAST.BCAST
```

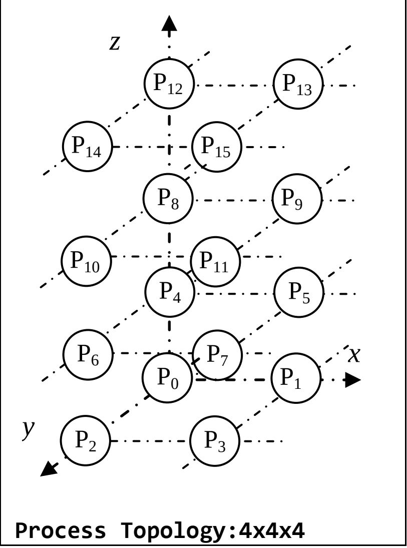


Figure 12. Communication Patterns in BT (Class: W, Iterations: 200) and Process Topology

Figure 13 lists the few resulted segments for positive segmentation strength. The small number of segments is expected due to the low number of communication patterns, resulting in low entropy. Segment S_1 represents the initialization phase and includes only broadcast collective communications. S_3 represents the Solve phase and consists of only one communication pattern that is repeated 201 times. Finally, S_4 represents the finalization phase where the verification of results as well as the output is printed. This is due to the repeating nature of the program.

S	p_s	p_e	L	S	D_{JS}	p_c	P
S_0	1	211	211	10.14	0.2	7	-
S_1	1	7	7	-0.42	0.23	6	S_0
S_2	8	211	204	4.64	0.11	208	S_0
S_3	8	208	201	-1	0	208	S_2
S_4	209	211	3	-1	0	211	S_2

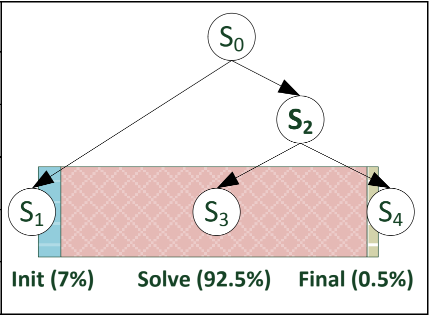


Figure 13. Recursive Segmentation (P_s: start position, P_e: end position, l: length, D_{JS}: Jensen-Shannon Divergence, p_c: cutting position of max divergence, s: Segmentation Strength, P: parent node)

5. Discussion and Threats to Validity

Our approach can be used both in SPMD and MPMD HPC systems. Naming the phase in MPMD systems will not be straightforward as in SPMD since the functionality in MPMD programs is decomposed among the processes.

The recursive segmentation algorithm provides very accurate results based on the specified threshold and segmentation strength. The algorithm derives the segments and sub-segments based on the entropy in the input sequence. The analyst may vary the threshold in order to achieve fine- or coarse-grained segments (phases in our context). Recursive segmentation is not linear with a complexity of *O (NLog(N))* where N represents the size of the sequence to be segmented [Li 02]. In our case, we applied segmentation on a sequence of communication patterns. Therefore, *N* refers here to the number of pattern occurrences in a trace and not the number of events in the original trace. For example the SMG2000 16x16x1 trace contains approximately 600 million events but only around 15 thousand patterns. Applying the recursive segmentation directly on raw traces may pose scalability issues. The complexity of extracting patterns from a trace using the algorithm proposed in Section 3.2 is *O(l*S)* where *l* refers to the length of the longest pattern and *S* the size of the raw trace since the algorithm may require multiple passes through the trace until the longest pattern is formed.

One way to improve the scalability of the approach is to filter out functions that implement low-level utilities or any other components of the system that may not add any value to the analysis. Utilities tend to be the functions that are repeated most frequently in the trace [Hamou-Lhadj 04]. The challenge, however, is to automatically distinguish the utilities from core functions. We intend to investigate this in future work. User-based filtering could be a possible approach. Users can choose to instrument smaller parts of the system that they want to focus their analysis on.

Another limitation of our approach is that we labelled the phases manually by referring to whatever documentation (including source code comments) available. In practice, manual labelling is not desirable. We need to investigate ways to automatically extract labels from various system artifacts. A possible solution is to use information retrieval techniques to mine source code comments and other artifacts such as method names, etc. A good example of this is the work by Medini et al. [Medini 12]. The authors proposed a labelling mechanism using text mining for labelling segments once extracted. Their labelling method can be used with any trace segmentation technique. Their work, however, is limited to simple Java programs. Extending this work to heterogeneous environments is necessary.

The selection of the dataset is one of the common threats to validity for this type of studies. It is possible that the traces extracted from these two systems share common properties that we are not aware of and therefore, invalidate our results. However, these two systems are used in many similar studies so we believe that they are representative systems for this research. This said, we acknowledge that we need to apply our approach to other systems.

Another threat to validity lies in the way we have selected the traces in this study. We selected the traces by running the systems using various configurations to avoid any bias. One may argue that a better approach would be to select traces based on other criteria such as the size of the traces or the number of distinct functions they contain, etc. We believe that longer and more complex traces may perhaps have an impact on the running time of the approach, but we are not convinced that the accuracy of our approach depends on the complexity of the traces. Besides, traces used in this paper are very large (they contain tens of millions of events); they should provide good coverage of the running systems.

In addition, we see a threat to validity that stems from the fact that we implemented the recursive segmentation algorithm based on the description of the approach in the paper [Li 02]. Unfortunately, we were not able to have access to the implementation of the authors. We tested our implementation on many examples to make sure it works properly.

The threshold, *t*, may be a threat to validity since a different threshold may lead to different results. In practice, we need to have a way to vary the threshold until satisfactory phases are obtained. This can be achieved by embedding this approach in a trace analysis tool and allowing users to interact with the tool to vary the threshold.

Moreover, we see a threat to validity in the trace generation tool we used. If the tool does not handle indeterminism well and other complex scenarios, this may affect our approach. This threat is mitigated by the fact that we used the Score-P tool suite, which is a well-supported infrastructure for tracing HPC systems, developed by HPC experts.

Finally, we see a threat to validity that stems from the fact that we only used one industrial system, which hinders the generalizability of our approach.

6. Conclusions and Future Work

We proposed a new approach for identifying execution phases in message passing programs based on the segmentation of the communication patterns sequence extracted from execution traces generated from SPMD HPC systems.

Our approach elaborates on the steps that are required to identify the execution phases supported by an explanatory example. We validated the results of our approach on large scale traces from the SMG2000 system and the BT NAS benchmark. The presented approach does not only identify the main program phases but also the sub-phases.

Our approach depends on the threshold *t* to determine the level of granularity of the detected phases. One possible way to determine this threshold is to have domain experts apply our technique to several post-mortem traces, analyze the resulting phases by varying *t*, and decide on the ones that are most suitable. The other analysts can then use the same *t* when analyzing similar traces of the same systems.

Our approach is based on MPI 2.0. We need to examine the changes to MPI, reflected in MPI 3.0, and see how these changes affect our approach. This may require updating the way some MPI operations are being handled by our approach.

In the future, we intend to enhance the phase detection approach by including a different segmentation technique for segmenting long homogeneous sequences such as the ones identified in the target systems. This will provide a way to further break down these long phases into sub-phases.

Moreover, we intend to further reduce the number of distinct communication patterns by considering the similarity among them resulting in a more homogeneous sequence.

We also intend to improve the performance of our algorithms by parallelizing them such as the per-process detection algorithm and the recursive segmentation technique. Moreover, we will use our technique in identifying performance bottlenecks by locating irregular communication patterns due to delays in computations as well as network overhead.

Finally, we should also investigate how this approach can be extended to support other inter-process communication models, in addition to SPMD.

Acknowledgment

The authors would like to acknowledge the support provided by the Natural Science and Engineering Council of Canada (NSERC) and the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No. IN131031.

7. References

Aguilar 14 X. Aguilar, K. Fürlinger, and E. Laure, “MPI trace compression using event flow graphs,” *In Proc. of the 20th International Euro-Par Conference on Parallel Processing (Euro-Par ’14),* pp. 1-12, 2014.

Aguilar 15 X. Aguilar, K. Fürlinger, and E. Laure, “Automatic On-Line Detection of MPI Application Structure with Event Flow Graphs,” *In Proc, of the 21th International Euro-Par Conference on Parallel Processing (Euro-Par '15),* pp. 70-81, 2015.

Akaike 78 H. Akaike, “A Bayesian analysis of the minimum AIC procedure,” *Annals of the Institute of Statistical Mathematics*, 30 (Part A), pp. 9-14, 1978.

Alawneh 12 L. Alawneh, A. Hamou-Lhadj, “Identifying computational phases from inter-process communication traces of HPC applications,” *In Proc. of the International Conference on Program Comprehension (ICPC 2012)*, pp. 133–142, 2012.

Alawneh 11 L. Alawneh and A. Hamou-Lhadj, “Pattern Recognition Techniques Applied to the Abstraction of Traces of Inter-Process Communication,” *In Proc. of the European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pp. 211-220, 2011.

Alawneh 14 L. Alawneh, A. Hamou-Lhadj, S. Shariyar Murtaza, Y. Liu, “A contextual approach for effective recovery of inter-process communication patterns from HPC traces,” *In Proc. of I In Proc. of the Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pp. 274-282, 2014.

Cornelissen 09 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.

Casas 07 M. Casas, R. M. Badia, and J. Labarta “Automatic phase detection of MPI applications,” *In Proc. of the 14th Conference on Parallel Computing Parallel Computing*, pp. 129-136,2007.

Casas 10 M. Casas, R. M. Badia, J. Labarta, “Automatic Phase Detection and Structure Extraction of MPI Applications*,” International Journal of High Performance Computing Applications*, 24(3), pp.335-360, 2010.

Chetsa 13 G. L. T. Chetsa, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, “A User Friendly Phase Detection Methodology for HPC Systems’ Analysis,” In Proc. of IEEE International Conference on Green Computing and Communications, Beijing (China), pp. 118–125, 2013.

Cormen 90 T. H. Cormen, C.E. Leiserson, R.L. Rivest, “Introduction to Algorithms,” *The MIT Press*, Cambridge, MA, 1990.

Geimer 09 M. Geimer, F. Wolf, B.J.N. Wylie, B. Mohr, “A scalable tool architecture for diagnosing wait states in massively-parallel applications,” *Journal of Parallel Computing*, 35(7), pp. 375–388, 2009.

González 09	J. González, J. Giménez, J. Labarta, “Automatic Detection of Parallel Applications Computation Phases,” In *Proc. of International Parallel & Distributed Processing Symposium (IPDPS)*, pp. 1-11, 2009.
González 12	J. González, K. Huck, J. Giménez, & J. Labarta, “Automatic Refinement of Parallel Applications Structure Detection,” *In Proc. of 26th International the Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pp. 1680–1687, 2012.
González 13	J. González, J. Giménez, J. Labarta, “Performance Analytics: Understanding Parallel Applications Using Cluster and Sequence Analysis,” *In Proc. of the 7th International Workshop on Parallel Tools for High Performance Computing*, pp. 1-17, 2013.
Gray 11	R. Gray, “Entropy and information theory,” 2nd Edition, New York, Springer, 2011.
Grosse 02	I. Grosse, P. Bernaola-Galván, P. Carpena , R. Román-Roldán, J. Oliver and H. E. Stanley, "Analysis of symbolic sequences using the Jensen-Shannon divergence," *Phys. Rev.* E, 65(4), pp.041905-1 -041905-16 , 2002.
Hamou-Lhadj 04	A. Hamou-Lhadj, and T. Lethbridge, "Reasoning About the Concept of Utilities," *ECOOP International Workshop on Practical Problems of Programming in the Large, Oslo, Norway, Lecture Notes in Computer Science (LNCS), Vol 3344, Springer-Verlag*, pp. 10-22, 2004.
Isaacs 15	K. E. Isaacs, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, “Ordering traces logically to identify lateness in message passing programs,” *IEEE Transactions on Parallel and Distributed Systems*, 2015.
Knüpfer 06	A. Knüpfer, B. Voigt, W.E. Nagel, H. Mix, “Visualization of repetitive patterns in event traces,” *In Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, p. 430-439, 2006.
Köckerbauer 10	T. Köckerbauer, T. Klausecker and D. Kranzlmüller, “Scalable Parallel Debugging with g-Eclipse,” *In Proc. of the 3rd International Workshop on Parallel Tools for High Performance Computing, published as a book chapter in Tools for High Performance Computing, Springer Berlin Heidelberg*, pp. 115-123, 2010.
Kunz 97	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)*, pp. 12-24, 1997
Li 02	W. Li, P. Bernaola-Galvan, F. Haghighi, I. Grosse, “Applications of recursive segmentation to the analysis of DNA sequences,” *Journal of Computers & Chemistry*, 26, pp. 491-510, 2002.

Maghraoui 05	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 (ICPP AM),* pp. 258-271, 2005.
Medini 12	S.Medini., G.Antoniol.,Y. Gueheneuc, M. Di Penta and P.Tonella, "SCAN: An Approach to Label and Relate Execution Trace Segments," *In Proc. of 29th Working Conference on Reverse Engineering,* pp. 135-144, 2012.
Newman 14	G. A. Newman, "A Review of high-performance computational strategies for modeling and imaging of electromagnetic induction data," Surveys in Geophysics, vol. 35, pp. 85-100, 2014.
Noeth 09	M. Noetha, P. Ratna, F. Muellera, M. Schulzb, B. R. de Supinskib, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing*," Journal of Parallel and Distributed Computing*, 69(8), pp. 696-710, 2009.
Pirzadeh 11a	H. Pirzadeh, A. Hamou-Lhadj, "A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension," *In Proc. of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '11)*, pp. 221-230, 2011.
Pirzadeh 11b	H. Pirzadeh, A. Hamou-Lhadj, "A Software Behaviour Analysis Framework Based on the Human Perception System: NIER Track*", In Proc. of the 33rd International Conference on Software Engineering (ICSE NIER Track)*, pp, 948-951, 2011.
Preissl 08	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 (ICPP),* pp. 230–237, 2008.
Preissl 10	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 39th International Conference on Parallel Processing (ICPP),* pp. 51-60, 2010.
Reiss 05	S. P. Reiss, "Dynamic detection and visualization of software phases", *In Proc. of the 3rd International Workshop on Dynamic Analysis (WODA),* ACM, pp. 1-6, 2005.
Shannon 48	C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal, vol.* 27, pp. 379-423, 1948.
SMG2000	Advanced Simulation and Computing Program: The ASC SMG2000 benchmark code. URL: http//www.llnl.gov/asc/purple/benchmarks/limited/smg/, 2001.
Tiwari 11	I. Tiwari, J. K. Hollingsworth, C. Chen, M. W. Hall, C. Liao, D.J. Quinlan, J. Chame, "Auto-tuning full applications: A case study," *The International Journal of High Perfornace Computing Applications*, 25(3), pp. 286-294, 2011.

Trahay 15	F. Trahay, E. Brunet, M. M. Bouksiaa, and J. Liao, “Selecting points of interest in traces using patterns of events,” *In Proc. of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP* ’15), pp. 70-77, 2015
Wolf 03	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.
Wolf 07	F. Wolf, B. Mohr, J. Dongarra, S. Moore, “Automatic analysis of inefficiency patterns in parallel applications,” *Wiley Journal of Concurrency and Computation: Practice and Experience*, 19(11), pp. 1481–1496, 2007.