Identifying Computational Phases from Inter-Process Communication Traces of HPC Applications

Luay Alawneh and Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve Blvd. West
Montreal, QC, Canada H3G 1M8
{l_alawne, abdelw}@ece.concordia.ca

Abstract— **Understanding the behaviour of High Performance Computing (HPC) systems is a challenging task due to the large number of processes they involve as well as the complex interactions among these processes. In this paper, we present a novel approach that aims to simplify the analysis of large execution traces generated from HPC applications. We achieve this through a technique that allows semi-automatic extraction of execution phases from large traces. These phases, which characterize the main computations of the traced scenario, can be used by software engineers to browse the content of a trace at different levels of abstraction. Our approach is based on the application of information theory principles to the analysis of sequences of communication patterns found in HPC traces. The results of the proposed approach when applied to traces of a large HPC industrial system demonstrate its effectiveness in identifying the main program phases and their corresponding sub-phases.**

Keywords - Program comprehension, Dynamic analysis, High Performance Computing Systems, Inter-process communication traces, Execution phases

I. INTRODUCTION

High Performance Computing (HPC) systems are designed to solve advanced computational problems that are highly challenging, complex, and time consuming. HPC is quickly becoming the paradigm of choice for building super powerful applications, especially since the emergence of multi-core and cloud computing platforms.

HPC applications, however, are also known to be very challenging to understand and debug due to the large number of processes they may involve and the complex parallel interactions among these processes [1]. In recent years, a number of studies have emerged to facilitate the understanding of the behaviour of HPC applications through tracing and monitoring techniques (e.g. [27-29]). The objective is to extract views from raw traces that can help software engineers understand the traced scenario. Traces, however, can be extraordinary large. The size of typical traces can easily reach millions of events long. Despite the advances in the area of trace abstraction and simplification techniques, there is a general consensus that the area is still in its infancy and that much more needs to be done.

In this paper, we propose a novel approach to facilitate the understanding and the analysis of large traces generated from HPC applications. Our approach aims to localize computational phases from large HPC traces. We define a computational phase as part of a trace where a particular program computation is invoked. For example, a trace that is generated from a compiler should contain events that represent the various compiler's computational phases including initialization of variables, parsing, preprocessing, lexical analysis, semantic analysis, and so on. Knowing where each of these phases occurs in the trace is usually a challenging task since there is no support at the programming language level of how to explicitly indicate the beginning and end of each phase. This is further complicated in the context of HPC applications where a phase can be performed by multiple processes running in parallel. But, if done properly, the recovery of computational phases (and their sub-phases) can reduce considerably the time and effort spent by software engineers on understanding what goes on in a trace.

The phase detection approach presented in this paper encompasses two main steps. First, we detect communication patterns that characterize the way processes communicate with each other throughout the execution of the program. We achieve this by applying the communication pattern detection algorithm presented in [12]. The second step, which is also the main contribution of this paper, consists of an approach for automatically grouping the extracted patterns into dense homogenous clusters that indicate the presence of computational phases. We achieve the second step using information theory concepts such as Shannon entropy [27] and the Jensen-Shannon Divergence measure [17].

The focus of this paper is on HPC applications that use the Message Passing Interface (MPI) [9] as their main inter-process communication standard. We sometimes use the term MPI traces throughout the paper to refer to HPC traces. We believe that this work is readily applicable to other communication mechanisms.

The rest of the paper is organized as follows. In Section 2, we present the related work followed by a description of inter-process communication traces in Section 3. In Section 4, we present our overall approach and briefly describe the algorithms used in the detection process. In Section 5, we present the effectiveness of our approach by applying it to a trace generated from a target system. We conclude our work in Section 6.

II. Related Work

The area of trace analysis for HPC applications with a focus on program comprehension tasks is relatively new. Most of the HPC-related studies fall within the realm of performance analysis. In what follows, we present the recent studies that are most relevant to our work.

In their study [8], Casas et al. applied the wavelet transform technique in the signal processing field to automatically detect the main execution phases in MPI applications. The algorithm identifies phases by separating execution regions based on their iterative frequency. The different MPI phases (initialization, computation, and output) are categorized based on their frequency of iterative behaviour where in the computational phase most of the parallel iterations exist. In [3], the authors extended this work that uses wavelet transform from signal processing in order to detect the different sub-phases in the computational phase. They based their approach on the iterative behaviour found in MPI traces where CPU bursts were followed by process communication. They derived the signals from different metrics that were based on inter-process communication and computing bursts. They assumed that the highest frequencies of communications (signals) appeared when there was a computational phase change. Our work can be seen as complimentary. Instead of looking at frequency and other usage metrics, we focus mainly on the trace content itself, i.e., its events and communication patterns.

Gonzalez et al. [2] presented an approach to facilitate the analysis of message passing parallel applications using the density-based clustering techniques to detect computational phases that occur between the parallel communications in the program. They applied the density-based approach on data obtained from performance counters provided by modern processors. The main objective of their work was to detect the most important regions of execution in the program. They used CPU bursts to outline the different regions in the program. A CPU burst was considered as a CPU computation region between two consecutive communications. Therefore, a burst was identified by the duration and the set of performance counters. In our experience, performance data requires a lot of fine-tuning in order to obtain accurate computational phases.

Pirzadeh and Hamou-Lhadj presented a novel phase detection approach that they called trace segmentation and which was inspired by the way the human perception system groups lines and dots into shapes and objects [23, 30]. They have developed several methods that could automatically group trace events into dense elements that formed computational phases. Their work, however, focuses on traces of routines calls and not inter-process communication traces. We plan in the future to study how their approach can be applied in the context of this study.

III. Message Passing Interface Traces

The Message Passing Interface (MPI) [9] is the de facto standard for inter-process communication in HPC programs. The main advantages that distinguish MPI from other message passing paradigms are its support for asynchronous communication, process group context, and process synchronization. Another important advantage is its portability since all existing implementations on different platforms are based on the same open accepted standard.

MPI provides point-to-point and collective types of communications. Point-to-point communication involves only two processes in an MPI program. MPI allows the same process to act as the sender and the receiver for the same message. The sending process posts a send operation that contains the destination process, the message data, the data type, the tag, and the communicator. The tag is an integer value that helps in identifying the incoming message at the receiving process. The receiver, on its side, should post a receive operation that matches the incoming message based on its data type, the tag value, and the source process. However, the receiving process may post a receive operation that can accept a message coming from any source in the group and that has any tag value.

MPI collective communication defines different types of operations for exchanging information among a group of processes defined as an MPI communicator. MPI assumes that all processes, in a communicator, must execute the same collective operations in the same order. In order to guarantee the synchronization among all processes in the communicator, MPI recommends the usage of the ‘barrier’ operation. It is worth mentioning that MPI collective operations are based on point-to-point operations. However, the communication mode in a collective communication must be blocking in order to enforce the execution of the same collective operation by all processes synchronously. Moreover, collective operations do not use tags as message identifier in order to strictly force the exchange of messages according to their order of execution. All processes must post collective operations that exactly match the size and data type of the exchanged data.

An MPI trace consists of events generated from each running process in the program. Each process events can be collected in a separate file. The trace from each process consists of routine call events as well as the MPI events. Inter-process communication traces are those events generated from MPI point-to-point and collective communication routine calls.

IV. Proposed Approach

Figure 1 shows our execution phase detection approach. The trace is first divided into multiple process traces in which the events of each process are grouped together. The next step is to detect communication patterns from the process traces. For this, we use an algorithm that we presented in [12] and that we will review in the upcoming subsections. These patterns are then input to the phase detection component. The phase detection method looks for changes in communication patterns throughout the program execution. Note that a phase may be composed of multiple patterns. The challenge is to automatically identify groups of homogenous patterns and distinguish them from each other. We achieve this by measuring the degree for which multiple patterns can be considered homogenous using the Jensen-Shannon divergence metric. The phase detection approach is discussed in more detail in Section IV.C. Finally, we analyze the execution phases. The result might necessitate further fine-tuning of the pattern detection technique or the phase detection algorithm until satisfactory phases are obtained. This last step is done manually.

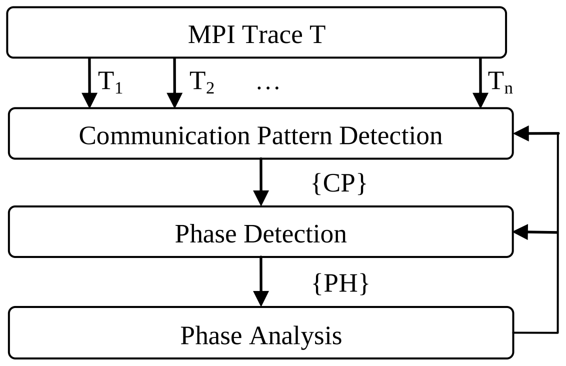


Figure 1. Phase Detection Approach

A. Trace Generation

There exist several tools that automatically instrument MPI applications by allowing users to specify places in the code where probes should be inserted. An Example of such tools are VampirTrace [25] and TAU [5]. We generate a trace based on a specified scenario determined by the input parameters. The resulting trace contains events of the entire system execution from initialization to outputting the results. HPC systems have different input parameters based on the problem to solve. In addition to the input data, we also need to specify the number of processes. We can vary the number of processes to increase or decrease the processing speed depending also on the capabilities of the host node. Once a trace is generated, we create a process trace for each process.

B. Communication Pattern Detection

A communication pattern in HPC applications represents a way the program processes communicate with each other to accomplish a specific task. Figure 2 depicts a sample trace generated from running four processes in parallel. Each horizontal line represents the events from each process. When matching the MPI events with the partner processes, a communication pattern is discovered. The example in the figure shows a nearest-neighbor communication pattern (with a 4 x 1 process topology) that is repeated three times at different locations in the trace. Non-MPI events are represented using the dark bars. The graph that is used to depict the communication events is known as the event graph [11] where the x-axis represents time and the y-axis represents the processes. The trace events flow from left to right.

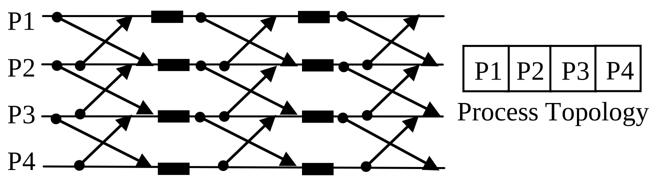


Figure 2. Repeating Communication Pattern

There are several communication patterns that are documented such as the wevefront pattern [10]. Typical MPI applications may be composed of a large number of communication patterns depending on the complexity and the computational task as well the requirements in terms of program performance and efficiency.

A process topology is the way the processes are represented on a grid (Cartesian) or a graph structure. For example, in Figure 2 (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 are arranged on two superposed 2D, forming a 3D grid.

As an MPI application undergoes several ad-hoc maintenance activities, it becomes challenging to know which patterns are supported. This has led several researchers to design methods for automatic recovery of communication patterns (e.g., [13]).

In [12], we studied this problem and proposed an efficient algorithm that detects communication patterns from large MPI traces. Our algorithm encompassed two main steps: First, we detected the repeating message passing events patterns for each process trace separately. To achieve this, we used the concept of n-grams extraction technique, found in statistical natural language processing. In the classical n-gram pattern detection approach [14], the algorithm looks for all n-size patterns in a sequence. This approach tends be exhaustive (and hence resource-consuming) especially when applied to long sequences with unknown pattern sizes. To overcome this issue, we developed a new algorithm that detected patterns as it passed through the trace. We used bi-grams (length = 2) as the minimum length of a pattern. In the algorithm, the pattern length increased whenever a new occurrence was detected. This concept was also used in the LZW data compression algorithm [15], where whenever a pattern already exists in the pattern database, it is revisited in the sequence. The algorithm appends the next symbol in the sequence to the end of the pattern. Our algorithm differs from the LZW algorithm in that it checks whether a pattern exists at the previous positions of its prefix pattern (‘ab’ is the prefix of ‘abc’).

The second step of the algorithm was to assemble the patterns detected in the previous step into communication patterns that combine multiple partner processes. We input the process patterns into this algorithm and started iterating on all corresponding patterns - the corresponding patterns of a pattern *PT1* are those patterns that have partner events with PT_1 - until a communication pattern was constructed. We applied the algorithm to traces generated from two MPI systems and obtained superior results compared to existing approaches both in terms of accuracy and efficiency. The detailed description of the algorithm along with an example is presented in [12].

C. Phase Detection

Our phase detection approach is inspired by studies in the field of bioinformatics, more particularly, the analysis of DNA sequences. In [7], the authors proposed a recursive algorithm for segmenting a DNA sequence into more homogeneous sub-domains. The algorithm follows the divide-and-conquer approach proposed in [16], which relies on information theory concepts. More precisely, the algorithm uses Shannon entropy [6, 27] and the Jensen-Shannon divergence measures [17] to guide the segmentation process.

We adapted this algorithm to the segmentation of a MPI trace, in which the symbols represent the communication patterns identified in the previous step. The length of the sequence is the number of instances of the patterns. It should be noted that another alternative would have been to apply the sequence segmentation to the original trace. This would however been impractical given the high number of events involved, hence the use of communication patterns.

The segmentation process starts by measuring the degree of heterogeneity of the sequence. For this, Shannon entropy is used [27]. Shannon entropy measures the amount of information in a sequence by assessing how much randomness exists in the sequence. A sequence for which all the symbols appear with the same probability will result in low entropy (meaning that the uncertainty about the data is at its minimum). On the other hand, the higher the entropy, the more variations exists in the data (i.e., the more heterogeneous the data is). The Shannon entropy H of a sequence S of length N with k distinct symbols is defined using the following equation [27].

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

Where N_j is the number of times symbol j appears in sequence S.

Once the Shannon entropy of a sequence is measured, the next step is to identify places in the sequence where heterogeneous behaviour occurs. This process is done recursively based on the following steps:

- For each position i in the sequence, we measure the entropy of the left subsequence and the right subsequence from position i. Note that the left and right subsequences must not be empty. H_l and H_r which represent the entropy of the left and right subsequences are computed as follow:

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

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

Where N_j^l is the number of times symbol j appears in the left subsequence S_l and N_j^r is the number of times symbol j occurs in the right subsequence S_r.

- For each two subsequences, we measure their similarity by comparing the entropy values using the Jensen-Shannon Divergence (D_{JS}) measure [17] and which is presented below. The higher D_{JS}, the more heterogeneous the subsequences are:

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

- We select the subsequences for which D_{JS} has the highest value and apply the segmentation process recursively to these subsequences until a stopping criterion is met, which is explained in what follows.

In order to determine the criterion for stopping the recursive segmentation process, Li et al. proposed to use the model selection framework presented in [7] where a model can be evaluated by a combination of the degree to which the model fits the data and the complexity of the model itself. In sequence segmentation, we have two models. The first model M_1 is represented by the whole sequence S whereas the second model M_2 is represented by the left and right subsequences (S_l and S_r) respectively. The objective is to find a model at the boundary between the under-fitting models (models that do not fit the data well) and over-fitting models (models that fit the data too well using many parameters). Li et al. [7] proposed to use the Bayesian Information Criterion (*BIC*) [18] in order to balance the goodness-of-fit of the model to the data with respect to the number of parameters in the model. The *BIC* is defined by:

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

Where L is the maximum likelihood of the model, K is the number of free parameters in the two models, and N is the sample (sequence) size. The value of K is calculated using (k_l + k_r + 1 – k) where k_l is the number of distinct parameters in S_l, k_r is the number of distinct parameters in S_r and k is the number of distinct parameters in S. In the following, we will explain how *BIC* can be used to derive the stopping criterion for recursive sequence segmentation based on Shannon entropy. The likelihood for S (before segmentation) is determined by:

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

Where p_j is equal to N_j/N (the probability of symbol j in sequence S). Therefore, the *log*-likelihood is determined by:

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

It can be easily shown that the *log*-likelihood (*log L1*) before segmentation is *equal to (- NH)* where H is the Shannon Entropy for the whole sequence S.

Additionally, the likelihood for the left and right subsequences (after segmentation) is determined 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. N_l is the cutting point (also length of left subsequence). The *log*-likelihood is determined by:

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

Similarly, it can be easily shown that *log* L2 = $-N_l H_l - N_r H_r$. The likelihood *L* is measured by the increase of likelihood from the two models as *L2/L1*. Therefore, the increase of log-likelihood is $log(L2/L1) = NH - (N_l H_l + N_r H_r)$ which is equal to ND_{JS} (see equation 4).

The maximized value of *L* (maximum likelihood) occurs at the point with the maximum D_{JS} value. In order for segmentation to continue, the *BIC* value should be reduced to the minimum (close to zero or $\Delta BIC < 0$). By replacing *L* by $N\hat{D}_{JS}$ in equation 5, it will lead to the following:

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

Where $\hat{D}_{JS}$ is the maximum Jensen-Shannon divergence value. This means that the segmentation will continue if the maximum D_{JS} value is above *log(N)K/2N*. The advantage of this approach is that the user's intervention is not required to determine the threshold value in order to stop segmentation. Therefore, the threshold value is calculated as:

$$\tau = \log(N)K / 2N \quad (11)$$

Li et al [7] proposed to use a measure of the segmentation strength *s* which is measured by the relative increase of $2ND_{JS}$ from the *BIC* threshold using the following:

$$s = \frac{2N\hat{D}_{JS} - \log(N)K}{\log(N)K} \quad (12)$$

Segmenting the sequence based on Equation 12 when s > 0 will have the same effect as segmenting the sequence when D_{JS} is greater than the dynamic threshold calculated based on Equation 10. In other words, the segmentation strength must always be positive value in order to continue the segmentation process. Moreover, the value of *s* can be adjusted to be greater than a user-specified value. Varying *s* will vary the numbers of detected subsequences. A larger *s* threshold value will result in a smaller and more fine-grained number of subsequences.

The output of the segmentation algorithm can be depicted in a binary tree where every subsequence is divided into two subsequences based on the position of the maximum D_{JS} value. The accuracy of the recursive segmentation algorithm is at the price of its relatively slower computational time since many passes through the data are needed to measure the DJs for left and right subsequences.

(1) Execution Trace

P1	$[B,B]_2,[R2R3]_{360},[R2S3]_{360},[R3S2]_{360},[S1S3]_{360},[R,R]_5$
P2	$[B,B]_2,[R4S1]_{360},[S1S4]_{360},[R1R4]_{360},[R2S4]_{360},[R,R]_5$
P3	$[B,B]_2,[R4S1]_{360},[R1R4]_{360},[S1S4]_{360},[R1S4]_{360},[R,R]_5$
P4	$[B,B]_2,[S3S2]_{360},[R2S3]_{360},[R3S2]_{360},[R2R3]_{360},[R,R]_5$

(2) Process Repeating Patterns

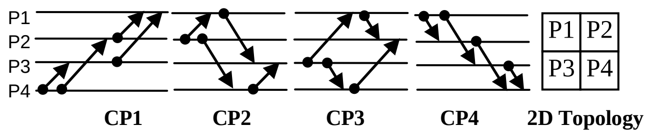
Point-to-Point Patterns:
P1:PT1 = [R2R3], PT2 = [R2S3], PT3 = [R3S2], PT4 = [S1S3]
P2:PT5 = [R4S1], PT6 = [S1S4] , PT7 = [R1R4], PT8 = [R2S4]
P3:PT9 = [R4S1],PT10 = [R1R4], PT11 = [S1S4], PT12 = [R1S4]
P4:PT13 = [S3S2], PT14 = [R2S3], PT15 = [R3S2], PT16 =[R2R3]

Collective Patterns on each process: [B,B] and [R,R]

(3) Communication Patterns (CP)

CP1: PT1, PT5, PT9, PT13, **CP2**: PT2, PT6, PT10, PT14
CP3: PT3, PT7, PT11,PT15, **CP4**: PT4, PT8, PT12,PT16
CP5: [B,B] ,**CP6**: [R,R]

(4) Point-to-point Communication Patterns (Wavefront)



(5) Communication Pattern Sequence (length = 1447)

$[CP5]_2$, $[CP1]_{360}$, $[CP2]_{360}$, $[CP3]_{360}$, $[CP4]_{360}$, $[CP6]_5$

(6) Phase Detection:

	From	To	s
S0	1	1443	273.0
S1	1	720	150.0
S2	721	1443	150.0
S3	1	361	2.9
S4	362	720	-1.0
S5	1	2	-1.5
S6	3	361	-1.1
S7	721	1079	-1.1
S8	1080	1443	7.7
S9	1080	1438	-1.1
S10	1439	1443	-1.2

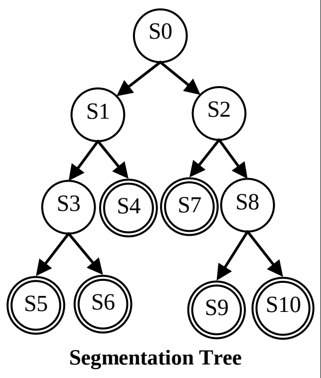


Figure 3. Phase Detection Example

We show the application of the phase detection algorithm through the example of Figure 3. This example is similar to a trace generated from running the Sweep3D [20] system on a 2x2 process topology. Figure 3(1) shows the sample trace which consists of four communicating processes. The events are represented in square brackets indicate the number of times the events are repeated. For example, $[R2R3]_{360}$ means that R2S3 is repeated 360 times in the trace (R2 means Receive from P2 and S3 means Send to P2). In this example, we do not show steps for detecting the repeating patterns and the construction of the communication patterns as they were given in [12]. Figure 3(2) shows the repeating patterns on each process and Figure 3(3) shows the communication patterns that were constructed from the process repeating patterns. For example, CP1 is constructed from the partner process patterns (PT1, PT5, PT9, and PT13).

The detected point-to-point communication patterns (in Figure 3(4)) correspond to the wavefront pattern where every process in the 2D topology sweeps data to the process in the opposite corner. Finally, the sequence of communication patterns is extracted ($[CP1]_{360}$ means that communication pattern 1 is repeated 360 times in the sequence).

Figure 3(6) shows the segmentation tree resulting from applying the phase detection algorithm. The double rounded nodes represent the detected computational phases. Table 3(6) shows the segmentation strength value for each segment. As can be seen, no further segmentation was performed for segments with negative segmentation strength values. Sweep3D has three distinct phases which are Initialize, Solve and Finalize. Our approach is also able to detect the sub-phases in the Solve phase. The following phases are detected using our approach:

- **S_5**: Initialize phase
- **S6**: Solve phase (sweep from P4 to P1)
- **S7**: Solve phase (sweep from P2 to P3)
- **S8**: Solve phase (sweep from P3 to P2)
- **S9**: Solve phase (sweep from P1 to P4)
- **S10**: Finalize phase

D. *Phase Analysis*

In this step, we verify the accuracy of the detected phases. This step is done semi-automatically. We start by mapping the phases to the original execution trace. Since each process has its own trace file, we need to map the segments to their locations in each process trace. For each process trace, the beginning of the phase will be based on the first pattern in the sequence and the end of the phase will be based on the end of the last pattern in the sequence. We use the routine-call tree in order to determine the routine that is performing this pattern. For example, if the pattern occurs at nesting level 5, then we go up in the call hierarchy until we find the highest routine call (without crossing any preceding communication patterns) that is responsible for performing the communication. We check that the routine is indeed responsible for the phase. We do this by referring to the source code or any available documentation. If not that, then the phase detection failed. In this case, we need to re-execute the pattern detection and the phase detection steps by changing the parameters.

V. CASE STUDY

In this section, we show the effectiveness of our approach by applying it to a large trace generated from the SMG2000 industrial HPC system [21]. This system is used by many other studies that target HPC applications [13].

SMG2000 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 is a SPMD (Single Program Multiple Data) program that uses data decomposition to solve the problem. SMG2000 performs a large number of non-nearest-neighbor point-to-point communication operations [22].

At a high-level, SMG2000 performs three distinct phases to solve the problem as reported in [24]. These phases are Initialization, Setup and Solve. The setup phase starts by a call to the HYPRE_StructSMGSetup routine and the Solve phase starts by a call to the HYPRE_StructSMGSolve routine. The initialization phase occurs before the setup phase and encompasses the trace events that occur before the HYPRE_StructSMGSetup routine. This information will be used in the validation of the detected phases. Our approach, as we will show in the subsequent section, also detects sub-phases in each phase.

A. *Trace Generation*

We used the VampirTrace [25] tracing tool to generate the traces from running SMG2000. The execution scenario is based on a 4x4x2 process topology (Figure 4) and a 2x2x2 input problem size.

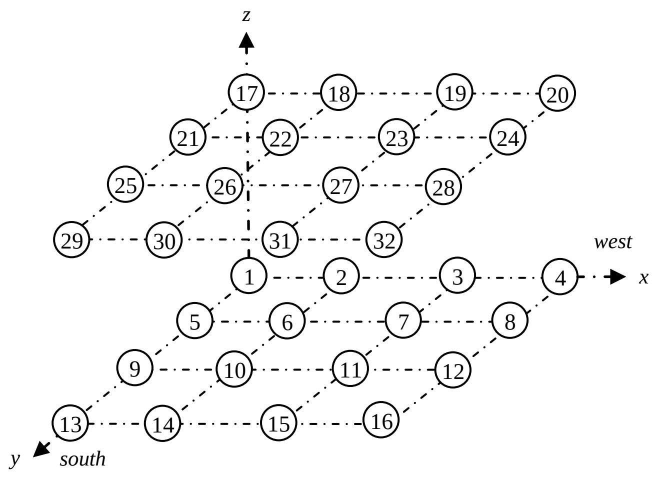


Figure 4. Process Topology for SMG2000 4x4x2

Table 1 presents some statistics about the generated trace. The total number of message passing events based on point-to-point communications is 248768. Moreover, each process exchanges data by performing 14 collective operations (a total of 448 collective communication events for all processes). Table 1 shows that this is relatively a large trace with more than 15 Million events.

TABLE 1. SMG2000 STATISTICS FOR SMG2000 TRACE (4X4X2 PROCESS TOPOLOGY AND 2X2X2 INPUT PROBLEM SIZE)

Trace Attribute	**Value**
Size of Trace	1 GB
Number of Processes	32
Total Number of Events	15392281
Point-to-point Communication Events	248768
Collective Communication Events	448

B. *Pattern Detection*

We used our pattern detection algorithm described in [12] to detect the communication patterns in the SMG2000 trace. The algorithm resulted in 47 distinct patterns (3 collective and 44 point-to-point communication patterns). The total number of patterns instances is 2065.

The validation of the communication patterns is performed using a combination of static and dynamic analysis. The static analysis part is to locate the routines that are responsible for the communication. In all communication routines, each process sends data to a group of processes and then receives data from the same group. The group of processes is determined in the calling routine and is passed to the routine responsible for handling the communication events. The dynamic analysis part is to trace these groups of processes for each process and then compare them to the partner processes in each pattern. Some of the patterns that were detected are described herein:

- *Pattern 1*: Each process communicates with its direct neighbours on each grid. For example, Process 7 will send to and receive from processes 2, 3, 4, 6, 8, 10, 11, 12, 18, 19, 20, 21, 22, 23, 24, 26, 27, and 28.
- *Pattern 2*: Each process communicates with its direct neighbours on each grid and the adjacent grid. Also, each process will communicate with its second West, South and South-West neighbours on the x-axis and y-axis on each grid. For example, Process 7 will send to and receive from processes 2, 3, 4, 6, 8, 10, 11, 12, 14, 15, 16, 18, 19, 20, 22, 23, 24, 26, 27, 28, 30, 31, 31 whereas Process 1 communicates with 2, 3, 5, 6, 7, 9, 10, 11, 17, 18, 19, 21, 22, 23, 25, 26, and 27.
- *Pattern 3*: Each process sends to the West process and receives from the East process on the same grid. No communication with processes on the other grid in this pattern. For example, Process 10 communicates with processes 9 and 11. However, Process 1 communicates with process 2 only since it does not have a direct process to its left on the grid.
- *Pattern 4*: Each process sends to the North process and receives from the South process on the same grid. No communication with other grids in this pattern. For example, Process 10 communicates with 6 and 14 while Process 1 communicates with process 5 only since it does not have a direct upper neighbor process on the same grid.

C. *Phase Detection*

We applied the recursive segmentation steps to the communication pattern sequence detected in the previous step. The results are presented in what follows. Figure 5 shows the Jensen-Shannon divergence distribution for each pattern position in the whole sequence. As we can see, the sequence can be split into two subsequences at peak point 443. Two sequences have emerged that we call S_1 (patterns positions 1 to 443) and S_2 (starting from position 444). The curve that represents sequence S_1 (position 1 to 443) in Figure 5 shows that the data is still highly heterogeneous, whereas the smooth curve for S_2 (positions 444 to 2065) shows high homogeneity. It is worth mentioning that when we mapped the first postion in S_2 (position 444) to the original trace, we found that it represents a call to the the routine HYPRE_StructSMGSolve, which seems to indicate that the Solve phase has started to take place at this position.

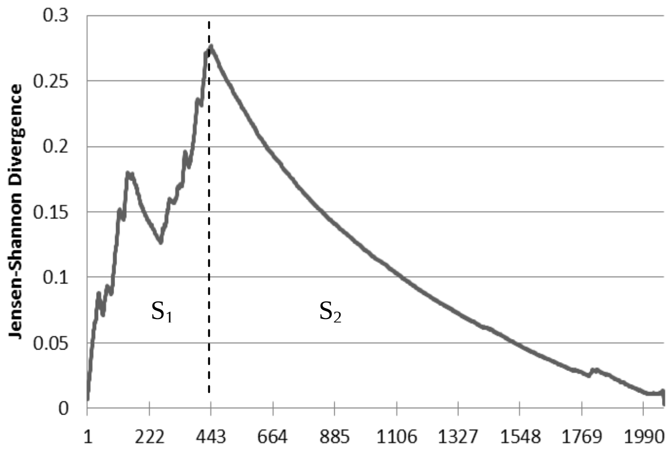


Figure 5. D_{JS} values for the whole sequence (max D_{JS} at 443, $\tau = 0.06$)

The recursive segmentation continues as long as the segmentation strength *s* is positive. As previously described, the segmentation strength *s* can be also specified by the user in order to control the number of detected sub-phases. A higher *s* value means a smaller number of phases. In this study, we segmented based on two values $s > 0$ and $s > 0.5$.

When using $s > 0$ (general case), the total number of segments (including S_0) was 68 and the number of leaf nodes (phases) was 34. However, when considering further segmentation with $s > 0.5$, the total number of segments was reduced to 27 and the number of leaf nodes was reduced to 14. We examined both computational phase sets obtained with s > 0 and s > 0.5 and found the difference is in the level of granularity of the phases. With s > 0, we obtained fine-grained phases than with s > 0.5. In this case study, we only show in Table 2 the resulting sequences from the recursive segmentation algorithm when allowing segmentation for *s* greater than 0.5. It is difficult to know in advance how to set *s* and even if we succeed to determine a proper limit for *s* for one system, there is no guarantee that it would work for another system. We anticipate that a tool that supports our technique to allow flexibility to the user to change *s* on the fly.

TABLE 2. RECURSIVE SEGMENTATION (P_S: START POSITION, P_E: END POSITION, L: LENGTH, D_{JS}: JENSEN-SHANNON DIVERGENCE, PC: CUTTING POSITION OF MAX DIVERGENCE, T: THRESHOLD, S: SEGMENTATION STRENGTH, AND P: PARENT NODE, HYPHEN(-) MEANS NO *S* FOR LENGTH = 1)

S	p_s	p_e	l	D_{JS}	p_c	τ	s	P
S_0	1	2065	2065	0.28	443	0.06	3.94	NA
S_1	1	443	443	0.33	145	0.19	0.73	S_0
S_2	444	2065	1622	0.02	2061	0.01	0.85	S_0
S_3	1	145	145	0.38	23	0.07	4.05	S_1
S_4	146	443	298	0.44	264	0.19	1.26	S_1
S_5	1	23	23	0.5	5	0.2	1.53	S_3
S_6	24	145	122	0.25	42	0.2	0.28	S_3
S_7	1	5	5	0.92	2	0.23	2.97	S_5
S_8	6	23	18	0.44	17	0.23	0.89	S_5
S_9	1	2	2	-0.25	1	0.5	-1.5	S_7
S_{10}	3	5	3	0.66	3	0.26	1.49	S_7
S_{11}	3	3	1	0	2	0	-	S_{10}
S_{12}	4	5	2	-0.25	4	0.5	-1.5	S_{10}
S_{13}	6	17	12	-0.04	16	0.3	-1.14	S_8
S_{14}	18	23	6	0.79	21	0.22	2.66	S_8
S_{15}	18	21	4	-0.12	20	0.5	-1.25	S_{14}
S_{16}	22	23	2	-0.25	22	0.5	-1.5	S_{14}
S_{17}	146	264	119	0.19	162	0.2	-0.06	S_4
S_{18}	265	443	179	0.28	294	0.15	0.95	S_4
S_{19}	265	294	30	0.76	276	0.16	3.63	S_{18}
S_{20}	295	443	149	0.33	365	0.48	-0.32	S_{18}
S_{21}	265	276	12	0.43	270	0.3	0.44	S_{19}
S_{22}	277	294	18	0.36	280	0.35	0.05	S_{19}
S_{23}	444	2061	1618	0.01	1821	0.06	-0.76	S_2
S_{24}	2062	2065	4	0.58	2062	0.25	1.31	S_2
S_{25}	2062	2062	1	0	2061	0	-	S_{24}
S_{26}	2063	2065	3	-0.17	2064	0.53	-1.32	S_{24}

Table 2 shows all the parameters used in the calculation of the segmentation process. The D_{JS} is the maximum divergence value of the point that the segmentation is performed at. It should be noted that the max D_{JS} must be always greater than τ in order to allow segmentation which is met by Equation 10. Figure 6 shows the hierarchy of the segments represented as a binary tree. The leaf nodes in the tree represent the detected sub-phases in the trace. By going up the hierarchy, we can get a coarse-grained view of the phases. The leaf nodes when the allowed segmentation strength is above 0.5 are (14 phases):

$$S_9.S_{11}.S_{12}.S_{13}.S_{15}.S_{16}.S_6.S_{17}.S_{21}.S_{22}.S_{20}.S_{23}.S_{25}.S_{26}$$

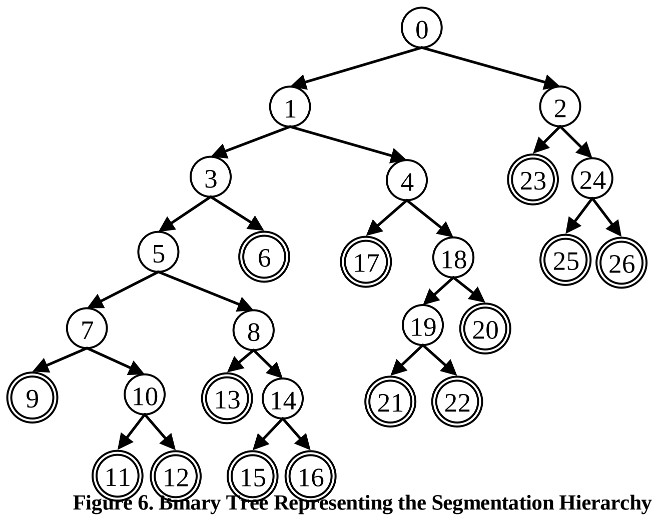


Figure 6. Binary Tree Representing the Segmentation Hierarchy

D. Phase Analysis

We mapped the phases to the original trace and analyzed the routines that were called at the beginning of each phase. The detailed descriptions of the routines of the SMG2000 are found on the SMG2000 website [21]. We used these descriptions to validate whether the phases we detected were valid or not. The following was concluded from our analysis.

Initialization Phase: This phase starts as phase S_9 and includes the phases that are in the sub-tree rooted at S_7. Table 3 describes the detected sub-phase of the initialization phase.

TABLE 3. INITIALIZATION SUB-PHASES

S	Description
S_9	This sub-phase uses the '*gather*' collective communication operation in the HYPRE_StructGridAssemble routine. Also, the hypre_InitializeTiming and hypre_BeginTiming routines are being called at the beginning of this sub-phase for tracking the timing of the initialization phase. Additionally, it contains the MPI_Init which is responsible for the initialization of MPI in each process.
S_{11}	The point-to-point communication pattern that was used in this phase is Pattern 1 described at the beginning of the case study. The main executed routine is HYPRE_StructMatrixAssemble which only found in this phase in the whole trace.
S_{12}	S_{12} uses the '*reduce*' collective operation and is responsible for tracking timing information at the end of the initialization phase (hypre_EndTiming and hypre_PrintTiming ,hypre_FinalizeTiming).

Setup Phase: The HYPRE_StructSMGSetup is responsible for starting the setup phase. It starts executing at point 6 in the sequence which corresponds to S_8 in Figure 6. The Setup phase spans the sub-trees rooted at S_8, S_6 and S_4. Table 4 provides a description of the sub-phases in the Setup phase.

TABLE 4. SETUP SUB-PHASES

S	Description
S_{13}	The call to HYPRE_StructSMGSetup is in this sub-phase. There are several routines that are distinct to this sub-phase. Also, The hypre_InitializeTiming and hypre_BeginTiming routines are being called in this phase to track the timing of the Setup phase.
S_{15} S_{16} S_6 S_{17} S_{21} S_{22}	These sub-phases are similar in terms of the routines they execute but they differ in terms of the communication patterns that are performed. S6, S17 are the longest phases and contain the highest number of communication patterns. The routines in the other phases (S_{15}, S_{16}, S_{21}, and S_{22}) are all a subset of the routines executed in these two sub-phases.
S_{20}	This sub-phase executes the same routines in S6 and S17 but it also contains the hypre_EndTiming, hypre_PrintTiming and hypre_FinalizeTiming to track the timing at the end of the Setup phase.

Solve Phase: The execution of HYPRE_StructSMGSolve starts at point 444 (belongs to S_2) and ends at point 2065 (in S_2). Therefore, the sub-tree rooted at S_2 corresponds to the Solve phase of the program. Table 5 presents the description of the sub-phases.

TABLE 5. SOLVE SUB-PHASES

S	Description
S_{23}	HYPRE_StructSMGSolve is executed at the beginning of S_{23} and indicates the start of the Solve phase. Also, in S_{23}, the hypre_InitializeTiming and hypre_BeginTiming routines are being called at the beginning of the Solve phase for tracking the timing of the phase. This phase represents the major execution in the Solve phase. It includes 1618 executed patterns. This indicates that the communication patterns used in this phase are highly homogeneous.
S_{25}	This phase is very short and performs only one communication pattern and the main routine that is executed is hypre_SMGResidual.
S_{26}	*Reduce* collective communication is used to track the timing (hypre_PrintTiming and hypre_EndTiming) information to mark the end of the initialization phase.

Figure 7 shows the main execution phases in the program where the length of each phase is based on the total execution time spent during that phase. The Finalize phase did not involve any inter-process communication. It started after the completion of the HYPRE_StructSMGSolve routine. It was identified based on the routine call tree where we considered the first sub-tree after all the communications as the Finalize phase. The Finalize phase contains the MPI_Finalize routine that is responsible for the termination of the MPI communication and also other routines that are responsible for the destruction of the grid that was constructed in the initialization phase.

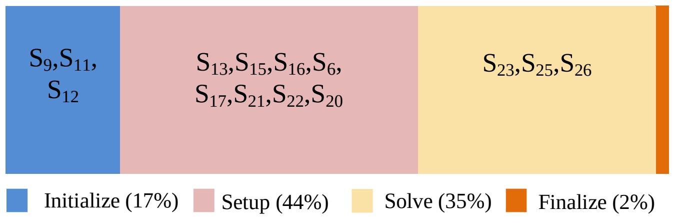


Figure 7. Detected Phases

I. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new approach for detecting execution phases in MPI programs based on the sequence of communication patterns extracted from MPI execution traces.

We presented all the steps that are needed in order to detect the execution phases along with an illustrative example. We validated the results of our phase detection approach on a trace of SMG2000 with respect the documented phases in [24]. Our phase detection approach did not only detect the main program phases but also the sub-phases.

In the future, we intend to improve the phase detection approach by applying a different segmentation technique for segmenting long homogeneous sequences such as the one found in S_{23}. This will provide a more detailed view of these types of long phases.

Moreover, we intend to further reduce the number of communication patterns by measuring the similarity among them. This will reduce the number of distinct patterns in the sequence which will result in a more homogeneous sequence which will affect the number of detected sub-phases accordingly.

We also intend to experiment with various segmentation strengths and study the effect of changing *s* on the resulting phases. Finally, we need to experiment with more systems and also compare our results with other studies.

Acknowledgment: This work is supported partially by the Natural Science and Engineering Council (NSERC) of Canada.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Q. Xu, J. Subhlok, R. Zheng, and S. Voss, "Logicalization of communication traces from parallel execution," In *Proc. of the 2009 IEEE International Symposium on Workload Characterization, (IISWC)*, pp. 34-43, 2009.

[5] TAU, Tracing and Analysis Utility. URL: http://www.cs.uoregon.edu/Research/tau.

[6] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, 27, pp. 379-423, 1948.

[7] W. Li, P. Bernaola-Galvan, F. Haghighi, I. Grosse, "Applications of recursive segmentation to the analysis of DNA sequences," *Journal of Computers & Chemistery,* 26, pp. 491-510, 2002.

[8] 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*, 2007.

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

[10] N. Palma, "Performance Evaluation of Interconnection Networks using Simulation: Tools and Case Studies," *PhD Dissertation*, Department of Computer Architecture and Technology, University, Spain, 2009.

[11] D. Kranzlmüller, "Event Graph Analysis for Debugging Massively Parallel Programs," *PhD Dissertation*, GUP Linz, Johannes Kepler University Linz, Austria, 2000.

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

[13] 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.

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

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

[16] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms," *The MIT Press*, Cambridge, MA, 1990.

[17] I. Grosse, P. Bernaola-Galván, P. Carpena, R. Román-Roldán, J. Oliver, H.E. Stanley, "Analysis of symbolic sequences using the Jensen-Shannon divergence," *Physical Review*. E, 65, 041905, 2002.

[18] H. Akaike, "A Bayesian analysis of the minimum AIC procedure.," *Annals of the Institute of Statistical Mathematics*, 30 (Part A), pp. 9-14, 1978.

[19] E. Terzi, P. Tsaparas, "Efficient Algorithms for Sequence Segmentation," *In Proc. of the SIAM International Conference on Data Mining,* pp. 314-325, 2006.

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

[21] Advanced Simulation and Computing Program: The ASC SMG2000 benchmark code. URL: http//www.llnl.gov/asc/purple/benchmarks/limited/smg/, 2001.

[22] 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, Bonn, Germany, 2006.

[23] 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.

[24] A. 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.

[25] VampirTrace, ZIH, Technische Universitat, Dresden, http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih.

[26] R. Gray, "Entropy and information theory," 2nd Edition, New York, Springer, 2011.

[27] J. Roberts and C. Zilles., "TraceVis: an execution trace visualization tool," *In Proc. of Workshop on Modeling, Benchmarking and Simulation (MoBS)*, Madison, USA, 2005.

[28] M. Noeth et al., "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, 69(8), pp. 696-710, 2009.

[29] 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.

[30] 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.