Pattern Recognition Techniques Applied to the Abstraction of Traces of Inter-Process Communication

Luay Alawneh and Abdelwahab Hamou-Lhadj
Software Behaviour Analysis 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*— The large number of processors in high performance computing and distributed applications is becoming a major challenge in the analysis of the way an application's processes communicate with each other. In this paper, we propose an approach that facilitates the understanding of large traces of inter-process communication by extracting communication patterns that characterize their main behavior. Two algorithms are proposed. The first one permits the recognition of repeating patterns in traces of MPI (Message Passing Interaction) applications whereas the second algorithm searches if a given communication pattern occurs in a trace. Both algorithms are based on the n-gram extraction technique used in natural language processing. Unlike existing work, our approach operates on the trace as it is generated (i.e. on the fly) and does not require complex and computationally-expensive data structures. We show the effectiveness and efficiency of our approach in detecting communication patterns from large traces generated from three target systems.**

Keywords- ***Trace of Inter-Process Communication; Pattern Recongintion; Message Passing Interface; Dynamic Analysis; Program Comprehension***

I. Introduction

Understanding how processes communicate in HPC (High Performance Computing) distributed applications can be a difficult task due to the large number of processes involved and the various ways they can communicate [1]. Recently, there has been an increase in the number of tools that focus on studying execution traces generated from HPC systems to help understand their behavior. Most of these studies rely on some sort of visualization methods to help engineers navigate through traces in an efficient way (e.g. [2, 3]). These approaches, however, have limited use when applied to large traces – As the trace size grows, usually due the number of processes involved, the visualization tool often creates clutter and conceals meaningful content, making it difficult to follow the process interactions [2, 3, 4-6]. There is clearly a need for trace abstraction methods that can reduce the clutter and hence facilitate the understanding and analysis of trace content despite the trace being massive.

In this paper, we propose a trace abstraction approach that is based on extracting communication patterns from inter-process communication traces. A pattern can be defined as a sequence of events repeated non-contiguously in a trace. Patterns have been shown to be an excellent way to understand the content of large traces since they often describe high-level concepts of the program under study [7]. Pattern recognition techniques have also been used successfully in the past to simplify the analysis of routine call traces [30]. This led us to embark on investigating ways to apply a pattern recognition approach to abstracting out the content of inter-process communication traces.

More specifically, we present, in this paper, two algorithms that aim to facilitate the understanding and analysis of inter-process communication traces. The first algorithm detects the exact repeating patterns in a trace. Detecting such patterns should reduce the effort to understand the program behavior by allowing software engineers to focus on examining the extracted communication patterns first before they decide to explore other aspects of the trace (if this would be needed at all). The second algorithm detects patterns in a trace that are similar to a pre-defined pattern (i.e., a known communication pattern provided as input). The objective is to allow software engineers to verify whether the traced scenario implements a specific communication pattern or not. This is particularly important in the context of distributed systems since some applications are implemented according to known (and documented) process communication topologies.

There exist some studies that also focus on detecting communication patterns in inter-process communication traces [4-6]. These studies, however, suffer from scalability problems due to the complexity of the algorithms they use and which depend on the use of computationally-expensive data structures (e.g. suffix trees) that are inefficient in terms of time and space [8]. Existing approaches also detect patterns in inter-process communication traces in a post-mortem manner. In other words, they process the trace after it has been entirely generated. This is often impractical for very large traces. In this paper, we present very efficient pattern recognition algorithms and that can process the trace as generated (i.e., on the fly). Our algorithms are based on n-gram extraction techniques, a concept used extensively in the area of natural language processing [9, 10]. The n-gram extraction approach has also been shown to be useful in detecting DNA patterns [11] and patterns in musical notes [12] efficiently.

Also, in this paper, we focus on target Single Program Multiple Data (SPMD) HPC applications that use the MPI (Message Passing Interface) standard [13]. MPI is the de facto standard for inter-process message passing paradigm, and it is used in most of today's distributed systems.

The rest of the paper is organized as follows. In Section 2, we present the related work followed by a definition of communication patterns in Section 3. In Section 4, we present our overall approach and describe the algorithms we have developed. In Section 5, we present the effectiveness of our approach by applying it to traces generated from three different systems. We conclude our work in Section 6.

II. Related Work

In this section, we present the research studies that cover trace abstraction techniques in parallel systems based on pattern detection and how these studies differ from ours.

Preissl et al. [4] proposed an algorithm for the detection of patterns in MPI traces. Their approach is based on detecting maximal repeats using compressed suffix trees. Then, they find the occurrences of the maximal repeats in the trace to be used in the communication pattern construction. Our approach does an on-the-fly detection of the maximal repeats and their occurrences in the trace. Also, we present an algorithm for finding similar patterns in the trace which is not targeted in their work.

Knüpfer et al. [14] proposed an algorithm based on the compressed complete call graph (cCCG) and the pattern graph (a derivative of the cCCG). This approach only looks for contiguous repetitions and does not detect those that are found interspersed in the trace. This approach is not comprehensive since it is necessary to detect non-contiguous repetitions, which will be more useful in reflecting the behavior embedded in a trace.

Roberts and Zilles [15] presented a trace visualization tool called TraceVis. It uses the trace graph (visualizes execution traces as instructions flowing through the processor pipeline) to detect regions of similar inter-process communications and processor activity. In this approach, pattern detection depends on the user's ability to identify similar inter-process communications in the trace. Though this may be possible for small traces, dealing with large traces that involve a large number of processes is almost always impossible using this approach.

Kunz et al. [5] presented a technique based on finite state automata to find patterns in the trace that match an input pattern. A problem with using finite state machines is the scalability problem for very large traces. Also, this work did not propose an algorithm for detecting repeating patterns in the trace.

Ma et al. [6] proposed an approach for comparing the communication patterns found in the traces generated from different systems in order to find the degree of similarity between them. The degree of similarity between two traces is measured using the correlation coefficient followed by an undirected communication graph that depicts the communication topology among the processes. Then, the similarity between the generated graphs is determined using graph isomorphism metrics. Our work targets the detection of repeating and similar patterns in a single trace file. In the future, we intend to compare communication patterns generated from different systems.

Köckerbauer et al. [16] proposed the use of a pattern matching technique to simplify the debugging of large message passing parallel programs by identifying patterns in the trace file that are similar to a predefined pattern. First, the user specifies a description of the communication pattern to be searched for in the trace file. This pattern description is then translated to abstract syntax trees. The ASTs are then scaled up to the number of processes in the trace (or the number of the target processes in the trace). The pattern matching process is run on each process trace individually. In their work, they used a hash-based search to detect exact and similar patterns on each process trace. Finally, the matching patterns are merged in order to get the communication pattern which should be exact or a variation of the specified pattern by the user.

Moore et al. proposed a pattern matching method for detecting patterns of inefficient behavior based on wait states in order to be used in KOJAK (a performance analysis tool for high performance parallel applications) [17]. These patterns of inefficient behavior are identified by converting the trace into a compact call-path profile which classifies patterns based on the time spent. This approach only looks for events that cause performance degradation and does not focus on the inter-process communication.

III. Communication patterns in MPI Traces

We define a pattern as a sequence of events that are repeating non-contiguously in a trace. We use the term *process pattern* to refer to a sequence of events that are repeated non-contiguously in one particular process in the trace file. We use the term *communication pattern* to refer to a group of process patterns that construct an inter-process communication pattern. For example, in Figure 1, Process 1 (P1) has two events (Send to P2 and Receive from P2) that are repeating non-contiguously in this sample trace. Similarly, the figure shows that the rest of the processes have non-contiguous repetitions of MPI communication events. Considering these process patterns, a communication pattern is constructed by combining the partner events among the processes. Therefore, Figure 1 shows a communication pattern repeated twice (each instance is shown in a dashed rectangle). The solid bars represent different events that are not part of the pattern.

MPI communication patterns may involve point-to-point (operations that involves only two processes) and/or collective operations (operations that involve all the processes). For example, a communication pattern may only involve MPI collective operations such as *MPI_Bcast* (an

MPI operation that can be used by a process to broadcast a message to all other processes), *MPI_Gather* (this is used by a process to collect information from other processes), which will be repeated on all the processes in the group of processes involved in this communication.

It should also be noted that a communication pattern depicts how the processes are communicating and not what data they are exchanging. For example, each pattern in Figure 1 may have different data but the processes are still communicating based on the same pattern.

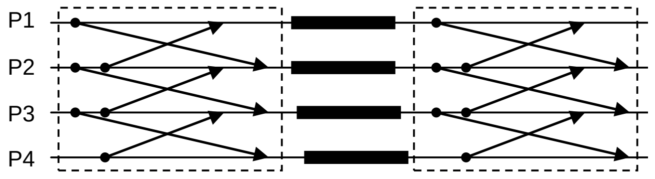


Figure 1. A Communication Pattern

In addition, some known communication patterns are well documented in the literature [18]. They are often used as guidelines on the proper way to implement in an inter-process communication mechanism (for more details about the list of documented communication patterns, please refer to [18]). For example, Figure **2**a presents the wavefront communication pattern that is used to sweep data from the first node to the last node diagonally as depicted in the 2D virtual topology in 2b. This pattern does not have a heavy communication load as only a few processes send messages at the same time.

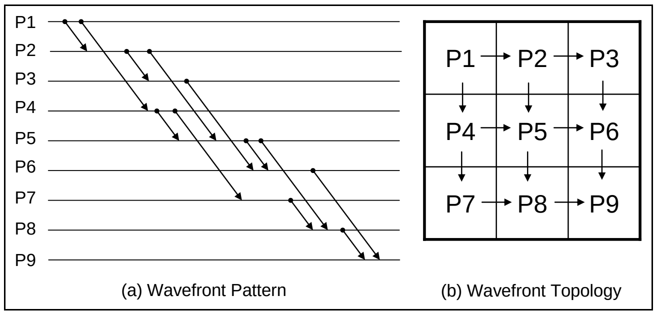


Figure 2. The wavefront pattern and topology

Figure 3 shows another example of a documented communication pattern, and which presents two patterns that are used in implementing collective communications. The Binary Tree pattern Figure 3a is used to implement All-to-One MPI collective operations. For example, the *MPI_Reduce* operation is implemented using this pattern. The Butterfly Pattern shown in Figure 3b is communication pattern that is used to implement All-to-All MPI collective operations.

Understanding the communication patterns (whether they are documented or not) that exist in an MPI program can help software engineers in debugging MPI applications and in performance optimization [4]. For example, a software engineer may decide to replace a point-to-point communication pattern by collective operations. Also, communication patterns can play an important role in revealing the process communication topology which usually helps in understanding the structure of the MPI program.

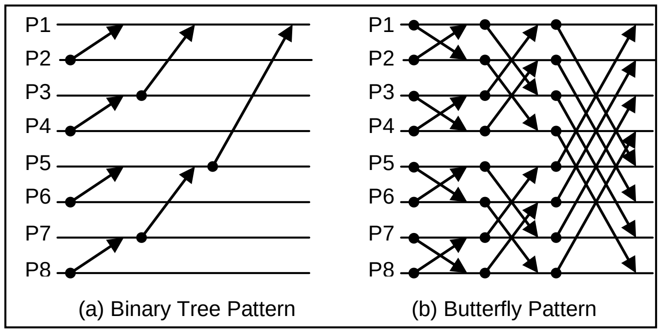


Figure 3. Examples of known communication patterns

IV. Pattern Recognition Approach

In this section, we describe the overall approach of applying pattern recognition techniques for extracting communication patterns from MPI traces. We also present the algorithms that we have developed to achieve this.

A. *Overview of the approach*

Our approach for detecting patterns in MPI traces and for searching if a given pattern exists in a trace is shown in Figures 4a and 4b respectively. The former aims to detect any non-contiguous repetitions of events in an MPI trace, no matter if the detected patterns are among the ones that are documented or not. The latter can be used by software engineers to verify if the processes in the traced scenario communicate according to a known communication pattern. We expect that software engineers would most likely use this capability to detect the existence of documented communication patterns (such as the wavefront pattern, the butterfly pattern, etc.) in a trace.

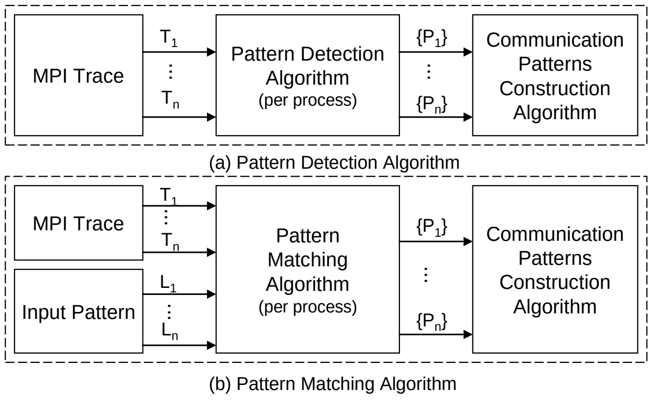

MPI Trace
T1
Tn
Pattern Detection Algorithm (per process)
{P1}
{Pn}
Communication Patterns Construction Algorithm
(a) Pattern Detection Algorithm
MPI Trace
Input Pattern
T1
Tn
L1
Ln
Pattern Matching Algorithm (per process)
{P1}
{Pn}
Communication Patterns Construction Algorithm
(b) Pattern Matching Algorithm

Figure 4. Pattern detection and pattern matching approach

The steps of our approach start in both cases by decomposing the input MPI trace into *n* trace files ($T_1 \ldots T_n$), each corresponding to a process in the trace. During this step, we also preprocess the information contained in a trace by removing contiguous repetitions due to loops, and by removing the message envelope (message size, tag and data type) since we are only interested in the way the processes communicate independently from the data they exchange. The pattern detection algorithm is used to detect repeated sequences in each process. The pattern matching algorithm is used to find the patterns in a trace that match a given pattern. In this case, the input pattern is also decomposed into *n* process patterns *($L_1 \ldots L_n$)*. Each process pattern L_i is compared to its process trace file T_i in order to extract its similar patterns. Note that the patterns do not have to be identical. A measure of similarity is discussed later in the paper.

After extracting the patterns from each process trace (for both algorithms), they are used as input for the communication patterns construction algorithm to generate the inter-process communication patterns. The three algorithms will be explained in detail in the following subsections.

B. *The Pattern Detection Algorithm*

Our algorithm for detecting patterns in each process trace operates on each process trace individually and detects their non-contiguous repetitions (process patterns). The MPI events in the trace file can be seen as a stream of data. This led us to use the concept of n-grams applied in statistical natural language processing to detect the patterns in an MPI trace. Our approach, however, extends existing applications of the n-gram extraction algorithms by varying the sizes of the n-gram dynamically whenever a pattern is detected, which is similar to the approach used in the Lempel-Ziv-Welch data compression algorithm [20]. This is particularly important in our approach since, unlike existing pattern recognition techniques used for MPI traces, our proposed algorithm has the advantage of operating on the trace as it is generated (i.e., on the fly), and it is capable of detecting repeating patterns in one single pass.

In a given sequence, an n-gram is a subsequence of *n* items in that sequence. In our study, the smallest n-gram that may be considered as a repeat is a bi-gram (two consecutive MPI events). We consider a repeat is a sequence that appears at least twice in a given string without overlapping. There are several types of repeats that may exist in a stream of data. We consider the following types of repeats that will be used later in the pattern detection algorithm. Let consider*p1* as the start position of substring 1, *p2* the start position of substring 2, and *l* is their length):

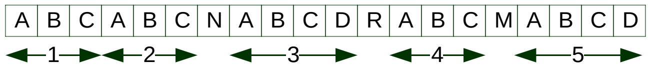
1. Tandem Repeats: repeats that are directly adjacent to each other.
   Given a string *S* of length *n*, a Tandem pair in S is a tuple (*p1*, *p2*, *l*) such that
   $\exists$ S[p1 .. p1 + l – 1] = S[p2 .. p2 + l – 1] and p2 > p1 and S[p2 - 1] = S[p1 + l – 1]. This type of repeat is usually generated from loops in the program.

2. Maximal Repeat (Interspersed Repeats): a *repeat* that cannot be extended to the left and to the right.

   Given a string *S* of length *n*, a maximal pair in S is a tuple (p1, p2, l) such that
   $\exists$ S[p1 .. p1 + l – 1] = S[p2 .. p2 + l – 1] and p2 > p1 and S[p1 + l] $\neq$ S[p2 + l] and S[p1 - 1] $\neq$ S[p2 - 1]

3. Super Maximal Repeat: a *maximal repeat* that does not occur in any other *maximal repeat*.

We show an example of the three types of the aforementioned repeats:



As we can see, 'ABC' is a maximal repeat that occurs three times in the string at 1, 2, 4. The occurrence at 2 is a tandem (contiguous) repeat since it is directly following the first occurrence of 'ABC'. Also, 'ABCD', which is found at 3 and 5, is a super maximal repeat since it contains another maximal repeat and is not contained in any other maximal repeat.

The process pattern detection algorithm is presented in Algorithm 1. We use three main objects in the algorithm. The n-gram object keeps track of the current n-gram and its position. A pattern object contains the pattern sequence, its positions in the trace and its frequency (number of occurrences). The Pattern List is the dictionary that holds the detected pattern objects. Moreover, we use two pointers that slide over the trace in order to return the next n-gram that will be used in detecting the patterns. Since the minimum length of a repeat is two, we should be able to read a bi-gram from the trace. Therefore, the two pointers are always adjacent so a bi-gram could be returned when needed. In the algorithm, we also show how the n-gram grows in size whenever a pattern is detected.

The first five lines are declarations that will be used by the algorithm. The *aNewPattern* indicates whether the current pattern is new or existing. The *aMatch* variable indicates whether the current pattern can be constructed from its prefix pattern at its previous positions (returned by the check pattern occurrences algorithm). The *tandemRepeats* is an integer value indicating how many times the current pattern is repeated contiguously right after its current position.

The algorithm starts by reading the first bi-gram, at line 6, which will be considered as the first pattern added to the detected patterns list. At line 10, the algorithm will check if the detected pattern is repeated contiguously in the following events in the trace. If the pattern is repeated contiguously more than once, then the two pointers will advance *((repeats - 1) * pattern size)* steps forward in the trace. The pointers will start at the beginning of the last detected tandem repeat since it may be part of a bigger pattern. The algorithm will repeatedly read the next bi-grams from the trace file and add them to the pattern list until a bi-gram match is detected. In this case, the algorithm will enter the *do-while* loop at line 15 and will add the next event from the trace to the right of the matching bi-gram which will result in a tri-gram. This occurs by the call to the *ConstructNGram* function at line 18, which is a utility function that constructs the n-gram.

```
Pattern Detection: this algorithm runs for each
process separately to find repeating patterns
Γ  checkPatternOccurence
advanceSteps = (tandemRepeats - 1) * patternSize
1.  PatternList: List of extracted patterns
2.  aNewPattern: Boolean
3.  aMatch: Boolean
4.  tandemRepeats: Integer
5.  currentPattern: Pattern
6.  while(next n-gram is not null){
7.     p = position of nextNGram
8.     aNewPattern = UpdatePatternList(nextNGram, p)
9.     currentPattern = getPattern(nextNGram)
10.    tandemRepeats = checkTandem(currentPattern)
11.    if (tandemRepeats > 1) then
12.      advancePointers(advanceSteps)
13.    end if
14.    if aNewPattern is false then
15.      do{
16.         aMatch = false
17.         currentPattern = getPattern(nextNGram)
18.         nextNGram  = constructNGram(nextNGram)
19.         UpdatePatternList(nextNGram , p)
20.         nextPattern = getPattern(nextNGram)
21.         aMatch = Γ(nextPattern,p, currentPattern)
22.         tandemRepeats = checkTandem(currentPattern)
23.         if (tandemRepeats > 1) then
24.            aMatch = true
25.            advancePointers(advanceSteps)
26.         end if
27.         if aMatch is false then
28.            remove nextPattern from PatternList
29.         end if
30.      } while(aMatch)
31.    end if
32. end while
```

Algorithm 1. Pattern Detection Algorithm

Then, the algorithm will check whether the tri-gram can be constructed from the previous occurrence of its bi-gram by calling the *checkPatternOccurence* (shown as Γ in the algorithm) function at line 21. In the *checkPatternOccurence* function, if the previous occurrence of the bi-gram can be constructed to match the detected tri-gram, the frequency of the tri-gram pattern will be incremented and the frequency of the bi-gram will be decremented. Since we have a repeating tri-gram, the algorithm will read the next event and add it to the tri-gram (line 18) and again check if the previous occurrence (line 21) of the tri-gram can be extended to match the new quad-gram. Again, at line 22, the algorithm will check whether the new constructed pattern has a tandem repeat or not, if yes, the two pointers will be advanced as described previously. As can be seen from the algorithm, the n-gram will grow in size whenever it has a match in the pattern list. If the constructed n-gram cannot be detected at any previous position of its prefix n-gram, then it will be removed from the pattern list at line 28.

We also present the C*heck Pattern Occurrence* in Algorithm 2. This algorithm is being called by the code presented in Algorithm 1 as *'checkPatternOccurence'* or Γ function. It is used to detect if the new pattern can also be detected at the previous positions of its prefix patterns (e..g., for a pattern '*abcd'* its prefix pattern is '*abc'*). If the pattern can be detected at the previous positions, the algorithm will return *true*.

```
Check Pattern Occurrence: checks if nextPattern can be
constructed from the previous positions of current Pattern.
Returns true if nextPattern can be found at its prefixPattern
previous positions
Signature: nextPattern, nextPatternPosition, prefixPattern
1.  curPosition: position of the prefixPattern
2.  aMatch = false
3.  for each curPosition of prefixPattern positions{
4.    if curPosition EQUALS nextPatternPosition then
5.       continue // get next position
6.    if nextPattern has curPosition then
7.       aMatch = true
8.       prefixPattern.decrementFrequency
9.       prefixPattern.removePosition(curPosition)
10.      continue //get next position
11.   end if
12.   currentNGram = prefixPattern.getNGram
13.   currentNGram.position = curPosition
14.   add next unigram to currentNGram at curPosition
15.   if nextPattern.NGram EQUALS currentNGram then
16.      aMatch = true
17.      prefixPattern.decrementFrequency
18.      prefixPattern.removePosition(curPosition)
19.      nextPattern.incrementFrequency
20.      nextPattern.addPosition(curPosition)
21.   end if
22. end for each
23. return aMatch
```

Algorithm 2. Check Pattern Occurrences Algorithm

The algorithm will iterate on the positions of the prefix pattern in order to find whether the next pattern can be detected at these positions (line 3). Line 4 makes sure not to continue the iteration when the prefix pattern position is the same as the next pattern position. Also, lines 6 through 10 make sure not to continue in the current iteration if next pattern already has the current position *curPosition*. If none of the conditions at line 4 and 6 is true, then the next unigram in the trace that follows the prefix pattern at *curPosition* will be appended to prefix pattern. Whenever the prefix pattern can be extended to match the new pattern, the frequency of the prefix pattern is decremented and its position is removed (lines 17 to 20 in Algorithm 2). In the following, we demonstrate using a short example how the n-gram based algorithm is able to detect the different types of repeats in the trace.

Figure 5 presents a trace of 17 point-to-point communication events (S2 means Send to 2 and R2 means Receive from 2). The algorithm starts by reading the first bi-gram ‘S2, S3’ at position 1 and add it as a new pattern to the pattern list. Since there is no contiguous repeat for the pattern, the next bi-gram ‘S3, R2’ will be read and added as a new pattern.

Trace:

1	2	3	4	**5**	6	7	**8**	9	10	**11**	12	13	14	**15**	16	17
S2	S3	R2	S5	S2	S3	R2	S2	S3	R2	S2	S3	R2	S4	S2	S3	R2

Execution:

#	Pattern				New?	Has Tandem?	Freq.	Pos.	Next Action
1	S2	S3			Yes	No	1	1	next bi-gram
2	S3	R2			Yes	No	1	2	next bi-gram
3	R2	S5			Yes	No	1	3	next bi-gram
4	S5	S2			Yes	No	1	4	next bi-gram
5	S2	S3			No	No	2	1, 5	Construct from current n-gram
6	S2	S3	R2		Yes	Yes	4	1, 5, 8, 11	Append event at position 14
7	S2	S3	R2	R4	Yes	No	1	11	next bi-gram
8	S4	S2			Yes	No	1	14	next bi-gram
9	S2	S3			No	No	2	8, 12	Construct from current n-gram
10	S2	S3	R2		No	No	5	1, 5, 8, 11, 15	End of Trace

Result:

Detected Pattern			Frequency	Positions
S2	S3	R2	5	1, 5, 8, 11, 15

Figure 5. Process Pattern Detection Example

Similarly, there is no contiguous repeat for this new pattern, therefore the algorithm will continue reading until it reads ‘S2, S3’ at position 5. Since this is an existing pattern, its frequency will be incremented and its position will be added to the pattern positions list. Again the algorithm will check for contiguous repeats which also do not exist in this case. However, since this is an existing pattern, the next uni-gram in the trace will be added to the pattern resulting in ‘S2, S3, R2’ as a new pattern. The algorithm will detect that there are two contiguous repeats (tandem) of this pattern. Also, the check pattern occurrence function will be called and detect that at position 1 (position of prefix pattern ‘S2, S3’ this new pattern can be detected. Then, the algorithm will append the next event following the last tandem repeat which will result in the pattern found at row 7 in the *Execution* table in Figure 3. When the algorithm reaches the end of the trace, it will find that ‘S2, S3, R2’ is the only maximal repeat with frequency more than 1 in the trace. This example shows how the n-gram-based algorithm is able to detect patterns (maximal repeats) in trace files of MPI applications.

C. *The Pattern Matching Algorithm*

In this section, we present our algorithm for extracting similar communication patterns in an MPI trace to a predefined input pattern. The pattern under study can be provided by the user or it can be provided from the list of patterns detected using our first algorithm presented in the previous section. The input communication pattern is stored as a list where each entry corresponds to the sequence of events of one process only. These events are inter-process communication events such as this send event ‘MPI_Send (target = P5, Size = 256)’.

Similar to the pattern detection algorithm, this algorithm detects similar patterns on each process trace separately. The output of this algorithm is input to the communication pattern construction algorithm presented in the next section. The degree of similarity between the patterns is determined by the number of shared events between them.

We use the Edit Distance [21] (also known as Levenshtein Distance) function to calculate the degree of similarity between the two patterns. In order to determine the areas in the trace that could potentially match the input pattern, we use the Lemma proposed by Jokinen and Ukkonen [22] for our filtration process. This Lemma is based on calculating the shared n-grams between the pattern and the target string. Several research studies for approximate string matching exist that are based on this Lemma [23, 24]. The Lemma is presented in the following:

<u>Lemma:</u> N-gram based Filter (Jokinen and Ukkonen [22])

Let a string *S1* of length *m* with at most *k* edit distance from another string *S2* of length *m*, then at least $m + 1 – kn + n$ of the n-grams in *S1* occur in *S2*.

The process of determining similar patterns consists of two steps. The first step is the filtration (fast) process which uses the above lemma, and the second step is the *edit distance* function (slower). We slide a window of length m, which is the length of the input pattern on a process trace until there is a potential match (window shares at least $m + 1 – kn + n$ with the pattern). A window that is identified as a potential match is verified using the edit distance function.

In order to reduce the number of verified windows, and to reduce the total execution time consequently, we use positioned n-grams to preprocess the pattern. We build a table for all the n-grams in the pattern with their positions in the pattern. We use the positioned n-grams table in the filtration process to shift the window to the right (in the trace) based on the position of the first n-gram found in the window under test. For example, if the position of the n-gram in the n-gram table is 3 and the same n-gram was found at position 5 in the window, then we slide the window to the right by two steps in order to avoid verifying two non-matching windows using the edit distance function.

Algorithm 3 describes our procedure for detecting communication patterns that are similar to a pattern *P*. As mentioned previously, this algorithm runs for every process separately. In line 5, it will iterate on each window in the trace. The window (w) may shift to the right based on its position in the n-gram positioned table (lines 6-8). Based on the number of shared n-grams between the pattern and the window determined in line 9, the edit distance will be computed in line 11. If edit distance is less than or equal to *k*, then the window *w* will be added to the *MatchingPatternList* at line 12 and the window will be shifted to start at the next adjacent window at line 13. Every process in the MPI trace should have its own *MatchingPatternList* which will be used in the algorithm described in the next section for the communication patterns construction. The *MatchingPatternList* contains the patterns and their start positions in the trace.

```
Pattern Matching: this algorithm runs for each
process separately to find similar patterns
p: pattern under study of size m
threshold = pattern size – n + 1 – k.n
k:maximum allowed edit distance
n: n-gram size
firstSharedNGramDisplacement: displacement between
position of first shared q-gram in w and its position in
the q-gram position table
1.  w: window of size m
2.  MatchingPatternList: List of matched windows
3.  // MatchingPatternList also holds the position of w
4.  sharedNGrams: Integer
5.  while(next w is not null){
6.      if (firstSharedNGramDisplacement > 0) then
7.        shiftWindow(firstSharedNGramDisplacement)
8.      end if
9.      sharedNGrams = countSharedNGrams(p, w)
10.     if sharedNGrams > threshold then
11.         if editDistance(p, w) <= k then
12.             add w to MatchingPatternList
13.             jump to next adjacent window
14.         end if
15.     end if
16. end while
```

Algorithm 3. Pattern Matching Algorithm

We show the correctness of the algorithm using the example shown in Figure 6. We used alphabets instead of MPI events for simplicity. The figure shows the input pattern and to its right its n-grams along with their positions (n-gram position table). The window size is the same as of the size of the pattern. We slide the window on the string and find the number of shared n-grams. For window #12 and window #22, the window is shifted to the right based on the position of the '*ab'* n-gram (line 7 in the algorithm). Also, since a match was detected at window # 16 with *k* = 1, the window was shifted to point at window # 22.

This example shows the usefulness of using the concept of n-grams in the filtration step. The filtration step reduces the execution time since it reduces the number of windows to be checked using the edit distance function for all the windows in the trace. The filtration step could be improved in order to avoid checking non-matching windows using the edit distance function. One more issue that needs to be tuned is the window size. In some cases, the window size should be decreased to minimum of (*m – k).* For example, window # 9 'b c d e f y' has an edit distance of 2 while if we consider the window as 'b c d e f' (size is *m – k + 1*) then the edit distance will be 1 which increases the degree of similarity to the input pattern. The same can be done for window # 10 since 'c d e f' has an edit distance of 2 while 'c d e f y e' has an edit distance of 5. Currently, we are handling these cases in another step (after the execution of the algorithm) by checking windows with at most *2k* edit distance and reducing there window size to verify if a shorter window may have a similar match to the input pattern. However, we have to keep in mind that a matching window may be contained in a larger pattern which is not the same as the input pattern. Therefore, the software engineer should be informed that a group of windows are similar to or match the input pattern but they exist in a larger pattern in the trace which means that the input pattern may be a subset of some patterns in the trace.

Input Pattern: a b c d e f → **0**: a b, **1**: b c, **2**: c d, **3**: d e, **4**: e f
Trace: a b c d m h k o b c d e f y e a b h d e f r s a b c d e f

m = 6, n = 2, k = 1, t >= m – n + 1 – kn → **t >= 3 shared n-grams**

W#	Window	Shared n-grams	ED	Action
1	a b c d m h	ab, bc, cd	2	
2	b c d m h k	bc, cd		Skip window
3	c d m h k o	cd		Skip window
4	d m h k o b			Skip window
5	m h k o b c	bc		Skip window
6	h k o b c d	bc, cd		Skip window
7	k o b c d e	bc, cd, de	3	
8	o b c d e f	bc, cd, de, ef	1	
9	b c d e f y	bc, cd, de, ef	2	
10	c d e f y e	cd, de, ef	5	
11	d e f y e a	de, ef		Skip window
12	e f y e a b	ab at position 4	4	Jump to w#16
13	f y e a b h			
14	y e a b h d			
15	e a b h d e			
16	a b h d e f	ab, de, ef	1	Jump to w#22
17	b h d e f r			
18	h d e f r s			
19	d e f r s a			
20	e f r s a b			
21	f r s a b c			
22	r s a b c d	ab at position 2		Jumpt to w#24
23	s a b c d e			
24	a b c d e f	ab,bc,cd,de,ef	0	Done

Figure 6. Example of applying the pattern matching algorithm

Once all the similar patterns were detected for each process. We start building the communication patterns using the *Communications Patterns Construction* algorithm presented in the next section. In order to consider the communication pattern as a similar match, we need to check whether the total edit distance (sum of edit distance from each process similar match) is still within the specified threshold. This is computed by relating the total number of errors (differences) to the total number of events in the constructed communication pattern. Therefore, some similar patterns per process may be within the specified threshold but their communication pattern may have an error that is larger than the threshold.

D. *Communication Patterns Construction Algorithm*

In this section, we present the algorithm for assembling the process patterns detected either through the pattern detection algorithm or the pattern matching algorithm into communication patterns that encompass all the communicating processes. We input the process detected patterns (detected in the previous steps) into this algorithm and start iterating on all corresponding patterns (for pattern *p1*, its corresponding patterns are those patterns that have partner events with *p1*) until a communication pattern is constructed. When using this algorithm to construct each process patterns detected using the pattern detection algorithm presented in Section 5, the output will be the set of all communication patterns that are repeating in the trace. On the other hand, when using this algorithm to construct the similar matching patterns on each process detected using the pattern matching algorithm presented in Section 6, the output will be the set of all communication patterns that are similar to the given input communication pattern.

The communication patterns construction algorithm is presented in Algorithm 4. It uses the ordered pattern positions list generated from the first step (any of the two previously presented algorithms). For each pattern position (line 5), the corresponding patterns on the other processes will be detected by locating their partner events. We iterate on the positions of each detected pattern since at different positions the same pattern may have different partner patterns which will result in different communication patterns. For each pattern, the algorithm will check if it is already part of a communication pattern. The communication pattern will be retrieved or a new communication pattern will be created accordingly (lines 6-11). In some cases, an event that is included in a pattern may have a partner event that is not included in any pattern. This single partner event will not be detected using the pattern detection algorithms since we consider the minimum pattern size as two events (bi-gram). This event will be added to the resulting communication pattern with the condition that its process does not have any other partner events at a dispersed location. The single events will be retrieved using the call at line 12 and they will be added to the communication pattern in the following '*for-each'* loop at line 12. At line 16, all the corresponding process patterns will be retrieved and then added (if not already exist) to the communication pattern inside the '*for-each'* loop at line 17.

```
Communication Pattern Construction: constructs the
communication patterns by iterating on the process patterns
AllProcessesPatternList: a list of all process patterns;
it contains all patterns detected in the previous step
getCorrespondingProcessPatternsList: returns the
corresponding patterns based on the partner events
1.  CommPatternList: communication  patterns list
2.  CommPattern: current communication pattern
3.  foreach ProcessPatternList in AllProcessesPatternList{
4.   foreach pattern in ProcessPatternList{
5.    foreach position i in pattern positions list{
6.      if pattern at position i is part of a Comm Pattern then
7.        CommPattern = getCommPattern(pattern, position)
8.      else
9.        create new CommPattern
10.       CommPatternList.add(CommPattern)
11.     end if
12.     getCorrespondingSingleEventList(pattern)
13.     foreach correspondingEvent{//not in a pattern
14.       CommPattern.add(correspondingEvent)
15.    end for each
16.    getCorrespondingProcessPatternsList(pattern)
17.    foreach correspondingPattern {
18.      if correspondingPattern ∉ CommPattern then
19.        CommPattern.add(correspondingPattern)
20.      end if
21.    end for each
22.   end for each
23.  end for each
24. end for each
25. ExtractDistinctCommunicationPatterns
```

Algorithm 4. Communication pattern Construction

After the algorithm finishes iterating on all the process patterns, it will output the distinct communication patterns at line 25. The resulting communication patterns may involve all or a subset of the processes in the trace.

We acknowledge that the number of resulting patterns might in some cases be considerably high and there is therefore a need to determine the most important ones for further investigation. Some factors for ranking patterns based on their importance that can be considered include the pattern frequency (most frequent patterns mean that the system depends greatly on this behavior to accomplish its objective), the number of events in the repeating pattern, the number of processes involved, etc. In this paper, we focused on presenting algorithms for detecting repeating and similar patterns in MPI traces. Pattern ranking is left as a subject for further studies.

V. CASE STUDY

We used three traces generated from three different applications to validate our proposed algorithms. In the following, we present our trace analysis results on each system trace. Our experiments were performed on a 1.83 GHz Intel Core 2 Duo CPU with 3.0 GB of RAM.

A. *Weather Research & Forecasting (WRF)*

We tested our pattern detection algorithm (presented in Section 5) on a trace file generated by the VampirTrace [25] trace analysis tool. The trace file had 336960 point-to-point events. It was generated from the Weather Research and Forecasting (WRF) Model (a collaborative effort from different partners in order to provide operational forecasting and atmospheric research needs) system. The tested instance of the application contains 16 processes. The trace file format used by the VampirTrace is called the Open Trace Format (OTF) [26] which comes with several APIs for reading the trace data.

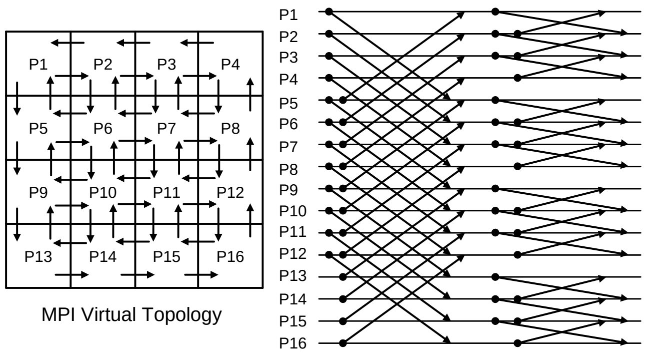


Figure 7. Detected Repeating Pattern in WRF

In this trace, we detected two main patterns, one consists of point-to-point operations and the other one is composed of collective operations. The right side of Figure 7 depicts the point-to-point communication pattern found in the WRF trace. The left side shows that the MPI virtual topology that is used in implementing the traced feature consists of a 2D-mesh were each process communicates with its direct neighbors only. The communication pattern and the topology facilitate the understanding of the communication behavior in the program. The total execution time to detect this pattern was 37 seconds which is still reasonable considering the trace length.

Our analysis shows that this repeating pattern exists in different contexts of the program. Here, a context means the function that the pattern occurs in. The detected pattern is repeated 3510 in the trace file. The point-to-point communication pattern exists in the START_DOMAIN_EM and SOLVE_EM functions. START_DOMAIN_EM is called once in the program and SOLVE_EM function is called 100 times. The START_DOMAIN_EM call occurs before the SOLVE_EM calls. The detected pattern in the execution trace helped us locate the important communications in the program. These inter-process communications were used in setting up the data to compute several weather parameters such as moisture coefficients, the diagnostic quantities pressure and others.

The execution trace contained two collective patterns (patterns from MPI collective operations) as shown in Figure 8. The *root* process in the collective operations is *P1*. Moreover, Pattern 2 shows in the first 3 elements of Pattern 1 but was detected at different locations in the trace that were not part of the occurrences of Pattern 1.

Collective Pattern 1	Collective Pattern 2
MPI_Bcast MPI_Gather MPI_Gatherv MPI_Gather MPI_Scatterv	MPI_Bcast MPI_Gather MPI_Gatherv
60 repetitions	116 repetitions

Figure 8. Detected Collective Pattern

B. *Scalar Pentadiagonal (SP) - NAS Parallel Benchmark*

We also tested the algorithm presented in Section 5 on another trace generated from the Scalar Pentadiagonal (SP) included in the NAS Parallel Benchmark [27]. This trace contained 38544 point-to-point events. SP solves three sets of uncoupled systems of equations in the *x*, *y*, and in the *z* dimensions starting with the x-dimension. The trace file was collected from a run that consisted of 4 processes collaborating in order to solve a synthetic computational fluid dynamics (CFD) problem.

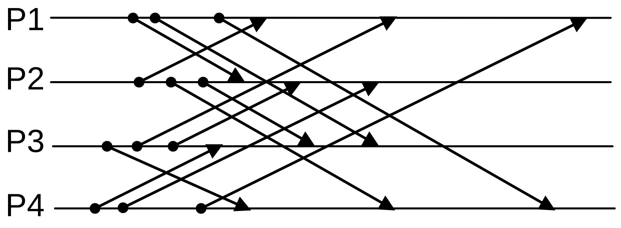


Figure 9. Detected Pattern in SP

After applying the pattern detection algorithm, we noticed that the processes of this application communicate according to the sequence shown in Figure 9. This pattern was detected in 401 non-contiguous locations in the trace. As it can be seen from the figure, all the processes communicate the solution of their parts of the equations to all other processes in the program. The total execution time to detect this pattern was 12 seconds. We intend to compare the performance of our approach to other studies in future work, although we believe, based on the study presented in [4] and where the authors tackled the problem of efficient detection of communication patterns, we believe that the obtained total execution time shows that our algorithm is considered efficient.

C. *2D Solution to Cellular Nuclear Burning – FLASH 2.0*

The largest trace file in our case study was generated from the two-dimensional implementation of the Cellular Nuclear Burning problem [28]. Flash solves complex systems of equations for hydrodynamics and nuclear burning which uses Paramesh library [29] for adaptive mesh refinement on rectangular grid. The generated trace file contained 633490 point-to-point MPI events generated from 16 processes. We were able to detect 202 distinct patterns.

Some of these patterns were repeated a few times and others were repeated for a few thousand times. Figure 8 shows a detected pattern from the same trace. The total execution time for detecting the patterns was 228 seconds. This long execution time is due to the large number of distinct patterns in the trace.

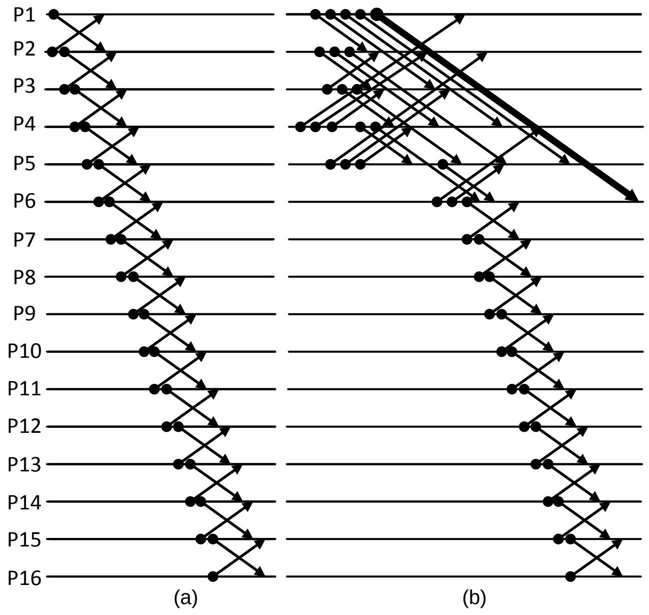


Figure 10. Two detected patterns in the 2D cellular problem

Figure 8 shows two patterns that were detected using the pattern detection algorithm. Pattern 10a is repeated 927 times and pattern 10b was only repeated 5 times in the trace. It can be seen in the two patterns that processes P6 to P15 have the same communications (process patterns). That is why in the communication pattern construction algorithm we iterate on all the positions of the detected process patterns. If not all of the positions were taken into account then some of the communication patterns will not be detected in the trace. These two patterns are used in filling the guard cells in the mesh. We also detected more complex patterns that we cannot include in this paper due to space limitation.

We also tested the pattern matching algorithm on this trace to detect similar patterns to an input pattern. In this case study, we were able to detect similar patterns that differ in message size, tag value, and that have different number of communications. For example, when considering the message envelope for pattern 8b, we detected 4 instances of the pattern when the size of the message sent from P1 to P6 is 24. The input pattern differs from the detected patterns in the message size which is 0. In this example, a maximum edit distance of 1 was allowed.

We detected many other similar patterns using the similar pattern detection algorithm. In the case studies, we found out that when n increases, the total execution time increases. This can be justified since the number of verified windows using the edit distance function increases. Moreover, in some cases, we found that the window size should be less than the size of the pattern but also not less than $m - k$ in order to have a similar match.

VI. CONCLUSION & FUTURE WORK

In this paper, we presented a new approach for detecting exact and similar patterns in MPI execution traces. Our approach is based on the concept of n-grams applied in different areas such as statistical natural language processing, DNA and Musical notes. To the best of our knowledge, this is the first work that utilizes this concept for detecting inter-process communication patterns.

We presented two algorithms and demonstrated their results on three trace files. The results showed that the n-gram concepts can be used to detect the repeating patterns in traces generated from parallel systems. In the future, we are planning on improving the similar pattern detection algorithm by applying sampling on the trace file. This will result in a faster algorithm but with a tradeoff on its accuracy thereof.

We also intend to apply the algorithms on traces that involve a larger number of processes in order to verify the second part of the algorithm for assembling the inter-process communication patterns. Finally, we will present a pattern ranking scheme to categorize the detected patterns based on different criteria.

REFERENCES

[1] K. El Maghraoui, B. K. Szymanski, 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*, pp. 258-271, 2005.

[2] The Vampir Performance Visualizer, http://www.vampir.eu

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

[4] 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 ICPP,* pp. 230–237, 2008.

[5] T. Kunz, M. F. H. Seuren, “Fast detection of communication patterns in distributed executions,” *In Proc. of CASCON*, 1997.

[6] Chao Ma, Yong Meng Teo, Verdi March, Naixue Xiong, Ioana Romelia Pop, Yan Xiang He, Simon See, "An approach for matching communication patterns in parallel applications," ipdps, pp.1-12, 2009 IEEE International Symposium on Parallel&Distributed Processing, 2009.

[7] H. Safyallah and K. Sartipi, “Dynamic Analysis of Software Systems using Execution Pattern Mining”, In Proc. 14th IEEE International Conference on Program Comprehension, 2006, pp. 84-88.

[8] Gonzalo Navarro , Veli Mäkinen, Compressed full-text indexes, ACM Computing Surveys (CSUR), v.39 n.1, p.2-es, 2007.

[9] P. F. Brown, V. J. Della Pietra, P. V. deSouza, J. C. Lai, and R. L. Mercer, “Class-based n-gram models of natural language”, Computational Linguistics, vol. 18, pp. 467–479, 1992.

[10] Kim JY, Shawe-Taylor J (1994) Fast string matching using an n-gram algorithm. Softw Pract Exper 94(1):79–88.

[11] Jie Zheng, Stefano Lonardi, "Discovery of Repetitive Patterns in DNA with Accurate Boundaries," bibe, pp.105-112, Fifth IEEE Symposium on Bioinformatics and Bioengineering (BIBE'05), 2005

[12] Nikunj Patel, Padma Mundur, An n-gram based approach to finding the repeating patterns in musical data. Proceedings of the European IMSA. Grindelwald, Switzerland, 2005.

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

[14] Knüpfer, A., Voigt, B., Nagel, W.E., Mix, H.: Visualization of repetitive patterns in event traces. In: Kågström, B., Elmroth, E.,

Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 430–439. Springer, Heidelberg (2007).

[15] J. Roberts and C. Zilles. TraceVis: an execution trace visualization tool. In Proc.MoBS 2005, Madison, USA, 2005.

[16] Thomas Köckerbauer, Christof Klausecker and Dieter Kranzlmüller, Scalable Parallel Debugging with g-Eclipse, Book Chapter, Springer Berlin Heidelberg, 2010.

[17] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. A Scalable Approach to MPI Application Performance Analysis. LNCS, 3666:309–316, 2005.

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

[19] Z. Volkovich and et al., “The method of n-grams in large-scale clustering of DNA texts,” Pattern Recognition, 2005.

[20] T. A Welch., “A technique for high-performance data compression”, Computer, Vol. 17, No. 6, pp. 8-19, June 1984.

[21] A. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” Soviet Physics Doklady, vol. 10, no. 8, pp. 707-710, 1966.

[22] Jokinen, P., and Ukkonen, E. Two algorithms for approximate string matching in static texts. In In Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science, 240–248. 1991.

[23] X. Cao, S. C. Li, and A. K. H. Tung. Indexing DNA Sequences Using q-grams. In Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA), 2005.

[24] Rasmussen K, Stoye J, Myers EW. Efficient q-Gram Filters for Finding All ϵ-Matches Over a Given Length. Journal of Computational Biology. 2006;13:296–308.

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

[26] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, W. Nagel, Introducing the Open Trace Format (OTF), in: Proc. of the International Conference on Computational Science (ICS), 2006, pp. 526–533.

[27] NAS Parallel Benchmarks, http://www.nas.nasa.gov/Resources/Software/npb.html.

[28] FLASH3 User's Guide, http://flash.uchicago.edu/website/codesupport/flash3_ug_3p2/node31.html

[29] MacNeice, P., Olson, K.M., Mobarry, C., de Fainchtein, R. and Packer, C., PARAMESH: a parallel adaptive mesh refinement community toolkit. Comput. Phys Commun. v126. 330.

[30] Abdelwahab Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension", Ph.D. Dissertation, School of Information Technology and Engineering (SITE), University of Ottawa., 2005.