TECHNIQUES TO FACILITATE THE UNDERSTANDING OF INTER-PROCESS COMMUNICATION TRACES

LU'AY ALAWNEH

A THESIS
IN
THE DEPARTMENT
OF
ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2012

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: **Lu'ay Alawneh**

Entitled: **Techniques to Facilitate the Understanding of Inter-Process Communication Traces**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________________Chair

______________________________________ External Examiner

______________________________________ External to Program

______________________________________ Examiner

______________________________________ Examiner

______________________________________Thesis Supervisor

Approved by ____

____

Dr. Robin A.L. Drew, Dean
Faculty of Engineering & Computer Science

Abstract

High Performance Computing (HPC) systems play an important role in today's heavily digitized world, which is in a constant demand for higher speed of calculation and performance. HPC applications are used in multiple domains such as telecommunication, health, scientific research, and more. With the emergence of multi-core and cloud computing platforms, the HPC paradigm is quickly becoming the design of choice of many service providers.

HPC systems are also known to be complex to debug and analyze due to the large number of processes they involve and the way these processes communicate with each other to perform specific tasks. As a result, software engineers must spend extensive amount of time understanding the complex interactions among a system's processes. This is usually done through the analysis of execution traces generated from running the system at hand. Traces, however, are very difficult to work with due to the overwhelming size of typical traces. The objective of this research is to present a set of techniques that facilitates the understanding of the behaviour of HPC applications through the analysis of system traces. The first technique consists of building an exchange format called MTF (MPI Trace Format) for representing and exchanging traces generated from HPC applications based on the MPI (Message Passing Interface) standard, which is a de facto standard for inter-process communication for high performance computing systems. The design of MTF is validated against well-known requirements for a standard exchange format.

The second technique aims to facilitate the understanding of large traces of inter-process communication by automatically extracting communication patterns that characterize their main behaviour. Two algorithms are presented. 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.

Finally, we developed a technique to abstract MPI traces by detecting the different execution phases in a program based on concepts from information theory. Using this approach, software engineers can examine the trace as a sequence of high-level computational phases instead of a mere flow of low-level events.

The techniques presented in this thesis have been tested on traces generated from real HPC programs. The results from several case studies demonstrate the usefulness and effectiveness of our techniques.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Abdelwahab Hamou-Lhadj for his constructive scientific advice, encouragement, abundance help, and invaluable assistance. His supervision throughout the different stages of my PhD studies provided me with inspiration to explore new research ideas from different research fields. I would like to thank Dr. Hamou-Lhadj for his encouragement and support.

I would like to thank the committee members for all the feedback they provided throughout the course of my PhD research.

I would like to thank all the members of the Software Behaviour Analysis (SBA) Research Lab at Concordia University for sharing knowledge and for their support.

I would like to thank my parents and my family members for the moral support to complete this research.

Special thanks to my precious wife Yasmin for her love and support throughout the entire course of my PhD research.

Dedication

To my parents
To my wonderful wife Yasmin
To my beautiful daughters Tala & Zeina

List of Abbreviations

ATEMPT	A Tool Event ManiPulaTion
BIC	Bayesian Information Criterion
cCCG	compressed Complete Call Graph
CST	Compressed Suffix Tree
CTF	Compact Trace Format
DNA	Deoxyribonucleic Acid
ED	Edit Distance function
EMF	Eclipse Modeling Framework
EPILOG	Event Processing, Investigating, and Logging
ITL	Intel Trace Analyzer
GraX	Graph Exchange Format
GXL	Graph Exchange Language
HPC	High Performance Computing
KDM	Knowledge Discovery Metamodel
KOJAK	Kit for Objective Judgment and Knowledge-based Detection of Performance Bottlenecks
LZW	Lemel-Ziv-Welch
MPI	Message Passing Interface
MPMD	Multiple Program, Multiple Data
NAS	NASA Advanced Supercomputing
OTF	Open Trace Format
RSF	Rigi Standard Form
SDDF	Self-Defining Data Format

SLOG	Scalable Log format
SMG	Semicoarsening Multigrid Solver
SPMD	Single Program, Multiple Data
STF	Structured Trace Format
TA	Tuple Attribute Language
TAU	Tuning and Analysis Utilities
UML	Unified Modeling Language
VAMPIR	Visualization and Analysis of MPI Resources
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1. Introduction & Motivations

This thesis targets the analysis of traces generated from inter-process communication applications that use the message passing paradigm. The objective is to develop techniques to facilitate the understanding of the content of large inter-process communication traces. In the following, we will motivate the idea behind this research and outline the main contributions of this work.

1.1 Problem Statement

High Performance Computing (HPC) benefits from parallel computing systems in order to solve computation-intensive scientific problems. As opposed to sequential computing, parallel computing decomposes the problem into sub-problems that run on different computational units in order to solve the problem in a reasonable amount of time. In most cases, the computational units need to collaborate in order to complete a specific task. This collaboration is achieved using two main programming paradigms which are the shared memory and distributed memory paradigms. In shared memory, processes collaborate by sharing the same memory space. On the other hand, a distributed memory application consists of many processes running on different distributed processors that interact using the message passing model. These parallel programs may consist of thousands of processes that are coordinating to solve a specific large scale problem. In this thesis, we focus on distributed memory applications with specific interest in programs that use the Message Passing Interface [MPI] (an accepted standard for writing parallel applications using message passing) for inter-process communication.

Although the benefits of HPC applications are numerous, they tend to be difficult to debug and analyze, causing significant delays in production and maintenance times. This is mainly due to the large number of inter-communicating processes they involve and the size of data to be processed. Therefore, it becomes necessary to develop program analysis techniques that can facilitate the understanding of these types of applications.

Program analysis techniques are grouped in two categories: Static analysis and dynamic analysis. Static analysis techniques study the source code and the available documentation. They do not involve the execution of the system. Despite their popularity, static analysis techniques tend to be conservative to the understanding of the behavioural aspects of software, especially in the context of parallel systems where system attributes can only be detected during run-time. Dynamic analysis techniques, the focus of this thesis, revolve around the examination of traces generated from running an instrumented version of the software system. Dynamic analysis of software systems has the advantage of being precise since it depicts the system's actual behaviour. Dynamic analysis, however, suffers from the huge volume of data that is generated, which hinders any viable analysis. There is a need for techniques that enable software engineers to understand and analyze large traces despite the trace being massive.

The objective of this thesis is two-fold:

- Build an exchange trace format that leverages the synergy among the various trace analysis tools.
- Develop techniques to reduce the size of traces to allow their analysis. Using these techniques, software engineers can browse a trace at a higher level of abstraction than the low-level events.

In the next section, we discuss the motivations behind the selection of the analysis of message passing programs. Section 1.3 presents the main contributions of this thesis. Finally, the outline of the thesis is presented in Section 1.4.

1.2 The focus on traces of inter-process communications

In this work, we focus on inter-process communication traces generated from HPC applications which use the MPI standard as the inter-process communication model. HPC applications are used in different domains such as bioinformatics, cryptography, telecommunications and others. These applications tend to be complex and require excessive inter-process communication in order to achieve their goals. Consequently, it becomes more difficult to maintain and understand these types of applications when compared to sequential programs. One of the main factors that hinder the comprehension of such applications is the excessive inter-process interactions. Therefore, understanding the inter-process communication can provide valuable insight into the behaviour of HPC applications. Our motivations behind studying inter-process communication traces can be summarized as follows:

1. The wide acceptance of the message passing model for inter-process communication.
2. The complexity of parallel programs as a result of the large number of their communicating processes and the huge amount of data to be processed.
3. The need for a standard exchange format for the available tools for dynamic analysis of parallel message-passing applications which is expected to improve the synergy among them.

4. The need for new techniques for inter-process communication trace abstraction in order to facilitate the understanding of the large amounts of trace data generated from executing these systems.

1.3 Thesis Contributions

The main contributions of this thesis are discussed in what follows:

1.3.1 Exchange Format of MPI Traces

Recently, there has been an increase in the number of tools to help software engineers analyze the behaviour of HPC applications. These tools provide several features that facilitate the understanding and analysis of the information contained in inter-process communication traces generated from running an HPC application. They, however, use different formats to represent traces, which hinders interoperability and sharing of data. We address this by proposing an exchange format called MTF (MPI Trace Format) for representing and exchanging traces generated from HPC applications based on the MPI standard. The design of MTF is validated against well-known requirements for a standard exchange format, with an objective being to lead the work towards standardizing the way MPI traces are represented in order to allow better synergy among tools. We have also developed a set of queries to facilitate the retrieval of data from MTF traces. Additionally, we have applied concepts from graph theory in order to represent MTF traces in a more compact format. The model and its ability to scale to large traces are tested against traces generated from running large HPC programs.

1.3.2 Communication Patterns Extraction

We propose a new approach that facilitates the understanding of large traces of inter-process communication by extracting communication patterns that characterize the main behaviour embedded in a trace. Two algorithms are proposed. The first one permits the recognition of repeating patterns in traces of MPI 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. In this thesis, we also present a pattern detection technique that overcome the main limitation of existing approaches and which lies in the fact that they generate many patterns among which many are noise. This appears to be due to the fact that they treat a trace as a mere string of events for which they apply various pattern matching techniques. In other words, they are blind to the different parts of a trace. In this thesis, we propose an approach that uses the routine call tree to guide the pattern extraction process. We show the effectiveness and efficiency of our approach in detecting communication patterns from large traces generated from different HPC programs systems.

1.3.3 Execution Phase Detection

We present a novel approach that aims to simplify the analysis of large execution traces generated from HPC applications through the semi-automatic extraction of computational 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 large HPC industrial systems demonstrate its effectiveness in identifying the main program phases and their corresponding sub-phases.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 – Background

This chapter starts by presenting the main concepts in MPI. Then, we review existing trace abstraction techniques. The chapter details the related work that targets the development of trace formats for traces generated from HPC. Moreover, it details the state of the art of the communication patterns detection approaches followed by the latest work conducted on detecting phases in MPI programs.

Chapter 3 – MPI Trace Format

This chapter starts by describing the domain of MPI traces followed by a presentation of the requirements for having a standard exchange format. The MTF metamodel and its components are presented in the chapter. Furthermore, an approach for compacting traces of MPI programs is presented. The chapter is concluded by different case studies that demonstrate the usefulness of the model in terms of the application of different queries and the scalability of the model.

Chapter 4 – Communication Patterns

The chapter starts by illustrating the communication patterns and their importance in understanding the inter-process communication behaviour in the program. Then, the chapter details the different techniques that are used in the detection of communication patterns. Finally, different case studies are presented to demonstrate the effectiveness of the presented techniques.

Chapter 5 – Phase Detection

This chapter presents a new approach for detecting execution phases in MPI programs based on the concepts in information theory. The chapter starts by explaining the importance of identifying the different execution phases in the program then it presents the methodology for the detection of execution phases. The chapter concludes with a case study that illustrates the different steps in the phase detection process and the accuracy of the results.

Chapter 6 – Conclusion

This chapter discusses the contributions of the thesis and the directions for future work. The chapter then concludes with some closing remarks.

Chapter 2. Background

2.1 Introduction

In this chapter, we start by explaining the message passing interface (MPI) along with its main functions. Then, we discuss the different trace abstraction techniques. Thereafter, we survey the state of the art of related research studies that target the dynamic analysis of MPI programs. First, we present the prominent execution trace formats for MPI trace analysis tools. Second, we discuss a list of trace analysis and visualization tools for HPC MPI programs. Third, the most relevant communication pattern detection studies are explained. Finally, we present the existing execution phase detection techniques for MPI programs.

2.2 Message Passing Interface

Message passing is an effective inter-process communication paradigm that enables the exchange of data and synchronization among processes in parallel programs. Existing software libraries that facilitate this kind of communication among processes are called Message Passing Environments (MPE). The most popular message passing environment is the Message Passing Interface (MPI) which has become a standard in the industry and academia. The primary goals of MPI are efficient communication and portability. Although several message-passing libraries exist on different systems, MPI is popular for the following reasons:

- Fully Asynchronous: process communications and computations can overlap.
- Group Membership: Processes may be grouped based on context.

- Synchronization Variables: these variables are used to enforce synchronization. They include the source and destination information, message labelling, and context information.
- Portability: the MPI specification is publicly available for implementation on any environment.

MPI is a framework that facilitates the inter-process communication in parallel programs based on message passing. Every process consists of a program counter and address space and may also have multiple threads (program counters and associated stacks) sharing a single address space. MPI targets the communication among processes which have separate address spaces. Figure 2.1 depicts a typical distributed parallel environment. It is composed of different processors that contain one or more processes and a mean for inter-process communication which is in this case based on MPI. Processes in MPI programs are arranged in a specific process topology. A process topology is the way the processes are virtually represented on a grid (Cartesian) or a graph structure.

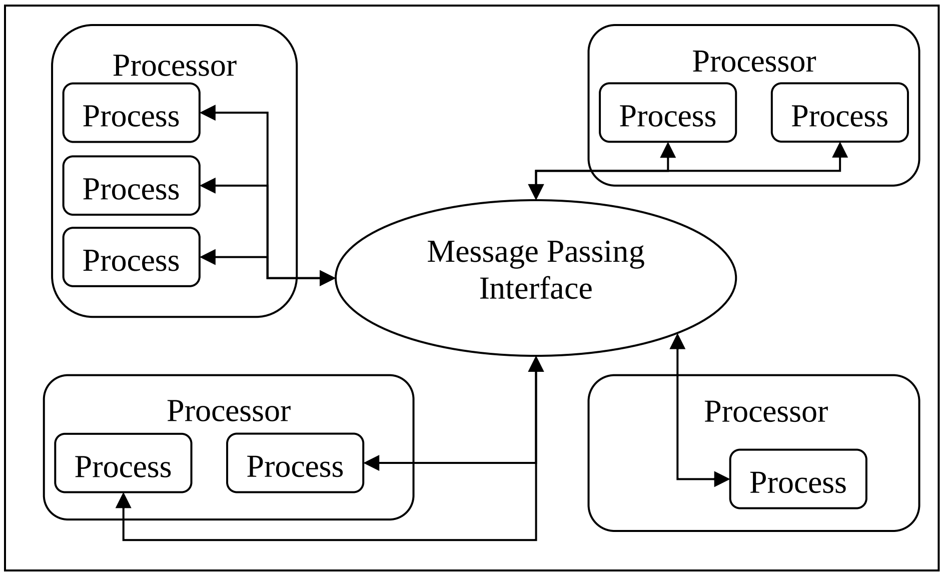


Figure 2.1 Message Passing Environment

MPI supports two types of programming paradigms [MPI]:

1. SPMD (Single Program, Multiple Data): All the processes will run the same program on multiple sets of data in order to complete the task.
2. MPMD (Parallel Programs, Multiple Data): Processes will run different programs on multiple sets of data in order to complete the task or set of tasks.

The MPI library routines provide a set of functions that support the following [MPI]:

- Point-to-point communication.
- Collective communication.
- Communication contexts.
- Process topologies.
- Data-type manipulation.

The following sections will provide a detailed study on the point-to-point and collective MPI communications.

2.2.1 Point-to-Point Communications

Point-to-point communication involves sending and receiving messages between two processes [MPI]. This is the simplest form of data transfer in a message-passing model. One process acts as the sender and the other acts as the receiver. The message consists of an envelope that indicates the source, destination, tag, communicator and data. There are two modes in communication in point-to-point MPI operations:

- *Blocking:* the program will not return from the subroutine call until the copy to/from the system buffer has finished.
- *Non-blocking:* the program immediately returns from the subroutine call. It is not assured that the copy to/from the system buffer has completed so that the program

has to check for the completion of the copy. MPI uses different routines to check or to wait for the completion of the operation.

Message exchange should occur between two processes that belong to the same group. A group of processes in MPI is defined as a *Communicator*. A communicator is an object that represents a group of processes and their communication medium or context. These processes exchange messages to transfer data. Communicators encapsulate a group of processes such that communication is restricted to processes within that group. A message is sent with a specific user-defined tag value that can be used at the receiver to identify the incoming message. Also, a receiving process may accept a message regardless of the *tag* by specifying MPI_ANY_TAG as the *tag* in the posted *Receive*.

Table 2.1. Blocking and Non-blocking Send Operations

Mode	**Blocking**	**Non-blocking**	**Description**
Standard	MPI_Send	MPI_Isend	MPI may buffer the message
Buffered	MPI_Bsend	MPI_Ibsend	A Send operation may start and complete without waiting for a posted matching Receive.
Synchronous	MPI_Ssend	MPI_Issend	A Send operation may start whether or not a matching Receive has been posted. However, the operation will not complete successfully unless a matching Receive is posted and started to receive the message
Ready	MPI_Rsend	MPI_Irsend	A Send operation that uses the Ready mode of communication cannot start unless the matching Receive is already posted.

MPI provides four types of Send-operations, each of them available in a blocking and a non-blocking variant. Table 2.1 provides the names of the four operations for each mode of communication. Receive operations can be blocking and non-blocking and can match any of the modes of the send operation. A blocking receive operation can match a non-blocking send operation and vice-versa.

Table 2.2 shows the two different MPI receive operations. The length of the received message must be less than or equal to the length of the receive buffer. A message can be received only if its envelope matches the Source, Tag, and Communicator in the Receive operation unless specified otherwise.

Table 2.2. Blocking and Non-blocking Send Operations

Mode	**Non-blocking**	**Description**
Blocking	MPI_Recv	The process will block until the operation is completed.
Non-Blocking	MPI_Irecv	The process will resume after posting the receive operation.

In non-blocking mode of communication, the process can use the MPI_Wait operation in order to wait for the completion of the Send/Receive operation. Moreover, the process may use the MPI_Test to check for the completion of the operation. The completion of a send indicates that the sending process is free to access the send buffer. The completion of a Receive indicates that the Receive buffer contains the message and it is ready to be accessed by the receiver.

2.2.2 Collective Communications

Collective communication involves exchanging information among a group (communicator) of processes. MPI provides a set of routines that handle this type of communication. Basically, these collective routines are based on the point-to-point routines. Thus, a combination of point-to-point MPI operations can achieve the same functionality of the collective ones. However, collective communication routines do not use tags for message send and receive operations. Table 2.3 presents the different types of collective operations defined in MPI. Collective communication can be one-to-many, many-to-one, or many-to-many. The single originating process in the one-to-many routines or the single receiving process in the many-to-one routines is called the *root*.

Table 2.3. MPI Collective Operations

Operation	**Description**
Barrier	Used to block the calling process until all processes have entered the function. Forces synchronization among the processes in the communicator.
Broadcast	MPI_Bcast operation is used to broadcast a message from a root process to all other processes in the communicator.
Gather	MPI_Gather collects the contents of each process' data and send it to the root process, which stores the messages in rank order.
Scatter	MPI_Scatter is a one-to-all type of communication and is the opposite of MPI Gather.
AllGather	MPI Allgather gathers the data from each process in the communicator and sends them to all the processes in the communicator so all the processes will have the same copy of each process' data.
All-To-All	MPI_Alltoall is an extension of MPI_ALLGATHER where each process sends distinct data to every other process in the communicator. The jth block sent from Process *i* is received by Process *j* and is placed in the ith block of the receiver's buffer.
Reduce	MPI_Reduce will store the result of a specific arithmetic operation in the root process.

A basic rule for collective communication is that all processes must execute the same collective communication operations in the same order. This enforces synchronization among the group's processes. However, MPI does not guarantee this synchronization and recommends using the Barrier operation. Collective operations in MPI have the following rules:

- Type matching conditions are stricter than the ones in point-to-point communication.
- The amount of data sent must be exactly the same as the amount specified by the receiver.
- Collective operations come in blocking versions only.
- Collective operations do not use a tag argument which means that they are matched strictly according to the order of execution.
- Collective operations come in standard mode only.

It is recommended to use the collective operations when needed instead of using point-to-point operations for that purpose.

2.3 Trace Abstraction Techniques

Execution trace size is one of the major drawbacks of the dynamic analysis of software systems. Therefore, in order to make dynamic analysis a favourable approach it is necessary to provide means for reducing the amount of trace data without losing its main characteristics. In this section, we present the main trace abstraction techniques found in the literature.

2.3.1 Sampling

Sampling [Chan 03] is used effectively in the dynamic analysis of software systems and is performed by processing a number of sampled events from the trace rather than processing the whole trace file. The sampling method can be performed in different ways such as selecting every *nth* event from the trace file, randomly or using a customized method. Sampling helps in reducing the trace file size but with a drawback that some key events may have been skipped using this method. In the scope of our work, we do not intend to apply sampling to traces of inter-process communication applications.

2.3.2 Filtering

Filtering the trace data based on different factors such as the type of objects, the time interval, a slice from an object type and others is another way of reducing the amount of trace data to the software engineer [Hamou-Lhadj 05]. This is another effective abstraction technique that is found in many trace analysis tools (e.g. [JumpShot]). In traces for method calls, filtering also includes techniques such as stack depth limitation (the nesting level of the method in the trace) determined by a threshold. Only method calls that appear up to the specified threshold are taken into account during the analysis. In inter-process communication traces, filtering may be used by hiding some processes and their corresponding events, hiding specific types of events and others such as showing messages with size greater than a specific value.

2.3.3 Grouping

Grouping [Cornelissen 08] or clustering is an abstraction technique that groups events or processes (in case of parallel applications) according to specific criteria. This technique is different from sampling and filtering in that it attempts to apply some rules to group the objects under study to provide a higher level of abstraction. The result of this technique is another set of objects that simplify the understanding of the trace under study.

2.3.4 Utility Removal

An execution trace contains a lot of information that in many cases may not be useful in understanding the program [Hamou-Lhadj 05]. Therefore, removing these elements will reduce the size of the trace and will make it more beneficial for program comprehension. These elements are called utilities. Utilities could be methods, classes, packages, processes and threads that do not implement important functionality of the system. They are used to provide support to the functions that implement the core functionality.

2.3.5 Pattern Detection

Software programs repeat the same or similar behaviour throughout the program run which can be extracted and presented to the software engineer. These repeated behaviours can be detected in the trace files using different techniques. This repeating behaviour is known as patterns. A pattern is a sequence of events that is repeating non-contiguously in the trace file. In traces of method calls, a repeated pattern is a sequence of method calls (at different nesting levels) that are repeated non-contiguously in the trace file. In inter-communication traces, a pattern is a set of inter-process communications that are repeating non-contiguously throughout the trace. Detecting repeating patterns reduces the effort of understanding the trace file as the scattered patterns in the trace file are presented to the software engineer automatically.

2.3.6 Visualization Techniques

Trace visualization [Cornelissen 2009] plays a significant role in program comprehension since it abstracts the trace data into different views that provide meaningful information to the software engineer. Trace visualization is considered an abstraction technique since each visualization view presents different information that may be very high-level or very detailed based on the objective of the analysis. Most visualization techniques provide some features that allow the user to abstract the trace data by grouping events, hiding events, highlighting events and others.

2.4 Inter-process Communication Trace Formats

In this section, we present different types of trace formats. Some of these formats are generic and can be applied to traces of MPI programs. Another set of trace formats is designed specifically to carry traces of HPC programs that use MPI for inter-process communication.

2.4.1 Self-Defining Data Format (SDDF)

The Self Defining Data Format is one of the leading trace formats that have been used for representing trace data generated from distributed applications [Aydt 94]. It is a general-purpose format that is designed to be a meta-format for defining data record structures. SDDF trace files consist of a header and packet sections. The header determines the type of encoding used in the trace file (binary or ASCII). The binary representation of SDDF can be used when compactness is sought. On the other hand, the ASCII representation is used when portability and readability are needed. The packets describe information about the trace files such as the time the trace was generated. The main packet, which defines the data record structures, is called the 'Record Descriptor'. The trace data exists in the 'Record Data' packet which is represented using the Record Descriptor packet. Another advantage of using SDDF is its flexibility. Therefore, trace format developers can define new trace formats by extending the meta-format provided by SDDF. SDDF, however, is not specifically designed to support MPI operations, which renders its applicability to support traces generated from HPC systems based on MPI a difficult task. Also, it does not suggest a well-known data carrier for exchanging the trace data.

2.4.2 Pajé

Pajé trace format is a generic trace format that provides the ability to define the structure of the traces based on the targeted problem [Kergomm 03]. Similar to SDDF, the trace data format of Pajé is self-defined. The meta-format (the trace structure) is defined in the trace file in a hierarchical manner that classifies all types of traceable elements. A Pajé trace file is composed of two definition categories that define the format of the generic instructions about the experiment and the format of the event traces respectively. Pajé, also, contains two data categories (the trace data) which represent instances of the two definition categories. The trace file contains the definition of the events followed by the events themselves. Events with different unique identifiers can have the same names. This allows adding different fields for the same event type based on the tracing requirement. Though the Pajé trace format provides flexible ways of defining different event formats, it is difficult to represent all the properties of MPI traces such as matching point-to-point operations and their corresponding wait and test statements.

2.4.3 EPILOG

The Event Processing, Investigating, and Logging (EPILOG) format is a binary trace format for representing traces of MPI and OpenMP (a paradigm for shared memory programming) applications [Wolf 04]. An EPILOG trace file consists of two sections. A header which contains information related to the EPILOG file such as the EPILOG version number. The second part is the records section. EPILOG uses two record types; the definition record and the event record. Each record consists of a header and a body. The header defines the length and the type of the record body. Definition records are used to define the types and objects that will be used in the trace file. For example, a definition record can be used to define the trace for the MPI send operation. Also, EPILOG defines records for the communicator and the locations in the MPI application so they can be referenced by other record definitions. The event records are used to capture run-time information. EPILOG provides a trace format specifically designed for MPI traces. However, a main drawback of using EPILOG is the fact that it provides a binary trace format that hinders portability of the trace format on different platforms.

2.4.4 Structured Trace Format

The Structured Trace Format (STF) handles traces generated from large applications using several physical files [STF]. The purpose is to properly control the size problem of large trace files to avoid having trace files that take up more than ten gigabytes. STF defines a set of files mainly the index file (locates other STF files), the declaration file, the event data file and the statistics file. The declaration file defines the record formats of the traced units such as the methods Enter and Exit. The data file contains the trace data based on the format defined in the declaration file. Finally, the statistics file contains some profiling information based on the trace. The Intel Trace Collector (ITC) tool [STF] produces traces in the STF format. STF traces can be analyzed using the Intel Trace Analyzer (ITA) performance analysis tool. This trace format does not meet the simplicity requirement for a standard exchange format as it is complex to use since it requires managing different types of data files. Moreover, this trace format is proprietary which contradicts the openness requirement for a standard exchange format.

2.4.5 Open Trace Format

The Open Trace Format (OTF) uses different streams (files) to represent trace data for HPC parallel applications [Knüpfer 06a]. A stream usually corresponds to one process in the program. However, traces of one process must exist in one stream only in order to preserve the execution of the process' events. Each stream contains definitions for the trace events such as the routine names, the MPI operations used in the trace file as well as the information regarding the processes and the MPI communicators in the application. The definitions of the traces are followed by the events traced in the program. Some statistical information may also follow the trace events in the stream. OTF defines an index file that is used to map each process to its stream (file). This file is used by the OTF library to locate and map the streams for each process. OTF uses ASCII encoding in order to be presented as a platform independent trace file format. Finally, OTF uses compression techniques in order to provide reduced trace file size. Based on our experiments, we believe that OTF is an efficient trace file format. However, it does not use a popular data carrier which makes it difficult to be read by other tools. Moreover, OTF stores the events sequentially without taking the scalability problem into account.

2.4.6 Scalable Log format (SLOG)

The Scalable Log format (SLOG-2) [Margaris 09] is a hierarchical trace file format that is built with the intention to support the visualization of huge trace files efficiently. Its main purpose is to enable only loading the displayed time window in memory without the need to load the whole trace file which may exceed in some cases multiple gigabytes. Therefore, this trace format avoids removing some trace data in order to reduce the file size. Each hierarchy represents a level of abstraction which is composed of different time intervals.

The deeper we go in the hierarchy the more intervals that we discover. The SLOG file format has a binary tree structure that is defined recursively with the root node being the interval from 0 to the last event end time in the trace.

2.4.7 Paraver Trace Format

The Paraver Trace Format [Paraver] uses one file to store the trace data. It defines the following record types: Enter/Leave events for routine calls, Atomic events for capturing performance counters information, and communication events for point-to-point and collective communication events. In addition to timestamp sorting of events, Paraver permits the sorting of events by their event type. Paraver provides the description of events based on their physical and logical locations by using two fixed hierarchies. The logical location description contains threads, processes and applications. The physical location contains CPUs, Nodes of multiple CPUs and systems of multiple nodes. Moreover, Paraver supports additional configuration files that are used to configure the display of event types.

2.4.8 TAU Trace Format

TAU (Trace Analysis Utilities) trace format [Shende 04] uses a binary encoding for trace events. It is used by the TAU profiling tool [Shende 05]. The trace format uses a single file to define and store the trace data. Initially, traces are gathered from each process separately and then merged into the single file. All record types use the exact same number of bytes to represent the events, which limits the extensibility of the trace format.

2.4.9 Research Studies that target the Scalability of MPI Traces

In this section, we present a number of research studies that target the scalability problem of MPI traces.

2.4.9.1 ScalaTrace

Noeth et al. [Noeth 09] presented ScalaTrace that provides a compressed trace format for MPI traces. The compression takes place at two stages: intra-process compression followed by inter-process compression. At the process level, they represent the identical sequences of MPI events caused from loops using one regular section descriptor (RSD) which specifies how many times the sequence is repeated. The intra-process compression is then followed by an inter-process compression using a binary tree where similar RSDs with matching counts are merged. The main advantage of their approach is that the compression preserves the temporal ordering of events. However, this approach has the following two main disadvantages:

- The approach only targets Single Process Multiple Data (SPMD) applications where all processes behave similarly which makes their approach useful for these cases only.
- Even though the approach keeps the ordering of events, it is still lossy as it provides approximate timestamps and not the exact values that were collected at the tracing time.

Moreover, this study only provided compression of MPI events in the program and did not take into account other kind of information such as user routine calls.

2.4.9.2 Construction and Compression of CCG for Post-mortem Trace Analysis

Knüpfer et al. [Knüpfer 05] proposed the usage of compressed Complete Call Graphs (cCCG) in order to represent traces of single and parallel process programs. In parallel process programs, each process trace will have its own cCCG. The cCCG is a directed acyclic graph as in their approach they tend to combine regular patterns into common sub-trees. Representing the routine call tree as a directed acyclic graph was previously conducted by [Larus 99, Reiss 01] and later improved by [Hamou-Lhadj 04]. However, Knupfer et al. do not look for identical sub-trees. They search for compatible trees by comparing the sub-trees' top nodes only and assuming that if all the references of the child nodes of the two compared root nodes are pointing to the same sub-tree then the two sub-trees are considered to be compatible. This trade-off for time complexity reduces the accuracy of the compression algorithm. Furthermore, they represent the timing information as delta times (duration) instead of the timestamps that are gathered at execution time. In order to recover the original timestamp, the traversal of the graph from the root node to the designated node is required. Two sub-trees are considered similar when the delta times in both the sub-trees' nodes deviate within a specified bound. Therefore, when considering a small deviation bound, the number of similar sub-trees will be very low which will result in a lower compression ratio. When constructing the CCG, they take the graph branching factor into consideration (number of direct children to the node). If the branching factor is beyond a threshold, then artificial nodes will be inserted into the graph between the parent and its children by splitting the children into two or more groups.

2.4.9.3 ScalaExtrap

Wu et al. [Wu 11] presented an approach for the extrapolation of an application's communication traces and their execution times from small traces in order to simulate traces at larger scale. The extrapolation method is based on the communication topology identification at smaller numbers of processes. They proposed the usage of a set of linear equations in order to obtain the relation between communication traces for traces with different number of processes that will enable the extrapolation of communication traces.

Regarding the extrapolation of timing information, they employed curve fitting approaches to represent trends in delta times over traces with varying number of processes. The authors only considered SPMD HPC MPI programs with stencil and mesh process arrangements that only exploit one communication pattern throughout their execution. However, when considering more complex problems such as SMG2000 which has several communication patterns and have varying communication behaviour when considering different problem sizes, the presented extrapolation technique will be limited.

2.4.9.4 Logicalization of Communication Traces from Parallel Execution

This work [Qu 09] presents a framework to automatically construct a single logical trace that is a representative of the overall parallel execution when the communication pattern is a regular stencil. The approach is based on identifying the communication topology of the application and converting all point-to-point communication calls between physical processes to logical calls representing the global communication pattern. The methodology is independent of the numbering of processes in the system. The key contribution is an algorithmic framework to identify the global communication topology from distributed message exchange data that is effective and efficient. This work provides only a logical representation of the complete execution trace. Therefore, the resulting trace is lossy and cannot recover the original trace from the logical one.

2.5 Visualization Techniques for Inter-process Communication Traces

Visualization techniques for traces generated from parallel applications can be divided into three main types; behavioural, structural and statistical. Behavioural techniques visualize the execution of the program over time. Structural techniques are used to describe the structure of communication such as the communication topology among processes.

Statistical techniques present summary information about the execution trace such as the number of events, the size of data exchanged and so on. These techniques have implementations in 2D and 3D space diagrams. In the following, we present the state-of-the-art of the research studies that have been proposed in the literature for the visualization of inter-process communication traces.

2.5.1 Message Passing Visualization with ATEMPT

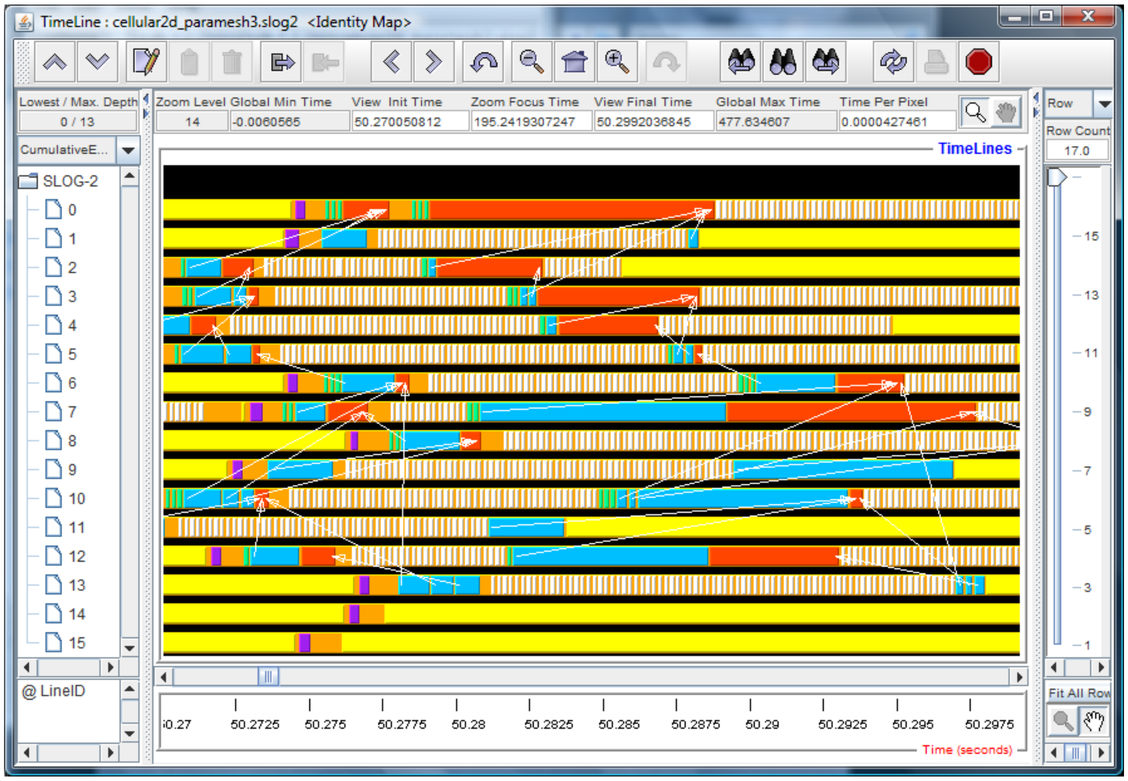
ATEMPT [Kranzlmüller 95] "A Tool Event ManiPuliaTion" is a tool that applies the concept of event graphs for visualizing communications among the processes in parallel applications. An event graph [Kranzlmüller 00] consists of a horizontal line for each process, vertices that represent the event and directed edges between the events which represent the process communication or the sequential program flow. In inter-process communication applications, the edges are used to represent messages exchanged among the program processes. The purpose of ATEMPT is to help software engineers detecting errors such as a send event with no receive event, and performance analysis of parallel applications. One main advantage of ATEMPT is that it applies the concept of trace abstraction to limit the analysis to the points of interest. However, the abstraction is performed in a semi-automatic way by allowing the user to specify the main areas of interest in the graph.

2.5.2 ParaGraph

Paragraph [Heath 03] is a performance and behavioural visualization tool of parallel programs based on MPI. It is a post-mortem tool that displays execution traces pictorially in an animated manner. Also, it provides some graphical statistical views that provide summaries about the performance of the application under test. ParaGraph was initially developed based on PICL (Portable Instrumented Communication Library) in 1989 and was modified later to support the new message passing specification (MPI). ParaGraph supports views for processor utilization such as the utilization count, Gantt chart, Kiviat diagram and concurrency profile. Also, ParaGraph supports several displays that depict the communication among the processes in the program such as the space-time diagram and the communication matrix. Also, ParaGraph supports an animated view that has a node for each process' status (busy, overhead, idle, sending, receiving, or collective communication) and arcs between the nodes to represent the communication activity between the processes. ParaGraph contains many other views that we cannot include in this context for space limitation.

2.5.3 JumpShot

JumpShot-4 [Jumpshot] is a visualization tool that supports the SLOG-2 trace format. An advantage of JumpShot-4 is the Level-of-detail support which means that it does not need to read the whole trace file into memory. It only reads the data needed at each level of abstraction. The main view in JumpShot is the space-time view which also provides a Gantt-like chart for each process in order to show the activities each process is involved in. Moreover, it uses arrows to depict the messages among the different processes in the program. Figure 2.2 shows an example of the JumpShot-4 tool. Each horizontal line belongs to a process in the program which contains all the actions that were performed by a process. Also, as can be seen, the arrows show the messages being exchanged among the processes. As can be seen, the program provides zooming functions, filtration, searching, scrolling and others.


TimeLine : cellular2d_paramesh3.slog2 <Identity Map>

Lowest / Max. Depth 0 / 13
Zoom Level 14
Global Min Time -0.0060565
View Init Time 50.270050812
Zoom Focus Time 195.2419307247
View Final Time 50.2992036845
Global Max Time 477.634607
Time Per Pixel 0.0000427461
Row Count 17.0

Figure 2.2. Screenshot of Jumpshot Tool

2.5.4 Pajé Visualization Tool

Pajé [Paje] is a versatile trace based visualization tool designed to help performance debugging of large-sized parallel applications. From trace files, recorded during the execution of parallel programs, Pajé builds a graphical representation of the behaviour of these programs, to help programmers identify their "performance errors". Pajé provides two types of visualization techniques to represent graphically containers, state, events, variables and links. The first and most used is the space-time window, which actually draws a Gantt-chart display that uses arrows to represent interactions among processes. The second type of display is used to dynamically show statistical information about a selected slice of time in the space-time window. Pajé is designed to be interactive, scalable and extensible which, according to its developers, enables it to handle a very large amount of traces efficiently.

2.5.5 Vampir

Vampir [Vampir] is a commercial performance and visualization tool that is supported by the Center for Information Services and High Performance Computing (ZIH) of TU Dresden. The main objective of Vampir is to support scalable visualization of inter-process communication of OTF traces generated from MPI using the VampirTrace tracing tool. Vampir contains several display views such as the message statistics view, matrix chart, summary chart, Gantt-charts, summary timeline and counter timeline. Vampir has a set of flexible filter operations, which are used to reduce the amount of information displayed and to help its users to spot more easily performance problems. Figure 2.3 shows a screenshot of Vampir. Furthermore, Vampir provides a hierarchical visualization, based on Gantt charts, which allows users to view trace data in different levels of abstraction such as process, thread, and the process cluster. An advantage of using the hierarchical technique is its scalability. This technique supports up to 50 times more processes than only using the Gantt chart.

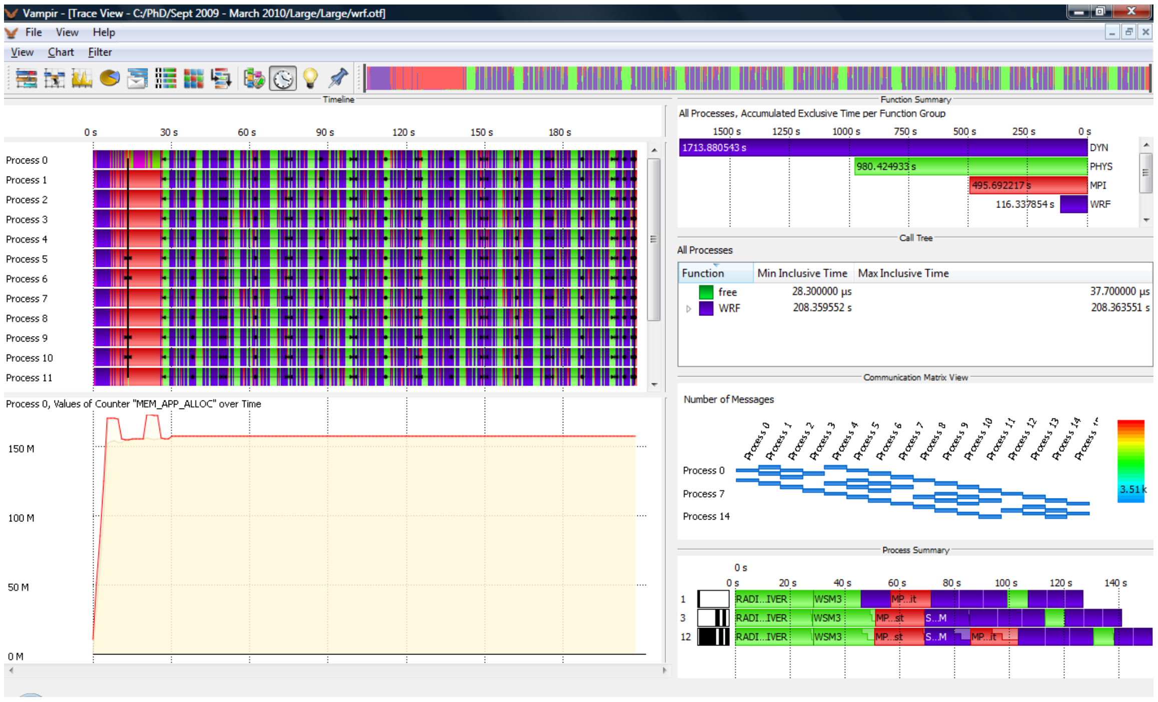


Figure 2.3. Screenshot of Vampir Tool

2.5.6 ParaProf

ParaProf [Paraprof] is a visualization tool for parallel applications. It is part of the Tuning and Analysis Utilities (TAU) [Shende 05] project, a joint project between the University of Oregon, Los Alamos National Laboratory, in the United States, and Julich Research Center, Germany. ParaProf is designed to be portable, extensible and scalable and is organized in four main components. The visualization component supports 3D visualizations, thread-based views, function-based views and phase-based views. The 3D views include Triangle Mesh Plot (provides metrics for program functions and threads), 3D Bar Plot, and the 3D Scatter Plot. The thread-based view provides statistics for each thread and a call graph of the functions executed in the program. The function-based views include statistical information depicted using function bar chart and function histograms.

Finally, the phase-based views show statistical data related to each execution phase in the parallel program.

2.5.7 Visual Analysis of Inter-Process Comm. for Large-Scale Parallel Computing

Muelder et al. [Muelder 09] proposed a new visualization approach for understanding communication behaviours and identifying performance for large scale parallel programs that consist of thousands to millions of processes. In their approach, they focus on the system as a whole before digging down into individual processes or MPI calls. They propose three views with different levels of abstraction. The highest level of abstraction view presents the system as a whole and provides information on how the overall communication is impacting the system performance. A more detailed view considers the communications among groups of processes (ignores individual processes). In this view, the MPI calls can be viewed regardless of the number of participating processes. The third view shows the details for individual views and individual MPI calls. Furthermore, they used opacity scaling to resolve the overlapping of the plotted MPI calls.

2.6 Communication Patterns Detection

In this section, we present the state of the art of the research studies that targeted the detection of communication patterns in inter-process communication applications.

2.6.1 Detecting Patterns in MPI Communication Traces

The authors [Preissl 08] proposed an algorithm for the detection of repeating patterns in MPI traces. Their approach is based on compressed suffix trees to detect the maximal repeats in every process trace separately. For each process trace, they select certain maximal repeats and not all of them by using seed events or sub-graph properties and in some cases they used static analysis to determine the most important areas in the code and use their events accordingly. They only consider a subset of the maximal repeats since it takes a prohibitive time to compute the communication patterns based on all maximal repeats. However, our analysis shows that the same maximal repeat may be part of different communication patterns. Therefore, this factor should be taken into consideration when filtering most of the detected maximal repeats. After selecting the start repeats, they start building the communication pattern starting from one maximal repeat on process *i*. Then, they compute the maximal and minimal intervals by locating the matching events on the other processes. This step is done iteratively until the communication pattern is complete and all the maximal repeats were included in the iterations.

2.6.2 Exploitation of Dynamic Communication Patterns through Static Analysis

In [Preissl 10], the authors applied their communication pattern detection approach supported by static analysis in order to detect point-to-point communication patterns that correspond to collective MPI operations. The objective of this work is to replace point-to-point communication patterns by collective MPI operations that have better performance than using an equivalent communication based on MPI point-to-point operations.

2.6.3 Automatic analysis of inefficiency patterns in parallel applications

Wolf et al. [Wolf 07] utilized the knowledge from virtual topologies in order to identify patterns of inefficient behaviour due to long wait states caused from inefficient application of the parallel programming model. The communication topology (virtual topology) is used to identify the phases of inter-process communication in the program. This work is different than our communication pattern detection since it only looks for patterns of inefficient behaviour resulting from processes in long wait states. Also, they presume knowledge of the communication topology in the program which helps them in identifying the different parallel communication phases in the program. Our approach looks for inter-process communication patterns by investigating message passing events.

2.6.4 Visualization of Repetitive Patterns in Event Traces

The authors [Knüpfer 06b] proposed an algorithm to remove contiguous repeating patterns from the trace in order to reduce the size of the trace. The algorithm is based on the compressed complete call graph (cCCG) and the pattern graph (a derivative of the cCCG). An advantage of using cCCG is that it references all call sequences that are equal with respect to call structure and temporal behaviour, which improves trace compression. In their algorithm, they only detect contiguous pattern repetitions. They claim that patterns found at interspersed locations are identified as the same pattern which is not the case when studying large traces with hundreds of distinct patterns. Moreover, this approach does not detect communication patterns. It only detects repeating patterns on each process trace separately.

2.6.5 TraceVis: An Execution Trace Visualization Tool

TraceVis [Roberts 05] is a trace visualization tool for parallel program executions. In TraceVis, the pattern detection algorithm depends on the human ability to process enormous amounts of visual data. The trace graph view in TraceVis is used to locate regions of similar inter-process communications. Though this may be possible for reasonable trace sizes, dealing with huge traces that involve a large number of processes is merely impossible.

2.6.6 Fast Detection of Communication Patterns in Distributed Executions

The authors [Kunz 97] presented a technique based on finite state automata to find communication patterns in the trace that match an input pattern. The pattern matching algorithm is performed by determining the longest process pattern in the input communication pattern which will be used as the search string in the pattern matching algorithm. They start building the communication pattern by locating the partner events on the other process traces. This approach is only concerned with detecting patterns based on a pre-defined input pattern. In our work, we propose two algorithms for detecting repeating patterns and matching a pre-defined pattern.

2.6.7 An Approach for Matching Communication Patterns in Parallel Applications

The authors [Ma 08] proposed an approach for comparing the communication patterns found in the traces generated from different systems to find the degree of similarity between them. The degree of similarity between two applications 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. This work is different from our work as it compares traces generated from different systems.

2.6.8 Scalable Parallel Debugging with g-Eclipse

The authors [Köckerbauer 10] 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 user's specified pattern.

2.6.9 A Scalable Approach to MPI Application Performance Analysis

Moore et al. [Moore 05] proposed a pattern matching method for detecting patterns of inefficient behaviour based on wait states in order to be used in KOJAK (a performance analysis tool for high performance parallel applications) [KOJAK]. These patterns of inefficient behaviour are identified by converting the trace into a compact call-path profile which classifies patterns based on the time spent. This approach only looks for events that cause performance degradation and does not focus on the inter-process communication patterns.

2.7 Phase Detection

The execution of a program exhibits a similar cyclic behaviour which can be identified as several execution phases [Gu 06]. In the literature, several studies investigated the usefulness of the program execution phases in performance optimization, reducing profiling overhead, system reengineering, and in program comprehension. In MPI programs, there exist a small number of studies that target the detection of execution phases.

2.7.1 Automatic Phase Detection of MPI Applications

In their study [Casas 07], 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. A region with high frequency of iterations will be separated from a low frequency one. This work targets the detection of computation phases in MPI programs. The different MPI phases (initialization, computation, and output) are categorized based on their frequency of iterative behaviour where in the computation phase most of the parallel iterations exist. The objective of this work is to provide the analyst with an initial abstraction level that provides an overview about the system under study before studying the source code. Casas et al. indicated that the computation phase in MPI programs is usually large and more effort should be invested in an algorithm that identifies the sub-computational phases.

2.7.2 Automatic Detection of Parallel Applications Computation Phases

Gonzalez et al. [González 09] presented an approach to facilitate the analysis of message passing parallel applications using the density-based clustering techniques to detect computation phases that occur between the parallel communications in the program. They apply the density-based approach on data obtained from performance counters provided by modern processors. The main objective of this work is to detect the most important regions of execution in the program. They use CPU bursts to outline the different regions in the program. A CPU burst is considered as a CPU computation region between two consecutive communications. Therefore, a burst is identified by the duration and the set of performance counters.

2.7.3 Automatic Phase Detection and Structure Extraction of MPI Applications

Casas et al. [Casas 10] extended the previous work that uses wavelet transform from signal processing in order to detect the different sub-phases in the computational phase. They base their approach on the iterative behaviour found in MPI traces where CPU bursts are followed by process communication. They derive the signals from different metrics that are based on inter-process communication and computing bursts. They assume that the highest frequencies of communications (signals) appear in the computation phases. Therefore, their approach detects regions with highest frequencies and identifies them as the computational phase in the program.

2.8 Summary

The focus of this thesis is on developing techniques to facilitate the understanding of inter-process communication traces. Therefore, the work in this thesis lies within the domain of program comprehension. We focus on two research problems which are the modeling of MPI execution traces and their abstraction. In abstraction, we target communication patterns detection and matching techniques and execution phase detection techniques.

This chapter targeted a survey of the related studies. In the following, we comment on the surveyed research studies.

- None of the surveyed trace formats targeted the development of an exchange format that meets the requirements for a standard one. Existing trace formats are not scalable to carry very large execution traces. Additionally, approaches that targeted the scalability proposed lossless trace formats which may cause the loss of potential trace information.

- The existing communication pattern detection techniques do not take into account the quality of the detected patterns. Usually, they detect a large set of false positives. Also, existing techniques do not scale up to large traces.
- Only a few phase detection techniques have been proposed in the literature. These techniques focus mainly on performance analysis. We believe that our work on detecting execution phases from execution traces with a focus on program comprehension is considered unique and novel.

Chapter 3. MPI Exchange Format (MTF)

3.1 Introduction

Several techniques and tools have emerged to facilitate the analysis of HPC applications (e.g. [TAU, Vampir, and Heath 03]). These tools come with many features including trace analysis algorithms, visualization layouts, optimization algorithms, pattern detection methods, and others that can help in studying the runtime behaviour of these applications for performance analysis, debugging, deadlock detection, and so on. These tools, however, do not interoperate due to a lack of a common exchange format for representing HPC traces. Clearly, a common trace format that enables synergy and sharing of data among tools is needed, and reduces the effort and cost required to represent HPC traces.

The objective of this chapter is to present MTF (Message Passing Interface Trace Format), an exchange format that we have developed to represent runtime information generated from HPC applications. The focus is on inter-process communication traces based on the message passing paradigm, with a particular interest in MPI [MPI]. MTF supports the modeling of MPI operations, the application's processes and the way they interact in a specific usage scenario, and the routine calls that are executed by each process during a particular execution.

There exist several exchange formats in the literature for HPC-generated traces (presented in Chapter 2), but most of them do not scale up to large traces or they support lossy versions of the original trace. Many of them are also proprietary and represent traces in binary format which hinders their portability and understandability.

MTF is built with several requirements in mind to facilitate its adoption and enable it to become a standard exchange format for traces generated from HPC applications. One of the key requirements that we have carefully addressed is the ability for MTF to support very large traces. This is particularly important in the context of traces since typical traces may contain millions of events, especially if generated from HPC applications that involve a large number of computing nodes (which is very common in practice). The specification of MTF is openly available. The MTF model itself is represented as an Ecore model developed using the Eclipse Modeling Framework (EMF) [EMF]. MTF also reuses existing data carriers such as XML. We have also developed a query language and an API that can be readily used to extract information from MTF models. In sum, we believe that MTF supports key features that can make it a common exchange format for representing and sharing information generated from HPC systems, and if adopted, we believe it can lead the work towards a standard exchange format for MPI traces.

The rest of the chapter is organized as follows. In Section 3.2, we present the domain of MPI traces. In Section 3.3, we present the requirements for a standard exchange format. Section 3.4 presents the MTF metamodel and its main components. Section 3.5 presents the MTF tool support. In Section 3.6, we present an approach for compacting MPI traces based on the directed acyclic graph which is supported by MTF. Section 3.7 presents the validation of MTF. Finally, Section 3.8 presents a case study that shows the effectiveness of MTF to support large traces generated from different systems and benchmarks. We conclude the chapter in Section 3.9.

3.2 The Domain of MPI Traces

An MPI trace depicts the execution of the running processes in the program along with the messages exchanged among them. HPC applications often follow the Single Program Multiple Data (SPMD) paradigm in which the program tasks are run in parallel on multiple processors to maximize performance.

As mentioned in the background chapter, communication among processes is based on executing MPI operations supported by the MPI environment. MPI supports two communication modes: point-to-point and collective communications. Point-to-point operations are blocking and non-blocking operations. They only involve two processes (a sender and a receiver). On the other hand, collective operations involve all the processes in a communicator that is specified in the call. Collective operations can only run in blocking mode in order to guarantee the synchronization among the processes. The MPI specifications [MPI] provide detailed description of the various MPI operations. An MPI trace can be considered as a set of streams of data, where each stream corresponds to one process in the program. Each trace contains the routines executed by the process, the MPI operations invoked by the process to communicate with other processes, the messages sent and received, and many other details such as timestamps.

Figure 3.1 shows an example of two processes that execute in parallel four functions *f1*, *f2*, *f3*, and *f4*. The label on the edge is added here to show the order of execution within each process. The interaction between these two processes is also shown as typical Send and Receive MPI operations along the exchanged messages. The message object is created by merging the atomic sent-message and received-message events on the sender and receiver respectively.

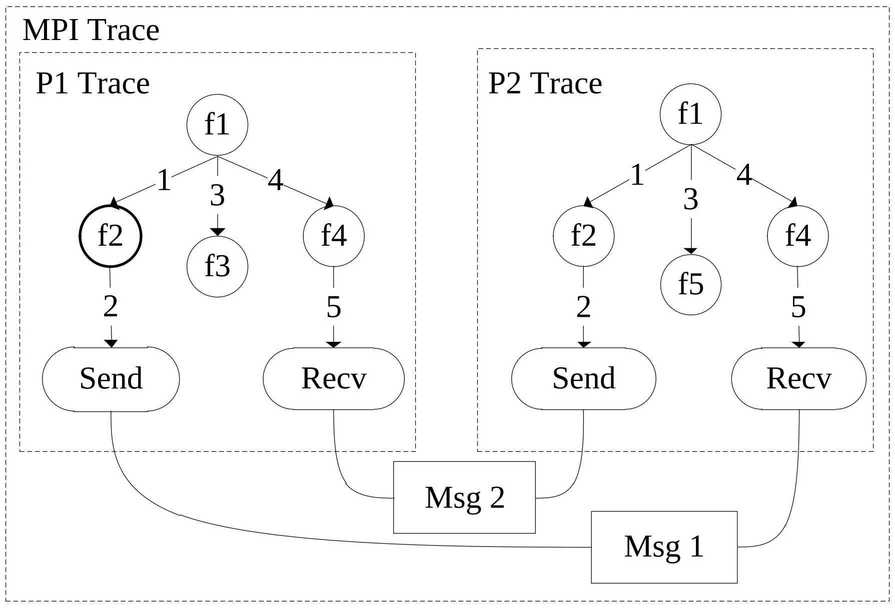


Figure 3.1. MPI Trace Representation

3.3 Requirements for the Design of MTF

A trace format should meet certain requirements in order to qualify as a common exchange format. These requirements are summarized in [St-Denis 00] and include expressiveness, scalability, openness, simplicity, and transparency. Although our proposed metamodel is developed to meet most of these requirements, in this thesis, we focus on expressiveness, scalability, extensibility, and openness. We used these key requirements as guiding principles in the design of MTF.

3.3.1 Expressiveness

An exchange format should be expressive enough to capture the needed information to enable various types of analyses. After studying the MPI specifications and the related research studies, it has become clear that all the information needed for MPI operations must be captured in order to be used during the analysis phase. For example, when tracing an MPI_Send operation, we need to store information about the sender, receiver, data type, tag value, communicator, size of sent data, and the address of send buffer. We also need to record the routines executed by each process and the order of execution to be able to identify where in the program a specific communication of multiple processes occur. MTF was carefully designed to provide support for all these concepts.

3.3.2 Scalability

An exchange format should be scalable to support a large amount of information efficiently and in a way that does not degrade access to the instance data. This is particularly important in the area of trace analysis since the size of typical trace files can easily reach tens to hundreds of gigabytes. To achieve this, we employed a compaction scheme presented by Hamou-Lhadj et al. [Hamou-Lhadj 04] and in which the authors used graph-theory concepts to compact large traces of routine calls in the design of their exchange format CTF (Compact Trace Format) [Hamou-Lhadj 04].

3.3.3 Extensibility

Exchange formats should be easily extended in order to support new or different data types. Also, they should be extended without affecting previous versions of the trace data. This is important especially when the analysis tools evolve and may acquire new types of data to cope with the emerging analysis techniques. We believe that the design of the MTF model follows sound object-oriented concepts that make it readily extensible to support additional trace elements.

3.3.4 Openness

In order to qualify for a standard exchange format, we believe that a trace format should be freely available to its users along with the metamodel, the semantics of its components, and the syntactic form. This also opens the door for further improvements to the model or possibilities to customize it to specific needs. MTF specifications are open and we are working on the finalization of the implementation of the model along with the interfaces that will allow querying the trace data.

3.4 MTF Components

In this section, we present the MTF exchange format. The definition of an exchange format involves two main components [Bowman 00]: A metamodel (also called a schema) that describe the abstract syntax or the structure of the entities to exchange and the way they are connected, and the syntactic form, which describes how the instance data of the metamodel is represented in a trace file.

3.4.1 MTF Metamodel

Figure 3.2 shows a UML class diagram that describes the MTF metamodel. The entities of this metamodel are discussed in the following subsections. The exact definition of the classes of the metamodel including their attributes, associations, constraints, and semantics are presented in Appendix A using as similar template as the OMG[1] template for defining the UML metamodel.

3.4.1.1 Usage Scenario

The Scenario class is used to describe a certain usage scenario which is used in generating one or more execution traces. MTF permits that a usage scenario can be represented by different traces showing both normal and exceptional executions.

[1] http://www.omg.org/uml

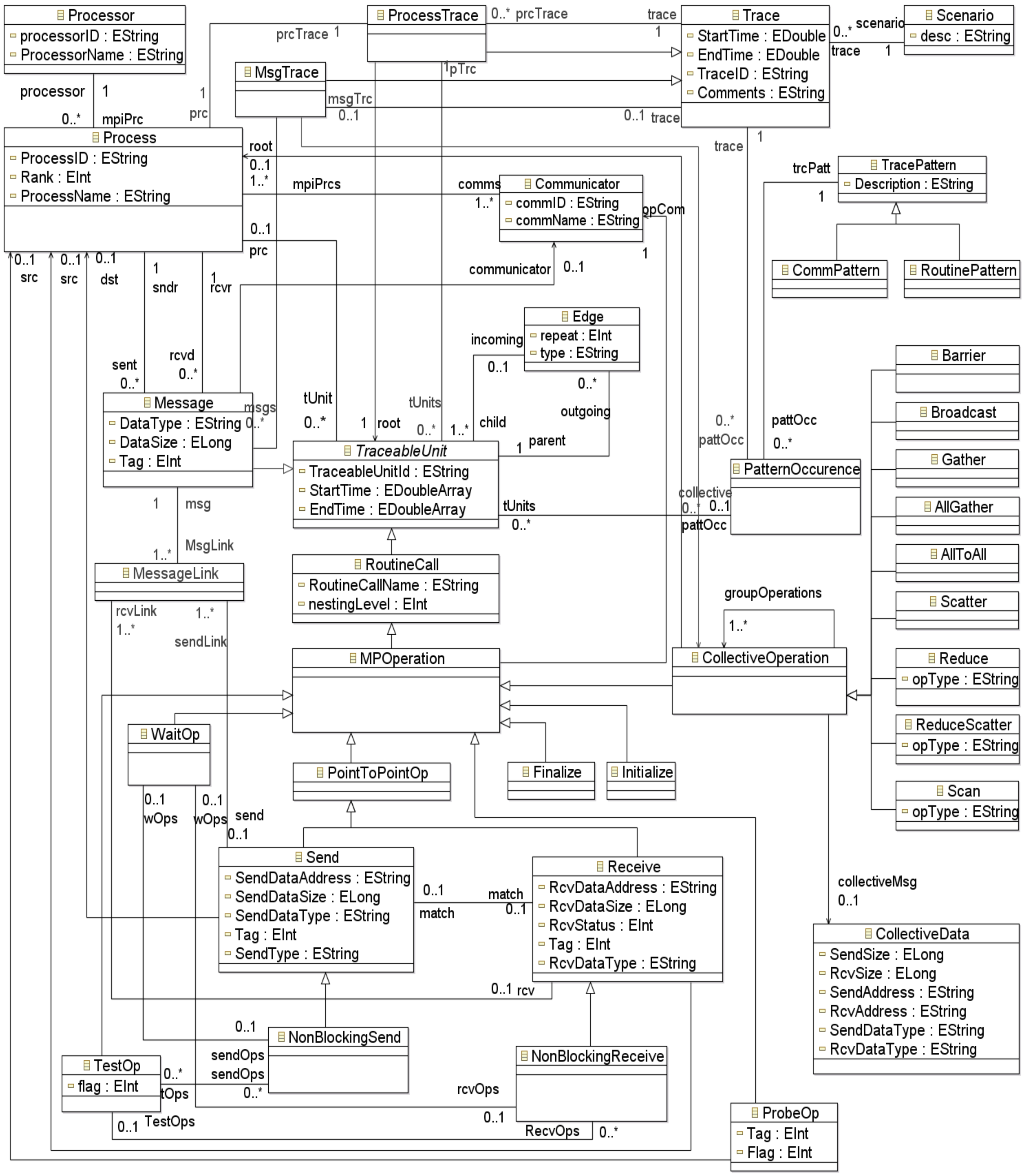


Figure 3.2. The MTF Metamodel

3.4.1.2 Trace Types

The class Trace is used to describe information about the collected trace such as the name, the time the trace was collected, etc. To create specialized types of traces, one can simply extend this class. In our metamodel, we define the MsgTrace class to represent traces of point-to-point messages exchanged in the application. On the other hand, the class ProcessTrace is used to represent all the traces generated from a particular process in the program. The class Trace is a concrete class and it is meant to represent the whole execution trace of the program.

3.4.1.3 Processor and Process

The Processor class is used to capture the machine and the node on which a process is running. A process in the MPI program is represented using the Process class.

3.4.1.4 Traceable Unit

An execution trace generated from running HPC applications contains different kinds of information such as routine calls, MPI operation calls, messages, I/O operations and others. In MTF, the abstract class TraceableUnit is used for extending the metamodel with any kind of events that may be generated during the program execution. Therefore, the extensibility requirement is captured by our metamodel using the TraceableUnit abstract class.

3.4.1.5 Edge

The Edge class is used to represent the traces in a graph structure. The type attribute specifies the type of edges to be used. The model supports three types which are the sequence, fork-sequence and recursive edges. We will show an example of each edge in Figure 3.6.

3.4.1.6 Message

The Message class represents the messages exchanged using point-to-point operations only. It captures information regarding the sender, receive, data size, data type and tag value. An Instance of the MessageLink class is used as a link between a message and its corresponding MPI operation. Each message is linked to two MPI operations.

3.4.1.7 MPI Operations

The MPOperation is the base class for all types of operations defined in the MPI specifications. This class is further specialized to represent specific MPI operations such as Initialize, Finalize, point-to-point operations (represented by the class PointToPointOp) and the Collective operations (represented by the class CollectiveOperation). The PointToPointOp class is extended into specific operations modeling blocking *send* and *receive* MPI operations (represented using the classes Send and Receive), non-blocking send and receive operations (classes NonBlockingSend and NonBlockingReceive).

The metamodel also depicts the relationship between the non-blocking operations and the *wait* and *test* operations represented by the WaitOp and TestOp respectively.

Collective operations (run in blocking mode only) such as a barrier and broadcast are represented using classes that inherit directly from the CollectiveOperation class. It should be noted that the presented metamodel in Figure 6 does not include all of the implemented classes that represent the MPI operations to avoid cluttering the model.

3.4.1.8 Collective Data

The data exchanged during the execution of collective operations is modeled using the CollectiveData class. It represents the information about the data being exchanged by each process when executing a collective MPI operation. MPI requires that all the processes in a communicator be involved in the collective communication.

3.4.1.9 Trace Patterns

Traces may contain several patterns that are defined as sequences of events that are repeated non-contiguously in a trace. MPI applications may contain two types of patterns which depict specific behaviours in the program. The communication patterns may be detected in the point-to-point and collective messages and the routine call patterns may be detected in the routine call events in the trace. According to Hamou-Lhadj et al. [Hamou-Lhadj 04], who presented an exchange format for representing traces of routine calls, the analysis of patterns found in a trace might reveal important information about the behaviour of the system. In MTF, the class TracePattern is the base class for the CommPattern (represents communication patterns) and the RoutinePattern (represents routine call patterns). Moreover, the class PatternOccurrence represents a single occurrence of a give pattern in the trace.

3.4.1.10 Well-formedness of MTF

The well-formedness of MTF is supported by adding the necessary constraints that must be met in order to provide a correct representation of the MPI traces. Table 3.1 outlines the main constraints that are supported in the metamodel. The complete list of constraints can be found in the description of MTF in Appendix A.

Table 3.1. Main Constraints in MTF Metamodel

1	Instances of MPOperation class are always leaves (they do not have an outgoing edge).
2	Data type between matching point-to-point operations must match unless MPI_BYTE data type is specified.
3	A call to MPI_Init must precede any other MPI call in the program, except for MPI_Initialized routine that can be used to check if MPI_Init has been called or not.
4	Every process in the MPI environment must call MPI_Finalize before exiting unless a call to MPI_Abort has been made.
5	The StartTime of an MPI_Wait statement cannot occur before the StartTime of the corresponding Send or Receive operations.
6	A collective operation should match the same type of collective operation in all other processes. Therefore, the maximum number of matched operations may not exceed the number of processes in a communicator.
7	The end-time for a Barrier object of one process cannot be before the start-time for any of the matched Barrier objects of the other processes.
8	An object of type Barrier cannot reference an object of type CollectiveData.
9	The type signature (SendSize, SendDataType) for MPI_Bcast at the root process must be equal to the type signature of the matching MPI_Bcast on all processes (receiving processes) in the communicator.
10	In a Gather operation, The receiving buffer for non-root process should be equal to null.
11	Instances of AllGather do not reference a root process.
12	Instances of AllToAll do not reference a root process.
13	Only an edge with a fork-sequence type can have more than one child node.

3.4.2 Syntactic Form

The syntactic form of an exchange format describes the way the data (instances of the abstract syntax metamodel) is carried. There exist several data carriers including XMI (XML Metadata Interchange) [XMI-OMG], GXL (Graph Exchange Language) [Holt 00], TA (Tuple Attributes language) [Holt 98], etc. These syntactic forms vary depending on whether they are based on XML or not, their ability to carry the metamodel as well as the instance data, their compactness, etc.

We suggest that an adequate syntactic form that can be used with MTF should have the following characteristics:

1. It should be compact in order to be able to handle very large traces and enable the scalability of the trace analysis tools.
2. It needs to be able to carry the metamodel as well as the data (instance of the metamodel). This will allow tools to check the consistency of the data against the metamodel.
3. It should be open and portable. This excludes proprietary and binary syntactic forms that are dependent on a particular technology.
4. It should have tool support available such as parsers and viewers.
5. It should be adopted by tool vendors. This requirement favors well accepted data carriers such as the ones that have been standardized (e.g. XMI).

Except for Requirement 1, all other requirements can be met by a known XML-based language such as GXL, which is widely accepted in academia and industry [Holt 00]. However, when the GXL file is loaded into memory, the XML tags, which are considered verbose, will not be part of the loaded trace.

GXL is built on a number of pre-existing syntactic forms for exchanging software artefacts such as GraX [Ebert 99], TA [Holt 98], and RSF [Müller 88]. Figure 3.3 shows an example using GXL to represent an MPI trace which is used in the case study of this chapter to show the effectiveness of MTF to capture large MPI traces.

```
<gxl>
<graph>
<node id = “scen001”>
<attr name = “description”>
<string> Weather Research and Forecasting Model
Test</string>
</attr>
</node>
<node id = “trace001”>
<attr name = “startTime”>
<double> 12:00:00 </double> </attr>
<attr name = “endTime”>
<double> 12:00:40 </double> </attr>
<attr name = “comments”> <string> Sample MPI
trace of Weather Research and Forecasting Model
code </string></attr>
</node>
<node id = “PRCR00001”>
<attr name = “ProcessorName”>
<string> Processor 1</string> </attr></node>
<node id = “PRC00001”>
<attr name ="rank">
<int> 0 </int></attr>
<attr name ="ProcessName">
<string> Process 1 </int></attr></node>
<node id = “PRC00002”>
<attr name ="rank">
<int> 1 </int></attr>
<attr name ="ProcessName">
<string> Process 2 </int></attr></node>
--- REMAINING PROCESS NODES {2 - 15}
<node id = “COMM 1000000000”>
<attr name ="COMMName">
<string> MPI Communicator 0
</string></attr></node>
<node id = “trc000001”>
<attr name ="MPOperationName">
<string> MPI_Init </string></attr>
<attr name ="startTime">
<double> 0.00070105 </double></attr>
<attr name ="endTime">
<double> 0.0008256 </double></attr></node>
```

```
<node id = “trc000002”>
<attr name ="MPOperationName">
<string> MPI_Init </string></attr>
<attr name ="startTime">
<double> 0.00070185 </double></attr>
<attr name ="endTime">
<double> 0.0008311 </double></attr>
</node>
--- REMAINING MPI_Init NODES
<node id = “trc000017”>
<attr name ="MPOperationName">
<string> MPI_Bcast </string></attr>
<attr name ="startTime">
<double> 0.001653567 </double></attr>
<attr name ="endTime">
<double> 0.0233165 </double></attr>
</node>
<node id = “trc000018”>
<attr name ="MPOperationName">
<string> MPI_Bcast </string></attr>
<attr name ="startTime">
<double> 0.00172138 </double></attr>
<attr name ="endTime">
<double> 0.0297359 </double></attr>
</node>

--- REMAINING TRACE NODES
trace001
<edge from = “scen001” to =
“trace001”></edge>
<edge from = “trace001”to =
“trc000001”></edge>
<edge from = “trc000001” to =
“PRC00002”></edge>
<edge from = “trace001”to =
“trc000002”></edge>
<edge from = “trace001”to =
“trc000003”></edge>

--- REMAINING EDGES
</graph>
</gxl>
```

Figure 3.3. An example of an MPI trace captured with MTF and carried by GXL

3.5 MTF Tool Support

In this section, we present a prototype tool that we have developed to support the analysis of MTF traces. Our tool is written in Java as an Eclipse plug-in. Figure 3.4 shows the architecture of the tool.

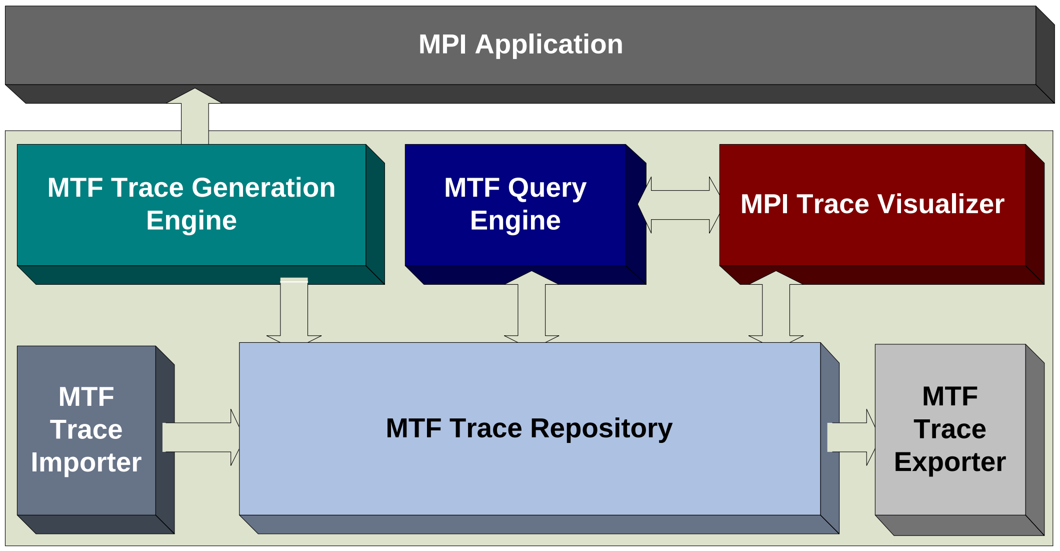


Figure 3.4 The MTF Tool Architecture

The tool consists of four main components presented here and discussed in more detail in the subsequent sections:

- The MPI trace repository: We used EMF (Eclipse Modeling Framework) [EMF] to create an Ecore model from which we generated the implementation of the MPI metamodel classes. The MPI trace query engine: We have developed a powerful query language that can retrieve all sort of information from an MPI trace modeled in MTF.
- The MPI Trace Generation Engine: We have developed an engine that permits generating traces in the form of MTF (carried in GXL).

- The MPI Visualizer: The visualizer aims to visualize MPI traces in a usable manner. The implementation of this component is not completed, and therefore, it is not included in this chapter.
- MTF Trace Importer and the MTF Trace Exporter are two modules used to convert the MTF traces from and to other trace formats respectively. We developed importers for OTF [OTF] and SLOG [SLOG] trace format, two commonly traces format used for MPI traces.

3.5.1 The MTF Trace Repository

The MTF trace repository is based on the Eclipse Modeling Framework (EMF), which is a modeling framework and code generation facility for building applications based on a structured data model [EMF]. The advantages of using EMF are as follows:

1. It explicitly represents the data model which gives a clear understanding of the data structure.
2. It generates an implementation from the model automatically.
3. If there is an update to the model, the corresponding implementation is also updated automatically.
4. It provides the flexibility to import a UML model (such as the MTF class diagram) created using any supported UML CASE tool such as Rational Rose [Rose].

In our work, we created an Ecore model by importing the MTF class diagram into EMF. We were then able to generate a Java implementation of the class diagram that is used by the other components of the tools such as the query engine.

3.5.2 MTF Query Language

In order to facilitate the use of MTF, we have implemented a set of queries in our EMF-based tool for accessing and retrieving of specific information about MPI traces. Every query has an implementation that can retrieve information about traces related to a single, group, or all the processes in a specific communicator.

Table 3.2 shows the part of the query that determines which processes the query should run on. For example, when specifying a query with (3-6) as the process parameter, it means that the query will only return a slice of a trace that involves processes 3 to 6 inclusive. In the following, we explain the different types of queries implemented in our toolset for MPI traces.

Table 3.2. Processes Specified in a Query

Process (p_n)	Traces related to one process only.
Processes (p_m - p_n)	Traces related to a sequence of processes.
Processes (p_a, p_c, p_m,…, p_n)	Traces related to a selected number of processes.
Processes in Communicator c_1	All processes in an MPI communicator.

3.5.2.1 Point-to-Point-Related Queries

Point-to-point related queries retrieve information that pertains to MPI point-to-point operations. Table 3.3 shows the information that the queries supported by our tool are capable of retrieving for point-to-point processes.

Table 3.3 Point-to-Point Queries

1	All *point-to-point* operations for a specific set of processes.
2	All *Send* operations for a specific set of processes.
3	All *Receive* operations for a specific set of processes.
4	All *point-to-point* operations sent and/or received between time t_1 and time t_2 for a set of processes where size of data is less than, equal to, or greater than *size$_n$*.

3.5.2.2 Collective-Related Queries

Collective related queries retrieve information that pertains to collective operations. Since collective operations involve all the processes in a communicator, we have only implemented the queries that are related to traces of one process or all the processes in a communicator. Table 3.4 shows the collective queries supported by our tool.

Table 3.4. Collective Queries

1	All *Collective* operations related to one process or all the processes in a communicator.
2	All traces related to a *specific* collective operation for all processes in the group.
3	All *Collective* operations executed between time t_1 and time t_2 related to one process in a communicator.
4	All *Collective* operations executed between time t_1 and time t_2 related to one process in a communicator where size of data sent/received is less than, equal to, or greater than $size_n$.

3.5.2.3 Message-Related Queries

Message-related queries target traces of messages exchanged in point-to-point operations. Table 3.5 shows the main queries used to retrieve information related to messages transferred using point-to-point operations.

Table 3.5. Message-Related Queries

1	All messages in the MPI trace.
2	All messages exchanged among a group of processes.
3	All messages exchanged among a group of processes between time t_1 and time t_2 related to where size of data sent/received is less than, equal to, or greater than $size_n$.

Figure 3.5 shows a few simple query examples that can be used in our tool to retrieve information from the trace under study.

Example 1: retrieve all messages in Communicator C1

SELECT ALL MESSAGES IN COMM(C1)

Example 2: retrieve all messages between process 1 and process 2

SELECT ALL MESSAGES BETWEEN PROCESS(1,2) IN COMM(C1)

Example 3: retrieve all point-to-point operations between process 1 and process 2

SELECT POINT_TO_POINT_OPERATIONS BETWEEN PROCESS(1,2) IN COMM(C1)

Example 4: retrieve all collective messages among all processes in communicator C1

SELECT COLLECTIVE_OPERATIONS AMONG ALL PROCESSES IN COMM(C1)

Example 5: retrieve all Broadcast messages that Process 1 performed

SELECT BROADCAST FOR PROCESS(1) IN COMM(C1)

Figure 3.5. Simple Query Examples

This query language can also be used to compute statistical information such as the time duration of a particular process in a MPI communication, the number of bytes a process sent to other processes and the number of bytes a process received from other processes during MPI communications. Also, we provide some queries for retrieving profiling information from the MPI execution trace. For this purpose, we define the following functions:

Process-fan-in: A process fan-in represents the number of bytes received by a process. This includes messages received by point-to-point as well as collective operations. A process fan-in includes data received using the following operations.

$$\textit{Bytes Received(p)} = \sum_{p\,=\,\text{receiver}} \text{Message.DataSize} + \sum_{p} \text{CollectiveData.RcvSize}$$

Process-fan-out: A process fan-out consists of the number of bytes sent by a process. This includes messages sent by point-to-point as well as collective operations. A process fan-out includes data sent using the following operations.

$$\textit{Bytes Sent(p)} = \sum_{p\,=\,\text{sender}} \text{Messages.DataSize} + \sum_{p} \text{CollectiveData.SendSize}$$

3.5.3 MTF Trace Generation Engine

Trace generation is another important feature in a trace analysis tool. We built our own tracing API which generates MPI traces based on our proposed trace format, MTF. We used the MPI standard Profiling Interface (PMPI) [MPI], for the instrumentation of the various MPI operations in a given program.

3.6 Scalability of MPI Traces

In this section, we present a set of techniques for compacting MPI execution traces that are based on graph theory. First we present some rules that can be used to normalize the original call tree and then we present a technique to convert the normalized graph into a directed acyclic graph.

3.6.1 Call Graph Normalization

The trace of each process in an MPI program can be represented as a routine call tree where MPI routines are at the leaf level. Usually, these programs generate many contiguously repeating events in the execution trace. These contiguous occurrences can be collapsed resulting in a normalized version of the original graph. This increases the possibility of finding similar sub-trees in the call graph as will be illustrated in the directed acyclic graph example.

Contiguous repetitions are often caused by the presence of loops and recursive calls in the code or the way the scenario is executed. Removing these repetitions from a trace can considerably reduce its size as shown by Hamou-Lhadj et al. in [Hamou-Lhadj 09]. Contiguous repetitions can be removed by collapsing the repetitions into one node in the graph. However, a trace file needs also to provide all of its original data including the timestamps. We therefore propose to keep an array of timestamps associated with the remaining node. For example, if we have the following repetitive events (*A*, t_1), (*A*, t_2), and (*A*, t_3), where *A* is the event and t_i represents the timestamp, then we can collapse them into one node (*A*,{t_1, t_2, t_3}) that keeps track of the timestamps in an array. Note that we only consider the routine name. If the value of the parameters for each call needs to be preserved then this compaction will fail. However, it is usually sufficient to understand that a particular routine is executed to build a mental model of the program without having to worry about the details of the call.

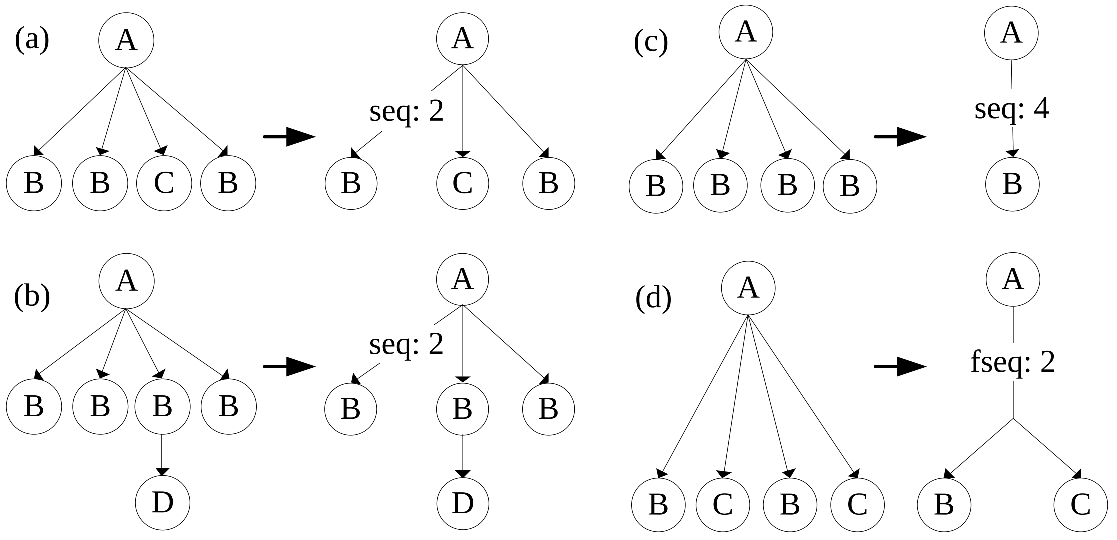


Figure 3.6. Collapse Contiguous Calls

Figure 3.6 shows four examples of how we collapse repetitive nodes in the trace. As mentioned earlier, the numbers on the edges represent the order of calls and are added here for clarification. Collapsed nodes should be at the same nesting level of calls. Example 3a shows that only the first two occurrences of ‘B’ can be collapsed. Example 3b shows that since the third occurrence of ‘B’ is calling ‘D’, then only the first two occurrences of ‘B’ can be collapsed. Example 3c shows that all four occurrences of ‘B’ can be collapsed since they all occur at the same nesting level and none of them is calling another node. The edge from ‘A’ to ‘B’ includes the order of its occurrence along with the number of repetitions. Moreover, in Figure 3d another type of edge is used. We call this as a fork-sequence which indicates that the ‘B, C’ sequence is repeated twice in the graph and is being called by ‘A’. The fork-sequence edge is the only edge type that allows more than one child node. This is a constraint that is added to our metamodel.

Also, nodes that occur from recursive calls can be collapsed into one node. For example, Figure 3.7 shows that ‘A’ is repeated 5 times in the tree resulting from recursive calls in the program. We collapse recursive calls by keeping the first call to ‘A’ and then by using a recursive edge with the number of repetitions to another node called ‘A’ which represents the recursive calls.

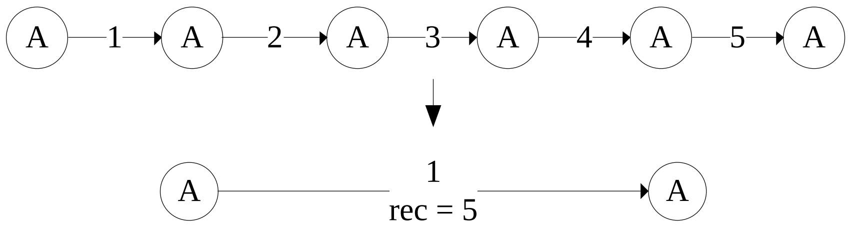


Figure 3.7. Collapse Recursive Calls

Messages exchanged between two processes can also be collapsed into one message node if they are identical while keeping track of the message timestamps in an array. Figure 3.8 shows an example depicting how the same message can be collapsed into one message node while keeping the associated timestamps. The metamodel in the next section shows that a Message class is associated with the Send and Receive classes using the MessageLink class. A message instance may have many MessageLink instances to a Send and Receive operations. The MessageLink class will simplify the retrieval of the timestamps from the timestamp array in the Message node.

As can be seen from the previous example, there are three types of edges; the sequence edge '*seq*', the recursive edge '*rec*', and the fork-sequence edge '*fseq*'. These edge types are represented by an attribute in the MTF metamodel.

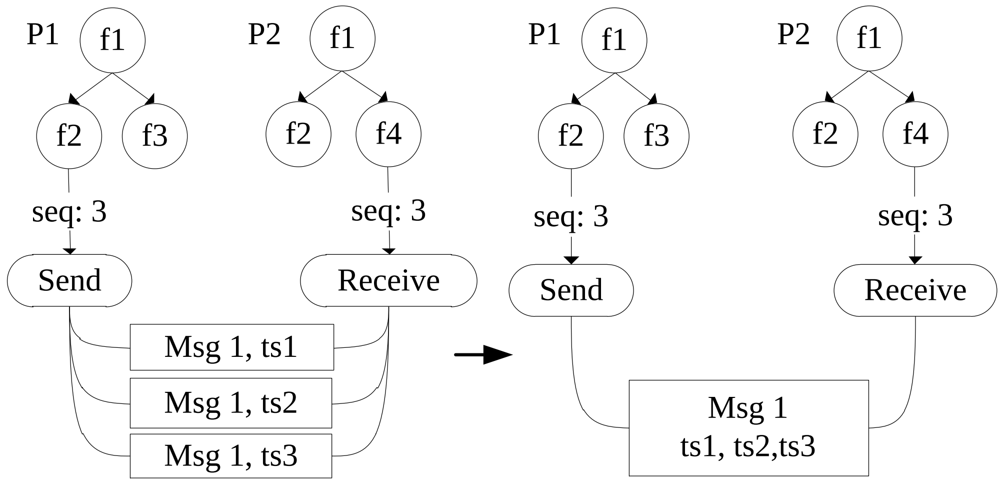


Figure 3.8. Message Compaction Example

3.6.2 Converting Call Graph to an Ordered Directed Acyclic Graph

Our second compaction mechanism consists of representing repetitions that appear non-contiguously in the trace (also known as trace patterns) only once in a trace. For this purpose, we adapted the compactness scheme presented by Hamou-Lhadj and Lethbridge [Hamou-Lhadj 04] and in which the authors proposed to transform a call tree into an ordered Directed Acyclic Graphs (DAG) where similar sub-trees are represented only once [Downey 80]. The authors showed that this transformation provided maximum compactness of the trace data while it preserved the order of calls and other attributes of the original trace.

In order to convert the call tree into an ordered directed acyclic graph, we used a variant of Valiente's algorithm [Valiente 00] which was modified by Hamou-Lhadj et al. [Hamou-

Lhadj 04] and applied it to traces of routine calls. Valiente's approach is a bottom-up approach for finding isomorphic trees where it traverses a tree from the leaves to the root node. The algorithm assigns each node a certificate number. Two nodes n_1 and n_2 will have the same certificate number if they belong to two sub-trees rooted at n_1 and n_2 that are isomorphic. Each node will have a signature value which is a concatenation of the node label and the certificate values of its child nodes. The signature value will be used in the calculation of the certificates. A leaf node will have its label as its signature. Therefore, in a bottom-up fashion, nodes with the same signature will be assigned the same certificate value.

Figure 3.9 shows an example of converting a tree into an ordered DAG after removing contiguous repetitions (Figure 3.9b). It should be noted that the presented graph have ordered edges from left to right. As shown in Figure 3.9b, two edges are of type *seq* (represents a sequence of the same event) and another two are of type *rec* (represents a set of recursive calls). The edge contains the number of repetitions which indicates how many times the node is originally represented. Figure 5c shows the final DAG which contains 9 nodes and 11 edges compared to 23 nodes and 22 edges in the original tree.

This simple example shows that the DAG provides a good compaction ratio compared to the original tree. It should be noted that without the graph normalization step, the three sub-trees in Figure 3.9a (with bolded nodes) will not be considered equivalent and the conversion to DAG will not be efficient. Similarly, the two sub-trees that represent the recursive calls for *F* will not be considered equivalent. We believe that this is the first time this technique is used for MPI traces.

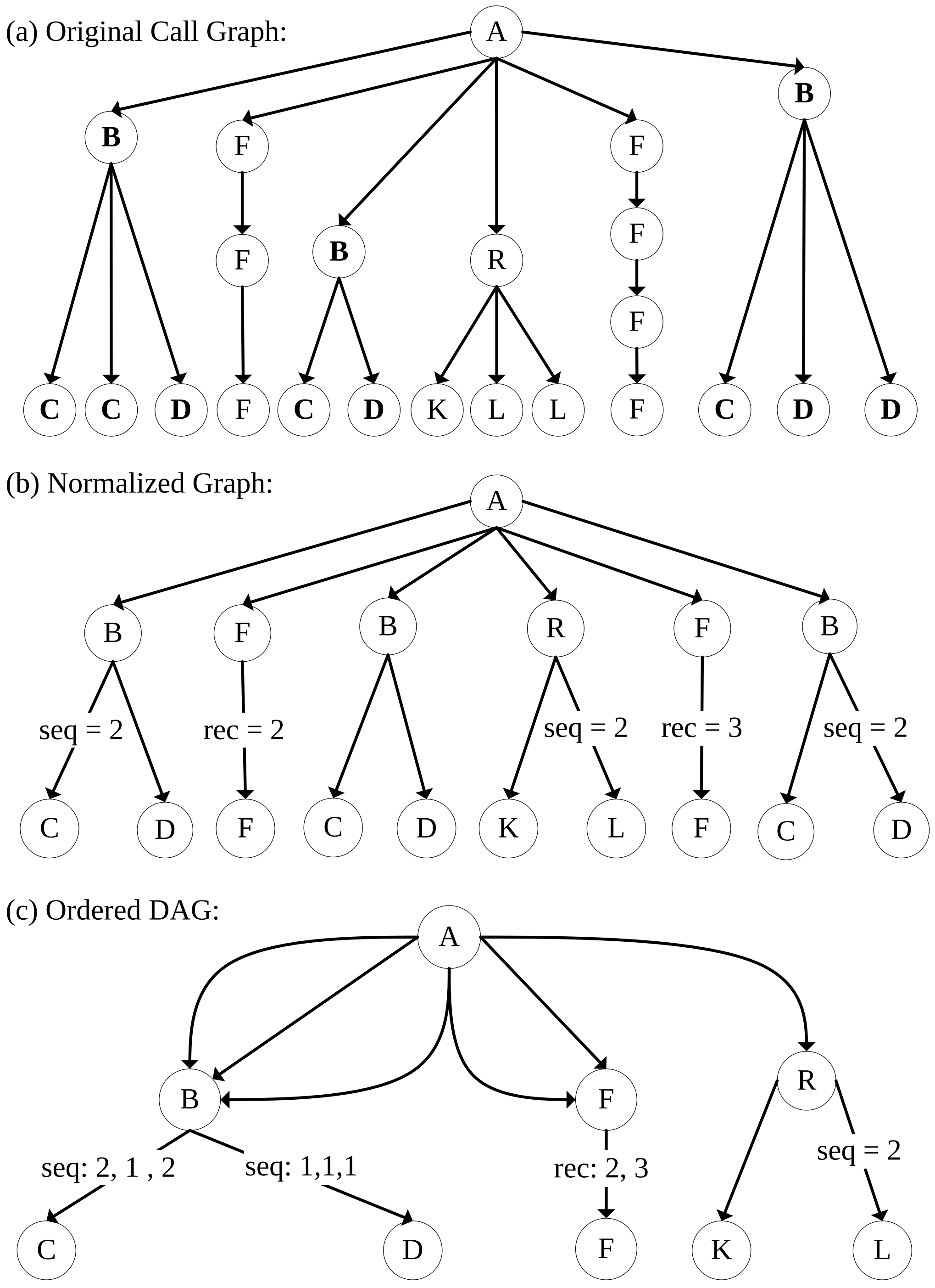


Figure 3.9. Tree to DAG Conversion Example

3.7 Validation of MTF

In this section, we discuss how MTF meets the requirements for a standard exchange format that we presented in Section 3. Table 3.6 summarizes the evaluation of MTF with respect of each requirement. As shown in Table 3.6, the design of MTF meets many of these requirements. It is expressive, fully supporting MPI functions. It is built with simplicity in mind using proper and well recognized modeling practices. It is also designed with transparency in mind by suggesting a data carrier that can not only carry MTF instance data but MTF metamodel (i.e., the abstract syntax) as well. This will allow tools that do not support MTF to check the well-formedness of an MTF trace with respect to the metamodel by reconstructing, on the fly, the metamodel from the MTF file. The design of MTF also favours reuse of an existing solution. First, many object-oriented design techniques have been used to build the MTF metamodel, which should readily enable tool builders to support MTF. Also, we recommend reusing an existing data carrier (e.g., GXL) rather than creating a new one so as to avoid reinventing the wheel. We also believe that MTF is easily extendible.

Table 3.6. Validating MTF against requirement for a standard exchange format

Requirement	**Justification**
Expressiveness	MTF supports all the necessary information for MPI point-to-point and collective operations that enable the analysis of MPI traces using MPI trace analysis tools.
Scalability	We showed how MTF is capable or representing MPI traces as a directed acyclic graph. Also, we showed how contiguous events can be supported using the list of timestamps.
Extensibility	MTF can be extended in many ways to support new types of traces by extending the Trace and the TraceableUnit classes.
Openness	MTF is provided as a metamodel and has been published in two different please. Also, a website will be shortly made available from which MTF specifications and accompanying tools can be downloaded.

3.8 Case Study

This section includes two parts for validating the scalability and the querying of MTF.

3.8.1 Scalability of MTF

In this section, we provide some results that show the usefulness of the compaction approach. Furthermore, we provide some results gathered from running some of the queries implemented in MTF. We used a 1.83 GHz Intel Core 2 Duo CPU with 3.0 GB of RAM for our experiments. In order to show the ability for MTF to represent MPI traces generated from large systems in a compact form, we tested it on several trace files generated by the VampirTrace tracing tool [VampirTrace].

VampirTrace generates traces in the OTF format presented in Chapter 2. The OTF format does not apply any compaction on the trace events themselves. It uses zlib [Gailly 02] to compress the trace file into several streams. However, the number of events in the uncompressed OTF file maps exactly to the number of events generated from the target system. In our study, we take OTF traces and apply our compaction techniques on them. More precisely, we load OTF traces as a call tree. Each call tree represents the calls executed by one process. The point-to-point messages are linked to their corresponding MPI calls as was shown previously in Figure 3.1. Then, we perform our collapsing rules on the nodes in the tree as well as on the point-to-point messages. Finally, we convert each call tree into a DAG which will result in an MTF representation of the original OTF trace. We targeted four programs provided by the NAS Parallel Benchmark [NAS]. We used the VampirTrace tracing tool to generate traces in OTF format. Also, we tested it on an OTF trace file that is generated from the Weather Research and Forecasting (WRF) model [WRF]. The scalability study is also applied to large traces generated from SMG2000

[SMG2000] and Sweep3D [Sweep3D] programs. In the following, we show the compaction gain obtained by turning OTF traces into MTF.

- **The NAS Parallel Benchmarks (NPB 3.3)**

The NAS parallel benchmarks [NAS] are a suite of benchmarks for performance evaluation of parallel supercomputers. They are developed and maintained by the NASA Advanced Supercomputing (NAS) Division (formerly the NASA Numerical Aerodynamic Simulation Program) based at the NASA Ames Research Center. In this case study, we target four programs that are part of the NPB suite (CG, MG, LU, and SP). We briefly describe each target program along with the results of the compaction rate on two traces from each program generated by the VampirTrace tool.

<u>CG:</u> This program represents a Conjugate Gradient method to compute an approximation to the smallest Eigen value of a large and sparse symmetric positive definite matrix. This kernel is useful for unstructured grid computations in order to test irregular long distance communication that employs unstructured matrix vector multiplication. We tested our compaction algorithm on two traces generated from running CG on 16 and 32 processes respectively. Table 3.7 shows the test results along with the compaction rate obtained after applying our compaction method. The results show that the compaction rate obtained using MTF is almost 78% in both cases, which is considerably high. We can also notice that the number of nodes that represent routines in all MTF traces is considerably low compared to the original traces (561 instead of 3509121 in the case of 16 processes). This is normal since the traced program is relatively small; it does not contain a lot of routines. In OTF, each call is represented as a separate object, which significantly increases the number of times the same routine appears in the trace. This number becomes higher as the number of processes increases. This demonstrates the need to represent routine calls of HPC applications as ordered DAGs.

MG: This program represents a simplified MultiGrid kernel which requires highly structured long distance communication and is used to test short and long distance data communication. We tested our compaction algorithm on two traces generated from running MG on 16 and 32 processes respectively. The results in Table 3.7 show that the compaction in both cases is almost 50%, which is satisfactory but also shows that further improvements to our approach are needed to obtain better results. For example, we can improve the way we measure the way two sequences of calls are deemed similar. In this thesis, we are only considering identical matching. Perhaps, we need to consider other matching criteria such as ignoring the number of contiguous repetitions when comparing two sequences of calls. However, the resulting MTF model will lose some information about the original traces. Further studies should be conducted to investigate ways to balance compaction and the quality of the information that we want to capture.

LU: This problem performs a synthetic computational fluid dynamics (CFD) calculation by solving regular-sparse, block (5 X 5) lower and upper triangular systems. We tested MTF on two traces with 32 and 64 processes. Table 3.7 shows the compaction rate for the trace of 32 processes and 64 processes respectively. The trace of 64 processes contains more than 18 million events (nodes). It has a slightly smaller compaction rate (65%) compared to the 32 processes' trace (69%).

SP: This problem offers a solution of multiple, independent systems of non- diagonally dominant, scalar, and pentadiagonal equations. SP solves three sets of uncoupled systems of equations in the x, y, and in the z dimensions starting with the x-dimension. This problem only accepts a square number of processes. In the case of 64 processes the compaction rate was 73.1%. However, when considering 100 processes, the compaction rate was reduced to 66%. Table 3.7 shows the details for the MTF compaction of the SP traces.

- **Weather Forecasting & Research (WRF) Model:**

WRF [39] is a next-generation mesoscale numerical weather prediction system developed to help in both operational forecasting and atmospheric research studies. We ran the compaction technique on a trace that is generated from the WRF model on 16 processes. The results in Table 3.7 show that the compaction rate is 51%.

- **SWEEP3D**

Sweep3D [Sweep3D] models a 3D discrete ordinates neutron transport and represents the heart of a real ASCI application. This code was developed at LLNL and is included in the ASCI Blue Benchmark Suite. We generated two traces from running the program using 16 and 32 processes. The compaction rate for the trace generated from running 16 processes is 44% as shown in Table 3.7. However, the compaction gain increased when for traces generated from running the program on 32 processes. This shows that for larger traces (with more processes) the gain achieved may be higher.

- **SMG2000**

SMG2000 [SMG2000] is a parallel semicoarsening multigrid solver applied for linear systems based on finite difference, finite volume or finite element discretization of the diffusion equation on logical rectangular grids. In the case of SMG2000, we tested the compaction algorithm on three traces generated from running the program on 16, 32, and 64 processes respectively. As can be seen in Table 3.7, the compaction rate in the three cases is around 50%. SMG2000 is a very complex system in terms of inter-process communication and shows to have many different patterns.

As shown in Table 3.7, we have clearly demonstrated that using MTF results in a significant reduction in the number of model elements, which in our point of view, can improve the scalability of analysis tools. It is worth mentioning that the compaction algorithm took in some cases several hours to complete which necessitates the search for faster algorithms such as the VF2 [Cordella 01] and *nauty* [McKay 81] algorithms that have linear time and space complexities.

Table 3.7. Empirical Results (#P: number of Processes, N: number of Nodes, E: number of Edges, A =∑(N0, E0, M0) , B = ∑(Nc, Ec, Mc), CR: the Compaction Rate = (1 – B / A) * 100%, M :number of Messages, 0: before compaction, c: after compaction)

	#P	**N0**	**E0**	**M0**	**A**	**Nc**	**Ec**	**Mc**	**B**	**CR (%)**
CG	16	3509121	3509105	47104	7065330	561	1479281	42716	1522558	78
CG	32	7139585	7139553	134656	14413794	1121	3039969	119252	3160342	78
MG	16	609874	609858	11024	1230756	648	608280	7588	616516	49
MG	32	692690	692658	21728	1407076	561	689428	15001	704990	50
LU	32	10473947	10473915	1644936	22592798	1518	6009007	986054	6996579	69
LU	64	18310623	18310559	3542924	40164106	2990	12088359	2046493	14137842	68
SP	64	9525649	9525585	1232256	20283490	2881	4340289	1112352	5455522	73
SP	100	14359525	14359425	2406600	31125550	4501	8465901	2188443	10658845	66
WRF	16	272373	272357	25680	570410	8779	245752	21881	276412	51
Sweep	16	962244	962228	239616	2164088	546	960772	239472	1200790	44
Sweep	32	4867550	4867518	1181578	10916646	672	4867518	380198	5248388	52
SMG	16	2095262	2095246	489148	4679656	336	2095246	179543	2275125	51
SMG	32	2084228	2084196	519168	4687592	1090	2081284	518902	2601276	44
SMG	64	10593512	10593448	2662152	23849112	1344	10593448	778816	11373608	52

3.8.2 Querying MTF

In Table 3.8, we present part of the results obtained by querying the MTF trace data using our proposed query language. Since collective operations are executed on all processes simultaneously, we can see that all the processes execute the same number of collective operations as expected. Also, since the program uses non-blocking point-to-point operations, we noticed that the MPI_wait operation was used by all processes to represent non-blocking calls. For example, Process 5 has 3210 MPI_wait operations that were used to detect the completion of the 1605 MPI_Isend and 1605 MPI_Irecv operations. Finally, the size of data helps in identifying which process or processes have the highest load in the program.

Table 3.8. MPI Trace Statistics

P	**Init**	**Fin**	**Wait**	**Bcast**	**Gather**	**Scatterv**	**Isend**	**Irecv**	**Sent (bytes)**	**Received (bytes)**
P1	1	1	2140	640	120	60	1070	1070	159205808	565756448
P2	1	1	3210	640	120	60	1605	1605	213522608	186419232
P3	1	1	3210	640	120	60	1605	1605	213522608	186419232
P4	1	1	2140	640	120	60	1070	1070	158508560	131405184
P5	1	1	3210	640	120	60	1605	1605	236278352	209174976
P6	1	1	4280	640	120	60	2140	2140	289913264	262809888
P7	1	1	4280	640	120	60	2140	2140	289913264	262809888
P8	1	1	3210	640	120	60	1605	1605	234899216	207795840
P9	1	1	3210	640	120	60	1605	1605	236278352	209174976
P10	1	1	4280	640	120	60	2140	2140	289913264	262809888
P11	1	1	4280	640	120	60	2140	2140	289913264	262809888
P12	1	1	3210	640	120	60	1605	1605	234899216	207795840
P13	1	1	2140	640	120	60	1070	1070	159198128	132094752
P14	1	1	3210	640	120	60	1605	1605	213522608	186419232
P15	1	1	3210	640	120	60	1605	1605	213522608	186419232
P16	1	1	2140	640	120	60	1070	1070	158508560	131405184
Total	16	16	51360	10240	1920	60	25680	25680	3591519680	3591519680

We also provide an example of using two of the implemented queries in the MTF metamodel to query information about the trace generated from running SWEEP3D on 16 processes.

- We queried the number of point-to-point messages exchanged between each pair of processes in the program. The results show that based on a rectangular grid, each process is communicating with its direct neighboring processes only. All neighboring processes sent 4992 messages to each other.
- In running the query for calculating the size of data sent from on process to another, the results showed that all the pair processes exchanged the same amount of data (38338560 bytes).

This shows that MTF queries are able to collect detailed information from the target traces that can be used for statistical analysis of the execution trace.

3.9 Summary

We presented a new exchange format for MPI traces generated from HPC applications, called MTF. MTF is built with the requirements for a standard trace exchange format. We provided a detailed specification of the abstract syntax (metamodel) of MTF in the form of a UML class diagram and an associated documentation. We also discussed the syntactic form that should be used with MTF. We also presented the main components in MTF that will be part of a toolkit for generating and querying MTF traces. MTF is lossless but traces can be represented using a compact format as a directed acyclic graph constructed from the original routine call tree. MTF supports different levels of abstractions such as inter-process communication traces and routine call traces. Finally, we showed how MTF can represent large MPI traces generated from different MPI HPC programs and benchmarks.

Additionally, we tested MTF using different queries supported using the proposed query language.

Chapter 4. Communication Pattern Detection

4.1 Introduction

High Performance Computing (HPC) systems that use the message passing paradigm for inter-process communication tend to follow specific communication patterns throughout their execution. These communication patterns play an important role in the analysis of HPC by providing detailed views of the inter-process communication behaviour in the program. These views can in turn help in the understanding of the overall program behaviour. Moreover, they provide useful information about the parallel programs such as their parallel structures and communication topologies. This information can be further exploited for debugging and the validation of the actual behaviour with respect to the intended inter-process communication.

However, as the system undergoes several ad-hoc maintenance tasks, it becomes difficult to know which patterns are being supported. This is further complicated by the fact that documentation is rarely updated when changes to the system are made, making it almost impossible to know which parts of the system follow specific communication patterns. Several approaches for detecting repeating communication patterns in parallel programs [Preissl 08, Kunz 97, Ma 09] have been proposed. However, these approaches are purely syntactic. In other words, they treat a message passing trace as a mere string for which they apply the pattern matching methods. This often results, as we will show in this chapter, in a large number of patterns among which many of them are noise. These approaches do not guarantee the detection of all valid patterns either. To further complicate matters, using these techniques, software engineers need to identify the valid patterns among all the ones that are detected. This task is usually done manually, which hinders the practical value of these approaches. There is therefore a need for techniques that can automatically identify valid patterns.

In this thesis, we present a pattern detection approach that uses additional information about a trace to guide the detection process. More precisely, we use the routine calls invoked in an MPI process trace to act as delimiters that can indicate the beginning and end of valid patterns. The objective is to improve the quality of the detected communication patterns as well as reducing the number of false positives.

In addition to this, we propose another algorithm that 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 [Palma 09].

The rest of the chapter is organized as follows. Section 4.2 gives an overview of the communication patterns. Section 4.3 presents the main approach for communication pattern detection and matching. The repeating communication patterns detection approach and the algorithms for detecting repeating patterns on each process trace separately are presented in Section 4.4. Section 4.5 presents the communication pattern matching algorithm. Section 4.6 presents the algorithm for removing contiguous repeats in message passing traces. The communication patterns construction algorithm is presented in Section 4.7 followed by a case study in Section 4.8. Finally, the chapter is summarized in Section 4.9.

4.2 Communication Patterns

An inter-process communication pattern describes the way several program processes interact to accomplish a specific task. HPC applications may have one or more communication patterns throughout their execution. Generally, a pattern can be viewed as a sequence of events that are repeated non-contiguously in a trace. In parallel programs, a communication pattern is more complex than that since it involves multiple processes - each represented in a trace file that we call a process trace. We refer to patterns that are repeated in one process trace as process patterns. A communication pattern is usually a collection of process patterns.

MPI communication patterns may involve point-to-point operations (operations that involve 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), and MPI_Gather (this is used by a process to collect information from other processes).

An example of a communication pattern is shown in Figure 4.1. The figure 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 on the partner processes, a communication pattern will be generated. The figure represents a 2D-nearest-neighbor communication pattern (with a 4 x 1 process topology) that is repeated three times at different locations in the graph. Non-MPI events are represented using dark bars. The graph that we used to depict the communication events is the event graph [Kranzlmüller 00] where time is on the x-axis and the events flow from left to right.

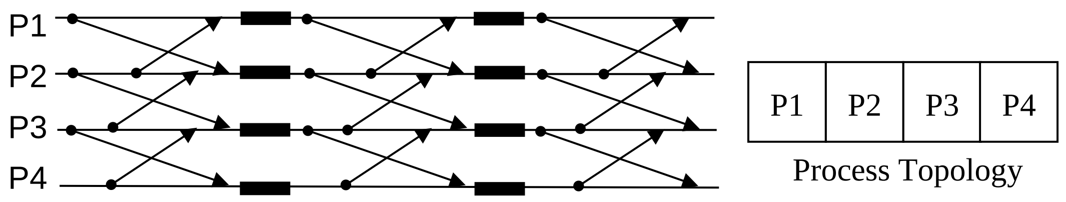


Figure 4.1. Repeating Communication Pattern (top) and Process Topology (bottom)

A process topology is the way the processes are arranged in a certain structure. MPI has two types of process topologies which are the Cartesian (this example) and the graph topologies [MPI].

When detecting communication patterns, we look for the way the program processes are communicating and not what data they are exchanging. For example, each pattern instance in Figure 4.1 may have different data but the processes are still communicating based on the same pattern.

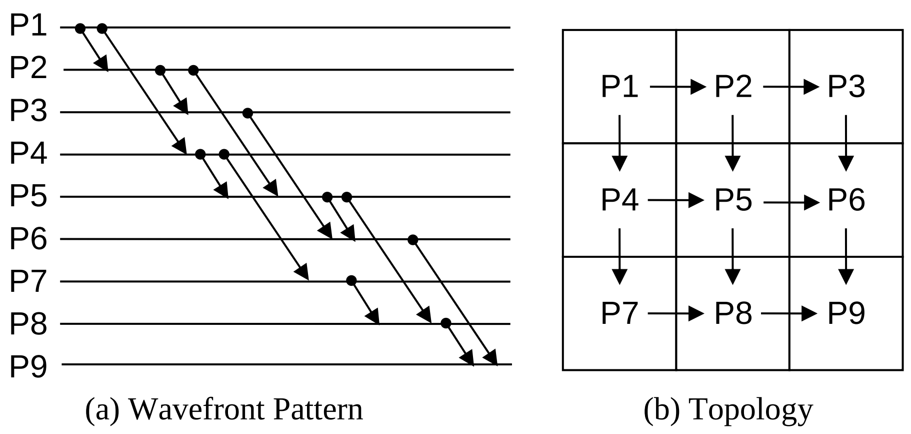


Figure 4.2. The wavefront pattern and topology

In addition, some known communication patterns are well documented in the literature [Palma 09]. They are often used as guidelines for the proper way to implement an inter-process communication mechanism (for more details about the list of documented communication patterns, please refer to [Palma 09]). For example, Figure 4.2a 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 process topology in Figure 4.2b. A wavefront pattern represents a sweep where processes should first receive the messages from other processes before sending to the next ones. For example, P5 should first wait for messages from P2 and P4 before sending to P6 and P8.

Figure 4.3 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 4.3a) 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 4.3b is a communication pattern that is used to implement All-to-All MPI collective operations.

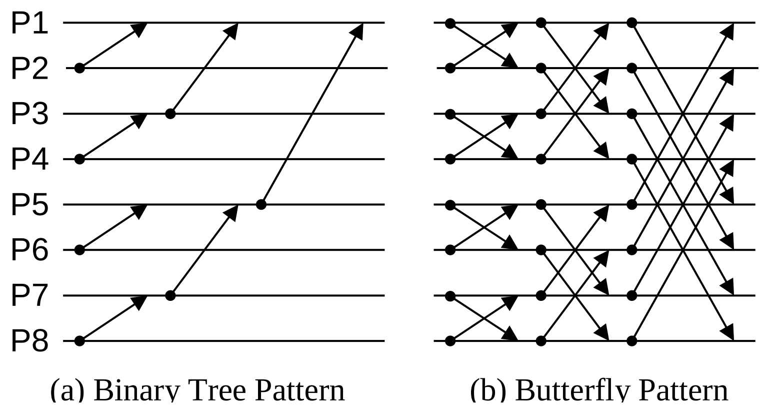


Figure 4.3. Examples of known communication patterns

Detecting communication patterns from message passing programs helps software engineers in understanding the inter-process communication behaviour in these programs by providing abstract views from the whole execution trace. Also, it has been shown that these patterns can help software engineers in debugging MPI applications and in performance optimization [Preissl 08]. For example, a software engineer may decide to replace a point-to-point communication pattern by collective operations [Preissl 10]. 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 as a whole and determining the different computational phases in the program.

4.3 Overall Approach

The objective of this chapter is two-fold: (a) detecting patterns in MPI traces no matter if they are among the documented ones or not, and (b) searching if a given pattern exists in a trace to help software engineers verify if the processes in the traced scenario communicate according to a known communication pattern. We anticipate 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.

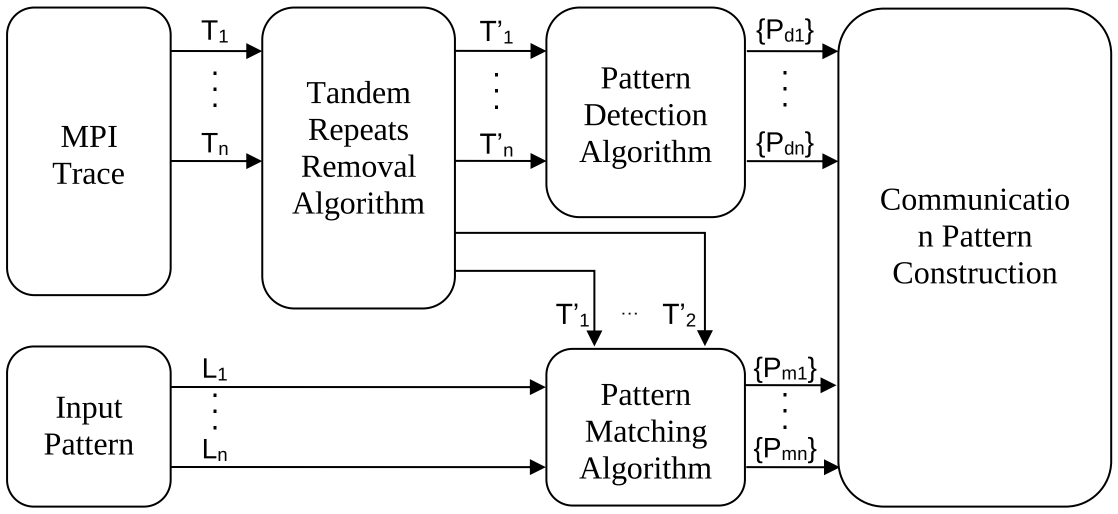

MPI Trace
T1
Tn
Tandem Repeats Removal Algorithm
T'1
T'n
Pattern Detection Algorithm
{Pd1}
{Pdn}
Communication Pattern Construction
T'1
...
T'2
Input Pattern
L1
Ln
Pattern Matching Algorithm
{Pm1}
{Pmn}

Figure 4.4. Pattern detection and pattern matching approach

The approach for achieving both objectives is presented in Figure 4.4. In both cases, we first start 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 ignoring 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… 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 Chapter. An additional step that may be required before the detection and matching processes start is the removal of contiguous (or tandem) repeats from each process trace separately. Removing contiguously repeating events may reduce the trace size and improve the quality of detected patterns. The algorithm for removing contiguously repeating MPI events is discussed in the chapter.

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. In the following, we present each algorithm in the presented approach in a separate section.

4.4 Repeating Communication Patterns Detection

In this section, we present the communication patterns detection approach in MPI traces. The main idea is to initially detect the repeating patterns on each process trace and then construct the communication patterns by matching the partner repeats (patterns) found on different processes in the program.

Each process trace can be viewed as a stream of events which contains repeating sequences of events. 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 p_1 as the start position of Substring S_1, p_2 the start position of substring S_2, and l is their length):

1. Tandem (contiguous) Repeats: repeats that are directly adjacent to each other. Given a string S of length n, a Tandem repeat in S is a tuple (p_1, p_2, l) such that

   $\exists$ S[p_1 .. $p_1 + l - 1$] = S[p_2 .. $p_2 + l - 1$] and $p_2 > p_1$ and S[p_2 - 1] = S[$p_1 + l - 1$].
2. Maximal (interspersed) Repeat: a *repeat* that cannot be extended to the left and to the right. Given a string S of length n, a maximal repeat in S is a tuple (p1, p2, l) such that

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

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

When considering each process trace as a string that contains message passing events, we can utilize existing data mining techniques to detect the repeating patterns in each process trace. The main advantage of this approach is that it only deals with the message passing events which makes the trace size smaller than when considering other kinds of events such as routine calls. However, this approach has numerous disadvantages.

- It may result in a large number of patterns with many patterns as false positives due to three main reasons. First, a pure syntactic approach allows the detection of overlapping patterns; this case can be easily seen in Figure 4.5. Second, many detected patterns might end up as a combination of other patterns (a combination of valid and invalid patterns). Finally, in many cases it is difficult to determine the beginning of the pattern. For example, when considering this process trace of message passing events 'R3S2S3R2R3S2S3R2R3S2S3R2R3S2S3R2R3' (where S2 means ‘Send to process 2’ and R2 means ‘Receive from process 2’), the sequence 'R3S2S3R2' will be detected as the pattern. This is due to the existence of the R3 event at the beginning of the trace. However, the real pattern in this case is 'S2S3R2R3'. This case can be easily found when testing the pattern detection algorithm on different trace files.

- Some valid patterns may not be discovered at all since they exist within a larger invalid pattern. This usually occurs when the trace has a large number of events and there are a lot of repetitions in the trace.

Considering only message passing events results in the detection of very long patterns that are a composition of different adjacent patterns that are repeated in the same sequence in the trace. Therefore, processing all of these patterns is time consuming and requires in many cases the user's intervention in order to determine the valid patterns.

Sequence: (mirrors a process trace generated from Sweep3D)

abababacacacbdbdbdadadadabababacacacbdbdbdadadadabababacacacbdbdbdadadad

Detected Patterns: Number in brackets shows how many times the maximal repeat occurs in the sequence above

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

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

Figure 4.5. Pattern Detection Based on Syntactic Methods

Some of these limitations can be illustrated in the example of Figure 4.5, which is taken from a real system execution. The presented sequence simulates a large trace that is generated from running the Sweep3D [Sweep3D] program. We denote the MPI events as symbols for simplicity. The valid communication patterns for this application are known and documented in [Sweep3D]. Sweep3D implements a wavefront pattern with a sweep from each corner in the process topology to its opposite corner. The example shows that 16 patterns were detected despite the fact that only five patterns are valid patterns. In addition, the approach missed two valid patterns 'ab' and 'ad'. This shows that when applying a pattern detection approach directly to a trace of message passing events alone the quality of the detected patterns is low. The longest valid pattern 'ababababacacacbdbdbdadadad' is a supermaximal repeat that can be composed from the smaller valid patterns.

4.4.1 Detailed Repeating Patterns Detection Approach

Figure 4.6 presents our detailed approach for detecting communication patterns in MPI traces. First, the traces of MPI operations and routine calls are collected. Then, we build the routine call tree for each MPI process. This can be done by simply computing the nesting level (using the event entry and exit events) for each routine call (including the MPI events which occur at the leaf level in the tree). Therefore, the whole routine call tree does not need to be present in memory at the same time. We extract the MPI events from the trace along with the routine calls that occur directly at the higher level in the call tree.

The routine calls with their timestamps will generate unique constructs in the trace and will not appear in any detected pattern since they exist only once in the trace (the timestamp is unique for each routine call). This will guarantee the detection of accurate patterns since the routine event can identify the start and end positions of the repeats. In some cases, when the direct callers of the MPI routines are wrapper functions, the routine call events at the direct higher nesting level will be selected instead.

The size of the trace can be reduced by removing the contiguous repeats before the detected process at Step 3. Also, another advantage of removing the contiguous repeats is that it enables the detection of patterns in their general form. For example, 'ababcdcdefef' can be represented as 'abcdef' when removing the contiguous repeats in the trace. After detecting all the process patterns in Step 4, the construction of the communication patterns will be handled using the communication pattern construction algorithm in Step 5. All the detected patterns will be then stored in the pattern database, which is the result of the approach.

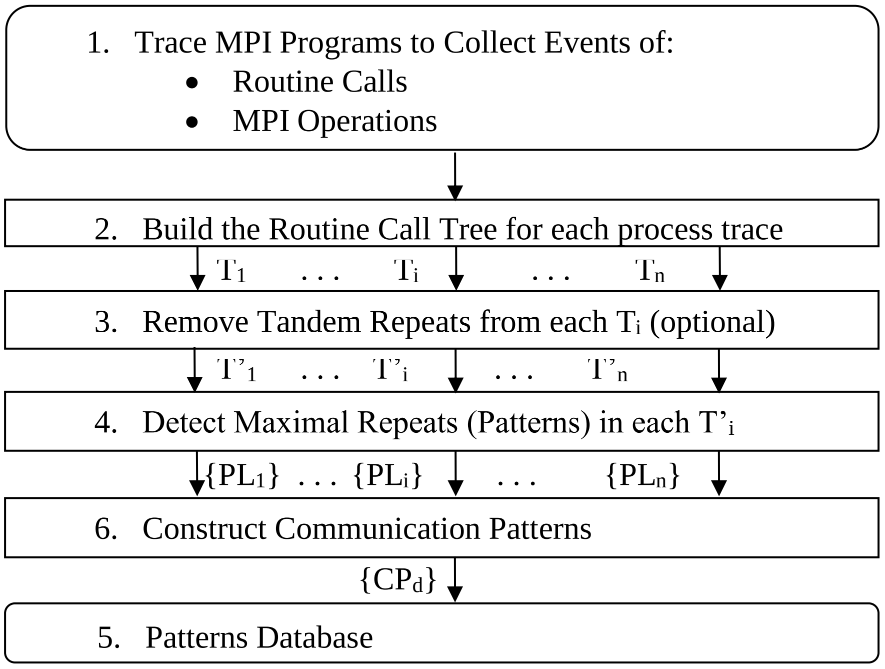

1. Trace MPI Programs to Collect Events of:
• Routine Calls
• MPI Operations
2. Build the Routine Call Tree for each process trace
T1 . . . Ti . . . Tn
3. Remove Tandem Repeats from each Ti (optional)
T'1 . . . T'i . . . T'n
4. Detect Maximal Repeats (Patterns) in each T'i
{PL1} . . . {PLi} . . . {PLn}
6. Construct Communication Patterns
{CPd}
5. Patterns Database

Figure 4.6. Detailed Repeating Pattern Detection Approach

In the following, we detail on the two different versions that are used in the process repeating patterns detection. The tandem repeats detection algorithm and the communication patterns construction algorithm are presented in Section 4.6 and Section 4.7 respectively.

4.4.2 Process Repeating Patterns Detection

The pattern detection algorithm uses the concept of n-grams found in statistical natural language processing. In the classical n-gram pattern detection approach [Karp 72], the algorithm looks for all n-size patterns in a string. However, this approach is too costly especially when used for long strings with unknown patterns sizes. Therefore, we developed a new algorithm that detects patterns as it goes through the trace. We used bi-grams (length = 2) as the minimum length of a pattern. The pattern length increases whenever a new occurrence is detected. This is borrowed from the LZW data compression algorithm [Welch 84], where whenever a sequence already exists in the pattern database, the algorithm appends the next character in the text to the end of the sequence. However, our algorithm differs from the LZW algorithm in that it tries to detect a pattern at the other positions of its prefix pattern ('ab' is the prefix of 'abc'). This algorithm runs on each process trace separately and detects all process patterns which will then be input to the communication pattern construction algorithm. We developed two versions of the n-gram based pattern detection algorithm. In the following section, we present the Reverse Pattern Lookup Algorithm followed by the Reverse-Forward Pattern Lookup algorithm in Section 4.4.2.2.

4.4.2.1 Reverse Pattern Lookup Algorithm

In this section, we present our initial version of the pattern detection algorithm. Algorithm 4.1 uses 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 (LZW starts by reading a character from the string), 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 is similar to the LZW approach). This occurs by the call to the *ConstructNGram* function at line 18, which is a utility function that adds the next event in the trace to the current 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 = checkPatternOccurence(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 4.1. Reverse Pattern Lookup 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* function at line 21 (this is not part of the LZW algorithm and this is the main difference that enables our algorithm to detect complete maximal repeats).

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 list at line 28.

We also present the C*heck Pattern Occurrence* in Algorithm 4.2. This algorithm is being called by the code presented in Algorithm 1 as *'checkPatternOccurence'* or Ƭ 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'*). 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.

CheckPatternOccurrence: *checks if nextPattern can be constructed from the previous positions of current Pattern.*

Returns *true* if *nextPattern* can be found at its *prefixPattern's* 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 EQ 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 4.2. Check pattern occurrences

Figure 4.7 presents an example of 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?**	**Tand?**	**Freq.**	**Pos.**	**Next Action**
1	S2	S3			Yes	No	1	1	Get next bi-gram
2	S3	R2			Yes	No	1	2	Get next bi-gram
3	R2	S5			Yes	No	1	3	Get next bi-gram
4	S5	S2			Yes	No	1	4	Get next bi-gram
5	S2	S3			No	No	2	1, 5	Const. from cur. 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	Get next bi-gram
8	S4	S2			Yes	No	1	14	Get 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

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

Figure 4.7. Reverse Pattern Lookup 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 unigram 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 5. 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.

4.4.2.2 Reverse-Forward Pattern Lookup Algorithm

In this section, we present a modified version of the algorithm presented in the previous section. The first difference in this algorithm is that we first detect all the bi-grams in the trace along with their starting positions. Another difference is that in the previous versions, the algorithm checks for contiguous repeats (tandem repeats) as it reads the events from the trace. In this version, we do not check for tandem repeats as this step is currently handled before the detection process. The first line in Algorithm 4.3 calls the *ExtractBiGrams* (presented in the code snippet below) routine at line 1 which is responsible for the extraction of all the bi-grams and their start positions in the trace. After detecting all the bi-grams, it removes the bi-grams that only exist once in the trace since they are not part of any repeating pattern. The advantage of having the bi-grams information available prior to the detection process is that the algorithm will be able to construct a pattern at all its positions that start with its prefix bi-gram directly after encountering its first occurrence in the trace. In the previous algorithm, it is only possible to detect a pattern after encountering its starting bi-gram at least twice in the trace. This is one of the main differences between the current version and the previous one.

In *ExtractBiGrams*, whenever a bi-gram at position *i* is read it will be added to the pattern-positions-list of that bi-gram. The algorithm will continue reading the remaining bi-grams until it reaches the end of the trace. At this stage, all detected bigrams will have a frequency of zero to indicate that they are still not part of the final detected patterns-list. The frequency attribute will be updated during the pattern detection process.

Routine: ExtractBiGrams

```
1. for i = 0 to trace.size - 2
2.     bigram = trace[i] + trace[i+1]
3.     if bigram ∉ PatternsList then
4.         addToPatternList(bigram)
5.         bigram.frequency = 0
6.     end for
7.     addToPositionsList(bigram, i)
8.     remove bigrams with one position only
9. end for
```

Algorithm 4.3 continues at line 2 by reading the first bi-gram from the trace. This while-loop will stop when it reaches the end of the trace. At line 3, the if-statement will check whether the new pattern (bi-gram) exists in the patterns-list (all bi-grams were detected and added to the patterns-list along with their starting positions in the trace prior to the detection process, only those bi-grams that exist only once were removed from the patterns-list). At line 4, the algorithm enters the do-while loop where the actual detection logic exists. Line 5 defines the *overlap* variable that holds the number of overlaps the pattern has. Two occurrences of a pattern overlap when the last event's position in the first occurrence is greater than the start position of the second occurrence of the pattern. For example, the two occurrences of pattern *aba* in *ababa* overlap and *overlap* will be equal 1. Line 6 defines the *distance* variable which holds the distance between the start positions of the first two overlapping occurrences of a pattern.

Algorithm: Pattern Detection – Maximal Repeats Detection
This algorithm runs for each process separately to find repeating patterns

```
1.  ExtractBiGrams
2.  while [(pattern = nextBiGram) is not null]
3.    if pattern ∈  PatternsList then
4.     do
5.        overlap              = 0   // number of overlapping patterns
6.        distance             = -1  // distance between two overlapping patterns
7.        matches              = 0   // number of matches for new pattern
8.        prevPattern          = pattern //points to the previous pattern
9.        latestEvent          = pattern.addNextEvent() //add next event to pattern
10.       if latestEvent is null OR not MPI_EVENT then break
11.       if pattern IS NOT NEW then continue //get next bi-gram
12.       else UpdatePatternList(pattern , pattern.position)
13.       prevPosition = prevPattern.firstPosition
14.       prevMatch = false
15.       for i = 0  to  prevPattern.positions.size
16.          currentPosition = prevPattern.positions.get(i)
17.          nextEventIndex = currentPosition + pattern.length - 1
18.          if nextEventIndex GT trace.size - 1 then break
19.          if trace.get(nextEventIndex) EQ latestEvent then
20.                if currentPosition NE pattern.position then
21.                   nextPattern.add(currentPosition)
22.                   matches++
23.                end-if
24.                if currentPosition GT  prevPosition AND prevMatch AND
                             currentPosition – pattern.length LT prevPosition then
25.                    overlap++
26.                    if distance EQ -1 THEN distance = currentPosition – prevPosition
27.                end-if
28.                prevMatch = true
29.           else prevMatch = false
30.           end-if
31.              prevPosition = currentPosition
32.       end-for
33.       if overlap GT 0 then
34.            if overlap GT matches – distance then  matches = 0
35.            if overlap EQ matches then
36.                   pattern.lineIndex = pattern.lastPosition + distance
37.            else   nextNGram.lineIndex = pattern.position + pLength - 1
38.       end-if
39.       if matches EQ  0 then
40.           remove pattern from PatternsList
41.       else
42.            pattern.incrementFrequency(matches - overlap)
43.            prevPattern.decrementFrequency(matches - overlap)
44.        end-if
45.      while matches GT  0 //do-while loop
46.   end-if
47. end-while
```

Algorithm 4.3. Reverse-Forward Pattern Lookup Algorithm

The *matches* variable is used to calculate the number of matches a pattern has at the other occurrences (except the current position) of its prefix pattern (previous pattern). Therefore, when all occurrences of a pattern are overlapping, the number of matches will be equal to the number of overlaps. For example, the first *aba* in the previous example has one overlap and one match in the trace which means that it is not a true pattern.

The *prevPattern* holds the value of the previous pattern. The *latestEvent* variable gets the value of the next event in the trace (the event to the right of the current pattern) at Line 9. At this point, the *pattern* variable has one extra event and the *prevPattern* holds the value of *pattern* prior to appending the new event. Line 10 will check whether the *latestEvent* is *null* (end of the trace) or if the event is not an inter-process communication event. If the event is *null* then it means that the detection process is complete. If the event is not a message passing event then the algorithm will read a new bi-gram from the trace. At line 11, if *pattern* already exists in the *patterns-list*, then it means that this pattern was already detected and the algorithm will continue to read the next bi-gram in the trace. The next bi-gram starts at the position of the *latestEvent* in the trace unless it is a non-message passing event. If *pattern* is not in the *patterns-list*, then it will be added along with its current position at line 12. At line 13, the *prevPosition* variable will be set to hold the value of the first position of *prevPattern*. The variable *prevMatch* (set to *false* at line 14) will be used later and it indicates whether there was a match or not at the other pattern position. The *for-loop* at line 15 will iterate over all the positions of the *prevPattern*. This loop contains the logic that is used to verify whether *pattern* exists at the other positions of its prefix pattern (*prevPattern*). At line 16, the *currentPosition* will get the value of the *i*-th position of *prevPattern* positions. Line 17 will calculate the position of the next event (*nextEventIndex)* in the trace that will be appended to *prevPattern.* Line 18 will check if the value of the *nextEventIndex* is still less than the size of the trace. If it is larger than the trace size, then the loop will break and the algorithm will read the next bi-gram if it did not yet reach the end of the trace. Line 19 will check if the event at *nextEventIndex* is equal to the *latestEvent* read at line 8. If the two events are equal then it implicitly means that *pattern* exists at the *i*-th position of *prevPattern*. In the previous version, we were comparing the pattern as a whole which is not necessary. The *if-statement* at line 20 will check if *currentPosition* does not exist in the *positions-list* of *pattern*. If the condition is *true*, *currentPosition* will be added to the *positions-list* of *pattern* at line 21. At line 22, the *matches* variable will be incremented since there is a match. The condition at line 24 will check if the two occurrences of the pattern are overlapping. First the condition will check if *currentPosition* is greater than *prevPosition* and then it will check if there was a previous match using the *prevMatch* variable.

Finally, the condition will check if the expression '*currentPosition – pLength (*pattern length*)'* is less than the value of previous position. If this condition is met, then it means that there is an overlap and the value of *overlap* will be incremented at line 25. At line 26, the algorithm calculates the distance between the two overlapping occurrences of the pattern. This value is only calculated for the first overlapping pair of a pattern. That is why we initialize *distance* value to -1. At line 28, the *prevMatch* is set to *true* when there is a match, otherwise it will be set to *false* at line 29. Line 31 assigns the value of *currentPosition* to *prevPosition* to use it in the next iteration of the *for-loop*. When the *for-loop* iterates on all the positions of *prevPattern*, the condition of the *if-statement* at line 33 is evaluated. The expression after the *if-statement* at line 34 resets the value of *matches* to *zero* if the condition (*overlap* > *matches* – *distance*) is met. In the following example, we show why this expression is being used. Consider the following trace which has a long repeating pattern:

ababababacacacbdbdbdadadad ababababacacacbdbdbdadadad ababababacacacbdbdbdadadad

For the first pattern 'ab', when we add the next event it will be 'aba' that will have *matches* = 8, *overlap* = 6 and *distance* = 2 which means that the condition will return *false* and the matches will not be reset to zero. If this condition was true then the long pattern 'ababababacacacbdbdbdadadad' will not be detected. Therefore, using this expression, longer true patterns that are composed of overlapping shorter patterns can be detected. In the previous version of the algorithm we did not have this validation. Therefore, a case like this example which is a snapshot of a Sweep3D trace will not have the long pattern detected which represents the global communication behaviour.

If the number of overlaps is equal to the number of matches, then it means that all the occurrences of the pattern are overlapping. Therefore, at line 36, the pointer that reads from the trace is advanced *distance* steps to the right of the last position of previous pattern otherwise the pointer will be moved to the last event of the current pattern at line 38. For example, when the long pattern 'ababababacacacbdbdbdadadad' is extended to the right, it will be 'ababababacacacbdbdbdadadada' which is not a true pattern. However, *overlap* will be equal to *matches* which is equal to 2 in this case and the pattern will be removed from the *patterns-list*. In the case of suffix tree, this longer pattern will be returned as a repeat at the end of the detection process. The statement at line 40 will be executed if there were no matches to *pattern* in the trace which will remove *pattern* from the *patterns-list*. If there were matches to *pattern* then the *frequency* of *pattern* will be incremented by '*matches* -

overlap' and the *frequency* of *prevPattern* will be decremented by '*matches - overlap*' (lines 42 and 43 respectively). If there were matches to *pattern* then the *do-while* loop will continue and it will append another event to *pattern* which will continue until there are no more matches.

Finally, if the algorithm did not reach the end of the trace, it will read a new bi-gram at line 2 and the algorithm will execute until the end of the trace. At the end of the algorithm, all the patterns will be detected but only the ones with frequency more than 1 will be considered as true patterns.

In Figure 4.8, we present the same example presented in the previous section in order to outline the main differences in the two versions of the algorithm. The first step (1) is to extract the bi-grams (bi-grams table) and then remove the patterns that exist only once in the trace. Then, the pattern detection will start at step (2). The first bigram in the trace is S2S3 which already exists in the *patterns-list* (extracted using the *ExtractBiGrams* routine). The frequency of S2S3 will be set to 5 at this stage. The advantage here is that we already know all the positions that this bi-gram exists at. Therefore, we can check if we can extend it to a larger pattern at all its positions early in the detection process. In the first version, at this point we only know that this bi-gram exists at position 0 only. By appending the event at line 2 in the trace, the pattern S2S3R2 will be our next candidate. The algorithm will check if it exists at the other positions of its prefix pattern S2S3. Step 2 in the detection process shows that S2S3R2 exists at all other positions of S2S3.

Therefore, the pattern is detected at an earlier stage in the detection process. The frequency of S2S3 will be decremented to 0 and the frequency of S2S3R2 will be 5. Then, the event at line 4 will be appended to S2S3R2 resulting in the longer pattern S2S3R2R5.

(1) 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

(1) Extracting BiGrams Table

	BiGram	**Positions**
1	S2S3	1, 5, 8, 11, 15
2	S3R2	2, 6, 9, 12, 16
3	R2S5	3
4	S5S2	4
5	R2S2	7, 10
6	R2S4	13
7	S4S2	14

$\rightarrow$

	BiGram	**Positions**
1	S2S3	1, 5, 8, 11, 15
2	S3R2	2, 6, 9, 12, 16
3	R2S2	7, 10

(2) Pattern Detection Execution

#	**Pattern**				**i**	**New?**	**Positions**	**Next Action**
1	S2	S3			1	No	1, 5, 8, 11, 15	Add next event to S2S3 → S2S3R2
2	S2	S3	R2		1	Yes	1, 5, 8, 11, 15	Does S2S3R2 exist at positions 5 → add 5 Does S2S3R2 exist at positions 8 → add 8 Does S2S3R2 exist at positions 11 → add 11 Does S2S3R2 exist at positions 15 → add 15
3	S2	S3	R2	S5	1	No	1	Check if it exists at all positions of S2S3R2 → It only exists once → remove it from the db
4	S5	S2			4	Yes	3	Not in BiGrams table → get next bi-gram at 5
5	S2	S3			5	No	1, 5, 8, 11, 15	Add next event to S2S3 → S2S3R2
6	S2	S3	R2		5	No	1, 5, 8, 11, 15	Already exists → Append event at position 8
7	S2	S3	R2	S2	5	Yes	5, 8	Does S2S3R2S2 exist at positions 1 → No Does S2S3R2S2 exist at positions 8 → add 8 Does S2S3R2S2 exist at positions 11 → No Does S2S3R2S2 exist at positions 15 → No → matches = 1 and overlap = 1 → remove pattern and advance pointer to 11
8	S2	S3			11	No	1, 5, 8, 11, 15	Not in BiGrams table → get next bi-gram
9	S2	S3	R2		11	No	1, 5, 8, 11, 15	Add next event to S2S3 → S2S3R2
10	S2	S3	R2	S4	11	No	1, 5, 8, 11, 15	Does not exist at any other positions → remove
11	S4	S2			14	Yes	14	Not in BiGrams table → get next bi-gram at 15
12	S2	S3			15	No	1, 5, 8, 11, 15	Add next event to S2S3 → S2S3R2
13	S2	S3	R2		15	No	1, 5, 8, 11, 15	End of Trace

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

Figure 4.8. Reverse-Forward Pattern Lookup Example

However, this pattern does not exist at any other positions of S2S3R2 and will be removed from the *patterns-list*. The algorithm will then read the bi-gram at line 4 which does not exist in the bi-grams list. Therefore, it will not be added to the patterns-list and the next bi-gram will be read from the trace. S2S3 at line 5 already exists in the trace therefore the algorithm will add the event at line 7 resulting in S2S3R2 which already exists in the *patterns-list*.

The event next to S2S3R2 at position 8 will be appended resulting with the pattern S2S3R2S2. This pattern exists at two overlapping locations. According to the algorithm, since matches = 1 and overlap = 1 then the pattern will be removed from the patterns-list and the index *i* will be advanced to position 11. At position 11, the bi-gram S2S3 exists as well as S2S3R2.

Then, S4 at position 14 will be appended resulting in S2S3R2S4 which does not exist at any other position and will be removed as well. Bi-gram S4S2 does not exist in the patterns-list therefore the next bi-gram S2S3 will be read which already exists. Finally, the constructed pattern S2S3R2 already exists in the *patterns-list*. At this point, the algorithm reached the end of the trace and based on the frequencies of the patterns only S2S3R2 will be detected as a true pattern in the trace.

Algorithm's Complexity:

The presented algorithm runs in linear time with respect to the trace size (*n*). The *ExtractBiGrams* routine only requires *n* steps to execute. The complexity of the pattern detection algorithm can be measured as follows:

- Steps required to Execute *ExtractBiGrams*: *n*

- Steps required to read the trace events (lines 2 & 9; together, these two lines will read the trace events from left to right): *n*
- Steps required to Execute Pattern Detection: $\sum^{P} R_i$ *where* P is the total number of repeats (not only the ones with frequency more than 1) and R_i is the number of occurrences for each repeat. The detection of every occurrence of the pattern adds one step to the total execution time. Therefore, for each pattern, the total number of steps that will be added to the total execution time of the algorithm will be the number of its occurrences in the trace (R).

$$Complexity = 2n + \sum_{i=1}^{p} R_i \qquad (4.1)$$

Since the number of repeats in a string is always less than *n* [Grissa 07] and the number of occurrences for each pattern is linear with n it is easy to deduce that the algorithm's complexity will run in O(n).

With respect to the space complexity of the algorithm, it was implemented by representing every detected pattern by a unique hash code. Therefore, since the maximum number of patterns is always less than *n*, the algorithm's memory usage will depend on the number of patterns and the size of the hash code for each pattern. This also guaranties that the algorithm's space usage will grow linear with the size of the input trace.

4.5 Communication Pattern Matching

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 the 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 finds similar patterns on each process trace separately. The output of this algorithm is input to the communication pattern construction algorithm presented later. The degree of similarity between the patterns is determined by the number of shared events between them.

We use the *Edit-Distance* [Levenshtein 66] (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 [Jokinen 91] 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 [Cao 05, Rasmussen 06]. The Lemma is presented in the following:

Lemma: N-gram based Filter (Jokinen and Ukkonen [Jokinen 91])

Let a string S_1 of length m with at most k edit distance from another string S_2 of length m, then at least $m+1-kn+n$ of the n-grams in S_1 occur in S_2.

The process of determining similar patterns consists of two steps. The first step is the filtration process which uses the above lemma, and the second step is the *edit-distance* function. 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 to avoid verifying two non-matching windows using the *edit-distance* function.

```
Pattern Matching: runs for each process separately
p: pattern under study of size m
threshold = pattern size – n + 1 – k.n
k:maximum allowed edit distance
-----------------------------------------------------------------
firstSharedNGramDisplacement: displacement between position of first shared
n-gram in w and its position in the n-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 4.4. Pattern matching

Algorithm 4.4 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 construction of the communication patterns. The *MatchingPatternList* contains the patterns and their start positions in the trace.

We demonstrate our pattern matching algorithm using the example shown in Figure 4.9. 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 (ED) function. 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.

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 4.9. Example of the pattern matching algorithm

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 their 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.

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 all *edit distances* 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.

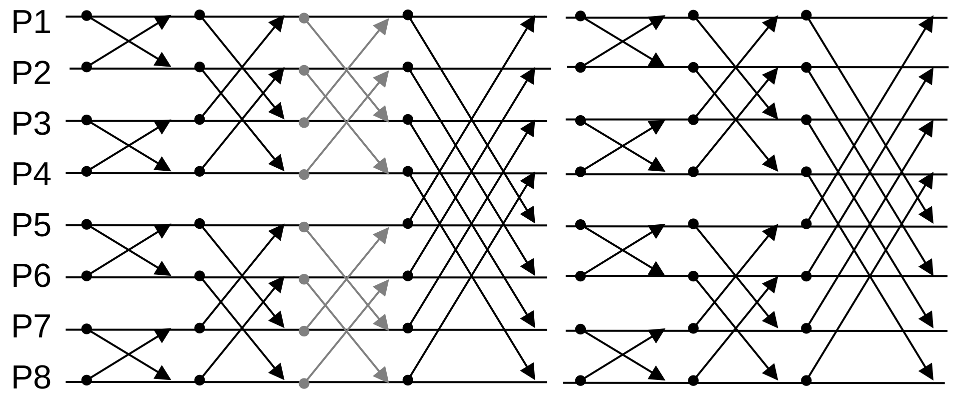
The size of the input communication pattern is based on the number of processes involved in the communication. Therefore, in order to detect a wavefront pattern (for example) on a grid topology of 5x5, the input pattern will be different than when detecting it on a grid of 2x2. Therefore, the knowledge about the communication pattern should be applied in order to extrapolate the pattern from a small process topology to a larger one. For example, the events for Process 1 for a sweep from P1 to P4 in the 2x2 topology will be ‘Send to 2, Send to 3’. However, in case of 5x5 the events for P1 will be ‘Send to 2, Send to 6’.

4.6 Tandem Repeats Removal Algorithm

MPI traces may contain two or more communication patterns that are not identical but correspond to the same communication behaviour. This can be due to:

- The varying number of iterations (loops) at different stages in the program.
- The ordering of events at different stages in the program.
- Different number of events.

In the case of differences caused by loops, in order to detect these similar repeating behaviours in the trace, we need to abstract the trace by removing events caused by these extra iterations. These events appear contiguously in each process trace and can be detected and removed prior to the communication pattern detection process. Patterns detected after removing the contiguously repeating patterns will be in their general form.



(a) An Instance of Butterfly Pattern (b) General Butterfly Pattern

Figure 4.10. Butterfly Pattern with Contiguous Repeats

For example, Figure 4.10 shows two examples of a butterfly communication pattern. This pattern is used in implementing MPI collective operations. Figure 4.10a depicts a butterfly pattern that is not appearing in its general form. By removing the contiguous repeats (grey) from the trace we can represent the pattern in its general form as shown in Figure 4.10b. More complex cases may exist resulting from an excessive number of contiguously repeating patterns.

For the purpose of studying the communication behaviour in MPI programs, these contiguous repeats will only increase the effort of mining the important communication patterns in the trace file. Moreover, in some cases, the excess of these contiguous repeats in the trace will prevent the discovery of the communication pattern. Therefore, there should be a technique to remove these contiguously repeating patterns. Another advantage of removing the contiguously repeating patterns is the reduction of the trace size which will make the process of mining the communication patterns faster.

The removal of contiguous repeats is performed on each process trace separately. We developed this algorithm based on the concept of n-grams. However, in this algorithm we used only bi-grams to help in detecting the tandem repeats on each process trace. This algorithm is iterative; therefore it will be repeated until all the tandem repeats are removed from the trace.

Algorithm 4.5 represents the main loop for detecting the tandem repeats. The algorithm will repeat until the input size is fixed (all tandem repeats are removed). First, the algorithm will extract all adjacent bi-grams from the input trace. Extracted bi-grams and their positions will be stored in a hash table where the key is the bi-gram and the value is the array of starting positions in the trace for each bi-gram. Then, the algorithm will start by reading a bi-gram from the input trace, if the bi-gram exists in the bi-gram hash table then the *detectPossibleTandem* function will be called to detect if there is a tandem repeat. If the value of *index* returned by the *detectPossibleTandem* function is greater than the value of *i*, then it means that one or more tandem repeat was detected and then the bi-gram at *index* position will be read from the input trace. The algorithm will repeat iteratively until all tandem repeats are detected and removed from the trace. The reason why we iterate until the size of the trace is fixed is that when removing some tandem repeats, new tandem repeats may appear in the trace.

```
Detect Tandem Repeats
1.  traceSize = 0
2.  ht // holds the repeats and their positions
3.  tr // the list of tandem repeats and their positions
4.  while (traceSize != trace.size)
5.    Extract All Bi-Grams from trace
6.    Keep Bi-Grams that are repeated in trace
7.    index = 0
8.    for (i = 0; i < trace.size; i++)
9.      current.addTwoGrams(trace)
10.     if current is-not-in-Bi-Grams list then continue
11.         index = detectPossibleTandem(current, i, trace, tr)
12.          if ( index > i ) then
13.               i = index
14.               index = 0
15.               current = ""
16.          end if
17.     end-i-for-loop
18.     traceSize = trace.size
19.     removeTandemRepeats(trace, tr)
20.     clear tr
21.     clear ht
22. end-while
```

Algorithm 4.5. Tandem Repeats Detection

The *detectPossibleTandem* algorithm is presented in Algorithm 4.6. It will verify if there is a contiguous repeat at two consecutive positions of a specific bi-gram (referred to as key in the algorithm). If a contiguous repeat is detected, it will check if there is another tandem repeat right to the already detected one. If the algorithm does not detect any tandem repeat it will return the same initial value of index (index has position as initial value). If one or more repeats were detected, the algorithm will return the value of *index* which is the position of the next bi-gram after the tandem repeat.

```
Detect Possible Tandem
Parameter List: key, position, trace, tr
1.  index = position
2.  positions = get-all-positions-of( key )
3.  i = get index of position in positions
4.  if i is last index in positions then return index
5.  pLength = positions[i+1] – position
6.  $s_1$ = trace.sublist(position, position + pLength)
7.  while (i < positions.size - 2)
8.        $position_1$ = positions[i]
9.        $position_2$ = positions[i + 1]
10.       if ($position_2$ + pLength – 1 $\geq$ trace.size) return index
11.       if ($position_2$ – position1 > pLength) return index
12.           $gram_1$ = trace[$position_2$ – 1]
13.           $gram_2$ = trace[$position_2$ + pLength – 1]
14.           if ($gram_1$ == $gram_2$) then
15.               $s_2$ = trace.sublist($position_2$, $position_2$ + pLength)
16.               if ($s_1$.equals($s_2$)) then
17.                   tr.add($position_2$, $s_1$)
18.                   index = $position_2$ + pLength
19.               else
20.                   return index
21.               end if
22.       else
23.           return index
24.       end if
25.     i++
26. end-while
27. return index
```

Algorithm 4.6. Possible Tandem Repeats Detection

Algorithm 4.6 starts by assigning the value of position to the index variable at line 1. At line 2, the algorithm will get the list of positions (sequence) of the key (bi-gram) passed in the parameters list and will get the index *i* of *position* in the positions list. The possible pattern length *pLength* is calculated at line 5. At line 6 the possible pattern s_1 is extracted from the input. It should be noted that we calculate *pLength* and s_1 before entering the while loop. When the algorithm enters the loop at line 7 it will get the positions values at *i* and *i* +1 using the statements at lines 8 and 9 respectively. At line 10, the algorithm will return if the string starting at *$position_2$* to the end of the trace is less than the *pLength* value. The $gram_1$ and $gram_2$ variables at lines 12 and 13 hold the events from the trace found at positions (*$position_2$* -1) and (*$position_2$* + *pLength* – 1) respectively. If these two values are different, the algorithm does not need to check the whole events to confirm equality. If they are equal, the string of events from *$position_2$* to *$position_2$* + *pLength* will be extracted and then will be checked for equality. If the two strings are equal, then a tandem repeat is detected and the value of *$position_2$* and *s_1* will be added to the tandem repeats hash table. The algorithm will loop until there are no additional tandem repeats detected. The algorithm will return the value of the index just right after the last detected tandem repeat. Figure 4.11 presents an example to illustrate the tandem repeats detection using our algorithm. The figure depicts a trace of 17 events (we use alphabets to represent events for simplicity). We only present the iterations for key = ‘ab’ since it is the only one that will result in the detection of tandem repeats which is shown in the *Execution* part of the figure. The Bi-grams and Positions columns are retrieved in Algorithm 4.5. The two bi-grams ‘dm’ and ‘ma’ will be removed from the list since they occurred only once in the trace and certainly will not help in detecting a tandem repeat. The example shows that there are two tandem repeats ‘abcd’ at positions 4 and 8 respectively. The figure clearly explains the 3 steps that were executed to detect the tandem repeats at the two different locations.

There exist several approaches for detecting tandem repeats using suffix trees [Stoye 02, Adjeroh 03]. These approaches have the limitations of using the suffix trees in terms of space complexity. Our approach depends on the concept of n-grams and does not have the space limitation caused by the large suffix trees.

Trace:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	c	d	a	b	c	d	a	b	c	d	m	a	b	c	d

Execution:

Bi-grams	**Positions**	#	**Execution** (iterations from the Detect Possible Tandem)
ab	0, 4, 8, 13		*key* = ab, *position* = 0 , *i* = 0, *pLength* = 4, s_1= abcd
bc	1, 5, 9, 14	1	$Position_1$ = 0, $position_2$ = 4, $gram_1$ = d, $gram_2$ = d
cd	2, 6, 10, 15		s_2 = abcd → $s_2 == s_1$ → add s_1 and *position* 4 to *TR*
da	3, 7	2	$Position_1$ = 4, $position_2$ = 8, $gram_1$ = d, $gram_2$ = d
dm*	11		s_2 = abcd → $s_2 == s_1$ → add s_1 and *position* 8 to *TR*
ma*	12	3	$Position_1$ = 8, $position_2$ = 13
* will be removed			13 – 8 > 4 → *return* index = 12

Tandem Repeats (TR) Table

Pattern	Positions
Abcd	4, 8

Trace after removing tandem repeats

0	1	2	3	4	5	6	7	8
a	b	c	d	m	a	b	c	d

Figure 4.11. Tandem Repeats Removal Example

4.7 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 p_1, its corresponding patterns are those patterns that have partner events with p_1) until a communication pattern is constructed. When using this algorithm to construct each process patterns (maximal repeats) detected using the pattern detection algorithm presented in Section 4.4, 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 4.5, the output will be the set of all communication patterns that are similar to the given input communication pattern.

The communication pattern construction algorithm is presented in Algorithm 4.7. We introduce the following definitions to help in understanding the algorithm:

1. *CP* (pt_j , po_k): returns the communication pattern cp_m that the process pattern pt_j found at position po_k belongs to. If pt_j does not already belong to a communication pattern, CP will create a new one and return it as cp_m.
2. *PEL*(pt_j , po_k): returns the list of partner events *pel* found in other process traces (events that do not belong to any detected pattern but will be part of a communication pattern).
3. *PPL*(pt_j , po_k): returns the process patterns with which pattern pt_j has partner events.

The algorithm starts iterating on each process at line 1. At line 2, the algorithm iterates on each detected pattern. For each pattern position (line 3), 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 the construction of different communication patterns. At line 4, we retrieve the communication pattern for pattern pt_j at position po_k. If ptj at position po_k is already part of a communication pattern, then it will be returned using *CP*. Otherwise, a new communication pattern will be created and returned by *CP*. We retrieve the partner single events list (*pel*) for pattern pt_j at position po_k at line 5 using *PEL*. We use the partner events list, since an event that is included in a pattern may have a partner event that is not included in any pattern at a partner process. The single partner events will not be detected using the process pattern detection algorithm since we consider the minimum pattern size as two events (bi-gram).

Communication-Patterns-Construction

```
1.  for-each process $pr_i$
2.    for-each pattern $pt_j \in pr_i$-patterns-list
3.      for-each position $po_k \in pt_j$-positions-list
4.        $cp_m = CP(pt_j, po_k)$
5.        $pel = PEL(pt_j, po_k)$
6.        for-each $e \in pel$
7.          if $e$ at $pos(e) \notin cp_m$ then
8.              add $e$ to $cp_m$
9.              add $pos(e)$ to $cp_m$
10.         end-if
11.       end-for-each
12.       $cpl = PPL(pt_j, po_k)$
13.       for-each $p \in cpl$
14.         if $p$ at $pos(p) \notin cp_m$ then
15.             add $p$ to $cp_m$
16.             add $pos(p)$ to $cp_m$
17.         end-if
18.       end-foreach
19.     end-foreach
20.   end-foreach
21. end-foreach
```

Algorithm 4.7. Communication Patterns Construction

We iterate on the *pel* (lines 6-11) where every single event (along with its position in the trace) will be added to the resulting communication pattern with the condition that its process does not have any other partner events that belong to a detected process pattern. At line 12, all the partner process patterns will be retrieved and then added (if they do not already exist) to the communication pattern inside (lines 13-18). After the algorithm finishes iterating on all the process patterns, it will output the distinct communication patterns. The resulting communication patterns may involve all or a subset of the processes in the trace.

We present an example that illustrates the communication pattern detection approach. We first present traces generated from running four processes represented as routine call trees in Figure 4.12. Routines that appear at the leaf level are removed since they do not hold any MPI events. In Figure 4.13, we present the different stages of the communication pattern detection approach. Figure 4.13(b) shows the extracted events that will be entered into the repeating process pattern detection algorithm. We extracted the MPI events (e.g. S2 and R3) and their direct calling methods along with their timestamps. Any routine call events that appear at the same nesting level with the message passing events will be removed from the trace.

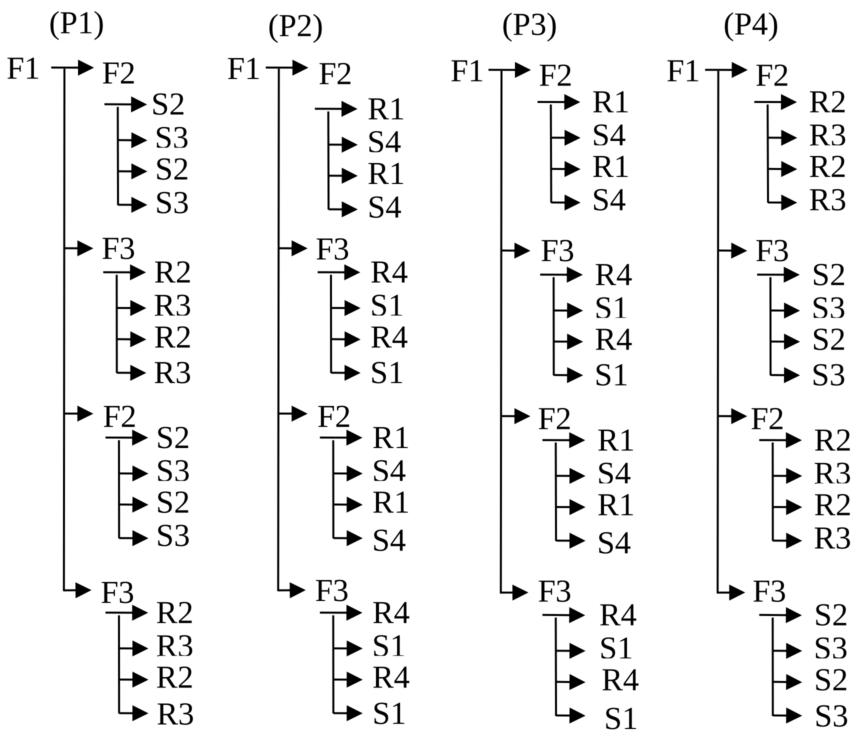


Figure 4.12. Step1: Sample Traces from four Parallel Processes

(a) Process Traces Extracted from Routine Call Trees

P1:	(F2t1) S2S3S2S3 (F3t2) R2R3R2R3 (F2t3) S2S3S2S3 (F3t4) R2R3R2R3
P2:	(F2t1) R1S4R1S4 (F3t2) R4S1R4S1 (F2t3) R1S4R1S4 (F3t4) R4S1R4S1
P3:	(F2t1) R1S4R1S4 (F3t2) R4S1R4S1 (F2t3) R1S4R1S4 (F3t4) R4S1R4S1
P4:	(F2t1) R2R3R2R3 (F3t2) S2S3S2S3 (F2t3) R2R3R2R3 (F3t4) S2S3S2S3

(b) Traces after removing contiguous repeats

	1	2	3	4	5	6	7	8	9	10	11	12
P1	(F2t1)	S2	S3	(F3t2)	R2	R3	(F2t3)	S2	S3	(F3t4)	R2	R3
P2	(F2t1)	R1	S4	(F3t2)	R4	S1	(F2t3)	R1	S4	(F3t4)	R4	S1
P3	(F2t1)	R1	S4	(F3t2)	R4	S1	(F2t3)	R1	S4	(F3t4)	R4	S1
P4	(F2t1)	R2	R3	(F3t2)	S2	S3	(F2t3)	R2	R3	(F3t4)	S2	S3

(c) Detected Process Patterns

P1:PT1 = [S2,S3], PT2 = [R2,R3] **P2**: PT3 = [R1,S1], PT4 = [R4,S1]
P3:PT5 = [R1,S1], PT6 = [R4,S1] **P4**: PT7 = [R2,R3], PT8 = [S2,S3]

(d) Execution

P	Pattern	CP?*	Corresponding Patterns	Communication Pattern (CP)
P1	PT1 at 2	No	PT3 at 2, PT5 at 2	CP1{PT1,PT3,PT5}
P1	PT1 at 8	No	PT3 at 8, PT5 at 8	CP2{PT1,PT3,PT5}
P1	PT2 at 5	No	PT4 at 5, PT6 at 5	CP3{PT2,PT4,PT6}
P1	PT2 at 11	No	PT4 at 11, PT6 at 11	CP4{PT2,PT4,PT6}
P2	PT3 at 2	CP1	PT1 at 2, PT7 at 2	CP1{PT1,PT3,PT5,PT7}
P2	PT3 at 8	CP2	PT1 at 8, PT7 at 8	CP2{PT1,PT3,PT5,PT7}
P2	PT4 at 5	CP3	PT2 at 5, PT8 at 5	CP3{PT2,PT4,PT6, PT8}
P2	PT4 at 11	CP4	PT2 at 11, PT8 at 11	CP4{PT2,PT4,PT6, PT8}
P3	PT5 at 2	CP1	PT1 at 2, PT7 at 2	CP1{PT1,PT3,PT5,PT7}
P3	PT5 at 8	CP2	PT1 at 8, PT7 at 8	CP2{PT1,PT3,PT5,PT7}
P3	PT6 at 5	CP3	PT2 at 5, PT8 at 5	CP3{PT2,PT4,PT6, PT8}
P3	PT6 at 11	CP4	PT2 at 11, PT8 at 11	CP4{PT2,PT4,PT6, PT8}
P4	PT7 at 2	CP1	PT3 at 2, PT5 at 2	CP1{PT1,PT3,PT5,PT7}
P4	PT7 at 8	CP3	PT3 at 8, PT5 at 8	CP2{PT1,PT3,PT5,PT7}
P4	PT8 at 5	CP2	PT4 at 5, PT6 at 5	CP3{PT2,PT4,PT6, PT8}
P4	PT8 at 11	CP4	PT4 at 11, PT6 at 11	CP4{PT2,PT4,PT6, PT8}

* CP?: Does this process pattern already belong to a communication pattern?

(e) Detected Distinct Communication Patterns

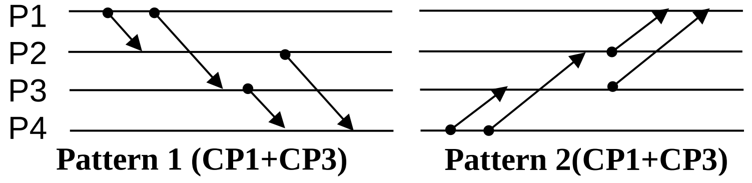


Figure 4.13. Communication Construction Example

In this example, we removed contiguous repeats prior to the detection process as shown in Figure 4.13(b). We consider the first event to be at position 1. We apply the pattern detection algorithm on each process trace and detected the patterns PT1 to PT8.

The detected process patterns are shown in Figure 4.13(c). In Figure 4.13(d), we show execution steps that lead to the construction of the communication patterns.

We start by selecting pattern PT1 from process P1. The algorithm iterates on all the positions where PT1 appears (positions 2 and 8) and locates all the corresponding patterns at the other processes. Four communication patterns are created CP1 to CP4. At the end of the algorithm, only two distinct communication patterns are extracted. The detected patterns correspond to a wavefront pattern. Pattern 1 is a sweep from P1 to P4 and pattern 2 is a sweep from P4 to P1 as shown in Figure 4.13(e).

4.8 Case Studies

In this section, we test our pattern detection and pattern matching approaches on several traces generated from well-known benchmarks and real HPC applications. In Section 4.8.1, we provide two case studies with a comparison with other pattern detection techniques. We will provide the results from applying both the syntactic-directed and the knowledge-directed approaches. In Section 4.8.2, we provide several communication patterns that were detected from traces of different programs.

4.8.1 Repeating Pattern Detection Comparison

In this section, we test our repeating pattern detection approach on two traces generated from Sweep3D and SMG2000. The analysis includes comparison with the syntactic-directed approach.

4.8.1.1 Sweep3D

Sweep3D [Sweep3D] includes the streaming and the scattering operators. The streaming operator is solved by sweeps (wavefront) from each angle to the opposite angle in the grid. The scattering operator is solved iteratively. Sweep3D parallelism is based on the wavefront communication patterns. In case of a 2-dimensional grid, the sweep3D will have four sweeps (wavefront) from each corner to the opposite corner.

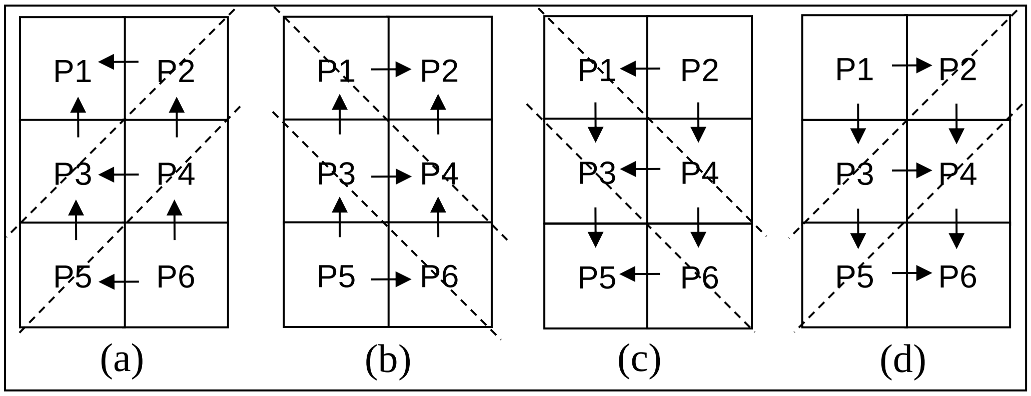


Figure 4.14. Wavefront Pattern (2x3 Process Topology)

Figure 4.14 shows the four sweeps in a 2x3 process topology. Each sweep sends data from a corner to its opposite corner in the grid. In case of a 3-dimensional grid, Sweep3D will consist of eight sweeps (originating from each corner) per iteration.

We tested our approach on six traces generated from running the program using different process topologies and variable number of iterations. In all cases, Sweep3D had the same communication behaviour, i.e., wavefront pattern. The global communication pattern (composition of all wavefront patterns) was repeated the same number of times as the number of iterations (specified as input to the program). Figure 4.15 presents the detected communication wavefront patterns and the global communication pattern composed from the four separate wavefront patterns.

As we can see, the first wavefront is from P6 to P1 followed by the wavefront from P2 to P5. The next wavefront is from P5 to P2 followed by the last wavefront from P1 to P6. The four wavefront patterns compose together a global communication pattern that is repeated 12 times in the trace for a 2x3 process topology and 12 iterations. In each global communication pattern, each single wavefront pattern is repeated 30 times. Without detecting the occurrences of the contiguous repeats it would not be possible to represent the pattern in the compact form shown in Figure 4.15. Hence, it would be large and cluttered and it would require more effort to understand the communication behaviour otherwise. The pattern in Figure 4.15(a) corresponds to the sweep shown in Figure 4.14(a). The pattern in Figure 4.15(b) corresponds to the sweep shown in Figure 4.14(b) and so on for cases Figure 4.15(c) and Figure 4.15(d). This shows that our approach is capable of detecting the valid patterns in a system that uses the wavefront pattern as its communication pattern.

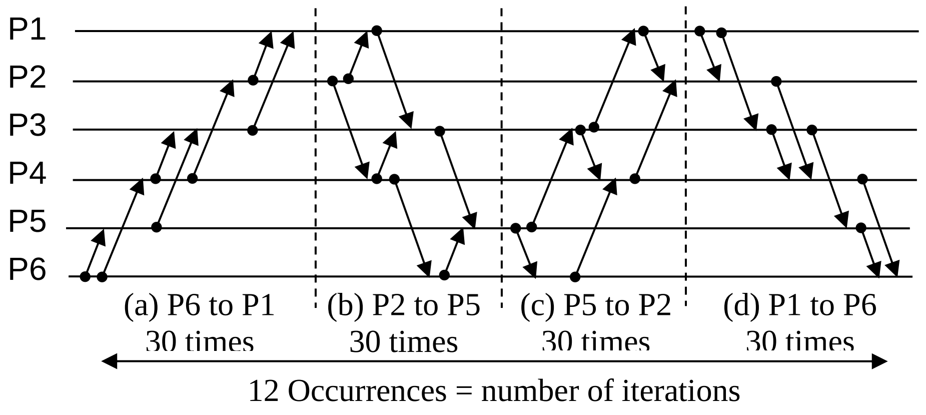


Figure 4.15. Detected Communication Patterns

As indicated in [Preissl 08], the detected communication pattern is large and was not presented in their work. In this work, we present the detected pattern in its compact form which provides the software engineer with a clear understanding of the communication behaviour in the program.

Table 4.1 shows the number of detected patterns when using the syntactic approaches (i.e., the ones that only process messages passed between processes) based on the suffix tree algorithm [Sadakane 07] and our n–gram algorithm. We also show the detected patterns based on routine call tree approach presented in this chapter using the n-gram algorithm.

Table 4.1. Number of Detected Repeats for P1 for Sweep3D (2x3 process topology and 12 iterations) System (Relevant Patterns = 5)

Pattern Detection Technique	**P**	**FP**	**TN**	**Precision**	**Recall**
Syntactic Matching based on Suffix Tree Method	133	129	1	3%	80%
Syntactic Matching based N-Gram Method	20	15	0	40%	100%
Routine call-directed Approach Based on the N-Gram Method	5	0	0	100%	100%

The number of repeats detected using the suffix tree approach for process P1 is 133 which is very large for such a small trace of P1 (2880 messages). Moreover, 129 of the detected patterns are not valid patterns. Furthermore, the approach missed one valid pattern. In the n-gram approach without the routine call tree, the number of detected repeats is 20 with no false positives and no true negatives. The other processes have the same number of repeats for both methods and the same number of false positives and false negatives. This is due to the nature of the Sweep3D which is repetitive and only follows several wavefront communication behaviours. When applying the detection with the support of the routine call trees for each process, we detected five patterns, which reflect the valid patterns of the Sweep3D application. The table also shows that our algorithm for this trace has precision and recall values that are 100%. We were able to calculate the precision and recall for this system as we already know the communication behaviour in Sweep3D. Therefore, this can be used as a validation for our communication pattern detection approach.

We also compared the results in terms of performance (see Table 4.2). We used different process topologies to vary the number of processes. CST stands for stands for Compressed Suffix Tree [Sadakane 07] which is an algorithm used to detect communication patterns by processing message passing events. As we can see, the routine call trees based technique performs better than CST, thanks to our previous n-gram algorithm. In other words, a call tree based approaches not only improves effectiveness (i.e. quality of the patterns) but it is also efficient if combined with an efficient extraction algorithm.

Table 4.2. Performance Analysis for Sweep3D Traces

Process Topology	**It.**	**Messages**	**Routine call with n-gram**	**n-gram (s)**	**CST (s)**
2 x 3	12	20160	0.78	0.72	5.23
6 x 3	12	51840	2.45	1.674	5.700
5 x 5	40	256000	5.83	4.20	43.00
7 x 4	74	532800	9.156	7.20	40.40
8 x 8	120	2150400	28.74	22.27	285.64
8 x 16	120	4454400	56.45	52.23	480.83

4.8.1.2 SMG2000

SMG2000 [SMG2000] uses a complex communication pattern [Geimer 06]. The parallelism is achieved by data decomposition. SMG2000 performs a large number of non-nearest-neighbor point-to-point communication operations and can be considered a stress test for the network subsystems of a machine [Wolf 08]. We tested our pattern detection approach on several traces generated from different scenarios by varying the number of processes and the problem size.

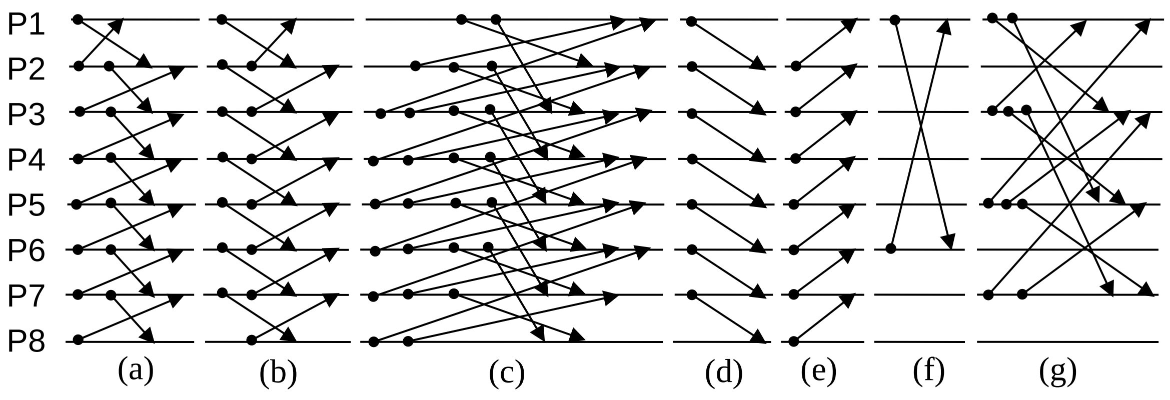


Figure 4.16. SMG2000 Communication Patterns (Topology: 8x1x1 Problem Size: 2x2x2)

Figure 4.16 shows the seven detected repeating patterns for the 8x1x1 process topology and a 2x2x2 problem size. Figure 4.16(a) (61 occurrences) and Figure 4.16(b) (17 occurrences) correspond to the nearest-neighbor communication pattern where every process is only communicating with its direct neighbors. It can be noticed that Figure 4.16(a) and Figure 4.16(b) correspond to the same pattern with a difference in the events ordering. Figure 4.16(c) (18 occurrences) is a more complex case where the processes communicate with non-direct neighbor processes, which corresponds to the specifications of SMG2000. For Figure 4.16(d) (137 occurrences) and Figure 4.16(e) (76 occurrences), it can be noticed that for P1 and P8 there is only one event in each case which we do not detect using our n-gram approach as we consider a bi-gram as the smallest pattern. However, the communication pattern construction algorithm adds these single partner events to the communication pattern as previously described in the algorithm. The pattern in Figure 4.16(d) moves the data from P1 to P8 while the pattern in Figure 4.16(e) moves the data in the opposite direction. The pattern in Figure 4.16(f) (36 occurrences) shows that there is a repeating communication pattern between processes P1 and P5 only. The pattern in Figure 4.16(g) (18 occurrences) shows that the communication pattern involves only processes P1, P3, P5 and P7. In [Preissl 08], only the pattern in Figure 4.16(a) was presented as the main communication pattern. In our work, we present all the detected patterns that correspond to the same scenario.

Table 4.3 shows the resulting patterns when applying the various techniques to the trace of process P2 for SMG2000 with 8x1x1 process topology and 2x2x2 problem size. When applying the n-gram approach directly on the trace generated from P2 (1028 message passing events) without considering the routine call tree, the number of detected process patterns was 25. The suffix tree approach resulted in 52 patterns. However, the number of detected patterns when applying the algorithm using the routine call tree was reduced to 10. When removing contiguous repeats from the trace, the number of patterns with the routine-call directed approach was reduced to 5 which is the exact number of patterns).

Table 4.3. Detected Repeats for P2 for SMG2000 (8x1x1 Process Topology and 2x2x2 Problem Size, Relevant Patterns = 5, P: Patterns)

Pattern Detection Technique	**P**	**FP**	**TN**	**Precision**	**Recall**
Syntactic Matching based on Suffix Tree Method	52	48	0	10%	100%
Syntactic Matching based N-Gram Method	25	20	0	20%	100%
Routine call-directed Approach Based on the N-Gram Method	10	5	0	50%	100%

In another scenario where we used a 2x2x2 process topology (3D mesh) and 2x2x2 problem size another set of patterns was detected. However, without the knowledge of the routine calls, some patterns were not detected using any of the pure string matching techniques. Moreover, the number of detected patterns for P1 when using the suffix tree on the trace of message passing events was 208 which is considerably high compared to the true number of patterns which is 25.

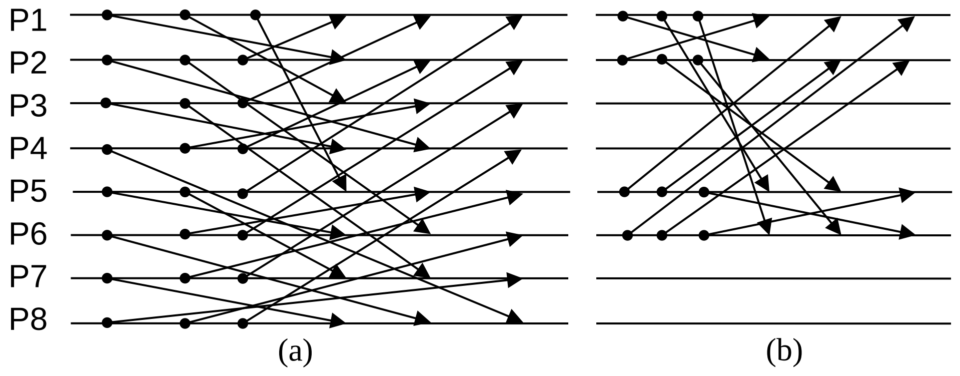


Figure 4.17. SMG2000 Detected Patterns using Routine-Call Directed Approach

Figure 4.17 shows an example of two communication patterns that were not detected when applying the pattern detection approach directly to the message passing events. The Pattern in Figure 4.17(a) involves all the processes in the trace and depicts a nearest neighbour communication pattern. For example, P1 communicates with P2, P3 and P5 which are its neighbours. However, the pattern in Figure 4.17(b) involves only four processes from the trace and shows that all processes communicate with each other.

Table 4.4. Performance Analysis for SMG2000 Traces

Topology	**Problem Size**	**MPI Events**	**Routine call tree n-gram (s)**	**n-gram (s)**	**CST (s)**
8x1x1	2x2x2	9312	1.25	0.98	3.33
2x2x2	2x2x2	25416	1.33	1.40	17.13
4x4x2	2x2x2	248768	12.56	10.82	70.96
16x1x1	10x10x10	978296	73.98	68.71	387.07
32x1x1	10x10x10	2363156	162.14	147.65	804.12
64x1x1	10x10x10	5324304	359.54	354.32	1204.90

This example shows that using the routine call tree in the detection process helps in improving the quality of the detection process by uncovering patterns that cannot be detected directly from a trace of message passing events. Similar to the Sweep3D example, our approach performs also better in terms of execution time than a pure suffix tree based approach as can be seen in Table 4.4.

It is clear that the two patterns in Figure 4.17 when represented by the event graph are not very easy to follow due to the irregular order of events as opposed to the patterns in Figure 4.16. This opens another research question on how to find a better visualization technique than event graphs to represent communication patterns.

4.8.2 Sample of Detected Patterns on Target Systems

In this section, we present sample detected patterns applied to different systems using our n-gram based techniques.

4.8.2.1 NAS Parallel Benchmark

In this section, we target three programs (LU, CG, and MG) that are part of the NAS Parallel Benchmark suite (described in Section 3.8.1). We briefly describe each target program along with the detected communication patterns.

4.8.2.1.1 NAS-LU

LU is similar to Sweep3D in that it uses diagonal pipelining method (wavefront) method, to perform communication of partition boundaries. An iteration in LU consists of two sweeps [Mudalige 08], one sweep starting from the top-left corner to the bottom-right corner in the process topology followed by a sweep in the opposite direction. In the following, we test our approach on a trace generated from LU in order to verify if the communication pattern used in LU corresponds to a wavefront pattern. The tested trace is generated from running 8 processes with a 2x4 process topology, a problem size of 64 x

64 x 64, and 250 iterations. When applying the pattern detection algorithm on the trace (without considering the routine call graph) we were able to detect the global communication behaviour in the trace. It should be noted that according to the NPB documentation found at [NAS-Changes] a dummy iteration was added before the time step loop in LU for consistency with the other benchmarks in NPB. This justifies the number of occurrences of the global communication pattern to be 251 rather than 250. The global communication pattern is depicted in Figure 4.18. When applying the suffix tree approach on the LU trace, we were not able to detect the global communication behaviour. In Figure 4.18, it is noticed that the two sweeps are preceded by a 2D-nearest neighbor pattern in all the occurrences. This pattern is used to perform some computations prior to the sweeps. The suffix tree approach was only able to detect each repeating pattern separately. However, our approach is able to detect each repeating pattern separately as well as the global communication pattern shown in Figure 4.18. Moreover, the number of detected repeats using the suffix tree approach for P1 (for example) is 383 repeats where in our approach it was only 6 repeats. The problem with suffix tree is that it detects a large number of repeats which most of them are considered as false positives. Also, using our algorithm, there are two cases of the communication pattern that are preceded with two occurrences of the 2D-nearest neighbour pattern and there is one case that the two sweeps were followed by one occurrence of the 2D nearest neighbour pattern.

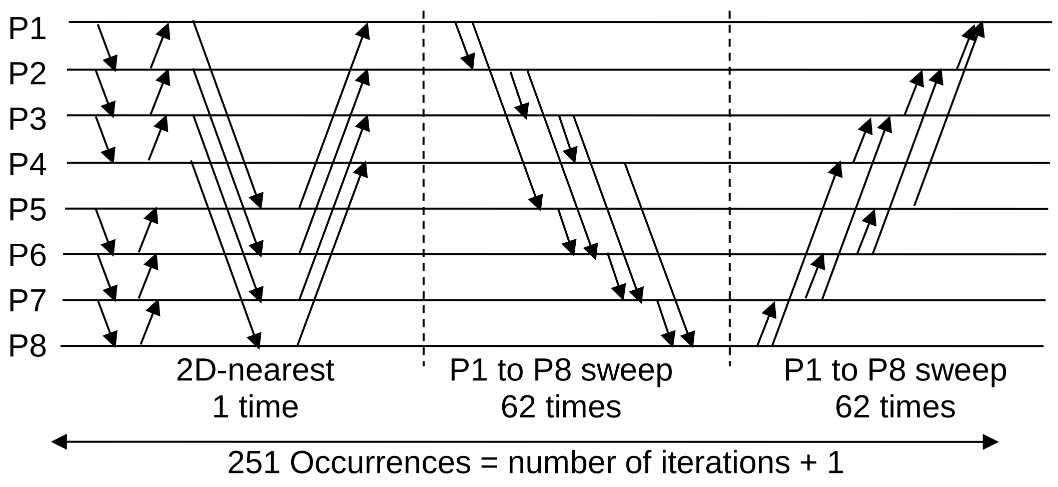


Figure 4.18. LU Global Communication Pattern

When using the information from the routine call graph, we were able to detect the patterns shown in Table 4.5. The numbers of occurrences for each pattern are consistent with the number presented in Figure 4.18.

Table 4.5. Patterns Detected with Routine Call Graph

Wavefront Pattern	**Occurrences**
Sweep from P1 to P8	15562 = 251 x 62
Sweep from P8 to P1	15562 = 251 x 62
2D-nearest neighbor	254 = 251 + 3

We apply the pattern matching technique to the LU program since we already know that the communication pattern used in the program is a wavefront pattern. The pattern matching algorithm differs than the pattern detection one since we need to provide an input pattern and then look for it in the trace. Therefore, the pattern needs to be entered properly and should match the number of processes and their topology. We used the same example (2x4 process topology). The first step is to generate the pattern events for each process separately. Then, we can match each process patterns separately and construct the communication pattern. The process topology is the most important factor when building the input pattern since it determines the partner processes that each process will communicate with. Moreover, it will determine the originating process in the wavefront. When considering P1 as the originating process the input pattern for each process should look like the events shown in Table 4.6. The pattern matching algorithm will use the events for each process to look for the matching process patterns in the trace. After detecting all the occurrences of each pattern in the trace, the communication construction algorithm will start by matching the partner events based on their positions in the trace.

Table 4.6 Input Pattern for Wavefront originating from P1

Process	**Input Pattern**
P1	Send to P2, Send to P5
P2	Receive from P1, Send to P3, Send to P6
P3	Receive from P2, Send to P4, Send to P7
P4	Receive from P3, Send to P8
P5	Receive from P1, Send to P6
P6	Receive from P5, Receive from P2, Sent to P7
P7	Receive from P6, Receive from P3, Send to P8
P8	Receive from P7, Receive from P4

In this example, we set the error value to be 0 which means that we are looking for exact matches to the input pattern. However, the error value can be set to another value when we are looking for similar patterns to the input one. It should be noted that the ordering of events may be different, for example P1 may send to P5 before sending to P2. These different combinations can be handled since our algorithm uses the *edit distance* function. After running the pattern matching algorithm the number of detected wavefront patterns that originate from P1 and end at P8 was 15562. This validates the pattern matching algorithm since we already know the number of wavefronts originating from P1. Moreover, it validates that the repeating pattern detection algorithm is correct since also the number of detected patterns was verified using the pattern matching algorithm.

4.8.2.1.2 NAS-CG

This kernel is useful for unstructured grid computations in order to test irregular long distance communication that employs unstructured matrix vector multiplication. We tested our algorithm on the NAS CG (class W) benchmark.

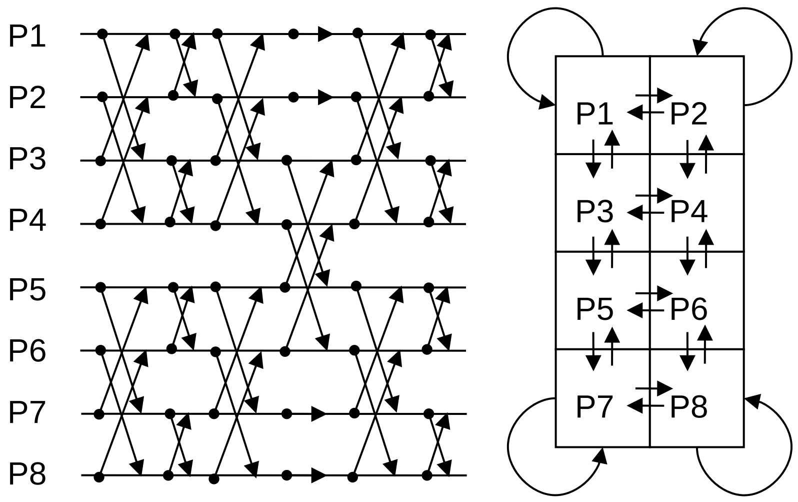


Figure 4.19. NAS CG Pattern and Topology

Figure 4.19 shows the communication pattern (left) and its corresponding MPI virtual topology. The main communication behaviour in CG follows a 2D-stencil which was detected by our algorithm as shown in Figure 4.19.

As can be seen from the pattern, processes (P1 to P4) form a sub-group and processes (P5 to P8) form another sub-group of communication. Data is being exchanged between the two sub-groups through processes (P3 to P6) at the center of the communication pattern. Moreover, to validate the detection algorithm we compared the communication topology to the one presented in [Cappello 00] and found that they are identical. In [Lee 09], the authors presented some analysis regarding the two-subgroups and how they communicate. In our work, we were able to provide the full communication pattern described using the event graph.

4.8.2.1.3 NAS-MG

The MG kernel follows only one communication behaviour. The process topology (Figure 4.20a) corresponds to a 3D mesh (2x2x4). Figure 4.20b shows the detected communication pattern for an instance of NAS MG (class A) running on 16 processes. The total number of messages in the trace is 22048. The communication pattern shows that processes communicate to the nearest neighbour on their layer and the adjacent layer. Also, processes on the side layers communicate with each other. For example, P1 communicates with P13 and P4 communicates with P16. The NAS MG is used to test near and far communications. This can be easily noticed in the detected patterns. For example, P1 communicates with the far process P13 and also it communicates with its near neighbors (P1, P3 and P5). The communication pattern is detected 109 times in the trace.

When using the knowledge-directed approach, all the communications occurred within the *comm3* routine. According to [Lu 04] they mentioned that every process when executing *comm3* it sends 6 messages and receives 6 messages exchanged in the three dimensions coordinate. Our dynamic analysis approach proves these results and extends it by representing the communication pattern using an event graph.

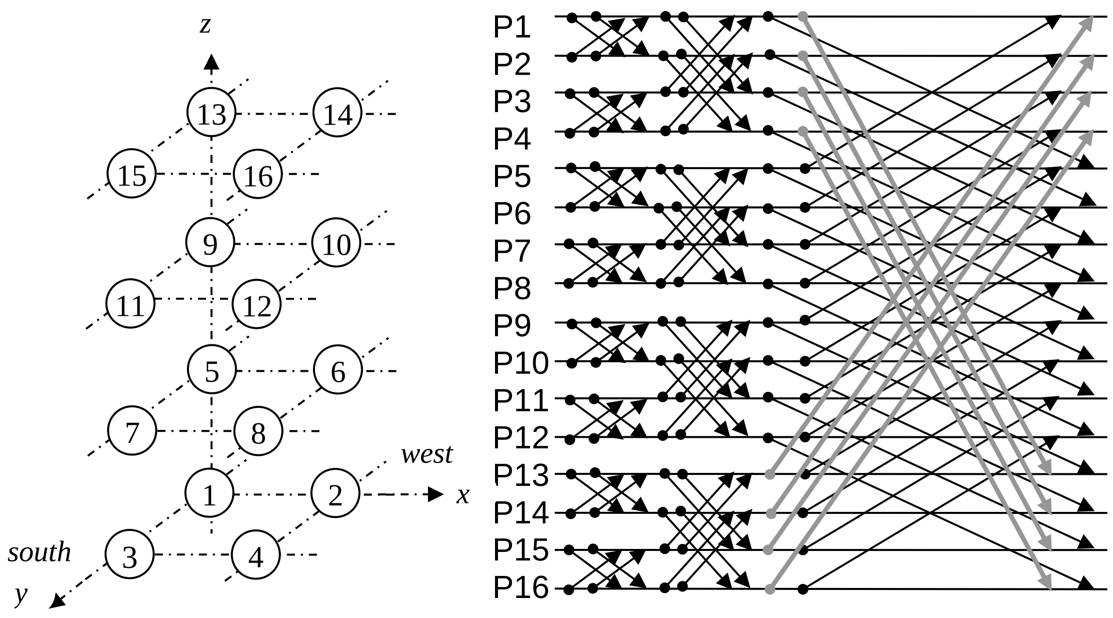


Figure 4.20. NAS MG Class A Communication Topology & Pattern

4.8.2.2 Weather Forecasting & Research Model

We tested our pattern detection algorithm on a trace file generated by the VampirTrace [VampirTrace] trace analysis tool (this trace is different than the one presented in the previous section but was generated from the same system). The trace file had 336960 point-to-point events. 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

Processes in WRF communicate based on a 2D nearest-neighbor topology. In the following, we tested our pattern detection approach on different traces generated from WRF. The results show that the communication pattern follows the same communication structure (2D nearest-neighbor or 2D-Stencil) as indicated in the program's documentation. For the first trace, the detected pattern occurred 535 times.

When using the suffix tree approach the number of resulting repeats was 534. However, when applying our n-gram algorithm there was only one pattern detected which is the only true pattern. In the suffix tree approach, it returned overlapping repeats. For example, when considering only this part of the whole trace for P1 ‘abcdabcdabcdabcdabcdabcdabcdabcdabcd’ (where *a*: Send to 5, *b*: Receive from 5, *c*: Send to 2, and *d*: Receive from 2) then the resulting patterns are shown in Table 4.7.

Table 4.7. Suffix Tree Detection Example of WRF Sample Trace

	Pattern	**Occurrences**	**Positions**
1	abcd	9	0, 4, 8, 12, 16, 20, 24, 28, 32
2	abcdabcd	8	0, 4, 8, 12, 16, 20, 24, 28
3	abcdabcdabcd	7	0, 4, 8, 12, 16, 20, 24
4	abcdabcdabcdabcd	6	0, 4, 8, 12, 16, 20
5	abcdabcdabcdabcdabcd	5	0, 4, 8, 12, 16
6	abcdabcdabcdabcdabcdabcd	4	0, 4, 8, 12
7	abcdabcdabcdabcdabcdabcdabcd	3	0, 4, 8
8	abcdabcdabcdabcdabcdabcdabcdabcd	2	0, 4

It clearly shows that in the suffix tree approach the number of detected repeats is quite high with respect to the true number of patterns in the trace. When applying the n-gram approach, only the first pattern ‘abcd’ was detected which makes the communication pattern construction algorithm much easier than when considering all the other detected patterns.

Figure 4.21 presents the communication pattern (right) and its corresponding communication topology which clearly shows a 2D stencil communication behaviour. When applying the pattern detection algorithm on a larger trace of the WRF application the same pattern was detected with 3510 occurrences.

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.

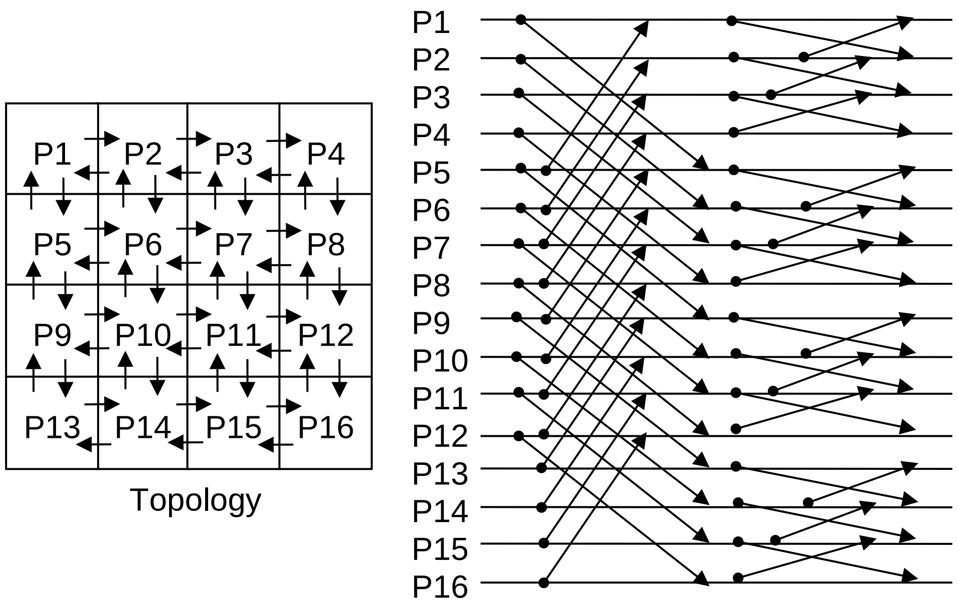


Figure 4.21. WRF Communication Pattern

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 and the diagnostic quantities pressure.

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 4.22. Detected Collective Pattern

The execution trace contained two collective patterns (patterns from MPI collective operations) as shown in Figure 4.22. 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.

4.8.2.3 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 [FLASH 2]. Flash solves complex systems of equations for hydrodynamics and nuclear burning which uses Paramesh library [Paramesh] 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. 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 4.23 shows two patterns that were detected using the pattern detection algorithm. The pattern in Figure 4.23a is repeated 927 times and pattern in Figure 4.23b 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 work 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 in Figure 4.23b, 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.

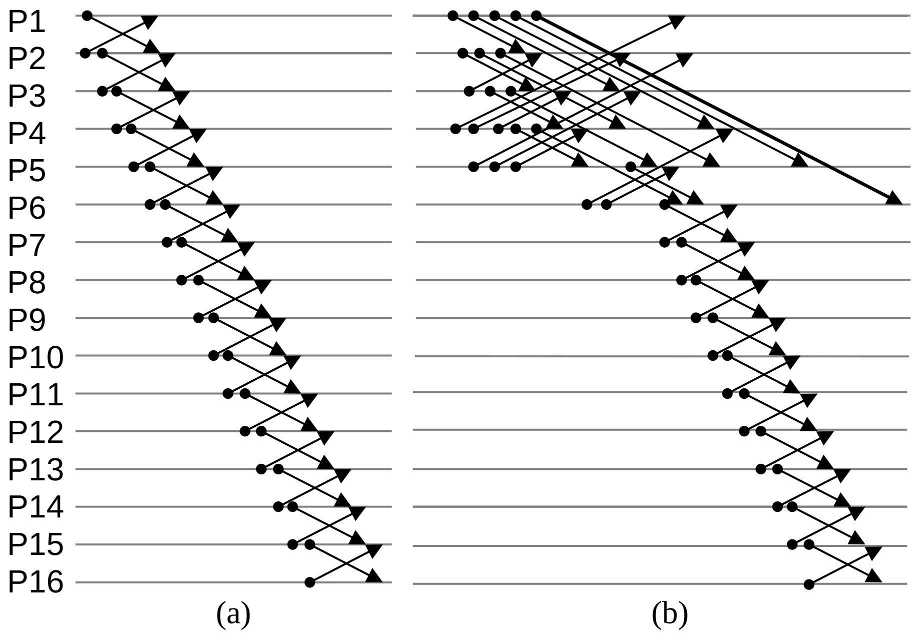


Figure 4.23. Two Detected Patterns in the 2D Cellular Problem

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.

4.9 Summary

In this chapter, we presented a new approach for detecting repeating communication patterns and matching similar communication 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. We presented several algorithms that we have developed to guide in the detection process. The presented algorithms are:

1. The detection of maximal repeats in a process trace. This algorithm extracts all the repeating sequences of MPI events in each process trace separately.
2. The detection and removal of tandem repeats in a process trace. This algorithm removes all contiguous repeats from the trace which reduces the size of the trace significantly.
3. An algorithm for finding similar patterns based on a predefined input pattern in the MPI trace. This algorithm runs on each process trace separately and finds the sequences on each process trace that match the input pattern.
4. The construction of communication patterns based on the detected process patterns gathered in 1 & 3.

We elaborated on the steps in each algorithm in a separate section and provided a running example that illustrates the algorithm.

We have shown how our approach for detecting repeating communication patterns in the trace utilizes the knowledge in the trace as opposed to the existing approaches that are syntactic where they only consider the MPI trace as a mere string of message passing events. The results showed that a knowledge-directed approach improves the quality of detection patterns in terms of reducing the numbers of false positives and true negatives respectively.

Chapter 5. Execution Phases in MPI Traces

5.1 Introduction

Programs are designed to have several execution phases where each phase is meant to represent a specific behaviour in the program such as its initialization, computations, and outputting the results. A phase can also be comprised of several sub-phases. Locating the phases in the execution trace can be utilized for different purposes such as program comprehension, reducing simulation time, system reconfiguration and adaptive optimizations [Gu 06].

In this thesis, we propose a novel approach for localizing computational phases in 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 presented phase detection approach 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 Chapter 4. The second step, which is also the main contribution of this chapter, 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 [Gray 11] and the Jensen-Shannon Divergence measure [Grosse 02]. The description and explanation of the phase detection approach along with two case studies from well-known HPC programs and benchmarks are presented in the following sections.

5.2 Phase Detection Approach

Figure 5.1 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 Chapter 4. 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. 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. In the following section, we discuss our phase detection approach in more detail.

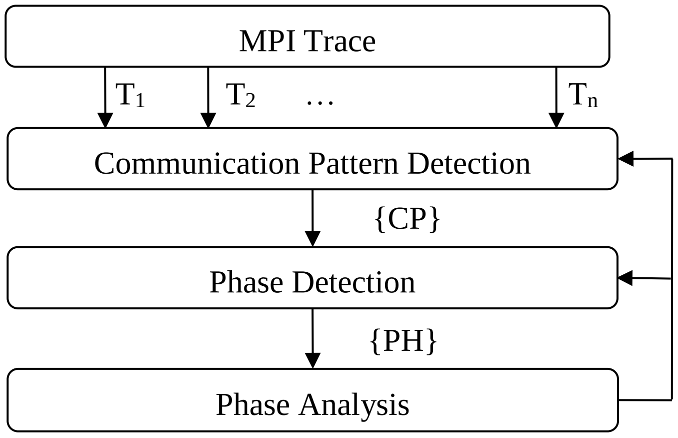


Figure 5.1. Phase Detection Approach

5.2.1 Phase Detection

Our phase detection approach is inspired by studies in the field of bioinformatics, more particularly, the analysis of DNA sequences. In [Li 02], 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 [Cormen 90], which relies on information theory concepts. More precisely, the algorithm uses Shannon entropy [Shannon 48, Gray 11] and the Jensen-Shannon divergence measures [Grosse 02] 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 [Gray 11]. 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 [Gray 11].

$$H = -\sum_{j=1}^{k} \frac{N_j}{N} \log \frac{N_j}{N} \qquad (5.1)$$

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:

1. 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:

$$H_l = -\sum_{j=1}^{k} \frac{N_j^l}{i} log \frac{N_j^l}{i} \qquad (5.2)$$

$$H_r = -\sum_{j=1}^{k} \frac{N_j^r}{N-i} log \frac{N_j^r}{N-i} \qquad (5.3)$$

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.

2. For each two subsequences, we measure their similarity by comparing the entropy values using the Jensen-Shannon Divergence (D_{JS}) measure [Grosse 02] and which is presented below. The higher D_{JS}, the more heterogeneous the subsequences are:

$$D_{JS} = H - \frac{i}{N} H_l - \frac{N-i}{N} H_r \quad (5.4)$$

3. 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 [Li 02] 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. [Li 02] proposed to use the Bayesian Information Criterion (*BIC*) [Akaike 78] 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:

$$BIC = -2\,log(L) + log(N)K \quad (5.5)$$

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:

$$L1(S) = \prod_{j=1}^{k} p_j^{N_j} \qquad (5.6)$$

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

$$\log L1(S) = \sum_{j=1}^{k} N_j \log \frac{N_j}{N} \qquad (5.7)$$

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:

$$L2(S_l, S_r, N_l) = \prod_{j=1}^{k_l} (p_j^l)^{N_j^l} \prod_{j=1}^{k_r} (p_j^r)^{N_j^r} \qquad (5.8)$$

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:

$$\log L2(S_l, S_r, N_l) = \sum_{j=1}^{k_l} N_j^l \log \frac{N_j^l}{N} + \sum_{j=1}^{k_r} N_j^r \log \frac{N_j^r}{N} \qquad (5.9)$$

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 5.4).

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 log(*L)* by $N\hat{D}_{JS}$ in Equation 5.5, it will lead to the following:

$$2N\hat{D}_{JS} > \log(N)K \quad (5.10)$$

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:

$$\tau = \log(N)K/2N \quad (5.11)$$

$\hat{D}_{JS}$ should be greater than τ in order for segmentation to continue. Li et al [Li 02] 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:

$$s = \frac{2N\hat{D}_{JS} - log(N)K}{log(N)K} \quad (5.12)$$

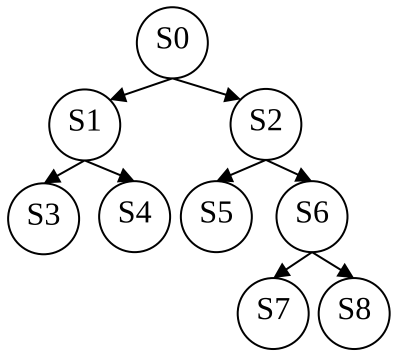
Segmenting the sequence based on Equation 5.12 when s > 0 will have the same effect as segmenting the sequence when D_{JS} is greater than the dynamic threshold calculated based on Equation 5.10. In other words, the segmentation strength must always be positive value in order to continue the segmentation process. Moreover, the value of s can be adjusted to be greater than a user-specified value s_0 where $s > s_0 > 0$. Varying s_0 will vary the numbers of detected subsequences. A larger s threshold value s_0 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 D_{JS} for left and right subsequences.

The graph in Figure 5.2c clearly shows the borders of each segment in the sequence. It shows that at points 3, 9, 17 and 21 there are peak divergence values. The algorithm will select the highest divergence value (0.97 at position 9). Then, it will run the same algorithm for the left and the right subsequences for further segmentation. Figure 2b shows the segmentation tree and how each sequence is further segmented into left and right segments. The table presented in Figure 5.2a shows the values that correspond to each subsequence during the segmentation process. Also, the tree that corresponds to the subsequences is shown in Figure 5.2b. This example demonstrates usefulness of the Shannon entropy and the Jensen-Shannon divergence in sequence segmentation.

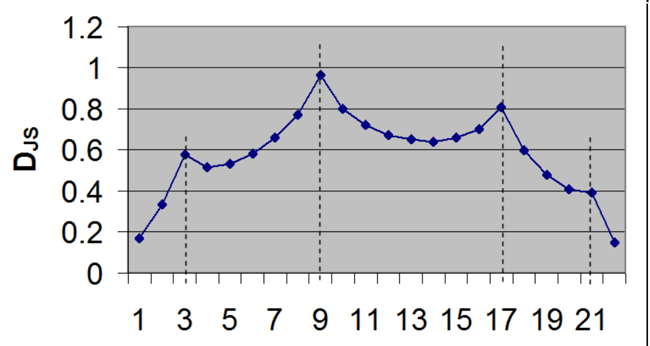
S	AAABBBBBBCCCCCCCCDDDDEE				
	p_s	p_e	D_{js}	p_c	s
S_0	1	23	0.97	9	8.81
S_1	1	9	0.79	3	3.49
S_2	10	23	0.96	17	6.06
S_3	1	3	-0.17	-	-1.0
S_4	4	9	-0.08	-	-1.19
S_5	10	17	0.0	-	-1.0
S_6	18	23	0.79	21	2.66
S_7	18	21	-0.12	-	-1.0
S_8	22	23	-0.25	-	-1.5

(a) Segmentation Data



(b) Segmentation Tree

p	D_{JS}	p	D_{JS}
1	0.168	12	0.671
2	0.332	13	0.65
3	0.577	14	0.64
4	0.515	15	0.66
5	0.533	16	0.7
6	0.583	17	0.81
7	0.661	18	0.6
8	0.772	19	0.48
9	0.965	20	0.41
10	0.8	21	0.39
11	0.72	22	0.15



(c) D_{JS} values for each element in S_0

Figure 5.2. Heterogeneous Sequence Segmentation Example (p_s: start position, p_e: end position, H: Shannon Entropy, $\hat{D}_{JS}$: Jensen-Shannon Divergence, p_c: cutting point, τ: threshold, and *s*: segmentation strength)

5.2.2 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.

5.3 Case Study

In this section, we show the effectiveness of our approach by applying it to two large traces generated from the NAS BT benchmark and the SMG2000 industrial HPC system.

5.3.1 SMG2000

In this section, we show the effectiveness of our approach by applying it to a large trace generated from the SMG2000 (described in Section 4.8.1) industrial HPC system [SMG2000]. SMG2000 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 [Geimer 06].

At a high-level, SMG2000 performs three distinct phases to solve the problem as reported in [Tiwari 11]. 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. We used the

VampirTrace [VampirTrace] tracing tool to generate the traces from running SMG2000. The execution scenario is based on a 4x4x2 process topology (Figure 5.3) and a 2x2x2 input problem size.

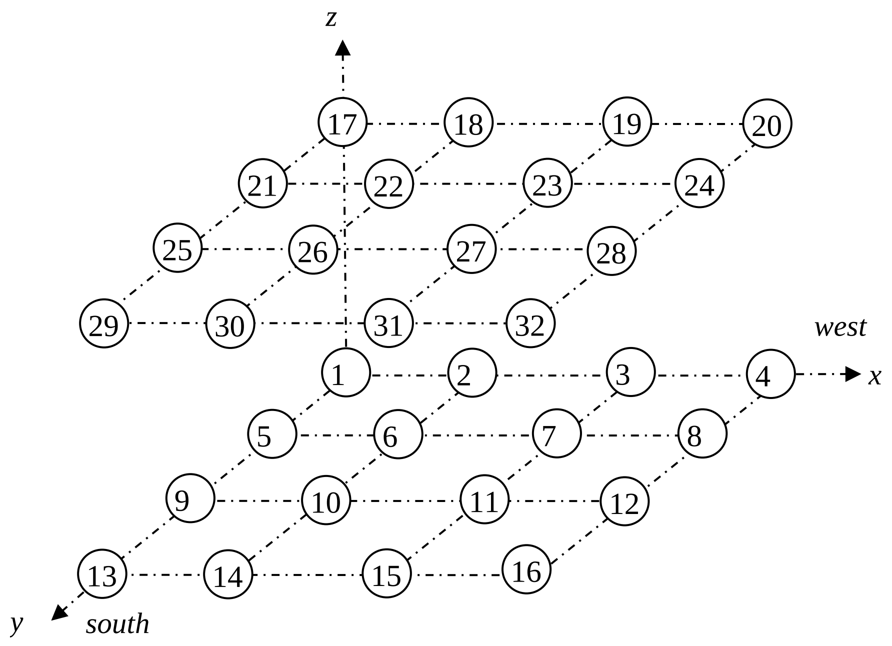


Figure 5.3. Process Topology for SMG2000 4x4x2

Table 5.1 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 5.1 shows that this is relatively a large trace with more than 15 Million events.

Table 5.1. SMG2000 Statistics for SMG2000 Trace

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

5.3.1.1 Pattern Detection

We used our pattern detection algorithm described in Chapter 4 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. We present the list of all point-to-point communication patterns in Appendix B.

5.3.1.2 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 5.4 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 5.4 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.

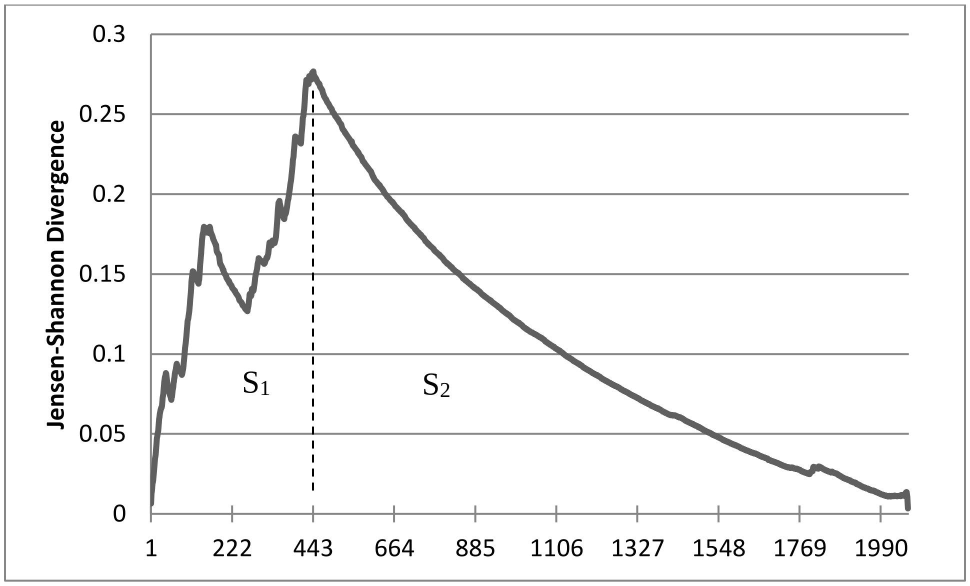


Figure 5.4. D_{JS} values for the whole sequence (max D_{JS} at 443, $\tau = 0.06$)

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 67 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.

Table 5.2. Recursive Segmentation (p_s: start position, p_e: end position, l: length, D_{JS}: Jensen-Shannon Divergence, p_c: cutting position of max divergence, τ: threshold, s: Segmentation Strength, and P: parent node, hyphen (-) means no s for length = 1)

	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}	24	42	19	0.42	37	0.34	0.25	S_{6}
S_{18}	43	145	103	0.26	87	0.52	-0.51	S_{6}
S_{19}	24	37	14	0.45	29	0.41	0.11	S_{17}
S_{20}	38	42	5	0.92	39	0.23	2.97	S_{17}
S_{21}	24	29	6	0.88	25	0.22	3.08	S_{19}
S_{22}	30	37	8	0.97	33	0.19	4.18	S_{19}
S_{23}	24	25	2	-0.25	24	0.5	-1.5	S_{21}
S_{24}	26	29	4	0.81	27	0.25	2.22	S_{21}
S_{25}	26	27	2	-0.25	26	0.5	-1.5	S_{24}
S_{26}	28	29	2	-0.25	28	0.5	-1.5	S_{24}
S_{27}	30	33	4	0.81	31	0.25	2.22	S_{22}
S_{28}	34	37	4	0.81	35	0.25	2.22	S_{22}
S_{29}	30	31	2	-0.25	30	0.5	-1.5	S_{27}
S_{30}	32	33	2	-0.25	32	0.5	-1.5	S_{27}
S_{31}	34	35	2	-0.25	34	0.5	-1.5	S_{28}
S_{32}	36	37	2	-0.25	36	0.5	-1.5	S_{28}
S_{33}	38	39	2	-0.25	38	0.5	-1.5	S_{20}
S_{34}	40	42	3	0.75	41	0.26	1.85	S_{20}
S_{35}	40	41	2	-0.25	40	0.5	-1.5	S_{34}
S_{36}	42	42	1	0	41	0	-	S_{34}
S_{37}	146	264	119	0.19	162	0.2	-0.06	S_{4}
S_{38}	265	443	179	0.28	294	0.15	0.95	S_{4}
S_{39}	265	294	30	0.76	276	0.16	3.63	S_{38}
S_{40}	295	443	149	0.33	365	0.48	-0.32	S_{38}
S_{41}	265	276	12	0.43	270	0.3	0.44	S_{39}
S_{42}	277	294	18	0.36	280	0.35	0.05	S_{39}
S_{43}	265	270	6	-0.08	269	0.43	-1.19	S_{41}
S_{44}	271	276	6	0.79	274	0.22	2.66	S_{41}
S_{45}	271	274	4	-0.12	273	0.5	-1.25	S_{44}
S_{46}	275	276	2	-0.25	275	0.5	-1.5	S_{44}
S_{47}	277	280	4	0.81	278	0.25	2.22	S_{42}
S_{48}	281	294	14	0.45	286	0.41	0.11	S_{42}
S_{49}	277	278	2	-0.25	277	0.5	-1.5	S_{47}
S_{50}	279	280	2	-0.25	279	0.5	-1.5	S_{47}
S_{51}	281	286	6	0.88	282	0.22	3.08	S_{48}
S_{52}	287	294	8	0.97	290	0.19	4.18	S_{48}
S_{53}	281	282	2	-0.25	281	0.5	-1.5	S_{51}
S_{54}	283	286	4	0.81	284	0.25	2.22	S_{51}
S_{55}	283	284	2	-0.25	283	0.5	-1.5	S_{54}
S_{56}	285	286	2	-0.25	285	0.5	-1.5	S_{54}
S_{57}	287	290	4	0.81	288	0.25	2.22	S_{52}
S_{58}	291	294	4	0.81	292	0.25	2.22	S_{52}
S_{59}	287	288	2	-0.25	287	0.5	-1.5	S_{57}
S_{60}	289	290	2	-0.25	289	0.5	-1.5	S_{57}
S_{61}	291	292	2	-0.25	291	0.5	-1.5	S_{58}
S_{62}	293	294	2	-0.25	293	0.5	-1.5	S_{58}
S_{63}	444	2061	1618	0.01	1821	0.06	-0.76	S_{2}
S_{64}	2062	2065	4	0.58	2062	0.25	1.31	S_{2}
S_{65}	2062	2062	1	0	2061	0	-	S_{64}
S_{66}	2063	2065	3	-0.17	2064	0.53	-1.32	S_{64}

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 5.2 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.

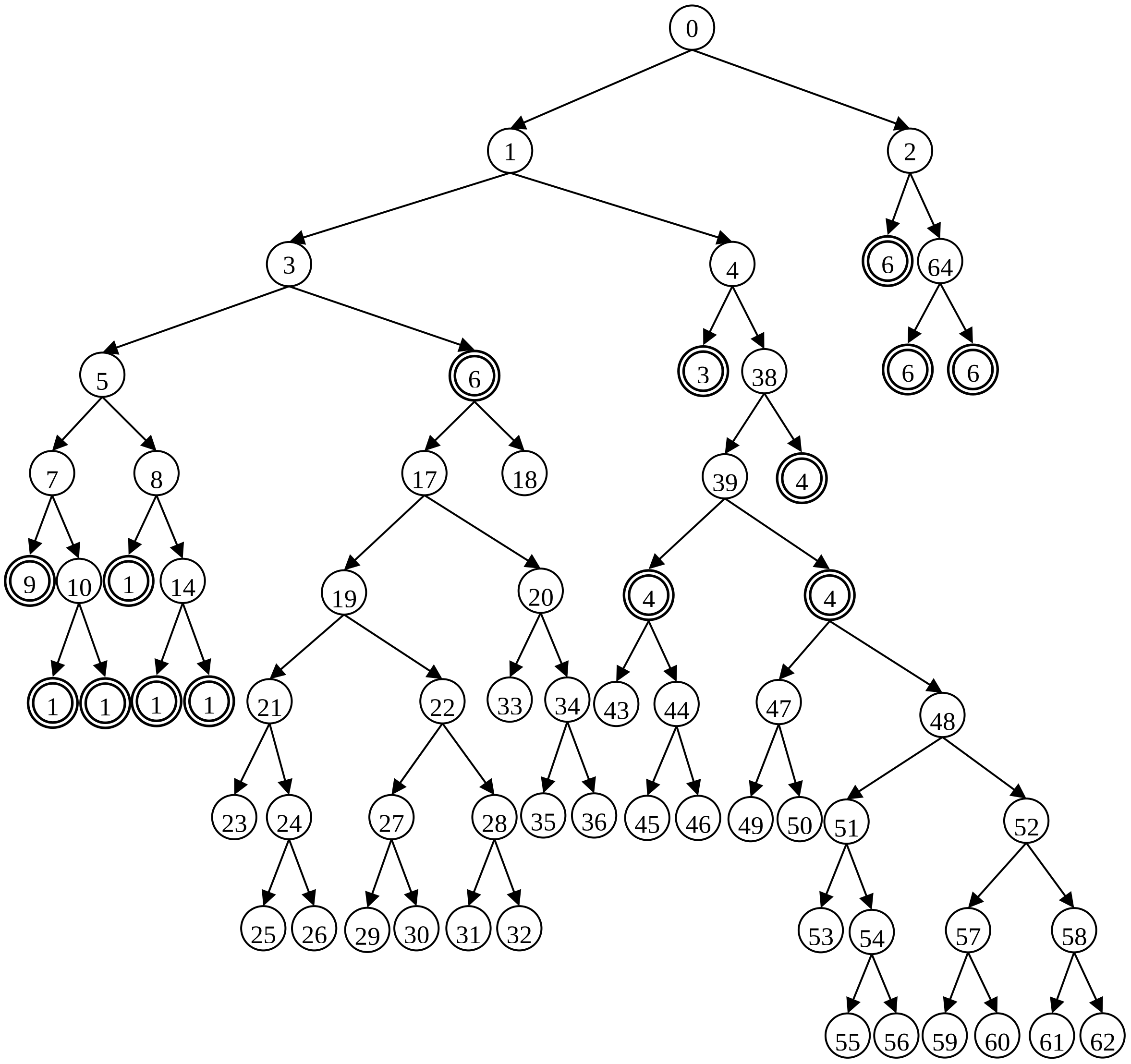


Figure 5.5. Binary Tree Representing the Segmentation Hierarchy (SMG 2000)

Figure 5.5 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. The detected sub-phases for segmentation strength greater than 0 are (33 phases):

$S_9.S_{11}.S_{12}.S_{13}.S_{15}.S_{16}.S_{23}.S_{25}.S_{26}.S_{29}.S_{30}.S_{31}.S_{32}.S_{33}.S_{35}.S_{36}.S_{18}.S_{43}.S_{45}.S_{46}.S_{49}.S_{50}.S_{53}.S_{55}.$
$S_{56}.S_{59}.S_{60}.S_{61}.S_{62}.S_{40}.S_{63}.S_{65}.S_{66}$

By going up the hierarchy, we can get a coarse-grained view of the phases. The leaf nodes (double rounded) when the allowed segmentation strength is above 0.5 are (14 phases):

$S_9.S_{11}.S_{12}.S_{13}.S_{15}.S_{16}.S_6.S_{37}.S_{41}.S_{42}.S_{40}.S_{63}.S_{65}.S_{66}$

It is clear how changing the segmentation strength can affect the number of detected phases in the trace.

5.3.1.3 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 [SMG2000]. 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 at phase S_9 and includes the phases that are in the sub-tree rooted at S_7. Table 5.3 describes the detected sub-phases of the initialization phase.

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 5.5. The Setup phase spans the sub-trees rooted at S_8, S_6 and S_4. Table 5.4 provides a description of the sub-phases in the Setup phase.

Table 5.3. 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).

Table 5.4. 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. S_6 and S_{17} 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 S_6 and S_{17} 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 5.5 presents the description of the sub-phases.

Table 5.5. 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 5.6 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.

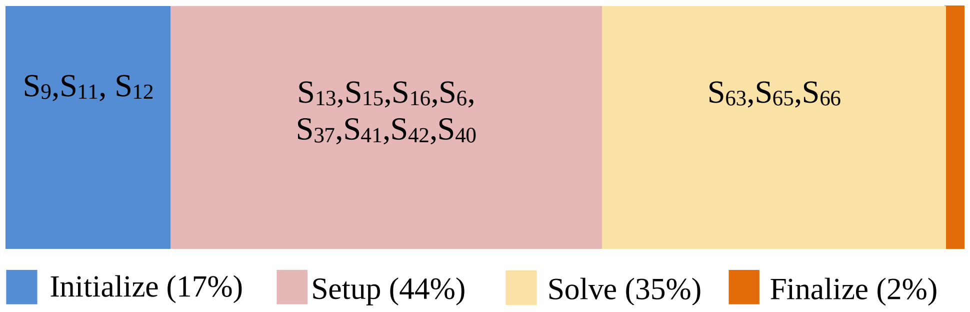


Figure 5.6. Detected Phases in SMG2000

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.

5.3.2 NAS BT

The Block Tridiagonal benchmark is part of the NAS PB [NAS] suite. It uses an implicit algorithm to solve the 3-D compressible Navier-Stokes equations. We generated the trace using VampirTrace [VampirTrace]. Table 5.6 shows some statistics on the generated trace. The process topology is presented in Figure 5.7.

Table 5.6. Statistics for BT Trace

Trace Attribute	**Value**
Size of Trace	0.43GB
Number of Processes	16
Number of Iterations	200
Input Size	24x24x24
Total Number of Events	6856270
Point-to-point Communication Events	154560
Collective Communication Events	160

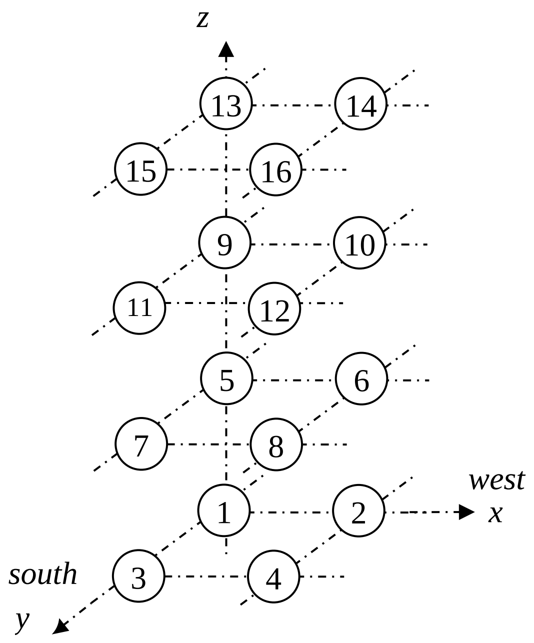


Figure 5.7. NAS BT Process Topology

According to Geisler et al. [Geisler 99], the execution of NAS BT is divided into seven distinct execution phases as follows:

1. Initialization: sets all the initial values.
2. Copy Faces: exchanges boundary values between neighboring processes.
3. Solve Phase:
    a. X Solve: solves the problem in the x-dimension.
    b. Y Solve: solves the problem in the y-dimension.
    c. Z Solve: solves the problem in the z-dimension.
4. Add: performs a matrix update.
5. Final Clean up: verifies the solution integrity, cleans up data, and prints the final results.

In the following, we present the steps that were involved in the phase detection process.

5.3.2.1 Pattern Detection

We used our pattern detection algorithm described in Chapter 4 to detect the communication patterns in the NAS BT trace. The algorithm resulted in 16 distinct patterns (3 collective and 13 point-to-point communication patterns). The total number of patterns instances is 7446 (sequence length). The collective communications are Broadcast, Reduce and All-Reduce.

Table 5.7 presents the events involved in the communication pattern that is used in the Copy Faces routine. This pattern is repeated 201 times in the trace. This is a complex pattern that involves near and far 2-way neighbour communication. This pattern is represented textually due to its complexity which will result in cluttering when represented using the event graph. The reason why the pattern is repeated 201 times instead of 200 (number of iterations) is that there is a dummy iteration before the time step function [NAS Changes].

Table 5.7. Communication Pattern used in Copy Faces (P: Process, e: event, S2: Send to 2, R2: Receive from 2)

P	**e1**	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**	**e8**	**e9**	**e10**	**e11**	**e12**
P1	S2	S4	S5	S13	S8	S14	R2	R4	R5	R13	R8	R14
P2	S3	S1	S6	S14	S5	S15	R3	R1	R6	R14	R5	R15
P3	S4	S2	S7	S15	S6	S16	R4	R2	R7	R15	R6	R16
P4	S1	S3	S8	S16	S7	S13	R1	R3	R8	R16	R7	R13
P5	S6	S8	S9	S1	S12	S2	R6	R8	R9	R1	R12	R2
P6	S7	S5	S10	S2	S9	S3	R7	R5	R10	R2	R9	R3
P7	S8	S6	S11	S3	S10	S4	R8	R6	R11	R3	R10	R4
P8	S5	S7	S12	S4	S11	S1	R5	R7	R12	R4	R11	R1
P9	S10	S12	S13	S5	S16	S6	R10	R12	R13	R5	R16	R6
P10	S11	S9	S14	S6	S13	S7	R11	R9	R14	R6	R13	R7
P11	S12	S10	S15	S7	S14	S8	R12	R10	R15	R7	R14	R8
P12	S9	S11	S16	S8	S15	S5	R9	R11	R16	R8	R15	R5
P13	S14	S16	S1	S9	S4	S10	R14	R16	R1	R9	R4	R10
P14	S15	S13	S2	S10	S1	S11	R15	R13	R2	R10	R1	R11
P15	S16	S14	S3	S11	S2	S12	R16	R14	R3	R11	R2	R12
P16	S13	S15	S4	S12	S3	S9	R13	R15	R4	R12	R3	R9

Figure 5.8 presents four communication patterns that are used in the X Solve routine. Each pattern only involves four processes in the program. These four patterns always occur together. However, the patterns are disconnected and cannot construct one global communication pattern that involves all the processes in the trace. Therefore, each pattern will be represented as a separate symbol in the pattern sequence. Each pattern instance is repeated 603 times in the trace.

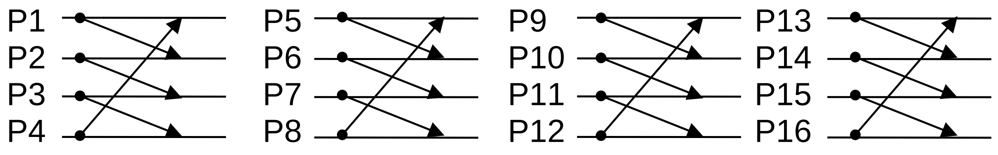


Figure 5.8. Communication Pattern used in X Solve

Figure 5.9 shows another case of a disconnected communication pattern (a pattern that does not involve all the processes in the trace). This communication pattern is used in the X Solve Cell routine. Similar to the pattern in Figure 5.8, each pattern instance will be represented as a separate symbol in the pattern sequence. Each pattern repeats 603 times in the trace.

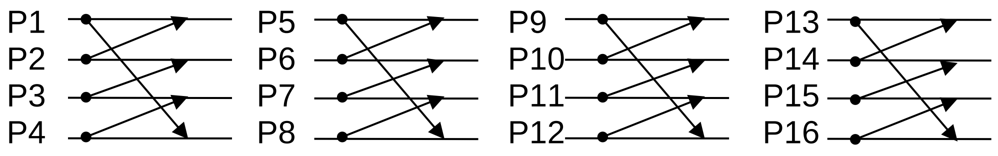


Figure 5.9. Communication Pattern used in X Solve Cell

Figure 5.10 shows the communication patterns used in Y Solve, Y Solve Cell, Z Solve, and Z Solve Cell respectively. These patterns are also repeated 603 times in the trace.

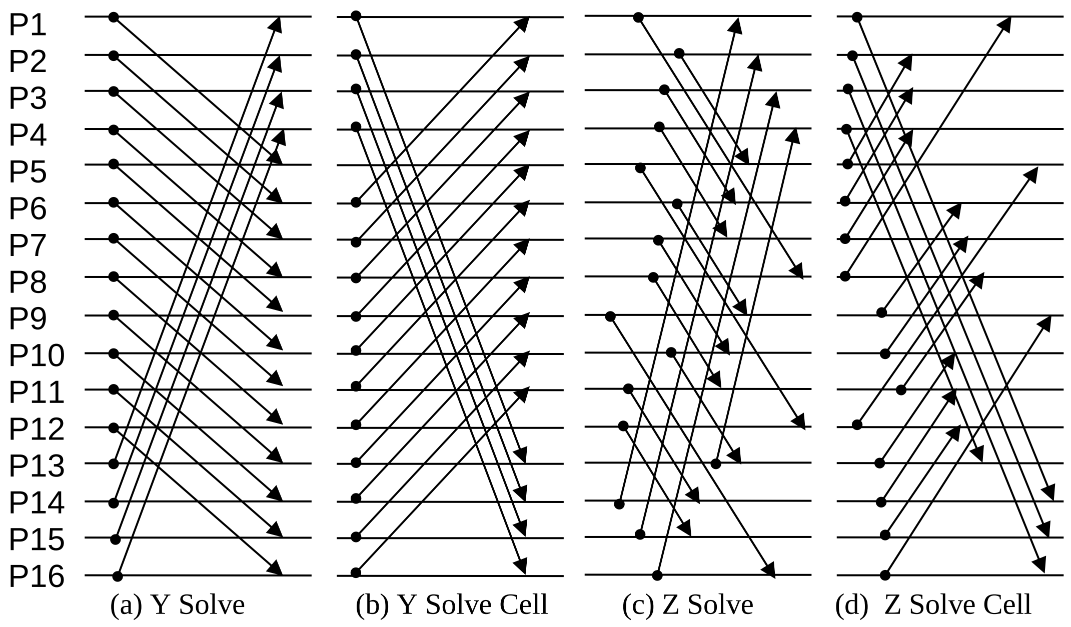


Figure 5.10. Communication Patterns in Y Solve and Z Solve

5.3.2.2 Phase Detection

We applied the recursive segmentation steps to the communication pattern sequence detected in the previous step. Table 5.8 lists the segmentation steps for segmentations strength greater than zero ($s_0 > 0$). The segmentation resulted in 17 segments (including S_7). Segment S_7 is very long and cannot be further segmented using the recursive segmentation algorithm. This is due to the repeating nature of the BT program (the program has 201 iterations that are identical in terms of communication patterns).

Table 5.8. Recursive Segmentation (p_s: start position, p_e: end position, l: length, D_{JS}: Jensen-Shannon Divergence, p_c: cutting position of max divergence, τ: 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	T	s	P
S_0	1	7446	7446	0.01	18	0.01	0.72	NA
S_1	1	18	18	0.95	6	0.12	7.2	**S_0**
S_2	19	7446	7428	0.01	7433	0	0.49	**S_0**
S_3	1	6	6	0.57	5	0.22	1.63	**S_1**
S_4	7	18	12	0.24	8	0.45	-0.47	**S_1**
S_5	1	5	5	-0.1	4	0.46	-1.22	**S_3**
S_6	6	6	1	0	5	0	-	**S_3**
S_7	19	7433	7415	0	33	0.01	-0.44	**S_2**
S_8	7434	7446	13	1.03	7439	0.14	6.23	**S_2**
S_9	7434	7439	6	0.84	7436	0.22	2.92	**S_8**
S_{10}	7440	7446	7	1.01	7442	0.2	4.03	**S_8**
S_{11}	7434	7436	3	-0.17	7435	0.53	-1.32	**S_9**
S_{12}	7437	7439	3	-0.17	7438	0.53	-1.32	**S_9**
S_{13}	7440	7442	3	-0.17	7441	0.53	-1.32	**S_{10}**
S_{14}	7443	7446	4	0.69	7445	0.25	1.75	**S_{10}**
S_{15}	7443	7445	3	0.21	7443	0.53	-0.6	**S_{14}**
S_{16}	7446	7446	1	0	7445	0	-	**S_{14}**

Figure 5.11 depicts the binary tree that resulted from the segmentation algorithm for the communication pattern sequence. The leaf nodes in the tree represent the detected sub-phases in the trace. The detected sub-phases for segmentation strength greater than 0 are (33 phases):

$$S_5.S_6.S_4.S_7.S_{11}.S_{12}.S_{13}.S_{15}.S_{16}$$

In the following section, we present the detected phases using the recursive segmentation algorithm in more detail.

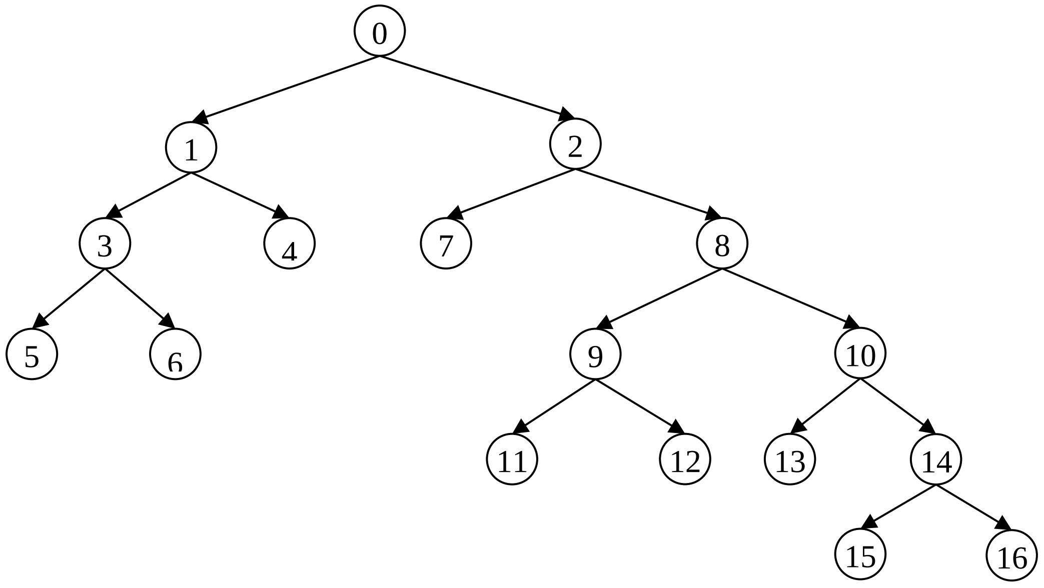


Figure 5.11. Binary Tree Representing the Segmentation Hierarchy (BT)

5.3.2.3 Phase Analysis

Table 5.9 presents the detected phases in the BT trace. S_5 maps to the Initialization phase that exists in every MPI program. S6 contains the pattern presented in Table 5.7 which corresponds to the Copy Faces phase in the program. S_4 corresponds to the X-Solve phase. S_7 is a very long phase with repeating behaviour of (Copy Faces, X Solve, X Solve Cell, Y Solve, Y Solve Cell, Z Solve and Z Solve Cell) sub-phases. It is clear that S_7 is the longest phase that corresponds to the Solve phase in BT. S_{11} corresponds to the Y Solve Cell sub-phase. The Z Solve and Z Solve Cell sub-phases are detected in S_{12} and S_{13} respectively. The Add, Verify and Copy Faces sub-phases occur in the S_{15} segment. Finally, S16 is responsible for printing the results.

The recursive segmentation algorithm was able to detect most of the program sub-phases. The S7 sub-phase is very homogeneous due to its repetitive nature. Further analysis could be applied to S7 in order to detect the repeating behaviour.

Table 5.9. BT Detected Sub-Phases

Phase	**Description**
S_5	Initialization
S_6	Copy Faces
S_4	X Solve
S_7	Very long phase. Starts with X Solve Cell and contains (Copy Faces, X Solve, X Solve Cell, Y Solve, Y Solve Cell, Z Solve and Z Solve Cell sub-phases)
S_{11}	Y Solve Cell
S_{12}	Z Solve
S_{13}	Z Solve Cell
S_{15}	Add, Verify, Copy Faces
S_{16}	Print Results

5.4 Summary

In this chapter, 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 two traces of SMG2000 system and NAS BT benchmark with respect to the documented phases in [Tirawi 11] and [Geisler 99] respectively. Our phase detection approach did not only detect the main program phases but also the corresponding sub-phases.

Chapter 6. Conclusions & Future Work

Dynamic analysis holds a lot of potential in helping with program comprehension tasks. However, the large amount of data in typical execution traces generated from instrumented versions of HPC systems hinders the applicability of dynamic analysis techniques. This led to the emergence of many techniques and tools to facilitate the understanding of the traces of HPC programs.

In this thesis, we presented several techniques that aim to simplify and improve the analysis of traces of HPC programs that use MPI for inter-process communication. In the following section, we summarize the contributions of this thesis.

6.1 Thesis Contributions

MPI Trace Format: different trace formats limit the interoperability among trace analysis tools. We have developed an exchange format for traces of MPI programs. We studied the domain of MPI traces and provided the exchange format as a metamodel. The MTF metamodel is built to meet the requirements for a standard exchange format. It is built to be scalable, extensible, simple and maintainable. We provided a set of queries that can be applied to retrieve trace data. We also provided an example that shows how GXL carries the trace information. We ran different experiments on the metamodel that tested its ability to query information from the execution trace as well as its ability to scale to large execution traces. MTF was published in the Elsevier Future Computer Generation Systems journal and in the Proc. of the International Conference for Program Comprehension (ICPC) 2011.

Communication Patterns Detection Techniques: We presented a new approach for detecting communication patterns from MPI traces based on the concept of n-grams. We have developed different algorithms and showed their applicability on traces generated from HPC programs. We believe that our communication patterns detection approach outperforms the existing studies in terms of quality and performance. Communication patterns can help in understanding HPC programs as they reduce the effort of exploring the whole trace by providing an abstract view of the communication behaviour in the program. The pattern detection and matching approaches were published in the Proc. of the European Conference on Software Maintenance and Reengineering (CSMR), 2011.

Execution Phase Detection of HPC Programs: We presented a new approach for detecting execution phases in MPI programs based on information theory principles. To our knowledge, this is the first study that targets the detection of phases based on the inter-process communication behaviour in the program. We demonstrated the effectiveness of the phase detection technique using two large traces and the results showed the accuracy of the method. This work has been accepted for publication in the International Conference on Program Comprehension [Alawneh 12].

6.2 Directions for Future Research

In this section, we discuss possible future directions in our research.

6.2.1 Support of other message passing paradigms

In this thesis, we have presented a metamodel for trace information generated from HPC programs with specific focus on systems that use the MPI for inter-process communication. In order to support the neutrality requirement, we need to support other message passing models. Moreover, it should be possible to make the model open to any type of inter-process communication in distributed systems that use messages for exchanging data. MTF is designed to be extensible and should be able to accommodate any message passing model.

6.2.2 Support traces of inter-process communication based on shared memory

In this thesis, we did not target traces generated from inter-process communication based on the shared memory model. Different types of applications use this model for inter-process communication. Moreover, some systems use a hybrid of the message passing and the shared memory models. MTF is designed to be extensible and should be able to accommodate this new requirement.

6.2.3 MTF as part of the Knowledge Discovery metamodel (KDM)

Currently, MTF supports traces generated from routine calls and MPI. It has the main components to support traces generated from distributed applications that use MPI for inter-process communication. The Knowledge Discovery Metamodel (KDM) [KDM] is a metamodel that targets a widespread set of software applications, platforms and programming languages such as modern enterprise applications which involve multiple technologies and programming languages. The goal of KDM is to facilitate the integration between different tools that capture information about complex enterprise applications. In [Alawneh 09], we proposed that execution traces should be considered as a new domain. We proposed the usage of KDM to contain this domain. We need to investigate how MTF could be used with KDM.

6.2.4 Formal language for representing traces of inter-process communication

Execution traces generated from MPI programs should be expressed formally in a language that is similar or an extension to some formal languages such as π-calculus. Formal methods can also be used to model the various trace abstraction methods and enable their comparison without the need to generate traces.

6.2.5 Communication patterns visualization

In this work, we used the event graph [Kranzlmüller 00] for visualizing communication patterns. However, we have shown by example that this technique is limited to relatively small patterns. Moreover, the event graph will be of less benefit when the presented patterns are irregular and contain many process interactions. Proposing a new technique that is capable of solving this problem will add a great benefit to the existing trace analysis tools.

6.2.6 Metrics to categorize communication patterns

Complex HPC programs may have many different communication patterns. In many cases, it would be necessary to categorize these patterns based on different factors such as the number of messages, the number of processes involved in the communication, the size of data being exchanged and others. In the literature, we have seen different metrics that provide statistics based on each process separately. We believe that a new set of metrics that characterize the complexity of communication patterns can be very useful in speeding up the program comprehension process of inter-process communication traces.

6.2.7 Phase detection to support homogeneous segmentation

In this thesis, we targeted the detection of phases in heterogeneous sequences of communication patterns. As part of future work, we intend to extend the recursive segmentation algorithm in order to segment sequences of homogeneous communication patterns sequences. This becomes necessary since HPC programs tend to have repetitive communication behaviour which may result in long homogeneous sequences that will not be able to be segmented using the current recursive segmentation approach.

6.2.8 Experimenting with software engineers

As future work, we need to work with software engineers to further validate the techniques presented in this thesis. Software engineers can provide useful feedback that can further improve the trace abstraction techniques.

6.3 Closing Remarks

Large execution traces and the lack of a common exchange format for trace analysis tools of HPC programs limit the applicability of the dynamic analysis approach in the process of program comprehension. We have presented several techniques that cope with the problem of trace size. We showed the usefulness of these techniques using several case studies. The intention behind the development of these techniques is to reduce the impact of the size of traces on the process of understanding the content of these traces and the program in general.

References

[Adjeroh 03]	D. Adjeroh, J. Feng, “Locating all tandem repeat families in a sequence,” *In Proc. of IEEE Computational Systems Bioinformatics Conference*, Palo Alto, CA, pages 676–681, 2003.
[Akaike 78]	H. Akaike, “A Bayesian analysis of the minimum AIC procedure,” *Annals of the Institute of Statistical Mathematics 30 (Part A)*, pages 9-14, 1978.
[Alawneh 09]	L. Alawneh, A. Hamou-Lhadj, “Execution Traces: A New Domain that Requires the Creation of a Standard Metamodel,” *Book Chapter in the Book Series on Communications in Computer and Information Science*, Springer Berlin Heidelberg, pages 253-263, 2009.
[Alawneh 10]	L. Alawneh, A. Hamou-Lhadj, “An Exchange Format for Representing Dynamic Information Generated from High Performance Computing Applications,” In *Future Generation Computer Systems,* The International Journal of Grid Computing and eScience, 27(4), Elsevier Press, pages 381-394, 2011.
[Alawneh 11a]	L. Alawneh, A. Hamou-Lhadj, “Pattern Recognition Techniques Applied to the Abstraction of Traces of Inter-Process Communication,” *In Proc. of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 211-220, 2011.
[Alawneh 11b]	L. Alawneh, A. Hamou-Lhadj, “MTF: A Scalable Exchange Format for Traces of High Performance Computing Systems,” *In Proc. of the 19th IEEE International Conference on Program Comprehension (ICPC)*, pages 181-184, 2011.
[Alawneh 12]	L. Alawneh, A. Hamou-Lhadj, “Identifying Computational Phases from Inter-Process Communication Traces of HPC Applications,” *In Proc. Of the International Conference on Program Comprehension (ICPC 2012),* pages 133-142, 2012.
[Aloision 02]	G. Aloisio, D. Talia, “Grid Computing: Towards a New Computing Infrastructure,” *In Future Generation Computer Systems*, Vol. 18(8), Elsevier Science, pages v-vi, 2002.

[Aydt 94] R. A. Aydt, “The Pablo Self-Defining Data Format,” Technical report, Department of Computer Science, University of Illinois, 1994. http://wotug.kent.ac.uk/parallel/performance/tools/pablo/.

[Ball 99] T. Ball, “The Concept of Dynamic Analysis,” In *Proc. of the 7th European Software Engineering Conference*, Springer-Verlag, pages 216-234, 1999.

[Becker 07] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. N. Wylie, B. Mohr, “Automatic Trace-Based Performance Analysis of Metacomputing Applications,” In *Proc. of the International Parallel and Distributed Processing Symposium*, IEEE Computer Society, pages 1-10, 2007.

[Bowman 00] I. T. Bowman, M. W. Godfrey, and R. C. Holt, “Connecting Architecture Reconstruction Frameworks,” In *Journal of Information and Software Technology*, 42(2), pages 91-102, 2000.

[Brown 92] 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,” In *Journal of Computational Linguistics*, vol. 18, pages 467–479, 1992.

[Cao 05] X. Cao, S. C. Li, and A. K. H. Tung. “Indexing DNA Sequences Using q-grams,” In *Proc. of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 4-16, 2005.

[Cappello 00] F. Cappello, D. Etiemble, “MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks,” In *Proc. of High Performance Networking and Computing Conference (*SC2000*)*, pages 12-es, 2000.

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

[Casas 10] M.Casas , R. M. Badia , J. Labarta, “Automatic Phase Detection and Structure Extraction of MPI Applications,” *International Journal of High Performance Computing Applications*, v.24 n.3, pages 335-360, August 2010.

[Chan 03] A. Chan, R. Holmes, GC. Murphy, Ying ATT. Scaling an object-oriented system execution visualizer through sampling. In *Proc. of* *the 11th Int. Workshop on Program Comprehension (IWPC)*, IEEE, pages 237–244, 2003.

[Chan 08] 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), pages 155–165, 2008.

[Cordella 01] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *Proc. of the 3rd IAPR-TC15 Workshop on Graph based Representations in Pattern Recognition*, Ischia (Italy), pages 149-159, 2001.

[Cormen 90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," *2nd edition, MIT Press*, Cambridge, MA, 2001.

[Cornelissen 08] B. Cornelissen and L. Moonen, "On large execution traces and trace abstraction techniques," *Technical Report*, *TUD-SERG*, 2008.

[Cornelissen 09] B. Cornelissen, A. Zaidman, and A. van Deursen, "A Controlled Experiment for Program Comprehension through Trace Visualization," *Technical Report, TUD-SERG*, 2009.

[Downey 80] J.P. Downey, R. Sethi, R.E. Tarjan, "Variations on the Common Subexpression Problem," *Journal of the ACM*, Volume 27, Issue 4, pages 758-771, 1980.

[Ebert 99] J. Ebert, B. Kullbach, and A. Winter, "GraX – An Interchange Format for Reengineering Tools," In *Proc. of the 6th Working Conference on Reverse Engineering*, pages 89–98, 1999.

[EMF] Eclipse Modeling Framework, URL: http://www.eclipse.org/modeling/emf/.

[FLASH 2] FLASH 2.0 User's Guide, http://flash.uchicago.edu/

[Fürlinger 09] Karl Fürlinger, David Skinner, "Capturing and Visualizing Event Flow Graphs of MPI Applications," In *Proc. of the* 2nd *Workshop on Productivity and Performance (PROPER 2009)*, pages 218-227, 2009.

[Gailly 02] J. L. Gailly and M. Adler, "zlib 1.1.4 Manual," 2002. URL: http://www.zlib.net/manual.html.

[Geimer 06] 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, pages 303–312, Bonn, Germany, Springer, 2006.

[González 09] J. González, J. Giménez, J. Labarta, “Automatic Detection of Parallel Applications Computation Phases,” *In Proc. of International Parallel & Distributed Processing Symposium (IPDPS), pages 1-11,* 2009.

[Gray 11] R. Gray, “Entropy and information theory,” 2nd Edition, New York, *Springer,* 2011.

[Geisler 99] J. Geisler and V. Taylor, “Performance coupling: A methodology for predicting application performance using kernel performance,” In *Proc. of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, pages 125-134, 1999.

[Grissa 07] I. Grissa, G. Vergnaud, C. Pourcel, “CRISPRFinder: a web tool to identify clustered regularly interspaced short palindromic repeats,” *Nucleic Acids Res 35,* W52–57, 2007.

[Grosse 02] 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.

[Gu 06] D. Gu, C. A. Verbrugge, “A Survey of Phase Analysis: Techniques, Evaluation and Applications,” *Sable Technical Report* No. 2006-1, 2006.

[Hamou-Lhadj 04] A. Hamou-Lhadj and T. Lethbridge T., “A Metamodel for Dynamic Information Generated from Object-Oriented Systems,” In *Proc. of the First International Workshop on Meta-models and Schemas for Reverse Engineering*, Electronic Notes in Theoretical Computer Science Volume 94, pages 59-69, 2004.

[Hamou-Lhadj 05] A. 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.

[Heath 03] M. T. Heath and J. E. Finger, “Paragraph: A performance visualization tool for MPI,” A User guide, 2003. URL: http://www.csar.illinois.edu/software/paragraph/.

[Hermes] R. Strom, D. Bacon, A. Lowry, A. Goldberg, D. Yellin, and S. Yemini, “Hermes: A Language for Distributed Computing,” Series in Innovative Technology, *Prentice-Hall*, 482 pages, 1991.

[Holt 00] R. C. Holt, A. Winter, and A. Schürr A., “GXL: Toward a Standard Exchange Format,” In *Proc. of the 7th Working Conference on Reverse Engineering*, pages 162-171, 2000.

[Holt 98] R. C. Holt, “An Introduction to TA: The Tuple Attribute Language,” http://swag.uwaterloo.ca/pbs/papers/ta.html.

[Hong 96] C. Hong, B. Lee, G. On, D. Chi, “Replay for Debugging MPI Parallel Programs,” In *Proc. of the Second MPI Developers Conference*, pages 156-160, 1996.

[Huband 01] S. Huband, D. McDonald, “DEPICT: A topology-based debugger for MPI programs,” In *Proc. of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2001),* Springer, Berlin, pages 109–121, 2001.

[Jain 99] Jain, A. K., Murty, M. N., and Flynn, P. J., “Data clustering: A review,” *ACM Computing Surveys* Vo. 31, pages 264–323, 1999.

[Jokinen 91] P. Jokinen and E. Ukkonen “Two algorithms for approximate string matching in static texts,” In *Proc. of the 2nd Annual Symposium on Mathematical Foundations of Computer Science*, pages 240–248. 1991.

[Jumpshot] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider, “Toward scalable performance visualization with Jumpshot,” Journal of *High Performance Computing Applications*, Vo. 13(2), pages 277–288, 1999.

[Karp 72] R. Karp, R. E. Miller, A. L. Rosenberg. “Rapid Identification of Repeated Patterns in Strings, Trees and Arrays,” In *Proc. of 4th Symposium of Theory of Computing,* pages125-136, 1972.

[KDM] Object Management Group. Knowledge Discovery Metamodel: KDM Version 1.1, 2008.

[Kergomm 03] J. Chassin de Kergommeaux, B. de Oliveira Stein, and G. Mouni, "Paje Input Data Format," *Technical Report*, Intel GmbH, Brühl, Germany, 2003.

[Kim 94] JY. Kim, J. Shawe-Taylor, "Fast string matching using an n-gram algorithm," *Journal of Software Practice & Experience*, 94(1), pages 79–88, 1994.

[Kingsbury 07] B. Kingsbury, "Organizing processes and threads for debugging," In *Proc. of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 21-26, 2007.

[Knüpfer 05] A. Knüpfer, W.E. Nagel, "Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis," In *Proc. of the International Conference on Parallel Processing (ICCP 2005),* , pages 165–172, 2005.

[Knüpfer 06a] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel, "Introducing the open trace format (OTF)," In *Proc. of the International Conference on Computational Science (ICS),* pages 526-533, 2006.

[Knüpfer 06b] A. Knüpfer, B. Voigt, W.E. Nagel, H. Mix, "Visualization of repetitive patterns in event traces," In *Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA),* pages 430-439, 2006.

[Köckerbauer 10] T. Köckerbauer, T. Klausecker and D. Kranzlmüller, "Scalable Parallel Debugging with g-Eclipse," *Book Chapter, Springer Berlin Heidelberg,* pages 115-123, 2010.

[KOJAK] KOJAK, URL: http://icl.cs.utk.edu/kojak/

[Kranzlmüller 00] D. Kranzlmüller. Event Graph Analysis for Debugging Massively Parallel Programs. *PhD thesis*, GUP Linz, Joh. Kepler University Linz (September 2000). http://www.gup.uni-linz.ac.at/~dk/thesis.

[Kranzlmüller 02] D. Kranzlmüller, "Scalable Parallel Program Debugging with Process Isolation and Grouping," In *Proc. of the 16th International*

Parallel and Distributed Symposium (IPDPS2002), pages 109-115, 2002.

[Kranzlmüller 95] D. Kranzlmüller, S. Grabner, J. Volkert, "Message Passing Visualization with ATEMPT," In *Proc. of PARCO 1995 Conference on Parallel Computing*, Gent, Belgium, pages 653-656, 1995.

[Kunz 95] T. Kunz, J.P Black, "Using Automatic Process Clustering for Design Recovery and Distributed Debugging," *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, pages 515-527, 1995.

[Kunz 97] T. Kunz, M. F. H. Seuren, "Fast detection of communication patterns in distributed executions," In *Proc. of the conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 12, 1997.

[Larus 99] J.R. Larus, "Whole program paths," In *Proc. of the Conference on Programming Language Design and Implementation*, ACM Press, pages 259-269, 1999.

[Lee 09] I. Lee, "Characterizing communication patterns of nas-mpi benchmark programs," *In Proc. of Southeast Conference*, Atlanta, pages 158-163, 2009.

[Lethbridge 97] T. C. Lethbridge and N. Anquetil, "Architecture of a Source Code Exploration Tool: A Software Engineering Case Study," *Computer Science Technical Report TR-97-07*, University of Ottawa, 1997.

[Levenshtein 66] A. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pages 707-710, 1966.

[Li 02] W. Li, P. Bernaola-Galvan, F. Haghighi, I. Grosse, "Applications of recursive segmentation to the analysis of DNA sequences," *Journal of Computers & Chemistery 26*, pages 491-510, 2002.

[Lu 04] Q. Lu, J. Wu, D. Panda, P. Sadayappan, "Applying MPI derived Datatypes to the NAS Benchmarks: A case study," In *Proc. of the 2004 International Conference on Parallel Processing Workshops*, Montreal, Quebec, Canada, pages 538-545, 2004.

[Ma 09] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, S. See, "An approach for matching communication patterns in parallel applications," *In Proc. of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1-12, 2009.

[Maghraoui 05] K. El Maghraoui, B. K. Szymanski, C. A. Varela, "An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments," In *Proc. of the Sixth International Conference on Parallel Processing and Applied Mathematics (ICPP AM)*, pages 258-271, 2005.

[McKay 81] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, 30, pages 45-87, 1981.

[Moore 05] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. "A Scalable Approach to MPI Application Performance Analysis," *Lecture Notes of Computer Science (LNCS)* 3666, pages 309–316, 2005.

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

[Margaris 09] A. I. Margaris, "Log File Formats for Parallel Applications: A Review," *International Journal of Parallel Programming*, 37(2), pages 195-222, 2009.

[Mudalige 08] G.R. Mudalige, M.K. Vernon, S.A. Jarvis, "A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures," In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, pages 1-14, 2008.

[Muelder 09] C. Muelder, F. Gygi, and K.-L. Ma, "Visual analysis of inter-process communication for large-scale parallel computing," *IEEE Transactions on Viualization and Computer Graphics*, 15(6), pages 1129-1136, 2009.

[Müller 88] H. A. Müller, K. Klashinsky, "Rigi – A System for Programming in-the-large," In *Proc. of the 10th International Conference on Software Engineering*, pages 80-86, 1988.

[NAS] NAS Parallel Benchmarks, http://www.nas.nasa.gov/

[NAS-Changes] NAS Parallel Benchmarks Changes, http://www.nas.nasa.gov/Resources/Software/npb_changes.html

[Navarro 07] G. Navarro, V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys (CSUR)*, v.39 n.1, pages 2-es, 2007.

[Noeth 09] M. Noeth, P. Ratn, F. Mueller, M. Schulz, B. R. De Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*. Vo. 69, pages 696-710, 2009.

[Okita 03] M. Okita, F. Ino, K. Hagihara, "Debugging Tool for Localizing Faulty Processes in Message Passing Programs," In *Proc. of the Fifth International Workshop on Automated Debugging*, pages 127-142, 2003.

[OMG] Object Management Group, http://www.omg.org/uml

[PAJE] J. Kergommeaux and B. Stein, "Paje: an extensible and interactive and scalable environment for visualizing parallel executions," *Rapport de Recherche, No. 3919, Institut National de Recherche en Informatique et en Automatique (INRIA)*, France, 2000.

[Palma 09] N. Palma, "Performance Evaluation of Interconnection Networks using Simulation: Tools and Case Studies" *PhD Dissertation*, Department of Computer Architecture and Technology, University, Spain, 2009.

[Paramesh] P. MacNeice, K.M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, "PARAMESH: a parallel adaptive mesh refinement community toolkit," *Journal of Computer Physics Communications*, 26(3), pages 330-354, 2000.

[ParaProf] R. Bell, A. Malony, and S. Shende, "ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis," *Euro-Par*, pages 17-26, 2003.

[Paraver] CEPBA (European Center for Parallelism in Barcelona), Barcelona/Spain. PARAVER Version 3.0 Trace file Description, June 2001.

[Patel 05] Nikunj Patel, Padma Mundur, “An n-gram based approach to finding the repeating patterns in musical data,” In *Proc. of the European Internet and Multimedia Systems and Applications (IMSA)*. Grindelwald, Switzerland, pages 407-412, 2005.

[Preissl 08] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, “Detecting patterns in MPI communication traces,” *In Proc. of International Conference on Parallel Processing (ICPP)*, pages 230–237, 2008.

[Preissl 10] R. Preissl, B. R. de Supinski, M. Schulz, D. J. Quinlan, D. Kranzlmüller, T. Panas, “Exploitation of Dynamic Communication Patterns through Static Analysis,” In *Proc. of International Conference on Parallel Processing (ICPP)*, pages 51-60, 2010.

[Qu 09] Q. Xu, J. Subhlok, R. Zheng, S. Voss, “Logicalization of communication traces from parallel execution,” In *Proc. of the 2009 IEEE International Symposium on Workload Characterization, (IISWC)*, October 4-6, 2009, Austin, TX, USA. pages 34-43, 2009.

[Ranjan 08] R. Ranjan, A. Harwood, R. Buyya, “A case for cooperative and incentive-based federation of distributed clusters,” *Future Generation Computer*, Volume 24(4), pages 280-295, 2008.

[Rasmussen 06] K. Rasmussen, J. Stoye, EW Myers, “Efficient q-Gram Filters for Finding All ϵ-Matches Over a Given Length,” *Journal of Computational Biology*, Volume 13, pages 296–308, 2006.

[Reiss 01] S. P. Reiss and M. Renieris, “Encoding program executions,” *In Proc. of the 23rd International Conference on Software Engineering*, ACM Press, pages 221-230, 2001.

[Roberts 05] J. Roberts and C. Zilles. “TraceVis: an execution trace visualization tool,” *In Proc. of Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2005.

[Rose] Rational Rose, URL: http://www-01.ibm.com/software/rational/

[Safyallah 06] H. Safyallah and K. Sartipi, “Dynamic Analysis of Software Systems using Execution Pattern Mining,” In *Proc. of the 14th IEEE International Conference on Program Comprehension*, pages 84-88, 2006.

[Sadakane 07] K. Sadakane. “Compressed suffix trees with full functionality,” Journal of *Theory of Computing Systems*, Volume 41(4), pages 589–607, 2007.

[Shannon 48] C.E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, Volume 27, pages 379-423, 1948.

[Shende 04] S. Shende. TAU Source Code, Version 2.13.5. Personal Communications, 2004.

[Shende 05] S. Shende and A. D. Malony, “The TAU Parallel Performance System,” *International Journal of High Performance Computing Applications (ACTS Collection)*, Volume 20, pages 287-331, 2005.

[Singh 09] R. Singh and P. Graham, “Grouping MPI Processes for Partial Checkpoint and Co-migration,” In *Proc. of Euro-Par 2009 Parallel Processing*, Lecture Notes in Computer Science, Volume 5704. Springer Berlin Heidelberg, pages 69-80, 2009.

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

[Stamatakis 05] A. Stamatakis, M. Ott, T. Ludwig, and H. Meier, “DRAxML@home: A Distributed Program for Computation of Large Phylogenetic Trees,” In *Future Generation Computer Systems* (FGCS) 21(5), pages725-730, 2005.

[Stoye 02] J. Stoye, D. Gusfield, “Simple and Flexible Detection of Contiguous Repeats Using a Suffx Tree,” *Theorical Computer Sciences*, Volume 27(1-2), pages 843-856, 2002.

[St-Denis 00] G. St-Denis, R. Schauer, and R. K. Keller, “Selecting a Model Interchange Format: The SPOOL Case Study,” In *Proc. of the 33rd Annual Hawaii International Conference on System Sciences*, pages 4-7, 2000.

[STF] Intel Trace Collector User’s Guide. URL: http://www.uybhm.itu.edu.tr/documents/ITC-ReferenceGuide.pdf

[Sweep3d] Sweep3D, Accelerated strategic computing initiative. The ASCI Sweep3D Benchmark Code. URL:

http://public.lanl.gov/hjw/CODES/SWEEP3D/sweep3d.html, LANL 1995.

[Terzi 06] E. Terzi, P. Tsaparas, "Efficient Algorithms for Sequence Segmentation," In *Proc. of the SIAM International Conference on Data Mining*, pages 314-325, 2006.

[Tiwari 11] 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 Performance Computing Applications*, Volume 25(3), pages 286-294, 2011.

[Valiente 00] G. Valiente, "Simple and Efficient Tree Pattern Matching," *Research Report E-08034*, Technical University of Catalonia, 2000.

[Vampir] Vampir Performance Optimization Tool. URL: http://www.vampir.eu.

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

[Volko 05] Z. Volkovich and et al., "The method of n-grams in large-scale clustering of DNA texts*," Journal of Pattern Recognition*, Volume 38(11), pages 1902-1912, 2005.

[Welch 84] T. A Welch., "A technique for high-performance data compression," *Journal of Computer*, 17(6), pages 8-19, 1984.

[Wolf 04] F. Wolf and B. Mohr, "EPILOG Binary Trace-Data Format," Technical report, University of Tennessee, 2004.

[Wolf 07] F. Wolf, B. Mohr, J. Dongarra, S. Moore, Automatic analysis of inefficiency patterns in parallel applications, *Journal of Concurrency and Computation: Practice and Experience*, Special Issue: European–American Working Group on Automatic Performance Analysis (APART), 19(11), pages 1481–1496, 2007.

[Wolf 08] F. Wolf, D. Becker, M. Geimer, B. J. N. Wylie "Scalable performance analysis methods for the next generation of supercomputers," In *Proc. of the John von Neumann Institute for Computing (NIC) Symposium*, volume 39 of NIC-Series, pages 315-322, 2008.

[Woods 99] S. Woods, S. J. Carrière., and R. Kazman R., "A semantic foundation for architectural reengineering and interchange," In *Proc. of International Conference on Software Maintenance*, pages 391–398, 1999.

[WRF] Weather Research & Forecasting Model (WRF). URL: http://www.wrf-model.org.

[Wu 11] X. Wu and F. Mueller. "ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Program," In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10, 2011.

[XMI-OMG] XMI: XML Metadata Interchange ,URL: http://www.omg.org/technology/documents/formal/xmi.htm

[Xu 06] Q. Xu, J. Subhlok, R. Zheng, and S. Voss, "Localization of communication traces from parallel execution," In *Proc. of IEEE Int. Symposium on Workload Charactization (IISWC*), Austin, TX, pages 34-43, 2009.

[Xue 09] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, W. Zhang, and G. Voelker, "MPIWiz: subgroup reproducible replay of mpi applications," In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 251-260, 2009.

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

Appendix A - The detailed specification of MTF

1) Scenario

Semantics

Objects of the Scenario class represent the system scenario executed in order to generate the traces that need to be studied.

Attributes

- desc: Specifies a description of the usage scenario such as the name of the scenario, input data, etc.

Associations

- Trace [1..*]: References the execution traces that are generated after the execution of the usage scenario. One scenario may have more than one trace object.

2) Trace

Semantics

A concrete class representing common information about traces generated from the execution of the system.

Attributes

- TraceID: A unique identifier for the generated trace.
- StartTime: Specifies the starting time of the generation of the trace.
- EndTime: Specifies the ending time of the generation of the trace.
- Comments: Specifies comments that software engineers might need in order to describe the circumstances under which the trace is generated.

Associations

- Scenario [1]: References the usage scenario that is exercised so as to generate the trace.
- PatternOccurrence [*]: References the occurrences of the execution patterns that are invoked in the trace.
- ProcessTrace: A Trace may have many instances of ProcessTrace.
- MsgTrace: A Trace may have only one instance of MsgTrace.

Constraints

[1] StartTime and EndTime should be different

self.EndTime >= self.StartTime

3) ProcessTrace

Semantics

An object of the ProcessTrace represents a trace generated from all operations executed by. This class inherits from the Trace class.

Attributes

No additional attributes. The start and end time for a ProcessTrace are different from those of the parent Trace class. However, MTF is designed that the ProcessTrace call can access the StartTime and EndTime of the parent Trace class. The start time is the start time of the first routine call and the end time is end time of the lass routine call for a process.

Associations

- Trace: the ProcessTrace class has an association with its parent class. An instance of ProcessTrace can only belong to one instance of Trace class.
- Process: A ProcessTrace may have only one instance of Process class.
- TraceableUnit [0..*]: A reference to all instances of TraceableUnit class that are of types MPOperation and Message.

Constraints

- ProcessTrace references objects created from RoutineCall class only.

4) MsgTrace

Semantics

An object of the MsgTrace represents a trace generated from all the messages (instances of Message class). This class inherits from the Trace class.

Attributes

No additional attributes.

Associations

- Trace: the MsgTrace class has an association with its parent class. An instance of MsgTrace can only belong to one instance of Trace class.
- Message: A MsgTrace may have as many instances of the Message class in the trace.

– CollectiveData: A MsgTrace may have as many instances of the CollectiveData class in the trace.

Constraints

– MsgTrace references objects created from Message class only.
– No Start and End times are used for this class.

5) TracePattern

Semantics

TracePattern is an abstract class that represents communication and routine-call patterns in the trace.

Attributes

– desc: Specifies a textual description that a software engineer assigns to the execution pattern.

Associations

– PatternOccurrence [*] References the instances of the pattern in the trace.

Constraints

[1] The PatternOccurrence objects belong to the same trace.

6) CommPattern

Semantics

CommPattern inherits from the TracePattern class. It represents the inter-process communication patterns in the trace.

Attributes

– No additional attributes.

Constraints

[2] The CommPattern class only references instances of Message and CollectiveData classes.

7) RoutinePattern

Semantics

RoutinePattern inherits from the TracePattern class. It represents the routine call patterns in the trace.

Attributes

- No additional attributes.

Constraints

[3] The RoutinePattern class only references instance of RoutineCall class.

8) PatternOccurence

Semantics

This class represents the instances of an execution pattern.

Associations

- TracePattern [1] References the TracePattern object for which this object represents an occurrence of the pattern.
- Trace [1] References the Trace object where the pattern pointed to by the PatternOccurrence object appears.
- TraceableUnit [*] References the TraceableUnit instances that belong to the pattern occurrence.

Constraints

No additional constraints.

9) TraceableUnit

Semantics

This is an abstract meta-class which represents any traceable element in an execution trace. This class is not restricted to the Message Passing metamodel. Any execution trace metamodel can use this class.

Attributes

- TraceableUnitID: a unique identifier assigned to the traceable unit.
- StartTime: a timestamp specifies when the traceable unit started execution.
- EndTime: a timestamp specifies when the traceable unit finished execution.

Associations

- Process [1]   : references the Process object that represents the process in which this traceable unit is executed.

- MPITrace [1]: in our model, every TraceableUnit element belongs to one trace represented by the class MPITrace. Other traces such as method call traces should have another class defined such as 'MethodCallTrace' to capture traces of MPI operations.

- PatternOccurence [0..1]: a reference to the PatternOccurence class. Every traceable unit may belong to one pattern occurrence object.

Constraints

[1] The StartTime timestamp of TraceableUnit objects that belong to one process must be sorted in an ascending order. This guarantees the order of execution of the message passing operations. Traces of type Message and traces of type Point-to-point operation may have the same start or end times.

10) Edge

Semantics

Edge is a concrete class that represents an edge from a caller routine to a callee routine in the trace.

Attributes

- repeat: indicates how many instances of the callee are represented by the edge.
- type: indicates the type of the edge; recursive, sequence or fork-sequence.

Associations

- TraceableUnit [1] a parent traceable unit may have many outgoing edges.
- TraceableUnit [1..*] a child traceable unit may have only one incoming edge.

Constraints

No additional constraints.

11) Process

Semantics

This class represents a software process. Instances of this class may represent processes in a distributed environment or may represent processes running on the same processor.

Attributes

- ProcessID: a unique identifier in the model that identifies the process.
- Rank: the rank of the process in an MPI group.
- ProcessName: the name designated to the process in the trace.

Associations

- TraceableUnit [*]: a process may have many instances of traceable units.
- Communicator [*]: a process may belong to many MPI communicators.
- Processor [1]: a process runs on one processor only.

12) **Processor**

Semantics

This class represents the *processor* that a *process* runs on.

Attributes

- ProcessorID: a unique identifier is specified for every processor in the system.
- ProcessorName: the name designated to the processor in the trace.

Associations

- Process [*]: a *processor* may contain many running *processes*.

13) **Communicator**

Semantics

This class belongs to the Message Passing environment. A communicator represents a group of processes that communicate through message passing. Processes in a communicator are ranked from 0 to *n*-1, where *n* is the total number of processes.

Attributes

– CommID: the unique identifier for an MPI communicator.

Associations

– Process [1..*]: a communicator may contain one or many processes.
– MPOperation [*]: a communicator may be used by many message passing operations.

14) Message

Semantics

This class captures messages exchanged in point-to-point communications. Message is a direct child of the TraceableUnit meta-class.

Attributes

– DataType: the type of data in the message.
– DataSize: the size of data in the message.
– Tag: the tag sent in the message.

Associations

– MsgTrace: An instance of message belongs to one MsgTrace only.
– MessageLink: Message may have many instances of MessageLink.
– Process (sender): a message may have only one sender.
– Process (receiver): a message may have only one receiver.

Constraints

– Instances of the class Message only correspond to data exchanged in point-to-point operations.

15) MessageLink

Semantics

MessageLink is a concrete class that represents a link between an instance of Message and its corresponding point-to-point operations.

Attributes

– MessageLinkID: the id of the message link.

Associations

- Message[1] MessageLink may have an association with one instance of Message.
- Send [1]: a message link is associated with one Send operation.
- Receive [1]: a message link is associated with one Receive operation.

Constraints

No additional constraints.

16) RoutineCall

Semantics

Routine is a concrete class that represents all the instances of routine calls in the trace.

Attributes

- routineCallName: the name of the routine.
- nestingLevel: the nesting level of the routine in the call tree.

Associations

- No additional associations.

Constraints

No additional constraints.

17) MPOperation

Semantics

A concrete class is at the core of our message passing execution trace model. It acts as a super-class for every message passing operation such as Send, Receive, Gather and Broadcast. An MPOperation is a traceable element and is a direct child to the RoutineCall class.

Attributes

- No additional attributes.

Associations

- Communicator [0..1]: a message passing operation may reference up to one communicator object.

Constraints

No additional constraints.

18) Initialize

Semantics

This class models the MPI_Init routine which is responsible for the initialization of the MPI environment. It is the first MPI call in the program. The initialization of the MPI environment includes synchronization of processes and adding processes to the MPI_COMM_WORLD communicator. In our trace metamodel, MPI_Init inherits from MPOperation.

Associations

- MPI_Init is a child of the MPOperation class. Therefore, it will inherit all the associations of its parent class.

Constraints

[1] A call to MPI_Init must precede any other MPI call in the program, except for MPI_Initialized routine that can be used to check if MPI_Init has been called or not.

19) Finalize

Semantics

This class models the MPI_Finalize routine that is used to clean up the MPI state. Each process must call MPI_Finalize before it exits. Before calling MPI_Finalize, each process must ensure that all pending non-blocking communications are (locally) complete.

Associations

MPI_Finalize is a child of the MPOperation class. Therefore, it will inherit all the associations of its parent class.

Constraints

[1] Every process in the MPI environment must call MPI_Finalize before exiting unless a call to MPI_Abort has been made.

20) **PointToPointOperation**

Semantics

This class is the super-class for blocking and non-blocking point to point operations in the message passing environment. It inherits directly from the MPOperation class.

Constraints

[1] Datatype between matching point-to-point operations must match unless MPI_BYTE data type is specified.

21) **Send**

Semantics

This class represents a message passing *send* operation. *Send* is a direct child of the MPOperation class. Blocking Send operations are directly instantiated from the *Send* class. Non-blocking operations are instantiated from the *NonBlockingSend* class described below.

Attributes

- SendDataAddress: address of sent data.
- SendDataSize: number of sent elements.
- SendDataType: the type of data being sent to destination process.
- Tag: the tag value (integer) sent with the message.
- SendType: this attribute specifies the type of the send operation (Standard, Buffered, Synchronous and Ready).

Associations

- Process [0..1]: the receiving process.
- Receive [0..1]: a message passing send may reference (match) zero or one message passing receive operations.

Constraints

[1] Send operation must specify a receiving process.
[2] A blocking Send with *SendType ≠ Buffered* cannot terminate before a matching *Receive* is posted (end time of send operation must be after start time of receive operation).
[3] A blocking Send with SendType = Synchronous cannot terminate before a matching Receive is posted.

22) **NonBlockingSend**

Semantics

This class represents non-blocking send operations. A process that makes a non-blocking send call proceeds right after the *send* call has been made.

Attributes

No additional attributes.

Associations

- WaitOperation [0..1]: an object of a non-blocking send operation may be referenced by one WaitOperation object.
- TestOperation [0..*]: an object of a non-blocking send operation may be referenced by zero or more TestOperation objects.

23) **Receive**

Semantics

This class represents the message passing *Receive* operation. It is a direct child of the *PointToPointOperation* class. Matching the Send and Receive operations is done by comparing the values to the instances of the Messsage class.

Attributes

- RcvDataAddress: address of the received message buffer at the receiver.
- RcvDataSize: number of elements received at the Receive address.
- Tag: an integer value that should be matched with the coming process unless if specified as MPI_ANY_TAG.

Associations

- Send [0..1]: a message passing receive may reference (match) zero or one message passing send operations.
- Process [0..1]: represents the sender of the message. A receive operation may specify MPI_ANY_SOURCE, in this case the Source process can not be determined as part of the trace for the receive operation. The source will be determined once the message is received at the receiver.

Constraints

No additional constraints.

24) **NonBlockingReceive**

Semantics

This class represents a trace of a non-blocking message passing *Receive* operation. It provides a handle to an object that will be used to check for the completion of the receive operation. A process that uses a non-blocking receive will proceed after calling the receive operation.

Attributes

No additional attributes.

Associations

- WaitOperation [0..1]: an object of a non-blocking receive class may be referenced by one WaitOperation objects.
- TestOperation [0..*]: an object of a non-blocking receive class may be referenced by zero or more TestOperation objects.

25) **WaitOperation**

Semantics

This class represents the different types of Wait operations provided by MPI which can be used to wait and check for the completion of non-blocking message passing operations.

Attributes

No additional attributes.

Associations

- NonBlockingSend [1]: a wait statement references the non-blocking send object that it is performing the wait operation for.
- NonBlockingReceive [1]: a wait statement references the non-blocking receive object that it is performing the wait operation for.

Constraints

[1] The StartTime of an MPI_Wait statement cannot occur before the StartTime of the corresponding Send or Receive operations.

26) TestOperation

Semantics

This class represents traces of the different Test operations provided by MPI. An MPI Test is similar to MPI Wait except that the process does not wait for the completion of the non-blocking operation.

Attributes

- Flag: this flag returns true if the non-blocking operation has completed successfully, false otherwise.

Associations

- NonBlockingSend [0..*]: a test statement references the non-blocking send class that it is performing the test operation for.

- NonBlockingReceive [0..*]: a test statement references the non-blocking receive class that it is performing the test operation for.

Constraints

[1] The StartTime of an MPI_Test statement cannot occur before the StartTime of the corresponding Send or Receive operations.

27) ProbeOperation

Semantics

An MPI probe operation is used to check whether there is an incoming message that matches the Source, Tag, and Communicator except for MPI_ANY_SOURCE and MPI_ANY_TAG.

Attributes

- Tag: this is an integer value that is sent with the message.
- Flag: indicates whether the incoming message matches the expected one.

Associations

- Process [0..1]: specifies the source process (sending process).

Constraints

- If MPI_ANY_SOURCE is indicated, ProbeOperation will not have a reference to the Sending process.

28) **CollectiveOperation**

Semantics

This abstract class is the parent class of all the collective operations in the message passing environment. Collective operations involve all the processes in a communicator.

Associations

- CollectiveData [0..1]: Collective operations other than Barrier will reference one object of the CollectiveData.
- Process [0..1]: represents the root process in the collective operation.

Constraints

[1] A collective operation should match the same type of collective operation in all other processes. Therefore, the maximum number of matched operations may not exceed the number of processes in a communicator.

29) **CollectiveData**

Semantics

This class describes the data being exchanged in a collective operation as well as the address of the exchanged data for each process. The Barrier operation does not involve any data exchange. Therefore, the MPI_Barrier operation does not instantiate a CollectiveData association.

Attritbues

- SendSize: the size of sent data.
- RcvSize: the size of received data.
- SendAddress: the address of sent data.
- RcvAddress: the address of received data.
- SendDataType: the data type of sent data.
- RcvDataType: the data type of received data.

Associations

- CollectiveOperation [1]: an instance of CollectiveData may belong to one CollectiveOperation object.
- MsgTrace [1]: CollectiveData instance belongs to one instance of MsgTrace only.

Constraints

[1] An object of type Barrier cannot reference an object of type CollectiveData.

30) **Barrier**

Semantics

This class represents the message barrier operation (MPI_Barrier) in a message passing environment. It inherits directly from the CollectiveOperation class.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

[1] The end-time for a Barrier object of one process cannot be before the start-time for any of the matched Barrier objects of the other processes.
[2] A Barrier object cannot have an associated instance of class *CollectiveData*.

31) **Broadcast**

Semantics

This class represents the broadcast operation (MPI_Bcast) in the message passing environment. It inherits directly from the CollectiveOperation class.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

[1] The type signature (SendSize, SendDataType) for MPI_Bcast at the root process must be equal to the type signature of the matching MPI_Bcast on all processes (receiving processes) in the communicator.
[2] The root process must belong to the communicator group.

32) **Gather**

Semantics

This class represents the gather operation (MPI_GATHER and MPI_GATHERV) in a message passing environment. It inherits directly from the CollectiveOperation class. In

MPI_Gather, the root process receives the messages and stores them in rank order. The receiving buffer (RcvAddress) for non-root processes is ignored for this operation.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

[1] The type signature (SendSize, SendDataType) for MPI_Gather at the root must be equal to the type signature of the matching MPI_Gather on all processes (sending processes) in the communicator.
[2] The gathered (received) message should be sorted based on the process rank in the communicator.
[3] The root process must belong to the communicator.
[4] The receiving buffer for non-root process should be equal to null.

33) **Scatter**

Semantics

This class represents the scatter operation (MPI_Scatter and MPI_Scatterv) in a message passing environment. It inherits directly from the CollectiveOperation class. The send buffer is ignored for all non-root processes.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

[1] The type signature (SendSize, SendDataType) for MPI_Scatter at the root must be equal to the type signature of the matching MPI_Scatter on all processes (receiving processes) in the communicator.

34) **Reduce**

Semantics

This class represents the Reduce operation (MPI_Reduce) in a message passing environment. Every process will send a value to the root process.

Attributes

- OpType: the type of the executed operation on the received data at the root process.

Associations

No additional associations.

Constraints

[1] All processes provide input buffers and output buffers of the same length, with elements of the same type.

35) **Allgather**

Semantics

Traces from MPI_ALLGATHER and MPI_ALLGATHERV are captured using the Allgather class. This class is a direct subclass of the CollectiveOperation class.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

[1] Instances of AllGather do not reference a root process.

36) **AllToAll**

Semantics

Traces from MPI_ALLTOALL and MPI_ ALLTOALLV are captured using the AllToAll class.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

[1] Instances of AllGather do not reference a root process.

37) **ReduceScatter**

Semantics

Traces from MPI_REDUCE_SCATTER are captured using the ReduceScatter class.

Attributes

- OpType: the type of the executed operation on the received data at the root process.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

38) **Scan**

Semantics

Traces from MPI_Scan operation are captured using the Scan class. The Scan class is a subclass of CollectiveOperation class. A Scan operation is used to perform a prefix reduction on data exchanged across the group. For a process with rank *i*, the scan operation returns, in the receive buffer, the reduction of the values in the send buffers of processes with ranks *0,...,i* (inclusive).

Attributes

- OpType: the type of the executed operation on the received data at the root process.

Associations

No additional associations.

Constraints

No additional constraints.

Appendix B – SMG2000 Communication Patterns

In the following, we present the point-to-point communication patterns that were detected in SMG2000 in Chapter 5. A process p is represented in the 3D grid shown in Figure 5.3 as follows. $P_{i,j,k}$ where *i* is the x-position and *j* is the y-position and *k* is the z-position in the grid. For example, process P1 is represented as $P_{1,1,1}$ and process P2 is represented as $P_{2,1,1}$ and process P7 is represented as $P_{3,2,1}$ and P10 is represented as $P_{2,3,1}$ and P27 is represented as $P_{3,3,2}$. A process does not communicate with itself.

1. *Pattern 1*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i,j,k\pm1}$, $P_{i,j\pm1,k\pm1}$, $P_{i\pm1,j,k\pm1}$, and $P_{i\pm1,j\pm1,k\pm1}$. For example, Process 7 will send and receive from processes 2, 3, 4, 6, 8, 10, 11, 12, 18, 19, 20, 21, 22, 23, 24, 26, 27, and 28.
2. *Pattern 2*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i,j,k\pm1}$, $P_{i,j\pm1,k\pm1}$, $P_{i\pm1,j,k\pm1}$, $P_{i\pm1,j\pm1,k\pm1}$, and $P_{i\pm1,j\pm2,k\pm1}$. For example, Process 7 will send 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, 32 whereas Process 1 communicates with 2, 3, 5, 6, 7, 9, 10, 11, 17, 18, 19, 21, 22, 23, 25, 26, and 27.
3. *Pattern 3*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i,j,k\pm1}$, $P_{i,j\pm1,k\pm1}$, $P_{i,j\pm2,k\pm1}$, $P_{i\pm1,j,k\pm1}$, $P_{i\pm1,j\pm1,k\pm1}$, $P_{i\pm,j\pm2,k\pm1,}$ $P_{i\pm2,j,k\pm1}$, $P_{i\pm1,j\pm2,k\pm1}$, and $P_{i\pm2,j\pm2,k\pm1}$. For example, Process 7 will send and receive from processes 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32 whereas Process 1 communicates with 2, 3, 5, 6, 7, 9, 10, 11, 17, 18, 19, 21, 22, 23, 25, 26, and 27.
4. *Pattern 4*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i,j,k\pm1}$, $P_{i,j\pm1,k\pm1}$, $P_{i\pm2,j,k\pm1}$ and $P_{i\pm2,j\pm1,k\pm1}$. For example, Process 7 will send and receive from processes 1,

3, 5, 9, 11, 21, 23, 25, and 27 whereas Process 1 communicates with 3, 5, 7, 17, 19, 21, and 23.

5. *Pattern 5*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm1,k}$. For example, Process 7 will send and receive from processes 3 and 11 whereas Process 1 communicates with process 5 only.
6. *Pattern 6*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm2,k}$. For example, process 7 will send to and receive from 5 whereas process 1 will send to and receive from process 3.
7. *Pattern 7*: Each process $P_{i,j,k}$ will send to $P_{i,j-1,k}$ and receive from process $P_{i,j+1,k}$. For example, process 7 will send to 3 and receive from 11 whereas process 1 will receive from 5.
8. *Pattern 8*: Each process $P_{i,j,k}$ will send to $P_{i-1,j,k}$ and receive from process $P_{i+1,j,k}$. For example, process 27 will send to 26 and receive from 28.
9. *Process 9*: Each process $P_{i,j,k}$ will receive from $P_{i-1,j,k}$ and send to process $P_{i+1,j,k}$. For example, process 27 will send to 28 and receive from 26.
10. *Process 10*: Processes $P_{2,j,1}$ and $P_{2,j,2}$ will send to $P_{4,j,1}$ and $P_{4,j,2}$ respectively and processes $P_{3,j,1}$ and $P_{3,j,2}$ will send to $P_{1,j,1}$ and $P_{1,j,2}$ respectively. Therefore, processes (2, 6, 10, 14, 18, 22, 26, and 30) will send to the second direct neighbor to the West on the same grid and processes (3, 7, 11, 15, 19, 23, 27, and 31) will send to the second direct neighbor to the East on the same grid.
11. *Process 11*: Processes $P_{2,j,1}$ and $P_{2,j,2}$ will receive from $P_{4,j,1}$ and $P_{4,j,2}$ respectively and processes $P_{3,j,1}$ and $P_{3,j,2}$ will receive from $P_{1,j,1}$ and $P_{1,j,2}$ respectively. Therefore, processes (2, 6, 10, 14, 18, 22, 26, and 30) will receive from the second direct neighbor

to the West on the same grid and processes (3, 7, 11, 15, 19, 23, 27, and 31) will receive from the second direct neighbor to the East on the same grid.

12. *Pattern 12*: Each process $P_{i,j,k}$ will send to $P_{i-1,j,k}$ and $P_{i+1,j,k}$. For example, process 27 will send to 26 and 28.
13. *Pattern 13*: Each process $P_{i,j,k}$ will send to $P_{i+1,j,k}$ and receive from process $P_{i-1,j,k}$. For example, process 7 will send to 8 and receive from 6.
14. *Pattern 14:* Each process $P_{i,j,k}$ will send to $P_{i,j+1,k}$ and receive from process $P_{i,j-1,k}$. For example, process 7 will send to 11 and receive from 3.
15. *Pattern 15:* Each process $P_{i,j,k}$ will send to and receive from $P_{i\pm1,j,k}$, $P_{i,j\pm1,k}$, $P_{i\pm1,j\pm1,k}$ and receive from process $P_{i,j-1,k}$. For example, process 10 will send to and receive from 5, 6, 7, 9, 11, 13, 14, and 15.
16. *Pattern 16:* Process $P_{i,j,k}$ will send and receive from processes $P_{i,j,k\pm1}$, $P_{i\pm1,j,k\pm1}$, $P_{i\pm2,j,k\pm1}$, $P_{i,j\pm2,k\pm1}$, $P_{i\pm1,j\pm2,k\pm1}$, $P_{i\pm2,j\pm2,k\pm1}$. For example, Process 7 will send and receive from processes 5, 6, 8, 13, 14, 15, 16, 21, 22, 23, 24,29, 30, 31, and 32 whereas Process 1 communicates with 2, 3, 9, 10, 11, 17, 18, 19, 25, 26, and 27.
17. *Pattern 17:* Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j,k\pm1}$, $P_{i\pm2,j,k\pm1}$, $P_{i\pm2,j\pm2,k\pm1}$, $P_{i,j\pm2,k\pm1}$. For example, process 5 will send to and receive from 7, 13, 15, 21, 23, 29, and 31.
18. *Pattern 18:* Process $P_{i,j,k}$ will send to processes $P_{i,j-1,k}$, $P_{i\pm1,j-1,k}$ and will receive from to processes $P_{i,j+1,k}$, $P_{i\pm1,j+1,k}$. For example, process 7 will send to processes 2, 3, and 4 and will receive from processes 10, 11, and 12.

19. *Pattern 19:* Processes $P_{i,2,1}$, $P_{i,4,1}$, $P_{i,2,2}$ and $P_{i,4,2}$ will send one message to $P_{i-1,2,1}$, $P_{i-1,4,1}$, $P_{i-1,2,2}$ and $P_{i-1,4,2}$ respectively. Therefore, processes (5, 6, 7, 8, 13, 14, 15, 16, 21, 22, 23, 24, 29, 30, 31 and 32) will send to the direct North process on their same grid.

20. *Pattern 20:* Process $P_{i,j,k}$ will send to processes $P_{i,j+1,k}$, $P_{i\pm1,j+1,k}$ and will receive from to processes $P_{i,j-1,k}$, $P_{i\pm1,j-1,k}$. For example, process 7 will send to processes 10, 11, and 12 and will receive from processes 2, 3, and 4.

21. *Pattern 21:* Processes $P_{i,2,1}$, $P_{i,4,1}$, $P_{i,2,2}$ and $P_{i,4,2}$ will send one message to $P_{i+1,2,1}$, $P_{i+1,4,1}$, $P_{i+1,2,2}$ and $P_{i+1,4,2}$ respectively. Therefore, processes (5, 6, 7, 8, 13, 14, 15, 16, 21, 22, 23, 24, 29, 30, 31 and 32) will send to the direct South process on their same grid.

22. *Pattern 22:* Process $P_{i,j,k}$ will receive from processes $P_{i,j-1,k\pm1}$, $P_{i\pm1,j-1,k\pm1}$,$P_{i,j+1,k\pm1}$, and $P_{i\pm1,j+1,k\pm1}$. For example, process 8 will receive from 3, 4, 11, 12, 19, 20, 27 and 28.

23. *Pattern 23:* Process $P_{i,j,1}$ will receive from processes $P_{i,j,2}$. For example, process 6 will receive from 22 whereas process 27 will send to process 11 (process 11 will receive from process 27).

24. *Pattern 24:* Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm1,k}$. For example, process 7 will send to and receive from processes 11 and 3.

25. Pattern 25: Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm1,k}$ and $P_{i\pm1,j,k}$. For example, process 6 will send to and receive from 10, 2, 5 and 7.

26. Pattern 26: Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm1,k}$ and $P_{i\pm1,j\ i\pm,k}$. For example, process 7 will send to and receive from 11, 12, 10, 2, 3 and 4.

27. Pattern 27: Process $P_{i,j,k}$ will send to and receive from $P_{i\pm1,j,k}$, $P_{i,j\pm1,k}$, $P_{i\pm1,j\pm1,k}$ and receive from process $P_{i,j-1,k}$ (this is same as PT15 but the order of messages is random).

28. *Pattern 28:* Process $P_{i,j,k}$ will send processes $P_{i,j\pm1,k}$. For example, process 14 will send to 10 only whereas process 7 will send to 11 and 3.

29. *Pattern 29:* Process $P_{i,j,k}$ will receive from processes $P_{i,j\pm1,k}$. For example, process 14 will receive from 10 only whereas process 7 will receive from 11 and 3.

30. *Pattern 30:* Process $P_{i,j,1}$ will send to process $P_{i,j,2}$. Therefore, each process on the first grid (upper) will send to its direct neighbor on the adjacent grid.

31. *Pattern 31:* Process $P_{i,j,k}$ will send and receive from processes $P_{i\pm1,j,k}$, $P_{i,j\pm2,k}$, $P_{i\pm1,j\pm2,k}$, and $P_{i\pm2,j\pm2,k}$. For example, Process P7 sends to and receives from 2, 3, 4, 6, 8, 10, 11, 12, 14, 15, and16 whereas P1 sends to and receives from 2, 5, 6, 9, and 10.

32. *Pattern 32:* Process $P_{i,j,k}$ will send and receive from processes $P_{i\pm1,j,k}$, $P_{i\pm2,j,k}$, $P_{i,j\pm2,k}$, $P_{i\pm1,j\pm2,k}$, and $P_{i\pm2,j\pm2,k}$. For example, Process P7 sends to and receives from 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, and16 whereas P1 sends to and receives from 2, 3 5, 6, 7, 9, 10 and 11.

33. *Pattern 33:* Process $P_{i,j,k}$ will send and receive from processes $P_{i\pm1,j,k}$, $P_{i\pm2,j,k}$, and $P_{i,j\pm2,k,}$. For example, Process P7 sends to and receives from 1, 3, 5, 9, 10, 1113, and 15 whereas P1 sends to and receives from 1, 3 7, 9, and 11.

34. *Pattern 34*: Each process $P_{i,j,k}$ will send to processes $P_{i,j,k\pm1}$, $P_{i,j\pm1,k\pm1}$, $P_{i\pm1,j,k\pm1}$, and $P_{i\pm1,j\pm1,k\pm1}$ and will receive from $P_{i,j,k}$, $P_{i,j\pm1,k}$, $P_{i\pm1,j,k}$, and $P_{i\pm1,j\pm1,k}$.

35. *Pattern 35*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm1,k}$, $P_{i\pm1,j,k}$, $P_{i\pm2,j,k}$, $P_{i\pm1,j\pm2,k}$, and $P_{i\pm2,j\pm2,k}$. For example, process 7 will send to and receive from 5, 6, 8 , 13, 14, 15, and 16.

36. *Pattern 36*: Each process $P_{i,j,k}$ will send to and receive from processes $P_{i\pm2,j,k}$, $P_{i,j\pm2,k}$, and $P_{i\pm2,j\pm2,k}$. For example, process 7 will send to and receive from 5, 13, and 15.

37. *Pattern 37*: Process $P_{i,j,k}$ will receive from processes $P_{i,j+1,k}$, $P_{i\pm1,j+1,k}$, $P_{i,j-1,k}$, $P_{i\pm1,j-1,k}$. For example, process 7 will receive from processes 10, 11, 12, 2, 3, and 4.
38. *Pattern 38*: Process $P_{i,j,1}$ will receive from processes $P_{i,j,2}$, $P_{i\pm1,j,2}$, $P_{i,j\pm1,2}$ and $P_{i\pm1,j\pm1,2}$. For example, process 7 receives from processes 18, 19, 20, 22, 23, 24, 26, 27, and 28.
39. *Pattern 39*: Process $P_{i,j,1}$ will receive from processes $P_{i,j,2}$, $P_{i\pm1,j,2}$, $P_{i,j\pm1,2}$. For example, process 7 receives from processes 19, 22, 23, 24, and 27.
40. *Pattern 40*: Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm1,k}$, and $P_{i\pm1,j,k}$. For example, process 7 will send and receive from 11, 8, and 3. This is similar to pattern 25 with the difference in the order of messages.
41. *Pattern 41*: Process $P_{i,j,1}$ will send to processes $P_{i,j,2}$, $P_{i\pm1,j,2}$, $P_{i,j\pm1,2}$. For example, process 7 receives from processes 19, 22, 23, 24, and 27.
42. *Pattern 42*: Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j,k\pm1}$.
43. *Pattern 43*: Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j,k\pm1}$, $P_{i,j\pm1,k}$ and $P_{i\pm1,j,k}$.
44. *Pattern 44*: Process $P_{i,j,k}$ will send to and receive from processes $P_{i,j\pm1,k}$, $P_{i\pm1,j,k}$, $P_{i,j,k\pm1}$, $P_{i,j\pm1,k\pm1}$ and $P_{i\pm1,j,k\pm1}$.