An Exchange Format for Representing Dynamic Information Generated from High Performance Computing Applications

Luay Alawneh and Abdelwahab Hamou-Lhadj
Software Behaviour Analysis Lab
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montréal, Québec, Canada
{l_alawne, abdelw}@ece.concordia.ca

Abstract

High Performance Computing (HPC) systems tend to be complex to debug and analyze due to the large number of processes they involve and the way they communicate with each other to perform specific tasks. Recently, there has been an increase in the number of tools to help software engineers analyze the behavior 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. In this paper, 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 (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, 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 an MTF toolkit that supports the generation of MTF traces equipped with a query engine to facilitate the retrieval of data from MTF traces. Finally, we show how MTF can carry a large trace generated using a commercial off the shelf MPI trace analysis tool.

Keywords: High Performance Computing Systems, Inter-Process Communication Traces, Message Passing Interface, Standard Exchange Format

1. Introduction

High Performance Computing (HPC) systems such as the ones used in grid computing have been shown to be useful in a variety of domains ranging from solving computation-intensive scientific problems to powerful back-office data processing units used in large enterprise applications (e.g. [Stamatakis 05, Aloision 02, Ranjan 08]). These applications take advantage of multiple processes running on autonomous computers that communicate through a computer network to achieve a common goal. Although the benefits of HPC applications are numerous, they tend to be difficult to debug and analyze, causing significant delays in production and maintenance time [Becker 2007]. This is mainly due to the large number of inter-communicating processes they involve. To address this issue, several techniques and supporting tools have been proposed (e.g. [TAU, Hong 1996, Heath 2003]). These tools provide many features that enable software engineers to examine the run-time behavior of these applications for performance analysis, debugging, deadlock detection, etc. Although these tools have common features, each of them has its own advantages and specialized functions. Currently the only way to take full advantage of the functions they provide is to convert the data generated from HPC systems from one format to another. This is due to the fact that they use different formats for representing HPC traces, which hinders interoperability and sharing of data. Writing converters to and from all available formats is usually a tedious task, which is in many times impractical. What is needed is a standard exchange format for representing run-time information generated from HPC systems that can be readily used by different tool vendors. There are many other advantages for having a standard exchange format such as:

- Reducing the effort required to represent HPC traces.
- Allowing researchers to use different tools on the same input, which can help compare the techniques supported by each tool.
- Enabling software engineers to combine the techniques from different tools without having to worry about how the data is represented.

In this paper, we present an exchange format, called MTF (MPI Trace Format) that we have developed to represent run-time information generated from HPC applications. We believe that MTF can lead the work towards a standard exchange format for representing and sharing information generated from HPC systems. The focus is on modeling inter-process communication traces based on message passing, which is perhaps the most common communication paradigm used in most of today's high performance computing distributed systems. In particular, we target systems built using the Message Passing Interface (MPI) framework [MPI], which is the de facto standard for inter-process communication in HPC parallel applications running on distributed systems.

Organization of the paper: In Section 2, we present the related work and the background on MPI traces. In Section 3, we present the requirements that guided the design of MTF. Section 4 presents MTF along with the semantics of its components. In Section 5, we discuss MTF tool support. In Section 6, we show how MTF meets some key requirements for a standard exchange format. We present a case study where we applied MTF to represent an MPI trace file generated by VampirTrace [VampirTrace], a library for the generation of MPI traces in Section 7. We conclude the paper in Section 8.

2. Background and Related Work

In the section, we first provide necessary background on the MPI framework that is needed to understand the content of this paper including the point-to-point and collective mechanisms defined in MPI for inter-process communication. We also discuss related work by surveying existing MPI trace formats found in the literature.

2.1 The Message Passing Interface

A parallel program is composed of several processes running on different processors that need to collaborate in order to execute a specific set of tasks. Usually, processes cannot have direct access to each others' address space. Therefore, a mean for communication is needed in order to enable information exchange among the program's processes. The message passing mechanism, which consists of having processes communicate through the exchange of messages, has been adopted as the mechanism of choice for inter-process communication. It is based on the transfer of a message from one process to another by sending a copy of the message from the sending process to the receiving one which in turn expects an incoming message at a predefined location in its memory space.

In 1992, an effort for standardizing the operations that can be used by processes to exchange messages has started as a result of a meeting held at the workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia. In 1994, the first version of the MPI (Message Passing Interface) specification [MPI] was released. The work continued on improving the standard until 1997 where the second version known as MPI 2.0 [MPI] was released, which is the version currently in effect.

The main advantages that distinguish MPI from other message passing paradigms are its support for asynchronous communication, process group context, and process synchronization. Another important advantage is its portability since all existing implementations on different platforms are based on the same open accepted standard.

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

MPI supports blocking and non-blocking modes of point-to-point communication. In the blocking mode, the process must wait for the completion of the posted operation. Thus, the process cannot resume any other computations until a confirmation is received that a matching operation has been posted. On the other hand, non-blocking operations return immediately in order to allow the process to continue with its computations which demonstrates the advantage of asynchronous communication. The non-blocking mode requests the MPI library to perform, when possible, the posted operation. A process can use ‘wait’ and ‘test’ MPI routines to wait or check for the completion of the non-blocking operation respectively.

MPI collective communication defines different types of operations for exchanging information among a group of processes defined as an MPI communicator. MPI assumes that all processes, in a communicator, must execute the same collective operations in the same order. In order to guarantee the synchronization among all processes in the communicator, MPI recommends the usage of the ‘barrier’ operation.

It is worth mentioning that MPI collective operations are based on point-to-point operations. However, the communication mode in a collective communication must be blocking in order to enforce the execution of the same collective operation by all processes synchronously. Moreover, collective operations do not use tags as message identifier in order to strictly force the exchange of messages according to their order of execution. All processes must post collective operations that exactly match the size and data type of the exchanged data.

Furthermore, collective communication involves operations that have one process sending data to the other processes, one process receiving data from all other processes or all processes sending data to all other processes in the communicator. Another type of collective operations is the reduction operations which involve collecting data from the other processes in the communicator and then performing a predefined operation on the received data.

2.2 Existing Trace Formats

Recording run-time information requires a trace format that is expressive, scalable, portable, and extensible. Moreover, it should provide efficient ways for accessing the trace information through an effective query language. In the following, we surveyed execution trace formats for MPI traces and other message passing environments. Most of these formats, however, are either proprietary or built without taking into consideration the aforementioned requirements for an exchange format. The list of the studies included in this survey is by no means exhaustive but we believe that it is representative of the state of the art.

2.2.1 Pablo Self-Defining Data Format (SDDF)

SDDF is perhaps one of the leading trace formats that have been used for performance analysis of distributed applications [Aydt 1994]. It is a general-purpose language that can be considered as 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 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.

SDDF is designed to provide trace formats in both binary and ASCII representations. The reason behind this is that the binary representation can be used when compactness is sought. On the other hand, the ASCII representation is used when portability and readability are needed. 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.

2.2.2 Pajé Trace Format

Pajé trace format is a generic trace format that provides the ability to define the structure of the traces based on the targeted problem [Kergommeaux 2003]. 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. When the trace file is generated, the event unique numbers are replaced with the event name from the event definition which is used to ignore the event definitions afterwards in order to process the trace file using the event names. Though 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.

2.2.3 EPILOG Trace Format

EPILOG (Event Processing, Investigating, and Logging) is a binary data format for capturing traces of MPI and OpenMP (a paradigm for shared memory programming) applications [Wolf 2004]. An EPILOG trace file consists of a header preceding the trace records. This header contains information related to the EPILOG file such as the EPILOG version number. 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, it is limited in many ways including the fact that it provides a binary trace format that hinders portability of the trace format on different platforms.

2.2.4 Structured Trace Format (STF)

The main idea behind the Structured Trace Format (STF) is to handle traces generated from large applications using several physical files [STF]. The intention is to properly handle 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, the declaration file, the event data file and the statistics file. The index file is used to locate the other STF files. The declaration file defines the record formats of the traced units such as method 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.

2.2.5 Open Trace Format (OTF)

OTF is a trace format that uses different streams (files) to represent trace data for HPC parallel applications [Knüpfer 2006]. A stream may contain traces corresponding to one or more process. 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 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, similar to other trace formats, does not provide all the information that can be traced from MPI applications that are needed, for example for debugging purposes, such as the data types and the memory addresses of the exchanged data.

3. Requirements for the Design of MTF

In this section, we present the requirements that we used to guide the design of MTF. These requirements are based on known requirements for developing a standard exchange which are described in [Bowman 1999, Lethbridge 1997, St-Denis 2000, Woods 1999]. The validation of MTF against these requirements is presented in Section 4.

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 was 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.

3.2 Scalability

An exchange format should be scalable to support a large amount of information efficiently in a way that does not degrade access to its 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.

3.3 Simplicity

This requirement for an exchange format dictates the need for a trace format specification that is easy to understand so as to facilitate its adoption by tool vendors. Also, simplicity requires clean and complete documentation of the design of the exchange format.

3.4 Transparency

Transparency ensures that the information is represented without any alteration. Therefore, we need to provide well-defined mechanisms in order to generate traces in the form of MTF.

3.5 Neutrality

Neutrality refers to an exchange format that is not specific to a particular language or platform.

3.6 Extensibility

Extensibility is an important requirement when building an exchange format. Exchange formats should be easily extended in order to support new or different data types.

3.7 Completeness

Completeness mandates that an exchange format should include the necessary information during the exchange process. An exchange format that satisfies this requirement should provide the data as well as the structure (i.e. the metamodel, ‘known as the schema’) which can be used to interpret the carried data. This enables tools to validate the carried data with regard to the provided metamodel.

3.8 Solution Reuse

It is important to build an exchange format that reuses some existing technologies to avoid reinventing the wheel. For example, an exchange format can be carried using an existing data carrier language such as XML instead of creating a new one.

3.9 Popularity

In order to meet the popularity requirement (i.e. acceptance by several users), an exchange format needs to meet the previously mentioned requirements. Moreover, it should be delivered with an API that will allow tool vendors to generate and query traces.

4. MTF Components

In this section, we present the MTF exchange format. The definition of an exchange format involves two main components [Bowman 1999]: A metamodel (also called a schema) that describes 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.

4.1 MTF Metamodel

Figure 1 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.

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

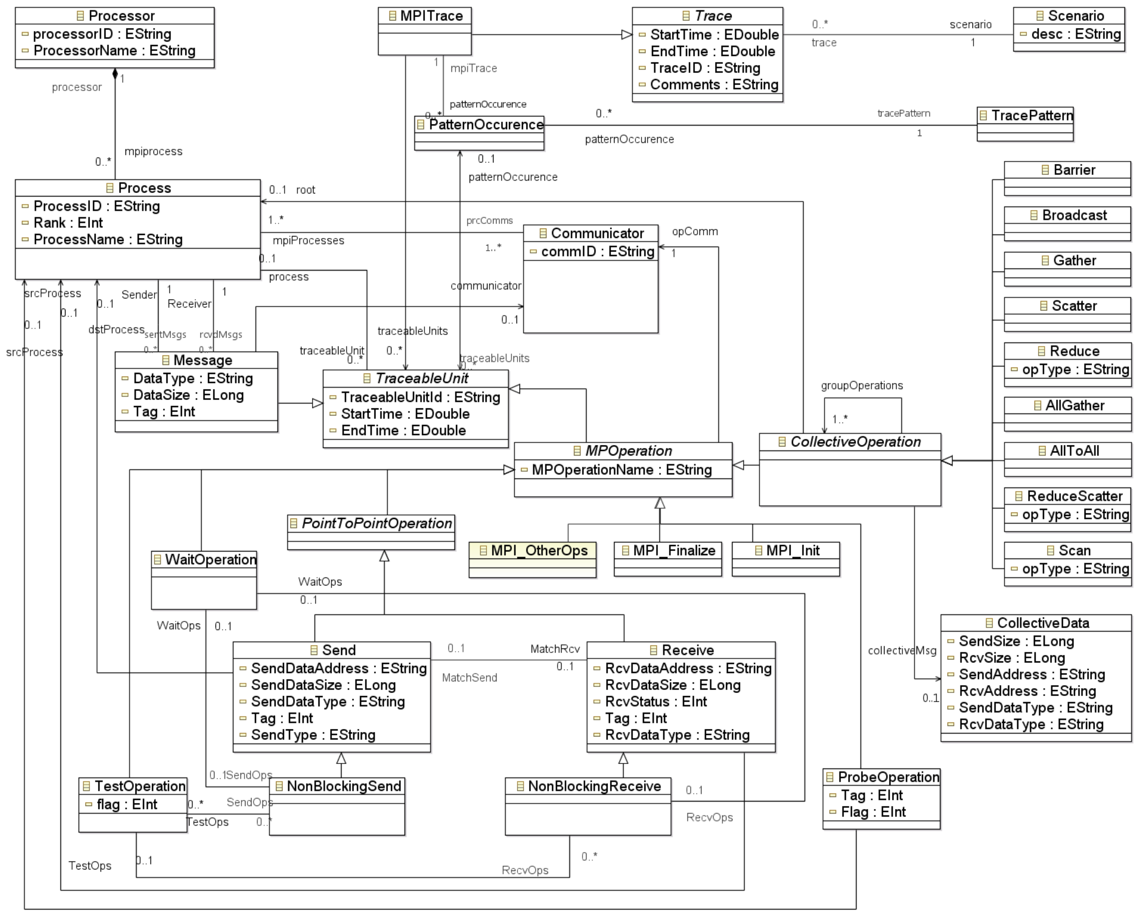

Figure 1. The MTF metamodel

4.1.1 Usage Scenario

An execution trace is obtained by executing a usage scenario described in the Scenario class. Our metamodel accepts that a scenario can be represented by several execution traces in order to support situations where different traces might be needed to understand a particular scenario. Traces for the same scenario can, for example, be used to detect anomalies caused by non-deterministic behavior of MPI applications.

4.1.2 Trace Types

The abstract 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 MPITrace class to represent traces of MPI applications.

4.1.3 Processor and Process

The Processor and Process classes are used to capture the process and the machine (including the node) a process is running on.

4.1.4 Traceable Unit

A traceable unit (modeled using the TraceableUnit class) is used to represent any event contained in traces of an MPI system such as the MPI operations, routine calls, program statements, messages, and any other type of a traceable unit in the program. Although our focus is on modeling MPI operations, the TraceableUnit abstract class is added to enable our metamodel to be extended to capture other types of dynamic information.

4.1.5 MPI Operations

The MPI operations are represented using the class MPIOperation, which is a base class to many other MPI operations including MPI_Finilize, MPI_Init, a probe, a wait, a test, point-to-point operations, and collective operations. The Point-to-point operations are further specialized into specific operations modeling blocking send and receive operations (represented using the classes Send and Receive), non-blocking send and receive operations (classes NonBlockingSend and NonBlockingReceive). Similarly, the specific collective operations such as a barrier, broadcast, etc. are represented using classes that inherit from the CollectiveOperation class. Collective operations are only blocking operations and involve all the processes in the application. The attributes needed for each of these operations are also modeled although some of them are not shown in the diagram of Figure 1 to avoid cluttering the diagrams.

4.1.6 Message

The class Message is used to capture the messages exchanged using point-to-point operations only. Data exchanged using collective operations can be detected from the collective call for each process. Since point-to-point operations allow the receiving process to post a receive operation that does not match exactly the incoming message, the information in the MPI_Recv call cannot be used to refer to the received message. Therefore, the message needs to be checked in order to determine the sending process as well as the information regarding data such as size, data type and receiving buffer address.

4.1.7 Collective Data

Collective data (modeled using the class CollectiveData) represent the information about the data being exchanged by each process when executing a collective MPI operation. MPI requires that all the processes post the same data type and size when executing a collective MPI call.

4.1.8 Trace Patterns

We also modeled trace patterns, which are defined as sequences of events that are repeated non-contiguously in a trace. This is based on the work of Hamou-Lhadj et al. [Hamou-Lhadj 04], where the authors proposed an exchange format for representing traces of routine calls in which trace patterns are modeled as separate entities. According to the authors, the analysis of trace patterns might reveal important information about the behavior of the system. Similarly, we propose that MPI trace patterns might be needed to understand various aspects of an MPI system. We therefore provide support for it in our metamodel using the classes TracePattern and PatternOccurrence (which represents a single occurrence of a given pattern). The analysis of MPI traces using trace patterns is out of the scope of this paper.

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 2000], TA (Tuple Attributes language) [Holt 1998], 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. In this paper, 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 2000]. It supersedes a number of pre-existing syntactic forms for exchanging software artefacts such as GraX [Ebert 1999], TA [Holt 1998], and RSF [Müller 1988]. Figure 2 shows an example using GXL to represent an MPI trace which is used in the case study of this paper to show the effectiveness of MTF to capture MPI traces.

```
<gxl>

<graph>
<node id = “scen001”>
<attr name = “description”>
<string> Test of  Weather Research and Forecasting Model
code</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 2. An example of an MPI trace captured with MTF and carried by GXL

An XML-based language, however, tends to be very verbose due to the excessive use of XML tags. This may cause scalability issues when applied to carry MTF traces since traces, in general, tend to be excessively large. A possible alternative is to explore non XML formats such as TA. These formats, however, are not widely accepted which goes against some of the above requirements. The decision on which syntactic form should be used with MTF is a subject of future studies.

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 shows the architecture of the tool, which consists of four main components, which are 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 paper.
- MTF Trace Importer and the MTF Trace Exporter are two modules used to convert the MTF traces from and to other trace formats respectively.

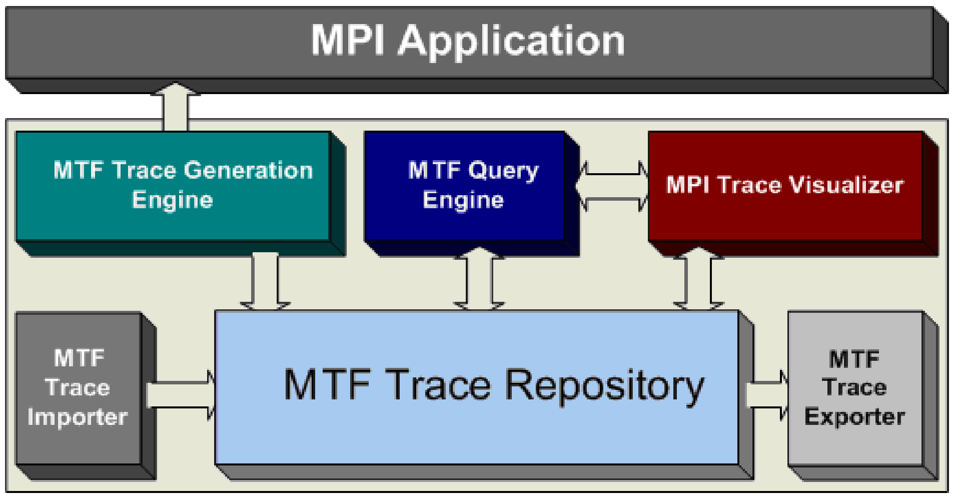

Figure 3 The MTF Tool Architecture

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.

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 1. 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 ($p_a, p_c, p_m, \ldots, p_n$)	Traces related to a selected number of processes.
Processes in Communicator c_1	All processes in an MPI communicator.

Table 1 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.

5.2.1 Point-to-Point-Related Queries

Point-to-point related queries retrieve information that pertains to MPI point-to-point operations. Table 2 shows the information that the queries supported by our tool are capable of retrieving for point-to-point processes.

Table 2 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 specific set of processes where size of data sent/received is less than, equal, or greater than $size_n$.

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 shows the collective queries supported by our tool.

Table 3. 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, or greater than $size_n$.

5.2.3 Message-Related Queries

Message-related queries target traces of messages exchanged in point-to-point operations. Table 4 shows the main queries used to retrieve information related to messages transferred using point-to-point operations.

Table 4. 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, or greater than $size_n$.

Figure 4 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 4. Simple Query Examples

This query language can also be used to compute statistical information such as the time a process was involved in MPI communications, 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 M-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.

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

Process M-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.

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

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 use the MPI standard Profiling Interface (PMPI) [MPI], for the instrumentation of the various MPI operations in the program.

6. 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 5 summarizes the evaluation of MTF with respect of each requirement. As shown in Table 5, 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 meramodel by reconstructing, on the fly, the metamodel from the MTF file. The design of MTF also favors 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. We deliberately made use of abstract classes to facilitate the creation of specialized classes that would represent other types of data not captured by MTF (e.g., functions used in other types of inter-process communication platforms than MPI).

However, we recognize that MTF requires further improvements to meet the scalability requirement. As it isn MTF will require modeling every event of an MPI trace and does support any compaction scheme. A possible solution to this is to improve the MTF model by representing non-contiguous repeated events only once. This will require investigating ways to transform MPI event streams into a structure in which similar sequences are represented only once.

Table 5. 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 recommend using a compact syntactic form to carry MTF traces. This, however, will not guarantee an exchange format that is scalable enough to carry trace files of the size of gigabytes. We are still working on improving the metamodel by investigating ways to represent the information that is duplicated in an MPI trace only once. One direction is to adopt the techniques presented in [Reiss 01] which are inspired from data compression techniques in information theory to reduce the amount of information found in traces.
Simplicity	We believe that the MTF metamodel is simple to understand since it maps well to MPI operations. In Appendix A, the detailed specification of MTF is provided.
Transparency	We have developed an API for generating and querying MTF-based MPI traces as shown in Section 5.
Neutrality	Currently, MTF supports only MPI generated traces. MPI is implemented as a middleware to allow applications that run on MPI to be neutral to the specific technology platform. However, MTF only supports MPI traces and does not support other message passing frameworks.
Extensibility	MTF can be extended in many ways to support new types of traces by

	extending the Trace and the TraceableUnit classes.
Completeness	We address this requirement by recommending a syntactic form (e.g. GXL) that supports the exchange of MTF metamodel as well as the instance data.
Solution Reuse	In this work, we suggest reusing an existing syntactic form such as GXL. We also built the MPI query language as an Eclipse plug-in using the EMF (Eclipse Modeling Framework) capabilities. Additionally, we used the MPI profiling functions in building our API for trace generation.
Popularity	We believe that the need for a standard exchange format for MPI traces is an important issue to which we have provided a complete solution. Therefore, we believe that MTF can lead towards the work of standardizing MPI traces.

7. Case Study

In order to show the ability for MTF to represent MPI traces generated from large systems, we tested it using a sample trace generated by the VampirTrace [VampirTrace] trace analysis tool. These traces are provided on the Vampir tool website [Vampir]. They have been generated from the Weather Research and Forecasting Model system running on HPC-System SGI Altix 4700, which is composed of 1024 dual-core Intel Itanium processors and has 6.5 TB main memory. The trace file format generated by the VampirTrace is called the Open Trace Format (OTF) which comes with several APIs for reading the trace data. However, we built our own code, which is a proprietary format. OTF files are compressed using the zlib [Gailly 2002] data compression format, which requires the use of special libraries (e.g. OTFDUMP library [Knüpfer 2006]) to convert the content of the file into a human readable version.

The size of the original OTF compressed file was 4.12 MB. However, the size of the uncompressed file increased dramatically to 79.4 MB. This file also includes traces of non-MPI routines. The size of the MPI trace extracted from the file was 39.4 MB, which is almost half the size of the complete trace file. We converted the OTF file into MTF using GXL as the syntactic form. The size of the resulting GXL trace file was 46.2 MB, which is larger than the size of the MPI traces in the OTF file. This is expected since OTF is not based on GXL. The increase in the size was due to the XML-like syntax added to the trace data which accounts for almost 17% more than the original trace data. This shows that GXL may not be the best carrier for MTF and that a non-XML language should be considered. The MTF trace was fed to our trace analysis tool. We have noticed some scalability problems due mainly to the fact that the tool does not support any optimization technique of memory in its actual state. It is provided in this paper as a proof of concept and we intend to continue working on improving it in the future.

Table 6. MPI Trace Statistics

	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

In Table 6, 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. Moreover, when querying the point-to-point operations, we were able to identify the MPI virtual topology used in the program. Every process communicates with its neighbors in the grid (west, east, south and north). For example, Process 7 communicates with Processes 3, 6, 8 and 11. However, Process 1 only communicates with Processes 2 and 5 since it does not have an east and a north neighboring processes. Detecting the MPI virtual topology helps in identifying which processes to include in the study of the behavior of inter-process communication. 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.

8. Conclusion and Future Directions

In this paper, 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, which we believe can facilitate its adoption. 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 built an MTF toolkit to allow users to generate and query MTF traces. Finally, we showed how MTF can represent a large trace generated from a commercial MPI trace analysis tool.

An immediate future direction is to continue to use MTF to represent traces and study ways of optimizing it so it could handle extremely large traces. In addition, we need to work on defining a compact but yet expressive data carrier that can be used with MTF. Finally, we need to create converters that would convert the formats used by other tools into MTF to encourage tool vendors to adopt it.

Acknowledgment

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[Aydt 1994] Ruth 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/.

[Becker 2007] 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, 2007.

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

[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,* pp. 89–98, 1999.

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

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

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

[Hamou-Lhadj 2004] 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,* pp. 59-69, 2004.

[Heath 2003] 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/.

[Hong 1996] C-Eui Hong, B-Sik Lee, G-W. On, D-H. Chi, "Replay for Debugging MPI Parallel Programs", *In Proc. of the Second MPI Developers Conference,* pp. 156-160, 1996.

[Holt 1998] R. C. Holt, "An Introduction to TA: The Tuple Attribute Language", http://swag.uwaterloo.ca/pbs/papers/ta.html

[Holt 2000] 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,* pp. 162-171, 2000.

[Lethbridge 1997] 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.

[Knüpfer 2006] 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),* pp. 526-533, 2006.

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

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

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

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

[St-Denis 2000] 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*, 2000.

[TAU] TAU User's Guide.
URL : http://www.cs.uoregon.edu/research/tau/docs/newguide/.

[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.

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

[Woods 1999] 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, pp. 391–398, 1999.

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

[Xue 2009] 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*, pp. 251-260, 2009.

[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), pp.725-730, 2005.

[Aloision 02] G. Aloisio, D. Talia, Guest editorial: grid computing: Towards a new computing infrastructure, Future Generation Computer System 18 (8) 2002.

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

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

An abstract 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.

Constraints

[1] StartTime and EndTime should be different

```
self.EndTime >= self.StartTime
```

3) MPITrace

Semantics

An object of the MPITrace represents a trace generated from MPI operations. This class inherits from the abstract Trace class.

Attributes

No additional attributes.

Associations

- PatternOccurrence [*]: References the occurrences of the execution patterns that are invoked in the trace.
- TraceableUnit [0..*]: A reference to all instances of TraceableUnit class that are of types MPOperation and Message.

Constraints

- MPITrace only references objects created from MPOperation and Message classes.

4) TracePattern

Semantics

An object of the class TracePattern represents a sequence of message passing operations that is repeated in a non-contiguous manner 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 (i.e. MPITrace object that contains the pattern occurrences).

5) 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.
- MPITrace [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.

6) 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. For example, a Method Call is a type of TraceableUnit. Thus, a metamodel for capturing method call traces, ex. Compact Trace Format (CTF) [Hamou-Lhadj 2004], can inherit from 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.

7) **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.

8) **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*.

9) **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.

10) **MPOperation**

Semantics

This abstract 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 TraceableUnit class.

Attributes

– MPOperationName: The name of the MPI operation.

Associations

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

Constraints

No additional constraints.

11) MPI_Init

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.

12) MPI_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.

13) MPI_OtherOps

Semantics

This class represents all the other MPI operations that do not have a concrete class defined specifically to capture their traces.

Attributes

No additional attributes.

Associations

No more attributes associations than the ones assigned to its parent classes.

Constraints

No additional constraints.

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

- Sender [1]: the sending process.
- Receiver [1]: the receiving process.

Constraints

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

15) **PointToPointOperation**

Semantics

This abstract 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.

16) **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.

17) **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.

18) **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.

19) **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.

20) **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.

21) 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.

22) 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.

23) **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.

24) **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.

Constraints

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

25) **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*.

26) **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.

27) **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.

28) **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.

29) **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.

30) **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.

31) **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.

32) **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.

33) **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.