Towards a Common Metamodel for Traces of High Performance Computing Systems to Enable Software Analysis Tasks

Luay Alawneh
Department of Software Engineering
Jordan University of Science & Technology
Irbid, Jordan
lmalawneh@just.edu.jo

Abdelwahab Hamou-Lhadj
SBA Research Lab
Electrical and Computer Engineering
Concordia University
Montreal, QC, Canada
abdelw@ece.concordia.ca

Jameleddine Hassine
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
jhassine@kfupm.edu.sa

***Abstract*—There exist several tools for analyzing traces generated from HPC (High Performance Computing) applications, used by software engineers for debugging and other maintenance tasks. These tools, however, use different formats to represent HPC traces, which hinders interoperability and data exchange. At the present time, there is no standard metamodel that represents HPC trace concepts and their relations. In this paper, we argue that the lack of a common metamodel is a serious impediment for effective analysis for this class of software systems. We aim to fill this void by presenting MTF2 (MPI Trace Format2)—a metamodel for representing HPC system traces. MTF2 is built with expressiveness and scalability in mind. Scalability, an important requirement when working with large traces, is achieved by adopting graph theory concepts to compact large traces. We show through a case study that a trace represented in MTF2 can be in average 49% smaller than a trace represented in a format that does not consider compaction.**

Keyword*—High performance computing systems, inter-process communication traces, metamodeling, HPC trace analysis.

I. Introduction

HPC systems are used in a variety of application domains that require high processing capacity – particularly speed of calculation. With the emergence of multi-core systems and cloud computing, HPC systems are expected to be more commonly used. This is accelerated by the need to process large data, hence contributing to solving the infamous Big Data problem.

Typical HPC systems are designed where multiple processes interact with each other to solve a specific computational problem. These processes follow communication patterns and are arranged through process topologies [30]. A topology varies depending on the available resources. A powerful supercomputer can be configured to run hundreds if not thousands of processes at the same time. Speed and performance are two apparent benefits of HPC, but this comes with a price. It is often challenging for developers to understand and analyze these systems [11]. This is mainly caused by the inherent complexity of inter-process communication, further complicated by the overwhelming size of run-time data (traces) used to depict this communication.

Fortunately, there exist a good set of HPC analysis tools to overcome these challenges (e.g., [9, 18]). They offer plethora of features that facilitate maintenance tasks. Examples of features include trace abstraction algorithms, search capabilities, various visualization methods, and so on. These tools, however, use different formats to represent traces, which hinders interoperability and sharing of data. To take advantage of their many features, we need to create data converters. This task is not only tedious but also time consuming and unproductive. Besides, not all existing formats are publicly available. Clearly, there is a need to start working towards a common (and open) trace model to enable synergy among HPC tools. This way, software engineers can focus on the analysis itself and not on how the data is represented.

Our effort towards standardizing traces of HPC systems dates back to 2009 when we first presented a UML-based HPC trace model [2], called MTF (MPI[1] Trace Format). At the time, it was a mere flat format that modeled sequences of MPI call operations (Send, Receive, etc.). It was only applicable to small traces because it did not support any compaction mechanism. Furthermore, it did not have support for other components such as user-defined routines and communication patterns, much needed for the analysis and performance debugging of HPC systems (see [4]). In fact, MTF was an experimental attempt to understand the complexity of the HPC trace domain (which was substantially more than what we anticipated). We used this knowledge to build the present (and major) revision of MTF, MTF2.

MTF2 is built with expressiveness and scalability in mind. Expressiveness is achieved by providing support to a wide spectrum of concepts of the HPC trace domain (see Section III.B for more details). These concepts include additional MPI

[1]MPI stands for Message Passing Interface, a standard used for inter-process communication.

operations, user-defined functions, communication patterns, and process topologies. The objective is to enable the use of MTF2 in a broad range of applications with the hope to facilitate its adoption.

Scalability, an important requirement for a trace model because of the large size of typical traces, is achieved by adopting a mechanism for compacting HPC traces without loss of information. We showed in an early achievement paper paper that traces of HPC systems can be made smaller using graph theory concepts [1]. Early experiments involving a small system showed promising results. This paper builds on earlier work by providing the MTF2 model (abstract syntax, design guidelines, MTF2 expressiveness), improving the compaction mechanism, and experimenting with larger traces generated from three systems.

Note that we distinguish between the concepts of *compaction* and *compression*. A compressed file needs to be uncompressed before processing. A compacted file will never need to be 'uncompacted'. In other words, we change the way the original data is represented. For example, the rooted tree in Figure 1a can be represented as an ordered directed acyclic graph (DAG) as shown in Figure 1b by capturing common subtrees only once. In this case, we say that the graph is a *compact* representation of the tree. We can always retrieve the original tree from the ordered DAG. That is, the compaction is lossless. Compression, as defined in Information Theory [19], can be further applied to the compact form (or the original trace) to save disk space.

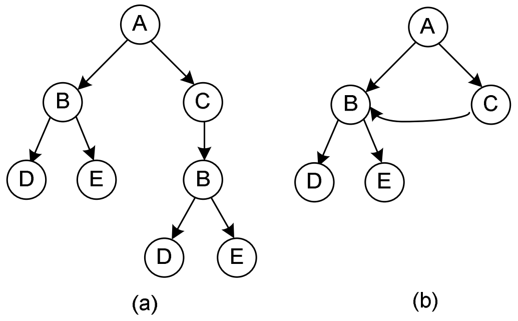

Fig. 1. Turning a rooted tree into an ordered DAG

We applied MTF2 to traces of millions of events, generated from three industrial systems. We obtained a compact model that is in average 49% smaller than a model generated from a trace format that does not support any compaction. Moreover, MTF2 specification is openly available. The meatamodel is represented as an Ecore model developed using Eclipse. We have also developed a query language and an API that can be readily used to extract information from MTF2 traces. In sum, we believe that MTF2 supports key features that can inspire the design of a standard metamodel for representing and sharing information generated from HPC systems.

Moreover, we hope that MTF2 contributes to advancing research in software maintenance of HPC systems, which, despite their importance, have yet to receive the attention they deserve from the software maintenance research community compared to traditional systems.

The rest of the paper is organized as follows: In Section II, we present related work. In Section III, we present MTF2. In Section IV, we present empirical results that show the effectiveness of MTF2 to support large traces generated from different systems. Threats to validity are presented in Section V, followed by a conclusion.

II. Review of Existing HPC Trace Formats

We surveyed trace formats used for traces generated from HPC applications that use the message passing paradigm for inter-process communication.

The Paraver[2] Trace Format 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. TAU [25] trace format uses a binary encoding for trace events. 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. Unlike MTF2, Paraver does not use any compaction technique. The traces need to be uncompressed before processing.

The Open Trace Format (OTF) uses different streams (files) to represent trace data for HPC applications [13]. A stream corresponds to one process in the program. 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. Just like Paraver, OTF does not compact the traces. In other words, a tool that uses OTF needs to uncompress the data ending with the same amount of data as in the original trace. In other words, OTF compresses the data, but does not compact it.

Noeth et al. [18] presented ScalaTrace that provides a compressed trace format for HPC 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-

[2] http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview

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, 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. This study only provided compression of MPI events in the program and did not take into account other kinds of information such as user-defined routines.

Knüpfer et al. [14] proposed the usage of compressed Complete Call Graphs (cCCG) in order to represent traces of single and parallel process programs. cCCG is an approach rather than an exchange format. Knupfer et al. do not look for identical subtrees. They search for compatible trees by comparing only the subtrees' top nodes, assuming that if all the references of the child nodes of the two compared root nodes are pointing to the same subtree then the two subtrees are considered to be compatible. Though interesting, using such compression techniques will result in a lossy model.

Representing routine call trees as a directed acyclic graph was previously proposed by other authors such as Reiss et al. [22], Larus et al. [15], and Hamou-Lhadj et al. in [10]. Hamou-Lhadj et al. have even proposed to use this technique to create a metamodel called CTF (Compact Trace Format) for object-oriented systems [10]. The focus of CTF, however, is on single-threaded system and the mapping between trace information and standard metamodels for representing static information such as KDM (Knowledge Discovery Metamodel) [12]. Merging MTF2 and CTF to provide a full-fledged model for dynamic analysis (i.e., that can support various programming paradigms) is subject of future work.

III. MTF2 (MPI TRACE FORMAT2)

In this section, we present the MTF2 model. We start by discussing the principles that guided the design of MTF2. The domain of our model is then described with a particular focus on the compaction scheme we used to reduce the number of model elements.

A. Guiding Principles

To build MTF2, we followed known guidelines for designing standard exchange formats such as the ones described in [23]. An exchange format is defined using a metamodel and a data carrier. Since in this paper, we only focus on the metamodel, we use the terms metamodel and exchange format interchangeably. The data carrier is not dealt with in this paper. We adhere to the idea that a data carrier should be defined independently from the metamodel.

We focus, in this paper, on four requirements that we believe should be given priority: expressiveness, scalability, openness, and transparency. These requirements are also verifiable. The other requirements described in [23] such as extensibility, simplicity, neutrality are also important but hard to verify due to their subjective nature. Nevertheless, we intend in future work to address these requirements as well.

Expressiveness: We define expressiveness as the ability for a model to support a rich set of information that is needed by the analysis tools. We studied the HPC domain as well as the MPI specification very carefully to understand the HPC trace domain. We also worked with HPC analysis tools such as Vampir[3] and JumpShot[4] [17] to understand the features they support and the data they use. We identified five categories of data that must be supported by MTF2 to be expressive: (1) MPI operations with their arguments, (2) user-defined functions, (3) trace communication patterns and topologies, (4) process and processor information, and (5) usage scenario information.

Scalability: A trace metamodel must support extremely large traces. This is because typical (and most interesting) traces tend to be considerably large (Giga bytes of data). MTF2 achieves scalability by supporting a trace compaction framework based on graph transformations. This is further discussed in the next section.

Openness: This requirement necessitates that the metamodel be publicly available. This also opens the door for further improvements to the model or possibilities to customize it to specific needs. MTF2 specifications are open. The model is built using Eclipse EMF[5]. A website[6] is made publicly available from which MTF2 specifications and the accompanying tools and traces can be downloaded.

Transparency: This requirement refers to the ability for a metamodel to represent data without alteration. As we will see in the next section, MTF2 is a lossless exchange format. However, we may argue that, in the context of execution traces, we might want to sacrifice some data for better compaction. For example, in the design of ScalaTrace, Noeth et al. [18] keep track of timestamp intervals instead of every timestamp associated with each event. Knowing the interval might be sufficient to perform some debugging or performance analysis tasks. In our design, we take a different approach. We provide support for a lossless representation at all time, but leave it up to the user to modify the format as needed, as long as the changes do not violate MTF2 metamodel and constraints.

B. The MTF2 Domain

An HPC 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 [30] in which the program tasks are run in parallel on multiple processors to maximize performance.

[3]http://www.vampir.eu

[4]http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/

[5]http://www.eclipse.org/modeling/emf/

[6]http://www.encs.concordia.ca/~abdelw/sba/mtf

Communication among processes is based on executing MPI operations, which can be grouped into two categories: 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 provide detailed description of the various MPI operations [16]. In its current state, the MTF2 metamodel supports the most widely used collective operations such asBarrier, AlltoAll, etc. The metamodel can be extended with other operations if needed.

An HPC 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, tag value, communicator, size of sent data, and the address of send buffer.

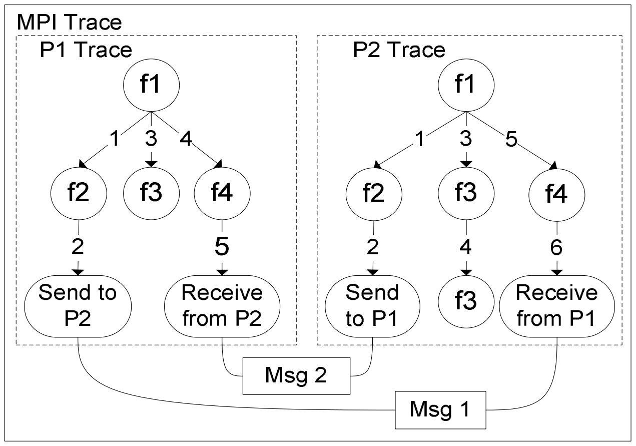

Fig. 2. HPC Trace Representation

Figure 2 shows an example of two processes that execute in parallel four user-defined 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 shown as typical Send and Receive MPI operations along with the exchanged messages. The message object is created by merging the atomic sent-message and received-message events on the sender and receiver respectively.

HPC programs in particular are designed to follow specific communication patterns that characterize their process communication topology [30]. Examples of such patterns include the butterfly pattern, the wavefront pattern [23], etc. It is important for an exchange format to support the modeling of these patterns because they provide important insight to designers into how the application functions. Also, many HPC tools support algorithms for the extraction of communication patterns from traces. An exchange format should have model elements that represent these patterns once extracted.

C. Compaction Framework

In order to provide a scalable representation of HPC traces, we propose a compaction framework that is composed of two trace compaction methods: Call Graph Normalization and Call Tree Transformation.

Call Graph Normalization: The trace of each process in an MPI program can be represented as a routine call tree. The tree contains user-defined functions and MPI operations (which are also functions to the MPI library). MPI operations appear at the leaf level. Usually, these programs generate many contiguously repeating events in the execution trace. 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 [11].

Contiguous repetitions can be removed by collapsing the repetitive calls into one node. However, to be compliant with the transparency requirement, we need also to keep track of the 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 here that we only consider the routine names. If the argument values of the user-defined routines need to be preserved, then this compaction alone will not be sufficient. It needs to be augmented with other data structures to keep track of the arguments of each call. 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.

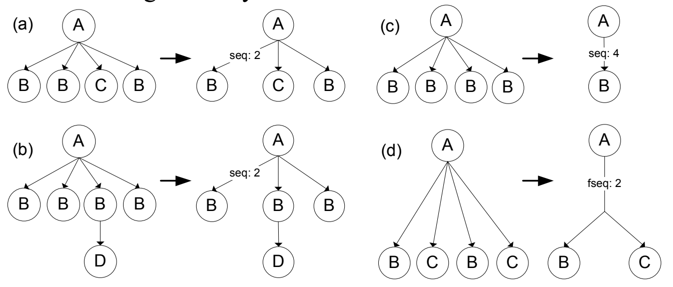

Fig. 3. Collapsing contiguous calls

Figure 3 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 2a shows that only the first two occurrences of ‘B’ can be collapsed. Example 2b shows that since the third occurrence of ‘B’ is calling ‘D’, then only the first two occurrences of ‘B’ can be collapsed. Example 2c 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 a fork-sequence which indicates that the ‘B, C’ sequence is repeated twice in the graph and is being called by ‘A’.

Also, nodes that occur from recursive calls can be collapsed into one node. For example, Figure 4 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.

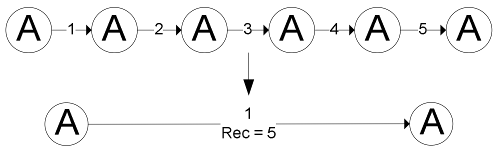

Fig. 4. Collapse Recursive Calls

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 5 shows an example depicting how the same message can be collapsed into one message node. The MTF2 metamodel, presented 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 Send and Receive operations. The MessageLink class will simplify the retrieval of the timestamps from the timestamp array in the Message node.

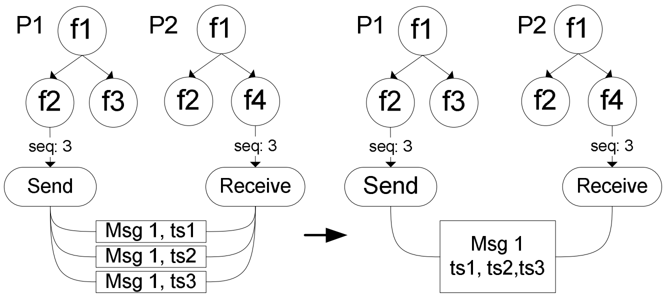

Fig. 5. Example of message compaction

As we can see 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 of the class Edge in the MTF2 metamodel (see Figure 8).

Call tree transformation: The Call Tree Transformation approach is inspired by the compactness scheme presented by Hamou-Lhadj and Lethbridge [11]. In the effort to understand the complexity embedded in method call traces of single-threaded object-oriented systems, the authors proposed to transform a call tree into an ordered DAG where similar sub-trees are represented only once. 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.

Figure 6 shows an example of converting a tree into an ordered DAG after removing contiguous repetitions (Figure 6b). There exist several algorithms (see [5, 6] for an example) that perform this transformation in O(Nd) time where N is the number of nodes in the tree and *d* is the maximum degree of the tree. More discussion on how to generate MTF2 traces using this transformation is presented in Subsection III.E.

It should be noted that the graph edges are ordered from left to right to be able to reproduce back the original tree, if needed. As shown in Figure 6b, 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 6c 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 6a (with bolded node labels) 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.

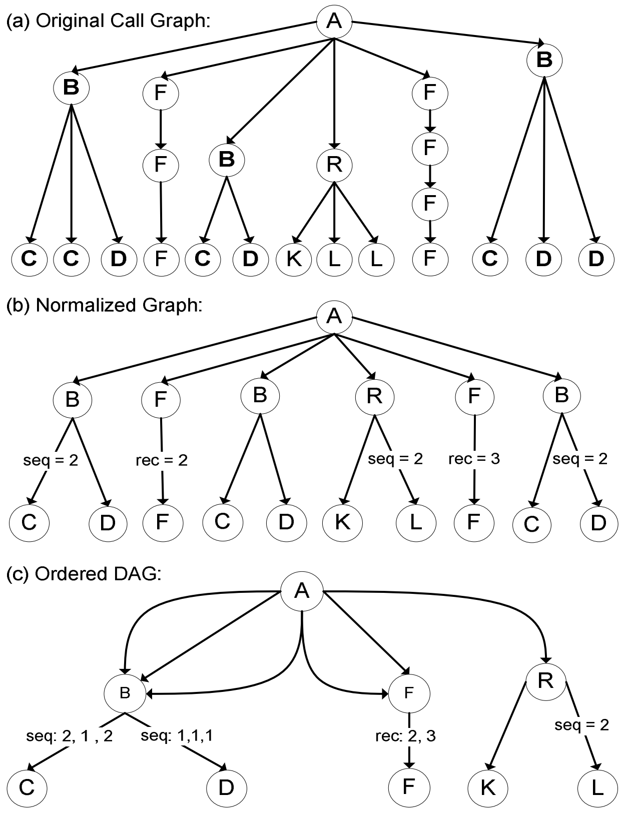

Fig. 6. Tree to DAG Conversion Example

The proposed transformation is lossless, i.e., it keeps the original data. The DAG can be converted back to the tree. More compaction can be achieved if a matching function is designed to measure similarity among sub-trees. In other words, two sub-trees can be mapped to one if deemed similar without necessarily being identical. For example, the two sub-trees 'A calls B, C, and D' and 'A calls B, C, D, C' can be considered similar and mapped to the same sub-graph since they differ only slightly in content.

It is tempting to consider similarity when working with traces to further reduce the size of the trace model. The danger with lossy transformation is that we might end up losing information needed for analysis. Trade-off between completeness and compaction should be carefully examined.

It is worth noting that MTF2 metamodel is designed in such a way that it supports lossy compaction as well. This is because it does not depend on the way sub-trees are mapped. An MTF2 model simply supports the ordered DAG (see subsequent section) no matter if it is originated from using identical matching or similarity.

D. The MTF2 Metamodel

The MTF2 metamodel is shown in Figure 7. It improves over the previous versions in many aspects. The main changes we made are indicated in dashed line rectangles (see Figure 7). The first change, which is perhaps the most important one, consists of modeling a trace as a DAG and not as a tree. The Edge class and the two associations that link it to the TraceableUnit class are used to represent the call tree as an ordered DAG. The type attribute of the Edge class specifies the normalization type applied to the sub-tree. The model supports three types which are the sequence, fork-sequence and recursive edges as described in the previous section.

Another important change consists of the introduction of the RoutineCall class that is used to model user-defined functions. We considered MPI operations as just another type of routines as depicted in the inheritance relationship between the classes MPIOperation and RoutineCall.

Moreover, we added support to communication patterns as shown by the CommunicationPattern class. As mentioned earlier, communication patterns are important concepts in understanding the behaviour of HPC systems [3]. Many analysis tools implement algorithms to recover such patterns from traces of HPC systems. We believe that addressing explicitly these concepts in the model will facilitate the handling of patterns. For example, once a pattern is identified, it is modeled as an object of the CommunicationPattern class. The user can assign to it a description. The patterns can then be saved and retrieved during the next explorations. Note that, in the model, we distinguish between communication patterns and routine call patterns (hence the RoutinePattern class). By the latter, we mean patterns of function calls that do not necessarily depict communication patterns. Routine call patterns have also been used to simplify the analysis of function call traces. Unlike communication patterns, routine call patterns might or might not involve MPI operations. More discussion on the usefulness of both types of patterns in the analysis of HPC systems can be found in [3].

Furthermore, we have made several other refinements to the original model to improve its expressiveness (e.g., added new collective operations) and simplicity. For example, we introduced the concept of ProcessTrace (see Figure 7 box 4) to represent a trace of only one process. This is useful by itself to manage the complexity of manipulating the trace in a tool. For example, if a user wants to remove (or hide) a particular process, then it is sufficient to filter out the corresponding object from the internal model.

We also introduced a message trace (see class MsgTrace), which represents a trace of messages exchanged between processes. In some situations, it is useful to only examine the messages that are exchanged. This is particularly important for forensic analysis of traces, for example, extracted due to attacks. In such case, the user does not need to trace other data (which will result in less tracing overhead) if only the messages are needed for analysis.

Finally, we added several constraints to enforce model consistency at run-time. The main constraints that were added to MTF2 are:

- 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.
- An object of type Barrier cannot reference an object of type CollectiveData.
- 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.
- In a Gather operation, the receiving buffer for non-root process should be equal to null.
- Instances of AllGather & AllToAll do not reference a root process.
- Only an edge with a fork-sequence type can have more than one child node.

E. Generating Traces in MTF2

Generating traces directly in MTF2 requires a mechanism that can convert a trace (as a call tree) into an ordered DAG. Saving the trace as a tree and later converting it into an ordered DAG (and hence modeling it in MTF2) defeats the purpose of having a compact model in the first place. An MTF2 must never be saved as a tree.

The problem of converting a rooted tree into an ordered DAG has been the topic of many studies in graph theory. It is often referred to as the common subexpression problem. It has applications in a variety of areas in computer science including compiler design, symbolic manipulation of code, and computer algebra. Several algorithms have been proposed to solve this problem, among which, perhaps the first one is the one proposed by Downey et al. in [5]. An improvement to Downey's algorithm was proposed by Flajolet et al. [6].

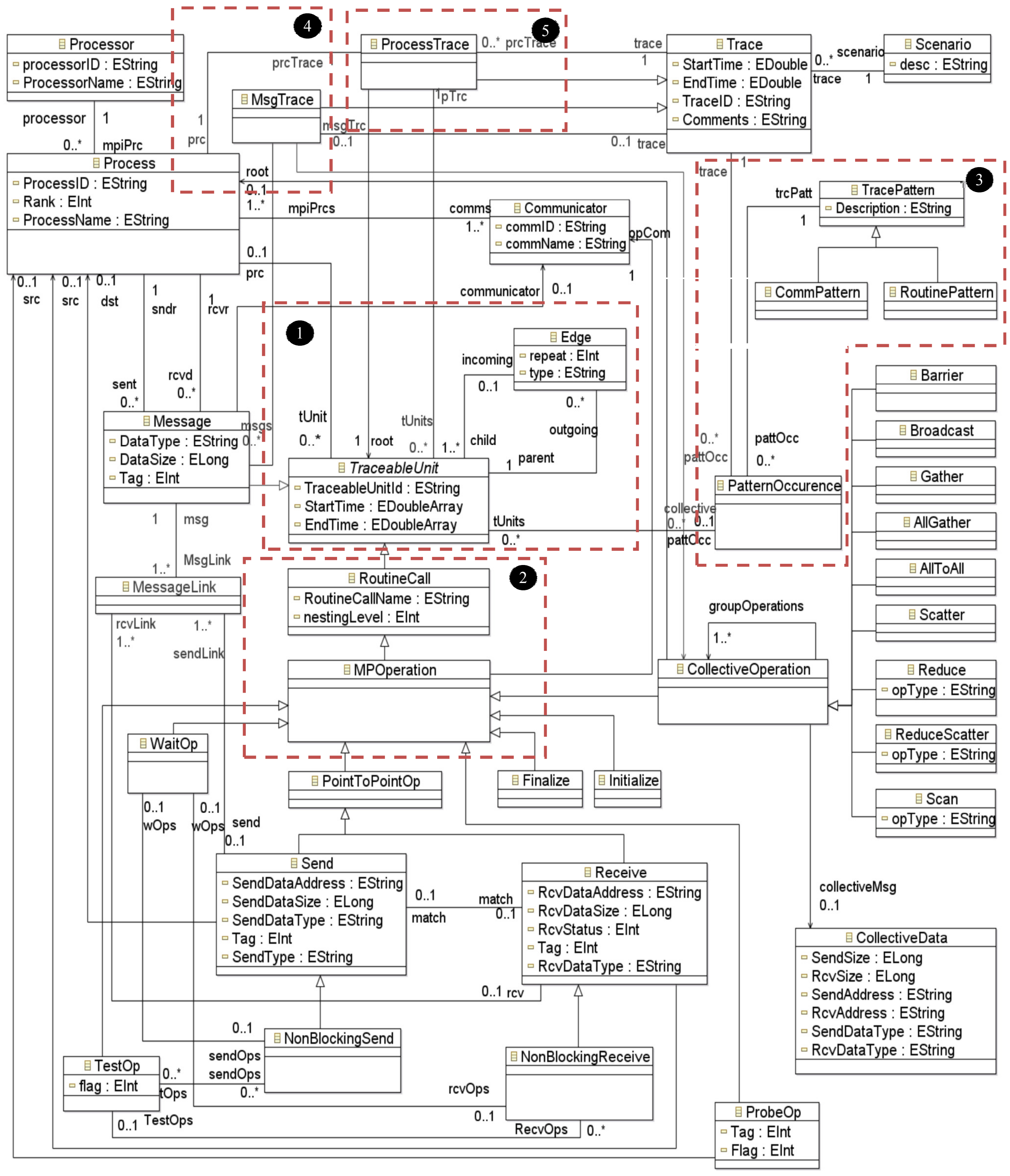

Fig. 7. MTF2 Metamodel

The authors proposed a top-down recursive procedure that solves this problem in an expected linear time (O(N), N being the size of the tree) assuming that the degree of the tree is bounded by a constant [6]. Flajolet et al.'s algorithm is shown in Figure 8. The algorithm uses a global table that keeps track of the subtrees that have been visited. It builds a signature for each node of the tree, which consists of the node label and the unique identifiers (UIDs) of its children. A UID is a global variable that is first set to zero and is incremented whenever a new node is encountered.

The algorithm, as defined in [6], works on binary trees. To adapt it to transforming call trees, we simply need to consider all the children of each node by modifying the code in the rectangle of Figure 8. It should be noted that the complexity of the algorithm when applied to call trees is O(Nd) where *N* is the size of the tree and *d* is the largest degree of the tree. A degree of a node is the number of its incident edges.

```
function UID(T : tree) : integer;
begin
  if T = nil
      then return(0)
      else
        triple := <root(T),UID(left(T)),UID(right(T))>;
        if Found(triple,Table)
           then return(value_found)
           else counter := counter+1;
                Insert pair (triple,counter) in Table;
                return(counter)
        fi
  fi
end;
```

Fig. 8. Flajolet et al.'s algorithm for transforming a tree into a DAG (taken from [6])

It is important to emphasize that Flajolet et al.'s algorithm transforms the tree on the fly. In other words, we do not need to save the tree before transforming it. This is important so as to generate traces directly as ordered DAGs and not as trees.

An example of applying the algorithm (after adapting it to call trees) to the tree of Figure 1a is shown in Table 1. To reproduce the ordered DAG, we simply need to follow the signatures, starting with the one that has the highest UID (here it is A 2 3). In each step, we replace the UID with the corresponding signature. For example, after A 2 3, we obtain two branches, B 0 1 and C 2, by extending the UIDs 2 and 3. Note that the subtree B 0 1 is represented in this table only once.

IV. Evaluation of the Compaction of MTF2

In this section, we show the ability for MTF2 to model traces generated from various HPC systems. We also generated traces in another format, called OTF [13] for comparison. We chose OTF because of its popularity. It is also used in commercial tools such as Vampir.

TABLE 1. BUILDING THE GLOBAL TABLE FOR THE TREE IN FIGURE 1A

Signature	UID
D	0
E	1
B 0 1	2
C 2	3
A 2 3	4

One way to compare the scalability of two formats is to measure the size difference of their corresponding trace files. Obviously, this comparison is sensitive to the syntactic form (i.e., the data carrier) used by each format. Since our focus is to assess the effectiveness of the compaction scheme used by MTF2, we take a different approach. We compare both formats at the object level, i.e., the number of nodes and edges, loaded into memory. This is more interesting than examining the trace files because trace files can always be compressed to reduce disk space.

In this case study, we proceed as follows: we generate traces in OTF, load them as call trees, and then perform our compaction rules on the tree nodes as well as on the point-to-point messages. This will result in an MTF2 object model of the original OTF trace. We then measure the gain obtained from compacting OTF traces using MTF2 compaction methods. Note that, in practice, we do not need to have OTF in order to generate MTF2 traces. MTF2 traces can directly be generated using a native tracer that implements the algorithm presented in the previous section.

We use the following metrics:

Given:

N: The number of node objects
E: The number of edge objects
M: The number of message objects

We measure the size, *A*, of the trace *before* compaction as the total number of node, edge, and message objects. Note that we are using the subscript *0* to mean '<u>before compaction</u>'.

$$A = \sum(N_0, E_0, M_0)$$

Similarly, we measure the size, B, of the trace *after* compaction as the total number of node, edge, and message objects. Note that we are using the subscript *1* to mean '<u>after compaction</u>'.

$$B = \sum(N_1, E_1, M_1)$$

We measure the compaction rate, CR, as follows:

CR (Compaction Rate) = (1 – B / A) * 100%

CR varies from 0 to 100%. It converges to 0 if little gain is obtained. It is close to 100% when the gain in terms of size is high.

TABLE 2. EMPIRICAL RESULTS (#P IS NUMBER OF PROCESSES, N IS NUMBER OF NODES, E IS NUMBER OF EDGES, M IS NUMBER OF MESSAGES, $A = \sum(N_0, E_0, M_0)$, $B = \sum(N_1, E_1, M_1)$, CR (COMPACTION RATE) = (1 – B / A) * 100%, 0 SUBSCRIPT MEANS BEFORE COMPACTION, 1 MEANS AFTER COMPACTION

System	#P	N_0	E_0	M_0	A	N_1	E_1	M_1	B	CR
WRF	16	272373	272357	25680	570410	8779	245752	21881	276412	51%
SWEEP3D	16	962244	962228	239616	2164088	546	960772	239472	1200790	44%
SWEEP3D	32	4867550	4867518	1181578	10916646	672	4867518	380198	5248388	52%
SMG2000	16	2095262	2095246	489148	4679656	336	2095246	179543	2275125	51%
SMG2000	32	2084228	2084196	519168	4687592	1090	2081284	518902	2601276	44%
SMG2000	64	10593512	10593448	2662152	23849112	1344	10593448	778816	11373608	52%

The target HPC systems used in this study are WRF (the Weather Research and Forecast) [28], SMG2000 [24] and SWEEP3D [26]. WRF 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. SWEEP3D models a 3D discrete ordinates neutron transport and represents the heart of a real ASCI application. This code is included in the ASCI Blue Benchmark Suite. We generated two traces from running the program using 16 and 32 processes. SMG2000 is a parallel multi-grid solver applied to linear systems. We tested the compaction algorithm on three traces generated from running the program on 16, 32, and 64 processes respectively.

As we can see in Table 2, except for the two first traces, the other ones contain millions of events. Applying the MTF2 compaction mechanism to these traces results in 44% to 52% compaction rate. We can further improve the compaction gain by considering lossy compaction, conjecturing that some maintenance tasks may not need to the full model. One way to achieve this is by introducing matching criteria by which two subtrees can be considered similar even if they are not identical. For example, if two subtrees differ only with a certain number of functions and MPI operations, we may consider them similar and collapse them into the same subexpression. The edit distance can, for example, be used to measure the similarity between two subtrees. Another matching criterion could be to map the timestamps into a timestamp interval. This way, we do not need to keep track of every single timestamp. We conjecture that this could be useful for maintenance task that do not require timing information. In fact, we can further combine multiple criteria for better compaction. The challenge is to determine which criteria best fit specific maintenance tasks and how these criteria, once identified, can be combined.

V. THREATS TO VALIDITY

A threat to the validity of our conclusions exists because we used traces of both user-defined functions and MPI operations. One may argue that many analysis tools may only require MPI operations

A threat to internal validity exists in the way we collected traces for the case study. We use the Vampire tool to generate OTF traces that we then turned into MTF2. This is because we have not developed a native tracer that generates MTF2 traces yet. This threat is mitigated by carefully testing, using a variety of scenarios, that the conversion from OTF to MTF2 is performed properly.

A threat to external validity exists in generalizing the results of this study as we have only experimented with three open source systems. Though these systems are also used in other studies such as [30], we need to conduct additional studies on large industrial systems.

VI. CONCLUSION AND FUTURE WORK

We presented an exchange format, called MTF2, for representing HPC traces generated from HPC applications. MTF2 is built with trace compaction in mind to allow it to be scalable to large traces. MTF2 is also expressive enough to carry data that describe various behavioural aspects of HPC systems. MTF2 is based on graph theory concepts to achieve an acceptable compaction level. The evaluation of our approach is demonstrated to be efficient when applied to complex HPC commercial applications. MTF2 is implemented as an Ecore model using the Eclipse Modeling Framework. The format is open and available for download.

An immediate future direction is to continue experimenting with large traces to more precisely establish the compaction gain range for MTF2. We also plan to further investigate ways to reduce the number of edges between the nodes of the ordered DAG. Finally, we need to create native tracers that can generate automatically traces in MTF2 (i.e., using the compaction mechanism) to have software engineers use the new format.

ACKNOWLEDGEMENT

This work is supported by the NSERC Natural Science and Engineering Research Council (NSERC).

References

[1] L. Alawneh and A. Hamou-Lhadj, “MTF: A Scalable Exchange Format for Traces of High Performance Computing Systems,” *In Proc. of the International Conference on Program Comprehension (Early Research Achievement Track),* pp. 181 - 184, 2011.

[2] L. Alawneh and A. Hamou-Lhadj, “An exchange format for representing dynamic information generated from High Performance Computing applications,” *Elsevier Journal on Future Generation Computer Systems, 27(4),* pp. 381-394, 2011.

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

[4] 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*, pp. 1-10, 2007.

[5] J. P. Downey, R. Sethi, and R. E. Tarjan, “Variations on the Common Subexpression Problem,” *Journal of the ACM, 27(4),* pp. 758-771, 1980.

[6] P. Flajolet, P. Sipala, J-M Steyaert , "Analytic variations on the common subexpression problem," *In Proc. of the 7th International Colloquium on Automata, languages and programming,* pp. 220-234, 1990.

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

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

[9] M.T. Heath, J.E. Finger, “Paragraph: a performance visualization tool for MPI,” 2003 – Technical paper Available online. URL:http://www.csar.illinois.edu/software/paragraph/

[10] A. Hamou-Lhadj and T. Lethbridge, "A Metamodel for the Compact but Lossless Exchange of Execution Traces," *The Springer Journal of Software and Systems Modeling (SoSym), 11(1),* pp. 77-98, 2012.

[11] A. Hamou-Lhadj and T. C. Lethbridge, “Understanding the Complexity Embedded in Routine Call Traces with a Focus on Program Comprehension Tasks,” *IET Software Journal,* pp. 161 – 177, 2009.

[12] KDM (Knowledge Discovery Metamodel) specification available online: http://www.omg.org/spec/KDM/1.3/

[13] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, W. Nagel, “Introducing the Open Trace Format (OTF),” *In Proc. of the International Conference on Computational Science,* pp. 526–533, 2006.

[14] 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*, pp. 165 – 172, 2005.

[15] J. R. Larus, “Whole program paths,” *In Proc. of the Conference on Prog. Lang. Design and Implementation,* pp. 259-269, 1999.

[16] MPI (Message Passing Interface) Specification. URL: http://www.mpi-forum.org.

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

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

[19] M. Nelson, J-L G., The Data Compression Book, Wiley, 1995

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

[21] 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,* pp. 230–237, 2008.

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

[23] 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.

[24] SMG2000: Advanced Simulation and Computing Program. URL: http://www.llnl.gov/asc/purple/benchmarks/limited/smg/

[25] S. Shende and A. D. Malony, “The TAU Parallel Performance System,” *International Journal of High Performance Computing Applications, (20)2,* pp. 287 – 311, 2005.

[26] Sweep3D, Accelerated strategic computing initiative. http://public.lanl.gov/hjw/CODES/SWEEP3D/sweep3d.html.

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

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

[29] F. Wolf, B. Mohr, “EPILOG binary trace-data format, Technical Report,” Technical Report FZJ-ZAM-IB-2004-06, 2004.

[30] L. T. Yang, M. Guo. High-performance computing: paradigm and infrastructure. ISBN: 978-0-471-65471-1, Wiley, 2005.