Runtime MPI Correctness Checking with a Scalable Tools Infrastructure

Kurzfassung zur Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Tobias Hilbrich
geboren am 5. Oktober 1983 in Dresden

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Wolfgang E. Nagel

Dresden, December 18, 2014

Abstract—Increasing computational demand of simulations motivates the use of High Performance Computing (HPC) systems. At the same time, the parallelism of these systems poses challenges to application developers. The Message Passing Interface (MPI) is a de-facto standard for distributed memory programming in HPC. At the same time, its use enables complex parallel programing errors such as races, communication errors, and deadlocks. Automatic tools can assist application developers in the detection and removal of such errors.

This thesis considers tools that detect such errors during an application run and advances them towards a combination of both precise checks (neither false positives nor false negatives) and scalability. Novel hierarchical correctness analyses and a distributed transition system concept for MPI enable this combination. At the same time, this thesis investigates novel concepts to develop HPC tools and proposes an abstraction for fine-grain tool development with reusable parts. This approach incorporates common tasks such as instrumentation, scalability concepts, and a high degree of flexibility that lets tool developers freely decide where parts of the tool functionality execute. The use of the proposed tools infrastructure abstraction in wide ranges of parallel runtime tools could greatly increase component reuse, as well as reduce development costs. An application study with up to 16,384 application processes demonstrates the applicability of both the proposed runtime correctness concepts and of the proposed tools infrastructure.

1 Introduction

The development of massively parallel software is crucial to satisfy increasing computational demands of modern simulations. At the same time, the High Performance Computing (HPC) systems that support such parallelism follow hardware trends towards increasing compute core counts. The Message Passing Interface (MPI) [11] is a de-facto standard for distributed memory programming [4] that allows programmers to utilize this parallelism. MPI enables the development of highly scalable applications, but offers few extensions to enforce its correct use. The standard includes more than one hundred functions to which specific rules and restrictions apply. Particularly, software faults that involve incorrect MPI usage can manifest into failures that corrupt application results, as an application crash, or as a deadlock.

Tools play a critical role in aiding developers during the development of correct and efficient applications. Especially, they support application developers in the removal of MPI related software defects with a variety of techniques [6]. Automated runtime correctness tools form one of these techniques and detect usage errors of the MPI standard during an application run.

1.1 Runtime MPI Correctness Tools

Various tools to detect MPI usage errors at runtime exist, e.g., ISP [17], Marmot [7], Umpire [18], MPI-Check [10], and TAC [13]. These tools differ in the set of defects that they can detect and in the techniques that they apply to detect them. A study in the thesis that is associated with this document—*this/the thesis* in the following—finds that existing tools: Avoid heuristics that can introduce false positives/negatives, but have limited scalability due to the use of a centralized instance that runs a large portion of the correctness analysis; Or that these tools apply heuristics in order to achieve a certain degree of scalability, but can fail to detect some defects or provide spurious error reports.

Tools with limited scalability restrict their applicability in practice, since some failures may not occur in smaller application runs, and need to be understood at a certain scale [3, 8, 16]. At the same time, tools with heuristics that introduce opportunity for false positives or false negatives reduce the confidence that application developers can have in them. Thus, the thesis addresses the question:

Can an MPI runtime correctness tool provide both a high degree of scalability without the use of correctness analyses that can report false positives/negatives?

1.2 Parallel Tools Infrastructures

The development of runtime correctness tools requires components for common tasks. This includes instrumentation that observes relevant application activities, communication to forward information within the tool, and scalability services. Since the development of portable, efficient, and scalable tool components is challenging [9, 15], reusable components preserve lessons learned and simplify tool development. Parallel tool infrastructures or frameworks—*tool infrastructures* in the following—can both directly provide tool components, e.g., an instrumentation service, and can provide abstractions that allow a reuse of tool functionality. Thus, tools that use the same tool infrastructure can share parts of their implementations.

A current tool development trend is the use of infrastructures with Tree-Based Overlay Network (TBON) capabilities [1, 3, 5, 12], since they provide a powerful approach for scalability. The TBON concept uses a hierarchy of processing nodes—*places* in the following—that allows tools to condense information as it progresses towards the root of the hierarchy. Aggregations and filters on all hierarchy layers provide a step-by-step means to condense the information. TBON-based tools such as STAT [3] and DDT [2] operate for applications that run on close to, or even more than, one million compute cores. Of these two examples, STAT already uses the tools infrastructure MRNet [14] that provides TBON services.

However, an evaluation of existing infrastructures in this thesis underlines that current TBON approaches use distinct interfaces for specific parts of the TBON network. These interfaces differentiate between front-end code that executes on the root of the tree, back-end code that executes on the application processes, and module code that executes on the intermediate levels of the hierarchy. Thus, as an example, a piece of tool functionality that usually operates on the application processes cannot be migrated to a higher hierarchy layer of the tree directly. This unnecessarily restricts component reuse. Consequently, a key question that this thesis addresses is:

> *Can an infrastructure provide a single concept to put tool functionality flexibly onto the application processes, the root of the TBON, or intermediate hierarchy layers?*

In summary, this thesis follows two goals: First, it advances runtime correctness tools for MPI with concepts and algorithms for scalable and precise checks. Second, it proposes a novel abstraction for a parallel tools infrastructure that simplifies the development of such a correctness tool. The abstraction and the assumptions for this infrastructure must apply to the correctness tool use case, but should be general enough to apply to a wide range of runtime tools. Such capabilities should particularly include an abstraction that enables reuse of parts of the tool implementation across different tools. Challenging correctness analyses such as deadlock detection then provide a means to test both the scalability features and the abstraction of the infrastructure that this thesis proposes. At the same time, these correctness capabilities advance the scalability of previous precise approaches for runtime MPI correctness tools.

2 Tool Infrastructure Concept

The subsequent concept for novel tools infrastructures relies on a notion of *events* and *analyses* (sometimes also referred to as event-action [19]):

Event:	Occurrence of information that the runtime tool must know about, and
Analysis:	The processing that needs to take place upon perceiving an event.

2.1 Analyses

Events and analyses capture the activities of a wide range of existing HPC tools, such as:

- Tracing tools for performance analysis store information on a function invocation (event) into a trace buffer, from where they are processed (analysis);

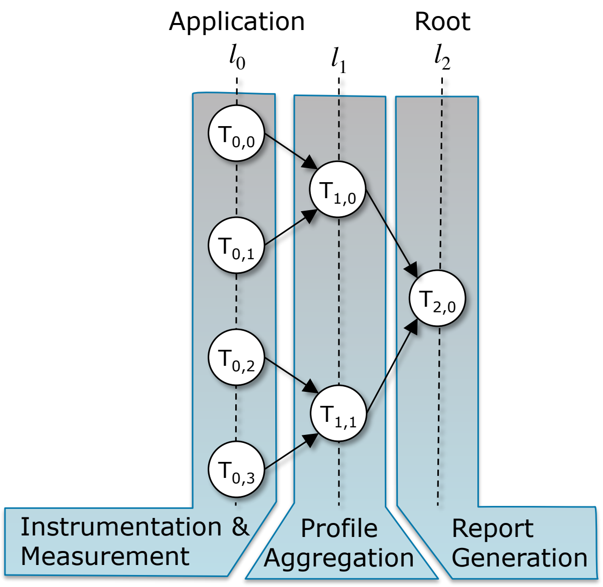

Figure 1: *Illustration of a tool layout and analyses for a profiling tool.*

- A debugging tool retrieves a stack trace (analysis) due to a request from the graphical user interface (event);
- A sampling-based performance analysis tool retrieves a stack trace (analysis) when a new sample is triggered (event); or
- A runtime correctness tool for MPI analyzes whether send and receive datatypes match (analysis) when a pair of send and receive operations is observed (events).

Consequently, the goal of the subsequent abstraction is that the overall functionality of a tool derives from a set of events and analyses, as well as an association between the two. A tool developer then focus on the fundamental questions required for their particular tool:

What are the activities (analyses) that the tool must implement and which events trigger them?

With this approach, a tool developer can implement the tool functionality as a set of analyses and describe their relation to events of interest. Distributing analyses across a runtime tool to apply them to application processes or further tool owned processing capabilities, triggering them when events occur, and handling the instrumentation that observes relevant events can be left to an infrastructure then, rather than the tool developer.

The following example illustrates this idea using a simplified profiling tool. A profiling tool creates basic execution profiles and maps them to source code regions to aid in performance optimization. Such a tool would observe when the application enters or leaves functions. Summarizing this information for all functions of equal name, the tool provides basic profiles that identify the functions in which the application spends most time. Figure 1 illustrates an instance of a profiling tool for four application processes/threads depicted as nodes $T_{0,0}$–$T_{0,3}$. Instrumentation, i.e., observing function enters and leaves, takes place directly on the application, along with adding up the time that each function consumes. The box with *instrumentation & measurement* in the figure illustrates the association of this tool functionality to the application processes. The tool in the figure employs a TBON layout for two purposes: First, the root $T_{2,0}$ of the TBON can create a summary report that could average the profiling data from all processes. Second, the intermediate layer with $T_{1,0}$ and $T_{1,1}$ aids in averaging the profiling data, by applying a step-wise aggregation of the data. This *event aggregation* replaces multiple input events with a single/few summarized events. As a result, tool nodes (places) on each hierarchy layer receive a constant amount of events and higher layers do not become a scalability bottleneck.

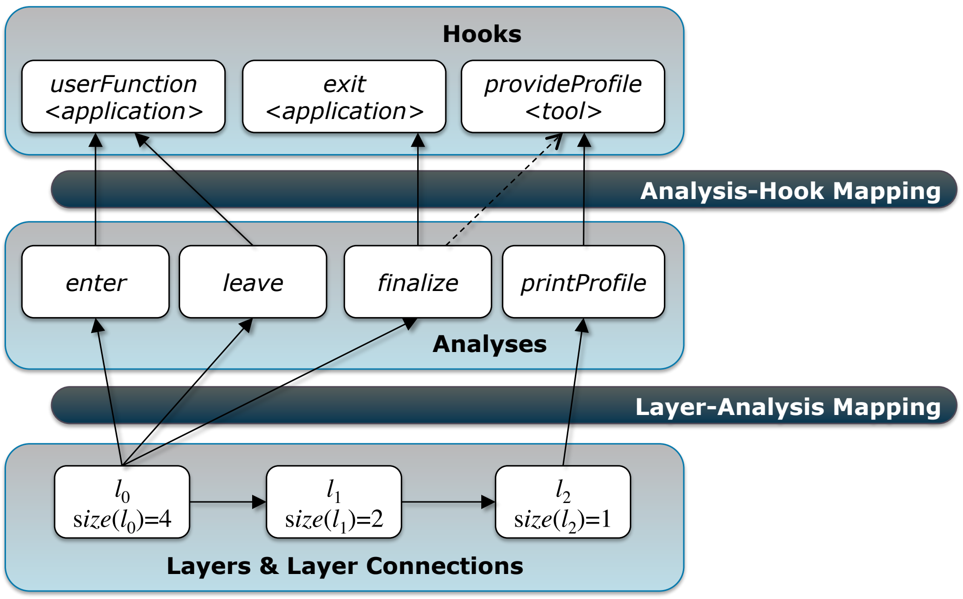

Figure 2: *Illustration of the proposed mapping-based abstraction for the profiling tool example from Figure 1.*

2.2 Abstraction

The proposed parallel tools infrastructure abstraction employs the notion of events and analyses. Then, it combines them with a notion for a tool layout and defines which events form the input of an analysis on the layout. Following the profiling tool example with the high-level activities from Figure 1, Figure 2 illustrates the subsequent terms.

Analyses represent available actions for tools. A selection from these analyses then makes up the overall functionality of a tool instance. A set of available analyses A represents them. For the profiling tool, an analysis enter could observe when the application enters a function, an analysis leave could then observe when the application leaves a function, an analysis $\mathrm{finalize}$ could trigger the creation and forwarding of the profiling data before the application exits, and an analysis $\mathrm{printProfile}$ could write the resulting profiling data into a file for user investigation. Figure 2 (middle) illustrates the analyses $A = \{\mathrm{enter}, \mathrm{leave}, \mathrm{finalize}, \mathrm{printProfile}\}$. The thesis considers *modules* as an additional indirection step to package multiple analyses that operate on the same data together. The following definitions are a simplification from the thesis and omit the handling for modules.

Layers and Layer Connections Figure 1 illustrates a possible layout for the example tool. The proposed abstraction defines such layouts with a set of *layers* $L = \{l_0, l_1, \dots\}$ that uses l_0 as the layer that consists of the application processes or threads. A *layer tree* $\mathcal{L} = (L, E_{\mathcal{L}} \subseteq L \times L)$ then connects these layers to indicate which layers send information to which other layers. This forwarding represents the *towards-root* direction for scalable tools that apply event aggregation, as in existing TBON tools. The layer tree $\mathcal{L}$ must form a directed tree with root l_0, i.e., the *towards-root* direction always originates from the application layer, but multiple or partial hierarchies could still be used. The example uses $L = \{l_0, l_1, l_2\}$ and $E_{\mathcal{L}} = \{(l_0, l_1), (l_1, l_2)\}$. The lower third of Figure 2 illustrates this layer tree.

Further, a function $\mathrm{size} : L \to \mathbb{N} \setminus \{0\}$ associates a *place* count with each layer ($\mathbb{N}$ being the set of natural numbers). A place represents an application process/thread or a tool process/thread. Thus, $T_{1,0}$, $T_{1,1}$, and $T_{2,0}$ in Figure 1 could be additional processes, as well as additional threads on the application processes $T_{0,0}$–$T_{0,3}$. In the example: $\mathrm{size}(l_0) = 4$, $\mathrm{size}(l_1) = 2$, and $\mathrm{size}(l_2) = 1$.

Connection rules then create a *tool topology graph* $\mathcal{T} = (P, E_{\mathcal{T}})$ from a layer tree, e.g., to create the layout in Figure 1 from the layer tree in Figure 2 (bottom). The node set of this graph is the set of all places $P = \{T_{i,j} : l_i \in L \text{ and } 0 \leq j < \text{size}(l_i)\}$, i.e., for each layer l_i the rules create as many places $T_{i,j}$ as the size function indicates ($0 \leq j < \text{size}(l_i)$). A connection rule then connects places of connected layers to create the set of arcs $E_{\mathcal{T}} \subseteq P \times P$.

Layer-Analysis-Mapping To associate tool functionality in the form of analyses with specific places, a *layer-analysis-mapping* specifies the analyses that each layer executes. The function $m_{L,A} : L \to \mathcal{P}(A)$ specifies this mapping. Figure 1 illustrates the association for the example tool, which could use $m_{L,A}(l_0) = \{\text{enter}\}$, $m_{L,A}(l_1) = \emptyset$, and $m_{L,A}(l_2) = \{\text{printProfile}\}$. Figure 2 illustrates $m_{L,A}$ with arrows between layers and analyses.

Hooks and Analysis-Hook-Mapping The proposed abstraction takes care of the instrumentation of the application, i.e., of observing relevant (application) activities, in order to simplify tool development. This requires that the infrastructure is aware of what can be instrumented, which is specified as a set of *hooks* H. A hook $h \in H$ is some activity that the infrastructure can instrument in order to observe it. When during tool runtime, an instrumented hook h is triggered, the runtime of the infrastructure creates an *event* with information on the observation of hook h. Examples of hooks are function calls or APIs such as the profiling interface of MPI.

The profiling tool example uses $H = \{\text{userFunction}, \text{exit}, \text{provideProfile}\}$. The userFunction hook observes all function calls of the application and could rely on a compiler-based or source-to-source instrumentation of the application. The exit hook observes when the application attempts to exit, e.g., with an *atexit* handler. When it is triggered, it must pass the profiling data to the printProfile analysis. To do so:

A tool can use events to both observe application activities and to inject events that communicate information within the tool.

The third hook uses this concept and implements part of the tool functionality, rather than to observe application activities. With the provideProfile hook, the finalize analysis can inject an event that carries this data. Any analysis that is interested in the data, e.g., printProfile, can then observe it. Figure 2 (top) illustrates these hooks and highlights that the first two hooks serve for application activities, while the third hook implements part of the tool functionality.

The above example already illustrates a notion of *an analysis is interested in events of a hook*. An *analysis-hook-mapping* specifies this notion in the proposed abstraction. The function $m_{A,H} : A \to \mathcal{P}(H)$ defines this mapping. The enter and leave analyses must observe the userFunction hook, i.e., $m_{A,H}(\text{enter}) = \{\text{userFunction}\}$ and $m_{A,H}(\text{leave}) = \{\text{userFunction}\}$. Additionally, the finalize analysis observes the exit hook, while the printProfile analysis observes the provideProfile hook. Figure 2 illustrates these mappings with arrows between the analyses and hooks. The arrow with the dashed line illustrates that the finalize analysis would provide the measured profiling data with the provideProfile hook, i.e., that it will trigger it at runtime.

Event-Flow The previous specifications detail an abstract representation of a tool. An instantiation of the proposed abstraction can use this information to map it to the system. While the previous definitions conveyed a notion of their intention, no semantics specified which analyses should be triggered for which events yet. So called *event-flow* definitions define the workflow within the tool:

- What events must the instrumentation system of the tool observe?
- Towards which other places must a place forward events?
- Which analysis must be triggered when the instrumentation system observes an event or if a place receives an event from another place?

Based on a mapping function $\mathrm{execute} : P \rightarrow \mathcal{P}(A)$ that associates a set of analyses with each place[1], the first definition of the event-flow specifies which analyses must be triggered when it observes or receives an event for a hook:

Event-Flow 1 (Trigger) *A place $T \in P$ that observes or receives an event for a hook h must trigger an analysis $a \in \mathrm{execute}(T)$ exactly if $h \in m_{A,H}(a)$.*

This definition follows the intuition that when a place receives or observes an event, it triggers any analysis that is mapped to the hook that created the event.

The remaining event flow definitions depend on *communication directions*. For the *towards-root* direction, events progress from application processes towards higher level hierarchy layers, e.g., towards the root of a TBON. The tool topology graph $\mathcal{T}$, e.g., the one in Figure 1, determines which events a place can perceive. A place T can perceive events that are directly triggered on T and any event that a predecessor in the graph can perceive. As an example, in the figure, $T_{0,0}$ only perceives its own events, whereas the root $T_{2,0}$ can perceive events from all places. To minimize tool overhead, a place must only perceive events for hooks to which some of its analyses are mapped, or which a descendant in the tool topology graph must perceive. The relation $\mathrm{requiresInformation} \subseteq P \times H$ formally defines the hooks whose events a place must observe as $(T, h) \in \mathrm{requiresInformation}$ exactly if $\exists T' \in P$ and $a \in A$:

- $T' \in \{T\} \cup \mathrm{successors}(T, \mathcal{T})$,
- $a \in \mathrm{execute}(T')$, and
- $h \in m_{A,H}(a)$.

The relation includes pairs of a place and a hook if the place itself or a direct successor of the place executes at least one analysis that is mapped to the respective hook $\big(\mathrm{successors}(T, \mathcal{T})$ is the set of successor places of T in the tool topology graph $\mathcal{T}\big)$. This relation directly allows a definition of what hooks a place must instrument, i.e., observe, and towards which places a place forwards events:

Event-Flow 2 (Observe) *A place $T \in P$ must observe (i.e., instrument) a hook $h \in H$ exactly if $(T, h) \in \mathrm{requiresInformation}$.*

Event-Flow 3 (Forward) *A place $T \in P$ must forward an event of hook h, which it observes or receives from another place, to a direct successor $T' \in P$ exactly if $(T', h) \in \mathrm{requiresInformation}$ where $(T, T') \in E_{\mathcal{T}}$.*

For the example, these definitions require that places $T_{0,0}$–$T_{0,3}$ observe the hook $\mathrm{provideProfile}$, even though the analysis mappings do not assign them an analysis that is mapped to this hook. This is due to the fact that place $T_{2,0}$ executes the analysis $\mathrm{printProfile}$ that is mapped to $\mathrm{provideProfile}$, while $T_{2,0}$ is a successor of $T_{0,0}$–$T_{0,3}$ in the tool topology graph.

These event-flow definitions formally define the workings of a tool that follows the proposed abstraction. This thesis incorporates event aggregation into these definitions and considers further communication directions in addition.

2.3 Prototype and Further Infrastructure Contributions

Based on the aforementioned concept, the thesis provides a prototype implementation called GTI. This infrastructure includes the mapping-based abstraction as well as additional contributions to parallel tools infrastructures:

- An integration of event aggregation into the concept,
- A concept for order preserving event aggregation,
- An intralayer communication direction to overcome shortcomings of tree layouts, and
- A crash handling concept to allow event processing to complete past an application crash.

[1]This mapping is defined by the layer-analysis mapping and the concept of event aggregations, which are not handled in this document.

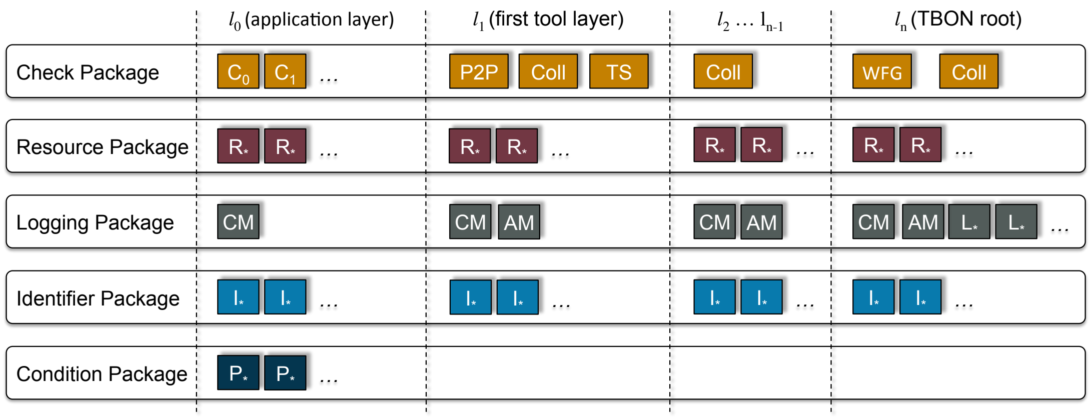

Figure 3: *An illustration of the association of modules—as sets of associated analyses—to important layers of basic MUST instantiations. The figure groups modules into functionality packages.*

3 Distributed MPI Correctness Analysis

The prototype implementation GTI enables an implementation of a novel runtime correctness tool called MUST. Its implementation only needs to provide the actual tool functionality as a set of GTI modules and specifications. These specifications describe the MPI standard as a set of hooks, tool internal hooks for MUST's internal communication, and analysis-hook mappings. The goal of MUST is to combine precise MPI correctness analysis with a high degree of scalability.

3.1 Architecture

Figure 3 summarizes the modules of MUST. Each of these modules (colored boxes with white labels) include a set of associated analyses of the proposed abstraction. Additionally, modules are associated with distinct functionality packages. The modules of the *identifier package* provide basic information on MPI ranks and call locations, e.g., a call name or a call stack. Thus, most other modules require these services to identify which MPI rank created an event. Modules of the *check package* create *correctness messages* when they detect a failure. The *logging package* provides services to report these correctness messages to the tool user. The *resource package* then provides information on handles that identify MPI datatypes, communicators, groups, requests, and operations. The individual modules in this package track all default instances of each of these resources and monitor resource creation, destruction, and modification. *Check modules* use module them to retrieve detailed information about involved MPI resources. The *condition package* provides modules that evaluate whether MPI operations satisfy a specific condition. They serve to filter out unnecessary events and to preprocess events for tool internal communication.

The modules of MUST operate on specific layers of a TBON hierarchy, Figure 3 highlights the default association for the individual functionality packages. Key correctness analysis in the thesis are the *P2PMatch* module ("P2P" in Figure 3), which provides a distributed analysis of MPI point-to-point operations; the *CollectiveMatch* module ("Coll" in Figure 3), which provides a distributed analysis of MPI collective operations; the *TransitionSystem* module ("TS" in Figure 3), which implements a distributed transition system for MPI deadlock detection; and the *WfgManager* module ("WFG" in Figure 3), which provides a graph-based dependency analysis to create detailed deadlock reports. The thesis describes these modules and their distributed analysis concepts in detail, while the following summarizes the *CollectiveMatch* module and its analysis.

Process 0	Process 1
`MPI_Bcast`(root:0)	`MPI_Bcast`(root:1)

Figure 4: *Even if processes execute collective operations of the same type, they can specify incompatible arguments as in this example. For* `MPI_Bcast` *the MPI standard requires all participating processes to identify the same process with the* root *argument.*

3.2 Distributed Analysis of Collectives

Collective communication operations of MPI provide data transfers and reduction operations between groups of processes. MPI communicators specify these process groups. For a collective communication to take place, all involved processes issue a collective operation of the same name with a communicator argument that exactly includes the involved processes. The term *collective* refers to the overall communication between all members of a process group and the term *collective operation* refers to a single MPI call invocation of a member of the group. MPI usage errors for collective operations involve deadlocks, MPI datatype matching defects, and further kinds of inconsistent arguments. The *CollectiveMatch* module of MUST employs a distributed detection scheme to implement correctness analyses for this purpose. The module issues checks to detect inconsistent arguments and type matching defects.

3.2.1 Representative Operations

Figure 4 illustrates an example defect that involves inconsistent collective operation arguments. In the example, processes 0 and 1 specify different values for the root argument. An implementation for a correctness analysis to detect this failure must compare arguments of collective operations that belong to the same collective. A comparison such as “Do all processes use the same root argument?” is transitive. Thus, it is not necessary to compare all pairs of operations with each other. Rather, any comparison scheme that compares pairs of operations, such that the transitive closure of these pairs includes all pairs of distinct operations, suffices. As an example, the approach in Umpire uses one operation o_x of each collective as a representative and compares all other operations against o_x.

To provide a scalable and distributed comparison scheme, this thesis proposes a step-wise hierarchical comparison. It requires a layer l with exactly one place, where l and all its predecessor layers in the layer tree execute a hierarchical comparison as follows: If a place on these layers can receive operations $o_0, o_1, \ldots, o_n$ of a collective, it selects a representative o_x $(0 \leq x \leq n)$ and compares all remaining operations that it receives to o_x. The place then forwards an event for operation o_x and removes the events for the other operations from the primary communication direction. Further, the place stores in the event for operation o_x that this event is a representative for $o_0, o_1, \ldots, o_n$. Places use this information to determine when they perceived all operations for a collective. If a place detects an inconsistency during a comparison, it reports it to the user.

The hierarchical comparison scheme satisfies the above condition that the transitive closure of all compared operations includes all pairs of distinct operations (without proof). As opposed to a centralized scheme on a single place, the hierarchical scheme provides an efficient distribution. For a k-ary TBON layout, and collectives that involve all processes, each place receives information on k operations and forwards information on one operation. Particularly, the workload on each place is independent of the application scale if k remains constant when scale increases.

The following checks apply to collective communications:

Call: Consistent type of operation, e.g., all operations are a `MPI_Bcast`, within one collective;

Root: Same root for all operations of a collective that uses a root process;

Op: Compatible reduction operator (`MPI_Op`) for collectives that use such an operator; and

Types: Type matching for all data transfers of the collective.

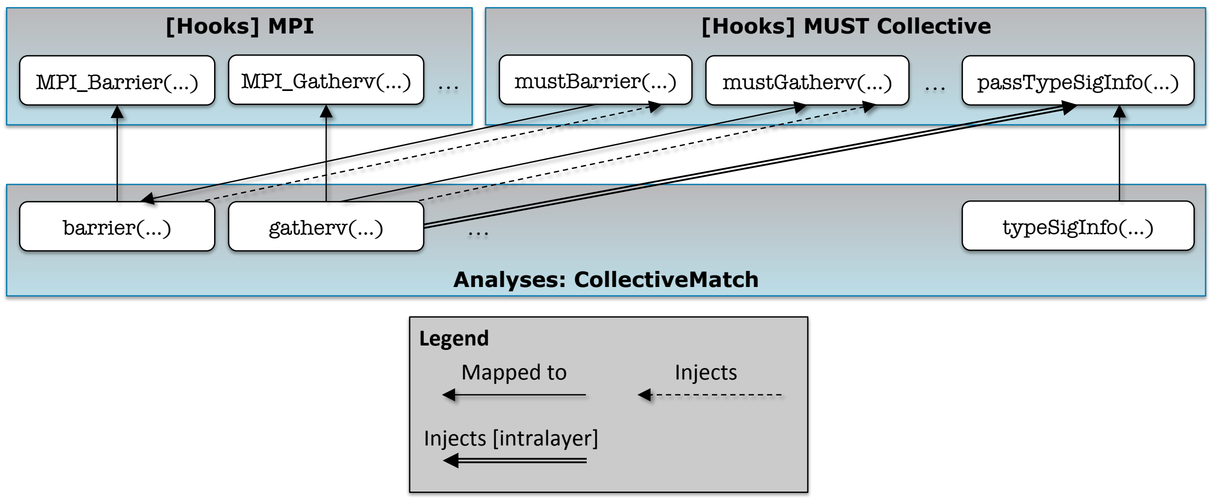

Figure 5: *Illustration of the analyses of the* CollectiveMatch *module, which enables a distributed comparison of MPI collective operations, along with their relevant hooks and mappings.*

All of these checks are transitive with the exception of the type matching rules of the irregular collectives `MPI_Gatherv`, `MPI_Scatterv`, `MPI_Alltoallv`, and `MPI_Alltoallw`. Collectives such as `MPI_Gather` specify transitive type matching rules where the root of the collective and all other processes must use an identical type signature. Type matching checks for such collectives can use a transitive scheme. The irregular version of this collective (`MPI_Gatherv`), however, can specify distinct type signatures between the root process of the collective and each other participating process. Thus, for irregular collectives, a transitive type matching scheme is inapplicable. The *CollectiveMatch* module handles all transitive checks with the hierarchical scheme. For the collectives that use non-transitive type matching rules, the thesis presents a scheme that exchanges type matching information within a single hierarchy layer.

3.2.2 Mapping to Tools Infrastructure Abstraction

Figure 5 illustrates the design of the *CollectiveMatch* module, which implements the hierarchical analysis of MPI collective operations. The module uses one analysis for each collective, e.g., `barrier` for `MPI_Barrier`. Additionally, as to inject events for representative collective operations that represent multiple operations, the design uses one tool internal hook (per type of collective) to inject these events, e.g., `mustBarrier`. When the *CollectiveMatch* module determines that all operations of a collective are consistent, it injects a new representative event with the hook. The hooks that the *CollectiveMatch* module uses include an additional argument that represents information on how many collective operations each event represents. This information supports the *CollectiveMatch* module when it determines whether all reachable operations of a collective arrived.

For the collectives with non-transitive type matching rules, such as `MPI_Gatherv`, the `passTypeSigInfo` hook supports type signature comparisons within a single hierarchy layer.

4 Summary

This document summarizes a novel approach to check the consistency of MPI collective operations, while the associated thesis presents approaches for correctness analyses of MPI point-to-point operations and a distributed deadlock detection scheme. Overall, the design and the prototype implementation of MUST fill the gap of a precise tool that offers a high degree of scalability. The thesis introduces time complexities for these correctness analyses that highlight that the proposed distributed analysis schemes for point-to-point operations, collective operations, and for a transition system that supports

MPI deadlock detection can analyze MPI operations with time costs that compare to their overheads within an MPI implementation.

At the same time, the development of runtime tools for high performance computing systems is expensive and involves many common components and tasks. Recent developments in parallel tool infrastructures provide one approach to avoid reoccurring development costs for such tools. Moreover, abstractions and patterns can guide tool developers towards scalable implementations. Several scalable tools use tree-overlay abstractions, such as provided by the abstractions of MRNet. However, current infrastructures fail to provide some requirements for runtime correctness tool such as MUST, including a crash handling scheme to tolerate an application crash or a communication system that enables data exchanges within a hierarchy layer of a TBON. Additionally, the abstractions of infrastructures such as MRNet employ different interfaces on the application processes, the root of the TBON, or its intermediate layers. This layer-specific distinction unnecessarily reduces flexibility in tool component reuse. Thus, this thesis contributes a concept for a novel tool infrastructure that employs a mapping-based abstraction. Most importantly, this abstraction uses a single concept to apply tool analyses onto application processes, the root of a hierarchy of tool places, or on an intermediate hierarchy level. This contribution overcomes a shortcoming of existing infrastructures. Further contributions of the proposed tool infrastructure concepts include event order preserving aggregations, a communication scheme that enables data exchanges within a hierarchy layer, and a scheme for application crash handling.

The MUST prototype then serves as a challenging test case for the prototype infrastructure implementation GTI. GTI both supports all of MUST's requirements and allows this tool to be specified as a collection of modules—representing the tool analyses—and specifications that provide the analysis-event mappings. In addition, MUST does not implement any of the usual tool components, e.g., instrumentation, tool internal communication, or spawning extra tool processes/threads. Widespread use of an abstraction such as in GTI would drastically simplify tool integrations and combinations.

Further, the thesis presents an empirical study with synthetic stress tests and two widely used HPC benchmarks to evaluate the prototypes of GTI and MUST. The stress tests with up to 4,096 application processes (8,191 total MPI processes) underline that the analyses in MUST and the overheads within GTI can match the scalability that their theoretic time complexities suggest. The two benchmark suites at 2,048 and 16,384 application processes respectively (2,731 and 27,307 total MPI processes) highlight that the MUST analyses provide low overheads that should not limit the applicability of the approach. No other runtime deadlock detection approach for MPI reports results at similar scale. A comparison of the overheads of the distributed transition system to a state-of-the-art centralized runtime deadlock detection approach highlights that the approach in this thesis provides two orders of magnitude lower overheads at 512 processes already.

References

[1] Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP '08)*, pages 578–585, 2008.

[2] Allinea Software. Allinea DDT. `http://www.allinea.com/products/ddt/`. Last visited on 09/12/2014.

[3] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack Trace Analysis for Large Scale Debugging. In *Proceedings of the 2010 IEEE 21th International Parallel and Distributed Processing Symposium*, IPDPS '07, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[4] Mark Bull. Unified European Applications Benchmark Suite. Deliverable D7.4 from PRACE Second Implementation Phase Project. `http://www.prace-ri.eu/IMG/pdf/d7.4_3ip.pdf`, 2013. Last visited on 09/12/2014.

[5] James Galarowicz. Project Final Report: Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools around Open|SpeedShop. Technical report, Krell Institute, 2014.

[6] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Formal Analysis of MPI-Based Parallel Programs. *Communications of the ACM*, 54(12):82–91, 2011.

[7] Bettina Krammer and Matthias S. Müller. MPI Application Development with MARMOT. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 893–900. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[8] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, and Todd Gamblin. Probabilistic Diagnosis of Performance Faults in Large-scale Parallel Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 213–222, New York, NY, USA, 2012. ACM.

[9] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons learned at 208K: Towards Debugging Millions of Cores. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 26:1–26:9, Piscataway, NJ, USA, 2008. IEEE Press.

[10] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012. Last visited on 09/12/2014.

[12] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller. A Framework for Scalable, Parallel Performance Monitoring. *Concurrency and Computation: Practice and Experience*, 22(6):720–735, 2010.

[13] Patrick Ohly and Werner Krotz-Vogel. Automated MPI Correctness Checking: What if there was a magic option? In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, 2007.

[14] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, New York, NY, USA, 2003. ACM.

[15] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. Benchmarking the MRNet Distributed Tool Infrastructure: Lessons Learned. In *Proceedings of the 2004 IEEE 18th International Parallel and Distributed Processing Symposium*, IPDPS 2004, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[16] Anne M. Stark. Bug repellent for supercomputers proves effective. `https://www.llnl.gov/news/newsreleases/2012/Nov/NR12-11-02.html`, 2012. Last visited on 09/12/2014.

[17] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 285–286, New York, NY, USA, 2008. ACM.

[18] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[19] Roland Wismüller, Jörg Trinitis, and Thomas Ludwig. OCM–A Monitoring System for Interoperable Tools. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, pages 1–9. ACM Press, 1998.