# D3.4.1 – Debugging design document

## *WP3: Development environment*

| | |
|---|---|
| **Project Acronym** | CRESTA |
| **Project Title** | Collaborative Research Into Exascale Systemware, Tools and Applications |
| **Project Number** | 287703 |
| **Instrument** | Collaborative project |
| **Thematic Priority** | ICT-2011.9.13 Exascale computing, software and simulation |

| | |
|---|---|
| **Due date:** | M10 |
| **Submission date:** | 31/07/2012 |
| **Project start date:** | 01/10/2011 |
| **Project duration:** | 36 months |
| **Deliverable lead organisation** | Allinea Software LTD (ASL) |
| **Version:** | 1.0 |
| **Status** | Final |
| **Author(s):** | David Lecomber (ASL), Tobias Hilbrich (TUD) |
| **Reviewer(s)** | Mats Hamrud (ECMWF), Jeremy Nowell (UEDIN) |

| **Dissemination level** | |
|---|---|
| <PU/PP/RE/CO> | *PU - Public* |

# Version History

| Version | Date | Comments, Changes, Status | Authors, contributors, reviewers |
|---|---|---|---|
| 0.1 | 28/06/2012 | First draft for review | Tobias Hilbrich (TUD), David Lecomber (ASL) |
| 1.0 | 11/07/2012 | Final draft with incorporated review comments | Tobias Hilbrich (TUD) |

# Table of Contents



# Index of Figures

# 1 Executive Summary

This document describes designs, extension steps, and ideas that will allow the debugger Allinea DDT and the automatic runtime correctness tool MUST to cope with exascale needs. We use deliverable D3.1 “State of the art and gap analysis” as a roadmap for these extensions. We will provide a second version of this document in project month 24 to extend this document and add detail or new knowledge where necessary.

The transition of Allinea DDT to exascale requires extensions to cope with new paradigms and heterogeneous approaches to high performance computing as one important step. Also, we address advances in the automatic detection of anomalies to reduce the debugging time of complex, possibly heterogeneous or scaled to millions of processes, applications. Finally, we add more awareness of models, application structure, or libraries to DDT to present the user with more meaningful information. One important use case is the display of information for MPI resource handles, where Allinea DDT and other debuggers currently provide no resource state.

For the runtime error detection tool MUST the proposed extensions fall into scalability and programming paradigm support. Current runtime error detection tools scale to 100-1,000 processes at most, which is even with todays scale less than unsatisfactory. The design that we propose to distribute key correctness checks of MUST targets a scale up to 10,000 processes. Further extensions might then further enhance MUST’s scalability to cope with 100,000 or 1,000,000 processes. Note that while exascale systems will likely have far more parallel cores, with hybrid programming approaches, the total number of MPI processes might be in this range. Furthermore, the design highlights the specific steps that would allow MUST to provide automatic correctness checks for other paradigms.

As both Allinea DDT and MUST identify programming paradigms as an important goal for improvement, it is crucial to notice that the experience within CRESTA has not yet identified a primary or highly promising programing paradigm for exascale development. As our resources within the project are limited, we expect that for both tools we will only be able to provide extensions for one single programming paradigm or model. Thus, we keep our extensions more general within this document, while we hope to have identified the model of choice for the second version of this document.

A further open gap for both tools is fault tolerance. If median-time-to-failure is below the runtime of an average application, as many expectations for Exascale indicate, both tools need techniques to identify and react to failures. As this effect is barely studied at the current project state we will extend our design document for this important topic in the second version of this document. An important input for this is deliverable D2.5.2, which will be completed in month 30. We hope that we can use first insights from this deliverable around project month 20.

Integrations of DDT and MUST can provide users with a unified user interface that provides access to both tools. Besides the advantages in usability, an integration can also combine the advantages of the tools to lead to a deeper understanding of a software bug or a reduced time to solution. As a result, we describe first steps for a tool integration and resulting extensions to advance this integration. The highest integration degree would allow MUST to operate within DDT’s tree based overlay network. While such an integration is highly desirable we see this as a second priority that only comes after addressing the before-mentioned areas of improvement.

Finally, we investigated the testing processes of both MUST and DDT. We identified that both tools already use very elaborate and extensive functionality tests. Extensions to the existing testing procedures will be added by covering the CRESTA benchmark suite from D2.6.1 to detect performance issues of the tools. Also we will add a system where both MUST and DDT operate to test integration components.

# 2 Introduction

In this document ('Debugging Design Document, D3.4.1') we present designs to extend the existing tools DDT and MUST towards the gaps and high level goals that we identified in D3.1 'State of the art and gap analysis'.

We first describe the current tools for debugging/correctness within CRESTA - Allinea DDT and MUST. We then describe the changes to be made according to the priorities from the Gap Analysis for each tool. Section 2 presents the first designs for the Allinea DDT and Section 3 presents first designs for MUST.

These primary recommendations from the Gap Analysis - in the order addressed in this Design Document - were:

- Programming models are in a state of flux - due to recent seismic hardware shifts and may require specific debugging or correctness support.
- Analysis of applications with automated correctness checking is important and existing tools do not scale: the MUST package from TU Dresden should be extended to improve scale.
- Automated anomaly detection is increasingly important as the scale of concurrency grows - and architectures change - therefore detection of changes between runs and between processes is important.
- Tool integration should be developed to enable independent developers of tools to lever existing platforms to remove the hard problems of scalability and portability.
- Application or model awareness should be investigated - for example improving the understanding of MPI objects and their presentation to users in tools, or the display of task lists with task parallel models.
- Fault tolerance is an area that that should be investigated or prototyped but is expected to be not yet ready at the system level for production level tools activity within the time-frame of the project.

Given the early state of the CRESTA project, we provide first design hints for these architecture and system specific topics, but expect that we can provide a much more detailed design in the second version of this deliverable. We address the state of fault tolerance in Section 4 and present first insights on tool integration in Section 5. Finally we present how we ensure that any extension that we provide is well tested and suitable for production use (Section 6). In the following we first provide a short overview to the two mentioned tools Allinea DDT and MUST, and then highlight the purpose of this document.

## 2.1 Allinea DDT

In terms of debugging the state of the art for scalable debugging is Allinea DDT – which has proven the feasibility of Petascale debugging for the existing PRACE prototypes and other large systems by reaching 220,000 cores (the largest machine at the time of measurement) and provided fraction-of-a-second responsiveness with global operations at full scale.

The Allinea DDT architecture is modern graphical debugging tool with a bespoke tree overlay network for communication and message broadcast and aggregation, and at the leaf (and any debugging nodes) of the tree, a daemon and a full-strength "command line" debugger (usually the open source GDB). This tree architecture has been essential to scalability.

Allinea DDT provides support for most paradigms found in high performance computing - from GPU programming models through OpenMP and MPI to PGAS languages. Platform support is similarly broad - covering the majority of HPC systems in use today.

## 2.2 MUST

MUST (Marmot Umpire Scalable Tool) is an automatic runtime error detection tool for parallel software. Currently it focuses on the detection of MPI programming errors, while its extensibility allows us to advance the tool towards other programming paradigms in the future. Automatic error detection tools are less versatile than debuggers, however, they offer a quick and easy option to detect many error types. The individual error outputs can often include helpful details on the surroundings and details of the error, as well as sometimes its root cause. Finally, crucial is the notion that some errors may not manifest in an actual application crash or hang, for some systems or runs. In these cases silent errors may lead to erroneous calculations that might stay undetected without these tools.

While debuggers already showed their scalability to leadership scale machines, e.g., Jaguar experiments with 220,000 cores, automatic runtime error detection tools fail to scale their more detailed correctness checks to more than 100-1,000 cores. Thus, a primary goal is to advance the scalability of MUST towards at least 10,000 cores and potentially even more. We will detail our design for scalable MPI runtime error detection with MUST in the first part of this section. Furthermore, new paradigms are playing an increasing role in the development of parallel applications. We present a design to extend MUST for the use of other paradigms in the second part of this section.

## 2.3 Purpose

This document presents specific ideas and designs to overcome the identified gaps that current debugging tools have. This particularly includes expectations of changes for exascale systems, which in our case includes the challenges:

- Scale
- Programing models
- Fault tolerance
- Complexity of software and its bugs

The challenges *scale* and *complexity of software and its bugs* can be predicted more clearly than the impact of *programing models* or *fault tolerance*. As a result, we can show more detailed designs to cover *scale* and *complexity of software and its bugs* within this design document. For *scale*, Allinea DDT already supports test cases with 100.000 processes very well, so extensions in this direction primarily address MUST. For *complexity of software and its bugs,* we provide new designs for Allinea DDT to allow easier root-cause analysis and paradigm/model specific insights. The developments in the two remaining areas (*programming models* and *fault tolerance*) remain unpredictable at the moment. As a result, our designs for these problems form a basis for an extension in the second version of the deliverable.

## 2.4 Glossary of Acronyms

| | |
|---|---|
| **MPI** | Message Passing Interface |
| **TBON** | Tree Based Overlay Network |
| **MUST** | Marmot Umpire Scalable Tool |
| **GTI** | Generic Tools Infrastructure |
| **P2P** | Point-to-point |
| **WFG** | Waiting-for graph |

# 3 Debugger Design Proposals

We will propose to address the core problems identified as a result of the survey and initial document D3.1 ('State of the art and gap analysis') and their impact on the debugger architecture.

## 3.1 Programming Models

The trend in hardware is presently towards heterogeneity - in many forms. We already see GPUs or the latest Intel Xeon Phi as accelerators that are located on PCI Express buses, and in time it is likely such components will become more closely integrated. Processor platforms normally outside of HPC such as ARM are also adopting heterogeneity with concepts for power efficiency such as big.LITTLE - whereby a number of cores are only powered up at points within an application that can use the concurrency. Formerly separate components such as the GPU have moved into the die with products such as the AMD Fusion.

This heterogeneity requires programmer intervention in order to maximize performance (or even to access the hardware in most cases). In particular - with offloaded computations there is a need for the ability to debug synchronous and asynchronously launched offloaded computation at the same time as debugging the host processor on the node. The hardware combination is part of a larger system with multiple nodes - and the software must run across the whole platform simultaneously.

The most common form of hybrid code until recently was typically OpenMP within a node, MPI between nodes - this has changed with CUDA or OpenCL now as an alternative to the OpenMP - often leaving multiple cores on a node idle whilst a GPU executes an (eg.) CUDA kernel. This *may* change with the advent of the Intel Xeon Phi architecture which supports – amongst other programming models - OpenMP - and could lead to reestablishment of OpenMP + MPI as a dominant combination. The processor roadmaps of the main vendors will have a clear but as yet uncertain impact on the development choices made by application developers.

For a debugger such as Allinea DDT this rise of heterogeneity should be considered and in particular:

- New platforms must be supported as a route to understanding the impact on programmability and debugging. Architecturally the designs for the changes depend on the platforms expected for exascale - but in the immediate term systems such as accelerators from NVIDIA and Intel are likely candidates.
- The control of thread (including GPU thread) level parallelism within a node should become more intuitive - allowing the same form of control that a user has over the MPI processes.
- The display of data as shown for MPI parallelism within Allinea DDT must include similar thread-parallel displays also - for example automated comparison across threads is a worthy addition to comparison across processes.

New programming models other than these mainstream coprocessor models did not have sufficient interest from the application developers in CRESTA to consider at this point in time.

## 3.2 Automated Anomaly Detection

Automatic identification of anomalous values is becoming important. Firstly, users have expressed concerns – in both the survey for the existing Gap Analysis document, and we are aware anecdotally, that the volume of data is increasing as applications grow and is already unmanageable for many applications. Secondly, debugging is both deductive and iterative, and yet iteration is not a process that we humans do well. At

current scale, and as we reach higher scales, we can automatically identify anomalies that happen - differences with previous successful runs, and with processes that are successful within the current task. Identifying earlier that a value is invalid would be helpful even at current application scale.

Identifying anomalous application activity is also important - current approaches, for example, viewing merged stacks of processes are helpful but need to be extended. Identifying, for example, the path of execution that led to a particular issue would be helpful. This may take the form of comparing with a previous successful application run. This could cover both data changes, and process activity. Automated methods for asserting data integrity should be investigated that would allow, for example, a developer to efficiently detect incorrect values. This could involve developing both standard libraries for data verification, and model specific libraries and, for example, the addition of consistency checks support - for example a checksum capability on function entrance for large arrays.

This would mean exploring or develop in-process methods for check-summing: vast data-sets require the fastest methods to access data. Running at the debugger layer through OS debugging APIs could be too slow but an in-process dynamically loaded library could provide fast support.

Logging and comparing different runs is another desirable way of finding why two codes or runs have given different results, but the naive implementation is too slow. If each function call creates a delay to log the function-event, then debugger-level logging will impose too much delay. Compiler-level instrumentation is an option - but not universally supported. Therefore, dynamic instrumentation needs to be compared against selective debugging-level interruption.

Application/Library Model Awareness: Better integration of layered models and the debugger should be investigated with, for example, awareness of MPI communicators and the internals of request object or integration with runtime of task based parallel frameworks to visualize internal task lists.

The first example to be tackled will be MPI handles. Presently there is no standard for MPI handle debugging - this meaning the request objects or communicators are typically “opaque” to the debugger and provide no useful information for the user.

- TU-Dresden has a wrapper library to MPI and a series of GDB python scripts that could be integrated into DDT. The wrapper library wraps calls to the MPI handle creation, for example MPI communicator objects, creating its own MPI objects and then the gdb python scripts work to provide more sensible “pretty printing” of the internal objects. This python based printing of variables is already supported by DDT for global settings by virtue of it using the GDB debugger underneath. The pretty printers and wrapper library will fit within a modest extension to the existing preload plugin capability in DDT.
    - Add support for scripts using the python API for pretty printing in the definition files for the plugin architecture.
    - For specified plugin scripts, ensure support for every MPI by transferring the python scripts to compute nodes in systems where necessary.
- Investigate the proposed MPI 3 Handle Extension (for inclusion in the MPI 3.1 standard) whereby information similar to the TU-Dresden library is included by default in the MPI instead and the debugger queries an interface similar to the MPI Message Queue Debugging API to explore the handles. Discussions and designs for this are already in draft form between Allinea and the Open MPI team. A prototype is needed to consider whether this method is appropriate.

Alternative models beyond MPI - such as OpenMP 3 Task Parallelism - can be considered in the next design document.

Extensions to the debugging interface to enable more application aware debugging will also be considered and reported in the next design document. In particular we will investigate

- Specific library hooks to query consistency of internal data types.
- Specific library code or support for representing the opaque data at a higher level.
- Use of an API to enable debugger/runtime interaction - for example exposing a programmable interface to the debugger.

# 4 MUST

We describe MUST's extensions towards Exascale needs in this section.

## 4.1 Scalability

MUST analyses MPI events in order to analyze their correctness in terms of the MPI standard specification. An important notion is that two types of correctness checks exist:

- Local checks: Only require information from one process, e.g., "Is the datatype in the MPI_Send call committed?" and can be executed directly on the application processes; and
- Non-local checks: Require information from more than one process, e.g., "Do all processes in the MPI_Bcast call use the same type signature?". These require a communication infrastructure to gather all the information that is necessary to run the correctness check.

The local checks satisfactorily scale with the number of MPI tasks, while the non-local checks impose scalability challenges. Prior correctness tools such as Marmot [4], Umpire [5], and ISP [3] must use a single process or thread to run all non-local correctness checks, which is a major scalability limitation. MUST includes both types of correctness checks as well. The centralised design allows correctness tools to handle 100-1,000 cores at most. We propose a design that is intended to scale to at least 10,000 cores, while its extension to higher scales should be feasible. We will first present an overview of our design and then detail its individual components.

This design also reflects progress within an ongoing cooperation with the Los Alamos National Laboratories and the Lawrence Livermore National Laboratories. While these cooperations focus on achieving low runtime error detection overheads for about 10,000 cores, we want to extend MUST scalability as far as possible within the CRESTA scope. The experience with the design below and its limitations for actual test cases will motivate a more detailed layout for further scaling in the second version of this design document.

### 4.1.1 Component Overview

MUST uses a module approach, where distinct modules implement particular tasks. These modules have clearly defined interfaces and cooperate with each other. The second important functionality in MUST is the use of Tree Based Overlay Networks (TBONs) that allow us to distribute workload onto multiple levels of extra processes or threads. Both the module and the TBON concept are in fact not implemented within MUST, but rather in the infrastructure it employs, which is called the Generic Tools Infrastructure (GTI). TUD develops GTI along with the Lawrence Livermore National Laboratory and the MUST tool is currently its key use-case. This design describes how we can use GTI's TBON functionality to derive distributed non-local correctness checks.

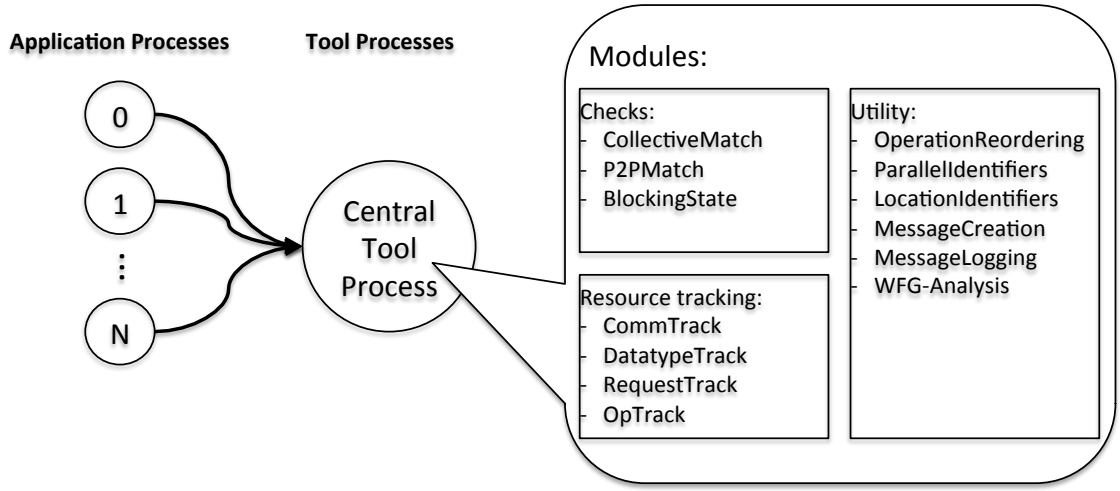


**Figure 1: Centralised correctness checking with MUST**

In terms of non-local correctness checking, MUST uses the layout and the modules in Figure 1. The left side of the figure illustrates the central tool layout of MUST. A single process receives MPI events from all application processes. The tool process runs the modules that are listed on the right side of the figure. This includes the following modules for correctness checking:

- CollectiveMatch: checks MPI collective calls for their correctness;
- P2PMatch: checks MPI point-to-point calls for their correctness; and
- BlockingState: detects deadlocks.

The other module groups “Resource tracking” and “Utility” are used by these correctness checks to function correctly. While the three correctness checking modules assume that they receive events from all MPI processes, all the other modules can already be used in a distributed fashion. So our design focuses on distributing these three modules.

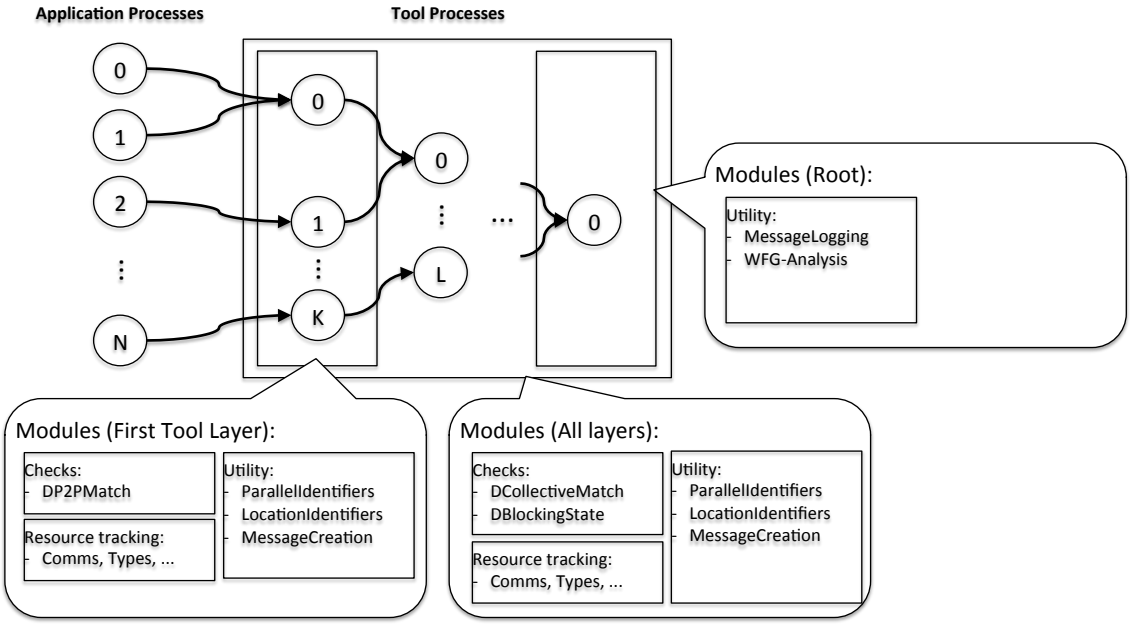


**Figure 2: Distributed correctness checking with MUST**

Figure 2 illustrates the distribution of the non-local correctness checks, it shows the application processes at the left and sketches a tree based overlay network to its right. Our design will analyze point-to-point matching on the first non-application layer of the TBON. Collective MPI operations and deadlocks will be analyzed throughout the complete TBON. Finally, the root process of the tree will run the actual graph analysis that we use for deadlock detection and it will log correctness messages. We detail

these components in the following. Note that we simplified the modules for the distributed wait-state tracking.

### 4.1.2 Distributed P2P Analysis

We propose to match and analyze the correctness of point-to-point messages on the first non-application layer of GTI's TBON. The key observation is that a distributed message matching in the whole tree is unsatisfactory [2]. The limitation is that the root would have to match 50% of all possible communication pairs for general communication patterns. Thus, depending on an application's communication pattern, severe load balancing problems might arise. As a result, we instead only run this analysis on the first tool level of the TBON. In order to communicate information between these tool processes we use a so-called intra-layer communication.

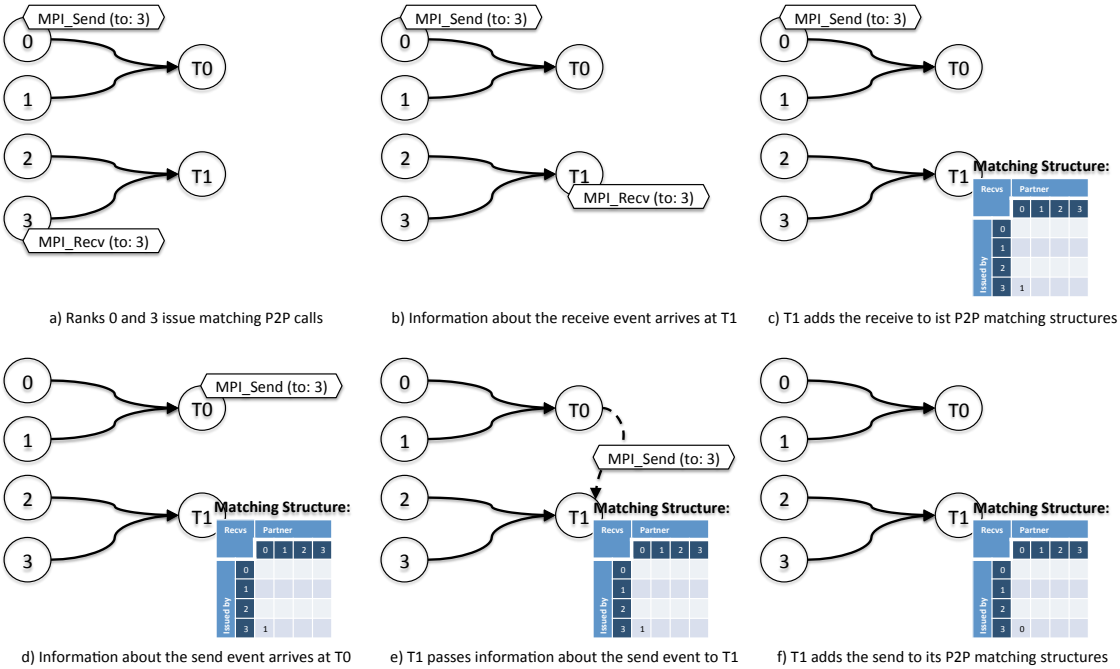


**Figure 3: Illustration of intra layer communication based P2P matching**

Figure 3 illustrates the intra layer communication based point-to-point matching that we propose. Figure 3a shows that rank 0 issues a send that has rank 3 as its destination, while rank 3 issues a matching receive for this send. Once issued, the events that represent these MPI calls can arrive in any order on the tool processes T0 and T1. As this example uses a binary TBON, the receive event of rank 3 will arrive at T1 (Figure 3b). We directly match receive events on the tool processes that process them, i.e., T1 in our example. T1 will analyze the receive and determine that it is not aware of a matching send call for this event, as a result, it adds the event to an internal data structure for point-to-point matching, which we illustrate with a small table in Figure 3c. When information about the send event arrives at T0 (Figure 3d), we determine that the matching receive for this send will arrive at T1 instead of T0. So in order to match the message we transfer the send event information to T1 (Figure 3e). Finally, when the information on the send arrives at T1, this process determines that a matching receive is available. During this matching T1 can run any necessary correctness checks, e.g., type matching. Note that this requires not only to transfer specific send events within a TBON layer, but also requires us to transfer information about MPI resources—such as communicators or datatypes—within the layer. This complicates the implementation of the design, as MUST's resource system needs to be aware of such "remote" resources.

We propose to transfer send events with intra layer communication as MPI's push semantic guarantees that each send call specifies a destination rank, whereas receive events might specify that they match a send from any process

(source=MPI_ANY_SOURCE). Thus, using intra-layer communication for receive events would add unnecessary complexity.

This design relies on the availability of an intra-layer communication mechanism within a TBON. Our experience with MUST and GTI indicates that this poses no severe restrictions. MUST currently uses an MPI-based TBON communication system that utilizes an MPI_COMM_WORLD virtualization. Using an intra-layer communication in this setting is straight forward. The actual intra layer communication can be implemented by regular communication protocols of GTI.

### 4.1.3 Distributed Collective Analysis

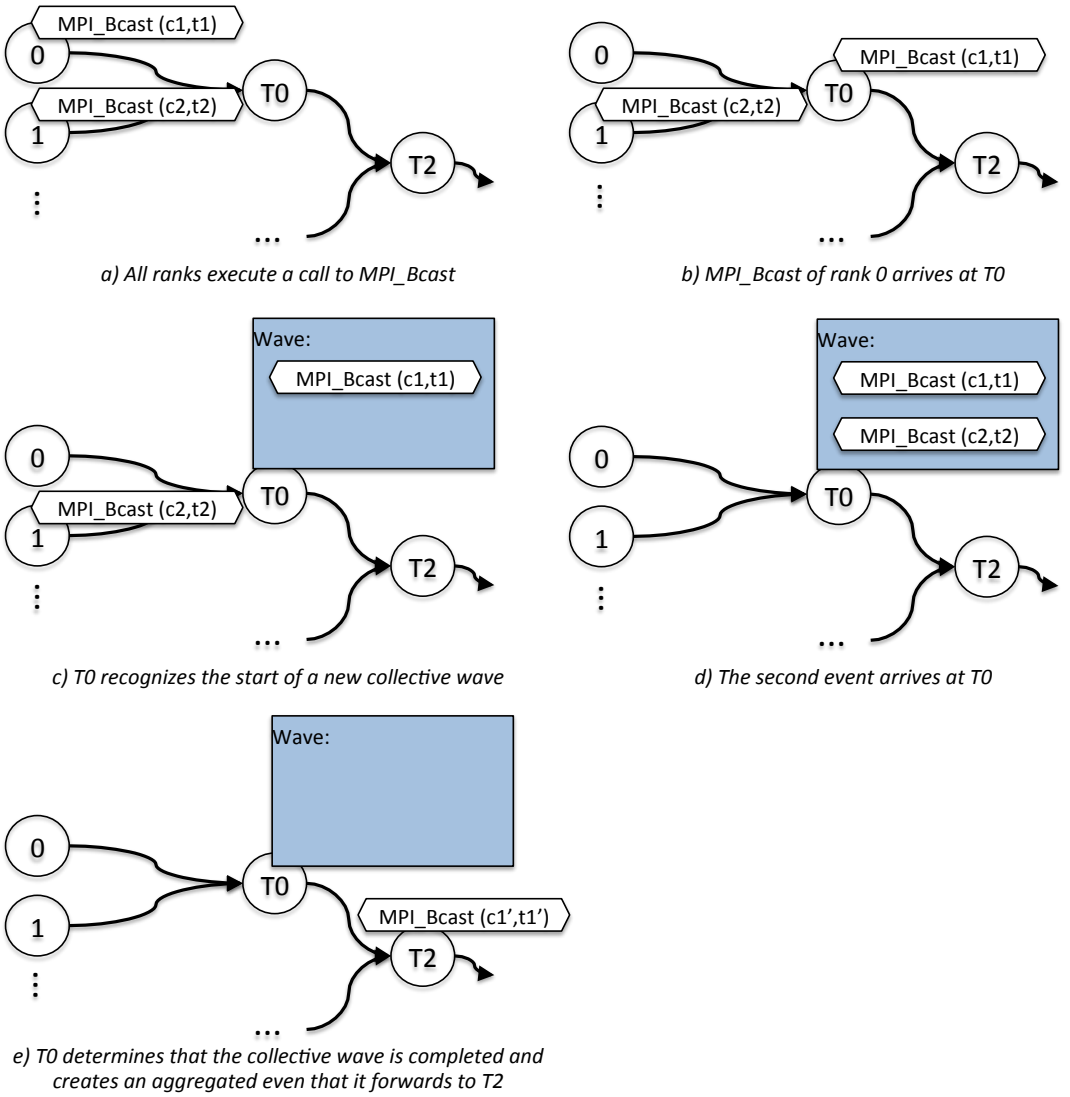


**Figure 4: Illustration of event aggregation for MPI collectives**

We propose to analyze the correctness of MPI collective events within GTI's TBON. Most correctness checks for MPI collective operations can use aggregations for their implementation. Figure 4 illustrates this concept, where each application process issues an MPI_Bcast operation (Figure 4a). When the first MPI_Bcast event arrives at T0, it recognizes this as a new wave of events and creates a respective data structure. Note that the event is not forwarded towards T2 (Figure 4c). When the second event arrives at T0 (Figure 4d), the TBON node determines that it received all events that belong to this wave and runs all correctness checks (Figure 4d). Finally, it creates a new event that represents the information from the two incoming events. For MPI collective operations this is possible without increasing event size. This event is then forwarded to the next TBON layer (Figure 4e).

An analysis of all MPI collective operations reveals that all non-local correctness checks can be distributed in this fashion and that event aggregation with constant event size is feasible. The only exceptions are communication calls where pairs of processes can use distinct type signatures, this includes:

- MPI_Scatterv,
- MPI_Gatherv,
- MPI_Alltoallv, and
- MPI_Alltoallw.

Note that MPI_Allgatherv is not included in this list, as it requires all processes to span the same type-signatures with its count array. Handling these four calls with the aggregation mechanism would require us to store arrays of type signatures in the aggregated events. This would first of all lead to a non-constant event size and furthermore also lead to a load imbalance, as the root of the TBON would have to execute a majority of the type matching checks. As a result, we propose to handle the type matching for these four operations with intra-layer communication.

For MPI_Scatterv and MPI_Gatherv, the root process needs to scatter the count array along with the datatype in use within one TBON layer that provides intra-layer communication. For MPI_Alltoallv and MPI_Alltoallw, each process scatters its send-counts array and its type(s) across a TBON layer. This handling should not exceed the complexity of the original MPI communications, and should thus provide an acceptable overhead. Note that these four calls also have expected scalability limits for Exascale needs, as they use arrays that are sized according to the number of processes in use.

### 4.1.4 Distributed Wait-State Analysis

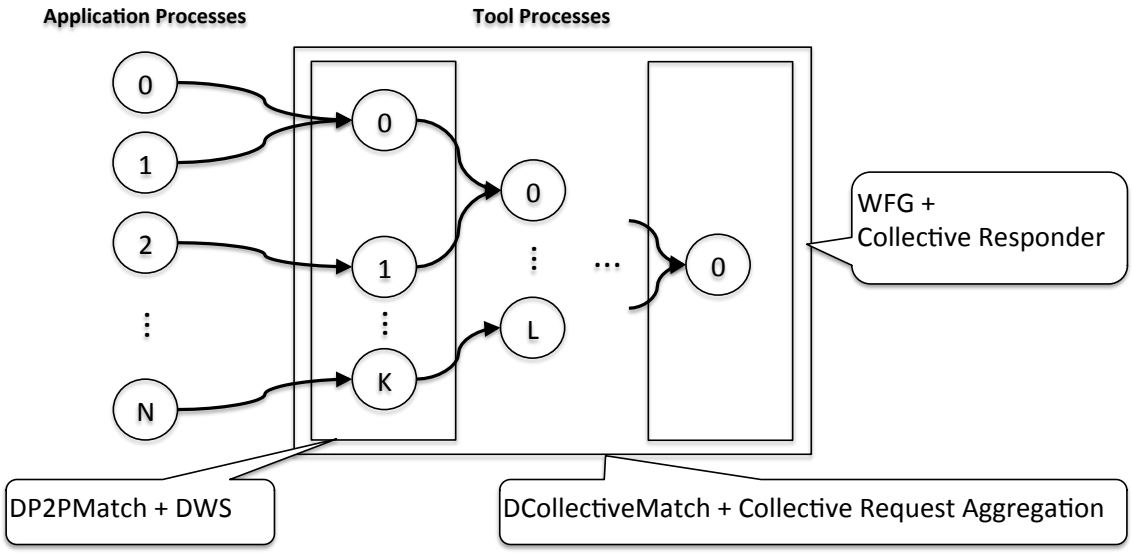


**Figure 5: Illustration of distributed wait-state modules**

The most challenging non-local analysis is the MPI deadlock detection. It basically consists of two parts: a wait-state tracking and a graph based deadlock detection. The wait-state tracking causes the higher overhead, as it needs to consider each single possibly blocking MPI operation. It then decides whether the current operation can complete, i.e. if all of its matching operations can actually be issued. The graph based deadlock detection is only executed if we suspect the presence of a deadlock, and is thus the less critical overhead. We propose the following design to distribute the wait-state analysis:

- Each P2P and collective operation gets an associated timestamp that captures their order within a process

- P2P and collective matching add matched operations to queues of completed operations—Completed [operation] Queue (CQ)—operations that are ordered by the timestamp of these operations
- The Distributed Wait-State (DWS) analysis runs on the same layer as the P2P matching (Figure 5, first tool layer)
- DWS starts at the beginning of the CQ and determines whether this operation can safely complete
- The current operation considered by DWS in the CQ is considered the “active” operation
- If the active operation can complete, the next operation in the queue becomes the active operation
- DWS on a TBON node uses upstream and intra-layer communication to determine whether the active operation can complete, that is whether all matching operations also became active already (Figure 5, “Collective Request Aggregation” and “Collective Responder” Modules cooperate with DWS for that)

The root of the TBON runs a centralized graph analysis (Figure 5, WFG Module).

Note that the layer on which DWS runs is only aware of whether all connected processes called a certain collective call, but not whether all other ranks also called the collective call. We propose to use a communication directed towards the root for that, along with a downwards broadcast that is started by the root once that that all processes indicated that the given collective became active.

In summary we use the following requests for the intra-layer and up/down communication:

- Collective-Became-Active [Upwards-Aggregated]
- Collective-Commit [Downwards-Broadcasted]
- P2P-Became-Active [Intra-layer]
- P2P-Commit [Intra-layer]

When a collective operation becomes active on a node that runs DWS, it issues the request “Collective-Became-Active” and sends it towards the TBON root. This event is aggregated along its way if its communicator and timestamp—timestamp within the communicator—match. Once the root received a complete wave for this request it sends a notification event “Collective-Commit”. When DWS receives this request it can advance its active op to the next operation in CQ. When a P2P operation becomes active on a DWS node and the operation is a receive operation, it sends a “P2P-Became-Active” request to the node that hosts the send that matches the receive operation. Once the DWS node that hosts the send operation receives this request, while the given send also became active or was already completed, it sends the “P2P-Commit” back to the node that hosts the receive. The DWS node that hosts the sender op can advance the active op once it saw the “P2P-Became-Active” request, while the DWS node that hosts the receive op can advance its active op when it receives the “P2P-Commit” request. Note that less communication is necessary if the send and/or the receive operation is non-blocking. Also for completion calls, e.g., MPI_Waitall, array versions of such requests will be necessary.

Finally, to actually detect deadlocks, the root of the TBON runs the WFG module. This module uses a timeout to start deadlock detection in given intervals. When it starts a deadlock detection it requests wait-for dependency information about the active operation of all DWS nodes, for that it uses the following request:

- Request-Active-Ops [Downwards-Broadcast]

The DWS nodes answer with the requested wait-for information that they send back to the WFG module. The WFG module waits until it received all the wait-for information,

applies it to a WFG, and then runs a deadlock detection on this graph. Note that this graph analysis is currently still centralized, but should scale to 10,000 cores at least.

### 4.1.5 Asynchronous Communication

Besides the distribution of MUST's non-local correctness checks it is important that MUST uses a highly efficient communication means. In particular, experiments indicate that application events need to be transferred in a non-blocking fashion, i.e., the application must be allowed to continue its execution while MUST evaluates new events for correctness in parallel. However, in the presence of application crashes this may result in MUST not detecting a usage error before the application crashes. Thus we need an asynchronous communication medium that operates while the application crashes.

We propose the following design to handle this:

- On each computing node of the system we use one core for a MUST tool process
- The remaining cores on the nodes are used for the application processes
- These processes that are on the machine nodes of the application form the second TBON layer (application is the first layer)
- All other layers are distributed across the remaining nodes
- Communication between the application processes (first layer) and the tool process on the second layer uses shared memory
- The second TBON layer communicates via MPI with the remaining layers

We then use signal-handlers and error-handlers on the application processes to catch an application crash. These handler routines notify the communication system that a "panik" signal needs to be raised, which is forwarded to the root of the TBON. The root then broadcasts a "complete-analysis" signal downwards in the tree. When nodes receive this signal they must receive all available incoming events, process them and then finish their execution.

GTI's flexibility in its communication protocol and timing allow various platform dependent strategies for application crash handling. The use of shared-memory between the application layer and the first tool layer appears to be a most promising and portable selection.

### 4.1.6 Expected Impact and Limitations

In summary we propose distributed systems for:

- Point-to-point matching (DP2PMatch);
- Collective matching (DCollectiveMatch); and
- Wait-state tracking (DWS).

In addition we propose an asynchronous communication system that allows MUST to detect MPI usage errors even if the application crashes.

The distributed systems should scale to 10,000 cores and beyond. While this may still be low for Exascale applications, this is a major improvement to the previous centralized approach. Two factors should also be considered for this scalability level: First, Exascale applications are likely hybrid and may thus use less processes than cores are available; Second, Even if MUST does not supports full system test cases, it can at least support smaller test or debugging runs. Finally, it may be an option to only run some of MUST's correctness checks to improve its scalability where necessary.

Our performance expectations are based on the following analysis. DP2PMatch executes a coarsened version of the applications communication pattern. Even if each node that executes DP2PMatch receives events from multiple application processes, it can provide low overheads as its intra-layer communication can use event buffering to

use high-bandwidth communication. DCollectiveMatch needs to run a TBON based aggregation, however if we fix the fan-in (number of application/TBON-nodes connected to a TBON node), the number of events to process stays constant with increasing scale. As our event aggregation keeps the event size constant, the cost of analyzing a single event also stays constant. The DWS system will likely have the highest overhead, it uses the same communications as DP2PMatch and DCollectiveMatch in combination (In order to communicate the different request types). However, it uses the downwards directed broadcast in addition. The main difference here is that DWS always sends one or multiple requests, and then waits for a reply; whereas DP2PMatch and DCollectiveMatch can continue their execution irrespective of the availability of a reply. So the DWS communication will be more latency bound than the communications of DP2PMatch and DCollectiveMatch. An evaluation of a DWS prototype must show whether this design leads to acceptable overheads.

Besides these distributed components, we see potential scalability limitations in:

- Resource tracking
- Graph-based deadlock detection

MUST's distributed analyses require information about MPI resources (communicators, datatypes, requests, groups, reduce operations), currently we communicate information about these resources to all TBON nodes that need them. For 10,000 cores or more this might lead to expensive bookkeeping, depending on the number of MPI resources that an application uses. We will investigate this and propose extensions to MUST's resource system in the second version of this deliverable if necessary.

The graph-based deadlock detection runs on the root of the TBON, for N processes it will need to analyze a graph with about N nodes. As a result, this is a scalability bottleneck, but its impact depends on the number and type of the active wait-for conditions. Furthermore, we only rarely execute this analysis, so it only needs to return within an acceptable runtime. We will investigate the resulting overheads and also propose design extensions if necessary.

## 4.2 Paradigms

As architectures become more complex and heterogeneous, new parallel programing paradigms and abstractions arise. Automatic correctness support for these paradigms is highly desirable, but requires that automatic error detection tools understand the paradigm in question. This both requires an instrumentation mechanism to intercept correctness relevant events, and to add new correctness checks for these events. MUST's infrastructure GTI allows us to easily add new types of correctness checks. However, the instrumentation system is very dependent upon the paradigm and can't easily be generalized within GTI.

With the limited resources within the CRESTA project it is unrealistic to advance MUST towards more than one paradigm. As there is currently no clear candidate for this paradigm, we sketch first steps towards support for additional paradigms in this design document. This specifically includes Co-Array Fortran as a PGAS language and CUDA as an accelerator-based paradigm.

### 4.2.1 PGAS Language Correctness Checking

PGAS languages are available as library-based paradigms, e.g., GPI [6], and as language extensions, e.g., UPC, Co-Array Fortran, and Chapel. In terms of instrumentation, library based PGAS paradigms are easier to handle, as they directly provide an interface to intercept. Language based PGAS paradigms are compiled into an intermediate form and implemented by some communication layer. Intercepting events is more difficult in this case.

The types of correctness errors that occur in PGAS languages appear to be more typical to threading errors as detected by tools like the Intel Thread Checker, e.g., a

write to remote memory happens in parallel to a local read to the same location on the remote side. So detecting all types of PGAS usage errors will require tools to trace each single memory access, which we assume to be too expensive for a scalable runtime tool. So advances in fine-grained memory race detection are outside the expertise and resources available within the CRESTA project. However, further errors of PGAS applications include synchronization errors like concurrent write operations to equal memory regions or a lack of synchronization primitives between DMA operations. Finally, due to locking and barrier synchronization, deadlocks may also manifest in PGAS languages. The later two can also be detected without tracing each single memory access. So first correctness checks within MUST would more likely fall into the detection of coarse-grained synchronization errors and deadlocks. If low overhead memory access tracing is available, experiments with per memory access based error detection could be possible.

From our early experience, for first correctness checks of a language based PGAS paradigm, we would require an instrumentation API that provides us with information on:

- Memory ranges that are global
- Synchronization calls
- Communication/writes into global memory
- Communication/reads from global memory

Details of such an API would be added if a language based PGAS paradigm becomes a primary candidate for application development within CRESTA.

### 4.2.2 CUDA/OpenCL Correctness Checking

Approaches such as CUDA and OpenCL are library-based extensions, which simplifies their instrumentation. The key difference to other paradigms is that both CUDA and OpenCL use a host and a kernel language. The kernel language is usually a modified subset of an existing language like C or C++. Instrumentation of kernel language events is very challenging as the functionality that is available on the kernel level is very limited. As a result, correctness checks of MUST for accelerator languages would focus on the host side. This primarily includes process local correctness checks, e.g., is the device or the kernel in the correct state for a certain API call. Also, this includes a few potential non-local checks, e.g., are no two processes using the same GPU device. Note that such an extension would still provide MPI correctness checking to MPI/GPGPU hybrid applications.

### 4.2.3 Generic Correctness Checking Overview

GTI is agnostic of any particular programing paradigm, i.e., not specific to MPI tools. However, it is currently only used for MPI runtime tools, and may thus still need extensions to handle other paradigms. Once GTI can be used for a certain paradigm, a tool developer can specify which events he intercepts, which may require extensions to PnMPI (A basis infrastructure used by GTI). Afterwards, the tool developer can directly specify analysis modules, which can implement correctness checks in the case of runtime error detection. In summary, GTI is prepared for handling other paradigms, but will require instrumentation extensions (PnMPI), possible extensions for GTI itself, and a MUST extension to add the actual correctness checks, which we describe in the following.

PnMPI is a virtualization of the MPI profiling interface, it allows multiple tools to intercept MPI events at the same time. GTI uses PnMPI to intercept MPI calls and manage its modules. In order to handle additional paradigms, PnMPI needs to be able to intercept events of this paradigm. For CUDA, OpenCL, or a library based PGAS language, this requires us to extend PnMPI such that it creates a wrapper library to intercept any event of interest. In the case of a language based PGAS paradigm, PnMPI needs to provide hook or callback functions that work together with a vendor

provided instrumentation API, or with a source to source translation that adds events to the PGAS language.

Based on a specification of events to intercept, GTI creates wrapper functions to fetch events that are already intercepted by PnMPI. If PnMPI provides additional instrumentation mechanism, GTI needs to be extended such that it is aware of these additional mechanisms. Our specification for events to intercept needs to distinguish the interception mechanism in this case.

Further, our primary communication protocol for GTI uses MPI based communication, in the case of a pure PGAS/OpenCL/CUDA application; this communication medium may not be available. So depending on the paradigm of choice, it may be necessary to add additional communication protocols to GTI.

Finally, to make use of such PnMPI and GTI extensions we need to develop an event specification for the new events of interest within MUST. In addition we need to implement the new correctness checks and map them to the events that we describe. Furthermore, depending on the paradigm, it may be necessary to provide certain base services or conversion function to convert some event arguments into representations that can be analyzed on all nodes of our TBON.

# 5 Fault Tolerance

Expectations [7] for Exascale systems indicate that mean-time-to-failure may lower than a day. As a result, applications, systemware, operating system, and any type of runtime tool need to be aware of possible failures and may also need to recover from them. This may include the use of spare nodes or cores to replace failed components.

These effects need to be considered for the development of debugging and runtime error detection tools for Exascale systems. Work package 2 evaluates operating system and programming model changes to handle such hardware faults in D2.5.2, which will be released in month 30 of the project. Thus, at the current project state, there is close to no indication of what these mechanisms and designs might look like. This includes an indication of a mean-time-to-failure that is to be expected. Also possible advances in hardware may render these concerns unnecessary altogether. As an immediate consequence we will not address any modifications for fault tolerance in this design document. Our hope is to use the insights that are available around project month 20, to include them in the second version of this design document (month 24).

# 6 Tool Integration

While individual tools may provide application developers or system administrators valuable insights, there is usually a high reluctance to learn and understand new tools. At the same time, combining multiple tools may lead to deeper and more meaningful insights than with just a single tool. Debuggers and runtime error detection tools are an example for such a case. The obvious integration direction is to incorporate an automatic error detection tool into the debugger, this leads to the following benefits:

- Tool user only needs to learn the debugger usage
- Errors detected by an automatic runtime tool can directly be investigated with the debugger
- Automatic runtime tools may share knowledge with the debugger

As a result, we want to investigate potential integrations between DDT and MUST in order to provide these advantages to the tool users. An early integration between DDT and Marmot (a predecessor of MUST) already showed a first integration approach that allowed Marmot to stop the execution when it detected an error. Afterwards, the user could investigate error details with the debugger. An important notion is that both MUST and DDT rely on the use of a Tree Based Overlay Network (TBON), an interesting question is whether an integration could allow the tools to share this infrastructure for easier deployment and lower resource consumption. In theory, graphical user interfaces within DDT could control the behavior of MUST and even apply certain correctness checks to particular data. However, while such integration is extremely promising in terms of usability and reduced time to solution, the individual DDT and MUST extensions in the preceding sections are crucial to provide helpful debugging tools for Exascale. Thus, the integration stays an optional research direction that we can only follow if progress in the other development areas is successful.

Sharing TBON functionality between the tools is a key for a deep and long lasting tool integration. As this is a basic and crucial component in either of the tools, it is important to identify similarities and differences between the TBON usage and instantiation in the two tools. We want to identify integration goals as a first step in this design document. Afterwards, we will present the existing approach and first insights for sharing a TBON.

## 6.1 Goals

Tool integration between DDT and a tool like MUST should provide the following functions at least:

- Starting MUST as a plugin within DDT (within DDTs user interfaces)
- Stopping the debugger if MUST a detects errors
- Displaying MUST's output within DDT
- Exporting environmental variables that control MUST/GTI/PnMPI

These basic features allow users to load MUST into DDT and to gain basic benefits of an integration. The next section describes details on how we implemented such a first integration. The following complications arise for such a basic integration:

- MUST performs a code generation steps before it runs with a certain application. This code generation also needs to be handled by an integration, a straight forward solution is the usage of MUST's own utilities to perform this generation step.
- MUST starts the MPI application with additional processes per default, their presence will confuse DDT users
- In order to operate in a scalable and fast manner, MUST needs to use asynchronous communication methods. As a result, it will usually detect non-local correctness errors only after the application stepped over the respective

MPI calls. In such a case, stopping the application where the error happens is not directly possible.

After handling the basic integration goals we must address these three issues to improve the user experience with such an integration.

Once we handled these first steps additional more advanced integration goals could include:

- Sharing TBON functionality between DDT and MUST
- Allowing the specification of user driven runtime checks within DDT, MUST could be used to implement these checks
- Allowing user to select certain correctness checks for MUST, i.e., advanced and tool specific options that can be selected during the DDT run configuration
- MUST can provide DDT with information about MPI state, e.g., for MPI handles (See Section 0)
- Sharing static/debugger knowledge, e.g., DDT might tell MUST the type of a variable; This can enhance the precision of some MUST checks

## 6.2 Enabling Tool Integration

Allinea DDT has a basic capability for integrating simple components that operate purely via shared library preloading. This has been used previously in the ITEA PARMA project for integration of Allinea DDT and the forerunner of MUST, MARMOT - but also supports the Intel Message Checker which targets a similar objective of MPI usage verification.

The plugin model as it currently stands is able to—via an XML configuration file—specify libraries to preload and default breakpoints and tracepoints. At a default breakpoint, a message - consisting of a string and severity level can be shown to the user. Error checkers will call this function in their ordinary course of execution - but when running in the debugger the default breakpoint action will then cause a message dialog to be shown to the user.

For example, the Intel Message Checker plugin file consists of this small XML file.

```
<plugin name="Intel Message Checker 7.1" description = "Enables
MPI message checking when using Intel MPI 3.0 or later">

     <preload name="libVTmc.so" />

     <breakpoint location="MessageCheckingBreakpoint"
action="message_box" message_variable="error" />

</plugin>
```

When the Message Checker detects a problem the usage error is shown inside Allinea DDT with the application still “alive” - enabling the full context of the application to then be understood by the programmer.

As Allinea DDT is able to launch applications with any MPI and understands how to do library preloading for each of the implementations, this enables any tool that uses the MPI profiling interface (PMPI) to work together and be configured to run with no effort from the user.

The aim of this part of the CRESTA project is to enable an extended mechanism to work for tools such as MUST that require scalable infrastructures such as a tree network.

The design of the necessary integration components and APIs that need exposing is underway but not complete and will be refined over the next three months by Allinea and TU-Dresden.

Initially we have begun to “clean” (refactor) various parts of the existing DDT code-base to enable easier access of 3rd party components to Allinea DDT. We have used the infrastructure with a profiling plugin also - to demonstrate that another tool quite distinct from debugging can use such an infrastructure. This enabling work will help to include other tools.

The design required to support tools for scalability must expose scalable tree operations - and hence the current simple framework is inadequate.

Exposing the tree capability requires:

- The ability to declare a library and support dynamic loading of the library into the tree network at the tree nodes
- The library will need to conform to an API (yet to be defined) to allow the operators - merge/broadcast functions in the tree to apply to specific defined message types.
- Additional launch issues—for example any impact on the application on the numbering or number of MPI tasks—as some MPI tools define extra processes as part of their internal processing.

# 7 Testing Plan

We categorize the software extensions that we include into three fields:

- Allinea DDT extensions
- MUST extensions
- Integration components

Extensions to Allinea DDT and MUST can be tested separately while any integration component can only be tested where both tools are available. This also motivates the use of a plugin concept for integration, as a DDT installation should not require the presence of a MUST installation. Testing goals either include functionality tests or performance/scalability tests. We will detail how we will test both tools for correct functionality in the following. For performance and scalability purposes we will use all qualifying tests of the benchmark suite from WP 2 to test both tools in regular intervals. Further, as co-design teams identify successful or promising use cases for Allinea DDT or MUST, we will include these respective co-design applications into regular performance tests. For tests that address the integration of the two tools we will identify a system where both tools are available. We will use regular tests on this system to test the tool integration.

## 7.1 DDT Functionality Tests

Allinea DDT has a comprehensive testing suite, which also includes remote testing to enable access to more extreme machines such as those provided by vendors such as Cray, SGI and IBM. This will be extended with test cases specific to the extensions proposed here. Presently roughly 120 compound test cases exist which are each run against about 10 machines with 6 MPI installations and circa 4 compilers. This is in addition to many unit tests within the code.

One or two of the (more liberally licensed) applications from the CRESTA project will be included as part of the test applications and specific tests build for these.

The test suite is driven by a Javascript-like interface to provide full depth integration testing of DDT—with an API enabling basic debugging operations and GUI state to be driven and queried.

## 7.2 Must Functionality Tests

MUST uses CTest for automatic testing. We currently have a total of 697 test cases that are constituted by various correct or incorrect MPI applications. The test cases include the use of the current centralized MUST components and the use of early prototypes of distributed checking components. We run these test cases on 3 different systems every night and summarize the test results on a dashboard. The test cases also measure the required runtime which allows us to detect overall runtime changes. In order to test MUST with more complex applications we use SPEC MPI2007 and the NAS Parallel Benchmarks (NPB) after each larger functionality extension.

# 8 References


[1] Joachim Protze, Tobias Hilbrich, Andreas Knüpfer, Bronis R. de Supinski, Matthias S. Müller, "Holistic Debugging of MPI Derived Datatypes", Parallel Distributed Processing Symposium (IPDPS), (2012)

[2] Tobias Hilbrich, Matthias S. Müller, Bronis R. de Supinski, Martin Schulz, Wolfgang E. Nagel, "GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems", Parallel Distributed Processing Symposium (IPDPS), (2012)

[3] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, Robert M. Kirby, “ISP: A Tool for Model Checking MPI Programs”, PPOPP, (2008)

[4] Bettina Krammer, Matthias S. Müller, “MPI Application Development with MARMOT“, PARCO, (2005)

[5] Jeffrey S. Vetter, Bronis R. de Supinski, “Dynamic Software Testing of MPI Applications with Umpire“, Supercomputing, (2000)

[6] Franz-Josef Pfreundt, “GPI and MCTP”, Fraunhofer Institut für Techno- und Wirtschaftsmathematik ITWM, http://www.gpi-site.com/cms/sites/default/files/GPI_Whitepaper.pdf, (accessed June 2012)

[7] Herbert Huber, Riccardo Brunino, “D4.3 Working Group Report on Hardware roadmap, links and vendors”, European Exascale Software Initiative, (2011)