D3.4.2 – 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 Exa-scale computing, software and simulation

Due date:	M24
Submission date:	30/09/2013
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	ASL
Version:	1.0
Status	Final
Author(s):	David Lecomber (ASL), Tobias Hilbrich (TUD), Mark O'Connor (ASL), Joachim Protze (TUD)
Reviewer(s)	Michele Weiland (EPCC), Mats Aspnäs (ABO), Alistair Hart (Cray)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	28/06/2012	Copy from Google Docs and complete document pass	Tobias Hilbrich (TUD), David Lecomber (ASL)
0.2	11/07/2012	Incorporated review comments	Tobias Hilbrich (TUD)
0.3	29/07/2013	Update Allinea sections with more detail	Mark O'Connor (ASL)
0.5	16/08/2013	Update MUST sections to current state	Joachim Protze (TUD)
0.5.1	23/8/2013	Include design changes as a result of co-design feedback, included more detail of designs completed so far	Mark O'Connor (ASL)
0.6	23/8/2013	Merge of changes on MUST/DDT parts	Joachim Protze (TUD)
0.7	26/08/2013	Reading pass (primarily MUST and introductory parts)	Tobias Hilbrich (TUD)
0.8	30/08/2013	Pass over Allinea sections	David Lecomber (ASL)
0.9	12/09/2013	Incorporated internal review comments	Joachim Protze (TUD)
0.9.1	16/09/2013	Handled several internal review comments	Tobias Hilbrich (TUD)
0.9.2	17/09/2013	Allinea modifications after internal review.	David Lecomber (ASL)
1.0	17/09/2013	Final version for submission	Joachim Protze (TUD)

Table of Contents

Index of Figures

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 adapt towards Exascale needs. We use deliverable D3.1 “State of the art and gap analysis” as a roadmap for these extensions. We extend the first version of this document from project month 10 and refine our designs and plans where we gained additional knowledge or feedback.

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 the categories of scalability and support for programming paradigms. Current runtime error detection tools scale to 100-1,000 processes at most which, even with today’s scale, is unsatisfactory. The design that we propose (to distribute key correctness checks of MUST) targets 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 allow MUST to provide automatic correctness checks for other paradigms.

Both Allinea DDT and MUST identify parallel programming paradigms as an important goal for improvement. However, within the CRESTA consortium there is no single paradigm (non MPI/OpenMP) that all application developers want to explore. Given the number of available parallel programming paradigms this is no surprise.We will extend both tools towards one additional paradigm, where Allinea DDT already supports a wide range of paradigms. Thus, we focus on support for Xeon Phi as the primary architecture to support for Allinea DDT and use a PGAS library implementation as the target paradigm for MUST. We select the library implementation since our instrumentation approach is oriented to interfere library calls. Checks for the selected PGAS paradigm will provide functionality that we can reuse for different PGAS implementations in the future.

At the current state of development towards Exascale systems we still see no clear indication towards fault tolerance requirements and techniques. Thus, we closely watch suggestions from upcoming CRESTA deliverables such as D2.5.2, but do not plan to implement such techniques within the timeframe of CRESTA.

Integrations of DDT and MUST can provide users with a unified user interface that provides access to both tools. Besides the advantages in usability, integration can also combine the advantages of the tools to lead to a deeper understanding of a software bug or reduced time to solution. As a result, we describe first steps for tool integration and resulting extensions to advance this integration. In particular, we identified an integration scheme that allows Allinea DDT to highlight errors that MUST detects in an asynchronous manner, i.e. where the application has already progressed beyond the error invocation at the time MUST detects the error.

Finally, we investigated the testing processes of both MUST and DDT. We identified that both tools already use very elaborate and extensive functionality tests.

Change Notes

This second version of the document is an update of document D3.4.1.

Key modifications include:

- Based on co-design feedback we shift development priorities from DDT features such as automatic array checksums and OpenMP 3 Task Parallelism towards improved CUDA and Xeon Phi support (Section 2.2);
- Increased detail in plan for application logging and comparisons via Allinea DDT (Section 2.2);
- We present first promising measurements of intra-layer communication (Section 3.1);
- We describe studies regarding correctness checking for further new paradigms (PGAS paradigms in Section 3.2.1 and tasking in Section 3.2.3);
- We select a paradigm for MUST correctness check extensions (Section 3.2.4),
- We update our plans towards fault tolerance (Section 0);
- We present a refined design for an Allinea DDT and MUST integration (Section 5);
- We add a design for a simplified batch mode for Allinea DDT as a direct result of the co-design process (Section 2); and
- We add Section 5.3 on tool extensibility and co-design work done to ensure other tools can benefit from Allinea DDT's parallel framework and Allinea MAP's performance monitoring data.

1 Introduction

In this document ("Debugging Design Document, D3.4.2") 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 and 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 designs that extend Allinea DDT and Section 3 presents designs that extend 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 massive 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 leverage 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.

We address the state of fault tolerance in Section 0 and present our design of 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 5.3). In the following we first provide a short overview to the tools Allinea DDT and MUST, and then highlight the purpose of this document.

1.1 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. During 2012 Allinea DDT was used at over 700,000 cores.

The Allinea DDT architecture essentially consists of a Qt4-library based user interface, 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.

1.2 MUST

MUST (Marmot Umpire Scalable Tool) is an automatic runtime error detection tool for parallel software. It currently 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, but 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, a crucial notion is that for some systems or runs, errors may not manifest in an actual application crash or hang. 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. ORNL 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.

1.3 Purpose

This document presents specific ideas and designs to overcome the identified gaps in current debugging tools. 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 *programming 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 hard to predict. Allinea DDT already offers wide support for novel programming paradigms such as the CUDA API - and the Xeon Phi architecture support has been developed within CRESTA. MPI correctness checks remain the focus of MUST, but we will explore checks for the GASPI PGAS API to evaluate its potential for other paradigms. Finally, due to limited input for techniques to implement fault tolerance, we detail no designs towards this challenge.

1.4 Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
CQ	Completed Queue
CUDA	Compute Unified Device Architecture
DDT	Allinea DDT – the parallel debugger
DWS	Distributed Wait State
GASPI	Global Address Space Programming Interface
GDB	GNU Project Debugger
GPGPU	General Purpose Graphics Processing Unit
GPI	Global address space Programming Interface
GPU	Graphics Processing Unit
GTI	Generic Tools Infrastructure
GUI	Graphical User Interface
ISP	In-situ Partial Order
MPI	Message Passing Interface

MUST	Marmot Umpire Scalable Tool
ORNL	Oak Ridge National Laboratory
P2P	Point-to-point
PGAS	Partitioned Global Address Space
PRACE	Partnership for Advanced Computing in Europe
PSTI	Workshop on Parallel Software Tools and Tool Infrastructures
SPEC	Standard Performance Evaluation Corporation
TBON	Tree Based Overlay Network
WFG	Waiting-for graph
XML	Extensible Markup Language

2 Debugger Design Proposals

These proposed features and enhancements to Allinea DDT are designed to address the core problems identified in attaining rapid correctness at Exascale.

2.1 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 APU – which combine the CPU and GPU on one chip.

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 GPU hybrid code until recently was typically OpenMP within a node, MPI between nodes - this has changed with CUDA, OpenACC or OpenCL now as an alternative to the OpenMP - potentially leaving multiple cores on a node idle whilst a GPU executes a CUDA kernel.

Additionally, the Intel Xeon Phi architecture is seeing strong interest amongst co-design partners and support for an OpenMP + MPI combination on Intel Xeon Phi will be developed for Allinea DDT. A prototype implementation has been made available for feedback and on-going application development.

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 when surveyed during the first feedback exercise with partners (D3.1).

2.2 Automated Anomaly Detection

Automatic identification of anomalous values is becoming important. Firstly, the volume of trace and debug data is increasing as applications and systems grow. 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 were discussed with co-design partners, but demand for improved heterogeneous platform support (NVIDIA CUDA + Intel Xeon Phi) was significantly higher. Enhancing and deepening support on these platforms will be prioritized over automated data integrity work.

Our research suggests that addressing Exascale debugging effectively hinges on well-executed implementations of:

- Additional automated correctness checking, – specifically integration with the MPI correctness checking proposed and implemented in MUST, see Sections 3 and 5.
- Enhancing and automating the ability to log and compare different runs, described by call paths, variable and invariant values and comparisons or statistical analyses over many processes in a scalable and minimally performance-impacting manner.
- Simpler, more flexible ways to launch and use debugging tools in complex batch script-driven environments. This was a direct result of co-design feedback during the CRESTA project.
- The continuing blurring of lines between debugging and profiling, as at increasingly large core counts the inability to scale efficiently is both frequently caused by bugs and is by definition a software defect at Exascale. See Section 5 for more details.

Building on extensible data collection to deliver easy access to low-overhead performance reporting, giving application developers and system owners a quick check on whether an application is performing well on the current system and scale - see Section 5 for more details on this.

Logging and comparing different runs is a desirable and efficient way to discover why two codes or runs have given different results - but a naive implementation logging all calls and variables is too slow and data-intensive for Exascale-class codes. Therefore a combination of dynamic instrumentation and job-wide debugging-level interruption with statistical aggregation is proposed. This approach will minimise the amount of redundant and unvisualisable data generated during program execution.

An initial logging implementation has been made available to CRESTA partners, allowing debugging runs to be automatically logged, saved, reviewed and shared. Implementation of comparison between logging runs across different scales, architectures or code revisions is also planned.

These changes are also enhancing the existing offline debugging reports (a form of non- interactive debugging) present in Allinea DDT.

2.3 Enabling Integration with Tools in Exascale Simulations

In moving towards automated debugging, one of the feedback items from the D3.1 deliverable and co-design with CRESTA partners was the requirement to make it easier to take a tool and run it inside existing workflows.

Existing applications tend to have complex dependencies and configurations. They can have workflows and frameworks that do not fit with existing shrink-wrapped “click and run” types of development tool. One example is the IFS code from ECMWF which uses its own workflow management tool to create packaged runs of the IFS application. These are responsible for obtaining initial data, configuring initial parameters, setting up the runtime environment – and executing a sequence of linked steps through the batch scheduling systems.

Exascale simulations are likely to bring more coupling of codes – and workflows such as the IFS scenario. This led to a requirement to make launching and configuration in these scenarios more seamless. A change to reduce the integration necessary for Allinea DDT to run to being a single "prefix" command inside existing jobs will be prototyped.

2.4 Application or Model Aware Debugging

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 is MPI handles. Presently there is no standard for MPI handle debugging - this means that the request objects or communicators are typically "opaque" to the debugger and provide no useful information for the user.

- TU-Dresden [1] 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 approach of printing 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. Designs for this have reached a limited prototype implementation form between Allinea and the Open MPI team. It is not clear whether this extension will appear in the MPI 3.1 standard.

Alternative models beyond MPI, such as OpenMP 3 Task Parallelism, have been investigated, but – based on D3.1 feedback - improving support for CUDA – particularly more general-purpose features such as dynamic parallelism – and Intel Xeon Phi were preferred and have been implemented in prototypes.

3 MUST

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

3.1 Scalability

MUST analyses MPI events in order to detect usage errors of the MPI interface. 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 [5], Umpire [6], and ISP [2] 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 on-going collaboration with the Los Alamos National Laboratories and the Lawrence Livermore National Laboratories. While these collaborations 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. From early prototypes we identified extension directions that we summarize in this document.

3.1.1 Component Overview

MUST uses a modular 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.

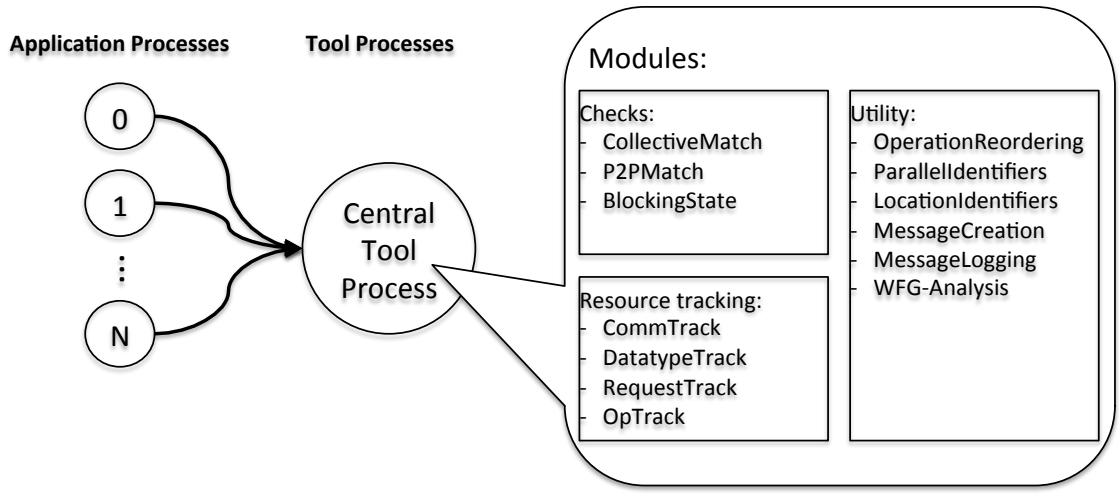


Figure 1 Centralised correctness checking with MUST

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.

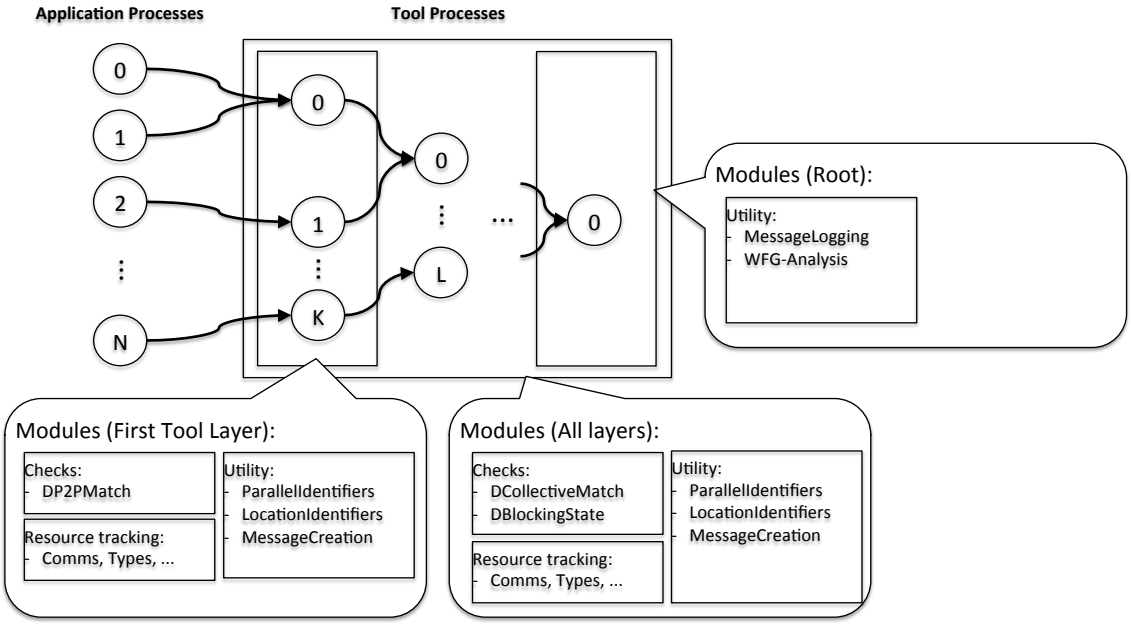


Figure 2 Distributed correctness checking with MUST

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 analyse point-to-point matching on the first non-application layer of the TBON. Collective MPI operations and deadlocks will be analysed 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.

3.1.2 Distributed P2P Analysis

We propose to match and analyse 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 has 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.

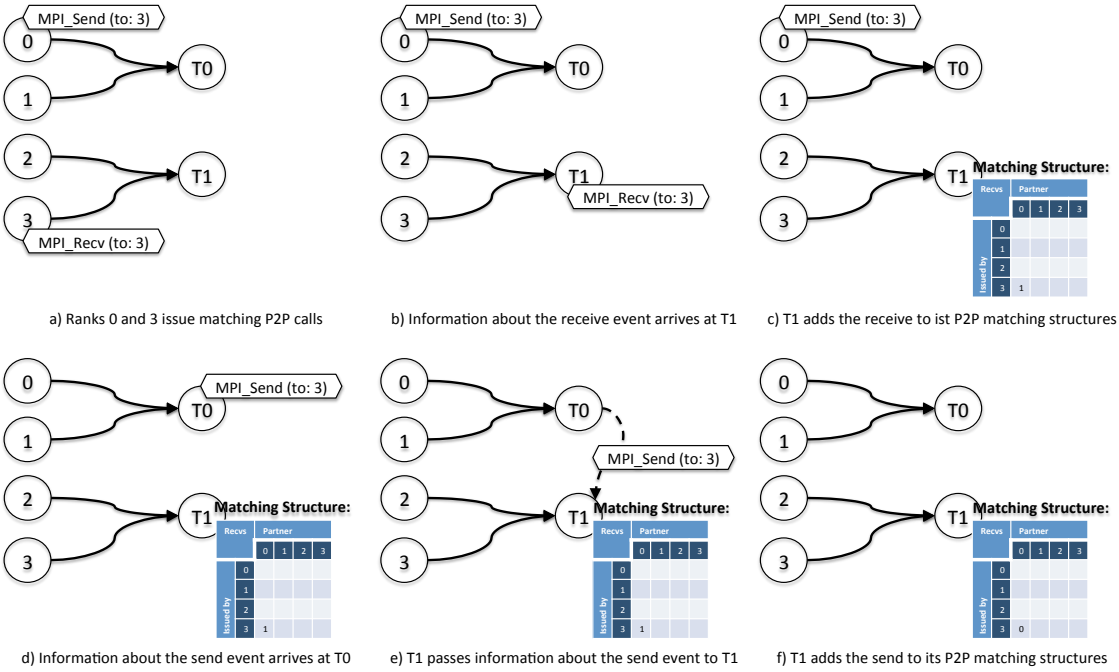


Figure 3 Illustration of intra layer communication based P2P matching

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 match receive events on the tool processes that process them directly, i.e. T1 in our example. T1 will analyse the receive event and determine that it is not aware of a matching send call for this event. As a result, T1 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 on 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.

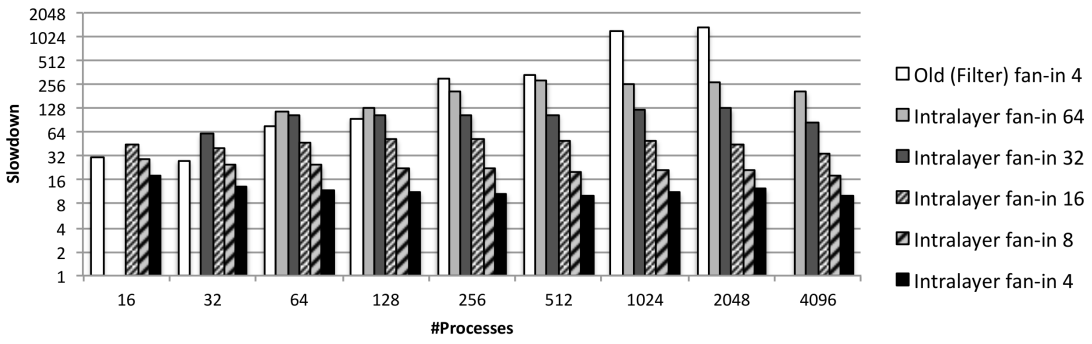
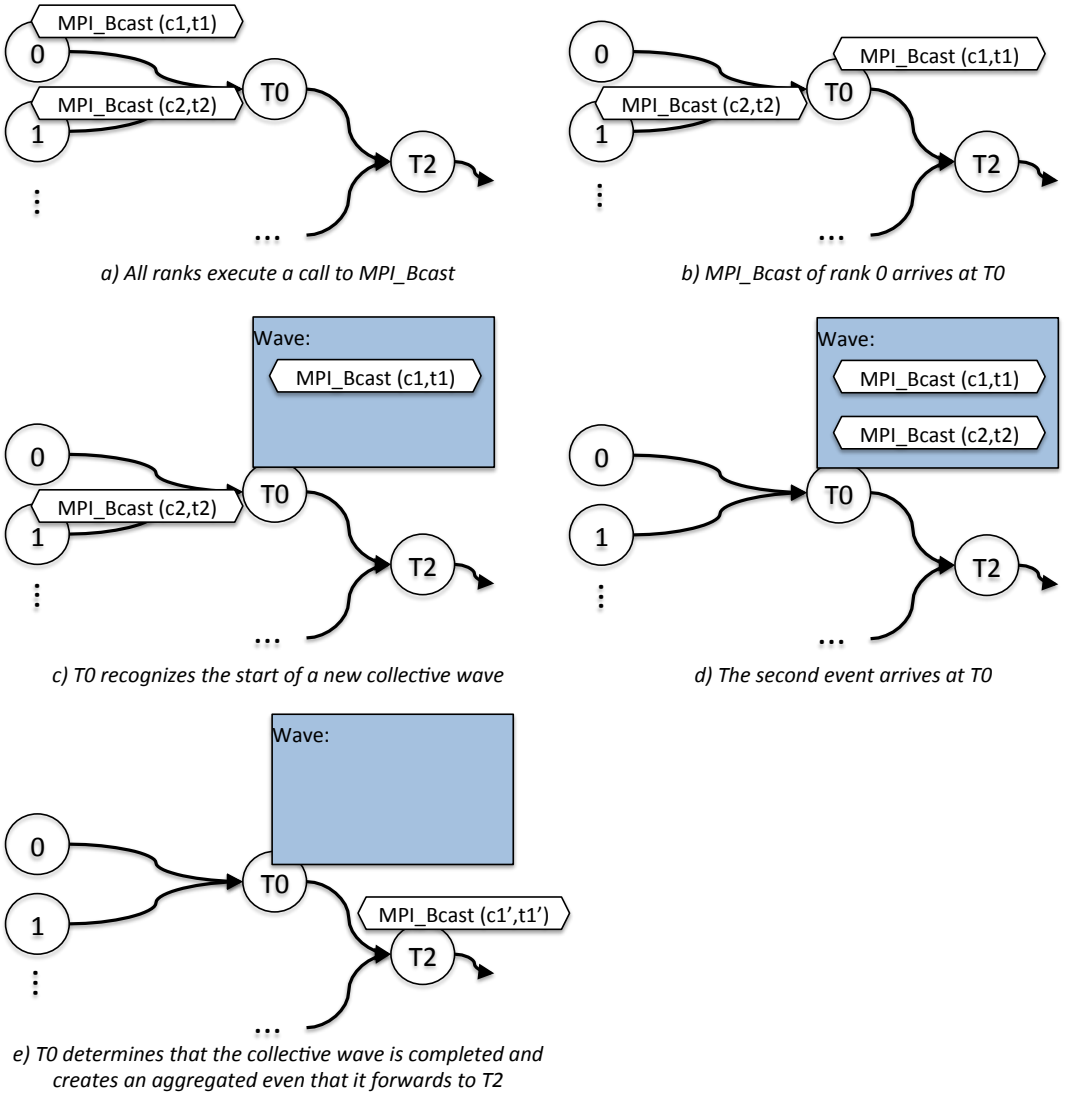


Figure 4: Overhead measurement of a prototype for point-to-point matching

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 utilises an MPI_COMM_WORLD virtualization. Using an intra-layer communication in this setting is straight-forward. The actual intra layer communication is implemented by regular communication protocols of GTI. In a paper [3] we published at PSTI'13 we showcase the impact of the intra layer communication on the overhead of point-to-point matching. For a synthetic benchmark example and an early prototype of point-to-point matching we see a constant slowdown while scaling from 16 to 4096 application processes in Figure 4. The figure presents a slowdown that we calculate as the ratio of application runtime with MUST to the runtime of a reference run.

3.1.3 Distributed Collective Analysis



a) All ranks execute a call to MPI_Bcast

b) MPI_Bcast of rank 0 arrives at T0

c) T0 recognizes the start of a new collective wave

d) The second event arrives at T0

e) T0 determines that the collective wave is completed and creates an aggregated even that it forwards to T2

Figure 5: Illustration of event aggregation for MPI collectives

We propose to analyse the correctness of MPI collective events within GTI's TBON. Most correctness checks for MPI collective operations can use aggregations for their implementation. Figure 5 illustrates this concept, where each application process issues an MPI_Bcast operation (Figure 5a). When the first MPI_Bcast event arrives at T0, it recognises this as a new wave of events and creates a respective data structure. Note that the event is not forwarded towards T2 (Figure 5c). When the second event arrives at T0 (Figure 5d), the TBON node determines that it received all events that belong to this wave and runs all correctness checks (Figure 5d). 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 5e).

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.

3.1.4 Distributed Wait-State Analysis

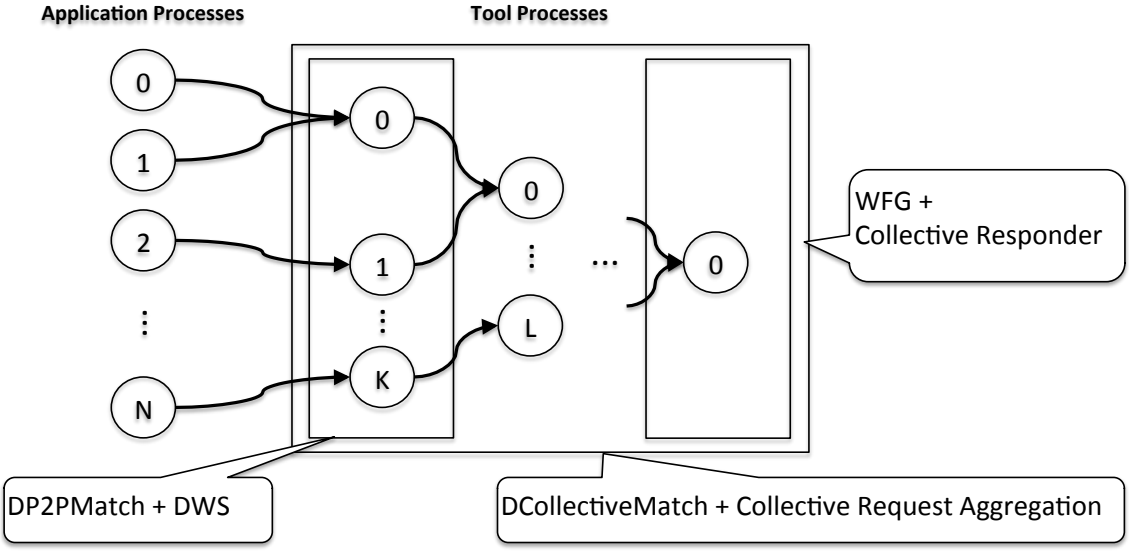


Figure 6: Illustration of distributed wait-state modules

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 centralised 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 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 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 deadlock detection on this graph. Note that this graph analysis is currently still centralised, but it should scale to 10,000 cores at least.

3.1.5 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 "panic" 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.

The use of an alternative communication medium (shared-memory) along with the use of immediate communication ensures that we can transfer all event information for error situations where some processes hang in blocking communication calls. However, communicating towards application processes that hang is challenging. Thus, the shutdown of such processes will be controlled by their responsible tool process. 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.

3.1.6 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 centralised 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 support 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 analysing 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.

MUST modules must receive events from application processes and then repeat the communication pattern of the application (coarsened). This highlights that rank-to-core placement impacts MUST's performance. If an application uses a tuned rank-to-core binding, then MUST may cause perturbation that degrades this optimisation; and MUST itself should use a similar tuned binding that reduces the overhead of its communication pattern replay. Thus, at higher scales rank-to-core binding can heavily influence the overhead of MUST. This notion remains an important direction for future work that should integrate with placement optimisation frameworks.

Besides the 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. With our current prototypes of MUST, we experience no degradation due to heavy MPI resource usage even with our intra-layer communication system that forwards tracked resources between nodes of a single tool layer. For different target applications or at increased scale, this may change and require future extensions.

The graph-based deadlock detection runs on the root of the TBON. For N processes it needs to analyse a graph with up to N nodes and N^2 arcs. 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, but do not consider extensions to this implementation within CRESTA

3.2 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 requires an instrumentation mechanism to both 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 generalised within GTI.

We sketch steps towards support for additional paradigms in this design document to evaluate development costs for their support and benefits from checks for these paradigms. This includes PGAS languages such as Coarray Fortran and accelerator-based paradigms such as CUDA. Based on this evaluation, we decided to focus on PGAS paradigms within CRESTA. Since the development of an instrumentation interface would consume considerable development resources we select a target paradigm that already provides such an interface. Co-design activities explore Coarray

Fortran, but since this paradigm provides no instrumentation interface, we cannot provide support for this paradigm within CRESTA. Therefore we decide to use a library implementation of the PGAS paradigm to prototype correctness checks for a PGAS language. These checks will provide functionality that we can reuse for different PGAS implementations in the future.

3.2.1 PGAS Language Correctness Checking

PGAS languages are available as library-based paradigms, e.g. GASPI / GPI [7], OpenSHMEM and as language extensions or individual languages, e.g. UPC, Coarray 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. Detecting all types of PGAS usage errors will therefore require tools to trace each single memory access, which limits the applicability of a scalable runtime tool such as MUST, since it induces high overheads. 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 synchronisation errors like concurrent write operations to equal memory regions or a lack of synchronisation primitives between DMA operations. Finally, due to locking and barrier synchronisation, deadlocks may also manifest in PGAS languages. The latter two can also be detected without tracing each single memory access. First correctness checks within MUST fall into the category of detection of coarse-grained synchronisation errors and deadlocks.

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

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

Within the CRESTA co-design activities the application IFS explores Coarray Fortran as a PGAS paradigm. Experiments with the paradigm show performance benefits that motivate the use of such paradigms. However, since Coarray Fortran provides no instrumentation interface, we cannot directly apply correctness checks to this PGAS implementation. Thus, we will prototype MUST correctness checks for a PGAS library implementation instead, since this offers a readily available instrumentation interface.

Since the first version of this document the GASPI consortium released Version 1.0 of the GASPI standard. This standard includes a profiling interface that offers a name shifting mechanism like MPI.

OpenSHMEM doesn't define a profiling interface like GASPI. This makes the interception of library calls a bit less portable.

The constraints given by these interfaces are very similar, given by the nature of the PGAS approach. The decision for one of this libraries, or both will be done while implementing the library wrapper. Based on one of this interfaces we will implement prototype checks for MUST. This will include checks for:

- Integrity checks for argument values (within the defined boundaries); (application local check)
- Mutual excluded calls according to the standard (e.g. collectives, MPI/GASPI); (aggregated, centralised checks)
- Checks for: "A valid one-sided communication request requires that the local and the remote segment are allocated, that there is a connection between the

local and the remote GASPI process and that the remote segment has been registered on the local GASPI process." (check with intra-layer communication)
- Coarse-grained checks for races based on library calls. (check with intra-layer communication)

The MPI interoperability described in the GASPI standard is minimal. It states that there shall be no running MPI operation when GASPI is active and vice versa. For this reason we cannot use MPI as communication means between application and tool processes. This restriction can impact the productivity and availability of our prototype checks. GTI provides a posix shared memory communication means that is restricted to a shared memory node. This would suffice for a small prototype, but no scalable tool.

3.2.2 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. whether the device or the kernel is in the correct state for a certain API call. Also, this includes a few potential non-local checks that can also document sources for performance degradation, 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.

3.2.3 Tasking Paradigm Correctness Checking

Tasking paradigms such as OmpSs, Charm++ and DAGuE could potentially increase the programmability of Exascale systems. These paradigms offer promising properties towards fault tolerance and load balancing. Since a runtime system manages the actual communication calls for such paradigms runtime correctness checks can only check for a limited number of correctness error classes. First, deadlocks are likely rare for such paradigms, since a runtime system automatically issues communication and synchronisation primitives. Second, a key correctness issue for tasking are data races on shared memory. As for PGAS languages, this requires instrumentation for each memory access, which may drastically increase runtime overhead for an approach such as MUST. Correctness checks for a tasking paradigm would be most powerful if they could check task-specific input/output properties to reveal programming errors. However, such checks require a specification language that provides the correctness tool input on input/output requirements of each task.

3.2.4 Extensions to Provide GASPI/OpenSHMEM Checks

GTI as a base infrastructure of MUST is designed to be programing paradigm agnostic, i.e. is not limited to MPI tools. However, GTI provides communication and tool node start-up services that differ between target paradigms. As an example, for MPI we use a communicator virtualisation that allows us to use MPI for tool node start-up and for communication. Thus, to support GASPI checks we must adapt our communication modules and our tool node start-up mechanisms for this paradigm. Since both consume large amounts of development time, we will first explore execution modes that use MPI and GASPI simultaneously to reuse existing mechanisms. If this is not feasible we will explore more manual mechanisms to avoid costly development of additional tool components, such that we can focus on the development of the actual checks.

For OpenSHMEM we can use the MPI communication means of GTI.

Once we can execute GTI tools with GASPI/OpenSHMEM applications (automatically, semi-automatically, or manually) we must provide information on the GASPI/OpenSHMEM API to tool modules. This requires extensions to PnMPI (a basis

infrastructure used by GTI) to support the new instrumentation interface. Afterwards, the tool developer can directly specify analysis modules, which can implement correctness checks.

4 Fault Tolerance

Expectations [8] for Exascale systems indicate that mean-time-to-failure may be lower than a day. If the mean-time-to-failure becomes too low, check pointing approaches may become unfeasible. Such a compute system requires that applications, system ware, operating system, and/or any type of runtime tool needs to be aware of failures and handle them such that an application can continue its execution. 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. At the current project state we have no results about an expected mean-time-of-failure or indications about fault tolerance mechanisms. Particularly, the latest US-project on Exascale operating systems (ARGO [9]) appears to address fault tolerance primarily within the operating system level. Thus, it remains unclear whether (and what) techniques for fault tolerance need to be applied to which software layers. The MPI forum also did not provide fault tolerance support for its recent MPI-3 standard. Possibly check pointing remains a viable option. Thus, we will not address any modifications for fault tolerance in this design document.

5 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 tool users. An early integration between DDT and Marmot (a predecessor of MUST) already allowed Marmot to stop the execution when it detected an error. Afterwards, the user could investigate error details with the debugger. Both MUST and DDT rely on the use of a Tree Based Overlay Network (TBON), and an interesting question is whether an integration could allow the tools to share this infrastructure for easier deployment and lower resource consumption in the future. We include experience with Allinea MAP that reuses components from Allinea DDT in that respect. In theory, graphical user interfaces within DDT could control the behaviour 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.

5.1 Goals

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 detects errors,
- Displaying MUST's output within DDT, and
- 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 step before it runs with an application. This code generation also needs to be handled when integrated with DDT. A straightforward solution is to use 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 the integrated tool.

Within the scope of CRESTA we will implement the following integration features:

- 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, implemented by passing arguments to the mustrun command.
- MUST can provide DDT with information about MPI state, e.g. for MPI handles (See Section 0)

5.2 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 set 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 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.

Allinea DDT will be modified to allow users to add MUST checking to a normal debugging session at start-up time. The tools will integrate as follows:

1. Allinea DDT launches MUST’s “mustrun” with a special identifying argument or environment variable to inform MUST that this session is being debugged.
2. MUST performs its usual start-up behaviour, but launches “ddt-client” instead of executing the underlying mpirun as it would do in a non-debugger run.
3. Allinea DDT detects the ddt-client launch, and starts mpirun and the rest of the processes under debugger control using the arguments passed by MUST.

As mentioned above, MUST allocates additional processes for the analysis TBON. For cases in which DDT needs to know in advance how many processes will be required in total (e.g. when requesting resources from a job scheduler) it will use the existing “mustrun –must:info” functionality of MUST to get this information.

Using this start-up methodology, Allinea DDT and MUST do not need to share the same scalable tree, vastly simplifying implementation and maintenance. MUST will continue to manage its processes and communication as usual. Allinea DDT will hide MUST-specific processes from the user using the same functionality as implemented previously when adding Marmot support. MUST decides in the MPI_Init function

whether a process becomes a tool or an application process, the tool processes will never leave the MPI_Init function, so DDT is able to hide the tool processes after the application processes left the MPI_Init function.

For scalability the analysis within the TBON runs asynchronously. When an error is identified, the application has typically made progress. Therefore DDT cannot break at the call where the error occurred. For applications with a reproducible communication scheme we will provide a simplistic replay mode where we:

- Restart the application run,
- Ensure that the communication scheme is retained,
- Break at the erroneous MPI call, and
- Display the recorded error message.

This mode is not applicable for applications with non-deterministic communication schemes, e.g. dynamically load balanced applications.

5.3 Tool Extensibility

Allinea MAP, a performance profiling tool, is now part of the Allinea tools platform. Co-design work with some CRESTA members was undertaken to ensure that this meets the identified needs to locate and understand all causes of performance bottlenecks, including algorithmic errors and other software defects. Important feedback from this process led Allinea MAP to produce XML output files suitable for integration with and analysis by other applications.

One design requirement is building a common extensible platform that other tools are able to benefit from, without having to re-implement high-scalability start-up and data merging.

A tool that takes advantage of this tool extensibility is being prototyped that will enable quick, low-overhead performance reports of unmodified application binaries, allowing application and system owners to characterize the performance of a code and quickly identify whether a code is currently in need of deeper analysis.

6 Testing Plan

We categorise 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 at 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 an HPC system where both tools are available. We will use regular tests on this system to test the tool integration.

6.1 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 – this is planned for M24-M36.

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.

6.2 MUST Functionality Tests

MUST uses CTest for automatic testing. We currently have a total of 697 test cases that represent various correct or incorrect synthetic 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. To test MUST with more complex applications, we use SPEC MPI2007 and the NAS Parallel Benchmarks (NPB) after each larger functionality extension.

7 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] Tobias Hilbrich, Joachim Protze, Bronis R. de Supinski, Martin Schulz, Matthias S. Mueller and Wolfgang E. Nagel, " Intralayer Communication for Tree-Based Overlay Networks", Parallel Software Tools and Tool Infrastructures Workshop (PSTI), (2013)

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

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

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

[7] 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)

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

[9] Argonne National Laboratory, Mathematics and Computer Science, Feature Stories, (08/07/13), http://www.mcs.anl.gov/articles/designing-new-operating-system-exascale-architectures, (accessed August 2013)