Chapter 5
MUST: A Scalable Approach to Runtime Error Detection in MPI Programs

Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller

Abstract The Message-Passing Interface (MPI) is large and complex. Therefore, programming MPI is error prone. Several MPI runtime correctness tools address classes of usage errors, such as deadlocks or non-portable constructs. To our knowledge none of these tools scales to more than about 100 processes. However, some of the current HPC systems use more than 100,000 cores and future systems are expected to use far more. Since errors often depend on the task count used, we need correctness tools that scale to the full system size. We present a novel framework for scalable MPI correctness tools to address this need. Our fine-grained, module-based approach supports rapid prototyping and allows correctness tools built upon it to adapt to different architectures and use cases. The design uses P^nMPI to instantiate a tool from a set of individual modules. We present an overview of our design, along with first performance results for a proof of concept implementation.

5.1 Introduction

The Message Passing Interface (MPI) [1, 2] is the de-facto standard for programming HPC (High Performance Computing) applications. Even the first version of this interface offers more than 100 different functions to provide various types of data transfers. Thus MPI usage is error prone and debugging tools can greatly in-

Tobias Hilbrich
GWT-TUD GmbH, Chemnitzer Str. 48b, 01187 Dresden, Germany
e-mail: tobias.hilbrich@zih.tu-dresden.de

Martin Schulz, Bronis R. de Supinski
Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
e-mail: {schulzm, bronis}@llnl.gov

Matthias S. Müller
Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, D-01062 Dresden, Germany
e-mail: matthias.mueller@tu-dresden.de

crease MPI programmers' productivity. Many types of errors can occur with MPI usage including invalid arguments, errors in type matching, race conditions, deadlocks and portability errors. Existing tools that detect some of these errors use one the following three approaches: static source code analysis, model checking or runtime error detection.

Runtime error detection is usually the most practical of these approaches for tool users, since it can be deployed transparently and avoids the potentially exponential analysis time of static analysis or model checking. However, these tools are generally limited to the detection of errors that occur in the executed control flow of the application and, thus, may not identify all potential errors. Several runtime error detection tools for MPI exist; however, our experience is that none of these tools covers all types of MPI errors. Further, none is known to scale to more than about 100 processes. With current systems that utilize more than 100,000 cores it is becoming increasingly difficult to apply these tools, even to small test cases.

This paper presents MUST, a new approach to runtime error detection in MPI applications. It draws upon our previous experience with the existing tools Marmot [3] and Umpire [4] and is specifically designed to overcome the scalability limitations of current runtime detection tools while facilitating the implementation of additonal detection routines. MUST relies on a fine grain design in the form of modules that are loaded into P^nMPI [5]. The next section will present the experiences and issues that we discovered during our development of Marmot and Umpire. Section 5.3 presents the goals and general design ideas of MUST, while Section 5.4 covers several key design details of MUST. In Section 5.5 we present initial experimental results with a proof of concept implementation of the MUST design. Finally, Sections 5.6 and 5.7 present related work and our conclusions.

5.2 Experiences from Marmot and Umpire

This section presents insights into our two predecessor MPI correctness checking tools: Marmot [3] and Umpire [4]. Marmot provides a wide range of local and global checks and offers good usability and integration into several other tools. Umpire's strength is a runtime deadlock detection algorithm that detects all actual deadlocks in MPI-1.2 as well as some potential deadlocks on alternate execution paths. While both tools have been very successful and have helped users debug their codes, they both are first generation MPI checker tools and have inherent limitations, upon which we focus in the following.

In particular, our analysis focuses on two things: first, the communication system for MPI trace records; second, the separation of tool internal infrastructure and the actual correctness checks. The communication system is necessary for checks (e.g., deadlock detection or type matching) that require global knowledge of MPI calls, i.e., data from more than one process. Thus, such checks require a system to communicate records for MPI calls. The separation of tool internal infrastructure and the actual correctness checks is important in order to enhance existing checks and to add further correctness checks that are used for new features or new versions of the MPI standard. We first analyze these aspects for Marmot and then cover Umpire.

5.2.1 Marmot

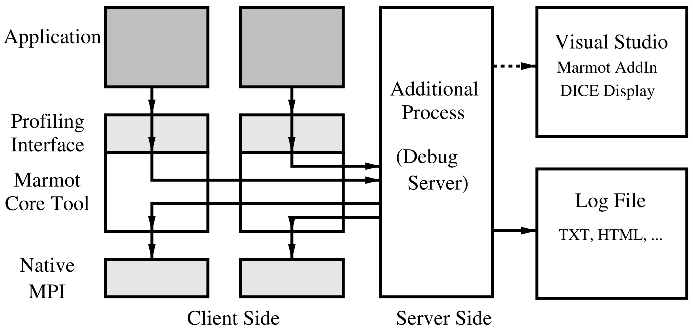


Fig. 5.1: Marmot trace communication design.

Marmot is an MPI runtime checker written in C++ that covers MPI-1.2 and parts of MPI-2. Its communication system is sketched in Figure 5.1. Marmot's MPI wrappers intercept any MPI call issued by the application. Marmot then performs two steps before executing the actual MPI call: first, it checks for correctness of the MPI call locally; second, it sends a trace record for this MPI call to the "DebugServer", a global manager process. The application process continues its execution only after it receives a ready-message from the *DebugServer*. As a result, it is guaranteed that all non-local checks executed at the *DebugServer*, as well as all local, are finished before the actual MPI call is issued. This synchronous checking ensures that all errors are reported before they can actually occur, which removes the need to handle potential application crashes. The *DebugServer* also executes a timeout based deadlock detection. While this approach can detect many deadlocks, it can lead to false positives. Also, it is not possible to highlight the MPI calls that lead to a deadlock with this strategy. Additionally, the *DebugServer* performs error logging in various output formats and can send error reports via TCP socket communication to arbitrary receivers. The main disadvantage of this synchronous or blocking communication system is its high impact on application performance. In particular, the runtime overhead increases significantly as the number of MPI processes increases since the *DebugServer* is a centralized bottleneck. Also, the blocking communication with the *DebugServer* can lead to high latency even at small scales, which – especially for latency bound applications – is a disadvantage.

The separation of tool internal infrastructure and the actual MPI correctness checks is not well solved for Marmot. It uses one C++ class for each MPI call and uses multiple abstract classes to build a hierarchy for all MPI calls. Checks are implemented as methods of these classes and are called before the PMPI call is issued. This has two disadvantages: First, checks for one MPI call are often distributed to multiple objects making it hard to determine which checks are used for a certain MPI call. Second, our experience with Marmot shows that there is no reasonable hierarchy for MPI calls that also builds a good hierarchy for all the different types of checks. Thus, many checks in Marmot are either implemented in very abstract classes or are implemented in multiple branches of the object hierarchy, which leads to code redundancy. The implementation of the checks uses a multitude of static variables that are stored in the more abstract classes of the hierarchy. These variables represent state information for the MPI system leading to checks being very tightly coupled with Marmot's class hierarchy.

The development of Marmot occurred concurrently with multiple workshops on parallel programming tools that included hands-on sessions. The experiences from these workshops guided the development of Marmot. One of the commonly asked-for features are integrations into widely accepted tools like debuggers, IDEs, or performance tools. As a result, Marmot provides multiple usability enhancing tools and integrations that help users in applying the tool. These efforts help new users to apply the tool easily, which is an important factor for the success of Marmot.

5.2.2 Umpire

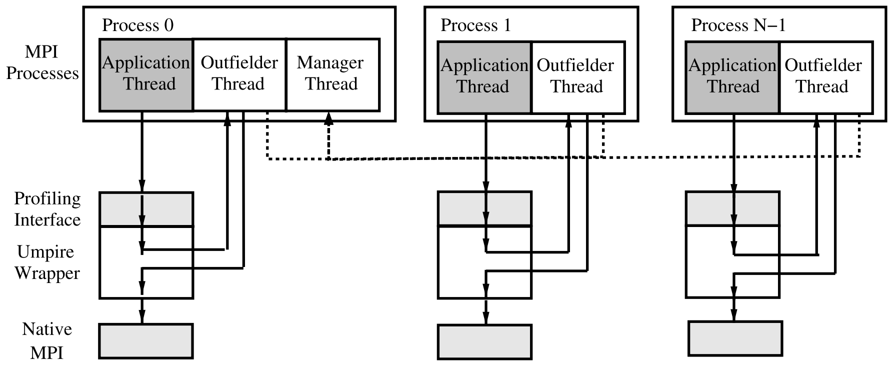


Fig. 5.2: Umpire trace communication design.

The MPI correctness checker Umpire is written in C and focuses on non-local MPI checks. It executes both a centralized deadlock detection and type matching at a central manager. Figure 5.2 sketches the trace transfer that is implemented in

Umpire. The first difference to Marmot is that Umpire spawns extra threads for each MPI process. It spawns an “outfielder” thread for all processes. In addition, it spawns a “manager” thread on one process (usually process 0). The *outfielder* thread asynchronously transfers trace records to the centralized manager, which is executed on the *manager* thread.

Similarly to Marmot, Umpire’s wrappers intercept any MPI call issued by the application. However, Umpire minimizes immediate application perturbation. The application thrad only builds a trace record for the MPI call, which it transfers to the *outfielder* thread of that process through shared memory. Each *outfielder* thread aggregates the trace records that it receives and sends them to the *manger* thread. This send happens if the buffer limit is exceeded or when a timeout occurs. This communication is implemented with either MPI or shared memory depending on the system architecture. Umpire’s communication system is designed to incur low runtime overhead, which is achieved with the asynchronous transfer of trace records to the central manager. Due to the asynchronous design, the central manager is no longer a bottleneck. However, it still limits performance since it must analyze trace records of all processes. Further, performance tests with Umpire show that the efficiency of the asynchronous transfer depends highly on the interleaving of the communication of the application and the MPI communication of the *outfielder* threads [6].

As with Marmot, the separation of internal infrastructure and correctness checks is incomplete with Umpire. The checks that are executed at the centralized manager are tightly coupled to a large structure that represents internal state as well as MPI state. All checks are directly coupled to this structure. Also, some of the different checks of the central manager are dependent on each other and need to use internal data from each other. This applies to a smaller extent to local checks which tend to need less state information. Umpire currently only implements a small number of local checks. Additional local checks may be added by extending the wrapper generation of Umpire, since checks can be issued in the wrappers or other generated files.

5.3 Introduction to MUST

We present MUST (Marmot Umpire Support Tool), a new approach to runtime MPI correctness checking. We designed MUST to overcome the limitations to scalability and extensibility of Umpire and Marmot and their hard coded trace communication with a centralized manager. Its design focuses on the following goals:

1. Correctness
2. Scalability
3. Usability
4. Portability

The correctness goal is the most important one and comes with two sub-goals: first, the tool must not give false positives; second, the tool should detect all MPI related errors that manifest themselves in an execution with MUST. We restrict this second sub-goal to runs in which errors actually occur, as the detection of all potential errors would likely incur an intolerable runtime overhead, which would limit the applicability of the tool.

The second goal, scalability, is one of the main motivations for this new approach to MPI checking. The tool must scale at least to small or medium sized test cases on next generation HPC systems. With the current trend towards high numbers of computing cores, this means at least a range of 1,000 to 10,000 processes. Our goal is to offer a full set of correctness features for 1,000 processes at a runtime overhead of less than 10%, and a restricted set of correctness features for 10,000 processes at the same runtime overhead.

The further goals, usability and portability, are important to achieve a successful tool that will find acceptance with both application programmers and HPC support staff. A common problem with many HPC tools is that they require the application developer to recompile and relink the application, which can be very time consuming for larger applications. Therefore, we aim to avoid this requirement with MUST. Further, tools must be adaptable to special HPC systems that impose restrictions such as no support for threads.

We address both issues with P^nMPI [5], an infrastructure for MPI tools. P^nMPI simplifies MPI tool usage by allowing tools to be added dynamically, removing the need to recompile and offering flexibility in the choice and combination of PMPI-based tools. Only the P^nMPI core is linked to the application, instead of a certain MPI tool. If the MPI tools are available as shared libraries, P^nMPI supports the application of any number of MPI tools simultaneously. Thus, at execution time, the tool user can decide which tools he wants to apply to an application.

P^nMPI achieves this flexibility by virtualizing the MPI Profiling interface. It considers each MPI tool as a module and arranges these modules in stacks that specify the order in which MPI calls are passed to the modules. These modules may also cooperate with each other by offering services to or using services from other modules. Further, special P^nMPI modules allow more enhanced features like condition-based branching in stacks. This infrastructure provides flexibility combined with advantages to tool usability. As a result, we base the design of MUST on P^nMPI and use fine grained modules that can be composed to form an instance of MUST. With this basic infrastructure, we can easily adapt the MUST tool to specialized scenarios such as when only an individual correctness check is of interest.

A further important aspect of MUST is the notion that the overall tool will consist of three layers. The bottom layer is provided by P^nMPI and its modules that provide the basic infrastructure and composability of the tool. The actual correctness checks form the upper layer of MUST. The remaining middle layer has to provide service tasks like trace record generation and the communication of trace records to processes and threads that are exclusively allocated to the tool, which are used to offload correctness analyses. A further task is the management of these processes and threads for error cases, startup and shutdown. This task is tool agnostic and needed for many HPC tools. As a result, we want to provide this layer of functionality as a decoupled set of modules that is also available to other tool developers. Thus, we name this layer of functionality "Generic Tool Infrastructure" (GTI).

5.4 MUST Design

This section introduces some of the key design ideas of MUST. As discussed in the last section, our design uses P^nMPI for the underlying infrastructure along with a set of fine grain modules that implement the MPI checks. A first important aspect of the MUST design is the ability to execute correctness checks either in an application process itself (in the critical path) or in extra processes or threads that are used to offload these analyses (away from the critical path). This choice can provide a low runtime overhead while supporting portability. The first part of this section introduces the concepts that we use to achieve this goal. Afterwards we present an overview of the overall components of the MUST design, and highlight their tasks. A further aspect of the design is the communication of trace records. We present an overview of how different types of modules combine to implement this communication. These modules are part of the GTI layer and can be used by other tools.

5.4.1 Offloading of Checks

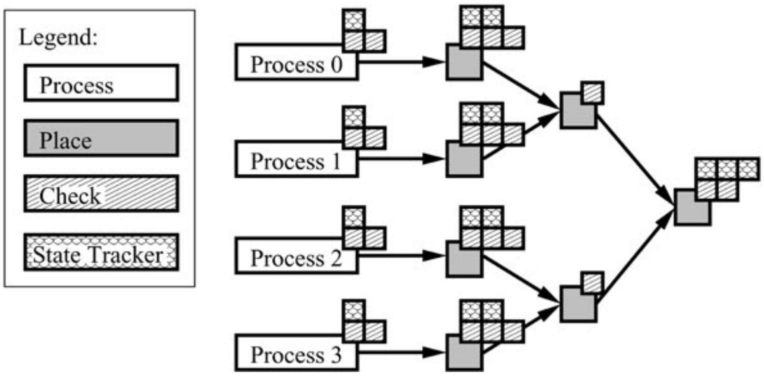


Fig. 5.3: Example instantiation of places, checks, state trackers and a communication network.

The option to execute correctness checks on additional processes or threads is one of MUST's most important aspects. We refer to such a process or thread by the term "place". Marmot and Umpire both execute some checks on an extra place (the *manager* thread for Umpire and the *DebugServer* process for Marmot). However, both tools do not support the selection of the place of execution freely, as these checks are explicitly aware of being executed at a certain place. Moving such a check into the critical path, or a check being executed in the critical path to another place is not easily possible in either Marmot or Umpire.

The main problem is that the execution of checks often requires background information on the state of MPI. It is possible within the application process for a synchronous tool to query such information with MPI calls, while it is not possible on additional processes that do not have access to the MPI library. Similarly, if the MPI process has proceeded beyond the MPI call, the relevant state may have changed. Also, the required information often must be gathered and updated during application execution. For example, determining which requests are currently active requires the sequence of request initiations and completions. While much of the work can be offloaded to MPI emulation, the gathering of the basic information must take place in the application processes themselves.

MUST uses the concept of "state trackers" to solve this problem. All information that a check requires but is not directly available from the arguments of the MPI call that triggered this check, must be provided by state trackers. These trackers are implemented as independent modules and may gather different types of data during the application's runtime and provide it to checks when needed. If multiple checks require the same state tracker, a single instance of the state tracker can provide this information. In order to support the placement of a check at any place, the MUST system has to determine which state trackers are required on each place. This strategy provides a transparent way to implement checks that can be offloaded to places.

Figure 5.3 shows an example distribution of places, checks, state trackers and a trace communication network. It uses four application processes and seven extra places to offload checks. The checks are highlighted as little boxes in the top right corner of the places or application processes. Each place or application process may also need state trackers that are indicated by little boxes above the checks.

5.4.2 Major Components

Figure 5.4 shows the main components of MUST and parts of their overall interaction. The correctness checks and the tool infrastructure are provided as modules from MUST, P^nMPI, and the GTI (top row). We summarize a further set of components in the top right of the figure as "descriptions", which describe properties of some of the modules and formalize what checks apply to the arguments of specific MPI calls. They also characterize the dependencies of checks and state trackers. A GUI (middle left) provides users with options to individualize MUST for their needs, e.g., to specify the checks being used, to add extra processes/threads that offload checks, or to define the layout of the trace communication network. A default configuration will usually by sufficient for smaller test cases, while large scale

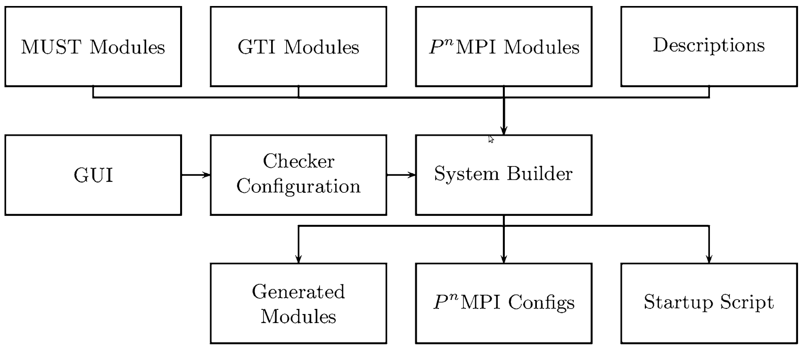


Fig. 5.4: Major components in MUST; arcs denote input/output dependencies.

tests will need a specifically tailored configuration. The system builder component uses the selected tool configuration, the list of available modules and the various descriptions to create the configuration files for P^nMPI and additional intermediate modules, including specialized MPI wrappers to create and forward the necessary trace records. An additional startup script may be provided to simplify the startup of the application with MUST.

5.4.3 Trace Communication System

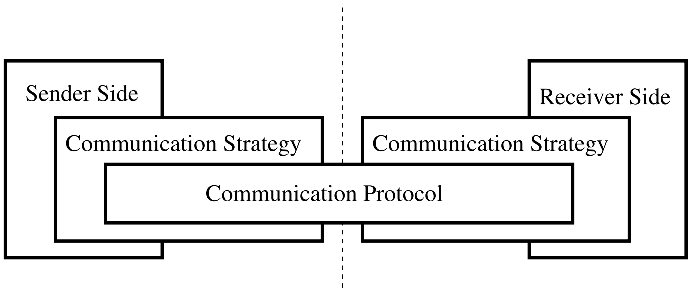


Fig. 5.5: Composition of places with communication strategies and communication protocols.

An important aspect of MUST's design is its encapsulation of how to transport trace records from one process or thread to another. An efficient communication of trace records primarily depends on two things: first, an efficient communication medium that optimizes the use of the underlying system where possible; second, an efficient strategy to use these communication media. Thus, we must use shared memory when communicating on node or rely on InfiniBand instead of Ethernet if both networks are available. It will usually be very inefficient to transfer tiny trace records with single messages with a TCP/InfiniBand/MPI based communication. Also, it will be more effective not to wait until the message has been received for most media.

The GTI component of MUST solves this problem by combining two different types of modules to implement a communication. The first type of module, a "communication strategy", decides when to send what information: it may send trace records immediately or it may delay the transfer of trace records and aggregate them into larger messages. The second type of module, a "communication protocol", implements the communication for a particular communication medium, e.g., TCP/IP, InfiniBand, SHMEM, or MPI.

Figure 5.5 shows how we compose these modules on the sender and receiver side. By selecting appropriate combinations of these two module types, we can provide a flexible, adaptable and high performance communication of MPI communication traces. One instantiation of MUST may use multiple combinations, e.g., a shared memory communication protocol to transfer MPI trace records to an extra thread and a TCP/IP communication protocol to transfer trace records from this thread to a further place used to offload checks.

5.5 Initial Experiments

We developed a proof of concept implementation of a subset of the MUST design in order to verify our ideas, as well as to perform first performance studies. The implementation provides the features necessary to use extra places and transfer trace records to them. One of our early questions is the feasibility of our runtime overhead goals. The question at hand is, whether we can transfer the trace records from the application processes to extra places without perturbing the application. We use initial experiments with our proof of concept implementation to study this problem. We use two different communication layouts and three different communication strategies to study different communication approaches. Our tests intercept all MPI calls and create a trace record for each. We send these trace records from the application processes to extra places and measure the runtime overhead that results from this extra communication. However, the receiver side only receives and unpacks these trace records; no checks are executed. We use NPB3.3-MPI as target applications and run our tests on a 1152 node AMD Opteron Linux cluster with a DDR InfiniBand network. Each node has 8 cores on four sockets and 16 GB of main memory that is shared between all cores.

As the communication protocol we use MPI itself, as it provides an easily available and highly optimized communication medium. It also offers a simple way to allocate extra processes for MUST. We use P^nMPI based virtualization to split an allocated `MPI_COMM_WORLD` into multiple disjoint sets. The application uses one of these sets as its `MPI_COMM_WORLD`, which is transparent to the application itself. The remaining sets can be used for MUST. MPI based communication between all of the sets is possible. We use two different communication layouts, which are “1-to-1”, a best case layout where each application process has one extra process that receives its trace records, and “all-to-1”, a centralized manager case where all application processes send their trace records to one extra process. The first layout helps to determine what runtime overhead to expect for a case where checks can be well distributed and no centralized manager needs to receive records from all processes. The second case captures the limits of a communication with a centralized manager, as in Umpire and Marmot.

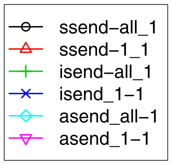
We use three different communication strategies to implement different communication schemes. These are:

Ssend: Sends one message for each trace record, waits for the completion of the receive of each message before it continues execution.

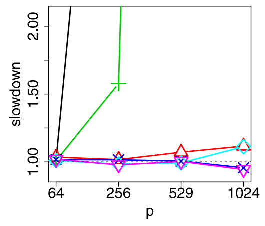
Isend: Sends one message for each trace record, does not wait for the completion of the receive of the message. With the MPI based communication protocol this is implemented with an `MPI_Isend` call.

Asend: Aggregates multiple trace records into one message, sends the message when either the aggregate buffer is full (100KB) or a flush is issued. As with *Isend*, the sender does not wait for the completion of the receive of the message.

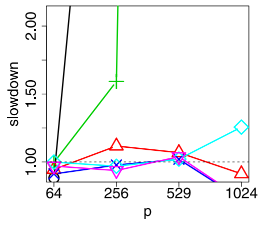
The *Ssend* strategy is very similar to the communication currently used in Marmot. Besides its obvious performance disadvantage, it simplifies handling of application crashes as it guarantees that trace records were sent out from the application before a crash can occur. The *Isend* strategy is still simple to implement and should overcome the performance problems of the *Ssend* strategy. The *Asend* strategy, which is similar to Umpire’s communication strategy, is our most complex strategy, but offers multiple optimizations that may provide a low runtime overhead. In particular, we expect that this method will achieve higher bandwidth, due to the aggregation of the trace records. However, its performance benefit will depend on a good interleaving of the communication: we expect a high runtime penalty if the aggregated messages are sent while the application is in a communication phase. On the other hand, sending the aggreagated messages while the application is in a computation phase will incur close to no runtime overhead, particularly on systems with communication co-processors. Experiments with Umpire already highlighted the significance of this timing behavior [6]. As a result, we instrument the NPB kernels to issue a flush of the aggregated buffer when the application enters a computation phase. This removes the need for an automatic detection of computation phases and represents a close to best case scenario. For a final system, we will have to apply a heuristic to guess when the application is entering a computation phase.



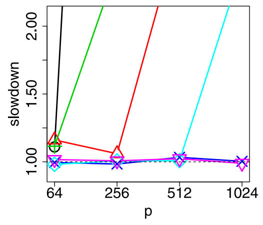
(a) Legend



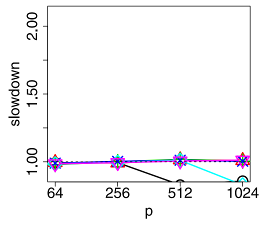
(b) bt



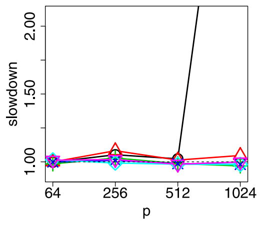
(c) sp



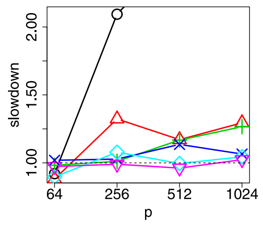
(d) cg



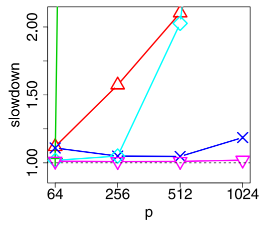
(e) ep



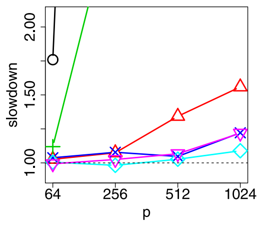
(f) ft



(g) is



(h) lu



(i) mg

Fig. 5.6: Runtime overhead for different implementations of a trace transfer.

Figure 5.6 summarizes the performance results for NPB3.3-MPI with problem size D when using the three different types of trace communication and both place configurations. Subfigure 5.6a shows the legend for the different communication layout and communication strategy combinations. The remaining figures show the runtime overhead for 64 to 1024 processes for these combinations. The all-to-1 cases fail to scale to 1024 processes for most of the kernels. Where the *Ssend* and *Isend* versions of the all-to-1 communication even fail for 256 processes for most kernels, the *Asend* strategy scales to up to 512 processes. Its main advantage is the reduction in messages arriving at the centralized manager, which leads to a lower workload. For the 1-to-1 cases, the *Ssend* strategy incurs a slowdown of up to 3 and hence fails to meet our performance goals. However, its slowdown does not necessarily increase with scale. Both the *Isend* and *Asend* strategies for the 1-to-1 cases incur a low runtime overhead, even at scale. These strategies only fail to achieve the desired less than 10% runtime overhead for the kernel mg. However, the problem size D of NPB3.3-MPI is a challenging test case at 1024 processes, as the fraction of the total execution time spent in MPI is very high at this scale. We expect better results for most applications.

5.6 Related Work

Several other MPI message checkers exist beyond Marmot [3] and Umpire [4] including MPI-Check [7] and ISP [8]. Both of these tools are not reported to scale to more than a hundred processes. Especially the complex analysis of alternative executions in ISP limits its scalability dramatically. We hope to combine our efforts with the ones for ISP in future work, as both tools have the same basic needs.

The generic tool infrastructure component of MUST relates to a wide range of infrastructure and scalability work. This includes P^nMPI [5], as well as infrastucture tools like MRNet [9], which we may use to implement several of the GTI components. Also, existing HPC tools like VampirServer [10] and Scalasca [11], or debuggers like DDT [12] and Totalview [13] may implement well adapted communication schemes that can be used for the GTI components. Further, these tools, as well as upcoming tools may employ modules of the GTI to implement their communications and may thus benefit from this component.

5.7 Conclusions

This paper presents a novel approach to create a runtime infrastructure for scalable MPI correctness checking. As far as we know, existing approaches – like Marmot and Umpire – lack the scalability needed for large HPC systems. Further, these tools use static communication systems that are hard to adapt to different types of systems. Also the implementation of new checks and the extension of existing ones is hard for these tools, as their checks are tightly coupled to their internal data structures and infrastructures. Our approach overcomes these problems by using a fine-grained module-based design that uses P^nMPI. We present an overview of this design and highlight our most important concepts that allow the offloading of checks to extra processes and threads. Further, we present a flexible communication system that promises an efficient transfer of trace records between different processes or threads. To demonstrate the feasibility of our design and to highlight the performance capabilities of our communication system, we present a performance study with a proof of concept implementation. This study shows that our ambitious runtime overhead goals are feasible, even at scale. In particular we demonstrate full MPI tracing for up to 1024 processes while transferring the trace records to extra processes without perturbing the application.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-10.ps (1995)
2. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi-20.ps (1997)
3. Krammer, B., Bidmon, K., Müller, M.S., Resch, M.M.: MARMOT: An MPI Analysis and Checking Tool. In Joubert, G.R., Nagel, W.E., Peters, F.J., Walter, W.V., eds.: PARCO. Volume 13 of Advances in Parallel Computing., Elsevier (2003) 493–500
4. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. Supercomputing, ACM/IEEE 2000 Conference (04-10 Nov. 2000) 51–51
5. Schulz, M., de Supinski, B.R.: P^{N}MPI Tools: A Whole Lot Greater Than the Sum of Their Parts. In: Supercomputing 2007 (SC'07). (2007)
6. Hilbrich, T., de Supinski, B.R., Schulz, M., Müller, M.S.: A Graph Based Approach for MPI Deadlock Detection. In: ICS '09: Proceedings of the 23rd international conference on Supercomputing, New York, NY, USA, ACM (2009) 296–305
7. Luecke, G.R., Zou, Y., Coyle, J., Hoekstra, J., Kraeva, M.: Deadlock Detection in MPI Programs. Concurrency and Computation: Practice and Experience **14**(11) (2002) 911–932
8. Vakkalanka, S.S., Sharma, S., Gopalakrishnan, G., Kirby, R.M.: ISP: A Tool for Model Checking MPI Programs. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2008) 285–286
9. Roth, P.C., Arnold, D.C., Miller, B.P.: MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In: SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2003) 21
10. Brunst, H., Kranzlmüller, D., Nagel, W.E.: Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. The International Series in Engineering and Computer Science, Distributed and Parallel Systems **777** (2005) 92–102
11. Wolf, F., Wylie, B., Abraham, E., Becker, D., Frings, W., Fuerlinger, K., Geimer, M., Hermanns, M., Mohr, B., Moore, S., Szebenyi, Z.: Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In: Proceedings of the 2nd HLRS Parallel Tools Workshop, Stuttgart, Germany (July 2008)
12. Edwards, D.J., Minsky, M.L.: Recent Improvements in DDT. Technical report, Alinea, Cambridge, MA, USA (1963)
13. Totalview Technologies: Totalview - Parallel and Thread Debugger. http://www.totalviewtech.com/products/totalview.html (July 2009)