An Interface for Integrated MPI Correctness Checking

Tobias HILBRICH [a], Matthias JURENZ [b], Hartmut MIX [b], Holger BRUNST [b], Andreas KNÜPFER [b], Matthias S. MÜLLER [b], and Wolfgang E. NAGEL [b]

[a] *GWT-TUD GmbH*
Chemnitzer Straße 48b, 01187 Dresden, Germany
tobias.hilbrich@zih.tu-dresden.de

[b] *Center for Information Services and High Performance Computing (ZIH)*
Technische Univeristät Dresden, D-01062 Dresden, Germany
{matthias.jurenz, hartmut.mix, ...}@tu-dresden.de

Abstract. Usage errors of the widely accepted Message-Passing Interface (MPI) are common and complicate the development process of parallel applications considerably. Some of these errors are hard to track, especially when they only occur in certain application runs or on certain platforms. Runtime correctness checking tools for MPI simplify the detection of these errors. However, they usually need the MPI profiling interface for their analysis. This paper addresses two issues related to correctness tools: First, due to the exclusive usage of the MPI profiling interface, it is not possible to use such tools in conjunction with other MPI tools, which are also based on the profiling interface. Second, correctness checking tools usually lack the ability to provide a detailed history of the events leading to an error, whereas such a history is provided naturally by tracing frameworks. We introduce the Universal MPI Correctness Interface (UniMCI) to overcome the first problem. This interface provides functions that invoke correctness checking and return detected errors in a manner that is independent of the correctness checker in use. Furthermore, we demonstrate the applicability of UniMCI with an implementation that uses the Marmot correctness checker and an exemplary integration of the interface into the VampirTrace performance analysis framework. As a result, we can provide a history for detected correctness events, which provides detailed information for debugging. Finally, we present a study using the SPEC MPI2007 benchmark to demonstrate the feasibility and applicability of our approach.

Keywords. Correctness checking, Message-Passing Interface, Tools, Marmot, Vampir

Introduction

Usage of the widely accepted Message-Passing Interface (MPI) standard [1] is simplified by a variety of tools for performance optimization, debugging, or other tasks. Using multiple tools simultaneously can help pinpoint issues faster, e.g., the combination of a tracing and a correctness tool can provide the history that leads to a correctness event and add further details to a detected error. Thus, it simplifies the identification of the root cause of an error.

Such runtime tools usually employ the MPI profiling interface (PMPI), which enables an interception of MPI calls along with an associated analysis of these

(a) Integrations without UniMCI (b) Integrations with UniMCI

Figure 1. Integrating a host tool with correctness checking tools.

calls. The usage of the profiling interface is exclusive, therefore, a simultaneous usage of two tools is not immediately possible. The P^nMPI [2] tool already solves this problem by allowing multiple tools to intercept MPI calls. Further problems of combining two tools are: First, the output of both tools should be merged into one combined output. And second, tools incorporating correctness outputs should not depend upon a specific MPI correctness tool, but rather use a generic interface that works with any correctness tool.

Our contribution is the design and implementation of the so called Universal MPI Correctness Interface (UniMCI) that addresses these problems. This paper is structured as follows: Section 1 introduces UniMCI and its design. Section 2 presents a tool combination of the correctness checker Marmot [3] and the tracing tool VampirTrace [4], to show the applicability of UniMCI. Section 3 provides an example application that demonstrates some of the benefits of this tool combination. We present a performance study for the VampirTrace-Marmot combination in Section 4, as such tool combinations have an impact on performance. Finally, we present related work and our conclusions in Sections 5 and 6.

1. UniMCI

Tool integration is necessary to incorporate MPI correctness checking functionality into another MPI tool, e.g., into a performance tool. We refer to the tool using the functionality of a correctness tool as the *host* tool, whereas the tool being used for correctness checking is referred to as the *guest* tool. One possible solution to combine a host and a guest tool is an integration that is specific to these two tools, e.g., a certain performance tool is integrated with a certain correctness tool. However, this direct approach would require the host tool to provide one integration for each guest tool. To avoid these extensive development costs and to simplify tool integration, we present the Universal MPI Correctness Interface (UniMCI). It provides correctness checking functionality independent of the actual correctness tool that is used. As a result, the host tool only needs one UniMCI integration to utilize any correctness checking tool that implements UniMCI. This is illustrated in Figures 1(a) and 1(b).

UniMCI is designed to be as portable and as independent of other tools as possible. As the MPI profiling interface can only be used by one tool at a time, a method is needed that allows both the host and the guest tool to analyze each MPI call. UniMCI offers two distinct solutions for this problem: first, name-shifted functions to pass MPI call arguments to the guest tool, and second, a prototype solution with P^nMPI [2]. The P^nMPI based solution will be released with a future UniMCI version and simplifies the tool coupling. However, it requires that a P^nMPI installation is present and that both the host and the guest tool support P^nMPI, which usually requires both tools to be available as a shared library. We will only present details for the name-shifted interface due to space considerations.

```
/*Wrapper for MPI_Send*/
int MPI_Send(void* buf, int count, MPI_Datatype type, int dest, int tag,
    MPI_Comm comm)
{
  ...
  UNIMCI_check_pre_MPI_Send(buf, count, type, dest, tag, comm, FILE, LINE, ID);
  check_unimci_return();
  PMPI_Send(buf, count, type, dest, tag, comm);
  UNIMCI_check_post_MPI_Send(buf, count, type, dest, tag, comm, FILE, LINE, ID);
  check_unimci_return();
  ...
}

/*Error evaluation function*/
void check_unimci_return()
{
  UNIMCI_MSG *msg;
  while (UNIMCI_has_msg ())
  {
    UNIMCI_pop_msg (&msg);
    /*Do something with msg, e.g., log it into output*/
    UNIMCI_msg_free (&msg);
  }
}
```

Figure 2. Host tool using UniMCI, with name-shifted interface.

The name-shifted interface provides two functions for each MPI call: one to analyze the initial arguments of the MPI call, called *pre* check, and one to analyze the results of the MPI call, called *post* check. All runtime MPI checkers will need an analysis of the initial MPI call arguments to detect errors in the given arguments, which is invoked with the pre check function of UniMCI. The post check is usually needed for MPI calls that create resources, e.g., `MPI_Isend`, which creates a new request. MPI correctness tools have to add these new resources to their internal data structures, in order to be aware of all valid handles and their respective state. By splitting the analysis of the MPI call into two parts it is possible to return check result to the host tool before the actual MPI call is issued, which is important to guarantee that errors are handled before the application might crash.

Results of checks are returned with additional interface functions that have to be issued after each check function. A simple correctness message record is used to return problems detected by the guest tool. Figure 2 provides an example of how a host tool can use UniMCI in its MPI wrappers. Note that even though the number of additional source lines needed for UniMCI seems high, most MPI wrappers are created by wrapper generators, which simplifies the integration of the extra code. The additional arguments *FILE*, *LINE*, and *ID* are used to provide source information to the guest tool. *FILE* and *LINE* identify the source code location of the wrapped MPI call. The *ID* argument is used to pass an identifier to the guest tool that is used in correctness messages resulting from this call. This functionality may be used by the host tool to correlate correctness events with past MPI calls. Future versions of the interface will contain extensions and rules to handle asynchronous correctness checking tools and multi-threaded applications.

2. Marmot-VampirTrace with UniMCI

This section presents details on a first correctness tool that supports UniMCI and a first host tool that utilizes the interface. As a guest tool, we use Marmot [3], a portable MPI checker, which is already integrated into various tools, IDE's, and debuggers. As a host tool, we choose VampirTrace [4], a performance event tracing tool.

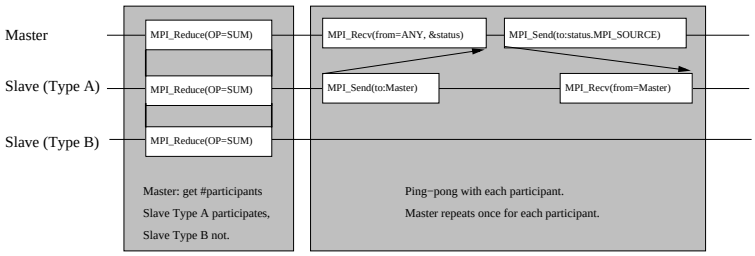


Figure 3. Communication pattern for master and slaves in example 4.

Marmot executes various local checks to detect MPI usage errors and non-portable constructs. Further, one of the MPI processes is used to execute global checks, e.g., a timeout based deadlock detection. This process is also informed about each MPI call, which causes significant overhead. As tool combinations with performance tools will become impractical if the execution of the guest tool is too expensive, a low overhead version of Marmot was used for the implementation of UniMCI by disabling this global process. This does not affect Marmot's powerful local checks.

Marmot's implementation of UniMCI is mostly straightforward. As a first step, Marmot's core functionality was refactored in order to clearly separate pre checks from post checks. Further, a new logger was introduced to store MPI messages for retrieval by the host tool. In order to provide the necessary functions to retrieve Marmot's messages, we extended Marmot's external interface. The name-shifted interface is implemented with generated functions that are very similar to Marmot's regular MPI wrappers. Marmot's global process is disabled to remove its performance bottleneck.

The first host tool that uses UniMCI is VampirTrace. It detects the presence of a UniMCI installation during its *configure* step. If an installation is available it queries the provided *unimci-config* tool for the libraries and flags needed to compile and link with UniMCI. VampirTrace uses the name-shifted interface in its MPI wrappers. This is an optional feature of VampirTrace and is controlled with environmental variables.

VampirTrace stores UniMCI correctness events in a separate *marker* file. Each marker is associated with a timestamp and a process. Further, each marker has a type, a name, and a textual message. This data is used during visualization with *VampirServer* [5]. Each marker is added to the global timeline at its respective process and timestamp. Detailed information can be retrieved by clicking on the visualized marker.

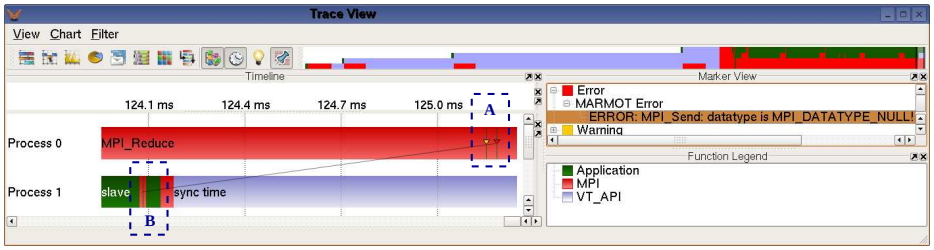
3. Example

To demonstrate the benefits of a VampirTrace and Marmot combination, we present an artificial example with an MPI usage error. The usual synthetic MPI usage error is simple to solve and becomes obvious when the error is detected by a correctness tool. However, some errors are more complex and result from a series of MPI calls. VampirTrace is able to visualize such a history of calls, which simplifies debugging in its presence. As a result, here we present an error which is more complicated to track.

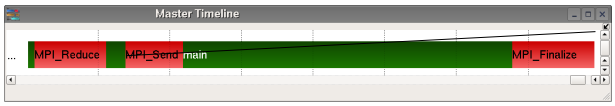
For space considerations we can't present the full example code here, rather we will highlight the parts that are related to the correctness error. The exam-

```
33 for (i = 0; i < num_participating_slaves; i++)
34 {
35   MPI_Recv (&buf, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
36   MPI_Send (sendbuf[buf], 1, type[buf], status.MPI_SOURCE, tag, MPI_COMM_WORLD);
37 }
```

Figure 4. Example of a complex potential MPI error (master process).



(a) Global timeline for processes 0 and 1, zoomed into the error region.



(b) Detail information for process 1, zoomed into the area labeled *B* in (a).

Figure 5. Vampir visualization for the example from Figure 4

ple uses a master slave communication were the communication pattern is repeated in multiple iterations. The communication used for each of the iterations is schemed in Figure 3. Each slave computes whether it is going to participate in this iteration's communication or not. Slaves that participate are of *type A* and will do a ping-pong communication with the master process, whereas slaves that do not participate are of *type B*. An `MPI_Reduce` is used to determine the total number of slaves that are going to participate in an iteration. The master repeats a wildcard receive and a send, in order to implement the ping-pong communication with each of the participating slaves. Figure 4 presents the lines of code that are used by the master process to implement the ping-pong communication. Note that the send uses a datatype and a buffer based on the value received in the preceding wildcard receive. Finally, an additional communication is started after all the iterations ended. During this final communication each slave sends one message to the master, whereas the master calls `MPI_Recv` once for each slave.

This code contains a potential MPI error, which manifests for some, but not all runs. The application is likely to crash if the error manifests. When Marmot is used without any other tool and the error manifests, it is able to detect the MPI call that causes the application crash. Marmot detects the usage of an invalid datatype(`MPI_DATATYPE_NULL`) in the `MPI_Send` call of line 36 on the master process. However, this output is only of limited use, as the datatype used in this call depends on the value received in the `MPI_Recv` call of line 35. As this is a wildcard receive, it is unclear which value was received there.

When using the UniMCI based Marmot and VampirTrace integration, it is possible to analyze the series of events that lead to this situation. Figure 5(a) shows a VampirServer visualization for a trace file that contains the error. It shows a global timeline of the process activities. The light gray bars represent MPI calls, whereas the dark gray areas represent the execution of the actual application.

Depicted is a zoomed-in part of the area in which the error occurred. The Marmot messages are represented with little triangles on the respective MPI calls that caused the message. Two areas of the timeline are highlighted as *A* and *B* in the figure. The second triangle in area A is the marker that points to the detected error. The error text is displayed to the right of the timeline. The line from area B to area A in Figure 5(a) shows the message that was received by the `MPI_Recv` call of line 35. As discussed above, this call determines which datatype is used by the `MPI_Send` call that causes the error. Figure 5(b) shows details for the area B from the timeline in Figure 5(a). It identifies the individual activities of the process that sends this message (process 1). The presence of the `MPI_Finalize` call after the send call reveals that this send call should not have matched the receive of the master process. It is not a send call from one of the iterations, but rather the send call from the final communication that is executed after all the iterations have ended. This reveals the source of the error, a slave of type B, which does not participate in the ping-pong communication of the last iteration, may already enter the final communication. Thus, the master process may receive a wrong message in line 35.

Solutions to this problem are either synchronization calls, e.g., usage of a `MPI_Barrier` or different tags for the communications.

4. Performance Results

This section presents a performance study for the VampirTrace and Marmot combination introduced in this paper. Both tools incur an overhead at runtime to perform their respective analyses, of interest is whether the combined overhead limits the applicability of the tool combination. We use the SPEC MPI2007 [6] benchmark as a challenging test suite. It contains 13 different application codes from various fields of science. The benchmark uses three different sizes for its data sets, from which we use the largest one (mref). All applications are strong scaling, which allows us to analyze the impact of message sizes on the total tool overhead. For our experiments we use 32, 64, and 128 MPI processes with three different versions of the benchmarks. One version without any tools, one version that only uses VampirTrace, and one version with the VampirTrace-Marmot integration from this paper. As all of the codes in the SPEC MPI2007 benchmark are widely used production codes, the presence of MPI usage error is very unlikely. We use an SGI Altix 4700 to run our experiments.

Figure 6 presents the slowdown of VampirTrace and the VampirTrace-Marmot combination for all of the SPEC MPI2007 applications at different scales. Both VampirTrace and the tool combination incur an overhead of less than 10% for all applications when using 32 processes. Only *121.pop* is an exception that is known to be very challenging for MPI runtime tools, as it uses a very high amount of small messages.

For the applications *113.GemsFDTD*, *115.fds4*, *122.tachyon*, and *126.lammps*, the tool combination still has a low overhead at a scale of 128 processes. For the remaining applications the overhead increases with scale. As these are strong scaling problems it is expected that the tool overhead increases with scale. This results from a decreased average message size. As intercepting and analyzing MPI calls with VampirTrace and Marmot is independent of the message size, the overhead per MPI call increases, as the actual communication takes less time. Both

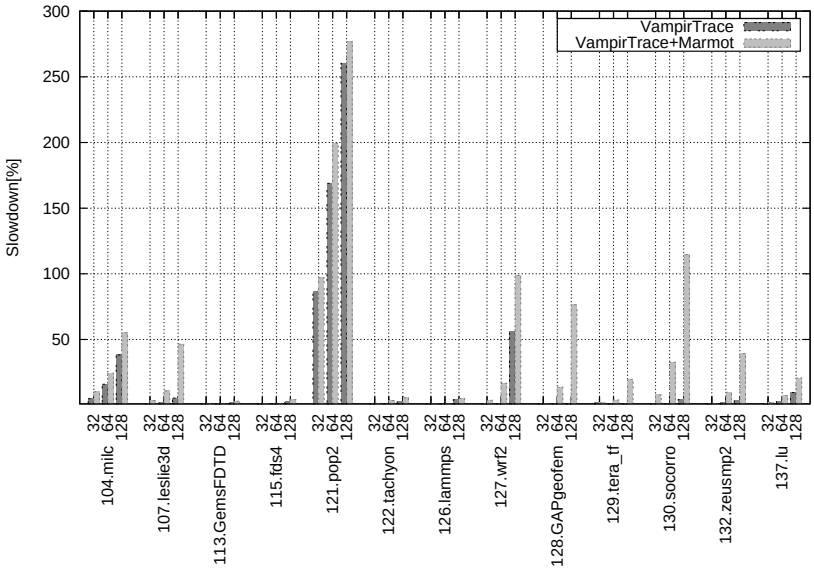


Figure 6. Slowdown for SPEC MPI2007 with 32 to 128 tasks.

104.milc and *137.lu* cause VampirTrace and Marmot to incur a higher overhead at increased scale. Whereas for *107.leslie3d*, *128.GAPgeofem*, *129.tera_tf*, *130.socorro*, and *132.zeusmp2* the increase in Marmot's overhead is dominating. This results from increased complexity of correctness checks. A common case where this happens is the usage of one or multiple MPI request for each process. With increased scale, the total number of concurrently existing requests increases, which adds complexity to the verifications of *MPI_Request* arguments. These overheads may be reduced in future Marmot releases as its performance was – up to now – only of second concern, as long as its usage was still feasible.

5. Related Work

Employing UniMCI as an interface for MPI correctness checking is a reliable and portable way of integrating correctness data into other tools. Future versions of the interface will also offer a P^nMPI [2] based interface that further simplifies the usage of UniMCI. Additional guest tools that might implement this interface are MPI-Check [7], Umpire [8], and ISP [9]. Some of these tools provide more powerful global checks, e.g., Umpire and ISP. As a result, an UniMCI implementation from these tools would provide UniMCI users with a larger set of correctness features.

Usage of UniMCI may also affect a variety of different host tools. Other performance tools, besides VampirTrace, that might use UniMCI are Scalasca [10] and Tau [11]. With a UniMCI integration, these tools may provide MPI correctness checking to users that are not aware of the underlying guest tools. Especially an integration of UniMCI into Scalasca is very promising, as both Scalasca and Marmot use the same output format.

6. Conclusions

This paper presents UniMCI, a universal interface for incorporating runtime MPI correctness checking functionality into host tools such as tracing frameworks. Two distinct modes of usage provide a straightforward integration of UniMCI into

existing tools. The first mode uses a name-shifted interface, whereas the second mode uses P^nMPI. We demonstrate the applicability of UniMCI with a sample implementation of the interface in the Marmot correctness checker and the VampirTrace monitoring tool. Extensions to VampirServer enable the visualization of identified correctness events collected by VampirTrace.

A detailed example demonstrates the advantage of a VampirTrace and Marmot combination based on UniMCI. This example shows that understanding source code errors is simplified when using a combined output of these two tools. A further contribution of this paper is a performance study for the SPEC MPI2007 benchmark. While the concurrent usage of Marmot and VampirTrace adds up their respective overheads, we demonstrate the applicability of this tool combination for multiple applications with up to 128 MPI processes.

Acknowledgments
The research presented in this paper has partially been supported by the German Ministry for Research and Education (BMBF) through the ITEA2 project "ParMA" [12] (No 06015, June 2007 – May 2010).

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1. http://www.mpi-forum.org/docs/mpi21-report.pdf, September 2008.

[2] M. Schulz and B.R. de Supinski. P^NMPI Tools A Whole Lot Greater Than the Sum of Their Parts. In *Supercomputing 2007 (SC'07)*, 2007.

[3] B. Krammer, K. Bidmon, M.S. Müller, and M.M. Resch. MARMOT: An MPI Analysis and Checking Tool. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *PARCO*, volume 13 of *Advances in Parallel Computing*, pages 493–500. Elsevier, 2003.

[4] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M.S. Müller, and W.E. Nagel. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, pages 139–155. Springer Verlag, July 2008.

[5] H. Brunst, D. Kranzlmüller, and W.E. Nagel. Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. *The International Series in Engineering and Computer Science, Distributed and Parallel Systems*, 777:92–102, 2005.

[6] SPEC MPI2007 Benchmark Suite for MPI. http://www.spec.org/mpi2007/.

[7] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[8] J.S. Vetter and B.R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. *Supercomputing, ACM/IEEE 2000 Conference*, pages 51–51, 04-10 Nov. 2000.

[9] S.S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R.M. Kirby. 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*, pages 285–286, New York, NY, USA, 2008. ACM.

[10] F. Wolf, B. Wylie, E. Abraham, D. Becker, W. Frings, K. Fuerlinger, M. Geimer, M. Hermanns, B. Mohr, S. Moore, and Z. Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Tools for High Performance Computing*, pages 157–167. Springer Verlag, July 2008.

[11] D. Brown, S. Hackstadt, A. Malony, and B. Mohr. Program Analysis Environments for Parallel Language Systems: The TAU Environment. In *In Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing*, pages 162–171, 1994.

[12] ParMA: Parallel Programming for Multi-core Architectures - ITEA2 Project (06015). http://www.parma-itea2.org/.