John von Neumann Institute for Computing

Coupling DDT and Marmot for Debugging of MPI Applications

Bettina Krammer, Valentin Himmler, David Lecomber

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 653-660, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Coupling DDT and Marmot for Debugging of MPI Applications

Bettina Krammer[1], Valentin Himmler[1], and David Lecomber[2]

[1] HLRS - High Performance Computing Center Stuttgart
Nobelstrasse 19, 70569 Stuttgart, Germany
E-mail: {krammer, himmler}@hlrs.de

[2] Allinea Software,
The Innovation Centre, Warwick Technology Park, Gallows Hill
Warwick, CV34 6UW, UK
E-mail: david@allinea.com

Parallel programming is a complex, and, since the multi-core era has dawned, also a more common task that can be alleviated considerably by tools supporting the application development and porting process. Existing tools, namely the MPI correctness checker Marmot and the parallel debugger DDT, have so far been used on a wide range of platforms as stand-alone tools to cover different aspects of correctness debugging. In this paper we will describe first steps towards coupling these two tools to provide application developers with a powerful and user-friendly environment.

1 Introduction

The Message Passing Interface (MPI) has been a commonly used standard[1,2] for writing parallel programs for more than a decade, at least within the High Performance Computing (HPC) community. With the arrival of multi-core processors, parallel programming paradigms such as MPI or OpenMP will become more popular among a wider public in many application domains as software needs to be adapted and parallelised to exploit fully the processor's performance. However, tracking down a bug in a distributed program can turn into a very painful task, especially if one has to deal with a huge and hardly comprehensible piece of legacy code.

Therefore, we plan to couple existing tools, namely the MPI correctness checker, Marmot, and the parallel debugger, DDT, to provide MPI application developers with a powerful and user-friendly environment. So far, both tools have been used on a wide range of platforms as stand-alone tools to cover different aspects of correctness debugging. While (parallel) debuggers are a great help in examining code at source level, e.g. by monitoring the execution, tracking values of variables, displaying the stack, finding memory leaks, etc., they give little insight into *why* a program actually gives wrong results or crashes when the failure is due to incorrect usage of the MPI API. To unravel such kinds of errors, the Marmot library has been developed. The tool checks at run-time for errors frequently made in MPI applications, e.g. deadlocks, the correct construction and destruction of resources, etc., and also issues warnings in the case of non-portable constructs.

In the following sections, we will shortly describe both tools and discuss our first considerations and results towards integrating the two of them.

1.1 DDT - Distributed Debugging Tool

Allinea Software's DDT[21] is a source-level debugger for scalar, multi-threaded and large-scale parallel C, C++ and Fortran codes. It provides complete control over the execution of a job and allows the user to examine in detail the state of every aspect of the processes and threads within it.

Control of processes is aggregated using groups of processes which can be run, single stepped, or stopped together. The user can also set breakpoints at points in the code which will cause a process to pause when reaching it.

When a program is paused, the source code is highlighted showing which lines have threads on them and, by simply hovering the mouse, the actual threads present are identified. By selecting an individual process, its data can be interrogated - for example to find the local variables and their values, or to evaluate specific expressions.

There are many parts of the DDT interface that help the user get a quick understanding of the code at scale - such as the parallel stack view which aggregates the call stacks of every process in a job and displays it as tree, or the cross-process data comparison tools.

Specifically to aid MPI programming, there is a window that examines the current state of MPI message queues - showing the send, receive and unexpected receive buffers. However, this can only show the current state of messages - and bugs due to the historic MPI behaviour are not easy to detect with only current information.

1.2 Marmot - MPI Correctness Checker

Marmot[9–11,14] is a library that uses the so-called PMPI profiling interface to intercept MPI calls and analyse them during runtime. It has to be linked to the application in addition to the underlying MPI implementation, not requiring any modification of the application's source code nor of the MPI library. The tool checks if the MPI API is used correctly and checks for errors frequently made in MPI applications, e.g. deadlocks, the correct construction and destruction of resources, etc. It also issues warnings for non-portable behaviour, e.g. using tags outside the range guaranteed by the MPI-standard. The output of the tool is available in different formats, e.g. as text log file or html/xml, which can be displayed and analysed using a graphical interface. Marmot is intended to be a portable tool that has been tested on many different platforms and with many different MPI implementations.

Marmot supports the complete MPI-1.2 standard for C and Fortran applications and is being extended to also cover MPI-2 functionality[12,13].

Figure 1 illustrates the design of Marmot. Local checks including verification of arguments such as tags, communicators, ranks, etc. are performed on the client side. An additional MPI process (referred to as *debug server*) is added for the tasks that cannot be handled within the context of a single MPI process, e.g. deadlock detection. Another task of the debug server is the logging and the control of the execution flow. Every client has to register at the debug server, which gives its clients the permission for execution in a round-robin way. Information is transferred between the original MPI processes and the debug server using MPI.

In order to ensure that the debug server process is transparent to the application, we map `MPI_COMM_WORLD` to a Marmot communicator that contains only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they will also automatically exclude the debug server process. This mapping is done at start-up time

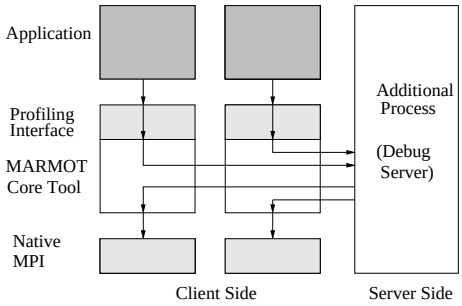

Figure 1. Design of Marmot.

in the `MPI_Init` call, where we also map all other predefined MPI resources, such as groups or datatypes, to our own Marmot resources. When an application constructs or destructs resources during run-time, e.g. by creating or freeing a user-defined communicator, the Marmot maps are updated accordingly. Having its own book-keeping of MPI resources, independently of the actual MPI implementation, Marmot can thus verify correct handling of resources.

1.3 Related Work

Debugging MPI programs can be addressed in various ways. The different solutions can be roughly grouped in four different approaches:

1. *classical debuggers:* Among the best-known parallel debuggers are the commercial debuggers DDT[21] and Totalview[22]. It is also possible to attach the freely available debugger gdb[20] to single MPI processes.

2. The second approach is to provide a *special debug version* of the MPI library, for instance for checking collective routines, e.g. mpich[7] or NEC MPI[18].

3. Another possibility is to develop tools for *run-time analysis* of MPI applications. Examples of such an approach are MPI-CHECK[16], Umpire[19] and Marmot[9]. MPI-CHECK is currently restricted to Fortran code and performs argument type checking or finds problems such as deadlocks. Like Marmot, Umpire[19] uses the profiling interface.

4. The fourth approach is to collect all information on MPI calls in a trace file for *post-mortem analysis*, which can be analysed by a separate tool after program execution[15, 17]. A disadvantage with this approach is that such a trace file may be very large. However, the main problem is guaranteeing that the trace file is written in the presence of MPI errors, because the behaviour after an MPI error is implementation defined.

2 Coupling DDT and Marmot

The ultimate goal of coupling both tools is to create a common software development environment with debugging and correctness checking facilities at the same time, comple-

menting each other and, thus, offering the opportunity to detect a maximum number of bugs on different levels, for example, bugs being due to incorrect usage of the MPI API, memory leaks, etc. In order to do so, a number of considerations have to be taken into account and adaptations have to be implemented in both tools.

- **Marmot's `MPI_Init`:** As described in Section 1.2, Marmot requires one additional process for the debug server, which is always running on the process with highest rank. That means while the first n processes call our redefined `MPI_Init`, i.e. call `PMPI_Init`, take part in the special Marmot start-up and return from the `MPI_Init` call to proceed with the application's code, the debug server (process $n + 1$) always remains in `MPI_Init` to execute Marmot's debug server code. When executing an application with DDT and Marmot, DDT's environment variable `DDT_MPI_INIT` has to be set to the value `PMPI_Init` to circumvent problems with attaching DDT also to this last process, because the DDT start-up procedure normally requires a process to return from the `MPI_Init` call.

  While the debug server process is transparent to the application in the Marmot-only approach, i.e. from the user's point of view, showing silent signs of life by announcing errors and producing a logfile without being visible as additional MPI process, it is in the coupled approach currently also displayed in DDT's graphical interface and is, thus, visible to the user in the same way as one of the original application's MPI processes. As this may be confusing to users, the ultimate goal will be to hide Marmot's debug server process within DDT completely and to allow steering of Marmot through a plugin.

- **Marmot's breakpoints:** One reason for currently still displaying this debug server process – though it has nothing to do with the actual application itself – is that it allows users to set breakpoints that will be hit when Marmot finds an error or issues a warning. For this purpose, we implemented simple functions, named e.g. `insertBreakpointMarmotError`, that are called by Marmot's debug server in such an event, see Fig.2 left. The screenshot on the bottom shows the application's code pausing at the erroneous line, with the last process pausing in `MPI_Init` as explained above. As it has control over the execution flow, hitting one of the debug server's special breakpoints implies that the application processes cannot continue with the erroneous MPI call nor perform any further MPI calls, respectively.

  As part of the integration with Marmot, DDT will automatically add breakpoints into these "stub" functions and will thus alert the user to a Marmot-detected error or warning by pausing the program.

- **Error dialogues:** Having detected an error with Marmot, the question is how to get the warning message across to the user within DDT. Currently, a user will find such messages in Marmot's log file, or in DDT's variables or stderr windows at runtime, as shown in Fig.3.

  Simple text and error code are shown in DDT's local variables panel at the breakpoints inside the "stub" error functions - these will be passed as arguments to those functions. Extra integration is planned, such as displaying more detailed information and providing extra windows to display the context of errors more clearly.

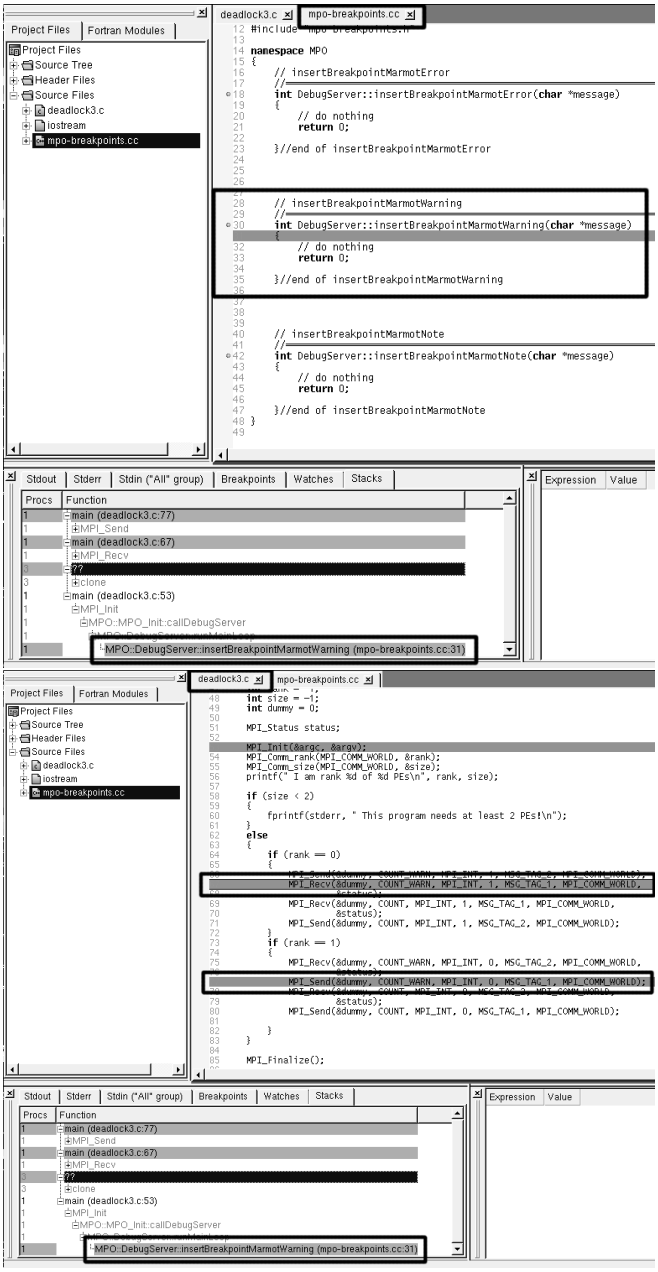

Figure 2. Setting breakpoints in Marmot's debug server process (top) causes program to stop at an error/warning detected by Marmot - in this case, a warning for using `COUNT_WARN= 0` in send/recv calls (bottom)

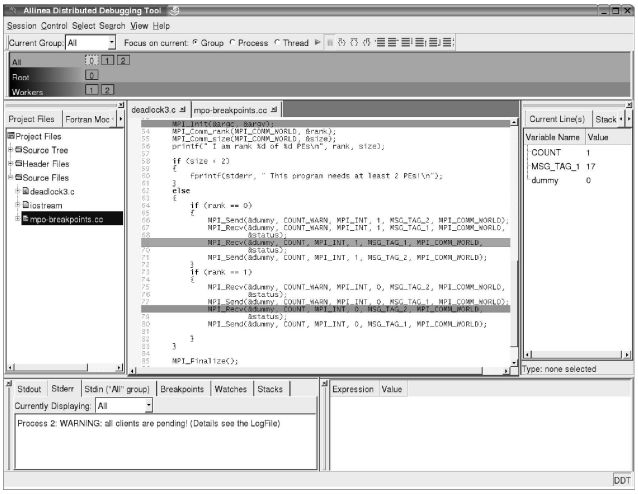

Figure 3. Deadlock warning from Marmot

- **Dynamic linking:** Currently, Marmot is built as static libraries per default. While this is an advantage concerning the performance, it is a drawback in the integration process of Marmot and DDT. Beside the fact that the size of statically linked applications can be very large compared to those dynamically linked, a more severe issue is, that as soon as an application is linked statically against the Marmot libraries, this application cannot be run without Marmot. Our goal is to be able to activate Marmot for a given run within DDT, and to disable it for another, using one and the same binary. The trick is to preload the dynamic Marmot libraries by using the environment variable `LD_PRELOAD`.

  This will tell the dynamic linker to first look in the Marmot libraries for any MPI call, provided that also the MPI libraries themselves are dynamic. For switching off Marmot, the variable will simply be unset. Note that in this case neither at compile time nor at link time the application has to be aware of the very existence of Marmot. Only at runtime the dynamic linker decides whether the MPI calls will be intercepted by Marmot or not. Switching Marmot on and off could then be as easy as ticking a checkbox in the DDT GUI, which consequently takes care of the preloading and automatically adds one extra process for the debug server.

  Although not all MPIs export the `LD_PRELOAD` setting to all processes in a job seamlessly, DDT already has the mechanism to do this for every MPI - and therefore extending it to preload the Marmot libraries will be a simple task.

3 First Results

Running applications with Marmot in conjunction with DDT has been tested on various HLRS clusters using different MPI Implementations, e.g. different vendor MPIs or Open Source implementations such as mpich[3,4] or Open MPI[5,6].

As a proof of concept, some simple programs were used for testing. For instance, the example shown in Fig.2 and 3 first triggers a warning by using send and receive calls with count 0 and then an error by using send and receive calls in wrong order (deadlock).

Building and preloading shared libraries were successfully tested on an Intel Xeon cluster with Open MPI 1.2.3 and IntelMPI 3.0, using the Intel Compiler 10.0. For both MPI implementations the build process generally involves the following steps: generating objects with *position independent code* and building shared libraries from these object files.

A given MPI application is invoked e.g. by the command `mpirun -np <n> ./myMPIApp`, where n denotes the number of processes. The same application can be run with Marmot by setting the appropriate environment variable and adjusting the number of processes:

```
env LD_PRELOAD="<marmotlib>" mpirun -np <n+1> ./myMPIApp.
```

4 Concluding Remarks

We have presented the DDT parallel debugger and the Marmot MPI correctness checker, which have been used successfully as stand-alone tools so far. Recently we have made first efforts to combine both tools so that MPI application developers get better insight into *where* and *why* their programs crash or give wrong results. The approach taken looks promising and could be verified using simple test programs.

Future work includes tests with real applications and technical improvements, e.g. a tighter integration of the display of warnings and message queues, thus being more user-friendly, or enhancements in Marmot's build process.

Acknowledgements

The research presented in this paper has partially been supported by the European Union through the IST-031857 project "int.eu.grid"[14] (May 2006 – April 2008) and by the German Ministery for Research and Education (BMBF) through the ITEA2 project "ParMA"[8] (June 2007 – May 2010). Marmot is being developed by HLRS in collaboration with ZIH Dresden.

References

1. Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, (1995). `http://www.mpi-forum.org`
2. Message Passing Interface Forum, *MPI-2: Extensions to the Message Passing Interface*, (1997). `http://www.mpi-forum.org`
3. mpich Homepage. `www.mcs.anl.gov/mpi/mpich`
4. W. Gropp, E. L. Lusk, N. E. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI Message Passing Interface standard*. Parallel Computing, **22**, 789–828, (1996).
5. Open MPI Homepage. `http://www.open-mpi.org`

6. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham and T. S. Woodall, *Open MPI: goals, concept, and design of a next generation MPI implementation*, in: 11th European PVM/MPI, LNCS **3241**, pp. 97–104, (Springer, 2004).
7. E. Lusk, C. Falzone, A. Chan and W. Gropp, *Collective error detection for MPI collective operations*, in: 12th European PVM/MPI, LNCS **3666**, pp. 138–147, (Springer, 2005).
8. ParMA: Parallel Programming for Multi-core Architectures - ITEA2 Project (06015). `http://www.parma-itea2.org/`
9. B. Krammer, K. Bidmon, M. S. Müller and M. M. Resch, *MARMOT: An MPI analysis and checking tool.*, in: PARCO 2003, Dresden, Germany, (2003).
10. B. Krammer, M. S. Müller and M. M. Resch, *MPI application development using the analysis tool MARMOT*, in: ICCS 2004, LNCS **3038**, pp. 464–471. (Springer, 2004).
11. B. Krammer, M. S. Mueller and M. M. Resch, *Runtime checking of MPI applications with MARMOT*. in: *ParCo 2005*, Malaga, Spain, (2005).
12. B. Krammer, M. S. Müller and M. M. Resch, *MPI I/O analysis and error detection with MARMOT*, in: Proc. EuroPVM/MPI 2004, Budapest, Hungary, September 19–22, 2004, LNCS vol. **3241**, pp. 242–250, (Springer, 2004).
13. B. Krammer and M. M. Resch, *Correctness checking of MPI one-sided communication using MARMOT*, in: Proc. EuroPVM/MPI 2006, Bonn, Germany, September 17–20, 2006, LNCS vol. **4192**, pp. 105–114, (Springer, 2006).
14. B. Krammer, *Experiences with MPI application development within int.eu.grid: interactive European grid project*, in: Proc. GES2007, Baden-Baden, Germany, May 2–4, (2007).
15. D. Kranzlmüller, *Event graph analysis for debugging massively parallel programs*, PhD thesis, Joh. Kepler University Linz, Austria, (2000).
16. G. Luecke, Y. Zou, J. Coyle, J. Hoekstra and M. Kraeva, *Deadlock detection in MPI programs*, Concurrency and Computation: Practice and Experience, **14**, 911–932, (2002).
17. B. Kuhn, J. DeSouza and B. R. de Supinski, *Automated, scalable debugging of MPI programs with Intel Message Checker*, in: SE-HPCS '05, St. Louis, Missouri, USA, (2005). `http://csdl.ics.hawaii.edu/se-hpcs/papers/11.pdf`
18. J. L. Träff and J. Worringen, *Verifying collective MPI calls*, in: 11th European PVM/MPI, LNCS vol. **3241**, pp. 18–27, (Springer, 2004).
19. J. S. Vetter and B. R. de Supinski, *Dynamic software testing of MPI applications with Umpire*, in: SC 2000, Dallas, Texas, (ACM/IEEE, 2000). CD-ROM.
20. gdb. The GNU Project Debugger. `http://www.gnu.org/manual/gdb`.
21. DDT. `http://www.allinea.com/index.php?page=48`
22. Totalview. `http://www.totalviewtech.com/productsTV.htm`.