H L R S

ZIH

Center for Information Services & High Performance Computing

Detecting MPI Usage Errors with Marmot

Version 1.1, published June 2009

First version published: June 2008

Contents

Contributors and Acknowledgements

Blasius Czink, HLRS
Tobias Hilbrich, ZIH
Bettina Krammer, UVSQ - LRC ITACA
Matthias Müller, ZIH
Kiril Dichev, HLRS
Valentin Himmler
Ronny Zschitzschmann

Introduction

Debugging and Porting Parallel MPI Applications

Parallel programming is a complex, and since the multi-core era has dawned, also a more and more common task. In addition to all the problems known from serial programming, application developers now have to find and fix bugs in a jungle of parallel tasks that may not behave deterministically and may be subject to new error classes such as deadlocks or race conditions. Understanding the order of execution of parallel code and ensuring correctness of software, especially when running on a higher number of processors, puts high demands on application developers. The Message Passing Interface (MPI) is widely used to write parallel programs using message passing, but - as a further example of the difficulties of parallel programming - it does not guarantee portability between different MPI implementations. When an application runs without any problems on one platform but crashes or gives wrong results on another platform, developers tend to blame the compiler, the architecture or the MPI implementation. In many cases the problem is a subtle programming error in the application undetected on the platforms used previously. Finding this kind of bugs can be a very strenuous and difficult task. In order to alleviate this process considerably, tools that support the application development and porting process are necessary.

Marmot an MPI Correctness Checker

In this paper we present the Marmot tool, an automated correctness checker for MPI applications during runtime. The tool checks all the parameters passed to MPI calls and detects usage errors like introduction of irreproducibility, deadlocks, incorrect management of resources such as communicators, groups, data-types etc., or the use of non-portable constructs. In order to detect global usage errors like deadlocks an additional MPI process is used.

Debugging with Marmot

Marmot is usually either applied during application development or when porting an existing MPI application to another system. In order to use Marmot it is necessary to link additional libraries to the application. Once this is done the application is executed as normal, except that one additional process is needed, and MPI usage errors are reported. Finding all errors is usually an iterative process in which different sets of input data are used and possible solutions to issued errors are tested. Marmot's output files also give links to the MPI standard document to enhance the developers understanding of the identified problems. Once the application is presumably error-free the additional Marmot libraries should be removed in order to enable production runs with optimal performance.

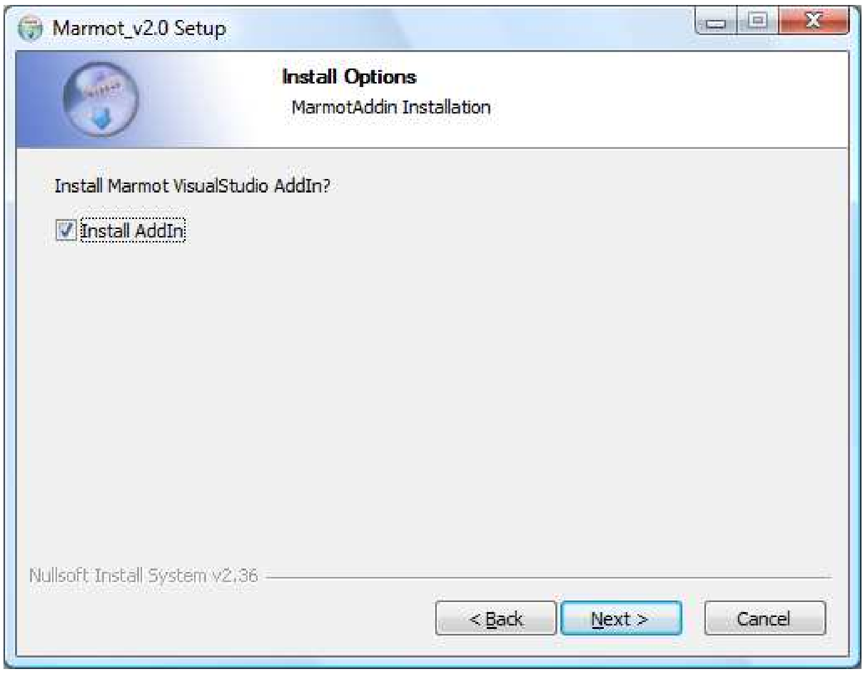
Installation and Deployment

Download

Marmot is available for download as a source distribution or a precompiled version for Microsoft® Windows® HPC Server 2008, Microsoft HPC Pack 2008 and Visual Studio 2008 from the projects website (http://www.hlrs.de/organization/av/amt/research/marmot). For a different configuration using CCPv1 and Visual Studio 2005 the user will have to rebuild Marmot from source. Detailed information on configuring / building Marmot with *CMake* is available in the Manual.

Installation

Installing a precompiled version of Marmot is as easy as launching the setup-application and selecting a destination folder. In case of an installed Visual Studio 2005 or 2008 the setup will offer an option to install the Marmot-AddIn, which is recommended. The AddIn will be installed for the current user only.



Marmot_v2.0 Setup

Install Options

MarmotAddin Installation

Install Marmot VisualStudio AddIn?

[✓] Install AddIn

Nullsoft Install System v2.36

< Back   Next >   Cancel

Figure: 1 Marmot Setup

It is not necessary to deploy any files on the compute cluster nodes since Marmot is statically linked to an application in the current distribution. The Microsoft Runtime Libraries should be available though. If not they can be easily installed with *classrun* and a silent install of the *vcredist*-package.

What is installed

- **bin** - contains a copy of the MarmotAddin.dll and compile-wrapper scripts
- **include** - Marmot header files
- **lib** - Marmot libraries (debug versions with a “D” suffix)
- **examples** – Example Visual Studio projects

- **share -** contains Marmot documentation (Userguide, MPI-Standard-Reference) and CMake module files (used in conjunction with CMake)

It is not necessary though helpful to set an environment-variable MARMOT_HOME to the Marmot installation directory.

Using Marmot

Example Code

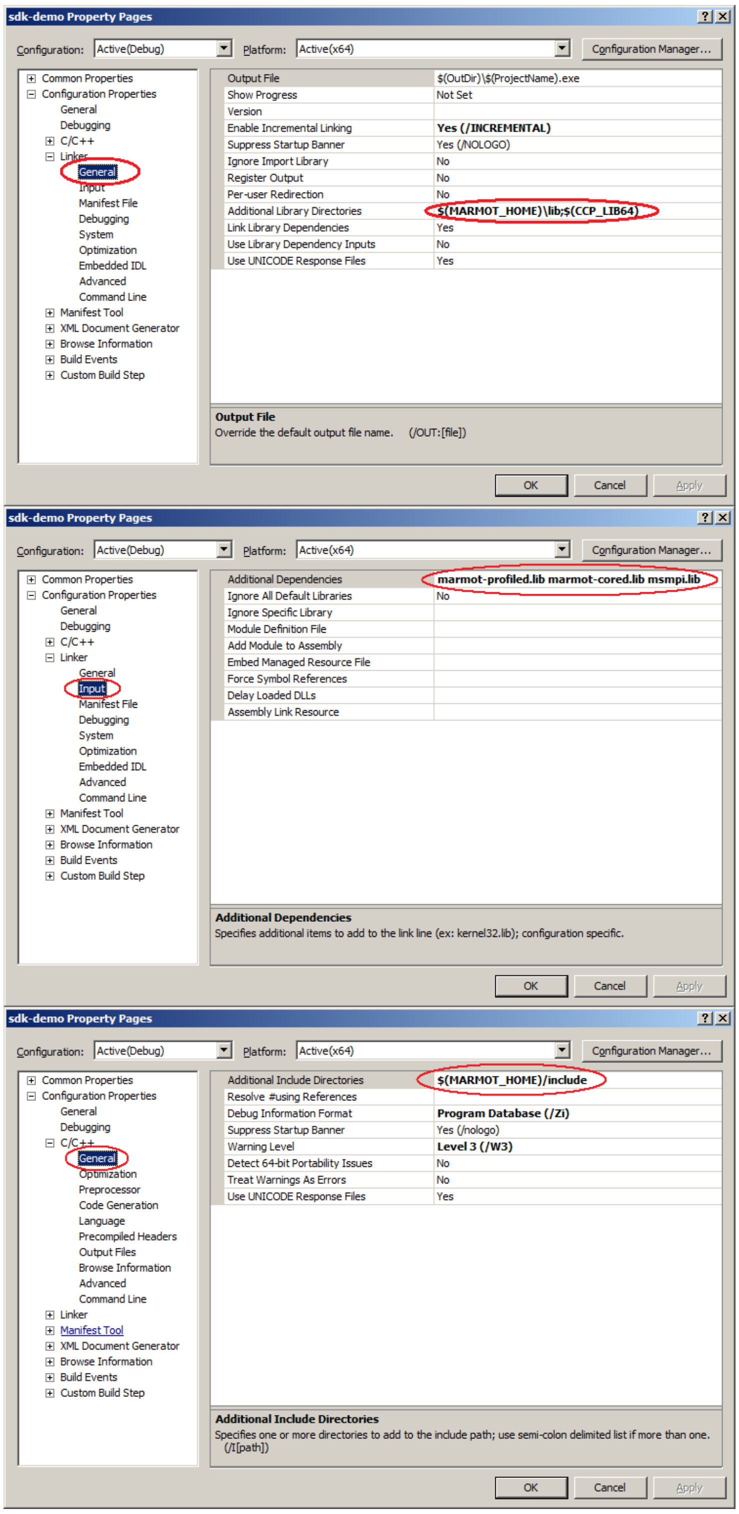
In order to demonstrate the usage of Marmot we introduce a simple erroneous MPI application. The application performs a ring communication where each process sends a message to its successor and receives a message from its predecessor. In order to avoid deadlock (as the standard mode send of MPI might be blocking) an asynchronous send call is used. In addition a user defined data-type is used for the communication. The summarized C code of this application is presented below:

```
 // Initialize MPI
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

// Create a datatype
 MPI_Type_contiguous(2, MPI_INT, &cont2Int);
 MPI_Type_commit(&cont2Int);

 // Send to successor process and receive from predecessor process
 MPI_Isend(sendBuf, 1, cont2Int, (rank+1)%size, MSG_TAG, MPI_COMM_WORLD, &request);
 MPI_Recv(recvBuf, 1, cont2Int, (rank+size-1)%size, MSG_TAG, MPI_COMM_WORLD, &status);

 // Finalize MPI
 MPI_Finalize();
```

This code contains two MPI usage errors. The first one is a missing completion call like **MPI_Wait** for the issued asynchronous communication. The second one is more subtle and very common among many MPI applications and concerns the missing **MPI_Type_free** call for the user defined data-type. In the next sections we present how Marmot is applied to this example application and how to interpret and use the output given by Marmot.

Compilation and Linking

A Visual Studio project for the above example is located in the **examples** subdirectory in the Marmot installation path. In order to use Marmot for this project it is necessary to add certain libraries. This is done in two steps, first by adding the library path to Marmot as an absolute path or set the environment variable MARMOT_HOME to the Marmot installation directory and use "**$(MARMOT_HOME)\lib"**. In the second step add the Marmot libraries to the project. It is important to list them before the MPI library and to use the appropriate version that matches the configuration (debug versions of Marmot libraries have a "D" suffix attached). As a minimum the libraries "marmot-profile.lib" and "marmot-core.lib" have to be added. Finally overwrite the include path specified for the MS-MPI header with the include directory of Marmot. By doing so Marmot gains extra information about the source code locations of the MPI calls. This whole process is shown in Figure 2.



sdk-demo Property Pages

Configuration: Active(Debug) Platform: Active(x64)

Output File	$(OutDir)\$(ProjectName).exe
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**$(MARMOT_HOME)\lib;$(CCP_LIB64)**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Use UNICODE Response Files	Yes

Output File
Override the default output file name. (/OUT:[file])

sdk-demo Property Pages

Configuration: Active(Debug) Platform: Active(x64)

Additional Dependencies	**marmot-profiled.lib marmot-cored.lib msmpi.lib**
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

sdk-demo Property Pages

Configuration: Active(Debug) Platform: Active(x64)

Additional Include Directories	**$(MARMOT_HOME)/include**
Resolve #using References	
Debug Information Format	**Program Database (/Zi)**
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level 3 (/W3)**
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

Additional Include Directories
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

Figure 2: Visual Studio configuration changes necessary to add Marmot to your MPI application

Running with Marmot

Marmot is a library that uses the PMPI profiling interface to intercept MPI calls and analyze them during runtime. 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. 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.

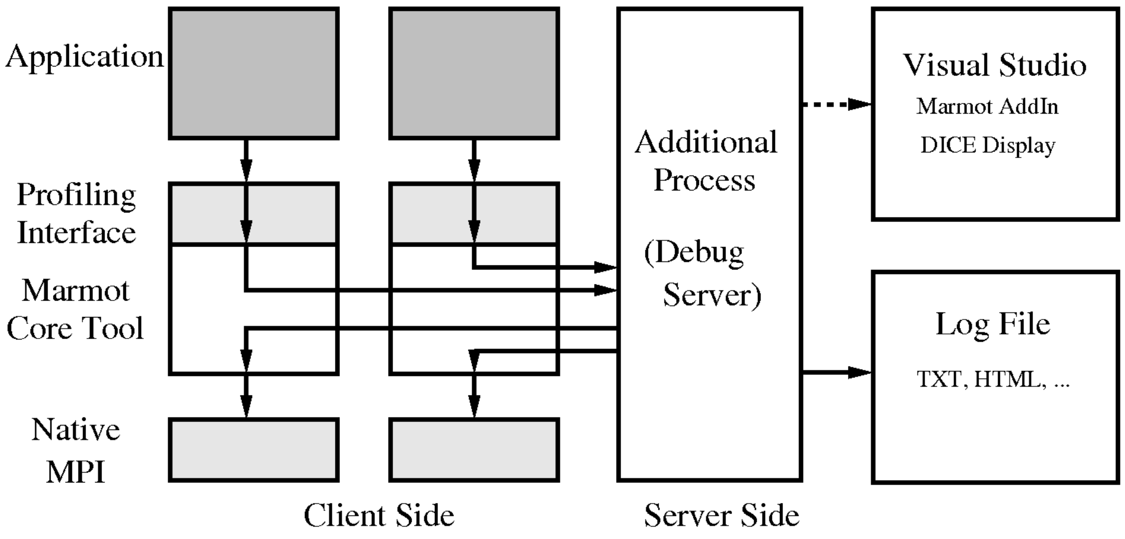


Figure 3: Design of Marmot

Since it can be quite tedious to lookup errors and warnings in log-files an AddIn was developed to better integrate Marmot into Visual Studio. The AddIn launches the application selected as "Startup Project" in Visual Studio and communicates with the debug server built into Marmot. Due to the fact that the AddIn is running in the context of Visual Studio the windows firewall should be set up so that it allows inbound connections from the cluster nodes to "devenv.exe".

Short Recipe

For a quick test of your application with Marmot do the following:

1. Install Marmot
2. Add the libraries and paths mentioned in "Compilation and Linking" to your configuration
3. Compile and link your application
4. Choose the **Startup Project**
5. Click on "Set Command Line" in the Marmot AddIn toolbar
6. Adjust the "Command Line" if needed. (Prepending "cmd /K" for example will keep the command window open. Additional environment variables could be specified right after mpiexec. See the Marmot manual for environment variables used by Marmot). Hit the return key to confirm any changes made.
7. Click on the "Application Launch" button to start your application and check the Marmot output (described later)

Configuration

Marmot's configuration can be adjusted by environment variables. Here are some of the more influential variables:

- MARMOT_HOME: Sets the path of the Marmot installation. If not set the registry is used to determine the installation path. This variable is quite useful to switch between different builds of Marmot.
- MARMOT_LOGFILE_TYPE: Sets the type of the log-file generated by Marmot. "0" enables ASCII logging (default). "1" enables HTML logging.
- MARMOT_LOG_FLUSH_TYPE: "0" flush on error (default). "1" immediate flush.

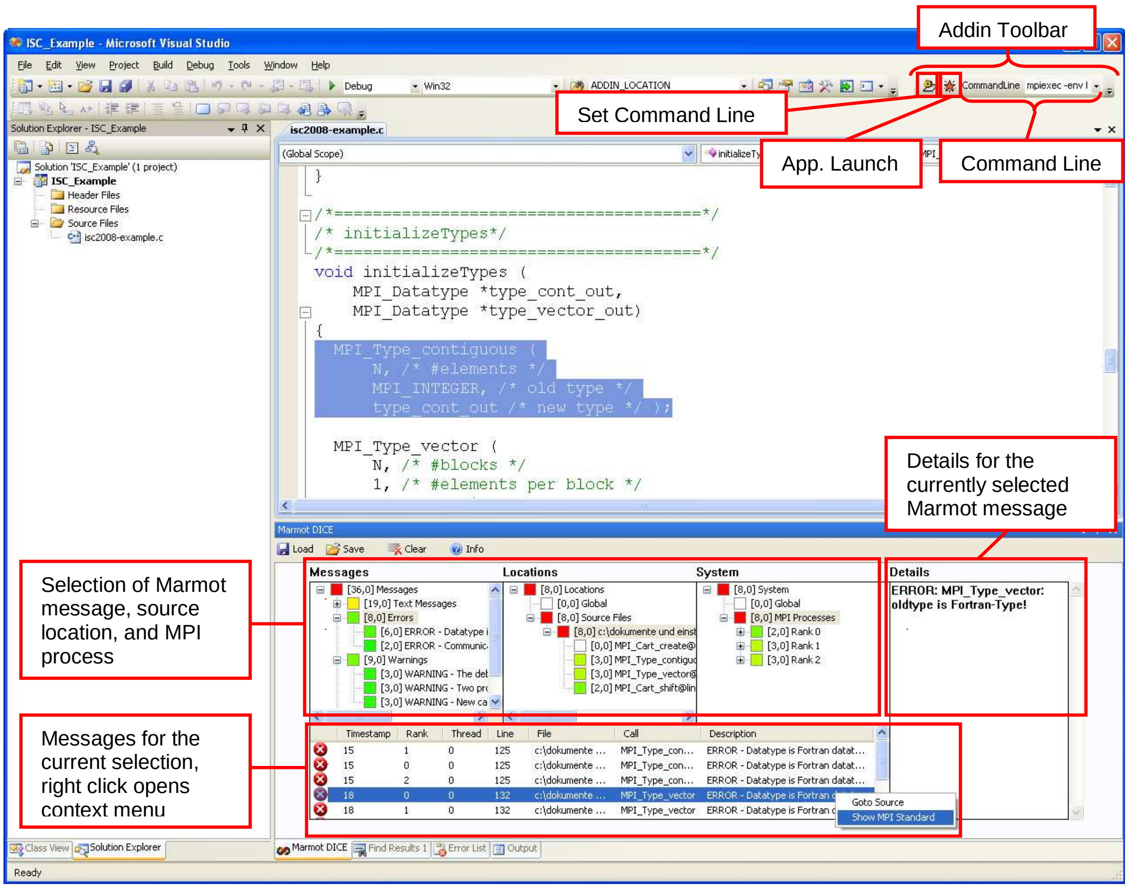
The following environment variables are used in conjunction with the Visual Studio AddIn:

- MARMOT_ADDIN_TIMEOUT: Timeout for the AddIn to wait for an inbound connect of the Debug-Server. Defaults to 5000 milliseconds if not set.
- MARMOT_LOG_HOST: Specifies the computer the AddIn is running on (Name or IP).
- MARMOT_LOG_PORT: Specifies the port used by the AddIn to listen for incoming connects.

Please note that usually the AddIn sets MARMOT_LOG_HOST and MARMOT_LOG_PORT automatically and passes those environment variables to the spawned processes for example with mpiexec's "-env" option.

Marmot Output

The output of Marmot is either stored in an HTML or text based log file or sent to the Microsoft Visual Studio AddIn. Using the AddIn will display Marmot's output in a tool window named "DICE". The DICE window provides an overview of the Marmot output in three windows. The first window summarizes all the messages detected by Marmot, the second window summarizes were these messages occurred, and the third window shows on which processes these messages occurred. By selecting elements in these three windows it is possible to precisely select messages of interest, which are displayed in a message list below the three overview windows. When clicking on one of the messages of the message list window, detailed information is displayed in the rightmost window. Further, by double clicking on a message in the message list the associated source code location is opened in the editor, whereas a right click offers the option to open a reference to the MPI standard. The DICE window and the AddIn are shown in Figure 4.



Addin Toolbar
Set Command Line
App. Launch
Command Line
Details for the currently selected Marmot message
Selection of Marmot message, source location, and MPI process
Messages for the current selection, right click opens context menu
ISC_Example - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Help
Debug
Win32
ADDIN_LOCATION
CommandLine mpiexec -env l
Solution Explorer - ISC_Example
Solution 'ISC_Example' (1 project)
ISC_Example
Header Files
Resource Files
Source Files
isc2008-example.c
(Global Scope)
initializeTy
Marmot DICE
Load
Save
Clear
Info
Goto Source
Show MPI Standard
Class View
Solution Explorer
Marmot DICE
Find Results 1
Error List
Output
Ready

```
}

/*=====================================*/
/* initializeTypes*/
/*=====================================*/
void initializeTypes (
    MPI_Datatype *type_cont_out,
    MPI_Datatype *type_vector_out)
{
  MPI_Type_contiguous (
      N, /* #elements */
      MPI_INTEGER, /* old type */
      type_cont_out /* new type */ );

  MPI_Type_vector (
      N, /* #blocks */
      1, /* #elements per block */
```

Messages
- [36,0] Messages
  - [19,0] Text Messages
  - [8,0] Errors
    - [6,0] ERROR - Datatype i
    - [2,0] ERROR - Communic.
  - [9,0] Warnings
    - [3,0] WARNING - The det
    - [3,0] WARNING - Two pr
    - [3,0] WARNING - New ca

Locations
- [8,0] Locations
  - [0,0] Global
  - [8,0] Source Files
    - [8,0] c:\dokumente und einst
      - [0,0] MPI_Cart_create@
      - [3,0] MPI_Type_contigu
      - [3,0] MPI_Type_vector@
      - [2,0] MPI_Cart_shift@lin

System
- [8,0] System
  - [0,0] Global
  - [8,0] MPI Processes
    - [2,0] Rank 0
    - [3,0] Rank 1
    - [3,0] Rank 2

Details

ERROR: MPI_Type_vector: oldtype is Fortran-Type!

Timestamp	Rank	Thread	Line	File	Call	Description
15	1	0	125	c:\dokumente ...	MPI_Type_con...	ERROR - Datatype is Fortran datat...
15	0	0	125	c:\dokumente ...	MPI_Type_con...	ERROR - Datatype is Fortran datat...
15	2	0	125	c:\dokumente ...	MPI_Type_con...	ERROR - Datatype is Fortran datat...
18	0	0	132	c:\dokumente ...	MPI_Type_vector	ERROR - Datatype is Fortran d
18	1	0	132	c:\dokumente ...	MPI_Type_vector	ERROR - Datatype is Fortran d

Figure 4: Marmot AddIn Toolbar and DICE display

Common MPI Usage Errors

Parallel programming is a complex challenge and offers many pitfalls. Detection of some MPI usage errors is very hard and only possible by combining different tools. Very common programming errors are:

- **Deadlocks:** Marmot contains a mechanism to automatically detect deadlocks and notify the user where and why they have occurred. In general, deadlocks are caused by the non-occurrence of something else, for example mismatched send/receive operations or mismatched collective calls. One can distinguish between *real* deadlocks, which occur inevitably, and *potential* deadlocks, which may occur only under certain circumstances, e.g. depending on data races or on the implementation, for instance, if a standard send is implemented as a buffered send or not. In this code snippet process 0 and process 1 exchange messages between each other.

```
if (rank == 0)
{ // send to 1 and receive from 1
   MPI_Send(...);
   MPI_Recv(...);
}
else if (rank == 1)
{
   // send to 0 and receive from 0
   MPI_Send(...);
   MPI_Recv(...);
}
```

  If the MPI_Send is implemented in buffered mode, for example for small message sizes, this code will not deadlock, otherwise it will. Currently Marmot's deadlock detection is based on a timeout mechanism and therefore finds all real deadlocks. Marmot's debug server surveys the time each process is waiting in an MPI call. If this time exceeds a certain user-defined limit on all processes at the same time, the debug process issues a deadlock warning. The user is then able to trace the last few calls on each node. It is also possible that attaching Marmot (or any other tool) to an application slightly changes the execution flow in such a way that a potential deadlock becomes apparent.

- **Data races:** Potential race conditions can be caused by various reasons, e.g. by the use of a receive call with the wildcard MPI_ANY_SOURCE as source argument or the wildcard MPI_ANY_TAG as tag argument, by the use of random numbers, or by the fact that nodes do not behave exactly the same. Some users also rely on collective calls being synchronizing, however, the only necessarily synchronizing collective call is the MPI_Barrier. Other collective calls can be synchronizing or not, depending on their implementation. For example, assume that any of the send calls on processes 1 and 2 match to any of the receive calls on process 0:

```
if (rank == 0)
{
   MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
   MPI_Bcast(....);
   MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...);
}
else if (rank == 1)
{
   MPI_Send(...);
   MPI_Bcast(...);
}
```

```
else if (rank == 2)
{
   MPI_Bcast(...);
   MPI_Send(...);
}
```

If the MPI_Bcast is synchronizing process 0 will have to receive the message from process 1 first. If it is not then the message order will not be deterministic: either the message from process 1 or from process 2 can be received first. At present, Marmot indicates the use of wildcards, but it does not construct dependency graphs to view the different possible executions nor does it use methods like record and replay to identify and track down bugs in parallel programs or to compare different runs.

- **Mismatches:** Mismatches in arguments of one call can be detected locally and are sometimes even detected by the compiler. Examples are wrong type or number of arguments. Mismatches are also seen in arguments involving more than one call, e.g. in send/receive pairs or in collective calls. Special attention is needed when comparing matched pairs of derived data-types because it is legal to send, for example two (MPI_INT, MPI_DOUBLE) and to receive one (MPI_INT, MPI_DOUBLE, MPI_INT, MPI_DOUBLE), or to send one (MPI_INT, MPI_DOUBLE) and to receive one (MPI_INT, MPI_DOUBLE, MPI_INT, MPI_DOUBLE) (a so-called *partial* receive). MPI implementations usually abort an application when there is a data-type mismatch, e.g. send an MPI_INT and receive an MPI_DOUBLE, but no exact diagnosis of the mismatch is given.

- **Resource handling:** This is an area in MPI where incorrect usage may result in fatal errors with almost no obvious link to the real cause. Since they are very difficult to find, we place special focus into detecting them. Marmot is able to keep track of the proper construction, usage and destruction of all MPI resources, such as communicators, groups, data-types, etc. As these resources are “opaque” objects and therefore implementation-dependent, Marmot has its own book-keeping of these resources and, thus, duplicates the management done by the underlying MPI library. Marmot also checks if requests and other arguments (tags, ranks, etc.) are used correctly, e.g. if an active requests is reused.

- **Memory and other resource exhaustion:** Non-blocking calls such as MPI_Isend etc. can complete without issuing a matching test or wait call. However, the number of available request handles is limited (and implementation defined). Therefore requests should always be freed, as should allocated communicators, data-types, etc. Marmot gives a warning when a request is reused, and also when there are active or non-freed requests left at the MPI_Finalize. Another issue is reusing memory that is still in use, for example by reading/writing from/into a buffer by an unfinished send/receive operation. As Marmot has only information about all the issued MPI calls it is not capable to detect these usage errors, however when buffers already owned by MPI exclusively are passed to MPI in another call an error is given. In some cases, Marmot checks if buffers are overwritten by mistake, e.g. for MPI_Gatherv and similar collective calls, it is verified whether on the root process data is overridden due to an erroneous array of displacements.

- **Portability:** The MPI standard leaves many decisions to the implementors, for example how to implement opaque objects and handles to these objects, if to implement MPI_Send as buffered call or not, if to implement collective calls as synchronizing calls, if to make the implementation thread-safe or not, etc. Some of these issues can already be detected at compile time when the application is ported to another environment, some can be found at runtime by Marmot, e.g. using a tag beyond the guaranteed limit.

Conclusion

Having already proven to be a useful MPI application development tool on many Linux platforms, Marmot has recently been ported to Windows. This paper presents a Microsoft Visual Studio® AddIn that simplifies the usage of Marmot. The AddIn helps to start MPI applications build with Marmot, and automatically makes the Marmot output available in Visual Studio. A variety of common MPI usage errors exists and Marmot helps to detect and solve most of them. The functionality of this tool has been tested successfully with real world applications.

Future work includes technical improvements e.g. a better deadlock detection mechanism or full support for the MPI-2 standard. Another aspect is to improve performance and scalability of the tool, especially for communication-intensive applications. We also aim at constantly improving user-friendliness, e.g. by simplifying the build process with Marmot.

References

[1] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, June 1995. http://www.mpi-forum.org.

[2] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, July 1997. http://www.mpi-forum.org.

[3] Marmot. http://www.hlrs.de/organization/av/amt/research/marmot/

[4] Message Passing Interface (MPI) Tutorial. https://computing.llnl.gov/tutorials/mpi/

[5] Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI I/O Analysis and Error Detection with Marmot. In Recent Advances In Parallel Virtual Machine And Message Passing. 11th European PVM/MPI Users' Group Meeting.LNCS 3241, pages 242 - 250, Springer, 2004.

[6] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. Marmot An MPI analysis and checking tool. In Proceedings of PARCO 2003, pages 493-500, Elsevier, 2004.

[7] Bettina Krammer, Matthias S. Müller and Michael M. Resch. MPI Application Development Using the Analysis Tool Marmot, In Proceedings of ICCS 2004, LNCS 3038, pages 464 - 471, Springer 2004.

[8] Bettina Krammer, Valentin Himmler, David Lecomber. Coupling DDT and Marmot for Debugging of MPI Applications. In Proc. of ParCo 2007, Jülich/Aachen, Germany, September 4-7, 2007. NIC Series, Vol. 38, pp. 653-660

[9] The Cross-Platform Makefile Generator. http://www.cmake.org

Feedback

Did you find problems with this tutorial? Do you have suggestions that would improve it? Send us your feedback or report a bug at marmot-supp@lists.gforge.hlrs.de.

More Information and Downloads

Additional information
http://www.hlrs.de/organization/av/amt/research/marmot/

Download Marmot for Windows
http://www.hlrs.de/organization/av/amt/research/marmot/downloads/