Tool for Analysing and Checking MPI Applications

April 30, 2010

Contents

1 Introduction

1.1 What is Marmot?

- Marmot is a library written in C++, which has to be linked to your application in addition to the existing MPI library.
- It will check whether your application conforms to the MPI standard and will issue warnings if there are errors or non-portable constructs.
- You do not need to modify your source code, you only need one additional process working as Marmot's debug server.
- Marmot's output is a human-readable text file, an HTML file, or uses a format that allows display in other tools, e.g. Cube.
- The tool can be configured via environment variables.

1.2 Design of Marmot

Marmot makes use of the so-called "profiling interface" defined in the MPI standard, i.e. it intercepts the MPI calls from the application for examination before they are passed to the underlying MPI implementation. Marmot maps MPI resources such as communicators, datatypes etc. to its own resources to keep track of proper construction and usage.

1.3 Current Status

- Full support for MPI-1.2
- Partial MPI-2.1 support (The user is notified if your application uses MPI calls that are not checked by Marmot)
- Support for the C/Fortran language bindings of MPI
- Support for hybrid applications that use MPI and OpenMP
- Output available for the Cube [SCALASCA] Visualizer
- Integration into Microsoft Visual Studio available

1.4 Future Plans

- Extended MPI-2 support
- Further integration into IDE's and high performance tools
- Enhanced deadlock detection

2 Installation

2.1 Software requirements

2.1.1 MPI implementation

1. Marmot needs to be built and configured for the MPI implementation that you want to use with your applications. Marmot verifies the calls made by the program with the use of the so called profiling interface (PMPI). This profiling interface is part of the MPI standard. Any MPI implementation that conforms to the MPI standard needs to provide this interface. Therefore, this requirement should not limit the selection of possible MPI libraries.

2. Marmot needs to detect the MPI implementation, i.e. its mpi.h and its libraries

2.1.2 CMake

Marmot uses CMake as its build system, in order to build Marmot from source you will need an installed CMake version. We strongly suggest to use a recent version of CMake, to simplify the build process. For linux systems, if you have no administrator priviliges, you can still install CMake into your home directory.

2.1.3 OpenMP

Marmot supports hybrid programs using MPI and OpenMP. In order to do so, Marmot has to use an internal synchronisation, thus enabling full MPI_THREAD_MULTIPLE support. This synchronisation uses OpenMP directives which also allows Marmot to gather additional information about the threads being used. See Section 2.3 on how to build Marmot with OpenMP support. Your compiler must, of course, support OpenMP as well.

2.1.4 C++ compiler

1. Marmot is implemented in C++. The compiler should implement the ISO/IEC 14882 language specification of C++. We have succeeded in using gcc 2.96 or later, earlier versions might not work properly. Intel Compilers are an alternative, they are available for no cost for non-commercial use on linux platforms. For example, on our local environment, Intel compilers version 10.0 have been used successfully. Further, IBM xlc++ compilers have been used successfully too.

2. To link Marmot to a C or Fortran application with the C/Fortran linker instead of the C++ linker, some C++ libraries will have to be linked, too (for example libstdc++, using gcc or g77). Marmot tries to determine these libraries, thus you are strongly encouraged to link your MPI applications with the Marmot compiler wrappers (see Section 3.1.1).

2.1.5 Fortran Compiler

To support the Fortran binding of the MPI standard a Fortran compiler is required.

2.1.6 C compiler

To support C applications, a C compiler is required (any C compiler should do).

2.1.7 Other tools that users need

A version of *make* (tested with GNU make 3.79.1 and later versions) for compilation. Also, some of the Marmot helper tools, e.g. compiler wrappers, need the *bash* shell and an *awk* interpreter (UNIX only).

2.1.8 Other tools that users do not necessarily need[1]

1. Doxygen [DOXYGEN] (tested with version 1.2.14 and later versions) is used to automatically generate documentation. This documentation is supposed to provide sufficient information for developers even if they do not have Doxygen.

2. Scalasca [SCALASCA] or a stand alone installation of Cube, if you want to use Cube logging (see Section 3.3)

2.2 Hardware requirements

Marmot does not require any specific hardware (any UNIX or Windows environment should do). It has been tested on the following platforms:

- LINUX IA32/IA64 clusters
- SGI Altix 4700
- Opteron clusters
- SUN clusters
- NEC SX6, SX8
- IBM Power6 systems with AIX
- Windows Server 2003 & 2008 cluster

[1] Marmot's distribution comes with all the required files, the users just have to run "cmake" and "make"

2.3 Configuring and Building

2.3.1 Basic Steps

Marmot uses CMake for its configuration, as a first step you run the cmake tool to create platform specific make files or project files. CMake provides a command line tool and also graphical user interfaces for an easy configuration process. Basically you just have to create a folder for example inside the Marmot trunk, e.g. "MyMarmotBuild", change into this folder and issue the command `$ccmake ..`, `$cmake-gui`, or `$cmake` (You should prefer ccmake/cmake-gui over cmake).

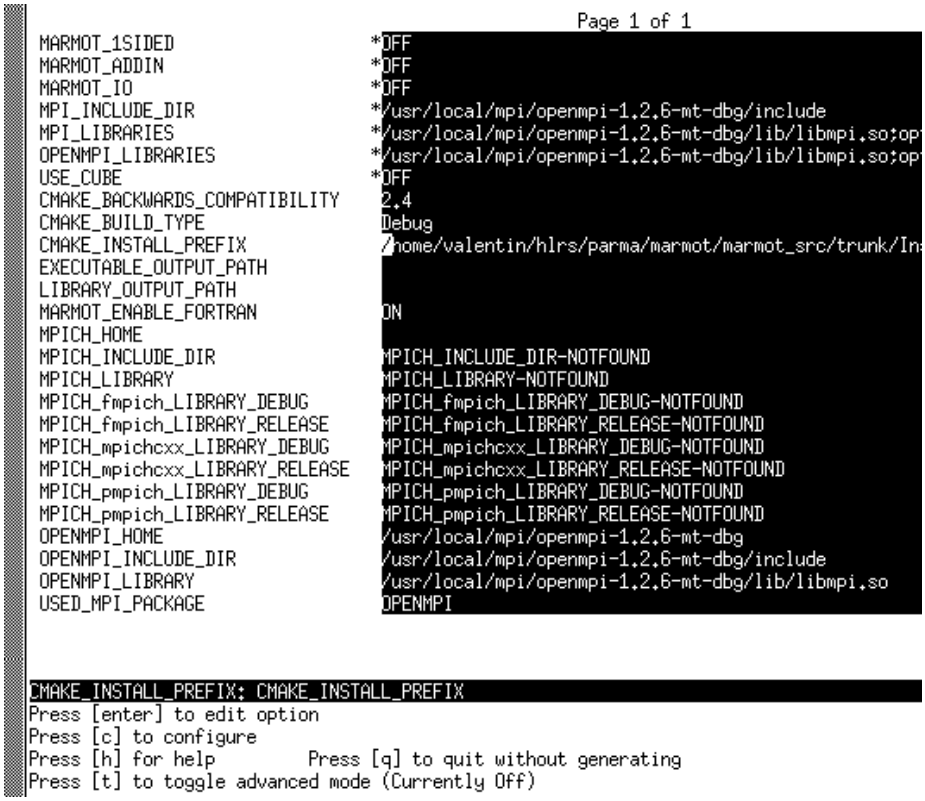

Figure 1: Ncurses display of CMake

You can then adapt the configuration to your needs. Some of the important CMake configuration options for Marmot are:

`CMAKE_INSTALL_PREFIX` determines the installation path

`MARMOT_USED_MPI_PACKAGE` determines which MPI find module is used. You may choose: `MPICH`, `OPENMPI` or `MPI`. MPI works for almost all MPI implementation and is a common choice.

`MARMOT_ADDIN` when switched to ON the AddIn for VisualStudio is compiled (Windows specific)

`MARMOT_ENABLE_FORTRAN` specifies whether Marmot is built with Fortran support (enabled per default on Linux)

`MARMOT_USE_CUBE` specifies whether Marmot is built with Cube support

In CMake frontends such as ccmake or cmake-gui you may switch to an advanced mode where you can alter all the cmake-variables relevant to the configuration process. Changing the compilers used for your installation is not possible after you have started the CMake configuration. You should set the `CC`, `CXX`, `F77`, and `FC` environment variables before the first call of ccmake.

After the configuration step with CMake you will have platform specific make files, e.g. in Linux these may be regular make files, in Windows these may be a VisualStudio project. Build these files, e.g., in Linux run `make && make install`, in Windows open the project right-click the INSTALL target and build the solution. This will build and install Marmot.

2.3.2 Installation with OpenMP support

A Marmot installation with OpenMP support can only be used for applications that are compiled and linked with OpenMP flags. As this may have a performance impact for non-OpenMP applications, we advice to create an extra installation of Marmot that is configured with OpenMP support. Many MPI implementations use a different MPI library for hybrid MPI/OpenMP applications that use MPI_THREAD_MULTIPLE. For Marmot you should specify this library during configuration with cmake, i.e. by setting the MPI library to `-lmpi_mt` (assuming your multithreaded MPI library is named "libmpi_mt"). Also do not forget to enable the `MARMOT_USE_OPENMP` option in cmake.

2.3.3 Known Issues

On some systems, or for some MPI implementations, it may happen that issues during the installation or ussage of Marmot arise. The following list mentiones all currently known issues and their workaroundlable:

- IBM compilers on AIX (Power6):
  - C++ linking: A default installation of Marmot links C++ STL libraries to your C and Fortran applications, on AIX systems with IBM compilers this may lead to deadlocks when running your applications. Marmot's compiler wrappers support a special linking step for C and Fortran applications were the C++ compiler is used instead of the native compiler. Set the cmake variable "MARMOT_STDCXX_LIBS" to something invalid – e.g. "BREAK!" – to enforce this behavior.

- 64bit builds: CMake supports no ARFLAGS, which are needed to build 64 bit Marmot libraries. This issue can be overcome by setting the environment variable "OBJECT_MODE" to "64" before runing cmake/cmake-gui/ccmake for the first time.
- All in all, a Marmot build on AIX with IBM compilers and POE may look like (assumed no mpicc/.. of some other MPI is in the path):

```
cd <Marmot-Source>
mkdir BUILD-64 && cd BUILD-64
export OBJECT_MODE=64
CC=xlc CXX=xlc++ FC=xlf F77=xlf F90=xlf90 \
cmake ../ \
     -DCMAKE_INSTALL_PREFIX=<INSTALL-PATH> \
     -DMARMOT_STDCXX_LIBS=BREAK!
```

If you encounter any other issues, you are welcome to contact *marmot-supp@gforge.hlrs.de* for help.

2.4 DDT Plugin

Marmot supports Allinea's parallel debugger DDT with a plugin. This plugin makes it possible to run a debugging session with DDT and at the same time to perform Marmot's checks.

2.4.1 Prerequisites

To use Marmot's DDT plugin you need

- DDT from Allinea [ALLINEA], along with a valid Licence, version 2.3.1 or above,
- An MPI installation with shared libraries,
- Marmot with shared libraries and configured to support Cube

2.4.2 Installing DDT

Installation instructions for DDT can be found in the DDT manual. After installing DDT you will find the following folder and files in the DDT folder:

```
$ls
```

```
bin doc examples help icons lib Licence plugins script templates wizard
```

To use plugins you need to create a plugin folder. Change into the DDT folder and issue

```
$mkdir plugins
```

2.4.3 Installing the Marmot Plugin

Installing the Marmot plugin is as easy as copying the appropriate XML file into DDT's plugin folder. An XML file is generated and placed in Marmot's installation directory in share/marmot/examples/marmot_ddt_plugin.xml

```
$cp marmot_ddt_plugin.xml <DDT_DIR>/plugins/
```

Please note that you have to set the environment variable `MARMOT_LOGFILE_TYPE` to 3 for the plugin to work properly. Now you may startup DDT to see whether the Marmot plugin has been recognized. Go to *Session* → *New Session*→ *Run...* and switch to the *Advanced* view. You should now see the Marmot plugin:

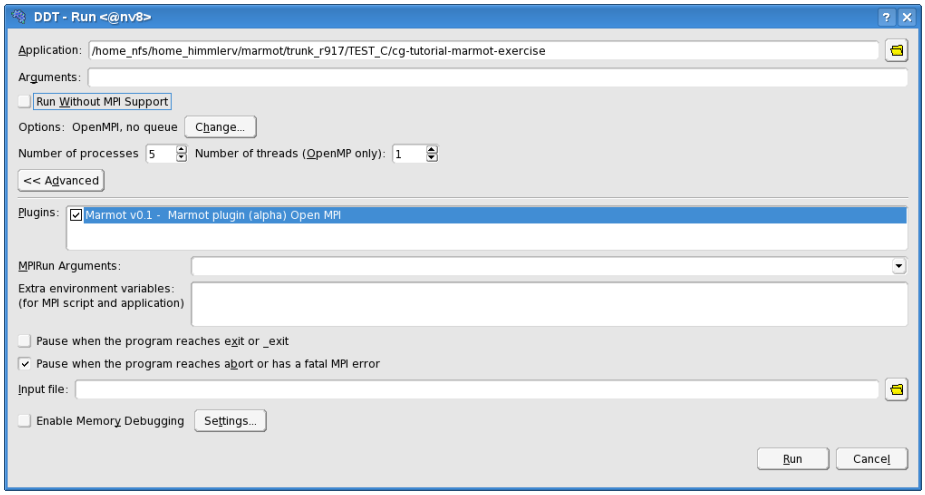

Figure 2: DDT plugin selection

3 Usage

3.1 Compilation

3.1.1 C and C++ programs

To compile a C/C++ application with Marmot, you can use *marmotcc* and *marmotcxx* which are wrapper scripts invoking the C or C++ MPI compilers of the underlying MPI implementation (Or native compilers with extra libraries, if no MPI compilers are available). Compilation of a C/C++ application should be as easy as typing

`$marmotcc -o basic basic.c` or `$marmotcxx -o basic basic.cc`

However, in rare situation this may lead to problems, e.g. in the linking step. In this case, you may have to modify *marmotcc*/*marmotcxx* for your needs (Or ask an administrator or Marmot developer for help). To see what exactly *marmotcc*/*marmotcxx* does, you can type

```
$marmotcc --marmot-verbose -o basic basic.c
```

and should see something like

```
mpicc -I/usr/local/marmot/include -o basic basic.c -L/usr/local/marmot/lib
-lmarmot-profile -lmarmot-core -L/usr/local/mpi/openmpi-1.2.8/lib -lmpi
-lpthread -L/usr/lib -lxml2 -L/usr/local/cube/lib -lcube3 -L/usr/lib
-lstdc++
```

If you attempt to modify the linking step or change the library order, make sure you link the libraries in the correct order, i.e. the Marmot libraries have to be linked prior to the MPI libraries. Otherwise Marmot simply won't work, note that you won't get an error message during the linking step due to a wrong order.

Further, the compiler wrappers try to redirect the MPI header. If successful, your application will use Marmot's provided MPI header, which in turn includes the MPI implementation provided header. As a result, Marmot can add source code information to MPI calls, thus providing more detailed output.

To get basic usage hints just type *marmotcc*/*marmotcxx* without any arguments.

3.1.2 Fortran programs

Compiling fortran programs basically works the same way as compiling C programs. One can use *marmotf77*/*marmotf90*:

```
$marmotf77 -o basic basic.f
```

If you attempt to modify the linking step or change the library order, be sure to link the Marmot libs prior to the MPI libs.

The Marmot compiler wrappers for Fortran attempt a source to source translation, which adds additional source code data to the MPI calls. For some Fortran applications this may fail, use the extra argument `--marmot-noinst` in these cases. You can also run *marmotf77*/*marmotf90* without any arguments to see all of their options.

3.1.3 Hybrid programs

A Marmot installation configured with OpenMP support works like a normal installation. You can use*marmotcc* (*marmotf77* resp.) to compile and link your applications. The compiler wrappers will automatically include the necessary MPI library and the OpenMP flag, include paths and libraries.

3.2 Running the application

3.2.1 Invocation

To run the application, one has to add an additional process working as debug server, i.e. one needs $(n + 1)$ instead of n processes:

```
$mpirun -np (n+1) foo
```

Marmot's output is written to a logfile (see Section 3.3).

Note on hybrid programs:
Running a program compiled with an OpenMP supporting version of Marmot works as usual. But you might have to set additional environmental variables in order to enable MPI/OpenMP interoperability on your system.

3.2.2 Influential environment variables

The following environment variables affect Marmot's behaviour at runtime:

`MARMOT_DEBUG_MODE`	0: errors, 1: errors and warnings, 2: errors, warnings and remarks are reported (default)
`MARMOT_LOGFILE_TYPE`	0: ASCII Logging (default), 1: HTML Logging, 2: CUBE Logging (when enabled via configure),
`MARMOT_LOG_FILTER_COUNT`	Limits how often a sepecific problem is recoreded (default: 2)
`MARMOT_LOG_FLUSH_TYPE`	0: Flush on Error (default), 1: Immediate Flush
`MARMOT_SERIALIZE`	0: code is not serialized, 1: code is serialized (default)

MARMOT_TRACE_CALLS	1: calls are traced with output to stderr, traceback in case of a deadlock is possible (default), 0: calls are traced without output to stderr, traceback in case of a deadlock is possible, -1: calls are not traced, traceback in case of a deadlock is NOT possible.
MARMOT_MAX_PEND_COUNT	maximum number of calls that are traced back (default 10)
MARMOT_MAX_TIMEOUT_DEADLOCK	maximum message time (default 10^6 μs)
MARMOT_MAX_TIMEOUT_SERIALIZE	maximum message time (default 10^3 μs)

3.3 Marmot's output

Marmot's output can be set to one of currently three modes. *ASCII* logging, *HTML* logging and *Cube* logging. The mode is set via the environment variable *MARMOT_LOGFILE_TYPE* (see Section 3.2.2). In the following, the different modes are compared by having a look at the output of a small program named *deadlock1.c* in the directory *TEST_C*. The source code can be seen in Figure 3.

As the name suggests, this program deadlocks. So let's have a look at Marmot's output, depending on the logging mode.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    const int COUNT = 1;
    const int MSG_TAG_1 = 17;
    const int MSG_TAG_2 = 18;
    int rank = -1;
    int size = -1;
    int dummy = 0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf(" I am rank %d of %d PEs\n", rank, size);
    if (size < 2)
    {
      fprintf(stderr, " This program needs at least 2 PEs!\n");
    }
    else
    {
      if (rank == 0)
      {
         MPI_Recv(&dummy, COUNT, MPI_INT, 1, MSG_TAG_1, \\
         MPI_COMM_WORLD, &status);
         MPI_Send(&dummy, COUNT, MPI_INT, 1, MSG_TAG_2, \\
         MPI_COMM_WORLD);
      }
      if (rank == 1)
      {
         MPI_Recv(&dummy, COUNT, MPI_INT, 0, MSG_TAG_2, \\
         MPI_COMM_WORLD, &status);
         MPI_Send(&dummy, COUNT, MPI_INT, 0, MSG_TAG_1, \\
         MPI_COMM_WORLD);
      }
    }
    MPI_Finalize();
    return 0;
}
```

Figure 3: deadlock1.c

3.3.1 ASCII logging

```
1: Note from rank 0 with  Text: performing
On Call: MPI_Init

2: Note from rank 1 with  Text: performing
On Call: MPI_Init

3: Note from rank 0 with  Text: performing
On Call: MPI_Comm_rank

4: Note from rank 1 with  Text: performing
On Call: MPI_Comm_rank

5: Note from rank 0 with  Text: performing
On Call: MPI_Comm_size

6: Note from rank 1 with  Text: performing
On Call: MPI_Comm_size

7: Note from rank 0 with  Text: performing
On Call: MPI_Recv

8: Note from rank 1 with  Text: performing
On Call: MPI_Recv

8: Error global message with  Text: WARNING: all clients are pending!
Last calls (max. 10) on node 0:
 timestamp 1: MPI_Init(*argc, ***argv)

 timestamp 3: MPI_Comm_rank(comm = MPI_COMM_WORLD, *rank)

 timestamp 5: MPI_Comm_size(comm = MPI_COMM_WORLD, *size)

 timestamp 7: MPI_Recv(*buf, count = 1, datatype = MPI_INT, source = 1, tag = 17, comm = MPI_COMM_WORLD, *status)


Last calls (max. 10) on node 1:
 timestamp 2: MPI_Init(*argc, ***argv)

 timestamp 4: MPI_Comm_rank(comm = MPI_COMM_WORLD, *rank)

 timestamp 6: MPI_Comm_size(comm = MPI_COMM_WORLD, *size)

 timestamp 8: MPI_Recv(*buf, count = 1, datatype = MPI_INT, source = 0, tag = 18, comm = MPI_COMM_WORLD, *status)
```

Figure 4: ASCII logging (excerpt)

The output file in ASCII logging mode is named *Marmot_EXE_YYYYMMDD_hhmmss.txt*, where EXE denotes the name of your executable, and YYYYMMDD_hhmmss is a timestamp.

3.3.2 HTML logging

38	2	0	**Error**	Text: ERROR: MPI_Type_struct: datatype [0] is Fortran-Type! Call: MPI_Type_struct	cg-tutorial-marmot-exercise.c line: 230	Infos see MPI-Standard
38	4	0	**Error**	Text: ERROR: MPI_Type_struct: datatype [0] is Fortran-Type! Call: MPI_Type_struct	cg-tutorial-marmot-exercise.c line: 230	Infos see MPI-Standard
40	4	0	**Warning**	Text: WARNING: MPI_Type_struct: datatype [1] is only optional! Call: MPI_Type_struct	cg-tutorial-marmot-exercise.c line: 230	Infos see MPI-Standard
42	1	0	**Warning**	Text: WARNING: MPI_Type_struct: blocklength[0] = 0! Call: MPI_Type_struct	cg-tutorial-marmot-exercise.c line: 230	Infos see MPI-Standard
42	2	0	**Warning**	Text: WARNING: MPI_Type_struct: datatype [1] is only optional! Call: MPI_Type_struct	cg-tutorial-marmot-exercise.c line: 230	Infos see MPI-Standard
44	1	0	**Error**	Text: ERROR: MPI_Type_struct: datatype [0] is Fortran-Type! Call: MPI_Type_struct	cg-tutorial-marmot-exercise.c line: 230	Infos see MPI-Standard
47	3	0	**Note**	Text: NOTE: MPI_Type_commit: Datatype already committed! Argument: datatype Information for Resource of type MPI_Datatype: created at cg-tutorial-marmot-exercise.c line: 239 commited at cg-tutorial-marmot-exercise.c line: 242 not yet freed. Call: MPI_Type_commit	cg-tutorial-marmot-exercise.c line: 243	Infos see MPI-Standard
50	5	0	**Note**	Text: NOTE: MPI_Type_commit: Datatype already committed! Argument: datatype Information for Resource of type MPI_Datatype: created at cg-tutorial-marmot-exercise.c line: 239 commited at cg-tutorial-marmot-exercise.c line: 242 not yet freed. Call: MPI_Type_commit	cg-tutorial-marmot-exercise.c line: 243	Infos see MPI-Standard

Figure 5: HTML logging (excerpt)

The output file in HTML logging mode is named *MarmotLog_EXE_YYYYMMDD_hhmmss.html*.

3.3.3 CUBE logging

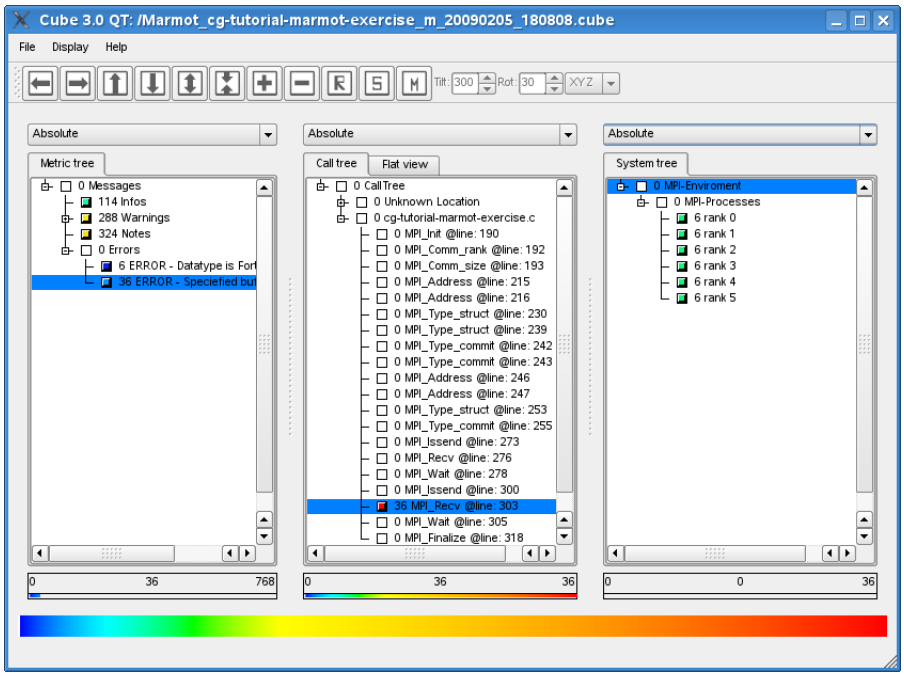

Figure 6: CUBE logging

The output file in CUBE logging mode is named *MarmotLog_EXE_YYYYMMDD_hhmmss.cube*. Further a folder named *MARMOT_HTML* is created, which contains detailed information. To view the log file with the Cube browser, type

`$cube MarmotLog_EXE_YYYYMMDD_hhmmss.cube` or `$cube3 MarmotLog_EXE_YYYYMMDD_hhmmss.cube` for the most recent version of the Cube browser.

3.3.4 Running DDT with Marmot's plugin

Install the Marmot plugin as described in section 2.4. Compile your program with the original MPI compiler wrappers (e.g. *mpicc*) and don't forget to include debug information in the executable (usually with *-g*):

```
$mpicc -g -o yourprogram yourprogram.c
```

Launch DDT and go to *Session* → *New Session* → *Run ...*. In the advanced view, select the Marmot plugin:

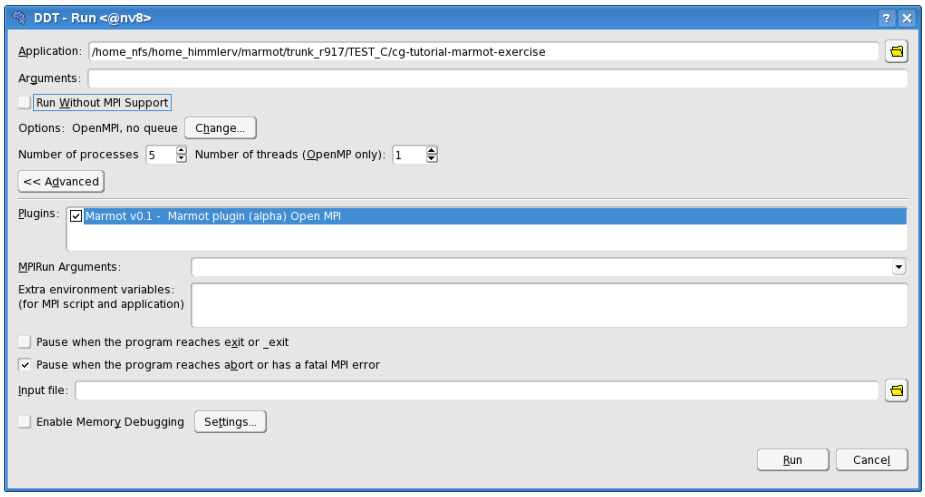

Figure 7: DDT plugin selection

Run the application with the original number of processes. DDT will automatically add one process for Marmot's debug server, which is not displayed. When Marmot detects an error DDT will pause the execution and pop up a window:

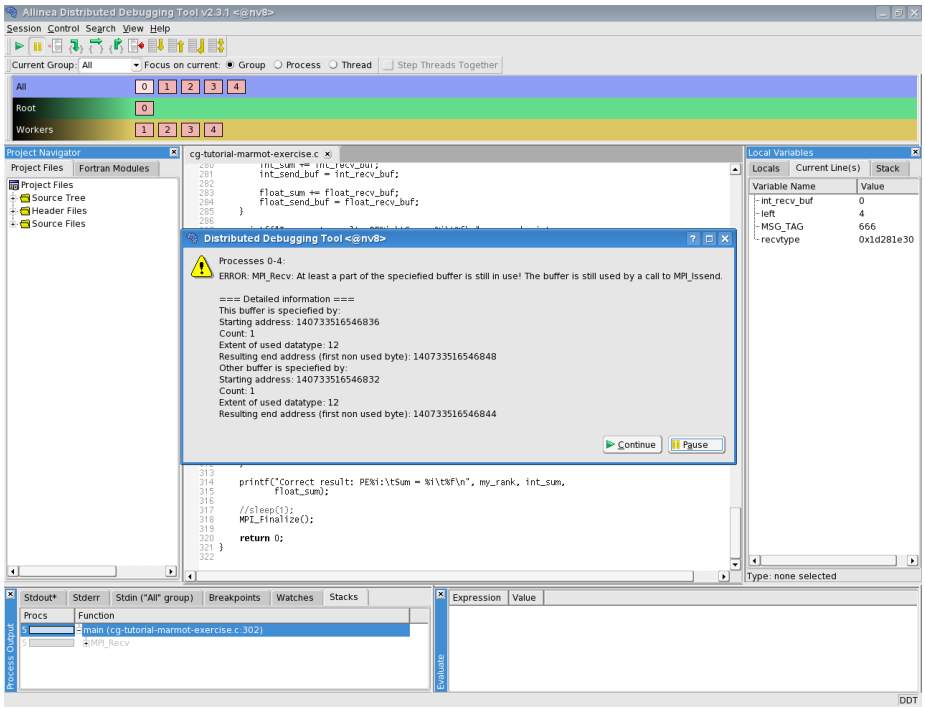

Figure 8: Marmot detects error from within DDT

When run from within DDT, Marmot still creates its logfile for later analysis.

References

[SCALASCA] Scalable Performance Analysis of Large-Scale Applications, FZ Jülich, available at http://www.fz-juelich.de/jsc/scalasca/

[DOXYGEN] Doxygen, http://www.doxygen.org/

[CMAKE] CMake, http://www.cmake.org

[ALLINEA] Allinea Software, http://www.allinea.com