DELIVERABLE D2.3

PART II: PROTOTYPE DOCUMENTATION FOR MPI CODE DEBUGGING AND VERIFICATION

Task 2.2 MPI Code Debugging and Verification

Document Filename:	**CG2.2-D2.3-v2.0-UST008-PrototypeDoc.doc**
Work package:	**WP2 Grid Application Programming Environment**
Partner(s):	**USTUTT, CSIC**
Lead Partner:	**USTUTT**
Config ID:	**CG2.2-D2.3-v2.0-UST008-PrototypeDoc**
Document classification:	**PUBLIC**

Abstract: This document specifies the first prototype for CrossGrid Task 2.2 'MPI code debugging and verification'. This tool aims at verifying the correctness of parallel, distributed grid applications using the MPI paradigm.

Delivery Slip

	Name	Partner	Date	Signature
From	Bettina Krammer, Matthias Müller	USTUTT	16/01/2003	
Verified by	Celso Martínez Rivero, Jesus Marco, Isidro Gonzalez Caballero	CSIC	07/02/2003	
Approved by				

Document Log

Version	Date	Summary of changes	Author
1.0	16/01/2002	Draft version	Bettina Krammer, Matthias Mueller
1.1	17/02/2003	Final version Supplemented user manual and added information on distribution.	Bettina Krammer, Matthias Müller
2.0	08/05/2003	Modified version after EU-review, including work done during project month 12-14. Extended executive summary, comparison with related work. Summary of current status added with usage summary table, references updated, structure of code directory explained, state of the art added, contribution to grid technology added, Marmot functionality added, installation and configuration updated, scheme for makefiles for Fortran added, user support section added, description of tests with applications added, known bugs added.	Bettina Krammer, Matthias Müller

CONTENTS

List of Figures

List of Tables

EXECUTIVE SUMMARY

DOCUMENT STRUCTURE:

This document describes the first prototype of the MPI debugging and verification tool (MARMOT), which aims at verifying the correctness of parallel, distributed grid applications using the MPI paradigm.

Section 1 specifies the purpose of this tool and lists definitions, abbreviations and acronyms used throughout this document.

Section 2 contains references concerning the source code and provides information on how to contact the implementors.

Section 3 describes the status of the implementation. We assume that the MPI standards, that the Software Requirements Specification (CG-2.2-DOC-0003-1-0-FINAL-C) and that the Design Document (CG2.2-D2.2-v1.1-UST001-VerifDesign) are well known.

Section 4 shows the functionality of the prototype in comparison to the current state of the art.

Section 5 is intended to be a short user's manual, enabling the user to install the verification tool and to use it.

Section 6 explains how the verification tool has been tested so far.

Section 7 contains further issues affecting the functioning of the prototype, e.g. known bugs.

SUMMARY OF THE CURRENT STATUS:

According to the technical annex the milestone at project month 12 should "contain the subset of MPI calls required by the end user applications and is running on a local environment". This includes 23 different calls of the MPI interface for the C and Fortran language binding. This objective has been met. In addition to simple tests the functionality has been demonstrated with one end user application with C language binding (Task 1.3 HEP) and one with Fortran binding (Task 1.4.3 Meteo).

The following table summarizes the interaction and status with different applications and kernels of CrossGrid. Three stages of usage are distinguished: "applicable to" means that the tool can be applied to the program developed in that WP, "tested with" means that the tool was tested with the program of that task by MARMOT developers, "used by" means that MARMOT was used by the application developer of that WP and applied to the developed application of that WP.

	Applicable to	Tested with	Used by
WP 1.1 BioMed	**Yes**	No[1]	No? [2]
WP 1.2 Flooding	**Yes**	No[1]	No? [2]
WP 1.3 HEP	**Yes**	**Yes**	**Yes**
WP 1.4 METEO	**Yes**	**Yes**	**Yes**
WP 2.3 benchmarks	**Yes**	**Yes** [3]	No [3]

table 1 Usage Summary

[1] Application not available in cvs

[2] No user feedback has been provided yet

[3] The focus of the first prototype was on functionality and availability for the applications of WP1. Performance tuning and testing with the benchmarks is a target for the next version.

1. INTRODUCTION

1.1. PURPOSE

This document specifies the first prototype of the MPI code debugging and verification tool, within Task 2.2 'MPI code debugging and verification'. The intended audience is both the Task itself and dependent tasks.

The products of Task 2.2 are:

- A portable tool for verifying the standard conformance of an MPI program. This tool is a library that needs to be linked to the application, no modification of the application source code is required at all.
- Some simple programs to test the tool. A wide range of correct and incorrect test programs involving various MPI calls is provided. Thus, users can check whether and how the verification tool works after installation.

Details on how to use this tool will follow below.

1.2. DEFINITIONS, ABBREVIATIONS, ACRONYMS

MPI Message Passing Interface (http://www-unix.mcs.anl.gov/mpi/)

API Application Program Interface

SRS Software requirements specification

HEP High Energy Physics

Throughout this document the terms "MPI code debugging and verification tool" , "MPI verification tool" and "MPI code verification tool" are used as synonyms. This tool is also referred to as MARMOT 1.0 or simply MARMOT.

2. REFERENCES

MPI Standards: http://www.mpi-forum.org/docs/docs.html
Version 1.1: document from June 12, 1995.
Version 1.2: described in chapter 3 of the MPI 2.0 standard.
Version 2.0: document from July 18, 1997.

CrossGrid Documents:

SRS Document: CrossGrid Deliverable D2.1, Task 2.2 Software Requirements Specification. June 2002. CG-2.2-DOC-0003-1-0-FINAL-C

Design Document: CrossGrid Deliverable D2.2, Task 2.2 Design Document. September 2002. CG2.2-D2.2-v1.1-UST001-VerifDesign

Prototype Document: CrossGrid Deliverable D2.3, Task 2.2 Prototype Documentation. February 2003. CG2.2-D2.3-v1.1-UST006-PrototypeDoc

Report Document: CrossGrid Deliverable D2.4, Task 2.2 Internal Progress Report on Software Evaluation and Testing. April 2003. CG2.2-D2.4-v1.2-UST006-ReportDoc

Technical Annex: CrossGrid Project Technical Annex 1
http://www.eu-crossgrid.org/CrossGridAnnex1_v3.1.doc

Existing Debugging Tools for MPI Applications:

Totalview: http://www.etnus.com/Products/TotalView/

Umpire: Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In SC2000: High Performance Networking and Computing Conf. ACM/IEEE, 2000.

P2d2: http://www.nas.nasa.gov/Groups/Tools/Projects/P2D2/
D. Cheng and R. Hood. A Portable Debugger for Parallel and Distributed Programs. In Proc. Supercomputing'94, pp 723-732, Washington D.C., Nov. 1994.

MPI-CHECK: http://www.hpc.iastate.edu/MPI-CHECK.htm
Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, Yan Zou. MPI-CHECK: a Tool for Checking Fortran 90 MPI Programs. Concurrency and Computation: Practice and Experience. 2003, vol. 15, pp 93-100.

DDT: The Distributed Debugging Tool
http://www.streamline-computing.com/softwaredivision_1.shtml

mpigdb: http://www-unix.mcs.anl.gov/mpi/mpich/docs/userguide/node26.htm#Node29

Other Software:

ISO/IEC 14882: 1998(E) Programming languages - C++

GCC: the GNU Compiler Collection
http://www.gnu.org/software/gcc/gcc.html

KCC: the kcc compiler

http://developer.intel.com/software/products/kcc/

ICC: the Intel compiler
http://www.intel.com/software/products/compilers/clin/

GDB: The GNU Project debugger,
http://www.gnu.org/manual/gdb/

MPICH: Portable implementation of MPI
http://www-unix.mcs.anl.gov/mpi/mpich/

MPICH-G, MPICH-G2: Grid-enabled implementation of the MPI v1.1 standard
http://www3.niu.edu/mpi/

PACX-MPI: PArallel Computer eXtension: The Grid-Computing library Extending MPI for Computational Grids
http://www.hlrs.de/organization/pds/projects/pacx-mpi/
Edgar Gabriel, Michael Resch, Thomas Beisel, Rainer Keller. Distributed Computing in a heterogenous computing environment. In Vassil Alexandrov, Jack Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, pp. 180-188, Springer, 1998.

Doxygen: http://www.doxygen.org/

2.1. SOURCE CODE

The MPI code verification tool is now also referred to as MARMOT.

The source code is available from the FZK CVS-repository (http://gridportal.fzk.de/), under crossgrid/wp2/wp2_2-verif/src/MARMOT. It is recommended to have a look at the README file before using MARMOT.

- The README file contains information on how to install and configure MARMOT and on how to run applications with it. This information is basically the same as described in this prototype documentation, however, the README file is regularly updated to keep pace with the ongoing development of MARMOT.
- The KnownBugs file contains further information on known bugs, which may also be interesting for users.

MARMOT contains several subdirectories:

- DOC/AUTO contains documentation automatically generated by Doxygen when compiling MARMOT. As described in the Design Document, MARMOT is a library written in C++ and contains:
  - 128 classes representing MPI calls
  - several base classes
  - several classes representing MPI Datatypes etc.
  - miscellaneous classes representing the debug server etc.

  The Doxygen tool is therefore a convenient tool to browse through the respective class hierarchy and to examine specific classes or methods more closely (open DOC/AUTO/html/index.html with your browser). However, a really detailed description of all these classes and their functionality that is comparable e.g. to the way MPI calls are described in the MPI standards cannot be provided yet.

In general, the information provided in the MARMOT/README file will suffice for users, the documentation generated by doxygen is rather relevant to those who are interested in the implementation of MARMOT.

- LIB contains the three libraries libdmpi.a, libmpo.a and libfmpo.a, which are the essential parts of MARMOT. In fact, only these three libraries are required to make an application run with MARMOT, i.e. these three libraries have to be linked to the user's application.
- SRC/SRC contains the various source files representing MPI calls etc., precisely for the C language binding.
- SRC/INCLUDE contains the necessary header files.
- SRC/PROFILE is needed for the use of the MPI profiling interface.
- SRC/FORTRAN contains the source files that are necessary to support the Fortran language binding on top of the implementation of the C language binding.
- TEST_C contains simple test programs written in C.
- TEST_F contains simple test programs written in Fortran.

The following sections 4, 5 and 6 are intended to help the user to handle MARMOT. Apart from this manually-written documentation, the Doxygen tool has been used to automatically generate further documentation.

2.2. CONTACT INFORMATION

The following persons are responsible for the MPI code verification tool MARMOT:

Bettina Krammer, Matthias Mueller
High Performance Computing Centre Stuttgart (HLRS)
Allmandring 30, D-70550 Stuttgart, Germany
Phone: (+49-711) 685-8038
Fax: (+49-711) 6787626
Mail: {krammer, mueller}@hlrs.de
URL : http://www.hlrs.de

3. IMPLEMENTATION STRUCTURE

The MPI verification tool MARMOT is a tool that tries to verify the correct usage of MPI. Both the C and Fortran language binding of MPI standard 1.2 are supported. The C interface was implemented first, since PM12 the Fortran interface is also available. The Fortran interface is implemented as a wrapper on top of the C interface. Future developments and improvements will therefore be available at the same time for both language bindings.

MARMOT is a library that is linked to the MPI application in addition to the existing MPI library and that allows a detailed analysis of this application at runtime. The implementation of MARMOT is based on the model given in the Design Document, i.e. it is a library written in C++ and is specified as follows.

3.1. MPI INTERFACE

The technical basis of the verification tool is the MPI profiling interface which allows a detailed analysis of the MPI application at runtime. The MPI verification tool is therefore not a replacement of the MPI library, but an add on. It will be used in addition to the existing MPI library. The input to the tool consists of a sequence of MPI calls together with the provided arguments. The possible calls and the arguments are described in the MPI standard. A detailed description of the profiling interface can be found in chapter 8 of MPI 1.1 and in section 4.18 of MPI 2.0.

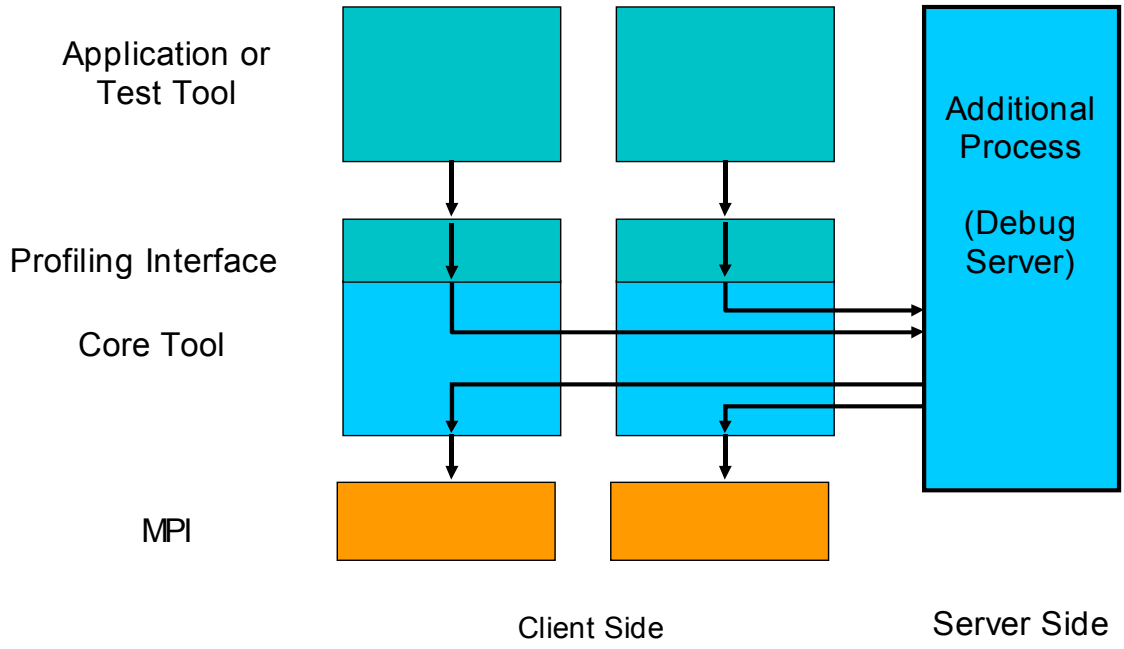

figure 1 Design of MARMOT

Figure 1 shows a graphical representation of how the MPI calls of the application or test tool are intercepted by the MPI profiling interface. Before the calls are passed onto the "real" MPI library they are inspected by the verification tool. The profiling interface is implemented with a simple layer that passes the control to the core of the verification tool. For some parts of the verification process a global view of the program execution is required. This part is handled by the so called debug server,

which is an additional process. This process is part of the library and added transparently for the application.

3.2. DEBUG SERVER

As stated in the Design Document, the additional process working as debug server is intended to execute the following tasks:

- Control the execution flow: the server does not only monitor the program flow, but does also control which process starts the next MPI call. This approach imposes a huge performance penalty.

  If required by the end-user, this will be implemented as an option.
- Signal conditions, e.g. deadlocks.
- Check matching send/receive pairs for consistency: within the MPI standard a message matches if the source, tag and communicator match. It is the responsibility of the user to provide a corresponding datatype on the sender and receiver. The tool will check whether this is the case.

  Such an approach that is reliable, but relatively inexpensive to compute has not been implemented yet.
- Output log (report errors etc.).

3.3. DEBUG CLIENT

The client checks described in the Design Document are all implemented, e.g. the verification of MPI_Request usage, of tag ranges etc.

3.4. CLASSES AND METHODS

The classes and methods have been implemented as proposed in the Design Document.

3.5. OUTPUT

The verification tool MARMOT provides the output that is described in the Design Document in more detail.

- Violations of the MPI standard are reported as error.
- Unusual behaviour or possible problems is reported as warnings.
- Notes are displayed when harmless but remarkable behaviour occurs.
- The MPI calls are traced on each node throughout the whole application.
- When detecting a deadlock the last few calls (as configured by the user) can be traced back on each node.
- The output is written to standard output (stdout and stderr) by default and can be redirected to a file by the user. The format is human readable.

3.6. CONFIGURATION

The following configurations can be set by the user via environmental variables:

- Three levels of debug mode:
  1. errors;
  2. errors and warnings;

3. errors, warnings and remarks.

- Tracing, i.e. whether MPI calls shall be traced or not.
- Maximum message time, i.e. how long a processing unit is allowed to stay idle before it is reported as pending.
- Maximum number of MPI calls that should be traced back in the case of a deadlock.

3.7. TEST TOOLS

In order to check the correctness of the verification tool, some simple test programs are provided, see also section 6.

3.8. SUMMARISING COMPARISON OF IMPLEMENTATION AGAINST DESIGN MODEL

To avoid a mere repetition of all that is said in the Design Document, only some short sentences have been given above to illustrate the status of the implementation. The comparison of the implementation against the design model can be summarised as follows:

- Accordance: in general, the design model has been realized, with some exceptions following below.
- Descrepancy:
    1. The control of the execution flow is not yet implemented as a user option.
    2. Send/receive pairs are not yet checked for consistency.

4. PROTOTYPE FUNCTIONALITY

4.1. STATE OF THE ART

Since MARMOT is a tool that aims at improving the software development productivity by automatically catching programming errors we compare MARMOT with other possible approaches and tools.

4.1.1. Classical debuggers

- Gdb: The freely available gdb debugger has currently no support for MPI. There are ongoing developments to use gdb as a backend debugger in conjunction with a front-end that supports MPI. One example is the commercial debugger DDT available by streamline computing, another is the mpigdb effort inside the mpich implementation.
- Totalview: has support for MPI and is available across many platforms and MPI implementations.
- P2d2: unfortunately p2d2 is not freely available. It has support for MPI and also is able to work in a Grid environment, e.g. in conjunction with MPICH-G.

The disadvantage of this approach is that is does not catch incorrect usage of MPI, but only helps to analyze the situation after the incorrect usage has produced other errors e.g. a segmentation violation.

4.1.2. Debug Versions of the MPI Library

Theoretically a debug version of the MPI library could catch all cases of incorrect usage of this library. However, the primary design goals of an MPI library are the correct behaviour and performance for correct MPI programs. Consistency and other internal tests are normally too expensive, especially if knowledge about the state of a remote process or even global knowledge about the MPI program is required.

4.1.3. Tools Designed to Catch MPI Programming Errors

- MPI-CHECK: this tool is designed to find problems like deadlocks, and performs argument type checking. It performs compile and run-time checking. At compile time a source code modification is performed. Since a language parser is required, it is currently only available for F77 and F90. This is a disadvantage shared with many other freely available tools which require language parsers. Languages like C++ are difficult to parse and therefore not supported.
- Umpire: similar to MARMOT it also uses the profiling interface to avoid the problem of language parsing. In contrast to MARMOT it does not use MPI as transport mechanism between the clients and the debug server, but a shared memory mechanism. It is therefore limited to shared memory platforms. Distributed MPI runs with Grid enabled MPI libraries as planned in the CrossGrid project are not possible with umpire.

4.1.4. MARMOT'S Functionality vs. the current Tools Technology

The summary of advantages of the approach taken with MARMOT compared to current technology are:

- It supports all programming languages that can make use of the two language bindings for MPI 1.2: F77, F90, F95, C, C++ and others.
- It supports all platforms that are supported by MPI: distributed memory systems, shared memory systems, cluster of SMPs and even Metacomputers on the Grid.

- It works automatically without user intervention and inspection at runtime.
- It is designed to improve the portability by reporting non-portable constructs.

4.2. CONTRIBUTION TO GRID TECHNOLOGY

The motivation to develop MARMOT is based on the experiences with PACX-MPI and a large number of applications using PACX-MPI. In a large number of occasions the portability problems addressed by MARMOT showed up. In every case this resulted in long and expensive debug sessions where MPI developpers and application developers had to find the reason for erratic application behaviour when it was executed on the Grid.

MARMOT was designed to support portability issues between all MPI implementations. Strictly speaking a Grid enabled MPI is just another MPI implementation, although problems are much more likely to occur due to the dynamic and heterogeneous environment plagued with problems like low bandwidth and high latencies.

Currently there is no MPI check and verification tool available on the Grid. This is caused by the limitation of the current tool technology as described in the previous chapter. MARMOT is the first tool in that area that works on the Grid. It already has shown to work with two major Grid enabled MPI implementations (MPICH-G2 and PACX-MPI) in first simple tests.

A new contribution will also be the specific tests for heterogeneous runs that are planned for the next prototype. Although heterogeneous runs are not part of CrossGrid we regard this feature as essential to support applications like multi-physics simulations where different parts run best on different platforms.

4.3. MARMOT'S FUNCTIONALITY

From the user point of view the verification tool is a library that has to be linked to the application. If this library is linked to the application a report will be generated at runtime which contains comments regarding the MPI standard conformance of the MPI program. The library consists of the debug clients and one debug server. No source code modification is required, only an additional process has to be added at start-up time, i.e. the user will have to run the job with mpirun for n+1 instead of n processes. The debug server will catch one process automatically, i.e. if the user forgets to add one process only n-1 instead of n processes will be available to the application.

The prototype supports the C and Fortran language binding of MPI standard 1.2, in particular the calls used by the applications of WP1 as enumerated in the SRS Document:

MPI calls	Check whether…
MPI_Barrier	communicator is valid
MPI_Bcast	communicator, count, datatype and rank of root are valid
MPI_Cart_create	communicator is valid
MPI_Cart_rank	communicator and coordinates are valid
MPI_Cart_shift	communicator is valid
MPI_Comm_rank	communicator is valid
MPI_Comm_size	communicator is valid
MPI_Finalize	active requests and pending messages are left
MPI_Gatherv	communicator, count, datatype, rank of root, displacements are valid

MPI_Get_processor_name	currently no checks
MPI_Init	currently no checks
MPI_Pack	counts, communicator and datatype are valid.
MPI_Recv	communicator, count, datatype, rank, tag are valid
MPI_Reduce	communicator, count, datatype, rank of root, operator are valid
MPI_Scatterv	communicator, count, datatype, rank of root, displacements are valid
MPI_Send	communicator, count, datatype, rank, tag are valid
MPI_Sendrecv	communicator, count, datatype, rank, tag are valid
MPI_Type_commit	datatype is valid
MPI_Type_extent	datatype is valid
MPI_Type_hvector	count and datatype are valid, blocklength is >0
MPI_Type_struct	count and entries in datatype[] are valid, blocklength is >0
MPI_Unpack	count, size and datatype are valid
MPI_Wtime	currently no checks

table 2 Supported MPI calls

5. USER MANUAL

The following section contains a short introduction on how to install and use MARMOT.

5.1. DEPENDENCIES

As MARMOT is intended to be a portable tool running on any environment, there is no specific hardware required. The following software is required:

- MPI library: since the program that is to be verified is written using MPI, this library is needed to run the program. The MPI verification tool verifies the calls made by the program with the use of the so called profiling interface. 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.
- C++ compiler: the MPI verification tool is implemented in C++. The compiler should implement the ISO/IEC 14882 language specification of C++. gcc version 3.0.4 or newer is required for a sufficient support of the C++ language standard. Intel Compilers are an alternative, they are available for no cost for non-commercial use on linux platforms. For example, on our local environment, the KAI C++ 4.0 compiler has been used successfully. We have also succeeded in using gcc 3.1.
- To support the Fortran binding of the MPI standard a Fortran compiler is required. The same Fortran compiler should be used to compile the application.
- Doxygen (tested with version 1.2.14) is used to automatically generate documentation.
- make (tested with GNU make version 3.79.1)

Further details can be found in the SRS Document.

5.2. INSTALLATION

The following steps explain how to install and configure MARMOT.

1. Download the code from the FZK CVS-repository (http://gridportal.fzk.de/):

   ```
   cvs -z3 co crossgrid/wp2/wp2_2-verif
   ```

   or go to http://gridportal.fzk.de/cgi-bin/viewcvs.cgi/crossgrid/crossgrid/wp2/wp2_2-verif/src/ and click on the "download tarball" button.

   Have a look at the `MARMOT/README` file, which contains the up-to-date information on the installation of MARMOT.
2. Go to the directory `crossgrid/wp2/wp2_2-verif/src/MARMOT`.

The next four steps explain how to adjust the Makefiles.

3. Go to the directory `MARMOT/SRC/SRC` and change the MPI and CC paths in the `Makefile`.
4. Go to the directory `MARMOT/SRC/PROFILE` and change the MPI and CC paths in the `Makefile` according to step3.
5. Go to the directory `MARMOT/TEST_C` and change the MPI and CC paths in the `Makefile` according to step 3.
6. Go to the directory `MARMOT/TEST_F` and change the MPI and CC resp. FC paths in the `Makefile` according to step 3.
7. Go to the directory `MARMOT/DOC/AUTO` and change `doxygen.config` if necessary.
8. Execute

```
$ cd MARMOT
$ ./configure
$ make
```

to compile the source code. To clean up the files that were generated by `make`, execute

```
$ make clean
```

If you use

```
$ make distclean
```

and want to compile MARMOT again, you must execute

```
$ cd MARMOT
$ autoconf
$ ./configure
$ make
```

5.3. CONFIGURATION

- Set the environmental variable `DEBUG_MODE` to the desired value. The values for the debug mode mean

  0: only errors,

  1: errors and warnings,

  2: errors, warnings and remarks are reported.

  The default value is 2. For example in bash one would issue the command

  ```
  $ export DEBUG_MODE=1
  ```

- Set the environmental variable `INTERFACE_MODE` for the interface mode (default 0). The values mean

  0: C Interface

  1: Fortran interface

  E.g. in bash one would simply issue the command

  ```
  export INTERFACE_MODE=1
  ```

  to get the fortran interface. If the user forgets to set the Fortran interface mode for Fortran applications, MARMOT may issue error messages that are specific to the C language binding and do not apply for the Fortran language binding.

- Set the environmental variable `MAX_TIMEOUT` for the maximum message time, i.e. set a value in microseconds for `MAX_TIMEOUT` (default 1000) to limit the maximum message time.

  E.g. in bash one would simply issue the command

  ```
  export MAX_TIMEOUT=100000
  ```

- Set the environmental variable `TRACE_CALLS`, whether calls shall be traced back or not (default value 1). The values mean

  1: calls are traced with output to stderr, traceback in case of a deadlock is possible

  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.

  The number of calls to be traced back in case of deadlock can be set via `MAX_PEND_COUNT`.

  E.g. in bash one would simply issue the command

```
export TRACE_CALLS=0
```

- Set the environmental variable `MAX_PEND_COUNT` for the maximum number of MPI calls that can be traced back, i.e. set an int value for `MAX_PEND_COUNT` (default 10).

  E.g. in bash one would simply issue the command

```
export MAX_PEND_COUNT=2
```

Environmental Variable	Short Description	Value	Description of values	Default
DEBUG_MODE	Set extent of reporting messages	0 1 2	only errors errors & warnings errors, warnings & remarks are reported	2
INTERFACE_MODE	Set interface	0 1	C interface Fortran interface	0
MAX_TIMEOUT	Set value for maximum message time	Int >= 0	maximum message time	1000 [microseconds]
TRACE_CALLS	Set extent of tracing	1 0 -1	tracing with output to stderr, traceback in case of a deadlock is possible tracing without output to stderr, traceback in case of a deadlock is possible no tracing, traceback in case of a deadlock is NOT possible.	1
MAX_PEND_COUNT	Set value for maximum number of traced-back calls	Int	Maximum number of calls to trace-back	10

table 3 Environment Variables

5.4. RUNNING

The following examples show how to run an application with MARMOT, which is a library to be linked to the application, i.e. the application has to be rebuilt. To finally run the application, an additional process working as debug server has to be added.

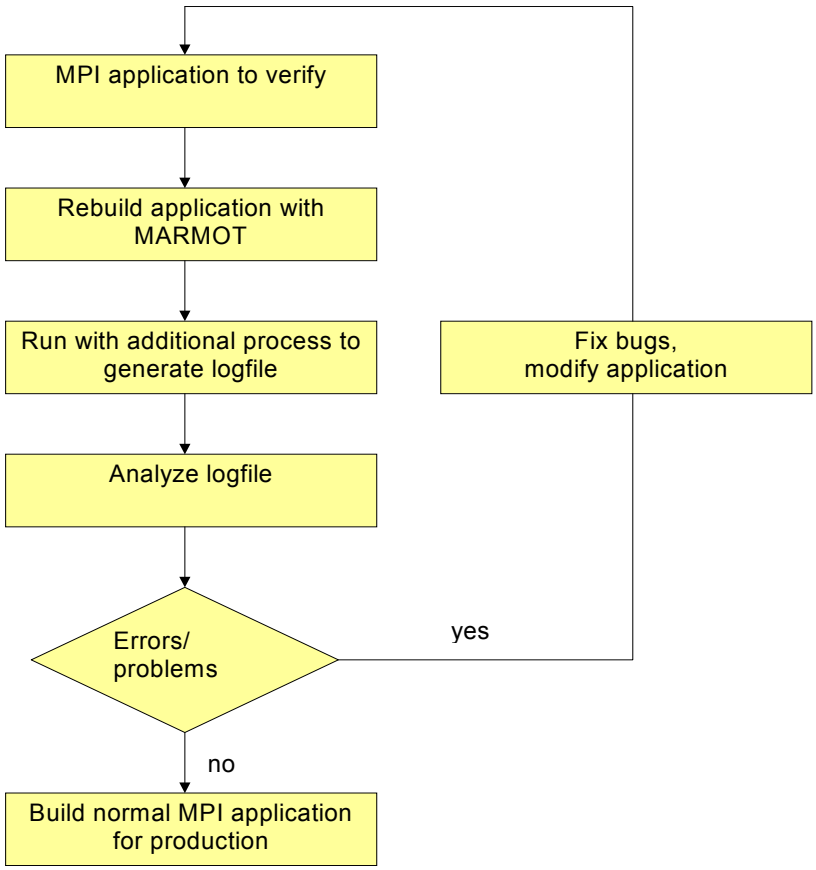

figure 2 How to use an application with MARMOT

One category of examples are the simple test programs that come along with the verification tool. Another category are applications from other tasks, e.g. the HEP application from Task1.3, the meteo-application from task 1.4.3 or the benchmarks from task 2.3. Both these categories are also used for internal tests (see section 6).

5.4.1. Scheme for Makefiles for C programs

The test programs contained in the directory `MARMOT/TEST_C` were compiled using the following partly user-specific Makefile settings (the MPI paths and CC paths may have to be adjusted by the user, see section 5.2 on installation)

```
CC = gcc
INCDIR = ../SRC/INCLUDE
LIBDIR = ../LIB
```

```
GCC_LIBDIR = /opt/lib
GCC_LIB = stdc++

MPIDIR = /home/rusbetti/mpich-1.2.4
MPI_INCDIR = $(MPIDIR)/include
MPI_LIBDIR = $(MPIDIR)/lib
MPI_LIBS = -lpmpich -lmpich

CFLAGS = -I$(INCDIR) -I$(MPI_INCDIR) -g
LDFLAGS = -static

foo: foo.o
$(CC)   -o   foo   foo.o   $(LDFLAGS)   -L$(LIBDIR)   -ldmpi   -lmpo   -
L$(MPI_LIBDIR) $(MPI_LIBS) -L$(GCC_LIBDIR) -l$(GCC_LIB)

foo.o: foo.c
            $(CC) $(CFLAGS) -c foo.c
```

If the user wants to run `foo` with e.g. 4 processes, an additional process will have to be added, i.e. `foo` is executed by typing the command

```
$ mpirun -np 5 foo
```

To set a value for the environment variable debug_mode, e.g. that only errors are reported, one would execute (in bash)

```
$ export DEBUG_MODE=0
$ mpirun -np 5 foo
```

To illustrate this with some concrete examples, we regard the following two simple test programs `MARMOT/TEST_C/deadlock1.c` and `MARMOT/TEST_C/request-reuse1.c`.

5.4.2. Example (deadlock)

The simple test program `MARMOT/TEST_C/deadlock1.c` looks as follows:

```
/*
**  This program produces a deadlock.
**  At least 2 nodes are required to run the program.
**
**  Rank 0 recv a message from Rank 1.
**  Rank 1 recv a message from Rank 0.
**
**  AFTERWARDS:
**  Rank 0 sends a message to Rank 1.
**  Rank 1 sends a message to Rank 0.
 */

#include <stdio.h>
#include "mpi.h"

int main( int argc, char** argv ){
  int rank  = 0;
  int size  = 0;
  int dummy = 0;
  MPI_Status status;
  MPI_Init( &argc, &argv );
```

```
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  if( size < 2 ){
    fprintf( stderr, " This program needs at least 2 PEs!\n" );
  }
  else {
    if (rank == 0){
      MPI_Recv( &dummy,1,MPI_INT,1,17,MPI_COMM_WORLD,&status );
      MPI_Send( &dummy,1,MPI_INT,1,18,MPI_COMM_WORLD );
    }
    if(rank == 1){
      MPI_Recv( &dummy,1,MPI_INT,0,18,MPI_COMM_WORLD,&status );
      MPI_Send( &dummy,1,MPI_INT,0,17,MPI_COMM_WORLD );
    }
  }
  MPI_Finalize();
}
```

To compile this program, one proceeds as described in section 5.4.1.
To run this program, one needs at least 2 processes plus an additional debug process:

```
$ mpirun -np 3 deadlock1
```

This produces the output

```
1 rank 1 performs MPI_Init
2 rank 0 performs MPI_Init
3 rank 0 performs MPI_Comm_rank
4 rank 1 performs MPI_Comm_rank
5 rank 0 performs MPI_Comm_size
6 rank 1 performs MPI_Comm_size
7 rank 0 performs MPI_Recv
8 rank 1 performs MPI_Recv
8 Rank 0 is pending!
8 Rank 1 is pending!
 WARNING: deadlock detected, all clients are pending
Last calls (max. 10) on node 0:
 timestamp= 2: MPI_Init(*argc, ***argv)
 timestamp= 3: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp= 5: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp=   7:   MPI_Recv(*buf,   count=-1,   datatype=non-predefined
datatype, source=-1, tag=-1, comm=MPI_COMM_NULL, *status)

Last calls (max. 10) on node 1:
 timestamp= 1: MPI_Init(*argc, ***argv)
 timestamp= 4: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp= 6: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp=   8:   MPI_Recv(*buf,   count=-1,   datatype=non-predefined
datatype, source=-1, tag=-1, comm=MPI_COMM_NULL, *status)
```

5.4.3. Example (request-reuse)

The simple test program `MARMOT/TEST_C/request-reuse1.c` looks as follows:

```
/*
** Here we re-use a request we didn't free before
*/
```

```
#include <stdio.h>
#include <assert.h>
#include "mpi.h"

int main(int argc,char **argv)
{
  int size    = -1;
  int rank    = -1;
  int value   = -1;
  int value2  = -1;
  MPI_Status  send_status, recv_status;
  MPI_Request send_request, recv_request;

  printf("We call Irecv and Isend with non-freed requests.\n");
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf( " I am rank %d of %d PEs\n",rank,size );

  if( rank == 0 ){
    /* This is just to get the request used */
    MPI_Isend(&value,1,MPI_INT,1,16,MPI_COMM_WORLD,&recv_request);

    /* going to receive the message and reuse a non-freed request*/
    MPI_Irecv(&value,1,MPI_INT,1,17,MPI_COMM_WORLD,&recv_request);
    MPI_Wait(&recv_request, &recv_status);
    assert(value=19);
  }
  if(rank == 1){
    value2 = 19;
    /* this is just to use the request */
    MPI_Isend(&value,1,MPI_INT,0,18,MPI_COMM_WORLD,&send_request);

    /* going to send the message */
    MPI_Isend(&value2,1,MPI_INT,1,17,MPI_COMM_WORLD,&send_request);
    MPI_Wait(&send_request, &send_status);
  }
  MPI_Finalize();
  return 0;
}
```

To compile this program, one proceeds again as described in section 5.4.1.
To run this program, one needs at least 2 processes plus an additional debug process:

```
$ mpirun -np 3 request-reuse1
```

This produces the output (written to the standard output)

```
We call Irecv and Isend with non-freed requests.
We call Irecv and Isend with non-freed requests.
We call Irecv and Isend with non-freed requests.
1 rank 1 performs MPI_Init
2 rank 0 performs MPI_Init
3 rank 0 performs MPI_Comm_size
4 rank 1 performs MPI_Comm_size
5 rank 0 performs MPI_Comm_rank
6 rank 1 performs MPI_Comm_rank
7 rank 0 performs MPI_Isend
```

```
8 rank 1 performs MPI_Isend
9 rank 0 performs MPI_Irecv
10 rank 1 performs MPI_Isend
 I am rank 0 of 2 PEs
ERROR: MPI_Irecv Request is still in use !!
11 rank 0 performs MPI_Wait
 I am rank 1 of 2 PEs
ERROR: MPI_Isend Request is still in use !!
12 rank 1 performs MPI_Wait
12 Rank 0 is pending!
13 rank 1 performs MPI_Finalize
WARNING: MPI_Finalize: There are still pending messages!
13 Rank 1 is pending!
 WARNING: deadlock detected, all clients are pending
Last calls (max. 10) on node 0:
 timestamp= 2: MPI_Init(*argc, ***argv)
 timestamp= 3: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp= 5: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp=   7:   MPI_Isend(*buf,   count=1,   datatype=non-predefined
datatype, dest=1, tag=16, comm=MPI_COMM_NULL, *request)
 timestamp=   9:   MPI_Irecv(*buf,   count=1,   datatype=non-predefined
datatype, source=1, tag=17, comm=MPI_COMM_NULL, *request)
 timestamp= 11: MPI_Wait(*request, *status)

Last calls (max. 10) on node 1:
 timestamp= 1: MPI_Init(*argc, ***argv)
 timestamp= 4: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp= 6: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp=   8:   MPI_Isend(*buf,   count=1,   datatype=non-predefined
datatype, dest=0, tag=18, comm=MPI_COMM_NULL, *request)
 timestamp=  10:   MPI_Isend(*buf,   count=1,   datatype=non-predefined
datatype, dest=1, tag=17, comm=MPI_COMM_NULL, *request)
 timestamp= 12: MPI_Wait(*request, *status)
 timestamp= 13: MPI_Finalize()
```

However, for larger programs it seems more convenient to redirect the output to a file that can be properly analyzed. To do so, one executes

```
$ mpirun -np 3 request-reuse1 >  all.log 2>&1
```

Thus, the output from above is written to the file `all.log`.

Sometimes the user may wish to separate the application's output from MARMOT's output, i.e. the user has a log file containing the reports from the application

```
We call Irecv and Isend with non-freed requests.
We call Irecv and Isend with non-freed requests.
We call Irecv and Isend with non-freed requests.
 I am rank 0 of 2 PEs
 I am rank 1 of 2 PEs
```

and another log file containing the reports from MARMOT

```
1 rank 1 performs MPI_Init
2 rank 0 performs MPI_Init
3 rank 0 performs MPI_Comm_size
4 rank 1 performs MPI_Comm_size
5 rank 0 performs MPI_Comm_rank
6 rank 1 performs MPI_Comm_rank
7 rank 0 performs MPI_Isend
```

```
8 rank 1 performs MPI_Isend
9 rank 0 performs MPI_Irecv
10 rank 1 performs MPI_Isend
ERROR: MPI_Irecv Request is still in use !!
11 rank 0 performs MPI_Wait
ERROR: MPI_Isend Request is still in use !!
12 rank 1 performs MPI_Wait
12 Rank 0 is pending!
13 rank 1 performs MPI_Finalize
WARNING: MPI_Finalize: There are still pending messages!
13 Rank 1 is pending!
 WARNING: deadlock detected, all clients are pending
Last calls (max. 10) on node 0:
 timestamp= 2: MPI_Init(*argc, ***argv)
 timestamp= 3: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp= 5: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp=   7:   MPI_Isend(*buf,   count=1,   datatype=non-predefined
datatype, dest=1, tag=16, comm=MPI_COMM_NULL, *request)
 timestamp=   9:   MPI_Irecv(*buf,   count=1,   datatype=non-predefined
datatype, source=1, tag=17, comm=MPI_COMM_NULL, *request)
 timestamp= 11: MPI_Wait(*request, *status)

Last calls (max. 10) on node 1:
 timestamp= 1: MPI_Init(*argc, ***argv)
 timestamp= 4: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp= 6: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp=   8:   MPI_Isend(*buf,   count=1,   datatype=non-predefined
datatype, dest=0, tag=18, comm=MPI_COMM_NULL, *request)
 timestamp=  10:   MPI_Isend(*buf,   count=1,   datatype=non-predefined
datatype, dest=1, tag=17, comm=MPI_COMM_NULL, *request)
 timestamp= 12: MPI_Wait(*request, *status)
 timestamp= 13: MPI_Finalize()
```

To achieve this, one can run the following command

```
$ mpirun -np 3 request-reuse1 > stdout.log 2> stderr.log
```

to get the two corresponding log files `stdout.log` and `stderr.log`.

To redirect for example the application's output to standard output and MARMOT's output to a log file, one can use the command

```
$ mpirun -np 3 request-reuse1 2> stderr.log
```

Finally, if the user wants to set a different value for the environment variable debug_mode, that e.g. only errors are reported, one will execute

```
$ export DEBUG_MODE=0
$ mpirun -np 3 request-reuse1 2> stderr.log
```

5.4.4. Example (HEP application from Task 1.3, benchmarks from task 2.3)

To run other C applications with MARMOT, e.g. the HEP application from WP1 or a benchmark from task 2.3, one simply proceeds according to the scheme given in section 5.4.1.

5.4.5. Scheme for Makefiles for Fortran programs

The test programs contained in the directory `MARMOT/TEST_F` were compiled using the following partly user-specific Makefile settings (the MPI paths and CC resp. FC paths will have to be adjusted

by the user, see section 5.2 on installation)

```
CC = g++
FC = g77

INCDIR = ../SRC/INCLUDE
LIBDIR=../LIB

GCC_LIBDIR  = /opt/lib
GCC_LIBS    = -lm -lfrtbegin -lg2c -lgcc_s

MPIDIR=/home/rusbetti/mpich-1.2.4
MPI_INCDIR=$(MPIDIR)/include
MPI_LIBDIR=$(MPIDIR)/lib
MPI_LIBS= -lpmpich -lmpich

CFLAGS = -I$(INCDIR) -I$(MPI_INCDIR) -g
LDFLAGS =

foo: foo.o
$(CC) -o foo foo.o $(LDFLAGS) -L$(LIBDIR) -ldmpi -lfmpo -lmpo -
L$(MPI_LIBDIR) $(MPI_LIBS) -L$(GCC_LIBDIR) $(GCC_LIBS)


.f.o:
                    $(FC) -I$(MPI_INCDIR) -c $<
```

Fortran programs can then be run like the C programs described above.

To illustrate this with a concrete example, we regard the following simple test program `MARMOT/TEST_F/ring1.f`, which produces a deadlock.

5.4.6. Example (deadlock)

The simple test program `MARMOT/TEST_F/ring1.f` looks as follows:

```
PROGRAM ring

IMPLICIT NONE

INCLUDE "mpif.h"

INTEGER to_right
PARAMETER(to_right=201)
INTEGER ierror, my_rank, size
INTEGER right, left
INTEGER  i, sum
INTEGER send_buf, recv_buf
INTEGER status(MPI_STATUS_SIZE)

CALL MPI_INIT(ierror)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)

right = mod(my_rank+1,      size)
left  = mod(my_rank-1+size, size)
sum = 0
```

```
      send_buf = my_rank

      DO i = 1, size
           CALL MPI_SSEND(send_buf, 1, MPI_INTEGER, right,
     &                    to_right, MPI_COMM_WORLD,
     &                    ierror)

           CALL MPI_RECV(recv_buf, 1, MPI_INTEGER, left, to_right,
     &                   MPI_COMM_WORLD, status, ierror)

           sum = sum + recv_buf
           send_buf = recv_buf
      END DO

      WRITE(*,*) "PE", my_rank, ": Sum =", sum

      CALL MPI_FINALIZE(ierror)

      END
```

This program is run by executing

```
export INTERFACE_MODE=1
```

to invoke the Fortran interface (i.e. to suppress error reports that are specific to the C language binding) and

```
mpirun -np 5 ring1
```

The output looks as follows:

```
1 rank 0 performs MPI_Init
2 rank 2 performs MPI_Init
3 rank 1 performs MPI_Init
4 rank 3 performs MPI_Init
5 rank 0 performs MPI_Comm_rank
6 rank 1 performs MPI_Comm_rank
7 rank 2 performs MPI_Comm_rank
8 rank 3 performs MPI_Comm_rank
9 rank 0 performs MPI_Comm_size
10 rank 1 performs MPI_Comm_size
11 rank 2 performs MPI_Comm_size
12 rank 3 performs MPI_Comm_size
13 rank 0 performs MPI_Ssend
14 rank 1 performs MPI_Ssend
15 rank 2 performs MPI_Ssend
16 rank 3 performs MPI_Ssend
16 Rank 0 is pending!
16 Rank 1 is pending!
16 Rank 2 is pending!
16 Rank 3 is pending!
 WARNING: deadlock detected, all clients are pending
Last calls (max. 10) on node 0:
 timestamp= 1: MPI_Init(*argc, ***argv)
 timestamp= 5: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp= 9: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp=  13:  MPI_Ssend(*buf,  count=-1,  datatype=non-predefined
datatype, dest=-1, tag=-1, comm=MPI_COMM_NULL)

Last calls (max. 10) on node 1:
 timestamp= 3: MPI_Init(*argc, ***argv)
```

```
 timestamp= 6: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp= 10: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp=  14:  MPI_Ssend(*buf,  count=-1,  datatype=non-predefined
datatype, dest=-1, tag=-1, comm=MPI_COMM_NULL)

Last calls (max. 10) on node 2:
 timestamp= 2: MPI_Init(*argc, ***argv)
 timestamp= 7: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp= 11: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp=  15:  MPI_Ssend(*buf,  count=-1,  datatype=non-predefined
datatype, dest=-1, tag=-1, comm=MPI_COMM_NULL)

Last calls (max. 10) on node 3:
 timestamp= 4: MPI_Init(*argc, ***argv)
 timestamp= 8: MPI_Comm_rank(comm=MPI_COMM_NULL, *rank)
 timestamp= 12: MPI_Comm_size(comm=MPI_COMM_NULL, *size)
 timestamp=  16:  MPI_Ssend(*buf,  count=-1,  datatype=non-predefined
datatype, dest=-1, tag=-1, comm=MPI_COMM_NULL)
```

(Note that the traced-back calls still show the C language binding…)

5.4.7. Example (meteo-application from Task1.4.3)

To run other Fortran applications with MARMOT, e.g. the meteo-application from WP1, one simply proceeds according to the scheme given in section 5.4.5.

5.5. USER SUPPORT

- Users may submit and survey bugs via the bug-tracking system that can be found at http://gridportal.fzk.de/bugs/?group=cg-wp2-2
- There's also a public forum named "user-feedback", where users can post their feedback to and which is available at http://gridportal.fzk.de/forum/?group=cg-wp2-2
- The file MARMOT/KnownBugs is regularly updated.
- Questions and suggestions can also be sent directly to Bettina Krammer, krammer@hlrs.de , or to Matthias Müller, mueller@hlrs.de .

5.6. INTERFACE DESCRIPTION

The tool provides the API as defined in the MPI standard version 1.2.

6. INTERNAL TESTS

In order to test MARMOT, some simple programs containing various MPI calls are delivered in the `TEST_C` and the `TEST_F` directory. These test programs may be erroneous or not. Another test programs were delivered by WP1 and WP2, namely the HEP application from Task 1.3, the meteo-application from Task 1.4.3 and a benchmark from Task 2.3.

6.1. SIMPLE TEST PROGRAMS

In order to check the correctness of the verification tool, two categories of test programs were used.

- Correct MPI programs: they still run correctly when using the tool.
- Incorrect MPI programs: they are used to test whether the errors are detected by the tool. Examples are deadlocks and incorrect recycling of requests.

These test programs are provided together with the library to allow users to:

- check whether the library works correctly after installation on arbitrary machines
- see how the verification tool deals with correct/incorrect MPI code.

Some of these examples were shown in the last section.

6.2. HEP APPLICATION FROM TASK 1.3

The C interface of MARMOT was also tested with the HEP-applicaton from task 1.3.

- On a local environment: MARMOT did neither report errors/warnings/remarks nor deadlocks. The HEP application from WP1 generates the same output both with and without the verification tool MARMOT linked to it. In the latter case, running for example `cg-ann-light.exe` with 5 processes (i.e.with 4 plus one additional debug process), the output log merely consists of reports like the last few given below:

```
5214 rank 1 performs MPI_Bcast
5215 rank 2 performs MPI_Bcast
5216 rank 3 performs MPI_Bcast
5217 rank 0 performs MPI_Reduce
5218 rank 1 performs MPI_Reduce
5219 rank 2 performs MPI_Reduce
5220 rank 3 performs MPI_Reduce
5220 Rank 0 is pending!
5221 rank 1 performs MPI_Bcast
5221 Rank 2 is pending!
Epoch 99 error=1.156534e-06
Epoch 100 error=1.120634e-06
5222 rank 3 performs MPI_Bcast
5223 rank 2 performs MPI_Bcast
5224 rank 0 performs MPI_Bcast
5224 Rank 0 is pending!
 End of training message
 Finalize MPI
```

```
 Finalize MPI

5225 rank 0 performs MPI_Finalize
5226 rank 1 performs MPI_Finalize

 Finalize MPI

5227 rank 2 performs MPI_Finalize

 Finalize MPI

5228 rank 3 performs MPI_Finalize
5228 Rank 0 is pending!
5228 Rank 1 is pending!
5228 Rank 2 is pending!
```

- On the testbed: : MARMOT did not report any errors/warnings/remarks, but issued unjustified deadlock warnings.

6.3. METEO-APPLICATION FROM TASK 1.4.3

The Fortran interface of MARMOT was tested with the meteo-application `STEMII_mpi.exe` from Task 1.4.3.

The example below shows a fragment of the log file when the meteo-application runs on 4 processors (i.e. on 5 including the additional debug process) on a local environment. MARMOT does not report any errors/warnings, but remarks on the use of `MPI_Scatterv` and `MPI_Gatherv`. It issues unjustified deadlock warnings.

```
589 rank 1 performs MPI_Scatterv
590 rank 2 performs MPI_Scatterv
591 rank 0 performs MPI_Scatterv
592 rank 3 performs MPI_Scatterv
REMARK: MPI_Scatterv: Datatype has holes, so we don't check whether
it's correct,but displs[1] reads data/ holes read from other nodes!
REMARK: MPI_Scatterv: Datatype has holes, so we don't check whether
it's correct,but displs[2] reads data/ holes read from other nodes!
REMARK: MPI_Scatterv: Datatype has holes, so we don't check whether
it's correct,but displs[3] reads data/ holes read from other nodes!
592 Rank 0 is pending!
592 Rank 1 is pending!
592 Rank 2 is pending!
592 Rank 3 is pending!
 WARNING: deadlock detected, all clients are pending
```

```
Last calls (max. 1) on node 0:
 timestamp=   591:   MPI_Scatterv(*sendbuf,   *sendcounts,   *displs,
sendtype=non-predefined     datatype,     *recvbuf,     recvcount=16,
recvtype=non-predefined datatype, root=0, comm=MPI_COMM_NULL)

Last calls (max. 1) on node 1:
 timestamp=   589:   MPI_Scatterv(*sendbuf,   *sendcounts,   *displs,
sendtype=non-predefined     datatype,     *recvbuf,     recvcount=15,
recvtype=non-predefined datatype, root=0, comm=MPI_COMM_NULL)

Last calls (max. 1) on node 2:
 timestamp=   590:   MPI_Scatterv(*sendbuf,   *sendcounts,   *displs,
sendtype=non-predefined     datatype,     *recvbuf,     recvcount=15,
recvtype=non-predefined datatype, root=0, comm=MPI_COMM_NULL)

Last calls (max. 1) on node 3:
 timestamp=   592:   MPI_Scatterv(*sendbuf,   *sendcounts,   *displs,
sendtype=non-predefined     datatype,     *recvbuf,     recvcount=15,
recvtype=non-predefined datatype, root=0, comm=MPI_COMM_NULL)

593 rank 0 performs MPI_Barrier
594 rank 1 performs MPI_Barrier
594 Rank 0 is pending!
595 rank 2 performs MPI_Barrier
596 rank 3 performs MPI_Barrier

[…]

605 rank 0 performs MPI_Gatherv
REMARK: MPI_Gatherv: Datatype has holes, so we don't check whether
it's correct,but displs[1] overrides data/ holes from other nodes!
REMARK: MPI_Gatherv: Datatype has holes, so we don't check whether
it's correct,but displs[2] overrides data/ holes from other nodes!
REMARK: MPI_Gatherv: Datatype has holes, so we don't check whether
it's correct,but displs[3] overrides data/ holes from other nodes!
606 rank 1 performs MPI_Gatherv
```

6.4. BENCHMARKS FROM TASK 2.3

When the `xhpl` benchmark from Task 2.3 runs on 4 processors (i.e. on 5 including the additional debug process) on a local environment MARMOT does not report any errors/warnings, but remarks on the use of `MPI_Type_commit`, e.g.

```
230 rank 1 performs MPI_Address
231 rank 2 performs MPI_Wait
232 rank 3 performs MPI_Wait
```

```
233 rank 0 performs MPI_Address
234 rank 1 performs MPI_Address
235 rank 2 performs MPI_Irecv
236 rank 3 performs MPI_Irecv
237 rank 0 performs MPI_Type_struct
238 rank 1 performs MPI_Type_struct
239 rank 2 performs MPI_Send
240 rank 3 performs MPI_Send
241 rank 0 performs MPI_Type_commit
242 rank 1 performs MPI_Type_commit
243 rank 2 performs MPI_Wait
243 Rank 3 is pending!
NOTE: MPI_Type_commit: Datatype already committed!
244 rank 3 performs MPI_Wait
245 rank 0 performs MPI_Iprobe
NOTE: MPI_Type_commit: Datatype already committed!
246 rank 1 performs MPI_Iprobe
247 rank 2 performs MPI_Address
248 rank 3 performs MPI_Address
249 rank 0 performs MPI_Iprobe
250 rank 1 performs MPI_Iprobe
251 rank 2 performs MPI_Address
252 rank 3 performs MPI_Address
```

6.5. RESULTS OF TESTS AND EVALUATION

First tests of MARMOT show that the basic functionality works reliably and that it can be configured by the user as described above. MARMOT's design goals are attained by now as follows:

- Portability: MARMOT does neither require any specific hardware nor any specific MPI implementation. Details concerning these requirements can be found in the SRS and the Prototype documents. MARMOT works on different platforms using different MPI implementations, however, the complete range of possible platforms and MPI implementations has not been tested yet.
- Reproducibility: MARMOT issues the warnings regarding reproducibility as planned.
- Reliability: so far MARMOT always worked reliable with the internal test programs and with the applications from WP 1.3 (HEP) and WP 1.4 (METEO) and the benchmarks from WP 2.3. I.e. MARMOT did not modify the intended behaviour of the programs.
- Scalability: the automatic analysis during runtime works as expected. However, it should be noted that MARMOT has not yet undergone performance measurements. Huge performance impacts could limit the scalability or even the usability significantly.

7. ISSUES

7.1. KNOWN BUGS

- Running the HEP-application and the meteo-application with MARMOT shows that MARMOT issues deadlock warnings though there isn't a deadlock. This behaviour needs further investigation. The current plan is to replace the simple deadlock detection with a more advanced algorithm. The exact kind of algorithm that is required for a reliable detection for distributed runs on the Grid is currently under examination.
- Bugs or suggestions can be reported as described in section 5.5.

7.2. FURTHER ISSUES

This first prototype aims at providing a reliable, portable tool – any performance measurements have not yet been done. More details can be found in the Report Doc.