A Survey of MPI Related Debuggers and Tools*

Subodh Sharma Ganesh Gopalakrishnan Robert M. Kirby

School of Computing, University of Utah
Salt Lake City, UT 84112, U.S.A.

UUCS-07-015

1 Overview

Message Passing Interface is a widely used standard in the High Performance and Scientific Computing Community for writing programs that can exploit the capability of parallel platforms. However, the inherent complexity and the size of the communication standard have made it difficult for programmers to use it efficiently and more importantly *correctly*. There are numerous tools and debuggers written by various academic/industry communities to find bugs in the MPI programs written by users. Some of them are MPI-CHECK (Iowa state Univ, [12]), MPIDD (UNBC, Canada [6]), UMPIRE (LLNL, [15]), Intel Message Checker (Intel, [5]), MARMOT (HLRS, [8]) and TotalView ([1]). A brief analysis and comparison of these tools are presented below. In addition, this report presents an overview of the debugging support build into some of the currently popular MPI libraries.

2 A Survey of MPI Debugging Tools

2.1 MPI-CHECK

MPI-CHECK supports only FORTRAN programs. The version that supports C/C++ is under developement. Unlike other tools like UMPIRE and MARMOT, MPI-CHECK does not use the MPI profiling Interface to capture the calls and analyze them; instead, using a macro-like mechanism, they instrument the programs where the MPI calls are replaced with modified calls that have extra arguments. These arguments provide information such as line number in the source code where the call was made, the MPI function name and its arguments. The information is stored in a database known as the Program Database (PDB). The process of checking is split in to two phases. In phase one, instrumentation of MPI programs is performed followed by their compilation. In phase two, execution of the instrumented MPI code under the control of the MPI-CHECK server takes place. The errors captured by MPI-CHECK as explained in [12, 11] are illustrated below:

* This work is supported by NSF CNS-0509379, SRC 2005-TJ-1318, and Microsoft HPC Institutes Program.

– Mismatch in argument type, kind, rank or number. Some checks can be done statically. For instance if tag, source, or destination arguments of MPI point to point routines are constants, then checking can be performed prior to run-time. For instance, if all sends have a tag of 2, then all receives with constant tags must also have a tag of 2.
– When the bounds of the message buffer exceed the allocated size. Size of the buffer can be calculated from the count and datatype arguments of the MPI routine. For dynamically allocated arrays, MPI-CHECK instruments the program by replacing malloc calls with special routines (macros with extra arguments). These routines essentially update the PDB with the file name, line number, start address of the message buffer and the block size information.
– Potential and real deadlock detection by creating dependency graphs from calls made for point-to-point or collective communication. It is an overly conservative approach. MPI-CHECK reports a time-out deadlock for cases where the dependency graph is not resolved in a user specified time.
– Negative message lengths.
– MPI calls before MPI_Init or after MPI_Finalize.
– Inconsistencies between the declared type of a message and its associated datatype argument.
– Actual arguments which violate the INTENT attribute.

MPI-CHECK intruments the MPI user programs to a large extent in order to check them. In contrast, UMPIRE and MARMOT need to relink the code in order to use the PMPI interface. MPI-CHECK, however, can check message buffer types and bounds and correct usage of the dynamic memory. These functionalities, which are absent in MPIDD and UMPIRE, come with an extra price. The MPI-CHECK method of checking invloves significant overhead of instrumenting the user code and building the PDB.

2.2 MARMOT

MARMOT is a tool to analyze MPI programs by trapping communication calls using the MPI profiling interface. It performs all argument verification like tags, communicators, ranks, etc. locally on the client side. MARMOT also detects potential and real deadlocks. However, the mechanism employed to detect deadlocks is different from that of MPI-CHECK. In MARMOT dependency graph is not created. Instead, a time-out mechanism is used to conclude the presence of a deadlock. Some of the checks performed by MARMOT as explained in [8–10, 7] are summarized below.

– More than one call to MPI_Init in an application.
– Any pending messages or active requests in any communicator at the time of MPI_Finalize.
– Checks the validity of the communicators used in calls. Also inspects the validity of datatype argument and for MPI_Type_struct and MPI_Type_hvector it also inspects if the count and block-length are greater than zero.

- For point-to-point and other collective communication calls made, it inspects the correctness of communicator, rank, tag, count, and datatype arguments. For instance, MARMOT will issue a warning if ranks or tags used are beyond valid ranges.
- Detects possible real deadlocks, using a time-out mechanism.
- MARMOT keeps track of construction, usage and destruction of all MPI resources such as communicators, groups, datatypes, etc. It checks if requests and other arguments are used correctly. For instance, MARMOT issues a warning if an active request is reused.
- Gives warnings if there are active non-freed requests left at MPI_Finalize.
- MARMOT also detects the erroneous use of MPI I/O (defined in MPI-2 standard) interface which may go undetected by the MPI implementation.
  - File Manipulation: MARMOT keeps track of all the opened files so as to be able to generate a warning when MPI_File_delete is called on files which are still open or there are still outstanding non-blocking requests or split collective operations when MPI_Finalize is called. Furthermore, MARMOT also generates a warning for groups (created by MPI_File_get_group) that are not freed before MPI_Finalize.
- One Sided Communication : With the kind of setup MARMOT has, it can check if there are any pending RMA (Remote Memory Accesses) function calls left when window is to be freed. MARMOT can also check the validity of RMA call arguments like target rank, window, displacement, datatype, etc.

MARMOT supports the complete MPI-1.2 standard; however, not all possible checks are performed by it. For instance, checks for data races are not performed. Furthermore, checks for safe reuse of buffer after the successful transmission of data are also currently not handled by MARMOT.

2.3 Intel Message Checker (IMC)

Intel Message Checker is an MPI correctness tool which has a centralized mechanism to detect errors/deadlocks like MARMOT and UMPIRE. However, UMPIRE and MARMOT are purely runtime checking tools. IMC, on the other hand is a post-mortem analyzer. The component of IMC called "TRACE collector", collects information of each MPI call in a trace file using a library file libVTmc.so which is similiar to the PMPI interface. This trace file is then analyzed by a checking engine after the execution. IMC offers several features of interest. Some of them are illustrated below:

- Extensive time stamping and event filtering facility.
- Support for Java tracing on platforms that support JVM Profiler Interface.
- Prints errors with lines tagged with process ranks. However, MPI applications that use process spawning and attachment are not supported in IMC.

The errors trapped by Intel message checker as explained in [5, 2] are summarized below:

– Mismatch of send and receive calls caused by incorrect specification of message sender or receiver.
– Potential or real deadlocks. In case of real deadlocks, a cyclic dependency is created from call stacks. Potential deadlocks are identified by the time-out mechanism. The wait time is a configurable entity.
– Errors caused by touching the send bufffers before the Isend operation is completed. However, the successful reporting of error is done in a lazy fashion. The error is reported later when the damage is detected.
– Different order of collective and reduction operation calls.
– Erroneous specification of different message lengths in matching send and receive operations.
– Mismatch of checksums of sent and received messages.
– Infinite loop or abnormal program termination in an MPI function call.
– Incorrect specification of sending and receiving data types. The Intel Trace Collector allows hash signatures to be computed piece-meal for the constituent data types. This helps in easier and faster matching of data types across send-receive calls.
– Memory leaks occur when communicator is freed and there are still outstanding buffered messages yet to be received. Trace Collector catches such a bug and gives a warning.
– Datatypes that are not freed or requests that are prematurely freed, are detected and a warning is generated.
– Errors caused by specifying overlapped receive buffers in different communication operations running in parallel.
– Prints a list of unfreed requests at the time of MPI_Finalize.

IMC can suffer from several impediments. The trace files generated can be large. Furthermore, the generation of trace files in the presence of an MPI error cannot be guaranteed, as the behaviour after an MPI error is implementation defined. Reading a memory location that is already under use, is not allowed. However, such a scenario remains undetected in ITC (Intel Trace Collector) as reads do not modify the buffers.

2.4 UMPIRE

UMPIRE, developed at LLNL, is another MPI program correctness checker. It is a tool that dynamically analyzes MPI programming errors using a profiling interface like MARMOT and MPIDD. It performs checking at two levels. First it checks at the local level where it uses all the task-local information to perform the checks. For instance, tests regarding the checksum on non-blocking send buffers can be carried out at this level. The second check is performed at a global level. It digs out more subtle errors like deadlocks, consistency errors, etc. at the global level. UMPIRE uses time-out mechanism and dependency graphs to detect deadlocks. Few of the errors that UMPIRE uncovers as explained in [15] are summarized below:

– Deadlocks caused due to blocking calls.

– Deadlocks involving spinloops over non-blocking completion calls.
– Ordering of collective communication calls within a communicator.
– Detects configuration dependent buffer deadlocks.
– Mismatch collective call operations.
– UMPIRE does extensive resource tracking. Consequently it is able to unearth resource leaks. For instance, applications can repeatedly create opaque objects without freeing them, leading to memory exaustion. Or there can be lost requests due to overwriting of request handles.
– Errant writes to send buffers before non-blocking sends are completed.

2.5 MPIDD

MPIDD, like UMPIRE has a central manager that traps all MPI calls using PMPI; however UMPIRE runs as a separate process and communicate using shared memory with different processes. MPIDD runs as another MPI process and the trapped information is sent to the central detector using MPI calls as explained in [6]. MPIDD is essentially a deadlock detection tool. It creates a dependency graph to figure out potential/real deadlocks. The detection algorithm is a Depth First Search for cycles in the dependency graph. The architecture of MPIDD suggests that it should be able to do all the argument verification tests that other tools perform. This can be done by the wrappers component of MPIDD.

A succinct comparative study of the above mentioned tools is presented in Table 1:

2.6 TotalView

TotalView is an industrial strength debugger. It is designed especially for complex multi-process or multi-threaded applications. It provides two decision points to users to control the execution of a program as explained in [1].

– Users can select an appropriate command.
– Users are provided with an option to decide upon the scope of the chosen command.

The execution control commands are: Go, Halt, Step, Kill, Next, etc. One can execute these control commands at the Group, Process or Thread scoping level.

1. **Group Scoping:** Executes the chosen command on all the processes that define that Group.
2. **Process Scoping:** Executes the chosen command on a single selected process. If the process has several threads, then the command influences all the threads owned by the process.

Tools	MPI-CHECK	MARMOT	UMPIRE	IMC	MPIDD
Deadlock Detection Method	T,D	T	T,D	T,D	D
Argument Verification	√	√	√	√	√
Use PMPI/Macros	Macros	PMPI	PMPI	PMPI	PMPI
Static/Runtime Checking	S,R	R	R	Trace	R
Check of buffer type and bounds	√	-	-	√	-
Data Race detection (Consistency Checks)	√	-	√	-	-
Buffer Reuse (Incomplete request)	-	-	√	√	-
Resource Leak Checks	-	√	√	√	-
Mismatch of Collective Operations	√	√	√	√	-
MBT	-	-	-	-	-

Table 1. Comparitive Study of MPI Debugging Tools. T: Time based deadlock detection; D: Dependency Graph based deadlock detection; S: Static; R: Runtime Trace: Analysis on Trace; MBT: Model Based Testing; - Not Available; √: Available

3. **Thread Scoping:** Executes the chosen command on a single specified thread of a Multi-threaded process. Thread scoping offers interesting subtleties. Normally, all processes in a thread stop when any one of them encounters a breakpoint. However, TotalView provides additional expressions that include intrinsic variables and builtin statements, using which thread-specific breakpoints can be implemented.

TotalView has a GUI showing message queues. The message queues capture the following information:

- Pending Receives
- Pending Sends
- Buffer contents can be viewed.
- Unexpected messages - messages sent to a process which did not post a matching receive.

These message queues can be thought of dependency graphs. Consequently, users can identify deadlocks by viewing the message queues.TotalView also displays MPI call arguments. Sanity of MPI call arguments can thus be validated against user's intentions.

TotalView provides support for hybrid codes where multiple threads can exist within an MPI process. When communication has to be performed, one thread makes the required MPI call. It is still not very clear whether programs that utilize MPI_THREAD_MULTIPLE functionality wherein multiple threads can make an MPI call, can be debugged under TotalView environment.

3 Debugging Support in MPI Libraries

Some popular MPI implementations also provide debugging facilities in the form of flags or separate profilers. For instance Open-MPI has a set of in-built debugging parameters for the MPI layer. Flags as explained in [3] are listed below:

3.1 OpenMPI

- **mpi_param_check:** If set to true, and when Open MPI is compiled with parameter checking enabled (the default), the parameters to each MPI function can be passed through a series of correctness checks. Problems such as passing illegal values (e.g., NULL or MPI_DATATYPE_NULL or other "bad" values) will be discovered at run time and an MPI exception will be invoked.
- **mpi_show_handle_leaks:** If set to true, OMPI will display lists of any MPI handles that were not freed before MPI_FINALIZE (e.g., communicators, datatypes, requests, etc.).
- **mpi_no_free_handles:** If set to true, do not actually free MPI object when their corresponding MPI "free" function (e.g., do not free communicators when MPI_COMM_FREE is invoked)is called. This can be helpful in tracking down any use of MPI handles after they have been freed.
- **mpi_show_mca_params:** If set to true, show a list of all MCA parameters and their values during MPI_INIT. This can be quite helpful for reproducability of MPI applications.
- **mpi_show_mca_params_file:** If set to a non-empty value, and if the value of mpi_show_mca_params is true, then output the list of MCA parameters to the filename value. If this parameter is an empty value, the list is sent to stderr.
- **mpi_keep_peer_hostnames:** If set to a true value, send the list of all hostnames involved in the MPI job to every process in the job. This can help the specificity of error messages that Open MPI emits if a problem occurs (i.e., Open MPI can display the name of the peer host that it was trying to communicate with), but it can somewhat slow down the startup of large-scale MPI jobs.
- **mpi_abort_delay:** Prints out an identifying message when MPI_ABORT is invoked showing the hostname and PID of the process that invoked MPI_ABORT, and then delay that many seconds before exiting. This allows a user to manually come in and attach a debugger when an error occurs.
- **mpi_abort_print_stack:** Prints out a stack trace when MPI_ABORT is invoked.

3.2 LAM-MPI

LAM-MPI implementation has a GUI-based tool support for MPI debugging and visualizing called as XMPI. Extensive book-keeping is done for running MPI applications. LAM-MPI is now succeeded by Open MPI. Following are the few key features provided by XMPI as explained in [4]:

– Runtime snapshot of MPI process synchronization.
– Runtime snapshot of unreceived message synchronization.
– Extensive detailing on information like communicator, datatype, tag, message and length.
– A highly integrated snapshot from communication trace timeline.
– A matrix display of unreceived message sources.
– Process group and datatype type map displays.

3.3 MPICH

In MPICH implementation, one can execute mpirun in a controlled environment of a debugger, say, TotalView. The option can be specified while executing mpirun with an extra flag -dgb set to the appropriate script.

4 Conclusions

MPI programs are notoriously difficult to debug. Tools such as MARMOT, UMPIRE, IMC, MPI-CHECK, and MPIDD are capable of detecting many errors in MPI programs. However, these tools do not guarantee to explore systematically all the execution interleavings of a program. For instance, MPI programs can have many sources of nondeterminism. There may be potential bugs lurking behind the cover of nondeterminism. It would simpily not suffice to explore just one execution interleaving of such a program. In [14], SPIN is extended with MPI non-blocking constructs to perform Model Checking based verification. One may argue to an extent that MPI-SPIN is reliable and expandable but the fact remains that modeling programs is a laborious and error prone task. Efforts, as mentioned in [13] are directed at removing the modeling overhead and to carry out similar checks as were performed in the tools discussed. It also guarantees that all interleavings of a program are being tested systematically. The disadvantage of this method include reduced execution speeds. In summary, no single method is superior and a variety of approaches need to be supported.

References

1. http://www.llnl.gov/computing/tutorials/totalview/.
2. http://support.intel.com/support/performancetools/cluster/analyzer/.
3. http://www.open-mpi.org/faq/?category=debugging.
4. http://www.lam-mpi.org/software/xmpi/.
5. Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 78–82, New York, NY, USA, 2005. ACM Press.
6. W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, 2006.

7. Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI I/O Analysis and Error Detection with MARMOT. In *PVM/MPI*, pages 242–250, 2004.
8. Bettina Krammer, Matthias S. Mller, and Michael M. Resch. MPI Application Development using the analysis tool MARMOT. In *ICCS*, volume LNCS, pages 464–471. Springer, 2004.
9. Bettina Krammer, Matthias S. Mller, and Michael M. Resch. Runtime Checking of MPI appications with MARMOT. 2005.
10. Bettina Krammer and Michael M. Resch. Correctness Checking of MPI One-Sided Communication using MARMOT. In *PVM/MPI*, pages 105–114, 2006.
11. Pavel Krusina and Glenn Luecke. MPI-CHECK for C/C++ MPI Programs. 2003.
12. G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003.
13. Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Robert Palmer, Rajeev Thakur, and William Gropp. Practical Model Checking Method for Verifying Correctness of MPI Programs. In *EuroPVM/MPI*, September 2007. Submitted.
14. Stephen F. Siegel. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2007.
15. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. pages 70–70, 2000.