MPI I/O Analysis and Error Detection with MARMOT

Bettina Krammer, Matthias S. Müller, Michael M. Resch

High Performance Computing Center Stuttgart
Allmandring 30, D-70550 Stuttgart, Germany
{krammer, mueller, resch}@hlrs.de

Abstract. The most frequently used part of MPI-2 is MPI I/O. Due to the complexity of parallel programming in general, and of handling parallel I/O in particular, there is a need for tools that support the application development process. There are many situations where incorrect usage of MPI by the application programmer can be automatically detected. In this paper we describe the MARMOT tool that uncovers some of these errors and we also analyze to what extent it is possible to do so for MPI I/O.
Keywords: MPI, Tools, Parallel Programming, Analysis, I/O

1 Introduction

The Message Passing Interface (MPI) is a widely used standard [12] for writing parallel programs. The availability of implementations on essentially all parallel platforms is probably the main reason for its popularity. Yet another reason is that the standard contains a large number of calls for solving standard parallel problems in a convenient and efficient manner. However, a drawback of this is that the MPI-1.2 standard, with its 129 calls, has a size and complexity that makes it possible to use the MPI API incorrectly.

Version 2 of the MPI standard [13] extends the functionality significantly, adding about 200 functions. This further increases the API's complexity and therefore the possibilities for introducing mistakes. Several vendors offer implementations of MPI-2 and there are already open source implementations [2, 3, 6], which cover at least some of the new features. Due to the demand for I/O support on the one hand and the availability of the free ROMIO implementation [15] on the other hand, MPI I/O is probably the most widely used part of MPI-2. The MPI I/O chapter describes 53 different calls.

2 Related Work

Finding errors in MPI programs is a difficult task that has been addressed in various ways by existing tools. The solutions can be roughly grouped into four different approaches: classical debuggers, special MPI libraries and other tools that may perform a run-time or post-mortem analysis.

1. Classical debuggers have been extended to address MPI programs. This is done by attaching the debugger to all processes of the MPI program. There are many parallel debuggers, among them the very well-known commercial debugger Totalview [1]. The freely available gdb debugger currently has no support for MPI; however, it may be used as a back-end debugger in conjunction with a front-end that supports MPI, e.g. mpigdb. Another example of such an approach is the commercial debugger DDT by the company Streamline Computing, or the non-freely available p2d2 [7, 14].
2. The second approach is to provide a debug version of the MPI library (e.g. mpich). This version is not only used to catch internal errors in the MPI library, but also to detect some incorrect uses of MPI calls by the programmer, e.g. a type mismatch between sending and receiving message pairs [5].
3. Another possibility is to develop tools and environments dedicated to finding problems within MPI applications. Three different message-checking tools are under active development at present: MPI-CHECK [11], Umpire [16] and MARMOT [8]. MPI-CHECK is currently restricted to Fortran code and performs argument type checking or finds problems such as deadlocks [11]. Like MARMOT, Umpire [16] uses the MPI profiling interface. These three tools all perform their analysis at runtime.
4. The fourth approach is to collect all information on MPI calls in a trace file, which can be analyzed by a separate tool after program execution [10]. A disadvantage with this approach is that such a trace file may be very large. However, the main problem is guaranteeing that the trace file is written in the presence of MPI errors, because the behavior after an MPI error is implementation defined.

The approach taken by tools such as MARMOT has the advantage that they are able to combine extensive checking with ease of use. However, since they offer specific support for MPI problems, they have to be extended to cover the new functionality.

3 Short Description of MARMOT

MARMOT uses the MPI profiling interface to intercept MPI calls and analyze them, as illustrated in Fig. 1. It issues warnings if the application relies on non-portable MPI constructs, and error messages if erroneous calls are made. MARMOT can be used with any standard-conforming MPI implementation and it may thus be deployed on any development platform available to the programmer. The tool has been tested on Linux Clusters with IA32/IA64 processors, IBM Regatta and NEC SX systems. It currently supports the MPI-1.2 interface, although not all possible checks (such as consistency checks) have been implemented yet. Functionality and performance tests have been performed with test suites, microbenchmarks and real applications [8, 9].

Local checks including verification of arguments such as tags, communicators, ranks, etc. are performed on the client side. For all tasks that cannot be

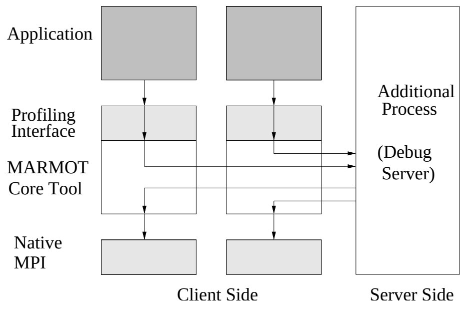

Fig. 1. Design of MARMOT.

handled within the context of a single MPI process, e.g. deadlock detection, an additional MPI process is added. Information is transferred between the original MPI processes and this additional process using MPI. An alternative approach would be to use a thread instead of an MPI process and shared memory communication instead of MPI [16]. The advantage of the approach taken here is that the MPI library does not need to be threadsafe.

In order to ensure that the additional debug process is transparent to the application, we map `MPI_COMM_WORLD` to a MARMOT communicator that contains only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they will also automatically exclude the debug server process. A similar strategy is used to map all other MPI resources, such as groups, datatypes, etc. to the corresponding MARMOT groups, datatypes, etc., and vice versa. Thus, MARMOT is able to keep track of the proper construction, usage and freeing of these resources.

4 Short Description of MPI I/O

Version 2 of MPI [13] provides a high-level interface to parallel I/O that enables the creation of portable applications. To support high performance I/O and a convenient interface for the user, MPI-2 not only includes Unix/Posix-like read and write operations, but also collective I/O operations and operations with strided memory and/or file access. Among other things, it also introduces the concept of split collective operations. I/O access is based on the concept of MPI predefined or derived datatypes.

The MPI-2 standard distinguishes between MPI calls for file manipulation, e. g. `MPI_File_open` (see Table 1), calls for file view such as `MPI_File_set_view` (see Table 2), calls for data access with many different read/write options (see Table 3) and calls for file pointer manipulation (see Table 4), calls for file interoperability such as `MPI_File_get_type_extent` and calls for consistency such as `MPI_File_sync`.

5 Possible Checks

File management errors are common, and MPI-2 therefore also includes error handling. According to the MPI standard, some of the errors related to these I/O calls are to be flagged in error classes: for example, errors related to the access mode in `MPI_File_open` belong to the class `MPI_ERR_AMODE`. However, the goal of MARMOT is not to detect errors of this kind. Our attention is focussed on the kind of errors that stem from erroneous use of the interface, and these are often not detected by the implementation itself. The question, of course, is to what extent such errors can be uncovered by verification tools such as MARMOT. Since the MPI standard also allows implementation dependent behavior for I/O calls, tools like MARMOT may help the user detect non-portable constructs.

5.1 Classification of I/O calls and general checks

In addition to their grouping according to functionality, several other criteria can be used to classify MPI calls. One possibility is to distinguish between collective and non-collective calls. As with all other collective routines, care has to be taken to call the collective functions in the same order on each process and, depending on the call, to provide the same value for certain arguments. For the numerous data access routines, two additional criteria are positioning and synchronism (see Table 3). The first of these distinguishes calls based on their use of explicit offsets, individual file pointers and shared file pointers. The second one groups the calls into blocking, nonblocking and split collective routines.

In the following subsections, we consider the possible checks for these classes of calls in more detail.

5.2 File Manipulation

MPI-2 contains several functions for creating, deleting and otherwise manipulating files. Table 1 classifies calls as either collective or non-collective. For the collective calls, the general checks described in Section 5.1 are applied. All query functions are non-collective. The only non-collective function that is not a query function is `MPI_File_delete`. If it is invoked when a process currently has the corresponding file open, the behavior of any accesses to the file (as well as of any outstanding accesses) is implementation dependent. MARMOT must therefore keep track of the names of all currently open files. For the sake of simplicity, we assume a global namespace when comparing the filenames on different processes.

There is not much that needs to be verified for query functions. One exception is the group that is created in calls to `MPI_File_get_group`. MARMOT issues a warning if the group is not freed before `MPI_Finalize` is called.

For `MPI_File_get_info` and `MPI_File_set_info` a portability warning will be issued if a key value is used that is not reserved by the standard. In addition, for the appropriate keys an error is flagged if the value provided is not the same on all processes. For the reserved keys, we can also check whether the value given has the appropriate data type (integer, string, comma separated list, ...).

Non-collective	*Collective*
MPI_File_delete	MPI_File_open
	MPI_File_close
	MPI_File_preallocate
MPI_File_get_size	MPI_File_set_size
MPI_File_get_group	
MPI_File_get_amode	
MPI_File_get_info	MPI_File_set_info

Table 1. Classification of File Manipulation functions.

The two calls `MPI_File_open` and `MPI_File_close` require more attention. The user is responsible for ensuring that all outstanding non-blocking requests and split collective operations have been completed by a process before it closes the file. MARMOT must therefore keep track of these operations and verify this condition when the file is closed. The user is also responsible for closing all files before `MPI_Finalize` is called. It is thus necessary to extend the checks already implemented for this call in MARMOT to determine whether there are any open files.

The access mode argument of `MPI_File_open` has strong implications. First, it is illegal to combine certain modes (e.g. `MPI_MODE_SEQUENTIAL` with `MPI_MODE_-RDWR`). Second, certain modes will make specific operations on that file illegal. If the file is marked as read only, any write operation is illegal; a read operation is illegal in write-only mode. If `MPI_MODE_SEQUENTIAL` was specified in `MPI_File_open`, the routines with explicit offset, the routines with individual pointers, `MPI_File_seek`, `MPI_File_seek_shared`, `MPI_File_get_position`, and `MPI_File_get_position_shared` must not be called.

5.3 File Views

Table 2 lists the calls related to setting/getting file views. For the collective call `MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)`, the values for the `datarep` argument and the extents of `etype` must be identical in all processes in the group, whereas the values for the `disp`, `filetype` and `info` arguments may vary. However, the range of values allowed for `disp` depends on the mode chosen in `MPI_File_open`. Calling `MPI_File_set_view` is erroneous if there are non-blocking requests or split collective operations on the file handle `fh` that have not completed.

Special attention has to be paid to verifying the correct usage of the datatypes `etype` and `filetype` in `MPI_File_set_view` and `MPI_File_get_view`. However these are based on MPI predefined or derived datatypes, so MARMOT can determine whether the datatypes are properly constructed, committed or freed in just the same way that it already handles similar MPI-1.2 calls. Absolute addresses must not be used in the construction of the `etype` and `filetype`.

noncollective	*collective*
`MPI_File_get_view`	`MPI_File_set_view`

Table 2. Classification of File View functions.

Moreover, the displacements in the typemap of the `filetype` are required to be non-negative and monotonically non-decreasing, the extent of any hole in the `filetype` must be a multiple of the `etype`'s extent and neither the `etype` nor the `filetype` is permitted to contain overlapping regions once the file is opened for writing.

5.4 Data Access

Table 3 shows that the MPI standard is very flexible with regard to data access, where it provides 30 different calls to meet the user's requirements. There are again collective and non-collective calls, which can be grouped into synchronous, i. e. blocking, and asynchronous, i. e. non-blocking and split collective calls, and according to positioning, i. e. into calls using explicit offsets, individual file pointers or shared file pointers.

positioning	**synchronism**	**coordination**	
		noncollective	*collective*
explicit offsets	*blocking*	`MPI_File_read_at`	`MPI_File_read_at_all`
		`MPI_File_write_at`	`MPI_File_write_at_all`
	nonblocking & split collective	`MPI_File_iread_at`	`MPI_File_read_at_all_begin`
			`MPI_File_read_at_all_end`
		`MPI_File_iwrite_at`	`MPI_File_write_at_all_begin`
			`MPI_File_write_at_all_end`
individual file pointers	*blocking*	`MPI_File_read`	`MPI_File_read_all`
		`MPI_File_write`	`MPI_File_write_all`
	nonblocking & split collective	`MPI_File_iread`	`MPI_File_read_all_begin`
			`MPI_File_read_all_end`
		`MPI_File_iwrite`	`MPI_File_write_all_begin`
			`MPI_File_write_all_end`
shared file pointer	*blocking*	`MPI_File_read_shared`	`MPI_File_read_ordered`
		`MPI_File_write_shared`	`MPI_File_write_ordered`
	nonblocking & split collective	`MPI_File_iread_shared`	`MPI_File_read_ordered_begin`
			`MPI_File_read_ordered_end`
		`MPI_File_iwrite_shared`	`MPI_File_write_ordered_begin`
			`MPI_File_write_ordered_end`

Table 3. Classification of Data Access routines of MPI-2 [13].

For all calls, the offset arguments must never be negative and the type signatures of the datatype argument must match the signature of some number of contiguous copies of the etype argument of the current view. The datatype for reading must not contain overlapping regions.

As buffering of data is implementation dependent, the only way to guarantee that data has been transferred to the storage device is to use the `MPI_File_sync routine`. Non-blocking I/O routines follow the naming convention of the MPI 1.2 standard and are named `MPI_File_iXXX`. Just as with the well-known MPI-1.2 non-blocking calls, it is necessary to insert calls like `MPI_Test` or `MPI_Wait` etc. that request completion of operations before an attempt is made to access the data. The split collective calls are named `MPI_File_XXX_begin` and `MPI_File_XXX_end`, respectively. They must be inserted in matching begin/end pairs of calls; no other collective I/O operation is permitted on their file handle between them. A file handle may have at most one active split collective operation at any time in an MPI process.

The data access routines that accept explicit offsets are named `MPI_File_XXX_at` when they are non-collective and `MPI_File_XXX_at_all_YYY` when they are collective. The routines with shared pointers are named `MPI_File_XXX_shared` when they are non-collective and `MPI_File_XXX_ordered_YYY` when they are collective. They may only be used when all processes share the same file view.

File Pointer Manipulation Table 4 gives an overview of routines for pointer manipulation. With the exception of `MPI_File_get_byte_offset`, it is erroneous to use them if `MPI_MODE_SEQUENTIAL` is set. The seek functions are permitted to have a negative offset, i. e. to seek backwards; however, the user must ensure that the seek operation does not reach a negative position in the view.

individual file pointer	*shared file pointer*	*view-relative offset*
`MPI_File_seek`	`MPI_File_seek_shared`	
`MPI_File_get_position`	`MPI_File_get_position_shared`	
		`MPI_File_get_byte_offset`

Table 4. Classification of File Pointer Manipulation functions.

5.5 File Interoperability

MPI-2 guarantees that a file written by one program can be read by another program, independent of the number of processes used. However, this is only true if both programs make use of the same MPI implementation. In order to allow file sharing between different implementations, several conditions have to be fulfilled. First, the data has to be written using the data format “external 32”. This must be specified in the `MPI_File_open` call. Second, the datatypes for etypes and filetypes have to be *portable*. A datatype is *portable*, if it is a predefined datatype, or is derived from a predefined datatype using only the type constructors `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_indexed`, `MPI_Type_indexed_block`, `MPI_Type_create_subarray`, `MPI_Type_dup`, and `MPI_Type_Create_Darray`. Other datatypes may contain platform-dependent byte displacements for padding.

5.6 Consistency

The consistency semantics of MPI-2 describes the validity and outcome of *conflicting* data access operations, which occur when two data access operations overlap and at least one of them is a write access. The user is able to control the consistency behavior by setting the function `MPI_File_set_atomicity` when desired. If atomic mode is thus set, MPI guarantees that the data written by one process can be read immediately by another process. A race condition may exist only if the order of the two data access operations is not defined. More care has to be taken in the default, non-atomic mode, where the user is responsible for ensuring that no write *sequence* in a process is concurrent with any other *sequence* in any other process. In this context, a *sequence* is a set of file operations bracketed by any pair of the functions `MPI_File_sync`, `MPI_File_open`, and `MPI_File_close` [4].

To find potential race conditions, MARMOT has to analyze which data is written in which order by all participating processes. For all conflicting data access operations, it is necessary to verify that they are separated by an appropriate synchronization point. We are currently investigating strategies for doing this in a portable and efficient way within the limitations imposed by the design of MARMOT.

There are other potential problems related to the contents of the file. For example, the value of data in new regions created by a call to `MPI_File_set_size` is undefined, and it is therefore erroneous to read such data before writing it.

6 Conclusions and Future Work

In this paper, we analyzed the MPI I/O Interface with regard to potential errors that can be made by the user. Altogether more than 50 different error types have been identified. We found that in most cases these errors can be detected by tools like MARMOT following the approach taken for MPI-1. The rules for data consistency are an exception, and new solutions have to be considered for them. This is currently work in progress. Since the implementation is not yet completed, another open question is the performance impact MARMOT will have on the application.

7 Acknowledgments

The development of MARMOT is partially supported by the European Commissionion through the IST-2001-32243 project “CrossGrid”.

References

1. WWW. http://www.etnus.com/Products/TotalView.
2. Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
3. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
4. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, 1999.
5. William D. Gropp. Runtime Checking of Datatype Signatures in MPI. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908 of *Lecture Notes In Computer Science*, pages 160–167. Springer, Balatonfüred, Lake Balaton, Hungary, Sept. 2000. 7th European PVM/MPI Users' Group Meeting.
6. William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.
7. Robert Hood. Debugging Computational Grid Programs with the Portable Parallel/Distributed Debugger (p2d2). In *The NASA HPCC Annual Report for 1999.* NASA, 1999. http://hpcc.arc.nasa.gov:80/reports/report99/99index.htm.
8. Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI Analysis and Checking Tool. In *Proceedings of PARCO 2003*, Dresden, Germany, September 2003.
9. Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI Application Development Using the Analysis Tool MARMOT. In M. Bubak, G. D. van Albada, P. M. Sloot, and J. J. Dongarra, editors, *Computational Science — ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471, Krakow, Poland, June 2004. Springer.
10. D. Kranzlmueller, Ch. Schaubschlaeger, and J. Volkert. A Brief Overview of the MAD Debugging Activities. In *Fourth International Workshop on Automated Debugging (AADEBUG 2000)*, Munich, 2000.
11. Glenn Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock Detection in MPI Programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.
12. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. http://www.mpi-forum.org.
13. Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org.
14. Sue Reynolds. System software makes it easy. *Insights Magazine*, 2000. NASA, http://hpcc.arc.nasa.gov:80/insights/vol12.
15. Rajeev Thakur, Robert Ross, Ewing Lusk, and William Gropp. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Argonne National Laboratory, January 2002. Technical Memorandum ANL/MCS-TM-234.
16. J.S. Vetter and B.R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference (SC 2000)*, Dallas, Texas, 2000. ACM/IEEE. CD-ROM.