Collective Error Detection for MPI Collective Operations[*]

Chris Falzone
University of Pennsylvania at Edinboro
Edinboro, Pennsylvania 16444

Anthony Chan, Ewing Lusk, and William Gropp
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, Illinois 60439

Abstract. An MPI profiling library is a standard mechanism for intercepting MPI calls by applications. Profiling libraries are so named because they are commonly used to gather performance data on MPI programs. Here we present a profiling library whose purpose is to detect user errors in the use of MPI's collective operations. While some errors can be detected locally (by a single process), other errors involving the consistency of arguments passed to MPI collective functions must be tested for in a collective fashion. While the idea of using such a profiling library does not originate here, we take the idea further than it has been taken before (we detect more errors) and offer an open-source library that can be used with any MPI implementation. We describe the tests carried out, provide some details of the implementation, illustrate the usage of the library, and present performance tests.

Keywords: MPI, collective, errors, datatype, hashing

1 Introduction

Detection and reporting of user errors are important components of any software system. All high-quality implementations of the Message Passing Interface (MPI) Standard [6, 2] provide for runtime checking of arguments passed to MPI functions to ensure that they are appropriate and will not cause the function to behave unexpectedly or even cause the application to crash. The MPI collective operations, however, present a special problem: they are called in a coordinated way by multiple processes, and the Standard mandates (and common sense requires) that the arguments passed on each process be consistent with the arguments passed on the other processes. Perhaps the simplest example is the case of `MPI_Bcast`:

[*] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, SciDAC Program, under Contract W-31-109-ENG-38.

```
MPI_Bcast(buff, count, datatype, root, communicator)
```

in which each process must pass the same value for root. In this case, "consistent" means "identical," but more complex types of consistency exist. No single process by itself can detect inconsistency; the error check itself must be a collective operation.

Fortunately, the MPI profiling interface allows one to intercept MPI calls and carry out such a collective check before carrying out the "real" collective operation specified by the application. In the case of an error, the error can be reported in the way specified by the MPI Standard, still independently of the underlying MPI implementation, and without access to its source code.

The profiling library we describe here is freely available as part of the MPICH2 MPI-2 implementation [4]. Since the library is implemented entirely as an MPI profiling library, however, it can be used with any MPI implementation. For example, we have tested it with the IBM MPI implementation for Blue Gene/L [1].

The idea of using the MPI profiling library for this purpose was first presented by Jesper Träff and Josef Worringen in [7], where they describe the error-checking approach taken in the NEC MPI implementation, in which even local checks are done in the profiling library, some collective checks are done portably in a profiling library as we describe here, and some are done by making NEC-specific calls into the proprietary MPI implementation layer. The datatype consistency check in [7] is only partial, however; the sizes of communication buffers are checked, but not the details of the datatype arguments, where there is considerable room for user error. Moreover, the consistency requirements are not on the datatypes themselves, but on the datatype *signatures*; we say more about this in Section 3.1.

To address this area, we use a "datatype signature hashing" mechanism, devised by William Gropp in [3]. He describes there a family of algorithms that can be used to assign a small amount of data to an MPI datatype signature in such a way that only small messages need to be sent in order to catch most user errors involving datatype arguments to MPI collective functions. In this paper we describe a specific implementation of datatype signature hashing and present an MPI profiling library that uses datatype signature hashing to carry out more thorough error checking than is done in [7]. Since extra work (to calculate the hash) is involved, we also present some simple performance measurements, although one can of course use this profiling library just during application development and remove it for production use.

In Section 2 we describe the nature and scope of the error checks we carry out and compare our approach with that in [7]. Section 3 lays out details of our implementation, including our implementation of the hashing algorithm given in [3]; we also present example output. In Section 4 we present some performance measurements. Section 5 summarizes the work and describes future directions.

2 Scope of Checks

In this section we describe the error checking carried out by our profiling library. We give definitions of each check and provide a table associating the checks made on the arguments of each collective MPI function with that function. We also compare our collective error checking with that described in [7].

2.1 Definitions of Checks

The error checks for each MPI(-2) collective function are shown in Table 1. The following checks are made:

call checks that all processes in the communicator have called the same collective function in a given event, thus guarding against the error of calling `MPI_Reduce` on some processes, for example, and `MPI_Allreduce` on others.

root means that the same argument was passed for the `root` argument on all processes.

datatype refers to datatype signature consistency. This is explained further in Section 3.1.

`MPI_IN_PLACE` means that every processes either did or did not provide `MPI_IN_PLACE` instead of a buffer.

op checks operation consistency, for collective operations that include computations. For example, each process in a call to `MPI_Reduce` must provide the same operation.

local leader and tag test consistency of the `local_leader` and tag arguments. They are used only for `MPI_Intercomm_create`.

high/low tests consistency of the `high` argument. It is used only for `MPI_Intercomm_merge`.

dims checks for `dims` consistency across the communicator.

graph tests the consistency of the graph supplied by the arguments to `MPI_Graph_create` and `MPI_Graph_map`.

amode tests for `amode` consistency across the communicator for the function `MPI_File_open`.

size, datarep, and flag verify consistency on these arguments, respectively.

etype is an additional datatype signature check for MPI file operations.

order checks for the collective file read and write functions, therefore ensuring the proper order of the operations. According to the MPI Standard [2], a `begin` operation must follow an `end` operation, with no other collective file functions in between.

2.2 Comparison with Previous Work

This work can be viewed as an extension of the NEC implementation of collective error checking via a profiling library presented in [7]. The largest difference between that work and this is that we incorporate the datatype signature hashing mechanism described in Section 3, which makes this paper also an extension of [3], where the hashing mechanism is described but not implemented. In the NEC implementation, only message lengths, rather than datatype signatures, are checked. We do not check length consistency since it would be incorrect to do so in a heterogeneous environment. We also implement our library as a

Table 1. Checks performed on MPI functions

MPI_Barrier	call
MPI_Bcast	call, root, datatype
MPI_Gather	call, root, datatype
MPI_Gatherv	call, root, datatype
MPI_Scatter	call, root, datatype
MPI_Scatterv	call, root, datatype
MPI_Allgather	call, datatype, MPI_IN_PLACE
MPI_Allgatherv	call, datatype, MPI_IN_PLACE
MPI_Alltoall	call, datatype
MPI_Alltoallw	call, datatype
MPI_Alltoallv	call, datatype
MPI_Reduce	call, datatype, op
MPI_AllReduce	call, datatype, op, MPI_IN_PLACE
MPI_Reduce_scatter	call, datatype, op, MPI_IN_PLACE
MPI_Scan	call, datatype, op
MPI_Exscan	call, datatype, op
MPI_Comm_dup	call
MPI_Comm_create	call
MPI_Comm_split	call
MPI_Intercomm_create	call, local leader, tag
MPI_Intercomm_merge	call, high/low
MPI_Carte_create	call, dims
MPI_Carte_map	call, dims
MPI_Graph_create	call, graph
MPI_Graph_map	call, graph
MPI_Comm_spawn	call, root
MPI_Comm_spawn_multiple	call, root
MPI_Comm_connect	call, root
MPI_Comm_disconnect	call
MPI_Win_create	call
MPI_Win_fence	call
MPI_File_open	call, amode
MPI_File_set_size	call, size
MPI_File_set_view	call, datarep, etype
MPI_File_set_automicity	call, flag
MPI_File_preallocate	call, size
MPI_File_seek_shared	call, order
MPI_File_read_all_begin	call, order
MPI_File_read_all	call, order
MPI_File_read_all_end	call, order
MPI_File_read_at_all_begin	call, order
MPI_File_read_at_all	call, order
MPI_File_read_at_all_end	call, order
MPI_File_read_ordered_begin	call, order
MPI_File_read_ordered	call, order
MPI_File_read_ordered_end	call, order
MPI_File_write_all_begin	call, order
MPI_File_write_all	call, order
MPI_File_write_all_end	call, order
MPI_File_write_at_all_begin	call, order
MPI_File_write_at_all	call, order
MPI_File_write_at_all_end	call, order
MPI_File_write_ordered_begin	call, order
MPI_File_write_ordered	call, order
MPI_File_write_ordered_end	call, order

pure profiling library. This precludes us from doing some MPI-implementation-dependent checks that are provided in the NEC implementation, but allows our library to be used with any MPI implementation. In this paper we also present some performance tests, showing that the overhead, even of our unoptimized version, is acceptable. Finally, the library described here is freely available.

3 Implementation

In this section we describe our implementation of the datatype signature matching presented in [3]. We also show how we use datatype signatures in coordination with other checks on collective operation arguments.

3.1 Datatype Signature Matching

An MPI datatype signature for n different datatypes $type_i$ is defined to ignore the relative displacement among the datatypes as follows [6]:

$$Typesig = \{type_1, type_2, \ldots, type_n\}. \tag{1}$$

A datatype hashing mechanism was proposed in [3] to allow efficient comparison of datatype signature over any MPI collective call. Essentially, it involves comparison of a tuple (α, n), where α is the hash value and n is the total number of basic predefined datatypes contained in it. A tuple of form $(\alpha, 1)$ is assigned for each basic MPI predefined datatype (e.g. `MPI_INT`), where α is some chosen hash value. The tuple for an MPI derived datatype consisting of n basic predefined datatypes $(\alpha, 1)$ becomes (α, n). The combined tuple of any two MPI derived datatypes, (α, n) and (β, m), is computed based on the hashing function:

$$(\alpha, n) \oplus (\beta, m) \equiv (\alpha \wedge (\beta \triangleleft n), n + m), \tag{2}$$

where $\wedge$ is the bitwise exclusive or (xor) operator, $\triangleleft$ is the circular left shift operator, and $+$ is the integer addition operator. The noncommutative nature of the operator $\oplus$ in equation (2) guarantees the ordered requirement in datatype signature definition [1].

One of the obvious potential hash collisions is caused by the $\triangleleft$ operator's circular shift by 1 bit. Let us say there are four basic predefined datatypes identified by tuples $(\alpha, 1)$, $(\beta, 1)$, $(\gamma, 1)$, and $(\lambda, 1)$ and that $\alpha = \lambda \triangleleft 1$ and $\gamma = \beta \triangleleft 1$. For $n = m = 1$ in equation (2), we have

$$\begin{aligned} (\alpha, 1) \oplus (\beta, 1) &\equiv (\alpha \wedge (\beta \triangleleft 1), 2) \\ &\equiv ((\beta \triangleleft 1) \wedge \alpha, 2) \\ &\equiv (\gamma \wedge (\lambda \triangleleft 1), 2) \\ &\equiv (\gamma, 1) \oplus (\lambda, 1), \end{aligned} \tag{3}$$

If the hash values for all basic predefined datatypes are assigned consecutive integers, there will be roughly a 25 percent collision rate as indicated by equation (3). The simplest solution for avoiding this problem is to choose consecutive odd integers for all the basic predefined datatypes. Also, there are composite predefined datatypes in the MPI standard (e.g., MPI_FLOAT_INT), whose hash values are chosen according to equation (2) such that

$$MPI\_FLOAT\_INT = MPI\_FLOAT \oplus MPI\_INT$$

.

The tuples for MPI_UB and MPI_LB are assigned $(0,0)$, so they are essentially ignored. MPI_PACKED is a special case, as described in [3].

More complicated derived datatypes are decoded by using MPI_Type_get_envelope() and MPI_Type_get_content() and their hashed tuple computed during the process.

3.2 Collective Datatype Checking

Because of the different comunication patterns and the different specifications of the send and receive datatypes in various MPI collective calls, a uniform method of collective datatype checking is not attainable. Hence five different procedures are used to validate the datatype consistency of the collectives. The goal here is to provide error messages at the process where the erroneous argument has been passed. To achieve that goal, we tailor each procedure to match the communication pattern of the profiled collective call. For convenience, each procedure is named by one of the MPI collective routines being profiled.

Collective Scatter Check

1. At the root, compute the sender's datatype hash tuple.
2. Use PMPI_Bcast() to broadcast the hash tuple from the root to other processes.
3. At each process, compute the receiver's datatype hash tuple locally and compare it to the hash tuple received from the root.

A special case of the collective scatter check is when the sender's datatype signature is the same as the receiver's. This special case can be refered to as a collective bcast check. It is used in the profiled version of MPI_Bcast(), MPI_Reduce(), MPI_Allreduce(), MPI_Reduce_scatter(), MPI_Scan(), and MPI_Exscan().

The general collective scatter check is used in the profiled version of MPI_Gather() and MPI_Scatter().

Collective Scatterv Check

1. At the root, compute the vector of the sender's datatype hash tuples.
2. Use PMPI_Scatter() to broadcast the vector of hash tuples from the root to the corresponding process in the communicator.
3. At each process, compute the receiver's datatype hash tuple locally and compare it to the hash tuple received from the root.

The collective scatterv check is used in the profiled version of MPI_Gatherv() and MPI_Scatterv().

Collective Allgather Check

1. At each process, compute the sender’s datatype hash tuple.
2. Use `PMPI_Allgather()` to gather other senders’ datatype hash tuples as a local hash tuple vector.
3. At each process, compute the receiver’s datatype hash tuple locally, and compare it to each element of the hash tuple vector received.

The collective allgather check is used in the profiled version of `MPI_Allgather()` and `MPI_Alltoall()`.

Collective Allgatherv Check

1. At each process, compute the sender’s datatype hash tuple.
2. Use `PMPI_Allgather()` to gather other senders’ datatype hash tuples as a local hash tuple vector.
3. At each process, compute the vector of the receiver’s datatype hash tuples locally, and compare this local hash tuple vector to the hash tuple vector received element by element.

The collective allgatherv check is used in the profiled version of `MPI_Allgatherv()`.

Collective Alltoallv/Alltoallw Check

1. At each process, compute the vector of the sender’s datatype hash tuples.
2. Use `PMPI_Alltoall()` to gather other senders’ datatype hash tuples as a local hash tuple vector.
3. At each process, compute the vector of the receiver’s datatype hash tuples locally, and compare this local hash tuple vector to the hash tuple vector received element by element.

The difference between collective alltoallv and collective alltoallw checks is that alltoallw is more general than alltoallv; in other words, alltoallw accepts a vector of `MPI_Datatype` in both the sender and receiver.

The collective alltoallv check is used in the profiled version of `MPI_Alltoallv()`, and the collective alltoallw check is used in the profiled version of `MPI_Alltoallw()`.

3.3 Example Output

In this section we illustrate what the user sees (on `stderr`) when a collective call is invoked incorrectly.

Example 1. In this example, run with five processes, all but the last process call `MPI_Bcast`; the last process calls `MPI_Barrier`.

```
aborting job:
Fatal error in MPI_Comm_call_errhandler:

VALIDATE BARRIER (Rank 4) --> Collective call (BARRIER) is Inconsistent with Rank 0’s (BCAST).

rank 4 in job 204  ilsig.mcs.anl.gov_32779   caused collective abort of all ranks
  exit status of rank 4: return code 13
```

Example 2. In this example, run with five processes, all but the last process give `MPI_CHAR`; but the last process gives `MPI_INT`.

```
aborting job:
Fatal error in MPI_Comm_call_errhandler:

VALIDATE BCAST (Rank 4) --> Datatype Signature used is Inconsistent with Rank 0s.

rank 4 in job 205  ilsig.mcs.anl.gov_32779   caused collective abort of all ranks
  exit status of rank 4: return code 13
```

Example 3. In this example, run with five processes, all but the last process use 0 as the `root` parameter; the last process uses its rank.

```
aborting job:
Fatal error in MPI_Comm_call_errhandler:

VALIDATE BCAST (Rank 4) --> Root Parameter (4) is inconsistent with rank 0 (0)

rank 4 in job 207  ilsig.mcs.anl.gov_32779   caused collective abort of all ranks
  exit status of rank 4: return code 13
```

4 Experiences

Here we describe our experiences with the collective error checking profiling library in the areas of usage, porting, and performance.

After preliminary debugging tests gave us some confidence that hte library was functioning correctly, we applied it to the collective part of the MPICH2 test suite. This set of tests consists of approximately 70 programs, many of which carry out multiple tests, that test the MPI-1 and MPI-2 Standard compliance for MPICH2. We were surprised (and strangely satisfied, although simultaneously embarrassed) to find an error in one of our test programs. One case in one test expected a datatype of one `MPI_INT` to match a vector of `sizeof(int)` `MPI_BYTE`s. This is incorrect, although MPICH2 allowed the program to execute.

To test a real application, we linked FLASH [5], a large astrophysics application utilizing many collective operations, with the profiling library and ran one of its model problems. In this case no errors were found.

A profiling library should be automatically portable among MPI implementations. The library we describe here was developed under MPICH2. To check for portability and to obtain separate performance measurements, we also used it in conjunction with IBM's MPI for BlueGene/L [1], without encountering any problems.

We carried out performance tests on two platforms. On BlueGene/L, the collective and datatype checking library and the test codes were compiled with `xlc` with `-O3` and linked with IBM's MPI implementation available on BlueGene/L.

The performance of the collective and datatype checking library of a 32-process job is listed in Table 2, where each test case is linked with and without the collective and datatype checking library.

Similarly on a IA32 Linux cluster, the collective and datatype checking library and the test codes were compiled with `gcc` with `-O3` and linked with MPICH2-1.0.1. The performance results of the library are tabulated in Table 3.

Table 2. The maximum time taken (in seconds) among all the processes in a 32-process job on BlueGene/L. Count is the number of MPI_Doubles in the datatype, and N_{itr} refers to the number of times the MPI collective routine was called in the test. The underlined digits indicates that the corresponding digit could be less in one of the processes involved.

Test Name	count×N_{itr}	No CollChk	With CollChk
MPI_Bcast	1×10	$0.00026\underline{9}$	$0.00288\underline{0}$
MPI_Bcast	1K×10	$0.00050\underline{5}$	$0.00386\underline{1}$
MPI_Bcast	128K×10	$0.03142\underline{6}$	$0.13513\underline{8}$
MPI_Allreduce	1×1	$0.00003\underline{9}$	$0.00031\underline{8}$
MPI_Allreduce	1K×1	$0.00023\underline{3}$	$0.0005\underline{86}$
MPI_Allreduce	128K×1	$0.02\underline{2263}$	$0.03\underline{2420}$
MPI_Alltoallv	1×1	$0.00004\underline{3}$	$0.00025\underline{2}$
MPI_Alltoallv	1K×1	$0.0001\underline{68}$	$0.0005\underline{40}$
MPI_Alltoallv	128K×1	$0.01\underline{5357}$	$0.035\underline{828}$

Table 3. The maximum time taken (in seconds) on a 8-process job on Jazz, an IA32 Linux cluster. Count is the number of MPI_Double in the datatype, and N_{itr} refers to the number of times the MPI collective routine has been called in the test.

Test Name	count×N_{itr}	No CollChk	With CollChk
MPI_Bcast	1×10	$0.0\underline{34332}$	$0.093\underline{795}$
MPI_Bcast	1K×10	$0.0\underline{22218}$	$0.06\underline{9825}$
MPI_Bcast	128K×10	$1.704\underline{995}$	$1.7\underline{30708}$
MPI_Allreduce	1×1	$0.000\underline{423}$	$0.006\underline{863}$
MPI_Allreduce	1K×1	$0.003\underline{896}$	$0.005\underline{795}$
MPI_Allreduce	128K×1	$0.\underline{233541}$	$0.2\underline{36214}$
MPI_Alltoallv	1×1	$0.000\underline{320}$	$0.009\underline{682}$
MPI_Alltoallv	1K×1	$0.00\underline{2415}$	$0.003\underline{593}$
MPI_Alltoallv	128K×1	$0.\underline{271676}$	$0.\underline{355068}$

Both Tables 2 and 3 indicate that the cost of the collective and datatype checking library diminishes as the size of the datatype increases. The cost of collective checking can be significant when the datatype size is small. One would like the performance of such a library to be good enough that it is convenient to use and does not affect the general behavior of the application it is being applied to. On the other hand, performance is not absolutely critical, since it is basically a debug-time tool and is likely not to be used when the application is in production. Our implementation at this stage does still present a number of opportunities for optimization, but we have found it highly usable.

5 Summary

We have presented an effective technique to safeguard users from making easy-to-make but hard-to-find mistakes that often lead to deadlock, incorrect results, or worse. The technique is portable and available for all MPI implementations. We have also presented a method for checking datatype signatures in collective operations. We intend to extend the datatype hashing mechanism to point-to-point operations as well.

This profiling library is freely available as part of MPICH2 [4] in the MPE subdirectory, along with other profiling libraries.

References

1. G. Almási, C. Archer, J. G. Casta nos, M. Gupta, X. Martorell, J. E. Moreira, W. D. Gropp, S. Rus, and B. Toonen. MPI on BlueGene/L: Designing an efficient general purpose messaging solution for a large cellular system. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number LNCS2840 in Lecture Notes in Computer Science, pages 352–361. Springer Verlag, 2003.
2. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions.* MIT Press, Cambridge, MA, 1998.
3. William D. Gropp. Runtime checking of datatype signatures in MPI. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 160–167, September 2000.
4. MPICH2 Web page. `http://www.mcs.anl.gov/mpi/mpich2`.
5. R. Rosner, A. Calder, J. Dursi, B. Fryxell, D. Q. Lamb, J. C. Niemeyer, K. Olson, P. Ricker, F. X. Timmes, J. W. Truran, H. Tufo, Y. Young, M. Zingale, E. Lusk, and R. Stevens. Flash code: Studying astrophysical thermonuclear flashes. *Computing in Science and Engineering*, 2(2):33, 2000.
6. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core,* 2nd edition. MIT Press, Cambridge, MA, 1998.
7. Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. (submitted).

The following notice is required by DOE as part of the manuscript submission; it would not appear in the final version of the paper.