A Portable Method for Finding User Errors in the Usage of MPI Collective Operations*

Chris Falzone
University of Pennsylvania at Edinboro
Edinboro, Pennsylvania 16444

Anthony Chan
University of Chicago
Chicago, IL 60615

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, including those involving datatype inconsistencies) and present 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

One measure of the quality of a software system is its ability to identify user errors at an early stage and provide sufficient information for the user to correct the error. To this end, all high-quality implementations of the Message Passing Interface (MPI) Standard [10, 4] 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

*This work was partially 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, under Contract DE-AC02-06CH11357. It was also partially supported by the Department of Energy ASC/Alliances Center for Astrophysical Thermonuclear Flashes at the University of Chicago, Contract B523820.

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`:

```
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 MPI profiling interface consists primarily of a requirement in the MPI Standard that all MPI functions (normally called by application programs using function names employing the "`MPI_`" prefix) be also callable with a name using the corresponding "`PMPI_`" prefix as well. This simple requirement allows a library writer, without access to the source code of the MPI implementation, to intercept all MPI calls by interposing at link time his own library of "`MPI_`" functions, which carry out some useful specialized operation devised by the library writer. The specialized functions call "`PMPI`" functions required by the application. Such a special library of "`MPI_`" functions is called a *profiling library*.

The profiling library we describe here is freely available as part of the MPICH2 MPI-2 implementation [7]. 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's MPI implementation for Blue Gene/L [1] and OpenMPI [8].

The idea of using the MPI profiling library for this purpose was first presented by Jesper Träff and Joachim Worringen in [11], 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 [11] 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 [5]. 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 [11]. 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.

An earlier, compact version of this paper appeared in [2]. In this revision we describe the idea of MPI profiling library in general (above) and report on further portability and performance experiments, including tests using OpenMPI. We have expanded the section on datatype hashing to increase clarity and reference other work that has appeared since the original version of this paper was written. In addition, we describe how the library is used as a part of the MPE library distributed with MPICH, and provide more examples of usage and sample output. We have also added a section describing future work.

In Section 2 we describe the nature and scope of the error checks we carry out and compare our approach with that in [11]. Section 3 lays out details of our implementation, including our implementation of the hashing algorithm given in [5]; we also describe how usage of the library is made convenient in the MPICH2 environment and show some example output. In Section 4 we present some performance measurements. Section 5 describes areas in which we intend to extend and improve the capabilities of the library. Section 6 summarizes the paper.

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 [11].

2.1 Definitions of Checks

The error checks for each MPI collective function are shown in Tables 1 and 2. There are five categories of tests; these are described below:

These checks apply to most collective routines:

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 test ensures both that the counts and the datatypes are consistent in the collective call. This is explained further in Section 3.1.

The following applies only to collective computation and some collective communication routines:

`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.

The following apply only to intercommunicator create and merge:

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`.

The following apply only to collective topology routines:

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`.

The following apply only to collective I/O operations:

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 [4], a `begin` operation must follow an `end` operation, with no other collective file functions in between.

One check that is not included in this table is that the same communicator is passed by each of the processes making the collective call, and that all processes in the communicator are making the call. An approach to handling this is proposed in Section 5.

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 [11]. 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 [5], 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 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 [5]. We also show how we use datatype signatures in coordination with other checks on collective operation arguments.

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_Cart_create	call, dims
MPI_Cart_map	call, dims
MPI_Graph_create	call, graph
MPI_Graph_map	call, graph

Table 1: Checks performed on MPI-1 functions

3.1 Datatype Signature Matching

In MPI, the amount of data to send or receive is described by the tuple (`count`, `datatype`) that indicates `count` copies of the `datatype` are used to describe the data. Because datatypes in MPI may be built from combinations of basic datatypes (such as `MPI_INT`), the number of basic items that are communicated cannot be determined from the `count` alone. Instead, the number and type of basic types must be compared (basic types are just the MPI counterparts of the basic types in the language; e.g., `MPI_INT` corresponds to the C `int` type). The basic types within an MPI datatype are called the *MPI datatype signature* of that MPI datatype. More formally, an MPI datatype signature for a datatype constructed from n different basic datatypes $type_i$ is simply

$$Typesig = \{type_1, type_2, \ldots, type_n\}. \quad (1)$$

(In MPI, the *type map* combines the information on the basic types and the displacement in memory that is to be used in reading or writing the data from or to memory; the displacement information is not relevant in our checks.)

A datatype hashing mechanism was proposed in [5] to allow efficient comparison of datatype signature over any MPI collective call. Essentially, it involves comparison of

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

Table 2: Checks performed on MPI-2 functions

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 (Equation 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 [5].

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. Computing the hash value in the case where there "count" value is greater than one exploits the $\log(n)$ algorithm described in [5].

While the simple hash function above is adequate for detecting mismatches of the basic datatypes, more sophisticated hash functions can be used that are more accurate for common derived datatypes. For example, see [6] where computationally efficient hash functions based on Galois Fields are described and tested against a collection of MPI derived datatypes.

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 Usage

In this section we illustrate how users enable the collective and datatype checking library when linking their applications in the case of MPICH; for other MPI implementations, they would follow the appropriate procedures for linking in profiling libraries.

The collective and datatype checking library is freely available as part of MPICH2 [7] in the MPE subdirectory, along with other profiling libraries. MPE provides convenient compiler wrappers to allow for easy access of different profiling libraries. When MPE is installed separately with non-MPICH2 MPI implementation, e.g. IBM's MPI for BlueGene/L or OpenMPI, two compiler wrappers, "mpecc" for C program and "mpefc" for Fortran program, are created. Available MPE profiling options for "mpecc" and "mpefc" are as follows; the option `-mpicheck` invokes the checks described in this paper.

```
-mpilog     : Automatic MPI and MPE user-defined states logging.
              This links against -llmpe -lmpe.

-mpitrace   : Trace MPI program with printf.
              This links against -ltmpe.

-mpianim    : Animate MPI program in real-time.
              This links against -lampe -lmpe.

-mpicheck   : Check MPI Program with the Collective & Datatype
              Checking library.  This links against -lmpe_collchk.

-graphics   : Use MPE graphics routines with X11 library.
              This links against -lmpe <X11 libraries>.

-log        : MPE user-defined states logging.
              This links against -lmpe.
```

```
-nolog      : Nullify MPE user-defined states logging.
              This links against -lmpe_null.

-help       : Print this help page.
```

To invoke the collective and datatype checking library, one can do

```
mpecc -mpicheck -o mpi_pgm mpi_pgm.c
```

Since MPICH2 provides a more comprehensive set of compiler wrappers, (`mpicc`, `mpicxx`, `mpif77`, and `mpif90`) than MPE's, the MPE profiling options are available through a -mpe=[MPE option] switch provided by these wrappers, e.g. the following commmand will link the user program with the collective and datatype checking library.

```
mpicc -mpe=mpicheck -o mpi_pgm mpi_pgm.c
```

3.4 Example Output

We show here sample output (that would appear on `stderr)` from the collective and datatype checking library to indicate an incorrect use of MPI collective call.

Example 1. In this example, run with four processes with MPICH2, all but the last process call `MPI_Bcast`; the last process calls `MPI_Barrier`.

```
Starting MPI Collective and Datatype Checking
[cli_3]: aborting job:
Fatal error in MPI_Comm_call_errhandler:

Collective Checking: BARRIER (Rank 3) --> Collective call (BARRIER) is Inconsistent with Rank 0's (BCAST).

rank 3 in job 4  jlogin1_42089   caused collective abort of all ranks
  exit status of rank 3: return code 1
```

Example 2. In this example, run with four processes with MPICH2, all but the last process give `MPI_INT`; but the last process gives `MPI_BYTE`.

```
Starting MPI Collective and Datatype Checking
[cli_3]: aborting job:
Fatal error in MPI_Comm_call_errhandler:

Collective Checking: BCAST (Rank 3) --> Inconsistent datatype signatures detected between rank 3 and rank 0.

rank 3 in job 3  jlogin1_42089   caused collective abort of all ranks
  exit status of rank 3: return code 1
```

Example 3. In this example, run with four processes with OpenMPI, all but the last process give `MPI_INT`; but the last process gives `MPI_BYTE`.

```
Starting MPI Collective and Datatype Checking
Collective Checking: BCAST --> no error
Collective Checking: BCAST --> no error
[jlogin1:09505] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:09505] *** on communicator MPI_COMM_WORLD
[jlogin1:09505] *** MPI_SUCCESS: no errors
[jlogin1:09505] *** MPI_ERRORS_ARE_FATAL (goodbye)
```

```
Collective Checking: BCAST --> Inconsistent datatype signatures detected between rank 3 and rank 0.

[jlogin1:09513] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:09513] *** on communicator MPI_COMM_WORLD
[jlogin1:09513] *** MPI_SUCCESS: no errors
[jlogin1:09513] *** MPI_ERRORS_ARE_FATAL (goodbye)
Collective Checking: BCAST --> no error
[jlogin1:09509] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:09509] *** on communicator MPI_COMM_WORLD
[jlogin1:09509] *** MPI_SUCCESS: no errors
[jlogin1:09509] *** MPI_ERRORS_ARE_FATAL (goodbye)
[jlogin1:09501] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:09501] *** on communicator MPI_COMM_WORLD
[jlogin1:09501] *** MPI_SUCCESS: no errors
[jlogin1:09501] *** MPI_ERRORS_ARE_FATAL (goodbye)
[jlogin1:09492] [0,0,0]-[0,1,0] mca_oob_tcp_msg_recv: readv failed with errno=104
3 additional processes aborted (not shown)
```

Example 4. In this example, run with four processes with MPICH2, all but the last process use 0 as the `root` parameter; the last process uses its rank.

```
Starting MPI Collective and Datatype Checking
rank 3 in job 2  jlogin1_42089   caused collective abort of all ranks
  exit status of rank 3: killed by signal 9
[cli_3]: aborting job:
Fatal error in MPI_Comm_call_errhandler:

Collective Checking: BCAST (Rank 3) --> Root Parameter (3) is inconsistent with rank 0 (0)
```

Example 5 In this example, run with four processes with OpenMPI, all but the last process use 0 as the `root` parameter; the last process uses its rank.

```
Starting MPI Collective and Datatype Checking
Collective Checking: BCAST --> no error
[jlogin1:07718] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:07718] *** on communicator MPI_COMM_WORLD
[jlogin1:07718] *** MPI_SUCCESS: no errors
[jlogin1:07718] *** MPI_ERRORS_ARE_FATAL (goodbye)
Collective Checking: BCAST --> no error
[jlogin1:07725] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:07725] *** on communicator MPI_COMM_WORLD
[jlogin1:07725] *** MPI_SUCCESS: no errors
[jlogin1:07725] *** MPI_ERRORS_ARE_FATAL (goodbye)
Collective Checking: BCAST --> no error
[jlogin1:07729] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:07729] *** on communicator MPI_COMM_WORLD
[jlogin1:07729] *** MPI_SUCCESS: no errors
[jlogin1:07729] *** MPI_ERRORS_ARE_FATAL (goodbye)
Collective Checking: BCAST --> Root Parameter (3) is inconsistent with rank 0 (0)
[jlogin1:07734] *** An error occurred in MPI_Comm_call_errhandler
[jlogin1:07734] *** on communicator MPI_COMM_WORLD
[jlogin1:07734] *** MPI_SUCCESS: no errors
[jlogin1:07734] *** MPI_ERRORS_ARE_FATAL (goodbye)
3 additional processes aborted (not shown)
```

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 the 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. That is, part of the code to test the use of datatypes contained the fragment:

```
sendtype->datatype = MPI_INT;
sendtype->count    = 1;
recvtype->datatype = MPI_BYTE;
recvtype->count    = sizeof(int);
```

This is incorrect, although MPICH2 allowed the program to execute.

To test a real application, we linked FLASH [9], 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] and OpenMPI [8, 3], without encountering any problems. However, we noticed that incorrect MPI collective programs sometimes behaved differently on different implementations This is perfectly acceptable, since the MPI Standard does not specify the behavior of erroneous programs. But it does make it particularly useful to identify and report such errors before the actual collective call takes place.

We carried out performance tests on three platforms. On BlueGene/L, the collective and datatype checking library and the test codes were compiled with `xlc` of version 8.0 and linked with the IBM's MPI implementation (V1R3M1_400_2006-061024) available on BlueGene/L.

Test Name	count×N_{itr}	No CollChk	With CollChk
MPI_Bcast	1×10	0.000028	0.001543
MPI_Bcast	1K×1	0.000031	0.000424
MPI_Bcast	128K×1	0.003121	0.032495
MPI_Allreduce	1×10	0.000063	0.001898
MPI_Allreduce	1K×1	0.000136	0.000543
MPI_Allreduce	128K×1	0.009167	0.038532
MPI_Alltoallv	1×10	0.000423	0.002264
MPI_Alltoallv	1K×1	0.000175	0.000812
MPI_Alltoallv	128K×1	0.015522	0.074069

Table 3: The maximum time taken (in seconds) among all the processes in a 32-process MPI job on BlueGene/L. Where count is the number of MPI_Double 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.

The performance of the collective and datatype checking library of a 32-process job is listed in Table 3, 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` of version 4.0.2 and linked with MPICH2-1.0.5 and OpenMPI-1.1.2. The performance results of the library are tabulated in Table 4 and 5.

Test Name	count×N_{itr}	No CollChk	With CollChk
MPI_Bcast	1×10	0.040800	0.178053
MPI_Bcast	1K×1	0.011457	0.045029
MPI_Bcast	128K×1	0.197951	0.240698
MPI_Allreduce	1×10	0.006159	0.111160
MPI_Allreduce	1K×1	0.005569	0.047161
MPI_Allreduce	128K×1	0.304521	0.341304
MPI_Alltoallv	1×10	0.001572	0.093878
MPI_Alltoallv	1K×1	0.003105	0.042199
MPI_Alltoallv	128K×1	0.270906	0.293240

Table 4: The maximum time taken (in seconds) on a 32-process MPICH2 job on Jazz, an IA32 Linux cluster. Where count is the number of MPI_Double in the datatype, and N_{itr} refers to the number of times the MPI collective routine was called in the test.

Tables, 3, 4 and 5, show that the relative 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 not likely 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.

Test Name	count×N_{itr}	No CollChk	With CollChk
MPI_Bcast	1×10	0.043394	0.157876
MPI_Bcast	1K×1	0.009069	0.044477
MPI_Bcast	128K×1	0.160046	0.181880
MPI_Allreduce	1×10	0.051406	0.176388
MPI_Allreduce	1K×1	0.230335	0.274223
MPI_Allreduce	128K×1	1.624644	1.660873
MPI_Alltoallv	1×10	0.010516	0.207629
MPI_Alltoallv	1K×1	0.010338	0.046493
MPI_Alltoallv	128K×1	0.348700	0.357958

Table 5: The maximum time taken (in seconds) on a 32-process OpenMPI job on Jazz, an IA32 Linux cluster. Where count is the number of MPI_Double in the datatype, and N_{itr} refers to the number of times the MPI collective routine was called in the test.

5 Future Work

We are pursuing a number of extensions and enhancements to this collective error-checking library With respect to the current implementation, we need to forward any error returns from the MPI library to the user when the error handler is not `MPI_ERRORS_ABORT`; this is particularly true when `MPI_ERRORS_RETURN` has been selected as the error handler for MPI routines.

In an MPI-2 environment, we can reduce the cost of comparing datatype signatures by making use of the attribute caching functions to save the hash value on the MPI datatype; thus we only need to compute the hash function once for a derived datatype. This can be done when the datatype is commited with `MPI_Type_commit`, reducing the cost within the collective routines.

A similar approach for detecting datatype mismatches can be applied to point-to-point operations. In this case, we would also like to enable the option of an implementation-specific approach, as it is relatively easy to include the datatype hash value in the message envelope used within the MPI implementation. Of course, we will provide a fully portable version that does not rely on interfacing with the internals of the MPI implementations.

One weakness of our approach is that it assumes that all processes in the communicator are calling a collective operation. While we do check for the error of calling different collective operations on the same communicator, we do not check for calling collective operations on different communicators. For example, consider this code fragment in a single-threaded program:

```
MPI_Comm_split( ..., &comm1 );
MPI_Comm_split( ..., &comm2 );
if (rank < size/2) {
    MPI_Bcast( ..., comm1 );
    MPI_Bcast( ..., comm2 );
}
else {
    MPI_Bcast( ..., comm2 );
    MPI_Bcast( ..., comm1 );
}
```

While something so clearly wrong is unlikely to occur so obviously in a code, this set of collective operations could occur as a result of complex (and erroneous) logic in managing several communicators, such as communicators for row and column computations in a matrix. Our library will not catch these; instead, a deadlock will occur within the collective operations that we use to implement the checks. To catch these errors, it is necessary to use point-to-point operations on a private communicator. In MPI-1 programs, this can be accomplished by creating a dup of `MPI_COMM_WORLD` and using a point-to-point implementation of an Allreduce operation to compare a communicator id, also maintained by the colcheck library, to ensure that collective operations are properly ordered in the MPI code. By using nonblocking operations and implementing a time-out, we can also handle errors where some processes do not make a collective call at all, as in this example:

```
if (rank == 0) MPI_Bcast( ..., 0, MPI_COMM_WORLD );
else           MPI_Recv( ..., 0, MPI_COMM_WORLD );
```

(This error is seen when a user uses `MPI_Bcast` as if it is a "SendAll" instead of an MPI collective operation.)

Multithreaded programs introduce additional complexity. For example, the above mechanism for detecting mismatched use of communicators in collective calls is no longer valid because different threads are permitted (and in fact are encouraged) to make MPI collective calls on different communicators. Detecting user errors, such as the use of `MPI_Bcast` as a SendAll, will require either a timeout check (with an arbitrary, user-controllable timeout period) or more sophisticated deadlock detection, such as determining that every MPI thread (not just process) is in a blocking wait.

Finally, while the performance overhead for this profiling library is probably acceptable for use in testing rather than production runs, the current implementation admits of optimizations that have not yet been carried out. The optimizations with the most impact are those which will reduce the number of "extra" collective operations carried out by the checking library.

6 Summary

In this paper we have described an effective technique for MPI programmers to use for detecting easy-to-make but hard-to-find mistakes that often lead to deadlock, incorrect results, or worse. The technique demonstrates an important use of the MPI "profiling library" idea. We have demonstrated the library's portability by running it with three different MPI implementations and measuring its performance. We have implemented the datatype hashing algorithm presented elsewhere and demonstrated its effectiveness. Our final section outlines future improvements that are under way.

References

[1] G. Almási, C. Archer, J. G. Castaños, 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] Chris Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective error detection for MPI collective operations. In dieter Kranzlmüller Beniamino di Martino and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*, number 3666 in Lecture Notes in Computer Science, pages 138–147. Springer, 2005.

[3] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implemen-

tation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[4] 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.

[5] 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.

[6] Julien Langou, George Bosilca, Graham E. Fagg, and Jack Dongarra. Hash functions for datatype signatures in MPI. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *PVM/MPI*, volume 3666 of *Lecture Notes in Computer Science*, pages 76–83. Springer, 2005.

[7] MPICH2 Web page. `http://www.mcs.anl.gov/mpi/mpich2`.

[8] OpenMPI Web page. `http://www.open-mpi.org`.

[9] 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.

[10] 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.

[11] Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. In Dieter Kranslmüller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 3241 in Springer Lecture Notes in Computer Science, pages 18–27, 2004.