John von Neumann Institute for Computing

The MPI/SX Collectives Verification Library

J.L. Träff, J. Worringen

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 909-916, 2006.

http://www.fz-juelich.de/nic-series/volume33

The MPI/SX Collectives Verification Library

Jesper Larsson Träff[a], Joachim Worringen[a]

[a]C&C Research Laboratories, NEC Europe Ltd., Rathausallee 10, D-53757 Sankt Augustin, Germany

This paper summarizes and discusses the functionality of an extended MPI library for verifying correct use and consistency of all collective functions of the MPI-2 standard. The library is part of the MPI/SX implementation for the NEC SX-series of parallel vector-computers, as well as NEC implementations of MPI for other platforms. We give examples of the use of the verification library, and in particular report on the overheads entailed. The library could have been implemented as a stand-alone, portable interface by using the MPI "profiling interface" as defined by the MPI standard. We discuss obstacles of a portable implementation, and instead argue to support collective verification as part of any good MPI implementation.

1. Introduction

MPI, the *Message Passing Interface* MPI [4, 9], contains a large number of functions that are collective over a set of processes, meaning that all the processes in the set must call the function in order to let it complete on all processes. This includes collective communication and reduction operations like `MPI_Bcast` and `MPI_Reduce`, but also functions for creating new sets of processes (*communicators*), spawning processes, creating windows for one-sided communication or opening files are collective in this sense. All collective functions of MPI explicitly or implicitly pose certain *consistency requirements* among the parameters passed by the different processes, and sticking to these rules is essential for correct execution of the application. As a simple example, the processes calling the `MPI_Bcast` function must supply the same `root` parameter. Violating consistency conditions may lead to unpredictable, but often fatal behavior of the application, entirely dependent, however, on the MPI implementation at hand. The application may deadlock (as could be the case if a different `root` parameter was given in an `MPI_Bcast` or `MPI_Reduce` call), give wrong results (as could be the case if non-matching datatypes and counts were given in the call), or crash, possibly at some time later in the application (this could be the case if different process group parameters were given to a communicator creating function). On the other hand, especially for irregular collectives like `MPI_Gatherv` or `MPI_Reduce_scatter`, the consistency requirements are quite complex, and it is easy to make mistakes. Thus, it would help an application developer to have the MPI library check parameter consistency in the use of the collective MPI functions.

Consistency requirements are conditions that require communication to check, and are thus expensive compared to simple argument checks that can be performed locally by each process (is the root argument in range? is the datatype committed?). For high-performance use such checks are therefore prohibitive. Consequently, the MPI standard does not specify a behavior in case of errors, and it is legal for an MPI implementation to deadlock or even crash if collective functions are not called consistently. A solution to the dilemma would be to have a separate MPI library to be used during application development/debugging to catch consistency errors. Such a library could either be implemented stand-alone and in a portable fashion using the profiling interface feature of the MPI standard, or implemented as a separate library for a specific MPI implementation. The latter course has been followed for MPI/SX.

2. Related Work

The portable tool *MARMOT* is described in a series of papers [5, 6, 7]. It uses a single (additional) MPI process to act as a control process: the MPI processes running the user application log all MPI function calls with this control process, which in turn performs a global analysis of the function calls and the communication patterns. This allows to give detailed diagnostic output i.e. for a deadlock situation. However, this approach is inherently non-scalable, and we believe that it is possible to achieve a similar verification quality following our distributed approach if a separate deadlock-detection mechanism is available.

Umpire [12] follows a similar approach as MARMOT, but relies on out-of-band shared-memory communication to have a dedicated thread of process 0 analyze the MPI function calls of all other processes that they logged in the shared memory area. The main focus of Umpire is deadlock detection. Obviously, the requirement that all processes of the application to be verified run on a single node limits the scalability and applicability of this approach especially on cluster machines with few CPUs per node.

A deadlock and use checker for MPI programs in Fortran 90 and Fortran 77 named *MPI-CHECK* is described in [8]. Next to some checks of correct use of MPI functions which are done in part by instrumenting the Fortran code at source level, its main focus is on distributed deadlock detection. It does not use a central control instance, but instead uses a handshake-protocol on top of the MPI point-to-point communication functions to validate each send and receive call with its destination and source, respectively.

These tools do deadlock checking, which for MPI/SX is left to the MPI implementation (suspend/resume mechanism). Extensive local checks are also performed by the MPI/SX library. Thus the MPI/SX verification library focuses entirely on collective consistency requirements.

Intel Message Checker [1] does mostly checking for point-to-point communication and is an offline, trace-based tool. This means that the verification is necessarily a three-step process: first run the application to create the trace file, then run the analyzer software with this trace file as input, and finally visualize the results using a graphical tool. This indirect approach complicates the verification for the user, and depending on the size of the generated trace, may even make it infeasible.

Recently MPICH2 has incorporated a checking interface similar to the approach of MPI/SX. The MPICH2 approach is portable by using the MPI profiling interface [3]. It extends our approach by doing datatype signature checking using hash-values. On the other hand, it does not perform a verification as complete as our approach as it is not able to decode opaque MPI objects like `MPI_Group` or `MPI_Win`, does not handle inter-communicators, and is missing some other verifications, i.e. in MPI-IO (see Chapter 5).

We summarize the comparison of the different approaches in Table 1, which is based on published information. It shows the general architecture of an approach, lists the supported types of communicators for collective operations, the type of checks performed for point-to-point operations, the availability of the profiling interface when using the verification library, the portability of the software to different MPI libraries, the occasions on which the use of MPI datatypes is verified, the capability of checking the use of opaque MPI objects (like `MPI_Win` and `MPI_Group`) and the degree of support for the MPI-2 standard.

3. Design and Implementation of the MPI/SX verification library

The approach to verification in MPI/SX is described in more detail in [11]. Local checks that can be performed fast (in constant time, independent of the number of processes, data size etc.) are

Table 1
Key characteristics of different MPI verification approaches.

	architecture	collective	point-to-point checking	PMPI
NEC MPI/SX	distributed	intra & inter	deadlock	available
MPICH2	distributed	intra	datatype	used
MARMOT	centralized (distributed memory)	intra	deadlock	used
Umpire	centralized (shared memory)	intra	deadlock, buffer	used
MPI-CHECK	distributed, instrumentation	intra	deadlock	used
Intel MC	tracefile (offline)	intra	deadlock, buffer,datatype	used
	portable	opaque objects	datatype checking	MPI-2
NEC MPI/SX	no (NEC MPI)	yes	setup of file view	full
MPICH2	yes	no	communication (hash)	partially
MARMOT	yes	yes	construction	no
Umpire	limited (SMP only)	no	no	no
MPI-CHECK	limited (Fortran only)	no	no	no
Intel MC	limited (Intel MPI & MPICH2)	unknown	communication (partly)	partially

always performed by MPI/SX, as they are often valuable for catching irritating errors (e.g. rank out of range, non-committed datatype) and do not hamper performance. Non-local consistency checks on arguments to collective operations all require (expensive) communication, and are therefore exclusively done by the verification library. To verify that all processes calling e.g. `MPI_Bcast` with the same `root` argument it suffices for some process, say rank 0, to broadcast its value of the `root` argument to the other processes, which in turn check this against their own value for the `root` argument. A process detecting an inconsistency could report the error, in which case it would normally make sense to abort the application. In the spirit of MPI the error handler associated with the communicator of the `MPI_Bcast` call should be called. In case the user has changed the error handler to not abort, for instance by using `MPI_ERRORS_RETURN` this could again lead to highly unpredictable behavior, since only the processes that detected the wrong `root` argument would be aware of the error condition. To avoid this, the action of the MPI/SX verification library is implemented to be *symmetric*. All processes will be informed of a possible error condition and can thus all invoke the error handler. The cost of this an extra `MPI_Allreduce` for each verified condition. Overall, the total cost per collective operation is from two to eight extra collective calls (either `MPI_Bcast`, `MPI_Gather`, `MPI_Alltoall`, or `MPI_Allreduce`), all with small data (from a single `MPI_Aint` up to as many `MPI_Aint` as there are processes in the communicator).

Collective verification of this sort can in principle be implemented in MPI itself, and made available in a portable fashion by using the profiling interface mechanism of MPI. However there are some tedious obstacles to this approach. A minor problem is that some MPI objects (for instance `MPI_Aint`, `MPI_Op`) are not first-class citizens, and therefore (formally) cannot be exchanged in communication operations. These must therefore be mapped to objects that can be used in communication operations. More severe difficulties of this sort are the extraction of the processes from an `MPI_Group` object which is necessary when verifying for instance `MPI_Comm_create`, or the extraction of the underlying communicator from a one-sided communication window. These difficulties are trivial to overcome from within an actual MPI implementation. A further advantage of having a special library is that the profiling interface is not “used up”, meaning that other profiling can be done together with the verification library.

4. Using the verification library

To use the library, only relinking of the aplication with `-lvmpi` is needed. It is also possible to use a version of the verification library that provides a profiling interface by using `-lvpmpich`. In this case, the additional collective operations performed by the verification library will *not* be visible to the library using the profiling interface. This is due to the fact that the verification library uses internal hooks to these operations.

The level of verification and/or reporting is controlled either by the MPI profiling interface function `MPI_Pcontrol(level,...)` or by setting the environment variable `MPIVERIFY`.

- `level 0`: disabled
- `level 1`: return an error code to the error handler
- `level 2`: print an additional description of the problem

By default, the verification level is set to 2. Setting the level to 0 will cause only minimal run-time overhead of a few `if` statements per collective operation. Depending on the individual requirements, this allows to use the verification library also for production code. All other verification levels will, if a problem occurs, call the active error handler of the communicator in whose context the function was called. In such a case the error handler is called by all processes.

5. Examples

We have tested the verification library with some existing applications. Finding errors in an application in production is expected to be rare, as these have already been tested by other means.

We expect the verification library to be more successful when it is used during application development. We could resolve a bug in such an application where a series of `MPI_Bcast` operations was performed. The first operation broadcasted the amount of data for the following operations. However, there was a mismatch in these values on the root process by which too little data was broadcasted in the subsequent operations. This problem was not discovered within the broadcast operations, but showed up much later in the application as data corruption. The verification library could detect the mismatch between amount of data received and expected.

We have created a test suite for the verification library. This test suite contains some typical errors, like setting the send and receive counts wrong for an `MPI_Alltoallv` operation:

```
for (i = 0; i < nbr_processes; i++) {
  send_count[i] = i;
  recv_count[i] = nbr_processes - i; /* ERROR: should be 'my_rank' */
}
```

Here, each process should receive an amount of data proportional to its rank. However, this requires that each process sets all entries of the `recv_count` array to its own rank.

Test cases for MPI-IO deal with using process-individual file views with shared file pointers (which is not allowed by the MPI standard), or defining non-contiguous file views where the extents of gaps are not multiples of the extent of the elementary data type. The latter problem is local to each process and only occurs within a collective operation.

The MPI/SX verification library detects all these errors. It is more interesting to observe how MPI libraries without an explicit verification capability handle such problems. We checked this with

our own MPI library and with the latest version of MPICH2 (1.0.2), which can be considered as a reference implementation. Some problems lead to deadlocks with both libraries, while some, like the `MPI_Alltoallv` problem described above, behave differently.

This problem returned an (unspecified) `MPI_ERR_TRUNCATE` error code with our library, but deadlocked with MPICH2. The reason for this deadlock was found in the chosen algorithm: In MPICH2, empty messages are not sent within `MPI_Alltoallv`, However, in this case, the receiver erroneously waits for a (non-empty) message. The algorithm in our library is different and thus reacts differently. With the verification library, the following diagnostic message is output:

```
VERIFY MPI_ALLTOALLV(0): sendsize[1]=4 != expected recvsize(1)[0]=16
VERIFY MPI_ALLTOALLV(1): sendsize[0]=0 != expected recvsize(0)[1]=12
VERIFY MPI_ALLTOALLV(2): sendsize[0]=0 != expected recvsize(0)[2]=8
VERIFY MPI_ALLTOALLV(3): sendsize[0]=0 != expected recvsize(0)[3]=4
```

Some of the verifications done within the MPI-IO operations are local. As the overhead is actually not very large and some of these functions like `MPI_File_set_view` are not as performance critical as the collective communication functions, we decided to activate most verifications also in the default version of the MPI/SX library. Still, the extended diagnostic messages are only printed by the verification library. The standard library only returns an error code which needs to be decoded by `MPI_Error_string`. However, this leads to the effect that 8 of the 9 problems for MPI-IO are also detected without the verification library when using MPI/SX. The original ROMIO as used in MPICH2, too, generates an error for one of the problems tested.

6. Verification overhead

Naturally, the verification functionality incurs extra overhead. The nominal overhead per collective operation, measured as number of additional collective operations, is from two to eight operations. The amount of data exchanged per operation is small, in the worst case proportional to the number of processes in the communicator, and in most cases just a single `MPI_Aint`.

6.1. Synthetic Benchmark

We used a standard benchmark for collective communication operations to measure the actual overhead of the verification library on different platforms. Although the verification does not only take place in the communication operations, the methods used and thus the overhead implied is very similar.

We express this overhead as a *relative slowdown*, which means we give a percentage of how much additional time is needed for a collective operation to complete at all processes. Figure 1 shows these numbers for runs of this benchmark on an NEC SX-8 machine, using 32 processes on 4 nodes and for an IA-32 based cluster with Myrinet 2000 interconnect, using 32 processes on 16 nodes.

As can be seen from the charts, the overhead for small message sizes, which means short execution times of the non-verified operation, is very significant. It varies between a factor of 4 for `MPI_Allgather` up to a factor of 14 for `MPI_Gather` and `MPI_Scatter`. The high overhead for the latter operations relates to the relatively short execution time of the non-verified operation.

With increasing data size, the execution time of the non-verified operation increases, while the time required for the verification remains constant. This leads to a decrease of the relative overhead of the verification library. Summarizing, the overhead of the verification library is non-negligible for individual collective operations with small data.

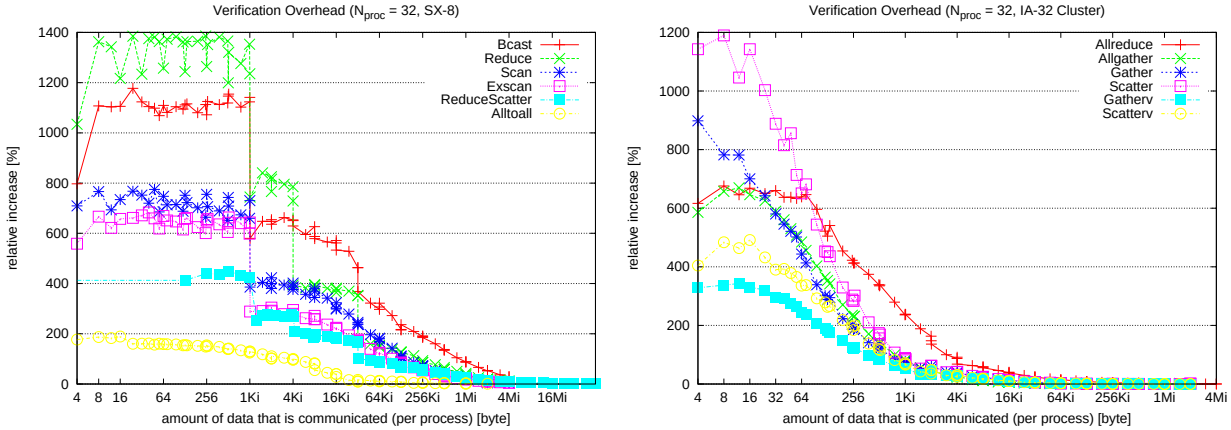

Figure 1. Relative overhead of verification library for some collective operations on NEC SX-8 and generic IA-32 cluster

6.2. Application Benchmark

The overhead of the verification library as evaluated in the previous section is significant, especially for small data sizes and collective operations with a short native run time. It is obvious that for applications where the performance is bound by the performance of the MPI collectives for small data, use of the verification library imposes a significant overhead. We wanted to know how much this actually affects the run time of an actual application.

For this evaluation, we chose the *hypre* [2] package of preconditioners and linear solvers. We ran the test case `ij` from the test suite using the default algebraic multi-grid solver (index 0) with a $50 \cdot N_{proc} \times 100 \times 100$ grid, with N_{proc} being the number of processes. In this configuration, each process has a peak memory usage of about 1GB. The only difference between the two executables was the additional option `-lvmpi` when linking the application.

We executed the test case on 8 nodes of the aforementioned cluster, with each node running 2 processes ($N_{proc} = 16$). To get statistically valid data, we performed 99 runs of each variant of the test case. In each `ij` run, more than 5000 collective operations are executed. About 70% of these operations are `MPI_Allreduce` with 4-byte integer values. The remaining operations are calls to `MPI_Allgather(v)` and `MPI_Gather` with small data sizes below 128 bytes. Based on the results of the synthetic benchmarks, we estimated the overhead that would be introduced by using the verification library to about 2s additional runtime. The runtime with the standard MPI library is in the range of 140s. The results of our tests are presented in Table 2. We give the average (arithmetic mean) of all run times and the related standard deviation as well as the 90^{th}-quantile to better handle the outliers.

The overhead calculated from the average execution time is very small, and it contrasts to the expected overhead of about 2 seconds. The reason for this may be found in the value of the standard deviation which is larger than the expected overhead. Calculating the overhead based on the 90^{th}-quantile results in a value closer to the expected overhead.

The large standard deviation surprised us as the cluster was operated as a production system with a batch system in which the nodes are exclusive for the individual application. The distribution plot in Figure 2 shows that the execution times actually vary between 137 and 164 seconds for the same

Table 2
Run time of *hypre* test case `ij` with and without verification library.

	w/o verify	w/ verify	overhead
avg. runtime [s]	139.78	140.10	0.32
standard deviation	2.96	3.86	n/a
90^{th}-quantile [s]	141.64	142.27	0.63

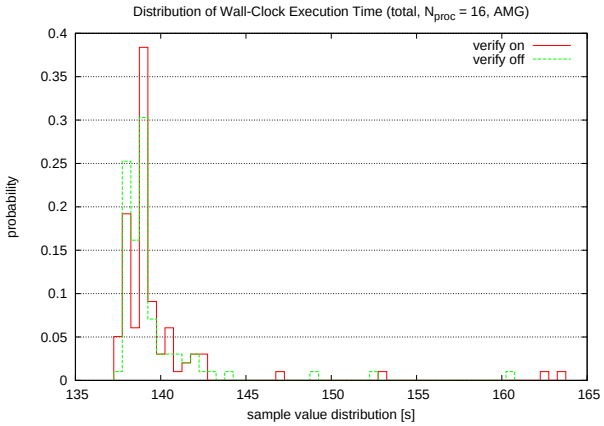

Figure 2. Normalized distribution of execution time of *hypre* `ij` test case.

problem on the same set of nodes. This shows two things: first, simply averaging across a small number of runs can easily give bogus results, and secondly that in this case, the actual overhead of the verification library could thus even be tolerated for a production environment.

7. Conclusion

The MPI/SX verification library for collective operations is an easy-to-use tool to verify the correctness of an MPI application. It covers all MPI functions which need to be called by all processes of an inter- or intra-communicator to complete. Because our implementation can access internal data structures of the MPI/SX library, it is able to cover more potential errors than a portable approach reasonably could.

We continue to work on the verification library in order to create a comprehensive tool not only for collective operations, but for the complete range of MPI functions. To achieve this, we will implement techniques to ensure the correct use of MPI datatypes, and will extend the verification to include non-collective MPI operations. To validate the correct use of MPI datatypes, we will make use of the available technique in MPI/SX for the space- and time-efficient representation of derived datatypes [10]. In contrast to approaches that use a hash-value of a datatype, using the complete representation of a datatype can never lead to false diagnostics or undetected errors. Aspects of verification for non-collective operations are correct use of message buffers and MPI requests. Another important aspect, namely the detection of deadlocks, is already implemented in MPI/SX outside the scope of the verification library.

8. Acknowledgments

We thank our colleague Jens Georg Schmidt for support with the application benchmarking. All data for the performance evaluation was processed using *perfbase* [13].

References

[1] Jayant DeSouza, Bob Kuhn, and Bronis R. de Supinski. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Second International Workshop on Software Engineering for High Performance Computing System Applications in conjunction with 27th International Conference on Software Engineering*, 2005.

[2] R.D. Falgout and U.M. Yang. hypre: a library of high performance preconditioners. In *Computational Science (3) - ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641, 2002.

[3] Chris Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective error detection for MPI collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 138–147, 2005.

[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 Extensions. MIT Press, 1998.

[5] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing (ParCo)*, 2003.

[6] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI application development using the analysis tool MARMOT. In *International Conference on Computational Science (ICCS)*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471, 2004.

[7] Bettina Krammer, Matthias S Müller, and Michael M. Resch. MPI I/O analysis and error detection with MARMOT. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 242–250, 2004.

[8] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[9] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.

[10] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116, 1999.

[11] Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 18–27, 2004.

[12] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing (SC)*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm#01`.

[13] Joachim Worringen. Experiment management and analysis with perfbase. In *Proceedings of the IEEE International Conference on Cluster Computing*, Boston, September 2005. IEEE Computer Society Press.