ANL/MCS/CP-99346

# A Standard Interface for Debugger Access to Message Queue Information in MPI

James Cownie[1] and William Gropp[2] *

[1] Etnus Inc., Framingham MA, USA
jcownie@etnus.com,
WWW home page: http://www.etnus.com/
[2] Mathematics and Computer Science Division, Argonne National Laboratory
gropp@mcs.anl.gov,
WWW home page: http://www.mcs.anl.gov/~gropp

RECEIVED
OCT 13 1999
OSTI

**Abstract.** This paper discusses the design and implementation of an interface that allows a debugger to obtain the information necessary to display the contents of the MPI message queues. The design has been implemented in the TotalView debugger, and dynamic libraries that conform to the interface exist for MPICH, as well as the proprietary MPI implementations from Compaq, IBM, and SGI.

## 1 Introduction

In any debugging task one of the debugger's main objectives is to make visible information about the state of the executing program.

Debuggers for sequential processes provide easy access to the state of the process, allowing the user to observe the values of variables, machine registers, stack backtraces, and so on. When debugging message-passing programs all of this information is still required; however, additional information specific to the message passing model is also needed if the user is to be able to debug the problems, such as deadlock that are specific to this new environment. Unfortunately the message-passing state of the program is not easily accessible to the user (or the debugger), because it is represented by data structures in the message-passing library, whose format and content is implementation dependent.

To address this problem, we describe an interface within the debugger itself that allows library-specific code to extract information describing the *conceptual* message-passing state of the program so that this can be displayed to the user.

## 2 The User's Model of MPI Internal State

Since each MPI communicator represents an independent communication space, the highest level of the user's model is the communicator. Within a communi-

* This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

## DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

cator there are, conceptually at least, three distinct "message queues," which represent the state of the MPI subsystem. These are

**Send Queue:** represents all of the outstanding send operations.
**Receive Queue:** represents all of the oustanding receive operations.
**Unexpected Message Queue:** represents messages that have been sent to this process, but have not yet been received.

The send and receive queues store information about all of the unfinished send and receive operations that the process has started within the communicator. These might result either from blocking operations such as `MPI_Send` and `MPI_Recv` or nonblocking operations such as `MPI_Isend` or `MPI_Irecv`. Each entry on one of these queues contains the information that was passed to the function call that initiated the operation. Nonblocking operations will remain on these queues until they have completed and have been collected by a suitable `MPI_Wait`, `MPI_Test`, or one of the related multiple completion routines. The unexpected message queue represents a different class of information, since the elements on this queue have been created by MPI calls in other processes. Therefore, less information is available about these elements. In all cases the order of the queues represents the order that the MPI subsystem will perform matching (this is important where many entries could match, for instance when wild-card tag or source is used in a receive operation).

Note that these queues are *conceptual*; they are a description of how a user can think about the progression of messages through an MPI program. As we discuss below, an MPI implementation may have fewer or more queues, and the queues within an implementation may be different. The interface described in this paper addresses how to extract these *conceptual* queues from the implementation so that they can be presented to the user independently of the particular MPI implementation.

## 3 Queues in Actual MPI Implementations

The MPICH implementation of MPI [1] maintains only two queues in each process: a queue of posted receives and a queue of unexpected messages. There is no explicit queue of send operations; instead, all of the information about an incomplete send operation (e.g., an `MPI_Isend`) is maintained in the associated `MPI_Request`. In addition, operations associated with all communicators live in the same queues, being distinguished internally by a context identifier.

Other organizations of message queues are possible. For example, Morimoto et al [2] describe a system where posted receives from a particular source are sent to the processor that is expected to send the message; in this implementation there is a queue of unmatched receives that send operations try to match before sending data to a destination. This does not conflict with the queue model that we have described; it is always possible for an MPI implementation to present the three queues that this interface requires. However, this internal queue of "expected receivers" is not visible to the user via the debug interface described here.

## 4 How to Extract State for a Debugger

While it would be possible to define a set of data structures that an MPI library must implement solely so that a debugger could extract the data it needed, specifying such implementation details was at conflict with the aim of the MPI Forum of defining an interface that would allow implementations to achieve low latency. Therefore, although we considered the idea of making such a proposal to the MPI-2 Forum, we decided against it, since it would clearly never be accepted.

Another apparently attractive solution would be to provide MPI calls that could be made inside a process to return information about the MPI library state. Unfortunately, this approach has a significant problem: it requires that there be a thread in the process that the debugger can use at any time to call the inquiry function. In many cases of interest no such thread exists; for instance, when debugging core files, there are no threads at all, or when debugging a deadlocked set of processes, all of the threads are likely to be in the system calls underlying the send or receive operations. These arguments lead to the conclusion that accessing the library state must be driven by the debugger, rather than code running inside the process.

One way to permit the debugger to display MPI library state would be explicitly to code the knowledge of the MPI library's internal data structures into the debugger. While this works (and indeed was the way that we initially implemented MPI queue display for MPICH in TotalView [3]), it is unattractive because it links the debugger and MPI library versions together inextricably and it requires that the writers of the debugger have access to the details of the MPI implementation. For vendor-specific MPI implementations (when the MPI vendor and debugger vendor are different), such access is hard or impossible to obtain (often requiring the involvement of lawyers).

### 4.1 Use of a Library Dynamically Linked into the Debugger

The solution that we have now adopted (and which is supported by TotalView) is to have the debugger use the Unix `dlopen` call to load a dynamic library (the "debug DLL") at run time, when the debugger knows which MPI implementation is being used. This debug DLL is provided and distributed by the writers of the target MPI library.

This allows the debugger to be insulated from the internals of the MPI library, so that changes to the MPI implementation do not require the debugger to be rebuilt. Instead, the MPI implementation's specific debug DLL must be changed, but this is the responsibility of the MPI implementors. It allows the debugger to support multiple MPI implementations on the same system (for instance, both MPICH and a vendor MPI implementation) and to be portable, requiring no changes to its MPI support to work with many different MPI implementations. Finally, it allows implementors of MPI to provide their users with high-level debugging support without requiring access to the source code of the debugger.

## 5 The Interface between Debugger and Debug DLL

All calls to the debug DLL from the debugger are made through entry points whose names are known to the debugger. However, all calls back to the debugger from the debug DLL are made through a table of function pointers that is passed to the initialization entrypoint of the debug DLL. This procedure ensures that the debug DLL is independent of the specific debugger from which it is being called.

### 5.1 Opaque Objects Passed through the Interface

The debugger needs to identify to the debug DLL a number of different objects (`mqs` for "message queue system"):

`mqs_image`: an executable image file
`mqs_process`: a specific process
`mqs_type`: a named target type (`struct` or `typedef`)

However, the debugger does not want to expose its internal representations of these types to the debug DLL, which has no need to see the internal structure of these objects, but merely uses them as keys to identify objects of interest, or to be passed back to the debugger through a callback.

These objects are therefore defined in the interface file as `typedefs` of undefined structures and are always passed by reference. The use of these opaque types allows the debugger the freedom either to pass true pointers to its internal data structures or to pass some other key to the debug DLL from which it can later retrieve its internal object. We prefer `typedefs` of undefined structures rather than simply using `void *`, since using the `typedefs` provides more compile-time type checking over the interface.

For reasons of efficiency it is important that the debug DLL be able easily to associate information with some of these debugger-owned objects. For instance, it is convenient to extract information about the address at which a global variable of interest to the debug DLL lives only once for each process being debugged, rather than every time that the debug DLL needs access to the variable. Similarly, the offset of a field in a structure that the debug DLL needs to understand is constant within a specific executable image, and again should be looked up only once. Callbacks are therefore provided by the debugger to allow the debug DLL to store and retrieve information associated with image and process objects. Since retrieving the information is a callback, the debugger has the option either of extending its internal data structures to provide space for an additional pointer or of implementing a hash table to associate the information with the process key.

### 5.2 Concrete Objects Passed through the Interface

To allow the debugger to obtain useful information from the debug DLL, concrete types are defined to describe a communicator and a specific element on a message queue.

The information in the mqs_communicator structure includes the communicator's size, the local rank of the process within the communicator, and the name of the communicator as defined by the MPI implementation or set by the user using the MPI-2 [4] function MPI_Comm_set_name, which was added to the standard specifically to aid debugging and profiling.

The mqs_pending_operation structure contains enough information to allow the debugger to provide the user with details both of the arguments to a receive and of the incoming message that matched it. All references to other processes are available in the mqs_pending_operation structure both as indices into the group associated with the communicator and as indices into MPI_COMM_WORLD. This avoids any need for the debugger to concern itself explicitly with this mapping.

### 5.3 Target Independence

Since the code in the debug DLL is running inside the debugger, it could be running on a completely different machine from the debugged process. Therefore, the interface uses explicit types to describe target types, rather than using canonical C types. The interface header definition file defines the types mqs_taddr_t and mqs_tword_t that are appropriate types for the debug DLL to use on the host to hold, respectively, a target address (void *) and a target word (long).

It is also possible that although the debugger is running locally on the same machine as the target process, the target process may have different properties from the debugger. For instance, on AIX, IRIX, or Solaris 7, it is possible to execute both 32- and 64-bit processes. To handle this situation, the debugger provides a callback that returns type size information for a specific process. To handle the possibility that the byte ordering may be different between the debug host and the target, the debugger provides a callback to perform any necessary byte reordering when viewing the target store as an object of a specific size.

The debugger also provides callbacks to the debug DLL to allow it to find the address of a global symbol, to look up a named type, to find the size of a type, and to find the offset of a field within a (struct) type. Each of these calls takes as an argument a specific process.

These callbacks enable the debug DLL to be entirely independent of the target process, as is demonstrated by the fact that the MPICH implementation of the debug DLL contains no target-specific #ifdefs yet is successfully used on a variety of both big- and little-endian 32- and 64-bit systems.

### 5.4 Services Provided to the Debug DLL by the Debugger

As well as the services already described for extracting symbols and types and for supporting target independence, the debugger provides the debug DLL with a function for reading the store of a target process. This is the most fundamental service provided, since without it the debug DLL would have no access to target runtime state information.

## 6 Extracting the Information

The debug DLL functions called from the debugger to extract the information are structured as iterators: one to iterate over all of the communicators in a process, and a second to iterate over all of the messages in a specific message queue. This avoids the problems of store allocation that would arise were the interface defined in terms of arrays or lists of objects.

Since a communicator and associated translation groups can be of significant size (in MPICH a communicator and its associated groups occupy at least 176 bytes on a 32-bit machine), reading all of the communicator information each time that a message queue is displayed could be slow. Therefore a call to the routine `mqs_update_communicator_list` is made before the communicator iterator is initialized if the debugger knows that the process has run. This allows the DLL to hold the communicator information locally and to update it only when the communicator list has been changed. In the MPICH library such changes are detected by the use of a sequence number that is incremented whenever a communicator is created, destroyed, or modified (e.g., has its name changed).

To extract the communicator list, the debugger iterates over all of the communicators in the process using code similar to the following C++ example taken directly from TotalView. The MPI debug support library has been encapsulated in the C++ `dll` object, whose methods are trivial wrappers for the functions in the debug DLL itself.

```
/* Iterate over each communicator displaying the messages */
mqs_communicator comm;

for (dll->setup_communicator_iterator (process);
     dll->get_communicator (process, &comm) == mqs_ok;
     dll->next_communicator(process))
{
  /* Do something on each communicator, described by comm */
}
```

To extract information about a specific message queue in the currently selected communicator, code like this (again taken with only trivial edits from TotalView) is used.

```
int errcode = dll->setup_operation_iterator (process, which_queue);

if (errcode != mqs_ok)
  return false;  /* Nothing to be done */

mqs_pending_operation op;
while (dll->next_operation (process, &op) == mqs_ok)
  {
      /* Display the information about the operation from op */
  }
```

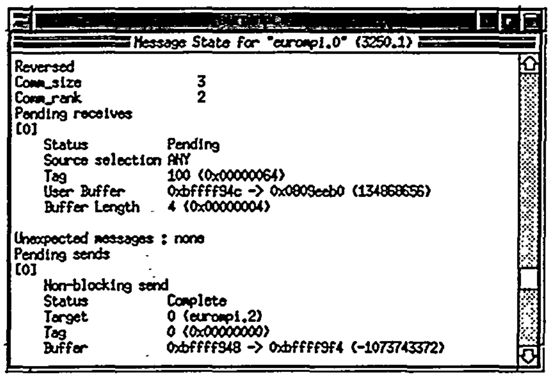
## 7 Implementation of the Debug DLL for MPICH

To provide a send queue for the debugger, we added a small amount of code to the MPICH implementation to maintain a list of `MPI_Requests` associated with MPI Send operations; completion of these operations (with `MPI_Test`, `MPI_Wait`, or the related multiple-completion routines) removes the request from the list. This extra list is conditionally compiled into MPICH when MPICH is configured with the `--enable-debug` switch. The run-time cost of this change is minimal when the process is not being debugged, amounting to one additional conditional branch in each of the immediate send operations, and in each of the test or wait operations when completing a request created by an immediate send.

The rest of the implementation of the debug DLL uses existing MPICH data structures; changes in MPICH internals are invisible to the debugger. For example, a change in `MPI_Request` to handle requests freed by the user before the operation completed only required rebuilding the DLL (along with the MPICH library); this change appeared in the next release of MPICH without requiring any change in TotalView.

## 8 Use of the Message Queue Data

Message queue data extracted from an MPI program can be used in many ways. The most basic is just to display it to the user, as in Fig. 1.



```
Message State for "eurompi.0" (3250.1)
Reversed
Comm_size             3
Comm_rank             2
Pending receives
[0]
    Status            Pending
    Source selection  ANY
    Tag               100 (0x00000064)
    User Buffer       0xbffff94c -> 0x0809eeb0 (134868656)
    Buffer Length     4 (0x00000004)

Unexpected messages : none
Pending sends
[0]
    Non-blocking send
    Status            Complete
    Target            0 (eurompi.2)
    Tag               0 (0x00000000)
    Buffer            0xbffff948 -> 0xbffff9f4 (-1073743372)
```

**Fig. 1.** Sample display generated by TotalView using the interface with MPICH

The data can also be used as input to tools that give higher-level information; for instance, Chan [5] describes a "Wait state graph" tool that uses the information extracted from an MPI implementation through this interface to show the user the message dependencies in an MPI program.

## 9 Availability of Implementations

In addition to the portable implementation that supports MPICH, implementations of libraries that conform to this interface have been written to support the proprietary MPI implementations of Compaq, IBM, and SGI.

The source code for the interface header and the MPICH implementation of the debug DLL are distributed with the MPICH source code, available from `http://www.mcs.anl.gov/mpi/mpich`. These files are open source; all rights are granted to everyone.

The code contains descriptions of other functions that have not been discussed here for lack of space, including conversion of error codes to strings and support for checking that the DLL and the client debugger are implementing compatible versions of the interface.

## 10 Conclusions

The interface described has proved to be useful on many systems. It successfully separates an MPI enabled debugger from the details of the MPI implementation and provides sufficient hooks to allow the debug DLL also to be written in a portable manner.

While the interface will require extensions to handle some of the features of MPI-2 (for instance, an index into `MPI_COMM_WORLD` will no longer be sufficient to identify a process), and other useful extensions such as the ability to display MPI data types, groups, or MPI-2 remote memory access windows are possible, the functionality already provided is extremely useful.

The interface is publicly available, and we encourage both debugger and MPI implementors to fetch the files and use the interface.

## References

1. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
2. Kenjii Morimoto, Takashi Matsumoto, and Kei Hiraki. Implementing MPI with the memory-based communication facilities on the SSS-CORE operating system. In Vassuk Alexandrov and Jack Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998. 5th European PVM/MPI Users' Group Meeting.
3. Etnus, Inc. TotalView User's Manual. Available from `http://www.etnus.com`.
4. Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.
5. Bor Chan. `http://www.llnl.gov/sccd/lc/DEG/TV-LLNL-Rel-3.html#waittree`.