Tools and Debugging Interfaces to MPI

Version 1.0

MPI Forum Working Group on Tools
Accepted by the Message Passing Interface Forum
(date tbd.)

Acknowledgments

Author
John Hancock

Contributing Authors
Furby F. Furby, Godzilla, and Katniss

Editor
Octocat

Reviewers
Murphy

Contents

Chapter 1

Introduction

A wonderful introduction will be here.

Put a citation here so that bibtex is happy: [1]

Chapter 2

Background

Need some background here.

Chapter 3

Overview

The message queue interface is used by tools and debuggers to extract information describing the conceptual message-passing state of the MPI application so that this can be displayed to the user.

Within each MPI communication space, there are three distinct message queues, which represent the MPI subsystem. They are:

1. Send Queue: represents all of the outstanding send operations.
2. Receive Queue: represents all of the outstanding receive operations.
3. Unexpected Message Queue: represents all the messages that have arrived at the process, but have not been received yet.

The send and receive queues store information about all of the unfinished send and receive operations that the process has started within the commnunicator. 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 funciton call that tinitiated the operation. Non-blocking 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 unepxected 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 (e.g., the datatype that was used by the sender). 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. The number of actual *queues* is implementation dependent. The interface described here addresses how to extract these conceptual queues from the imlementation so that they can be presented to the user independently of the particular MPI implementation. For example, the MPICH implemetation of MPI maintains only two queues, the Receive Queue and the Unexpected Message Queue. There is no explicit queue of send operations; instead all of the information about an incomplete send operation is maintained in the associated `MPI\_Request`. *I don't think this is true anymore within MPICH. If compiled with HAVE_DEBUGGER_SUPPORT, MPICH chains the send requests into a list, but Dave can confirm.*

Chapter 4

Definitions

4.1 MPI Process Definition

An MPI process is defined to be a process that is part of the MPI application as described in the MPI standard.

In this document, the rank of a process is assumed to be relative to MPI_COMM_WORLD (recall that this version of the MPIR interface does not support MPI-2 dynamic processes). For example, the phrase "MPI rank 0 process" denotes the process that is rank 0 in MPI_COMM_WORLD.

4.2 "Starter" Process Definition

The starter process is the process that is primarily responsible for launching the MPI job. The starter process may be a separate process that is not part of the MPI application, or the MPI rank 0 process may act as a starter process. By definition, the starter process contains functions, data structures, and symbol table information for the MPIR Process Acquisition Interface.

The MPI implementation determines which launch discipline is used, as described in the following subsections.

The MPI Rank 0 Process as the Starter Process

The MPICH-1 p4 channel is implemented such that the MPI rank 0 process launches the remaining MPI processes of the MPI application. In the MPICH-1 p4 channel implementation, the MPI rank 0 process is the starter process.

A Separate mpiexec as the Starter Process

Most MPI implementations use a separate `mpiexec` process that is responsible for launching the MPI processes. In these implementations, the `mpiexec` process is the starter process. Note that the name of the starter process executable varies by implementation; `mpirun` is a name commonly used by several implementations, for example. Other names include (but are not limited to) `srun` and `prun`.

4.3 MPIR Node Definitions

For the purposes of this document, the host node is defined to be the node running the tool process, and a target node is defined to be a node running the target application processes the tool is controlling. A target node might be the host node, that is, the target application processes might be running on the same node as the tool process.

Chapter 5

Debugger/MPI Interaction Model

The debugger will have access to the message queue functionality by loading a shared library provided by the MPI implementation. This allows the debugger to be insulated from the internals of the MPI library so that it can support multiple MPI implementations. Furthermore, MPI implementations can provide their users with debugging support without requiring source access to the debugger. The debugger learns about the location of this shared library by reading variable `mpimsgq_dll_locations` from the MPI Starter Process. The symbol is guaranteed to be NULL before the MPI Starter Process initializes it with a list of shared library names that it provides. The debugger should search this list and find one that is compatible to it (e.g., 32 bit vs 64 bit). More specifically:

1. At any time, the debugger searches for the public symbol `mpimsgq_dll_locations` in the MPI starter process (type (char**))
2. If the symbol is found and the symbol's value is NULL, try again later (meaning: the MPI implementation has not yet filled in relevant information)
3. If the symbol's value is non-NULL, the debugger goes through the NULL-terminated filenames in the string array (the last entry in the array will be NULL) and tries to dynamically load the DLL filename. This step assumes that `dlopen()` (or equivalent) will safely fail to load any DLL that is not suitable for the current platform (e.g., wrong endian, wrong bitness, wrong OS, ...etc.). If the load is successful and the DLL is suitable (e.g., the debugger can check the `mqs_version_string()` and `mqs_version_compatibility()` outputs), the debugger can continue with its logic. Otherwise (the load is unsuccessful), the debugger should continue down the list of dll names and repeat the loading process for each DLL name until it is successful.
4. If the symbol was not found, or if none of the DLLs was found to be suitable, the debugger should search for the public symbol `MPIR_dll_name` in the MPI Starter Process' process space and attempt to load the DLL name provided by that symbol.

Rationale. While the Message Queue Display interface has not been standardized, many MPI implementations and tool/debuggers have been relying on the existing mechanism of using the variable `MPIR_dll_name`. However, this mechanism limits the DLL names that can be chosen at compile-time, and is usually a DLL that is the same bitness as the installed MPI. When the debugger bitness is different from that of the MPI application it is debugging, user level workaround is usually required so that the proper DLL can be loaded. (*End of rationale.*)

All calls to the debug DLL from the debugger are made from 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 intialization entrypoint of the debug DLL. This procedure ensures that the debug DLL is independent of the specific debugger from which it is being called.

Chapter 6

Interface Specifications

Unless otherwise noted, all definitions are required and are provided in the interface header file.

6.1 mpimsgq_dll_locations

Global variable definition:

```
char* mpimsgq_dll_locations
```

> Definition is required.
> Definition is contained within the address space of the starter process.
> Variable is written by the starter process, and read by the tool.

`mpimsgq_dll_locations` is a argv-style array of DLL names populated by the starter process. The last entry of the array must be NULL. The names indicate the location of the list of DLLs provided by the MPI implementation that provide the message queue functionality. The debugger/tool can iterate this list to find a suitable shared library.

6.2 mqs_tword_t

`mqs_tword_t` is a target independence typedef name that is the appropriate type for the DLL to use on the host to hold a target word (`long`).

6.3 mqs_taddr_t

`mqs_tword_t` is a target independence typedef name that is the appropriate type for the DLL to use on the host to hold a target address (`void*`)

6.4 mqs_target_type_sizes

Type definition:

```
typedef struct
{
```

```
  int short_size;
  int int_size;
  int long_size;
  int long_long_size;
  int pointer_size;
} mqs_target_type_sizes;
```

mqs_target_type_sizes is a type definition for a struct that holds the size of common types in the target address space. The debug DLL will use the callback mqs_get_type_-sizes_ft provided by the debugger, which takes a variable of type mqs_target_type_-sizes) and populate it with the size information that it has based on the target host.

6.5 Opaque Types Passed Through the Interface

The following three types are opaque type that are defined within the debugger and are exposed to the debug DLL as undefined typedef's. The debug DLL has no need to see the internal structure of this type, but merely uses them as keys to identify objects of interest, or to be passed back to the debugger through some callback.

1. mqs_image
2. mqs_process
3. mqs_type

The following two types are opaque types defined within the debugger and are cast to concrete types within the debug DLL for the debug DLL's internal processing.

1. mqs_image_info

6.6 mqs_process_info

6.7 Constants and Enums

6.7.1 mqs_lang_code

```
typedef enum {
  mqs_lang_c     = 'c',
  mqs_lang_cplus = 'C',
  mqs_lang_f77   = 'f',
  mqs_lang_f90   = 'F'
} mqs_lang_code;
```

This enum is used by both the debug DLL and the debuger to deal with the different language type that the original target code was based on.

6.7.2 mqs_interface_version

This constant defines the version of the interface header

6.7.3 mqs_result

```
typedef enum {
  mqs_ok = 0,
  mqs_no_information,
  mqs_end_of_list,
  mqs_first_user_code = 100
}mqs_result;
```

This enum defines the various result code for the message queue functionality

6.7.4 mqs_error

```
enum
{
  MQS_INVALID_PROCESS = -1
};
```

This constant provides a value for the debugger to return error indicating an invalid process index.

6.7.5 mqs_op_class

```
typedef enum
{
  mqs_pending_sends,
  mqs_pending_receives,
  mqs_unexpected_messages
} mqs_op_class;
```

This enum is used by the debugger to indicate which queue it is interested in.

6.7.6 mqs_status

```
typedef enum
{
  mqs_st_pending, mqs_st_matched, mqs_st_complete
} mqs_status;
```

This enum is used to indicate the status of a message in the message queue.

6.8 Concrete Objects Passed Through the Interface

6.8.1 mqs_communicator

Type definition:

```
typedef struct
{
  mqs_taddr_t unique_id; /* A unique tag for the communicator */
```

```
  mqs_tword_t local_rank;/* The rank of this process Comm_rank */
  mqs_tword_t size;      /* Comm_size  */
  char        name[64];  /* the name if it has one */
} mqs_communicator;
```

6.8.2 mqs_pending_operation

Type defintion:

```
typedef struct
{
  mqs_status  status;
  mqs_tword_t desired_local_rank;
  mqs_tword_t desired_global_rank;
  int         tag_wild;
  mqs_tword_t desired_tag;
  mqs_tword_t desired_length;
  int         system_buffer;
  mqs_taddr_t buffer;

  /* Fields valid if status >= matched or it is a send */
  mqs_tword_t actual_local_rank;
  mqs_tword_t actual_global_rank;
  mqs_tword_t actual_tag;
  mqs_tword_t actual_length;

  char extra_text[5][64];
} mqs_pending_operation;
```

This structure contains enough information to allow the debugger to provide the user with details about both of the arguments to a receive and of the incoming message that matched it. All refereces 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

6.9 Callbacks Provided by the Debugger

6.9.1 mqs_basic_callbacks

Type definition:

```
typedef struct mqs_basic_callbacks
{
  mqs_malloc_ft           mqs_malloc_fp;
  mqs_free_ft             mqs_free_fp;
  mqs_errorstring_ft      mqs_errorstring_fp;
  mqs_put_image_info_ft   mqs_put_image_info_fp;
  mqs_get_image_info_ft   mqs_get_image_info_fp;
```

```
  mqs_put_process_info_ft mqs_put_process_info_fp;
  mqs_get_process_info_ft mqs_get_process_info_fp;
} mqs_basic_callbacks;
```

mqs_malloc_ft

mqs_free_ft

mqs_errorstring_ft

mqs_put_image_info_ft

mqs_get_image_info_ft

mqs_put_process_info_ft

mqs_get_process_info_ft

6.9.2 mqs_image_callbacks

Type definition:

```
typedef struct mqs_image_callbacks
{
  mqs_get_type_sizes_ft   mqs_get_type_sizes_fp;
  mqs_find_function_ft    mqs_find_function_fp;
  mqs_find_symbol_ft      mqs_find_symbol_fp;
  mqs_find_type_ft        mqs_find_type_fp;
  mqs_field_offset_ft     mqs_field_offset_fp;
  mqs_sizeof_ft           mqs_sizeof_fp;
} mqs_image_callbacks;
```

mqs_get_type_sizes_ft

mqs_find_function_ft

mqs_find_symbol_ft

mqs_find_type_ft

mqs_field_offset_ft

mqs_sizeof_ft

6.9.3 mqs_process_callbacks

Type definition:

```
typedef struct mqs_process_callbacks
{
  mqs_get_global_rank_ft       mqs_get_global_rank_fp;
  mqs_get_image_ft             mqs_get_image_fp;
  mqs_fetch_data_ft            mqs_fetch_data_fp;
  mqs_target_to_host_ft        mqs_target_to_host_fp;
} mqs_process_callbacks;
```

mqs_get_global_rank_ft

mqs_get_image_ft

mqs_fetch_data_ft

mqs_target_to_host_ft

6.10 Callbacks Provided by the DLL

6.10.1 mqs_setup_basic_callbacks

6.10.2 mqs_version_string

6.10.3 mqs_version_compatibility

6.11 Miscellaneous

6.12 mqs_dll_taddr_width

6.13 mqs_dll_error_string

6.14 Executable Image Related Functions

6.14.1 mqs_setup_image

Setup debug information for a specific image, this must save the callbacks, and use those functions for accessing this image. The DLL should use the `mqs_put_image_info` and `mqs_get_image_info` functions to associate the information it wants to keep with the image. The debugger will call `mqs_destroy_image_info` when it no longer wants to keep information about the given executable. This will be called once for each executable image in the parallel program.

6.14.2 mqs_image_has_queue

This function returns whether this image have the necessary symbols to allow access to the message queue. This function is called once for each image, and the information cached within the debugger.

6.15 Query Functions

ANH: DO WE REALLY NEED THESE?

These functions provide the message queue query functionality. The model here is that the debugger calls down to the library to initialize an iteration over a specific class of things, and then keeps calling the "next" function until it returns `mqs_false`. For communicators the stepping is separated from extracting information, because the debugger will need the state of the communicator iterator to qualify the selections of the operation iterator. `mqs_true` is returned when the description has been updated; `mqs_false` means there is no more information to return, and therefore the description contains no useful information. There is only one of each type of iteration running at once, so the library should save the iteration state in the `mqs_process_info`.

6.15.1 mqs_update_communicator_list

6.15.2 mqs_setup_communicator_iterator

6.15.3 mqs_get_communicator

6.15.4 mqs_get_comm_group

6.15.5 mqs_next_communicator

6.15.6 mqs_setup_operation_iterator

6.15.7 mqs_next_operation

Chapter 7

The MPI Handle Introspection Interface

A wonderful chapter will be here.

Bibliography

[1] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing Interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.