The MPIR Process Acquisition Interface

Written by: John V. DelSignore, Jr.
Last updated: August 6, 2010

Contents

1 Background

Back in the mid-1980s, parallel programming techniques for HPC were still evolving, but by the early 1990s an effort to produce a standard message passing interface was under way, and the first MPI standard was completed in May 1994. The TotalView debugger, an already mature parallel debugger for the BBN Uniform System and Oakridge National Laboratory's Parallel Virtual Machine (PVM), was quickly adapted to support debugging MPI applications.

In early 1995, TotalView's Jim Cownie and Argonne National Laboratory's Bill Gropp and Rusty Lusk decided to join forces and develop debugging interfaces for use with MPICH, one of the first widely available MPI implementations. Two interfaces were developed: one for process discovery and one for message queue extraction. Coined the "MPIR" interfaces, the MPI debugging interfaces eventually became de facto standards implemented by various MPI providers such as Compaq, HP, IBM, LAM, MPI Software Technologies, Open MPI, Quadrics, SCALI, SGI, and other implementations of MPI.

Even though the MPIR debugging interfaces are still widely used today by a number of MPI implementations and tool vendors, MPIR has not yet been standardized. Bill Gropp writes, "...there never was a formal 'MPIR' spec for the process discovery - there was a hack that was created for the first prototype implementation with MPICH and the ch_p4 device, but this was never intended to be a standard. Unfortunately, like so many prototypes, since there wasn't a standard, this part of the implementation was reverse-engineered into other implementations." In addition to being implemented by many MPI vendors, the MPIR Process Acquisition Interface used for process discovery has been extended by some vendors to better suit their needs.

This document describes the *state of the field* for the MPIR Process Acquisition Interface. It describes how the MPIR Process Acquisition Interface is currently used by several MPI implementations and tools.

2 MPIR Process Acquisition Interface Overview

The **MPIR Process Acquisition Interface**, also known as the **MPI Automatic Process Acquisition (MPI APA) Interface**, is used by tools such as debuggers and performance analyzers to locate MPI processes that are part of an MPI job. The tool can then automatically attach to the MPI processes in the job with no additional information required from the tool user. The MPI APA interface supports both launching an MPI job under the control of a tool and attaching a tool to an already running job.

The MPI APA interface is *not* an application programming interface (API). It is a rendezvous protocol used between the tool (such as a debugger or performance analyzer) and the MPI implementation. The MPI APA interface requires that the tool read symbol table information and trace the starter process, including starting and stopping the process, reading the memory and registers of the starter process, planting breakpoints, and handling events. MPI APA defines the starter process as the process that contains information about the location of the MPI processes. The starter process may or may not be an MPI process itself.

The MPI APA interface provides three fundamental pieces of information about each MPI process in an MPI job, as follows:

1. The location of the process in the form of a name that is resolvable to an IP address or node number.
2. The name of the executable the MPI process is running.
3. The process ID of the MPI process.

Collectively for each process, this information is known as the MPIR process descriptor. The MPI job control runtime system gathers the MPIR process descriptors into a single MPIR process descriptor table that is located in the memory of the starter process.

The MPI runtime system raises an event to the tool by setting an integer global variable and calling a breakpoint function in the starter process. The defined set of events is limited to MPI job spawn and abort. When the tool receives an MPI job spawn event, it reads the MPIR process descriptor out of the starter process and attaches to MPI processes.

Under the MPIR interface, the tool does not create the parallel job. The MPI job control runtime system creates the job and the tool attaches to the MPI processes *after* the processes have been created.

The MPIR interface also allows the MPI process to specify the path to the MPIR Message Queue Display (MQD) shared library, which allows the tool to display the state of the message queues in the MPI processes.

3 MPIR Process Acquisition Interface Definitions

This section contains definitions of terms used in the MPIR Process Acquisition Interface.

3.1 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 in **MPI_COMM_WORLD**. For example, "the MPI rank 0 process" means the MPI process that is rank 0 in **MPI_COMM_WORLD**.

3.2 "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.

3.2.1 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 MPICH-1 p4 channel implementation, the MPI rank 0 process is the starter process.

3.2.2 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 and "mpirun" is a name commonly used by several implementations. Other names include "orterun", "srun", and "prun".

3.3 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.

4 MPIR Process Acquisition Interface Requirements

The MPIR Process Acquisition Interface requires the tool to contain several capabilities typically found in a debugger.

4.1 Symbol Table Requirements

The MPIR Process Acquisition Interface requires the starter process contain symbol table information for the functions, data structures, and types defined by the interface. The symbol table information must be contained within the starter process executable or a shared library used by the starter process. If the implementation places the symbol table information in a shared library, the shared library may either be loaded as a requirement of the starter process executable, or dynamically loaded at runtime by the MPIR starter process.

The symbol table requirements are as follows:

- The starter process compilation unit(s) containing the functions, data structures, and types defined by the interface must be built with symbolic debugging information (for example, on Linux, that typically means compiling with the "-g" option).
- The starter process implementation must ensure that the functions, data structures, types, and symbol table information are not discarded as a result of compiler or linker optimizations.
- The packaging, distribution, and installation procedures for the starter process implementation must ensure that the symbol table information is not stripped, separated or otherwise discarded.

4.2 Debugger-Level Process Control Requirements

The debugger-level process-control requirement is as follows:

- The MPIR Process Acquisition Interface requires that the tool be able to exercise debugger-level control over the starter process. The tool is required to be able control the execution, read and write the address space, plant breakpoints, and handle breakpoint events in the starter process.

5 MPIR Process Acquisition Interface Limitations

The MPIR Process Acquisition Interface has the following limitation:

- The MPIR Process Acquisition Interface does not support the dynamic process creation or communication facilities of MPI 2, such as **MPI_Comm_spawn**, **MPI_Comm_accept**, or **MPI_Comm_connect**.

6 MPIR Process Acquisition Interface Models

The MPIR Process Acquisition Interface supports several process acquisition and synchronization models to accommodate various MPI implementations and tools deployment scenarios, as described in the following subsections.

6.1 MPIR Job Launch and Attach Models

The MPIR Process Acquisition Interface supports both launching an MPI application under the control of the tool, and attaching the tool to an already running MPI application.

Under the MPIR **job launch model**, the tool is invoked on the starter process executable, which in turn starts the MPI application. Consider the following example command, where **tool** is the tool executable, **mpiexec** is the starter process executable and **mpiapp** is the MPI application executable:

```
$ tool tool-args mpiexec mpiexec-args mpiapp
$
```

Under the MPIR **job attach model**, the MPI job is already running when the tool is attached to the starter process, as shown in the following example commands:

```
$ mpiexec mpiexec-args mpiapp &
[Shell prints the process ID "pid" of the background mpiexec process]
$
```

Some amount of time elapses, and the tool is attached to the mpiexec process (perhaps because the job is hung), as follows:

```
$ tool tool-args –pid pid mpiexec
$
```

6.2 MPI Process Synchronization Models

Under the job launch model the MPI implementation must ensure that the MPI processes do not return from **MPI_Init**. This requirement guarantees that the tool can acquire the MPI processes early in their lifetime.

The MPIR Process Acquisition Interface provides an optional interface (see the description of MPIR debug gate variable named **MPIR_debug_gate (**§9.8**)**) that allows the tool to synchronize with the start up of the MPI processes. The goal of the MPIR debug gate is to prevent the MPI processes from "running away," before the tool has a chance to attach to them.

An MPI implementation is not required to use the **MPIR_debug_gate** variable for synchronization. In fact, implementations are encouraged to use a synchronization technique that does *not* involve the use of **MPIR_debug_gate** in the MPI processes. Other synchronization techniques include the following:

- The MPI processes form a barrier with the starter process. During startup, the MPI processes are created and allowed to run to barrier. The starter process joins the barrier, but only *after* it has set **MPIR_debug_state (**§9.6) to

MPIR_DEBUG_SPAWNED (§9.6) and called the **MPIR_Breakpoint** (§9.9) function to notify the tool of the job spawn event.

- The starter process arranges for the MPI processes to be created in a stopped state, before they have executed any user-mode instructions. For example, on Posix-like systems, the MPI processes may be stopped on exit from execve. After the starter process has set **MPIR_debug_state** to **MPIR_DEBUG_SPAWNED** and called the **MPIR_Breakpoint** function to notify the tool of the job spawn event, the MPI processes are continued.

When an implementation uses a synchronization technique that does not require the tool to set **MPIR_debug_gate**, and does not require the tool to attach to and continue the MPI process, it should define the symbol **MPIR_partial_attach_ok (**§9.13) in the starter process. If possible, an MPI implementation that does not require the tool to set **MPIR_debug_gate** should avoid defining **MPIR_debug_gate** in the MPI processes.

7 MPIR Process Acquisition Interface Extensions

MPIR has been extended by some vendors, as described in the following subsections.

7.1 Tool Daemon Launch Extension

The MPIR Process Acquisition Interface does not specify how the tool launches its daemons, and does not support tool daemon launch.

However, the interface has been extended on IBM Blue Gene systems to support tool daemon launch, and that extension has also been implemented in Open MPI. The extension provides support only for tool daemon launch, not for communication between the tool and its daemons; the tool is responsible for establishing its own communication channels.

The tool daemon launch extension adds two character array variables to the MPIR interface that are named **MPIR_executable_path** (§9.16) and **MPIR_server_arguments** (§9.17).

They are used as follows:

1. Immediately after launching or attaching to the starter process, the tool writes the path name of its tool daemon's executable file to the **MPIR_executable_path** variable, and writes the tool daemon's command line arguments to the **MPIR_server_arguments** variable.
2. The tool then sets **MPIR_being_debugged (**§9.3) to a non-zero value, and continues the starter process.
3. The starter process notices that the value of **MPIR_being_debugged** changed to a non-zero value, and launches the executable named by the **MPIR_executable_path** variable and passes the command line arguments contained in the **MPIR_server_arguments** variable.
4. Under the MPIR job launch model, the starter process also arranges for the MPI processes to be created when the tool daemon processes are created. The MPI job control runtime system must prevent the created MPI processes from running beyond the return from the application's call to MPI_Init. On Blue Gene, the MPI processes are created in a stopped state, and do not begin execution until after the starter process is continued from the job spawn event (see §9.6).
5. The starter process raises the job spawn event.
6. It is then the responsibility of the tool and its daemon to establish communications, and attach to the MPI processes.

See the description of the **MPIR_executable_path** and **MPIR_server_arguments** variables for more information.

7.2 Message Queue Display DLL Search Extension

The text from the follow Tex document should be integrated into this document:

https://svn.mpi-forum.org/svn/mpi-forum-docs/trunk/MPI-3.0/wg/tools/new-tools-chapter/chap-prof/symbol.tex

The text from that URL is as follows. Also see section 9.18.

```
\section{Locating Tool Interface Symbols}
\label{sec:symbol}

Several of the interfaces provided in this chapter are intended to be
used by external software development tools, not by MPI applications
themselves.  As such, the implementations of these interfaces may not
be included in MPI processes, meaning that the interfaces' symbols may
need to be located elsewhere, such as a dynamic shared object (DSO,
a.k.a., a ``plugin'').
%
However, the tool that wishes to make use of these interfaces
typically only has knowledge of the MPI process itself -- it may not
know where the supplemental tool interface implementations can be
found.

The MPI process therefore needs to provide location information as to
where the interface implementation(s) can be found.  The general
technique used by these interfaces is to provide a well-known global
\chgfeba{symbol}
in the MPI process that contains an array of DSO filenames.
%
An array is used because different tools may require different DSO
characteristics (e.g., one DSO compiled from 32 bit environments and
another compiled for 64 bit environments).

\begchangejan25
\begin{rationale}
  It is common for ``plugin'' types of interfaces to require DSOs to
  export a public symbol with a pre-defined / well-known name.  Tools
  that open plugins look for that symbol and use it as an initial
  point of entry.
  %
  MPI extends this idea by exporting a well-known symbol for each
  interface {\em type} that is supported by the MPI implementation;
  this well-known symbol then provides an array of eligible DSO
  filenames.

  One alternative to this scheme would be to have a {\em single}
  well-known symbol that both defines the list of available interface
  types and provides a pointer to an array of DSO filenames for that
  type.  Regardless of how the filename arrays are provided, the tool
  needs to have semantic knowledge of what interface each DSO exports.
  The approach of using unique public symbols, one for each interface
  type, was chosen for simplicity.
\end{rationale}

\begin{implementors}
  Note that the DSO may not have the same characteristics as the MPI
  process itself.  For example, a parallel debugger may be a 64 bit
  application and therefore require a 64 bit DSO, even though the MPI
  process it is debugging is a 32 bit application.
\end{implementors}

The array is an ``argv-style'' array of character pointers.  If the
symbol is present in the MPI process, its value must either be
\const{NULL} (indicating that no DSO implementations are currently
available) or a valid array of character pointers.  If the value is
non-\const{NULL}, the last pointer in the array must be \const{NULL}
to indicate the end of the array.  The filename strings themselves are
\const{const}.

External tools can examine each file in the array in an attempt to
find a suitable DSO.  No filename conventions are imposed; the tool
will need to check the characteristics of the DSO itself to know
whether it is suitable.  For example, a tool can attempt to
dynamically load each of the files in the array; the file that loads
successfully is likely a good candidate to be used by the tool.
\endchangejan25
```

```
Array values may be filled in at any time during the life of the MPI
process; it is the tool's responsibility to check array values as
relevant to see if DSO implementations become available (or
unavailable).  This scheme allows the MPI implementation to search for
valid DSO implementations at run-time, perhaps during
\mpifunc{MPI\_\-INIT}.
%
The array symbol name depends on the type of interface; each
interface's global symbol name will be specified in its description
later in this chapter.

\begin{implementors}
\begchangejan25
  The following pseudocode example shows one possible implementation
  of filling in the \const{mpimsgq\_\-dll\_\-locations} DSO filename
  array (see Section \ref{sec:tools:introspection:msgqueue}):

  \exindex{Filling in {\tt mpimsgq\_\-dll\_\-locations}}

%%HEADER
%%SKIP
%%ENDHEADER
\begin{verbatim}
/* This implementation's DSO filename locations are not known at
   compile time, so it sets the array pointer to NULL. */
char **mpimsgq_dll_locations = NULL;
extern const char *DSO_FILENAME_32_BIT;
extern const char *DSO_FILENAME_64_BIT;

/* The following function is called during MPI_INIT. */
void function_called_during_mpi_init(void)
{
    FILE *fp;

    fp = fopen(DSO_FILENAME_32_BIT, "r");
    if (fp != NULL) {
        argv_add(&mpimsgq_dll_locations, DSO_FILENAME_32_BIT);
        fclose(fp);
    }

    fp = fopen(DSO_FILENAME_64_BIT, "r");
    if (fp != NULL) {
        argv_add(&mpimsgq_dll_locations, DSO_FILENAME_64_BIT);
        fclose(fp);
    }
}
\end{verbatim}
\endchangejan25
\end{implementors}
```

8 MPIR Process Acquisition Interface Use Case

Figure 1 shows a collaboration diagram of a typical MPI session under the control of a tool using the MPIR Process Acquisition Interface. Note that the interface can be used in several different ways, thus there are several different possible relationships. The collaboration diagram depicts one of the possible (and most common) relationships for an MPI implementation that uses a separate starter executable named "mpiexec" to launch a set of MPI tasks running an executable named "a.out".

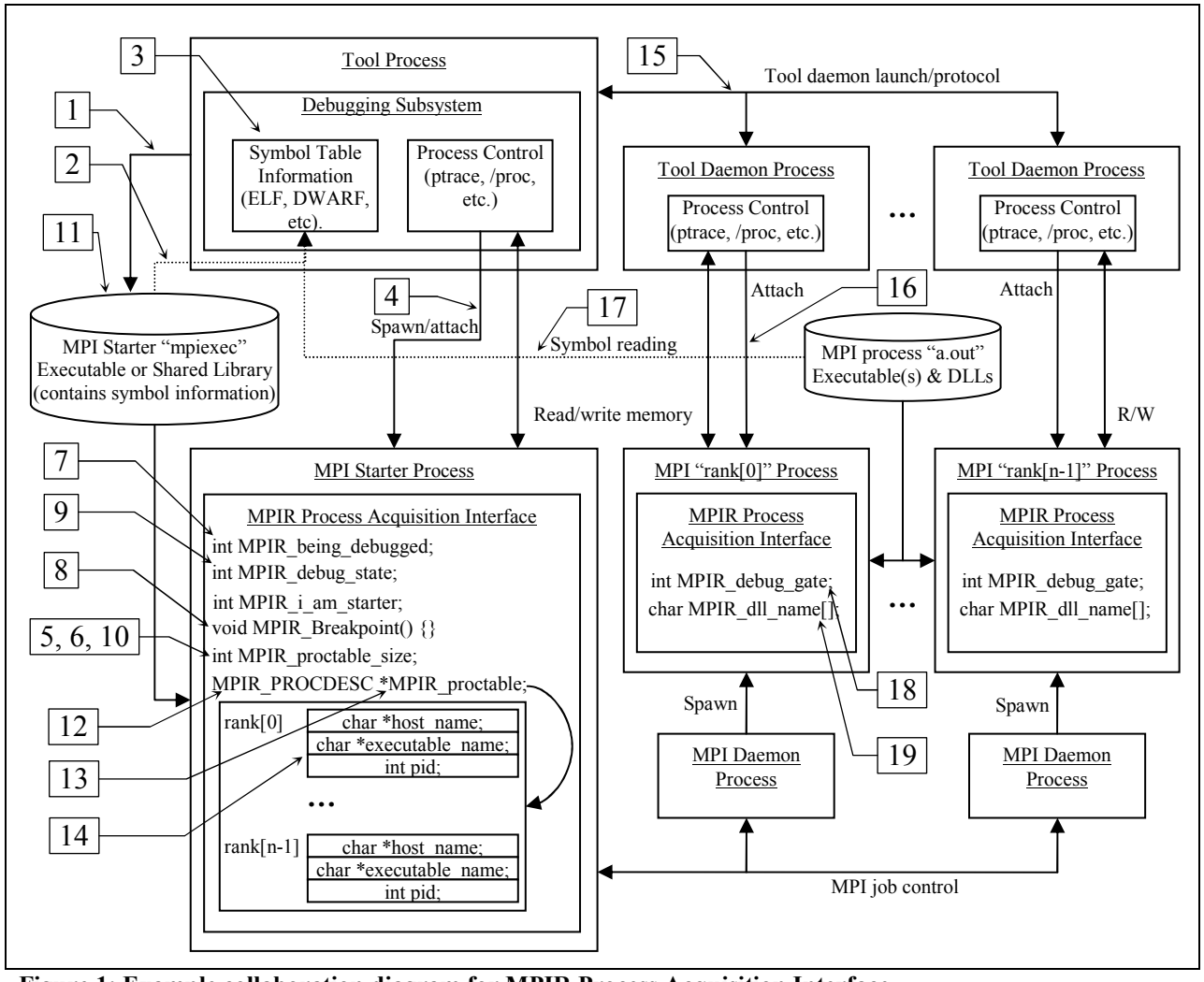

Figure 1: Example collaboration diagram for MPIR Process Acquisition Interface

Startup processing:

1. The tool is started on the "mpiexec" executable.
2. The debugging subsystem of the tool reads the symbol table information from the executable, and any shared libraries used by the executable.
3. The tool discovers the executable is a starter program by querying the symbol table for the **MPIR_Breakpoint** symbol. If the symbol exists, then the tool should consider this a starter program.
4. The tool spawns or attaches to the starter process, and arranges for the starter process to remain stopped after the spawn or attach operation completes.
5. The tool reads the value of **MPIR_proctable_size** (§9.5) out of the starter process to determine the number of MPI processes already created by the starter process. If the tool spawned the starter process, then this value is expected to be 0. However, if the tool attached to the starter process, then this value could be greater than zero.
6. If **MPIR_proctable_size** is greater than zero, then the tool should attach to any additional MPI processes that might exist. If the symbol **MPIR_i_am_starter** is *not* defined in the starter process, then the starter process *is* the MPI rank 0 process, thus the tool is already controlling the MPI rank 0 process.
7. The tool sets **MPIR_being_debugged** to 1 to inform the starter process that the debugger is present.
8. The tool plants a breakpoint at **MPIR_Breakpoint** to receive MPIR events (defined below). The tool may then continue the starter process.

Event processing:

9. If the starter process hits the breakpoint at **MPIR_Breakpoint** then by definition this raises an "MPIR event". The tool then reads the value of the **MPIR_debug_state** variable out of the starter process, and performs an action based on the variable's value, which typically results in the tool attaching to the MPI processes listed in the process descriptor table. See the description of **MPIR_debug_state** in section 9.6 for more information on MPIR events.

Process descriptor table processing:

10. The tool reads the value of **MPIR_proctable_size** variable out of the starter process to determine the number of MPI processes already created by the starter process. If this value is not greater than zero, then the table is empty.
11. The tool determines the size and member layout of the **MPIR_PROCDESC** (§9.2) process descriptor structure by reading the type information from symbol table of the starter process. The tool can find the process descriptor structure type information either using a lookup on the type name, or by looking up the **MPIR_proctable** (§9.4) variable and dereferencing the variable's type. The latter approach is more reliable because the name **MPIR_PROCDESC** is a typedef, which may not be available for by-name lookup.
12. The tool reads the value of the **MPIR_proctable** pointer variable out of the starter process. The pointer is the base address of the process descriptor table.

13. The tool reads the process descriptor table out of the starter process, starting at the **MPIR_proctable** pointer variable value. The length in bytes of the process descriptor table is the size of the **MPIR_PROCDESC** process descriptor structure in bytes multiplied by the value of **MPIR_proctable_size** variable.
14. The tool iterates over the process descriptor table information to identify the node name or IP address, executable path name, and process ID of each of the MPI processes.
15. The tool launches its tool daemons, if necessary, on the set of nodes being used by the MPI processes.
16. The tool attaches to the MPI processes.
17. The tool reads the symbol table information of the MPI processes.

MPI process attach:

18. If the symbol **MPIR_partial_attach_ok** is defined in the starter process, then this informs the tool that the initial startup barrier is implemented by the MPI system, and it is not necessary to set the **MPIR_debug_gate** variable in any of MPI processes. However, if the symbol **MPIR_partial_attach_ok** is *not* defined in the starter process, the tool must attach and set the **MPIR_debug_gate** variable to 1 in each MPI processes to release them from the gate, even if the tool user has instructed the tool to *not* attach to all of the MPI processes.
19. The tool reads the value of the **MPIR_dll_name** (§9.14) character string variable out of the MPI processes. The string is the path name of a shared library that the tool can dynamically load into its own address space to gather MPI message queue information for the MPI process. The MPI Message Queue Display (MQD) DLL is described in a separate document. As an optimization, the tool can assume that the value of the **MPIR_dll_name** character string variable is identical for all MPI processes that are running the same executable. For example, in a single program multiple data (SPMD) style program, all MPI processes are running the same executable so the tool may assume that the value of **MPIR_dll_name** is the same in each process. In a multiple program multiple data (MPMD) style program consisting of two executables named “a.out” and “b.out”, the tool may assume that all MPI processes executing “a.out” have the same **MPIR_dll_name** value, and all MPI processes executing “b.out” have the same **MPIR_dll_name** value, however the **MPIR_dll_name** value may differ between “a.out” and “b.out”.

9 MPIR Process Acquisition Interface Specification

The MPIR Process Acquisition Interface is specified as a set of C-language definitions. The following sections enumerate those definitions. Each subsection covers one definition, specifies if the definition belongs in the starter process or the MPI processes, states whether or not the definition is required, and describes how the definition is used.

9.1 VOLATILE

Macro definition:

```
#ifndef VOLATILE
#if defined(__STDC__) || defined(__cplusplus)
#define VOLATILE volatile
#else
#define VOLATILE
#endif
#endif
```

Definition is required.
Definition is used by the starter process and MPI processes.

The **VOLATILE** macro is defined to the **volatile** keyword for C++ and ANSI C. Declaring a variable **volatile** informs the compiler that the variable could be modified by an external entity and that its value can change at any time. **VOLATILE** is used with MPIR variables defined in the starter and MPI processes but are set by the tool.

9.2 MPIR_PROCDESC

Type definition:

```
typedef struct {
  char *host_name;
  char *executable_name;
  int   pid;
} MPIR_PROCDESC;
```

Definition is required.
Definition is contained within the symbol table of the starter process.

MPIR_PROCDESC is a **typedef** name for an anonymous structure that holds process descriptor information for a single MPI process. The structure must contain three members with the same names and types as specified above. The tool must use the symbol table information to determine the overall size of the structure, and offset and size of each of the structure's members.

The **host_name** member is a pointer to a null terminated character string in the address space of the starter process that specifies the target node location of the MPI process in the form of a host name or IP address string that must be translatable to an IP address. On Beowulf Distributed Process Space (BProc) systems, the string is an integer node number (e.g., "42").

The **executable_name** member is a pointer to a null terminated character string in the address space of the starter process that specifies the name of the executable the MPI process is running.

The string should be the full path name to the executable, which may be relative to the host node or target node.

The **pid** member is the integer process identifier of the MPI process. Note that historically **pid** was defined as a C language **int**, which is an integer of an unknown size, which might be smaller than the size of a process identifier for the target system. Implementations should define **pid** as an integer size that is large enough to hold a process identifier for the target system. For example, implementations should use **pid_t** as the type of **pid** provided that **pid_t** is an integer type.

The MPI implementation should share the host and executable name character strings across multiple process descriptor entries whenever possible. For example, if all of the MPI processes are executing "/path/a.out", then the executable name field in each process descriptor should point to the same null terminated character string. Sharing the strings enhances the tool's scalability by allowing it to cache data from the starter process and avoid reading redundant character strings.

9.3 MPIR_being_debugged

Global variable definition:

```
VOLATILE int MPIR_being_debugged;
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is written by the tool, and read by the starter process.

MPIR_being_debugged is an integer variable that is set or cleared by the tool to notify the starter process that a tool is present.

The tool sets the variable to 1 immediately after spawning or attaching to the starter process. The tool sets the variable to 0 immediately before detaching from the starter process.

The starter process may monitor the state of the variable and perform certain operations differently. For example, this variable might control whether or not the starter process forces the MPI processes to wait for the **MPIR_debug_gate** to be set.

9.4 MPIR_proctable

Global variable definition:

```
MPIR_PROCDESC *MPIR_proctable;
```

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.

MPIR_proctable is a pointer variable set by the starter process that points to an array of **MPIR_PROCDESC** structures containing **MPIR_proctable_size** elements. This array of structures is the process descriptor table.

The index position in the process descriptor table is the rank of the process in **MPI_COMM_WORLD**. For example, index 0 in the table specifies rank 0 in **MPI_COMM_WORLD**, index 1 in the table specifies rank 1 in MPI_COMM_WORLD, and so forth.

9.5 MPIR_proctable_size

Global variable definition:

```
int MPIR_proctable_size;
```

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.

MPIR_proctable_size is an integer variable set by the starter process that specifies the number of elements in the procedure descriptor table pointed to by the **MPIR_proctable** variable.

9.6 MPIR_debug_state

Macro definitions:

```
#define MPIR_NULL            0
#define MPIR_DEBUG_SPAWNED   1
#define MPIR_DEBUG_ABORTING  2
```

Global variable definition:

```
int MPIR_debug_state;
```

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.
Variable's initial value should be 0 (MPIR_NULL).

MPIR_debug_state is an integer variable set by the starter process that specifies the state of the MPI job at the point where the starter process calls the **MPIR_Breakpoint** function. The starter process can raise an MPIR event in the tool by setting **MPIR_debug_state** and calling the **MPIR_Breakpoint** function.

The tool must set a breakpoint at the **MPIR_Breakpoint** function and read the value of the **MPIR_debug_state** variable to process an MPIR event. The following events are defined:

- If the value is **MPIR_NULL** (0), then the tool should ignore the event and continue the starter process.

- If the value is **MPIR_DEBUG_SPAWNED** (1), then the starter process has spawned the MPI processes and filled in the process descriptor table. The tool can attach to any additional MPI processes that have appeared in the process descriptor table. This is known as a "job spawn event".
- If the value is **MPIR_DEBUG_ABORTING** (2), then the MPI job has aborted and the tool can notify the user of the abort condition. The tool can read the reason for aborting the job by reading the character string out of the starter process, which is pointed to by the **MPIR_debug_abort_string** variable in the starter process.

The tool may continue or leave the starter process stopped after processing the event. The tool decides when to continue the starter process. For example, a debugger may allow the user to control the execution of the starter process in conjunction with controlling the execution of the MPI processes. However, a performance analyzer might automatically continue the starter process after processing the MPIR event. Note that some MPI implementations may require that the starter process be running while one or more of the MPI processes are running. Therefore, the tool may be required to implicitly continue the starter process when any of the MPI processes are continued.

9.7 MPIR_debug_abort_string

Global variable definition:

```
char *MPIR_debug_abort_string;
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is written by the starter process, and read by the tool.

MPIR_debug_abort_string is a pointer to a null-terminated character string set by the starter process when MPI job has aborted. When an **MPIR_DEBUG_ABORTING** event is reported, the tool can read the reason for aborting the job by reading the character string out of the starter process. The abort reason string can then be reported it to the user, and is intended to be a human readable string.

9.8 MPIR_debug_gate

Global variable definition:

```
VOLATILE int MPIR_debug_gate;
```

Definition is *not* required.
Definition is contained within the address space of the MPI processes.
Variable is written by the tool, and read by the MPI processes.

MPIR_debug_gate is an integer variable that is set to 1 by the tool to notify the MPI processes that the debugger has attached. An MPI process may use this variable as a synchronization mechanism to prevent it from running away before the tool has time to attach to the process.

An MPI implementation is not required to use the **MPIR_debug_gate** variable for synchronization. However, the MPI job control runtime system must prevent the created MPI processes from running beyond the return from the application's call to **MPI_Init**.

9.9 MPIR_Breakpoint

Global subroutine definition:

```
void MPIR_Breakpoint() {}
```

Definition is required.
Definition is contained within the address space of the starter process.

MPIR_Breakpoint is the subroutine called by the starter process to notify the tool that an MPIR event has occurred. The starter process must set the **MPIR_debug_state** variable to an appropriate value before calling this subroutine. The tool must set a breakpoint at the **MPIR_Breakpoint** function, and when a thread running the starter process hits the breakpoint, the tool must read the value of the **MPIR_debug_state** variable to process an MPIR event.

9.10 MPIR_i_am_starter

Global variable definition:

```
int MPIR_i_am_starter;
```

Definition is required when the MPIR starter process is *not* also an MPI process.
This symbol must *not* be defined if the process is an MPI process.
Definition is contained within the address space of the starter process.
Variable is neither read nor written.

MPIR_i_am_starter is a symbol of any type (preferably int) that marks the process containing the symbol definition as a starter process that is *not* also an MPI process. This symbol serves as a flag to mark the process as a separate starter process or an MPI rank 0 process.

If MPI rank process 0 is the starter process, this symbol must *not* be defined.

9.11 MPIR_acquired_pre_main

Global variable definition:

```
int MPIR_acquired_pre_main;
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is neither read nor written.

MPIR_acquired_pre_main is a symbol of any type (preferably int) that informs the tool that under the MPI job launch model the MPI processes are stalled or stopped before entering the main subprogram (**main** in the C language).

If the symbol is defined, the tool may assume that the MPI processes to which it is attaching are stopped upon creation (for example, on exit from the **execve** system call or inside a shared library's init section code) and have not yet entered the main subprogram.

The presence or absence of this symbol may cause the tool to behave differently. For example, if this symbol is present, a debugger may choose to the display the source code of the main subprogram after acquiring the MPI processes during an MPI job launch startup. If the symbol is absent, then a normal display showing the place at which the code was executing may be shown.

9.12 MPIR_force_to_main

Global variable definition:

```
int MPIR_force_to_main;
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is neither read nor written.

MPIR_force_to_main is a symbol of any type (preferably int) that informs the tool that it should display the source code of the main subprogram after acquiring the MPI processes. The presence of the symbol **MPIR_force_to_main** does *not* imply that the MPI processes have been stopped before dynamic linking has occurred.

9.13 MPIR_partial_attach_ok

Global variable definition:

```
int MPIR_partial_attach_ok;
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is neither read nor written.

MPIR_partial_attach_ok is a symbol of any type (preferably int) that informs the tool that the MPI implementation supports attaching to a subset of the MPI processes.

If the symbol **MPIR_partial_attach_ok** is present, then this informs the tool that the initial startup synchronization is implemented in such a way that the tool need not attach nor continue MPI processes that the user is not interested in controlling. For example, the MPI implementation synchronization startup may be implemented as a barrier, rather than by having each of the MPI processes hang in a loop waiting for the **MPIR_debug_gate** variable to be set by the tool.

Thus, the tool need only release the starter process to release the whole MPI job, which can therefore be run *without* requiring the tool to acquire all of the MPI processes included in the MPI job. This is useful in tools that include the possibility of attaching to processes later in the tool session (for example, by selecting only processes in a specific communicator, or a specific

process in **MPI_COMM_WORLD**). This method of operation is preferred because operating on a subset of processes is a valuable feature on high-scale systems.

The tool may choose to ignore the presence of the **MPIR_partial_attach_ok** symbol and acquire all MPI processes. The presence of this symbol does not prevent the tool from setting the **MPIR_debug_gate** variable (if defined), which should have no effect.

The type or value of this symbol is not tested by the tool. The presence or absence of this symbol is all that is tested.

9.14 MPIR_dll_name

Global variable definition:

```
char MPIR_dll_name[];
```

Definition is *not* required.
Definition is contained within the address space of the MPI processes.
Variable is written by the MPI process and read by the tool.
The use of this variable is deprecated.

MPIR_dll_name is a null-terminated character string that contains the pathname of a dynamically loadable message queue debugging shared library. The shared library is intended to be dynamically loaded into the address space of the tool itself. An MPI implementation can use this variable to override any default message queue debugging library the tool uses.

Unfortunately, this interface has the limitation that allows for naming only one library, thus on systems where various tools could be using different ABIs (e.g., 32-bit vs. 64-bit), there is no single right library name value. To fix this problem, use the interface described in the "Message Queue Display DLL Search Extension" section.

9.15 MPIR_ignore_queues

Global variable definition:

```
int MPIR_ignore_queues;
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is neither read nor written.

MPIR_ignore_queues is a symbol of any type (preferably int) that informs the tool that MPI message queues support should be suppressed. This is useful when the MPIR Process Acquisition Interface is being used in a non-MPI environment.

9.16 MPIR_executable_path

Global variable definition:

```
char MPIR_executable_path[256];
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is written by the tool, and read by the starter process.

MPIR_executable_path is a null-terminated character string that is written by the tool into the address space of the starter process. The string is the path name of the tool daemon's executable file on the target node.

When the tool then sets **MPIR_being_debugged** to a non-zero value and continues the starter process, the starter process notices that the value of **MPIR_being_debugged** changed to a non-zero value, and launches the executable named by the **MPIR_executable_path** variable and passes the arguments contained in the **MPIR_server_arguments** variable.

9.17 MPIR_server_arguments

Global variable definition:

```
char MPIR_server_arguments[1024];
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is written by the tool, and read by the starter process.

MPIR_server_arguments is a sequence of zero or more null-terminated character strings followed by a null character that is written by the tool into the address space of the starter process. Each null-terminated character string is passed as a single argument to the tool daemon. A null character terminates the list. It is not possible to pass the empty string ("") as an argument.

For example, the following C string contains four arguments:

```
"-callback\010.0.0.10\0-name\0has spaces and\ttabs\0\0"
```

The starter process should split the above string into an argv-style vector as follows:

```
argv[0] = "-callback"
argv[1] = "10.0.0.10"
argv[2] = "-name"
argv[3] = "has spaces and\ttabs"
argv[4] = 0
```

When the tool then sets **MPIR_being_debugged** to a non-zero value and continues the starter process, the starter process notices that the value of **MPIR_being_debugged** changed to a non-zero value, and launches the executable named by the **MPIR_executable_path** variable and passes the arguments contained in the **MPIR_server_arguments** variable.

9.18 mpimsgq_dll_locations

Global variable definition:

```
char **mpimsgq_dll_locations;
```

Definition is *not* required.
Definition is contained within the address space of the starter process.
Variable is written by the starter process, and read by the tool.

This section is incomplete and needs work. The following text was taken from the following URL. Also see section 7.2.

https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MPI3Tools/dllapi

```
Short Version ¶

The following is a proposal for a new way to find and load the debugger MPI plugins in parallel
debuggers (most of the text below was taken from prior e-mails on this topic). The new mechanism
does not conflict with the current mechanism, and can (and should) be used in conjunction with
the current mechanism to support legacy MPI's and debuggers. The new mechanism provides a
standardized way for an MPI to specify its debugger DLLs at run time in an attempt to find a MPI
DLL+debugger combination that works.


In short: the new mechanism has the MPI starter process provide an array of DLL filenames that is
suitable to be read when the symbol is both present in the MPI process and has non-NULL values.
The debugger scans this array and loads whatever DLL is appropriate (if any).
Longer Version ¶

The ability to have a more flexible mechanism for specifying the DLL is motivated by a few
issues:

    * The DLL composition/type is dependent upon the debugger, not the MPI implementation. For
example, the installed MPI implementation may have been compiled for 32 bit, but the debugger is
64 bit (and therefore need a 64 bit DLL).
    * An MPI implementation may have/find multiple possible available DLLs at run time.
    * The MPI may/will likely not know exactly what the debugger wants.

As such, it is a straightforward extension to allow the MPI implementation to provide an argv-
style array of DLL filenames in global variables for each DLL type (this MPI3 WG may add more DLL
types; see other proposals). If the MPI process has the appropriate global variable available, it
must have a NULL value before the MPI implementation has assigned an array of possible DLL
filenames to it. At any time after the variable becomes non-NULL, the debugger may read the list
of filenames and use them to find an available DLL.

Real world example: Sun has run into exactly this problem. Sun provides customers with both 32
and 64 bit versions of their MPI (ClusterTools 7.x / Open MPI). However, Open MPI's build system
is incapable of building a DLL that does not match the bitness of the MPI that is building --
hence, each installation of MPI in CT 7.x has its own debugger DLL and assumes that it should be
used (simply put: the 32 bit install assumes that the 32 bit DLL should be used; the 64 bit
install assumes that the 64 bit DLL should be used). But in cases where the bitness of the MPI
does not match that of the debugger, user-level workarounds must be used to force Sun's dynamic
loader to open the appropriate MPI message queue DLL in the debugger.

The same scenario applies to other heterogeneous environments, too -- where the nodes on which
the MPI processes are running on are different architectures, operating systems, etc.

To review, the current MPI message queue DLL-finding-and-loading mechanism consists of the
following simple mechanism:

   1. the debugger searches for the public symbol MPIR_dll_name in the MPI starter process (type
(char *))
   2. if found, the debugger attempts to dynamically load the DLL filename
   3. if that succeeds, the message queue functionality proceeds
```

However, this is limited to names that can be chosen at compile-time, and -- at least for portable MPI implementations like Open MPI -- is usually a DLL that is the same bitness as the installed MPI.

We would like to trivially extend this mechanism to allow for other possibilities, especially on platforms where the MPI application under debug (and therefore the DLL that is naturally paired with it) may not match the bitness of the debugger. This will allow for greater freedom of both MPI implementers and debugger ISVs -- less coordination will be required to ensure that a viable debugger+MPI DLL combination can be found. In short, the MPI starter process may provide an array of DLLs to the debugger, and the debugger can choose which -- if any -- are suitable.

The new mechanism is as follows:

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
    1. If the symbol's value is NULL, try again later (meaning: the MPI implementation has not yet filled in relevant information)
    2. 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)
        1. Try 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.).
        2. 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), break out of the loop
        3. If the load is unsuccessful, it's not an error -- just loop around and try the next one
    3. If one of the DLLs was found to be suitable, the message queue functionality proceeds
3. If the symbol was not found, or if none of the DLLs was found to be suitable, proceed with the current MPI_dll_name mechanism

In short, the MPI implementation gives the debugger a list of possible candidate DLLs and lets the debugger choose which one it wants.

NOTE: Since the debugger may asynchronously read the mpimsgq_dll_locations variable, the MPI implementation should probably only set this value after the array has been completely setup.

Note that there are no naming conventions placed on the filenames that are returned in the arrays; an MPI implementation is free to name its DLLs whatever it wants. The debugger will try all of them.

Also note that the new mpimsgq_dll_locations array is the "preferred" mechanism -- it is tested first (because it provides more flexibility). If it is not present or no viable DLLs are found, the old MPIR_dll_name symbol is checked, just like it is in the current mechanism.

This combination approach will allow for legacy debuggers and MPI implementations to continue to be supported, but provide new flexibility for MPI+debugger combinations that support the new mechanism.

The above mechanism solves Sun's problems because Open MPI can be overridden (at run time) to provide a list of all available message queue DLLs in the ClusterTools installation -- both 32 and 64 bit. All of these filenames are passed back to the debugger; the first one that is able to be dynamically loaded and found to be suitable is used. For example (contrived paths):

```
mpimsgq_dll_locations[0] = "/opt/ct7.1/lib/openmpi/libompitv.so";
mpimsgq_dll_locations[1] = "/opt/ct7.1/lib64/openmpi/libompitv.so";
mpimsgq_dll_locations[2] = NULL;
```

Note, too, that more complex scenarios are possible, where a simple string substitution in the filename is not possible (or would be basically an entire substitution, such as the entire $libdir, and is therefore undesirable), such as:

```
mpimsgq_dll_locations[0] = "/opt/ct7.1/lib/openmpi/libompitv.so";
mpimsgq_dll_locations[1] = "/alternate/location/lib64/openmpi/libompitv.so";
mpimsgq_dll_locations[2] = NULL;
```