The MPIR Interfaces

Written by: John V. DelSignore, Jr.
Last updated: February, 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 attempts to describe the state of the field for the MPIR Process Acquisition Interface and MPIR Message Queue Display Interface, and is intended to serve as the foundation for a more formal standards document for each interface.

2 MPIR Process Acquisition Interface

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 rank processes that are part of an MPI job. The tool can then automatically attach to the MPI rank 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.

2.1 MPIR Process Acquisition Interface Overview

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 MPI 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 MPI starter process as the process that contains information on the location of the MPI rank processes. The MPI starter process may or may not be an MPI rank process itself.

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

1. The location of the process in the form of a host name or IP address.
2. The name of the executable the process is running.
3. The process ID of the process.

Collectively for each process, this information is known as the MPIR process descriptor, and is stored in a structure named **MPIR_PROCDESC**.

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 MPI starter process. The MPI rank 0 program may define an integer global variable named **MPIR_i_am_starter** to identify itself as the MPI starter process.

The length of the process descriptor table is stored in an integer global variable named **MPIR_proctable_size**. The process descriptor table is an array of MPIR_PROCDESC structures pointed to a global variable named **MPIR_proctable**.

The MPI runtime system can raise an MPIR event to the tool by setting an integer global variable named **MPIR_debug_state** and calling a breakpoint function named **MPIR_Breakpoint** in the MPI starter process. The defined set of MPIR events is limited to MPI job spawn and abort. The character string global variable named **MPIR_debug_abort_string** can be set to the reason why MPI aborted.

The tool can inform the MPI starter process that it is being debugged by setting an integer global variable named **MPIR_being_debugged** to a non-zero value. By reading the state of this variable, the MPI starter process may perform special actions and maintain extra internal information to facilitate debugging.

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 rank processes *after* the processes have been created. It is the responsibility of the MPI job control runtime system to prevent the created MPI rank processes from running beyond the return from their call to **MPI_Init**. The MPIR interface provides a mechanism where the MPI rank processes can stall waiting for the tool to attach. After the tool attaches to a rank process, it will set the integer global variable **MPIR_debug_gate** to 1 to allow the rank process to proceed.

The MPIR interface also allows the MPI rank process to specify the path to the MPIR Message Queue Display (MQD) shared library. If the symbol **MPIR_dll_name** is defined in the executable or shared library used by the rank process, then it is a character string that is the path name of the message queue debugging library to use for message queue extraction. This can be used to override the default shared library path name the tool would otherwise choose.

2.2 MPIR Process Acquisition Interface Definitions

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

2.2.1 MPI "Starter" Process Definition

The MPI starter process is the process that is primarily responsible for launching the MPI application. The MPI 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 MPI 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 follows.

2.2.1.1 MPI Rank 0 as the Starter Process

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

2.2.1.2 A Separate "mpirun" as the Starter Process

Most MPI implementations are implemented using a separate "mpirun" process that is responsible for launching the MPI rank processes. In these implementations, the "mpirun" process is the MPI starter process.

2.2.2 MPI "Rank" Process Definition

An MPI rank process is defined to be a process that is part of the MPI application and one that has a rank in MPI_COMM_WORLD.

2.2.3 MPIR Node Definitions

By definition, a **node** is a machine that is running a single operating system instance. For the purposes of this document, a **host node** is defined to be the node running the tool process, and the **target node** is defined to be the node running the target application processes 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.

2.3 MPIR Process Acquisition Interface Requirements

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

2.3.1 Symbol Table Requirements

The MPIR Process Acquisition Interface requires the MPI 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 MPI starter process executable or a shared library used by the MPI 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 MPI starter process executable, or dynamically loaded at runtime by the MPIR starter process.

The symbol table requirement for the MPI starter process requires the following:

1. The MPI 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).
2. The MPI 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.
3. The packaging, distribution, and installation procedures for the MPI starter process implementation must ensure that the symbol table information is not stripped, separated or otherwise discarded.

2.3.2 Debugger-Level Process Control Requirements

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

2.4 MPIR Process Acquisition Interface Limitations

The MPIR Process Acquisition Interface has the following limitations:

1. The MPIR Process Acquisition Interface supports MPI 1 level interfaces only. In particular, it 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**.
2. The MPIR Process Acquisition Interface is vulnerable to being "optimized out" due to compiler or linker optimizations, or being "discarded" due to symbol table stripping or other build or distribution errors. MPI implementations must take great care to ensure that the symbol table information is preserved throughout the build, packaging and distribution process.
3. The MPIR Process Acquisition Interface requires the tool to exert debugger-level control over the MPI starter process, which may be an additional burden on the tool (if the tool is not already a debugger). A further complication for the tool may occur in heterogeneous environments, where the node architecture of the MPI starter process may be different than the node architecture of the MPI rank processes.

2.5 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 follows.

2.5.1 MPIR Job Launch and Attach Models

The MPIR Process Acquisition Interface supports 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 MPI starter process executable, which in turn starts the MPI application. Consider the following example command, where

tool.exe is the tool executable, **mpirun.exe** is the MPI starter process executable and **app.exe** is the MPI application executable:

```
% tool.exe tool-args mpirun.exe mpirun-args app.exe
%
```

Under the MPIR **job attach model**, the MPI starter process starts the MPI application. At a later point in time, after the MPI job has started running, the tool is attached to the MPI starter process, as shown in the following example commands:

```
% mpirun.exe mpirun-args app.exe &
[Shell prints the process ID "pid" of the background mpirun.exe process]
%
```

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

```
% tool.exe tool-args -pid pid mpirun.exe
%
```

2.5.2 MPI Rank Process Synchronization Models

Under the job launch model, regardless of synchronization technique, the MPI implementation must ensure that the rank processes run no further than their calls to **MPI_Init**. This requirement guarantees that the tool can acquire the MPI rank processes fairly early in its lifetime.

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

As stated above, 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 the **MPIR_debug_gate** in the rank process. Other synchronization techniques include the following:

- The MPI rank processes form a barrier with the MPI starter process. During startup, the MPI rank processes are created and allowed to run to barrier. The MPI starter process joins the barrier, but only *after* it has stopped at the **MPIR_Breakpoint** function and notified the tool of the **MPIR_DEBUG_SPAWNED** event.
- The MPI starter process arranges for the MPI rank processes to be created in a stopped state, before they have executed any user-mode instructions. For example, on Unix systems, the rank processes may be stopped on exit from execve. After the MPI starter process has stopped at the **MPIR_Breakpoint** function and notified the tool of the **MPIR_DEBUG_SPAWNED** event, the MPI rank processes are continued.

When an implementation uses a synchronization technique that: (a) does not use the MPIR debug gate, and (b) does not require the tool to attach to and continue the MPI rank process, it should define the symbol **MPIR_partial_attach_ok** in the MPI starter process, and make sure that **MPIR_debug_gate** is *not* defined in the MPI rank processes.

2.6 MPIR Process Acquisition Interface Extensions

MPIR has been extended by some vendors, as follows.

2.6.1 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** and **MPIR_server_arguments**.

They are used as follows:

1. Immediately after launching or attaching to the MPI 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 arguments to the **MPIR_server_arguments** variable.
2. The tool then sets **MPIR_being_debugged** to a non-zero value, and continues the MPI starter process.
3. The MPI 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.
4. It is then the responsibility of the tool and its daemon to establish communications.

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

2.6.2 Message Queue Display DLL Search Extension

Jeff?

2.6.3 Object Handle DLL Extension

Jeff?

2.7 MPIR Process Acquisition Interface Use Case

Table 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 "mpirun" to launch a set of MPI tasks running an executable named "a.out".

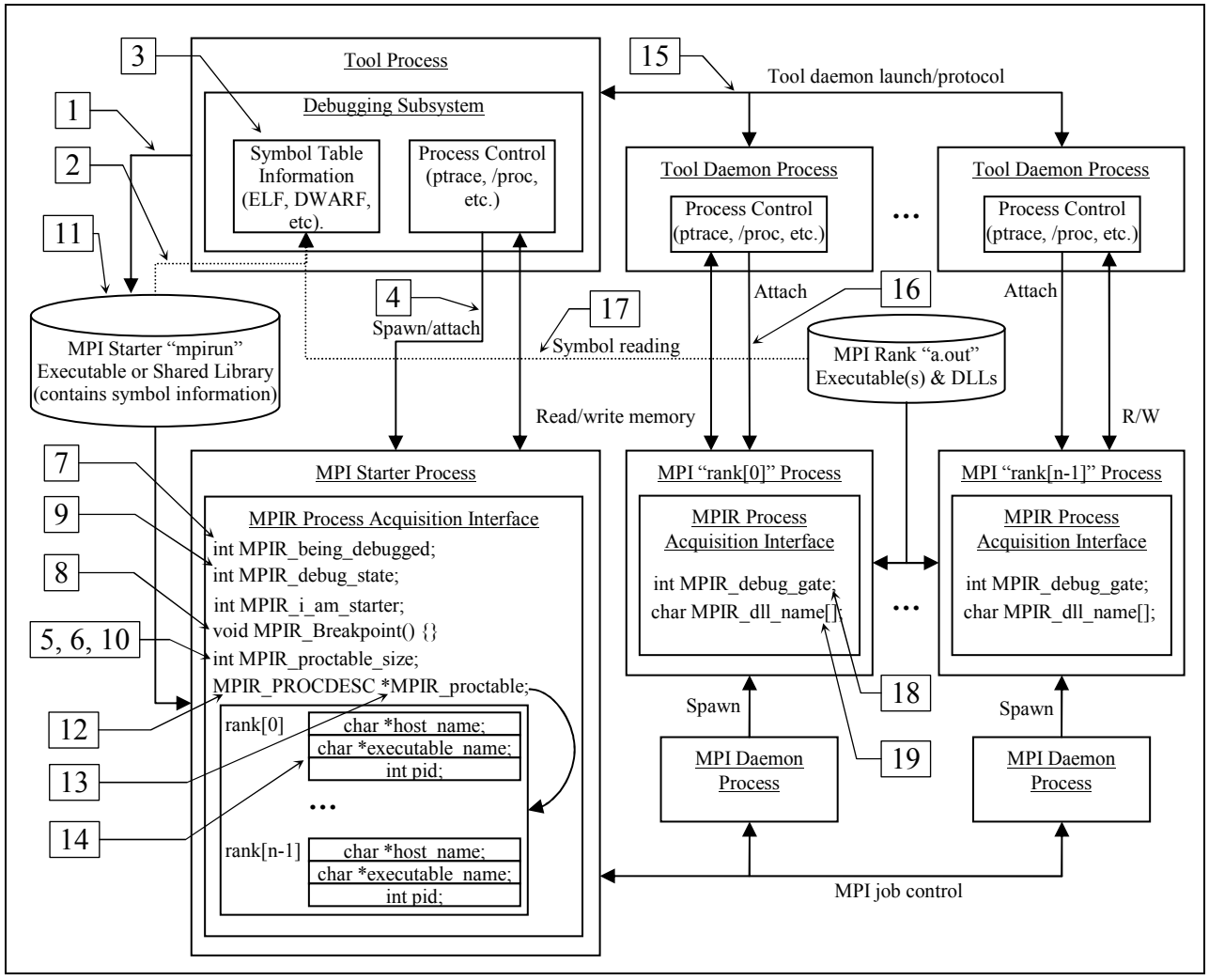

Table 1 Example collaboration diagram for MPIR Process Acquisition Interface

Startup processing:

1. The tool is started on the "mpirun" 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 an MPI starter program by querying the symbol table for the **MPIR_Breakpoint** symbol. If the symbol exists, then the tool should consider this an MPI starter program.
4. The tool spawns or attaches to the MPI 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** out of the MPI starter process to determine the number of rank processes already created by the MPI 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 rank processes that might exist. If the symbol **MPIR_i_am_starter** is *not* defined in the starter process, then the starter process *is* rank 0, thus the tool is already controlling rank 0.
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. The tool may then continue the MPI starter process.

Event processing:

9. If the starter process hits the breakpoint at **MPIR_Breakpoint**, the tool reads the value of the **MPIR_debug_state** variable out of the starter process, and performs one of the following actions:
   - 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 MPI starter process has spawned the MPI rank processes and filled in the process descriptor table. The tool can attach to any additional rank processes that have appeared in the process descriptor table.
   - 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.

Process descriptor table processing:

10. The tool reads the value of **MPIR_proctable_size** variable out of the MPI starter process to determine the number of rank processes already created by the MPI 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** 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** 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 rank processes.
15. The tool launches its tool daemons, if necessary, on the set of nodes being used by the rank processes.
16. The tool attaches to the rank processes.
17. The tool reads the symbol table information of the rank processes.

MPI rank process processing:

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 rank 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 rank processes to release them from the gate, even if the user has instructed the tool to *not* operate on all of the rank.
19. The tool reads the value of the **MPIR_dll_name** character string variable out of the rank 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 rank 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 rank processes that are running the same executable. For example, in a single program multiple data (SPMD) style program, all rank 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 rank processes executing “a.out” have the same **MPIR_dll_name** value, and all rank 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”.

2.8 MPIR Process Acquisition Interface Specification

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

2.8.1 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 MPI starter process and MPI rank 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 MPI starter and rank processes but are set by the tool.

2.8.2 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 MPI starter process.

MPIR_PROCDESC is a **typedef** name for an anonymous structure that holds process descriptor information for a single MPI rank 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 MPI starter process that specifies the target node location of the MPI rank process in the form of a host name or IP address. The host name or IP address string must be translatable to an IPv4 or IPv6 address.

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

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 rank 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 target process and avoid reading redundant character strings.

2.8.3 MPIR_being_debugged

Global variable definition:

```
VOLATILE int MPIR_being_debugged;
```

Definition is *not* required.
Definition is contained within the address space of the MPI starter process.

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

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

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

2.8.4 MPIR_proctable

Global variable definition:

```
MPIR_PROCDESC *MPIR_proctable;
```

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

MPIR_proctable is a pointer variable set by the MPI 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.

2.8.5 MPIR_proctable_size

Global variable definition:

```
int MPIR_proctable_size;
```

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

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

2.8.6 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 MPI starter process.

MPIR_debug_state is an integer variable set by the MPI starter process that specifies the state of the MPI job at the point where the MPI starter process calls the **MPIR_Breakpoint** function. The MPI 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 MPI starter process.
- If the value is **MPIR_DEBUG_SPAWNED** (1), then the MPI starter process has spawned the MPI rank processes and filled in the process descriptor table. The tool can attach to any additional rank processes that have appeared in the process descriptor table.
- 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.

2.8.7 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 MPI starter process.

MPIR_debug_abort_string is a pointer to a null-terminated character string set by the MPI 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 MPI starter process. The abort reason string can then be reported it to the user.

2.8.8 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 rank processes.

MPIR_debug_gate is an integer variable that is set to 1 by the tool to notify the MPI rank processes that the debugger has attached. An MPI rank 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 rank processes from running beyond the return from the application's call to **MPI_Init**.

2.8.9 MPIR_Breakpoint

Global subroutine definition:

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

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

MPIR_Breakpoint is the subroutine called by the MPI starter process to notify the tool that an MPIR event has occurred. The MPI 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 MPI starter process hits the breakpoint, the tool must read the value of the **MPIR_debug_state** variable to process an MPIR event.

2.8.10 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 rank process.
This symbol must *not* be defined if the process is a rank process.
Definition is contained within the address space of the MPI starter process.

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

If MPI rank process 0 is the starter process, as is the case with the MPICH 1 p4 channel implementation, this symbol must *not* be defined.

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.

2.8.11 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 MPI starter process.

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 rank 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 rank 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 rank 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.

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.

2.8.12 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 MPI starter process.

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 rank processes. The presence of the symbol **MPIR_force_to_main** does *not* imply that the MPI rank processes have been stopped before dynamic linking has occurred.

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.

2.8.13 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 MPI starter process.

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 rank 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 needn't attach nor continue MPI rank 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 rank 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 MPI starter process to release the whole MPI job, which can therefore be run *without* requiring the tool to acquire all of the MPI rank 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 rank 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 rank processes. The presence of this symbol does not prevent the tool from using the MPIR synchronization technique to acquire all of the processes, if it so chooses, because setting the **MPIR_debug_gate** variable (if present) is harmless.

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.

2.8.14 MPIR_dll_name

Global variable definition:

```
char MPIR_dll_name[];
```

Definition is *not* required.
Definition is contained within the address space of the MPI rank processes.
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.

2.8.15 MPIR_ignore_queues

Global variable definition:

```
int MPIR_ignore_queues;
```

Definition is *not* required.
Definition is contained within the address space of the MPI starter process.

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.

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.

2.8.16 MPIR_executable_path

Global variable definition:

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

Definition is *not* required.
Definition is contained within the address space of the MPI starter process.

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

When the tool then sets **MPIR_being_debugged** to a non-zero value and continues the MPI starter process, the MPI 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.

2.8.17 MPIR_server_arguments

Global variable definition:

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

Definition is *not* required.
Definition is contained within the address space of the MPI 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 MPI 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 MPI 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 MPI starter process, the MPI 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.

3 MPIR Message Queue Display

TBD.