

The MPIR Process Acquisition Interface

Version 1.1

MPI Forum Working Group on Tools
Accepted by the Message Passing Interface Forum
(March 1, 2018)

Acknowledgments

Author

John DelSignore

Contributing Authors

Dong Ahn, Ralph Castain, and Jeff Squyres

Editor

Martin Schulz

Reviewers

Bronis de Supinski, William Gropp, Marc-André Hermanns, David Lecomber,
Ashley Pittman, Alexander Supalov, and Greg Watson

Contents

1	Background	3
2	Overview	4
3	Definitions	6
3.1	MPI Process Definition	6
3.2	“Starter” Process Definition	6
	The MPI Rank 0 Process as the Starter Process	6
	A Separate “mpirexec” as the Starter Process	6
3.3	MPIR Node Definitions	7
4	Interface Requirements	8
4.1	Symbol Table Requirements	8
4.2	Debugger-Level Process Control Requirements	8
5	Limitations	9
6	Process Acquisition Modes	10
6.1	MPIR Job Launch and Attach Modes	10
6.2	MPI Process Synchronization	10
7	Optional Interface Extensions	12
7.1	Tool Daemon Launch Extension	12
7.2	Attach FIFO Extension	13
8	Use Case	14
9	Interface Specification	17
9.1	VOLATILE	17
9.2	MPIR_PROCDESC	17
9.3	MPIR_being_debugged	18
9.4	MPIR_proctable	19
9.5	MPIR_proctable_size	19
9.6	MPIR_debug_state	19
9.7	MPIR_debug_abort_string	20
9.8	MPIR_debug_gate	21
9.9	MPIR_Breakpoint	21
9.10	MPIR_i_am_starter	21

9.11	MPIR_acquired_pre_main	22
9.12	MPIR_force_to_main	22
9.13	MPIR_partial_attach_ok	22
9.14	MPIR_ignore_queues	23
9.15	MPIR_executable_path	23
9.16	MPIR_server_arguments	24
9.17	MPIR_attach_fifo	24
A	Changelog	25
A.1	Changes from Version 1.0 to Version 1.1	25
	Bibliography	26

1 Version 1.1: March 1, 2018 This document addressed clarifications and errata to the initial
2 document. Furthermore, it addressed several shortcomings in the layout and structure of
3 the document.

4 Version 1.0: October 11, 2010 This document codified the state of the practice for MPI
5 runtime systems to enable the debugging of MPI applications. This and future versions of
6 this side document to the MPI standard are ratified by the MPI Forum, but not an official
7 part of the standard itself.

Chapter 1

Background

Developing, debugging, and instrumenting parallel applications, by their distributed and multi-process nature, has always been difficult. By the mid-1990's, at least somewhat pushed by the completion of MPI-1.0 [4] in 1994, parallel application developers cited a real need for comprehensive debugging and instruction tools suitable use in parallel environments. In response, the TotalView debugger – an already mature parallel debugger for the BBN Uniform System and Oak Ridge 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/acquisition and one for message queue extraction. Coined the "MPIR" interfaces [1, 2], the MPI debugging interfaces eventually became *de facto* standards implemented by various MPI providers such as Compaq, HP, IBM, Intel, LAM/MPI, MPI Software Technologies, Open MPI, Quadrics, SCALI, SGI, Sun/Oracle 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 current 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.

Rationale. Note that this document does *not* introduce any improvements to the existing *de facto* use of the MPIR interface. For example, this document does not support MPI-2 [3] dynamic processes. Nor does it fix several of MPIR's well-known scalability issues. This document is solely intended to codify the current state of the art. (*End of rationale.*)

Chapter 2

Overview

The MPIR Process Acquisition Interface (MPAI), 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 MPAI interface supports both launching an MPI job under the control of a tool and attaching a tool to an already running job.

The MPAI 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 MPAI interface requires that a tool reads symbol table information and traces the starter process, including starting and stopping the process, reading the memory and registers of the starter process, planting breakpoints, and handling events. The MPAI 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 MPAI 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 the 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

¹ of the message queues in the MPI processes. Details of the MQD are not included in this
² document.

Chapter 3

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 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`.

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.

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`”.

1 3.3 MPIR Node Definitions

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

Chapter 4

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.

1 Chapter 5

2 Limitations

3 The MPIR Process Acquisition Interface has the following limitation:

- 4 • The MPIR Process Acquisition Interface does not support the dynamic process cre-
5 ation or communication facilities of MPI-2, such as MPI_COMM_SPAWN[
6 _MULTIPLE], MPI_COMM_ACCEPT, MPI_COMM_CONNECT, or MPI_COMM_JOIN.

Chapter 6

Process Acquisition Modes

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

6.1 MPIR Job Launch and Attach Modes

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 mode, 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 mode, 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 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 from above> mpiexec
$
```

6.2 MPI Process Synchronization

Under the job launch mode 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.

1 The MPIR Process Acquisition Interface provides an optional interface (see the de-
2 scription of MPIR debug gate variable named `MPIR_debug_gate` in Section 9.8) that allows
3 the tool to synchronize with the start up of the MPI processes. The goal of the MPIR debug
4 gate is to prevent the MPI processes from “running away” before the tool has a chance to
5 attach to them.

6 An MPI implementation is not required to use the `MPIR_debug_gate` variable for syn-
7 chronization. In fact, implementations are encouraged to use a synchronization technique
8 that does not involve the use of `MPIR_debug_gate` in the MPI processes. Other synchro-
9 nization techniques include the following:

- 10 • The MPI processes form a barrier with the starter process. During startup, the MPI
11 processes are created and allowed to run to barrier. The starter process joins the
12 barrier, but only after it has set `MPIR_debug_state` (Section 9.8) to `MPIR_DEBUG_-`
13 `SPAWNED` (Section 9.8) and called the `MPIR_Breakpoint` (Section 9.9) function to
14 notify the tool of the job spawn event.
- 15 • The starter process arranges for the MPI processes to be created in a stopped state,
16 before they have executed any user-mode instructions. For example, on Posix-like
17 systems, the MPI processes may be stopped on exit from `execve`. After the starter
18 process has set `MPIR_debug_state` to `MPIR_DEBUG_SPAWNED` and called the `MPIR_-`
19 `Breakpoint` function to notify the tool of the job spawn event, the MPI processes are
20 continued.

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

Chapter 7

Optional Interface Extensions

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

7.1 Tool Daemon Launch Extension

The original MPIR Process Acquisition Interface does not specify how the tool launches helper daemons (if required), nor does it support tool daemon launch. The original MPIR interface was extended on IBM Blue Gene systems to support tool daemon launch; that extension has also been implemented in Open MPI. The extension provides support only for tool daemon launch – it does not support 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` (Section 9.15) and `MPIR_server_arguments` (Section 9.16). They are used as follows:

1. Immediately after launching or attaching to the starter process, the tool writes the path name of its tool daemons executable file to the `MPIR_executable_path` variable, and writes the tool daemons command line arguments to the `MPIR_server_arguments` variable.
2. The tool then sets `MPIR_being_debugged` (Section 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 mode, 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 applications 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 (Section 9.6).
5. The starter process raises the job spawn event.

1 6. It is then the responsibility of the tool and its daemon to establish communications,
2 and attach to the MPI processes.

3 See the description of the `MPIR_executable_path` and `MPIR_server_arguments` vari-
4 ables for more information.

5 7.2 Attach FIFO Extension

6 To support the MPIR attach mode of the tool daemon launch extension (Section 7.1),
7 the starter process of MPI job control runtime systems periodically polls `MPIR_being_-`
8 `debugged` (Section 9.3). While this scheme works well for the systems that run their starter
9 on a dedicated node, it can significantly affect the MPI job's performance and scalability
10 for others that co-locate their starter with the MPI processes. To minimize such effects,
11 Open MPI added the `MPIR_attach_fifo` (Section 9.17) extension. This extension is used
12 as follows:

- 13 1. When the starter launches an MPI job, it locally opens a FIFO (named pipe) that is
14 owned and writable by its owner, writes the FIFO's path to the `MPIR_attach_fifo`
15 variable, and waits for a byte to arrive through the channel (e.g., one of the starter's
16 threads posts a blocking read on the channel).
- 17 2. Then, when the tool attaches to the starter, the tool sets `MPIR_being_debugged` to a
18 non-zero value, reads the path stored in `MPIR_attach_fifo`, opens and writes a byte
19 with a certain value into the FIFO.
- 20 3. In response, the starter wakes up, checks the value of the received byte, and performs
21 a predefined task associated with the value. If the value is 1, it reads the `MPIR_-`
22 `being_debugged` variable and reacts accordingly (see Section 7.1). The value of 0
23 is reserved for a no-op. 2-127 are reserved for MPIR future use and 128-255 are for
24 vendor-defined use.

25 The MPI job control runtime system is responsible for resetting the FIFO when the
26 tool finishes attaching from the MPI job and also for cleaning up the FIFO when the starter
27 terminates. The control system should also provide an option (e.g., a command line option
28 of the starter or a system configuration parameter) that allows the starter to fall back and
29 periodically poll the `MPIR_being_debugged` variable so as to ensure backward compatibility
30 with the tools that only support the original tool daemon launch interface (Section 7.1).
31 Given such an option, the starter should disable `MPIR_attach_fifo` support.

32 See the description of the `MPIR_attach_fifo` variable for more information (Sec-
33 tion 9.17).

Chapter 8

Use Case

Figure 8.1 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 `mpirexec` to launch a set of MPI tasks running an executable named “`a.out`”.

Startup processing:

1. The tool is started on the `mpirexec` 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_proactable_size` (Section 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_proactable_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.

- 1 reliable because the name `MPIR_PROCDDESC` is a typedef, which may not be available
2 for by-name lookup.
- 3 12. The tool reads the value of the `MPIR_proactable` pointer variable out of the starter
4 process. The pointer is the base address of the process descriptor table.
- 5 13. The tool reads the process descriptor table out of the starter process, starting at the
6 `MPIR_proactable` pointer variable value. The length in bytes of the process descriptor
7 table is the size of the `MPIR_PROCDDESC` process descriptor structure in bytes multiplied
8 by the value of `MPIR_proactable_size` variable.
- 9 14. The tool iterates over the process descriptor table information to identify the node
10 name or IP address, executable path name, and process ID of each of the MPI pro-
11 cesses.
- 12 15. The tool launches its tool daemons, if necessary, on the set of nodes being used by
13 the MPI processes.
- 14 16. The tool attaches to the MPI processes.
- 15 17. The tool reads the symbol table information of the MPI processes.

16 **MPI process attach:**

- 17 18. If the symbol `MPIR_partial_attach_ok` is defined in the starter process, then this
18 informs the tool that the initial startup barrier is implemented by the MPI system,
19 and it is not necessary to set the `MPIR_debug_gate` variable in any of MPI processes.
20 However, if the symbol `MPIR_partial_attach_ok` is not defined in the starter process,
21 the tool must attach and set the `MPIR_debug_gate` variable to 1 in each MPI processes
22 to release them from the gate, even if the tool user has instructed the tool to not attach
23 to all of the MPI processes.

1 Chapter 9

2 Interface Specification

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

7 9.1 VOLATILE

8 Macro definition:

```
9 #ifndef VOLATILE  
10 #if defined(__STDC__) || defined(__cplusplus)  
11 #define VOLATILE volatile  
12 #else  
13 #define VOLATILE  
14 #endif  
15 #endif
```

16 | Definition is required.

17 | Definition is used by the starter process and MPI processes.

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

22 9.2 MPIR_PROCDESC

23 Type definition:

```
24 typedef struct {  
25     char *host_name;  
26     char *executable_name;  
27     int pid;  
28 } MPIR_PROCDESC;
```

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

3 **MPIR_PROCDesc** is a typedef name for an anonymous structure that holds process de-
4 scriptor information for a single MPI process. The structure must contain three members
5 with the same names and types as specified above. The tool must use the symbol table
6 information to determine the overall size of the structure, and offset and size of each of the
7 structures members.

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

13 The **executable_name** member is a pointer to a null-terminated character string in
14 the address space of the starter process that specifies the name of the executable the MPI
15 process is running. The string should be the full path name to the executable, which may
16 be relative to the host node or target node.

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

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

29 9.3 MPIR_being_debugged

30 Global variable definition:

```
31 VOLATILE int MPIR_being_debugged;
```

32 | Definition is not required.
33 | Definition is contained within the address space of the starter
34 | process and/or any of the MPI processes which it started.
35 | Variable is written by the tool in any process where it exists.

36 **MPIR_being_debugged** is an integer variable that is set or cleared by the tool to notify
37 a process that a tool is present. The tool only changes the value of the symbol in any
38 process where it exists in two cases:

- 39 1. Immediately after spawning or attaching to a process, the tool sets the value to 1.
- 40 2. Immediately before detaching from a process, the tool sets the value to 0.

1 Any processes defining this variable may monitor its value and perform certain opera-
2 tions differently. For example, this variable might control whether or not the starter process
3 forces the MPI processes to wait for the `MPIR_debug_gate` to be set.

4 9.4 `MPIR_proctable`

5 Global variable definition:

```
6 MPIR_PROCDesc *MPIR_proctable;
```

7 | Definition is required.
8 | Definition is contained within the address space of the starter process.
9 | Variable is written by the starter process, and read by the tool.

10 `MPIR_proctable` is a pointer variable set by the starter process that points to an array
11 of `MPIR_PROCDesc` structures containing `MPIR_proctable_size` elements. This array of
12 structures is the process descriptor table.

13 The index position in the process descriptor table is the rank of the process in `MPI_-`
14 `COMM_WORLD`. For example, index 0 in the table specifies rank 0 in `MPI_COMM_WORLD`,
15 index 1 in the table specifies rank 1 in `MPI_COMM_WORLD`, and so forth.

16 9.5 `MPIR_proctable_size`

17 Global variable definition:

```
18 int MPIR_proctable_size;
```

19 | Definition is required.
20 | Definition is contained within the address space of the starter process.
21 | Variable is written by the starter process, and read by the tool.

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

25 9.6 `MPIR_debug_state`

26 Macro definitions:

```
27 #define MPIR_NULL          0  
28 #define MPIR_DEBUG_SPAWned 1  
29 #define MPIR_DEBUG_ABORTING 2
```

30 Global variable definition:

```
31 VOLATILE int MPIR_debug_state;
```

- 1 | Definition is required.
- 2 | Definition is contained within the address space of the starter process.
- 3 | Variable is written by the starter process, and read by the tool.
- 4 | Variables initial value should be 0 (MPIR_NULL).

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

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

- 12 • If the value is MPIR_NULL (0), then the tool should ignore the event and continue the
 13 starter process.
- 14 • If the value is MPIR_DEBUG_SPAWNED (1), then the starter process has spawned the
 15 MPI processes and filled in the process descriptor table. The tool can attach to any
 16 additional MPI processes that have appeared in the process descriptor table. This is
 17 known as a “job spawn event”.
- 18 • If the value is MPIR_DEBUG_ABORTING (2), then the MPI job has aborted and the
 19 tool can notify the user of the abort condition. The tool can read the reason for
 20 aborting the job by reading the character string out of the starter process, which is
 21 pointed to by the MPIR_debug_abort_string variable in the starter process.

22 The tool may continue or leave the starter process stopped after processing the event.
 23 The tool decides when to continue the starter process. For example, a debugger may al-
 24 low the user to control the execution of the starter process in conjunction with controlling
 25 the execution of the MPI processes. However, a performance analyzer might automatically
 26 continue the starter process after processing the MPIR event. Note that some MPI imple-
 27 mentations may require that the starter process be running while one or more of the MPI
 28 processes are running. Therefore, the tool may be required to implicitly continue the starter
 29 process when any of the MPI processes are continued.

30 9.7 MPIR_debug_abort_string

31 Global variable definition:

```
32 char *MPIR_debug_abort_string;
```

- 33 | Definition is not required.
- 34 | Definition is contained within the address space of the starter process.
- 35 | Variable is written by the starter process, and read by the tool.

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

1 9.8 MPIR_debug_gate

2 Global variable definition:

```
3 VOLATILE int MPIR_debug_gate;
```

4 | Definition is not required.

5 | Definition is contained within the address space of the MPI processes.

6 | Variable is written by the tool, and read by the MPI processes.

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

11 An MPI implementation is not required to use the MPIR_debug_gate variable for syn-
12 chronization. However, the MPI job control runtime system must prevent the created MPI
13 processes from running beyond the return from the applications call to MPI_INIT.

14 9.9 MPIR_Breakpoint

15 Global subroutine definition:

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

17 | Definition is required.

18 | Definition is contained within the address space of the starter process.

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

25 9.10 MPIR_i_am_starter

26 Global variable definition:

```
27 int MPIR_i_am_starter;
```

28 | Definition is required when the MPIR starter process is not also an MPI process.

29 | This symbol must not be defined if the process is an MPI process.

30 | Definition is contained within the address space of the starter process.

31 | Variable is neither read nor written.

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

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

1 9.11 MPIR_acquired_pre_main

2 Global variable definition:

```
3 int MPIR_acquired_pre_main;
```

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

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

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

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

18 9.12 MPIR_force_to_main

19 Global variable definition:

```
20 int MPIR_force_to_main;
```

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

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

28 9.13 MPIR_partial_attach_ok

29 Global variable definition:

```
30 int MPIR_partial_attach_ok;
```

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

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

36 If the symbol MPIR_partial_attach_ok is present, then this informs the tool that the
37 initial startup synchronization is implemented in such a way that the tool need not attach

1 nor continue MPI processes that the user is not interested in controlling. For example,
2 the MPI implementation synchronization startup may be implemented as a barrier, rather
3 than by having each of the MPI processes hang in a loop waiting for the `MPIR_debug_gate`
4 variable to be set by the tool.

5 Thus, the tool needs only to release the starter process to release the whole MPI job,
6 which can therefore be run without requiring the tool to acquire all of the MPI processes
7 included in the MPI job. This is useful in tools that include the possibility of attaching
8 to processes later in the tool session (for example, by selecting only processes in a specific
9 communicator, or a specific process in `MPI_COMM_WORLD`). This method of operation is
10 preferred because operating on a subset of processes is a valuable feature on high-scale
11 systems.

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

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

17 9.14 `MPIR_ignore_queues`

18 Global variable definition:

```
19 int MPIR_ignore_queues;
```

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

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

26 9.15 `MPIR_executable_path`

27 Global variable definition:

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

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

32 `MPIR_executable_path` is a null-terminated character string that is written by the
33 tool into the address space of the starter process. The string is the path name of the tool
34 daemons executable file on the target node.

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

1 9.16 MPIR_server_arguments

2 Global variable definition:

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

4 | Definition is not required.

5 | Definition is contained within the address space of the starter process.

6 | Variable is written by the tool, and read by the starter process.

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

12 For example, the following C string contains four arguments:

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

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

```
15 argv[0] = "-callback"
```

```
16 argv[1] = "10.0.0.10"
```

```
17 argv[2] = "-name"
```

```
18 argv[3] = "has spaces and \ttabs"
```

```
19 argv[4] = 0
```

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

24 9.17 MPIR_attach_fifo

```
25 char MPIR_attach_fifo[256];
```

26 | Definition is *not* required.

27 | Definition is contained within the address space of the starter process.

28 | Variable is written by the starter, and read by the tool.

29 MPIR_attach_fifo is a null-terminated character string that is written by the starter.
30 The tool reads it from the address space of the starter. The string is the path name of a
31 FIFO (named pipe) that is owned and writable by the owner of the starter process. The
32 FIFO resides on the node where the starter is running. The tool writes a byte with a value
33 into this FIFO, causing the starter to respond as follows:

```
34 0: NOP
```

```
35 1: Causing the starter to read the value of MPIR_being_debugged
```

```
36 2 – 127: Reserved by the MPIR interface for future use
```

```
37 128 – 255: Reserved for vendor-defined use
```

1 Annex A

2 Changelog

3 This annex summarizes changes from the previous versions of the *MPIR Process Acquisition*
4 *Interface* document to the version presented by this document. Only significant changes
5 that might either require implementation effort in MPI libraries or debuggers, or change
6 the understanding of specific mechanisms described by this document are presented.

7 A.1 Changes from Version 1.0 to Version 1.1

- 8 1. Clarifications to the availability of the variable `MPIR_being_debugged` in the starter
9 process.

1 Bibliography

- 2 [1] mpi-debug: Finding Processes. <http://www-unix.mcs.anl.gov/mpi/mpi-debug/>.
- 3 [2] James Cownie and William Gropp. A Standard Interface for Debugger Access to Message
4 Queue Information in MPI. In *Proceedings of the 6th European PVM/MPI Users' Group*
5 *Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*,
6 pages 51–58, Barcelona, Spain, September 1999.
- 7 [3] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk,
8 William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing
9 Interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors,
10 *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science,
11 pages 128–135. Springer Verlag, 1996.
- 12 [4] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of*
13 *Supercomputing*, pages 878–883. IEEE Computer Society Press, Nov 1993.