MPI Debugger Requirements for MPI Implementors

Chris January, Allinea Software Ltd.

February 2016

Contents

Chapter 1

Introduction

The MPIR Process Acquisition Interface [2] (hereafter called *MPIR* for short) documents the interface commonly used by MPI debuggers to take control of MPI jobs. The document is descriptive rather than normative.

This document outlines the high level requirements of the Allinea DDT MPI debugger, and shows how these requirements are met by *MPIR*. It also gives consideration to some issues not originally envisaged when *MPIR* was written.

We assume a model where MPI jobs are started by a separate `mpiexec` process which may variously be called `mpirun`, `poe`, etc., known as the *job starter*. This is the *Separate "`mpiexec`" as the Starter Process* model given in section 3.2 of *MPIR*. Where an alternative PMI-compliant job starter, such as SLURM's `srun`, is used in preference to the MPI implementation's own job starter the alternative PMI-compliant job starter must also implement the interfaces required by the MPI debugger.

Chapter 2

Taking Control of a Job

An MPI debugger may either be used to take control of a job at job launch time or may be used to attach to an already running job.

2.1 Job Launch Mode

When taking control of a job at launch time it is important the job does not make progress beyond MPI initialization (i.e. `MPI_Init`) before the debugger takes control. This is accompolished in *MPIR* by having the MPI processes form a barrier with the starter process (section 6.2). The debugger requires:

1. the job starter to signal the debugger when all the MPI processes are started and waiting at the barrier (`MPIR_Breakpoint` in sections 6.2 and 9.9 of *MPIR*).

2. the job starter to provide an interface for the debugger to retrieve a list of MPI processes (`MPIR_Proctable` in chapter 8 and section 9.4 of *MPIR*).

3. a mechanism for the debugger to launch its daemons on the compute nodes (discussed in chapter 3).

4. a mechanism for the debugger to signal the job starter to release the MPI processes from the barrier (in the *MPIR* model this occurs when the debugger resumes the starter process after it breaks at `MPIR_Breakpoint`).

2.2 Attach Mode

The debugger requires:

1. the job starter to provide an interface for the debugger to retrieve a list of MPI processes (`MPIR_Proctable` in chapter 8 and section 9.4 of *MPIR*).

2. a mechanism for the debugger to launch its daemons on the compute nodes (discussed in chapter 3).

2.3 Scalable Attaching

Attaching to a job requires `MPIR_Proctable` to be populated. Populating it at startup may be costly for large jobs, and the penalty will be incurred even if no debugger is ever attached to the job. An MPI implementation may, therefore, choose to delay initialization of `MPIR_Proctable` until `MPIR_being_debugged` is set to `1` in the job starter process. After setting `MPIR_being_debugger` the debugger will poll `MPIR_Proctable_size` until it has a non-zero value.

Allinea DDT does not implement the optional attach FIFO extension from section 7.2 of *MPIR*.

Chapter 3

Scalable Daemon Launch

After an MPI debugger obtains the list of MPI processes making up an MPI job (the *host*:*pid* pairs from `MPIR_Proctable`) it must launch its daemons on all the compute nodes taking part in the job in order to take control of the MPI processes.

When the number of compute nodes used by an MPI job is small it is feasible to programatically use SSH to login to each compute node to start the daemons. For larger jobs, however, this becomes infeasible.

Allinea DDT uses one of a number of different solutions to launch its daemons, depending on the MPI implementation. These are outlined below, in order of preference, so that our preferred mechanism comes first.

3.1 Environment

However the daemons are launched they should have the same general environment (environment variables, root filesystem, etc.) as the MPI processes. A minimal environment for tools (e.g. just a few environment variables, no `HOME` set, etc.) as seen on Blue Gene/Q is neither required nor desirable, neither is running the tools in a `chroot` environment.

3.2 start_tool

This mechanism follows the design of the Blue Gene/Q `start_tool` command[1, p. 9].

A separate command (or, alternatively, shared library function) is supplied to start the daemons on the compute nodes called, for example, `start_tool`. The `start_tool` command takes an argument to identify the MPI job in question. The job starter process must provide a mechanism to obtain this identifier. For example it may expose a `MPIR_job_id` global variable containing the MPI job identifier. Alternatively the process ID of the job starter may be used as the MPI job identifier, although this is less desirable as it prevents attaching to a job from a different login / MOM node to the one running the job starter process.

It is not necessary for the `start_tool` command to provide the MPIR interface (`MPIR_Proctable`, etc.) but doing so makes it possible attach to a job

from a different login / MOM node to the one running the job starter process.

3.3 Tool Daemon Launch Extension

MPIR defines an optional extension knows as the *Tool Daemon Launch Extension*. In this extension `MPIR_executable_path` and `MPIR_server_arguments` are defined as character array variables containing the executable path to the daemons, and arguments to pass to them, respectively.

In order to allow enough space for the arguments required by Allinea DDT's daemons, MPI implementations should ensure the `MPIR_server_arguments` array is at least 2048 characters long.

This mechanism has one main drawback: it is unclear how the tool arguments will be split and therefore how to quote them if they need quoting.

3.4 Job Starter Argument

With this mechanism an extra argument is injected into the job starter command line before it is launched by the debugger. For example:

```
mpirun -n 65536 myprogram arg1 arg2
```

becomes:

```
mpirun --tool /opt/allinea/forge/bin/ddt-debugger --tool-arguments
    "..." -n 65536 myprogram arg1 arg2
```

This mechanism has some drawbacks:

1. it is unclear how the tool arguments will be split and therefore how to quote them if they need quoting
2. it requires manipulating the job starter command line
3. it only works when the debugger is starting the MPI job and launching the job starter command itself
4. it does not support attaching

Chapter 4

Environment Variables

Allinea MAP requires the ability to set the `LD_PRELOAD` and `LD_LIBRARY_PATH` in the environment of the MPI processes, but not in the environment of any MPI daemons, the debugger daemons, etc.

Many MPI implementations provide control over the environment of the MPI processes through their job starter command. If this is not the case MAP requires some alternative facility to control the environment of the MPI processes.

It is preferable to be able to inject arguments at the start of the job starter command line to set environment variables in all MPI processes, even if multiple executables (MPMD) are used, rather than requiring the same environment variables to specified for each MPMD executable.

Chapter 5

Staging Of Debugger Daemon Files On The Compute Nodes

The total size of the files used by Allinea DDT's daemons may total several hundred megabytes (although the working set is not as large). Many network filesystems struggle to cope with thousands of compute nodes requesting the daemons' library and data files simultaneously. For this reason we optionally stage the files on the compute nodes. This requires the compute nodes to have a local well known temporary filesystem (such as `/tmp`) with sufficient free space ($\geq\approx 1\,\mathrm{GB}$). Staging the files on the local compute node filesystems in this way can significantly improve startup times, at the expense of the memory overhead on the compute nodes (assuming the filesystem is RAM-based, e.g. `tmpfs`).

Bibliography

[1] John Attinella, Sam Miller, and Gary Lakner. *IBM System Blue Gene Solution: Blue Gene/Q Code Development and Tools Interface*. IBM, may 2013.

[2] John DelSignore. The mpir process acquisition interface version 1.0. Technical report, MPI Forum Working Group on Tools, 2010.