Hewlett Packard Enterprise

HPE Performance Software — Message Passing Interface

User Guide

Abstract:
The HPE Performance Software — Message Passing Interface (HPE Performance MPI) software supports the MPI standard. HPE Performance MPI facilitates parallel programming on large systems and on computer system clusters.

Part Number: 007-3773-032

Published: August 2017

Edition: Revision 32

Record of Revision

Version	Description
001	March 2004 Original Printing. This manual documents the Message Passing Toolkit implementation of the Message Passing Interface (MPI).
002	November 2004 Supports the MPT 1.11 release.
003	June 2005 Supports the MPT 1.12 release.
004	June 2007 Supports the MPT 1.13 release.
005	October 2007 Supports the MPT 1.17 release.
006	January 2008 Supports the MPT 1.18 release.
007	May 2008 Supports the MPT 1.19 release.
008	July 2008 Supports the MPT 1.20 release.
009	October 2008 Supports the MPT 1.21 release.
010	January 2009 Supports the MPT 1.22 release.
011	April 2009 Supports the MPT 1.23 release.
012	October 2009 Supports the MPT 1.25 release.

013	April 2010 Supports the MPT 2.0 release.
014	July 2010 Supports the MPT 2.01 release.
015	October 2010 Supports the MPT 2.02 release.
016	February 2011 Supports the MPT 2.03 release.
017	March 2011 Supports additional changes for the MPT 2.03 release.
018	August 2011 Supports changes for the MPT 2.04 release.
019	November 2011 Supports changes for the MPT 2.05 release.
020	May 2012 Supports changes for the MPT 2.06 release.
021	November 2012 Supports changes for the MPT 2.07 release.
022	May 2013 Supports changes for the Performance Suite 1.6 release and the MPT 2.0.9 release.
023	November 2013 Supports the SGI Performance Suite 1.7 release, the MPT 2.09 release, and the MPI 1.7 release.
024	February 2014 Supports the SGI Performance Suite 1.7 release, the MPT 2.09 release, and the MPI 1.7 release. Clarifies SLURM support.

025	June 2014 Supports the SGI Performance Suite 1.8 release, the SGI MPT 2.10 release, and the SGI MPI 1.8 release. This is the last revision of this documentation with the title *Message Passing Toolkit (MPT) User Guide*.
026	May 2015 Supports the SGI Performance Suite 1.10 release, the SGI MPT 2.12 release, and the SGI MPI 1.10 release. This documentation is now called the *SGI MPI and SGI SHMEM User Guide*.
027	November 2015 Supports the SGI Performance Suite 1.11 release, the SGI MPT 2.13 release, and the SGI MPI 1.11 release.
028	May 2016 Supports the SGI Performance Suite 1.12 release, the SGI MPT 2.14 release, and the SGI MPI 1.12 release.
029	May 2016 Supports the SGI Performance Suite 1.12 release, the SGI MPT 2.14 release, and the SGI MPI 1.12 release and adds third-party compatibility information.
030	November 2016 Supports the SGI Performance Suite 1.13 release, the SGI MPT 2.15 release, and the SGI MPI 1.13 release and adds third-party compatibility information.
031	June 2017 Supports the HPE Performance Software – Message Passing Interface 1.0 release, the HPE MPT 2.16 release, and the SGI MPI 1.14 release.
032	August 2017 Supports the HPE Performance Software – Message Passing Interface 1.0 release and the MPT 2.16 release.

Contents

Tables

About This Guide

The Message Passing Interface (MPI) standard supports C and Fortran programs with a library and supporting commands. MPI operates through a technique known as *message passing*, which is the use of library calls to request data delivery from one process to another or between groups of processes. MPI also supports parallel file I/O and remote memory access (RMA).

The HPE Performance Software — Message Passing Interface (HPE Performance MPI) software supports the MPI standard. HPE Performance MPI facilitates parallel programming. This publication describes HPE Performance MPI 1.0, which supports the MPI 3.1 standard.

HPE Performance MPI supports the OpenSHMEM standard. The OpenSHMEM standard describes a low-latency library that supports RMA on symmetric memory in parallel environments. The OpenSHMEM programming model is a partitioned global address space (PGAS) programming model that presents distributed processes with symmetric arrays that are accessible via `PUT` and `GET` operations from other processes. This publication describes software that supports OpenSHMEM version 1.3. The HPE SHMEM programming model is the basis for the OpenSHMEM™ programming model specification that is being developed by the Open Source Software Solutions multivendor working group.

The following significant features make HPE Performance MPI the preferred implementation:

- Data transfer optimizations for NUMAlink, where available, including single-copy data transfer.
- Multirail InfiniBand support, which takes full advantage of the multiple InfiniBand fabrics available on HPE SGI 8600 and SGI ICE systems.
- Optimized MPI remote memory access (RMA) one-sided commands.
- Support for multiple application binary interfaces (ABIs), including MPICH and OpenMPI.

HPE's support for MPI and OpenSHMEM is built on top of the Message Passing Toolkit (MPT). MPT is a high-performance communications middleware software product. On some platforms, MPT uses Array Services to launch applications. MPT is optimized for large-scale, high-performance cluster computing.

Compatibility Information

The following table describes compatibility between HPE Performance MPI and other products.

Technology	Notes
Red Hat Enterprise Linux (RHEL) operating system	RHEL 6.X and RHEL 7.X
SLES operating system	SLES 12 SPX and SLES 11 SPX
CentOS operating system	CentOS 6.X and 7.X
Fortran 2008	Supports Fortran 2008.
Computing platforms	Platforms: • HPE SGI 8600 cluster systems • HPE Apollo 2000, 6000, and 6500 cluster systems • HPE Apollo 20 and 40 cluster systems • HPE Proliant systems • HPE Integrity MC 990 X systems • SGI ICE cluster systems • SGI Rackable cluster systems • SGI UV systems
Multi-rail InfiniBand (IB)	
Multi-rail Intel Omni-Path Architecture (OPA)	No support for MPI spawn
TCP/IP communication	
Mellanox Fabric Collective Accelerator (FCA) 3.x / HCOLL	
NVIDIA GPUDirect remote direct memory access (RDMA) over IB NVIDIA GPUDirect RDMA	Requires Mellanox Open Fabrics Enterprise Distribution (OFED). No support for MPI RMA passive windows.

Technology	Notes
Checkpoint-restart (CPR), supported through Berkeley Lab checkpoint restart (BLCR).	Supports jobs running over shared memory, InfiniBand, and TCP/IP. No support for CPR when using the following: • OpenSHMEM • MPI remote memory access (RMA) passive windows • MPI Spawn • Process managment interface (PMI), which is commonly used by the simple Linux utility for resource management (SLURM)
Third-party debugging and profiling tools: • Allinea DDT • RogueWave TotalView • Tuning and Analysis Utilities (TAU) • Vampir	Contact HPE for information about additional debugging and profiling tools.
Process management interfaces (PMIs), specifically PMIx and PMI2.	Supported when running under SLURM.
Third-party workload managers: • Altair PBS Professional • SLURM • UNIVA Grid Engine • IBM LSF • Moab / TORQUE	

Note: This documentation uses the term *cluster* to refer to cluster computers, cluster systems, or cluster nodes in HPE Apollo systems, HPE SGI 8600 systems, SGI ICE systems, and SGI Rackable systems. The term *cluster* does not pertain to HPE Integrity MC 990 X systems or to SGI UV computer systems because they are large-memory, single system image (SSI) systems.

If you use an HPE Integrity MC 990 X system, you can assume that all references to SGI UV systems also apply to HPE Integrity MC 990 X systems.

HPE Performance Software Suite Publications and General MPI Information

The HPE Performance Software Suite Publications are as follows:

- The *HPE Performance Software - Message Passing Interface User Guide* describes how to use the HPE Performance Software — Message Passing Interface software.
- The *HPE Performance Software - Message Passing Interface MPInside Reference Guide* describes the MPI profiling tool called MPInside.
- The *HPE Performance Software - Message Passing Interface Cpuset Software Guide* describes how to use cpusets.
- `MPInside`(3)

  This man page lists all the MPInside environment variables that HPE supports.
- `MPInside-exp`(3)

  This man page lists MPInside environment variables that HPE supports on an experimental basis. Use of these environment variables can generate unexpected results. The HPE Performance MPI documentation uses some of these experimental variables in examples and procedures.
- The HPE Performance Software — Message Passing Interface release notes contain information about the specific software packages included in each product. The release notes are available in the following locations:
  - The HPE Performance MPI release notes are posted to the following SGI website:

    https://support1-sgi.custhelp.com/app/answers/detail/a_id/6093

    Note: To access documentation through the SGI customer portal, first log in, and then navigate to the link. You can log in at the following website:

    https://support.sgi.com/login

  - On the product media. The release notes reside in a text file in the `/docs` directory on the product media.
  - On the system. After installation, the release notes and other product documentation reside in the `/opt/hpe/hpc/mpt/mpt-`*version_number*`/doc/README.relnotes` directory.

For example, you can use the following `rpm`(8) command to retrieve the location of the Array Services release notes:

```
# rpm -qi sgi-arraysvcs
/usr/share/doc/packages/sgi-arraysvcs/README.relnotes
```

Use a text editor or other command to display the file that the `rpm`(8) command returns.

For information about the MPI standard, see the following:

- The Message Passing Interface Forum's website, which is as follows:

  http://www.mpi-forum.org/

- *Using MPI — 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)*, by Gropp, Lusk, and Skjellum. ISBN-13: 978-0262571326.
- The University of Tennessee technical report. See reference [24] from *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum. ISBN-13: 978–0262571043.
- Journal articles in the following publications:
  - *International Journal of Supercomputer Applications*, volume 8, number 3/4, 1994
  - *International Journal of Supercomputer Applications*, volume 12, number 1/4, pages 1 to 299, 1998

HPE Websites

The following HPE websites might interest you:

- Hewlett Packard Enterprise Information Library:

  www.hpe.com/info/EIL

- Single Point of Connectivity Knowledge (SPOCK) Storage compatibility matrix:

  www.hpe.com/storage/spock

- Storage white papers and analyst reports:

  www.hpe.com/storage/whitepapers

Documentation Feedback

Hewlett Packard Enterprise is committed to providing documentation that meets your needs. To help us improve the documentation, send any errors, suggestions, or comments to Documentation Feedback (docsfeedback@hpe.com). When submitting your feedback, include the document title, part number, edition, and publication date located on the front cover of the document. For online help content, include the product name, product version, help edition, and publication date located on the legal notices page.

Configuring the Message Passing Toolkit (MPT)

This chapter includes the following topics:

- "About Configuring MPT" on page 1
- "Configuring MPT on an SGI UV Computer System" on page 5

About Configuring MPT

When you installed the HPE Performance Suite — Message Passing Interface (HPE Performance MPI) software, you also installed MPT. Before you can run any HPE Performance MPI programs, however, you need to configure the MPT software. The procedures in this chapter explain how to configure MPT.

High-performance computers often host several released versions of MPT. This environment provides users with the flexibility they need to develop and run HPE Performance MPI programs. The configuration instructions in this chapter explain how to accommodate these multiple versions if your site needs to have multiple versions installed.

The configuration procedure differs, depending on your platform, as follows:

- On a cluster computing system, the MPT installation and configuration procedure includes image-management steps.

  For information about how to configure MPT on an HPE SGI 8600 system, an SGI ICE system, an HPE Apollo 20 series system, an HPE Apollo 40 series system, or an SGI Rackable cluster system, see the following:

  HPE SGI Management Suite Installation and Configuration Guide

  For information about how to install and configure MPT and Array Services on an HPE Apollo 6500 series, an HPE Apollo 6000 series, an HPE Apollo 2000 series, see the following:

  "Installing the Array Services Software on HPE Apollo Cluster Systems" on page 2

- On an HPE Integrity MC990 X system or on a SGI UV system, see the following:

  "Configuring MPT on an SGI UV Computer System" on page 5

Installing the Array Services Software on HPE Apollo Cluster Systems

You need to install and configure Array Services software on the cluster before any HPE Performance MPI programs can run. The HPE factory installs Array Services on many HPE cluster systems. If you administer one of the following cluster systems, however, you need to install the Array Services software manually:

- HPE Apollo 6500
- HPE Apollo 6000
- HPE Apollo 2000

The `arrayconfig`(1M) command creates the following files on the compute service node to which you are logged in:

- `/etc/array/arrayd.conf`
- `/etc/array/arrayd.auth`

Note: If SLURM is installed on the cluster, you do not need to install Array Services. Clusters with SLURM do not use the Array Services software.

The following topics explain how to install and configure Array Services:

- "Installing and Configuring Array Services on HPE Apollo Cluster Systems" on page 2
- "Array Services Security Options" on page 4
- "Array Services Configuration Examples" on page 5

Installing and Configuring Array Services on HPE Apollo Cluster Systems

The following procedure explains how to install Array Services on HPE Apollo systems.

Procedure 1-1 To install Array Services

1. Log into the head node.
2. Write the list of cluster nodes to a temporary file:

```
$ cmu_show_nodes > nodes.txt
```

3. (Conditional) Edit the node list to include only the compute nodes that you want to include in the array.

   Complete this step if you do not want to include all the cluster nodes in the array.

4. Copy the list of cluster nodes to a directory of your choosing on one of the compute nodes.

   For example:

   ```
   $ scp nodes.txt first_compute_node:/somewhere
   ```

   The preceding command copies the `nodes.txt` file to the `somewhere` directory on `first_compute_node`.

5. Use the `ssh`(1) command to log into the compute node to which you wrote the node list.

   For example:

   ```
   $ ssh first_compute_node
   ```

6. Change to the directory in which the node list resides.

   For example:

   ```
   $ cd somewhere
   ```

7. Use the `arrayconfig`(8) command to configure the nodes into an array.

   The command format is as follows:

   `/usr/sbin/arrayconfig -a` *arrayname* `-f -m -A` *method node node* `...`

   The command parameters are as follows:

   - For *arrayname*, specify a name for the array. The default is `default`.

   - For *method*, specify `munge`, `none`, or `simple`. For information about the security levels, see the following:

     "Array Services Security Options" on page 4

   - For *node*, specify the nodes to include. For example `/tmp/nodelist` or a list of individual node names.

For example, the following command configures the nodes into an array and uses the `-D` parameter to distribute the `arrayd.conf` file:

```
$ arrayconfig -fmD -a default -A simple 'cat nodes.txt'
```

Array Services Security Options

The following list explains the Array Services security options.

`simple` (default)	Generates hostname/key pairs by using either the OpenSSL, `rand` command, 64–bit values (if available) or by using `$RANDOM` Bash facilities.
`munge`	Configures additional security provided by MUNGE. The installation process installs MUNGE by default.
`none`	You can configure `none` in one of the following ways:

- `none` on all nodes, including the login node. The login node can be a compute node that is designated for user logins and/or other services.

  When you specify `none`, Array Services disables all authentication.

  OR

- `none` on the the compute nodes and `noremote` on the login node. A security setting of `noremote` prevents remote logins to the array.

  When you specify `noremote` on the compute services nodes and specify `none` on the compute nodes, users must run their jobs directly from the compute services nodes. In this case, users cannot submit HPE Performance MPI jobs remotely.

"Manually Configuring Array Services on Multiple Hosts" on page 128 explains how to configure `noremote` on the login node.

Array Services Configuration Examples

Example 1. To specify that array `myarray` use MUNGE security and include all HPE Apollo compute nodes, type the following command:

```
# /usr/sbin/arrayconfig -a myarray -f -m -A munge /tmp/nodelist
```

Example 2. To specify that array `yourarray` use no security, include one compute service node, and include all HPE Apollo compute nodes, type the following command:

```
# /usr/sbin/arrayconfig -a yourarray -f -m -A none n1 /tmp/nodelist
```

Configuring MPT on an SGI UV Computer System

The information in the following procedures explains how to configure MPT on an SGI UV system:

- "Verifying Prerequisites" on page 5
- "(Optional) Installing the MPT Software Into a Nondefault Working Directory" on page 6
- "Adjusting File Resource Limits" on page 9
- "Completing the Configuration" on page 10

Verifying Prerequisites

The following procedure explains how to verify the MPT software's installation prerequisites.

Procedure 1-2 To verify prerequisites

1. As the root user, log into the SGI UV computer.
2. Verify that you have one of the following operating system software packages installed and configured:

- Red Hat Enterprise Linux (RHEL) 7 or 6
- SLES 12 or 11

You can type the following command to verify your operating system version:

```
# cat /etc/*release
```

3. Type the following command to verify that the HPE Performance MPI 1.0 release is installed:

```
# cat /etc/sgi-mpi-release
HPE Performance Software - Message Passing Interface 1.0
SGI MPI 1.14, Build xxxxxx.sles12sp2-xxxxxxxxxx
```

4. Proceed to one of the following:

   - "(Optional) Installing the MPT Software Into a Nondefault Working Directory" on page 6, which explains how to configure MPT in a way that lets you maintain more than one released version of the software on your SGI UV computer system.
   - "Adjusting File Resource Limits" on page 9, which assumes you want the MPT software to remain in the default installation directory.

(Optional) Installing the MPT Software Into a Nondefault Working Directory

Perform the procedure in this topic if you want to install MPT into a custom, nondefault working directory. You might want to perform the procedure in this topic if, for example, you have a nondefault filesystem or if you want to use HPE Performance MPI on one of the following platforms:

- HPE Apollo 6500 system
- HPE Apollo 6000 system
- HPE Apollo 2000 system

The RPM utility enables you to create, install, and manage relocatable packages. You can install a matched set of MPT RPMs in either the default location or an alternate location. The default location for installing the MPT RPM is `/opt/hpe/hpc/mpt/mpt-2.`*rel_level*. To install the MPT RPM in an alternate location, use the `--relocate` parameter to the `rpm` command. The `--relocate` parameter specifies an alternate base directory for the MPT software installation.

Either `/opt/hpe/hpc/mpt/mpt-2.`*rel_level* or both `/opt/hpe/hpc/mpt/mpt-2.`*rel_level* and `/usr/share/modules/modulefiles/mpt` can be relocated. The post installation script reconfigures the module file for the new location as long as the *oldpath* argument in the `rpm`(8) command precisely matches the description in the RPM info. The general format for the `rpm` command is as follows:

`rpm --relocate` *oldpath=newpath*

- For *oldpath*, specify the MPT software's current location.

  If you install the MPT software in an alternate location, the `rpm` command's *oldpath* argument must precisely match the relocation listed in the RPM for the environment module automatic modification feature to be correct.

- For *newpath*, specify the location to which you want to install the MPT software.

Procedure 1-3 To install the MPT software in an alternate location

1. Plan how to avoid problems related to uninstalled RPM dependencies.

   The following are two approaches:

   - Option 1: If you install from a system that does not run MPT jobs, it might be appropriate to use the `--nodeps` parameter on the `rpm`(8) command line. This parameter directs the `rpm`(8) command to ignore dependencies.

   - Option 2: If you install from a system or cluster nodes upon which MPT jobs need to run, type the following package manager commands on each cluster node or cluster node image to locally install the needed prerequisites on all the cluster nodes:

     – On SLES platforms, type the following command:

     ```
     # zypper install cpuset-utils arraysvcs xpmem libbitmask
     ```

     – On RHEL platforms, type the following command:

     ```
     # yum install cpuset-utils arraysvcs xpmem  libbitmask
     ```

2. Use the `rpm` command to specify an alternate location for the MPT software bundle.

Example 1. The following example shows how to install MPT in /usr/local/hpe/mpt/mpt-2.16 rather than in /opt, which is the default:

```
# rpm -i --relocate /opt/hpe/hpc/mpt/mpt-2.16=/usr/local/hpe/mpt/mpt-2.16 \
sgi-mpt-*.x86.rpm
```

Example 2: The following RHEL example shows how to install the modules, in addition to the total MPT software bundle, to /usr/local/hpe/mpt/mpt-2.16 and /usr/local/mod/mpt-2.16:

```
# rpm -i --relocate /opt/hpe/hpc/mpt/mpt-2.16=/usr/local/hpe/mpt/mpt-2.16 \
--relocate /usr/share/Modules/modulefiles/mpt=/usr/local/mod/mpt-2.16 \
sgi-mpt-*.x86_64.rpm
```

In the preceding RHEL example, note that the Modules directory in the argument to the second --relocate parameter begins with an uppercase letter.

Example 3. The following SLES example shows how to install the modules, in addition to the total MPT software bundle, to /usr/local/hpe/mpt/mpt-2.16 and /usr/local/mod/mpt-2.16:

```
# rpm -i --relocate /opt/hpe/hpc/mpt/mpt-2.16=/usr/local/hpe/mpt/mpt-2.16 \
--relocate /usr/share/modules/modulefiles/mpt=/usr/local/mod/mpt-2.16 \
sgi-mpt-*.x86_64.rpm
```

In the preceding SLES example, note that the modules directory in the argument to the second --relocate parameter begins with a lowercase letter.

Example 4:

The following example rpm command output shows the available relocations:

```
# rpm -qpi sgi-mpt-2.16-sgi*.x86_64.rpm
... Relocations: /opt/hpe/hpc/mpt/mpt-2.16 /usr/share/modules/modulefiles/mpt
```

Note: In the preceding output, the example shows only the significant message at the end of the output string.

---

3. Proceed to the following:

   "Adjusting File Resource Limits" on page 9

For more information about using the rpm command, see the rpm man page.

Adjusting File Resource Limits

The following procedure explains how to increase resource limits on the number of open files and enforce new security policies.

Procedure 1-4 To adjust file resource limits

1. Type the following command to retrieve the number of cores on this computer:

   ```
   # cat /proc/cpuinfo | grep processor | wc -l
   ```

   In the preceding line, the last character is a lowercase `L`, not the number `1`.

   This `cat`(1) command returns the number of cores on the SGI UV computer system.

2. Use a text editor to open file `/etc/security/limits.conf`.

3. Add the following line to file `/etc/security/limits.conf`:

   ```
   *      hard    nofile       limit
   ```

   For *limit*, specify an open file limit, for the number of MPI processes per host, based on the following guidelines:

Processes/host	***limit***
Fewer than 512	3000
Up to 1024	6000
Up to 2048	8192 (default)
4096 or more	21000

   MPI jobs require a large number of file descriptors, and on larger systems, you might need to increase the system-wide limit on the number of open files. The default value for the file-limit resource is 8192. For example, the following line is suitable for 512 MPI processes per host:

   ```
   *      hard    nofile  3000
   ```

4. Add the following line to file `/etc/security/limits.conf`:

   ```
   *      hard   memlock  unlimited
   ```

   The preceding line increases the resource limit for locked memory.

5. Save and close file `/etc/security/limits.conf`.
6. Use a text editor to open file `/etc/pam.d/login`, which is the Linux pluggable authentication module (PAM) configuration file.
7. Add the following line to file `/etc/pam.d/login`:

   ```
   session    required     /lib/security/pam_limits.so
   ```

8. Save and close the file.
9. (Conditional) Update other authentication configuration files as needed.

   Perform this step if your site allows other login methods, such as `ssh`, `rlogin`, and so on.

10. Proceed to the following:

    "Completing the Configuration" on page 10

Completing the Configuration

The following procedure explains how to complete the MPT configuration.

Procedure 1-5 To complete the MPT configuration

1. Run a test MPI program to make sure that the new software is working as expected.
2. (Conditional) Inform your user community of the location of the new MPT release on this computer.

   Perform this step if you moved the MPT software to a nondefault location.

   In this procedure's examples, the module files are located in the following directories:

   - On RHEL platforms:

     ```
     /opt/mpt/mpt-2.16/usr/share/Modules/modulefiles/mpt/mpt-2.16
     ```

   - On SLES platforms:

     ```
     /opt/mpt/mpt-2.16/usr/share/modules/modulefiles/mpt/mpt-2.16
     ```

Getting Started

This chapter includes the following topics:

- "About Running MPI Applications" on page 11
- "Loading the MPI Software Module and Specifying the Library Path" on page 11
- "Compiling and Linking the MPI Program" on page 13
- "Launching the MPI Application" on page 15
- "Compiling and Running OpenSHMEM Applications" on page 21
- "Building MPI Fortran Modules" on page 23
- "Using Huge Pages" on page 25
- "Using HPE Performance MPI in an SELinux Environment (RHEL Platforms Only)" on page 27

About Running MPI Applications

This chapter provides procedures for building MPI applications. It provides examples of the use of the `mpirun`(1) command to launch MPI jobs. It also provides procedures for building and running SHMEM applications.

The process of running MPI applications consists of the following procedures:

- "Loading the MPI Software Module and Specifying the Library Path" on page 11
- "Compiling and Linking the MPI Program" on page 13
- "Launching the MPI Application" on page 15

Loading the MPI Software Module and Specifying the Library Path

You need to ensure that programs can find the MPT library routines when the programs run.

The default locations for the include files, the `.so` files, the `.a` files, and the `mpirun` command are pulled in automatically. To ensure that the `mpt` software module is loaded, you can load site-specific library modules, or you can specify the library path on the command line before you run the program.

The following procedure explains how to specify the path to the MPI `libmpi.so` library.

Procedure 2-1 To determine the library path

1. (Optional) Set the library path in the `mpt` module file.

   Complete this step if your site uses module files.

   Sample module files reside in the following locations:

   - `/opt/hpe/hpc/mpt/mpt-`*mpt_rel*`/doc`
   - `/usr/share/modules/modulefiles/mpt/`*mpt_rel*

   To load the MPT module, type the following command:

   ```
   % module load mpt
   ```

2. Determine the directory into which the MPT software is installed.

   ```
   % ldd a.out
   libmpi.so => /tmp/usr/lib/libmpi.so (0x40014000)
   libc.so.6 => /lib/libc.so.6 (0x402ac000)
   libdl.so.2 => /lib/libdl.so.2 (0x4039a000)
   /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
   ```

   Line 1 in the preceding output shows the library path correctly as `/tmp/usr/lib/libmpi.so`. If you do not specify the correct library path, the MPT software searches incorrectly for the libraries in the default location of `/usr/lib/libmpi.so`.

3. Type the following command to set the library path:

   ```
   % setenv LD_LIBRARY_PATH /library_path/usr/lib
   ```

   For *library_path*, type the path to the directory in which the MPT software is installed.

Example 1. The following command uses information from the previous step to set the library path to `/tmp/usr/lib`:

```
% setenv LD_LIBRARY_PATH /tmp/usr/lib
```

Example 2. The following command assumes that the libraries reside in `/data/nfs/lib`, which might be the case if you installed MPT in an NFS-mounted file system:

```
% setenv LD_LIBRARY_PATH /data/nfs/lib
```

Compiling and Linking the MPI Program

You can use one of the MPI wrapper compiler commands to run your program, or you can call the compiler directly. The following topics explain these two alternatives:

- "Compiling With the Wrapper Compilers" on page 13
- "Compiling With the GNU or Intel Compilers" on page 14

Compiling With the Wrapper Compilers

The MPI wrapper compilers automatically incorporate the compiling and linking functions into the compiler command. If possible, use one of the following wrapper compiler commands to run your program:

- `mpif08 -I /`*install_path*`/usr/include `*file*`.f -L `*lib_path*`/usr/lib`
- `mpif90 -I /`*install_path*`/usr/include `*file*`.f -L `*lib_path*`/usr/lib`
- `mpicxx -I /`*install_path*`/usr/include `*file*`.c -L `*lib_path*`/usr/lib`
- `mpicc -I /`*install_path*`/usr/include `*file*`.c -L `*lib_path*`/usr/lib`

The variables in the preceding commands are as follows:

- For *install_path*, type the path to the directory in which the MPT software is installed.
- For *file*, type the name of your C or Fortran program file name.
- For *lib_path*, type the path to the library files.

For example:

```
% mpicc -I /tmp/usr/include simple1_mpi.c -L /tmp/usr/lib
```

Compiling With the GNU or Intel Compilers

This topic explains how to run an MPI program if you need to call the GNU or Intel compilers directly. When the MPT RPM is installed as default, the commands to build an MPI-based application using the `.so` files are as follows:

- To compile using GNU compilers, choose one of the following commands:

  ```
  % g++ -o myprog myprog.C -lmpi++ -lmpi
  % gcc -o myprog myprog.c -lmpi
  ```

- To compile programs with the Intel compilers, choose one of the following commands:

  ```
  % icc -o myprog myprog.c -lmpi       # C - version 8
  % mpif08 simple1_mpi.f               # Fortran 2008 wrapper compiler
  % mpif90 simple1_mpi.f               # Fortran 90 wrapper compiler
  % ifort -o myprog myprog.f -lmpi     # Fortran - version 8
  % mpicc -o myprog myprog.c           # MPI C wrapper compiler
  % mpicxx -o myprog myprog.C          # MPI C++ wrapper compiler
  ```

  Note: Use the Intel compiler to compile Fortran 90 programs.

- To compile Fortran programs with the Intel compiler and enable compile-time checking of MPI subroutine calls, insert a `USE MPI` statement near the beginning of each subprogram to be checked. Also, use the following command:

  ```
  % ifort -I/usr/include -o myprog myprog.f -lmpi   # version 8
  ```

  Note: The preceding command assumes a default installation. If your site has more than one version of MPT installed, or if your site installed MPT into a nondefault location, contact your system administrator to verify the location of the module files. For a nondefault installation location, replace `/usr/include` with the name of the relocated directory.

- The special case of using the Open64 compiler in combination with hybrid MPI/OpenMP applications requires separate compilation and link command lines. The Open64 version of the OpenMP library requires the use of the `-openmp` option on the command line for compiling, but it interferes with proper linking of MPI libraries. Use the following sequence:

```
% opencc -o myprog.o -openmp -c myprog.c
% opencc -o myprog myprog.o -lopenmp -lmpi
```

Launching the MPI Application

You can use either a workload manager or the `mpirun` command to launch an MPI application.

The following topics explain these alternatives:

- "Using a Workload Manager to Launch an MPI Application" on page 15
- "Using the `mpirun` Command to Launch an MPI Application" on page 17
- "Using MPI Spawn Functions to Launch an MPI Application" on page 19

Using a Workload Manager to Launch an MPI Application

When an MPI job is run from a workload manager like PBS Professional, Torque, or Load Sharing Facility (LSF), it needs to start on the cluster nodes and CPUs that have been allocated to the job. For multi-node MPI jobs, the command that you use to start this type of job requires you to communicate the node and CPU selection information to the workload manager. MPT includes one of these commands, `mpiexec_mpt`(1), and the PBS Professional workload manager includes another such command, `mpiexec`(1). The following topics describe how to start MPI jobs with specific workload managers:

- "PBS Professional" on page 15
- "Torque" on page 16
- "Simple Linux Utility for Resource Management (SLURM)" on page 17

PBS Professional

You can run MPI applications from job scripts that you submit through workload managers such as the PBS Professional workload manager.

Process and thread pinning onto CPUs is especially important on cache-coherent non-uniform memory access (ccNUMA) systems such as the SGI UV system series. Process pinning is performed automatically if PBS Professional is set up to run each application in a set of dedicated cpusets. In these cases, PBS Professional sets the PBS_CPUSET_DEDICATED environment variable to the value YES. This has the same effect as setting MPI_DSM_DISTRIBUTE=ON. Process and thread pinning are also performed in all cases if omplace(1) is used.

Example 1. To run an MPI application with 512 processes, include the following in the directive file:

```
#PBS -l select=512:ncpus=1
 mpiexec_mpt ./a.out
```

Example 2. To run an MPI application with 512 Processes and four OpenMP threads per process, include the following in the directive file:

```
#PBS -l select=512:ncpus=4
 mpiexec_mpt omplace -nt 4 ./a.out
```

Some third-party debuggers support the mpiexec_mpt(1) command. The mpiexec_mpt(1) command includes a -tv option for use with TotalView and includes a -ddt option for use with DDT. For more information, see Chapter 4, "Debugging MPI Applications" on page 39.

PBS Professional includes an mpiexec(1) command that enables you to run HPE Performance Software — Message Passing Interface (HPE Performance MPI) applications. PBS Professional's command does not support the same set of extended options that the mpiexec_mpt(1) supports.

For more information about the PBS Professional workload manager, see the following website:

http://www.pbsworks.com/SupportGT.aspx?d=PBS-Professional,-Documentation

Torque

When running Torque, HPE recommends that you use the following mpiexec_mpt(1) command to launch MPT jobs:

mpiexec_mpt [-n *P*] ./a.out

The *P* argument is optional. By default, the program runs with the original number of processes specified on the job initialization in Torque. To use *P*, specify is the total number of MPI processes in the application. This syntax applies whether running on a single host or a clustered system.

For more information, see the `mpiexec_mpt`(1) man page. The `mpiexec_mpt` command has a `-tv` option for use by MPT when running the TotalView Debugger with a workoad manager like Torque. For more information about using the `mpiexec_mpt` command `-tv` option, see "Using the TotalView Debugger with MPI Programs" on page 39.

Simple Linux Utility for Resource Management (SLURM)

HPE Performance MPI is adapted for use with the SLURM workload manager. If you want to use HPE Performance MPI with SLURM, use the SLURM `pmi2` MPI plug-in or the SLURM `pmix` MPI plug-in. HPE Performance MPI 1.8 or later requires SLURM 2.6 or later.

For general information about SLURM, see the following website:

http://slurm.schedmd.com

For more information about how to use MPI with SLURM, see the following website:

http://slurm.schedmd.com/mpi_guide.html

Using the `mpirun` Command to Launch an MPI Application

The `mpirun`(1) command starts an MPI application. Use the `mpirun`(1) command when you are not using a resource manager, such as PBS Professional.

For a complete specification of the command line syntax, see the `mpirun`(1) man page.

The following topics explain how to use the `mpirun` command to launch a variety of applications:

- "Launching a Single Program on the Local Host" on page 17
- "Launching a Multiple Program, Multiple Data (MPMD) Application on the Local Host" on page 18
- "Launching a Distributed Application" on page 18

Launching a Single Program on the Local Host

To run an application on the local host, enter the `mpirun` command with the `-np` argument. Your entry must include the number of processes to run and the name of the MPI executable file.

Example 1. The following command starts three instances of the `mtest` application, which is passed an argument list (arguments are optional):

```
% mpirun -np 3 mtest 1000 "arg2"
```

Launching a Multiple Program, Multiple Data (MPMD) Application on the Local Host

You are not required to use a different host in each entry that you specify on the `mpirun` command. You can start a job that has multiple executable files on the same host.

Example 1. The following command runs one copy of `prog1` and five copies of `prog2` on the local host, and both executable files use shared memory:

```
% mpirun -np 1 prog1 : -np 5 prog2
```

Launching a Distributed Application

You can use the `mpirun` command to start a program that consists of any number of executable files and processes, and you can distribute the program to any number of hosts. A host is usually a single machine, but it can be any accessible computer running the Array Services software. For a list of the available nodes on systems running Array Services software, type the following command:

```
% ainfo machines
```

You can list multiple entries on the `mpirun` command line. Each entry contains an MPI executable file and a combination of hosts and process counts for running it. This gives you the ability to start different executable files on the same or different hosts as part of the same MPI application.

The examples show various ways to start an application that consists of multiple MPI executable files on multiple hosts.

Example 1. The following command runs ten instances of the `a.out` file on `host_a`:

```
% mpirun host_a -np 10 a.out
```

Example 2. The following command launches ten instances of `fred` on each of three hosts. `fred` has two input arguments.

```
% mpirun host_a, host_b, host_c -np 10 fred arg1 arg2
```

Example 3. The following command launches ten instances of `fred`, with different numbers of instances on each processor:

```
% mpirun host_a -np 2, host_b -np 3, host_c -np 5 fred arg1 arg2
```

Example 4. The following command launches an MPI application on different hosts with different numbers of processes and executable files:

```
% mpirun host_a 6 a.out : host_b -np 26 b.out
```

Using MPI Spawn Functions to Launch an MPI Application

The following two functions enable the MPI spawn feature:

- `MPI_Comm_spawn`
- `MPI_Comm_spawn_multiple`

For information about how to use the spawn functions, see the following:

- "Specifying the Universe Size Automatically" on page 19
- "Specifying the Universe Size Directly" on page 20
- "Specifying the Universe Size on the mpirun Command" on page 20
- "Specifying Host Information" on page 20

For more information, see the `mpiexec_mpt`(1) man page.

Specifying the Universe Size Automatically

You can specify the universe size automatically when you specify the `-spawn` parameter on the `mpiexec_mpt` command and use the following MPI process creation functions:

- `MPI_Comm_spawn`
- `MPI_Comm_spawn_multiple`

For example, assume that when you submitted a job, you specified 10 processes. The following command starts three instances of the `mtest` MPI application, so the application can spawn seven more:

```
% mpiexec_mpt -spawn -np 3 mtest
```

Specifying the Universe Size Directly

You can specify the universe size directly by setting one of the following shell variables:

- `MPI_UNIVERSE_SIZE`

  Set `MPI_UNIVERSE_SIZE` to the maximum number of processes possible in the universe.

- `MPI_UNIVERSE`

  Set `MPI_UNIVERSE` to control both the universe size and the possible set of hosts that processes can run on.

  This variable specifies hosts upon which processes can be launched. The syntax for this variable is a list of hp_specs without a specified application or argument list. For example:

  – `"host_a, host_b"`

  – `"host_a 8, host_b 16"`

  – `"host_a, host_b 12"`

  – `"host_a, host_b -np 16"`

  If `MPI_UNIVERSE` is not specified, MPI spawn requests place new processes on the local host.

Specifying the Universe Size on the mpirun Command

If you use the `mpirun` command, you can use the `-up` parameter to specify the universe size.

Specifying Host Information

You can pass host information to `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` in `MPI_info` objects. When you use this method, you can specify hosts that are outside of the list of hosts specified to `MPI_UNIVERSE`. The following are the supported `MPI_info` keys and the values associated with the keys:

Info Key	**Value String**
hostfile	The name of the file that contains the lists of hosts upon which to spawn MPI processes. Use a space character or a tab character to separate multiple host names.
`MPI_SGI_NODELIST`	The list of hosts upon which to spawn MPI processes. Use a space character or a tab character to separate multiple host names.

The following examples call `MPI_Comm_spawn` using the `MPI_SGI_NODELIST` and `hostlist` `MPI_info` objects to specify the hosts on which to launch MPI processes.

Example 1:

```
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "MPI_SGI_NODELIST", "host_b host_b");
MPI_Comm_spawn("b.out", MPI_ARGV_NULL, 2, info,...);
```

Example 2:

```
char * list = "host_b host_b";
int fd = open("list.txt", O_WRONLY);
write(fd, list, strlen(list) + 1);
MPI_Info_set(info, "hostfile", "list.txt");
MPI_Comm_spawn("b.out", MPI_ARGV_NULL, 2, info,...);

% export MPI_UNIVERSE="host_a 4"
% export MPI_UNIVERSE_SIZE=20
% mpiexec_mpt -np 3 coupler
```

The preceding lines run two `b.out` processes on `host_b`. If the coupler program launches any spawn processes that do not specify the desired hosts in their `info` argument then they are placed within the defined `MPI_UNIVERSE`. At any single point in time, the sum of the number of starting processes, processes launched into the hosts in `MPI_UNIVERSE`, and hosts launched onto specified hosts cannot be greater than `MPI_UNIVERSE_SIZE`.

Compiling and Running OpenSHMEM Applications

The following procedure explains how to compile and run OpenSHMEM programs in general terms.

Procedure 2-2 Compiling and Running OpenSHMEM applications

1. Use one of the OpenSHMEM wrapper compiler commands to run your program or call the compiler directly.

   To use the wrapper compiler, use one of the following commands:

   - `oshcc`
   - `oshCC`
   - `oshfort`

   To compile the OpenSHMEM program directly, use GNU compiler or Intel compiler commands.

   - To compile OpenSHMEM programs with a GNU compiler, choose one of the following commands:
     - `g++ compute.C -lsma -lmpi`
     - `gcc compute.c -lsma -lmpi`
   - To compile OpenSHMEM programs with an Intel compiler, choose one of the following commands:
     - `icc compute.C -lsma -lmpi`
     - `icc compute.c -lsma -lmpi`
     - `ifort compute.f -lsma -lmpi`

2. Use the `mpirun` command or the `mpiexec_mpt` command to launch the OpenSHMEM application.

   When you are not using a resource manager, such as PBS Professional or SLURM, set the `-np` option on the command to request the desired number of processes to launch. The `NPES` variable has no effect on OpenSHMEM programs.

   The OpenSHMEM programming model supports single-host OpenSHMEM applications.

For more information, see the `intro_shmem`(3) man page.

Building MPI Fortran Modules

The `/opt/hpe/hpc/mpt/mpt-2.16/fortran_module_generator` directory contains all the source code needed to build your own Fortran modules. This topic explains how to use environment variables and the `make`(1) command to generate the modules.

The following procedure explains how to build MPI Fortran modules.

Procedure 2-3 To build MPI Fortran modules

1. Set the environment variables you need for the compiler and (optionally) for any additional flags you need.

   The environment variables are as follows:

Variable	**Content**
`FC`	This environment variable is required. Set this variable to the compiler command used to generate the modules.

`FCFLAGS`	This environment variable is optional.
	
	If you do not set this environment variable, Intel compiler support is assumed, and the system tags buffers with `!DEC$ ATTRIBUTES NO_ARG_CHECK`.
	
	If you set this environment variable, set it to one of the following values:
	
	• `USE_GCC_FORTRAN`, which tags buffers with the following:
	`!GCC$ NO_ARG_CHECK`
	
	• `USE_PGI_FORTRAN`, which tags buffers with the foillowing:
	`!DIR$ IGNORE_TKR`
	
	• `USE_TS29113_FORTRAN`, which tags buffers with the following:
	`TYPE(*), DIMENSION(..)`

Set FCFLAGS to this value if your compiler supports TS 29113.

Example 1. If you have an Intel compiler that supports TS 29113, specify the following environment variables:

```
FC=ifort FCFLAGS="-DUSE_TS29113_FORTRAN" make
```

Example 2. If you have a GNU Fortran compiler, specify the following environment variables:

```
FC=gfortran FCFLAGS="-DUSE_GCC_FORTRAN" make
```

Example 3. If you have a GNU Fortran compiler that supports TS 29113, specify the following environment variables:

```
FC=gfortran FCFLAGS="-DUSE_TS29113_FORTRAN" make
```

2. Choose an installation path, and run the following command:

   `DESTDIR=`*install_path* `make install`

   For *install_path*, specify the path to the generated `*.mod` files.

   For example:

   ```
   % make install DESTDIR=/opt/hpe/hpc/mpt/mpt-2.16/include/custom_dir
   ```

3. Set the `MPI_CUSTOM_FORTRAN_MODULES_PATH` environment variable to the directory path you set with the `make`(1) command.

4. Set the `MPIF90_F90` environment variable and the `MPIF08_F08` environment variable to the compiler that you used to generate the Fortran modules.

5. Set the `LD_LIBRARY_PATH` environment variable to include `MPI_CUSTOM_FORTRAN_MODULES_PATH`.

   Make sure that `MPI_CUSTOM_FORTRAN_MODULES_PATH` is ordered before MPT's `LD_LIBRARY_PATH`. This ordering avoids loading the default `libmpi_f08`.

   That is, if you use the following common idiom, make sure that the following idiom is processed after the MPT module is loaded:

   ```
   LD_LIBRARY_PATH=$MPI_CUSTOM_FORTRAN_MODULES_PATH:$LD_LIBRARY_PATH
   ```

   The HPE Performance MPI compiler helpers automatically detect the path to the HPE Performance MPI Fortran modules you built. Note that the compiler helpers need to know which compiler you are using. If you do not use `mpif90` or `mpif08`, you can simply use the following compiler options:

- For Fortran 2008, use `-I$MPI_CUSTOM_FORTRAN_MODULES_PATH -LMPI_CUSTOM_FORTRAN_MODULES_PATH -lmpi_f08`
- For Fortran 2003, use `-I$MPI_CUSTOM_FORTRAN_MODULES_PATH`

6. (Optional) Update the following environment variables in the module files for your compilers and for your users:

   - `MPI_CUSTOM_FORTRAN_MODULES_PATH`
   - `MPIF90_F90`
   - `MPIF08_F08`
   - `LD_LIBRARY_PATH`

Using Huge Pages

Huge pages optimize MPI application performance. The `MPI_HUGEPAGE_HEAP_SPACE` environment variable defines the minimum amount of heap space each MPI process can allocate using huge pages. If set to a positive number, `libmpi` verifies that enough `hugetlbfs` overcommit resources are available at program start-up to satisfy that amount on all MPI processes. The heap uses all available `hugetlbfs` space, even beyond the specified minimum amount. A value of 0 disables this check and disables the allocation of heap variables on huge pages. Values can be followed by K, M, G, or T to denote scaling by 1024, 1024^2, 1024^3, or 1024^4, respectively.

For information about the `MPI_HUGEPAGE_HEAP_SPACE` environment variable, see the `mpi`(1) man page.

The following steps explain how to configure system settings for huge pages.

Procedure 2-4 To configure system settings for huge pages

1. Type the following command to make sure that the current MPT software release module is installed:

```
sys:~ # module load mpt
```

2. Log in as the root user, and type the following command to configure the system settings for huge pages:

   ```
   sys:~ # mpt_hugepage_config -u
   Updating system configuration

   System config file:         /proc/sys/vm/nr_overcommit_hugepages
   Huge Pages Allowed:         28974 pages (56 GB)  90% of memory
   Huge Page Size:             2048 KB
   Huge TLB FS Directory:      /etc/mpt/hugepage_mpt
   ```

3. Type the following command to retrieve the current system configuration:

   ```
   sys:~ #  mpt_hugepage_config -v
   Reading current system configuration

   System config file:         /proc/sys/vm/nr_overcommit_hugepages
   Huge Pages Allowed:         28974 pages (56 GB)  90% of memory
   Huge Page Size:             2048 KB
   Huge TLB FS Directory:      /etc/mpt/hugepage_mpt   (exists)
   ```

4. When running your MPT program, make sure the `MPI_HUGEPAGE_HEAP_SPACE` environment variable is set to `1`.

   This activates the new `libmpi` huge page heap. Memory allocated by calls to the `malloc` function are allocated on huge pages.

5. Log in as the root user, and type the following command to clear the system configuration settings:

   ```
   sys:~ #  mpt_hugepage_config -e
   Removing MPT huge page configuration
   ```

6. To verify that the MPT huge page configuration has been cleared, type the following command to retrieve the system configuration again:

   ```
   uv44-sys:~ # mpt_hugepage_config -v
   Reading current system configuration

   System config file:         /proc/sys/vm/nr_overcommit_hugepages
   Huge Pages Allowed:         0 pages (0 KB)  0% of memory
   Huge Page Size:             2048 KB
   Huge TLB FS Directory:      /etc/mpt/hugepage_mpt   (does not exist)
   ```

For more information about how to configure huge pages for MPI applications, see the `mpt_hugepage_config`(1) man page.

Using HPE Performance MPI in an SELinux Environment (RHEL Platforms Only)

HPE supports Security-Enhanced Linux (SELinux) for single-host runs on SGI computer systems that run the Red Hat Enterprise Linux (RHEL) operating system. The following guidelines pertain to using SELinux:

- Configure SELinux. For configuration information, see the following manual:

  SGI UV System Software Installation and Configuration Guide

- Set the `MPI_USE_ARRAY` environment variable as follows:

  ```
  MPI_USE_ARRAY=false
  ```

  When set to `false`, Array Services is disabled. For more information about this environment variable, see the `MPI`(1) man page.

For more information about how to run HPE Performance MPI with security software, contact your technical support representative.

Using HPE Performance MPI With NVIDIA GPUs

HPE Performance MPI supports the use of data buffers in graphics processing unit (GPU) memory as the source or target of data movement by MPI or OpenSHMEM functions. To enable this feature, set `MPI_USE_CUDA=true`.

If your program sends GPU data to other hosts over Mellanox InfiniBand connections, HPE Performance MPI supports the GPUDirect RDMA feature.

For more information, see the documentation from NVIDIA and Mellanox about GPUDirect RDMA.

Chapter 3

Programming With the HPE Performance Software — Message Passing Interface (HPE Performance MPI)

This chapter includes the following topics:

About Programming With HPE Performance MPI

Portability is one of the main advantages MPI has over vendor-specific message passing software. Nonetheless, the MPI Standard offers sufficient flexibility for general variations in vendor implementations. In addition, there are often vendor-specific programming recommendations for optimal use of the MPI library. This chapter's topics explain how to develop or port MPI applications to SGI systems.

Job Termination and Error Handling

This section describes the behavior of the HPE Performance MPI implementation upon normal job termination. Error handling and characteristics of abnormal job termination are also described.

This section includes the following topics:

`MPI_Abort`

In the HPE Performance MPI implementation, a call to `MPI_Abort` has the following effect:

- The MPI job terminates, regardless of the communicator argument used.
- The error code value is returned as the exit status of the `mpirun` command.
- A stack traceback is displayed that shows where the program called `MPI_Abort`.

Error Handling

The MPI Standard describes MPI error handling. Although almost all MPI functions return an error status, an error handler is invoked before returning from the function. If the function has an associated communicator, the error handler associated with that communicator is invoked. Otherwise, the error handler associated with `MPI_COMM_WORLD` is invoked.

The HPE Performance MPI implementation provides the following predefined error handlers:

- `MPI_ERRORS_ARE_FATAL`. When called, causes the program to abort on all executing processes. This has the same effect as if `MPI_Abort` were called by the process that invoked the handler.
- `MPI_ERRORS_RETURN`. This handler has no effect.

By default, the `MPI_ERRORS_ARE_FATAL` error handler is associated with `MPI_COMM_WORLD` and any communicators derived from it. Hence, to handle the error statuses returned from MPI calls, it is necessary to associate either the `MPI_ERRORS_RETURN` handler or another user-defined handler with `MPI_COMM_WORLD` near the beginning of the application.

`MPI_Finalize` and Connect Processes

In the HPE implementation of MPI, all pending communications involving an MPI process must be complete before the process calls `MPI_Finalize`. If there are any pending `send` or `recv` requests that are unmatched or not completed, the application hangs in `MPI_Finalize`. For more details, see the MPI Standard.

If the application uses the MPI remote memory access (RMA) spawn functionality described in the MPI RMA standard, there are additional considerations. In the HPE implementation, all MPI processes are connected. The MPI RMA standard defines what is meant by connected processes. When the MPI RMA spawn functionality is used, `MPI_Finalize` is collective over all connected processes. Thus all MPI processes, both launched on the command line, or subsequently spawned, synchronize in `MPI_Finalize`.

Signals

In the MPI implementation, MPI processes are Linux processes. As such, the general rule regarding signal handling applies as it would to ordinary Linux processes.

In addition, certain signals can be propagated from the `mpirun` process to the other processes in the MPI job, whether they belong to the same process group on a single host or are running across multiple hosts in a cluster. These signals are as follows:

- `SIGURG`
- `SIGUSR1`
- `SIGINT`
- `SIGTERM`

To use this feature, the MPI program must have a signal handler that catches the signal. When the signal is sent to the `mpirun` process ID, the `mpirun` process catches the signal and propagates it to all MPI processes.

Buffering

Most MPI implementations use buffering for overall performance reasons, and some programs depend on it. However, you should not assume that there is any message buffering between processes because the MPI Standard does not mandate a buffering strategy. Table 3-1 on page 32 illustrates a simple sequence of MPI operations that cannot work unless messages are buffered. If sent messages are not buffered, each process hangs in the initial call, waiting for an `MPI_Recv` call to take the message.

Because most MPI implementations buffer messages to some degree, a program like this does not usually hang. The `MPI_Send` calls return after putting the messages into buffer space, and the `MPI_Recv` calls get the messages. Nevertheless, program logic like this is not valid according to the MPI Standard. Programs that require this sequence of MPI calls should employ one of the buffer MPI send calls, `MPI_Bsend` or `MPI_Ibsend`.

Table 3-1 Outline of Improper Dependence on Buffering

Process 1	Process 2
`MPI_Send(2,....)`	`MPI_Send(1,....)`
`MPI_Recv(2,....)`	`MPI_Recv(1,....)`

By default, the HPE implementation of MPI uses buffering under most circumstances. Short messages (64 or fewer bytes) are always buffered. Longer messages are also buffered, although under certain circumstances, buffering can be avoided. For performance reasons, it is sometimes desirable to avoid buffering. For further information on unbuffered message delivery, see "Programming Optimizations" on page 34.

Multithreaded Programming

HPE Performance MPI supports a hybrid programming model, in which MPI handles one level of parallelism in an application and POSIX threads or OpenMP processes are used to handle another level. When mixing OpenMP with MPI, for performance reasons, it is better to consider invoking MPI functions only outside parallel regions or only from within master regions. When used in this manner, it is not necessary to initialize MPI for thread safety. You can use `MPI_Init` to initialize MPI. However, to safely invoke MPI functions from any OpenMP process or when using Posix threads, MPI must be initialized with `MPI_Init_thread`.

When using `MPI_Thread_init`() with the threading level `MPI_THREAD_MULTIPLE`, link your program as follows:

- If you use the compiler wrappers for MPI or SHMEM, use the `-mt` option on the command line.
- If you want to call the compilers directly, use the `-lmpi_mt` parameter instead of the `-lmpi` parameter on the compiler command line.

For more information about compiling and linking MPI programs, see the `mpi`(1) man page.

Interoperability with the OpenSHMEM programming model

You can mix OpenSHMEM and MPI message passing in the same program. The application must be linked with both the OpenSHMEM and MPI libraries.

Start with an MPI program that calls `MPI_Init` (or `MPI_Init_thread()`) and `MPI_Finalize`. Next, add OpenSHMEM calls, and be aware that the PE numbers are equal to the MPI rank numbers in `MPI_COMM_WORLD`.

If your program uses both OpenSHMEM and MPI, make sure your program includes calls to the `shmem_init()` and `shmem_finalize()` library routines. This practice is similar to how you include calls to `MPI_Init()` (or `MPI_Init_thread()`) and `MPI_Finalize`.

When running the application across a cluster using OpenSHMEM and OpenSHMEM functions, some processes might not be able to communicate with other processes. You can use the `shmem_pe_accessible` and `shmem_addr_accessible` functions to determine whether an OpenSHMEM call can be used to access data residing in another process. Because the OpenSHMEM model functions only with respect to `MPI_COMM_WORLD`, these functions cannot be used to exchange data between MPI processes that are connected via MPI intercommunicators returned from MPI spawn-related functions.

For more information about the OpenSHMEM programming model, see the `intro_shmem`(3) man page.

Miscellaneous HPE Performance MPI Features

The following other characteristics of the HPE Performance MPI implementation might interest you:

- `stdin/stdout/stderr`.

In this implementation, `stdin` is enabled for only the process that is rank 0 in the first `MPI_COMM_WORLD`. Such processes do not need to be located on the same host as `mpirun`. The `stdout` and `stderr` results are enabled for all MPI processes in the job, whether started by `mpirun` or started by one of the MPI spawn functions.

- `MPI_Get_processor_name`

  The `MPI_Get_processor_name` function returns the Internet host name of the computer upon which the MPI process that started this subroutine is running.

Programming Optimizations

You might need to modify your MPI application to use the HPE Performance MPI optimization features.

The following topics describe how to use the optimized features of HPE's MPI implementation:

- "Using MPI Point-to-Point Communication Routines" on page 34
- "Using MPI Collective Communication Routines" on page 35
- "Using `MPI_Pack/MPI_Unpack`" on page 35
- "Avoiding Derived Data Types" on page 36
- "About Wild Cards" on page 36
- "Avoiding Message Buffering — Single Copy Methods" on page 36
- "Managing Memory Placement" on page 37

Using MPI Point-to-Point Communication Routines

MPI provides a number of different routines for point-to-point communication. The most efficient ones in terms of latency and bandwidth are the blocking and nonblocking `send/receive` functions, which are as follows:

- `MPI_Send`
- `MPI_Isend`
- `MPI_Recv`

- `MPI_Irecv`

Unless required for application semantics, avoid the synchronous send calls, which are as follows:

- `MPI_Ssend`
- `MPI_Issend`

Also avoid the buffered send calls, which double the amount of memory copying on the sender side. These calls are as follows:

- `MPI_Bsend`
- `MPI_Ibsend`

This implementation treats the ready-send routines, `MPI_Rsend` and `MPI_Irsend`, as standard `MPI_Send` and `MPI_Isend` routines. Persistent requests do not offer any performance advantage over standard requests in this implementation.

Using MPI Collective Communication Routines

The MPI collective calls are frequently layered on top of the point-to-point primitive calls. For small process counts, this can be reasonably effective. However, for higher process counts of 32 processes or more, or for clusters, this approach can be less efficient. For this reason, a number of the MPI library collective operations have been optimized to use more complex algorithms.

HPE's MPI collectives have been optimized for use with clusters. In these cases, steps are taken to reduce the number of messages using the relatively slower interconnect between hosts.

Some of the collective operations have been optimized for use with shared memory. The `MPI_Alltoall` routines also use special techniques to avoid message buffering when using shared memory. For more information, see "Avoiding Message Buffering — Single Copy Methods" on page 36.

Using `MPI_Pack`/`MPI_Unpack`

While `MPI_Pack` and `MPI_Unpack` are useful for porting parallel virtual machine (PVM) codes to MPI, they essentially double the amount of data to be copied by both the sender and receiver. Generally, either restructure your data or use derived data types to avoid using these functions. Note, however, that use of derived data types can lead to decreased performance in certain cases.

Avoiding Derived Data Types

Avoid derived data types when possible. In the HPE implementation, using derived data types does not generally lead to performance gains. Using derived data types might disable certain types of optimizations, for example, unbuffered or single copy data transfer.

About Wild Cards

The use of wild cards (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`) involves searching multiple queues for messages. While this is not significant for small process counts, for large process counts, the cost increases quickly.

MPT can make certain optimizations if the application does not make calls to variations of `MPI_Recv()` with `MPI_ANY_SOURCE`. When `MPI_WILDCARDS=false` is in effect, MPT assumes that the application does not contain receive calls with rank wild cards. This assumption enables MPT to make some bandwidth optimizations in its Intel Omni-Path Architecture code. MPT supports the `MPI_WILDCARDS` environment variable only on systems that include the Intel Omni-Path Architecture. For information about more environment variables that MPT supports on the Intel Onmi-Path Architecture, see the following:

"Tuning for Running Applications Over the Intel Omni-Path Interconnect" on page 61

Avoiding Message Buffering — Single Copy Methods

One of the most significant optimizations for bandwidth-sensitive applications in the MPI library is single-copy optimization, which avoids using shared memory buffers. However, as discussed in "Buffering" on page 31, some incorrectly coded applications might hang because of buffering assumptions. For this reason, this optimization is not enabled by default for `MPI_Send`, but you can use the `MPI_BUFFER_MAX` environment variable to enable this optimization at run time. The following guidelines show how to increase the opportunity for use of the unbuffered pathway:

- The MPI data type on the send side must be a contiguous type.
- The sender and receiver MPI processes must reside on the same host.

- The sender data must be globally accessible by the receiver. The HPE Performance MPI implementation allows data allocated from the static region (common blocks), the private heap, and the stack region to be globally accessible. In addition, memory allocated via the `MPI_Alloc_mem` function or the SHMEM symmetric heap accessed via the `shpalloc` or `shmalloc` functions is globally accessible.

Certain run-time environment variables must be set to enable the unbuffered, single-copy method. For information about how to set the run-time environment, see "Avoiding Message Buffering – Enabling Single Copy" on page 52.

Managing Memory Placement

SGI UV series systems have a ccNUMA memory architecture. For single-process and small multiprocess applications, this architecture behaves similarly to flat memory architectures. For more highly parallel applications, memory placement becomes important. MPI takes placement into consideration when it lays out shared memory data structures and the individual MPI processes' address spaces. Generally, you should not try to manage memory placement explicitly. To control the placement of the application at run time, however, see Chapter 7, "Run-time Tuning" on page 49.

Additional Programming Model Considerations

A number of additional programming options might be worth consideration when developing MPI applications for SGI systems. For example, using the SHMEM programming model can improve performance in the latency-sensitive sections of an application. Usually, this requires replacing MPI `send`/`recv` calls with `shmem_put`/`shmem_get` and `shmem_barrier` calls. The SHMEM programming model can deliver significantly lower latencies for short messages than traditional MPI calls. As an alternative to `shmem_get`/`shmem_put` calls, you might consider the MPI remote memory accesss (RMA) `MPI_Put`/ `MPI_Get` functions. These provide almost the same performance as the SHMEM calls, while providing a greater degree of portability.

Alternately, you might consider exploiting the shared memory architecture of SGI systems by handling one or more levels of parallelism with OpenMP, with the coarser grained levels of parallelism being handled by MPI. Also, there are special ccNUMA placement considerations to be aware of when running hybrid MPI/OpenMP applications. For further information, see Chapter 7, "Run-time Tuning" on page 49.

Debugging MPI Applications

This chapter includes the following topics:

MPI Routine Argument Checking

Debugging MPI applications can be more challenging than debugging sequential applications. By default, the HPE Performance Software — Message Passing Interface (HPE Performance MPI) implementation does not check the arguments to some performance-critical MPI routines, such as most of the point-to-point and collective communication routines. You can force HPE Performance MPI to always check the input arguments to MPI functions by setting the `MPI_CHECK_ARGS` environment variable. However, setting this variable might result in some degradation in application performance, so it is not recommended that it be set except when debugging.

Using the TotalView Debugger with MPI Programs

The Message Passing Toolkit (MPT) `mpiexec_mpt`(1) command has a `-tv` option for use by MPT with the TotalView Debugger. Note that the PBS Professional `mpiexec`(1) command does not support the `-tv` option. TotalView does not operate with MPI processes started via the `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` functions.

Example 1. To run an MPT MPI job with TotalView without a workload manager, type the following:

```
% totalview mpirun -a -np 4 a.out
```

Example 2. To run an MPT MPI job with the TotalView Debugger with a workoad manager, such as PBS Professional or Torque, type the following:

```
% mpiexec_mpt -tv -np 4 a.out
```

Using `idb` and `gdb` with MPI Programs

Because the `idb` and `gdb` debuggers are designed for sequential, non-parallel applications, they are generally not well suited for use in MPI program debugging and development. However, the use of the `MPI_SLAVE_DEBUG_ATTACH` environment variable makes these debuggers more usable.

If you set the `MPI_SLAVE_DEBUG_ATTACH` environment variable to a global rank number, the MPI process sleeps briefly in startup while you use `idb` or `gdb` to attach to the process. A message is printed to the screen, telling you how to use `idb` or `gdb` to attach to the process.

Similarly, if you want to debug the MPI daemon, setting `MPI_DAEMON_DEBUG_ATTACH` sleeps the daemon briefly while you attach to it.

Using the DDT Debugger with MPI Programs

Allinea Software's DDT product is a parallel debugger that supports MPT. You can run DDT in either interactive (online) or batch (offline) mode. In batch mode, DDT can create a text or HTML report that tracks variable values and shows the location of any errors. DDT records the data for a program's variables across all processes, and DDT logs values in the HTML output files as sparkline charts.

For information about how to configure Allinea for use with MPI on SGI systems, use the instructions in the Allinea user guide that is posted to the following website:

http://content.allinea.com/downloads/userguide.pdf

Example 1. The following command starts DDT in interactive (online) mode:

```
# ddt -np 4 a.out
```

Example 2. The following command generates a debugging report in HTML format:

```
# ddt -offline my-log.html -np 4 a.out
```

Example 3. Assume that you want to trace variables `x`, `y`, and `my_arr(x,y)` in parallel across all processes. The following command directs DDT to record the values of `x`, `y`, and `my_arr(x,y)` each time it encounters line 147:

```
# ddt -offline my-log.html -trace-at "my-file.f:147,x,y,my_arr(x,y)" -np 4 a.out
```

Example 4. You can specify batch (offline) DDT commands from within a queue submission script. Instead of specifying `mpiexec_mpt -np 4 a.out`, specify the following:

```
# ddt -noqueue -offline my-log.html -trace-at "my-file.f:147,x,y,my_arr(x,y)" -np 4 a.out
```

Using Valgrind With MPI Programs

Valgrind is a tool that can profile your program and can automatically detect memory management and threading bugs.

Valgrind is not compatible with the memory mapping functionality in MPT. When MPT detects that Valgrind is in use, MPT automatically enables the `MPI_MEMMAP_OFF` environment variable, which disables MPT's own memory mapping.

Chapter 5

Using `perfboost`

This chapter includes the following topics:

About `perfboost`

The performance boosting tool, `perfboost`, uses a wrapper library to run applications compiled against other Message Passing Interface (MPI) implementations under the Message Passing Toolkit (MPT) product on HPE hardware platforms. This chapter describes how to use `perfboost`.

Note: `perfboost` does not support the MPI C++ API.

Using `perfboost`

The following procedure explains how to use `perfboost` with an HPE Performance Software — Message Passing Interface (HPE Performance MPI) program.

Procedure 5-1 To use `perfboost`

1. Load the `perfboost` environment module.

   The module include the `PERFBOOST_VERBOSE` environment variable.

   If you set the `PERFBOOST_VERBOSE` environment variable, it enables a message when `PerfBoost` activates and also when the MPI application is completed through the `MPI_Finalize()` function. This message indicates that the `perfboost` library is active and also indicates when the MPI application completes through the `libperfboost` wrapper library.

The MPI environment variables that are documented in the MPI(1) man page are available to perfboost. MPI environment variables that are not used by MPT are currently not supported.

Note: Some applications redirect stderr. In this case, the verbose messages might not appear in the application output.

---

2. Type a command that inserts the perfboost command in front of the executable name along with the choice of MPI implementation to emulate.

   In other words, run the executable file with the MPT mpiexec_mpt(1) or the mpirun(1) command.

   The following are MPI implementations and corresponding command line options:

Implementation	**Command Line Option**
Platform MPI 7.1+	-pmpi
Intel MPI	-impi
OpenMPI	-ompi
MPICH1	-mpich
MPICH2, version 2 and later	-impi
MVAPICH2, version 2 and later	-impi

The following are some examples that use perfboost:

```
% module load mpt
% module load perfboost

% mpirun -np 32 perfboost -impi a.out arg1
% mpiexec_mpt perfboost -pmpi b.out arg1
% mpirun host1 32, host2 64 perfboost -impi c.out arg1 arg2
```

MPI Supported Functions

perfboost supports the commonly used elements of the C and Fortran MPI APIs. If a function is not supported, the job aborts and issues an error message. The message shows the name of the missing function. You can contact the customer support center at the following website to schedule a missing function to be added to `perfboost`:

https://support.sgi.com/caselist

Berkeley Lab Checkpoint/Restart (BLCR)

This chapter includes the following topics:

- "About BLCR" on page 47
- "BLCR Installation" on page 47
- "Using BLCR with MPT" on page 48

About BLCR

The Message Passing Toolkit (MPT) supports BLCR checkpoint/restart. This checkpoint/restart implementation allows applications to periodically save a copy of their state. Applications can resume from that point if the application crashes or if the job is aborted to free resources for higher-priority jobs.

The following are the implementation's limitations:

- BLCR does not checkpoint the state of any data files that the application might be using.
- When using checkpoint/restart, the Message Passing Interface (MPI) does not support certain features, including spawning and one-sided MPI.
- InfiniBand XRC queue pairs are not supported.
- Checkpoint files are often very large and require significant disk bandwidth to create in a timely manner.

For more information on BLCR, see the following:

http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/

BLCR Installation

To use checkpoint/restart with MPT, BLCR must first be installed.

Procedure 6-1 To install BCLR

1. Log in as the root user.

2. Install the `blcr-`, `blcr-libs-`, and `blcr-kmp-` RPMs.

   BLCR uses a kernel module that must be built against the specific kernel that the operating system is running. If the kernel module fails to load, you need to rebuild and reinstall. Install the `blcr-` source RPM. In the `blcr.spec` file, set the kernel variable to the name of the current kernel, then rebuild and install the new set of RPMs.

3. Type the following command to enable BLCR:

   ```
   # chkconfig blcr on
   ```

Using BLCR with MPT

To enable checkpoint/restart within MPT, you need to pass the `-cpr` option to `mpirun` or `mpiexec_mpt`. For example:

```
% mpirun -cpr hostA, hostB -np 8 ./a.out
```

To checkpoint a job, run the `mpt_checkpoint` command on the same host upon which `mpirun` is running. Make sure to pass the `mpt_checkpoint` command the PID of `mpirun` and the name with which you want to prefix all the checkpoint files. For example:

```
% mpt_checkpoint -p 12345 -f my_checkpoint
```

The preceding example command creates a `my_checkpoint.cps` metadata file and a number of `my_checkpoint.*.cpd` files.

To restart the job, pass the name of the `.cps` file to `mpirun`. For example:

```
% mpirun -cpr hostC, hostD -np 8 mpt_restart my_checkpoint.cps
```

You can restart the job on a different set of hosts, but the number of hosts must be the same. In addition, each host must have the same number of ranks as the corresponding host in the original run of the job.

Chapter 7

Run-time Tuning

This chapter includes the following topics:

About Run-time Tuning

This chapter describes the ways in which a user can tune the run-time environment to improve the performance of a Message Passing Interface (MPI) message passing application on SGI hardware platforms. None of these ways involve application code changes.

The run-time tuning topics are as follows:

- "Reducing Run-time Variability" on page 50
- "Tuning MPI Buffer Resources" on page 51
- "Avoiding Message Buffering – Enabling Single Copy" on page 52
- "Memory Placement and Policies" on page 53
- "Tuning MPI/OpenMP Hybrid Codes" on page 55

- "Tuning Running Applications Across Multiple Hosts" on page 57
- "Tuning for Running Applications over the InfiniBand Interconnect" on page 59
- "Tuning for Running Applications Over the Intel Omni-Path Interconnect" on page 61
- "MPI on SGI UV Systems" on page 62
- "Suspending MPI Jobs" on page 67

Reducing Run-time Variability

One of the most common problems with optimizing message passing codes on large, shared-memory computers is to achieve reproducible timings from run to run. To reduce run-time variability, you can take the following precautions:

- Do not oversubscribe the system. In other words, do not request more CPUs than are available, and do not request more memory than is available. Oversubscribing causes the system to wait unnecessarily for resources to become available, leads to variations in the results, and leads to less than optimal performance.
- Avoid interference from other system activity. The Linux kernel uses more memory on node 0 than on other nodes. Node 0 is also known as *the kernel node*. If your application uses almost all of the available memory per processor, the memory for processes assigned to the kernel node can unintentionally spill over to nonlocal memory. By keeping user applications off of the kernel node, you can avoid this effect.

  By restricting system daemons to run on the kernel node, you can also deliver an additional percentage of each application CPU to the user program.
- Avoid interference with other applications. If necessary, use cpusets to address this problem. The cpuset software enables you to partition a large, distributed memory host in a fashion that minimizes interactions between jobs running concurrently on the system. For more information about cpusets, see the following:

  HPE Performance Software - Message Passing Interface Cpuset Software Guide
- On a quiet, dedicated system, you can use the `dplace`(1) tool or the `MPI_DSM_CPULIST` environment variable to improve run-time performance repeatability. These approaches are not suited to shared, nondedicated systems.

- Use a workload manager such as Platform LSF from IBM or PBS Professional from Altair Engineering, Inc. These workload managers use cpusets to avoid oversubscribing the system and to avoid possible interference between applications.

Tuning MPI Buffer Resources

By default, the HPE Performance Software — Message Passing Interface (HPE Performance MPI) implementation buffers messages that are longer than 64 bytes. The system buffers these longer messages in a series of 16 KB buffers. Messages that exceed 64 bytes are handled as follows:

- If the message is 128 K in length or shorter, the sender MPI process buffers the entire message.

  In this case, the sender MPI process delivers a message header, also called a *control message*, to a mailbox. When an MPI call is made, the MPI receiver polls the mail box. If the receiver finds a matching receive request for the sender's control message, the receiver copies the data out of the buffers into the application buffer indicated in the receive request. The receiver then sends a message header back to the sender process, indicating that the buffers are available for reuse.

- If the message is longer than 128 K, the software breaks the message into chunks that are 128 K in length.

  The smaller chunks allow the sender and receiver to overlap the copying of data in a pipelined fashion. Because there are a finite number of buffers, this can constrain overall application performance for certain communication patterns. You can use the `MPI_BUFS_PER_PROC` shell variable to adjust the number of buffers available for each process, and you can use the MPI statistics counters to determine if the demand for buffering is high.

  Generally, you can avoid excessive numbers of retries for buffers if you increase the number of buffers. However, when you increase the number of buffers, you consume more memory, and you might increase the probability for cache pollution. *Cache pollution* is the excessive filling of the cache with message buffers. Cache pollution can degrade performance during the compute phase of a message passing application.

For information about statistics counters, see "MPI Internal Statistics" on page 76.

For information about buffering considerations when running an MPI job across multiple hosts, see "Tuning Running Applications Across Multiple Hosts" on page 57.

For information about the programming implications of message buffering, see "Buffering" on page 31.

Avoiding Message Buffering – Enabling Single Copy

It is possible to avoid the need to buffer messages for message transfers between MPI processes within the same host or message transfers that use devices that support remote direct memory access (RDMA), such as InfiniBand.

The following topics provide more information about buffering:

- "Buffering and `MPI_Send`" on page 52
- "Using the XPMEM Driver for Single Copy Optimization" on page 52

Buffering and `MPI_Send`

Many MPI applications are written to assume infinite buffering, so message buffering is not enabled by default for `MPI_Send`. For `MPI_Isend`, `MPI_Sendrecv`, and most collectives, this optimization is enabled by default for large message sizes. To disable this default, single-copy feature used for the collectives, use the `MPI_DEFAULT_SINGLE_COPY_OFF` environment variable.

Using the XPMEM Driver for Single Copy Optimization

MPI uses the XPMEM driver to support single-copy message transfers between two processes within the same host.

Enabling single-copy transfers can increase performance because this technique improves MPI’s bandwidth. On the other hand, single-copy transfers can introduce additional synchronization points, which can reduce application performance.

The `MPI_BUFFER_MAX` environment variable specifies the threshold for message lengths. Its value should be set to the message length, in bytes, beyond which you want MPI to use the single-copy method. In general, a value of 2000 or higher is beneficial for many applications.

During job startup, MPI uses the XPMEM driver, via the `xpmem` kernel module, to map memory from one MPI process to another. The mapped areas include the static (BSS) region, the private heap, the stack region, and (optionally) the symmetric heap region of each process.

Memory mapping allows each process to directly access memory from the address space of another process. This technique allows MPI to support single-copy transfers for contiguous data types from any of these mapped regions. For these transfers between processes residing on the same host, MPI uses the `bcopy` process to copy the data. The `bcopy` process also transfers data between two different executable files on the same host. For data residing outside of a mapped region (a `/dev/zero` region, for example), MPI uses a buffering technique to transfer the data.

Memory mapping is enabled by default. To disable it, set the `MPI_MEMMAP_OFF` environment variable. Memory mapping must be enabled to allow single-copy transfers, MPI remote memory access (RMA) one-sided communication, support for the SHMEM model, and certain collective optimizations.

Memory Placement and Policies

The MPI library takes advantage of NUMA placement functions that are available. Usually, the default placement is adequate. However, you can set one or more environment variables to modify the default behavior.

For a complete list of the environment variables that control memory placement, see the `MPI`(1) man page.

The following topics contain information on environment variables and tools that enable you to tune memory placement:

- "`MPI_DSM_CPULIST`" on page 53
- "`MPI_DSM_DISTRIBUTE`" on page 55
- "`MPI_DSM_VERBOSE`" on page 55
- "Using `dplace`" on page 55

MPI_DSM_CPULIST

The `MPI_DSM_CPULIST` environment variable allows you to select the processors to use for an MPI application. At times, specifying a list of processors on which to run a job can be the best means to insure highly reproducible timings, particularly when running on a dedicated system.

The setting is an ordered list that uses commas (`,`) and hyphens (`-`) to specify a mapping of MPI processes to CPUs. If running across multiple hosts, separate the per-host components of the CPU list with a colon (`:`). Wen using a hyphen-delineated list, you can specify CPU striding by specifying /*stride_distance* after the list.

For example:

Value	**CPU Assignment**
`8,16,32`	Place three MPI processes on CPUs 8, 16, and 32.
`32,16,8`	Place the MPI process rank zero on CPU 32, one on 16, and two on CPU 8.
`8-15/2`	Place the MPI processes 0 through 3 strided on CPUs 8, 10, 12, and 14.
`8-15,32-39`	Place the MPI processes 0 through 7 on CPUs 8 to 15. Place the MPI processes 8 through 15 on CPUs 32 to 39.
`39-32,8-15`	Place the MPI processes 0 through 7 on CPUs 39 to 32. Place the MPI processes 8 through 15 on CPUs 8 to 15.
`8-15:16-23`	Place the MPI processes 0 through 7 on the first host on CPUs 8 through 15. Place MPI processes 8 through 15 on CPUs 16 to 23 on the second host.

Note that the process rank is the `MPI_COMM_WORLD` rank. The interpretation of the CPU values specified in the `MPI_DSM_CPULIST` depends on whether the MPI job is being run within a cpuset, as follows:

- If the job is run outside of a cpuset, the CPUs specify *cpunum* values beginning with 0 and up to the number of CPUs in the system, minus one.
- If the job is run within a cpuset, the default behavior is to interpret the CPU values as relative processor numbers within the cpuset.

The number of processors specified should equal the number of MPI processes that are used to run the application. The number of colon-delineated parts of the list must equal the number of hosts used for the MPI job. If an error occurs in processing the CPU list, the default placement policy is used.

MPI_DSM_DISTRIBUTE

The MPI_DSM_DISTRIBUTE environment variable ensures that each MPI process gets a physical CPU and memory on the node to which it was assigned. MPI_DSM_DISTRIBUTE assigns MPI ranks, as follows:

- On systems that do not include InfiniBand interconnect, MPI_DSM_DISTRIBUTE assigns MPI ranks starting at logical CPU 0 and incrementing until all ranks have been placed.
- On systems that include InfiniBand interconnect, if the job spans hosts, MPI_DSM_DISTRIBUTE assigns MPI ranks starting with the CPU that is closest to the first InfiniBand host channel adapter (HCA).

If you set both MPI_DSM_DISTRIBUTE and MPI_DSM_CPULIST, MPI_DSM_CPULIST overrides MPI_DSM_DISTRIBUTE.

MPI_DSM_VERBOSE

Setting the MPI_DSM_VERBOSE environment variable directs MPI to display a synopsis of the NUMA and host placement options being used at run time.

Using dplace

The dplace tool offers another way to specify the placement of MPI processes within a distributed memory host. The dplace tool and MPI interoperate, and allow MPI to better manage placement of certain shared memory data structures.

For information about dplace with MPI, see the following:

- "dplace Command" on page 106
- The dplace(1) man page.
- The *Linux Application Tuning Guide*.

Tuning MPI/OpenMP Hybrid Codes

A hybrid MPI/OpenMP application is one in which each MPI process itself is a parallel threaded program. These programs often exploit the OpenMP paralllelism at the loop level while also implementing a higher-level parallel algorithm that uses MPI.

Many parallel applications perform better if the MPI processes and the threads within them are pinned to particular processors for the duration of their execution. For ccNUMA systems, this pinning ensures that all local, non-shared memory is allocated on the same memory node as the processor referencing the memory. For all systems, pinning can ensure that some or all of the OpenMP threads stay on processors that share a bus or perhaps a processor cache, which can speed up thread synchronization.

The Message Passing Toolkit (MPT) provides the `omplace`(1) command to help with the placement of OpenMP threads within an MPI program. The `omplace`(1) command causes the threads in a hybrid MPI/OpenMP job to be placed on unique CPUs within the containing cpuset. For example, the threads in a 2-process MPI program with 2 threads per process would be placed as follows:

- Rank 0, thread 0 on CPU 0
- Rank 0, thread 1 on CPU 1
- Rank 1, thread 0 on CPU 2
- Rank 1, thread 1 on CPU 3

The CPU placement is performed by dynamically generating a `dplace`(1) placement file and invoking `dplace`.

For more information, see the following:

- "NUMA Tools" on page 105
- The `omplace`(1) man page.
- The `dplace`(1) man page.
- The *Linux Application Tuning Guide for SGI X86-64 Based Systems.*
- The *HPE Performance Software - Message Passing Interface Cpuset Software Guide.*

Example 7-1 Running a Hybrid MPI/OpenMP Application

The following command line runs a hybrid MPI/OpenMP application with eight MPI processes that are two-way threaded on two hosts:

```
mpirun host1,host2 -np 4 omplace -nt 2 ./a.out
```

- When using the PBS workload manager to schedule the hybrid MPI/OpenMP job, use the following resource allocation specification:

  ```
  #PBS -l select=8:ncpus=2
  ```

- In addition, use the following `mpiexec_mpt` command:

  ```
  mpiexec_mpt -n 8 omplace -nt 2 ./a.out
  ```

For more information about running MPT programs with PBS, see the following:

"Using a Workload Manager to Launch an MPI Application" on page 15

Tuning Running Applications Across Multiple Hosts

When you run an MPI application across a cluster of hosts, you can use the environment variables in this topic to improve application performance across these hosts.

Table 7-1 on page 57 shows the interconnect types and the run-time environment settings and configurations that you can use to improve performance.

Table 7-1 Available Interconnects and the Inquiry Order for Available Interconnects

Interconnect Type	Default Order of Selection	Environment Variable Required
XPMEM	1	`MPI_USE_XPMEM`
Intel Omni-Path Architecture	2	`MPI_USE_OPA`
InfiniBand	3	`MPI_USE_IB`
InfiniBand Unreliable Datagram	4	`MPI_USE_UD`
TCP/IP	5	`MPI_USE_TCP`

Table 7-1 on page 57 shows the different types of interconnects that systems can employ as the multihost interconnect. When launched as a distributed application, MPI probes for these interconnects at job startup. For information about how to launch a distributed application, see "Using the `mpirun` Command to Launch an MPI Application" on page 17.

When MPI detects a high-performance interconnect, MPI attempts to use this interconnect, if it is available, on every host being used by the MPI job. If the interconnect is not available for use on every host, the library attempts to use the next slower interconnect until this connectivity requirement is met. Table 7-1 on page 57 specifies the order in which MPI probes for available interconnects.

The third column of Table 7-1 on page 57 indicates the environment variable you can set to pick a particular interconnect other than the default. In general, to insure the best application performance, allow MPI to pick the fastest available interconnect.

When using the TCP/IP interconnect, unless specified otherwise, MPI uses the default IP adapter for each host. To use a nondefault adapter, enter the adapter-specific host name on the `mpirun` command line.

The following environment variables enable you to tune your application for multiple hosts:

Variable	**Purpose**
`MPI_IB_RAILS`	When set to `1` and the MPI library uses the InfiniBand driver as the inter-host interconnect, MPT sends InfiniBand traffic over the first fabric that it detects. Default on all SGI UV systems. When set to `1+`, MPT sends all traffic across the first fabric, but if it encounters communication problems, it starts to use both fabrics. When set to `2`, the library tries to use multiple, available, separate, InfiniBand fabrics and splits the traffic across them.
`MPI_IB_SINGLE_COPY_BUFFER_MAX`	If MPI transfers data over InfiniBand and if the size of the cumulative data is greater than this value, then MPI attempts to send the data directly between the processes's buffers and not through intermediate buffers inside the MPI library. The default is 32767.
`MPI_USE_IB`	When set, the MPI library uses the InfiniBand driver as the interconnect when running across multiple hosts or running with multiple binaries. MPT requires the OFED software stack when the

InfiniBand interconnect is used. If InfiniBand is used, the `MPI_COREDUMP` environment variable is forced to `INHIBIT`, to comply with the InfiniBand driver restriction that no `fork()` actions occur after InfiniBand resources have been allocated.

The default is `false`.

For more information on these environment variables, see the ENVIRONMENT VARIABLES section of the `mpi`(1) man page.

Tuning for Running Applications over the InfiniBand Interconnect

When you run an MPI application across a cluster of hosts using the InfiniBand interconnect, there are run-time environment variables that you can set to to improve application performance. The following are these variables:

Variable **Purpose**

`MPI_HCOLL_IB_OFFLOAD`

Enables or disables the Mellanox fabric collectives accelerator (FCA) offload. If FCA offload is configured on your cluster, set `MPI_HCOLL_IB_OFFLOAD=true`.

Make sure that the Mellanox HCOLL libraries are specified in your library path. You can do this by either loading the `hpcx` software module, if available, or by making sure that the location of `libhcoll.so` is in your `LD_LIBRARY_PATH` environment variable.

The default is `MPI_COLL_HCOLL=false`.

`MPI_CONNECTIONS_THRESHOLD`

For very large MPI jobs, the time and resource cost to create a connection between every pair of ranks at job start time can be prodigious. When the number of ranks is set to at least this value, the MPI library creates InfiniBand connections on a demand basis. The default is 1025 ranks.

`MPI_IB_FAILOVER`

When the MPI library uses InfiniBand fabric and this variable is set, if an InfiniBand transmission error occurs, MPT tries to restart the connection to the other rank a certain number of times. The

MPI_IB_FAILOVER variable specifies the number of times MPT tries to restart the connection. MPT can handle a number of errors of this type between any pair of ranks equal to the value of this variable. The default is 32 times.

MPI_IB_PAYLOAD

When the MPI library uses InfiniBand fabric, it allocates memory for each message header that it uses for InfiniBand. If the size of data to be sent is not greater than this amount minus 64 bytes for the actual header, the data is inlined with the header. If the size is greater than this value, then the message is sent through remote direct memory access (RDMA) operations. The default is 16512 bytes.

MPI_IB_RNR_TIMER

When a packet arrives at an InfiniBand host channel adaptor (HCA) and there are no remaining receive buffers for it, the receiving HCA sends a negative acknowledgement (NAK) to the requestor. The requesting HCA tries again after some period of time, and this variable controls the delay time.

If you set a value higher than the default, performance can degrade in some circumstances. The higher value, however, is likely to improve fabric health significantly during high congestion. For precise translations of this value to delay times, see Table 45 of the official InfiniBand specification. The default is 14.

MPI_IB_TIMEOUT

When an InfiniBand card sends a packet, it waits some amount of time for an ACK packet to be returned by the receiving InfiniBand card. If it does not receive one, it sends the packet again. This variable controls the wait period. The time spent is equal to 4.096×2^n, where n is specified by the MPI_IB_TIMEOUT variable. By default, the variable is set to 18, and the time spent is expressed in microseconds.

MPI_NUM_MEMORY_REGIONS

For zero-copy sends over the InfiniBand interconnect, MPT keeps a cache of application data buffers registered for these transfers. This environment variable controls the size of the cache. If the application rarely reuses data buffers, it may make sense to set this value to 0 to

	avoid cache trashing. By default, this variable is set to 1024 (1K). The possible range is from 0 to 8192 (8K).
`MPI_NUM_QUICKS`	Controls the number of other ranks that a rank can receive from over InfiniBand using a short message fast path. This is 8 by default and can be any value between 0 and 32.

Tuning for Running Applications Over the Intel Omni-Path Interconnect

When you run an MPI application across a cluster of hosts using the Intel Omni-Path interconnect, there are run-time environment variables that you can set to to improve application performance. The following are these variables:

Variable	**Purpose**
`MPI_OPA_PAYLOAD`	When the MPI library uses Intel Omni-Path fabric, it allocates memory for each message header that it uses for transfer. If the size of the data to be sent is not greater than this amount minus 64 bytes for the actual header, the data is inlined with the header. If the size is greater than this value, the message is sent through remote direct memory access (RDMA) operations. The default is 16512 bytes.
`MPI_OPA_SINGLE_COPY_BUFFER_MAX`	If MPI transfers data over Intel Omni-Path fabric and if the size of the cumulative data is greater than this value, then MPI attempts to send the data directly between the processes's buffers and not through intermediate buffers inside the MPI library. The default is 32767.
`MPI_WILDCARDS`	If MPT can assume that an application does not make receive calls with rank wild cards, then MPT can perform some automatic optimizations. For more information, see the following:

"About Wild Cards" on page 36

MPI on SGI UV Systems

The SGI® UV™ series systems are scalable, nonuniform memory access (NUMA) systems that support a single Linux image of thousands of processors distributed over many sockets and many SGI UV hub application-specific integrated circuits (ASICs). The SGI UV hub is the heart of the SGI UV system compute blade. Each processor is a hyperthread on a particular core within a particular socket. Typically, each SGI UV hub connects to two sockets. All communication between the sockets and the SGI UV hub uses Intel QuickPath Interconnect (QPI) channels. The following information pertains to specific SGI UV systems:

- On SGI UV 3000 systems and SGI UV 300 systems, the SGI UV hub board assembly has an SGI UV hub ASIC with two identical hubs. Each hub supports one 9.6 GT/s QPI channel to a processor socket. On SGI UV 3000 systems and the SGI UV 300 systems, the hub has eight NUMAlink 7 ports that connect with the NUMAlink 7 interconnect fabric.
- On SGI UV 2000 systems, the SGI UV hub board assembly has an SGI UV hub ASIC with two identical hubs. Each hub supports one 8.0 GT/s QPI channel to a processor socket. The SGI UV 2000 series hub has eight NUMAlink 6 ports that connect with the NUMAlink 6 interconnect fabric.
- The SGI UV 1000 system's hub has four NUMAlink 5 ports that connect with the NUMAlink 5 interconnect fabric.

The SGI UV hub acts as a crossbar between the processors, local SDRAM memory, and the network interface. The hub ASIC enables any processor in the single-system image (SSI) to access the memory of all processors in the SSI.

When MPI communicates between processes on an SGI UV system, it uses the shared memory transfer method.

For more information about the SGI UV hub, SGI UV compute blades, QPI, and NUMAlink 5, or NUMAlink 6, see your hardware documentation.

The following topics contain more information about using MPI on SGI UV systems:

- "General Considerations" on page 63
- "Performance Problems and Corrective Actions" on page 63

- "Other ccNUMA Performance Considerations" on page 64

General Considerations

To run an MPI job optimally on an SGI UV system, it is best to pin MPI processes to CPUs and isolate multiple MPI jobs onto different sets of sockets and hubs. To accomplish this, you can configure a workload manager to create a cpuset for every MPI job. MPI pins its processes to the sequential list of logical processors within the containing cpuset by default, but you can control and alter the pinning pattern using the following:

- `MPI_DSM_CPULIST`. For more information, see "`MPI_DSM_CPULIST`" on page 53.
- `omplace`(1)
- `dplace`(1)

Performance Problems and Corrective Actions

The MPI library chooses buffer sizes and communication algorithms in an attempt to deliver the best performance to a wide variety of MPI applications automatically. The following list of performance problems can be remedied:

- Odd HyperThreads are idle.

  Most high performance computing MPI programs run best using only one HyperThread per core. When an SGI UV system has multiple HyperThreads per core, logical CPUs are numbered such that odd HyperThreads are the high half of the logical CPU numbers. Therefore, the task of scheduling only on the even HyperThreads can be accomplished by scheduling MPI jobs as if only half the full number exist, leaving the high logical CPUs idle. You can use the `cpumap`(1) command to determine if cores have multiple HyperThreads on your SGI UV system. The output shows the following:

  – The number of physical and logical processors.

  – Whether HyperThreading is on or off.

  – The way in which shared processors are paired. This information appears towards the bottom of the command’s output.

- MPI large message bandwidth is inappropriate.

Some programs transfer large messages via the `MPI_Send` function. To use unbuffered, single-copy transport in these cases, set `MPI_BUFFER_MAX=0`. For more information, see `MPI`(1).

- MPI small or near messages are very frequent.

  For small fabric hop counts, use shared memory message delivery. To deliver all messages within an SGI UV host via shared memory, set `MPI_SHARED_NEIGHBORHOOD=HOST`. For more information, see `MPI`(1).

Other ccNUMA Performance Considerations

MPI application processes typically perform better if their local memory is allocated on the socket assigned to execute the process. This cannot happen if memory on that socket is exhausted, either by the application itself or by other system consumption (for example, by file buffer cache).

You can use the `nodeinfo`(1) command to view memory consumption on the nodes assigned to your job, and you can use the `bcfree`(1)`bcfree` command to clear out excessive file buffer cache. PBS Professional workload manager installations can be configured to issue `bcfree`(1) commands in the job prologue.

For more information, see the PBS Professional documentation and `bcfree`(1).

Measuring Parallelization and Parallelizing Your Code

When tuning for performance, first assess the amount of code that is parallelized in your program. Use the following formula to calculate the amount of code that is parallelized:

```
p=N(T(1)-T(N)) / T(1)(N-1)
```

In this equation, T(1) is the time the code runs on a single CPU and T(N) is the time it runs on N CPUs. Speedup is defined as T(1)/T(N).

If *speedup*/`N` is less than 50% (that is, N>(2-*p*)/(1-*p*)), stop using more CPUs and tune for better scalability.

You can use one of the following to display CPU activity:

- The `top`(1) command.

- The `vmstat`(8) command.
- The open source Performance Co-Pilot tools. For example, `pmval`(1) (`pmval kernel.percpu.cpu.user`) or the visualization command `pmchart`(1).

Next, focus on using one of the following parallelization methodologies:

- "Using SGI MPI" on page 65
- "Using OpenMP" on page 65
- "Identifying OpenMP Nested Parallelism" on page 66
- "Using Compiler Options" on page 66
- "Identifying Opportunities for Loop Parallelism in Existing Code" on page 67

Using SGI MPI

The SGI Performance Suite includes the SGI Message Passing Toolkit (SGI MPT). SGI MPT includes both the SGI Message Passing Interface (SGI MPI) and SGI SHMEM. SGI MPI is optimized and more scalable for SGI UV series systems than the generic MPI libraries. SGI MPI takes advantage of the SGI UV architecture and SGI nonuniform memory access (NUMA) features.

Use the `-lmpi` compiler option to use MPI. For a list of environment variables that are supported, see the `mpi`(1) man page.

The `MPIO_DIRECT_READ` and `MPIO_DIRECT_WRITE` environment variables are supported under Linux for local XFS filesystems in SGI MPT version 1.6.1 and beyond.

MPI provides the MPI-2 standard MPI I/O functions that provide file read and write capabilities. A number of environment variables are available to tune MPI I/O performance. The `mpi_io`(3) man page describes these environment variables.

For information about performance tuning for MPI applications, see the following:

- *HPE Performance Software - Message Passing Interface MPInside Reference Guide*

Using OpenMP

OpenMP is a shared memory multiprocessing API, which standardizes existing practice. It is scalable for fine or coarse grain parallelism with an emphasis on performance. It exploits the strengths of shared memory and is directive-based. The

OpenMP implementation also contains library calls and environment variables. OpenMP is included with the C, C++, and Fortran compilers.

To use OpenMP directives, specify the `ifort -openmp` or `icc -openmp` compiler options. These options use the OpenMP front-end that is built into the Intel compilers. The latest Intel compiler OpenMP run-time library name is `libiomp5.so`. The latest Intel compiler also supports the GNU OpenMP library as an either/or option, in other words, do not mix-and-match the GNU library with the Intel version.

For more information, see the OpenMP standard at the following website:

http://www.openmp.org/wp/openmp-specifications

Identifying OpenMP Nested Parallelism

The following Open MP nested parallelism output shows two primary threads and four secondary threads, called master/nested:

```
% cat place_nested
firsttask cpu=0
thread name=a.out oncpu=0 cpu=4 noplace=1 exact onetime thread name=a.out oncpu=0
cpu=1-3 exact thread name=a.out oncpu=4 cpu=5-7 exact
```

```
% dplace -p place_nested a.out
Master thread 0 running on cpu 0
Master thread 1 running on cpu 4
Nested thread 0 of master 0 gets task 0 on cpu 0 Nested thread 1 of master 0 gets task 1 on cpu 1
Nested thread 2 of master 0 gets task 2 on cpu 2 Nested thread 3 of master 0 gets task 3 on cpu 3
Nested thread 0 of master 1 gets task 0 on cpu 4 Nested thread 1 of master 1 gets task 1 on cpu 5
Nested thread 2 of master 1 gets task 2 on cpu 6 Nested thread 3 of master 1 gets task 3 on cpu 7
```

For more information, see the `dplace`(1) man page.

Using Compiler Options

You can use compiler options to invoke automatic parallelization. Use the `-parallel` or `-par_report` options to the `ifort` or `icc` compiler commands. These options show which loops were parallelized and the reasons why some loops were not parallelized. If a source file contains many loops, it might be necessary to add the `-override_limits` flag to enable automatic parallelization. The code generated by the `-parallel` option is based on the OpenMP API. The standard OpenMP environment variables and Intel extensions apply.

There are some limitations to automatic parallelization:

- For Fortran codes, only `DO` loops are analyzed
- For C/C++ codes, only `for` loops using explicit array notation or those using pointer increment notation are analyzed. In addition, `for` loops using pointer arithmetic notation are not analyzed, nor does it analyze `while` or `do while` loops. The compiler also does not check for blocks of code that can be run in parallel.

Identifying Opportunities for Loop Parallelism in Existing Code

Another parallelization optimization technique is to identify loops that have a potential for parallelism, such as the following:

- Loops without data dependencies; a *data dependency conflict* occurs when a loop has results from one loop pass that are needed in future passes of the same loop.
- Loops with data dependencies because of temporary variables, reductions, nested loops, or function calls or subroutines.

Loops that do not have a potential for parallelism are those with premature exits, too few iterations, or those where the programming effort to avoid data dependencies is too great.

Suspending MPI Jobs

Internally, the HPE Performance MPI software uses the XPMEM kernel module tprovide single-copy operations to local data. The XPMEM kernel module prevents any pages used by these operations from paging. If an administrator needs to temporarily suspend an MPI application to allow other applications to run, they can unpin these pages so they can be swapped out and made available for other applications.

Each process of an MPI application that is using the XPMEM kernel module has a `/proc/xpmem/`*pid* file associated with it. File `/proc/xpmem/`*pid* includes the number of pages owned by this process that are prevented from paging by XPMEM. You can display the content of this file. For example:

```
# cat  /proc/xpmem/5562
pages pinned by XPMEM: 17
```

The following procedure explains how to unpin the pages for use by other processes.

Procedure 7-1 To unpin pages

1. Log in as the system administrator.
2. Suspend all the processes in the application.
3. Use the `echo`(1) command to unpin the pages.

   You can echoing any value into the `/proc/xpmem/`*pid* file.

   For *pid*, specify the process ID.

   The echo command does not return until that process's pages are unpinned.

   For example:

   ```
   # echo 1 > /proc/xpmem/5562
   ```

When the MPI application is resumed, the `XPMEM` kernel module prevents the pages from paging as they are referenced by the application.

Chapter 8

HPE Performance Software — Message Passing Interface (HPE Performance MPI) Performance Profiling

This chapter includes the following topics:

About HPE Performance MPI Performance Profiling

Performance profiling occurs when you run your MPI program or SHMEM program with a tool that can aggregate run time statistics. Profiling tools gather statistics that show the amount of time that your program spends in MPI, the number of messages sent, or the number of bytes sent. MPT includes profiling support in the `libmpi.so` library. When you use a profiling tool, the tool automatically replaces all `MPI_`*XXX* prototypes and function names with `PMPI_`*XXX* entry points.

This chapter describes the use of profiling tools to obtain performance information. Compared to the performance analysis of sequential applications, characterizing the performance of parallel applications can be challenging. Often it is most effective to first focus on improving the performance of MPI applications at the single process level.

It may also be important to understand the message traffic generated by an application. A number of tools can be used to analyze this aspect of a message passing application's performance, including HPE's MPInside and various third-party products.

The following topics contain more information about profiling:

- *HPE Performance Software - Message Passing Interface MPInside Reference Guide*. This manual explains how to use the MPInside profiling tool.

- "Using `perfcatch`(1)" on page 70
- "Writing Your Own Profiling Interface" on page 75
- "Using Third-party Profilers" on page 76
- "MPI Internal Statistics" on page 76

Using `perfcatch`(1)

You can use the `perfcatch` utility to profile the performance of an MPI program or SHMEM program. The `perfcatch` utility runs the MPI program with the wrapper library, `libmpi.so`, and writes MPI call profiling information information to `MPI_PROFILING_STATS`.

The following topics contain more information about `perfcatch`(1):

- "The `perfcatch`(1) Command" on page 70
- " `MPI_PROFILING_STATS` Results File Example" on page 71
- "Environment Variables Used With `perfcatch`(1)" on page 74

The `perfcatch`(1) Command

The following format shows how to use the `perfcatch` command:

```
mpiexec_mpt [ mpi_params ] perfcatch [ -i ] cmd [ args ]
```

By default, `perfcatch` assumes an MPT program. The `perfcatch` utility accepts the following options:

mpi_params	Optional. Specifies the MPI parameters needed to launch the program.
`-i`	Specifies to use Intel MPI.
cmd	Specifies the name of the executable program. For example, `a.out`.
args	Optional. Specifies additional command line arguments.

To use `perfcatch` with an HPE Performance MPI program, insert the `perfcatch` command in front of the executable file name, as the following examples show:

- **`mpiexec_mpt -np 64 perfcatch a.out arg1`**
- **`mpiexec_mpt host1 32, host2 64 perfcatch a.out arg1`**

To use `perfcatch` with Intel MPI, add the `-i` option, as follows:

```
mpiexec -np 64 perfcatch -i a.out arg1
```

For more information, see the `perfcatch`(1) man page.

`MPI_PROFILING_STATS` Results File Example

The `perfcatch`(1) utility's output file is called `MPI_PROFILING_STATS`. Upon program completion, the `MPI_PROFILING_STATS` file resides in the current working directory of the MPI process with rank 0.

This output file includes a summary statistics section followed by a rank-by-rank profiling information section. The summary statistics section reports some overall statistics. These statistics include the percent time each rank spent in MPI functions and the MPI process that spent the least and the most time in MPI functions. Similar reports are made about system time usage.

In the rank-by-rank profiling information, there is a list of every profiled MPI function called by a particular MPI process. The report includes the number of calls and the total time consumed by these calls. Some functions report additional information, such as average data counts and communication peer lists.

The following is an example `MPI_PROFILING_STATS` results file:

```
============================================================
PERFCATCHER version 22
(C) Copyright Hewlett Packard Enterprise Development LP.
This library may only be used on HPE hardware platforms.
See LICENSE file for details.
============================================================
MPI program profiling information
Job profile recorded Wed Jan 17 13:05:24 2007
Program command line:                                /home/estes01/michel/sastest/mpi_hello_linux
Total MPI processes                                  2

Total MPI job time, avg per rank                    0.0054768 sec
Profiled job time, avg per rank                     0.0054768 sec
Percent job time profiled, avg per rank             100%

Total user time, avg per rank                       0.001 sec
Percent user time, avg per rank                     18.2588%
Total system time, avg per rank                     0.0045 sec
Percent system time, avg per rank                   82.1648%

Time in all profiled MPI routines, avg per rank     5.75004e-07 sec
Percent time in profiled MPI routines, avg per rank 0.0104989%

Rank-by-Rank Summary Statistics
-------------------------------

Rank-by-Rank: Percent in Profiled MPI routines
        Rank:Percent
        0:0.0112245%    1:0.00968502%
  Least:  Rank 1      0.00968502%
  Most:   Rank 0      0.0112245%
  Load Imbalance:  0.000771%

Rank-by-Rank: User Time
        Rank:Percent
        0:17.2683%      1:19.3699%
  Least:  Rank 0      17.2683%
  Most:   Rank 1      19.3699%

Rank-by-Rank: System Time
        Rank:Percent
        0:86.3416%      1:77.4796%
  Least:  Rank 1      77.4796%
  Most:   Rank 0      86.3416%

Notes
-----

Wtime resolution is                   5e-08 sec

Rank-by-Rank MPI Profiling Results
----------------------------------

Activity on process rank 0

           Single-copy checking was not enabled.
comm_rank            calls:      1  time: 6.50005e-07 s  6.50005e-07 s/call

Activity on process rank 1

           Single-copy checking was not enabled.
comm_rank            calls:      1  time: 5.00004e-07 s  5.00004e-07 s/call

------------------------------------------------

recv profile

             cnt/sec for all remote ranks
local   ANY_SOURCE        0            1
 rank

------------------------------------------------

recv wait for data profile

             cnt/sec for all remote ranks
local        0            1
 rank
------------------------------------------------

recv wait for data profile
```

```
             cnt/sec for all remote ranks
local        0            1
 rank

------------------------------------------------

send profile

             cnt/sec for all destination ranks
  src        0            1
 rank

------------------------------------------------

ssend profile

             cnt/sec for all destination ranks
  src        0            1
 rank

------------------------------------------------

ibsend profile

             cnt/sec for all destination ranks
  src        0            1
 rank
```

Environment Variables Used With **`perfcatch`**(1)

The MPI performance-profiling environment variables are as follows:

Variable	**Description**
`MPI_PROFILE_AT_INIT`	Activates MPI profiling immediately, that is, at the start of MPI program execution. To use this environment variable, set it to any value. For example, set `MPI_PROFILE_AT_INIT` to 1.

`MPI_PROFILING_STATS_FILE`	Specifies the `perfcatch` output file. This is the file to which MPI profiling results are written. By default, the profiler writes to `MPI_PROFILING_STATS`.

Writing Your Own Profiling Interface

You can write your own profiler by using the MPI standard `PMPI_*` calls. In addition, either within your own profiling library or within the application itself, you can use the `MPI_Wtime` function call to time specific calls or sections of your code.

The following example output is for a single rank of a program that was run on 128 processors using a user-created profiling library that performs call counts and timings of common MPI calls. Notice that for this rank, most of the MPI time is spent in `MPI_Waitall` and `MPI_Allreduce`.

```
Total job time 2.203333e+02 sec
Total MPI processes 128
Wtime resolution is 8.000000e-07 sec

activity on process rank 0
comm_rank calls 1      time 8.800002e-06
get_count calls 0      time 0.000000e+00
ibsend calls    0      time 0.000000e+00
probe calls     0      time 0.000000e+00
recv calls      0      time 0.00000e+00   avg datacnt 0   waits 0       wait time 0.00000e+00
irecv calls     22039  time 9.76185e-01   datacnt 23474032 avg datacnt 1065
send calls      0      time 0.000000e+00
ssend calls     0      time 0.000000e+00
isend calls     22039  time 2.950286e+00
wait calls      0      time 0.00000e+00   avg datacnt 0
waitall calls   11045  time 7.73805e+01   # of Reqs 44078  avg data  cnt 137944
barrier calls   680    time 5.133110e+00
alltoall calls  0      time 0.0e+00       avg datacnt 0
alltoallv calls 0      time 0.000000e+00
reduce calls    0      time 0.000000e+00
allreduce calls 4658   time 2.072872e+01
bcast calls     680    time 6.915840e-02
gather calls    0      time 0.000000e+00
```

```
gatherv calls   0      time 0.000000e+00
scatter calls   0      time 0.000000e+00
scatterv calls  0      time 0.000000e+00

activity on process rank 1
...
```

Using Third-party Profilers

You can use third-party profiling tools with HPE Performance MPI. The following are examples of tools to consider:

- The TAU Performance System profiler from the University of Oregon. This software is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, and Python.
- The Allinea MAP profiler. The Allinea MAP profiler is part of the Allinea Forge toolkit

MPI Internal Statistics

MPI keeps track of certain resource utilization statistics. You can use these statistics to determine potential performance problems caused by a lack of MPI message buffers or other MPI internal resources.

To display MPI internal statistics, use the `MPI_STATS` environment variable or the `-stats` option on the `mpirun` command. MPI internal statistics are always being gathered, so displaying them does not cause significant additional overhead. In addition, one can sample the MPI statistics counters from within an application, allowing for finely grained measurements.

If the `MPI_STATS_FILE` environment variable is set, when the program completes, the system writes internal statistics to the file specified by this variable.

These statistics can be very useful in optimizing codes in the following ways:

- To determine if there are enough internal buffers and if processes are waiting (retries) to acquire them
- To determine if single copy optimization is being used for point-to-point or collective calls

For additional information on how to use the MPI statistics counters to help tune the run-time environment for an MPI application, see Chapter 7, "Run-time Tuning" on page 49.

Chapter 9

Troubleshooting and Frequently Asked Questions

This chapter includes the following topics:

About Troubleshooting

This chapter provides answers to the following questions that users might ask when they start to use HPE Performance Software — Message Passing Interface (HPE Performance MPI):

Why Is the `mpiexec_mpt`(1) Command Failing?

If the `mpiexec_mpt`(1) command fails, investigate the following:

- Look in `/var/log/messages` for any suspicious errors or warnings. For example, if your application tries to load a library that it cannot find, a message should appear here. Only the root user can view this file.
- Be sure that you did not misspell the name of your application.
- To find dynamic link errors, try the following:

  - Run your program without `mpiexec_mpt`(1). When you do not use `mpiexec_mpt`, the output includes dynamic link errors that might not otherwise be displayed. In addition, the output includes the following message:

    ```
    mpiexec_mpt must be used to launch all MPI applications
    ```

  - Run your program with `mpiexec_mpt`(1). Set the `LD_DEBUG` environment variable to `all`, which generates output that includes a set of messages for each symbol that `rld` resolves. This variable produces a lot of output, but it can help you find the cause of the link error.

- Verify that you set your remote directory properly. By default, `mpiexec_mpt`(1) attempts to place your processes on all machines into the directory that has the same name as `$PWD`. This should be the common case, but sometimes different functionality is required. For more information, see the `mpiexec_mpt`(1) man page's sections on `$MPI_DIR` and/or the `-dir` option.

- If you use a relative pathname for your application, verify that it appears in the `$PATH` environment variable. In particular, the `mpiexec_mpt`(1) command does not look in the working directory (`"."`) for your application unless `"."` appears in `$PATH`.

- Run the following command to verify that your array is configured correctly:

  ```
  /usr/sbin/ascheck
  ```

- Run the `mpiexec_mpt -verbose` command to verify that you are running the version of MPI that you think you are running.

- Be very careful when you set MPI environment variables from within your `.cshrc` or `.login` files because these files override any settings that you might later set from within your shell. The reason for this is that MPI creates the equivalent of a fresh login session for every job. The safe way to set things up is to test for the existence of `$MPI_ENVIRONMENT` in your scripts and set the other MPI environment variables only if it is undefined.

- If you are running in a Kerberos environment, you can experience unpredictable results because `mpiexec_mpt`(1) cannot pass tokens. For example, in some cases, if you use `telnet`(1) to connect to a host and then try to run `mpiexec_mpt` on that host, it fails. However, if you instead use `rsh`(1) to connect to the host, `mpiexec_mpt` succeeds. This might be because `telnet` is kerberized, but `rsh` is not. If you are running under such conditions, talk to your local administrator about the proper way to launch MPI jobs.

- Look in the following directory on all the machines you are using:

  ```
  /tmp/.arraysvcs
  ```

  In some cases, you might find a helpful `errlog` file.

- You can increase the verbosity of the Array Services daemon, `arrayd`, when you use the `-v` option to generate more debugging information. For more information, see the `arrayd`(8) man page.

- Check for error messages in the `/var/run/arraysvcs` directory.

Why Does My Code Run Correctly Until It Reaches `MPI_Finalize()` and Then Hang?

A code hang is almost always caused by a `send` request or a `recv` request that is either unmatched or not completed. These request types are as follows:

- An *unmatched request* is any blocking `send` request for which a corresponding `recv` request is never posted.

- An *incomplete request* is any nonblocking `send` or `recv` request that was never freed by a call to `MPI_Test()`, `MPI_Wait()`, or `MPI_Request_free()`.

Common examples are applications that call `MPI_Isend()` and then use internal means to determine when it is safe to reuse the send buffer. These applications never call `MPI_Wait()`. To fix such codes, update your code in one of the following ways:

- Insert a call to `MPI_Request_free()` immediately after all such `isend` operations.

- Add a call to `MPI_Wait()` at a later place in the code, prior to the point at which the send buffer must be reused.

- Set `MPI_REQUEST_DEBUG=true`, which causes MPT to check for this condition at `MPI_Finalize()` time.

Why Does My Hybrid Code (Using OpenMP) Stall on the `mpirun` Command?

If your application was compiled with the Open64 compiler, make sure you follow the instructions about using the Open64 compiler in combination with MPI/OpenMP applications described in the following topic:

"Compiling and Linking the MPI Program" on page 13

Why Do I Keep Receiving Warning Messages About the `MPI_REQUEST_MAX` Value Being Too Small?

The MPI library generates the following warning message when the `MPI_REQUEST_MAX` value is not set appropriately:

```
MPT Warning:  MPT has run out of preallocated request entries.
This may slow performance or fragment memory.
Please increase MPI_REQUEST_MAX.
```

The following are the conditions under which the MPI library generates the preceding message:

- The program uses a very large number of simultaneous transfer requests, much larger than the number of requests for which MPT preallocates. To fix this, use the `MPI_REQUEST_MAX` shell variable to increase the number of preallocated requests.
- The application calls `MPI_Isend()` or `MPI_Irecv()` and does not complete or free the requested objects. To fix this, use `MPI_Request_free()`, as described in the following:

  "Why Does My Code Run Correctly Until It Reaches `MPI_Finalize()` and Then Hang?" on page 82

Why Is It That I Do I Not See Any `stdout` And/Or `stderr` Output From My MPI Application?

All `stdout` output and `stderr` output is line-buffered, which means that the `mpirun`(1) command does not print any partial lines of output. This sometimes causes problems for codes that prompt the user for input parameters but do not end their prompts with a newline character. The only solution for this is to append a newline character to each prompt.

You can set the `MPI_UNBUFFERED_STDIO` environment variable to disable line-buffering. For more information, see the `MPI`(1) and `mpirun`(1) man pages.

How Do I Install the MPT Software On My Machine?

The MPT RPMs are included in the HPE Performance Software releases. In addition, you can obtain MPT RPMs from the customer portal at the following URL:

https://support.sgi.com

Where Can I Find More Information About the OpenSHMEM Programming Model?

See the `intro_shmem`(3) man page.

Why Does the `ps`(1) command Say That My Memory Use (`SIZE`) Is Higher Than Expected?

At job start-up, the MPI and OpenSHMEM libraries cross-map all user static and heap memory of the processes on the local host to provide optimization opportunities. The result is large virtual memory usage. The `ps`(1) command's `SIZE` statistic is telling you the amount of virtual address space used, not the amount of memory consumed. Even if all of the pages that you could reference were faulted in, most of the virtual address regions point to multiply-mapped (shared) data regions, and even in that case, actual per-process memory usage would be far lower than that indicated by `SIZE`.

What Does The `MPI: could not run executable` Message Mean?

This message means that something happened while `mpiexec_mpt`(1) was trying to launch your application, which caused it to fail before all of the MPI processes were able to handshake with it.

The mpiexec_mpt command directs arrayd to launch a shepherd process on each host and listens on a socket for those shepherds to connect back to it. Because the shepherds are children of arrayd, arrayd traps SIGCHLD and passes that signal back to mpiexec_mpt whenever one of the shepherds terminates. If mpiexec_mpt receives a signal before it establishes connections with every host in the job, it knows that something has gone wrong.

How Do I Combine MPI With *insert favorite tool here*?

Different MPI implementations use different methods of launching their worker processes. Some tools expect a method that is different from MPT's default. If you set the shell variable MPI_SHEPHERD=true, then MPT attempts to use a launch method that is similar to some other MPI implementations. Use of this option can disable some less-common features, such as spawning and checkpoint-restart support.

In general, the rule to follow is to run the mpiexec_mpt command on your tool and then run the tool on your application. Do not try to run the tool on mpiexec_mpt.

Also, because of the way that mpiexec_mpt sets up stdio, viewing the output from your tool might require a bit of effort. The most ideal case is when the tool directly supports an option to redirect its output to a file. In general, this is the recommended way to mix tools with mpiexec_mpt. Many tools, for example, dplace(1), do not support such an option. However, you might be able to wrap a shell script around the tool and have the script do the redirection, as in the following example:

```
> cat myscript
#!/bin/sh
#################################################################
# NOTE: The example shown is for illustrative purposes only and #
# has not been evaluated for use in a production environment.   #
#################################################################
setenv MPI_DSM_OFF
dplace -verbose a.out 2> outfile
> mpirun -np 4 myscript
hello world from process 0
hello world from process 1
hello world from process 2
hello world from process 3
> cat outfile
there are now 1 threads
Setting up policies and initial thread.
```

```
Migration is off.
Data placement policy is PlacementDefault.
Creating data PM.
Data pagesize is 16k.
Setting data PM.
Creating stack PM.
Stack pagesize is 16k.
Stack placement policy is PlacementDefault.
Setting stack PM.
there are now 2 threads
there are now 3 threads
there are now 4 threads
there are now 5 threads
```

Caution: The preceding script example is for illustrative purposes only and has not been evaluated for use in a production environment.

Why Do I See “stack traceback” Information When My MPI Job Aborts?

For information, see the `MPI_COREDUMP` environment variable description and the `MPI_COREDUMP_DEBUGGER` environment variable description on the `MPI`(1) man page.

Chapter 10

Array Services

This chapter includes the following topics:

- "About Array Services" on page 87
- "Installing and Configuring Array Services" on page 89
- "Array Services Commands and Arguments" on page 89
- "Array Services Environment Variables" on page 91
- "Obtaining Information About the Array" on page 92

About Array Services

The HPE Array Services software enables parallel applications to run on multiple hosts in a cluster, or *array*. Array Services provides cluster job launch capabilities for Message Passing Toolkit (MPT) jobs.

The array can consist of the following:

- Multiple server nodes on a cluster computing system
- Multiple physical machines

An array system is bound together with a high-speed network and the Array Services software. Array users can access the system with familiar commands for job control, authentication, and remote execution. Array Services facilitates global session management, array configuration management, batch processing, message passing, system administration, and performance visualization.

The Array Services software package includes the following:

- An array daemon that runs on each node. The daemon groups logically related processes together across multiple nodes. The process groups create a global process namespace across the array, facilitate accounting, and facilitate administration.

  The daemon maintains information about node configuration, process IDs, and process groups. Array daemons on the nodes cooperate with each other.

- Array configuration files. The files describes the array configuration and provides reference information for array daemons and user programs. Each node hosts a copy of each array configuration file.
- Commands, libraries, and utilities such as `ainfo`(1), `arshell`(1), and others.

The Message Passing Interface (MPI) of the HPE Performance Software — Message Passing Interface (HPE Performance MPI) software uses Array Services to launch parallel applications.

The HPE Performance MPI software distribution includes the MUNGE software. This optional, open-source product provides secure Array Services functionality. MUNGE allows a process to authenticate the UID and GID of another local or remote process within a group of hosts that have common users and groups. MUNGE authentication, which also includes the Array Services data exchanged in the array, is encrypted. For more information about MUNGE, see the MUNGE website at the following location:

http://dun.github.io/munge/

The Array Services package requires that the process sets service be installed and running. This package is provided in the `sgi-procset` RPM. Use one of the following command sets to verify that the process sets service is installed and running:

- On RHEL 7.*X* or SLES 12 SP*X* systems, type the following commands:

```
# rpm -q sgi-procset
# systemctl status procset
```

- On RHEL 6.*X* or SLES 11 SP*X* systems, type the following commands:

```
# rpm -q sgi-procset
# /etc/init.d/procset status
```

The following topics contain end-user information about Array Services:

- "Installing and Configuring Array Services" on page 89
- "Array Services Commands and Arguments" on page 89
- "Array Services Environment Variables" on page 91
- "Obtaining Information About the Array" on page 92

Note: For Array Services information that pertains to system administration, see the following:

Appendix B, "Array Services System Administration Information"

Installing and Configuring Array Services

The system administrator needs to install and configure the Array Services software before end users can use Array Services or run HPE Performance Software — Message Passing Interface (HPE Performance MPI) programs.

See one of the following for automated installation information:

- On an HPE SGI 8600 system, an HPE Apollo 40 system, an HPE Apollo 20 system, an SGI ICE system, or an SGI Rackable cluster system, see the following manual:

  HPE SGI Management Suite Installation and Configuration Guide

- On an HPE Apollo 6500 system, HPE Apollo 6000 system, or an HPE Apollo 2000 system, see the following:

  "Installing the Array Services Software on HPE Apollo Cluster Systems" on page 2

For information about the manual installation procedure, see the following:

Appendix B, "Array Services System Administration Information" on page 127

Array Services Commands and Arguments

When an application starts multiple processes on multiple nodes, a Linux process identifier (PID) and a process group identifier (PGID) are no longer adequate to manage the application. The Array Services commands enable you to view the entire array and to control processes on multiple nodes. You can type Array Services commands from any workstation connected to an array system. You do not have to be logged in to an array node.

To retrieve overview information about Array Services online, see the following `man`(1) page:

`array_services`(5)

The following topics contain more information about Array Services commands:

- "Array Services Commands" on page 90
- "Additional Information for the `ainfo`(1) Command and `array`(1) Command" on page 91

Array Services Commands

The following are the Array Services commands:

- `ainfo`(1)

  Retrieves information about the different arrays at your site and about the nodes included in each array. At most sites, there is only one array, but you can have multiple arrays at your site. The command output includes the hostnames for each node in each array at your site.

- `array`(1)

  Runs a system command on one or more nodes and returns output to `stdout`. As arguments, `array`(1) accepts several options and the one system command that you want to run on the array. There is a default set of system commands, but your system administrator determines the list of commands available to you at your site.

- `arshell`(1)

  Runs a system command remotely on a different node. As arguments, `arshell`(1) accepts several options and the one system command that you want to run on the remote node.

  The `arshell` command is like `rsh` in that it runs a command on another machine under the userid of the invoking user. Use of authentication codes makes Array Services somewhat more secure than `rsh`.

The `ainfo`(1), `array`(1), and `arshell`(1) online `man`(1) pages explain the arguments that each command accepts. For specific information about each command's arguments, see the individual command `man`(1) pages.

Additional Information for the `ainfo`(1) Command and `array`(1) Command

The `ainfo`(1) command and `array`(1) command support several common command line arguments. For comprehensive information about these commands, see the `ainfo`(1) and `array`(1) online `man`(1) pages.

The following information supplements the information on the `man`(1) pages:

- Your array administrator might have established an authentication code, which is a 64-bit number, for all or some of the nodes in an array.

  The `ainfo`(1) and `array`(1) commands accept the `-Kl` *number* and `-Kr` *number* options. For *number*, specify the 64–bit authentication key number for the local node or the remote node that you want to target with the command. The code applies to any command entered at that node or addressed to that node.

  Your system administrator can tell you if it is necessary to specify an authentication code.

  Note that in the case of `-Kl` *number*, the option letter is a lowercase letter “L”, for “local”.

- The `-l` and `-s` options work together. The `-l` option restricts the scope of a command to the node upon which the command is run. This option is a lowercase letter “L”, for “local”. By default, that is the node where the command is entered. When `-l` is not used, the command queries all nodes of the array. The `-s` option runs the command on a specified node of the array. These options work together as follows:

  – To query all nodes as seen by the local node, use neither option.

  – To query only the local node, use only `-l`.

  – To query all nodes as seen by a specified node, use only `-s`.

  – To query only a particular node, use both `-s` and `-l`.

Array Services Environment Variables

The Array Services commands depend on environment variables to define default values for the less-common command options. Table 10-1 summarizes these variables.

Table 10-1 Array Services Environment Variables

Variable Name	Use	Default When Undefined
`ARRAYD_FORWARD`	When defined with a string starting with the letter *y*, all commands default to forwarding through the array daemon (option `-F`).	Commands default to direct communication (option `-D`).
`ARRAYD_PORT`	The port (socket) number monitored by the array daemon on the destination node.	The standard number of 5434, or the number given with option `-p`.
`ARRAYD_LOCALKEY`	Authentication key for the local node (option `-Kl`).	No authentication unless the `-Kl` option is used.
`ARRAYD_REMOTEKEY`	Authentication key for the destination node (option `-Kr`).	No authentication unless `-Kr` option is used.
`ARRAYD`	The destination node, when not specified by the `-s` option.	The local node, or the node given with `-s`.

Obtaining Information About the Array

You can use Array Services commands and system commands to retrieve information about the array. For example, you can use the `ainfo`(1) command and the `array`(1) command to check the hardware components and the software workload on the array. In addition, you can use system commands, such as `who`(1), `top`(1), and `uptime`(1), to retrieve information about users and workload on a node. To obtain array-wide information, use these commands with the `array`(1) command.

The following topics include examples that show how to retrieve information about the array:

- "Retrieving Array Names" on page 93
- "Retrieving Node Names" on page 93
- "Retrieving User Names" on page 93
- "Retrieving Workload Information" on page 94

Retrieving Array Names

The following command shows how to retrieve the names of all arrays configured at your site:

```
homegrown% ainfo arrays
Arrays known to array services daemon
ARRAY DevArray
    IDENT 0x3381
ARRAY BigDevArray
    IDENT 0x7456
ARRAY test
    IDENT 0x655e
```

Your system adminstrator configures the arrays at your site. Different arrays might know different sets of other array names.

Retrieving Node Names

The following command uses the `-b` option of `ainfo`(1) command to retrieve a brief version of the information about all the nodes in the current array:

```
homegrown 175% ainfo -b machines
machine homegrown homegrown 5434 192.48.165.36 0
machine disarray disarray 5434 192.48.165.62 0
machine datarray datarray 5434 192.48.165.64 0
machine tokyo tokyo 5434 150.166.39.39 0
```

Retrieving User Names

Example 1. The following `array who` command retrieves the names of all users logged in to the array:

```
mynode% array who
frederik corfu        rummage.eng.sgi  -tcsh
stefaan  sf           yoga.eng.sgi  -tcsh
timo     tokyo        frost.ued.sgi   /bin/tcsh
wim      boston       sig.eng.sgi   vi +153 fs/procfs/prd
ruben    paris        mountain.eng.sgi   -tcsh
...
```

Example 2. The following command retrieves the names of users logged in to the node named `tokyo`. This command uses the `-l` and `-s` options.

```
homegrown 180% array -s tokyo -l who
joecd    tokyo        frummage.eng.sgi -tcsh
joecd    tokyo        frummage.eng.sgi -tcsh
benf     tokyo        einstein.ued.sgi. /bin/tcsh
yohn     tokyo        rayleigh.eng.sg vi +153 fs/procfs/prd
...
```

The preceding examples have been edited for brevity and security.

Retrieving Workload Information

Example 1. The following command shows how to use the `uptime`(1) command to retrieve information for the entire array:

```
homegrown 181% array uptime
   homegrown:  up 1 day,  7:40,  26 users,  load average: 7.21, 6.35, 4.72
    disarray:  up  2:53,  0 user,  load average: 0.00, 0.00, 0.00
    datarray:  up  5:34,  1 user,  load average: 0.00, 0.00, 0.00
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.15, 0.31, 0.29
```

Example 2. The following command shows how to use the `uptime`(1) command to retrieve information about a single node:

```
homegrown 182% array -l -s tokyo uptime
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.12, 0.30, 0.28
```

Example 3. The following command shows how to use the `top`(1) command to lists the processes that are currently using the most CPU time:

```
homegrown 183% array top
        ASH        Host           PID User           %CPU Command
----------------------------------------------------------------
0x1111ffff00000000 homegrown        5 root           1.20 vfs_sync
0x1111ffff000001e9 homegrown     1327 arraysvcs      1.19 atop
0x1111ffff000001e9 tokyo        19816 arraysvcs      0.73 atop
0x1111ffff000001e9 disarray      1106 arraysvcs      0.47 atop
0x1111ffff000001e9 datarray      1423 arraysvcs      0.42 atop
0x1111ffff00000000 homegrown       20 root           0.41 ShareII
0x1111ffff000000c0 homegrown    29683 kchang         0.37 ld
```

```
0x1111ffff0000001e homegrown     1324 root           0.17 arrayd
0x1111ffff00000000 homegrown      229 root           0.14 routed
0x1111ffff00000000 homegrown       19 root           0.09 pdflush
0x1111ffff000001e9 disarray      1105 arraysvcs      0.02 atopm
```

The output identifies each process by its internal array session handle (ASH) value. As an alternative, you could use the `-l` and `-s` options to select data about a single node.

Chapter 11

Using the Message Passing Toolkit (MPT) Plugin for Nagios

This chapter includes the following topics:

- "About the MPT Plugin for Nagios" on page 97
- "Installing the MPT Nagios Plugin on the Admin Node" on page 98
- "(Optional) Installing the MPT Nagios Plugin on an HPE SGI 8600 Rack Leader Controller or SGI ICE Rack Leader Controller" on page 101
- "Viewing MPT Messages From Within Nagios and Clearing the Messages" on page 102
- "(Optional) Modifying the Notification Email" on page 104

About the MPT Plugin for Nagios

Nagios is a web-based system monitoring tool that HPE automatically installs on cluster computing systems. Nagios enables you to monitor the cluster infrastructure. When you install the optional MPT plugin for Nagios, the MPT system log messages that typically appear in `/var/log/messages` also appear in the Nagios graphical user interface (GUI). The plugin scans the system log for messages that MPT has logged, and in the Nagios GUI, the plugin displays the number of error messages and warning messages that the plugin encountered in the scan.

The following topics provide more information about the MPT plugin for Nagios:

- "Installing the MPT Nagios Plugin on the Admin Node" on page 98
- "(Optional) Installing the MPT Nagios Plugin on an HPE SGI 8600 Rack Leader Controller or SGI ICE Rack Leader Controller" on page 101
- "Viewing MPT Messages From Within Nagios and Clearing the Messages" on page 102
- "(Optional) Modifying the Notification Email" on page 104

Installing the MPT Nagios Plugin on the Admin Node

The following procedure explains how to install the MPT Nagios plugin on the admin node.

Procedure 11-1 To install the MPT Nagios plugin on the admin node

1. Locate the HPE Performance Software installation DVD, and insert the DVD into the DVD reader on the admin node.

2. Log into the admin node as the root user.

3. Change to the RPM repository directory.

4. Type one of the following commands to install the plugin:

   - On RHEL 7 systems or RHEL 6 systems, type the following command:

     ```
     # yum install checkmpt-plugin
     ```

   - On SLES 12 systems or SLES 11 systems, type the following commands:

     ```
     # zypper in checkmpt-plugin
     ```

   The preceding commands install the following files:

   ```
   /opt/hpe/hpc/mpt/checkmpt-plugin/README
   /opt/sgi/nagios/libexec/check_mpt
   ```

5. Use a text editor to open file `/opt/hpe/hpc/mpt/checkmpt-plugin/README`, and leave the file open in a window on your desktop.

   This file contains a shorthand version of these installation instructions. Some steps in this installation procedure require you to insert specific lines into specific files, and it is easiest to copy the lines out of the README file and modify them as this procedure explains.

6. Type the following command to edit file `sudoers`:

   ```
   # visudo
   ```

7. Copy the following lines from the `README` file to the end of the `sudoers` file, and replace `<nagiosuser>` and `<PLUGINSDIR>` with values that are valid at your site:

   ```
   # check_mpt plugin for Nagios (needs access to syslogs)
   <nagiosuser> ALL=NOPASSWD: <PLUGINSDIR>/check_mpt
   ```

```
# end check_mpt
```

Replace the variables in the preceding lines as follows:

- Replace `<nagiosuser>` with the Nagios username assigned when Nagios was installed. By default, this username is `nagios`.
- Replace `<PLUGINSDIR>` with the directory in which the Nagios plugin resides. By default, this is `/opt/sgi/nagios/libexec`.

8. Save and close the `sudoers` file.
9. Use a text editor to open file `commands.cfg`.

   By default, this file resides in the following directory:

   ```
   /opt/sgi/nagios/etc/objects
   ```

10. Copy the following lines from the `README` file to the end of the `commands.cfg` file:

    ```
    # check_mpt command definition
        define command {
                command_name check_mpt
                command_line sudo $USER1$/check_mpt -W $ARG1$ -E $ARG2$
        }
        # end check_mpt
    ```

    You do not need to assign values to `$ARG1$` or `$ARG2$`. A later step in this procedure populates these arguments with values.

11. Save and close the `commands.cfg` file.
12. Use a text editor to open file `localhost.cfg`.

    By default, this file resides in the following directory:

    ```
    /opt/sgi/nagios/etc/objects
    ```

13. Copy the following lines from the `README` file to the end of the `localhost.cfg` file:

    ```
    # check_mpt service definition
    define service {
            use                     local-service
            host_name               localhost
    ```

```
        service_description   check_mpt
        check_command         check_mpt!10!5
        max_check_attempts    2
        normal_check_interval 2
        retry_check_interval  1
}
# end of check_mpt
```

The key lines in the preceding module have the following effects:

Line	Comment
`use local-service`	Use the generic Nagios template.
`host_name localhost`	Run on localhost or similar.
`service_description check_mpt`	Declare the service name.
`check_command check_mpt!10!5`	Is CRITICAL if >10 warnings / >5 errors.
`max_check_attempts 2`	If OK, try check again.
`normal_check_interval 2`	Run check every 2 minutes.
`retry_check_interval 1`	Retry every 1 minute.

14. Save and close file `localhost.cfg`.

15. Type the following command to verify the changes you made and to make sure that there are no conflicts:

    nagios_dir`/bin/nagios -v` *nagios_dir*`/etc/nagios.cfg`

    For *nagios_dir*, specify the Nagios home directory. By default, this directory is `/opt/sgi/nagios`.

16. Restart Nagios on the node.

    This command differs, depending on your platform, as follows:

    - To restart Nagios on RHEL 7 and SLES 12 platforms, type the following command:

      ```
      # systemctl restart Nagios
      ```

- To restart Nagios on RHEL 6 and SLES 11 platforms, type the following command:

  ```
  # service nagios restart
  ```

You need to restart Nagios after you change any of the Nagios `.cfg` files.

17. On the admin node, use a shell command to set the following environment variable:

    ```
    MPI_SYSLOG_COPY=1
    ```

    For example:

    ```
    # set MPI_SYSLOG_COPY=1
    ```

    Make sure to set this value in your shell before you run any HPE Performance Software — Message Passing Interface (HPE Performance MPI) or SHMEM applications.

18. (Optional) Leave the DVD in the admin node's DVD reader, and proceed to the following:

    "(Optional) Installing the MPT Nagios Plugin on an HPE SGI 8600 Rack Leader Controller or SGI ICE Rack Leader Controller" on page 101

(Optional) Installing the MPT Nagios Plugin on an HPE SGI 8600 Rack Leader Controller or SGI ICE Rack Leader Controller

In addition to the admin node, you can also install the plugin on one or more rack leader controllers (RLCs). The installation procedure is very similar to the procedure that explains how to install the plugin on the admin node. After you install the plugin on an RLC, you can start Nagios on that RLC to monitor the following:

- The messages on that RLC
- The messages related to that RLC's compute nodes

The following procedure explains how to install the plugin on an RLC.

Procedure 11-2 To install the MPT plugin on an RLC

1. From the admin node, use the `ssh` command to log into one of the RLCs as the root user.

2. Use the information in the following steps to install the plugin on the RLC:

   - Procedure 11-1, step 4 on page 98

     through

   - Procedure 11-1, step 17 on page 101

Viewing MPT Messages From Within Nagios and Clearing the Messages

The following procedure explains how to retrieve MPT messages and clear MPT messages.

Procedure 11-3 To retrieve and clear MPT messages

1. Log into one of the cluster nodes.

   If you log into the admin node and start Nagios from the admin node, Nagios displays information for the whole cluster.

   If you log into one of the RLCs and start Nagios from one of the RLCs, Nagios displays information for that RLC and its subordinate nodes.

2. Start Nagios.

   Type one of the following URLs into your browser:

   - To start Nagios on the admin node, type the following:

     `http://`*admin_name*`/nagios/`*rlc_name*

     For *admin_name*, type the hostname or IP address of the admin node.

   - To start Nagios on one of the RLCs, type the following:

     `http://`*admin_name*`/nagios/`*rlc_name*

     For *admin_name*, type the hostname or IP address of the admin node.

     For *rlc_name*, type the hostname or IP address of the RLC. For example, `r1lead`.

3. Type in the Nagios user's username and password.

   By default, the username is `nagiosadmin`. By default, the password is `sgisgi`.

4. Look for MPT information in the Nagios interface.

   By default, the plugin scans the messages in the `/var/log/messages` and reports messages to Nagios, as follows:

   - If you installed the plugin on the admin node, the plugin sends messages to Nagios for the admin node.
   - If you installed the plugin on one or more RLCs, the plugin sends messages to Nagios for the RLC and the RLC compute nodes. You need to start Nagios on the RLC to observe the messages related to that RLC.

   If you click an MPT message from within the Nagios interface, you retrieve more information about the message.

5. Use administrator commands to remedy the error conditions, if needed.

6. On the admin node, run the `check_mpt` command to clear the messages that Nagios reported.

   If you installed the plugin on the RLCs, run the `check_mpt` on RLCs, too.

   The MPT plugin works by scanning `/var/log/messages`, from beginning to end. To stop the plugin from repeatedly scanning the log file, a file offset is preserved. After you run the `check_mpt` command, the changes appear in Nagios after the next scan.

   The following examples show how to use options to the `check_mpt` command to direct the plugin to scan the system log according to your site preferences.

   Example 1. To direct the plugin to scan for only newly logged messages, use the `-C` option. The `-C` option clears all current message counts and requests that Nagios continue its scan for new messages. Also, the `-C` parameter changes the Nagios `CRITICAL` and `WARNING` status back to `OK` after you correct the reported error condition. To use this option, type the following command:

   ```
   # check_mpt -C
   ```

   Example 2. The `-X` parameter directs the plugin to start a new scan of `/var/log/messages`, clears the MPT message counts, and resets the offsets to 0. You can run `check_mpt` with the `-X` parameter after each log rotation. This command is as follows:

   ```
   # check_mpt -X
   ```

The `check_mpt` command accepts additional parameters. For more information on these parameters, type the following command to retrieve a usage statement:

```
# check_mpt -h
```

(Optional) Modifying the Notification Email

In addition to the notifications that Nagios reports in the Nagios GUI, Nagios also sends email notifications of alert conditions. If you modify the Nagios email configuration file, the Nagios email can include hostname information, which can let you identify the node upon which the error condition occurred more easily.

The `commands.cfg file` contains the following:

```
# 'notify-service-by-email-long' command definition
define command {
        command_name    notify-service-by-email-long
        command_line    /usr/bin/printf "%b" "***** Nagios *****\n\nNotification
Type: $NOTIFICATIONTYPE$\n\nService: $SERVICEDESC$\nHost: $HOSTALIAS$ \nAddress:
$HOSTADDRESS$\nState: $SERVICESTATE$\n\nDate/Time: $LONGDATETIME$\n\nAdditional
Info:\n\n$SERVICEOUTPUT$\n\n$LONGSERVICEOUTPUT$" | /usr/bin/mail -s "**
$NOTIFICATIONTYPE$ Service Alert: $HOSTALIAS$/$SERVICEDESC$ is $SERVICESTATE$ **"
$CONTACTEMAIL$
}
```

If you change `$HOSTALIAS$` to hostname, the Nagios emails include the hostname of the node upon which the error condition occurred. For example, the following file shows this enhancement:

```
# 'notify-service-by-email-long' command definition
define command {
        command_name    notify-service-by-email-long
        command_line    /usr/bin/printf "%b" "***** Nagios *****\n\nNotification
Type: $NOTIFICATIONTYPE$\n\nService: $SERVICEDESC$\nHost: `hostname` \nAddress:
$HOSTADDRESS$\nState: $SERVICESTATE$\n\nDate/Time: $LONGDATETIME$\n\nAdditional
Info:\n\n$SERVICEOUTPUT$\n\n$LONGSERVICEOUTPUT$" | /usr/bin/mail -s "**
$NOTIFICATIONTYPE$ Service Alert: $HOSTALIAS$/$SERVICEDESC$ is $SERVICESTATE$ **"
$CONTACTEMAIL$
}
```

For more information about Nagios and the Nagios email reporting feature, see your Nagios documentation.

High Performance Computing Tools

The following topics contain information about the high performance computing tools you can use with MPI programs:

- "NUMA Tools" on page 105
- "Flexible File I/O (FFIO)" on page 113

NUMA Tools

For information about the NUMA Tools, see the following:

- "About the NUMA Tools" on page 105
- "`dplace` Command" on page 106
- "`omplace` Command" on page 112
- The cpuset information in the *HPE Performance Software - Message Passing Interface Cpuset Software Guide*
- `taskset`(1) command information in the *Linux Application Tuning Guide*

About the NUMA Tools

The NUMA Tools enable data placement to specific memory locations, which minimizes communication overhead within an application.

The `dplace`(1) tool and and the cpuset tools are built upon the cpusets API. You can use these tools avoid poor data locality in your application caused by process or thread drift from CPU to CPU. The `omplace`(1) tool works like the `dplace`(1) tool and is designed for use with OpenMP applications. The differences among these tools are as follows:

- The `taskset`(1) command restricts execution to the listed set of CPUs when you specify the `-c` or `--cpu-list` option. The process is free to move among the CPUs that you specify.
- The `dplace`(1) tool differs from `taskset`(1) in that `dplace`(1) binds processes to specified CPUs in round-robin fashion. After a process is pinned, it does not

migrate, so you can use this for increasing the performance and reproducibility of parallel codes.

- Cpusets are named subsets of system cpus/memories and are used extensively in batch environments. For more information about cpusets, see the *HPE Performance Software - Message Passing Interface Cpuset Software Guide*.

`dplace` Command

You can use the `dplace`(1) command to improve the performance of processes running on your SGI nonuniform memory access (NUMA) machine.

By default, memory is allocated to a process on the node on which the process is executing. If a process moves from node to node while it is running, a higher percentage of memory references are made to remote nodes. Because remote accesses typically have higher access times, performance can degrade. CPU instruction pipelines also have to be reloaded.

The `dplace`(1) command specifies scheduling and memory placement policies for the process. You can use the `dplace` command to bind a related set of processes to specific CPUs or nodes to prevent process migrations. In some cases, this improves performance because a higher percentage of memory accesses are made to local nodes.

Processes always execute within a cpuset. The cpuset specifies the CPUs on which a process can run. By default, processes usually execute in a cpuset that contains all the CPUs in the system. For information about cpusets, see the *HPE Performance Software - Message Passing Interface Cpuset Software Guide*.

The `dplace`(1) command creates a placement container that includes all the CPUs, or a subset of CPUs, of a cpuset. The `dplace` process is placed in this container and, by default, is bound to the first CPU of the cpuset associated with the container. Then `dplace` invokes `exec` to run the command.

The command runs within this placement container and remains bound to the first CPU of the container. As the command forks child processes, the child processes inherit the container and are bound to the next available CPU of the container.

If you do not specify a placement file, `dplace` binds processes sequentially in a round-robin fashion to CPUs of the placement container. For example, if the current cpuset consists of physical CPUs 2, 3, 8, and 9, the first process launched by `dplace` is bound to CPU 2. The first child process forked by this process is bound to CPU 3. The next process, regardless of whether it is forked by a parent or a child, is bound to

CPU 8, and so on. If more processes are forked than there are CPUs in the cpuset, binding starts over with the first CPU in the cpuset.

For more information about `dplace`(1), see the `dplace`(1) man page. The `dplace`(1) man page also includes examples of how to use the command.

Example 12-1 Using the `dplace`(1) command with MPI Programs

The following command improves the placement of MPI programs on NUMA systems and verifies placement of certain data structures of a long-running MPI program:

```
% mpirun -np 64 /usr/bin/dplace -s1 -c 0-63 ./a.out
```

The `-s1` parameter causes `dplace`(1) to start placing processes with the second process, `p1`. The first process, `p0`, is not placed because it is associated with the job launch, not with the job itself. The `-c 0-63` parameter causes `dplace`(1) to use processors 0–63.

You can then use the `dlook`(1) command to verify placement of the data structures in another window on one of the slave thread PIDs. For more information about the `dlook` command, see the `dlook`(1) man page.

Example 12-2 Using the `dplace`(1) command with OpenMP Programs

The following command runs an OpenMP program on logical CPUs 4 through 7 within the current cpuset:

```
% efc -o prog -openmp -O3 program.f
% setenv OMP_NUM_THREADS 4
% dplace -c4-7 ./prog
```

Example 12-3 Using the `dplace`(1) command with OpenMP Programs

The `dplace`(1) command has a static load balancing feature, so you do not have to supply a CPU list. To place `prog1` on logical CPUs 0 through 3 and `prog2` on logical CPUs 4 through 7, type the following:

```
% setenv OMP_NUM_THREADS 4
% dplace ./prog1 &
% dplace ./prog2 &
```

You can use the `dplace -q` command to display the static load information.

Example 12-4 Using the `dplace`(1) command with Linux commands

The following examples assume that you run the `dplace` commands from a shell that runs in a cpuset consisting of physical CPUs 8 through 15.

Command	**Run Location**
`dplace -c2 date`	Runs the `date` command on physical CPU 10.
`dplace make linux`	Runs `gcc` and related processes on physical CPUs 8 through 15.
`dplace -c0-4,6 make linux`	Runs `gcc` and related processes on physical CPUs 8 through 12 or 14.
`taskset 4,5,6,7 dplace app`	The `taskset` command restricts execution to physical CPUs 12 through 15. The `dplace` command sequentially binds processes to CPUs 12 through 15.

Example 12-5 Using the `dplace` command and a debugger for verification

To use the `dplace` command accurately, you should know how your placed tasks are being created in terms of the `fork`, `exec`, and `pthread_create` calls. Determine whether each of these worker calls are an MPI rank task or are groups of pthreads created by rank tasks. Here is an example of two MPI ranks, each creating three threads:

```
cat <<EOF > placefile
firsttask cpu=0
exec name=mpiapp cpu=1
fork   name=mpiapp cpu=4-8:4 exact
thread name=mpiapp oncpu=4 cpu=5-7 exact thread name=mpiapp oncpu=8
cpu=9-11 exact EOF

#  mpirun is placed on cpu 0 in this example
#  the root mpiapp is placed on cpu 1 in this example

# or, if your version of dplace supports the "cpurel=" option:
# firsttask cpu=0
# fork   name=mpiapp cpu=4-8:4 exact
# thread name=mpiapp oncpu=4 cpurel=1-3 exact
# create 2 rank tasks, each will pthread_create 3 more
# ranks will be on 4 and 8
#  thread children on 5,6,7   9,10,11
dplace -p placefile mpirun -np 2 ~cpw/bin/mpiapp -P 3 -l


exit
```

You can use the debugger to determine if it is working. It should show two MPI rank applications, each with three pthreads, as follows:

```
>> pthreads | grep mpiapp
px *(task_struct *)e00002343c528000   17769   17769   17763  0        mpiapp
      member task: e000013817540000   17795   17769   17763  0      5 mpiapp
      member task: e000013473aa8000   17796   17769   17763  0      6 mpiapp
      member task: e000013817c68000   17798   17769   17763  0        mpiapp
px *(task_struct *)e0000234704f0000   17770   17770   17763  0        mpiapp
      member task: e000023466ed8000   17794   17770   17763  0      9 mpiapp
      member task: e00002384cce0000   17797   17770   17763  0        mpiapp
      member task: e00002342c448000   17799   17770   17763  0        mpiapp
```

You can also use the debugger to see a root application, the parent of the two MPI rank applications, as follows:

```
>> ps | grep mpiapp
0xe00000340b300000   1139  17763  17729      1  0xc800000   -  mpiapp
0xe00002343c528000   1139  17769  17763      0  0xc800040   -  mpiapp
0xe0000234704f0000   1139  17770  17763      0  0xc800040   8  mpiapp
```

These are placed as specified:

```
>> oncpus e00002343c528000 e000013817540000 e000013473aa8000
>> e000013817c68000 e0
000234704f0000 e000023466ed8000 e00002384cce0000 e00002342c448000
task: 0xe00002343c528000  mpiapp cpus_allowed: 4
task: 0xe000013817540000  mpiapp cpus_allowed: 5
task: 0xe000013473aa8000  mpiapp cpus_allowed: 6
task: 0xe000013817c68000  mpiapp cpus_allowed: 7
task: 0xe0000234704f0000  mpiapp cpus_allowed: 8
task: 0xe000023466ed8000  mpiapp cpus_allowed: 9
task: 0xe00002384cce0000  mpiapp cpus_allowed: 10
```

```
task: 0xe00002342c448000  mpiapp cpus_allowed: 11
```

Example 12-6 Using the `dplace`(1) command for compute thread placement troubleshooting

Sometimes compute threads do not end up on unique processors when using commands such a `dplace`(1) or `profile.pl`.

In this example, assume that the `dplace -s1 -c0-15` command bound 16 processes to run on 0-15 CPUs. However, output from the `top`(1) command shows only 13 CPUs running with CPUs 13, 14, and 15 still idle, and CPUs 0, 1 and 2 are shared with 6 processes.

```
263 processes: 225 sleeping, 18 running, 3 zombie, 17 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
             total  1265.6%    0.0%   28.8%   0.0%    11.2%    0.0%  291.2%

             cpu00  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu01   90.1%    0.0%    0.0%   0.0%     9.7%    0.0%    0.0%

             cpu02   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu03   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu04  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu05  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu06  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu07   88.4%    0.0%   10.6%   0.0%     0.8%    0.0%    0.0%

             cpu08  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu09   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu10   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu11   88.1%    0.0%   11.2%   0.0%     0.6%    0.0%    0.0%

             cpu12   99.7%    0.0%    0.2%   0.0%     0.0%    0.0%    0.0%
             cpu13    0.0%    0.0%    2.5%   0.0%     0.0%    0.0%   97.4%

             cpu14    0.8%    0.0%    1.6%   0.0%     0.0%    0.0%   97.5%

             cpu15    0.0%    0.0%    2.4%   0.0%     0.0%    0.0%   97.5%
 Mem:  60134432k av, 15746912k used, 44387520k free,       0k shrd,
672k buff
         351024k active,            13594288k inactive

  Swap: 2559968k av,       0k used, 2559968k free
  2652128k cached

    PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND

   7653 ccao      25   0  115G 586M  114G R    99.9  0.9   0:08   3 mocassin

  7656 ccao      25   0  115G 586M  114G R    99.9  0.9   0:08   6 mocassin

   7654 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   4 mocassin

   7655 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   5 mocassin

   7658 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   8 mocassin

   7659 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   9 mocassin

   7660 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08  10 mocassin

   7662 ccao      25   0  115G 586M  114G R    99.7  0.9   0:08  12 mocassin

   7657 ccao      25   0  115G 586M  114G R    88.5  0.9   0:07   7 mocassin

   7661 ccao      25   0  115G 586M  114G R    88.3  0.9   0:07  11 mocassin

   7649 ccao      25   0  115G 586M  114G R    55.2  0.9   0:04   2 mocassin

   7651 ccao      25   0  115G 586M  114G R    54.1  0.9   0:03   1 mocassin

   7650 ccao      25   0  115G 586M  114G R    50.0  0.9   0:04   0 mocassin

   7647 ccao      25   0  115G 586M  114G R    49.8  0.9   0:03   0 mocassin
```

```
7652 ccao       25   0  115G 586M  114G R    44.7  0.9   0:04   2 mocassin

7648 ccao       25   0  115G 586M  114G R    35.9  0.9   0:03   1 mocassin
```

Even if an application starts some threads executing for a very short time, the threads still have taken a token in the CPU list. Then, when the compute threads are finally started, the list is exhausted and restarts from the beginning. Consequently, some threads end up sharing the same CPU. To bypass this, try to eliminate the ghost thread creation, as follows:

- Check for a call to the system function. This is often responsible for the placement failure due to unexpected thread creation. If all the compute processes have the same name, you can do this by issuing a command such as the following:

  ```
  % dplace -c0-15 -n compute-process-name ...
  ```

- You can also run `dplace -e -c0-32` on 16 CPUs to understand the pattern of the thread creation. If this pattern is the same from one run to the other (unfortunately race between thread creation often occurs), you can find the right flag to `dplace`. For example, if you want to run on CPUs 0-3, with `dplace -e -C0-16` and you see that threads are always placed on CPU 0, 1, 5, and 6, then one of the following commands should place your threads correctly:

  ```
  dplace -e -c0,1,x,x,x,2,3
  ```

  or

  ```
  dplace -x24 -c0-3  # x24 =11000, place the 2 first and skip 3 before placing
  ```

`omplace` Command

The `omplace`(1) command controls the placement of MPI processes and OpenMP threads. This command is a wrapper script for `dplace`(1). Use `omplace`(1), rather than `dplace`(1), if your application uses MPI, OpenMP, pthreads, or hybrid MPI/OpenMP and MPI/pthreads codes. The `omplace`(1) command generates the proper `dplace`(1) placement file syntax automatically. It also supports some unique options, such as block-strided CPU lists.

The `omplace`(1) command causes the successive threads in a hybrid MPI/OpenMP job to be placed on unique CPUs. The CPUs are assigned in order from the effective CPU list within the containing cpuset. The CPU placement is performed by dynamically generating a placement file and invoking `dplace`(1) with the MPI job launch.

For example, to run two MPI processes with four threads per process, and to display the generated placement file, type a command similar to the following:

```
# mpirun -np 2 omplace -nt 4 -vv ./a.out
```

The preceding command places the threads as follows:

```
rank 0 thread 0 on CPU 0
rank 0 thread 1 on CPU 1
rank 0 thread 2 on CPU 2
rank 0 thread 3 on CPU 3
rank 1 thread 0 on CPU 4
rank 1 thread 1 on CPU 5
rank 1 thread 2 on CPU 6
rank 1 thread 3 on CPU 7
```

For more information, see the `omplace`(1) man page.

Flexible File I/O (FFIO)

The following topics explain how to use FFIO:

- "About FFIO" on page 113
- "Environment Variables" on page 114
- "FFIO Examples" on page 116
- "Multithreading Considerations" on page 118
- "Application Examples " on page 119
- "Event Tracing " on page 120
- "System Information and Issues " on page 121

About FFIO

Flexible File I/O (FFIO) can improve the file I/O performance of existing applications without having to resort to source code changes. The current executable remains unchanged. Knowledge of source code is not required, but some knowledge of how the source and the application software work can help you better interpret and optimize FFIO results. To take advantage of FFIO, all you need to do is to set some environment variables before running your application.

The FFIO subsystem allows you to define one or more additional I/O buffer caches for specific files to augment the Linux kernel I/O buffer cache. The FFIO subsystem then manages this buffer cache for you. In order to accomplish this, FFIO intercepts standard I/O calls such as open, read, and write, and replaces them with FFIO equivalent routines. These routines route I/O requests through the FFIO subsystem, which uses the user-defined FFIO buffer cache.

FFIO can bypass the Linux kernel I/O buffer cache by communicating with the disk subsystem via direct I/O. This bypass gives you precise control over cache I/O characteristics and allows for more efficient I/O requests. For example, doing direct I/O in large chunks (for example, 16 megabytes) allows the FFIO cache to amortize disk access. All file buffering occurs in user space when FFIO is used with direct I/O enabled. This differs from the Linux buffer cache mechanism, which requires a context switch in order to buffer data in kernel memory. Avoiding this kind of overhead helps FFIO to scale efficiently.

Another important distinction is that FFIO allows you to create an I/O buffer cache dedicated to a specific application. The Linux kernel, on the other hand, has to manage all the jobs on the entire system with a single I/O buffer cache. As a result, FFIO typically outperforms the Linux kernel buffer cache when it comes to I/O intensive throughput.

Environment Variables

To use FFIO, set one of the following environment variables: `LD_PRELOAD` or `FF_IO_OPTS`.

In order to enable FFIO to trap standard I/O calls, set the `LD_PRELOAD` environment variable, as follows:

```
# export LD_PRELOAD="/usr/lib64/libFFIO.so"
```

The `LD_PRELOAD` software is a Linux feature that instructs the linker to preload the indicated shared libraries. In this case, `libFFIO.so` is preloaded and provides the routines that replace the standard I/O calls. An application that is not dynamically linked with the `glibc` library cannot work with FFIO because the standard I/O calls cannot be intercepted. To disable FFIO, type the following:

```
# unset LD_PRELOAD
```

The FFIO buffer cache is managed by the `FF_IO_OPTS` environment variable. The syntax for setting this variable can be quite complex. A simple format for defining this variable is as follows:

```
export FF_IO_OPTS  'string(eie.direct.mbytes:size:num:lead:share:stride:0)'
```

You can use the following parameters with the `FF_IO_OPTS` environment variable:

string	Matches the names of files that can use the buffer cache.
size	Number of 4k blocks in each page of the I/O buffer cache.
num	Number of pages in the I/O buffer cache.
lead	The maximum number of read-ahead pages.
share	A value of 1 means a shared cache, 0 means private.
stride	Note that the number after the `stride` parameter is always 0.

Example 1. Assume that you want a shared buffer cache of 128 pages. Each page is to be 16 megabytes (that is, 4096*4k). The cache has a lead of six pages and uses a stride of one. The command is as follows:

```
% setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0)'
```

Each time the application opens a file, the FFIO code checks the file name to see if it matches the string supplied by `FF_IO_OPTS`. The file's path name is not considered when checking for a match against the string. For example, file names of `/tmp/test16` and `/var/tmp/testit` both match.

Example 2. This more complicated usage of `FF_IO_OPTS` builds upon the previous example. Multiple types of file names can share the same cache, as the following example shows:

```
% setenv FF_IO_OPTS 'output* test*(eie.direct.mbytes:4096:128:6:1:1:0)'
```

Example 3. You can specify multiple caches with `FF_IO_OPTS`. In the example that follows, files of the form `output*` and `test*` share a 128 page cache of 16 megabyte pages. The file `special42` has a 256–page private cache of 32 megabyte pages. The command, which uses the backslash (\) continuation character, is as follows:

```
% setenv FF_IO_OPTS 'output* test*(eie.direct.mbytes:4096:128:6:1:1:0) \
special42(eie.direct.mbytes:8192:256:6:0:1:0)'
```

Additional parameters can be added to `FF_IO_OPTS` to create feedback that is sent to standard output. For examples of this diagnostic output, see the following:

"FFIO Examples" on page 116

FFIO Examples

This topic includes some simple FFIO examples. Assume that `LD_PRELOAD` is set for the correct library, and `FF_IO_OPTS` is defined as follows:

```
% setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0)'
```

It can be difficult to tell what FFIO might or might not be doing even with a simple program. The examples in this topic use a small C program called `fio` that reads 4–megabyte chunks from a file for 100 iterations. When the program runs, it produces the following output:

```
% ./fio -n 100 /build/testit
Reading 4194304 bytes 100 times to /build/testit
Total time  = 7.383761
Throughput  = 56.804439 MB/sec
```

Example 1. You can direct a simple FFIO operations summary to standard output by making the following simple addition to `FF_IO_OPTS`:

```
% setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0, event.summary.mbytes.notrace )'
```

This new setting for `FF_IO_OPTS` generates the following summary on standard output when the program runs:

```
% ./fio -n 100 /build/testit
Reading 4194304 bytes 100 times to /build/testit
Total time  = 7.383761
Throughput  = 56.804439 MB/sec

event_close(testit)    eie <-->syscall   (496 mbytes)/( 8.72 s)=   56.85 mbytes/s
oflags=0x0000000000004042=RDWR+CREAT+DIRECT
sector size =4096(bytes)
```

```
cblks =0  cbits =0x0000000000000000
current file size =512 mbytes   high water file size =512 mbytes

function     times      wall     all     mbytes     mbytes       min         max        avg
             called     time    hidden  requested  delivered   request     request    request
    open          1     0.00
    read          2     0.61                32         32         16          16         16
    reada        29     0.01       0       464        464         16          16         16
    fcntl
       recall
       reada     29     8.11
       other      5     0.00
    flush         1     0.00
    close         1     0.00
```

Two synchronous reads of 16 megabytes each were issued, for a total of 32 megabytes. In addition, there were 29 asynchronous reads (`reada`) issued, for a total of 464 megabytes.

Example 2. You can generate additional diagnostic information by specifying the `.diag` modifier. The following is an example of the diagnostic output generated when the `.diag` modifier is used:

```
% setenv FF_IO_OPTS 'test*(eie.direct.diag.mbytes:4096:128:6:1:1:0 )'
% ./fio -n 100 /build/testit
Reading 4194304 bytes 100 times to /build/testit
Total time  = 7.383761
Throughput  = 56.804439 MB/sec

eie_close EIE final stats for file /build/testit
eie_close  Used shared eie cache 1
eie_close  128 mem pages of 4096 blocks (4096 sectors), max_lead = 6 pages
eie_close  advance reads used/started :       23/29    79.31%   (1.78 seconds wasted)
eie_close  write hits/total           :        0/0      0.00%
eie_close  read  hits/total           :       98/100   98.00%
eie_close  mbytes transferred    parent --> eie --> child      sync        async
eie_close                                 0           0        0             0
eie_close                               400         496        2            29 (0,0)
eie_close                        parent <-- eie <-- child

eie_close EIE stats for Shared cache 1
eie_close  128 mem pages of 4096 blocks
eie_close  advance reads used/started :       23/29    79.31%   (0.00 seconds wasted)
eie_close  write hits/total           :        0/0      0.00%
eie_close  read  hits/total           :       98/100   98.00%
eie_close  mbytes transferred    parent --> eie --> child      sync        async
eie_close                                0                      0            0
eie_close                              400         496          2             29 (0,0)
```

The preceding output lists information for both the file and the cache. In the `mbytes transferred` information, the lines in **bold** are for write and read operations, respectively. Only for very simple I/O patterns can the difference between (`parent --> eie`) and (`eie --> child`) read statistics be explained by the number of read aheads. For random reads of a large file over a long period of time, this is not the case. All write operations count as `async`.

You can generate additional diagnostic information by specifying the `.diag` modifier and the `.event.summary` modifier. The two modifiers operate independently from one another. The following specification uses both modifiers:

```
% setenv FF_IO_OPTS 'test*(eie.diag.direct.mbytes:4096:128:6:1:1:0, event.summary.mbytes.notrace )'
```

Multithreading Considerations

FFIO works with applications that use MPI for parallel processing. An MPI job assigns each thread a number or rank. The master thread has rank 0, while the remaining slave threads have ranks from 1 to *N*-l where *N* is the total number of threads in the MPI job. It is important to consider that the threads comprising an MPI job do not necessarily have access to each others' address space. As a result, there is no way for the different MPI threads to share the same FFIO cache. By default, each thread defines a separate FFIO cache based on the parameters defined by `FF_IO_OPTS`.

Having each MPI thread define a separate FFIO cache, based on a single environment variable (`FF_IO_OPTS`), can waste a lot of memory. Fortunately, FFIO provides a mechanism that allows you to specify a different FFIO cache for each MPI thread via the following environment variables:

```
setenv FF_IO_OPTS_RANK0 'result*(eie.direct.mbytes:4096:512:6:1:1:0)'
setenv FF_IO_OPTS_RANK1 'output*(eie.direct.mbytes:1024:128:6:1:1:0)'
setenv FF_IO_OPTS_RANK2 'input*(eie.direct.mbytes:2048:64:6:1:1:0)'
             .
             .
             .
setenv FF_IO_OPTS_RANKN-1 ...   (N = number of threads).
```

Each rank environment variable is set using the exact same syntax as `FF_IO_OPTS` and each defines a distinct cache for the corresponding MPI rank. If the cache is designated as shared, all files within the same ranking thread can use the same cache. FFIO works with SGI MPI, HP MPI, and LAM MPI. In order to work with MPI applications, FFIO needs to determine the rank of callers by invoking the `mpi_comm_rank_()` MPI library routine. Therefore, FFIO needs to determine the location of the MPI library used by the application. To accomplished this, set one, and only one, of the following environment variables:

- `setenv SGI_MPI /usr/lib`
- `setenv LAM_MPI`
- `setenv HP_MPI`

Note: LAM MPI and HP MPI are usually distributed via a third party application. The precise paths to the LAM and the HP MPI libraries are application dependent. See the application installation guide to find the correct path.

To use the rank functionality, both the MPI and `FF_IO_OPTS_RANK0` environment variables must be set. If either variable is not set, then the MPI threads all use `FF_IO_OPTS`. If both the MPI and the `FF_IO_OPTS_RANK0` variables are defined but, for example, `FF_IO_OPTS_RANK2` is undefined, all rank 2 files generate a `no match` with FFIO. This means that none of the rank 2 files are cached by FFIO. In this case, the software does not default to `FF_IO_OPTS`.

Fortran and C/C++ applications that use the `pthreads` interface create threads that share the same address space. These threads can all make use of the single FFIO cache defined by `FF_IO_OPTS`.

Application Examples

FFIO has been deployed successfully with several high-performance computing applications, such as Nastran and Abaqus. In a recent customer benchmark, an eight-way Abaqus throughput job ran approximately twice as fast when FFIO was used. The FFIO cache used 16–megabyte pages (that is, `page_size` = 4096) and the cache size was 8.0 gigabytes. As a rule of thumb, it was determined that setting the FFIO cache size to roughly 10-15% of the disk space required by Abaqus yielded reasonable I/O performance. For this benchmark, the `FF_IO_OPTS` environment variable was defined as follows:

```
% setenv FF_IO_OPTS '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
      *.nck* *.sct *.lop *.ngr *.elm *.ptn* *.stp* *.eig *.lnz* *.mass *.inp* *.scn* *.ddm
      *.dat* fort*(eie.direct.nodiag.mbytes:4096:512:6:1:1:0,event.summary.mbytes.notrace)'
```

For the MPI version of Abaqus, different caches were specified for each MPI rank, as follows:

```
% setenv FF_IO_OPTS_RANK0 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
      *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
      *.dat* fort*(eie.direct.nodiag.mbytes:4096:512:6:1:1:0,event.summary.mbytes.notrace)'

% setenv FF_IO_OPTS_RANK1 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
      *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
      *.dat* fort*(eie.direct.nodiag.mbytes:4096:16:6:1:1:0,event.summary.mbytes.notrace)'

% setenv FF_IO_OPTS_RANK2 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
      *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
      *.dat* fort*(eie.direct.nodiag.mbytes:4096:16:6:1:1:0,event.summary.mbytes.notrace)'

% setenv FF_IO_OPTS_RANK3 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
      *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
      *.dat* fort*(eie.direct.nodiag.mbytes:4096:16:6:1:1:0,event.summary.mbytes.notrace)'
```

Event Tracing

If you specify the `.trace` option as part of the `event` parameter, you can enable the event tracing feature in FFIO.

For example:

```
% setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0, event.summary.mbytes.trace )'
```

This option generates files of the form `ffio.events.`*pid* for each process that is part of the application. By default, event files are placed in `/tmp`. To chang this destination, set the `FFIO_TMPDIR` environment variable. These files contain time-stamped events for files using the FFIO cache and can be used to trace I/O activity such as I/O sizes and offsets.

System Information and Issues

The FFIO subsystem supports applications written in C, C++, and Fortran. C and C++ applications can be built with either the Intel or gcc compiler. Only Fortran codes built with the Intel compiler work with FFIO.

The following restrictions on FFIO must also be observed:

- The FFIO implementation of `pread/pwrite` is not correct. The file offset advances.
- Do not use FFIO for I/O on a socket.
- Do not link your application with the `librt` asynchronous I/O library.
- FFIO does not intercept calls that operate on files in `/proc`, `/etc`, and `/dev`.
- FFIO does not intercept calls that operate on `stdin`, `stdout`, and `stderr`.
- FFIO is not intended for generic I/O applications such as `vi`, `cp`, or `mv`, and so on.

Guidelines for Using the Message Passing Toolkit (MPT) on a Virtual Machine Within an SGI UV Computer System

This appendix section includes the following topics:

- "About MPT on a Virtual Machine" on page 123
- "Installing Software Within the Virtual Machine (VM)" on page 124
- "Adjusting SGI UV Virtual Machine System Settings" on page 124
- "Running HPE Performance Software — Message Passing Interface (HPE Performance MPI) Programs From Within a Virtual Machine (VM)" on page 126

About MPT on a Virtual Machine

You can configure a virtual machine (VM) on an SGI UV system. The VM creates a general-purpose computer, and MPT can run on that computer. When you use MPT from within a VM, however, you can expect differences in the computing environment and differences with regard to your application's behavior.

For information about how to configure a VM on an SGI hardware platform, see the documentation for Red Hat Enterprise Linux (RHEL) or for SLES.

If you are an administrator, use the information in the following topics to configure the VM environment appropriately:

- "Installing Software Within the Virtual Machine (VM)" on page 124
- "Adjusting SGI UV Virtual Machine System Settings" on page 124

If you are an application developer, use the information in the following topic to understand how your program might behave differently when running from within a VM:

- "Running HPE Performance Software — Message Passing Interface (HPE Performance MPI) Programs From Within a Virtual Machine (VM)" on page 126

Installing Software Within the Virtual Machine (VM)

The following procedure explains the software that you need to install in the VM in order for Message Passing Interface (MPI) programs to run on the VM.

Procedure A-1 To install the software for MPI programs

1. Install and configure the operating system (RHEL or SLES) and the HPE System Foundation Software on the SGI UV computer.

   For installation information, see the *SGI UV System Software Installation and Configuration Guide*.

2. Install and configure the VM according to your operating system vendor's instructions.

   Note that RHEL and SLES do not support InfiniBand technology from within a VM. Other OFED providers support InfiniBand technology from within a VM through single-root I/O virtualization (SR-IOV), but HPE does not support SR-IOV or other alternatives to the distribution-supplied OFED.

3. (Optional) Install the HPE System Foundation Software into the VM.

   For installation information, see the *SGI UV System Software Installation and Configuration Guide*.

4. Install the HPE Performance Software into the VM.

   For installation information, see the HPE Performance Software release notes.

5. Install MPT into the VM.

   For installation information, see Chapter 2, "Getting Started" on page 11.

Adjusting SGI UV Virtual Machine System Settings

For best performance, HPE recommends to change certain operating system settings after the software installation is complete.

The following procedure explains how to adjust the number of files that can be open at a given time.

Procedure A-2 To adjust system settings

1. Log into the SGI UV system as the root user.

2. Type `cpumap` command to retrieve the number of cores on the SGI UV computer.

   For example:

   ```
   # cpumap
   This is an SGI UV
   model name          : Genuine Intel(R) CPU @ 2.60GHz
   Architecture        : x86_64
   cpu MHz             : 2600.072
   cache size          : 20480 KB (Last Level)

   Total Number of Sockets                 : 16
   Total Number of Cores                   : 128   (8 per socket)
   Hyperthreading                          : ON
   Total Number of Physical Processors     : 128
   Total Number of Logical Processors      : 256   (2 per Phys Processor)

   UV Information
    HUB Version:                            UVHub  3.0
    Number of Hubs:                         16
    Number of connected Hubs:               16
    Number of connected NUMAlink ports:     128
   =============================================================================
   . . .
   ```

   The `Total Number of Cores` line reveals that there are 128 cores, 8 per socket.

3. Display the contents of the `/etc/sysctl-conf` file.

   For example, type the following command:

   ```
   # less /etc/sysctl.conf
   ...
   fs.file-max = 8204481
   ...
   ```

4. (Conditional) Use a text editor to open file `sysctl.conf` and increase the value of the `fs.file-max` parameter in the `/etc/sysctl.conf` file.

   Perform this step if the number of cores on your computer is greater than 512 and the `fs.file-max` parameter is set to less than 10,000,000.

   For optimum performance within a VM, set the `fs.file-max` parameter that is at least `10000000` on SGI UV systems with 512 cores or more.

Running HPE Performance Software — Message Passing Interface (HPE Performance MPI) Programs From Within a Virtual Machine (VM)

The following list explains some of the differences between running an MPI or SHMEM program on a native SGI hardware platform versus running an MPI or SHMEM program from within a VM hosted by an SGI UV system:

- Hardware-dependent features might not exist on a VM.

  When you run an MPI program on a VM, the environment detects the virtual nature of the platform and ignores any SGI hardware-specific features. The following hardware features are not available to an application that runs in a VM: NUMAlink, Superpages, the SGI UV timer, the HUB ASIC, and hardware performance counters. In addition, processor-specific performance diagnostics are limited.

  If your application uses hardware technologies that are not specific to SGI hardware platforms, you can expect that the VM can honor those non-specific technologies.

- Topology characteristics might be different.

  An application that relies on the topology of an SGI hardware platform needs to be run on a VM that was configured with topology that mimics the SGI hardware platform. MPI programs do not automatically use special topology characteristics effectively. If the application requires special heuristics for locality and placement, you need to configure that into the VM.

- XPMEM libraries are beneficial in very large VMs.

  HPE has tested XPMEM on VMs. XPMEM loads, and your application can call XPMEM routines successfully. However, XPMEM is useful only on systems with very large memory.

- No InfiniBand support.

  The RHEL and SLES operating systems do not support InfiniBand technology in VMs. Consult your system administrator to find out if single-root I/O virtualization (SR-IOV) is configured on the VM.

Array Services System Administration Information

This appendix contains the following topics:

- "About Installing and Configuring Array Services Manually" on page 127
- "Installing the Array Services Software on HPE Apollo Cluster Systems" on page 2
- "Manually Configuring Array Services on Multiple Hosts" on page 128
- "Changing the Security Access Level in the `AUTHENTICATION` parameter" on page 131
- "Configuring Nodes Into Arrays" on page 132
- "About the Array Configuration Files" on page 134
- "Designing Array Services Commands" on page 136
- "Testing Configuration Changes After Creating New Array Services Commands" on page 144

About Installing and Configuring Array Services Manually

The topics in this appendix section contain information for system administrators who support the Array Services software on a cluster computing system. These topics explain how to install Array Services manually and how to configure Array Services to suit your site's needs.

The topics in this appendix section are as follows:

- "Manually Configuring Array Services on Multiple Hosts" on page 128
- "Changing the Security Access Level in the `AUTHENTICATION` parameter" on page 131
- "Configuring Nodes Into Arrays" on page 132
- "About the Array Configuration Files" on page 134
- "Designing Array Services Commands" on page 136

- "Testing Configuration Changes After Creating New Array Services Commands" on page 144

Manually Configuring Array Services on Multiple Hosts

You can configure Array Services in an automated way or manually. The following list shows where you can find the standard, automated procedures:

- For cluster systems, use the information in the following:
  - *HPE SGI Management Suite Installation and Configuration Guide*
  - "Installing the Array Services Software on HPE Apollo Cluster Systems" on page 2
- For SGI UV computer systems, use the information in the following:

  Chapter 1, "Configuring the Message Passing Toolkit (MPT)" on page 1

The information in this appendix section explains how to configure Array Services in a manual way, which allows you to make customizations at installation time, if necessary.

The following procedure explains how to configure Array Services to run on multiple hosts.

Procedure B-1 To configure Array Services for multiple hosts

1. Log in as root on one of the hosts you want to include in the array.

   You must be logged in as an administrator to perform this procedure.

   For example, on an SGI ICE X system, log into one of the service nodes. You can include service nodes and compute nodes in the array.

2. (Optional) Install the MUNGE package from the HPE Performance Software — Message Passing Interface (HPE Performance MPI) software distribution.

   The optional MUNGE software package enables additional security for Array Services operations.

   During MUNGE installation, make sure of the following:

   - The MUNGE key that is used is the same across all the nodes in the array.

The MUNGE key resides in `/etc/munge/munge.key`.

- You configure a good time clock source, such as an NTP server. MUNGE depends on time synchronization across all nodes in the array.

To install MUNGE, use one of the following commands:

- On Red Hat Enterprise Linux platforms: `yum install munge`
- On SUSE Linux Enterprise Server platforms: `zypper install munge`

For more information about how to install MUNGE, see the HPE Performance MPI release notes.

3. Open file `/etc/array/arrayd.conf` with a text editor.
4. Edit the `/etc/array/arrayd.conf` file to list the machines in the array.

   This file enables you to configure many characteristics of an Array Services environment. The required specifications are as follows:

   - The array name.
   - The hostnames of the array participants.
   - A default destination array.

   For more information about the additional characteristics that you can specify in the `arrayd.conf` file, see the `arrayd.conf`(4) man page.

   For an example `arrayd.conf` file, see file `/usr/lib/array/arrayd.conf.template`.

   Example 1. The following lines specify an array name (`sgicluster`) and two hostnames. Specify each hostname on its own line. `array` and `machine` are keywords in the file.

```
array sgicluster
              machine host1
              machine host2
```

   Example 2. The following line sets a default array name.

```
destination array sgicluster
```

5. Save and close file `/etc/array/arrayd.conf`.

6. Use a text editor to open file `/etc/array/arrayd.auth`.

7. (Optional) Change the authentication method from the default of `NOREMOTE` to a method of your choosing.

   By default, the Array Services software does not allow remote access to the array. You can change this authentication method. For information about the various authentication methods, see the following:

   "Changing the Security Access Level in the `AUTHENTICATION` parameter" on page 131

   To change the access method, complete the following steps:

   - Search for the string `AUTHENTICATION NOREMOTE`, and insert a # character in column 1 to comment out the line.
   - Enable the security level under which you want Array Services to operate.

     This step specifies the authentication mechanism to use when Array Services messages pass between the Array Services daemons. Possible security levels are `NONE`, `SIMPLE`, or `MUNGE`, as follows:

     – If no authentication is required, remove the # character from column 1 of the `AUTHENTICATION NONE` line.

     – To enable simple authentication, ensure that there is no # in column 1 of the `AUTHENTICATION SIMPLE` line. This is the default.

     – To enable authentication through MUNGE, remove the # character from column 1 of the `AUTHENTICATION MUNGE` line.

       Make sure that MUNGE has been installed, as prescribed earlier in this procedure.

   - Save and close file `/etc/array/arrayd.auth`.

8. (Optional) Reset the default user account or the default array port.

   By default, the Array Services installation and configuration process sets the following defaults in the `/etc/array/arrayd.conf` configuration file:

   - A default user account of `arraysvcs`.

     Array Services requires that a user account exist on all hosts in the array for the purpose of running certain Array Services commands. If you create a different account, make sure to update the `arrayd.conf` file and set the user account permissions correctly on all hosts.

- A default port number of 5434.

  The `/etc/services` file contains a line that defines the `arrayd` service and port number as follows:

  ```
  sgi-arrayd   5434/tcp    # Array Services daemon
  ```

  You can set any value for the port number, but all systems mentioned in the `arrayd.conf` file must use the same value.

9. Type one of the following commands to restart Array Services:

   - On RHEL 7.*X* or SLES 12 SP*X* systems, type the following command:

     ```
     systemctl restart array
     ```

   - On RHEL 6.*X* or SLES 11 SP*X* systems, type the following command:

     ```
     /etc/init.d/array restart
     ```

10. Repeat the preceding steps on the other hosts or copy the `/etc/array/arrayd.conf` and `/etc/array/arrayd.auth` files to the other hosts.

    The Array Services feature requires that the configuration files on each participant host include the list of host participants and the authentication method. The files can contain additional, host-specific information.

Changing the Security Access Level in the `AUTHENTICATION` parameter

The `AUTHENTICATION` parameter in the `/etc/array/arrayd.auth` file specifies access to the array. The `AUTHENTICATION` parameter can have one of the following settings:

- `NOREMOTE` (default).

  When set to `NOREMOTE`, the `arrayd` daemon allows only local access to the array. That is, the `arrayd` daemon does not allow remote requests to access the array.

  Use `NOREMOTE` if the array is attached to a public network or if individual machines cannot be trusted.

- NONE.

  When set to NONE, the arrayd daemon assumes that remote users identify themselves accurately and honestly when making requests. In other words, if a request claims to be coming from user abc, the arrayd daemon assumes that it is in fact from user abc and not somebody spoofing abc.

  All requests from remote systems are authenticated using a mechanism that involves private keys that are known only to the superusers on the local and remote systems. Requests originating on systems that do not have these private keys are rejected. For more information, see the section on authentication information in the arrayd.conf(4) man page.

  This setting should be adequate for systems that are behind a network firewall or otherwise protected from hostile attack. In this situation, all the users inside the firewall are presumed to be non-hostile.

  Do not set AUTHENTICATION to NONE if the array is attached to a public network or if individual machines cannot be trusted.

- SIMPLE. Generates hostname/key pairs by using the OpenSSL rand command, 64–bit values (if available), or by using $RANDOM Bash facilities. For more information, see arrayd.auth(5).

- MUNGE.

  When set to MUNGE, uses the MUNGE credential encoder. For more information, see munge(1).

The Array Services daemon, arrayd, runs as root and does not support mapping of user, group, or project names between two different namespaces. All members of an array are assumed to share the same namespace for users, groups, and projects. Thus, if systems A and B are members of the same array, username abc on system A is assumed to be the same user as username abc on system B. This is most significant in the case of username root. Authentication should be used to prevent access to an array by machines using a different namespace.

Configuring Nodes Into Arrays

The following topics contain examples that show how to specify the nodes in an array in the arrayd.conf(5) file:

- "Specifying an Array Name and Machine Names" on page 133

- "Specifying IP Addresses and Ports" on page 133
- "Specifying Additional Attributes" on page 134

Specifying an Array Name and Machine Names

Often, the hostname of each node is the same as the node's name to the site domain name services (DNS). The following example defines an array where this is the case:

```
array simple
        machine congo
        machine niger
        machine nile
```

To access this array, the user needs to specify the the array name, `simple`, as the argument to the `-a` option on the `array` command and the `ainfo` command.

One array name should be specified in a `DESTINATION ARRAY` local option as the default array and reported by `ainfo dflt`. Local options are listed under "Configuring Local Options" on page 141.

Specifying IP Addresses and Ports

At your site, if a machine's IP address cannot be obtained from the given hostname, provide a `hostname` subentry to specify either a fully qualified domain name (FQDN) or an IP address, as follows:

```
array simple
        machine congo
            hostname congo.engr.hitech.com
            port 8820
        machine niger
            hostname niger.engr.hitech.com
        machine nile
            hostname "198.206.32.85"
```

The preceding example uses the `port` subentry to specify that `arrayd` in a particular machine use a different socket number than the default of 5434.

Specifying Additional Attributes

If you want the `ainfo` command to display certain strings, you can insert these values as subentries to the `array` entry. The following are some examples:

```
array simple
       array_attribute config_date="04/03/96"
       machine a_node
       machine_attribute aka="congo"
       hostname congo.engr.hitech.com
```

Tip: You can write code that fetches any array name, machine name, or attribute string from any node in the array.

About the Array Configuration Files

The Array Services configuration files are as follows:

- `/etc/array/arrayd.conf`
- `/etc/array/arrayd.auth`
- `/etc/sysconfig/array`

The configuration files contain array information, node information, authentication key information, and valid commands. The Array Services daemon reads each configuration file when it starts. Typically, the daemon starts on each node at boot time and then runs as a background process. The Array Services commands call the daemon process on each node to obtain information. You can also run the daemon from a command line. For example, you might want to run the daemon from a command line to check the syntax of a configuration file.

The following topics contain more configuration file information:

- "About Configuration File Formats and Contents" on page 134
- "About Loading Configuration Data" on page 135

About Configuration File Formats and Contents

A configuration file is a readable text file that contains the following types of entries:

- Array definition information, which describes this array and other known arrays, including array names and the node names and types.
- Command definitions, which specifies the usage and operation of a command that can be invoked through the `array` command.
- Authentication information, which specifies the authentication key numbers used to access the array. Not all arrays use authentication keys.
- Local options, which are options that modify the operation of the other entries or `arrayd`.

Within the configuration files, you can use blank lines, white space, and comment lines that begin with a pound character (#) for readability. Entries can be in any order in any of the Array Services configuration files.

Besides punctuation, entries have a keyword-based syntax. Keyword recognition is not case sensitive, but keywords appear in uppercase in this documentation and in the `man`(1) page. As the `arrayd.conf`(4) man page describes, the entries are formed from keywords, numbers, and quoted strings.

About Loading Configuration Data

When run as a command, the Array Services daemon, `arrayd`, accepts one or more file names as arguments. It reads them all and treats them like logical continuations. In effect, it concatenates them. If you do not specify any file names, it reads the following configuration files:

- `/etc/array/arrayd.conf`
- `/etc/array/arrayd.auth`
- `/etc/sysconfig/array`

  This file can contain a list of files and `arrayd` command-line options. The start-up script that launches `arrayd` at boot time reads this file.

Because configuration data can reside in two or more files, you can combine different strategies. For example:

- One file can have different access permissions than another. Typically, `/etc/array/arrayd.conf` is world-readable and contains the available `array` commands, while `/etc/array/arrayd.auth` is readable only by root and contains authentication codes.

- One node can have different configuration data than another. For example, certain commands might be defined only on certain nodes, or only the nodes used for interactive logins might know the names of all other nodes.
- You can use NFS-mounted configuration files. You could put a small configuration file on each machine to define the array and authentication keys, but you could have a larger file defining `array` commands that is NFS-mounted from one node.

After you modify the configuration files, you can make `arrayd` reload them by killing and restarting the daemon on each machine, as follows:

- To kill the daemon, use one of the following commands:
  - On RHEL 7 or SLES 12 systems, type the following:

    ```
    systemctl stop array
    ```

  - On RHEL 6 or SLES 11 systems, type the following:

    ```
    /etc/init.d/array stop
    ```

- To kill and restart the daemon in one operation, use one of the following commands:
  - On RHEL 7 or SLES 12 systems, type the following:

    ```
    systemctl restart array
    ```

  - On RHEL 6 or SLES 11 systems, type the following:

    ```
    /etc/init.d/array restart
    ```

The Array Services daemon on any node can access only the information in the configuration files on that node. One advantage to this design is that you can limit the use of particular nodes. At the same time, though, you need insure that common information is synchronized. "Designing New Array Commands" on page 142 summarizes a way to do this.

Designing Array Services Commands

By default, most Array Services commands run using the user, group, and project ID of either the user that issued the original command or `arraysvcs`. When you add new array commands to `arrayd.conf`, or when you modify existing array commands, always use the most restrictive IDs possible. This practice minimizes trouble if a hostile or careless user were to run that command. Avoid adding commands that run with more powerful IDs, such as user `root` or group `sys`, than the user. If such commands are necessary, analyze them carefully to ensure that an arbitrary user would not be granted any more privileges than expected, much the same as one would analyze a `setuid` program.

The user can invoke arbitrary system commands on single nodes using the `arshell` command. The user can also launch MPI programs that automatically distribute over multiple nodes. However, the only way to launch coordinated system programs on all nodes at once is to use the `array` command. This command does not accept any system command; it only permits execution of commands that the administrator has configured into the Array Services configuration file.

As the administrator, you can define any set of commands that your users need. You have complete control over how any single array node runs a command. For example, the definition can be different on different nodes. A command can simply invoke a standard system command, or, if you define a command as invoking a script, you can make a command arbitrarily complex.

About Substitution Syntax

The `arrayd.conf`(4) man page explains the syntax rules for entries in the configuration files. An important feature of this syntax is the use of several kinds of text substitution by which variable text is substituted into entries when run.

Most of the supported substitutions are used in command entries. These substitutions are performed dynamically each time the `array` command invokes a subcommand. At that time, substitutions insert values that are unique to the invocation of that subcommand. For example, the value `%USER` inserts the user ID of the user who is invoking the `array` command. Such a substitution has meaning only when the command runs.

Substitutions in other configuration entries are performed only once, at the time `arrayd` reads the configuration file. Only environment variable substitution makes sense in these entries. The environment variable values that are substituted are the values inherited by `arrayd` from the script that invokes it, which is as follows:

- On RHEL 7 or SLES 12 systems, the script is as follows:

  ```
  /usr/lib/systemd/system/array.service
  ```

- On RHEL 6 or SLES 11 systems, the script is as follows:

```
/etc/init.d/array
```

About Array Command Operations

When a user runs an `array` command, it has the following format:

`array` [*options*] *subcommand*

The specified *subcommand* operates on nodes as follows:

- If the user does not specify any options, the *subcommand* runs on the whole array.
- If the user specifies the `-l` option, the *subcommand* runs on the local node.
- If the user specifies the `-s` *node* option, the command runs on all nodes that *node* knows about.

  Remember that the destination node can be configured with only a subset of nodes. At each node, `arrayd` searches the configuration file for a `COMMAND` entry with the same name as the subcommand.
- If the user specifies both `-l` and `-s` *node*, the subcommand runs on the specified *node*.

For example, in the following command, `arrayd` processes the `uptime` subcommand on node `tokyo`:

```
array -s tokyo uptime
```

When `arrayd` finds the subcommand to be valid, it distributes the subcommand to every node that is configured in the default array at node `tokyo`.

In the `/etc/array/arrayd.conf` on `tokyo`, the `COMMAND` entry for `uptime` is as follows:

```
command uptime
        invoke /usr/lib/array/auptime %LOCAL
```

The `INVOKE` subentry tells `arrayd` how to run this command. In this case, it run a shell script, `/usr/lib/array/auptime`, and passes one argument, the name of the local node. This command runs on every node, with `%LOCAL` replaced by that node's name.

Command Definition Syntax Summary

The basic set of commands distributed with Array Services resides in `/etc/array/arrayd.conf`. Each `COMMAND` entry is defined using the subentries shown in Table B-1, which the `arrayd.conf`(4) man page also describes.

Table B-1 Subentries of a `COMMAND` Definition

Keyword	Meaning of Following Values
`COMMAND`	The name of the command as the user gives it to `array`.
`INVOKE`	A system command to be run on every node. Specify the full path to the system command. The argument values can be literals, user-supplied arguments, or other substitution values.
`MERGE`	A system command to be run only on the distributing node. Specify the full path to the system command. Its purpose is to gather the streams of output from all nodes and combine them into a single stream.
`USER`	The user ID under which the `INVOKE` and `MERGE` commands run. Typically specified as `USER %USER`, so as to run as the user who invoked `array`.
`GROUP`	The group name under which the `INVOKE` and `MERGE` commands run. Typically specified as `GROUP %GROUP`, so as to run in the group of the user who invoked `array`. For more information, see the `groups`(1) man page.
`PROJECT`	The project under which the `INVOKE` and `MERGE` commands run. Typically specified as `PROJECT %PROJECT`, so as to run in the project of the user who invoked `array`. For more information, see the `projects`(5) man page.
`OPTIONS`	A variety of options to modify this command. For more information, see Table B-3.

As with a shell script, system commands are often composed from a few literal values and many substitution strings. Table B-2 shows the substitutions that are supported, all of which are documented in detail in the `arrayd.conf`(4) man page:

Table B-2 Substitutions Used in a `COMMAND` Definition

Substitution	Replacement Value
`%1..%9`; `%ARG(`*n*`)`; `%ALLARGS`; `%OPTARG(`*n*`)`	Argument tokens from the user's subcommand. `%OPTARG` does not produce an error message if the specified argument is omitted.
`%USER`, `%GROUP`, `%PROJECT`	The effective user ID, effective group ID, and project of the user who invoked `array`.
`%REALUSER`, `%REALGROUP`	The real user ID and real group ID of the user who invoked `array`.
`%ASH`	The internal array session handle (ASH) number under which the `INVOKE` or `MERGE` command is to run. The term *array session* includes all the processes for one application, regardless of where the processes run. Typically, an array session includes the user's login shell and the programs the user started from the login shell. A batch job is an array session.
`%PID(`*ash*`)`	List of process identifier (PID) values for a specified ASH. `%PID(%ASH)` is a common use.
`%ARRAY`	The array name, either default or as given in the `-a` option.
`%LOCAL`	The hostname of the executing node.
`%ORIGIN`	The full domain name of the node where the `array` command ran and where the output is to be viewed.
`%OUTFILE`	List of names of temporary files, each containing the output from one node's `INVOKE` command. Valid only in the `MERGE` subentry.

The `OPTIONS` subentry permits a number of important modifications of the command execution. Table B-3 summarizes these.

Table B-3 Options of the COMMAND Definition

Keyword	Effect on Command
LOCAL	Does not distribute to other nodes. Effectively forces the -l option.
NEWSESSION	Runs the INVOKE command under a newly created ASH. The %ASH in the INVOKE line is the new ASH. The MERGE command runs under the original ASH, and %ASH substitutes as the old ASH in that line.
SETRUID	Sets both the real and the effective user ID from the USER subentry. Typically, USER sets only the effective UID.
SETRGID	Sets both the real and effective group ID from the GROUP subentry. Typically, GROUP sets only the effective GID.
QUIET	Discards the output of INVOKE, unless a MERGE subentry is present. If a MERGE subentry is present, passes INVOKE output to MERGE as usual, and discards the MERGE output.
NOWAIT	Discards the output and returns as soon as the processes are invoked. Does not wait for completion. A MERGE subentry is ineffective.

Configuring Local Options

The LOCAL entry specifies options to arrayd itself. Table B-4 summarizes the most important options.

Table B-4 Subentries of the LOCAL Entry

Subentry	Purpose
DIR	Pathname for the arrayd working directory, which is the initial, current working directory of INVOKE and MERGE commands. The default is /usr/lib/array.
DESTINATION ARRAY	Name of the default array, used when the user omits the -a option. When only one ARRAY entry is specified, it is the default destination.

Subentry	Purpose
USER, GROUP, PROJECT	Default values for COMMAND execution when USER, GROUP, or PROJECT are omitted from the COMMAND definition.
HOSTNAME	Value returned in this node by %LOCAL. Default is the hostname.
PORT	Socket to be used by arrayd.

If you do not supply LOCAL USER, GROUP, and PROJECT values, the default values for USER and GROUP are arraysvcs.

The HOSTNAME entry is needed whenever the hostname(1) command does not return a node name as specified in the ARRAY MACHINE entry. In order to supply a LOCAL HOSTNAME entry unique to each node, each node needs an individualized copy of at least one configuration file.

Designing New Array Commands

The /usr/lib/array/arrayd.conf.template file contains a basic set of commands. Examine this file carefully before defining commands of your own. Any new commands that you design become available to the users of the array system. You can develop new administrative commands, too.

Typically, a new command is defined with an INVOKE subentry that names a script written in sh, csh, or Perl syntax. You can use the substitution values to set up arguments to the script. You use the USER, GROUP, PROJECT, and OPTIONS subentries to establish the execution conditions of the script.

Within the invoked script, you can write any amount of logic to verify and validate the arguments and to run any command sequence. For an example of a script in Perl, see /usr/lib/array/aps, which is invoked by the array ps command.

Note: Perl is a particularly interesting choice for array commands because Perl has native support for socket I/O. In principle at least, you could build a distributed application in Perl in which multiple instances are launched by array and coordinate and exchange data using sockets. Performance would not rival the highly tuned MPI libraries, but development would be simpler.

The following example shows an administrator command called `reinit`, which reinitializes the Array Services configuration file on all nodes at once:

- The shell script in file `/usr/lib/array/arrayd-reinit` reinitializes each Array Services configuration file, on each node, simultaneously. The script is designed for RHEL 7 systems and SLES 12 systems and is as follows:

  ```
  #!/bin/sh
  #################################################################
  # NOTE: The example shown is for illustrative purposes only and #
  # has not been evaluated for use in a production environment.   #
  #################################################################
  # Script to reinitialize arrayd with a new configuration file
  # Usage:  arrayd-reinit <hostname:new-config-file>
  sleep 10                  # Let old arrayd finish distributing
  scp $1 /etc/array/
  systemctl restart array
  exit 0
  ```

  The script uses `rcp` to copy a specified file, presumably a configuration file such as `arrayd.conf`, into `/etc/array`. The script fails if `%USER` is not privileged. Then the script restarts `arrayd` to reread configuration files.

- The following is the command definition:

  ```
  command reinit
  #################################################################
  # NOTE: The example shown is for illustrative purposes only and #
  # has not been evaluated for use in a production environment.   #
  #################################################################
     invoke /usr/lib/array/arrayd-reinit %ORIGIN:%1
     user   %USER
     group  %GROUP
     options nowait   # Exit before restart occurs!
  ```

  The `INVOKE` subentry calls the script shown previously. The `NOWAIT` option prevents the daemon from waiting for the script to finish. The script stops the daemon.

Caution: The preceding example is for illustrative purposes only and has not been evaluated for use in a production environment.

Testing Configuration Changes After Creating New Array Services Commands

The configuration files contain many sections and options. You can use the `ascheck` command to perform a basic sanity check of all configuration files in the array.

After making a change, you can run the `arrayd` command with the `-c` and `-f` options to test an individual configuration file for correct syntax. For example, assume that you added a new command definition to `/etc/array/arrayd.local`. You can type the following command to check its syntax:

```
arrayd -c -f /usr/lib/array/arrayd.local
```

When testing new commands for correct operation, you need to see the warning and error messages produced by the `arrayd` command and by the processes that the `arrayd` command can spawn. Typically, the `stderr` messages from a daemon are not visible.

Procedure B-2 To test configuration changes

1. Notify the array's users that you are able to start an array configuration update.

   Users might experience a lack of response to `ainfo` and `array` commands.

2. Log into one of the nodes as the root user, and type one of the following commands:

   - On RHEL 7.*X* or SLES 12 SP*X* systems, type the following command:

     ```
     # systemctl stop array
     ```

   - On RHEL 6.*X* or SLES 11 SP*X* systems, type the following command:

     ```
     # /etc/init.d/array stop
     ```

3. In one shell window on that node, type the following `arrayd` command:

   ```
   # /usr/sbin/arrayd -n -v
   ```

   The preceding command prevents the `arrayd` command from moving into the background. The command remains attached to the shell terminal.

   Although `arrayd` becomes functional in this mode, it does not refer to the `/etc/sysconfig/array` file, so you need to specify explicitly all command line options, such as the names of nonstandard configuration files.

4. From another shell window on the same node (or on another node), issue `ainfo` and `array` commands to test the new configuration data.

   Observe that diagnostic output appears in the `arrayd` shell window.

5. From the shell window in which you typed the `/usr/sbin/arrayd -n -v` command, type CTRL-c to terminate the `arrayd` daemon.

6. Type one of the following commands to start the `arrayd` daemon:

   - On RHEL 7.*X* or SLES 12 SP*X* systems, type the following command:

     ```
     # systemctl start array
     ```

   - On RHEL 6.*X* or SLES 11 SP*X* systems, type the following command:

     ```
     # /etc/init.d/array start
     ```

Index