SGI MPI and SGI SHMEM User Guide

007–3773–026

New Features in This Manual

This revision adds the following information:

- A new procedure for installing SGI MPI in an alternate location.
- The `MPI_USE_ARRAY` environment variable, which toggles the Array Services software on or off and is designed for use in secure computing environments.
- The `MPI_COLL_IB_OFFLOAD` environment variable, which can be used if your cluster contains Mellanox Fabric Collective Accelerator (FCA) offset software.
- Updates to the default values for the `MPI_IB_PAYLOAD` and `MPI_IB_FAILOVER` environment variables.
- Miscellaneous technical and editorial corrections.

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*.

Contents

Tables

About This Guide

The SGI MPI software package facilitates parallel programming on large systems and computer system clusters. SGI MPI supports both the Message Passing Interface (MPI) standard and the OpenSHMEM standard, as follows:

- The 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).

  This publication describes SGI MPI 1.10, which supports the MPI 3.0 standard. SGI MPI includes significant features that make it the preferred implementation for use on SGI hardware. The following are some of these features:

  - 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 SGI® ICE™ systems.
  - Optimized MPI remote memory access (RMA) one-sided commands.
  - Interoperability with the SHMEM (LIBSMA) programming model.

- 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 SGI SHMEM, which supports OpenSHMEM version 1.1. The SGI 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.

SGI's support for MPI and OpenSHMEM is built on top of the SGI Message Passing Toolkit (MPT). SGI MPT is a high-performance communications middleware software product that runs on SGI's shared memory and cluster supercomputers. On some of these machines, SGI MPT uses SGI Array Services to launch applications. SGI MPT is optimized for all SGI hardware platforms. This document describes SGI MPT 2.12.

Related SGI Publications

The SGI Foundation Software release notes and the SGI Performance Suite release notes contain information about the specific software packages provided in those products. The release notes also list SGI publications that provide information about the products. The release notes are available in the following locations:

- Online at Supportfolio. After you log into Supportfolio, you can access the release notes. The SGI Foundation Software release notes are posted to the following website:

  https://support.sgi.com/content_request/194480/index.html

  The SGI Performance Suite release notes are posted to the following website:

  https://support.sgi.com/content_request/786853/index.html

  Note: You must sign into Supportfolio, at https://support.sgi.com/login, in order for the preceding links to work.

- On the product media. The release notes reside in a text file in the `/docs` directory on the product media. For example, `/docs/SGI-MPI-1.`*x*`-readme.txt`.

- On the system. After installation, the release notes and other product documentation reside in the `/usr/share/doc/packages/`*product* directory.

The *MPInside Reference Guide* describes SGI's MPInside MPI profiling tool.

All SGI software and hardware manuals and can be found in the SGI Technical Publications Library at the following website:

http://docs.sgi.com

SGI creates hardware manuals that are specific to each product line. The hardware documentation typically includes a system architecture overview and describes the major components. It also provides the standard procedures for powering on and powering off the system, basic troubleshooting information, and important safety and regulatory specifications.

The following procedure explains how to retrieve a list of hardware manuals for your system.

Procedure 0-1 To retrieve hardware documentation

1. Type the following URL into the address bar of your browser:

   `docs.sgi.com`

2. In the search box on the Techpubs Library, narrow your search as follows:

   - In the **search** field, type the model of your SGI system.

     For example, type one of the following: **"UV 300"**, **"ICE X"**, **Rackable**.

     Remember to enclose hardware model names in quotation marks (" ") if the hardware model name includes a space character.

   - Check **Search only titles**.
   - Check **Show only 1 hit/book**.
   - Click **search**.

Related Publications From Other Sources

Information about MPI is available from a variety of sources. 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

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in either of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system:

  http://www.sgi.com/support/supportcenters.html

SGI values your comments and will respond to them promptly.

Configuring the SGI Message Passing Toolkit (MPT) on a Standalone SGI UV Computer System

This chapter includes the following topics:

About Configuring SGI MPT

When you installed SGI Performance Suite, you also installed SGI MPT. Before you can run any SGI MPI programs, however, you need to configure the SGI MPT software. The procedures in this chapter explain how to configure SGI MPT.

SGI computers often host several released versions of SGI MPT. This environment provides users with the flexibility they need to develop and run 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 whether your SGI UV computer is partitioned or not, as follows:

- If you have an SGI UV system that is configured as a single system image (SSI), complete the following procedure:

  "Configuring SGI MPT on an SGI UV Computer System (Single System Image)" on page 2

- If you have an SGI UV system that is configured into two or more partitions, complete the following procedure:

  "Configuring SGI MPT on an SGI UV Computer System (Partitioned)" on page 7

Note: This chapter explains how to configure SGI MPT on a standalone SGI UV server.

For information about how to configure SGI MPT on an SGI computing system that is managed by SGI Management Center (SMC), see the following:

SGI Management Center (SMC) Installation and Configuration Guide for Clusters

Configuring SGI MPT on an SGI UV Computer System (Single System Image)

The information in the following procedures explains how to configure SGI MPT on a large, single SGI UV SSI:

Verifying Prerequisites

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

Procedure 1-1 To verify prerequisites

1. As the root user, log into the SGI UV computer.
2. (Conditional) Reboot the computer.

   Perform this step if the SGI UV computer was not rebooted after SGI Performance Suite was installed.

   If you do not know whether the computer has been rebooted, reboot at this time.
3. Verify that you have one of the following operating system software packages installed and configured:

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

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

```
# cat /etc/issue
```

4. Type a series of `cat`(1) commands to verify that the following required products from the SGI Performance Suite 1.10 release are installed:

   - SGI Accelerate
   - SGI MPI

   For example:

```
# cat /etc/sgi-accelerate-release
SGI Performance Suite 1.10, Build 710r19.sles11sp3-1404162103
# cat /etc/sgi-mpi-release
SGI MPI 1.10, Build 710r19.sles11sp3-1404162103
```

5. Proceed to one of the following:

   - "(Optional) Installing the SGI MPT Software Into a Nondefault Working Directory" on page 3, which explains how to configure SGI 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 5, which assumes you want the SGI MPT software to remain in the default installation directory.

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

Perform the procedure in this topic if you want to install SGI 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.

The RPM utility enables you to create, install, and manage relocatable packages. You can install a matched set of SGI MPT RPMs in either a default location or an alternate location. The default location for installing the SGI MPT RPM is `/opt/sgi/mpt/mpt-2.`*rel_level*. To install the SGI MPT RPM in an alternate

location, use the `--relocate` parameter to the `rpm` command. The `--relocate` parameter specifies an alternate base directory for the SGI MPT software installation.

Either `/opt/sgi/mpt/mpt-2.`*rel_level* or both `/opt/sgi/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* 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 SGI MPT software's current location.

  If you install the SGI 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 SGI MPT software.

Procedure 1-2 To install the SGI MPT software in an alternate location

1. Use the `rpm` command to specify an alternate locatiion for the SGI MPT software bundle.

   Example 1. The following example shows how to install SGI MPT in `/usr/local/sgi/mpt/mpt-2.12` rather than in `/opt`, which is the default:

   ```
   # rpm -i --relocate /opt/sgi/mpt/mpt-2.12=/usr/local/sgi/mpt/mpt-2.12 \
   sgi-mpt-*.x86_64.rpm
   ```

   Example 2: The following RHEL example shows how to install the modules, in addition to the total SGI MPT software bundle, to `/usr/local/sgi/mpt/mpt-2.12` and `/usr/local/mod/mpt`:

   ```
   # rpm -i --relocate /opt/sgi/mpt/mpt-2.12=/usr/local/sgi/mpt/mpt-2.12 \
   --relocate /usr/share/Modules/modulefiles/mpt=/usr/local/mod/mpt \
   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 SGI MPT software bundle, to `/usr/local/sgi/mpt/mpt-2.12` and `/usr/local/mod/mpt`:

```
# rpm -i --relocate /opt/sgi/mpt/mpt-2.12=/usr/local/sgi/mpt/mpt-2.12 \
--relocate /usr/share/modules/modulefiles/mpt=/usr/local/mod/mpt \
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-2.12-sgi*.x86_64.rpm
... Relocations: /opt/sgi/mpt/mpt-2.12 /usr/share/modules/modulefiles/mpt
```

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

2. Proceed to the following:

   "Adjusting File Resource Limits" on page 5

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 1-3 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 7

Completing the Configuration

The following procedure explains how to complete the SGI MPT configuration.

Procedure 1-4 To complete the SGI 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 SGI MPT release on this computer.

   Perform this step if you moved the SGI 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-2.12/usr/share/Modules/modulefiles/mpt/mpt-2.12`

   - On SLES platforms:

     `/opt/mpt/mpt-2.12/usr/share/modules/modulefiles/mpt/mpt-2.12`

Configuring SGI MPT on an SGI UV Computer System (Partitioned)

You can configure SGI MPT on an SGI UV computer system that is divided into two or more partitions. Generally, you configure each partition individually, and then you configure the partitions into an array. If you have several partitions, you can use only some of them for the SGI MPT array; you do not have to configure all the partitions into an array.

The information in the following procedures explains how to configure SGI MPT on a partitoned SGI UV server:

- "Verifying Prerequisites" on page 8
- "Configuring the OpenFabrics Enterprise Distribution (OFED) Software" on page 9
- "Adjusting File Resource Limits" on page 10
- "Creating a Directory and Removing the Current Software" on page 12

Verifying Prerequisites

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

Procedure 1-5 To verify prerequisites

1. (Conditional) Make sure that an NFS share is available.

   An NFS share is needed only if you plan to move the SGI MPT installation to a nondefault location on two or more partitions of an SGI UV computer.

2. As the root user, log into one of the partitions on the partitioned SGI UV computer.

3. (Conditional) Reboot the partition.

   Perform this step if the SGI UV computer was not rebooted after SGI Performance Suite was installed.

   If you do not know whether the computer has been rebooted, reboot at this time.

4. Verify that you have one of the following operating system software packages installed and configured:

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

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

   # **`cat /etc/issue`**

5. Type a series of `cat`(1) commands to verify that the following products from the SGI Performance Suite are installed:

- SGI Accelerate
- SGI MPI

For example:

```
# cat /etc/sgi-accelerate-release
SGI Performance Suite 1.10, Build 710r19.sles11sp3-1404162103
# cat /etc/sgi-mpi-release
SGI MPI 1.10, Build 710r19.sles11sp3-1404162103
```

6. Proceed to the following:

"Configuring the OpenFabrics Enterprise Distribution (OFED) Software" on page 9

Configuring the OpenFabrics Enterprise Distribution (OFED) Software

All SGI UV computers are equipped with NUMAlink technology. Some SGI UV computers are also equipped with InfiniBand hardware, which uses OFED software. The procedure in this topic explains how to test for the presence of InfiniBand hardware and how to specify the number of queue pairs (QPs) for the OFED software.

If you are installing a kernel-based virtual machine (KVM), be aware that neither SGI, nor RHEL, nor SLES support InfiniBand hardware within a KVM.

The following procedure explains how to adjust the `log_num_qp` parameter.

Procedure 1-6 To specify the `log_num_qp` parameter

1. Type the following command to determine whether this partition is equipped with InfiniBand hardware:

   ```
   # lspci | grep Mellanox
   ```

   Note whether the command returns information similar to the following, which informs you of the presence of InfiniBand hardware:

   ```
   03:00.0 Network controller: Mellanox Technologies MT27500 Family [ConnectX-3]
   ```

   If the `lspci` command returns nothing, this partition is not connected to InfiniBand hardware. You do not need to perform the rest of this procedure. Proceed to:

"Adjusting File Resource Limits" on page 10

2. Type one of the following commands to determine whether the OFED software is installed on this partition:

   - On SLES platforms, type the following:

     # **zypper info -t pattern ofed**

   - On RHEL platforms, type the following:

     # **yum grouplist "Infiniband Support"**

   The operating system packages include OFED by default.

3. Use a text editor to open file `/etc/modprobe.d/libmlx4.conf`.

4. Add a line similar to the following to file `/etc/modprobe.d/libmlx4.conf`:

   ```
   options mlx4_core log_num_qp=21
   ```

   The default maximum number of queue pairs is 2^{18} (262144).

   The `log_num_qp` parameter defines the $\log_2$ of the number of queue pairs (QPs). This step specifies the maximum number of queue pairs (QPs) for SHMEM applications. If the `log_num_qp` parameter is set to a number that is too low, the system generates the following message:

   ```
   MPT Warning: IB failed to create a QP
   ```

5. Save and close the file.

6. Proceed to the following:

   "Adjusting File Resource Limits" on page 10

Adjusting File Resource Limits

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

Procedure 1-7 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:

    "Creating a Directory and Removing the Current Software" on page 12

Creating a Directory and Removing the Current Software

The following procedure explains how to create an NFS-mounted directory and remove the SGI MPT software that currently resides on each partition.

Procedure 1-8 To create a directory and remove the existing software

1. Familiarize yourself with the current SGI MPT working directory structure, and create a directory for the SGI MPT software you want to configure at this time.

   By default, the product installs into the `/opt/sgi/mpt/mpt-2.12` directory. Make a plan for the nondefault structure at this time. In a partitioned environment, you install SGI MPT into a central NFS-mounted location.

   For example, use the `mkdir`(1) command to create the following alternate directory:

   ```
   # mkdir -p /nfsmount/sgimpi/mpt-2.12
   ```

   This documentation uses directory `/nfsmount/sgimpi/mpt-2.12` as an example nondefault working directory, configures SGI MPT 2.12 in that directory, and uses that example directory in the remaining steps of this configuration procedure.

2. Type the following command, and verify that all SGI MPT packages are in the default installation directory at this time:

   ```
   # rpm -qa | grep sgi-mpt
   ```

   Scan the `rpm` command output, and make sure that the following three SGI MPT packages appear:

   ```
   sgi-mpt-shmem-2.12-sgi710r6.sles11sp3
   sgi-mpt-2.12-sgi710r6.sles11sp3
   sgi-mpt-fs-2.12-sgi710r6.sles11sp3
   ```

3. Use a series of `rpm` commands to remove the SGI MPT packages from the default installation directory.

Your goal is to remove only the following packages:

- `sgi-mpt-shmem-`*release_number*
- `sgi-mpt-`*release_number*
- `sgi-mpt-fs-`*release_number*

The `rpm` command you need to use has the following format:

`rpm -e --nodeps` *package_name*

For *package_name*, specify the name of the package in the default directory at this time.

For example:

```
# rpm -e --nodeps sgi-mpt-shmem-2.10-sgi710r6.sles11sp3
# rpm -e --nodeps sgi-mpt-2.10-sgi710r6.sles11sp3
# rpm -e --nodeps sgi-mpt-fs-2.10-sgi710r6.sles11sp3
```

4. Proceed to one of the following:
   - "(Optional) Configuring the MUNGE Security Software" on page 13
   - "Updating Other Partitions or Continuing the Configuration" on page 14

(Optional) Configuring the MUNGE Security Software

Perform the procedure in this topic if you want to configure the MUNGE security software.

Array Services provides authentication services, but MUNGE provides additional authentication and security for Array Services operations. If you want to configure MUNGE, you need to configure it on each partition that you want to include in the array.

Procedure 1-9 To configure MUNGE

1. Verify that the partition is connected to a good time source, such as an NTP server.

   MUNGE depends on time synchronization across all nodes in the array.

2. Type one of the following commands to start the MUNGE installation, and respond to the installation prompts:

- On Red Hat Enterprise Linux platforms, type the following command:

  # **yum install munge**

- On SUSE Linux Enterprise Server platforms, type the following command:

  # **zypper install munge**

For more information about how to install MUNGE, see the SGI MPI release notes.

3. Type the following command to restart MUNGE:

   # **service munge restart**

4. Type the following command to verify the existance of a MUNGE key on the partition:

   # **md5sum /etc/munge/munge.key**

5. (Conditional) Copy one partition's MUNGE key to all of the partitions.

   Perform this step if this is the last partition that you need to configure.

   Immediately after you install MUNGE, each partition should have a unique key. When you run the partitions as an array, however, each partition needs to have the same key. After you have MUNGE installed on all the partitions that you want to include in the array, select one partition, and copy that partition's MUNGE key to file `/etc/munge/munge.key` on each of the other partitions.

6. Proceed to the following:

   "Updating Other Partitions or Continuing the Configuration" on page 14

Updating Other Partitions or Continuing the Configuration

At this point, at least one of the partitions on your SGI UV computer is configured correctly for SGI MPT.

The following procedure explains how to proceed.

Procedure 1-10 To assess progress

1. (Conditional) Configure additional partitions.

Make sure that you completed all the preceding procedures on all of the partitions that you want to include in the array before you continue with the procedures that follow.

If you want to include additional partitions in the array, repeat the following procedures on the additional partitions:

- "Verifying Prerequisites" on page 8
- "Configuring the OpenFabrics Enterprise Distribution (OFED) Software" on page 9
- "Adjusting File Resource Limits" on page 10
- "Creating a Directory and Removing the Current Software" on page 12
- "(Optional) Configuring the MUNGE Security Software" on page 13

2. Proceed to the following:

   "Configuring Array Services" on page 15

Configuring Array Services

SGI MPI depends on Array Services for several capabilities. During the configuration, your goal is to specify the partitions that you want to include in the array and to distribute the configuration files to each partition.

Table 1-1 on page 15 lists the documentation resources that contain additional configuration information.

Table 1-1 Array Configuration Resources

Topic	Documentation Resource
Advanced configuration information	"Additional Array Configuration Information" on page 91
Array Services overview	`array_services`(5)
Configuration file format	`arrayd.conf`(4) `/usr/lib/array/arrayd.conf.template`

Topic	Documentation Resource
Configuration file validator	`ascheck`(1)
Array Services configuration utility	`arrayconfig`(8)

The procedure in this topic uses the `arrayconfig`(8) command to specify SGI UV partitions for an array and to update the Array Services configuration files on each host.

Procedure 1-11 To configure Array Services for multiple partitions

1. (Optional) Synchronize and distribute secure shell (SSH) keys to each partition you want to include in the array.

   If you have SSH keys configured, you can complete work on one partition and log into the next without typing passwords each time. When you configure Array Services, it might be convenient for you if SSH is configured in each partition.

2. Plan the authentication method you want the Array Services software to use.

   Your choices are as follows:

   - `munge`. Specify `munge` if you configured the MUNGE software in the following procedure:

     "(Optional) Configuring the MUNGE Security Software" on page 13

   - `none`. Disables all authentication.
   - `noremote`. Disallows requests from remote systems.
   - `simple` (default). Generates hostname/key pairs by using the OpenSSL `rand` command, 64–bit values (if available), or by using `$RANDOM` Bash facilities.

   For more information about the authentication levels, see `arrayd.auth`(5).

3. Log in as root to the partition to which you expect users to log in when they want to run SGI MPI jobs.

   Run the `arrayconfig`(1M) command from the partition to which you expect users to run their SGI MPI jobs.

4. Use the `arrayconfig`(1M) command to specify the partitions that you want to include in the array.

The `arrayconfig` command configures the `/etc/array/arrayd.conf` and `/etc/array/arrayd.auth` files on each partition.

Type the `arrayconfig`(1M) command in the following format:

`/usr/sbin/arrayconfig -a` *`arrayname`* `-A` *`method`* `-D -m` *`hostname1 hostname2`* `...`

For *arrayname*, type a name for the array. For example, `sgicluster`. The default is `default`.

For *method*, type one of the following authentication methods: `munge`, `none`, `noremote`, or `simple` (default). For information about the authentication methods, see the `arrayd.auth`(4) man page.

For each *hostname*, specify the hostnames of the partitions upon which you installed the SGI Message Passing Toolkit (MPT) software. That is, for *hostname1*, *hostname2*, and so on, specify the hostnames of the partitions that you want to include in the array.

5. (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 `/usr/lib/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    # SGI 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.

6. Proceed to the following:

   "Enabling Cross-partition NUMAlink MPI Communication and Restarting Services" on page 18

Note: If you have trouble with the Array Services configuration, examine the Array Services manual configuration procedure in the following topic:

Appendix B, "Configuring Array Services Manually"

Enabling Cross-partition NUMAlink MPI Communication and Restarting Services

When you configure a large SGI UV system into two or more NUMAlink-connected partitions, the partitions act as separate, clustered hosts. The hardware supports efficient and flexible global memory access for cross-partition communication on such systems, but to enable this access, you need to load special kernel modules. If you do not enable cross-partition NUMAlink MPI communication at this time, users might receive the following message when they run an application:

```
MPT ERROR from do_cross_gets/xpmem_get, rc = -1, errno = 22
```

Depending on your operating system, perform one of the following procedures to ensure that the kernel modules load every time the system boots:

- "Enabling Cross-partition Communication and Restarting Services (RHEL)" on page 18
- "Enabling Cross-partition Communication and Restarting Services (SLES)" on page 19

Enabling Cross-partition Communication and Restarting Services (RHEL)

The following procedure explains how to load the kernel modules on one partition that hosts a RHEL operating system.

Procedure 1-12 To load the kernel modules at boot

1. As the root user, log into one of the partitions upon which you installed the SGI MPT software.
2. Type the following command:

   # **echo "modprobe xpc" >> /etc/sysconfig/modules/sgi-propack.modules**

3. Save and close the file.

4. Type one of the following command sequences:

   # **reboot -f**

   Or

   ```
   # modprobe xpc
   # modprobe xpmem
   # /etc/init.d/procset restart
   # /etc/init.d/array restart
   ```

5. Repeat the preceding steps on the other partitions in the array.
6. Proceed to the following:

   "Completing the Configuration" on page 20

Enabling Cross-partition Communication and Restarting Services (SLES)

The following procedure explains how to load the kernel modules on one partition that hosts a SLES operating system.

Procedure 1-13 To load the kernel modules at boot

1. As the root user, log into one of the partitions upon which you installed the SGI MPT software.
2. Use a text editor to open file `/etc/sysconfig/kernel`.
3. Within file `/etc/sysconfig/kernel`, search for the line that begins with `MODULES_LOADED_ON_BOOT`.
4. Add `xpc` to the list of modules that are loaded at boot time.
5. Save and close the file.
6. Type one of the following command sequences:

   # **reboot -f**

   Or

   ```
   # modprobe xpc
   # modprobe xpmem
   # /etc/init.d/procset restart
   # /etc/init.d/array restart
   ```

7. Repeat the preceding steps on the other partitions in the array.
8. Proceed to the following:

   "Completing the Configuration" on page 20

Completing the Configuration

The following procedure explains how to complete the SGI MPT configuration.

Procedure 1-14 To complete the SGI 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 SGI MPT release on this computer.

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

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

   - On RHEL platforms:

     `/nfsmount/sgimpi/mpt-2.12/usr/share/Modules/modulefiles/mpt/2.12`

   - On SLES platforms:

     `/nfsmount/sgimpi/mpt-2.12/usr/share/modules/modulefiles/mpt/2.12`

Getting Started

This chapter includes the following topics:

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.

Running MPI Jobs

The following procedure explains how to run an MPI application when the SGI Message Passing Toolkit (MPT) software is installed in an alternate location.

Procedure 2-1 To run jobs when SGI MPT is installed in an alternate location

1. Determine the directory into which the SGI MPT software is installed.
2. Type one of the following commands to compile your program.

   `mpif90 -I /`*`install_path`*`/usr/include `*`file`*`.f -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 SGI 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
```

3. (Conditional) Ensure that the program can find the SGI MPT library routines when it runs.

   You can use either site-specific library modules, or you can specify the library path on the command line before you run the program.

   If you use module files, set the library path in the `mpt` module file. Sample module files reside in the following locations:

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

   If you want to specify the library path as a command, type the following command:

   ```
   % setenv LD_LIBRARY_PATH /install_path/usr/lib
   ```

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

   Example 1. The following command assumes that the libraries reside in `/tmp`:

   ```
   % 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 SGI MPT in an NFS-mounted file system:

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

4. Type the following command to link the program:

   ```
   % 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 SGI MPT software searches incorrectly for the libraries in the default location of `/usr/lib/libmpi.so`.

5. Use the `mpirun`(1) command to run the program.

   For example, assume that you installed the SGI MPT software on an NFS-mounted file system (`/data/nfs`) in the alternate directory `/tmp`. Type the following command to run the program:

   ```
   % /data/nfs/bin/mpirun -v -a myarray hostA,hostB -np 1 a.out
   ```

Compiling and Linking MPI Programs

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, type the following command:

```
% module load mpt
```

When the SGI 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 SGI 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
  ```

Using `mpirun` to Launch an MPI Application

The `mpirun`(1) command starts an MPI application. For a complete specification of the command line syntax, see the `mpirun`(1) man page. This section summarizes the procedures for launching an MPI application.

The following topics explain how to use `mpirun` to launch an application:

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.

The following example 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. In the following example, one copy of `prog1` and five copies of `prog2` are run on the local host, and both executable files use shared memory:

```
% mpirun -np 1 prog1 : 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 Array Services software. For a list of the available nodes on systems running Array Services software, see the `/usr/lib/array/arrayd.conf` file.

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 in this section show various ways to start an application that consists of multiple MPI executable files on multiple hosts.

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

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

Example 2. When specifying multiple hosts, you can omit the `-np` option and list the number of processes directly. The following example launches ten instances of `fred` on three hosts. `fred` has two input arguments.

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

Example 3. The following example launches an MPI application on different hosts with different numbers of processes and executable files:

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

Using MPI Spawn Functions to Launch an Application

To use the MPI RMA process creation functions `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple`, use the `-up` option on the `mpirun` command to specify the universe size. For example, the following command starts three instances of the `mtest` MPI application in a universe of size 10:

```
% mpirun -up 10 -np 3 mtest
```

By using one of the preceding MPI spawn functions, `mtest` can start up to seven more MPI processes.

When running MPI applications on partitioned SGI UV systems that use the `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` functions, you might need to explicitly specify the partitions on which additional MPI processes can be launched. For more information, see the `mpirun`(1) man page.

Running MPI Jobs with a Workload Manager

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. SGI 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 27
- "Torque" on page 28

- "Simple Linux Utility for Resource Management (SLURM)" on page 29

PBS Professional

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

The following topics explain how to configure PBS job scripts to run MPI applications:

- "Specifying Computing Resources" on page 27
- "Running the MPI Application" on page 27
- "Examples" on page 28

Specifying Computing Resources

Within a script, use the `-l` option on a `#PBS` directive line. These lines have the following format:

`#PBS -l select=`*processes*`:ncpus=`*threads*`[:`*other_options*`]`

For *processes*, specify the total number of MPI processes in the job.

For *threads*, specify the number of OpenMP threads per process. For purely MPI jobs, specify `1`.

For more information about resource allocation options, see the `pbs_resources`(7) man page from the PBS Professional software distribution.

Running the MPI Application

Use the `mpiexec_mpt` command included in SGI MPT.

The `mpiexec_mpt` command is a wrapper script that assembles the correct host list and corresponding `mpirun` command before it runs the assembled `mpirun` command. The format is as follows:

`mpiexec_mpt` [**`-n`** ***processes***] **`./a.out`**

For *processes*, specify the total number of MPI processes in the application. Use this syntax on both a single-host and clustered systems. For more information, see the `mpiexec_mpt`(1) man page.

Examples

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 2-1 Running an MPI application with 512 Processes

To run an application with 512 processes, include the following in the directive file:

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

Example 2-2 Running an MPI application with 512 Processes and Four OpenMP Threads per Process

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 43.

PBS Professional includes an `mpiexec`(1) command that enables you to run SGI MPI applications. PBS Professional's command does not support the same set of extended options that the SGI `mpiexec_mpt`(1) supports.

Torque

When running Torque, SGI recommends the SGI MPT `mpiexec_mpt`(1) command to launch SGI MPT MPI jobs.

The basic syntax is as follows:

```
mpiexec_mpt -n P ./a.out
```

For *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. See the `mpiexec_mpt`(1) man page for more details.

The `mpiexec_mpt` command has a `-tv` option for use by SGI 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 43.

Simple Linux Utility for Resource Management (SLURM)

SGI MPI is adapted for use with the SLURM workload manager. If you want to use SGI MPI with SLURM, use the SLURM `pmi2` MPI plug-in. SGI 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

Compiling and Running SHMEM Applications

To compile SHMEM programs with a GNU compiler, choose one of the following commands:

```
% g++ compute.C -lsma -lmpi
% gcc compute.c -lsma -lmpi
```

To compile SHMEM programs with the Intel compiler, use the following commands:

```
% icc compute.C -lsma -lmpi
% icc compute.c -lsma -lmpi
% ifort compute.f -lsma -lmpi
```

Use `mpirun` to launch SHMEM applications. The `NPES` variable has no effect on SHMEM programs. To request the desired number of processes to launch, set the `-np` option on `mpirun`.

The SHMEM programming model supports both single-host SHMEM applications and SHMEM applications that span multiple partitions. To launch a SHMEM application on more than one partition, use the multiple host `mpirun` syntax, as follows:

```
% mpirun hostA, hostB -np 16 ./shmem_app
```

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

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 2-2 To configure system settings for huge pages

1. Type the following command to make sure that the current SGI 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 SGI 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. This makes single-copy MPI sends much more efficient when using the SGI UV global reference unit (GRU) for MPI messaging.

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 SGI 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 SGI MPI in an SELinux Environment (SGI UV Systems, RHEL Platforms Only)

SGI supports Security-Enhanced Linux (SELinux) on SGI UV computer systems that run the Red Hat Enterprise Linux (RHEL) operating system. If you want to use SGI MPI on an SGI UV system with SELinux Multi-Level Security (MLS) or Multi-Category Security (MCS) at a sensitivity level above S0, verify the following:

- SELinux for SGI UV systems is configured. For configuration information, see the following manual:

  SGI UV System Software Installation and Configuration Guide

- The `MPI_USE_ARRAY` environment variable is set 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 SGI MPI with security software, contact SGI technical support.

Programming With SGI MPI

This chapter includes the following topics:

About Programming With SGI 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 SGI MPI implementation upon normal job termination. Error handling and characteristics of abnormal job termination are also described.

This section includes the following topics:

• "MPI_Finalize and Connect Processes" on page 34

MPI_Abort

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

In addition, the `SIGURG` and `SIGUSR1` 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. To use this feature, the MPI program must have a signal handler that catches `SIGURG` or `SIGUSR1`. When the `SIGURG` or `SIGUSR1` signals are 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 36 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 SGI 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 37.

Multithreaded Programming

SGI 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 with `-lmpi_mt` instead of `-lmpi`.

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

Interoperability with the SHMEM programming model

You can mix SHMEM and MPI message passing in the same program. The application must be linked with both the SHMEM and MPI libraries. Start with an MPI program that calls `MPI_Init` and `MPI_Finalize`.

When you add SHMEM calls, the PE numbers are equal to the MPI rank numbers in `MPI_COMM_WORLD`. Do not call `start_pes()` in a mixed MPI and SHMEM program.

When running the application across a cluster and using SHMEM functions, some MPI processes might not be able to communicate with certain other MPI processes. You can use the `shmem_pe_accessible` and `shmem_addr_accessible` functions to determine whether a SHMEM call can be used to access data residing in another MPI process. Because the SHMEM 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 remote memory access (RMA) spawn-related functions.

SHMEM get and put functions are thread safe. SHMEM collective and synchronization functions are not thread safe unless different threads use different `pSync` and `pWork` arrays.

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

Miscellaneous SGI MPI Features

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

- `stdin`/`stdout`/`stderr`.

  In this implementation, `stdin` is enabled for only those MPI processes with 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 remote memory access (RMA) 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 SGI MPI optimization features.

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

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.

Most 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. On SGI UV systems, barriers and reductions have been optimized to use the SGI GRU hardware accelerator. 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 40.

Note: Collectives are optimized across partitions by using the XPMEM driver which is explained in Chapter 7, "Run-time Tuning". The collectives (except `MPI_Barrier`) try to use single-copy by default for large transfers unless `MPI_DEFAULT_SINGLE_COPY_OFF` is specified.

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 SGI 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.

Avoiding 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.

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 35, 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. In the case of a partitioned system, the processes can reside on any of the partitions.
- The sender data must be globally accessible by the receiver. The SGI 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 55.

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 53.

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 53.

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 SGI 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 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 SGI Message Passing Toolkit (MPT) `mpiexec_mpt`(1) command has a `-tv` option for use by SGI 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 SGI MPT MPI job with TotalView without a workload manager, type the following:

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

Example 2. To run an SGI 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 SGI 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 PerfBoost

This chapter includes the following topics:

About PerfBoost

SGI PerfBoost uses a wrapper library to run applications compiled against other MPI implementations under the SGI Message Passing Toolkit (MPT) product on SGI platforms. This chapter describes how to use PerfBoost software.

Note: PerfBoost does not support the MPI C++ API.

Using PerfBoost

The following procedure explains how to use PerfBoost with an SGI 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 SGI 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 SGI 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`
HP-MPI	`-pmpi`
Intel MPI	`-impi`
OpenMPI	`-ompi`
MPICH1	`-mpich`
MPICH2	`-impi`
MVAPICH2	`-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

SGI 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 SGI 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

The SGI Message Passing Toolkit (MPT) supports the Berkeley Lab Checkpoint/Restart (BLCR) checkpoint/restart implementation. This implementation allows applications to periodically save a copy of their state. Applications can resume from that point if the application crashes or 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, 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 https://ftg.lbl.gov/projects/CheckpointRestart.

BLCR Installation

To use checkpoint/restart with SGI MPT, BLCR must first be installed. This requires installing the `blcr-`, `blcr-libs-`, and `blcr-kmp-` RPMs. BLCR must then be enabled by root, as follows:

```
% chkconfig blcr on
```

BLCR uses a kernel module which must be built against the specific kernel that the operating system is running. In the case that the kernel module fails to load, it must be rebuilt and installed. Install the `blcr-` SRPM. 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.

Using BLCR with SGI MPT

To enable checkpoint/restart within SGI MPT, `mpirun` or `mpiexec_mpt` must be passed the `-cpr` option, for example:

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

To checkpoint a job, use the `mpt_checkpoint` command on the same host where `mpirun` is running. `mpt_checkpoint` needs to be passed the PID of `mpirun` and a name with which you want to prefix all the checkpoint files. For example:

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

This will create 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
```

The job may be restarted on a different set of hosts but there must be the same number of hosts and each host must have the same number of ranks as the corresponding host in the original run of the job.

Run-time Tuning

This chapter discusses ways in which the user can tune the run-time environment to improve the performance of an MPI message passing application on SGI computers. None of these ways involve application code changes. This chapter covers the following topics:

Reducing Run-time Variability

One of the most common problems with optimizing message passing codes on large shared memory computers is achieving 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 and leads to variations in the results and 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 called the kernel node in the following discussion). 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 the kernel node, you can avoid this effect.

Additionally, by restricting system daemons to run on the kernel node, you can also deliver an additional percentage of each application CPU to the user.

- Avoid interference with other applications. You can use cpusets to address this problem also. You can use cpusets to effectively partition a large, distributed memory host in a fashion that minimizes interactions between jobs running concurrently on the system. See the *SGI Cpuset Software Guide* for information about cpusets.
- On a quiet, dedicated system, you can use `dplace` or the `MPI_DSM_CPULIST` shell variable to improve run-time performance repeatability. These approaches are not as suitable for shared, nondedicated systems.
- Use a workload manager; for example, Platform LSF from Platform Computing Corporation or PBS Professional from Altair Engineering, Inc. These workload managers use cpusets to avoid oversubscribing the system and possible interference between applications.

Tuning MPI Buffer Resources

By default, the SGI 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 73.

For information about buffering considerations when running an MPI job across multiple hosts, see "Tuning for Running Applications Across Multiple Hosts" on page 59.

For information about the programming implications of message buffering, see "Buffering" on page 35.

Avoiding Message Buffering – Enabling Single Copy

For message transfers between MPI processes within the same host or transfers using devices that support remote direct memory access (RDMA), such as InfiniBand, it is possible under certain conditions to avoid the need to buffer messages. Because many MPI applications are written assuming infinite buffering, the use of this unbuffered approach is not enabled by default for `MPI_Send`. This section describes how to activate this mechanism 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 takes advantage of the XPMEM driver to support single copy message transfers between two processes within the same host or across partitions.

Enabling single copy transfers may result in better performance, since this technique improves MPI's bandwidth. However, single copy transfers may introduce additional synchronization points, which can reduce application performance in some cases.

The threshold for message lengths beyond which MPI attempts to use this single copy method is specified by the MPI_BUFFER_MAX shell variable. Its value should be set to the message length in bytes beyond which the single copy method should be tried. 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, whether between processes residing on the same host or across partitions, the data is copied using a bcopy process. A bcopy process is also used to transfer data between two different executable files on the same host or two different executable files across partitions. 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. Under certain circumstances, however, you might want to modify this default behavior. The easiest way to do this is by setting one or more MPI placement shell variables. Several of the most commonly used of these variables are described in the following sections. For a complete listing of memory placement related shell variables, see the MPI(1) man page.

MPI_DSM_CPULIST

The MPI_DSM_CPULIST shell variable allows you to manually select 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.

This setting is treated as a comma and/or hyphen delineated ordered list that specifies a mapping of MPI processes to CPUs. If running across multiple hosts, the per host components of the CPU list are delineated by colons. Within hyphen delineated lists CPU striding may be specified by placing /# after the list where # is the stride distance.

Note: This feature should not be used with MPI applications that use either of the MPI remote memory access (RMA) spawn related functions.

Examples of settings are as follows:

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. 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. When running 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 will be 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

Use the MPI_DSM_DISTRIBUTE shell variable to ensure that each MPI process gets a physical CPU and memory on the node to which it was assigned. MPI_DSM_DISTRIBUTE assigns MPI ranks, starting at logical CPU 0 and incrementing, until all ranks have been placed.

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

Using dplace for Memory Placement

The dplace tool offers another means of specifying the placement of MPI processes within a distributed memory host. The dplace tool and MPI interoperate to allow MPI to better manage placement of certain shared memory data structures when dplace is used to place the MPI job.

For instructions on how to use dplace with MPI, see the dplace(1) man page and 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 using 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 ensures that all local, non-shared memory is allocated on the same memory node as the processor referencing it. For all systems, it 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 SGI Message Passing Toolkit (MPT) provides the omplace(1) command to help with the placement of OpenMP threads within an MPI program. The omplace

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 detailed syntax and a number of examples, see the `omplace`(1) man page. For more information on `dplace`, see the `dplace`(1) man page. For information on using cpusets, see the *SGI Cpuset Software Guide.* For more information on using `dplace`, see the *Linux Application Tuning Guide.*

Example 7-1 How to Run a Hybrid MPI/OpenMP Application

Here is an example of how to run 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 a hybrid MPI/OpenMP job as shown in Example 7-1 on page 59, use the following resource allocation specification:

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

And use the following `mpiexec` command with the above example:

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

For more information about running SGI MPT programs with PBS, see"Running MPI Jobs with a Workload Manager" on page 26 .

Tuning for Running Applications Across Multiple Hosts

When you are running an MPI application across a cluster of hosts, there are additional run-time environment settings and configurations that you can consider when trying to improve application performance.

Systems can use the XPMEM interconnect to cluster hosts as partitioned systems, or use the InfiniBand interconnect or TCP/IP as the multihost interconnect.

When launched as a distributed application, MPI probes for these interconnects at job startup. For details of launching a distributed application, see "Launching a Distributed Application" on page 25. When a high performance interconnect is detected, 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 60 specifies the order in which MPI probes for available interconnects.

Table 7-1 Inquiry Order for Available Interconnects

Interconnect	Default Order of Selection	Environment Variable to Require Use
XPMEM	1	`MPI_USE_XPMEM`
InfiniBand	2	`MPI_USE_IB`
InfiniBand Unreliable Datagram	3	`MPI_USE_UD`
TCP/IP	4	`MPI_USE_TCP`

The third column of Table 7-1 on page 60 also indicates the environment variable you can set to pick a particular interconnect other than the default.

In general, to insure the best performance of the application, you should 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.

When using the InfiniBand interconnect, SGI MPT applications may not execute a `fork`() or `system`() call. The InfiniBand driver produces undefined results when an SGI MPT process using InfiniBand forks.

MPI_USE_IB

Requires the MPI library to use the InfiniBand driver as the interconnect when running across multiple hosts or running with multiple binaries. SGI 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()s may occur after InfiniBand resources have been allocated. Default: Not set

MPI_IB_RAILS

When this is set to 1 and the MPI library uses the InfiniBand driver as the inter-host interconnect, SGI MPT will send its InfiniBand traffic over the first fabric that it detects. If this is set to 2, the library will try to make use of multiple available separate InfiniBand fabrics and split its traffic across them. If the separate InfiniBand fabrics do not have unique subnet IDs, then the `rail-config` utility is required. It must be run by the system administrator to enable the library to correctly use the separate fabrics. Default: 1 on all SGI UV systems.

MPI_IB_SINGLE_COPY_BUFFER_MAX

When MPI transfers data over InfiniBand, if the size of the cumulative data is greater than this value then MPI will attempt to send the data directly between the processes's buffers and not through intermediate buffers inside the MPI library. Default: 32767

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 running an MPI application across a cluster of hosts using the InfiniBand interconnect, there are additional run-time environmental settings that you can consider to improve application performance, as follows:

MPI_COLL_IB_OFFLOAD

Enabled or disables the Mellanox fabric collectives accelerator (FCA) offload. If FCA offload is configured on your cluster, set `MPI_COLL_IB_OFFLOAD=true`. The default is `MPI_COLL_IB_OFFLOAD=false`.

You might also need to set Mellanox's `fca_ib_dev_name` and `fca_ib_port_num` environment variables to the name and port of the host channel adapter (HCA) you want to use. For example, `fca_ib_dev_name=mlx4_0 and fca_ib_port_num=1`.

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 may be prodigious. When the number of ranks is at least this value, the MPI library creates InfiniBand connections on a demand basis. The default is 1024 ranks.

MPI_IB_PAYLOAD

When the MPI library uses the InfiniBand fabric, it allocates some amount of 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_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 that wait period. The time spent is equal to 4 * 2 ^ `MPI_IB_TIMEOUT` microseconds. By default, the variable is set to 18.

MPI_IB_FAILOVER

When the MPI library uses InfiniBand and this variable is set, and an InfiniBand transmission error occurs, SGI MPT will try to restart the connection to the other rank. It will handle a number of errors of this type between any pair of ranks equal to the value of this variable. By default, the variable is set to 32.

MPI_NUM_MEMORY_REGIONS

For zero-copy sends over the InfiniBand interconnect, SGI MPT keeps a cache of application data buffers registered for these transfers. This environmental variable controls the size of the cache. It is 8 by default and can be any value between 0 and 32. If the application rarely reuses data buffers, it may make sense to set this value to 0 to avoid cache trashing.

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.

MPI on SGI UV Systems

Note: This section does not apply to SGI UV 10 systems or SGI UV 20 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 SGI UV Hub application-specific integrated circuits (ASICs). The 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. Each SGI UV Hub normally connects to two sockets. All communication between the sockets and the UV Hub uses Intel QuickPath Interconnect (QPI) channels. The SGI UV 1000 series Hub has four NUMAlink 5 ports that connect with the NUMAlink 5 interconnect fabric.

On SGI UV 2000 series systems, the UV Hub board assembly has a 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 four NUMAlink 6 ports that connect with the NUMAlink 6 interconnect fabric.

The 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. For more information on the SGI UV hub, SGI UV compute blades, QPI, and NUMAlink 5, or NUMAlink 6, see the *SGI Altix UV 1000 System User's Guide*, the *SGI Altix UV 100 System User's Guide* or *SGI UV 2000 System User's Guide*, respectively.

When MPI communicates between processes, two transfer methods are possible on an SGI UV system:

- By use of shared memory
- By use of the global reference unit (GRU), part of the SGI UV Hub ASIC

MPI chooses the method depending on internal heuristics, the type of MPI communication that is involved, and some user-tunable variables. When using the GRU to transfer data and messages, the MPI library uses the GRU resources it allocates via the GRU resource allocator, which divides up the available GRU resources. It fairly allocates buffer space and control blocks between the logical processors being used by the MPI job.

General Considerations

Running MPI jobs optimally on SGI UV systems is not very difficult. It is best to pin MPI processes to CPUs and isolate multiple MPI jobs onto different sets of sockets and Hubs, and this is usually achieved by configuring 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 `MPI_DSM_CPULIST` (see "`MPI_DSM_CPULIST`" on page 56), `omplace`(1), and `dplace`(1).

Job Performance Types

The MPI library chooses buffer sizes and communication algorithms in an attempt to deliver the best performance automatically to a wide variety of MPI applications. However, applications have different performance profiles and bottlenecks, and so user tuning may be of help in improving performance. Here are some application performance types and ways that MPI performance may be improved for them:

- 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 may 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 tells the number of physical and logical processors and if **Hyperthreading** is **ON** or **OFF** and how shared processors are paired (towards the bottom of the command's output).

If an MPI job uses only half of the available logical CPUs, set `GRU_RESOURCE_FACTOR` to 2 so that the MPI processes can utilize all the available GRU resources on a Hub rather than reserving some of them for the idle HyperThreads. For more information about GRU resource tuning, see the `gru_resource`(3) man page.

- MPI large message bandwidth is important.

  Some programs transfer large messages via the `MPI_Send` function. To switch on the use of unbuffered, single copy transport in these cases you can set `MPI_BUFFER_MAX` to 0. See the `MPI`(1) man page for more details.

- MPI small or near messages are very frequent.

  For small fabric hop counts, shared memory message delivery is faster than GRU messages. To deliver all messages within an SGI UV host via shared memory, set `MPI_SHARED_NEIGHBORHOOD` to "`host`". See the `MPI`(1) man page for more details.

Other ccNUMA Performance Issues

MPI application processes normally perform best if their local memory is allocated on the socket assigned to execute it. This cannot happen if memory on that socket is exhausted by the application or by other system consumption, for example, file buffer cache. Use the `nodeinfo`(1) command to view memory consumption on the nodes assigned to your job and use `bcfree`(1) to clear out excessive file buffer cache. PBS Professional workload manager installations can be configured to issue `bcfree` commands in the job prologue. For more information, see PBS Professional documentation and the `bcfree`(1) man page.

Suspending MPI Jobs

MPI software from SGI can internally use the XPMEM kernel module to provide direct access to data on remote partitions and to provide single copy operations to local data. Any pages used by these operations are prevented from paging by the XPMEM kernel module. If an administrator needs to temporarily suspend a 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 a MPI application which is using the XPMEM kernel module will have a /proc/xpmem/*pid* file associated with it. The number of pages owned by this process which are prevented from paging by XPMEM can be displayed by concatenating the /proc/xpmem/*pid* file, for example:

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

To unpin the pages for use by other processes, the administrator must first suspend all the processes in the application. The pages can then be unpinned by echoing any value into the /proc/xpmem/*pid* file, for example:

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

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

When the MPI application is resumed, the XPMEM kernel module will prevent these pages from paging as they are referenced by the application.

MPI Performance Profiling

SGI includes profiling support in the `libmpi.so` library. Profiling support replaces all `MPI_`*Xxx* prototypes and function names with `PMPI_`*Xxx* entry points.

This chapter describes the `perfcatch` utility used to profile the performance of an MPI program and other tools that can be used for profiling MPI applications. It covers the following topics:

Overview of `perfcatch` Utility

The `perfcatch` utility runs an MPI program with a wrapper profiling library that prints MPI call profiling information to a summary file upon MPI program completion. This MPI profiling result file is called `MPI_PROFILING_STATS`, by default (see " `MPI_PROFILING_STATS` Results File Example" on page 68). It is created in the current working directory of the MPI process with rank 0.

Using the `perfcatch` Utility

The syntax of the `perfcatch` utility is, as follows:

```
perfcatch [-v | -vofed | -i] cmd args
```

The `perfcatch` utility accepts the following options:

No option	Supports the SGI Message Passing Toolkit (MPT)
`-v`	Supports MPI
`-vofed`	Supports OFED MPI

`-i`	Supports Intel MPI

To use `perfcatch` with an SGI Message Passing Toolkit MPI program, insert the `perfcatch` command in front of the executable name, as the following examples show:

- **`mpirun -np 64 perfcatch a.out arg1`**
- **`mpirun 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 MPI profiling result file has a summary statistics section followed by a rank-by-rank profiling information section. The summary statistics section reports some overall statistics, including 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.

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

An example `MPI_PROFILING_STATS` results file is, as follows:

```
============================================================
PERFCATCHER version 22
(C) Copyright SGI.  This library may only be used
on SGI 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
```

MPI Performance Profiling Environment Variables

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.
MPI_PROFILING_STATS_FILE	Specifies the file where MPI profiling results are written. If not

specified, the file MPI_PROFILING_STATS is written.

Profiling MPI Applications

This section 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 Performance Co-Pilot and various third-party products. In this section, you can learn how to use these various tools with MPI applications. It covers the following topics:

Profiling Interface

You can write your own profiling 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 is actual output 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 being 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
...
```

MPI Internal Statistics

MPI keeps track of certain resource utilization statistics. These can be used to determine potential performance problems caused by lack of MPI message buffers and other MPI internal resources.

To turn on the displaying of 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 finer grain measurements. If the `MPI_STATS_FILE` variable is set, when the program completes, the internal statistics will be written to the file specified by this variable. For information about these MPI extensions, see the `mpi_stats` man page.

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 53.

Third-party Products

Two third-party tools that you can use with the SGI MPI implementation are Vampir from Pallas (`www.pallas.com`) and Jumpshot, which is part of the MPICH distribution. Both of these tools are effective for small, short-duration MPI jobs. However, the trace files these tools generate can be enormous for longer running or highly parallel jobs. This causes a program to run more slowly, but the larger problems is that the tools to analyze the data are often overwhelmed by the amount of data.

Troubleshooting and Frequently Asked Questions

This chapter provides answers to some common problems that users encounter when thay start to use SGI MPI and provides answers to other frequently asked questions. It covers the following topics:

What are some things I can try to figure out why `mpirun` is failing?

Here are some things to investigate:

- Look in `/var/log/messages` for any suspicious errors or warnings. For example, if your application tries to pull in 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 to run your program without `mpirun`. You will get the "`mpirun must be used to launch all MPI applications`" message, along with any dynamic link errors that might not be displayed when the program is started with `mpirun`.

  As a last resort, setting the environment variable `LD_DEBUG` to `all` will display a set of messages for each symbol that `rld` resolves. This produces a lot of output, but should help you find the cause of the link error.
- Be sure that you are setting your remote directory properly. By default, `mpirun` attempts to place your processes on all machines into the directory that has the same name as `$PWD`. This should be the common case, but sometimes different functionality is required. For more information, see the section on `$MPI_DIR` and/or the `-dir` option in the `mpirun` man page.
- If you are using a relative pathname for your application, be sure that it appears in `$PATH`. In particular, `mpirun` will not look in '.' for your application unless '.' appears in `$PATH`.
- Run `/usr/sbin/ascheck` to verify that your array is configured correctly.
- Use the `mpirun -verbose` option to verify that you are running the version of MPI that you think you are running.
- Be very careful when setting MPI environment variables from within your `.cshrc` or `.login` files, because these will override any settings that you might later set from within your shell (due to the fact 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 under a Kerberos environment, you may experience unpredictable results because currently, `mpirun` is unable to pass tokens. For example, in some cases, if you use `telnet` to connect to a host and then try to run `mpirun` on that host, it fails. But if you instead use `rsh` to connect to the host, `mpirun` succeeds. (This might be because `telnet` is kerberized but `rsh` is not.) At any rate, if you are running under such conditions, you will definitely want to talk to the local administrators about the proper way to launch MPI jobs.
- Look in `/tmp/.arraysvcs` on all machines you are using. In some cases, you might find an `errlog` file that may be helpful.

- You can increase the verbosity of the Array Services daemon (`arrayd`) using the `-v` option to generate more debugging information. For more information, see the `arrayd`(8) man page.
- Check error messages in `/var/run/arraysvcs`.

My code runs correctly until it reaches `MPI_Finalize()` and then it hangs.

This is almost always caused by `send` or `recv` requests that are either unmatched or not completed. An unmatched request is any blocking `send` for which a corresponding `recv` 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()`. You can fix such codes easily by inserting a call to `MPI_Request_free()` immediately after all such `isend` operations, or by adding a call to `MPI_Wait()` at a later place in the code, prior to the point at which the send buffer must be reused.

My hybrid code (using OpenMP) stalls 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 "Compiling and Linking MPI Programs" on page 23.

I keep getting error messages about `MPI_REQUEST_MAX` being too small.

There are two types of cases in which the MPI library reports an error concerning `MPI_REQUEST_MAX`. The error reported by the MPI library distinguishes these.

```
MPI has run out of unexpected request entries;
the current allocation level is: XXXXXX
```

The program is sending so many unexpected large messages (greater than 64 bytes) to a process that internal limits in the MPI library have been exceeded. The options here

are to increase the number of allowable requests via the MPI_REQUEST_MAX shell variable, or to modify the application.

```
MPI has run out of request entries;
the current allocation level is: MPI_REQUEST_MAX = XXXXX
```

You might have an application problem. You almost certainly are calling MPI_Isend() or MPI_Irecv() and not completing or freeing your request objects. You need to use MPI_Request_free(), as described in the previous section.

I am not seeing stdout and/or stderr output from my MPI application.

All stdout and stderr is line-buffered, which means that mpirun 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 can I get the SGI Message Passing Toolkit (MPT) software to install on my machine?

SGI MPT RPMs are included in the SGI Performance Suite releases. In addition, you can obtain SGI MPT RPMs from the following SGI Support website, under the **Downloads** link:

http://support.sgi.com

Where can I find more information about the SHMEM programming model?

See the intro_shmem(3) man page.

The ps(1) command says my memory use (SIZE) is higher than expected.

At MPI job start-up, MPI calls the SHMEM library to cross-map all user static memory on all MPI processes 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 being used, not the amount of memory being 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 `MPI: could not run executable` mean?

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

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

How do I combine MPI with *insert favorite tool here*?

In general, the rule to follow is to run mpirun on your tool and then the tool on your application. Do not try to run the tool on mpirun. Also, because of the way that mpirun sets up stdio, seeing 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 mpirun. Of course, not all tools (for example, dplace) support such an option. However, it is usually possible to make it work by wrapping a shell script around the tool and having the script do the redirection, as in the following example:

```
> cat myscript
     #!/bin/sh
     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
```

Why do I see “stack traceback” information when my MPI job aborts?

More information can be found in the MPI(1) man page in descriptions of the MPI_COREDUMP and MPI_COREDUMP_DEBUGGER environment variables.

Array Services

This chapter includes the following topics:

About Array Services

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

The array can consist of the following:

- Multiple single system images (SSIs) on an SGI UV system
- Multiple compute nodes plus a service node on an SGI ICE or SGI ICE X 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, login and password management, 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.

- An array configuration database. The database describes the array configuration and provides reference information for array daemons and user programs. Each node hosts a copy of the array configuration database.

- Commands, libraries, and utilities such as `ainfo`(1), `arshell`(1), and others.

The Message Passing Interface (MPI) of SGI MPI uses Array Services to launch parallel applications.

SGI includes MUNGE software in the SGI MPI software distribution. 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, which is at the following location:

http://munge.googlecode.com/

The Array Services package requires that the process sets service be installed and running. This package is provided in the `sgi-procset` RPM. You can type the following commands to verify that the process sets service is installed and running:

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

Retrieving the Array Services Release Notes

The following procedure explains how to find the array services release note information.

Procedure 10-1 To retrieve Array Services release note information

1. Type the following command to retrieve the location of the Array Services release notes:

   ```
   # rpm -qi sgi-arraysvcs
   /usr/share/doc/sgi-arraysvcs-3.7/README.relnotes
   ```

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

Managing Local Processes

Each UNIX process has a *process identifier* (PID), a number that identifies that process within the node where it runs. It is important to realize that a PID is local to the node; so it is possible to have processes in different nodes using the same PID numbers.

Within a node, processes can be logically grouped in *process groups*. A process group is composed of a parent process together with all the processes that it creates. Each process group has a *process group identifier* (PGID). Like a PID, a PGID is defined locally to that node, and there is no guarantee of uniqueness across the array.

Monitoring Local Processes and System Usage

You query the status of processes using the system command `ps`. To generate a full list of all processes on a local system, use a command such as the following:

```
ps -elfj
```

You can monitor the activity of processes using the command `top` for an ASCII display in a terminal window.

Scheduling and Killing Local Processes

You can schedule commands to run at specific times using the `at` command. You can kill or stop processes using the `kill` command. To destroy the process with PID 13032, use a command such as the following:

```
kill -KILL 13032
```

Summary of Local Process Management Commands

Table 10-1 on page 84 summarizes information about local process management.

Table 10-1 Information Sources: Local Process Management

Topic	Man Page
Process ID and process group	`intro`(2)
Listing and monitoring processes	`ps`(1), `top`(1)
Running programs at low priority	`nice`(1), `batch`(1)
Running programs at a scheduled time	`at`(1)
Terminating a process	`kill`(1)

Using Array Services Commands

When an application starts processes on more than one node, the PID and PGID are no longer adequate to manage the application. The Array Services commands enable you to view the entire array and to control the processes on multinode programs.

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. Table 10-2 on page 84 shows the commands that are common to Array Services operations.

Table 10-2 Common Array Services Commands

Topic	Man Page
Array Services Overview	`array_services`(5)
`ainfo` command	`ainfo`(1)
`array` command	`array`(1) or `arrayd.conf`(4)
`arshell` command	`arshell`(1)
`newsess` command	`newsess`(1)

About Array Sessions

Array Services is composed of a daemon–a background process that is started at boot time in every node–and a set of commands such as `ainfo`(1). The commands call on the daemon process in each node to get the information they need.

One concept that is basic to Array Services is the *array session*, which is a term for all the processes of one application, wherever they may execute. Normally, your login shell, with the programs you start from it, constitutes an array session. A batch job is an array session, and you can create a new shell with a new array session identity.

About Names of Arrays and Nodes

Each node is a server, and as such each node has a hostname. An array system as a whole has a name, too. In most installations there is only a single array, and you never need to specify which array you mean. However, it is possible to have multiple arrays available on a network, and you can direct Array Services commands to a specific array.

About Authentication Keys

It is possible for the array administrator to establish an authentication code, which is a 64-bit number, for all or some of the nodes in an array. There can be a single authentication code number for each node. Your system administrator can tell you if this is necessary.

When authentication keys are implemented, you need to specify the authentication key as the argument to the `-s` option on the command line of each Array Services command. The code applies to any command entered at that node or addressed to that node.

Array Services Commands

The Array Services package includes an array daemon, an array configuration database, and several commands. Some utilities enable you to retrieve information about the array. Other utilities let the administrator query and manipulate distributed array applications. The Array Services commands are as follows:

Command	Purpose
`ainfo` command	Queries the array configuration database. Retrieves information about processes.
`array` command	Runs a specified command on one or more nodes. Commands are predefined by the administrator in the configuration database.
`arshell` command	Starts a command remotely on a different 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) commands accept a common set of options plus some command-specific options. Table 10-3 on page 86 summarizes the common options. The default values of some options are set by environment variables.

Table 10-3 Array Services Command Option Summary

Option	Used In	Description
`-a` *array*	`ainfo`, `array`	Specify a particular array when more than one is accessible.
`-D`	`ainfo`, `array`, `arshell`	Send commands to other nodes directly, rather than through `arrayd` daemon.
`-F`	`ainfo`, `array`, `arshell`	Forward commands to other nodes through the `arrayd` daemon.
`-Kl` *number*	`ainfo`, `array`	Authentication key for the local node. This is a 64-bit number.
`-Kr` *number*	`ainfo`, `array`	Authentication key for the remote node. This is a 64-bit number.
`-l` . This letter is a lowercase letter “L”, for “local”.	`ainfo`, `array`	Execute in context of the destination node, not necessarily the current node.

Option	Used In	Description
`-l` *port*	`ainfo`, `array`, `arshell`	Nonstandard port number of array daemon.
`-s` *hostname*	`ainfo`, `array`	Specify a destination node.

Specifying a Single Node

The `-l` and `-s` options work together. The `-l` option restricts the scope of a command to the node where the command is executed. 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 scope of a query command is all nodes of the array. The `-s` option directs the command to be executed on a specified node of the array. These options work together in query commands 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`.

Common Environment Variables

The Array Services commands depend on environment variables to define default values for the less-common command options. These variables are summarized in Table 10-4.

Table 10-4 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

Any user of an array system can use Array Services commands to check the hardware components and the software workload of the array. The commands needed are `ainfo` and `array`.

Learning Array Names

If your network includes more than one array system, you can use `ainfo arrays` at one array node to list all the array names that are configured, as in the following example.

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

```
    IDENT 0x655e
```

Array names are configured into the array database by the administrator. Different arrays might know different sets of other array names.

Learning Node Names

You can use `ainfo machines` to learn the names and some features of all nodes in the current array, as in the following example.

```
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
```

In this example, the `-b` option of `ainfo` is used to get a concise display.

Learning Node Features

You can use `ainfo nodeinfo` to request detailed information about one or all nodes in the array. To get information about the local node, use `ainfo -l nodeinfo`. However, to get information about only a particular other node, for example node `tokyo`, use `-l` and `-s`, as in the following example:

```
homegrown 181% ainfo -s tokyo -l nodeinfo
Node information for server on machine "tokyo"
MACHINE tokyo
    VERSION  1.2
    8 PROCESSOR BOARDS
        BOARD: TYPE 15   SPEED 190
            CPU:   TYPE 9   REVISION 2.4
            FPU:   TYPE 9   REVISION 0.0
...
    16 IP INTERFACES  HOSTNAME tokyo   HOSTID 0xc01a5035
        DEVICE  et0    NETWORK    150.166.39.0    ADDRESS   150.166.39.39  UP
        DEVICE atm0    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
        DEVICE atm1    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
...
    0 GRAPHICS INTERFACES
    MEMORY
```

```
512 MB MAIN MEMORY
INTERLEAVE 4
```

The preceding example has been edited for brevity.

If the `-l` option is omitted, the destination node will return information about every node that it knows.

Learning User Names and Workload

The system commands `who`(1), `top`(1), and `uptime`(1) are commonly used to get information about users and workload on one server. The `array`(1) command offers array-wide equivalents to these commands.

Learning User Names

To get the names of all users logged in to the whole array, use `array who`. To learn the names of users logged in to a particular node, for example `tokyo`, use `-l` and `-s`, as in the following example:

```
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 example has been edited for brevity and security.

Learning Workload

Two variants of the `array` command return workload information. The array-wide equivalent of `uptime` is `array uptime`, as follows:

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

The command `array top` lists the processes that are currently using the most CPU time. The output identifies each process by its internal array session handle (ASH) value. The following is example output:

```
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 `-l` and `-s` options can be used to select data about a single node, as usual.

Additional Array Configuration Information

The system administrator has to initialize the array configuration database, a file that is used by the Array Services daemon in executing almost every `ainfo` and `array` command.

Security Considerations for Standard Array Services

The array services daemon, `arrayd`(1M), runs as root. As with other system services, if it is configured carelessly it is possible for arbitrary and possibly unauthorized user to disrupt or even damage a running system.

By default, most array commands are executed using the user, group, and project ID of either the user that issued the original command, or `arraysvcs`. When adding new array commands to `arrayd.conf`, or modifying existing ones, always use the most restrictive IDs possible in order to minimize 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.

In the default array services configuration, the `arrayd` daemon allows all the local requests to access `arrayd` but not the remote requests. In order to let the remote requests access the `arrayd`, the `AUTHENTICATION` parameter needs to be set to `NONE` in the `/usr/lib/array/arrayd.auth` file. By default it is set to `NOREMOTE`. When the `AUTHENTICATION` parameter is set to `NONE`, the `arrayd` daemon assumes that a remote user will accurately identify itself when making a request. 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`. This should be adequate for systems that are behind a network firewall or otherwise protected from hostile attack, and in which all the users inside the firewall are presumed to be non-hostile. On systems for which this is not the case, because they are attached to a public network or because individual machines cannot be trusted, the Array Services `AUTHENTICATION` parameter should be set to `NOREMOTE`. When `AUTHENTICATION` is set to `NONE`, all requests from remote systems are authenticated using a mechanism that involves private keys that are known only to the super-users on the local and remote systems. Requests originating on systems that do not have these private keys are rejected. For more details, see the section on authentication information in the `arrayd.conf`(4) man page.

The `arrayd` daemon does not support mapping 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 if necessary to prevent access to an array by machines using a different namespace.

About the Uses of the Configuration Files

The configuration files are read by the Array Services daemon when it starts. Typically, the daemon starts in each node during the system startup. You can also run the daemon from a command line in order to check the syntax of the configuration files.

The configuration files contain the following data, all of which is needed by `ainfo` and `array`:

- The names of array systems, including the current array but also any other arrays on which a user could run an Array Services command. `ainfo` reports this information.
- The names and types of the nodes in each named array, especially the hostnames that would be used in an Array Services command. `ainfo` reports this information.
- The authentication keys, if any, that must be used with Array Services commands. The `-Kl` and `-Kr` command options use this information. For more information, see "Array Services Commands" on page 85.
- The commands that are valid with the `array` command.

About Configuration File Format and Contents

A configuration file is a readable text file. The file contains entries of the following four types, which are detailed in later topics.

Array definition	Describes this array and other known arrays, including array names and the node names and types.
Command definition	Specifies the usage and operation of a command that can be invoked through the `array` command.
Authentication	Specifies authentication numbers that must be used to access the array.
Local option	Options that modify the operation of the other entries or `arrayd`.

Blank lines, white space, and comment lines beginning with a pound character (#) can be used freely for readability. Entries can be in any order in any of the files read by `arrayd`.

Besides punctuation, entries are formed with a keyword-based syntax. Keyword recognition is not case-sensitive; however keywords are shown in uppercase in this text and in the man page. The entries are primarily formed from keywords, numbers, and quoted strings, as detailed in the man page `arrayd.conf`(4).

Loading Configuration Data

The Array Services daemon, `arrayd`, can take one or more filenames as arguments. It reads them all, and treats them like logical continuations. In effect, it concatenates them. If no filenames are specified, it reads `/usr/lib/array/arrayd.conf` and `/usr/lib/array/arrayd.auth`. A different set of files, and any other `arrayd` command-line options, can be written into the file `/etc/config/arrayd.options`, which is read by the startup script that launches `arrayd` at boot time.

Since configuration data can be stored in two or more files, you can combine different strategies, for example:

- One file can have different access permissions than another. Typically, `/usr/lib/array/arrayd.conf` is world-readable and contains the available `array` commands, while `/usr/lib/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 in 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 the daemon and restarting it in each machine. The script `/etc/init.d/array` supports this operation:

To kill daemon, execute this command:

```
/etc/init.d/array stop
```

To kill and restart the daemon in one operation; perform the following command:

```
/etc/init.d/array restart
```

The Array Services daemon in any node knows only the information in the configuration files available in that node. This can be an advantage, in that you can limit the use of particular nodes; but it does require that you take pains to keep common information synchronized. "Designing New Array Commands" on page 102 summarizes an automated way to do this.

About Substitution Syntax

The `arrayd.conf`(4) man page explains the syntax rules for forming 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 they are executed.

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 no meaning except during execution of a command.

Substitutions in other configuration entries are performed only once, at the time the configuration file is read by `arrayd`. 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 `/etc/init.d/array`.

Testing Configuration Changes

The configuration files contain many sections and options. The Array Services command `ascheck` performs a basic sanity check of all configuration files in the array.

After making a change, you can test an individual configuration file for correct syntax by executing `arrayd` as a command with the `-c` and `-f` options. For example, suppose you have just added a new command definition to `/usr/lib/array/arrayd.local`. You can check its syntax with the following command:

```
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 `arrayd` and processes that it may spawn. The `stderr` messages from a daemon are not normally visible. You can make them visible by the following procedure:

1. On one node, kill the daemon, as follows:

   ```
   # /etc/init.d/array stop
   ```

2. In one shell window on that node, start `arrayd` with the options `-n -v`, as follows:

   # **`/usr/sbin/arrayd -n -v`**

   Instead of moving into the background, it remains attached to the shell terminal.

   Note: Although `arrayd` becomes functional in this mode, it does not refer to `/etc/config/arrayd.options`, so you need to specify explicitly all command-line options, such as the names of nonstandard configuration files.

   ---

3. From another shell window on the same or other nodes, issue `ainfo` and `array` commands to test the new configuration data. Diagnostic output appears in the `arrayd` shell window.

4. Terminate `arrayd` and use the following command to restart it as a daemon:

   # **`/usr/sbin/arrayd -v`**

During steps 1, 2, and 4, the test node might not respond to `ainfo` and `array` commands, so warn users that the array is in test mode.

Specifying Arrayname and Machine Names

The following lines are a simple example of an array definition within an `arrayd.conf` file:

```
array simple
        machine congo
        machine niger
        machine nile
```

The array name `simple` is the value the user must specify in the `-a` option. For more information, see "Array Services Commands" on page 85.

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 101.

Specifying IP Addresses and Ports

The simple `machine` subentries shown in the example are based on the assumption that the hostname is the same as the machine's name to domain name services (DNS).

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 shows how to use the `port` subentry to specify that `arrayd` in a particular machine uses 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 of attributes:

```
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.

Configuring Array Commands

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 database.

You can define any set of commands that your users need. You have complete control over how any single array node executes a command. For example, the definition can be different in different nodes. A command can simply invoke a standard system command, or, since you can define a command as invoking a script, you can make a command arbitrarily complex.

Operation of Array Commands

When a user invokes the `array` command, the subcommand and its arguments are processed by the destination node specified by `-s`. Unless the `-l` option was given, that daemon also distributes the subcommand and its arguments to all other array nodes that it knows about. Remember that the destination node might be configured with only a subset of nodes. At each node, `arrayd` searches the configuration database for a `COMMAND` entry with the same name as the `array` subcommand.

In the following example, the subcommand `uptime` is processed by `arrayd` in node `tokyo`:

```
array -s tokyo uptime
```

When `arrayd` finds the subcommand valid, it distributes it to every node that is configured in the default array at node `tokyo`.

The `COMMAND` entry for `uptime` is distributed in this form. You can read it in the file `/usr/lib/array/arrayd.conf`.

```
command uptime         # Display uptime/load of all nodes in array
        invoke /usr/lib/array/auptime %LOCAL
```

The `INVOKE` subentry tells `arrayd` how to execute this command. In this case, it executes a shell script `/usr/lib/array/auptime` , passing it one argument, the name of the local node. This command is executed at every node, with `%LOCAL` replaced by that node's name.

Summary of Command Definition Syntax

Look at the basic set of commands distributed with Array Services. This command set resides in `/usr/lib/array/arrayd.conf`. Each `COMMAND` entry is defined using the subentries shown in Table 10-5, which the `arrayd.conf`(4) man page also describes.

Table 10-5 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 executed on every node. The argument values can be literals, or arguments given by the user, or other substitution values.
MERGE	A system command to be executed only on the distributing node, 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. Usually given 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. Usually given 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. Usually given 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 10-7.

The system commands called by INVOKE and MERGE must be specified as full pathnames because arrayd has no defined execution path. As with a shell script, these system commands are often composed from a few literal values and many substitution strings. The substitutions that are supported, all of which are documented in detail in the arrayd.conf(4) man page, are summarized in Table 10-6.

Table 10-6 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.
%PID(*ash*)	List of 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 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 10-7 summarizes these.

Table 10-7 Options of the COMMAND Definition

Keyword	Effect on Command
LOCAL	Do not distribute to other nodes. Effectively forces the -l option.
NEWSESSION	Execute the INVOKE command under a newly created ASH. %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.

Keyword	Effect on Command
SETRUID	Set both the real and effective user ID from the USER subentry. Typically, USER only sets the effective UID.
SETRGID	Set both the real and effective group ID from the GROUP subentry. Typically, GROUP sets only the effective GID.
QUIET	Discard the output of INVOKE, unless a MERGE subentry is given. If a MERGE subentry is given, pass INVOKE output to MERGE as usual, and discard the MERGE output.
NOWAIT	Discard the output and return as soon as the processes are invoked. Do not wait for completion. A MERGE subentry is ineffective.

Configuring Local Options

The LOCAL entry specifies options to arrayd itself. The most important options are summarized in Table 10-8.

Table 10-8 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 given, it is the default destination.
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 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

A basic set of commands is distributed in the file /usr/lib/array/arrayd.conf.template. You should examine this file carefully before defining commands of your own. You can define new commands which then become available to the users of the array system.

Typically, a new command will be defined with an INVOKE subentry that names a script written in sh, csh, or Perl syntax. You 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 execute any sequence of commands. 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, since 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 administrator has need for distributed applications as well, since the configuration files are distributed over the array. Here is an example of a distributed command to reinitialize the Array Services database on all nodes at once. The script to be executed at each node, called /usr/lib/array/arrayd-reinit would read as follows:

```
#!/bin/sh
# Script to reinitialize arrayd with a new configuration file
# Usage:  arrayd-reinit <hostname:new-config-file>
sleep 10      # Let old arrayd finish distributing
rcp $1 /usr/lib/array/
/etc/init.d/array restart
```

```
exit 0
```

The script uses `rcp` to copy a specified file, presumably a configuration file such as `arrayd.conf`, into `/usr/lib/array`. This fails if `%USER` is not privileged. Then the script restarts `arrayd` to reread configuration files.

The command definition is as follows:

```
command reinit
   invoke /usr/lib/array/arrayd-reinit %ORIGIN:%1
   user   %USER
   group  %GROUP
   options nowait   # Exit before restart occurs!
```

The `INVOKE` subentry calls the restart script shown above. The `NOWAIT` option prevents the daemon's waiting for the script to finish because the script kills the daemon.

Guidelines for Using SGI MPT on a Virtual Machine Within an SGI UV Computer System

This appendix section includes the following topics:

About SGI 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 SGI 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 system, 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 105
- "Adjusting SGI UV Virtual Machine System Settings" on page 106

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 SGI MPI Programs From Within a Virtual Machine (VM)" on page 108

Installing Software Within the Virtual Machine (VM)

The following procedure explains the software that you need to install in the VM in order for 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 SGI 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 SGI does not support SR-IOV or other alternatives to the distribution-supplied OFED.

3. (Optional) Install the SGI Foundation Software into the VM.

   For installation information, see the *SGI UV System Software Installation and Configuration Guide*.

4. Install the SGI Performance Suite software into the VM.

   For installation information, see the SGI Performance Suite release notes.

5. Install SGI MPT into the VM.

   For installation information, see Chapter 2, "Getting Started" on page 21.

Adjusting SGI UV Virtual Machine System Settings

For best performance, SGI 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 SGI MPI Programs From Within a Virtual Machine (VM)

The following list explains some of the differences between running an MPI or SHMEM program on native SGI hardware 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, hardware performance counters, and global reference units (GRUs). In addition, processor-specific performance diagnostics are limited.

  If your application uses hardware technologies that are not specific to SGI systems, 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 system needs to be run on a VM that was configured with topology that mimics the SGI computer system. 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.

  SGI 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.

Configuring Array Services Manually

This appendix contains the following topics:

About Configuring Array Services Manually

The SGI MPT configuration procedures explain how to configure Array Services in an automated way on SGI UV partitioned systems and on SGI ICE X systems. The information in this appendix section explains how to configure Array Services in a manual way, which allows you to make customizations if necessary.

Configuring Array Services on Multiple Partitions or Hosts

The following procedure explains how to configure Array Services to run on multiple hosts, such as exist on an SGI UV partitioned system or an SGI ICE X system.

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 SGI 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 SGI MPI release notes.

3. Open file `/usr/lib/array/arrayd.conf` with a text editor.
4. Edit the `/usr/lib/array/arrayd.conf` file to list the machines in your cluster.

   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 `/usr/lib/array/arrayd.conf`.
6. Use a text editor to open file `/usr/lib/array/arrayd.auth`.

7. Search for the string `AUTHENTICATION NOREMOTE`, and insert a # character in column 1 to comment out the line.

8. 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.

   For information about the authentication methods, see the `arrayd.auth`(4) man page.

9. Save and close file `/usr/lib/array/arrayd.auth`.

10. (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 `/usr/lib/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    # SGI 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.

11. Type the following comand to restart Array Services:

    ```
    /etc/init.d/array restart
    ```

12. Repeat the preceding steps on the other hosts or copy the `/usr/lib/array/arrayd.conf` and `/usr/lib/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.

Index