Message Passing Toolkit (MPT) User Guide

007–3773–021

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.

Contents

Tables

About This Manual

This publication describes the SGI® implementation of the Message Passing Interface (MPI) 2.07.

MPI consists of a library, which contains both normal and profiling entry points, and commands that support the MPI interface. MPI is a component of the SGI Message Passing Toolkit (MPT).

MPT is a software package that supports parallel programming on large systems and clusters of computer systems through a technique known as *message passing*. Systems running MPI applications must also be running Array Services software version 3.7 or later. For more information on Array Services, see the *Linux Resource Administration Guide*.

Related Publications and Other Sources

Information about MPI is available from a variety of sources. Some of the following sources include pointers to other resources. For information about the MPI standard, see the following:

- The MPI Standard, which is documented online at the following website:

  http://www.mcs.anl.gov/mpi

- *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.

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

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

- 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

- The `comp.parallel.mpi` newsgroup.

The following SGI manuals also describe MPI:

- The *Linux Resource Administration Guide*, which provides information on Array Services.
- The *MPInside Reference Guide*, which describes the SGI MPInside MPI profiling tool.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at the following website:

  http://docs.sgi.com

  Manuals in various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- You can also view man pages by typing `man` *title* on a command line.

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 any 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.

- Send mail to the following address:

  SGI
  Technical Publications
  46600 Landing Parkway
  Fremont, CA 94538

SGI values your comments and will respond to them promptly.

Introduction

Message Passing Toolkit (MPT) is a software package that supports interprocess data exchange for applications that use concurrent, cooperating processes on a single host or on multiple hosts. Data exchange is done through message passing, which is the use of library calls to request data delivery from one process to another or between groups of processes.

The MPT package contains the following components and the appropriate accompanying documentation:

- Message Passing Interface (MPI). MPI is a standard specification for a message passing interface, allowing portable message passing programs in Fortran and C languages. MPI is the dominant programming model on large scale HPC systems and clusters today. MPT supports version 2.2 of the MPI standard specification.
- The SHMEM™ programming model. SHMEM 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. The SGI SHMEM programming model is the basis for the OpenSHMEM™ programming model specification which is being developed by the Open Source Software Solutions multi-vendor working group.

SGI MPT is highly optimized for all SGI hardware platforms. The *SGI Performance Suite 1.x Start Here* lists all current SGI software and hardware manuals and can be found on the SGI Technical Publications Library at the following website:

http://docs.sgi.com

This chapter provides an overview of the MPI software that is included in the toolkit. It includes descriptions of the MPI standard compliance, the basic components of MPI, and the basic features of MPI. Subsequent chapters address the following topics:

MPI Overview

MPI was created by the Message Passing Interface Forum (MPIF). MPIF is not sanctioned or supported by any official standards organization. Its goal was to develop a widely used standard for writing message passing programs.

SGI supports implementations of MPI that are released as part of the Message Passing Toolkit. The MPI standard is documented online at the following website:

http://www.mcs.anl.gov/mpi

MPI 2.2 Standard Compliance

The SGI MPI implementation complies with the MPI 2.2 standard.

MPI Components

The MPI library is provided as a dynamic shared object (DSO). A DSO is a file that ends in `.so`. The basic components that are necessary for using MPI are the `libmpi.so` library, the include files, and the `mpirun` command.

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

SGI MPI Features

The SGI MPI implementation offers a number of 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

- Multi-rail InfiniBand support, which takes full advantage of the multiple InfiniBand fabrics available on SGI® ICE™ systems
- Use of hardware fetch operations (`fetchops`), where available, for fast synchronization and lower latency for short messages
- Optimized MPI-2 one-sided commands
- Interoperability with the SHMEM (LIBSMA) programming model
- High-performance communication support for partitioned systems

Installing and Configuring the Message Passing Toolkit (MPT)

This chapter explains how to install the MPT software and how to configure your system to use it effectively. The information in this chapter also appears on your system in the MPT directory. To access the information online, change to the following directory:

`/opt/sgi/mpt/mpt-`*mpt_rel*`/doc/README.relnotes`

For *mpt_rel*, specify the MPT release number. For example, `2.07`.

This chapter includes the following topics:

Verifying Prerequisites

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

Procedure 2-1 To verify prerequisites

1. Verify that you have 25 Mbytes of free disk space in the installation directory.

   If you want to install the MPT software into the default installation directory, make sure there are 25 Mbytes of free space in `/opt/sgi/mpt`.

2. Verify that you have the following software installed and configured:

   - Red Hat Enterprise Linux (RHEL) 6.2, RHEL 6.3, or SUSE Linux Enterprise Server (SLES) 11 SP2.
   - OpenFabrics Enterprise Distribution (OFED) software. The operating system packages include OFED by default. To verify the OFED installation status, type one of the following commands:
     - On SLES 11 platforms, type the following:

       # **zypper info ---t pattern ofed**

     - On RHEL 6 platforms, type the following:

       # **yum grouplist "Infiniband Support"**

3. Proceed to the following:

   "Installing and Configuring MPT" on page 6

Installing and Configuring MPT

SGI distributes the MPT software as an RPM module. You can install the RPM itself, or you can use a `.cpio` file to install the RPM. In addition, you need to decide if you want to install the MPT software in the default location or if you want to install in a custom location.

Use one of the following procedures to install and configure the MPT software:

- "Installing the MPT RPM into the Default Location" on page 7
- "Installing the MPT RPM into an Alternate Location" on page 7
- "Using a .cpio File to Install the RPM into the Default Location or into an Alternate Location" on page 8

Installing the MPT RPM into the Default Location

The following procedure explains how to install the MPT RPM into its default location, which is `/opt/sgi/mpt`.

Procedure 2-2 To install the RPM into the default location

1. As the root user, log into the computer upon which you want to install the MPT software.
2. Type the following command:

   `# rpm -Uvh sgi-mpt-`*`mpt_rel`*`-sgi`*`lrp_rel`*`.x86_64.rpm`

   The variables in the preceding command are as follows:

   - For *mpt_rel*, type the release level of the MPT software that you want to install.
   - For *lrp_rel*, type the release level of the SGI Linux release product that includes the the MPT software.

   For example, to install the MPT 2.07 release, type the following command:

   ```
   # rpm -Uvh sgi-mpt-2.07-sgi707.x86_64.rpm
   ```

3. Proceed to:

   "Configuring Array Services" on page 13

Installing the MPT RPM into an Alternate Location

You can install the MPT RPM in an alternate location. If you install the MPT RPM into an alternate directory, users need to reset certain environment variables after the installation. The installation procedure guides you to "(Conditional) Resetting Enviroment Variables for Alternate-location Installations" on page 11 in a later step.

The following procedure explains how to install the MPT RPM into an alternate location.

Procedure 2-3 To install the RPM into an alternate location

1. As the root user, log into the computer upon which you want to install MPT.

2. Type the following command:

```
# rpm -i --relocate /opt/sgi/mpt/mpt-mpt_rel=/path \
--excludepath /usr sgi-mpt-mpt_rel-sgilrp_rel.x86_64.rpm
```

Note: The preceding command uses the \ character to signify a line break. Type this command all on one line before you press Enter.

The variables in the preceding command are as follows:

- For *mpt_rel*, type the release level of the MPT software that you want to install.
- For *path*, type the path to your alternate directory.
- For *lrp_rel*, type the release level of the SGI Linux release product that includes the the MPT software.

For example, to install the MPT 2.07 release into the `/tmp` directory, type the following command:

```
# rpm -i --relocate /opt/sgi/mpt/mpt-2.07=/tmp \
--excludepath /usr sgi-mpt-2.07-sgi707.x86_64.rpm
```

3. Proceed to one of the following:
   - "(Conditional) Enabling MPT for Cluster Environments for Alternate-location Installations" on page 10
   - "Configuring Array Services" on page 13

Using a .cpio File to Install the RPM into the Default Location or into an Alternate Location

The procedure in this topic explains how to use a `.cpio` file to install the MPT software into the default location or into an NFS file system that is shared by a number of hosts. In this case, it is not important or desirable for the RPM database on only one of the machines to track the versions of MPT that are installed. You do not need root permission to install the MPT software if you use a `.cpio` file.

The following procedure explains how to use a `.cpio` file to install the MPT software.

Procedure 2-4 To use a `.cpio` file to install the MPT RPM

1. As the root user, log into the computer upon which you want to install MPT.

2. Use the `cd` command to change to the installation directory.

   You can install the software in any directory to which you have write permission. The default installation directory is `/opt/sgi/mpt`.

   If you install the MPT software into an alternate directory, you need to reset your environment variables later in the installation procedure. The installation procedure guides you to "(Conditional) Resetting Enviroment Variables for Alternate-location Installations" on page 11 in a later step.

3. Type the following command to install the MPT software:

   `rpm2cpio -sgi-mpt-`*`mpt_rel`*`-*.rpm | cpio -idcmv`

   For *mpt_rel*, type the release version of MPT. For example, `2.07`.

   For example:

```
% rpm2cpio -sgi-mpt-2.07-*.rpm | cpio -idcmv
opt/sgi/mpt/mpt-2.07/bin/mpirun
opt/sgi/mpt/mpt-2.07/include/mpi++.h
opt/sgi/mpt/mpt-2.07/include/mpi.h
            ...
opt/sgi/mpt/mpt-2.07/lib/libmpi++.so
opt/sgi/mpt/mpt-2.07/lib/libmpi.so
opt/sgi/mpt/mpt-2.07/lib/libxmpi.so
            ...
```

4. List the files in the installation directory to confirm the installation.

   For example:

```
% ls -R /tmp/opt/sgi/mpt/mpt-2.06
bin  doc  include  lib  man

/tmp/opt/sgi/mpt/mpt-2.06/bin:
mpirun

/tmp/opt/sgi/mpt/mpt-2.06/include:
MPI.mod  mpi.h    mpi_ext.h   mpif.h             mpio.h    mpp
mpi++.h  mpi.mod  mpi_extf.h  mpif_parameters.h  mpiof.h

/tmp/opt/sgi/mpt/mpt-2.06/lib:
```

```
libmpi++.so*  libmpi.so*  libsma.so*  libxmpi.so*
           ...
```

5. Proceed to one of the following:

   - If you installed MPT in a nondefault location in a cluster environment, proceed to the following:

     "(Conditional) Enabling MPT for Cluster Environments for Alternate-location Installations" on page 10

   - If you installed MPT in a nondefault location, but not in a cluster environment, proceed to the following:

     "(Conditional) Resetting Enviroment Variables for Alternate-location Installations" on page 11

   - If you installed MPT in the default location, proceed to the following:

     "Configuring Array Services" on page 13

(Conditional) Enabling MPT for Cluster Environments for Alternate-location Installations

Perform this procedure if you installed MPT in an alternate location and you have MPT jobs that you need to run in a cluster environment.

The procedure in this topic explains how to copy all the MPT software modules to an NFS-mounted file system. This procedure enables the cluster nodes to access all the MPT software.

Procedure 2-5 To enable MPT in cluster environments if MPT was installed in an alternate location

1. As the root user, log into the computer upon which you installed the MPT software.

2. Type the following command to create a `tar` file of the MPT software modules:

   % **tar cf /*path*/mpt.*mpt_rel*.tar /opt/sgi/mpt/mpt-*mpt_rel***

   For *path*, type the path to your alternate directory.

   For *mpt_rel*, type the release level of the MPT software that you want to install.

3. Type the following command to copy the MPT software to the NFS file system:

   % **cp /*path*/mpt.*mpt_rel*.tar /*nfs_dir***

   The variables in the preceding command are as follows:

   - For *path*, type the path to your alternate directory.
   - For *mpt_rel*, type the release level of the MPT software.
   - For *nfs_dir*, type the NFS-mounted directory that the cluster-aware programs can access.

4. Use the cd(1) command to change to the NFS-mounted directory.

   For example:

   % **cd /data/nfs**

5. Type the following command to expand the file:

   % tar xf mpt.*mpt_rel*.tar

   For *mpt_rel*, type the release level of the MPT software.

6. Proceed to the following:

   "(Conditional) Resetting Enviroment Variables for Alternate-location Installations" on page 11

(Conditional) Resetting Enviroment Variables for Alternate-location Installations

Perform the procedure in this topic if you installed the MPT software into an alternate location.

The mpirun command assumes that the PATH and LD_LIBRARY_PATH environment variables are configured for the default MPT installation directory. The compilers, linkers, and loaders all use the values in these environment variables.

You can use the procedure in this topic to reset the environment variables either on the command line or in environment modules. For more information about environment modules, see "Running MPI Jobs" on page 19.

Procedure 2-6 To reset environment variables — Method 1, using commands

1. Open the configuration file appropriate to your shell, and change the following variables:

   - For the `tcsh` shell, the commands are as follows:

     ```
     setenv PATH /path/bin:${PATH}
     setenv LD_LIBRARY_PATH /path/lib
     ```

     For example:

     ```
     setenv PATH /tmp/bin:${PATH}
     setenv LD_LIBRARY_PATH /tmp/lib
     ```

   - For the `bash` shell, the commands are as follows:

     ```
     export PATH=/path/bin:${PATH}
     export LD_LIBRARY_PATH=/path/lib
     ```

     For example:

     ```
     export PATH=/tmp/bin:${PATH}
     export LD_LIBRARY_PATH=/tmp/lib
     ```

   For *path*, specify the full path to your alternate installation directory.

2. Proceed to the following:

   "Configuring Array Services" on page 13

Procedure 2-7 To reset environment variables — Method 2, using environment modules

1. Examine the following example module files:

   - `/opt/sgi/mpt/mpt-`*mpt_rel*`/doc/mpivars.sh`
   - `/opt/sgi/mpt/mpt-`*mpt_rel*`/doc/mpivars.csh`
   - On RHEL platforms, see the following file:

     `/usr/share/Modules/modulefiles/mpt/`*mpt_rel*

   - On SLES platforms, see the following file:

     `/usr/share/modules/modulefiles/mpt/`*mpt_rel*

   For *mpt_rel*, specify the release level of the MPT software that you installed.

2. Edit the module files and specify the variables needed.
3. Proceed to the following:

   "Configuring Array Services" on page 13

Configuring Array Services

Array Services must be configured and running on your SGI system or cluster before you start an MPI job. To configure Array Services, use one of the following procedures:

- If you have an SGI ICE series system, follow the procedure on the `arrayconfig_tempo`(8) man page to enable Array Services.
- If you have any other type of SGI system, use the procedure in this topic to configure Array Services.

  For more information about Array Services, see the `arrayconfig`(1) man page, the `arrayd.conf`(4) man page, and the *Linux Resource Administration Guide*.

Procedure 2-8 To enable array services

1. As the root user, log into one of the hosts.
2. Type the following command:

   # **`/usr/sbin/arrayconfig -m`** ***`host1`*** **`[`**`host2 ...`**`]`**

   For *host1* specify the hostname of the computer upon which you installed the MPT software. If you installed the MPT software on a cluster, specify the hostnames of the other computers in the cluster.

   This command creates the following two configuration files: `/etc/array/arrayd.conf` and `/etc/array/arrayd.auth`.
3. (Conditional) Copy files `/etc/array/arrayd.conf` and `/etc/array/arrayd.auth` to every cluster host.

   Perform this step if you installed the MPT software on multiple, clustered hosts.
4. Proceed to the following:

   "Configuring OFED for MPT" on page 14

Configuring OFED for MPT

You can specify the maximum number of queue pairs (QPs) for SHMEM and UPC applications when run on large clusters over OFED fabric. 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
```

SHMEM and UPC codes use the InfiniBand RC protocol for communication between all pairs of processes in the parallel job, which requires a large number of QPs. The `log_num_qp` parameter defines the $\log_2$ of the number of QPs. The following procedure explains how to specify the `log_num_qp` parameter.

Procedure 2-9 To specify the `log_num_qp` parameter

1. Log into one of the hosts upon which you installed the MPT software as the root user.
2. Use a text editor to open file `/etc/modprobe.d/libmlx4.conf`.
3. Add a line similar to the following to file `/etc/modprobe.d/libmlx4.conf`:

   ```
   options mlx4_core log_num_qp=21
   ```

   By default, the maximum number of queue pairs is 2^{17} (131072).
4. Save and close the file.
5. Repeat the preceding steps on other hosts.
6. Proceed to the following:

   "Restarting Services or Rebooting" on page 14

Restarting Services or Rebooting

The following procedure explains how to restart services. If you do not want to restart services, you can reboot your system.

Procedure 2-10 To restart services

1. Type the following commands:

```
# modprobe xpmem
# /etc/init.d/procset restart
# /etc/init.d/arrayd restart
```

2. (Conditional) Log into other hosts and type the commands in the preceding step.

   Perform this step if you want to run cluster-aware MPT programs.

   Make sure to reboot or to restart services on all cluster hosts.

3. Proceed to one of the following:

   - "(Conditional) Adjusting File Descriptor Limits" on page 15
   - "Adjusting the Resource Limit for Locked Memory" on page 16

(Conditional) Adjusting File Descriptor Limits

Perform the procedure in this chapter if you installed the MPT software on a large host with hundreds of processors.

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.

The following procedure explains how to increase the limit on the number of open files for all users.

Procedure 2-11 To increase the system limit on open files

1. As the root user, log in to the host upon which you installed the MPT software.
2. Use a text editor to open file `/etc/pam.d/login`.
3. Add the following line to file `/etc/pam.d/login`:

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

4. Save and close the file.
5. Use a text editor to open file `/etc/security/limits.conf`.

6. 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***
512	3000
1024	6000
8192	8192 (default)
4096	21000

   For example, the following line specifies 512 MPI processes per host:

   ```
   *     hard    nofile  3000
   ```

7. Save and close the file.

8. (Conditional) Update other files in the `/etc/pam.d` directory as needed.

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

   Modify the other files in the `/etc/pam.d` directory to accommodate the increased file descriptor limits.

9. (Conditional) Repeat the preceding steps on other hosts.

   Perform these steps if you installed the MPT software on more than one host.

10. Proceed to the following:

    "Adjusting the Resource Limit for Locked Memory" on page 16

Adjusting the Resource Limit for Locked Memory

The following procedure increases the resource limit for locked memory.

Procedure 2-12 To increase the resource limit

1. As the root user, log into the host upon which you installed the MPT software.

2. Use a text editor to open file `/etc/security/limits.conf`.
3. Add the following line to file `/etc/security/limits.conf`:

   ```
   *     hard   memlock  unlimited
   ```

4. (Conditional) Type the following commands to increase the resource limit for locked memory in the array services startup script:

   ```
   # sed -i.bak 's/ulimit -n/ulimit -l unlimited ; ulimit -n/' \
       /etc/init.d/array
   # /etc/init.d/array restart
   ```

5. (Conditional) Repeat the preceding steps on other hosts.

   Perform these steps if you installed the MPT software on multiple hosts.

6. (Conditional) Proceed to the following procedure if you installed the MPT software on multiple hosts:

   "(Conditional) Enabling Cross-partition NUMAlink MPI Communication" on page 17

(Conditional) Enabling Cross-partition NUMAlink MPI Communication

Perform the procedure in this topic if you installed the MPT software on multiple software partitions on an SGI UV system.

Large SGI UV systems can be configured into two or more NUMAlink-connected *partitions*. These 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. SGI recommends that you complete the procedure in this topic as part of the installation. If you do not perform this procedure during installation, you might receive the following message during the run of your 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.

Procedure 2-13 To load the kernel modules at boot (SLES)

1. As the root user, log into one of the hosts upon which you installed the 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. To the list of modules that are load at boot time, add `xpc`.
5. Save and close the file.
6. Reinitialize the kernel modules.

   To reinitialize the kernel modules, either reboot the system or type the following command:

   ```
   # modprobe xpc
   ```

7. Repeat the preceding steps on the other hosts.

Procedure 2-14 To load the kernel modules at boot (RHEL)

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

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

3. Save and close the file.
4. Reinitialize the kernel modules.

   To reinitialize the kernel modules, either reboot the system or type the following command:

   ```
   # modprobe xpc
   ```

5. Repeat the preceding steps on the other hosts.

Getting Started

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. It covers the following topics:

Running MPI Jobs

The following procedure explains how to run an MPI application when the MPT software is installed in an alternate location.

Procedure 3-1 To run jobs with MPT installed in an alternate location

1. Determine the directory into which the 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 -lmpi`

   `mpicc -I /`*`install_path`*`/usr/include `*`file`*`.c -L `*`lib_path`*`/usr/lib -lmpi`

   The variables in the preceding command are as follows:

   - For *install_path*, type the path to the directory in which the MPT software is installed.
   - For *file*, type the name of your C 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 -lmpi
   ```

3. (Conditional) Ensure that the program can find the 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 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 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 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 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
```

Once the MPT RPM is installed as default, the commands to build an MPI-based application using the .so files are as follows:

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

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

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

```
% ifort -o myprog myprog.f -lmpi  (Fortran - version 8)
% icc -o myprog myprog.c -lmpi    (C - version 8)
% mpif90 simple1_mpi.f            (Fortan 90)
% mpicc -o myprog myprog.c        (Open MPI C wrapper compiler)
% mpicxx -o myprog myprog.C       (Open 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 you installed MPT into a non-default 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.

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. 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 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.

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

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

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
```

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-2 Spawn Functions to Launch an Application

To use the MPI-2 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 above 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-2 `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 Work Load Manager

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

- "PBS Professional" on page 24
- "Torque" on page 26
- "SLURM" on page 26

PBS Professional

You can run MPI applications from job scripts that you submit through batch schedulers such as PBS Professional. The following procedures explain how to configure PBS job scripts to run MPI applications.

Procedure 3-2 To specify 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 on resource allocation options, see the `pbs_resources`(7) man page from the PBS Professional software distribution.

Procedure 3-3 To run the MPI application

Use the `mpiexec_mpt` command included in the SGI Message Passing Toolkit (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`(1) man page.

Process and thread pinning onto CPUs is especially important on cache coherent non-uniform memory access (ccNUMA) systems like 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 where `omplace`(1) is used.

Example 3-1 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 -n 512 ./a.out
```

Example 3-2 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 -n 512 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 5, "Debugging MPI Applications" on page 41.

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 MPT `mpiexec_mpt`(1) command to launch 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 MPT when running the TotalView Debugger with a batch scheduler like Torque. For more information on using the `mpiexec_mpt` command `-tv` option, see "Using the TotalView Debugger with MPI programs" on page 41.

SLURM

MPT can be used in conjunction with SGI® Simple Linux Utility for Resource Management (SLURM) product. The SLURM product has been adapted for MPT. Just use the `sgimpi` SLURM `mpi` plugin supplied in the product.

Type the following commands to get started:

```
% module load slurm
% module load mpt
% mpicc ...
% srun -n16 --mpi=sgimpi a.out
```

For information about how to install and configure SGI® SLURM, see the following:

Simple Linux Utility for Resource Management Install Guide

For general information about SLURM, see the following website:

https://computing.llnl.gov/linux/slurm/slurm.html

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

https://computing.llnl.gov/linux/slurm/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
```

You must use `mpirun` to launch SHMEM applications. The `NPES` variable has no effect on SHMEM programs. To request the desired number of processes to launch, you must 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

You can use huge pages to optimize the performance of your MPI application. The `MPI_HUGEPAGE_HEAP_SPACE` environment variable (see the `mpi`(1) man page) defines the minimum amount of heap space each MPI process would 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.

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

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

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

2. To configure the system settings for huge pages, as root user, perform the following:

   ```
   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. To view the current system configuration, perform the following command:

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

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

4. When running your MPT program, set the `MPI_HUGEPAGE_HEAP_SPACE` environment variable 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. To clear the system configuration settings, as root user, type the following:

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

6. To verify that the MPT huge page configuration has been cleared, retrieve the system configuration again, as follows:

   ```
   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.

Interoperation Between MPI, SHMEM, and UPC

The SGI UPC run-time environment depends on the SGI Message Passing Toolkit, which consists of the message passing interface (MPI) libraries and the SHMEM libraries. The MPI and SHMEM libraries provide job launch, parallel job control, memory mapping, and synchronization functions.

Just as with MPI or SHMEM jobs, you can use the `mpirun`(1) or `mpiexec_mpt`(1) commands to launch UPC jobs. UPC thread numbers correspond to SHMEM PE numbers and MPI rank numbers for `MPI_COMM_WORLD`.

For more information, see the *Unified Parallel C (UPC) User's Guide.*

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. The following 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.

`MPI_Abort`

In the SGI MPI implementation, a call to `MPI_Abort` causes the termination of the entire MPI job, 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

Section 7.2 of 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`. The handler, 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`. The 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 section 7.5 of the MPI Standard.

If the application uses the MPI-2 spawn functionality described in Chapter 5 of the MPI-2 Standard, there are additional considerations. In the SGI implementation, all MPI processes are connected. Section 5.5.4 of the MPI-2 Standard defines what is meant by connected processes. When the MPI-2 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 make use of 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 4-1 on page 33 illustrates a simple sequence of MPI operations that cannot work unless messages are buffered. If sent messages were not buffered, each process would hang in the initial call, waiting for an MPI_Recv call to take the message.

Because most MPI implementations do 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 4-1 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 35.

Multithreaded Programming

SGI MPI supports hybrid programming models, in which MPI is used to handle one level of parallelism in an application, while 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`. See the `mpi`(1) man page for more information about compiling and linking MPI programs.

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, some MPI processes may not be able to communicate with certain other MPI processes when using SHMEM functions. 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-2 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` man page.

Miscellaneous Features of SGI MPI

This section describes other characteristics of the SGI MPI implementation that might be of interest to application developers.

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`. `stdout` and `stderr` results are enabled for all MPI processes in the job, whether started by `mpirun` or started by one of the MPI-2 spawn functions.

MPI_Get_processor_name

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

Programming Optimizations

The following topics describe how to use the optimized features of SGI's MPI implementation. You might need to modify your MPI application to use these recommendations.

Using MPI Point-to-Point Communication Routines

MPI provides for 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. The barrier operation has also been optimized to use hardware fetch operations (`fetchops`). The `MPI_Alltoall` routines also use special techniques to avoid message buffering when using shared memory. For more details, see "Avoiding Message Buffering — Single Copy Methods" on page 37.

Note: Collectives are optimized across partitions by using the XPMEM driver which is explained in Chapter 8, "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. It is generally best to avoid the use of these functions by either restructuring your data or using derived data types. 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, use of derived data types does not generally lead to performance gains. Use of 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, avoiding the use of shared memory buffers. However, as discussed in "Buffering" on page 33, 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 49.

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. If you need to control the placement of the application at run time, however, see the following:

Chapter 8, "Run-time Tuning" on page 47

Using Global Shared Memory

The MPT software includes the Global Shared Memory (GSM) Feature. This feature allows users to allocate globally accessible shared memory from within an MPI or SHMEM program. The GSM feature can be used to provide shared memory access across partitioned SGI UV systems and to provide additional memory placement options within a single-host configuration.

User-callable functions are provided to allocate a global shared memory segment, free that segment, and provide information about the segment. Once allocated, the application can use this new global shared memory segment via standard loads and stores, just as if it were a System V shared memory segment. For more information, see the `GSM_Intro` or `GSM_Alloc` man pages.

Additional Programming Model Considerations

A number of additional programming options might be worth consideration when developing MPI applications for SGI systems. For example, the SHMEM programming model can provide a means to improve the performance of 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-2 `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 8, "Run-time Tuning" on page 47.

Debugging MPI Applications

Debugging MPI applications can be more challenging than debugging sequential applications. This chapter presents methods for debugging MPI applications. It covers the following topics:

MPI Routine Argument Checking

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 syntax for running SGI MPI with the TotalView Debugger (TVD) from TotalView Technologies is as follows:

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

Note that TVD is not expected to operate with MPI processes started via the `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` functions.

The MPT `mpiexec_mpt`(1) command has a `-tv` option for use by MPT with the TotalView Debugger. Note that the PBS Professional `mpiexec`(1) command does not support the `-tv` option.

To run an MPT MPI job with TotalView without a batch scheduler (same as the above example), type the following:

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

To run an MPT MPI job with Total View Debugger with a batch schduler, 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.

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: The MPI C++ API is not supported with PerfBoost.

Using PerfBoost

To use PerfBoost with an SGI MPT MPI program, first load the `perfboost` environmental module (see Example 6-1 on page 43). Then insert the `perfboost` command in front of the executable name along with the choice of MPI implementation to emulate. Launch the application with the SGI MPT `mpiexec_mpt`(1) or `mpirun`(1) command. The following are MPI implementations and corresponding command line options:

MPI Implementation	Command Line Option
Platform MPI 7.1+	`-pmpi`
HP-MPI	`-pmpi`
Intel MPI	`-impi`
OpenMPI	`-ompi`
MPICH1	`-mpich`
MPICH2	`-impi`
MVAPICH2	`-impi`

Example 6-1 Using the SGI `perfboost` Software

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
```

Environment Variables

The following environment variable is supported:

PERFBOOST_VERBOSE — Setting the PERFBOOST_VERBOSE environment variable 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 when the MPI application completes through the libperfboost wrapper library.

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

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.

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

Checkpoint/Restart

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 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 MPT

To enable checkpoint/restart within 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 *Linux Resource Administration 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 batch scheduler; for example, Platform LSF from Platform Computing Corporation or PBS Professional from Altair Engineering, Inc. These batch schedulers 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 128k 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 128k, the software breaks the message into chunks that are 128k 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 69.

For information about buffering considerations when running an MPI job across multiple hosts, see "Tuning for Running Applications Across Multiple Hosts" on page 53.

For information about the programming implications of message buffering, see "Buffering" on page 33.

Avoiding Message Buffering – Enabling Single Copy

For message transfers between MPI processes within the same host or transfers between partitions, 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`, `MPI_Alltoall`, `MPI_Bcast`, `MPI_Allreduce`, and `MPI_Reduce`, 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-2 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 discribed 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-2 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 will get a physical CPU and memory on the node to which it was assigned. If this environment variable is used without specifying an MPI_DSM_CPULIST variable, it will cause MPI to assign MPI ranks starting at logical CPU 0 and incrementing until all ranks have been placed. Therefore, it is recommended that this variable be used only if running within a cpuset on a dedicated system.

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.

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 *Linux Resource Administration Guide*. For more information on using `dplace`, see the *Linux Application Tuning Guide*.

Example 8-1 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 batch scheduler to schedule the a hybrid MPI/OpenMP job as shown in Example 8-1 on page 53, 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 MPT programs with PBS, see"Running MPI Jobs with a Work Load Manager" on page 24 .

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 23. 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 8-1 on page 54 specifies the order in which MPI probes for available interconnects.

Table 8-1 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
TCP/IP	3	MPI_USE_TCP

The third column of Table 8-1 on page 54 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, MPT applications may not execute a `fork`() or `system`() call. The InfiniBand driver produces undefined results when an 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. 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, 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_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_NUM_MEMORY_REGIONS

For zero-copy sends over the InfiniBand interconnect, 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_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 will create InfiniBand connections lazily 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 16384 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, 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 4.

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 batch scheduler 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 50), `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 batch scheduler 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

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 62). 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 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. Here are some examples:

```
mpirun -np 64 perfcatch a.out arg1
```
and

```
mpirun host1 32, host2 64 perfcatch a.out arg1
```

To use `perfcatch` with Intel MPI, add the `-i` options. An example is, 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.

MPI Supported Profiled Functions

The MPI supported profiled functions are, as follows:

Note: Some functions may not be implemented in all language as indicated below.

Languages	Function
C Fortran	`mpi_allgather`
C Fortran	`mpi_allgatherv`
C Fortran	`mpi_allreduce`
C Fortran	`mpi_alltoall`
C Fortran	`mpi_alltoallv`
C Fortran	`mpi_alltoallw`
C Fortran	`mpi_barrier`
C Fortran	`mpi_bcast`
C Fortran	`mpi_comm_create`
C Fortran	`mpi_comm_free`
C Fortran	`mpi_comm_group`
C Fortran	`mpi_comm_rank`
C Fortran	`mpi_finalize`
C Fortran	`mpi_gather`
C Fortran	`mpi_gatherv`
C	`mpi_get_count`
C Fortran	`mpi_group_difference`
C Fortran	`mpi_group_excl`
C Fortran	`mpi_group_free`
C Fortran	`mpi_group_incl`
C Fortran	`mpi_group_intersection`

C Fortran	mpi_group_range_excl
C Fortran	mpi_group_range_incl
C Fortran	mpi_group_union
C	mpi_ibsend
C Fortran	mpi_init
C	mpi_init_thread
C Fortran	mpi_irecv
C Fortran	mpi_isend
C	mpi_probe
C Fortran	mpi_recv
C Fortran	mpi_reduce
C Fortran	mpi_scatter
C Fortran	mpi_scatterv
C Fortran	mpi_send
C Fortran	mpi_sendrecv
C Fortran	mpi_ssend
C Fortran	mpi_test
C Fortran	mpi_testany
C Fortran	mpi_wait
C Fortran	mpi_wait

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-1 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 aquire 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 8, "Run-time Tuning" on page 47.

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 smaller, 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 even more problematic 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 users encounter when starting to use SGI MPI, as well as 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 arror.

- 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 descibed in "Compiling and Linking MPI Programs" on page 21.

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 MPT software to install on my machine?

MPT RPMs are included in the SGI Performance Suite releases. In addition, you can obtain MPT RPMs from the SGI Support website at

```
http://support.sgi.com
```
under "Downloads".

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.

Index

O

P

R

S

T

U