



Message Passing Toolkit (MPT) User's Guide

007-3773-016

COPYRIGHT

©1996, 1998-2010, 2011, SGI. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of SGI.

LIMITED RIGHTS LEGEND

The software described in this document is "commercial computer software" provided with restricted rights (except as to included open/free source) as specified in the FAR 52.227-19 and/or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS

SGI, Altix, the SGI logo, Silicon Graphics, IRIX, and Origin are registered trademarks and CASEVision, NUMALink, OpenMP, Performance Co-Pilot, ProDev, SHMEM, and SpeedShop are trademarks of Silicon Graphics International Corp. or its subsidiaries in the United States and other countries.

InfiniBand is a trademark of the InfiniBand Trade Association. Intel, Itanium, and Xeon are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Kerberos is a trademark of Massachusetts Institute of Technology. Linux is a registered trademark of Linus Torvalds in several countries. MIPS is a registered trademark and MIPSpro is a trademark of MIPS Technologies, Inc., used under license by SGI, in the United States and/or other countries worldwide. PBS Professional is a trademark of Altair Engineering, Inc. Platform Computing is a trademark and Platform LSF is a registered trademark of Platform Computing Corporation. PostScript is a trademark of Adobe Systems, Inc. TotalView and TotalView Technologies are registered trademarks and TVD is a trademark of TotalView Technologies. UNIX is a registered trademark of the Open Group in the United States and other countries.

New Features in This Manual

This update to the *Message Passing Toolkit (MPT) User's Guide* supports the MPT 2.03 release.

Major Documentation Changes

Performed the following:

- Updated release information in Chapter 2, "Administrating MPT" on page 5.
- Added information about the checkpoint/restart feature in Chapter 7, "Checkpoint/Restart " on page 33.
- Updated message length information "Tuning MPI Buffer Resources" on page 36.

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.

Record of Revision

- | | |
|-----|---|
| 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. |

Contents

About This Manual	xv
Related Publications and Other Sources	xv
Obtaining Publications	xvi
Conventions	xvi
Reader Comments	xvii
1. Introduction	1
MPI Overview	2
MPI 2.2 Standard Compliance	2
MPI Components	2
SGI MPI Features	2
2. Administrating MPT	5
Finding the MPT Release Notes	5
MPT Installation	5
Disk Space Requirements	6
Prerequisites	6
SGI Performance Suite Components	6
InfiniBand Software Stack	6
Installing the MPT RPM	6
Installing MPT Software in an Alternate Location	7
Using a <code>cpio</code> File for Installation	8
Using Dynamic Shared Libraries to Run MPI Jobs	9
Running MPI Jobs on a Cluster with MPT Alternate Installation	10
System Configuration	10
007-3773-016	vii

Starting Prerequisite Services	10
Configuring Array Services	11
Adjusting File Descriptor Limits	11
Adjusting Locked Memory Limits	12
3. Getting Started	13
Compiling and Linking MPI Programs	13
Using <code>mpirun</code> to Launch an MPI Application	14
Launching a Single Program on the Local Host	14
Launching a Multiple Program, Multiple Data (MPMD) Application on the Local Host	14
Launching a Distributed Application	15
Using MPI-2 Spawn Functions to Launch an Application	15
Running MPI Jobs with a Work Load Manager	16
PBS Professional	16
Torque	18
Compiling and Running SHMEM Applications	18
4. Programming with SGI MPI	21
Job Termination and Error Handling	21
MPI_Abort	21
Error Handling	22
MPI_Finalize and Connect Processes	22
Signals	22
Buffering	23
Multithreaded Programming	24
Interoperability with the SHMEM programming model	24
Miscellaneous Features of SGI MPI	25
stdin/stdout/stderr	25

MPI_Get_processor_name	25
Programming Optimizations	25
Using MPI Point-to-Point Communication Routines	25
Using MPI Collective Communication Routines	26
Using MPI_Pack/MPI_Unpack	26
Avoiding Derived Data Types	26
Avoiding Wild Cards	27
Avoiding Message Buffering — Single Copy Methods	27
Managing Memory Placement	27
Using Global Shared Memory	28
Additional Programming Model Considerations	28
5. Debugging MPI Applications	29
MPI Routine Argument Checking	29
Using the TotalView Debugger with MPI programs	29
Using <code>idb</code> and <code>gdb</code> with MPI programs	30
6. PerfBoost	31
Using PerfBoost	31
Environment Variables	32
MPI Supported Functions	32
7. Checkpoint/Restart	33
BLCR Installation	33
Using BLCR with MPT	34
8. Run-time Tuning	35
Reducing Run-time Variability	35
Tuning MPI Buffer Resources	36

Avoiding Message Buffering – Enabling Single Copy	37
Using the XPMEM Driver for Single Copy Optimization	38
Memory Placement and Policies	38
MPI_DSM_CPULIST	39
MPI_DSM_DISTRIBUTE	40
MPI_DSM_VERBOSE	40
Using <code>dplace</code> for Memory Placement	40
Tuning MPI/OpenMP Hybrid Codes	41
Tuning for Running Applications Across Multiple Hosts	42
MPI_USE_IB	43
MPI_IB_RAILS	43
MPI_IB_SINGLE_COPY_BUFFER_MAX	44
Tuning for Running Applications over the InfiniBand Interconnect	44
MPI_NUM_QUICKS	44
MPI_NUM_MEMORY_REGIONS	44
MPI_CONNECTIONS_THRESHOLD	44
MPI_IB_PAYLOAD	45
MPI_IB_TIMEOUT	45
MPI_IB_FAILOVER	45
MPI on Altix UV 100 and Altix UV 1000 Systems	45
General Considerations	46
Job Performance Types	46
Other ccNUMA Performance Issues	47
Suspending MPI Jobs	47
9. MPI Performance Profiling	49
Overview of <code>perfcatch</code> Utility	49
Using the <code>perfcatch</code> Utility	49

MPI_PROFILING_STATS Results File Example	50
MPI Performance Profiling Environment Variables	53
MPI Supported Profiled Functions	54
Profiling MPI Applications	55
Profiling Interface	56
MPI Internal Statistics	57
Third Party Products	57
10. Troubleshooting and Frequently Asked Questions	59
What are some things I can try to figure out why <code>mpirun</code> is failing?	59
My code runs correctly until it reaches <code>MPI_Finalize()</code> and then it hangs.	61
I keep getting error messages about <code>MPI_REQUEST_MAX</code> being too small.	61
I am not seeing <code>stdout</code> and/or <code>stderr</code> output from my MPI application.	62
How can I get the MPT software to install on my machine?	62
Where can I find more information about the SHMEM programming model?	62
The <code>ps(1)</code> command says my memory use (<code>SIZE</code>) is higher than expected.	62
What does <code>MPI: could not run executable</code> mean?	62
How do I combine MPI with <i>insert favorite tool here</i> ?	63
Why do I see “stack traceback” information when my MPI job aborts?	64
Index	65

Tables

Table 4-1	Outline of Improper Dependence on Buffering	23
Table 8-1	Inquiry Order for Available Interconnects	42

About This Manual

This publication documents the SGI implementation of the Message Passing Interface (MPI).

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 3.1 or later. For more information on Array Services, see Chapter 3, “Array Services” in the *Linux Resource Administration Guide*.

Related Publications and Other Sources

Material about MPI is available from a variety of sources. Some of these, particularly webpages, include pointers to other resources. Following is a grouped list of these sources.

The MPI standard:

- As a technical report: University of Tennessee report (reference [24] from *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum).
- As online PostScript or hypertext on the Web:
<http://www.mpi-forum.org/>
- As a journal article in the *International Journal of Supercomputer Applications*, volume 8, number 3/4, 1994. See also *International Journal of Supercomputer Applications*, volume 12, number 1/4, pages 1 to 299, 1998.

Book: *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, publication TPD-0011.

Newsgroup: `comp.parallel.mpi`

SGI manuals: *SpeedShop User's Guide* and *Linux Resource Administration Guide* for information on Array Services

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: <http://docs.sgi.com>. 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
<code>command</code>	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
<code>manpage(x)</code>	Man page section identifiers appear in parentheses after man page names.
<i>variable</i>	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.
- The SHMEM programming model. The SHMEM programming model is a distributed, shared-memory model that consists of a set of SGI proprietary message-passing library routines. These routines help distributed applications efficiently transfer data between cooperating processes. The model is based on multiple processes having separate address spaces, with the ability for one process to access data in another process' address space without interrupting the other process. The SHMEM programming model is not a standard like MPI, so SHMEM applications developed on other vendors' hardware might or might not work with the SGI SHMEM implementation.

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

- Chapter 2, "Adminstrating MPT" on page 5
- Chapter 3, "Getting Started" on page 13
- Chapter 4, "Programming with SGI MPI" on page 21
- Chapter 5, "Debugging MPI Applications" on page 29
- Chapter 6, "PerfBoost" on page 31

- Chapter 7, "Checkpoint/Restart " on page 33
- Chapter 8, "Run-time Tuning" on page 35
- Chapter 9, "MPI Performance Profiling" on page 49
- Chapter 10, "Troubleshooting and Frequently Asked Questions" on page 59

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 address:<http://www.mcs.anl.gov/mpi>

MPI 2.2 Standard Compliance

The SGI MPI implementation is compliant with the MPI 2.2 standard.

MPI Components

The MPI library is provided as a dynamic shared object (DSO) (a file with a name 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.

Profiling support is included 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 to use on SGI hardware:

- Data transfer optimizations for NUMALink where available, including single-copy data transfer

- Multi-rail InfiniBand support to take full advantage of the multiple InfiniBand fabrics available on SGI Altix 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 via XPMEM

Administrating MPT

This chapter is provided for system administrators who install, configure, and administer software on SGI Altix systems. It covers the following topics:

- "Finding the MPT Release Notes" on page 5
- "MPT Installation" on page 5
- "System Configuration" on page 10

Finding the MPT Release Notes

Find the latest MPT release notes on your system, as follows:

```
% rpm -qi sgi-mpt | grep README.relnotes
/opt/sgi/mpt/mpt-2.03/doc/README.relnotes
```

Next, change directory to the location found, and list the contents of the directory, as follows:

```
% cd /opt/sgi/mpt/mpt-2.03/doc
% ls
MPT_UG README.relnotes sgi-mpt.2.03.template
The release notes are in a file called README.relnotes.
```

MPT Installation

This section describes requirements and procedures for MPT installation. After you have installed the MPT and prerequisite software per the instructions in this section, be sure to perform the steps described in "System Configuration" on page 10.

Note: The MPI installation and configuration information found in this chapter is also available in the `README.relnotes` file in the `/opt/sgi/mpt/mpt-2.03/doc` directory.

Disk Space Requirements

Disk space requirements for the MPT product are substantially less than 20 Mbytes.

Prerequisites

This section describes software prerequisites for MPT.

SGI Performance Suite Components

A default install SGI Accelerate is recommended. This provides a number of software components required by MPT. The SGI Performance Suite RPMs required by MPT include the following:

- `sgi-arraysvcs` or `sgi-sarraysvcs`
- `sgi-procset`
- `libbitmask`
- `libcpuset`
- `cpuset-utils`
- `sgi-release`
- `xpmem`

InfiniBand Software Stack

If you are using the InfiniBand interconnect, you must ensure that one of the supported InfiniBand software stacks is installed.

Installing the MPT RPM

MPT is supplied as an RPM file. The name of the file contains the following information:

- Product (`sgi-mpt`)
- MPT Version (`2.03`)
- SGI MPI 1.1 software bundle, part of SGI Performance Suite 1.1

- Architecture (SGI IA-64 or x86_64 based systems)

For example, if the name of the MPT RPM for the MPT 2.03 release is `sgi-mpt-2.03-sgi703.ia64.rpm`. To install this RPM, log in as root and issue the following command:

```
# rpm -Uvh sgi-mpt-2.03-sgi703.ia64.rpm
```

Installing MPT Software in an Alternate Location

RPM provides a means for creating, installing, and managing relocatable packages. That is, the MPT RPM can be installed in either a default or alternate location.

The default location for installing the MPT RPM is `/opt/sgi/mpt/`. To install the MPT RPM in an alternate location, use the `--relocate` option, as shown in the following example. The `--relocate` option specifies the alternate base directory for the installation of the MPT software (in this case, `/tmp`).

```
# rpm -i --relocate /opt/sgi/mpt/mpt-2.03=/tmp --excludepath /usr sgi-mpt-2.03-sgi703.ia64.rpm
```

Note: If the MPT software is installed in an alternate location, MPT users must set the environment variables `PATH` and `LD_LIBRARY_PATH` to specify the search locations for the `mpirun` command and the run-time libraries, assuming the alternate location of `/tmp`, as follows:

```
setenv PATH /tmp/bin:${PATH}
export PATH=/tmp/bin:${PATH}

export LD_LIBRARY_PATH=/tmp/lib
export LD_LIBRARY_PATH=/tmp/lib
```

If the site is using environment modules to manage the user environment, then the alternate location should be placed in the `mpt` modulefile. This approach is the most convenient way to establish environment variable settings that enable MPT program developers and users to access the MPT software when installed in an alternate location. Sample modulefiles are located in `/opt/sgi/mpt/mpt-2.03/doc` and `/usr/share/modules/modulefiles/mpt/2.03`.

For more information, see "Using Dynamic Shared Libraries to Run MPI Jobs," later in this chapter.

Using a `cpio` File for Installation

The `cpio` file installation method described here is useful when the MPT software is installed in an NFS filesystem 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. Another advantage of the approach is that you do not need root permission to install the MPT software.

To install MPT using a `cpio` file, first convert the MPT RPM to a `cpio` file by executing the `rpm2cpio` command, as follows:

```
% rpm2cpio sgi-mpt-2.03-1.ia64.rpm > /tmp/sgi-mpt.cpio
```

Once you have created the `.cpio` file, you are free to install the software beneath any directory in which you have write permission. The following example demonstrates the process.

```
% cd /tmp
% cpio -idmv < sgi-mpt.cpio
opt/sgi/mpt/mpt-2.03/bin/mpirun
opt/sgi/mpt/mpt-2.03/include/mpi++.h
opt/sgi/mpt/mpt-2.03/include/mpi.h
...
opt/sgi/mpt/mpt-2.03/lib/libmpi++.so
opt/sgi/mpt/mpt-2.03/lib/libmpi.so
opt/sgi/mpt/mpt-2.03/lib/libxmpi.so
...
% ls -R /tmp/opt/sgi/mpt/mpt-2.03
bin doc include lib man

/tmp/opt/sgi/mpt/mpt-2.03/bin:
mpirun

/tmp/opt/sgi/mpt/mpt-2.03/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-2.03/lib:
libmpi++.so* libmpi.so* libsma.so* libxmpi.so*
...
```

If the MPT software is installed in an alternate location, set up an environment module to set environment variables which will be used by compilers, linkers, and runtime loaders to reference the MPT software.

Using Dynamic Shared Libraries to Run MPI Jobs

After you have installed the MPT RPM as default, use the following command to build an MPI-based application that uses the `.so` files:

For C programs, as follows:

```
% gcc simple1_mpi.c -lmpi
% mpirun -np 2 a.out
```

For Fortran programs:

```
% f77 -I/usr/include simple1_mpi.f -lmpi
% mpirun -np 2 a.out
```

The default locations for the `include` and `.so` files and the `mpirun` command are referenced automatically.

Assuming that the MPT package has been installed in an alternate location (under the `/tmp` directory), as described earlier in "Installing MPT Software in an Alternate Location" on page 7, the commands to compile, load, and check are, as follows:

```
% gcc -I /tmp/usr/include simple1_mpi.c -L/tmp/usr/lib -lmpi
% ldd a.out
libmpi.so => /usr/lib/libmpi.so (0x40019000)
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)
```

As shown above, compiling with alternate-location libraries does not mean that your program will run with them. Note that `libmpi.so` is resolved to `/usr/lib/libmpi.so`, which is the default-location library. If you are going to use an alternate location for the `.so` files, it is important to set the `LD_LIBRARY_PATH` environment variable. If the site is using environment modules, this can be done in the `mpt` modulefile. Otherwise, the user must set `LD_LIBRARY_PATH`, as in the following example:

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

This example shows the library being resolved to the correct file.

Running MPI Jobs on a Cluster with MPT Alternate Installation

For MPI jobs to run correctly in a cluster environment in which MPT has been installed in an alternate location, you must copy all of the pertinent pieces of MPT to an NFS-mounted filesystem. This is the only way in which all of the nodes in the cluster can access the software, short of installing the same MPT RPM on each node. The following method is one way to accomplish this (assuming /data/nfs is an NFS-mounted directory and MPT has been installed in the alternate location /tmp/usr):

```
node1 # tar cf /tmp/mpt.2.03.tar /tmp/usr
node1 # cp /tmp/mpt.2.03.tar /data/nfs
node1 # cd /data/nfs
node1 # tar xf mpt.2.03.tar
node1 # setenv LD_LIBRARY_PATH /data/nfs/lib
node1 # /data/nfs/bin/mpirun -v -a <arrayname> host_A,host_B -np 1 a.out
Replace the <arrayname> in the above example with an array services array name that
contains both host_A and host_B.
```

System Configuration

This section describes additional system configuration issues that a system administrator may need to address before running the SGI MPT software.

Starting Prerequisite Services

MPT requires that procset and array services be started and that the XPMEM kernel module be loaded. These tasks are performed automatically by a reboot of the system

occurring after the system configuration tasks in this section have been performed. If a reboot has not been performed, the following commands should be executed by root:

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

If you will be running MPT on a clustered system, these steps (or a reboot) must be performed for all hosts in the cluster.

Configuring Array Services

Array Services must be configured and running on all hosts in a cluster to perform the launch of MPI jobs. You can set up a simple Array Services configuration by executing the following two commands as root on all hosts of the cluster. List all host names on the `arrayconfig` command line:

```
/usr/sbin/arrayconfig -m host1 host2 ...
/etc/init.d/array restart
```

For a more elaborate configuration, consult the `arrayconfig(1)` and `arrayd.conf(4)` man pages and the "Installing and Configuring Array Services" section of the *Linux Resource Administration Guide*.

Adjusting File Descriptor Limits

On large hosts with hundreds of processors, MPI jobs require a large number of file descriptors. On these 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 steps can increase this limit for all users:

1. Add the following line to `/etc/pam.d/login`:

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

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

```
*          hard    nofile    DESC_LIMIT
```

Use the value of `DESC_LIMIT` that corresponds to the the desired maximum number of MPI processes per host, as follows:

MPI Processes Per Host	DESC_LIMIT Value
512	3000
1024	6000
4096	21000

For example, use " * hard nofile 3000 " to allow up to 512 MPI processes per host.

If other login methods are used (ssh, rlogin, and so on), and the increased file descriptor limits are desired, the corresponding files in /etc/pam.d should be modified as well.

Adjusting Locked Memory Limits

The OFED-based InfiniBand software stacks require the resource limit for locked memory to be set to a high value.

Increase the user hard limit by adding the following line to /etc/security/limits.conf:

```
*      hard  memlock  unlimited
```

If you are running on a system with an SGI ProPack software release prior to SGI ProPack 5 Service Pack 1, you will also need to patch the Array Services startup script /etc/init.d/array to ensure that arrayd is running with a high "memlock" hard limit. This is done by the following sequence, executed as root:

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

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:

- "Compiling and Linking MPI Programs" on page 13
- "Running MPI Jobs with a Work Load Manager" on page 16
- "Compiling and Running SHMEM Applications" on page 18

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. 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
% g77 -I/usr/include -o myprog myprog.f -lmpi
```

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

```
% efc -o myprog myprog.f -lmpi           (Fortran - version 7.1)
% ecc -o myprog myprog.c -lmpi           (C - version 7.1)
% ifort -o myprog myprog.f -lmpi         (Fortran - version 8)
% icc -o myprog myprog.c -lmpi           (C - version 8)
```

The `libmpi++.so` library is compatible with code generated by `g++ 3.0` or later compilers, as well as Intel C++ 8.0 or later compilers. If compatibility with previous `g++` or C++ compilers is required, the `libmpi++.so` released with MPT 1.9 (or earlier) must be used.

Note: You must use the Intel compiler to compile Fortran 90 programs.

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

```
% efc -I/usr/include -o myprog myprog.f -lmpi (version 7.1)
% ifort -I/usr/include -o myprog myprog.f -lmpi (version 8)
```

Note: The above command line assumes a default installation; if you have installed MPT into a non-default location, replace `/usr/include` with the name of the relocated directory.

Using `mpirun` to Launch an MPI Application

You must use the `mpirun` command to start MPI applications. For 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 launch 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 launch 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, or 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 launch 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`, you must specify the universe size by specifying the `-up` option on the `mpirun` command line. 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 Altix systems which use the MPI-2 `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` functions, it may be necessary to explicitly specify the partitions on which additional MPI processes may be launched. See the section "Launching Spawn Capable Jobs on Altix Partitioned Systems" on 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, or Load Sharing Facility (LSF), it needs to launch on the cluster nodes and CPUs that have been allocated to the job. For multi-node MPI jobs, this type of launch requires the use of an MPI launch command that interprets the node and CPU selection information for the workload manager that is in use. One of these commands, `mpiexec_mpt(1)`, is provided with MPT, and another such command, `mpiexec(1)`, ships with the PBS Professional workload manager software. The following section describes how to launch MPI jobs with specific workload managers and covers these topics:

- "PBS Professional" on page 16
- "Torque" on page 18

PBS Professional

Often MPI applications are run from job scripts submitted through batch schedulers like PBS Professional. This section provides some details about how to properly set up PBS job scripts to run MPI applications.

PBS job scripts can specify needed resource allocations using the `-l` option on a "#PBS" directive line. These lines will have the following form:

```
#PBS -l select=P:ncpus=T[:other options]
```

The value `P` should be set to the total number of MPI processes in the job, and the value `T` should be set to the number of OpenMP threads per process. For purely MPI jobs, `T` is 1. For more information on resource allocation options, see the `pbs_resources(7)` man page from the PBS Professional software distribution.

Each MPI application is executed with the `mpiexec` command that is delivered with the PBS Professional software packages. This is a wrapper script that assembles the

correct host list and corresponding `mpirun` command before executing the assembled `mpirun` command. The basic syntax is, as follows:

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

where `P` 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(1)` man page for more details.

Process and thread pinning onto CPUs is especially important on cache coherent non-uniform memory access (ccNUMA) systems like the SGI Altix 4700 or the SGI Altix UV 1000. 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 will set 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 Run an MPI application with 512 Processes

To run an application with 512 processes, perform the following:

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

Example 3-2 Run 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, perform the following:

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

The `mpiexec_mpt(1)` command is provided by the SGI Message Passing Toolkit (MPT). The `mpiexec_mpt` command launches a MPT MPI program in a batch scheduler-managed cluster environment. When running PBS Professional, `mpiexec_mpt` is an alternative to the `mpiexec(1)` command. Unlike the PBS Professional `mpiexec` command, `mpiexec_mpt` supports all MPT `mpirun` global options. The `mpiexec_mpt` command has a `-tv` option for use by MPT with the TotalView Debugger. For more information on using the `mpiexec_mpt` command `-tv` option, see "Using the TotalView Debugger with MPI programs" on page 29.

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
```

where `P` 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 29.

Compiling and Running SHMEM Applications

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

```
% g++ compute.C -lsma
% gcc compute.c -lsma
% g77 -I/usr/include compute.f -lsma
```

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

```
% ecc compute.C -lsma (version 7.1)
% ecc compute.c -lsma (version 7.1)
% efc compute.f -lsma (version 7.1)
% icc compute.C -lsma (version 8)
% icc compute.c -lsma (version 8)
% ifort compute.f -lsma (version 8)
```

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 single host SHMEM applications, as well as SHMEM applications that span multiple partitions. To launch a SHMEM

application on more than one partition, use the multiple host `mpirun` syntax, such as the following:

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

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

Programming with SGI MPI

Portability is one of the main advantages MPI has over vendor-specific message passing software. Nonetheless, the MPI Standard offers sufficient flexibility for general variations in vendor implementations. In addition, there are often vendor specific programming recommendations for optimal use of the MPI library. This chapter addresses topics that are of interest to those developing or porting MPI applications to SGI systems. It covers the following topics:

- "Job Termination and Error Handling" on page 21
- "Signals" on page 22
- "Buffering" on page 23
- "Multithreaded Programming" on page 24
- "Interoperability with the SHMEM programming model" on page 24
- "Miscellaneous Features of SGI MPI" on page 25
- "Programming Optimizations" on page 25
- "Additional Programming Model Considerations" on page 28

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 will hang 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 handling of signals 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 23 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 by 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
<code>MPI_Send(2,)</code>	<code>MPI_Send(1,)</code>
<code>MPI_Recv(2,)</code>	<code>MPI_Recv(1,)</code>

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 25.

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` (which does 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 launched via `mpirun`, or via 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 invoking this subroutine is running.

Programming Optimizations

This section describes ways in which the MPI application developer can best make use of optimized features of SGI's MPI implementation. Following recommendations in this section might require modifications to your MPI application.

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 (`MPI_Send`, `MPI_Isend`, `MPI_Recv`, and `MPI_Irecv`).

Unless required for application semantics, the synchronous `send` calls (`MPI_Ssend` and `MPI_Issend`) should be avoided. The buffered `send` calls (`MPI_Bsend` and `MPI_Ibsend`) should also usually be avoided as these double the amount of memory copying on the sender side. The ready `send` routines (`MPI_Rsend` and `MPI_Irsend`) are treated as standard `MPI_Send` and `MPI_Isend` in this implementation. 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 (32 processes or more) or for clusters, this approach can become 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 27.

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`) will 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 may lead to decreased performance in certain cases.

Avoiding Derived Data Types

In general, you should 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 23, some incorrectly coded applications might hang because of buffering assumptions. For this reason, this optimization is not enabled by default for `MPI_send`, but can be turned on by the user at run time by using the `MPI_BUFFER_MAX` environment variable. The following steps can be taken by the application developer to increase the opportunities for use of this 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 or, in the case of a partitioned system, the processes may 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 more details on how to set the run-time environment, see "Avoiding Message Buffering - Enabling Single Copy" on page 37.

Note: With the Intel 7.1 compiler, `ALLOCATABLE` arrays are not eligible for single copy, since they do not reside in a globally accessible memory region. This restriction does not apply when using the Intel 8.0/8.1 compilers.

Managing Memory Placement

SGI Altix 4000 and SGI Altix UV 1000 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 laying out shared memory data structures, and the individual MPI processes' address spaces. In general, it is not recommended that the application programmer try to manage memory placement explicitly. There are a number of means to control the placement of the application at run time, however. For more information, see Chapter 8, "Run-time Tuning" on page 35.

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 Altix systems and 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 35.

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" on page 29
- "Using the TotalView Debugger with MPI programs" on page 29
- "Using `idb` and `gdb` with MPI programs" on page 30

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), perform the following:

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

To run an MPT MPI job with Total View Debugger with a batch scheduler, such as, PBS Professional or Torque, perform 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.

Using PerfBoost

To use PerfBoost with an SGI MPT MPI program, first load the `perfboost` environmental module (see Example 6-1 on page 31). 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. Here is a list of 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

Here are some examples using `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 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.

<code>PERFBOOST_VERBOSE</code>	Setting the <code>PERFBOOST_VERBOSE</code> environment variable will enable a message when PerfBoost activates and also when the MPI application is completed through the <code>MPI_Finalize()</code> function. This message merely indicates that the PerfBoost library is active and also when the MPI application completes through the <code>libperfboost.wrapper</code> library.
--------------------------------	---

Note: Some applications will re-direct `stderr` in which case the verbose messages may not appear in the application output.

MPI Supported Functions

SGI PerfBoost supports the commonly used elements of the C & Fortran MPI APIs. If a function is not supported, the job will be aborted and an error printed showing the name of the missing function. Please contact SGI Customer Support Center at <https://support.sgi.com/caselist> to get a missing function scheduled for addition to PerfBoost.

Checkpoint/Restart

MPT 2.02 (or later) supports application checkpoint/restart by using the Berkeley Lab Checkpoint/Restart (BLCR) implementation. This allows applications to periodically save a copy of their state. They can then later resume from that point in time if the application crashes or the job is aborted to free up resources for higher priority jobs.

There are some important limitations to keep in mind, as follows:

- BLCR does not checkpoint the state of any data files that the application may be using.
- Certain MPI features including spawning and one-sided MPI are also not supported when using CPR.
- 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/CheckpointRestart/CheckpointRestart.shtml>

BLCR Installation

To use checkpoint/restart with MPT, BLCR must first be installed. This requires installing the `b1cr-`, `b1cr-libs-`, and `b1cr-kmp-` RPMs. BLCR must then be enabled by root, as follows:

```
% chkconfig b1cr 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 `b1cr-` SRPM. In the `b1cr.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` meta-data 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 -restart my_checkpoint.cps hostC, hostD -np 8 ./a.out
```

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" on page 35
- "Tuning MPI Buffer Resources" on page 36
- "Avoiding Message Buffering – Enabling Single Copy" on page 37
- "Memory Placement and Policies" on page 38
- "Tuning MPI/OpenMP Hybrid Codes" on page 41
- "Tuning for Running Applications Across Multiple Hosts" on page 42
- "Tuning for Running Applications over the InfiniBand Interconnect" on page 44
- "MPI on Altix UV 100 and Altix UV 1000 Systems" on page 45
- "Suspending MPI Jobs" on page 47

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 whose lengths exceed 64 bytes. Longer messages are buffered in a shared memory region to allow for exchange of data between MPI processes. In the SGI MPI implementation, these buffers are divided into two basic pools.

- For messages exchanged between MPI processes within the same host or between partitioned systems when using the XPMEM driver or when there are more than `MPI_BUFS_THRESHOLD` hosts, buffers from the "per process" pool (called the "per proc" pool) are used. Each MPI process is allocated a fixed portion of this pool when the application is launched. Each of these portions is logically partitioned into 16-KB buffers.
- For MPI jobs running across multiple hosts, a second pool of shared memory is available. Messages exchanged between MPI processes on different hosts use this pool of shared memory, called the "per host" pool. The structure of this pool is somewhat more complex than the "per proc" pool.

For an MPI job running on a single host, messages that exceed 64 bytes are handled as follows. For messages with a length of 128k or less, the sender MPI process buffers the entire message. It then delivers a message header (also called a control message) to a mailbox, which is polled by the MPI receiver when an MPI call is made. Upon finding a matching receive request for the sender's control message, the receiver

copies the data out of the shared memory buffer into the application buffer indicated in the receive request. The receiver then sends a message header back to the sender process, indicating that the shared memory buffer is available for reuse. Messages whose length exceeds 128k are broken down into 128k chunks, allowing the sender and receiver to overlap the copying of data to and from shared memory in a pipeline fashion.

Because there is a finite number of these shared memory buffers, this can be a constraint on the 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 the “per proc” pool. Similarly, you can use the `MPI_BUFS_PER_HOST` shell variable to adjust the “per host” pool. You can use the MPI statistics counters to determine if retries for these shared memory buffers are occurring.

For information on the use of these counters, see "MPI Internal Statistics" on page 57. In general, you can avoid excessive numbers of retries for buffers by increasing the number of buffers for the “per proc” pool or “per host” pool. However, you should keep in mind that increasing the number of buffers does consume more memory. Also, increasing the number of “per proc” buffers does potentially increase the probability for cache pollution (that is, the excessive filling of the cache with message buffers). Cache pollution can result in degraded performance during the compute phase of a message passing application.

There are additional buffering considerations to take into account when running an MPI job across multiple hosts. For further discussion of multihost runs, see "Tuning for Running Applications Across Multiple Hosts" on page 42.

For further discussion on programming implications concerning message buffering, see "Buffering" on page 23.

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 described in the following sections. For a complete listing of memory placement related shell variables, see the `MPI(1)` man page.

MPI_DSM_CPULIST

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

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

Note: This feature should not be used with MPI applications that use either of the MPI-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 cuset. If the job is run outside of a cuset, the CPUs specify *cpunum* values beginning with 0 and up to the number of CPUs in the system minus one. When running within a cuset, the default behavior is to interpret the CPU values as relative processor numbers within the cuset.

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 cuset 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 parallelism 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 41, 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 16 .

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 Voltaire 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 15. 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 42 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 42 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.

In addition to the choice of interconnect, you should know that multihost jobs may use different buffers from those used by jobs run on a single host. In the SGI implementation of MPI, the XPMEM interconnect uses the “per proc” buffers while the InfiniBand and TCP interconnects use the “per host” buffers. The default setting for the number of buffers per proc or per host might be too low for many applications. You can determine whether this setting is too low by using the MPI statistics described earlier in this section.

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 `ibhost` software stack from Voltaire 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 Altix 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 2048 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 Altix UV 100 and Altix UV 1000 Systems

The SGI Altix UV 100 and Altix UV 1000 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 Altix UV Hub application-specific integrated circuits (ASICs). The UV Hub is the heart of the SGI Altix UV 1000 or Altix UV 100 system compute blade. Each "processor" is a hyperthread on a particular core within a particular socket. Each Altix UV Hub normally connects to two sockets. All communication between the sockets and the UV Hub uses Intel QuickPath Interconnect (QPI) channels. The Altix UV Hub has four NUMALink 5 ports that connect with the NUMALink 5 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 Altix UV hub, Altix UV compute blades, QPI, and NUMALink 5, see the *SGI Altix UV 1000 System User's Guide* or the *SGI Altix UV 100 System User's Guide*, respectively.

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

- By use of shared memory
- By use of the global reference unit (GRU), part of the Altix 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 Altix 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 39), `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 Altix 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 Altix 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 Altix 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" on page 49
- "Using the `perfcatch` Utility" on page 49
- "MPI_PROFILING_STATS Results File Example" on page 50
- "MPI Performance Profiling Environment Variables" on page 53
- "MPI Supported Profiled Functions"
- "Profiling MPI Applications" on page 55

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 50). 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 Voltaire MPI
-vofed	Supports Voltaire 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

```

9: MPI Performance Profiling

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" on page 56
- "MPI Internal Statistics" on page 57
- "Third Party Products" on page 57

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 0.000000e+00  avg datacnt 0  waits 0  wait time 0.000000e+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 0.000000e+00  avg datacnt 0
waitall calls 11045    time 7.73805e+01  # of Reqs 44078  avg data cnt 137944
barrier calls 680      time 5.133110e+00
alltoall calls 0       time 0.0e+00  avg datacnt 0
alltoallv calls 0      time 0.000000e+00
reduce calls 0         time 0.000000e+00
allreduce calls 4658   time 2.072872e+01
bcast calls 680        time 6.915840e-02
```

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

```
activity on process rank 1
```

```
...
```

MPI Internal Statistics

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

To turn on the displaying of MPI internal statistics, use the `MPI_STATS` environment variable or the `-stats` option on the `mpirun` command. MPI internal statistics are always being gathered, so displaying them does not cause significant additional overhead. In addition, one can sample the MPI statistics counters from within an application, allowing for finer grain measurements. If the `MPI_STATS_FILE` variable is set, when the program completes, the internal statistics will be written to the file specified by this variable. For information about these MPI extensions, see the `mpi_stats` man page.

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

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

For additional information on how to use the MPI statistics counters to help tune the run-time environment for an MPI application, see Chapter 8, "Run-time Tuning" on page 35.

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?" on page 59
- "My code runs correctly until it reaches `MPI_Finalize()` and then it hangs." on page 61
- "I keep getting error messages about `MPI_REQUEST_MAX` being too small." on page 61
- "I am not seeing `stdout` and/or `stderr` output from my MPI application." on page 62
- "How can I get the MPT software to install on my machine?" on page 62
- "Where can I find more information about the SHMEM programming model?" on page 62
- "The `ps(1)` command says my memory use (`SIZE`) is higher than expected." on page 62
- "What does `MPI: could not run executable` mean?" on page 62
- "How do I combine MPI with *insert favorite tool here*?" on page 63
- "Why do I see "stack traceback" information when my MPI job aborts?" on page 64

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

Here are some things to investigate:

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

- To find dynamic link errors, try to run your program without `mpirun`. You will get the “`mpirun` must be used to launch all MPI applications” message, along with any dynamic link errors that might not be displayed when the program is started with `mpirun`.

As a last resort, setting the environment variable `LD_DEBUG` to `all` will display a set of messages for each symbol that `rld` resolves. This produces a lot of output, but should help you find the cause of the link error.

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

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

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

This is almost always caused by `send` or `recv` requests that are either unmatched or not completed. An unmatched request is any blocking `send` for which a corresponding `recv` is never posted. An incomplete request is any nonblocking `send` or `recv` request that was never freed by a call to `MPI_Test()`, `MPI_Wait()`, or `MPI_Request_free()`.

Common examples are applications that call `MPI_Isend()` and then use internal means to determine when it is safe to reuse the send buffer. These applications never call `MPI_Wait()`. You can fix such codes easily by inserting a call to `MPI_Request_free()` immediately after all such `isend` operations, or by adding a call to `MPI_Wait()` at a later place in the code, prior to the point at which the send buffer must be reused.

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

A

Administrating MPT

- finding release notes, 5
 - installation, 5
 - disk space requirements, 6
 - MPT RPM, 6
 - prerequisites, 6
 - installation in an alternate location, 7
 - using a cpio file for installation, 8
 - using dynamic shared libraries to run MPI jobs, 9
- Altix UV Hub, 45
- Argument checking, 29

B

- Berkeley Lab Checkpoint/Restart (BLCR), 33
- installation, 33
 - using with MPT, 34

C

- Cache coherent non-uniform memory access (ccNUMA) systems, 17, 47
- ccNUMA
- See also "cache coherent non-uniform memory access", 17, 47
- checkpoint/restart, 33
- Code hangs, 61
- Combining MPI with tools, 63
- Components, 2
- Configuring MPT
- adjusting file descriptor limits, 11
 - adjusting locked memory limits, 12

- configuring array services, 11
- starting prerequisite Services, 10

D

- Debuggers
- idb and gdb, 30
- Distributed applications, 15

F

- Features, 2
- Frequently asked questions, 59

G

- Getting started, 13
- Global reference unit (GRU), 45

I

- Internal statistics, 57
- Introduction, 1

M

- Memory placement and policies, 38
- Memory use size problems, 62
- MPI 2.2 standard compliance, 2
- MPI jobs, suspending, 47
- MPI launching problems, 62
- MPI on Altix UV systems, 45

- general considerations, 46
- job performance types, 46
- other ccNUMA performance issues, 47

MPI performance profiling, 49

- environment variables, 53
- results file, 50
- supported functions, 54

MPI-2 spawn functions

- to launch applications, 15

MPI_REQUEST_MAX too small, 61

mpirun command

- to launch application, 14

mpirun failing, 59

MPMD applications, 14

MPT software installation, 62

P

PerfBoost, 31

- environment variables, 32
- MPI supported functions, 32
- using, 31

perfcatch utility

- results file, 50
- See also "MPI performance profiling", 49
- using, 49

Profiling interface, 56

Profiling tools

- Jumpshot, 57
- third party, 57
- Vampir, 57

Programs

- compiling and linking, 13
- debugging methods, 29
- launching distributed, 15
- launching multiple, 14
- launching single, 14
- launching with mpirun, 14
- launching with PBS, 16
- launching with Torque, 18
- MPI-2 spawn functions, 15

- SHMEM programming model, 18
- with TotalView, 29

R

- Running MPI Jobs with a workload manager, 16

S

- SHMEM applications, 18
- SHMEM information, 62
- Single copy optimization
 - avoiding message buffering, 37
 - using the XPMEM driver, 38
- Stack traceback information, 64
- stdout and/or stderr not appearing, 62

T

- TotalView, 29
- Troubleshooting, 59
- Tuning
 - avoiding message buffering, 37
 - buffer resources, 36
 - enabling single copy, 37
 - for running applications across multiple hosts, 42
 - for running applications over the InfiniBand Interconnect, 44
 - memory placement and policies, 38
 - MPI/OpenMP hybrid codes, 42
 - reducing run-time variability, 35
 - using dplace, 40
 - using MPI_DSM_CPULIST, 39
 - using MPI_DSM_DISTRIBUTE, 40
 - using MPI_DSM_VERBOSE, 40
 - using the XPMEM driver, 38

U

Unpinning memory, 47
Using PBS Professional

to launch application, 16
Using Torque
to launch application, 18