Message Passing Toolkit: MPI Programmer’s Manual

007–3687–010

CONTRIBUTORS

Original text written by Julie Boney; updates by Jean Wilson, Steven Levine

Illustrations by Chrystie Danzer

Edited by Susan Wilkening

Production by Glen Traefald

Cover design by Sarah Bolles Design, and Dany Galgani, SGI Technical Publications

New Features in This Manual

A number of changes have been made to accomodate the differences between IRIX and Linux implementation of MPI.

Record of Revision

Version	Description
1.0	January 1996 Original Printing. This manual documents the Message Passing Toolkit implementation of the Message Passing Interface (MPI).
1.1	August 1996 This revision supports the Message Passing Toolkit (MPT) 1.1 release.
1.2	January 1998 This revision supports the Message Passing Toolkit (MPT) 1.2 release for UNICOS, UNICOS/mk, and IRIX systems.
1.3	February 1999 This revision supports the Message Passing Toolkit (MPT) 1.3 release for UNICOS, UNICOS/mk, and IRIX systems.
003	February 2000 This revision supports the Message Passing Toolkit (MPT) 1.4 release for IRIX systems.
004	October 2000 This revision supports the Message Passing Toolkit (MPT) 1.4.0.3 release for IRIX and beta release for Linux systems.
005	March 2001 This revision supports the Message Passing Toolkit (MPT) 1.5 release for IRIX and beta release for Linux systems.
006	May 2002 This revision supports the Message Passing Toolkit (MPT) 1.6 release for IRIX and beta release for Linux systems.
007	January 2003 This rewrite supports the Message Passing Toolkit (MPT) 1.6.1 release for Linux systems.

008	February 2003 This rewrite supports the Message Passing Toolkit (MPT) 1.7 release for Linux and IRIX systems.
009	June 2003 This update supports the Message Passing Toolkit (MPT) 1.8 release for Linux and IRIX systems.
010	November 2003 This update supports the Message Passing Toolkit (MPT) 1.9 release for Linux and IRIX systems.

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 Standard Compliance 2
  MPI Components 3
  MPI Features 3

2. Getting Started **5**
Compiling and Linking IRIX MPI Programs 5
Compiling and Linking Linux MPI Programs 6
Using `mpirun` to Launch an MPI Application 7
  Launching a Single Program on the Local Host 7
  Launching a Multiple Program, Multiple Data (MPMD) Application on the Local Host . 8
  Launching a Distributed Application 8
  Using MPI-2 Spawn Functions to Launch an Application 9
Compiling and Running SHMEM Applications on IRIX Systems 9
Compiling and Running SHMEM Applications on Linux Systems 10

3. Programming with SGI MPI **13**
Job Termination and Error Handling 13

MPI_Abort 13
Error Handling 13
MPI_Finalize and Connect Processes 14
Signals 14
Buffering 15
Multithreaded Programming 15
Thread Safety 16
Initialization 17
Query Functions 17
Requests 17
Probes 17
Collectives 18
Exception Handlers 18
Cancellation 18
Internal Statistics 18
Finalization 18
Interoperability with SHMEM 19
Miscellaneous Features of SGI MPI 19
stdin/stdout/stderr 19
MPI_Get_processor_name 19
Programming Optimizations 20
Using MPI Point-to-Point Communication Routines 20
Using MPI Collective Communication Routines 20
Using MPI_Pack/MPI_Unpack 21
Avoiding Derived Data Types 21
Avoiding Wild Cards 21
Avoiding Message Buffering — Single Copy Methods 22
Managing Memory Placement 23

Additional Programming Model Considerations 23

4. Debugging MPI Applications 25

MPI Routine Argument Checking 25

Using the ProDev™ WorkShop Debugger with MPI Programs 25

Setting Breakpoints 26

Finding Windows 27

Continuing and Stepping Processes 27

Rerunning a Process 27

Using TotalView with MPI programs 27

Using `dbx` and `gdb` with MPI programs 27

5. Profiling MPI Applications 29

Using Profiling Tools with MPI Applications 29

SpeedShop 29

`perfex` 30

`profile.pl` 30

`histx+` 31

Profiling Interface 31

MPI Internal Statistics 32

Performance Co-Pilot (PCP) 33

Third Party Products 36

6. Run-time Tuning 37

Reducing Run-time Variability 37

Tuning MPI Buffer Resources 38

Avoiding Message Buffering – Enabling Single Copy 39

Using Global Memory for Single Copy Optimization 39

Using the XPMEM Driver for Single Copy Optimization 40
Memory Placement and Policies 41
`MPI_DSM_CPULIST` 41
`MPI_DSM_DISTRIBUTE` (Linux only) 42
`MPI_DSM_MUSTRUN` (IRIX only) 43
`MPI_DSM_PPM` 43
`MPI_DSM_VERBOSE` 43
`PAGESIZE_DATA` and `PAGESIZE_STACK` 43
Using `dplace` for Memory Placement 44
Tuning MPI/OpenMP Hybrid Codes 44
Tuning for Running Applications Across Multiple Hosts 46

7. Troubleshooting and Frequently Asked Questions 49
What are some things I can try to figure out why `mpirun` is failing? 49
My code runs correctly until it reaches `MPI_Finalize()` and then it hangs. 50
I keep getting error messages about `MPI_REQUEST_MAX` being too small, no matter how large I set it. 51
I am not seeing `stdout` and/or `stderr` output from my MPI application. 51
How can I get the MPT software to install on my machine? 51
Where can I find more information about SHMEM? 52
The `ps`(1) command says my memory use (`SIZE`) is higher than expected. 52
What does `MPI: could not run executable` mean? 52
How do I combine MPI with *insert favorite tool here*? 52
Must I use `MPIO_Wait`() and `MPIO_Test`()? 53
Must I modify my code to replace calls to `MPIO_Wait`() with `MPI_Wait`() and recompile? . 54
Why do I see “stack traceback” information when my MPI job aborts? 54

Index 55

Figures

Figure 5-1 mpivis Tool 34
Figure 5-2 mpimon Tool 35

Tables

Table 3-1 Outline of Improper Dependence on Buffering 15

Table 3-2 Optimized MPI Collectives 21

Table 3-3 `MPI_Send/MPI_Recv` Bandwidth Speedup for Unbuffered vs. Buffered Methods 22

Table 6-1 Inquiry Order for Available Interconnects 46

Table 6-2 NPB FT Class C Running on 128 CPUs 47

About This Manual

This publication documents the SGI implementation of the Message Passing Interface (MPI) supported on Linux systems and SGI MIPS based systems running IRIX release 6.5 or later.

MPI consists of a library, a profiling library, 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*. IRIX systems running MPI applications must also be running Array Services software version 3.1 or later.

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 manual: *SpeedShop User's Guide*

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.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man` *title* on a command line.

Conventions

The following conventions are used throughout this document:

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

...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

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

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  `http://docs.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:

  Technical Publications
  SGI
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Introduction

Message Passing Toolkit (MPT) for Linux and IRIX 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.
- Logically shared, distributed memory (SHMEM) data-passing routines. SHMEM is a distributed, shared memory (SHMEM) library that consists of a set of SGI-proprietary message-passing library routines. These routines help distributed applications efficiently share data between cooperating processes. The model is based on multiple processes having separate address spaces, with the SHMEM-provided ability for one process to access data in another process' address space without interrupting the other process. SHMEM is not a standard like MPI, so SHMEM applications developed on other vendors' hardware might or might not work with the SGI SHMEM implementation.

This chapter provides an overview of the MPI software that is included in the toolkit. This overview includes a description of the MPI-2 Standard features that are provided, a description of the basic components of MPI, and a description of the basic features of MPI. Subsequent chapters address the following topics:

- Chapter 2, "Getting Started", page 5
- Chapter 3, "Programming with SGI MPI", page 13
- Chapter 4, "Debugging MPI Applications", page 25
- Chapter 5, "Profiling MPI Applications", page 29
- Chapter 6, "Run-time Tuning", page 37
- Chapter 7, "Troubleshooting and Frequently Asked Questions", page 49

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 on Linux systems and IRIX systems. The MPI Standard is documented online at the following address:

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

MPI-2 Standard Compliance

The SGI MPI implementation is compliant with the 1.0, 1.1, and 1.2 versions of the MPI Standard specification. In addition, the following MPI-2 features (with section numbers from the MPI-2 Standard specification) are provided:

Feature	**Section**
MPI-2 parallel I/O	9
A subset of MPI-2 one-sided communication routines (`put/get` model)	6
MPI `spawn` functionality	5.3
`MPI_Alloc_mem/MPI_Free_mem`	4.11
Transfer of handles	4.12.4
MPI-2 replacements for deprecated MPI-1 functions	4.14.1
Support for thread safety (IRIX only)	8.7
Extended language bindings for C++ and partial Fortran 90 support	10.1, 10.2.4
Generalized requests	4.5.2
New attribute caching functions	8.8

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 command.

Profiling support is included in the `libmpi.so` libraries. Profiling support replaces all `MPI_`*Xxx* prototypes and function names with `PMPI_`*Xxx* entry points.

MPI Features

The SGI MPI implementation offers a number of significant features that make it the preferred implementation to use on SGI hardware:

- Memory placement within IRIX ccNUMA hosts is handled automatically by the library
- Data transfer optimizations for NUMAlink, including single-copy data transfer
- 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 SHMEM (LIBSMA)
- High performance communication support for partitioned systems via XPMEM
- Thread safety on IRIX systems

Getting Started

This chapter provides procedures for building MPI applications on Linux and IRIX systems. 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.

Compiling and Linking IRIX MPI Programs

To use the 64-bit MPI library, choose one of the following commands, specifying the `mpi` library using the `-l` compiler option on the compiler command line::

```
% CC -64 compute.C -lmpi++ -lmpi
% cc -64 compute.c -lmpi
% f77 -LANG:recursive=on -64 compute.f -lmpi
% f90 -LANG:recursive=on -64 compute.f -lmpi
```

To use the 32-bit MPI library, choose one of the following commands:

```
% CC -n32 compute.C -lmpi++ -lmpi
% cc -n32 compute.c -lmpi
% f77 -n32 compute.f -lmpi
% f90 -n32 compute.f -lmpi
```

If the Fortran 90 compiler version 7.2.1 or later is installed, for compile-time checking of MPI subroutine calls, you can add the `-auto_use` option as follows:

```
% f90 -auto_use mpi_interface -LANG:recursive=on -64 compute.f -lmpi
% f90 -auto_use mpi_interface -n32 compute.f -lmpi
```

If your program does not perform MPI-2 one-sided operations like `put` and `get` to a local Fortran variable or array with the `SAVE` attribute, you can omit the `-LANG:recursive=on` option. Note that MPI-2 one-sided communication is not supported for the 32-bit MPI library, and so `-LANG:recursive=on` is not needed with `-n32`.

If the MPI application also uses OpenMP directives, you should link the application with the libraries listed in the following order:

```
% CC -mp -64 compute.C -lmp -lmpi++ -lmpi
% cc -mp -64 compute.c -lmp -lmpi
% f77 -mp -64 compute.f -lmp -lmpi
% f90 -mp -64 compute.f -lmp -lmpi
```

This order is not required, but under certain cases this order leads to better application performance. For further information about using hybrid applications, see "Tuning MPI/OpenMP Hybrid Codes", page 44.

If the MPI application uses the SGI pthreads library, use the following library order when linking the application:

```
% CC -64 compute.C -lmpi++ -lmpi -lpthread
% cc -64 compute.c -lmpi -lpthread
```

This order is necessary because the SGI MPI library contains internal initialization routines that might be required to be run prior to other initialization routines. The SGI `libpthread.so` library has one of these initialization routines that can conflict with the MPI routines. Use the linkage order shown above to ensure that they do not conflict.

Compiling and Linking Linux 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 use the 64-bit MPI library on Linux systems, 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 on Linux with the Intel compiler, use the following commands:

```
% efc -o myprog myprog.f -lmpi          (Fortran)
% ecc -o myprog myprog.c -lmpi          (C)
```

  For Linux the `libmpi++.so` library is not binary compatible with code generated by g++ 3.0 compilers. For this reason an additional library is supported for g++ 3.0 users as well as Intel C++ 8.0 users. The library is `libg++3mpi++.so` and can be linked in by using `-lg++3mpi++` instead of `-lmpi++`.

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

- To compile Fortran programs on Linux 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 the following command:

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

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.

Note: At the time this manual was written, the `MPI.mod` file included in MPT 1.9 was unusable by Intel efc compiler versions 8 and beyond. The supplied `MPI.mod` file is generated with efc version 7.1, build 20030605 and is accepted only by efc version 7 compilers.

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
```

Note that for IRIX systems running MPMD applications, the executable files must be compiled as either 32-bit or 64-bit applications.

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.

Compiling and Running SHMEM Applications on IRIX Systems

To compile a 64-bit SHMEM application on IRIX systems, choose one of the following commands:

```
% CC -64 compute.C -lsma
% cc -64 compute.c -lsma
% f77 -LANG:recursive=on  -64 compute.f -lsma
% f90 -LANG:recursive=on  -64 compute.f -lsma
```

To use the 32-bit SHMEM library, choose one of the following commands:

```
% CC -n32 compute.C -lsma
% cc -n32 compute.c -lsma
% f77 -LANG:recursive=on  -n32 compute.f -lsma
% f90 -LANG:recursive=on  -n32 compute.f -lsma
```

Note: It is generally not recommended to compile SHMEM applications as 32-bit executable files.

If the Fortran 90 compiler version 7.2.1 or later is installed, to get compile-time checking of MPI subroutine calls, you can add the `-auto_use` option as follows:

```
% f90 -auto_use shmem_interface -LANG:recursive=on -64 compute_shmem.f -lsma
% f90 -auto_use shmem_interface -LANG:recursive=on -n32 compute_shmem.f -lsma
```

If your program does not perform SHMEM one-sided operations like `put` and `get` to a local Fortran variable or array with the `SAVE` attribute, you can omit the `-LANG:recursive=on` option. This option prevents the compiler from holding these variables in registers across a subroutine call.

You do not need to use `mpirun` to launch SHMEM applications unless the MPI library was also linked with the application. Use the `NPES` environment variable to specify the number of SHMEM processes to use when running a SHMEM executable file. For example, the following command runs `shmem_app` on 32 processes:

```
% setenv NPES 32
% ./shmem_app
```

If MPI is also used in the executable file, you must use `mpirun` to launch the application, as if it were an MPI application.

Compiling and Running SHMEM Applications on Linux Systems

To use the 64-bit SHMEM library on Linux systems, 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 on Linux systems with the Intel compiler, use the following commands:

```
% ecc compute.C -lsma
% ecc compute.c -lsma
% efc compute.f -lsma
```

Unlike IRIX systems, with Linux systems you must use `mpirun` to launch SHMEM applications. The `NPES` variable has no effect on SHMEM programs running on Linux. To request the desired number of processes to launch, you must set the `-np` option on `mpirun`.

On Linux, 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.

Chapter 3

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.

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.

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.

There are additional concerns when using signals with multithreaded MPI applications. These are discussed in "Thread Safety", page 16.

Buffering

Most MPI implementations use buffering for overall performance reasons and some programs depend on it. However, you should not assume that there is any message buffering between processes because the MPI Standard does not mandate a buffering strategy. Table 3-1, page 15 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 3-1 Outline of Improper Dependence on Buffering

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

By default, the SGI implementation of MPI uses buffering under most circumstances. Short messages (64 or fewer bytes) are always buffered. Longer messages are also buffered, although under certain circumstances buffering can be avoided. For performance reasons, it is sometimes desirable to avoid buffering. For further information on unbuffered message delivery, see "Programming Optimizations", page 20.

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`.

Note: Multithreaded programming models are currently supported on IRIX systems only.

There are further considerations when using MPI with threads. These are described in the following sections.

Thread Safety

The SGI implementation of MPI on IRIX systems assumes the use of POSIX threads or processes (see the `pthread_create` or the OpenMP (`sprocs`) commands, respectively). Other threading packages might or might not work with this MPI implementation.

Each thread associated with a process can issue MPI calls. However, the rank ID in `send` or `receive` calls identifies the process, not the thread. A thread behaves on behalf of the MPI process. Therefore, any thread associated with a process can receive a message sent to that process.

It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. By using distinct communicators for each thread, the user can ensure that two threads in the same process do not issue conflicting communication calls.

All MPI calls on IRIX 6.5 or later systems are thread-safe. This means that two concurrently running threads can make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.

If you block an MPI call, only the calling thread is blocked, allowing another thread to execute, if available. The calling thread is blocked until the event on which it waits occurs. Once the blocked communication is enabled and can proceed, the call completes and the thread is marked runnable within a finite time. A blocked thread does not prevent progress of other runnable threads on the same process, and does not prevent them from executing MPI calls.

Initialization

To initialize MPI for a program that will run in a multithreaded environment, the user must call the MPI-2 function, `MPI_Init_thread()`. In addition to initializing MPI in the same way as `MPI_Init` does, `MPI_Init_thread()` also initializes the thread environment.

You can create threads before MPI is initialized, but before `MPI_Init_thread()` is called, the only MPI call these threads can execute is `MPI_Initialize`.

Only one thread can call `MPI_Init_thread()`. This thread becomes the main thread. Because only one thread calls `MPI_Init_thread()`, threads must be able to inherit initialization. With the SGI implementation of thread-safe MPI, for proper MPI initialization of the thread environment, a thread library must be loaded before the call to `MPI_Init_thread()`. This means that `dlopen` cannot be used to open a thread library after the call to `MPI_Init_thread()`.

Query Functions

The MPI-2 query function, `MPI_Query_thread()`, is available to query the current level of thread support. The MPI-2 function, `MPI_Is_thread_main()`, can be used to determine whether a thread is the main thread. The main thread is the thread that called `MPI_Init_thread()`.

Requests

More than one thread cannot work on the same request. A program in which two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent `MPI_Wait`{`any|some|all`} calls. In MPI, a request can be completed only once. Any combination of `wait` or `test` that violates this rule is erroneous.

Probes

A receive call that uses source and tag values returned by a preceding call to `MPI_Probe` or `MPI_Iprobe` receives the message matched by the probe call only if there was no other matching receive call after the probe and before that receive. In a multithreaded environment, it is the user's responsibility to use suitable mutual exclusion logic to enforce this condition. You can enforce this condition by making sure that each communicator is used by only one thread on each process.

Collectives

Matching collective calls on a communicator, window, or file handle is performed according to the order in which the calls are issued in each process. If concurrent threads issue such calls on the communicator, window, or file handle, it is the user's responsibility to use interthread synchronization to ensure that the calls are correctly ordered.

Exception Handlers

An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call. The exception handler can be executed by a thread that is distinct from the thread that will return the error code.

Cancellation

If a thread that executes an MPI call is canceled by another thread, or if a thread catches a signal while executing an MPI call, the outcome is undefined. When not executing MPI calls, a thread associated with an MPI process can terminate and can catch signals or be canceled by another thread.

Internal Statistics

The SGI internal statistics diagnostics are not thread-safe. MPI statistics are discussed in Chapter 5, "Profiling MPI Applications", page 29.

Finalization

The call to `MPI_Finalize` occurs on the same thread that initialized MPI (also known as the main thread). It is the user's responsibility to ensure that the call occurs only after all of the processes' threads have completed their MPI calls and have no pending communications or I/O operations.

Interoperability with SHMEM

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, not all MPI processes might be accessible when using SHMEM functions. You can use the `shmem_pe_accessible` function to determine whether a SHMEM call can be used to access data residing in another MPI process. Because SHMEM 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 functions should not be considered thread safe.

For more information about SHMEM, 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`

In this release of SGI MPI, 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.

Some 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.

Two 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`) on platforms on which these are available. 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", page 22. Table 3-2, page 21, lists the MPI collective routines optimized in this implementation.

Table 3-2 Optimized MPI Collectives

Routine	Optimized for Clusters	Optimized for Shared Memory
`MPI_Alltoall`	Yes	Yes
`MPI_Barrier`	Yes	Yes
`MPI_Allreduce`	Yes	No
`MPI_Bcast`	Yes	No

Note: On Altix systems these collectives are optimized across partitions by using the XPMEM driver which is explained in Chapter 6, "Run-time Tuning". These 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 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. Table 3-3, page 22, indicates the relative improvement in bandwidth for a simple ping/pong test using various message sizes. However, as discussed in "Buffering", page 15, 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 application should be built as a 64–bit executable file, or linked explicitly with the SHMEM library.
- The MPI data type on the send side must be a contiguous type.
- The sender and receiver MPI processes must reside on the same host.
- The sender data must be globally accessible. Globally accessible memory includes common block or static memory. Depending on the run-time environment, memory allocated via the Fortran 90 `allocate` statement might also be globally accessible. You can also access globally accessible memory by using the `MPI_Alloc_mem` function. In addition, the SHMEM symmetric heap accessed by using the `shpalloc` or `shmalloc` functions is also globally accessible.

Certain run-time environment variables must be set to enable the unbuffered, single copy method. In addition, on certain platforms, hardware is available to facilitate the single copy method without requiring a message buffer to be globally accessible. For more details on how to set the run-time environment, see "Avoiding Message Buffering – Enabling Single Copy", page 39.

Table 3-3 `MPI_Send/MPI_Recv` Bandwidth Speedup for Unbuffered vs. Buffered Methods

Message Length	O2000	O3000
8KB	1.2	1.1
1MB	1.2	1.2
10MB	1.8	1.6

Managing Memory Placement

Many multiprocessor SGI 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 6, "Run-time Tuning", page 37.

Additional Programming Model Considerations

A number of additional programming options might be worth consideration when developing MPI applications for SGI systems. For example, SHMEM 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. SHMEM 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. If you plan to call MPI in a manner requiring thread safety, see "Thread Safety", page 16. Also, there are special ccNUMA placement considerations to be aware of when running hybrid MPI/OpenMP applications. For further information, see Chapter 6, "Run-time Tuning", page 37.

Debugging MPI Applications

Debugging MPI applications can be more challenging than debugging sequential applications. This chapter presents methods for debugging MPI applications.

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 ProDev™ WorkShop Debugger with MPI Programs

Note: The ProDev WorkShop debugger (also known as CVD) is available on IRIX systems only.

Recent versions of the ProDev WorkShop debugger work well with MPI jobs running within a single host. You can use Debugger to debug MPI applications that make use of MPI-2 spawn functions. To use the Debugger, perform the following steps:

Procedure 4-1 Steps for Using the Debugger

1. Use the following command to bring up the Debugger:

   ```
   % cvd /usr/bin/mpirun
   ```

2. When the Debugger comes, up click on the **Admin** menu and select **Multiprocess View**.
3. When **Multiprocess View** appears, click on the **Config** menu, then the **Preferences** menu.
4. When the **Preferences** menu appears, check the first two unchecked boxes and click on **OK**. So that you do not need to set these menus the next time you bring up the Debugger, you can click on the **Save** button.

5. In the Debugger command window (bottom of the main window), enter the following commands:

   ```
   cvd> set $pendingtraps=true
   cvd> stop pgrp all in MPI_SGI_init
   ```

   (You can also use a function in your `a.out`) file.

6. In the command window (the top of the main window), enter the `mpirun` command with arguments, as in the following example:

   ```
   /usr/bin/mpirun -np 2 a.out
   ```

   Then click on the **Run** button.

7. Watch the **Multiprocess View** window as it forks processes. Eventually, it stops in `MPI_SGI_init` (or your function) in your program and the Debugger focuses on it. If you compiled with the `-g` option, it shows the source.

8. If you did not compile with the `-g` option, you can execute a `file` command to select a certain file and see the source, as in the following example:

   ```
   file ep.f
   ```

For complete details about using the Debugger, see the *ProDev WorkShop: Debugger User's Guide*.

Setting Breakpoints

The `pgrp` attribute on the `stop` command (see Procedure 4-1, step 5, page 26 above) indicates the setting of the breakpoint for any processes in the **Multiprocess View** window (including ones that will be spawned as slaves). You can set breakpoints by clicking just to the left of the line, but by default, they are for that particular process, not all processes in the **Multiprocess View** window.

You can change the process to add `pgrp` by clicking on the **Traps** menu in the Debugger and selecting both of the unchecked boxes. Note that when **Group Trap Default** is set, the `pgrp` attribute is added and when **Stop All Default** is set, the `all` attribute is added. The `all` attribute stops all processes in the Multiprocess View window when any process hits this breakpoint.

Finding Windows

To find various windows, use the **Views** menu. **Call Stack** and **Trap Manager** windows are very useful. You can also type `dbx` commands in the Debugger command window at the bottom of the main window.

Continuing and Stepping Processes

The buttons in the **Multiprocess View** window cause all processes to continue or step. Typically, you will want to use these. The buttons in the Debugger main window are for a single process, unless a button indicates **All**. You can set up the viewing of line numbers from the **Display** menu.

Rerunning a Process

If you want to rerun the process, simply click on the **Run** button. To temporarily turn off breakpoints, use the **Traps** menu in the **Trap Manager** window.

Using TotalView with MPI programs

The syntax for running SGI MPI with Etnus' TotalView is as follows:

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

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

Using dbx and gdb with MPI programs

Because the `dbx` 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 `dbx` or `gdb` to attach to the process. A message is printed to the screen, telling you how to use `dbx` 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. Both of these environment variables are available on IRIX and Linux.

Chapter 5

Profiling MPI Applications

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

Profiling tools such as SpeedShop can be effectively used to assess this performance aspect of message passing applications. 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 chapter, you can learn how to use these various tools with MPI applications.

Using Profiling Tools with MPI Applications

Two of the most common SGI profiling tools are SpeedShop and `perfex`. On Altix, `profile.pl` and `histx+` are commonly used. The following sections describe how to invoke these tools. Performance Co-Pilot (PCP) tools and tips for writing your own tools are also included.

Note: SpeedShop is available on IRIX systems only.

SpeedShop

You can use SpeedShop as a general purpose profiling tool or specific profiling tool for MPI potential bottlenecks. It has an advantage over many of the other profiling tools because it can map information to functions and even line numbers in the user source program. The examples listed below are in order from most general purpose to the most specific. You can use the `-ranks` option to limit the data files generated to only a few ranks.

General format:

```
% mpirun -np 4 ssrun [ssrun_options] a.out
```

Examples:

```
% mpirun -np 32 ssrun -pcsamp a.out       # general purpose, low cost
% mpirun -np 32 ssrun -usertime a.out     # general purpose, butterfly view
% mpirun -np 32 ssrun -bbcounts a.out     # most accurate, most cost, butterfly view
% mpirun -np 32 ssrun -mpi a.out          # traces MPI calls
% mpirun -np 32 ssrun -tlb_hwctime a.out  # profiles TLB misses
```

For further information and examples, see the *SpeedShop User's Guide*.

perfex

You can use `perfex` to obtain information concerning the hardware performance monitors.

General format:

```
% mpirun -np 4 perfex -mp [perfex_options] -o file a.out
```

Example:

```
% mpirun -np 4 perfex -mp -e 23 -o file a.out     # profiles TLB misses
```

Note: `perfex` is available on IRIX systems only.

profile.pl

On Altix systems, you can use `profile.pl` to obtain procedure level profiling as well as information about the hardware performance monitors. For further information, see the `profile.pl`(1) and `pfmon`(1) man pages.

General format:

```
% mpirun -np 4 profile.pl [profile.pl_options] ./a.out
```

Example:

```
% mpirun -np 4 profile.pl -s1 -c4,5 -N 1000 ./a.out
```

histx+

On Altix systems, `histx+` is a small set of tools that can assist with performance analysis and bottlenect identification.

General formats for `histx` (Histogram) and `lipfpm` (Linux IPF Performance Monitor):

% **mpirun -np 4 histx** [*histx_options*] **./a.out**

% **lipfpm** [*lipfpm_options*] **mmpirun -np 4 ./a.out**

Examples:

```
% mpirun -np 4 histx -f -o histx.out ./a.out
```

```
% lipfpm -f -e LOADS_RETIRED -e STORES_RETIRED mpirun -np 4 ./a.out
```

Profiling Interface

You can write your own profiling by using the MPI-1 standard `PMPI_*` calls. In addition, either within your own profiling library or within the application itself you can use the `MPI_Wtime` function call to time specific calls or sections of your code.

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

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

activity on process rank 0
comm_rank calls 1      time 8.800002e-06
get_count calls 0      time 0.000000e+00
ibsend calls    0      time 0.000000e+00
probe calls     0      time 0.000000e+00
recv calls      0      time 0.00000e+00   avg datacnt 0   waits 0       wait time 0.00000e+00
irecv calls     22039  time 9.76185e-01   datacnt 23474032 avg datacnt 1065
send calls      0      time 0.000000e+00
ssend calls     0      time 0.000000e+00
isend calls     22039  time 2.950286e+00
```

```
wait calls      0      time 0.00000e+00   avg datacnt 0
waitall calls   11045  time 7.73805e+01   # of Reqs 44078  avg data  cnt 137944
barrier calls   680    time 5.133110e+00
alltoall calls  0      time 0.0e+00       avg datacnt 0
alltoallv calls 0      time 0.000000e+00
reduce calls    0      time 0.000000e+00
allreduce calls 4658   time 2.072872e+01
bcast calls     680    time 6.915840e-02
gather calls    0      time 0.000000e+00
gatherv calls   0      time 0.000000e+00
scatter calls   0      time 0.000000e+00
scatterv calls  0      time 0.000000e+00

activity on process rank 1
...
```

SGI provides a freeware MPI profiling library that might be useful as a starting point for developing your own profiling routines. You can obtain this software at `http://freeware.sgi.com/index-by-alpha.html`.

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. For information about these MPI extensions, see the `mpi_stats` man page.

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

- To determine if there are enough internal buffers and if processes are waiting (retries) to aquire them
- To determine if single copy optimization is being used for point-to-point or collective calls
- To determine additional resource contention when using GSN networks

For additional information on how to use the MPI statistics counters to help tune the run-time environment for an MPI application, see Chapter 6, "Run-time Tuning", page 37.

Performance Co-Pilot (PCP)

In addition to the tools described in the preceding sections, you can also use the MPI agent for Performance Co-Pilot (PCP) to profile your application. The two additional PCP tools specifically designed for MPI are `mpivis` and `mpimon`. These tools do not use trace files and can be used live or can be logged for later replay.

For more information about configuring and using these tools, see the PCP tutorial in `/var/pcp/Tutorial/mpi.html`. Following are examples of the `mpivis` and `mpimon` tools.

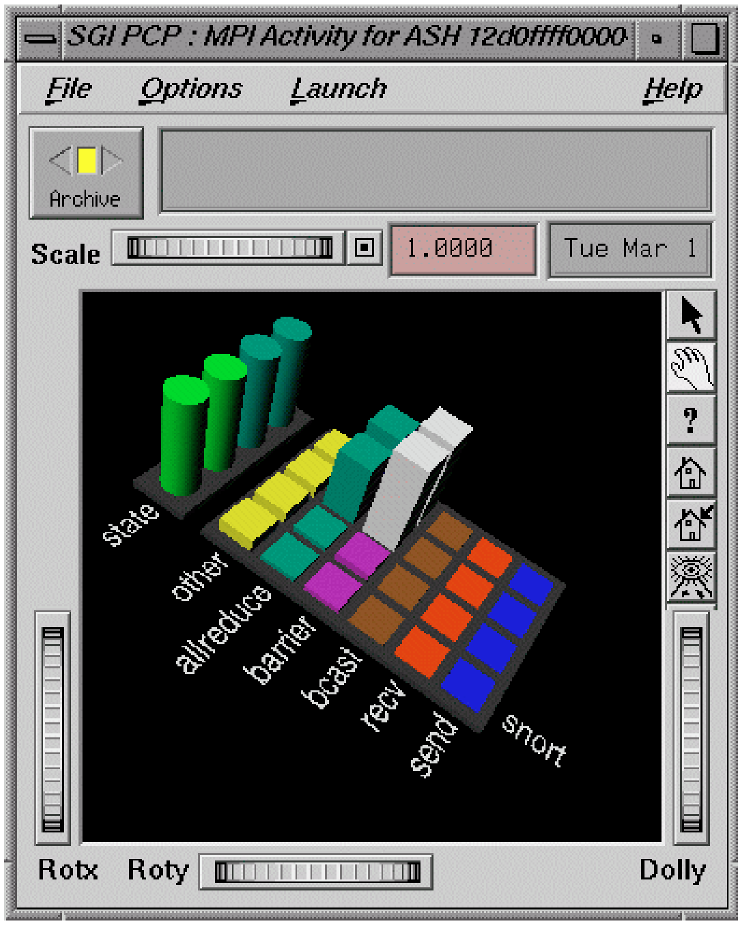

Figure 5-1 mpivis Tool

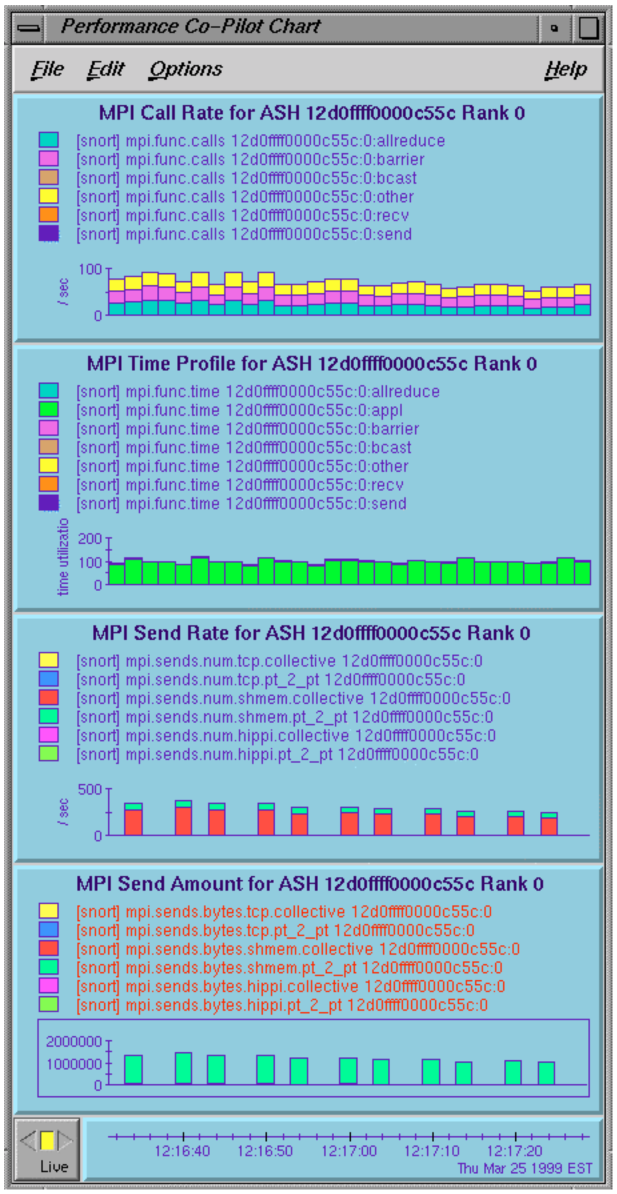
Performance Co-Pilot Chart
File
Edit
Options
Help
MPI Call Rate for ASH 12d0ffff0000c55c Rank 0
[snort] mpi.func.calls 12d0ffff0000c55c:0:allreduce
[snort] mpi.func.calls 12d0ffff0000c55c:0:barrier
[snort] mpi.func.calls 12d0ffff0000c55c:0:bcast
[snort] mpi.func.calls 12d0ffff0000c55c:0:other
[snort] mpi.func.calls 12d0ffff0000c55c:0:recv
[snort] mpi.func.calls 12d0ffff0000c55c:0:send
/ sec
100
0
MPI Time Profile for ASH 12d0ffff0000c55c Rank 0
[snort] mpi.func.time 12d0ffff0000c55c:0:allreduce
[snort] mpi.func.time 12d0ffff0000c55c:0:appl
[snort] mpi.func.time 12d0ffff0000c55c:0:barrier
[snort] mpi.func.time 12d0ffff0000c55c:0:bcast
[snort] mpi.func.time 12d0ffff0000c55c:0:other
[snort] mpi.func.time 12d0ffff0000c55c:0:recv
[snort] mpi.func.time 12d0ffff0000c55c:0:send
time utilizatio
200
100
0
MPI Send Rate for ASH 12d0ffff0000c55c Rank 0
[snort] mpi.sends.num.tcp.collective 12d0ffff0000c55c:0
[snort] mpi.sends.num.tcp.pt_2_pt 12d0ffff0000c55c:0
[snort] mpi.sends.num.shmem.collective 12d0ffff0000c55c:0
[snort] mpi.sends.num.shmem.pt_2_pt 12d0ffff0000c55c:0
[snort] mpi.sends.num.hippi.collective 12d0ffff0000c55c:0
[snort] mpi.sends.num.hippi.pt_2_pt 12d0ffff0000c55c:0
/ sec
500
0
MPI Send Amount for ASH 12d0ffff0000c55c Rank 0
[snort] mpi.sends.bytes.tcp.collective 12d0ffff0000c55c:0
[snort] mpi.sends.bytes.tcp.pt_2_pt 12d0ffff0000c55c:0
[snort] mpi.sends.bytes.shmem.collective 12d0ffff0000c55c:0
[snort] mpi.sends.bytes.shmem.pt_2_pt 12d0ffff0000c55c:0
[snort] mpi.sends.bytes.hippi.collective 12d0ffff0000c55c:0
[snort] mpi.sends.bytes.hippi.pt_2_pt 12d0ffff0000c55c:0
2000000
1000000
0
Live
12:16:40
12:16:50
12:17:00
12:17:10
12:17:20
Thu Mar 25 1999 EST

Figure 5-2 mpimon Tool

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.

A better approach is to use a general purpose profiling tool such as SpeedShop to locate the problem area and then to turn on and off the tracing just around the problematic areas of your code. With this approach, the display tools can better handle the amount of data that is generated.

Chapter 6

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.

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. Both the Linux and IRIX kernels use 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. One solution IRIX provides to solve this problem is the `boot_cpuset`(4) command. The `boot_cpuset` (`man boot_cpuset`) capability allows creation of a cpuset that contains the `init` process and all of its descendants, effectively preventing system functions from interfering with batch jobs running on the rest of the machine.

- Avoid interference with other applications. You can use cpusets or cpumemsets to address this problem also. You can use cpusets (for IRIX) or cpumemsets (for Linux) to effectively partition a large, distributed memory host in a fashion that minimizes interactions between jobs running concurrently on the system. See *IRIX Admin: Resource Administration* and the *Linux Resource Administration Guide* for information about cpusets and cpumemsets.

- 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, LSF from Platform Computing or PBSpro from Veridan. 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, 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 16 KB 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 16 KB are broken down into 16-KB 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", page 32. 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", page 46.

For further discussion on programming implications concerning message buffering, see "Buffering", page 15.

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_Bcase`, `MPI_Allreduce`, and `MPI_Reduce`, this optimization is enabled by default for large message sizes.

Using Global Memory for Single Copy Optimization

On IRIX systems, when global memory is used for single copy optimization, the sender's message data must reside in globally accessible memory. Globally accessible memory includes common block or static memory and memory allocated with the Fortran 90 `allocate` statement or `MPI_Alloc_mem` (with the `SMA_GLOBAL_ALLOC` environment variable set). In addition, applications linked against the SHMEM library can also access the LIBSMA symmetric heap via the `shpalloc` or `shmalloc` functions. Consequently, use of this feature might require changes to the application. Additional restrictions are described in "Avoiding Message Buffering — Single Copy Methods", page 22.

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 most applications running on a single host. To disable default single copy, use the `MPI_DEFAULT_SINGLE_COPY_OFF` environment variable.

Using the XPMEM Driver for Single Copy Optimization

MPI can take advantage of the XPMEM driver, a special cross-partition device driver, available on both IRIX and Linux systems, that allows the operating system to copy data between two processes within the same host or across partitions.

On systems running IRIX, this feature requires IRIX 6.5.13 or greater and is available only on Origin 300 and 3000 series servers. This option is not available on servers running Trusted IRIX. The MPI library uses the XPMEM driver to enhance single copy optimization (within a host) to eliminate some of the restrictions with only a slight (less that 5 percent) performance cost over the more restrictive single copy optimization using globally accessible memory.

You can enable this optimization if you set the `MPI_XPMEM_ON` and `MPI_BUFFER_MAX` environment variables. Note that if the sender data resides in globally accessible memory, the data is copied using a `bcopy` process. Otherwise, the XPMEM driver is used to transfer the data. Using the XPMEM form of single copy is less restrictive in that the sender's data is not required to be globally accessible. It is available for ABI N32 as well as ABI 64. This optimization also can be used to transfer data between two different executable files on the same host or two different executable files across IRIX partitions.

Note: Use of the XPMEM driver disables the ability to checkpoint/restart an MPI job.

On IRIX systems, under certain conditions, the XPMEM driver can take advantage of the block transfer engine (BTE) to provide increased bandwidth. In addition to having `MPI_BUFFER_MAX` and `MPI_XPMEM_ON` set, the send and receive buffers must be cache-aligned and the amount of data to transfer must be greater than or equal to `MPI_XPMEM_THRESHOLD`. The default value for `MPI_XPMEM_THRESHOLD` is 8192.

On systems running Linux, use of the XPMEM driver is required to support single-copy message transfers between two processes within the same host or across partitions. On Linux systems, during job startup, MPI uses the XPMEM driver (via the xpmem kernel module) to map memory from one MPI process onto another. The mapped areas include the static region, private heap, and stack 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 the XPMEM driver to copy the data.

Memory mapping is enabled by default on Linux. 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, and certain collective optimizations.

Memory Placement and Policies

The MPI library takes advantage of NUMA placement functions that are available on IRIX and Linux systems. Usually, the default placement is adequate. Under certain circumstances, however, you might want to modify this default behavior. The easiest way to do this is by setting one or more MPI placement shell variables. Several of the most commonly used of these variables are discribed in the following sections. For a complete listing of memory placement related shell variables, see the `MPI`(1) man page.

MPI_DSM_CPULIST

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

This setting is treated as a comma and/or hyphen delineated ordered list that specifies a mapping of MPI processes to CPUs. If running across multiple hosts, the per host components of the CPU list are delineated by colons.

Note: This feature will not be compatible with job migration features available in future IRIX releases. In addition, 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.
8-15,32-39	Place the MPI processes 0 through 7 on CPUs 8 to 15. Place the MPI processes 8 through 15 on CPUs 32 to 39.
39-32,8-15	Place the MPI processes 0 through 7 on CPUs 39 to 32. Place the MPI processes 8 through 15 on CPUs 8 to 15.
8-15:16-23	Place the MPI processes 0 through 7 on the first host on CPUs 8 through 15. Place MPI processes 8 through 15 on CPUs 16 to 23 on the second host.

Note that the process rank is the `MPI_COMM_WORLD` rank. The interpretation of the CPU values specified in the `MPI_DSM_CPULIST` depends on whether the MPI job is being run within a cpuset. If the job is run outside of a cpuset, the CPUs specify *cpunum* values given in the hardware graph (`hwgraph`(4)). When running within a cpuset, the default behavior is to interpret the CPU values as relative processor numbers within the cpuset. To specify *cpunum* values instead, you can use the `MPI_DSM_CPULIST_TYPE`(`MPI`(1)) shell variable.

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. To insure linking of the MPI processes to the designated processors, you should also set the `MPI_DSM_MUSTRUN` shell variable on IRIX only.

`MPI_DSM_DISTRIBUTE` (Linux only)

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. On Linux systems, 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. On Linux systems, therefore, it is recommended that this variable be used only if running within a cpumemset or on a dedicated system.

MPI_DSM_MUSTRUN (IRIX only)

Use the `MPI_DSM_MUSTRUN` shell variable to ensure that each MPI process will get a physical CPU and memory on the node to which it was assigned. It has been observed that using this shell variable has led to improved performance, especially on IRIX systems running version 6.5.7 and earlier. With the MPT 1.8 release, the `MPI_DSM_MUSTRUN` variable is deprecated on Linux. Use `MPI_DSM_DISTRIBUTE` instead.

MPI_DSM_PPM

The `MPI_DSM_PPM` shell variable allows you to specify the number of MPI processes to be placed on a node. Memory bandwidth intensive applications can benefit from placing fewer MPI processes on each node of a distributed memory host. On Origin 200 and Origin 2000 series servers, the default is to place two MPI processes on each node. On Origin 300 and Origin 3000 series servers, the default is four MPI processes per node. You can use the `MPI_DSM_PPM` shell variable to change these values. On Origin 300 and Origin 3000 series servers, setting `MPI_DSM_PPM` to 2 places one MPI process on each memory bus. On SGI Altix 3000 systems, setting `MPI_DSM_PPM` to 1 places one MPI process on each node.

MPI_DSM_VERBOSE

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

PAGESIZE_DATA and PAGESIZE_STACK

You can use the `PAGESIZE_DATA` and `PAGESIZE_STACK` variables to request nondefault page sizes (in kilobytes). Setting these variables can be helpful for applications that experience frequent TLB misses. You can ascertain this condition by using the ssrun or perfex profiling tools. However, these variables should be used with caution. Generally, system administrators do not configure the system to have many large pages per node. If very large page sizes are requested, you might lose good memory locality if the operating system is able to satisfy the large page request only with remote memory.

Note: Because these variables are associated with NUMA placement, disabling NUMA placement via the `MPI_DSM_OFF` shell variable disables the use of these page size shell variables.

Note: These shell variables are currently not available on Linux systems.

Using `dplace` for Memory Placement

The `dplace` tool offers another means of specifying the placement of MPI processes within a distributed memory host. This tool is available on both Linux and IRIX systems. Starting with IRIX 6.5.13, `dplace` 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. If this interoperability feature is undesirable, you can set the `MPI_DPLACE_INTEROP_OFF` shell variable.

For instructions on how to use `dplace` with MPI, see the `dplace`(1) man page.

Tuning MPI/OpenMP Hybrid Codes

Hybrid MPI/OpenMP applications might require special memory placement features to operate efficiently on ccNUMA Origin servers. This section describes a preliminary method for achieving this memory placement.

The basic idea is to space out the MPI processes to accommodate the OpenMP threads associated with each MPI process. In addition, assuming a particular ordering of library `init` code (see the `DSO` man page), this method employs procedures to insure that the OpenMP threads remain close to the parent MPI process. This type of placement has been found to improve the performance of some hybrid applications significantly.

To take partial advantage of this placement option, the following requirements must be met:

- When running the application, you must set the MPI_OPENMP_INTEROP shell variable.
- To compile the application, you must use a MIPSpro compiler and the -mp compiler option. This hybrid model placement option is not available with other compilers.
- The application must run on an Origin 300 or Origin 3000 series server.

To take full advantage of this placement option, you must be able to link the application such that the libmpi.so init code is run before the libmp.so init code. For instructions on how to link the hybrid application, see "Compiling and Linking IRIX MPI Programs", page 5. This linkage issue has been removed in the MIPspro 7.4 (and later versions) compilers. It may, however, remain in earlier compiler versions.

You can use an additional memory placement feature for hybrid MPI/OpenMP applications by using the MPI_DSM_PLACEMENT shell variable. Specification of a "threadroundrobin" policy results in the parent MPI process stack, data, and heap memory segments being spread across the nodes on which the child OpenMP threads are running.

MPI reserves nodes for this hybrid placement model based on the number of MPI processes and the number of OpenMP threads per process, rounded up to the nearest multiple of 4. For example, if 6 OpenMP threads per MPI process are going to be used for a 4 MPI process job, MPI will request a placement for 32 (4 X 8) CPUs on the host machine. You should take this into account when requesting resources in a batch environment or when using cpusets. In this implementation, it is assumed that all MPI processes start with the same number of OpenMP threads, as specified by the OMP_NUM_THREADS or equivalent shell variable at job startup.

This placement is not recommended if you set _DSM_PPM to a non-default value (for more information, see pe_environ). Also, it is suggested that the mustrun shell variables (MPI_DSM_MUSTRUN and _DSM_MUSTRUN) not be set when using this placement model.

On Linux systems the MPI_OPENMP_INTEROP variable is supported. However, the OpenMP threads are not actually pinned to a CPU but are free to migrate to any of the CPUs in the OpenMP thread group for each MPI rank. The pinning of the OpenMP thread to a specific CPU will be supported in a future release.

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.

IRIX hosts can be clustered using a variety of high performance interconnects. You can use the XPMEM interconnect to cluster Origin 300 and Origin 3000 series servers as partitioned systems. Other high performance interconnects include GSN and Myrinet. If none of these interconnects is available, MPI relies on TCP/IP to handle MPI traffic between hosts.

Systems running Linux can use the XPMEM interconnect to cluster hosts as partitioned systems, or rely on 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", page 8. 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 6-1, page 46 specifies the order in which MPI probes for available interconnects.

Table 6-1 Inquiry Order for Available Interconnects

Interconnect	Default Order of Selection	Environment Variable to Require Use	Environment Variable for Specifying Device Selection
XPMEM	1	`MPI_USE_XPMEM`	NA
GSN	2	`MPI_USE_GSN`	`MPI_GSN_DEVS`
Myrinet(GM)	3	`MPI_USE_GM`	`MPI_GM_DEVS`
TCP/IP	5	`MPI_USE_TCP`	NA

The third column of Table 6-1, page 46, also indicates the environment variable you can set to pick a particular interconnect other than the default. For example, suppose you want to run an MPI job on a cluster supporting both GSN and Myrinet (GM) interconnects. By default, the MPI job would try to run over the GSN interconnect. If for some reason you wanted to use the Myrinet (GM) interconnect, you would set the `MPI_USE_GM` shell variable before launching the job. This would cause the MPI library to attempt to run the job using the Myrinet (GM) interconnect. If the Myrinet interconnect cannot be used, the job will fail.

The XPMEM interconnect is an exception in that it does not require that all hosts in the MPI job need to be reachable via the XPMEM device. Message traffic between hosts not reachable via XPMEM will go over the next fastest interconnect. Also, when you specify a particular interconnect to use, you can set the `MPI_USE_XPMEM` variable in addition to one of the other four choices.

In general, to insure the best performance of the application, you should allow MPI to pick the fastest available interconnect.

When running in cluster mode, be careful about setting the `MPI_BUFFER_MAX` value too low. Setting it less than 16384 bytes could lead to a significant increase in the number of small control messages sent over the interconnect, possibly leading to performance degradation.

In addition to the choice of interconnect, you should know that multihost jobs use different buffers from those used by jobs run on a single host. In the SGI implementation of MPI, all of the previously mentioned interconnects rely on the "per host" buffers to deliver long messages. The default setting for the number of buffers 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.

In particular, you should examine the metric for retries allocating MPI per host buffers. High retry counts usually indicate that the `MPI_BUFS_PER_HOST` shell variable should be increased. Table 6-2, page 47 provides an example of application performance as a function of the number of "per host" message buffers. Here, the Fourier Transform (FT) class C benchmark was run on a cluster of four Origin 300 servers (32 CPUs each) using Myrinet. Note that the performance improves by almost a factor of three by increasing the `MPI_BUFS_PER_HOST` from the default of 32 buffers to 128 buffers per host.

Table 6-2 NPB FT Class C Running on 128 CPUs

`MPI_BUFS_PER_HOST` Setting	Execution Time (secs)
32 (default)	280
64	144

MPI_BUFS_PER_HOST Setting	Execution Time (secs)
128	108
256	104

When considering these MPI statistics, GSN users should also examine the counter for retries allocating MPI per host message headers. In cases in which this metric indicates high numbers of retries, it might be necessary to increase the MPI_MSGS_PER_HOST shell variable . Myrinet (GM) does not use this resource.

When using GSN or Myrinet high performance networks, MPI attempts to use all adapters (cards) available on each host in the job. You can modify this behavior by specifying specific adapter(s) to use. The fourth column of Table 6-1, page 46 indicates the shell variable to use for a given network. For details on syntax, see the MPI man page.

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.

Chapter 7

Troubleshooting and Frequently Asked Questions

This chapter provides answers to some common problems users encounter when starting to use SGI's MPI, as well as answers to other frequently asked questions.

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

Here are some things to investigate:

- On IRIX systems, look at the last few lines in `/var/adm/SYSLOG` for any suspicious errors or warnings. On Linux systems, look in `/var/log/messages`. For example, if your application tries to pull in a library that it cannot find, a message should appear here.
- Be sure that you did not misspell the name of your application.
- To find `rld`/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 `rld` link errors that might not be displayed when the program is started with `mpirun`.
- 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/etc/ascheck` to verify that your array is configured correctly.
- Be sure that you can execute `rsh` (or `arshell`) to all of the hosts that you are trying to use without entering a password. This means that either `/etc/hosts.equiv` or `~/.rhosts` must be modified to include the names of every host in the MPI job. Note that using the `-np` syntax (i.e. no hostnames) is equivalent to typing `localhost`, so a *localhost* entry will also be needed in one of the above two files.

- On IRIX systems, if you are using an `mpt` module to load MPI, try loading it directly from within your `.cshrc` file instead of from the shell. If you are also loading a MIPSpro module, be sure to load it after the `mpt` module.
- Use the `-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.

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, no matter how large I set it.

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.

Beginning with our MPT 1.2/MPI 3.1 release, 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.

Beginning with MPT 1.5.2, 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?

Message-Passing Toolkit software releases can be obtained at the SGI Software Download page at

```
http://www.sgi.com/products/evaluation/
```

Where can I find more information about SHMEM?

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, when running on IRIX hosts, MPI calls SHMEM 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.

With Array Services 3.2 or later and MPT 1.3 or later, many scenarios that generated this error message are now improved to be more descriptive.

Prior to Array Services 3.2, no diagnostic information was directly available. This was due to the highly decoupled interface between `mpirun` and `arrayd`.

`mpirun` 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
```

Must I use `MPIO_Wait()` and `MPIO_Test()`?

Beginning with MPT 1.8, MPT has unified the I/O requests generated from nonblocking I/O routines (such as `MPI_File_iwrite`()) and MPI requests from nonblocking message-passing routines (for example, `MPI_Isend`()). Formerly, these were different types of request objects and needed to be kept separate (one was called `MPIO_Request` and the other, `MPI_Request`). Under MPT 1.8 and later, however, this distinction is no longer necessary. You can freely mix request objects returned from I/O and MPI routines in calls to `MPI_Wait`(), `MPI_Test`(), and their variants.

Must I modify my code to replace calls to `MPIO_Wait()` with `MPI_Wait()` and recompile?

No. If you have an application that you compiled prior to MPT 1.8, you can continue to execute that application under MPT 1.8 and beyond without recompiling. Internally, MPT uses the unified requests, and for example, translates calls to `MPIO_Wait`() into calls to `MPI_Wait`().

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

This is a new feature beginning with MPT 1.8. 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

Argument checking, 25

B

Breakpoint setting, 26

C

Code hangs, 50
Combining MPI with tools, 52
Components, 3
Continuing processes, 27

D

Debuggers
  dbx and gdb, 27
  ProDev WorkShop , 25
Distributed applications, 8

F

Features, 3
Frequently asked questions, 49

G

Getting started, 5

H

histx tool, 31

I

Internal statistics, 32
Introduction, 1

M

Memory placement and policies, 41
Memory use size problems, 52
Modifying code for MPI_Wait, 54
MPI launching problems, 52
MPI-2 compliance, 2
MPI-2 spawn functions
  to launch applications, 9
MPI_REQUEST_MAX too small, 51
mpimon tool, 33
mpirun command
  to launch application, 7
mpirun failing, 49
mpivis tool, 33
MPMD applications, 8
MPT software installation, 51

P

perfex tool, 30
Performance Co-Pilot (PCP), 33
ProDev WorkShop debugger, 25
profile.pl tool, 30
Profiling interface, 31

Profiling library freeware, 32
Profiling tools
  Jumpshot, 36
  mpimon, 33
  mpivis, 33
  perfex, 29
  SpeedShop, 29
  third party, 36
  Vampir, 36
Programs
  compiling and linking, IRIX, 5
  compiling and linking, Linux, 6
  debugging methods, 25
  launching distributed, 8
  launching multiple, 8
  launching single, 7
  launching with mpirun, 7
  MPI-2 spawn functions, 9
  SHMEM, 9, 10
  with TotalView, 27

R

Rerunning processes, 27

S

SHMEM applications for IRIX, 9
SHMEM applications for Linux, 10
SHMEM information, 52
Single copy optimization
  avoiding message buffering, 39
  using global memory, 39
  using the XPMEM driver, 40
SpeedShop tool, 29
Stack traceback information, 54
stdout and/or stderr not appearing, 51
Stepping processes, 27

T

TotalView, 27
Troubleshooting, 49
Tuning
  avoiding message buffering, 39
  buffer resources, 38
  enabling single copy, 39
  for running applications across multiple hosts, 46
  memory placement and policies, 41
  MPI/OpenMP hybridcodes, 44
  reducing run-time variability, 37
  using dplace, 44
  using global memory, 39
  using MPI_DSM_CPULIST, 41
  using MPI_DSM_DISTRIBUTE, 42
  using MPI_DSM_MUSTRUN, 43
  using MPI_DSM_PPM, 43
  using MPI_DSM_VERBOSE, 43
  using PAGESIZE_DATA and PAGESIZE_STACK, 43
  using the XPMEM driver, 40

U

Using MPIO_Wait and MPIO_Test, 53

W

Window use, 27