Message Passing Toolkit: MPI Programmer’s Manual

007–3687–006

CONTRIBUTORS
Written by Julie Boney
Illustrations by Chrystie Danzer
Production by Glen Traefald

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

New Features

This revision of the *Message Passing Toolkit: MPI Programmer's Manual* supports the 1.6 release of the Message Passing Toolkit (MPT) for IRIX and Linux systems. The following new features are documented in this version of the manual:

- MPI-2 `MPI_Spawn` functionality
- New environment variables
- New optimization chapter
- New frequently asked questions chapter

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.

Contents

Figures

Tables

About This Manual

This publication documents the SGI version 1.6 implementation of the Message Passing Interface (MPI) supported on SGI MIPS based systems running IRIX release 6.5 or later and as a beta release on Linux systems. MPI is a component of the SGI Message Passing Toolkit (MPT).

IRIX systems running MPI applications must also be running Array Services software version 3.1 or later. MPI consists of a library, a profiling library, and commands that support MPI. The MPT 1.6 release is a software package that supports parallel programming across a network of computer systems through a technique known as *message passing*.

Related Publications and Other Sources

The *Message Passing Toolkit: PVM Programmer's Guide* contains additional information that might be helpful. You can obtain this document or any other SGI documentation from the SGI Technical Publications Library at `http://techpubs.sgi.com`.

Material about MPI is available from a variety of other sources. Some of these, particularly Web pages, 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.
- As text through the IRIS InSight library (for customers with access to this tool).

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

Newsgroup: `comp.parallel.mpi`

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. The following list describes the identifiers:

1	User commands
1B	User commands ported from BSD
2	System calls
3	Library routines, macros, and opdefs
4	Devices (special files)
4P	Protocols
5	File formats
7	Miscellaneous topics
7D	DWB-related information
8	Administrator commands

Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.

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.

SGI systems include all Linux systems and all MIPS based systems that run IRIX 6.5 or later.

Reader Comments

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

You can contact us 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 World Wide Web page:

  `http://techpubs.sgi.com`

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

- Send mail to the following address:

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

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

We value your comments and will respond to them promptly.

Chapter 1

Overview

The Message Passing Toolkit (MPT) for IRIX and Linux 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 1.6 package contains the following components and the appropriate accompanying documentation:

- Message Passing Interface (MPI)
- Logically shared, distributed memory (SHMEM) data-passing routines (IRIX only)

The Message Passing Interface (MPI) is a standard specification for a message passing interface, allowing portable message passing programs in Fortran and C languages.

This chapter provides an overview of the MPI software that is included in the toolkit, a description of the basic MPI components, and a list of general steps for developing an MPI program. Subsequent chapters address the following topics:

- Building MPI applications
- Using `mpirun` to execute applications
- Thread-safe MPI
- Setting environment variables
- Optimization and tuning
- Frequently asked questions

MPI Overview

MPI is a standard specification for a message passing interface, allowing portable message passing programs in Fortran and C languages. 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 available from the IRIS InSight library (for customers who have access to that tool), and is documented online at the following address:

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

The SGI MPT MPI implementation is compliant with the 1.0, 1.1, and 1.2 versions of the MPI standard specification. In addition, the following features from the MPI-2 standard specification are provided:

- MPI-2 parallel I/O, as described in section 9 of the MPI-2 standard
- MPI-2 one sided communication (put/get model), as described in section 6 of the MPI-2 standard
- `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`, as described in section 5.3 of the MPI-2 standard (IRIX only)
- `MPI_Alloc_mem/MPI_Free_mem`, as described in section 4.11 of the MPI-2 standard (IRIX only)
- Transfer handles, as described in section 4.12.4 of the MPI-2 standard
- MPI-2 replacements for deprecated MPI-1 functions, as described in section 4.14.1 of the MPI-2 standard

MPI Components

The MPI library is provided as a dynamic shared object (DSO) (a file with a name that ends in `.so`). The basic components that are necessary for using MPI are the `libmpi.so` library, the include files, and the `mpirun`(1) 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 Program Development

To develop a program that uses MPI, you must perform the following steps:

Procedure 1-1 Steps for MPI program development

1. Add MPI function calls to your application for MPI initiation, communications, and synchronization. For descriptions of these functions, see the online man pages or *Using MPI: Portable Parallel Programming with the Message-Passing Interface* or the MPI standard specification.
2. Build programs for the systems that you will use, as described in Chapter 2, "Building MPI Applications", page 5.
3. Execute your program by using the `mpirun`(1) command (see Chapter 3, "Using `mpirun` to Execute Applications", page 7).

Note: For information on how to execute MPI programs across more than one host or how to execute MPI programs that consist of more than one executable file, see Chapter 2, "Building MPI Applications", page 5.

Building MPI Applications

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

After you have added MPI function calls to your program, as described in Procedure 1-1, step 1, page 3, you can compile and link the IRIX or Linux program, as described in the following sections.

Compiling and Linking IRIX Programs

To use the 64-bit MPI library, choose one of the following commands:

```
CC -64 compute.C -lmpi++ -lmpi
```

```
cc -64 compute.c -lmpi
```

```
f77 -64 compute.f -lmpi
```

```
f90 -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, you can add the `-auto_use` option as follows to get compile-time checking of MPI subroutine calls:

```
f90 -auto_use mpi_interface -64 compute.f -lmpi
```

```
f90 -auto_use mpi_interface -n32 compute.f -lmpi
```

Compiling and Linking Linux Programs

The default locations for the include files, the `.so` files, the `.a` files, and the `mpi_launch` and `mpirun` commands 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 IA64 systems, choose one of the following commands:

```
g++ -o myprog myproc.C -lmpi++ -lmpi
```

```
gcc -o myprog myprog.c -lmpi
```

Chapter 3

Using `mpirun` to Execute Applications

The `mpirun`(1) command is the primary job launcher for the SGI implementation of MPI. The `mpirun` command must be used whenever a user wishes to run an MPI application on an IRIX or a Linux system. You can run an application on the local host only (the host from which you issued `mpirun`) or distribute it to run on any number of hosts that you specify. Note that several MPI implementations available today use a job launcher called `mpirun` and, because this command is not part of the MPI standard, each implementation's `mpirun` command differs in both syntax and functionality.

Syntax of the `mpirun` Command

The format of the `mpirun` command is as follows:

```
mpirun [global_options ] entry [ : entry ... ]
```

The *global_options* operand applies to all MPI executable files on all specified hosts. The following global options are supported:

Option	**Description**
`-a[rray]` *array_name*	(IRIX only) Specifies the array to use when launching an MPI application. By default, Array Services uses the default array specified in the Array Services configuration file, `arrayd.conf`.
`-cpr`	(IRIX systems only.) Allows users to checkpoint or restart MPI jobs that consist of a single executable file running on a single system. The absence of any host names in the `mpirun` command indicates that a job is running on a single system. For example, the following command is valid: `mpirun -cpr -np 2 ./a.out >&1/dev/null`

The following commands are not valid:

```
mpirun -cpr 2 ./a.out : 3 ./b.out
mpirun -cpr hosta -np 2 ./a.out>out 2>&1  mpirun -cpr hosta -np 2 ./a.out>out 2>&1  </dev/null
```

The first one is not valid because it consists of more than one executable file (`a.out` and `b.out`). The second one is not valid because even if submitted from `hosta`, it specifies a host name.

For interactive users, the preferred method of checkpointing the job is by `ASH`. This ensures that all of the user's processes specified in the `mpirun` command, plus daemons associated with the job, will be checkpointed. You can use the `array`(1) command to find the `ASH` of a job. Interactive users should also note that `stdin`, `stdout`, and `stderr` should not be connected to the terminal when this option is being used.

Use of this option requires Array Services 3.1 or later.

The default behavior will allow for jobs to be checkpointed if the above rules for invoking have been followed, but using the `-cpr` option is recommended because it provides specific error messages instead of silently disabling.

`-d[ir]` *path_name*
: Specifies the working directory for all hosts. In addition to normal path names, the following special values are recognized:

`.`	Translates into the absolute path name of the user's current working directory on the local host. This is the default.
~	Specifies the use of the value of `$HOME` as it is defined on each machine. In general, this value can be different on each machine.

`-f[ile]` *file_name*
: Specifies a text file that contains `mpirun` arguments.

`-h[elp]`	Displays a list of options supported by the `mpirun` command.
`-p[refix]` *prefix_string*	Specifies a string to prepend to each line of output from `stderr` and `stdout` for each MPI process. To delimit lines of text that come from different hosts, output to `stdout` must be terminated with a new line character. If a process's `stdout` or `stderr` streams do not end with a new line character, there will be no prefix associated with the output or error streams of that process from the final new line to the end of the stream.

If the `MPI_UNBUFFERED_STDIO` environment variable is set, the prefix string is ignored.

Some strings have special meaning and are translated as follows:

- `%g` translates into the global rank of the process producing the output. This is equivalent to the rank of the process in `MPI_COMM_WORLD` when not running in spawn capable mode. In the latter case, this translates to the rank of the process within the universe specified at job launch.
- `%G` translates into the number of processes in `MPI_COMM_WORLD`, or, if running in spawn capable mode, the value of the `MPI_UNIVERSE_SIZE` attribute.
- `%h` translates into the rank of the host on which the process is running, relative to the `mpirun`(1) command line. This string is not relevant for processes started via `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple`.
- `%H` translates into the total number of hosts in the job. This string is not relevant for processes started via `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple`.
- `%l` translates into the rank of the process relative to other processes running on the same host.

- %L translates into the total number of processes running on the host.
- `%w` translates into the world rank of the process, that is, its rank in `MPI_COMM_WORLD`. When not running in spawn capable mode, this is equivalent to `%g`.
- `%W` translates into the total number of processes in `MPI_COMM_WORLD`. When not running in spawn capable mode, this is equivalent to `%G`.
- %@ translates into the name of the host on which the process is running.

For examples of the use of these strings, first consider the following code fragment:

```
main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);

   printf("Hello world\n");

   MPI_Finalize();
}
```

Depending on how this code is run, the results of running the `mpirun` command will be similar to those in the following examples:

```
mpirun -np 2 a.out
Hello world
Hello world

mpirun -prefix ">" -np 2 a.out
>Hello world
>Hello world

mpirun -prefix "%g" 2 a.out
0Hello world
1Hello world
```

```
mpirun -prefix "[%g] " 2 a.out
[0] Hello world
[1] Hello world
```

```
mpirun -prefix "<process %g out of %G> " 4 a.out
<process 1 out of 4> Hello world
<process 0 out of 4> Hello world
<process 3 out of 4> Hello world
<process 2 out of 4> Hello world
```

```
mpirun -prefix "%@: " hosta,hostb 1 a.out
hosta: Hello world
hostb: Hello world
```

```
mpirun -prefix "%@ (%l out of %L) %g: " hosta 2, hostb 3 a.out
hosta (0 out of 2) 0: Hello world
hosta (1 out of 2) 1: Hello world
hostb (0 out of 3) 2: Hello world
hostb (1 out of 3) 3: Hello world
hostb (2 out of 3) 4: Hello world
```

```
mpirun -prefix "%@ (%h out of %H): " hosta,hostb,hostc 2 a.out
hosta (0 out of 3): Hello world
hostb (1 out of 3): Hello world
hostc (2 out of 3): Hello world
hosta (0 out of 3): Hello world
hostc (2 out of 3): Hello world
hostb (1 out of 3): Hello world
```

`-up` *u_size*	Specifies the value of the `MPI_UNIVERSE_SIZE` attribute to be used in supporting `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. This field must be set if either of these functions are to be used by the application being launched by `mpirun`. Setting this field implies the MPI job is being run in spawn capable mode.

`-v[erbose]`	Displays comments on what `mpirun` is doing when launching the MPI application.

The *entry* operand describes a host on which to run a program, and the local options for that host. You can list any number of entries on the `mpirun` command line.

In the common case (same program, multiple data (SPMD)), in which the same program runs with identical arguments on each host, usually, you need to specify only one entry.

Each entry has the following components:

- One or more host names (not needed if you run on the local host)
- Number of processes to start on each host
- Name of an executable program
- Arguments to the executable program (optional)

An entry has the following format:

host_list local_options program program_arguments

The *host_list* operand is either a single host (machine name) or a comma-separated list of hosts on which to run an MPI program.

The *local_options* operand contains information that applies to a specific host list. The following local options are supported:

Option	**Description**
`-f[ile]` *file_name*	Specifies a text file that contains `mpirun` arguments (same as *global_options*.) For more details, see "Using a File for `mpirun` Arguments".
`-np` *num_proc*	Specifies the number of processes on which to run.
`-nt` *num_tasks*	This option behaves the same as `-np`.

The *program program_arguments* operand specifies the name of the program that you are running and its accompanying options.

Using a File for `mpirun` Arguments

Because the full specification of a complex job can be lengthy, you can enter `mpirun` arguments in a file and use the `-f` option to specify the file on the `mpirun` command line, as in the following example:

```
mpirun -f my_arguments
```

The arguments file is a text file that contains argument segments. White space is ignored in the arguments file, so you can include spaces and newline characters for readability. An arguments file can also contain additional `-f` options.

Launching Programs on the Local Host Only

For testing and debugging, it is often useful to run an MPI program only on the local host without distributing it to other systems. To run the application locally, enter `mpirun` with the `-np` argument. Your entry must include the number of processes to run and the name of the MPI executable file.

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

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

You are not required to use a different host in each entry that you specify on the `mpirun`(1) command. You can launch a job that has two executable files on the same host. In the following example, both executable files use shared memory:

```
mpirun host_a -np 6 a.out : host_a -np 4 b.out
```

Launching a Distributed Program

You can use `mpirun`(1) to launch a program that consists of any number of executable files and processes and distribute it to any number of hosts. A host is usually a single machine, or, for IRIX systems, 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. Array Services is not supported on Linux systems currently, so an alternate launching mechanism is used.

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 following examples 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` or `-nt` option, listing 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, using an array called `test`:

```
mpirun -array test host_a 6 a.out : host_b 26 b.out
```

The following example launches an MPI application on different hosts out of the same directory on both hosts:

```
mpirun -d /tmp/mydir host_a 6 a.out : host_b 26 b.out
```

Launching a Program in Spawn Capable Mode on the Local Host

To use the MPI-2 process creation functions `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple`, the user 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 MPI application `mtest` in a universe of size 10:

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

Up to 7 more MPI processes can be started by `mtest` via one of the `spawn` commands.

Note: This implementation does not support spawn capable mode for MPI jobs launched via certain batch schedulers, nor does it support a spawn capability for MPI jobs spanning multiple hosts.

Chapter 4

Thread-Safe MPI

Note: The Linux implementation of MPI is not currently thread-safe.

The SGI implementation of MPI on IRIX systems assumes the use of POSIX threads or processes (see the `pthread_create`(3) or the `sprocs`(2) commands, respectively). MPI processes can be multithreaded. 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.

Threads are not separately addressable. To support both POSIX threads and processes (known as sprocs), thread-safe MPI must be run on an IRIX 6.5 system or later.

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`(3) 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_Initialized`(3).

Only one thread can call `MPI_Init_thread()`. This thread becomes the main thread. Since 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`(3c) 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 find out 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`(3) or `MPI_Iprobe`(3) will receive 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.

Signals

If a thread that executes an MPI call is cancelled 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 cancelled by another thread.

Internal Statistics

The SGI internal statistics diagnostics are not thread-safe.

Finalization

The call to `MPI_Finalize`(3) 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 the processes' threads have completed their MPI calls and have no pending communications or I/O operations.

Setting Environment Variables

This chapter describes the variables that specify the environment under which your MPI programs will run. Environment variables have default values if not explicitly set. You can change some variables to achieve particular performance objectives; others are required values for standard-compliant programs.

Setting MPI Environment Variables

Table 5-1, page 21 describes the MPI environment variables you can set for your programs. Unless otherwise specified, these variables are available for both Linux and IRIX systems.

Table 5-1 MPI Environment Variables

Variable	Description	Default
`MPI_ARRAY` (IRIX systems only)	Sets an alternative array name to be used for communicating with Array Services when a job is being launched.	Default name set in the `arrayd.conf` file.
`MPI_BAR_COUNTER` (IRIX systems only)	Specifies the use of a simple counter barrier algorithm within the `MPI_Barrier`(3) and `MPI_Win_fence`(3) functions.	Not enabled if job contains more than 64 PEs.

Variable	Description	Default
`MPI_BAR_DISSEM`	Specifies the use of the alternate barrier algorithm, the dissemination/butterfly, within the `MPI_Barrier`(3) and `MPI_Win_fence`(3) functions. This alternate algorithm provides better performance on jobs with larger PE counts. The `MPI_BAR_DISSEM` option is recommended for jobs with PE counts of 64 or greater.	Disabled if job contains less than 64 PEs; otherwise, enabled.
`MPI_BUFFER_MAX` (IRIX systems only)	Specifies a minimum message size, in bytes, for which the message will be considered a candidate for single-copy transfer. Currently, this mechanism is available only for communication between MPI processes on the same host. The sender data must reside in either the symmetric data, symmetric heap, or global heap. The MPI data type on the send side must also be a contiguous type. If the XPMEM driver is enabled (for single host jobs, see `MPI_XPMEM_ON` and for multihost jobs, see `MPI_USE_XPMEM`), MPI allows single-copy transfers for basic predefined MPI data types from any sender data location, including the stack and private heap. The XPMEM driver also allows single-copy transfers across partitions.	Not enabled.

Variable	Description	Default
	If cross mapping of data segments is enabled at job startup, data in common blocks will reside in the symmetric data segment. On systems running IRIX 6.5.2 or later, this feature is enabled by default. You can employ the symmetric heap by using the `shmalloc` (`shpalloc`) functions in LIBSMA. Testing of this feature has indicated that mMost MPI applications benefit more from buffering of medium-sized messages than from buffering of large messages, even though buffering of medium-sized messages requires an extra copy of data. However, highly synchronized applications that perform large message transfers can benefit from the single-copy pathway.	
`MPI_BUFS_PER_HOST`	Determines the number of shared message buffers (16 Kbytes each) that MPI is to allocate for each host. These buffers are used to send large messages.	16 pages (each page is 16 Kbytes)
`MPI_BUFS_PER_PROC`	Determines the number of private message buffers (16 Kbytes each) that MPI is to allocate for each process. These buffers are used to send large messages.	16 pages (each page is 16 Kbytes)

Variable	Description	Default
`MPI_BYPASS_CRC` (IRIX systems only)	Adds a checksum to each large message sent via HIPPI bypass. If the checksum does not match the data received, the job is terminated. Use of this environment variable might degrade performance.	Not set
`MPI_BYPASS_DEV_SELECTION` (IRIX systems only)	Specifies the algorithm MPI is to use for sending messages over multiple HIPPI adapters. Set this variable to one of the following values: • 0 - Static device selection. In this case, a process is assigned a HIPPI device to use for communication with processes on another host. The process uses only this HIPPI device to communicate with another host. This algorithm has been observed to be effective when interhost communication patterns are dominated by large messages (significantly more than 16K bytes).	1

Variable	Description	Default
	• 1 - Dynamic device selection. In this case, a process can select from any of the devices available for communication between any given pair of hosts. The first device that is not being used by another process is selected. This algorithm has been found to work best for applications in which multiple processes are trying to send medium-sized messages (16K or fewer bytes) between processes on different hosts. Large messages (more than 16K bytes) are split into chunks of 16K bytes. Different chunks can be sent over different HIPPI devices. • 2 - Round robin device selection. In this case, each process sends successive messages over a different HIPPI 800 device.	
`MPI_BYPASS_DEVS` (IRIX systems only)	Sets the order for opening HIPPI adapters. The list of devices does not need to be space-delimited (0123 is also valid). An array node usually has at least one HIPPI adapter, the interface to the HIPPI network. The HIPPI bypass is a lower software layer that interfaces directly to this adapter. The bypass sends MPI control and data messages that are 16 or fewer Kbytes.	0 1 2 3

Variable	Description	Default
	When you know that a system has multiple HIPPI adapters, you can use the `MPI_BYPASS_ DEVS` variable to specify the adapter that a program opens first. You can use this variable to ensure that multiple MPI programs distribute their traffic across the available adapters. If you prefer not to use the HIPPI bypass, you can turn it off by setting the `MPI_BYPASS_OFF` variable. When a HIPPI adapter reaches its maximum capacity of four MPI programs, it is not available to additional MPI programs. If all HIPPI adapters are busy, MPI sends internode messages by using TCP over the adapter instead of the bypass.	
`MPI_BYPASS_SINGLE` (IRIX systems only)	Allows MPI messages to be sent over multiple HIPPI connections if multiple connections are available. The HIPPI OS bypass multiboard feature is enabled by default. This environment variable disables it. When you set this variable, MPI operates as it did in previous releases, with use of a single HIPPI adapter connection, if available.	

Variable	Description	Default
`MPI_BYPASS_VERBOSE` (IRIX systems only)	Allows additional MPI initialization information to be printed in the standard output stream. This information contains details about the HIPPI OS bypass connections and the HIPPI adapters that are detected on each of the hosts.	
`MPI_CHECK_ARGS`	Enables checking of MPI function arguments. Segmentation faults might occur if bad arguments are passed to MPI, so this is useful for debugging purposes. Using argument checking adds several microseconds to latency.	Not enabled.
`MPI_COMM_MAX`	Sets the maximum number of communicators that can be used in an MPI program. Use this variable to increase internal default limits. (May be required by standard-compliant programs.)	256

Variable	Description	Default
MPI_DIR	Sets the working directory on a host. When an mpirun command is issued, the Array Services daemon on the local or distributed node responds by creating a user session and starting the required MPI processes. The user ID for the session is that of the user who invokes mpirun, so this user must be listed in the .rhosts file on the corresponding nodes. By default, the working directory for the session is the user's $HOME directory on each node. You can direct all nodes to a different directory (an NFS directory that is available to all nodes, for example) by setting the MPI_DIR variable to a different directory.	$HOME on the node. If using -np or -nt, the default is the current directory.
MPI_DPLACE_INTEROP_OFF (IRIX systems only)	Disables an MPI/dplace interoperability feature available beginning with IRIX 6.5.13. By setting this variable, you can obtain the behavior of MPI with dplace on older releases of IRIX.	Not enabled.
MPI_DSM_CPULIST (IRIX systems only)	Specifies a list of CPUs on which to run an MPI application. To ensure that processes are linked to CPUs, this variable should be used in conjunction with the MPI_DSM_MUSTRUN variable.	Not enabled.

Variable	Description	Default
`MPI_DSM_MUSTRUN` (IRIX systems only)	Enforces memory locality for MPI processes. Use of this feature ensures that each MPI process obtains a CPU and physical memory on the node to which it was originally assigned. This variable improves program performance on IRIX systems running release 6.5.7 and earlier, when running a program on a quiet system. With later IRIX releases, under certain circumstances, you do not need to set this variable. Internally, this feature directs the library to use the `process_cpulink`(3) function instead of `process_mldlink`(3) to control memory placement. You should not use `MPI_DSM_MUSTRUN` when the job is submitted to Miser (see `miser_submit`(1)) because this might cause the program to hang.	Not enabled.
`MPI_DSM_OFF` (IRIX systems only)	Turns off nonuniform memory access (NUMA) optimization in the MPI library.	Not enabled.

Variable	Description	Default
`MPI_DSM_PLACEMENT` (IRIX systems only)	Specifies the default placement policy to be used for the stack and data segments of an MPI process. Set this variable to one of the following values: • `firsttouch` - With this policy, IRIX attempts to satisfy requests for new memory pages for stack, data, and heap memory on the node where the requesting process is currently scheduled. • `fixed` - With this policy, IRIX attempts to satisfy requests for new memory pages for stack, data, and heap memory on the node associated with the memory locality domain (mld) with which an MPI process was linked at job startup. This is the default policy for MPI processes. • `roundrobin` - With this policy, IRIX attempts to satisfy requests for new memory pages in a round robin fashion across all of the nodes associated with the MPI job. It is generally not recommended to use this setting.	`fixed`

Variable	Description	Default
	• `threadroundrobin` - This policy is intended for use with hybrid MPI/OpenMP applications only. With this policy, IRIX attempts to satisfy requests for new memory pages for the MPI process stack, data, and heap memory in a roundrobin fashion across the nodes allocated to its OpenMP threads. This placement option might be helpful for large OpenMP/MPI process ratios. For non-OpenMP applications, this value is ignored.	
`MPI_DSM_PPM` (IRIX systems only)	Sets the number of MPI processes per memory locality domain (mld). For Origin 2000 systems, values of 1 or 2 are allowed. For Origin 3000 systems, values of 1, 2, or 4 are allowed.	Origin 2000 systems, 2; Origin 3000 systems, 4.
`MPI_DSM_TOPOLOGY` (IRIX systems only)	Specifies the shape of the set of hardware nodes on which the PE memories are allocated. Set this variable to one of the following values: • `cube` — A group of memory nodes that form a perfect hypercube. The number of processes per host must be a power of 2. If a perfect hypercube is unavailable, a less restrictive placement is used.	Not enabled.

Variable	Description	Default
	• `cube_fixed` — A group of memory nodes that form a perfect hypercube. The number of processes per host must be a power of 2. If a perfect hypercube is unavailable, the placement will fail, disabling NUMA placement. • `cpucluster` — Any group of memory nodes. The operating system attempts to place the group numbers close to one another, taking into account nodes with disabled processors. (Default for IRIX 6.5.11 and higher). • `free` — Any group of memory nodes. The operating system attempts to place the group numbers close to one another. (Default for IRIX 6.5.10 and earlier releases).	
`MPI_DSM_VERBOSE` (IRIX systems only)	Instructs `mpirun` to print information about process placement for jobs running on NUMA systems.	Not enabled.
`MPI_DSM_VERIFY` (IRIX systems only)	Instructs `mpirun` to run some diagnostic checks on proper memory placement of MPI data structures at job startup. If errors are found, a diagnostic message is printed to `stderr`.	Not enabled.

Variable	Description	Default
`MPI_GM_DEVS` (IRIX systems only)	Sets the order for opening GM(Myrinet) adapters. The list of devices does not need to be space-delimited (0321 is valid). The syntax is the same as for the `MPI_BYPASS_DEVS` environment variable. In this release, a maximum of 8 adapters are supported on a single host.	MPI will use all available GM(Myrinet) devices.
`MPI_GM_VERBOSE`	Allows some diagnostic information concerning messaging between processes using GM (Myrinet) to be displayed on `stderr`.	Not enabled.
`MPI_GROUP_MAX`	Sets the maximum number of groups that can be used in an MPI program. Use this variable to increase internal default limits. (May be required by standard-compliant programs.)	256
`MPI_GSN_DEVS` (IRIX 6.5.9 systems or later)	Sets the order for opening GSN adapters. The list of devices does not need to be quoted or space-delimited (0123 is valid).	MPI will use all available GSN devices.
`MPI_GSN_VERBOSE` (IRIX 6.5.9 systems or later)	Allows additional MPI initialization information to be printed in the standard output stream. This information contains details about the GSN (ST protocol) OS bypass connections and the GSN adapters that are detected on each of the hosts.	Not enabled.

Variable	Description	Default
`MPI_MSG_RETRIES`	Specifies the number of times the MPI library attempts to get a message header, if none are available. Each MPI message that is sent requires an initial message header. If one is not available after the specified number of attempts, the job will abort. Note that this variable no longer applies to processes on the same host, or when using the GM (Myrinet) protocol. In these cases, message headers are allocated dynamically on an as-needed basis.	500
`MPI_MSGS_MAX`	Controls the total number of message headers that can be allocated. This allocation applies to messages exchanged between processes on a single host, or between processes on different hosts when using the GM (Myrinet) OS bypass protocol. Note that the initial allocation of memory for message headers is 128 Kbytes.	Allow up to 64 Mbytes to be allocated for message headers. If you set this variable, specify the maximum number of message headers.

Variable	Description	Default
`MPI_MSGS_PER_HOST`	Sets the number of message headers to allocate for MPI messages on each MPI host. Space for messages that are destined for a process on a different host is allocated as shared memory on the host on which the sending processes are located. MPI locks these pages in memory. Use this variable to allocate buffer space for interhost messages. ⚠ **Caution:** If you set the memory pool for interhost packets to a large value, you can cause allocation of so much locked memory that total system performance is degraded. The previous description does not apply to processes that use the GM (Myrinet) OS bypass protocol. In this case, message headers are allocated dynamically as needed. See the `MPI_MSGS_MAX` variable description.	1024

Variable	Description	Default
`MPI_MSGS_PER_PROC`	This variable is effectively obsolete. Message headers are now allocated on an as-needed basis for messaging either between processes on the same host, or between processes on different hosts when using the GM (Myrinet) OS bypass protocol. You can use the new `MPI_MSGS_MAX` variable to control the total number of message headers that can be allocated.	1024
`MPI_OPENMP_INTEROP` (IRIX systems only)	Setting this variable modifies the placement of MPI processes to better accomodate the OpenMP threads associated with each process. **Note:** This option is available only on Origin 300 and Origin 3000 servers.	Not enabled
`MPI_REQUEST_MAX`	Sets the maximum number of simultaneous nonblocking sends and receives that can be active at one time. Use this variable to increase internal default limits. (May be required by standard-compliant programs.)	16384
`MPI_SHARED_VERBOSE`	Allows some diagnostic information concerning messaging within a host to be displayed on `stderr`.	Not enabled.

Variable	Description	Default
`MPI_SLAVE_DEBUG_ATTACH`	Specifies the MPI process to be debugged. If you set `MPI_SLAVE_DEBUG_ATTACH` to *N*, the MPI process with rank *N* prints a message during program startup, describing how to attach to it from another window using the dbx debugger on IRIX or the gdb debugger on Linux. You must attach the debugger to process *N* within ten seconds of the printing of the message.	Not enabled.
`MPI_STATIC_NO_MAP` (IRIX systems only)	Disables cross mapping of static memory between MPI processes. This variable can be set to reduce the significant MPI job startup and shutdown time that can be observed for jobs involving more than 512 processors on a single IRIX host. Note that setting this shell variable disables certain internal MPI optimizations and also restricts the use of MPI-2 one-sided functions. For more information, see the `MPI_Win` man page.	Not enabled.

Variable	Description	Default
`MPI_STATS`	Enables printing of MPI internal statistics. Each MPI process prints statistics about the amount of data sent with MPI calls during the `MPI_Finalize` process. Data is sent to `stderr`. To prefix the statistics messages with the MPI rank, use the `-p` option on the `mpirun` command. **Note:** Because the statistics-collection code is not thread-safe, this variable should not be set if the program uses threads.	Not enabled.
`MPI_TYPE_DEPTH`	Sets the maximum number of nesting levels for derived data types. (May be required by standard-compliant programs.) This variable limits the maximum depth of derived data types that an application can create. MPI logs error messages if the limit specified by `MPI_TYPE_DEPTH` is exceeded.	8 levels
`MPI_TYPE_MAX`	Sets the maximum number of derived data types that can be used in an MPI program. Use this variable to increase internal default limits. (May be required by standard-compliant programs.)	1024

Variable	Description	Default
`MPI_UNBUFFERED_STDIO`	Normally, `mpirun` line-buffers output received from the MPI processes on both the `stdout` and `stderr` standard IO streams. This prevents lines of text from different processes from possibly being merged into one line, and allows use of the `mpirun -prefix` option. Of course, there is a limit to the amount of buffer space that `mpirun` has available (currently, about 8,100 characters can appear between new line characters per stream per process). If more characters are emitted before a new line character, the MPI program aborts with an error message. Setting the `MPI_UNBUFFERED_STDIO` environment variable disables this buffering. This is useful, for example, when a program's rank 0 emits a series of periods over time to indicate progress of the program. With buffering, the entire line of periods will be output only when the new line character is seen. Without buffering, each period will be immediately displayed as soon as `mpirun` receives it from the MPI program. (Note that the MPI program still needs to call `fflush`(3) or `FLUSH(101)` to flush the `stdout` buffer from the application code.)	1024

Variable	Description	Default
	Additionally, setting MPI_UNBUFFERED_STDIO allows an MPI program that emits very long output lines to execute correctly. Note that if MPI_UNBUFFERED_STDIO is set, the mpirun -prefix option is ignored.	
MPI_USE_GSN (IRIX 6.5.12 systems or later)	Requires the MPI library to use the GSN (ST protocol) OS bypass driver as the interconnect when running across multiple hosts or running with multiple binaries. If a GSN connection cannot be established among all hosts in the MPI job, the job is terminated. GSN imposes a limit of one MPI process using GSN per CPU on a system. For example, on a 128-CPU system, you can run multiple MPI jobs, as long as the total number of MPI processes using the GSN bypass does not exceed 128.	Not set

Variable	Description	Default
	Once the maximum allowed MPI processes using GSN is reached, subsequent MPI jobs return an error to the user output, as the following example: `MPI: Could not connect all processes to GSN adapters. The maximum number of GSN adapter connections per system is normally equal to the number of CPUs on the system.` If there are a few CPUs still available, but not enough to satisfy the entire MPI job, the error will still be issued and the MPI job terminated.	
`MPI_USE_GM` (IRIX systems only)	Requires the MPI library to use the Myrinet(GM) OS bypass driver as the interconnect when running across multiple hosts or running with multiple binaries. If a GM connection cannot be established among all hosts in the MPI job, the job is terminated.	Not set
`MPI_USE_HIPPI` (IRIX systems only)	Requires the MPI library to use the HiPPI 800 OS bypass driver as the interconnect when running across multiple hosts or running with multiple binaries. If a HiPPI connection cannot be established among all hosts in the MPI job, the job is terminated.	Not set

Variable	Description	Default
`MPI_USE_TCP`	Requires the MPI library to use the TCP/IP driver as the interconnect when running across multiple hosts or running with multiple binaries.	Not set
`MPI_USE_XPMEM` (IRIX 6.5.13 systems or later)	Requires the MPI library to use the XPMEM driver as the interconnect when running across multiple hosts or running with multiple binaries. This driver allows MPI processes running on one partition to communicate with MPI processes on a different partition via the NUMAlink network. The NUMAlink network is powered by block transfer engines (BTEs). BTE data transfers do not require processor resources. The XPMEM (cross partition) device driver is available only on Origin 3000 and Origin 300 systems running IRIX 6.5.13 or greater. **Note:** Due to possible MPI program hangs, you should not run MPI across partitions using the XPMEM driver on IRIX versions 6.5.13, 6.5.14, or 6.5.15. This problem has been resolved in IRIX version 6.5.16.	Not set

Variable	Description	Default
	If all of the the hosts specified on the `mpirun` command do not reside in the same partitioned system, you can select one additional interconnect via the `MPI_USE` variables. MPI communication between partitions will go through the XPMEM driver, and communication between non-partitioned hosts will go through the second interconnect.	
`MPI_XPMEM_ON`	Enables the XPMEM single-copy enhancements for processes residing on the same host. The XPMEM enhancements allow single-copy transfers for basic predefined MPI data types from any sender data location, including the stack and private heap. Without enabling XPMEM, single-copy is allowed only from data residing in the symmetric data, symmetric heap, or global heap. Both the `MPI_XPMEM_ON` and `MPI_BUFFER_MAX` variables must be set to enable these enhancements. Both are disabled by default.	Not set

Variable	Description	Default
	If the following additional conditions are met, the block transfer engine (BTE) is invoked instead of `bcopy`, to provide increased bandwidth: • Send and receive buffers are cache-aligned. • Amount of data to transfer is greater than or equal to the `MPI_XPMEM_THRESHOLD` value. **Note:** The XPMEM driver does not support checkpoint/restart at this time. If you enable these XPMEM enhancements, you will not be able to checkpoint and restart your MPI job. The XPMEM single-copy enhancements require an Origin 3000 and Origin 300 servers running IRIX release 6.5.15 or greater.	
`MPI_XPMEM_THRESHOLD`	Specifies a minimum message size, in bytes, for which single-copy messages between processes residing on the same host will be transferred via the BTE, instead of `bcopy`. The following conditions must exist before the BTE transfer is invoked: • Single-copy mode is enabled (`MPI_BUFFER_MAX`).	8192

Variable	Description	Default
	• XPMEM single-copy enhancements are enabled (`MPI_XPMEM_ON`). • Send and receive buffers are cache-aligned. • Amount of data to transfer is greater than or equal to the `MPI_XPMEM_THRESHOLD` value. The XPMEM enhancements allow single-copy transfers for basic MPI types from any sender data location, including the stack and private heap. Without enabling XPMEM, single-copy is allowed only from data residing in the symmetric data, symmetric heap, or global heap. Both the `MPI_XPMEM_THRESHOLD` and `MPI_BUFFER_MAX` variables must be set to enable these enhancements. Both are disabled by default. If the following additional conditions are met, the block transfer engine (BTE) is invoked instead of `bcopy`, to provide increased bandwidth: • Send and receive buffers are cache-aligned. • Amount of data to transfer is greater than or equal to the `MPI_XPMEM_THRESHOLD` value.	

Variable	Description	Default
	In addition to enabling these single-copy enhancements, the `MPI_XPMEM_THRESHOLD` environment variable can be used to specify a minimum message size, in bytes, for which the message will be transferred via the BTE, provided the above conditions are met. If a value is not provided, a default of 8192 bytes will be used. **Note:** The XPMEM driver does not support checkpoint/restart at this time. If you enable these XPMEM enhancements, you will not be able to checkpoint and restart your MPI job. The XPMEM single-copy enhancements require Origin 3000 and Origin 300 servers running IRIX release 6.5.15 or greater.	
`MPI_XPMEM_VERBOSE`	Setting this variable allows additional MPI diagnostic information to be printed in the standard output stream. This information contains details about the XPMEM connections.	Not enabled

Variable	Description	Default
`PAGESIZE_DATA` (IRIX systems only)	Specifies the desired page size in kilobytes for program data areas. On Origin series systems, supported values include 16, 64, 256, 1024, and 4096. Specified values must be integer. **Note:** Setting `MPI_DSM_OFF` disables the ability to set the data pagesize via this shell variable.	Not enabled
`PAGESIZE_STACK` (IRIX systems only)	Specifies the desired page size in kilobytes for program stack areas. On Origin series systems, supported values include 16, 64, 256, 1024, and 4096. Specified values must be integer. **Note:** Setting `MPI_DSM_OFF` disables the ability to set the data pagesize via this shell variable.	Not enabled
`SMA_GLOBAL_ALLOC` (IRIX systems only)	Activates the LIBSMA based global heap facility. This variable is used by 64–bit MPI applications for certain internal optimizations, and is used as support for the `MPI_Alloc_mem` function. For additional details, see the `intro_shmem`(3) man page.	Not enabled
`SMA_GLOBAL_HEAP_SIZE` (IRIX systems only)	For 64 bit applications, specifies the per processes size of the LIBSMA global heap in bytes.	33,554,432 bytes

Internal Message Buffering in MPI

An MPI implementation can copy data that is being sent to another process into an internal temporary buffer so that the MPI library can return from the MPI function, giving execution control back to the user. However, according to the MPI standard, you should not assume that there is any message buffering between processes because the MPI standard does not mandate a buffering strategy. Some implementations choose to buffer user data internally, while other implementations block in the MPI routine until the data can be sent. These different buffering strategies have performance and convenience implications.

Most MPI implementations do use buffering for performance reasons and some programs depend on it. Table 5-2, page 49 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 `MPI_Send` 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.

The SGI implementation of MPI uses buffering under most circumstances. Short messages of 64 or fewer bytes are always buffered. On IRIX systems, longer messages are buffered unless the message to be sent resides in either a common block, the symmetric heap, or global shared heap and the sending and receiving processes reside on the same host. The MPI data type on the send side must also be a contiguous type. The message size must be greater than the size setting for `MPI_BUFFER_MAX` (see Table 5-1, page 21). If the XPMEM driver is enabled (for single host jobs, see `MPI_XPMEM_ON` and for multihost jobs, see `MPI_USE_XPMEM`), MPI allows single-copy transfers for basic MPI types from any sender data location, including the stack and private heap. The XPMEM driver also allows single-copy transfers across partitions. Under these circumstances, the receiver copies the data directly into its receive message area without buffering. Obviously, MPI applications with code segments equivalent to that shown in Table 5-2, page 49 will almost certainly deadlock if this bufferless pathway is available.

Note: This feature is not currently available on Linux systems.

Table 5-2 Outline of Improper Dependence on Buffering

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

Chapter 6

MPI Optimization and Tuning

This chapter provides information for maximizing the performance of the SGI MPI library. This information includes automatic or default optimizations, optimizations that the user can perform through the use of environment variables, optimizations for using MPI on IRIX clusters, and some general tips and tools for optimization.

Application Optimizations

The optimizations described in this section are performed by the MPI library automatically without changes from the user. For these optimizations to be effective, you must use the suggested MPI functions.

Optimized Point-to-Point Calls

The MPI library has been optimized to achieve low latency and high bandwidth for programs that use `MPI_Send/MPI_Recv` or `MPI_Isend/MPI_Irecv` point-to-point message passing. `MPI_Rsend` is treated the same as `MPI_Send` in this implementation. In addition, these point-to-point calls have been optimized for a high repeat rate. This allows applications that exchange data with the other processors to handle the requests at the same time without unnecessary waiting.

The diagram in Figure 6-1, page 53 shows what happens within the library when a message is sent from one process to another. In this example, a medium sized message (between 64 and 16384 bytes) is passed from one process to another on the same IRIX or Linux host. The numbers on the arrows indicate the order in which the following steps occur.

Procedure 6-1 Message passing process

1. The sender aquires a shared memory buffer and copies the `src` data into that buffer.
2. The sender performs a fetch and add operation of the `fetchop` variable that controls the receiver's message queue. The value obtained from this operation indicates the slot in which a message header can be placed in the receiver's queue. The message header contains MPI related data, such as tag, communicator, location of data, and so on. The sender then copies the message header into the receiver's queue.

3. The receiver, which is polling on its message queue, finds the message header and copies the message header out of the message queue.
4. The receiver copies the data from the shared memory buffer into the `dst` buffer specified by the application.
5. The receiver performs a fetch and add operation of the `fetchop` variable that controls the sender's message queue. The receiver then copies a message header to the sender's message queue, which acknowledges (`ACK`) that the message has been received.
6. The sender receives the `ACK` and marks the shared memory buffer and other internal data structures for reuse.

Short messages (64 bytes or less) are further optimized but do not need to use the shared memory buffers because the data actually fits in the message header itself.

The `MPI_Send/MPI_Recv` are blocking calls and are not required to be buffered. This is important for single copy optimization, described in "Single Copy Optimization", page 57.

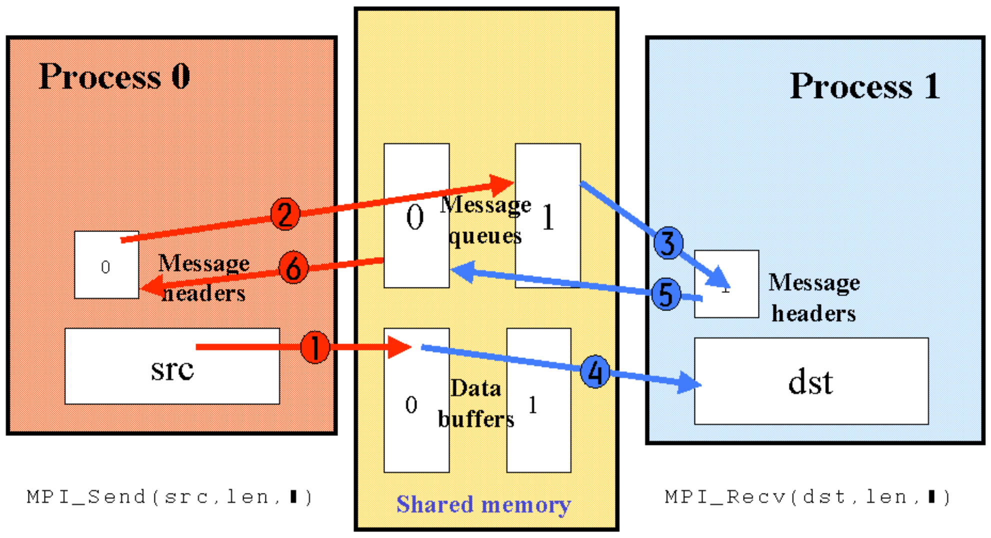

Figure 6-1 Message passing process

Optimized Collective Calls

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, the MPI library has optimized a number of the collective operations to make use of shared memory.

The `MPI_Alltoall` collective has been optimized to make use of symmetric data and the global heap. When the NAS FT parallel benchmark comparing two different versions of the SGI MPI library was run, the optimized `MPI_Alltoall` operation was almost an order of magnitude faster on high process counts than the nonoptimized buffered point-to-point `MPI_Alltoall`.

The `MPI_Barrier` call makes use of the fetchop barrier method for `MPI_COMM_WORLD` and similar communicators. Because it is the primary mechanism for synchronization, the barrier collective operation is critical for MPI-2 one-sided applications and SHMEM programs. The MPI library uses a tree or dissemination barrier mechanism for programs with 64 or more MPI processes. This implementation uses multiple fetch operations to minimize contention for HUB caches as well as to confine uncached loads to individual nodes to reduce traffic on the NUMA links.

Both the `MPI_Alltoall` collective and the `MPI_Barrier`call are also optimized for clusters. In addition, the `MPI_Bcast` and `MPI_Allreduce` collectives are optimized for clusters.

NUMA Placement

The MPI library takes advantage of NUMA placement functions that are available from IRIX. When running on IRIX 6.5.11 and later releases, the default topology (`MPI_DSM_TOPOLOGY`) is `cpucluster`. This allows IRIX to place the memories for that processor on any group of memory nodes of the hardware. IRIX attempts to place the group numbers close to one another, taking into account nodes with disabled processors.

If the user is running within a cpuset, only those CPUs and memory nodes specified within that cpuset are used.

In addition, MPI does NUMA placement optimization of key internal data structures to ensure it has good locality with respect to each CPU and does not create any unnecessary bottlenecks on specific memory nodes.

The `oview` command from Performance-CoPilot was used to generate Figure 6-2, page 55, which shows the placement of a 32 processor MPI job run on an 512 PE Origin 3000. Notice that the jobs (white bars) were placed on eight memory nodes near one another.

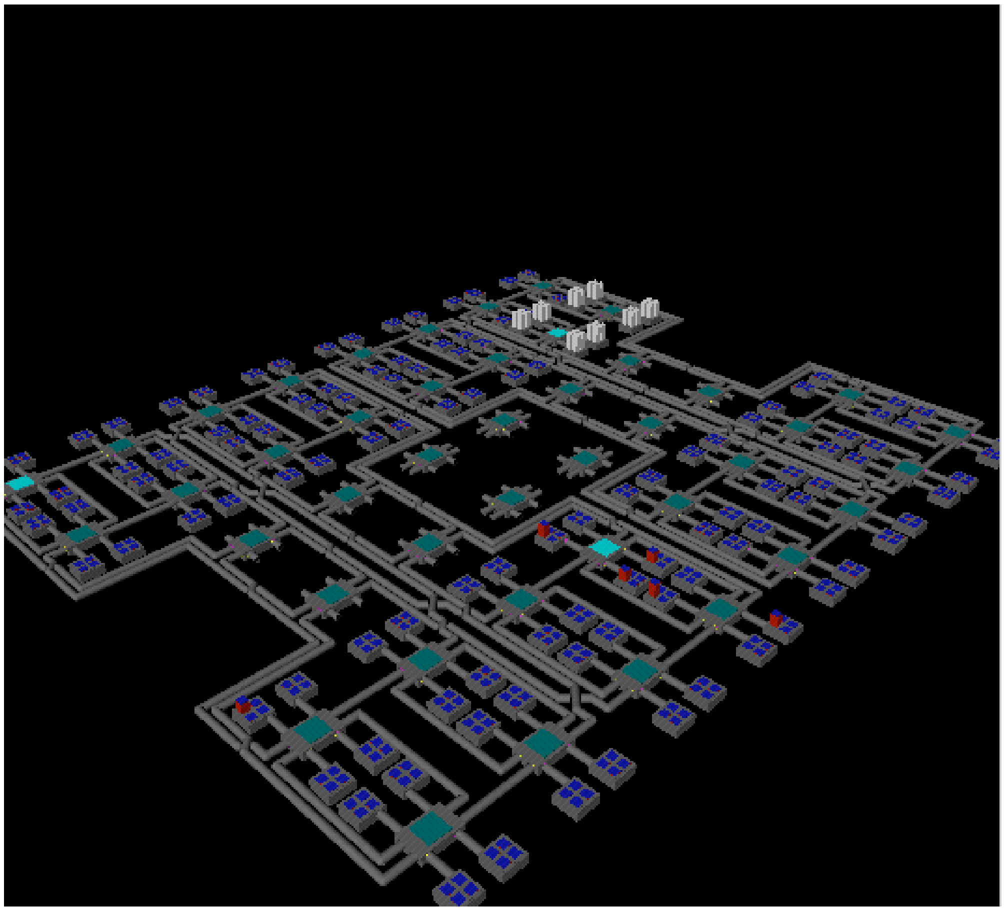

Figure 6-2 Placement optimization

MPI One-Sided Operations

Users who wish to get SHMEM like performance but write their code using a more portable interface can use the MPI-2 remote memory operations commonly called MPI one-sided. These interfaces are optimized and in fact use the SHMEM library to achieve very low latency and very high bandwidth.

Runtime Optimizations

You can tune the MPI library for performance by using environment variables described in the `MPI` man page. This section describes some of the common runtime optimizations.

Eliminating Retries

The MPI statistic counters (`-stats` option or `MPI_STATS` environment variable) can be used to tune the runtime environment of an MPI application. These counters are always accumulating statistics, so turning them on simply displays them.

One of these statistics is the number of retries. A retry indicates that the library spent time waiting for shared memory buffers to be made available before sending a message. The number of buffers can be increased to eliminate retries by adjusting the corresponding MPI environment variable. The most common ones that may need to be increased are `MPI_BUFS_PER_PROC` and `MPI_BUFS_PER_HOST`.

The following partial `-stats` output shows that 5672 retries were attempted for `PER PROC` data buffers for rank 3. In this case, the user should increase the `MPI_BUFS_PER_PROC` environment variable.

```
mpirun -stats -prefix "%g:" -np 8 a.out

...
3: *** Dumping MPI internal resource statistics...
3:
3:  0 retries allocating mpi PER_PROC headers for collective calls
3:  0 retries allocating mpi PER_HOST headers for collective calls
3:  0 retries allocating mpi PER_PROC headers for point-to-point calls
3:  0 retries allocating mpi PER_HOST headers for point-to-point calls
3:  0 retries allocating mpi PER_PROC buffers for collective calls
3:  0 retries allocating mpi PER_HOST buffers for collective calls
3:  5672 retries allocating mpi PER_PROC buffers for point-to-point calls
3:  0 retries allocating mpi PER_HOST buffers for point-to-point calls
3:  0 send requests using shared memory for collective calls
3:  6357 send requests using shared memory for point-to-point calls
3:  0 data buffers sent via shared memory for collective calls
3:  2304 data buffers sent via shared memory for point-to-point calls
3:  0 bytes sent using single copy for collective calls
3:  0 bytes sent using single copy for point-to-point calls
3:  0 message headers sent via shared memory for collective calls
```

```
3:  6357 message headers sent via shared memory for point-to-point calls
3:  0 bytes sent via shared memory for collective calls
3:  15756000 bytes sent via shared memory for point-to-point calls
...
```

Single Copy Optimization

One of the most significant optimizations for bandwidth sensitive applications that has been made in the MPI library has been single copy optimization. This section describes three types of single copy optimization. Each one of these utilize the fact that the library is not required to buffer the data. For this reason some nonstandard MPI codes that require buffering for their `MPI_Send/MPI_Recv` calls may experience program hangs. This is why this optimization is not enabled by default.

The MPI data type on the send side must also be a contiguous type.

Since single copy optimization does not use the shared memory data buffers, enabling it eliminates the problem described in "Eliminating Retries", page 56 concerning retries caused by too few per process data buffers (`MPI_BUFS_PER_PROC`).

For jobs that are run across a cluster, the messages sent between processes within a host use single copy optimization, if enabled.

Single copy transfers for point-to-point as well as collective operations are listed in the `-stats` output. Checking those statistics when attempting to use this optimization can be very helpful to determine if single copy was used.

Traditional Single Copy Optimization and Restrictions

The traditional approach requires that users make sure the senders data resided in globally accessible memory and that they set the `MPI_BUFFER_MAX` environment variable. This optimization uses special cross-mapping of memory from the SHMEM library and thus is available only for ABI 64.

Globally accessible memory includes common block or static memory and memory allocated with the Fortran 90 `allocate` statement (with the `SMA_GLOBAL_ALLOC` environment variable set). In addition, applications linked against the SHMEM library may also access the LIBSMA symmetric heap via the `shpalloc` or `shmalloc` functions.

Setting the `MPI_BUFFER_MAX` variable to any value will enable this optimization. However, you should use a value near 2000 because this optimization for messages of smaller size will often not yield better performance and sometimes may decrease performance slightly.

For a simple bandwidth test involving two MPI processors and large message lengths, the traditional single copy showed about 60% improvement in MB/sec.

Less Restrictive Single Copy Using the XPMEM Driver

One MPI library feature takes advantage of a special IRIX device driver known as XPMEM (cross partition) that allows the operating system to make a very fast copy between two processes within the same host or across partitions. This feature requires IRIX 6.5.15 or greater. The XPMEM driver has been used by the MPI library 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 traditional single copy optimization. You can enable this optimization if you set the `MPI_XPMEM_ON` and the `MPI_BUFFER_MAX` environment variables. The library tries to use the traditional single copy before trying to use this form of single copy. Using the XPMEM form of single copy is less restrictive in that the sender's data is not required to be globally accessible and it is available for ABI N32 as well as ABI 64. Also, this optimization can be used to transfer data between two different executable files on the same host or two different executable files across IRIX partitions.

Single Copy Using the XPMEM Driver and the BTE

In 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.

For the same bandwidth test mentioned in " Traditional Single Copy Optimization and Restrictions ", page 57, using the BTE showed more than a two-fold improvement in MB/sec over the traditional single copy.

NUMA Placement

Occasionally, it is useful to control NUMA placement. The `dplace` command can be very effective but sometimes difficult to use. Using cpusets can be effective in controlling NUMA placement, but to set up cpusets, you need to have root access.

The MPI library has introduced several environment variables to allow dplace and cpuset functionality when running on a quiet system. You can assign your MPI ranks to specific CPUs by using the `MPI_DSM_CPULIST` environment variable. You can use the `MPI_DSM_VERBOSE` environment variable to determine what CPUs and memory nodes were used, and which sysAD bus was used.

If you have a memory bound code that uses the memory of nodes in which the CPUs are idle, it might make sense to assign fewer CPUs per node to the application. You can do this by using the `MPI_DSM_PPM` environment variable. Note that the `MPI_DSM_MUSTRUN` environment variable must be set to ensure the MPI processes are pinned to the processors specified in the `MPI_DSM_CPULIST`.

If you are experiencing frequent TLB misses, you can increase the `PAGESIZE_DATA` environment variable. It is best to increase it slowly because setting it too high could cause even worse performance. Using the perfex or ssrun tools can help determine if TLB misses are excessive.

Optimizations for Using MPI on IRIX Clusters

When you are running an MPI application across a cluster of IRIX hosts, there are additional runtime 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. Origin 300 and Origin 3000 series servers can be clustered as partitioned systems using the XPMEM interconnect. Other high performance interconnects include GSN and Myrinet. The older HIPPI 800 interconnect technology is also supported by the SGI MPI implementation. If none of these interconnects is available, MPI relies on TCP/IP to handle MPI traffic between hosts.

When launched as a distributed application, MPI probes for these interconnects at job startup. Launching a distributed application is described in Chapter 3, "Using `mpirun` to Execute Applications", page 7. 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 60, 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`
HIPPI 800	4	`MPI_USE_HIPPI`	`MPI_BYPASS_DEVS`
TCP/IP	5	`MPI_USE_TCP`	`NA`

*These interconnects are available only on Origin 300 and Origin 3000 series computers.

The third column of Table 6-1, page 60 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 HIPPI interconnects. By default, the MPI job would try to run over the GSN interconnect. If for some reason, you wanted to use the HIPPI 800 interconnect, you would set the `MPI_USE_HIPPI` shell variable before launching the job. This would cause the MPI library to attempt to run the job using the HIPPI interconnect. The job will fail if the HIPPI interconnect cannot be used.

The XPMEM interconnect is exceptional 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.

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, the 'retries allocating mpi PER_HOST' metric should be examined. High retry counts indicate that the MPI_BUFS_PER_HOST shell variable should be increased. For example, when using Myrinet(GM), it has been observed that 256 or 512 are good settings for the MPI_BUFS_PER_HOST shell variable.

When considering these MPI statistics, GSN and HIPPI 800 users should also examine the 'retries allocating mpi PER_HOST headers' counters. It might be necessary to increase the MPI_MSGS_PER_HOST shell variable in cases in which this metric indicates high numbers of retries. Myrinet(GM) does not use this resource.

When using GSN, Myrinet, or HIPPI 800 high performance networks, MPI attempts to use all available adapters (cards) available on each host in the job. You can modify this behavior by specifying specific adpter(s) to use. The fourth column of Table 6-1, page 60 indicates the shell variable to use for a given network. For details on syntax, see the MPI man page. Users of HIPPI 800 networks have additional control over the way in which MPI uses multiple adapters via the MPI_BYPASS_DEV_SELECTION variable. For details on the use of this environment variable, see the MPI man page.

When using the TCP/IP interconnect, unless specified otherwise, MPI uses the default IP adpater for each host. To use a non-default adapter, the adapterr-specific host name can be used on the mpirun command line.

Using MPI with OpenMP

Hybrid MPI/OpenMP applications might require special memory placement features to operate efficiently on cc-NUMA Origin servers. A preliminary method for realizing this memory placement is available. The basic idea is to space out the MPI processes to accomodate the OpenMP threads associated with each MPI process. In addition, assuming a particular ordering of library init code (see the DSO(5) man page), procedures are employed 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 when more than four OpenMP threads are used by each MPI process.

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

- The user must set the MPI_OPENMP_INTEROP shell variable when running the application.
- The user must use a MIPSpro compiler and the -mp option to compile the application. This placement option is not available with other compilers.

- To take full advantage of this placement option, the user must be able to link the application such that the `libmpi.so init` code is run before the `libmp.so init` code. This is done by linking the MPI/OpenMP application as follows:

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

This linkage order insures that the `libmpi.so init` runs procedures for restricting the placement of OpenMP threads before the `libmp.so init` is run. Note that this is not the default linkage if only the `-mp` option is specified on the link line.

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. For more information, see Chapter 5, "Setting Environment Variables", page 21.

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.

Note: This placement is not recommended when setting `_DSM_PPM` to a non-default value (for more information, see `pe_environ`(5)). Also, if you are using `MPI_DSM_MUSTRUN`, it is important to also set `_DSM_MUSTRUN` to properly schedule the OpenMP threads.

Tips for Optimizing

Most techniques and tools for optimizing serial codes apply to message passing codes as well. Speedshop (ssrun) and perfex are tools that work well. The following tips for optimizing are described in this section:

- MPI constructs to avoid
- How to reproduce timings
- How to use MPI statistics
- A summary of the profiling tools available and how to get started using them

Avoiding Certain MPI Constructs

This section describes certain MPI constructs to avoid in performance critical sections of your application.

- The MPI library has been enhanced to optimize certain point-to-point and collective calls. Calls that have not been optimized and should be avoided in critical areas of an application are `MPI_Bsend`, `MPI_Ssend`, and `MPI_Issend`.

  The `MPI_Bsend` call is a buffered send and essentially doubles the amount of data to be copied by the sending process.

  The `MPI_Ssend` and `MPI_Issend` are synchronous sends that do not begin sending the message until they have received an acknowledgment (`ACK`) from the destination process that it is ready to receive the message. This significantly increases latency, especially for short messages (less than 64 bytes).

- 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 the receiver. It is best to avoid the use of these functions by either restructuring your data or using MPI derived data types.

- 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.

Reducing Runtime Variability

One of the most common problems with optimizing message passing codes is achieving reproducible timings from run to run. Use the following tips to reduce runtime variability:

- 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.

- Use cpusets to divide the host's CPUs and memory between applications or use the `MPI_DSM_CPULIST` environment variable on quiet systems to control NUMA placement.
- Use a batch scheduler like LSF from Platform Computing or PBSpro from Veridian, which can help avoid oversubscribing the system and poor NUMA placement.

Using MPI Statistics

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 simply lets you see them during `MPI_Finalize`. 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

Using Profiling

In addition to the MPI statistics, users can get additional performance information from several SGI or third party tools, or they can write their own wrappers. For long running codes or codes that use hundreds of processors, the trace data generated from MPI performance tools can be enormous. This causes the programs 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 to locate the problem area and then to turn on and off the tracing just around those areas of your code. With this approach the display tools can better handle the amount of data that is generated. Two third party tools that you can use in this way are Vampir from Pallas (`www.pallas.com`) and Jumpshot, which is part of the MPICH distribution.

Two of the most common SGI profiling tools are Speedshop and perfex. The following sections describe how to invoke Speedshop and perfex for typical profiling of MPI codes. Performance-CoPilot (PCP) tools and tips for writing your own tools are also included.

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
```

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
```

Performance-CoPilot

In addition to the tools described in the preceding sections, you can also use the MPI Agent for Performance-CoPilot (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.

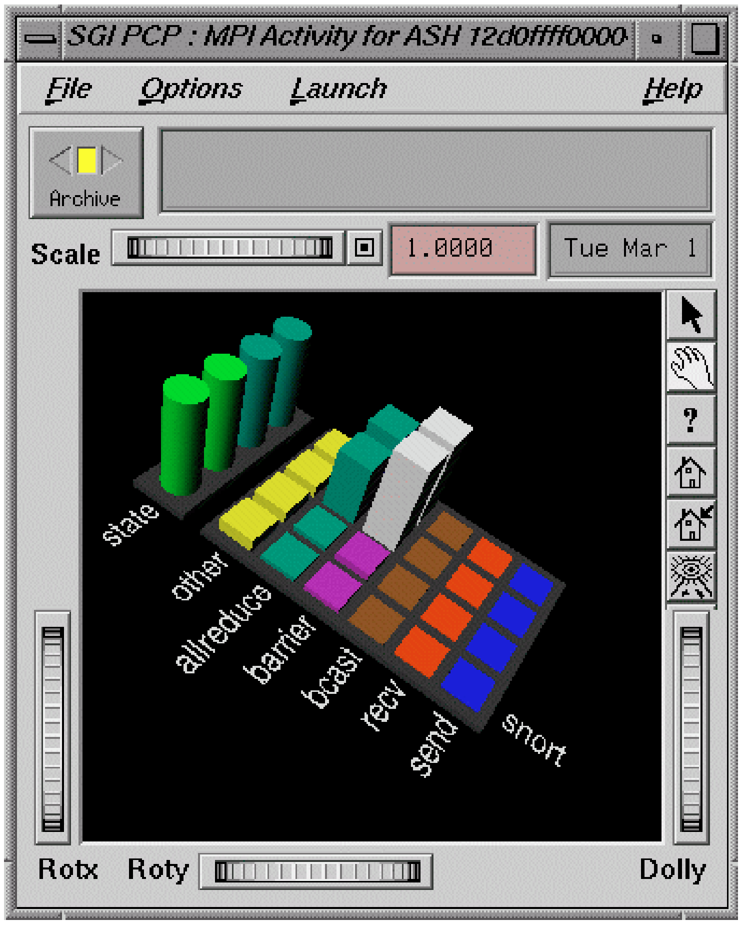

Figure 6-3 mpivis Tool

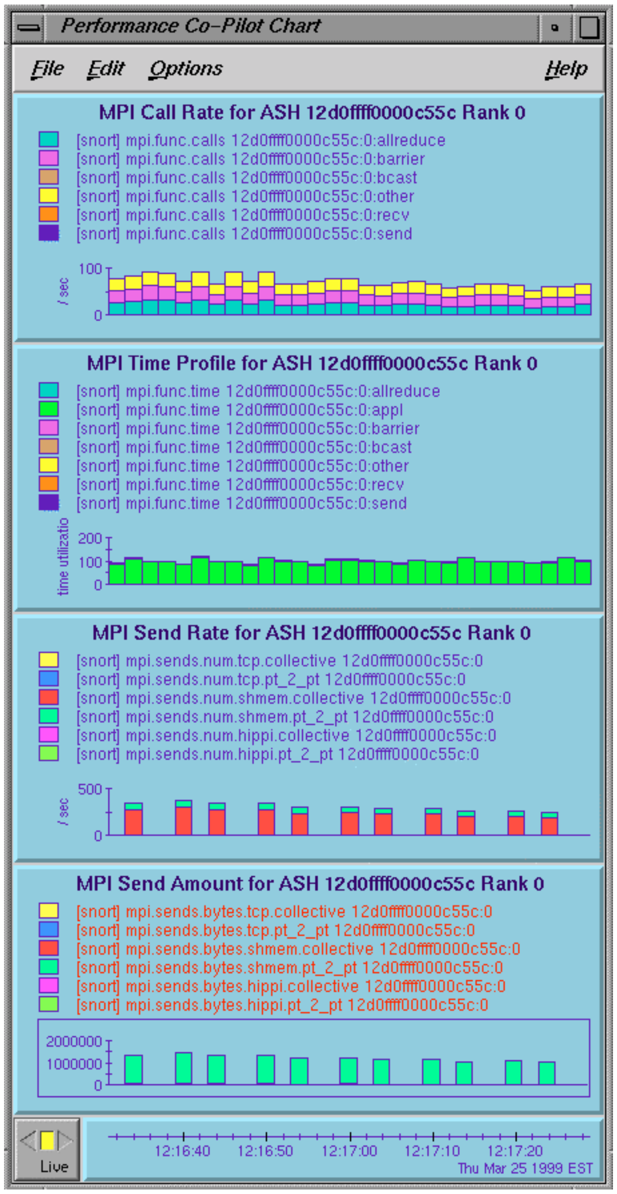
Performance Co-Pilot Chart
File
Edit
Options
Help
MPI Call Rate for ASH 12d0ffff0000c55c Rank 0
[snort] mpi.func.calls 12d0ffff0000c55c:0:allreduce
[snort] mpi.func.calls 12d0ffff0000c55c:0:barrier
[snort] mpi.func.calls 12d0ffff0000c55c:0:bcast
[snort] mpi.func.calls 12d0ffff0000c55c:0:other
[snort] 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 6-4 mpimon Tool

Additional profiling

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 datacnt 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
...
```

Chapter 7

Frequently Asked Questions

This chapter provides answers to frequently asked questions about MPI.

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

Here are some things to investigate:

- Look at the last few lines in `/var/adm/SYSLOG` for any suspicious errors or warnings. For example, if your application tries to pull in a library that it cannot find, a message should appear here.
- 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.
- 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 be in for a wild ride 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.

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
```

MPI with dplace

```
setenv MPI_DSM_OFF
     mpirun -np 4 dplace -place file a.out
```

Starting with IRIX 6.5.13, MPI interoperates with dplace so that MPI cc-NUMA functionality is not actually turned off. This might change the performance characteristics of MPI with previous releases of IRIX and dplace. To disable this interaction, the user needs to set the `MPI_DPLACE_INTEROP_OFF` shell variable.

MPI with perfex

```
mpirun -np 4 perfex -mp -o file a.out
```

The `-o` option to perfex became available only in IRIX 6.5, so on systems released earlier than IRIX 6.5, you must use a shell script as described previously. However, a shell script allows you to view only the summary for the entire job; individual statistics for each process are possible only via the `-o` option.

MPI with rld

```
setenv _RLDN32_PATH /usr/lib32/rld.debug
     setenv _RLD_ARGS "-log outfile -trace"
     mpirun -np 4 a.out
```

You can create more than one output file, depending on whether you are running out of your home directory and whether you use a relative pathname for the file. The first file will be created in the same directory from which you are running your application, and will contain information that applies to your job. The second file will be created in your home directory and will contain (uninteresting) information about the login shell that `mpirun` created to run your job. If both directories are the same, the entries from both are merged into a single file.

MPI with Totalview

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

In this special case, you must run the tool on `mpirun` and not the other way around. Note also that Totalview uses the `-a` option and therefore, it must always appear as the first option of the `mpirun` command.

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

MPI with SHMEM

It is easy to mix SHMEM and MPI message passing in the same program. 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. For more information, see the `shmem`(3) man page.

I am unable to malloc() more than 700-1000 MB when I link with libmpi.

On IRIX systems released before IRIX 6.5, there are no `so_locations` entries for the MPI libraries. The way to fix this is to requickstart all versions of `libmpi` as follows:

```
cd /usr/lib32/mips3
     rqs32 -force_requickstart -load_address 0x2000000 ./libmpi.so
     cd /usr/lib32/mips4
     rqs32 -force_requickstart -load_address 0x2000000 ./libmpi.so
     cd /usr/lib64/mips3
     rqs64 -force_requickstart -load_address 0x2000000 ./libmpi.so
     cd /usr/lib64/mips4
```

```
rqs64 -force_requickstart -load_address 0x2000000 ./libmpi.so
```

Note that this code requires root access.

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.

If the error message states

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

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.

If the error message states

```
 *** 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, MPI calls `libsma` 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.

I have other MPI questions. Where can I read more about MPI?

The `MPI`(1) and `mpirun`(1) man pages are good places to start. Also see the MPI standards at `http://www.mpi-forum.org/docs/docs.html`.

Index