

Message Passing Toolkit: MPI Programmer's Manual

007-3687-005

CONTRIBUTORS

Written by Julie Boney
Illustrations by Chris Wengelski
Production by Diane Ciardelli

COPYRIGHT

© 1996, 2001 Silicon Graphics, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND

The electronic (software) version of this document was developed at private expense; if acquired under an agreement with the USA government or any contractor thereto, it is acquired as "commercial computer software" subject to the provisions of its applicable license agreement, as specified in (a) 48 CFR 12.212 of the FAR; or, if acquired for Department of Defense units, (b) 48 CFR 227-7202 of the DoD FAR Supplement; or sections succeeding thereto. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy 2E, Mountain View, CA 94043-1351.

TRADEMARKS AND ATTRIBUTIONS

IRIS, IRIX, and Silicon Graphics are registered trademarks and IRIS InSight, SGI, and the SGI logo are trademarks of Silicon Graphics, Inc.

DynaWeb is a trademark of INSO Corporation. Kerberos is a trademark of Massachusetts Institute of Technology. MIPS is a trademark of MIPS Technologies, Inc. NFS is a trademark of Sun Microsystems, Inc. PostScript is a trademark of Adobe Systems, Inc. TotalView is a trademark of Bolt Beranek and Newman Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. X/Open is a registered trademark of X/Open Company, Ltd.

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

New Features

This revision of the *Message Passing Toolkit: MPI Programmer's Guide* supports the 1.5 release of the Message Passing Toolkit (MPT) for IRIX and Linux systems. The following new features are included with this release:

- GSN support
- Myrinet support
- Linux support (as a beta release)

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.

Contents

About This Manual	xi
Related Publications and Other Sources	xi
Conventions	xii
Reader Comments	xiii
1. Overview	1
MPI Overview	1
MPI Components	2
MPI Program Development	3
2. Building MPI Applications	5
Compiling and Linking IRIX Programs	5
Compiling and Linking Linux Programs	6
3. Using <code>mpirun</code> to Execute Applications	7
Syntax of the <code>mpirun</code> Command	7
Using a File for <code>mpirun</code> Arguments	12
Launching Programs on the Local Host Only	12
Using <code>mpirun(1)</code> to Run Programs in Shared Memory Mode	13
Launching a Distributed Program	13
4. Thread-Safe MPI	15
Initialization	15
Query Functions	16
Requests	16
007-3687-005	vii

Probes	16
Collectives	16
Exception Handlers	17
Signals	17
Internal Statistics	17
Finalization	17
5. Multiboard Feature for GSN and HiPPI 800 Networks	19
6. Setting Environment Variables	21
Setting MPI Environment Variables	21
Internal Message Buffering in MPI	32
7. MPI Troubleshooting and Application Tuning	35
What does MPI: could not run executable mean?	35
Can this error message be more descriptive?	35
Is there anything else I can do?	35
In the meantime, how can I figure out why mpirun is failing?	35
How do I combine MPI with other tools?	37
Combining MPI with dplace	38
Combining MPI with perfex	38
Combining MPI with rld	38
Combining MPI with TotalView	38
How can I allocate more than 700 to 1000 MB when I link with libmpi?	39
Why does my code run correctly until it reaches MPI_Finalize(3) and then hang?	39
Why do I keep getting error messages about MPI_REQUEST_MAX being too small, no matter how large I set it?	40
Why am I not seeing stdout or stderr output from my MPI application?	40
Index	41

Tables

Table 5-1	Algorithms for assigning multiple HiPPI adapters to MPI processes	20
Table 6-1	MPI Environment Variables	21
Table 6-2	Outline of Improper Dependence on Buffering	33

About This Manual

This publication documents the SGI version 1.5 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 1.5 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																				
<code>command</code>	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.																				
<code>manpage(x)</code>	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: <table><tbody><tr><td>1</td><td>User commands</td></tr><tr><td>1B</td><td>User commands ported from BSD</td></tr><tr><td>2</td><td>System calls</td></tr><tr><td>3</td><td>Library routines, macros, and opdefs</td></tr><tr><td>4</td><td>Devices (special files)</td></tr><tr><td>4P</td><td>Protocols</td></tr><tr><td>5</td><td>File formats</td></tr><tr><td>7</td><td>Miscellaneous topics</td></tr><tr><td>7D</td><td>DWB-related information</td></tr><tr><td>8</td><td>Administrator commands</td></tr></tbody></table> Some internal routines (for example, the <code>_assign_asgcmd_info()</code> routine) do not have man pages associated with them.	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
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																				
<i>variable</i>	Italic typeface denotes variable entries and words or concepts being defined.																				
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.																				
[]	Brackets enclose optional portions of a command or directive line.																				

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

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.

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 1.5 package contains the following components and the appropriate accompanying documentation:

- Parallel Virtual Machine (PVM) (IRIX only)
- 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
- Multiboard feature for GSN and HiPPI 800 networks
- Setting environment variables
- MPI troubleshooting and application tuning

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:

- Passing NULL arguments to `MPI_Init`.
- MPI I/O. MPT contains the ROMIO implementation of MPI I/O, in which a rich API for performing I/O in a message passing application is defined. Most of the standard-defined functionality is provided. For more information, see the `mpi_io(3)` man page.
- MPI one-sided communication (IRIX only). The `MPI_Win_create`, `MPI_Put`, `MPI_Get`, `MPI_Win_fence`, and `MPI_Win_free` routines are provided for single-host MPI jobs. For more information, see the `mpi_win(3)` man page.
- C++ bindings.
- Fortran 90 support for the `USE MPI` statement (IRIX only). Using the `USE MPI` statement instead of `INCLUDE 'mpif.h'` provides Fortran 90 programmers with parameter definitions and compile-time MPI subroutine call interface checking.
- MPI bindings for multi-threading inside an MPI process.

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
sgif90 -o myprog -I/usr/include myprog.f -lmpi
```

To use the 32-bit MPI library on Linux IA32 systems, choose one of the following commands:

```
g++ -o myprog myproc.C -lmpi++ -lmpi
gcc -o myprog myprog.c -lmpi
g77 -o myprog -I/usr/include myprog.f -lmpi
```

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
<code>-a[rray] array_name</code>	(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, <code>arrayd.conf</code> .
<code>-cpr</code>	(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 <code>mpirun</code> 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[file] 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. 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`.)
 - `%G` translates into the number of processes in `MPI_COMM_WORLD`.
 - `%h` translates into the rank of the host on which the process is running, relative to the `mpirun(1)` command line.
 - `%H` translates into the total number of hosts in the job.
 - `%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.
 - `%@` 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 "%@ (%1 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

```

`-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
<code>-f[file] file_name</code>	Specifies a text file that contains <code>mpirun</code> arguments (same as <i>global_options</i> .) For more details, see "Using a File for <code>mpirun</code> Arguments".
<code>-np np</code>	Specifies the number of processes on which to run.

`-nt nt` 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` or `-nt` 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
```

Using `mpirun(1)` to Run Programs in Shared Memory Mode

For running programs in MPI shared memory mode on a single host, the format of the `mpirun(1)` command is as follows:

```
mpirun -nt[nt ]programe
```

The `-nt` option specifies the number of tasks for shared memory MPI. A single UNIX process is run with multiple tasks representing MPI processes. The *programe* operand specifies the name of the program that you are running and its accompanying options.

Originally, the `-nt` option to `mpirun` was supported on IRIX systems for consistency across platforms. Because the default mode of execution on a single IRIX system is to use shared memory, the `-nt` option behaves the same as if you specified the `-np` option to `mpirun`. The following example runs ten instances of `a.out` in shared memory mode on `host_a`:

```
mpirun -nt 10 a.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
```

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.

Multiboard Feature for GSN and HiPPI 800 Networks

The SGI implementation of MPI automatically detects multiple high speed interconnect networks, including GSN and HiPPI 800 network adapters, and attempts to use the fastest interconnect by default. When multiple interface adapters are detected on a host, MPI attempts to use as many of them as possible when sending messages among hosts. During the initialization of the MPI job, each detected adapter is tested to determine which hosts it can reach. The adapter is then added to the list of available adapters for messages among the reachable hosts.

The multiboard feature is enabled by default. This implementation relaxes the requirements of earlier SGI MPI releases that the HiPPI interface adapters be located in the same board slot and have the same interface number, such as `hip0`.

When a high-speed interconnect network is detected, SGI MPI uses an OS Bypass protocol if the interconnect network is available for use on every host for the given MPI job. If the detected high speed interconnect network is not available for use on every host in a multihost MPI job, SGI MPI tries the next slower network until it finds one that meets this connection requirement. The order of attempted usage is GSN, HiPPI 800, and finally TCP/IP over the standard Ethernet interface. Note that one can redirect the TCP/IP traffic over an alternate Ethernet interface, such as TCP/IP-over-GSN or Gigabit Ethernet, by specifying the appropriate host name or IP address on the `mpirun` line.

When using the high speed OS Bypass protocol, the multiboard feature does not require that every host exist on a given subnet. This allows for network switch topologies that may be limited to 8 or 32 hosts per switch or per subnet. SGI MPI automatically detects the network topology during the startup phase. If every host is on at least one GSN or HiPPI 800 interconnect network, the OS Bypass protocol for that homogenous network, either GSN or HiPPI 800, is used.

When OS Bypass protocol is in use and multiple high speed GSN or HiPPI 800 connections are available between any two hosts, the MPI message traffic is sent over all of the available adapters. Note that SGI MPI automatically splits messages whose lengths exceed 16,384 bytes into smaller, 16,384-byte chunks. Individual chunks can be sent over different adapters, having a desirable load-balancing effect.

The multiboard feature allows users of the HiPPI 800 adapters to select the manner in which multiple adapters are assigned to MPI processes and the algorithm for

distributing the MPI job message traffic. You can change the method for selecting a HiPPI 800 adapter to use by setting the `MPI_BYPASS_DEV_SELECTION` environment variable. Table 5-1, page 20 describes the algorithms for the various settings.

Note that the GSN interconnect uses only the round robin selection method and is not affected by the `MPI_BYPASS_DEV_SELECTION` variable settings.

Table 5-1 Algorithms for assigning multiple HiPPI adapters to MPI processes

Setting	Description
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 16384 bytes).
1 (default)	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 (16,384 or fewer bytes) between processes on different hosts. Large messages (more than 16,384 bytes) are split into chunks of 16,384 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.

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 6-1, 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 6-1 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 <code>arrayd.conf</code> file.
MPI_BAR_DISSEM	Specifies the use of the alternate barrier algorithm, the dissemination/butterfly, within the <code>MPI_Barrier(3)</code> and <code>MPI_Win_fence(3)</code> functions. This alternate algorithm provides better performance on jobs with larger PE counts. The <code>MPI_BAR_DISSEM</code> option is recommended for jobs with PE counts of 64 or greater.	Disabled if job contains less than 64 PEs; otherwise, enabled.

Variable	Description	Default
MPI_BUFFER_MAX (IRIX systems only)	<p>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.</p> <p>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 <code>shmalloc</code> (<code>shpalloc</code>) functions in LIBSMA.</p> <p>Most 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.</p>	Not enabled.
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)

Variable	Description	Default
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)
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_DEVS (IRIX systems only)	<p>Sets the order for opening HIPPI adapters. The list of devices does not need to be space-delimited (0123 is also valid).</p> <p>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.</p> <p>When you know that a system has multiple HIPPI adapters, you can use the <code>MPI_BYPASS_DEVS</code> 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 <code>MPI_BYPASS_OFF</code> variable.</p>	0 1 2 3

6: Setting Environment Variables

Variable	Description	Default
	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_OFF (IRIX systems only)	Disables the HIPPI bypass.	Not enabled.
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.	
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.


Variable	Description	Default
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
MPI_DIR	Sets the working directory on a host. When an <code>mpirun</code> 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 <code>mpirun</code> , so this user must be listed in the <code>.rhosts</code> file on the corresponding nodes. By default, the working directory for the session is the user's <code>\$HOME</code> 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 <code>MPI_DIR</code> variable to a different directory.	<code>\$HOME</code> on the node. If using <code>-np</code> or <code>-nt</code> , the default is the current directory.
MPI_DSM_CPUCLUSTER (IRIX systems only)	When set on an Origin 2000 or an Origin 3000 system running IRIX 6.5.11 or greater, <code>TOPOLOGY_CPUCLUSTER</code> will be used for <code>mld</code> placement and the number of processes to be mapped to every memory (<code>MPI_DSM_PPM</code>) will be set to 1.	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 <code>process_cpulink(3)</code> function instead of <code>process_mldlink(3)</code> to control memory placement. You should not use <code>MPI_DSM_MUSTRUN</code> when the job is submitted to Miser (see <code>miser_submit(1)</code>) 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.
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_VERBOSE (IRIX systems only)	Instructs <code>mpirun</code> to print information about process placement for jobs running on NUMA systems.	Not enabled.

Variable	Description	Default
MPI_GM_ON	Enables use of GM (Myrinet) software. MPI attempts to establish Myrinet connections among all hosts involved in the job. If unable to do so, TCP/IP is used for interhost communication.	Not enabled.
MPI_GM_VERBOSE	Allows some diagnostic information concerning messaging between processes using GM (Myrinet) to be displayed on <code>stderr</code> .	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.

Variable	Description	Default
MPI_GSN_ON (IRIX 6.5.9 systems or later)	<p>Enables use of the GSN (ST protocol) bypass. MPI attempts to establish GSN connections among all hosts in the job. If unable to do so, HIPPI bypass connections will be attempted. If HIPPI is unavailable on all hosts, TCP/IP will be used for interhost communication.</p> <p>GSN imposes a limit of one MPI process using GSN per CPU on a system. So, 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.</p> <p>Once the maximum allowed MPI processes using GSN is reached, subsequent MPI jobs will return an error to the user output such as the following:</p> <pre>MPI: gsn_endpoint/st_endpoint: Resource temporarily unavailable</pre> <p>An error will also be printed to the SYSLOG file.</p> <p>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.</p>	Not enabled.
MPI_GSN_VERBOSE (IRIX 6.5.9 systems or later)	<p>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.</p>	Not enabled.

Variable	Description	Default
MPI_MAX_MSGS	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.
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

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.	1024
	<div style="display: flex; align-items: center;">  <p>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.</p> </div> <hr/> <p>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.</p>	
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

Variable	Description	Default
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 <code>stderr</code> .	Not enabled.
MPI_SLAVE_DEBUG_ATTACH	Specifies the MPI process to be debugged. If you set <code>MPI_SLAVE_DEBUG_ATTACH</code> to <i>N</i> , the MPI process with rank <i>N</i> prints a message during program startup, describing how to attach to it from another window using the <code>dbx</code> debugger on IRIX or the <code>gdb</code> debugger on Linux. You must attach the debugger to process <i>N</i> within ten seconds of the printing of the message.	Not enabled.

Variable	Description	Default
MPI_STATS	<p>Enables printing of MPI internal statistics. Each MPI process prints statistics about the amount of data sent with MPI calls during the <code>MPI_Finalize</code> process. Data is sent to <code>stderr</code>. To prefix the statistics messages with the MPI rank, use the <code>-p</code> option on the <code>mpirun</code> command.</p> <hr/> <p>Note: Because the statistics-collection code is not thread-safe, this variable should not be set if the program uses threads.</p> <hr/>	Not enabled.
MPI_TYPE_DEPTH	<p>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 <code>MPI_TYPE_DEPTH</code> is exceeded.</p>	8 levels
MPI_TYPE_MAX	<p>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.)</p>	1024

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 6-2, page 33 illustrates a simple sequence of MPI operations that cannot work unless messages are buffered. If sent messages were not buffered, each process would hang in the initial `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 also be equal to or greater than the size setting for `MPI_BUFFER_MAX` (see Table 6-1, page 21). 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 6-2, page 33 will almost certainly deadlock if this bufferless pathway is available.

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

Table 6-2 Outline of Improper Dependence on Buffering

Process 1	Process 2
<code>MPI_Send(2,)</code>	<code>MPI_Send(1,)</code>
<code>MPI_Recv(2,)</code>	<code>MPI_Recv(1,)</code>

MPI Troubleshooting and Application Tuning

This chapter provides answers to frequently asked questions about MPI.

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.

Can this error message be more descriptive?

No, because of the highly decoupled interface between `mpirun` and `arrayd`, no other information is directly available. `mpirun` asks `arrayd` to launch a master process on each host and listens on a socket for those masters to connect back to it. Because the masters are children of `arrayd`, whenever one of the masters terminates, `arrayd` traps `SIGCHLD` and passes that signal back to `mpirun`. If `mpirun` receives a signal before it has established connections with every host in the job, a problem has occurred. In other words, one of two possible bits of information is available to `mpirun` in the early stages of initialization: success or failure.

Is there anything else I can do?

You could create an `mpicheck` utility (similar to `ascheck`), which could run some simple experiments and look for things that are obviously broken from the `mpirun` point of view.

In the meantime, how can I figure out why `mpirun` is failing?

You can use the following checklist:

- 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.
- Check for misspelling of your application name.

- 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 `$PWD`. However, different functionality is required sometimes. For more information, see the `mpirun(1)` man page description of the `-dir` option.
- If you are using a relative path name for your application, be sure that it appears in `$PATH`. In particular, `mpirun` will not look in the `.` file for your application unless `.` appears in `$PATH`.
- Run `/usr/etc/ascheck` to verify that your array is configured correctly.
- Be sure that you can use `rsh` (or `arshell`) to connect to all of the hosts that you are trying to use, without entering a password. This means that either the `/etc/hosts.equiv` or the `~/.rhosts` file must be modified to include the names of every host in the MPI job. Note that using the `-np` syntax (that is, not specifying host names) is equivalent to typing `localhost`, so a `localhost` entry is also needed in either the `/etc/hosts.equiv` or the `~/.rhosts` file.
- If you are using an MPT module to load MPI, try loading it directly from within your `.cshrc` file instead of from the shell. If you are also loading a ProDev module, be sure to load it after the MPT module.
- To verify that you are running the version of MPI that you think you are, use the `-verbose` option of the `mpirun(1)` command.
- Be very careful when setting MPI environment variables from within your `.cshrc` or `.login` files, because these settings will override any settings that you might later set from within your shell (because MPI creates a fresh login session for every job). The safe way to set up environment variables is to test for the existence of `$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 might encounter difficulty because currently, `mpirun` cannot pass tokens. For example, if you use `telnet` to connect to a host and then try to run `mpirun` on that host, the process fails. But if you use `rsh` instead to connect to the host, `mpirun` succeeds. (This might be because `telnet` is kerberized but `rsh` is not.) If you are running under a Kerberos environment, you should talk to the local administrators about the proper way to launch MPI jobs.

How do I combine MPI with other tools?

As a general rule, you should run `mpirun` on your tool and then run the tool on your application. Do not try to run the tool on `mpirun`. Also, because of the way that `mpirun` sets up `stdio`, it might require some effort to see the output from your tool.

The simplest case is that in which 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`. However, not all tools (for example, `dplace`) support such an option. Fortunately, however, you can usually wrap a shell script around the tool and let the script perform the following redirection:

```
> 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
```

Combining MPI with `dplace`

To combine MPI with the `dplace` tool, use the following code:

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

Combining MPI with `perfex`

To combine MPI with the `perfex` tool, use the following code:

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

The `-o` option to `perfex` became available for the first time in IRIX 6.5. On earlier systems, you can use a shell script, as previously described. However, a shell script allows you to view only the summary for the entire job. You can view individual statistics for each process only by using the `-o` option.

Combining MPI with `rld`

To combine MPI with the `rld` tool, use the following code:

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

You can create more than one `outfile`, depending on whether you are running out of your home directory and whether you use a relative path name for the file. The first 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 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.

Combining MPI with TotalView

To combine MPI with the TotalView tool, use the following code:

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

In this one special case, you must run the tool on `mpirun` and not the other way around. Because TotalView uses the `-a` option, this option must always appear as the first option on the `mpirun` command.

How can I allocate more than 700 to 1000 MB when I link with `libmpi`?

At times, it might be necessary to change the `so_location` entries for the MPI libraries. To do this, you need 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: This procedure requires root access.

Why does my code run correctly until it reaches `MPI_Finalize(3)` and then hang?

This problem is almost always caused by `send` or `recv` requests that are either unmatched or incomplete. An unmatched request would be any blocking `send` request for which a corresponding `recv` request is never posted. An incomplete request would be any nonblocking `send` or `recv` request that was never freed by a call to `MPI_Test(3)`, `MPI_Wait(3)`, or `MPI_Request_free(3)`.

Common examples of unmatched or incomplete requests are applications that call `MPI_Isend(3)` and then use internal means to determine when it is safe to reuse the send buffer and, therefore, never call `MPI_Wait(3)`. You can fix such codes easily by inserting a call to `MPI_Request_free(3)` immediately after all such `send` requests.

Why do I keep getting error messages about `MPI_REQUEST_MAX` being too small, no matter how large I set it?

You are probably calling `MPI_Isend(3)` or `MPI_Irecv(3)` and not completing or freeing your request objects. You should use `MPI_Request_free(3)`, as described in the previous question.

Why am I not seeing `stdout` or `stderr` output from my MPI application?

Beginning with the MPI 3.1 release, all `stdout` and `stderr` output is line-buffered, which means that `mpirun` will 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 is to append a newline character to each prompt.

Index

A		argument file	12
		command	7
		for distributed programs	13
		for local host	12
		for shared memory	13
		MPT	
		components	1
		overview	1
		multiboard feature	19
B			
	building MPI applications		5
D			
	distributed programs		13
E			
	environment variable setting		21
F			
	frequently asked questions		35
I			
	internal message buffering		32
M			
	MPI		
	components		2
	overview		1
	mpirun		
007-3687-005			
		program development	3
		program segments	13
P			
S			
		shared memory	13
		sprocs	15
T			
		threads	
		collectives	16
		exception handlers	17
		finalization	17
		initialization	15
		internal statistics	17
		probes	16
		query functions	16
		requests	16
		signals	17
		thread-safe systems	15
		troubleshooting	35