

Message Passing Toolkit: MPI Programmer's Manual

007-3687-003

Copyright © 1996, 2000 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.

CONTRIBUTORS

Written by Julie Boney

Edited by Susan Wilkening

Illustrations by Chris Wengelski

Production by Susan Gorski

LIMITED AND RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Data clause at FAR 52.227-14 and/or in similar or successor clauses in the FAR, or in the DOD, DOE or NASA FAR Supplements. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy., Mountain View, CA 94043-1351.

IRIS, IRIX, and Silicon Graphics are registered trademarks and IRIS InSight 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 Limited. X/Open is a registered trademark of X/Open Company Ltd.

New Features in This Manual

This rewrite of the *Message Passing Toolkit: MPI Programmer's Manual* supports the 1.4 release of the Message Passing Toolkit for IRIX (MPT).

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.

Contents

About This Manual	xv
Related Publications	xv
Other Sources	xv
Obtaining Publications	xvi
Conventions	xvi
Reader Comments	xvii
 1. Overview	 1
MPI Overview	1
MPI Components	2
MPI Program Development	3
 2. Building MPI Applications	 5
 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	11
Launching Programs on the Local Host Only	11
Using <code>mpirun(1)</code> to Run Programs in Shared Memory Mode	12
Launching a Distributed Program	12
 4. Thread-Safe MPI	 15
Initialization	15
Query Functions	16
Requests	16
Probes	16
 007-3687-003	 vii

Collectives	16
Exception Handlers	17
Signals	17
Internal Statistics	17
Finalization	17
5. Multiboard Feature	19
6. Setting Environment Variables	21
Setting MPI Environment Variables	21
Internal Message Buffering in MPI	25
7. Launching Programs with NQE	27
Starting NQE	27
Submitting a Job with NQE	27
Checking Job Status with NQE	29
Getting More Information	30
8. MPI Troubleshooting	31
What does MPI: could not run executable mean?	31
Can this error message be more descriptive?	31
Is there something more that can be done?	31
In the meantime, how can we figure out why mpirun is failing?	31
How do I combine MPI with other tools?	33
Combining MPI with dplace	34
Combining MPI with perfex	34
Combining MPI with rld	34
Combining MPI with TotalView	34
How can I allocate more than 700 to 1000 MB when I link with libmpi?	35

Why does my code run correctly until it reaches <code>MPI_Finalize(3)</code> and then hang? . . .	35
Why do I keep getting error messages about <code>MPI_REQUEST_MAX</code> being too small, no matter how large I set it?	36
Why am I not seeing <code>stdout</code> or <code>stderr</code> output from my MPI application?	36
Index	37

Figures

Figure 7-1	NQE button bar	27
Figure 7-2	NQE Job Submission window	28
Figure 7-3	NQE Status window	29
Figure 7-4	NQE Detailed Job Status window	30

Tables

Table 6-1	MPI Environment Variables	21
Table 6-2	Outline of Improper Dependence on Buffering	25

About This Manual

This publication documents the Message Passing Toolkit for IRIX (MPT) 1.4 implementation of the Message Passing Interface (MPI) supported on SGI MIPS based systems running IRIX release 6.5 or later.

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 1.4 release is a software package that supports parallel programming across a network of computer systems through a technique known as *message passing*.

Related Publications

The following documents contain additional information that might be helpful:

- *Message Passing Toolkit: PVM Programmer's Manual*
- *Application Programmer's Library Reference Manual*
- *Installing Programming Environment Products*

To obtain the *Message Passing Toolkit: PVM Programmer's Manual*, see "Obtaining Publications," page xvi. To obtain the *Application Programmer's Library Reference Manual* and *Installing Programming Environment Products* manuals, contact the Minnesota Distribution Center at +651 683 5907. SGI employees can contact the Distribution Center by sending e-mail to orderdsk@sgi.com.

Other Sources

Material about MPI is available from a variety of other sources. Some of these, particularly World Wide 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 World Wide 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)

Books:

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

Newsgroup:

- `comp.parallel.mpi`

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at `http://techpubs.sgi.com`.

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><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></table>	1	User commands	1B	User commands ported from BSD	2	System calls	3	Library routines, macros, and opdefs
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 MIPS based systems running 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 can be found on the back cover.)

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

The MPT 1.4 package contains the following components and the appropriate accompanying documentation:

- Parallel Virtual Machine (PVM)
- Message Passing Interface (MPI)
- Logically shared, distributed memory (SHMEM) data-passing routines

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
- Setting environment variables
- Launching programs with NQE

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 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 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. 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. 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 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 program, as in the following examples:

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

```
CC -64 compute.C -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
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 higher 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
```


Using `mpirun` to Execute Applications

The `mpirun(1)` command is the primary job launcher for the MPT implementations of MPI. The `mpirun` command must be used whenever a user wishes to run an MPI application on an IRIX 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>	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>-d[ir] path_name</code>	Specifies the working directory for all hosts. In addition to normal path names, the following special values are recognized: <ul style="list-style-type: none"> <code>.</code> Translates into the absolute path name of the user's current working directory on the local host. This is the default. <code>~</code> Specifies the use of the value of <code>\$HOME</code> as it is defined on each machine. In general, this value can be different on each machine.
<code>-f[file] file_name</code>	Specifies a text file that contains <code>mpirun</code> 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 "%@ (%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
```

`-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 only one entry needs to be specified.

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 mpirun arguments (same as <i>global_options</i> .) For more details, see "Using a File for mpirun 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 on the local host only 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 -nt 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. Since 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 Origin system, or can be any accessible computer running Array Services software. Array Services software runs on IRIX systems and must be running to launch MPI programs. For available nodes on systems running Array Services software, see the `/usr/lib/array/arrayd.conf` file.

You can list multiple entries on the mpirun command line. Each entry contains an MPI executable file and a combination of hosts and process counts for running it. This gives you the ability to start different executable files on the same or different hosts as part of the same MPI application.

The 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

The SGI implementation of MPI 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 at 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.

Blocking MPI calls block the calling thread only, 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.

It is possible to 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 up to the user 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 at each process. If concurrent

threads issue such calls on the communicator, window, or file handle, it is up to the user 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 up to the user 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

MPI automatically detects multiple HIPPI network adapters and uses as many of them as possible when sending messages among hosts. The multiboard feature uses a "round robin" selection scheme in choosing the next available adapter over which to send the current message. The message is sent entirely over one adapter.

During the initialization of the MPI job, each detected adapter is tested to determine which hosts it can reach. It is then added to the list of available adapters for messages among the reachable hosts.

By means of the multiboard feature, messages are sent over as many HIPPI network adapters as are available between any pair of hosts.

The multiboard feature is enabled by default and relaxes the requirements of earlier MPI releases that the HIPPI interface adapters be located in the same board slot and have the same interface number, such as `hip0`. A series of new environment variables with this release allows the user to further specify the desired network connection.

Setting Environment Variables

This chapter describes the variables that specify the environment under which your MPI programs will run. Environment variables have predefined values. You can change some variables to achieve particular performance objectives; others are required values for standard-compliant programs.

Setting MPI Environment Variables


This section provides a table of MPI environment variables you can set for IRIX systems.

Table 6-1 MPI Environment Variables

Variable	Description	Default
MPI_ARRAY	Sets an alternative array name to be used for communicating with Array Services when a job is being launched.	The default name set in the <code>arrayd.conf</code> file
MPI_BUFS_PER_HOST	Determines the number of shared message buffers (16 KB each) that MPI is to allocate for each host. These buffers are used to send long messages.	16 pages (each page is 16 KB)
MPI_BUFS_PER_PROC	Determines the number of private message buffers (16 KB each) that MPI is to allocate for each process. These buffers are used to send long messages.	16 pages (each page is 16 KB)
MPI_BYPASS_DEVS	<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 Kbytes or shorter.</p>	0 1 2 3

Variable	Description	Default
	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. This variable can be used 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.	
	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.	
<code>MPI_BYPASS_OFF</code>	Disables the HIPPI bypass.	Not enabled
<code>MPI_BYPASS_SINGLE</code>	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.	
<code>MPI_BYPASS_VERBOSE</code>	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.	
<code>MPI_CHECK_ARGS</code>	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
<code>MPI_COMM_MAX</code>	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 <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 responding 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_OFF	Turns off nonuniform memory access (NUMA) optimization in the MPI library.	Not enabled
MPI_DSM_MUSTRUN	Specifies the CPUs on which processes are to run. You can set the <code>MPI_DSM_VERBOSE</code> variable to request that the <code>mpirun</code> command print information about where processes are executing.	Not enabled
MPI_DSM_PPM	Sets the number of MPI processes that can be run on each node of an IRIX system.	2
MPI_DSM_VERBOSE	Instructs <code>mpirun</code> to print information about process placement for jobs running on NUMA systems.	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

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 the MPI_MSGS_PER_HOST variable to allocate buffer space for interhost messages.	128
	 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.	
MPI_MSGS_PER_PROC	Sets the maximum number of buffers to be allocated from sending process space for outbound messages going to the same host. (May be required by standard-compliant programs.) MPI allocates buffer space for local messages based on the message destination. Space for messages that are destined for local processes is allocated as additional process space for the sending process.	128
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.)	1024
MPI_TYPE_DEPTH	Sets the maximum number of nesting levels for derived datatypes. (May be required by standard-compliant programs.) The MPI_TYPE_DEPTH variable limits the maximum depth of derived datatypes 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

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 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 25 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, often a program such as this will not hang. The `MPI_Send` calls return after putting the messages into buffer space, and the `MPI_Recv` calls get the messages. Nevertheless, program logic such as this is not valid by the MPI standard. The SGI implementation of MPI for IRIX systems buffers messages of all sizes. For buffering purposes, this implementation recognizes short message lengths (64 bytes or shorter) and long message lengths (longer than 64 bytes).

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>

Launching Programs with NQE

After an MPI program is debugged and ready to run in a production environment, it is often useful to submit it to a queue to be scheduled for execution. The Network Queuing Environment (NQE) provides this capability. NQE selects a node appropriate for the resources that an MPI job needs, routes the job to a node, and schedules it to run.

This chapter explains how to use the NQE graphical interface to submit an MPI program for execution.

Starting NQE

Before you begin, set your `DISPLAY` variable so that the NQE screens appear on your workstation. Then enter the `nqe` command, as shown in the following example:

```
setenv DISPLAY myworkstation:0
<nqe
```

Figure 7-1 shows the NQE button bar, which appears after your entry.

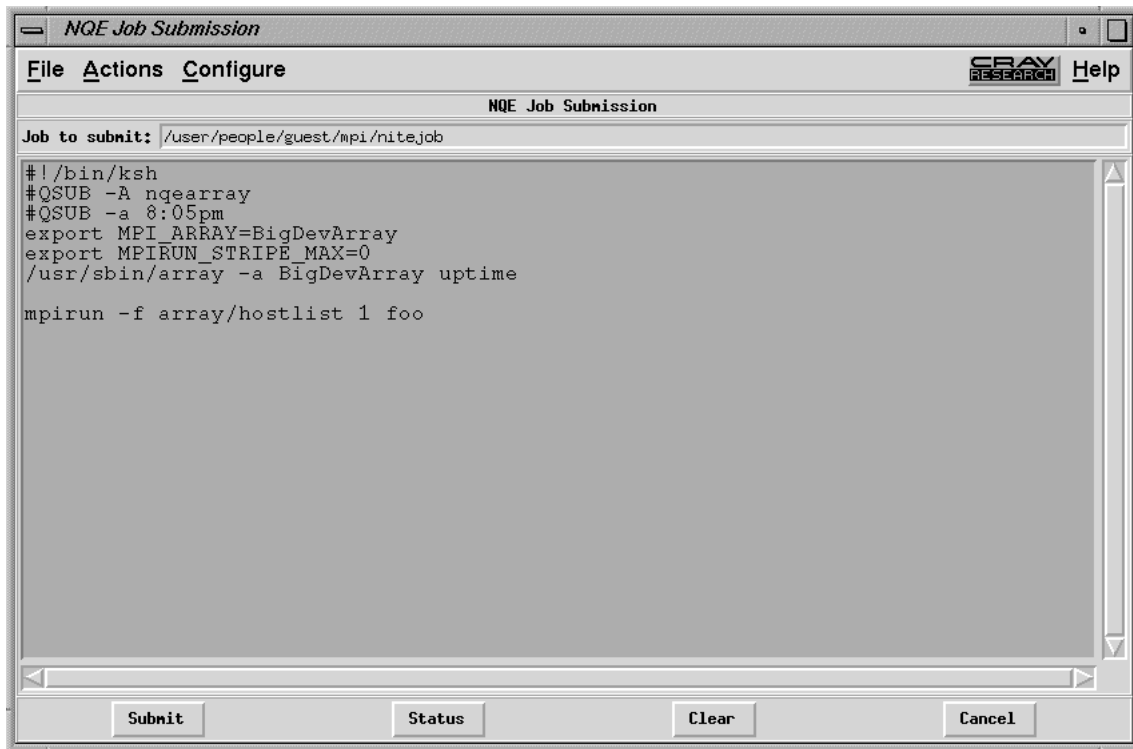


a11378

Figure 7-1 NQE button bar

Submitting a Job with NQE

To submit a job, click the **Submit** button on the **NQE Job Submission** window. Figure 7-2 shows the **NQE Job Submission** window with a sample job script ready to be submitted.



a11379

Figure 7-2 NQE Job Submission window

Notice in this figure that the difference between an NQE job request and a shell script lies in the use of the #QSUB identifiers. In this example, the directive #QSUB -A ngearray tells NQE to run this job under the ngearray project account. The directive #QSUB -a 8:05pm tells NQE to wait until 8:05 p.m. to start the job.

Also notice in Figure 7-2 that the MPI program is already compiled and distributed to the proper hosts. The file array/hostlist has the list of parameters for this job, as you can see in the output from the following cat command:

```
% cat array/hostlist
homegrown, disarray, dataarray
```

Checking Job Status with NQE

To see the status of jobs running under NQE, click the **Status** button to display the **NQE Status** window.

Figure 7-3 shows an example of the **NQE Status** window. Notice in this figure that the MPI job is queued and waiting to run.

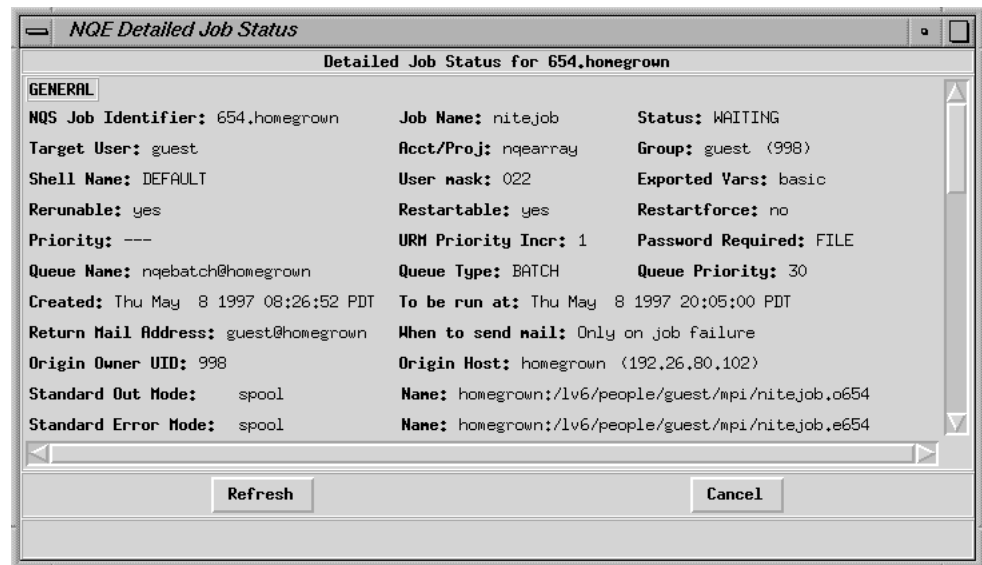


a11380

Figure 7-3 NQE Status window

To verify the scheduled starting time for the job, position the mouse cursor on the line that shows the job and double-click it.

This displays the **NQE Detailed Job Status** window, shown in Figure 7-4. Notice that the job was created at 8:26 PDT and is to run at 20:05 PDT.



a11381

Figure 7-4 NQE Detailed Job Status window

Getting More Information

For more information on using NQE, see the following NQE publications:

- *Introducing NQE*
- *NQE Release Overview*
- *NQE Installation*
- *NQE Administration*
- *NQE User's Guide*

To obtain the preceding SGI documentation, go to the SGI Technical Publications Library at <http://techpubs.sgi.com>.

For general information about NQE, see the following URL:

<http://www.sgi.com/software> (search for NQE)

MPI Troubleshooting

This chapter provides answers to frequently asked questions about MPI.

What does MPI: could not run executable mean?

It 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, that is an indication that something has gone wrong. In other words, there is one of two possible bits of information available to `mpirun` in the early stages of initialization: success or failure.

Is there something more that can be done?

One proposed idea is to 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 we 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` is unable to 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?

In general, the rule to follow is to 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. But fortunately, it is usually possible to "roll your own" by wrapping a shell script around the tool and having 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 will allow 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`?

On IRIX versions earlier than 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: 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 not completed. 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 bother to call `MPI_Wait(3)`. Such codes can be fixed 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 our 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 for this is to append a newline character to each prompt.

Index

A

adapter selection, 19

B

building MPI applications, 5

D

distributed programs, 12

E

environment variable setting, 21

F

frequently asked questions, 31

I

internal message buffering, 25

M

MPI
 components, 2
 overview, 1
mpirun
 argument file, 11

command, 7
for distributed programs, 12
for local host, 11
for shared memory, 12

MPT

 components, 1
 overview, 1
multiboard feature, 19

N

Network Queuing Environment (NQE), 27

P

program development, 3
program segments, 12

S

shared memory
 using mpirun, 12
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, 31