

Message Passing Toolkit: MPI Programmer's Manual

Document Number 007-3687-002

St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

Copyright © 1996, 1998, 1999 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 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 or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

The MPI implementation for the CRAY T3E system is derived from the implementation of MPI for the CRAY T3D system developed at Edinburgh Parallel Computing Centre. The software is supplied to Cray Research under license from The University of Edinburgh.

Autotasking, CF77, CRAY, Cray Ada, CraySoft, CRAY Y-MP, CRAY-1, CRInform, CRI/*TurboKiva*, HSX, LibSci, MPP Apprentice, SSD, SUPERCLUSTER, UNICOS, and X-MP EA are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, CRAY APP, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, CRAY J90se, CrayLink, Cray NQS, Cray/REELibrarian, CRAY S-MP, CRAY SSD-T90, CRAY SV1, CRAY T90, CRAY T3D, CRAY T3E, CrayTutor, CRAY X-MP, CRAY XMS, CRAY-2, CSIM, CVT, Delivering the power . . . , DGauss, Docview, EMDS, GigaRing, HEXAR, IOS, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNETH, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, UNICOS MAX, and UNICOS/mk are trademarks of Cray Research, Inc., a wholly owned subsidiary of Silicon Graphics, Inc.

DynaWeb is a trademark of INSO Corporation. IRIS, IRIX, and Silicon Graphics are registered trademarks and IRIS InSight and the Silicon Graphics logo are trademarks of Silicon Graphics, Inc. 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.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

New Features

Message Passing Toolkit: MPI Programmer's Manual

007-3687-002

This rewrite of *Message Passing Toolkit: MPI Programmer's Manual* supports the 1.3 release of the Cray Message Passing Toolkit and the Message Passing Toolkit for IRIX (MPT). New features include MPI statistics, MPI I/O, thread-safe MPI, multiboard adapter selection, and other performance improvements. With the MPT 1.3 release, support for XMPI has been dropped.

Record of Revision

<i>Version</i>	<i>Description</i>
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.

Contents

	<i>Page</i>
About This Manual	vii
Related Publications	vii
Other Sources	viii
Obtaining Publications	viii
Conventions	ix
Reader Comments	xi
Overview [1]	1
MPI Overview	1
MPI Components	2
MPI Program Development	2
Procedure 1: Steps for MPI program development	2
Building MPI Applications [2]	3
Building Applications on UNICOS Systems	3
Building Applications That Use Shared Memory MPI on UNICOS Systems	3
Shared Memory MPI Limitations	6
Building Files on UNICOS Systems When Using TCP	7
Building Applications on UNICOS/mk Systems	8
Building Applications on IRIX Systems	8
Using mpirun to Execute Applications [3]	9
Syntax of the mpirun Command	9
Using a File for mpirun Arguments (UNICOS or IRIX)	13
Launching Programs on the Local Host Only	13
Using mpirun(1) to Run Programs in Shared Memory Mode (UNICOS or IRIX)	14
007-3687-002	iii

	<i>Page</i>
Using the <code>mpirun(1)</code> Command on UNICOS/mk Systems	14
Executing UNICOS/mk Programs Directly	14
Launching a Distributed Program (UNICOS or IRIX)	15
Thread-Safe MPI [4]	17
Initialization	17
Query Functions	18
Requests	18
Probes	18
Collectives	18
Exception Handlers	19
Signals	19
Internal Statistics	19
Finalization	19
Multiboard Feature [5]	21
Setting Environment Variables [6]	23
Setting MPI Environment Variables on UNICOS and IRIX Systems	23
Setting MPI Environment Variables on UNICOS/mk Systems	27
Internal Message Buffering in MPI (IRIX Systems Only)	28
Launching Programs with NQE [7]	31
Starting NQE	31
Submitting a Job with NQE	31
Checking Job Status with NQE	33
Getting More Information	34
MPI Troubleshooting [8]	37
What does MPI: could not run executable mean?	37
Can this error message be more descriptive?	37

	<i>Page</i>
Is there something more that can be done?	37
In the meantime, how can we figure out why mpirun is failing?	37
How do I combine MPI with other tools?	38
Combining MPI with dplace	39
Combining MPI with perfex	40
Combining MPI with rld	40
Combining MPI with TotalView	40
How can I allocate more than 700 to 1000 MB when I link with libmpi?	41
Why does my code run correctly until it reaches MPI_Finalize(3) and then hang?	41
Why do I keep getting error messages about MPI_REQUEST_MAX being too small, no matter how large I set it?	41
Why am I not seeing stdout or stderr output from my MPI application?	42
Index	43
Figures	
Figure 1. NQE button bar	31
Figure 2. NQE NQE Job Submission window	32
Figure 3. NQE Status window	33
Figure 4. NQE Detailed Job Status window	34
Tables	
Table 1. assign examples	6
Table 2. MPI environment variables for IRIX systems only	23
Table 3. MPI environment variables for UNICOS and IRIX systems	25
Table 4. Environment variables for UNICOS/mk systems	27
Table 5. Outline of improper dependence on buffering	29

About This Manual

This publication documents the Cray Message Passing Toolkit and Message Passing Toolkit for IRIX (MPT) 1.3 implementation of the Message Passing Interface (MPI) supported on the following platforms:

- Cray PVP systems running UNICOS release 10.0 or later. The MPT 1.3 release requires a bugfix package to be installed on UNICOS systems running release 10.0 or later. The bugfix package, `MPT12_OS_FIXES`, is available through the `getfix` utility. It is also available from the anonymous FTP site `ftp.cray.com` in directory `/pub/mpt/fixes/MPT12_OS_FIXES`.
- CRAY T3E systems running UNICOS/mk release 1.5 or later
- Silicon Graphics MIPS based systems running IRIX release 6.2 or later
IRIX 6.2 systems running MPI require the kernel rollup patch 1650 or later.
IRIX 6.3 systems running MPI require the kernel rollup patch 2328 or later.

IRIX systems running MPI applications must also be running Array Services software version 3.0 or later. MPI consists of a library, a profiling library, and commands that support MPI. The MPT 1.3 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*

All of these are Cray publications and can be ordered from the Minnesota Distribution Center. For ordering information, see "Obtaining Publications."

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 fall issue of 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

The *User Publications Catalog* describes the availability and content of all Cray Research hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, call +1 651 683 5907. Silicon Graphics employees may send electronic mail to `orderdsk@sgi.com` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the Order Cray Software option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

<u>Convention</u>	<u>Meaning</u>																				
<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> <p>Some internal routines (for example, the <code>_assign_asgcmd_info()</code> routine) do not have man pages associated with them.</p>	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.																				

In this manual, references to Cray PVP systems include the following machines:

- CRAY C90 series
- CRAY C90D series
- CRAY EL series (including CRAY Y-MP EL systems)
- CRAY J90 series
- CRAY Y-MP E series
- CRAY Y-MP M90 series
- CRAY T90 series

Silicon Graphics systems include all MIPS based systems running IRIX 6.2 or later.

The following operating system terms are used throughout this document.

<u>Term</u>	<u>Definition</u>
UNICOS	Operating system for all configurations of Cray PVP systems
UNICOS/mk	Operating system for all configurations of CRAY T3E systems
UNICOS MAX	Operating system for all configurations of CRAY T3D systems
IRIX	Operating system for all configurations of MIPS based systems

The default shell in the UNICOS and UNICOS/mk operating systems, referred to in Cray Research documentation as the *standard shell*, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2-1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS and UNICOS/mk operating systems also support the optional use of the C shell.

Cray UNICOS version 10.0 is an X/Open Base 95 branded product.

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 part number of the document with your comments.

You can contact us in any of the following ways:

- Send electronic mail to the following address:

`techpubs@sgi.com`

- Send a facsimile to the attention of “Technical Publications” at fax number +1 650 932 0801.
- Use the Suggestion Box form on the Technical Publications Library World Wide Web page:

`http://techpubs.sgi.com/library/`

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

For Silicon Graphics IRIX based operating systems: 1 800 800 4SGI

For UNICOS or UNICOS/mk based operating systems or CRAY Origin2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389

We value your comments and will respond to them promptly.

Overview [1]

The Cray Message Passing Toolkit and 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 1.3 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

1.1 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. Silicon Graphics supports implementations of MPI that are released as part of the Message Passing Toolkit. These implementations are available on IRIX, UNICOS, and UNICOS/mk 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>

1.2 MPI Components

On UNICOS and UNICOS/mk systems, the MPI library is provided as a statically linked set of objects (a file with a name that ends in `.a`). On IRIX systems, 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.a` library (or `libmpi.so` for IRIX systems), the include files, and the `mpirun(1)` command.

For UNICOS/mk systems, in addition to the `libmpi.a` library, the `libpmpi.a` library is provided for profiling support. For UNICOS and IRIX systems, profiling support is included in the `libmpi.a` and `libmpi.so` libraries, respectively. Profiling support replaces all `MPI_XXX` prototypes and function names with `PMPI_XXX` entry points.

1.3 MPI Program Development

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

Procedure 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, page 3.
3. Execute your program by using one of the following methods:
 - `mpirun(1)` (available on all systems (see Chapter 3, page 9))
 - Execute directly (available on UNICOS/mk systems (see Section 3.3.3, page 14))

Note: On IRIX systems, 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 Section 2.3, page 8.

Building MPI Applications [2]

This chapter provides procedures for building MPI applications on UNICOS, UNICOS/mk, and IRIX systems.

2.1 Building Applications on UNICOS Systems

On UNICOS systems, the MPI library supports either a shared memory or a Transmission Control Protocol (TCP) communications model.

The following section provides information about the shared memory model, and information about compiling and linking shared memory MPI programs.

Note: Software included in the 1.3 release of the Message Passing Toolkit is designed to be used with the Cray Programming Environment. When building an application that uses the shared memory version of MPI, you must be using the Programming Environment 3.0 release or later. Before you can access the Programming Environment, the `PrgEnv` module must be loaded. For more information on using modules, see *Installing Programming Environment Products*, or, if the Programming Environment has already been installed on your system, see the online ASCII file `/opt/ctl/doc/README`.

Section 2.1.3, page 7, provides information that applies to MPI programs that use TCP for communication.

2.1.1 Building Applications That Use Shared Memory MPI on UNICOS Systems

To build an executable file that makes use of shared memory, perform the following steps:

1. Convert all global and static data to `TASKCOMMON` data.

In a multitasking environment, all members of the multitasking group can access all global or static data because they share one user address space. An MPI process is described as a separate address space. To emulate an MPI process in a multitasking environment, all global or static data that can be modified during the course of execution of a program must be treated as data local to each task. This is done by placing the data in `TASKCOMMON` blocks. `TASKCOMMON` storage is a mechanism that is used in multitasked programs to provide a separate copy of data for each member of the multitasking group. `TASKCOMMON` data is still globally accessible across functions within a multitasked program, but it is private to each specific

task. Fortran examples of global or static data that must be placed in `TASKCOMMON` include data that resides in `COMMON` blocks and data that appears in `DATA` or `SAVE` statements. In C, you must place all data that is declared static (either locally or globally) or data declared at a global level (outside of any function) in `TASKCOMMON`.

Because changing your program so that all global and static data is private is tedious and makes a program less portable, Cray provides support in the form of compile-time command line options to do the conversions. You can convert most global and static data to `TASKCOMMON` data automatically by using the following command line options:

- For C programs:

```
cc -h taskprivate
```

- For Fortran programs:

```
f90 -a taskcommon
```

When you are placing data in `TASKCOMMON`, there may be cases in which the compiler cannot do the conversion because of insufficient information. The compiler notes these cases by issuing a warning during compilation. For such cases, you must convert the data by hand. Most of the time, these cases are related to initialization that involves Fortran `DATA` or `SAVE` statements or C initialized static variables, and you might need to change only how or when the data is initialized for it to be successfully placed in `TASKCOMMON`.

The following is an example of a case that the compiler cannot handle:

```
int a;
int b = &a
```

If variable `a` resides in `TASKCOMMON`, its address will not be known until run time; therefore, the compiling system cannot initialize it. In this case, the initialization must be handled within the user program.

2. Use the `cc(1)` or `f90(1)` commands to build your shared memory MPI program, as in the following examples:

C programs:

```
cc -htaskprivate -D_MULTIP_ -L$MPTDIR/lib/multi file.c
```

For C programs, the `-D` and `-L` options are needed to access the reentrant version of `libc`. This version is required to provide safe access to `libc` routines in a multitasking environment. When the `mpt` module is loaded, the module software sets `$MPTDIR` automatically and points to the default MPT software library. (For information on using modules, see *Installing Programming Environment Products*.) To make compiling in C easier, the environment variable `$LIBCM` is also set automatically when the `mpt` module is loaded. You can use `$LIBCM` with the `cc(1)` command to request the reentrant version of `libc`. `$LIBCM` is set to the following value:

```
-D_MULTIP_ -L$MPTDIR/lib/multi
```

The following example uses `$LIBCM`:

```
cc -htaskprivate $LIBCM file.c
```

Fortran programs:

```
f90 -ataskcommon file.f
```

3. Select private I/O if private Fortran file unit numbers are desired.

In a multitasking environment, Fortran unit numbers are, by default, shared by all members of the multitasking group. This behavior forces all files to be shared among MPI processes. The user can request that files be private to each MPI process by specifying the private I/O option on the `assign(1)` command. The examples in Table 1, page 6, request private I/O.

Table 1. assign examples

Example	Description
<code>assign -P private u:10</code>	Specifies that unit 10 should be private to any MPI process that opens it.
<code>assign -P private p:%</code>	Specifies that all named Fortran units should be private to any MPI process that opens them. This includes all units connected to regular files and excludes units such as 5 and 6, which are connected to <code>stdin</code> , <code>stdout</code> , or <code>stderr</code> by default.
<code>assign -P global u:0</code> <code>assign -P global u:5</code> <code>assign -P global u:6</code> <code>assign -P global u:100</code> <code>assign -P global u:101</code> <code>assign -P global u:102</code>	This set of <code>assign</code> commands can be used in conjunction with <code>assign -P private g:all</code> to retain units connected by default to <code>stdin</code> , <code>stdout</code> , and <code>stderr</code> as global units. A unit connected to these standard files cannot be a private unit.

For more information on private I/O functionality on Cray PVP systems, see the `assign(1)` man page.

4. Run the application. To start or run an application that uses the shared memory version of MPI, you must use the `-nt` option on the `mpirun(1)` command (for example, `mpirun -nt 4 compute`).

You should also consider using Autotasking instead of message passing whenever your application is run on a UNICOS system. The communications overhead for Autotasking is orders of magnitude less than that for sockets, even on the same system, so it might be better to have only one fully autotasked MPI process on the UNICOS system. In many cases, you might be able to achieve this simply by invoking the appropriate compiler options and sending a larger file of input data to the MPI process on the UNICOS system.

2.1.2 Shared Memory MPI Limitations

Emulating a shared memory environment with the use of Cray multitasking software might cause unexpected program behavior. The goal is to preserve the original behavior as much as possible. However, it is not efficient or productive to completely preserve the original MPI behavior in a multitasked environment.

The intent is to document possible changes in behavior. For example, changes in behavior might occur with the use of signals; therefore, it is not recommended that signals be used with the shared memory version of MPI.

The shared memory implementation of MPI supports the running of only 32 MPI processes within a multitasking group. Because MPI processes must share CPU resources, running with more than the number of physical CPUs available on the UNICOS system will begin to degrade performance.

2.1.3 Building Files on UNICOS Systems When Using TCP

On UNICOS systems, after you have added the MPI function calls described in Procedure 1, step 1, page 2, a simple MPI program can be linked as follows:

```
cc -o compute compute.o
```

This command links the `compute.o` object code to produce the `compute` executable file.

If you are using more than one host, the executable files should be installed on the Cray systems that you will be using. By default, MPI uses the path name of the user-initiated executable file on all systems. You can override this file by using a process group file.

In some installations, certain hosts are connected in multiple ways. For example, an Ethernet connection may be supplemented by a high-speed FDDI ring. Usually, alternate host names are used to identify the high-speed connection. You must put these alternate names in your machine file.

If your hosts are connected in multiple ways, you must not use `local` in your machine file to identify the local host, but must use the name of the local host instead. For example, if hosts `host1` and `host2` have Asynchronous Transfer Mode (ATM) connected to `host1-atm` and `host2-atm`, respectively, the correct machine file is as follows:

```
host1-atm  
host2-atm
```

2.2 Building Applications on UNICOS/mk Systems

On UNICOS/mk systems, after you have added MPI function calls to your program, as described in Procedure 1, step 1, page 2, you can compile and link an MPI program, as in the following examples:

```
cc -o compute compute.c
or
f90 -o compute -X4 compute.f
```

If you have loaded the `mpt` module, the directory that contains the MPI include files is automatically searched, and the MPI library is automatically loaded.

2.3 Building Applications on IRIX Systems

On IRIX systems, after you have added MPI function calls to your program, as described in Procedure 1, step 1, page 2, 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
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
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 [3]

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 IRIX or UNICOS systems. On IRIX or UNICOS systems, 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. Use of the `mpirun` command is optional for UNICOS/mk systems and currently supports only the `-np` option. 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.

3.1 Syntax of the `mpirun` Command

The format of the `mpirun` command for UNICOS and IRIX 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:

<u>Option</u>	<u>Description</u>
<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.

-f[<i>file</i>] <i>file_name</i>	Specifies a text file that contains <code>mpirun</code> arguments.
-h[<i>elp</i>]	Displays a list of options supported by the <code>mpirun</code> command.
-p[<i>refix</i>] <i>prefix_string</i>	<p>Specifies a string to prepend to each line of output from <code>stderr</code> and <code>stdout</code> for each MPI process. Some strings have special meaning and are translated as follows:</p> <ul style="list-style-type: none"> • <code>%g</code> translates into the global rank of the process producing the output. (This is equivalent to the rank of the process in <code>MPI_COMM_WORLD</code>.) • <code>%G</code> translates into the number of processes in <code>MPI_COMM_WORLD</code>. • <code>%h</code> translates into the rank of the host on which the process is running, relative to the <code>mpirun(1)</code> command line. • <code>%H</code> translates into the total number of hosts in the job. • <code>%l</code> translates into the rank of the process relative to other processes running on the same host. • <code>%L</code> translates into the total number of processes running on the host. • <code>%@</code> 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:

<u>Option</u>	<u>Description</u>
<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 Section 3.2.

<code>-np <i>np</i></code>	Specifies the number of processes on which to run. (UNICOS/mk systems support only this option.)
<code>-nt <i>nt</i></code>	On UNICOS systems, specifies the number of tasks on which to run in a multitasking or shared memory environment. On IRIX systems, this option behaves the same as <code>-np</code> .

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

3.2 Using a File for `mpirun` Arguments (UNICOS or IRIX)

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.

3.3 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. On a UNICOS system, the following example uses a combination of shared memory and TCP. On an IRIX system, both executable files use shared memory:

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

3.3.1 Using `mpirun(1)` to Run Programs in Shared Memory Mode (UNICOS or IRIX)

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, and can be used on UNICOS systems only if you have compiled and linked your program as described in Section 2.1.1, page 3. 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.

The `-nt` option to `mpirun` is supported on IRIX systems for consistency across platforms. However, since the default mode of execution on a single IRIX system is to use shared memory, the 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
```

3.3.2 Using the `mpirun(1)` Command on UNICOS/mk Systems

The `mpirun(1)` command has been provided for consistency of use among IRIX, UNICOS, and UNICOS/mk systems. Use of this command is optional, however, on UNICOS/mk systems. If your program was built for a specific number of PEs, the number of PEs specified on the `mpirun(1)` command line must match the number that was built into the program. If it does not, `mpirun(1)` issues an error message.

The following example shows how to invoke the `mpirun(1)` command on a program that was built for four PEs:

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

3.3.3 Executing UNICOS/mk Programs Directly

Instead of using the `mpirun(1)` command, you can choose to launch your MPI programs on UNICOS/mk systems directly. If your UNICOS/mk program was built for a specific number of PEs, you can execute it directly, as follows:

```
./a.out
```

If your program was built as a *malleable* executable file (the number of PEs was not fixed at build time, and the `-Xm` option was used instead), you can execute it with the `mpprun(1)` command. The following example runs a program on a partition with four PEs:

```
mpprun -n 4 a.out
```

3.4 Launching a Distributed Program (UNICOS or IRIX)

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, CRAY J90, or CRAY T3E system, or can be any accessible computer running Array Services software. Array Services software runs on IRIX and UNICOS 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, the `-np` or `-nt` option can be omitted with the number of processes listed directly. On UNICOS systems, if you omit the `-np` or `-nt` option, `mpirun` assumes `-np` and defaults to TCP for communication. 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 [4]

The Silicon Graphics 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. MPI is not thread-safe on UNICOS or UNICOS/mk systems.

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.

4.1 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 Silicon Graphics 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()`.

4.2 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()`.

4.3 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.

4.4 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.

4.5 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.

4.6 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.

4.7 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.

4.8 Internal Statistics

The Silicon Graphics internal statistics diagnostics are not thread-safe.

4.9 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 [5]

On IRIX systems, 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 [6]

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.

6.1 Setting MPI Environment Variables on UNICOS and IRIX Systems

This section provides a table of MPI environment variables you can set for IRIX systems only, and a table of environment variables you can set for both UNICOS and IRIX systems. For environment variables for UNICOS/mk systems, see Section 6.2, page 27.

Table 2. MPI environment variables for IRIX systems only

Variable	Description	Default
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_BYPASS_DEVS	Sets the order for opening HIPPI adapters. The list of devices does not need to be space-delimited (0123 is also valid). An array node usually has at least one HIPPI adapter, the interface to the HIPPI network. The HIPPI bypass is a lower software layer that interfaces directly to this adapter. The bypass sends MPI control and data messages that are 16 Kbytes or shorter.	0 1 2 3

Variable	Description	Default
	<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. 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.</p> <p>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.</p>	
<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_DSM_OFF</code>	Turns off nonuniform memory access (NUMA) optimization in the MPI library.	Not enabled
<code>MPI_DSM_MUSTRUN</code>	Specifies the CPUs on which processes are to run. For jobs running on IRIX systems, 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
<code>MPI_DSM_PPM</code>	Sets the number of MPI processes that can be run on each node of an IRIX system.	2


Variable	Description	Default
MPI_DSM_VERBOSE	Instructs <code>mpirun</code> to print information about process placement for jobs running on NUMA systems.	Not enabled
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 <code>MPI_MSGS_PER_HOST</code> 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.	

Table 3. MPI environment variables for UNICOS and IRIX systems

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_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_CHECK_ARGS	Enables checking of MPI function arguments. Segmentation faults might occur if bad arguments are passed to MPI, so this is useful for debugging purposes. Using argument checking adds several microseconds to latency.	Not enabled
MPI_COMM_MAX	Sets the maximum number of communicators that can be used in an MPI program. Use this variable to increase internal default limits. (May be required by standard-compliant programs.)	256

Variable	Description	Default
MPI_DIR	Sets the working directory on a host. When an <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_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_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

Variable	Description	Default
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

6.2 Setting MPI Environment Variables on UNICOS/mk Systems

This section provides a table of MPI environment variables you can set for UNICOS/mk systems.

Table 4. Environment variables for UNICOS/mk systems

Variable	Description	Default
MPI_SM_POOL	Specifies shared memory queue. When MPI is started, it allocates a pool of shared memory for use in message passing. This pool represents space used to buffer message headers and small messages while the receiving PE is doing computations or I/O. The default of 1024 bytes is the number of bytes that can be pending.	1024 bytes
MPI_SM_TRANSFER	Specifies number of queue slots. Specifies the number of slots in the shared memory queue that can be occupied by a send operation at the receiver. A slot consists of four UNICOS/mk words. By default, a single send operation can occupy 128 slots (or buffer 512 words) while the receiving PE is doing computations or I/O.	128

Variable	Description	Default
MPI_BUFFER_MAX	Specifies maximum buffer size. Specifies a maximum message size, in bytes, that will be buffered for MPI standard, buffered, or ready send communication modes.	No limit
MPI_BUFFER_TOTAL	Specifies total buffer memory. Specifies a limit to the amount of memory the MPI implementation can use to buffer messages for MPI standard, buffered, or ready send communication modes.	No limit

6.3 Internal Message Buffering in MPI (IRIX Systems Only)

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 5, page 29, 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 Silicon Graphics 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 5. Outline of improper dependence on buffering

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

Launching Programs with NQE [7]

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 on IRIX systems to submit an MPI program for execution. For information on using NQE to submit UNICOS or UNICOS/mk programs, see the *NQE User's Guide*.

7.1 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 1 shows the NQE button bar, which appears after your entry.

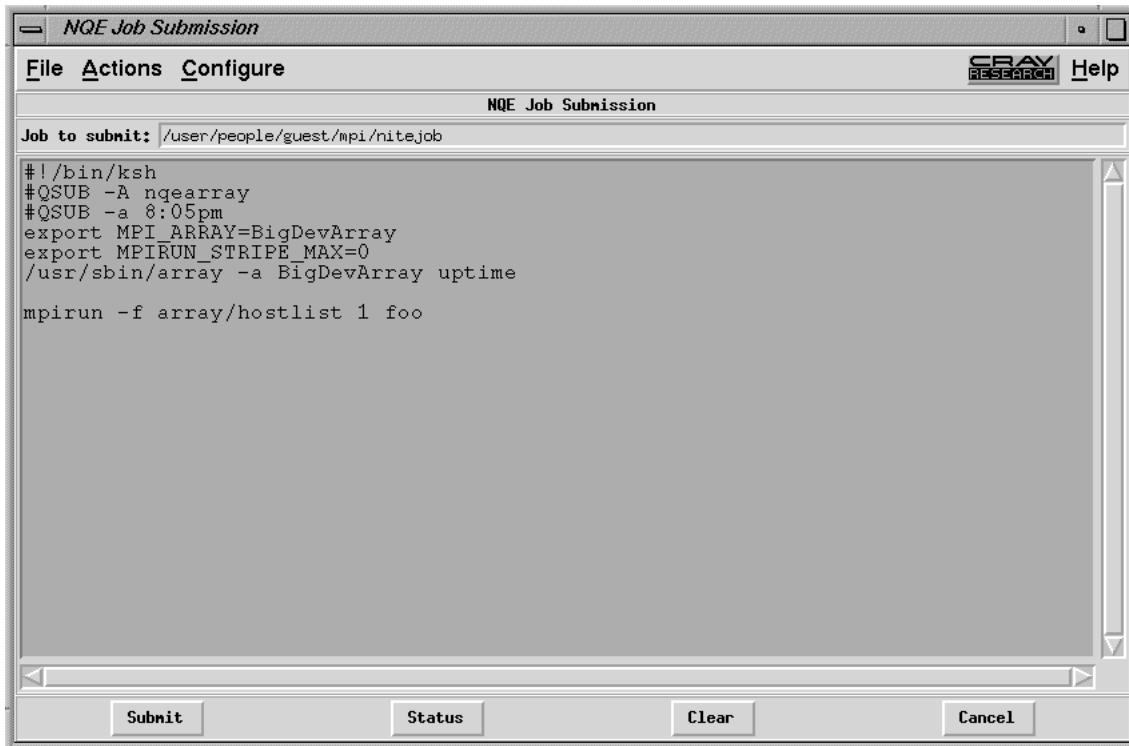


a11378

Figure 1. NQE button bar

7.2 Submitting a Job with NQE

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



a11379

Figure 2. NQE 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.

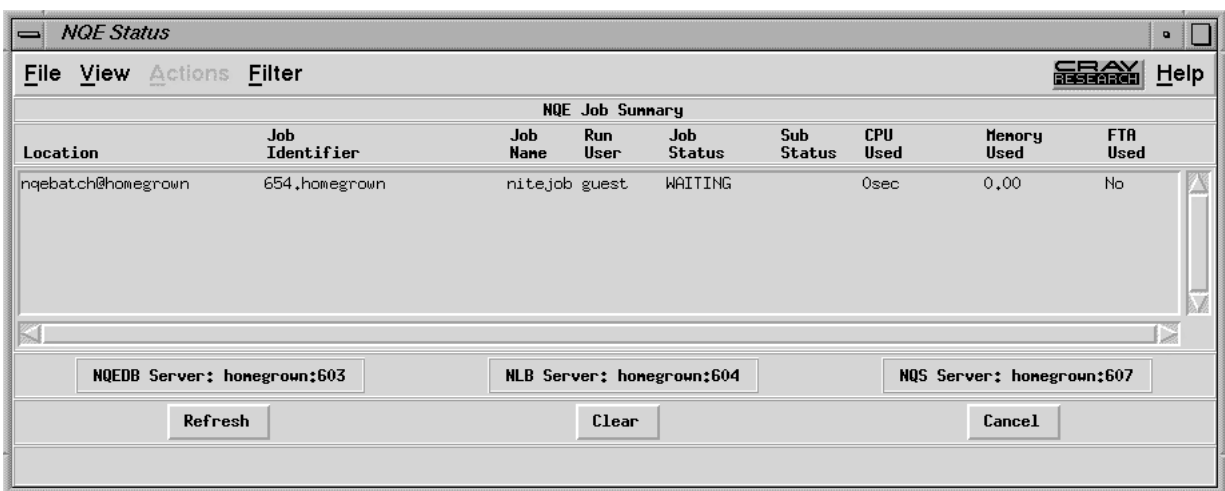
Also notice in Figure 2 that the MPI program is already compiled and distributed to the proper hosts. The file array/hostlist has the list of parameters for this job, as you can see in the output from the following cat command:

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


7.3 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 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 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 4. Notice that the job was created at 8:26 PDT and is to run at 20:05 PDT.



a11381

Figure 4. NQE Detailed Job Status window

7.4 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*

The preceding publications are also available in the Cray Online Software Publications Library at the following URL:

<http://www.cray.com/products/software/publications>

PostScript files of NQE publications are available through the Cray Online Publications Software Library. To download a publication, select Summary next to the book title you want on the Titles Web page, which is located at the following URL:

<http://www.cray.com/products/software/publications/dynaweb/docs/titles.html#N>

For general information about NQE, see the following URL:

<http://www.cray.com>

(search for NQE)

MPI Troubleshooting [8]

This chapter provides answers to frequently asked questions about MPI.

8.1 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.

8.1.1 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.

8.1.2 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.

8.1.3 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.

8.2 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
```

8.2.1 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
```

8.2.2 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.

8.2.3 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.

8.2.4 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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.

A

- adapter selection, 21
- application running, 6
- assign command, 6
- autotasking, 6

B

- building files
 - shared memory MPI, 3
 - TCP devices, 7
- building MPI applications
 - for shared memory, 3
 - on IRIX systems, 8
 - on UNICOS systems, 3
 - on UNICOS/mk systems, 8
 - using TCP, 7

D

- distributed programs, 15

E

- environment variable setting
 - UNICOS and IRIX systems, 23
 - UNICOS/mk systems, 27
- environment variables
 - buffering for UNICOS/mk, 28

F

- frequently asked questions, 37

H

- high-speed connections, 7

I

- internal message buffering, 28

M

- MPI
 - components, 2
 - overview, 1
- mpirun
 - argument file, 13
 - command, 9
 - for distributed programs, 15
 - for local host, 13
 - for shared memory, 14
 - on UNICOS/mk systems, 14
- MPT
 - components, 1
 - overview, 1
- multiboard feature, 21

N

- Network Queuing Environment (NQE), 31

P

- private files, 6
- program development, 2
- program segments, 15

S

- shared memory
 - file building, 3
 - limitations, 6
 - using mpirun, 14
- sprocs, 17

T

- TASKCOMMON storage, 3
- TCP file building, 7
- threads
 - collectives, 18
 - exception handlers, 19

- finalization, 19
- initialization, 17
- internal statistics, 19
- probes, 18
- query functions, 18
- requests, 18
- signals, 19
- thread-safe systems, 17
- troubleshooting, 37

U

- UNICOS/mk direct execution, 14

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3687-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
 - On the Internet: techpubs@sgi.com
 - For UUCP mail (through any backbone site): *[your_site]!sgi!techpubs*
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389