Sun™ MPI 5.0 Programming and Reference Manual

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 U.S.A.
650-960-1300

Part No. 816-0651-10
August 2001, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript™

Contents

Preface

The *Sun MPI Programming and Reference Manual* describes the Sun™ MPI library of message-passing routines and explains how to develop an MPI program on a Sun HPC system.

For the most part, this guide does not repeat information that is available in detail in the MPI Standard; it focuses instead on what is specific to the Sun MPI implementation.

You should be familiar with programming in C or Fortran, with parallel programming, and with the message-passing model.

Before You Read This Book

For general information about writing MPI programs, see Related Documentation on page ix. Sun MPI is part of the Sun HPC ClusterTools ™suite of software. For more information about running Sun MPI jobs, see the *Sun HPC ClusterTools 4 User's Guide*. Product notes for Sun MPI are included in *Sun HPC ClusterTools 4 Product Notes*.

Using UNIX Commands

This document does not describe how to use basic UNIX® commands. For that type of information, see:

- AnswerBook2™ online documentation for the Solaris™ software environment
- Other software documentation that you received with your system

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail`.

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename.*

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	`%`
C shell superuser	#
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	#

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
Sun HPC	*Read Me First: Guide to Sun HPC ClusterTools Documentation*	816-0646-10
Sun HPC ClusterTools software	*Sun HPC ClusterTools 4 Product Notes*	816-0647-10
	Sun HPC ClusterTools 4 Installation Guide	816-0648-10
	Sun HPC ClusterTools 4 User's Guide	816-0650-10
	Sun HPC ClusterTools 4 Performance Guide	816-0656-10
	Sun HPC ClusterTools 4 Administrator's Guide	816-0649-10
Sun S3L	*Sun S3L Programming Guide*	816-0652-10
	Sun S3L Reference Manual	816-0653-10
Prism™ development environment	*Prism User's Guide*	816-0654-10
	Prism Reference Manual	816-0655-10

There is a wealth of documentation on MPI available on the World Wide Web:

- The MPI home page, with links to specifications for the MPI-2 Standard:

  `http://www.mpi–forum.org`

- Additional Web sites that provide links to papers, talks, the MPI Standard, implementations, information about MPI-2, tutorials, plus pointers to many other sources:

  `http://www.erc.msstate.edu/mpi/`

  `http://www.arc.unm.edu/`

Accessing Sun Documentation Online

A broad selection of Sun system documentation is located at:

`http://www.sun.com/products-n-solutions/hardware/docs`

A complete set of Solaris documentation and many other titles are located at:

`http://docs.sun.com`

Ordering Sun Documentation

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at:

`http://www.fatbrain.com/documentation/sun`

Man Pages

Man pages are also available online for all the Sun MPI and MPI I/O routines and are accessible via the Solaris `man` command. These man pages are usually installed in `/opt/SUNWhpc/man`. Ask your system administrator for their location at your site.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number (806-0651-10) of your document in the subject line of your email.

CHAPTER 1

Introduction to Sun MPI

The MPI specification was developed by the MPI Forum, a group of software developers, computer vendors, academics, and computer-science researchers whose goal was to develop a standard for writing message-passing programs that would be efficient, flexible, and portable.

The outcome, known as the MPI Standard, was first published in 1993. The most recent version (MPI-2) was published in July 1997. It was well received, and there are several implementations available publicly.

Sun MPI is a library of message-passing routines, including all MPI 1.1–compliant routines and a subset of the MPI 2–compliant routines. Chapter 2 provides an overview of the routines, Appendix A summarizes them, Sun `man` pages provide detailed descriptions.

Sun MPI Features

- Optimized to run with Sun HPC ClusterTools 4 software using C, C^{++}, Fortran 77, or Fortran 90.
- Integration with the Sun™ ClusterTools Runtime Environment (CRE)
- Integration with Platform Computing's Load Sharing Facility (LSF) Suite
- Support for multithreaded programming
- Seamless use of different network protocols
- Multiprotocol support such that MPI picks the fastest available medium for each type of connection (such as shared memory and ATM)
- Communication by shared memory for fast performance on clusters of SMPs
- Finely tunable shared memory communication
- Optimized collectives for symmetric multiprocessors (SMPs) and clusters of SMPs
- MPI I/O support for parallel file I/O
- Prism support – Users can develop, run, and debug programs in the Prism programming environment.
- Implicit co-scheduling – The Sun HPC `spind` daemon enables certain processes of a given MPI job on a shared-memory system to be scheduled at approximately the same time as other related processes. This co-scheduling reduces the load on the processors, thus reducing the effect that MPI jobs have on each other.
- Limited support of one-sided communication routines
- Dynamic library
- MPI-2 dynamic support

Sun MPI and MPI I/O provide full F77, C, and C^{++} support, as well as Basic F90 support.

MPI I/O

File I/O in Sun MPI uses MPI 2–compliant routines for parallel file I/O. Chapter 4 describes these routines. Their `man` pages are provided online, and the routines are summarized in Appendix A.

CHAPTER **2**

Sun MPI Library

This chapter describes the Sun MPI library:

Note – Sun MPI I/O is described separately, in Chapter 4.

Types of Libraries

Sun MPI contains four types of libraries:

- *32- and 64-bit libraries* —If you want to take advantage of the 64-bit capabilities of Sun MPI, you must explicitly link to the 64-bit libraries. The 32-bit libraries are the default in each category.
- *Thread-safe and non-thread-safe libraries* —For multithreaded programs, link with the thread-safe library in the appropriate category unless the program has only one thread calling MPI. For programs that are not multithreaded, you can link against either the thread-safe or the default (non-thread-safe) library. However, non-multithreaded programs will have better performance using the default library, as it does not incur the extra overhead of providing thread-safety. Therefore, use the default libraries whenever possible for maximum performance.

The 32-bit libraries are the default. For full information about linking to libraries, see "Compiling and Linking" on page 26.

Sun MPI Routines

This section gives a brief description of the routines in the Sun MPI library. All the Sun MPI routines are listed in Appendix A with brief descriptions and their C syntax. For detailed descriptions of individual routines, see the man pages or the MPI Standard.

Point-to-Point Communication Routines

Point-to-point communication routines include the basic send and receive routines in both blocking and nonblocking forms and in four modes.

A *blocking send* blocks until its message buffer can be written with a new message. A *blocking receive* blocks until the received message is in the receive buffer.

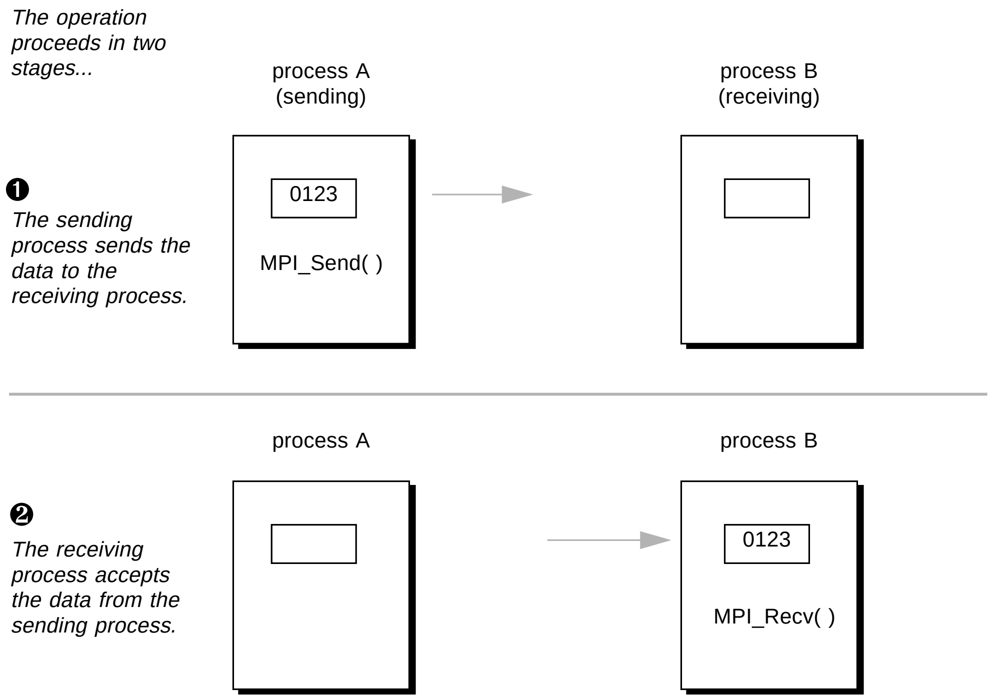
Nonblocking sends and receives differ from blocking sends and receives in that they return immediately and their completion must be waited or tested for. It is expected that eventually nonblocking send and receive calls will allow the overlap of communication and computation.

The four modes for MPI point-to-point communication are:

- *Standard* — the completion of a send implies that the message either is buffered internally or has been received. Users are free to overwrite the buffer that they passed in with any of the blocking send or receive routines, after the routine returns.
- *Buffered* — the user guarantees a certain amount of buffering space.
- *Synchronous* —rendezvous semantics occur between sender and receiver; that is, a send blocks until the corresponding receive has occurred.
- *Ready* — a send can be started only if the matching receive is already posted. The ready mode for sends is a way for the programmer to notify the system that the receive has been posted, so that the underlying system can use a faster protocol if it is available.

One-Sided Communication Routines

Standard MPI communication is two-sided. To complete a transfer of information, both the sending and receiving processes must call appropriate functions.


The operation proceeds in two stages...
process A (sending)
process B (receiving)
❶
The sending process sends the data to the receiving process.
0123
MPI_Send( )
process A
process B
❷
The receiving process accepts the data from the sending process.
0123
MPI_Recv( )

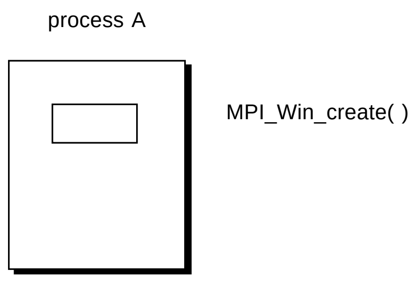
This form of communication requires regular synchronization between the sending and receiving processes. That synchronization can become complicated if the receiving process does not know which process is sending it the data it needs.

One-sided communication was developed to solve this problem and to reduce the amount of synchronization required even when both sending and receiving processes know each other's identities.

In one-sided communication, a process opens a window in memory and exposes it to all processes that belong to a particular communicator, provided they reside on the same node. As long as that window is open, any process in the communicator and node can *put* data into it and *get* data out of it.

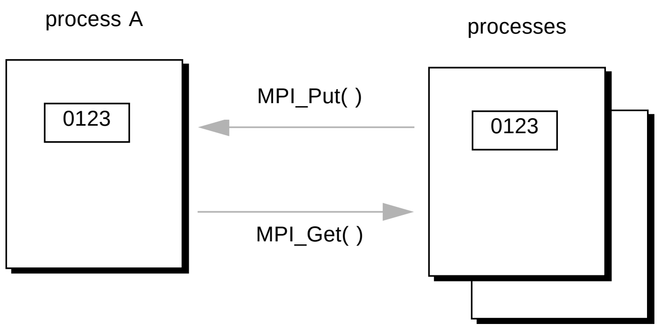
❶

A process opens a communications window and exposes it to the other processes in the same node and communicator.



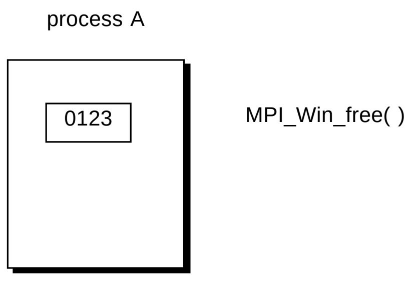
❷

Any process within the same communicator and node can transfer data directly into or out of that window...



❸

... until the original process closes the window.



The *put* requires no complementary operation from the process that opened the window, and is equivalent to the combination of a *send* and *receive* operation in two-sided MPI communication.

The functions used to implement one-sided MPI communication fall into three categories. They are summarized in Table 2-1. You can find their definitions in the MPI Standard. Also, Appendix A of this document provides syntax summaries.

TABLE 2-1 One-Sided Communication Routines

Window Creation	
`MPI_Win_create()`	Creates a window in memory and exposes it to all processes in the communicator and node.
`MPI_Win_free()`	Closes the window created with `MPI_Win_create`. Requires barrier synchronization.
`MPI_Win_get_group()`	Returns a duplicate of the group of the communicator used to create the window.
Data Transfer	
`MPI_Accumulate()`	Combines data from a process with data already in the window. Different from `MPI_Put` in that new data is appended to existing data instead of replacing it.
`MPI_Get()`	The calling process takes data directly from the window. The opposite of `MPI_Put`. Equivalent to the combination of a Send and Receive operation originated by the receiving process.
`MPI_Put()`	The calling process loads its data directly into the buffer of the target process. Equivalent to the combination of a Send and Receive operation originated by the sending process. The opposite of `MPI_Get`.
Synchronization	
`MPI_Win_fence()`	Blocks any process from operating on a particular window until all operations relating to that window have completed. Similar to `MPI_Barrier`, but applies to a window instead of a communicator.
`MPI_Win_lock()`	Starts an RMA access epoch. While the lock is in place, only the process whose rank is specified in the function call can be accessed by RMA operations on the window.
`MPI_Win_unlock()`	Closes an RMA access epoch begun with a call to `MPI_Win_lock()`.

Limitations

Sun’s implementation of one-sided communication has four limitations:

- Single Node

  The communication window can only be accessed by processes within the same node.

- Assertions

  Assertions are not supported. If you already have assertions in your one-sided code, Sun MPI will simply ignore them.

- `MPI_ALLOC_MEM`

  The size of the communication window cannot be larger than the value set by the `MPI_ALLOC_MEM` function.

- Unsupported functions

  These synchronization functions are not supported:

  - `MPI_Win_start()`
  - `MPI_Win_complete()`
  - `MPI_Win_post()`
  - `MPI_Win_wait()`
  - `MPI_Win_test()`

Collective Communication

Collective communication routines are blocking routines that involve all processes in a communicator. Collective communication includes broadcasts and scatters, reductions and gathers, all-gathers and all-to-alls, scans, and a synchronizing barrier call.

TABLE 2-2 Collective Communication Routines

`MPI_Bcast()`	Broadcasts from one process to all others in a communicator.
`MPI_Scatter()`	Scatters from one process to all others in a communicator.
`MPI_Reduce()`	Reduces from all to one in a communicator.
`MPI_Allreduce()`	Reduces, then broadcasts result to all nodes in a communicator.
`MPI_Reduce_scatter()`	Scatters a vector that contains results across the nodes in a communicator.
`MPI_Gather()`	Gathers from all to one in a communicator.

TABLE 2-2 Collective Communication Routines *(Continued)*

`MPI_Gatherv()`	Gathers information from all processes in a communicator
`MPI_Allgather()`	Gathers, then broadcasts the results of the gather in a communicator.
`MPI_Alltoall()`	Performs a set of gathers in which each process receives a specific result in a communicator.
`MPI_Scan()`	Scans (parallel prefix) across processes in a communicator.
`MPI_Barrier()`	Synchronizes processes in a communicator (no data is transmitted).

Many of the collective communication calls have alternative vector forms, with which different amounts of data can be sent to or received from different processes.

The syntax and semantics of these routines are basically consistent with the point-to-point routines (upon which they are built), but there are restrictions to keep them from getting too complicated:

- The amount of data sent must exactly match the amount of data specified by the receiver.
- There is only one mode, a mode analogous to the standard mode of point-to-point routines.

Managing Communicators, Groups, and Contexts

A distinguishing feature of the MPI Standard is that it includes a mechanism for creating separate worlds of communication, accomplished through *communicators*, *groups*, and *contexts*.

A *communicator* specifies a group of processes that will conduct communication operations within a specified context without affecting or being affected by operations occurring in other groups or contexts elsewhere in the program. A communicator also guarantees that, within any group and context, point-to-point and collective communication are isolated from each other.

A *group* is an ordered collection of processes. Each process has a rank in the group; the rank runs from 0 to *n*–1. A process can belong to more than one group; its rank in one group has nothing to do with its rank in any other group.

A *context* is the internal mechanism by which a communicator guarantees safe communication space to the group.

At program startup, two default communicators are defined:

- `MPI_COMM_WORLD`, which has as a process group all the processes of the job
- and `MPI_COMM_SELF`, which is equivalent to an identity communicator.

The process group that corresponds to MPI_COMM_WORLD is not predefined, but can be accessed using MPI_COMM_GROUP. One MPI_COMM_SELF communicator is defined for each process, each of which has rank zero in its own communicator. For many programs, these are the only communicators needed.

Communicators are of two kinds: *intracommunicators*, which conduct operations within a given group of processes; and *intercommunicators*, which conduct operations between two groups of processes.

Communicators provide a *caching* mechanism, which allows an application to attach attributes to communicators. Attributes can be user data or any other kind of information.

New groups and new communicators are constructed from existing ones. Group constructor routines are local, and their execution does not require interprocessor communication. Communicator constructor routines are collective, and their execution may require interprocess communication.

Note – Users who do not need any communicator other than the default MPI_COMM_WORLD communicator — that is, who do not need any sub- or supersets of processes — can simply plug in MPI_COMM_WORLD wherever a communicator argument is requested. In these circumstances, users can ignore this section and the associated routines. (These routines can be identified from the listing in Appendix A.)

Data Types

All Sun MPI communication routines have a data type argument. They may be primitive data types, such as integers or floating-point numbers, or they may be user-defined, derived data types that are specified in terms of primitive types.

Derived data types enable users to specify more general, mixed, and noncontiguous communication buffers, such as array sections and structures that contain combinations of primitive data types.

Fortran data types are listed in Table 2-3. Data types of Fortran used with the `-r8` flag are listed in TABLE 2-4 on page 12. C datatypes are listed in TABLE 2-5 on page 13.

TABLE 2-3 Fortran Data types

MPI Data Type	**Fortran Data Type**
`MPI_INTEGER`	`INTEGER` `INTEGER*4`
`MPI_INTEGER1`	`INTEGER*1 (Fortran 90 only)`
`MPI_INTEGER2`	`INTEGER*2`
`MPI_INTEGER4`	`INTEGER*4`
`MPI_INTEGER8`	`INTEGER*8`
`MPI_REAL`	`REAL` `REAL*4`
`MPI_REAL4`	`REAL*4`
`MPI_REAL8`	`REAL*8`
`MPI_REAL16`	`REAL*16`

`MPI_DOUBLE_PRECISION`	`REAL*8`
	`DOUBLE PRECISION`

`MPI_2DOUBLE_PRECISION`	`pair of DOUBLE PRECISION VARIABLES`[1]
`MPI_COMPLEX`	`COMPLEX`
`MPI_LOGICAL`	`LOGICAL`
`MPI_CHARACTER`	`CHARACTER(1)`
`MPI_DOUBLE_COMPLEX`	`DOUBLE COMPLEX`
`MPI_2REAL`	`pair of REALs`[1]
`MPI_INTEGER2`	`INTEGER*2`
`MPI_INTEGER4`	`INTEGER*4`
`MPI_2INTEGER`	`pair of INTEGERs`[1]
`MPI_BYTE`	no corresponding Fortran data type
`MPI_PACKED`	no corresponding Fortran data type

1. For use with MINLOC and MAXLOC

TABLE 2-4 Fortran -r8 Data types

MPI Data Type	Fortran -r8 Data Type
MPI_INTEGER	INTEGER*4
MPI_INTEGER1	INTEGER*1 (Fortran 90 only)
MPI_INTEGER2	INTEGER*2
MPI_INTEGER4	INTEGER*4
MPI_INTEGER8	INTEGER INTEGER*8
MPI_REAL	REAL*4
MPI_REAL4	REAL*4
MPI_REAL8	REAL REAL*8
MPI_REAL16	REAL*16 DOUBLE PRECISION
MPI_DOUBLE_PRECISION	REAL REAL*8
MPI_2DOUBLE_PRECISION	pair of REAL*8[1]
MPI_COMPLEX	COMPLEX*4
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_DOUBLE_COMPLEX	COMPLEX
MPI_2REAL	pair of REAL*4[1]
MPI_INTEGER2	INTEGER*2
MPI_INTEGER4	INTEGER*4
MPI_2INTEGER	pair of INTEGER*4
MPI_BYTE	no corresponding Fortran datatype
MPI_PACKED	no corresponding Fortran datatype

1. For use with MINLOC and MAXLOC

TABLE 2-5 C Datatypes

MPI Data Type	C Data Type
MPI_BYTE	no corresponding C datatype
MPI_PACKED	no corresponding C datatype
MPI_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long int
MPI_UNSIGNED_LONG	unsigned long int
MPI_LONG_LONG_INT	long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_2INT	pair of int[1]
MPI_FLOAT_INT	float and int[1]
MPI_DOUBLE_INT	double and int[1]
MPI_LONG_DOUBLE_INT	long double and int[1]
MPI_LONG_INT	long and int[1]
MPI_SHORT_INT	short and int[1]

1.For use with MINLOC and MAXLOC

Persistent Communication Requests

Sometimes within an inner loop of a parallel computation, a communication with the same argument list is executed repeatedly. The communication can be slightly improved by using a *persistent* communication request, which reduces the overhead for communication between the process and the communication controller. A persistent request can be thought of as a communication port or “half-channel.”

Managing Process Topologies

Process topologies are associated with communicators; they are optional attributes that can be given to an intracommunicator (not to an intercommunicator).

Recall that processes in a group are ranked from 0 to *n*–1. This linear ranking often reflects nothing of the logical communication pattern of the processes, which may be, for instance, a 2- or 3-dimensional grid. The logical communication pattern is referred to as a *virtual topology* (separate and distinct from any hardware topology). In MPI, there are two types of virtual topologies that can be created: Cartesian (grid) topology and graph topology.

You can use virtual topologies in your programs by taking physical processor organization into account to provide a ranking of processors that optimizes communication.

Environmental Inquiry Functions

Environmental inquiry functions include routines for starting up and shutting down error-handling routines and timers.

Few MPI routines may be called before `MPI_Init()` or after `MPI_Finalize()`. Examples include `MPI_Initialized()` and `MPI_Version()`. `MPI_Finalize()` may be called only if there are no outstanding communications involving that process.

The set of errors handled by MPI is dependent upon the implementation. See Appendix C for tables listing the Sun MPI error classes.

Programming With Sun MPI

Although there are about 190 non-I/O routines in the Sun MPI library, you can write programs for a wide range of problems using only six routines:

TABLE 2-6 Six Basic MPI Routines

`MPI_Init()`	Initializes the MPI library.
`MPI_Finalize()`	Finalizes the MPI library. This includes releasing resources used by the library.
`MPI_Comm_size()`	Determines the number of processes in a specified communicator.
`MPI_Comm_rank()`	Determines the rank of calling process within a communicator.
`MPI_Send()`	Sends a message.
`MPI_Recv()`	Receives a message.

This set of six routines includes the basic send and receive routines. Programs that depend heavily on collective communication may also include `MPI_Bcast()` and `MPI_Reduce()`.

The functionality of these routines means you can have the benefit of parallel operations without having to learn the whole library at once. As you become more familiar with programming for message passing, you can start learning the more complex and esoteric routines and add them to your programs as needed.

See Appendix A for a complete list of Sun MPI routines.

Fortran Support

Sun MPI provides basic Fortran support, as described in Section 10.2 of the MPI-2 standard. Essentially, Fortran bindings and an `mpif.h` file are provided, as specified in the MPI-1 standard. The `mpif.h` file is valid for both fixed- and free-source forms, as specified in the MPI-2 standard.

The MPI interface is known to violate the Fortran standard in several ways, but it causes few problems for FORTRAN 77 programs. Violations of the standard can cause more significant problems for Fortran 90 programs, however, if you do not follow the guidelines recommended in the standard. If you are programming in Fortran, and particularly if you are using Fortran 90, you should consult Section 10.2 of the MPI-2 standard for detailed information about basic Fortran support in an MPI implementation.

Recommendations for All-to-All and All-to-One Communication

The Sun MPI library uses the TCP protocol to communicate over a variety of networks. MPI depends on TCP to ensure reliable, correct data flow. TCP's reliability compensates for unreliability in the underlying network, as the TCP retransmission algorithms handles any segments that are lost or corrupted. In most cases, this works well with good performance characteristics. However, when doing all-to-all and all-to-one communication over certain networks, a large number of TCP segments may be lost, resulting in poor performance.

You can compensate for this diminished performance over TCP in these ways:

- When writing your own algorithms, avoid flooding one node with a lot of data.
- If you need to do all-to-all or all-to-one communication, use one of the Sun MPI routines to do so. They are implemented in a way that avoids congesting a single node with lots of data. The following routines fall into this category:
  - `MPI_Alltoall()` and `MPI_Alltoallv()` – These have been implemented using a pairwise communication pattern, so that every rank is communicating with only one other rank at a given time.
  - `MPI_Gather()`and `MPI_Gatherv()` – The root process sends ready-to-send packets to each nonroot-rank process to tell the processes to send their data. In this way, the root process can regulate how much data it is receiving at any one time. Using this ready-to-send method is associated with a minor performance cost, however. For this reason, you can override this method by setting the `MPI_TCPSAFEGATHER` environment variable to 0. (See Appendix B for information about environment variables.)

Signals and MPI

When running the MPI library over TCP, nonfatal `SIGPIPE` signals may be generated. To handle them, the library sets the signal handler for `SIGPIPE` to `ignore`, overriding the default setting (terminate the process). In this way, the MPI library can recover in certain situations. You should therefore avoid changing the `SIGPIPE` signal handler.

The Sun MPI Fortran and C^{++} bindings are implemented as wrappers on top of the C bindings. The profiling interface is implemented using weak symbols. This means a profiling library need contain only a profiled version of C bindings.

The `SIGPIPE`s may occur when a process first starts communicating over TCP. This happens because the MPI library creates connections over TCP only when processes actually communicate with one another. There are some unavoidable conditions where `SIGPIPE`s may be generated when two processes establish a connection. If you want to avoid any SIGPIPEs, set the environment variable MPI_FULLCONNINIT, which creates all connections during MPI_Init() and avoids any situations that may generate a SIGPIPE. For more information about environment variables, see Appendix B.

Multithreaded Programming

When you are linked to one of the thread-safe libraries, Sun MPI calls are thread safe, in accordance with basic tenets of thread safety for MPI mentioned in the MPI-2 specification. As a result:

- When two concurrently running threads make MPI calls, the outcome is as if the calls executed in some order.
- Blocking MPI calls block the calling thread only. A blocked calling thread does not prevent progress of other runnable threads on the same process, nor does it prevent them from executing MPI calls. Thus, multiple sends and receives are concurrent.

Guidelines for Thread-Safe Programming

Each thread within an MPI process may issue MPI calls; however, threads are not separately addressable. That is, the rank of a send or receive call identifies a process, not a thread, which means that no order is defined for the case where two threads call MPI_Recv() with the same tag and communicator. Such threads are said to be *in conflict*.

If threads within the same application post conflicting communication calls, data races will result. You can prevent such data races by using distinct communicators or tags for each thread.

In general, adhere to these guidelines:

- You must not have a request serviced by more than one thread. Although you may have an operation posted in one thread and then completed in another, you may not have the operation completed in more than one thread.
- A data type or communicator must not be freed by one thread while it is in use by another thread.
- Once MPI_Finalize() is called, subsequent calls in any thread will fail.
- You must ensure that a sufficient number of lightweight processes (LWPs) are available for your multithreaded program. Failure to do so may degrade performance or even result in deadlock.

- You cannot stub the thread calls in your multithreaded program by omitting the threads libraries in the link line. The `libmpi.so` library automatically calls in the threads libraries, which effectively override any stubs.

The following sections describe more specific guidelines that apply for some routines. They also include some general considerations for collective calls and communicator operations that you should be aware of.

`MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Waitsome()`

In a program where two or more threads call one of these routines, you must ensure that they are not waiting for the same request. Similarly, the same request cannot appear in the array of requests of multiple concurrent wait calls.

`MPI_Cancel()`

One thread must not cancel a request while that request is being serviced by another thread.

`MPI_Probe()`, `MPI_Iprobe()`

A call to `MPI_Probe()` or `MPI_Iprobe()` from one thread on a given communicator should not have a source rank and tags that match those of any other probes or receives on the same communicator. Otherwise, correct matching of message to probe call may not occur.

Collective Calls

Collective calls are matched on a communicator according to the order in which the calls are issued at each processor. All the processes on a given communicator must make the same collective call. You can avoid the effects of this restriction on the threads on a given processor by using a different communicator for each thread.

No process that belongs to the communicator may omit making a particular collective call; that is, none should be left “dangling.”

Communicator Operations

Each of the communicator functions operates simultaneously with each of the noncommunicator functions, regardless of what the parameters are and whether the functions are on the same or different communicators. However, if you are using multiple instances of the same communicator function on the same communicator where all parameters are the same, it cannot be determined which threads belong to which resultant communicator. Therefore, when concurrent threads issue such calls, you must ensure that the calls are synchronized in such a way that threads in different processes participating in the same communicator operation are grouped together. Do this either by using a different base communicator for each call or by making the calls in single-thread mode before actually using them within the separate threads.

Note also these special situations:

- If you are using multiple instances of the same function with differing parameters and multiple threads, you must use different communicators.
- When using splits with multiple instances of the same function with the same parameters, but with different threads at the split, you must use different communicators.

For example, suppose you wish to produce several communicators in different sets of threads by performing `MPI_Comm_split()` on some base communicator. To ensure proper, thread-safe operation, you should replicate the base communicator with `MPI_Comm_dup()` (in the root thread or in one thread) and then perform `MPI_Comm_split()` on the resulting duplicate communicators.

- Do not free a communicator in one thread if it is still being used by another thread.

Error Handlers

When an error occurs as a result of an MPI call, the handler may not run on the same thread as the thread that made the error-raising call. In other words, you cannot assume that the error handler will execute in the local context of the thread that made the error-raising call. The error handler may be executed by another thread on the same process, distinct from the one that returns the error code. Therefore, you cannot rely on local variables for error handling in threads; instead, use global variables from the process.

Profiling Interface

The Prism development environment, a component of Sun HPC ClusterTools software, can be used in conjunction with the TNF probes and libraries included with Sun MPI for profiling your code. (TNF is included with the Solaris operating environment.) See Appendix D for information about the TNF probes and "Choosing a Library Path" on page 29 for information about linking to the "trace" or TNF libraries. See the *Prism User's Guide* for more information about the TNF viewer built into the Prism environment.

Sun MPI also meets the requirements of the profiling interface described in Chapter 8 of the MPI-1 Standard. You may write your own profiling library or choose from a number of available profiling libraries, such as those included with the multiprocessing environment (MPE) from Argonne National Laboratory. (See "MPE: Extensions to the Library" on page 21 for more information.) The *User's Guide for mpich, a Portable Implementation of MPI* includes more detailed information about using profiling libraries.

FIGURE 2-1 on page 21 illustrates how the software fits together. In this example, the user is linking against a profiling library that collects information on `MPI_Send()`. No profiling information is being collected for `MPI_Recv()`.

C profiling interfaces are needed even for Fortran programs. If there is profiling for both the Fortran and C version of an MPI function, then a Fortran call will encounter both profilings.

Be sure you make the library dynamic. If you make it static, it can encounter the linker oddities described in Section 8.4.3 of the MPI 1.1 standard.

To compile the program, the user's link line would look like this:

```
# cc ..... -llibrary-name -lmpi
```

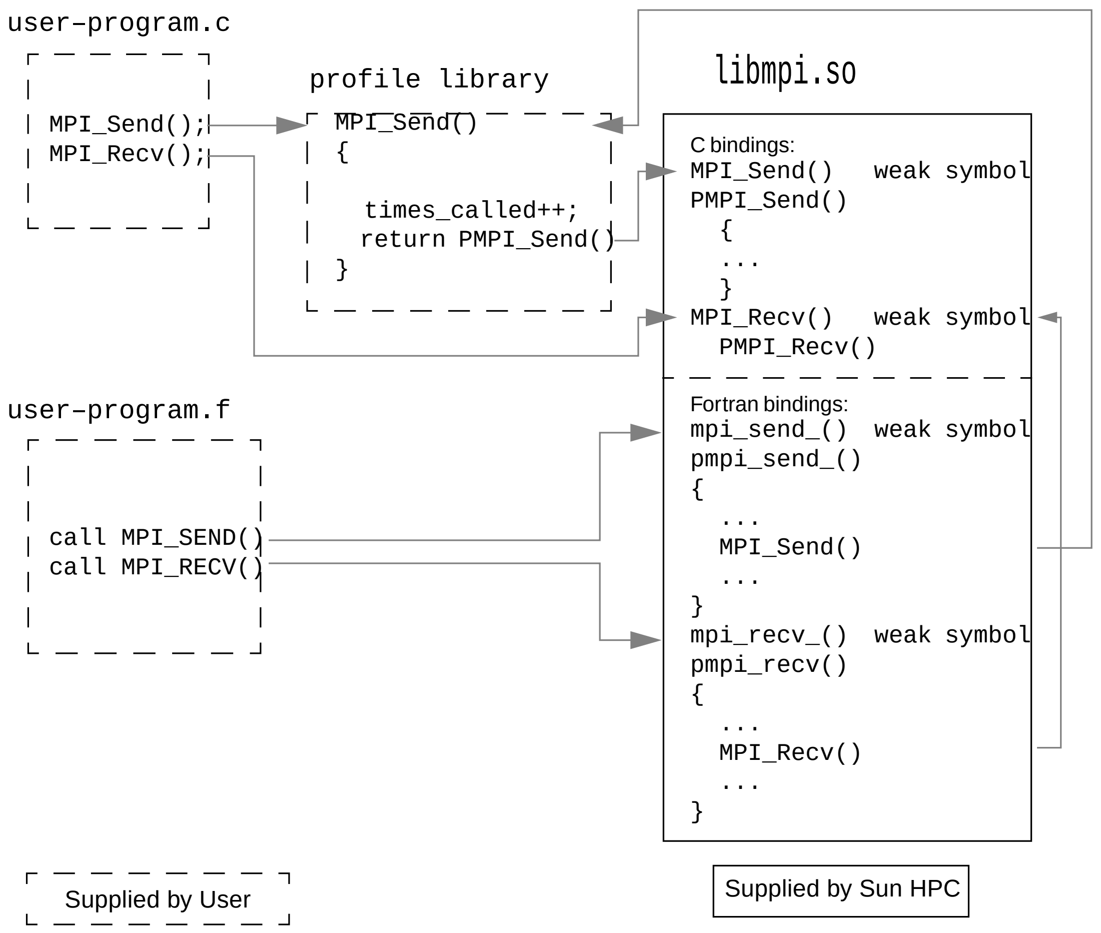


FIGURE 2-1 Sun MPI Profiling Interface

MPE: Extensions to the Library

Although the Sun MPI library does not include or support the multiprocessing environment (MPE) available from Argonne National Laboratory (ANL), it is compatible with MPE. If you would like to use these extensions to the MPI library, see the following instructions for downloading it from ANL and building it yourself. Note that this procedure may change if ANL makes changes to MPE.

▼ To Obtain and Build the MPE

The MPE software is available from Argonne National Laboratory. The `mpe.tar.gz` file is about 240 Kbytes.

1. **Use `ftp` to obtain the file.**

```
ftp://ftp.mcs.anl.gov/pub/mpi/misc/mpe.tar.gz
```

2. **Use `gunzip` and `tar` to decompress the software.**

```
# gunzip mpe.tar.gz

# tar xvf mpe.tar
```

3. **Change your current working directory to the `mpe` directory, and execute `configure` with the arguments shown.**

```
# cd mpe

# configure -cc=cc -fc=f77 -opt=-I/opt/SUNWhpc/include
```

4. **Execute a `make`.**

```
# make
```

Note – Sun MPI does not include the MPE error handlers. You must call the debug routines `MPE_Errors_call_dbx_in_xterm()` and `MPE_Signals_call_debugger()` yourself.

Refer to the *User's Guide for mpich, a Portable Implementation of MPI* for information on how to use MPE. It is available at the Argonne National Laboratory web site:

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

CHAPTER 3

Getting Started

This chapter describes how to develop, compile and link, execute, and debug a Sun MPI program. The chapter focuses on what is specific to the Sun MPI implementation and, for the most part, does not repeat information that can be found in related documents. Information about programming with the Sun MPI I/O routines is in Chapter 4.

Header Files

Include syntax must be placed at the top of any program that calls Sun MPI routines.

- For C and C++, use

  ```
  #include <mpi.h>
  ```

- For Fortran, use

  ```
  INCLUDE 'mpif.h'
  ```

These lines enable the program to access the Sun MPI version of the `mpi` header file, which contains the definitions, macros, and function prototypes required when compiling the program. Ensure that you are referencing the *Sun* MPI `include` file.

The `include` files are usually found in `/opt/SUNWhpc/include/` or `/opt/SUNWhpc/include/v9/`. If the compiler cannot find them, check that they exist and are accessible from the machine on which you are compiling your code. The location of the `include` file is specified by a compiler option (see "Compiling and Linking" on page 26).

Sample Code

Two simple Sun MPI programs are available in /opt/SUNWhpc/examples/mpi and are included here in their entirety. In the same directory you will find the Readme file, which provides instructions for using the examples, and the make file Makefile.

CODE EXAMPLE 3-1 Simple Sun MPI Program in C: connectivity.c

```
/*
 * Test the connectivity between all processes.
 */

#pragma ident "@(#)connectivity.c 1.1 99/02/02"

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <unistd.h>

#include <mpi.h>

int
main(int argc, char **argv)
{
    MPI_Status  status;
    int         verbose = 0;
    int         rank;
    int         np;                 /* number of processes in job */
    int         peer;
    int         i;
    int         j;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if (argc>1 && strcmp(argv[1], "-v")==0)
        verbose = 1;

    for (i=0; i<np; i++) {
        if (rank==i) {
            /* rank i sends to and receives from each higher rank */
            for(j=i+1; j<np; j++) {
                if (verbose)
                  printf("checking connection %4d <-> %-4d\n", i, j);
                MPI_Send(&rank, 1, MPI_INT, j, rank, MPI_COMM_WORLD);
```

CODE EXAMPLE 3-1 Simple Sun MPI Program in C: `connectivity.c` *(Continued)*

```
                MPI_Recv(&peer, 1, MPI_INT, j, j, MPI_COMM_WORLD, &status);
             }
         } else if (rank>i) {
             /* receive from and reply to rank i */
           MPI_Recv(&peer, 1, MPI_INT, i, i, MPI_COMM_WORLD, &status);
             MPI_Send(&rank, 1, MPI_INT, i, rank, MPI_COMM_WORLD);
         }
     }

     MPI_Barrier(MPI_COMM_WORLD);
     if (rank==0)
         printf("Connectivity test on %d processes PASSED.\n", np);

     MPI_Finalize();
     return 0;
 }
```

CODE EXAMPLE 3-2 Simple Sun MPI Program in Fortran: `monte.f`

```
 !
 ! Estimate pi via Monte-Carlo method.
 !
 ! Each process sums how many of samplesize random points generated
 ! in the square (-1,-1),(-1,1),(1,1),(1,-1) fall in the circle of
 ! radius 1 and center (0,0), and then estimates pi from the formula
 ! pi = (4 * sum) / samplesize.
 ! The final estimate of pi is calculated at rank 0 as the average of
 ! all the estimates.
 !
         program monte

         include 'mpif.h'

         double precision drand
         external drand

         double precision x, y, pi, pisum
         integer*4 ierr, rank, np
         integer*4 incircle, samplesize

         parameter(samplesize=2000000)

         call MPI_INIT(ierr)
         call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
         call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)

 !       seed random number generator
```

CODE EXAMPLE 3-2 Simple Sun MPI Program in Fortran: `monte.f` *(Continued)*

```
        x = drand(2 + 11*rank)

        incircle = 0
        do i = 1, samplesize
           x = drand(0)*2.0d0 - 1.0d0     ! generate a random point
           y = drand(0)*2.0d0 - 1.0d0

           if ((x*x + y*y) .lt. 1.0d0) then
              incircle = incircle+1       ! point is in the circle
           endif
        end do

        pi = 4.0d0 * DBLE(incircle) / DBLE(samplesize)

!       sum estimates at rank 0
         call MPI_REDUCE(pi, pisum, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
      &         0 , MPI_COMM_WORLD, ierr)

        if (rank .eq. 0) then
!          final estimate is the average
           pi = pisum / DBLE(np)
              print '(A,I4,A,F8.6,A)','Monte-Carlo estimate of pi by ',np,
      &          ' processes is ',pi,'.'
        endif

        call MPI_FINALIZE(ierr)
        end
```

Compiling and Linking

Sun MPI programs are compiled with ordinary C, C++, or Fortran compilers, just like any other C, C++, or Fortran program, and linked with the Sun MPI library.

The `mpf77`, `mpf90`, `mpcc`, and `mpCC` utilities may be used to compile Fortran 77, Fortran 90, C, and C++ programs, respectively. For example, you might use this entry

```
% mpf77 -fast -xarch=v8plusa -o a.out a.f -lmpi
```

to compile a Fortran 77 program that uses Sun MPI. See the `man` pages for more information on these utilities.

For performance, the single most important compilation switch is `–fast`. This is a macro that expands to settings appropriate for high performance for a general set of circumstances. Because its expansion varies from one compiler release to another, you may prefer to specify the underlying switches. To see what `–fast` expands to, use `–v` for “verbose” compilation output in Fortran, and -# for C. Also, `–fast` assumes native compilation, so you should compile on UltraSPARC™ processors.

The next important compilation switch is `–xarch`. Forte™ Developer 6 compilers set `-xarch` by default when you select `-fast` for native compilations. If you plan to compile on one type of processor and run the program on another type (non-native compilation), be sure to use the `-xarch` flag. Also use it to compile in 64-bit mode. For UltraSparc II, specify:

```
–xarch=v8plusa
```

or

```
–xarch=v9a
```

after `–fast` for 32-bit or 64-bit binaries, respectively. This version is only supported on Solaris 8 software. For UltraSparc III, specify:

```
–xarch=v8plusb
```

or

```
–xarch=v9b
```

The `v8plusb` and `v9b` flags apply only to UltraSparc III, and do not work with UltraSparc II. For more information, see the *Sun HPC ClusterTools 4 Performance Guide* and the documents that came with your compiler.

If you will be using the Prism debugger, you must compile your program with any of these Sun Forte™ Compilers for C/C^{++} or Fortran: release 6, release 6 update 1, or release 6 update 2 (see “Debugging” on page 31).

TABLE 3-1 Compile and Link Line Options for Sun MPI and Sun MPI I/O

When using . . .	Use . . .

C (nonthreaded example)	Use `mpcc` (below), or, if you prefer:
	`% cc` *filename*`.c -o` *filename* `\`
	`-I/opt/SUNWhpc/include -L/opt/SUNWhpc/lib \`
	`-R/opt/SUNWhpc/lib -lmpi`

C^{++}
Note that *x.y* represents the version of your C^{++} compiler.
	Use `mpCC` (below), or, if you prefer:
	`% CC` *filename*`.cc -o` *filename* `\`
	`-I/opt/SUNWhpc/include -L/opt/SUNWhpc/lib \`
	`-R/opt/SUNWhpc/lib -L/opt/SUNWhpc/lib/SC`*x.y* `\`
	`-R/opt/SUNWhpc/lib/SC`*x.y* `-mt -lmpi++ -lmpi`

`mpcc`, `mpCC`	`% mpcc -o` *filename* *filename*`.c -lmpi`
	`% mpCC -o` *filename* *filename*`.cc -mt -lmpi`

Fortran 77 (nonthreaded example)	Use `mpf77` (below), or, if you prefer:
	`% f77 -dalign` *filename*`.f -o` *filename* `\`
	`-I/opt/SUNWhpc/include -L/opt/SUNWhpc/lib \`
	`-R/opt/SUNWhpc/lib -lmpi`

Fortran on a 64-bit system	`% f77 -dalign` *filename*`.f -o` *filename* `\`
	`-I/opt/SUNWhpc/include/v9 \`
	`-L/opt/SUNWhpc/lib/sparcv9 \`
	`-R/opt/SUNWhpc/lib/sparcv9 -lmpi`

Fortran 90	Replace `mpf77` with `mpf90`, or `f77` with `f90`:

`mpf77`, `mpf90`, `mpf95`	`% mpf77 -o` *filename* `-dalign` *filename*`.f -lmpi`
	`% mpf90 -o` *filename* `-dalign` *filename*`.f -lmpi`
	`% mpf95 -o` *filename* `-dalign` *filename*`.f -lmpi`

Multithreaded programs and programs containing nonblocking MPI I/O routines	To support multithreaded code, replace `-lmpi` with `-lmpi_mt`. This change also supports programs with nonblocking MPI I/O routines. Note that `-lmpi` can be used for programs containing nonblocking MPI I/O routines, but `-lmpi_mt` *must* be used for multithreaded programs.

Note – For the Fortran interface, the `-dalign` option is necessary to avoid the possibility of bus errors. (The underlying C or C++ routines in Sun MPI internals assume that parameters and buffer types passed as REALs are double-aligned.)

Note – If your program has previously been linked to any static libraries, you will have to relink it to `libmpi.so` before executing it.

Choosing a Library Path

The paths for the MPI libraries, which you must specify when you are compiling and linking your program, are listed in the following table.

TABLE 3-2 Sun MPI Libraries

Category	Description	Path: `/opt/SUNWhpc/lib/`...
32-Bit Libraries	Default, not thread-safe	`libmpi.so`
	C^{++} (in addition to `libmpi.so`)	`SC6.0/libmpi++.so`
	Thread-safe	`libmpi_mt.so`
64-Bit Libraries	Default, not thread-safe	`sparcv9/libmpi.so`
	C^{++} (in addition to `sparcv9/libmpi.so`)	`SC6.0/libmpi++.so`
	Thread-safe	`sparcv9/libmpi_mt.so`

Overriding the Runtime Library

As shown in the sample compile and link lines in TABLE 3-1 on page 28, you use the `–R` flag in the compile and link line to specify the path for a runtime library when you are compiling. At run time, you can override the library specified in the `–R` argument by setting the `LD_LIBRARY_PATH` environment variable. For example, to link to the 32-bit trace libraries before running your program, type:

```
% setenv LD_LIBRARY_PATH /opt/SUNWhpc/lib/tnf
```

(This is a C shell example.)

Stubbing Thread Calls

The `libthread.so` libraries are automatically linked into the respective `libmpi.so` libraries. This means that any thread-function calls in your program can be resolved by the `libthread.so` library. Simply omitting `libthread.so` from the link line does not cause thread calls to be stubbed out; you must remove the thread calls yourself. For more information about the `libthread.so` library, see its man page. (For the location of Solaris man pages at your site, see your system administrator.)

Basic Job Execution

The *Sun HPC ClusterTools 4 User's Guide* and the `mprun` man page provide detailed information about running jobs with the CRE (ClusterTools Runtime Environment). Likewise, the *LSF Batch User's Guide*, the *Sun HPC ClusterTools 4 User's Guide*, and the `lsfintro` and `bsub` man pages provide thorough instructions for executing jobs with the LSF Suite. This section provides basic information about executing jobs with either environment.

Before starting your job, you may want to set one or more environment variables, which are also described in Appendix B and in the *Sun HPC ClusterTools 4 Performance Guide*.

Note – Both CRE and LSF accommodate parallel jobs of up to 2048 processes running on as many as 64 nodes.

Executing With CRE

When using the CRE, parallel jobs are launched using the `mprun` command. For example, to start a job with six processes named `mpijob`, use this command:

```
% mprun -np 6 mpijob
```

Executing With LSF Suite

Parallel jobs can either be launched by the LSF Parallel Application Manager (PAM) or be submitted in queues configured to run PAM as the parallel job starter. LSF's bsub command launches both parallel interactive and batch jobs. For example, to start a batch job named mpijob on four CPUs, use this command:

```
% bsub –n 4 pam mpijob
```

To launch an interactive job, add the –I argument to the command line. For example, to launch an interactive job named earth on a single CPU in the queue named sun, which is configured to launch jobs with PAM:

```
% bsub –q sun –Ip –n 1 earth
```

Debugging

Debugging parallel programs is notoriously difficult, since you are in effect debugging a program potentially made up of many distinct programs executing simultaneously. Even if the application is an SPMD (single process, multiple data) application, each instance may be executing a different line of code at any instant. The Prism development environment eases the debugging process considerably and is recommended for debugging with Sun HPC ClusterTools software.

Debugging With the Prism Environment

Note – To run the graphical version of the Prism environment, you must be running the Solaris 8 operating environment with either OpenWindows™ or the Common Desktop Environment (CDE), and with your DISPLAY environment variable set correctly. See the *Prism User's Guide* for information.

This section provides a brief introduction to the Prism development environment.

You can use a Prism session to debug more than one Sun MPI job at a time. To debug a child or client program it is necessary to launch an additional Prism session. If the child program is spawned using calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, Prism can (if enabled) debug the child program as well.

However, if an MPI job connects to another job, the current Prism session has control only of the parent or server MPI job. It cannot debug the children or clients of that job. This might occur, for example, when an MPI job sets up a client/server connection to another MPI job with `MPI_Comm_accept` or `MPI_Comm_connect`.

With the exception of programs using calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, to use the Prism environment to debug a Sun MPI program the program must be written in the SPMD (single program, multiple data) style. In other words, all processes that make up a Sun MPI program must be running the same executable.

`MPI_Comm_spawn_multiple` can create multiple executables with only one job id. Therefore, you can use the Prism environment to debug jobs with different executables that have been spawned with this command.

Starting Up Prism

To start Prism on a Sun MPI program, use the `–np` option to the `prism` command to specify how many processes you want to start. For example,

```
% prism –np 4 foo
```

launches Prism on executable `foo` with four processes.

This starts up a graphical version of Prism with your program loaded. You can then debug and visualize data in your Sun MPI program.

You can also attach Prism to running processes. First determine the job id (not the individual process id), or *jid*, using `mpps`. (See the *Sun HPC ClusterTools 4 User's Guide* for further information about `mpps`.) Then specify the jid at the command line with the `–np` option:

```
% prism –np 4 foo 12345
```

This will launch Prism and attach it to the processes running in job 12345.

One important feature of the Prism environment is that it lets you debug the Sun MPI program at any level of detail. You can look at the program as a whole or at subsets of processes within the program (for example, those that have an error condition), or at individual processes, all within the same debugging session. For complete information, see the *Prism User's Guide*.

Debugging With MPE

The multiprocessing environment (MPE) available from Argonne National Laboratory includes a debugger that can also be used for debugging at the thread level. For information about obtaining and building MPE, see "MPE: Extensions to the Library" on page 21.

CHAPTER 4

Programming With Sun MPI I/O

File I/O in Sun MPI is fully MPI-2 compliant. MPI I/O is specified as part of that standard, which was published in July, 1997. Its goal is to provide a library of routines featuring a portable parallel file system interface that is an extension of the MPI framework. See “Related Documentation” on page ix for more information about the MPI-2 standard.

The closest thing to a standard in file I/O is the UNIX file interface, but UNIX does not provide efficient coordination among multiple simultaneous accesses to a file, particularly when those accesses originate on multiple machines in a cluster. Another drawback of the UNIX file interface is its single-offset interface, that is, its lack of aggregate requests, which can also lead to inefficient access. The MPI I/O library provides routines that accomplish this coordination. Furthermore, MPI I/O allows multiple simultaneous access requests to be made to take advantage of the Sun HPC parallel file system, PFS. It is currently the only application programming interface through which users can access PFS. For more information about PFS, see the *Sun HPC ClusterTools 4 Administrator’s Guide* and the `pfsstat` man page.

Note – A direct interface to PFS is not available to the user. Currently, the only way to access PFS is through Sun’s implementation of MPI I/O or Solaris command-line utilities.

Using Sun MPI I/O

MPI I/O models file I/O on message passing; that is, writing to a file is analogous to sending a message, and reading from a file is analogous to receiving a message. The MPI library provides a high-level way of partitioning data among processes, which saves you from having to specify the details involved in making sure that the right pieces of data go to the right processes. This section describes basic MPI I/O concepts and the Sun MPI I/O routines.

Data Partitioning and Data Types

MPI I/O uses the MPI model of communicators and derived data types to describe communication between processes and I/O devices. MPI I/O determines which processes are communicating with a particular I/O device. Derived data types can be used to define the layout of data in memory and of data in a file on the I/O device. (For more information about derived data types, see "Data Types" on page 10.) Because MPI I/O builds on MPI concepts, it's easy for a knowledgeable MPI programmer to add MPI I/O code to a program.

Data is stored in memory and in the file according to MPI data types. Herein lies one of MPI and MPI I/O's advantages: Because they provide a mechanism whereby you can create your own data types, you have more freedom and flexibility in specifying data layout in memory and in the file.

The library also simplifies the task of describing how your data moves from processor memory to the file and back again. You create derived data types that describe how the data is arranged in the memory of each process and how it should be arranged in that part of the disk file associated with the process.

The Sun MPI I/O routines are described in "Routines" on page 38. But first, to be able to define a data layout, you will need to understand some basic MPI I/O data-layout concepts. The next section explains some of the fundamental terms and concepts.

Definitions

The following terms are used to describe partitioning data among processes. FIGURE 4-1 on page 37 illustrates some of these concepts.

- An *elementary data type* (or `etype`) is the unit of data access and positioning. It can be any MPI basic or derived data type. Data access is performed in elementary-data-type units, and offsets (see below) are expressed as a count of elementary data types.
- The *file type* (or `filetype`) is used to partition a file among processes; that is, a file type defines a template for accessing the file. It is either a single elementary data type or a derived MPI data type constructed from elementary data types. A file type may contain "holes," or extents of bytes that will not be accessed by this process.

- A file *displacement* (or `disp`) is an absolute byte position counted from the beginning of a file. The displacement defines the location where a view begins (see Figure 4-1, below).
- A *view* defines the current set of data visible and accessible by a process from an open file in terms of a displacement, an elementary data type, and a file type. The pattern described by a file type is repeated, beginning at the displacement, to define the view.
- An *offset* is a position relative to the current view, expressed as a count of elementary data types. Holes in the view's file type are ignored when calculating this position.

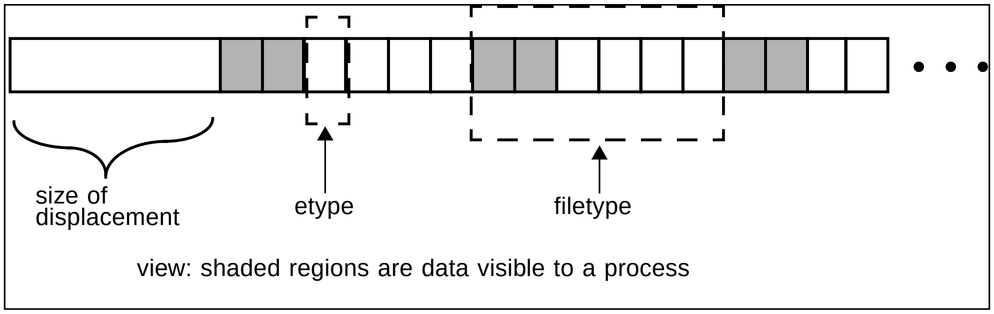


FIGURE 4-1 Displacement, the Elementary Data Type, the File Type, and the View

For a more detailed description of MPI I/O, see Chapter 9, "I/O," of the MPI-2 standard.

Note for Fortran Users

When writing a Fortran program, you must declare the variable `ADDRESS` as

```
INTEGER*MPI_ADDRESS_KIND ADDRESS
```

`MPI_ADDRESS_KIND` is a constant defined in `mpi.h`. This constant defines the length of the declared integer.

Routines

This release of Sun MPI includes all the MPI I/O routines, which are defined in Chapter 9, “I/O,” of the MPI-2 standard.

Code samples that use many of these routines are provided in “Sample Code” on page 45.

File Manipulation

Collective coordination	**Noncollective coordination**
`MPI_File_open()`	`MPI_File_delete()`
`MPI_File_close()`	`MPI_File_get_size()`
`MPI_File_set_size()`	`MPI_File_get_group()`
`MPI_File_preallocate()`	`MPI_File_get_amode()`

`MPI_File_open()` and `MPI_File_close()` are collective operations that open and close a file, respectively—that is, all processes in a communicator group must together open or close a file. To achieve a single-user, UNIX-like open, set the communicator to `MPI_COMM_SELF`.

`MPI_File_delete()` deletes a specified file.

The routines `MPI_File_set_size()`, `MPI_File_get_size()`, `MPI_File_get_group()`, and `MPI_File_get_amode()` get and set information about a file. When using the collective routine `MPI_File_set_size()` on a UNIX file, if the size that is set is smaller than the current file size, the file is truncated at the position defined by *size*. If *size* is set to be larger than the current file size, the file size becomes *size*.

When the file size is increased this way with `MPI_File_set_size()`, new regions are created in the file with displacements between the old file size and the larger, newly set file size. Sun MPI I/O does not necessarily allocate file space for such new regions. You may reserve file space either by using `MPI_File_preallocate()` or by performing a read or write to unallocated bytes. `MPI_File_preallocate()` ensures that storage space is allocated for a set quantity of bytes for the specified file; however, its use is very “expensive” in terms of performance and disk space.

The routine `MPI_File_get_group()` returns a communicator group, but it does not free the group.

File Hints

The opaque `info` object allows you to provide hints for optimization of your code, making it run faster or more efficiently, for example. These hints are set for each file, using the `MPI_File_open()`, `MPI_File_set_view()`, `MPI_File_set_info()`, and `MPI_File_delete()` routines. `MPI_File_set_info()` sets new values for the specified file's hints. `MPI_File_get_info()` returns all the hints that the system currently associates with the specified file.

When using UNIX files, Sun MPI I/O provides four hints for controlling how much buffer space it uses to satisfy I/O requests: `noncoll_read_bufsize`, `noncoll_write_bufsize`, `coll_read_bufsize`, and `coll_write_bufsize`. These hints may be tuned for your particular hardware configuration and application to improve performance for both noncollective and collective data accesses. For example, if your application uses a single MPI I/O call to request multiple noncontiguous chunks that form a regular strided pattern in the file, you may want to adjust the `noncoll_write_bufsize` to match the size of the stride. Note that these hints limit the size of MPI I/O's underlying buffers but do not limit the size of how much data a user can read or write in a single request.

File Views

The `MPI_File_set_view()` routine changes the view the process has of the data in the file, specifying its displacement, elementary data type, and file type, as well as setting the individual file pointers and shared file pointer to 0. `MPI_File_set_view()` is a collective routine; all processes in the group must pass identical values for the file handle and the elementary data type, although the values for the displacement, the file type, and the info object may vary. However, if you use the data-access routines that use file positioning with a shared file pointer, you must also give the displacement and the file type identical values. The data types passed in as the elementary data type and the file type must be committed.

You can also specify the type of data representation for the file. See "File Interoperability" on page 44 for information about registering data representation identifiers.

Note – Displacements within the file type and the elementary data type must be monotonically nondecreasing.

Data Access

The 35 data-access routines are categorized according to file positioning. Data access can be achieved by any of these methods of file positioning:

- By explicit offset
- By individual file pointer
- By shared file pointer

In the following subsections, each of these methods is discussed in more detail.

While *blocking* I/O calls will not return until the request is completed, *nonblocking* calls do not wait for the I/O request to complete. A separate "request complete" call, such as `MPI_Test()` or `MPI_Wait()`, is needed to confirm that the buffer is ready to be used again. Nonblocking routines have the prefix `MPI_File_i`, where the `i` stands for immediate.

All the nonblocking collective routines for data access are "split" into two routines, each with `_begin` or `_end` as a suffix. These *split collective* routines are subject to the semantic rules described in Section 9.4.5 of the MPI-2 standard.

Data Access With Explicit Offsets

Synchronism	Noncollective coordination	Collective coordination

Blocking	`MPI_File_read_at()`
	`MPI_File_write_at()`
		`MPI_File_read_at_all()`
		`MPI_File_write_at_all()`

Nonblocking or split collective	`MPI_File_iread_at()`
	`MPI_File_iwrite_at()`
		`MPI_File_read_at_all_begin()`
		`MPI_File_read_at_all_end()`
		`MPI_File_write_at_all_begin()`
		`MPI_File_write_at_all_end()`

To access data at an explicit offset, specify the position in the file where the next data access for each process should begin. For each call to a data-access routine, a process attempts to access a specified number of file types of a specified data type (starting at the specified offset) into a specified user buffer.

The offset is measured in elementary data type units relative to the current view; moreover, holes are not counted when locating an offset. The data is read from (in the case of a read) or written into (in the case of a write) those parts of the file specified by the current view. These routines store the number of buffer elements of a particular data type actually read (or written) in the status object, and all the other fields associated with the status object are undefined. The number of elements that are read or written can be accessed using `MPI_Get_count()`.

MPI_File_read_at() attempts to read from the file by the associated file handle returned from a successful MPI_File_open(). Similarly, MPI_File_write_at() attempts to write data from a user buffer to a file. MPI_File_iread_at() and MPI_File_iwrite_at() are the nonblocking versions of MPI_File_read_at() and MPI_File_write_at(), respectively.

MPI_File_read_at_all() and MPI_File_write_at_all() are collective versions of MPI_File_read_at() and MPI_File_write_at(), in which each process provides an explicit offset. The split collective versions of these nonblocking routines are listed in the table at the beginning of this section.

Data Access With Individual File Pointers

Synchronism	Noncollective coordination	Collective coordination

Blocking	MPI_File_read()
	MPI_File_write()
		MPI_File_read_all()
		MPI_File_write_all()

Nonblocking or split collective	MPI_File_iread()
	MPI_File_iwrite()
		MPI_File_read_all_begin()
		MPI_File_read_all_end()
		MPI_File_write_all_begin()
		MPI_File_write_all_end()

For each open file, Sun MPI I/O maintains one individual file pointer per process per collective MPI_File_open(). For these data-access routines, MPI I/O implicitly uses the value of the individual file pointer. These routines use and update only the individual file pointers maintained by MPI I/O by pointing to the next elementary data type after the one that has most recently been accessed. The individual file pointer is updated relative to the current view of the file. The shared file pointer is neither used nor updated. (For data access with shared file pointers, please see the next section.)

These routines have similar semantics to the explicit-offset data-access routines, except that the offset is defined here to be the current value of the individual file pointer.

MPI_File_read_all() and MPI_File_write_all() are collective versions of MPI_File_read() and MPI_File_write(), with each process using its individual file pointer.

MPI_File_iread() and MPI_File_iwrite() are the nonblocking versions of MPI_File_read() and MPI_File_write(), respectively. The split collective versions of MPI_File_read_all() and MPI_File_write_all() are listed in the table at the beginning of this section.

Pointer Manipulation

`MPI_File_seek`

`MPI_File_get_position`

`MPI_File_get_byte_offset`

Each process can call the routine `MPI_File_seek()` to update its individual file pointer according to the update mode. The update mode has the following possible values:

- `MPI_SEEK_SET` – The pointer is set to the offset.
- `MPI_SEEK_CUR` – The pointer is set to the current pointer position plus the offset.
- `MPI_SEEK_END` – The pointer is set to the end of the file plus the offset.

The offset can be negative for backwards seeking, but you cannot seek to a negative position in the file. The current position is defined as the elementary data item immediately following the last-accessed data item.

`MPI_File_get_position()` returns the current position of the individual file pointer relative to the current displacement and file type.

`MPI_File_get_byte_offset()` converts the offset specified for the current view to the displacement value, or absolute byte position, for the file.

Data Access With Shared File Pointers

Synchronism	Noncollective coordination	Collective coordination

Blocking	`MPI_File_read_shared()`
	`MPI_File_write_shared()`
		`MPI_File_read_ordered()`
		`MPI_File_write_ordered()`
		`MPI_File_seek_shared()`
		`MPI_File_get_position_shared()`

Nonblocking or split collective	`MPI_File_iread_shared()`
	`MPI_File_iwrite_shared()`
		`MPI_File_read_ordered_begin()`
		`MPI_File_read_ordered_end()`
		`MPI_File_write_ordered_begin()`
		`MPI_File_write_ordered_end()`

Sun MPI I/O maintains one shared file pointer per collective `MPI_File_open()` (shared among processes in the communicator group that opened the file). As with the routines for data access with individual file pointers, you can also use the current value of the shared file pointer to specify the offset of data accesses implicitly. These routines use and update only the shared file pointer; the individual file pointers are neither used nor updated by any of these routines.

These routines have similar semantics to the explicit-offset data-access routines, except:

- The offset is defined here to be the current value of the shared file pointer.
- Multiple calls (one for each process in the communicator group) affect the shared file pointer routines as if the calls were serialized.
- All processes must use the same file view.

After a shared file pointer operation is initiated, it is updated, relative to the current view of the file, to point to the elementary data item immediately following the last one requested, regardless of the number of items actually accessed.

`MPI_File_read_shared()` and `MPI_File_write_shared()` are blocking routines that use the shared file pointer to read and write files, respectively. The order of serialization is not deterministic for these noncollective routines, so you need to use other methods of synchronization if you wish to impose a particular order.

`MPI_File_iread_shared()` and `MPI_File_iwrite_shared()` are the nonblocking versions of `MPI_File_read_shared()` and `MPI_File_write_shared()`, respectively.

`MPI_File_read_ordered()` and `MPI_File_write_ordered()` are the collective versions of `MPI_File_read_shared()` and `MPI_File_write_shared()`. They must be called by all processes in the communicator group associated with the file handle, and the accesses to the file occur in the order determined by the ranks of the processes within the group. After all the processes in the group have issued their respective calls, for each process in the group, these routines determine the position of the shared file pointer after all processes with ranks lower than this process's rank had accessed their data. Then data is accessed (read or written) at that position. The shared file pointer is then updated by the amount of data requested by all processes of the group.

The split collective versions of `MPI_File_read_ordered()` and `MPI_File_write_ordered()` are listed in the table at the beginning of this section.

`MPI_File_seek_shared()` is a collective routine, and all processes in the communicator group associated with the particular file handler must call `MPI_File_seek_shared()` with the same file offset and the same update mode. All the processes are synchronized with a barrier before the shared file pointer is updated.

The offset can be negative for backwards seeking, but you cannot seek to a negative position in the file. The current position is defined as the elementary data item immediately following the last-accessed data item, even if that location is a hole.

`MPI_File_get_position_shared()` returns the current position of the shared file pointer relative to the current displacement and file type.

File Interoperability

`MPI_Register_datarep()`

`MPI_File_get_type_extent()`

Sun MPI I/O supports the basic data representations described in Section 9.5 of the MPI-2 standard:

- *native* – With native representation, data is stored exactly as in memory, in other words, in Solaris/UltraSPARC data representation. This format offers the highest performance and no loss of arithmetic precision. It should be used only in a homogeneous environment, that is, on Solaris/UltraSPARC nodes running Sun HPC ClusterTools software. It also may
- be used when the MPI application will perform the data type conversions itself.
- *internal* – With internal representation, data is stored in an implementation-dependent format, such as for Sun MPI.
- *external32* – With external32 representation, data is stored in a portable format, prescribed by the MPI-2 and IEEE standards.

These data representations, as well as any user-defined representations, are specified as an argument to `MPI_File_set_view()`.

You may create user-defined data representations with `MPI_Register_datarep()`. Once a data representation has been defined with this routine, you may specify it as an argument to `MPI_File_set_view()`, so that subsequent data-access operations will call the conversion functions specified with `MPI_Register_datarep()`.

If the file data representation is anything but native, you must be careful when constructing elementary data types and file types. For those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used.

`MPI_File_get_type_extent()` can be used to calculate the extents of data types in the file. The extent is the same for all processes accessing the specified file. If the current view uses a user-defined data representation, `MPI_File_get_type_extent()` uses one of the functions specified in setting the data representation to calculate the extent.

File Consistency and Semantics

Noncollective coordination	Collective coordination

`MPI_File_get_atomicity()`	`MPI_File_set_atomicity()`
	`MPI_File_sync()`

The routines ending in `_atomicity` allow you to set or query whether a file is in atomic or nonatomic mode. In *atomic mode*, all operations within the communicator group that opens a file are completed as if sequentialized into some serial order. In *nonatomic mode*, no such guarantee is made. In nonatomic mode, `MPI_File_sync()` can be used to ensure weak consistency.

The default mode varies with the number of nodes you are using. If you are running a job on a single node, a file is in *nonatomic* mode by default when it is opened. If you are running a job on more than one node, a file is in *atomic* mode by default.

`MPI_File_set_atomicity()` is a collective call that sets the consistency semantics for data-access operations. All the processes in the group must pass identical values for both the file handle and the Boolean flag that indicates whether atomic mode is set.

`MPI_File_get_atomicity()` returns the current consistency semantics for data-access operations. Again, a Boolean flag indicates whether the atomic mode is set.

Note – In some cases, setting atomicity to `false` may provide better performance. The default atomicity value on a cluster is `true`. The lack of synchronization among the distributed caches on a cluster will often prevent your data from completing in the desired state. In these circumstances, you may suffer performance disadvantages with atomicity set to `true`, especially when the data accesses overlap.

Sample Code

This section provides sample code to get you started with programming your I/O using Sun MPI. The first example shows how a parallel job can partition file data among its processes. That example is then adapted to use a broad range of other I/O programming styles supported by Sun MPI I/O. Finally, the last code sample illustrates the use of the nonblocking MPI I/O routines.

Remember that MPI I/O is part of MPI, so be sure to call `MPI_Init()` before calling any MPI I/O routines, and call `MPI_Finalize()` at the end of your program, even if you use only MPI I/O routines.

Partitioned Writing and Reading in a Parallel Job

MPI I/O was designed to enable processes in a parallel job to request multiple data items that are noncontiguous within a file. Typically, a parallel job partitions file data among the processes.

One method of partitioning a file is to derive the offset at which to access data from the rank of the process. The rich set of MPI derived types also makes it easy to partition file data. For example, you could create an MPI vector type as the filetype passed into `MPI_File_set_view()`. Since vector types do not end with a hole, you would make a call to either `MPI_Type_create_resized()` or `MPI_Type_ub()` to complete the partition. This call would extend the extent to include holes at the end of the type for processes with higher ranks. You can create a partitioned file by passing different displacements to `MPI_File_set_view()`. Each of these displacements would be derived from the process' rank. Consequently, offsets would not need to be derived from the ranks because only the data in the portion of the partition belonging to the process would be visible to the process.

The following example uses the first method that derives the file offsets directly from the rank of the process. Each process writes and reads `NUM_INTS` integers starting at the offset rank * `NUM_INTS`. It passes an explicit offset to the MPI I/O data-access routines `MPI_File_write_at()` and `MPI_File_read_at()`. It calls `MPI_Get_elements()` to find out how many elements were written or read. To verify that the write was successful, it compares the data written and read as well as set up an `MPI_Barrier()` before calling `MPI_File_get_size()` to verify that the file is the size expected upon completion of all the writes of the process.

Note that `MPI_File_set_view()` was called to set the view of the file as essentially an array of integers instead of the UNIX-like view of the file as an array of bytes. Thus, the offsets that are passed to `MPI_File_write_at()` and `MPI_File_read_at()` are indices into an array of integers and not a byte offset.

CODE EXAMPLE 4-1 Example code in which each process writes and reads `NUM_INTS` integers to a file using `MPI_File_write_at()` and `MPI_File_read_at()`, respectively.

```
/* wr_at.c
 *
 * Example to demonstrate use of MPI_File_write_at and MPI_File_read_at
 *
*/

#include <stdio.h>
#include "mpi.h"

#define NUM_INTS 100

void sample_error(int error, char *string)
{
```

CODE EXAMPLE 4-1 Example code in which each process writes and reads `NUM_INTS` integers to a file using `MPI_File_write_at()` and `MPI_File_read_at()`, respectively. *(Continued)*

```
  fprintf(stderr, "Error %d in %s\n", error, string);
  MPI_Finalize();
  exit(-1);
}

void
main( int argc, char **argv )
{
  char filename[128];
  int i, rank,  comm_size;
  int *buff1, *buff2;
  MPI_File fh;
  MPI_Offset disp, offset, file_size;
  MPI_Datatype etype, ftype, buftype;
  MPI_Info info;
  MPI_Status status;
  int result, count, differs;

  if(argc < 2) {
    fprintf(stdout, "Missing argument: filename\n");
    exit(-1);
  }
  strcpy(filename, argv[1]);

  MPI_Init(&argc, &argv);

  /* get this processor’s rank */
  result = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(result != MPI_SUCCESS)
    sample_error(result, "MPI_Comm_rank");

  result = MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  if(result != MPI_SUCCESS)
    sample_error(result, "MPI_Comm_size");

  /* communicator group MPI_COMM_WORLD opens file "foo"
     for reading and writing (and creating, if necessary) */
  result = MPI_File_open(MPI_COMM_WORLD, filename,
        MPI_MODE_RDWR | MPI_MODE_CREATE, (int)NULL, &fh);
  if(result != MPI_SUCCESS)
    sample_error(result, "MPI_File_open");

  /* Set the file view which tiles the file type MPI_INT, starting
     at displacement 0.  In this example, the etype is also MPI_INT.  */
  disp = 0;
  etype = MPI_INT;
```

CODE EXAMPLE 4-1 Example code in which each process writes and reads NUM_INTS integers to a file using MPI_File_write_at() and MPI_File_read_at(), respectively. *(Continued)*

```
   ftype = MPI_INT;
   info = (MPI_Info)NULL;
   result = MPI_File_set_view(fh, disp, etype, ftype, (char *)NULL, info);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_File_set_view");

   /* Allocate and initialize a buffer (buff1) containing NUM_INTS integers,
      where the integer in location i is set to i. */
   buff1 = (int *)malloc(NUM_INTS*sizeof(int));
   for(i=0;i<NUM_INTS;i++) buff1[i] = i;

   /* Set the buffer type to also be MPI_INT, then write the buffer (buff1)
      starting at offset 0, i.e., the first etype in the file. */
   buftype = MPI_INT;
   offset = rank * NUM_INTS;
   result = MPI_File_write_at(fh, offset, buff1, NUM_INTS, buftype, &status);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_File_write_at");

   result = MPI_Get_elements(&status, MPI_BYTE, &count);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_Get_elements");
   if(count != NUM_INTS*sizeof(int))
     fprintf(stderr, "Did not write the same number of bytes as requested\n");
   else
     fprintf(stdout, "Wrote %d bytes\n", count);

   /* Allocate another buffer (buff2) to read into, then read NUM_INTS
      integers into this buffer.  */
   buff2 = (int *)malloc(NUM_INTS*sizeof(int));
   result = MPI_File_read_at(fh, offset, buff2, NUM_INTS, buftype, &status);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_File_read_at");

   /* Find out how many bytes were read and compare to how many
      we expected */
   result = MPI_Get_elements(&status, MPI_BYTE, &count);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_Get_elements");
   if(count != NUM_INTS*sizeof(int))
     fprintf(stderr, "Did not read the same number of bytes as requested\n");
   else
     fprintf(stdout, "Read %d bytes\n", count);

   /* Check to see that each integer read from each location is
      the same as the integer written to that location. */
```

CODE EXAMPLE 4-1 Example code in which each process writes and reads NUM_INTS integers to a file using MPI_File_write_at() and MPI_File_read_at(), respectively. *(Continued)*

```
  differs = 0;
  for(i=0; i<NUM_INTS; i++) {
    if(buff1[i] != buff2[i]) {
      fprintf(stderr, "Integer number %d differs\n", i);
      differs = 1;
    }
  }
  if(!differs)
    fprintf(stdout, "Wrote and read the same data\n");

  MPI_Barrier(MPI_COMM_WORLD);

  result = MPI_File_get_size(fh, &file_size);
  if(result != MPI_SUCCESS)
    sample_error(result, "MPI_File_get_size");

  /* Compare the file size with what we expect */
  /* To see a negative response, make the file preexist with a larger
     size than what is written by this program */
  if(file_size != (comm_size * NUM_INTS * sizeof(int)))
    fprintf(stderr, "File size is not equal to the write size\n");

  result = MPI_File_close(&fh);
  if(result != MPI_SUCCESS)
    sample_error(result, "MPI_File_close");

  MPI_Finalize();

  free(buff1);
  free(buff2);
}
```

Data Access Styles

You can adapt the example above to support the I/O programming style that best suits your application. Essentially, there are three dimensions on which to choose an appropriate data access routine for a particular task: file pointer type, collective or noncollective, and blocking or nonblocking.

You need to choose which file pointer type to use: explicit, individual, or shared. The example above used an explicit pointer and passed it directly as the offset parameter to the MPI_File_write_at() and MPI_File_read_at() routines. Using an explicit pointer is equivalent to calling MPI_File_seek() to set the individual file pointer to offset, then calling MPI_File_write() or MPI_File_read(), which is directly analogous to calling UNIX lseek() and write() or read(). If each process accesses the file sequentially, individual file pointers save you the effort of recalculating offset for each data access. A different shared file pointer could be used in situations where all the processes needed to cooperatively access a file in a sequential way, such as to write log files.

Collective data-access routines enable you to enforce some implicit coordination among the processes in a parallel job when making data accesses. For example, if a parallel job alternately reads in a matrix and performs computation on it, but cannot progress to the next stage of computation until all processes have completed the last stage, then a coordinated effort between processes when accessing data might be more efficient. The example above could easily append the suffix `_all` to `MPI_File_write_at()` and `MPI_File_read_at()` to make the accesses collective. By coordinating the processes, you could achieve greater efficiency in the MPI library or at the file system level in buffering or caching the next matrix. In contrast, noncollective accesses are used when it is not evident that any benefit would be gained by coordinating disparate accesses by each process. UNIX file accesses are noncollective.

Overlapping I/O With Computation and Communication

MPI I/O also supports nonblocking versions of each of the data-access routines; that is, the data-access routines that have the letter `i` before `write` or `read` in the routine name (`i` stands for *immediate*). By definition, nonblocking I/O routines return immediately after the I/O request has been issued and do not wait until the I/O request has completed. This functionality enables you to perform computation and communication at the same time as the I/O. Since large I/O requests can take a long time to complete, this provides a way to more efficiently utilize your program's waiting time.

As in the previous example, parallel jobs often partition large matrices stored in files. These parallel jobs may use many large matrices, or matrices that are too large to fit into memory at once. Thus, each process may access the multiple and/or large matrices in stages. During each stage, a process reads in a chunk of data, then performs some computation on it (which may involve communicating with the other processes in the parallel job). While performing the computation and communication, the process could issue a nonblocking I/O read request for the next chunk of data. Similarly, once the computation on a particular chunk has completed, a nonblocking write request could be issued before performing computation and communication on the next chunk.

The following example code illustrates the use of a nonblocking data-access routine. Note that like nonblocking communication routines, the nonblocking I/O routines require a call to `MPI_Wait()` to wait for the nonblocking request to complete, or repeated calls to `MPI_Test()` to determine when the nonblocking data access has completed. Once complete, the write or read buffer is available for use again by the program.

CODE EXAMPLE 4-2 Example code in which each process reads and writes NUM_BYTES bytes to a file using the nonblocking MPI I/O routines MPI_File_iread_at() and MPI_File_iwrite_at(), respectively. Note the use of MPI_Wait() and MPI_Test() to determine when the nonblocking requests have completed.

```
/* iwr_at.c
 *
 * Example to demonstrate use of MPI_File_iwrite_at and MPI_File_iread_at
 *
*/

#include <stdio.h>
#include "mpi.h"

#define NUM_BYTES 100

void sample_error(int error, char *string)
{
  fprintf(stderr, "Error %d in %s\n", error, string);
  MPI_Finalize();
  exit(-1);
}

void
main( int argc, char **argv )
{
  char filename[128];
  char *buff;
  MPI_File fh;
  MPI_Offset offset;
  MPI_Request request;
  MPI_Status status;
  int i, rank, flag, result;

  if(argc < 2) {
    fprintf(stdout, "Missing argument: filename\n");
    exit(-1);
  }
  strcpy(filename, argv[1]);

  MPI_Init(&argc, &argv);

  result = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(result != MPI_SUCCESS)
    sample_error(result, "MPI_Comm_rank");

  result = MPI_File_open(MPI_COMM_WORLD, filename,
        MPI_MODE_RDWR | MPI_MODE_CREATE,
```

CODE EXAMPLE 4-2 Example code in which each process reads and writes NUM_BYTES bytes to a file using the nonblocking MPI I/O routines MPI_File_iread_at() and MPI_File_iwrite_at(), respectively. Note the use of MPI_Wait() and MPI_Test() to determine when the nonblocking requests have completed. *(Continued)*

```
         (MPI_Info)NULL, &fh);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_File_open");

   buff = (char *)malloc(NUM_BYTES*sizeof(char));
   for(i=0;i<NUM_BYTES;i++) buff[i] = i;

   offset = rank * NUM_BYTES;
   result = MPI_File_iread_at(fh, offset, buff, NUM_BYTES,
              MPI_BYTE, &request);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_File_iread_at");

   /* Perform some useful computation and/or communication */

   result = MPI_Wait(&request, &status);

   buff = (char *)malloc(NUM_BYTES*sizeof(char));
   for(i=0;i<NUM_BYTES;i++) buff[i] = i;
   result = MPI_File_iwrite_at(fh, offset, buff, NUM_BYTES,
               MPI_BYTE, &request);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_File_iwrite_at");

   /* Perform some useful computation and/or communication */

   flag = 0;
   i = 0;
   while(!flag) {
      result = MPI_Test(&request, &flag, &status);
      i++;
      /* Perform some more computation or communication, if possible */
   }

   result = MPI_File_close(&fh);
   if(result != MPI_SUCCESS)
     sample_error(result, "MPI_File_close");

   MPI_Finalize();

   fprintf(stdout, "Successful completion\n");

   free(buff);
 }
```

APPENDIX **A**

Sun MPI and Sun MPI I/O Routines

The tables in this appendix list the Sun MPI and Sun MPI I/O routines, along with the C syntax of the routines and a brief description of each. For more information about the routines, see their online man pages, usually found in

```
/opt/SUNWhpc/man
```

Your system administrator can tell you where they are installed at your site.

Sun MPI Routines

TABLE A-1 on page 64 lists the Sun MPI routines in alphabetical order. The following sections list the routines by functional category.

Point-to-Point Communication

Blocking Routines

```
MPI_Send()
MPI_Bsend()
MPI_Ssend()
MPI_Rsend()
MPI_Recv()
MPI_Sendrecv()
MPI_Sendrecv_replace()
```

Nonblocking Routines

```
MPI_Isend()
MPI_Ibsend()
MPI_Issend()
MPI_Irsend()
MPI_Irecv()
```

Communication Buffer Allocation

```
MPI_Buffer_attach()
MPI_Buffer_detach()
```

Status Data Structure

```
MPI_Get_count()
MPI_Get_elements()
```

Persistent (Half-Channel) Communication

```
MPI_Send_init()
MPI_Bsend_init()
MPI_Rsend_init()
MPI_Ssend_init()
MPI_Recv_init()
MPI_Start()
MPI_Startall()
```

Completion Tests

```
MPI_Wait()
MPI_Waitany()
MPI_Waitsome()
MPI_Waitall()
MPI_Test()
MPI_Testany()
MPI_Testsome()
MPI_Testall()
MPI_Request_free()
MPI_Cancel()
MPI_Test_cancelled()
```

Probing for Messages (Blocking and Nonblocking)

```
MPI_Probe()
MPI_Iprobe()
```

Packing and Unpacking Functions

```
MPI_Pack()
MPI_Pack_size()
MPI_Unpack()
```

Derived Data Type Constructors and Functions

MPI_Address(): *Deprecated – Use* MPI_Get_address()
MPI_Type_commit()
MPI_Type_contiguous()
MPI_Type_create_indexed_block()
MPI_Type_create_keyval()
MPI_Type_delete_attr()
MPI_Type_dup()
MPI_Type_free_keyval()
MPI_Type_get_attr()
MPI_Type_set_attr()
MPI_Type_get_contents()
MPI_Type_get_envelope()
MPI_Type_get_name()
MPI_Type_set_name()
MPI_Type_create_resized()
MPI_Type_free()
MPI_Type_get_true_extent()
MPI_Type_hvector(): *Deprecated* – Use MPI_Type_create_hvector()
MPI_Type_indexed()
MPI_Type_hindexed(): *Deprecated* – Use MPI_Type_create_hindexed()
MPI_Type_struct(): *Deprecated* – Use MPI_Type_create_struct()
MPI_Type_lb(): *Deprecated* – Use MPI_Type_get_extent()
MPI_Type_ub(): *Deprecated* – Use MPI_Type_get_extent()
MPI_Type_vector()
MPI_Type_extent(): *Deprecated* – Use MPI_Type_get_extent()
MPI_Type_size()

One-Sided Communication

Initialization

MPI_Win_create()
MPI_Win_free()
MPI_Win_get_group()

Communication Calls

```
MPI_Put()
MPI_Get()
MPI_Accumulate()
```

Synchronization Calls

```
MPI_Win_fence()
MPI_Win_lock()
MPI_Win_unlock()
```

The Sun HPC ClusterTools implementation of one-sided communication does not support these synchronization functions from the standard:

- `MPI_Win_start()`
- `MPI_Win_complete()`
- `MPI_Win_post()`
- `MPI_Win_wait()`
- `MPI_Win_test()`

Collective Communication

Barrier

```
MPI_Barrier()
```

Broadcast

```
MPI_Bcast()
```

Processor Gather and Scatter

```
MPI_Gather()
MPI_Gatherv()
MPI_Allgather()
MPI_Allgatherv()
MPI_Scatter()
MPI_Scatterv()
MPI_Alltoall()
MPI_Alltoallv()
```

Global Reduction and Scan Operations

```
MPI_Reduce()
MPI_Allreduce()
MPI_Reduce_scatter()
MPI_Scan()
MPI_Op_create()
MPI_Op_free()
```

Groups and Communicators

Group Management

Group Accessors

```
MPI_Group_size()
MPI_Group_rank()
MPI_Group_translate_ranks()
MPI_Group_compare()
```

Group Constructors

```
MPI_Comm_group()
MPI_Group_union()
MPI_Group_intersection()
MPI_Group_difference()
MPI_Group_incl()
MPI_Group_excl()
MPI_Group_range_incl()
MPI_Group_range_excl()
MPI_Group_free()
```

Communicator Management

Communicator Accessors

```
MPI_Comm_size()
MPI_Comm_rank()
MPI_Comm_compare()
```

Communicator Constructors

```
MPI_Comm_dup()
MPI_Comm_create()
MPI_Comm_split()
MPI_Comm_free()
```

Intercommunicators

```
MPI_Comm_test_inter()
MPI_Comm_remote_group()
MPI_Comm_remote_size()
MPI_Intercomm_create()
MPI_Intercomm_merge()
```

Communicator Attributes

MPI_Keyval_create(): *Deprecated* – Use MPI_Comm_create_keyval().
MPI_Keyval_free(): *Deprecated* – Use MPI_Comm_free_keyval().
MPI_Attr_put(): *Deprecated* – Use MPI_Comm_set_attr().
MPI_Attr_get(): *Deprecated* – Use MPI_Comm_get_attr().
MPI_Attr_delete(): *Deprecated* – Use MPI_Comm_delete_attr().

Process Topologies

MPI_Cart_create()
MPI_Dims_create()
MPI_Graph_create()
MPI_Topo_test()
MPI_Graphdims_get()
MPI_Graph_get()
MPI_Cartdim_get()
MPI_Cart_get()
MPI_Cart_rank()
MPI_Cart_coords()
MPI_Graph_neighbors()
MPI_Graph_neighbors_count()
MPI_Cart_shift()
MPI_Cart_sub()
MPI_Cart_map()
MPI_Graph_map()

Process Creation and Management

Establishing Communication

MPI_Close_port()
MPI_Comm_accept()
MPI_Comm_connect()
MPI_Comm_disconnect()
MPI_Open_port()

Process Manager Interface

MPI_Comm_get_parent()
MPI_Comm_spawn()
MPI_Comm_spawn_multiple()

Environmental Inquiry Functions and Profiling

Startup and Shutdown

MPI_Init()
MPI_Finalize()
MPI_Finalized()
MPI_Initialized()
MPI_Abort()
MPI_Get_processor_name()
MPI_Get_version()

Error Handler Functions

MPI_Errhandler_create(): *Deprecated* – Use MPI_Comm_create_errhandler().
MPI_Errhandler_set(): *Deprecated* – Use MPI_Comm_set_errhandler().
MPI_Errhandler_get(): *Deprecated* – Use MPI_Comm_get_errhandler().
MPI_Errhandler_free()
MPI_Error_string()
MPI_Error_class()

Info Objects

`MPI_Info_create()`
`MPI_Info_delete()`
`MPI_Info_dup()`
`MPI_Info_free()`
`MPI_Info_get()`
`MPI_Info_get_nkeys()`
`MPI_Info_get_nthkey()`
`MPI_Info_get_valuelen()`
`MPI_Info_set()`

Timers

`MPI_Wtime()`
`MPI_Wtick()`

Profiling

`MPI_Pcontrol()`

Miscellaneous

Associating Information With Status

`MPI_Status_set_cancelled()`
`MPI_Status_set_elements()`

Generalized Requests

`MPI_Grequest_complete()`
`MPI_Grequest_start()`

Naming Objects

```
MPI_Comm_get_name()
MPI_Comm_set_name()
MPI_Type_get_name()
MPI_Type_set_name()
```

Threads

```
MPI_Query_thread()
```

Handle Translation

```
MPI_Comm_c2f()
MPI_Comm_f2c()
MPI_Group_c2f()
MPI_Group_f2c()
MPI_Info_c2f()
MPI_Info_f2c()
MPI_Op_c2f()
MPI_Op_f2c()
MPI_Request_c2f()
MPI_Request_f2c()
MPI_Type_c2f()
MPI_Type_f2c()
```

Status Conversion

```
MPI_Status_c2f()
MPI_Status_f2c()
```

MPI Routines: Alphabetical Listing

TABLE A-1 Sun MPI Routines

Routine and C Syntax	Description
`MPI_Abort`(MPI_Comm *comm*, int *errorcode*)	Terminates MPI execution environment.
`MPI_Accumulate`(void **origin_addr*, int *origin_count*, MPI_Datatype *origin_datatype*, int *target_rank*, MPI_Aint *target_disp*, int *target_count*, MPI_Datatype *target_datatype*, MPI_Op *op*, MPI_Win *win*)	Combines the contents of the origin buffer with that of a target buffer.
`MPI_Address`(void **location*, MPI_Aint **address*)	*Deprecated:* Use instead `MPI_Get_address()`. Gets the address of a location in memory.
`MPI_Allgather`(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, MPI_Comm *comm*)	Gathers data from all processes and distributes it to all.
`MPI_Allgatherv`(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int **recvcount*, int **displs*, MPI_Datatype *recvtype*, MPI_Comm *comm*)	Gathers data from all processes and delivers it to all. Each process may contribute a different amount of data.
`MPI_Alloc_mem`(MPI_Aint *size*, MPI_Info *info*, void **baseptr*)	Allocates a specified memory segment.
`MPI_Allreduce`(void **sendbuf*, void **recvbuf*, int *count*, MPI_Datatype *datatype*, MPI_Op *op*, MPI_Comm *comm*)	Combines values from all processes and distributes the result back to all processes.
`MPI_Alltoall`(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, MPI_Comm *comm*)	Sends data from all to all processes.
`MPI_Alltoallv`(void **sendbuf*, int **sendcounts*, int **sdispls*, MPI_Datatype *sendtype*, void **recvbuf*, int **recvcounts*, int **rdispls*, MPI_Datatype *recvtype*, MPI_Comm *comm*)	Sends data from all to all processes, with a displacement. Each process may contribute a different amount of data.
`MPI_Attr_delete`(MPI_Comm *comm*, int *keyval*)	*Deprecated:* Use instead `MPI_Comm_delete_attr()`. Deletes attribute value associated with a key.
`MPI_Attr_get`(MPI_Comm *comm*, int *keyval*, void **attribute_val*, int **flag*)	*Deprecated:* Use instead `MPI_Comm_get_attr()`. Retrieves attribute value by key.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
`MPI_Attr_put`(MPI_Comm *comm*, int *keyval*, void **attribute_val*)	*Deprecated:* Use instead `MPI_Comm_set_attr()`. Stores attribute value associated with a key.
`MPI_Barrier`(MPI_Comm *comm*)	Blocks until all processes have reached this routine.
`MPI_Bcast`(void **buffer*, int *count*, MPI_Datatype *datatype*, int *root*, MPI_Comm *comm*)	Broadcasts a message from the process with rank `root` to all other processes of the group.
`MPI_Bsend`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*)	Basic send with user-specified buffering.
`MPI_Bsend_init`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Builds a handle for a buffered send.
`MPI_Buffer_attach`(void **buf*, int *size*)	Attaches a user-defined buffer for sending.
`MPI_Buffer_detach`(void **buf*, int **size*)	Removes an existing buffer (for use in `MPI_Bsend()`, etc.).
`MPI_Cancel`(MPI_Request **request*)	Cancels a communication request.
`MPI_Cart_coords`(MPI_Comm *comm*, int *rank*, int *maxdims*, int **coords*)	Determines process coordinates in Cartesian topology given rank in group.
`MPI_Cart_create`(MPI_Comm *comm_old*, int *ndims*, int **dims*, int **periods*, int *reorder*, MPI_Comm **comm_cart*)	Makes a new communicator to which Cartesian topology information has been attached.
`MPI_Cart_get`(MPI_Comm *comm*, int *maxdims*, int **dims*, int **periods*, int **coords*)	Retrieves Cartesian topology information associated with a communicator.
`MPI_Cart_map`(MPI_Comm *comm*, int *ndims*, int **dims*, int **periods*, int **newrank*)	Maps process to Cartesian topology information.
`MPI_Cart_rank`(MPI_Comm *comm*, int **coords*, int **rank*)	Determines process rank in communicator given Cartesian location.
`MPI_Cart_shift`(MPI_Comm *comm*, int *direction*, int *disp*, int **rank_source*, int **rank_dest*)	Returns the shifted source and destination ranks, given a shift direction and amount.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Cart_sub(MPI_Comm *comm*, int **remain_dims*, MPI_Comm **comm_new*)	Partitions a communicator into subcommunicators that form lower-dimensional Cartesian subgrids.
MPI_Cartdim_get(MPI_Comm *comm*, int **ndims*)	Retrieves Cartesian topology information associated with a communicator.
MPI_Close_port(char **port_name*)	Releases the specified network address.
MPI_Comm_accept(char **port_name*, MPI_Info *info*, int *root*, MPI_Comm *comm*, MPI_Comm **newcomm*)	Establishes communication with a client (collective).
MPI_Comm_c2f(MPI_Comm *comm*)	Translates a C handle into a Fortran handle.
MPI_Comm_compare(MPI_Comm *comm1*, MPI_Comm *comm2*, int **result*)	Compares two communicators.
MPI_Comm_connect(char **port_name*, MPI_Info *info*, int *root*, MPI_Comm *comm*, MPI_Comm **newcomm*)	Establishes communication with a server (collective).
MPI_Comm_create(MPI_Comm *comm*, MPI_Group *group*, MPI_Comm **newcomm*)	Creates a new communicator from a group.
MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn **function*, MPI_Errhandler **errhandler*)	Creates an error handler that can be attached to communicators.
MPI_Comm_create_keyval(MPI_Comm_copy_attr_function **comm_copy_attr_fn*, MPI_Comm_delete_attr_function **comm_delete_attr_fn*, int **comm_keyval*, void **extra_state*)	Generates a new attribute key.
MPI_Comm_delete_attr(MPI_Comm *comm*, int *comm_keyval*)	Deletes attribute value associated with a key.
MPI_Comm_disconnect(MPI_Comm **comm*)	De-allocates communicator object and sets handle to `MPI_COMM_NULL` (collective).
MPI_Comm_dup(MPI_Comm *comm*, MPI_Comm **newcomm*)	Duplicates an existing communicator with all its cached information.
MPI_Comm_f2c(MPI_Fint *comm*)	Translates a Fortran handle into a C handle.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Comm_free(MPI_Comm **comm*)	Marks the communicator object for deallocation.
MPI_Comm_free_keyval(int **comm_keyval*)	Frees attribute key for communicator cache attribute.
MPI_Comm_get_attr(MPI_Comm *comm*, int *comm_keyval*, void **attribute_val*, int **flag*)	Retrieves attribute value by key.
MPI_Comm_get_errhandler(MPI_Comm *comm*, MPI_Errhandler **errhandler*)	Retrieves error handler associated with a communicator.
MPI_Comm_get_name(MPI_Comm *comm*, char **comm_name*, int **resultlen*)	Returns the name that was most recently associated with a communicator.
MPI_Comm_get_parent(MPI_Comm **parent*)	Returns the parent intercommunicator of current spawned process.
MPI_Comm_group(MPI_Comm *comm*, MPI_Group **group*)	Accesses the group associated with a communicator.
MPI_Comm_rank(MPI_Comm *comm*, int **rank*)	Determines the rank of the calling process in a communicator.
MPI_Comm_remote_group(MPI_Comm *comm*, MPI_Group **group*)	Accesses the remote group associated with an intercommunicator.
MPI_Comm_remote_size(MPI_Comm *comm*, int *size*)	Determines the size of the remote group associated with an intercommunicator.
MPI_Comm_set_attr(MPI_Comm *comm*, int *comm_keyval*, void **attribute_val*)	Stores attribute value associated with a key.
MPI_Comm_set_errhandler(MPI_Comm *comm*, MPI_Errhandler **errhandler*)	Attaches a new error handler to a communicator.
MPI_Comm_set_name(MPI_Comm *comm*, char **comm_name*)	Associates a name with a communicator.
MPI_Comm_size(MPI_Comm *comm*, int **size*)	Determines the size of the group associated with a communicator.
MPI_Comm_spawn(char **command*, char **argv*[ ], int *maxprocs*, MPI_Info *info*, int *root*, MPI_Comm *comm*, MPI_Comm **intercomm*, int *array_of_errcodes*[ ])	Spawns a number of identical binaries.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
`MPI_Comm_spawn_multiple`(int *count*, char **array_of_commands*[ ], char ***array_of_argv*[ ], int *array_of_maxprocs*[ ], MPI_Info *array_of_info*[ ], int *root*, MPI_Comm *comm*, MPI_Comm **intercomm*, int *array_of_errcodes*[ ])	Spawns multiple binaries, or the same binary with multiple sets of arguments.
`MPI_Comm_split`(MPI_Comm *comm*, int *color*, int *key*, MPI_Comm **newcomm*)	Creates new communicators based on colors and keys.
`MPI_Comm_test_inter`(MPI_Comm *comm*, int **flag*)	Tests whether a communicator is an intercommunicator.
`MPI_Dims_create`(int *nnodes*, int *ndims*, int **dims*)	Creates a division of processors in a Cartesian grid.
`MPI_Errhandler_create`(MPI_Handler_function **function*, MPI_Errhandler **errhandler*)	*Deprecated:* Use instead `MPI_Comm_create_errhandler( )`. Creates an MPI error handler.
`MPI_Errhandler_free`(MPI_Errhandler **errhandler*)	Frees an MPI error handler.
`MPI_Errhandler_get`(MPI_Comm *comm*, MPI_Errhandler **errhandler*)	*Deprecated:* Use instead `MPI_Comm_get_errhandler()`. Gets the error handler for a communicator.
`MPI_Errhandler_set`(MPI_Comm *comm*, MPI_Errhandler *errhandler*)	*Deprecated:* Use instead `MPI_Comm_set_errhandler()`. Sets the error handler for a communicator.
`MPI_Error_class`(int *errorcode*, int **errorclass*)	Converts an error code into an error class.
`MPI_Error_string`(int *errorcode*, char **string*, int **resultlen*)	Returns a string for a given error code.
`MPI_Finalize`( )	Terminates MPI execution environment.
`MPI_Finalized`(int **flag*)	Checks whether `MPI_Finalize()` has completed.
`MPI_Free_mem`(void **base*)	Frees memory that has been allocated using `MPI_Alloc_mem`.
`MPI_Gather`(void **sendbuf*, int **sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*)	Gathers values from a group of processes.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Gatherv(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int **recvcounts*, int **displs*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*)	Gathers into specified locations from all processes in a group. Each process may contribute a different amount of data.
MPI_Get(void **origin_addr*, int *origin_count*, MPI_Datatype *origin_datatype*, int *target_rank*, MPI_Aint *target_disp*, int *target_count*, MPI_Datatype *target_datatype*, MPI_Win *win*)	Copies data from the target memory to the origin.
MPI_Get_address(void **location*, MPI_Aint **address*)	Gets the address of a location in memory.
MPI_Get_count(MPI_Status **status*, MPI_Datatype *datatype*, int **count*)	Gets the number of top-level elements received.
MPI_Get_elements(MPI_Status **status*, MPI_Datatype *datatype*, int **count*)	Returns the number of basic elements in a data type.
MPI_Get_processor_name(char **name*, int **resultlen*)	Gets the name of the processor.
MPI_Get_version(int **version*, int **subversion*)	Returns the version of the standard corresponding to the current implementation.
MPI_Graph_create(MPI_Comm *comm_old*, int *nnodes*, int **index*, int **edges*, int *reorder*, MPI_Comm **comm_graph*)	Makes a new communicator to which graph topology information has been attached.
MPI_Graph_get(MPI_Comm *comm*, int *maxindex*, int *maxedges*, int **index*, int **edges*)	Retrieves graph topology information associated with a communicator.
MPI_Graph_map(MPI_Comm *comm*, int *nnodes*, int **index*, int **edges*, int **newrank*)	Maps process to graph topology information.
MPI_Graph_neighbors(MPI_Comm *comm*, int *rank*, int *maxneighbors*, int **neighbors*)	Returns the neighbors of a node associated with a graph topology.
MPI_Graph_neighbors_count(MPI_Comm *comm*, int *rank*, int **nneighbors*)	Returns the number of neighbors of a node associated with a graph topology.
MPI_Graphdims_get(MPI_Comm *comm*, int **nnodes*, int **nedges*)	Retrieves graph topology information associated with a communicator.
MPI_Grequest_complete(MPI_Request *request*)	Reports that a generalized request is complete.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Grequest_start(MPI_Grequest_query_function **query_fn*, MPI_Grequest_free_function **free_fn*, MPI_Grequest_cancel_function **cancel_fn*, void **extra_state*, MPI_Request **request*)	Starts a generalized request and returns a handle to it.
MPI_Group_c2f(MPI_Group *group*)	Translates a C handle into a Fortran handle.
MPI_Group_compare(MPI_Group *group1*, MPI_Group *group2*, int **result*)	Compares two groups.
MPI_Group_difference(MPI_Group *group1*, MPI_Group *group2*, MPI_Group **group_out*)	Makes a group from the difference of two groups.
MPI_Group_excl(MPI_Group *group*, int *n*, int **ranks*, MPI_Group **newgroup*)	Produces a group by reordering an existing group and taking only unlisted members.
MPI_Group_f2c(MPI_Fint *group*)	Translates a Fortran handle into a C handle.
MPI_Group_free(MPI_Group **group*)	Frees a group.
MPI_Group_incl(MPI_Group *group*, int *n*, int **ranks*, MPI_Group **group_out*)	Produces a group by reordering an existing group and taking only listed members.
MPI_Group_intersection(MPI_Group *group1*, MPI_Group *group2*, MPI_Group **group_out*)	Produces a group at the intersection of two existing groups.
MPI_Group_range_excl(MPI_Group *group*, int *n*, int *ranges*[ ][3], MPI_Group **newgroup*)	Produces a group by excluding ranges of processes from an existing group.
MPI_Group_range_incl(MPI_Group *group*, int *n*, int *ranges*[ ][3], MPI_Group **newgroup*)	Creates a new group from ranges of ranks in an existing group.
MPI_Group_rank(MPI_Group *group*, int **rank*)	Returns the rank of this process in the given group.
MPI_Group_size(MPI_Group *group*, int **size*)	Returns the size of a group.
MPI_Group_translate_ranks(MPI_Group *group1*, int *n*, int **ranks1*, MPI_Group *group2*, int **ranks2*)	Translates the ranks of processes in one group to those in another group.
MPI_Group_union(MPI_Group *group1*, MPI_Group *group2*, MPI_Group **group_out*)	Produces a group by combining two groups.
MPI_Ibsend(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Starts a nonblocking buffered send.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Info_c2f(MPI_Info *info*)	Translates a C handle into a Fortran handle.
MPI_Info_create(MPI_Info **info*)	Creates a new info object.
MPI_Info_delete(MPI_Info **info*, char **key*, char **value*)	Deletes a key / value pair from *info*.
MPI_Info_dup(MPI_Info *info*, MPI_Info **newinfo*)	Duplicates an info object.
MPI_Info_f2c(MPI_Fint *info*)	Translates a Fortran handle into a C handle.
MPI_Info_free(MPI_Info **info*)	Frees *info* and sets it to `MPI_INFO_NULL`.
MPI_Info_get(MPI_Info **info*, char **key*, char **value*)	Retrieves key value for an *info* object.
MPI_Info_get_nkeys(MPI_Info *info*, int **nkeys*)	Returns the number of currently defined keys in info.
MPI_Info_get_nthkey(MPI_Info *info*, int *n*, char **key*)	Returns the *n*th defined key in *info*.
MPI_Info_get_valuelen(MPI_Info *info*, char **key*, int **valuelen*, int **flag*)	Retrieves the length of the key value associated with an *info* object.
MPI_Info_set(MPI_Info **info*, char **key*, char **value*)	Adds a key / value pair to *info*.
MPI_Init(int **argc*, char ****argv*)	Initializes the MPI execution environment.
MPI_Initialized(int **flag*)	Indicates whether `MPI_Init()` has been called.
MPI_Intercomm_create(MPI_Comm *local_comm*, int *local_leader*, MPI_Comm *peer_comm*, int *remote_leader*, int *tag*, MPI_Comm **newintercomm*)	Creates an intercommunicator.
MPI_Intercomm_merge(MPI_Comm *intercomm*, int *high*, MPI_Comm **newintracomm*	Creates an intracommunicator from an intercommunicator.
MPI_Iprobe(int *source*, int *tag*, MPI_Comm *comm*, int **flag*, MPI_Status **status*)	Nonblocking test for a message.
MPI_Irecv(void **buf*, int *count*, MPI_Datatype *datatype*, int *source*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Begins a nonblocking receive.
MPI_Irsend(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Begins a nonblocking ready send.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
`MPI_Isend`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Begins a nonblocking send.
`MPI_Issend`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Begins a nonblocking synchronous send.
`MPI_Keyval_create`(MPI_Copy_function **copy_fn*, MPI_Delete_function **delete_fn*, int **keyval*, void **extra_state*)	*Deprecated:* Use instead `MPI_Comm_create_keyval()`. Generates a new attribute key.
`MPI_Keyval_free`(int **keyval*)	*Deprecated:* Use instead `MPI_Comm_free_keyval()`. Frees attribute key for communicator cache attribute.
`MPI_Op_c2f`(MPI_Op *op*)	Translates a C handle into a Fortran handle.
`MPI_Op_create`(MPI_User_function **function*, int *commute*, MPI_Op **op*)	Creates a user-defined combination function handle.
`MPI_Op_f2c`(MPI_Fint *op*)	Translates a Fortran handle into a C handle.
`MPI_Op_free`(MPI_Op **op*)	Frees a user-defined combination function handle.
`MPI_Open_port`(MPI_Info *info*, char **port_name*)	Establishes a network address for a server to accept connections from clients.
`MPI_Pack`(void **inbuf*, int *incount*, MPI_Datatype *datatype*, void **outbuf*, int *outsize*, int **position*, MPI_Comm *comm*)	Packs data of a given data type into contiguous memory.
`MPI_Pack_size`(int *incount*, MPI_Datatype *datatype*, MPI_Comm *comm*, int **size*)	Returns the upper bound on the amount of space needed to pack a message.
`MPI_Pcontrol`(int *level*, ...)	Controls profiling.
`MPI_Probe`(int *source*, int *tag*, MPI_Comm *comm*, MPI_Status **status*)	Blocking test for a message.
`MPI_Put`(void **origin_addr*, int *origin_count*, MPI_Datatype *origin_datatype*, int *target_rank*, MPI_Aint *target_disp*, int *target_count*, MPI_Datatype *target_datatype*, MPI_Win *win*)	Copies data from the origin memory to the target.
`MPI_Query_thread`(int **provided*)	Returns the current level of thread support.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Recv(void **buf*, int *count*, MPI_Datatype *datatype*, int *source*, int *tag*, MPI_Comm *comm*, MPI_Status **status*)	Performs a standard receive.
MPI_Recv_init(void **buf*, int *count*, MPI_Datatype *datatype*, int *source*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Builds a persistent receive request handle.
MPI_Reduce(void **sendbuf*, void **recvbuf*, int *count*, MPI_Datatype *datatype*, MPI_Op *op*, int *root*, MPI_Comm *comm*)	Reduces values on all processes to a single value.
MPI_Reduce_scatter(void **sendbuf*, void **recvbuf*, int **recvcounts*, MPI_Datatype *datatype*, MPI_Op *op*, MPI_Comm *comm*)	Combines values and scatters the results.
MPI_Request_c2f(MPI_Request *request*)	Translates a C handle into a Fortran handle.
MPI_Request_f2c(MPI_Fint *request*)	Translates a Fortran handle into a C handle.
MPI_Request_free(MPI_Request **request*)	Frees a communication request object.
MPI_Request_get_status(MPI_Request *request*, int **flag*, MPI_Status **status*)	Accesses information associated with a request without freeing the request.
MPI_Rsend(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*)	Performs a ready send.
MPI_Rsend_init(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Builds a persistent ready send request handle.
MPI_Scan(void **sendbuf*, void **recvbuf*, int *count*, MPI_Datatype *datatype*, MPI_Op *op*, MPI_Comm *comm*)	Computes the scan (partial reductions) of data on a collection of processes.
MPI_Scatter(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*)	Sends data from one job to all other processes in a group.
MPI_Scatterv(void **sendbuf*, int **sendcounts*, int **displs*, MPI_Datatype *sendtype*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*)	Scatters a buffer in parts to all processes in a group.
MPI_Send(int **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*)	Performs a standard send.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
`MPI_Send_init`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Builds a persistent send request handle.
`MPI_Sendrecv`(void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, int *dest*, int *sendtag*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, int *source*, int *recvtag*, MPI_Comm *comm*, MPI_Status **status*)	Sends and receives two messages at the same time.
`MPI_Sendrecv_replace`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *sendtag*, int *source*, int *recvtag*, MPI_Comm *comm*, MPI_Status **status*)	Sends and receives using a single buffer.
`MPI_Ssend`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*)	Performs a synchronous send.
`MPI_Ssend_init`(void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, MPI_Request **request*)	Builds a persistent synchronous send request handle.
`MPI_Start`(MPI_Request **request*)	Initiates a communication using a persistent request handle.
`MPI_Startall`(int *count*, MPI_Request *array_of_requests*[ ])	Starts a collection of requests.
`MPI_Status_c2f`(MPI_Status **c_status*, MPI_Fint **f_status*)	Translates a C status into a Fortran status.
`MPI_Status_f2c`(MPI_Fint **f_status*, MPI_Status **c_status*)	Translates a Fortran status into a C status.
`MPI_Status_set_cancelled`(MPI_Status **status*, int *flag*)	Sets *status* to indicate a request has been cancelled.
`MPI_Status_set_elements`(MPI_Status **status*, MPI_Datatype *datatype*, int *count*)	Modifies opaque part of *status* to enable `MPI_Get_elements()` to return *count*.
`MPI_Test`(MPI_Request **request*, int **flag*, MPI_Status **status*)	Tests for the completion of a send or receive.
`MPI_Test_cancelled`(MPI_Status **status*, int **flag*)	Tests whether a request was cancelled.
`MPI_Testall`(int *count*, MPI_Request *array_of_requests*, int **flag*, MPI_Status **array_of_statuses*)	Tests for the completion of all of the given communications.
`MPI_Testany`(int *count*, MPI_Request *array_of_requests*[ ], int **index*, int **flag*, MPI_Status *status*)	Tests for completion of any of the given communications.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Testsome(int *incount*, MPI_Request *array_of_requests*[ ], int **outcount*, int **array_of_indices*, MPI_Status **array_of_statuses*)	Tests for some given communications to complete.
MPI_Topo_test(MPI_Comm *comm*, int **top_type*)	Determines the type of topology (if any) associated with a communicator.
MPI_Type_c2f(MPI_Datatype *datatype*)	Translates a C handle into a Fortran handle.
MPI_Type_commit(MPI_Datatype **datatype*)	Commits a data type.
MPI_Type_contiguous(int *count*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates a contiguous data type.
MPI_Type_create_darray(int *size*, int *rank*, int *ndims*, int *array_of_gsizes*[ ], int *array_of_distribs*[ ], int *array_of_dargs*[ ], int *array_of_psizes*[ ], int *order*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates an array of data types.
MPI_Type_create_hindexed(int *count*, int *array_of_blocklengths*, MPI_Aint *array_of_displacements*[ ], MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates an indexed data type with offsets in bytes.
MPI_Type_create_hvector(int *count*, int *blocklength*, MPI_Aint *stride*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates a vector (strided) data type with offset in bytes.
MPI_Type_create_indexed_block(int *count*, int *blocklength*, int *array_of_displacements*[ ], MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates an indexed block.
MPI_Type_create_keyval(MPI_Type_copy_attr_function **type_copy_attr_fn*, MPI_Type_delete_attr_function **type_delete_attr_fn*, int **type_keyval*, void **extra_state*)	Generates a new attribute key.
MPI_Type_create_resized(MPI_Datatype *oldtype*, MPI_Aint *lb*, MPI_Aint *extent*, MPI_Datatype **newtype*)	Returns a new data type with new extent and upper and lower bounds.
MPI_Type_create_struct(int *count*, int *array_of_blocklengths*[ ], MPI_Aint *array_of_displacements*[ ], MPI_Datatype *array_of_types*[ ], MPI_Datatype **newtype*)	Creates a `struct` data type.
MPI_Type_create_subarray(int *ndims*, int *array_of_sizes*[ ], int *array_of_subsizes*[ ], int *array_of_starts*[ ], int *order*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates a data type describing a subarray of an array.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
`MPI_Type_delete_attr`(MPI_Datatype *type*, int *type_keyval*)	Deletes attribute value associated with a key.
`MPI_Type_dup`(MPI_Datatype *type*, MPI_Datatype **newtype*)	Duplicates a data type with associated key values.
`MPI_Type_extent`(MPI_Datatype *datatype*, MPI_Aint **extent*)	*Deprecated:* Use instead `MPI_Type_get_extent()`. Returns the extent of a data type, the difference between the upper and lower bounds of the data type.
`MPI_Type_f2c`(MPI_Fint *datatype*)	Translates a Fortran handle into a C handle.
`MPI_Type_free`(MPI_Datatype **datatype*)	Frees a data type.
`MPI_Type_free_keyval`(int **type_keyval*)	
`MPI_Type_get_attr`(MPI_Datatype *type*, int *type_keyval*, void **attribute_val*, int **flag*)	Returns the attribute associated with a data type.
`MPI_Type_get_contents`(MPI_Datatype *datatype*, int *max_integers*, int *max_addresses*, int *max_datatypes*, int *array_of_integers*[ ], MPI_Aint *array_of_addresses*[ ], MPI_Datatype *array_of_datatypes*[ ])	Returns information about arguments used in creation of a data type.
`MPI_Type_get_envelope`(MPI_Datatype *datatype*, int **num_integers*, int **num_addresses*, int **num_datatypes*, int **combiner*)	Returns information about input arguments associated with a data type.
`MPI_Type_get_extent`(MPI_Datatype *datatype*, MPI_Aint **lb*, MPI_Aint **extent*)	Returns the lower bound and extent of a data type.
`MPI_Type_get_name`(MPI_Datatype *type*, char **type_name*, int **resultlen*)	Gets the name of a data type.
`MPI_Type_get_true_extent`(MPI_Datatype *datatype*, MPI_Aint **true_lb*, MPI_Aint **true_extent*)	Returns the true lower bound and extent of a data type's corresponding type map, ignoring `MPI_UB` and `MPI_LB` markers.
`MPI_Type_hindexed`(int *count*, int **array_of_blocklengths*, MPI_Aint **array_of_displacements*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	*Deprecated:* Use instead `MPI_Type_create_hindexed()`. Creates an indexed data type with offsets in bytes.
`MPI_Type_hvector`(int *count*, int *blocklength*, MPI_Aint *stride*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	*Deprecated:* Use instead `MPI_Type_create_hvector()`. Creates a vector (strided) data type with offset in bytes.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
`MPI_Type_indexed`(int *count*, int **array_of_blocklengths*, int **array_of_displacements*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates an indexed data type.
`MPI_Type_lb`(MPI_Datatype *datatype*, MPI_Aint **displacement*)	*Deprecated:* Use instead `MPI_Type_get_extent()`. Returns the lower bound of a data type.
`MPI_Type_set_attr`(MPI_Datatype *type*, int *type_keyval*, void **attribute_val*)	Stores attribute value associated with a key.
`MPI_Type_set_name`(MPI_Comm *comm*, char **type_name*)	Sets the name of a data type.
`MPI_Type_size`(MPI_Datatype *datatype*, int **size*)	Returns the number of bytes occupied by entries in the data type.
`MPI_Type_struct`(int *count*, int **array_of_blocklengths*, MPI_Aint **array_of_displacements*, MPI_Datatype **array_of_types*, MPI_Datatype **newtype*)	*Deprecated:* Use instead `MPI_Type_create_struct()`. Creates a `struct` data type.
`MPI_Type_ub`(MPI_Datatype *datatype*, MPI_Aint **displacement*)	*Deprecated:* Use instead `MPI_Type_get_extent()`. Returns the upper bound of a data type.
`MPI_Type_vector`(int *count*, int *blocklength*, int *stride*, MPI_Datatype *oldtype*, MPI_Datatype **newtype*)	Creates a vector (strided) data type.
`MPI_Unpack`(void **inbuf*, int *insize*, int **position*, void **outbuf*, int *outcount*, MPI_Datatype *datatype*, MPI_Comm *comm*)	Unpacks a data type into contiguous memory.
`MPI_Wait`(MPI_Request **request*, MPI_Status **status*)	Waits for an MPI send or receive to complete.
`MPI_Waitall`(int *count*, MPI_Request *array_of_requests*[ ], MPI_Status *array_of_statuses*[ ])	Waits for all of the given communications to complete.
`MPI_Waitany`(int *count*, MPI_Request *array_of_requests*[ ], int **index*, MPI_Status **status*)	Waits for any of the given communications to complete.
`MPI_Waitsome`(int *incount*, MPI_Request *array_of_requests*[ ], int **outcount*, int *array_of_indices*[ ], MPI_Status *array_of_statuses*[ ])	Waits for some given communications to complete.
`MPI_Win_c2f`(MPI_Win *win*)	Translates a C handle into a Fortran handle.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
MPI_Win_create(void **base*, MPI_Aint *size*, int *disp_unit*, MPI_Info *info*, MPI_Comm *comm*, MPI_Win **win*)	Opens a communication window in memory.
MPI_Win_create_errhandler(MPI_Win_errhandler_fn **function*, MPI_Errhandler **errhandler*)	Creates an error handler that can be attached to windows.
MPI_Win_create_keyval(MPI_Win_copy_attr_function **win_copy_attr_fn*, MPI_Win_delete_attr_function **win_delete_attr_fn*, int **win_keyval*, void **extra_state*)	Creates a caching attribute that can be associated with a window.
MPI_Win_delete_attr(MPI_Win *win*, int *win_keyval*)	Deletes the attribute created with `MPI_Win_create_keyval`.
MPI_Win_f2c(MPI_Fint *win*)	Translates a Fortran handle into a C handle.
MPI_Win_fence(int *assert*, MPI_Win *win*)	Synchronizes RMA calls on a window.
MPI_Win_free(MPI_Win **win*)	Frees the window object and returns a null handle.
MPI_Win_free_keyval(int **win_keyval*)	Releases a window attribute.
MPI_Win_get_attr(MPI_Win *win*, int *win_keyval*, void **attribute_val*, int **flag*)	Obtain the value of a window attribute.
MPI_Win_get_errhandler(MPI_Win *win*, MPI_Errhandler **errhandler*)	Retrieves the error handler currently associated with a window.
MPI_Win_get_group(MPI_Win *win*, MPI_Group **group*)	Returns a duplicate of the group of the communicator used to create the window.
MPI_Win_get_name(MPI_Win *win*, char **win_name*, int **resultlen*)	Returns the last name associated with a window object.
MPI_Win_lock(int *lock_type*, int *rank*, int *assert*, MPI_Win *win*)	Starts an RMA access epoch, during which only the window at the process with the specified rank can be accessed.
MPI_Win_set_attr(MPI_Win *win*, int *win_keyval*, void **attribute_val*)	Associates an attribute with a window.
MPI_Win_set_errhandler(MPI_Win *win*, MPI_Errhandler *errhandler*)	Attaches a new error handler to a window.
MPI_Win_set_name(MPI_Win *win*, char **win_name*)	Assigns a name to a window.

TABLE A-1 Sun MPI Routines *(Continued)*

Routine and C Syntax	Description
`MPI_Win_unlock`(int *rank*, MPI_Win *win*)	Completes an RMA access epoch started by a call to `MPI_Win_lock()`.
double **`MPI_Wtick`**( )	Returns the resolution of `MPI_Wtime()`.
double **`MPI_Wtime`**( )	Returns an elapsed time on the calling processor.

Sun MPI I/O Routines

TABLE A-2 on page 82 lists the Sun MPI I/O routines in alphabetical order. The following sections list the routines by functional category.

File Manipulation

Collective coordination	Noncollective coordination
`MPI_File_open()`	`MPI_File_delete()`
`MPI_File_close()`	`MPI_File_get_size()`
`MPI_File_set_size()`	`MPI_File_get_group()`
`MPI_File_preallocate()`	`MPI_File_get_amode()`

File Info

Noncollective coordination	Collective coordination
`MPI_File_get_info()`	`MPI_File_set_info()`

Data access

Data Access With Explicit Offsets

Synchronism	Noncollective coordination	Collective coordination

Blocking	`MPI_File_read_at()`
	`MPI_File_write_at()`
		`MPI_File_read_at_all()`
		`MPI_File_write_at_all()`

Nonblocking or split collective	`MPI_File_iread_at()`	`MPI_File_read_at_all_begin()`
		`MPI_File_read_at_all_end()`

	`MPI_File_iwrite_at()`	`MPI_File_write_at_all_begin()`
		`MPI_File_write_at_all_end()`

Data Access With Individual File Pointers

Synchronism	Noncollective coordination	Collective coordination

Blocking	`MPI_File_read()`
	`MPI_File_write()`
		`MPI_File_read_all()`
		`MPI_File_write_all()`

Nonblocking or split collective	`MPI_File_iread()`	`MPI_File_read_all_begin()`
		`MPI_File_read_all_end()`

	`MPI_File_iwrite()`	`MPI_File_write_all_begin()`
		`MPI_File_write_all_end()`

Data Access With Shared File Pointers

Synchronism	Noncollective coordination	Collective coordination

Blocking	`MPI_File_read_shared()`
	`MPI_File_write_shared()`
		`MPI_File_read_ordered()`
		`MPI_File_write_ordered()`
		`MPI_File_seek_shared()`
		`MPI_File_get_position_shared()`

Nonblocking or split collective	`MPI_File_iread_shared()`	`MPI_File_read_ordered_begin()`
		`MPI_File_read_ordered_end()`

	`MPI_File_iwrite_shared()`	`MPI_File_write_ordered_begin()`
		`MPI_File_write_ordered_end()`

Pointer Manipulation

```
MPI_File_seek()
MPI_File_get_position()
MPI_File_get_byte_offset()
```

File Interoperability

```
MPI_Register_datarep()
MPI_File_get_type_extent()
```

File Consistency and Semantics

```
MPI_File_set_atomicity()
MPI_File_get_atomicity()
MPI_File_sync()
```

Handle Translation

```
MPI_File_f2c()
MPI_File_c2f()
```

MPI I/O Routines: Alphabetical Listing

TABLE A-2 Sun MPI I/O Routines

Routine and C Syntax	Description
MPI_File_c2f(MPI_File *file*)	Translates a C handle into a Fortran handle.
MPI_File_close(MPI_File **fh*)	Closes a file (collective).
MPI_File_create_errhandler(MPI_File_errhandler_fn **function*, MPI_Errhandler **errhandler*)	Creates an MPI-style error handler that can be attached to a file.
MPI_File_delete(char **filename*, MPI_Info *info*)	Deletes a file.
MPI_File_f2c(MPI_File *file*)	Translates a Fortran handle into a C handle.
MPI_File_get_amode(MPI_File *fh*, int **amode*)	Returns mode associated with open file.
MPI_File_get_atomicity(MPI_File *fh*, int **flag*)	Returns current consistency semantics for data-access operations.
MPI_File_get_byte_offset(MPI_File *fh*, MPI_Offset *offset*, MPI_Offset **disp*)	Converts a view-relative offset into an absolute byte position.
MPI_File_get_errhandler(MPI_Comm *file*, MPI_Errhandler **errhandler*)	Gets the error handler for a file.
MPI_File_get_group(MPI_File *fh*, MPI_Group **group*)	Returns the process group of file.
MPI_File_get_info(MPI_File *fh*, MPI_Info **info_used*)	Returns a new info object containing hints.
MPI_File_get_position(MPI_File *fh*, MPI_Offset **offset*)	Returns current position of individual file pointer.
MPI_File_get_position_shared(MPI_File *fh*, MPI_Offset **offset*)	Returns current position of the shared file pointer (collective).
MPI_File_get_size(MPI_File *fh*, MPI_Offset **size*)	Returns current size of file.
MPI_File_get_type_extent(MPI_File *fh*, MPI_Datatype *datatype*, MPI_Aint **extent*)	Returns the extent of the data type in a file.
MPI_File_get_view(MPI_File *fh*, MPI_Offset **disp*, MPI_Datatype **etype*, MPI_Datatype **filetype*, char **datarep*)	Returns process's view of data in file.

TABLE A-2 Sun MPI I/O Routines *(Continued)*

Routine and C Syntax	Description
MPI_File_iread(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Request **request*)	Reads a file starting at the location specified by the individual file pointer (nonblocking, noncollective).
MPI_File_iread_at(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Request **request*)	Reads a file at an explicitly specified offset (nonblocking, noncollective).
MPI_File_iread_shared(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Request **request*)	Reads a file using the shared file pointer (nonblocking, noncollective).
MPI_File_iwrite(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Request **request*)	Writes a file starting at the location specified by the individual file pointer (nonblocking, noncollective).
MPI_File_iwrite_at(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Request **request*)	Writes a file at an explicitly specified offset (nonblocking, noncollective).
MPI_File_iwrite_shared(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Request **request*)	Writes a file using the shared file pointer (nonblocking, noncollective).
MPI_File_open(MPI_Comm *comm*, char **filename*, init *amode*, MPI_Info *info*, MPI_File **fh*)	Opens a file (collective).
MPI_File_preallocate(MPI_File *fh*, MPI_Offset *size*)	Preallocates storage space for a portion of a file (collective).
MPI_File_read(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Reads a file starting at the location specified by the individual file pointer.
MPI_File_read_all(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Reads a file starting at the locations specified by individual file pointers (collective).
MPI_File_read_all_begin(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*)	Reads a file starting at the locations specified by individual file pointers; beginning part of a split collective routine (nonblocking).
MPI_File_read_all_end(MPI_File *fh*, void **buf*, MPI_Status **status*)	Reads a file starting at the locations specified by individual file pointers; ending part of a split collective routine (blocking).
MPI_File_read_at(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Reads a file at an explicitly specified offset.

TABLE A-2 Sun MPI I/O Routines *(Continued)*

Routine and C Syntax	Description
MPI_File_read_at_all(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Reads a file at explicitly specified offsets (collective).
MPI_File_read_at_all_begin(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*)	Reads a file at explicitly specified offsets; beginning part of a split collective routine (nonblocking).
MPI_File_read_at_all_end(MPI_File *fh*, void **buf*, MPI_Status **status*)	Reads a file at explicitly specified offsets; ending part of a split collective routine (blocking).
MPI_File_read_ordered(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Reads a file at a location specified by a shared file pointer (collective).
MPI_File_read_ordered_begin(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*)	Reads a file at a location specified by a shared file pointer; beginning part of a split collective routine (nonblocking).
MPI_File_read_ordered_end(MPI_File *fh*, void **buf*, MPI_Status **status*)	Reads a file at a location specified by a shared file pointer; ending part of a split collective routine (blocking).
MPI_File_read_shared(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Reads a file using the shared file pointer (blocking, noncollective).
MPI_File_seek(MPI_File *fh*, MPI_Offset *offset*, int *whence*)	Updates individual file pointers.
MPI_File_seek_shared(MPI_File *fh*, MPI_Offset *offset*, int *whence*)	Updates the global shared file pointer (collective).
MPI_File_set_atomicity(MPI_File *fh*, int *flag*)	Sets consistency semantics for data-access operations (collective).
MPI_File_set_errhandler(MPI_File *file*, MPI_Errhandler *errhandler*)	Sets the error handler for a file.
MPI_File_set_info(MPI_File *fh*, MPI_Info *info*)	Sets new values for hints (collective).
MPI_File_set_size(MPI_File *fh*, MPI_Offset *size*)	Resizes a file (collective).
MPI_File_set_view(MPI_File *fh*, MPI_Offset *disp*, MPI_Datatype *etype*, MPI_Datatype *filetype*, char **datarep*, MPI_Info *info*)	Changes process's view of data in file (collective).
MPI_File_sync(MPI_File *fh*)	Makes semantics consistent for data-access operations (collective).

TABLE A-2 Sun MPI I/O Routines *(Continued)*

Routine and C Syntax	Description
`MPI_File_write`(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Writes a file starting at the location specified by the individual file pointer.
`MPI_File_write_all`(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Writes a file starting at the locations specified by individual file pointers (collective).
`MPI_File_write_all_begin`(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*)	Writes a file starting at the locations specified by individual file pointers; beginning part of a split collective routine (nonblocking).
`MPI_File_write_all_end`(MPI_File *fh*, void **buf*, MPI_Status **status*)	Writes a file starting at the locations specified by individual file pointers; ending part of a split collective routine (blocking).
`MPI_File_write_at`(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Writes a file at an explicitly specified offset.
`MPI_File_write_at_all`(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Writes a file at explicitly specified offsets (collective).
`MPI_File_write_at_all_begin`(MPI_File *fh*, MPI_Offset *offset*, void **buf*, int *count*, MPI_Datatype *datatype*)	Writes a file at explicitly specified offsets; beginning part of a split collective routine (nonblocking).
`MPI_File_write_at_all_end`(MPI_File *fh*, void **buf*, MPI_Status **status*)	Writes a file at explicitly specified offsets; ending part of a split collective routine (blocking).
`MPI_File_write_ordered`(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Writes a file at a location specified by a shared file pointer (collective).
`MPI_File_write_ordered_begin`(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*)	Writes a file at a location specified by a shared file pointer; beginning part of a split collective routine (nonblocking).

TABLE A-2 Sun MPI I/O Routines *(Continued)*

Routine and C Syntax	Description
`MPI_File_write_ordered_end`(MPI_File *fh*, void **buf*, MPI_Status **status*)	Writes a file at a location specified by a shared file pointer; ending part of a split collective routine (blocking).
`MPI_File_write_shared`(MPI_File *fh*, void **buf*, int *count*, MPI_Datatype *datatype*, MPI_Status **status*)	Writes a file using the shared file pointer (blocking, noncollective).
`MPI_Register_datarep`(char **datarep*, MPI_Datarep_conversion_function **read_conversion_fn*, MPI_Datarep_conversion_function **write_conversion_fn*, MPI_Datarep_extent_function **dtype_file_extent_fn*, void **extra_state*)	Defines data representation.

APPENDIX **B**

Environment Variables

Many environment variables are available for fine-tuning your Sun MPI environment. All 39 Sun MPI environment variables are listed here with brief descriptions. The same descriptions are also available on the `MPI` man page. If you want to return to the default setting after setting a variable, simply unset it (using `unsetenv`). The effects of some of the variables are explained in more detail in the *Sun HPC ClusterTools Performance Guide*.

The environment variables are listed here in six groups:

- "Informational" on page 87
- "General Performance Tuning" on page 88
- "Tuning Memory for Point-to-Point Performance" on page 89
- "Numerics" on page 92
- "Tuning Rendezvous" on page 92
- "Miscellaneous" on page 93

Informational

MPI_PRINTENV

When set to 1, causes the environment variables and `hpc.conf` parameters associated with the MPI job to be printed out. The default value is 0.

MPI_QUIET

If set to 1, suppresses Sun MPI warning messages. The default value is 0.

MPI_SHOW_ERRORS

If set to 1, the MPI_ERRORS_RETURN error handler prints the error message and returns the error. The default value is 0.

MPI_SHOW_INTERFACES

When set to 1, 2, or 3, information regarding which interfaces are being used by an MPI application prints to stdout. Set MPI_SHOW_INTERFACES to 1 to print the selected internode interface. Set it to 2 to print all the interfaces and their rankings. Set it to 3 for verbose output. The default value, 0, does not print information to stdout.

General Performance Tuning

MPI_POLLALL

When set to 1, the default value, all connections are polled for receives, also known as *full polling*. When set to 0, only those connections are polled where receives are posted. Full polling helps drain system buffers and so lessen the chance of deadlock for “unsafe” codes. Well-written codes should set MPI_POLLALL to 0 for best performance.

MPI_PROCBIND

Binds each MPI process to its own processor. By default, MPI_PROCBIND is set to 0, which means processor binding is off. To turn processor binding on, set it to 1. The system administrator may enable or disable processor binding by setting the pbind parameter in the hpc.conf file to on or off. If this parameter is set, the MPI_PROCBIND environment variable is disabled. Performance can be enhanced with processor binding, but very poor performance will result if processor binding is used for multithreaded jobs or for more than one job at a time.

MPI_SPIN

Sets the spin policy. The default value is 0, which causes MPI processes to spin nonaggressively, allowing best performance when the load is at least as great as the number of CPUs. A value of 1 causes MPI processes to spin aggressively, leading to best performance if extra CPUs are available on each node to handle system daemons and other background activities.

Tuning Memory for Point-to-Point Performance

MPI_RSM_CPOOLSIZE

The requested size, in bytes, to be allocated per stripe for buffers for each remote-shared-memory connection. This value may be overridden when connections are established on the basis of the size of the segment allocated. The default value is 16384 bytes.

MPI_RSM_NUMPOSTBOX

The number of postboxes per stripe per remote-shared-memory connection. The maximum number of postboxes depends on the value of `rsm_maxsegsize`. The default is 15 postboxes.

MPI_RSM_PIPESIZE

The limit on the size (in bytes) of a message that can be sent over remote shared memory through the buffer list of one postbox per stripe. The default is 8192 bytes. This size also depends on the blocksize used for sending data. The maximum size is equal to min (cpoolsize/2, (10 * max(blk1sz,blk2sz))).

MPI_RSM_SBPOOLSIZE

If set, `MPI_RSM_SBPOOLSIZE` is the requested size in bytes of each RSM send buffer pool. An RSM send buffer pool is the pool of buffers on a node that a remote process would use to send to processes on the node. A multiple of 1024 must be used. If unset, the size of bufferpool is equal to cpoolsize times processes per node. The max value allowed is maxsegsize minus the memory used for postboxes.

MPI_RSM_SHORTMSGSIZE

The maximum size, in bytes, of a message that will be sent via remote shared memory without using buffers. The default value is 384 bytes. The upper limit is determined by the number of postboxes available.

MPI_RSM_STRONGPARTITION

If set to 1, the RSM protocol module will use strong partition to manage memory. Every connection will have a set of blocks in the buffer pool reserved for communication. Otherwise, the pool will shared by all the receivers. The default value is 0.

MPI_SHM_CPOOLSIZE

The amount of memory, in bytes, that can be allocated to each connection pool. When `MPI_SHM_SBPOOLSIZE` is not set, the default value is 24576 bytes. Otherwise, the default value is `MPI_SHM_SBPOOLSIZE`.

MPI_SHM_CYCLESIZE

The limit, in bytes, on the portion of a shared-memory message that will be sent via the buffer list of a single postbox during a cyclic transfer. The default value is 8192 bytes. A multiple of 1024 that is at most `MPI_SHM_CPOOLSIZE`/2 must be used.

MPI_SHM_CYCLESTART

Shared-memory transfers that are larger than `MPI_SHM_CYCLESTART` bytes will be cyclic. The default value is 24576 bytes.

MPI_SHM_NUMPOSTBOX

The number of postboxes dedicated to each shared-memory connection. The default value is 16.

MPI_SHM_PIPESIZE

The limit, in bytes, on the portion of a shared-memory message that will be sent via the buffer list of a single postbox during a pipeline transfer. The default value is 8192 bytes. The value must be a multiple of 1024.

MPI_SHM_PIPESTART

The size, in bytes, at which shared-memory transfers will start to be pipelined. The default value is 2048. Multiples of 1024 must be used.

MPI_SHM_SBPOOLSIZE

If set, `MPI_SHM_SBPOOLSIZE` is the size, in bytes, of the pool of shared-memory buffers dedicated to each sender. A multiple of 1024 must be used. If unset, then pools of shared-memory buffers are dedicated to connections rather than to senders.

MPI_SHM_SHORTMSGSIZE

The size (in bytes) of the section of a postbox that contains either data or a buffer list. The default value is 256 bytes.

Note – If `MPI_SHM_PIPESTART`, `MPI_SHM_PIPESIZE`, or `MPI_SHM_CYCLESIZE` is increased to a size larger than 31744 bytes, then `MPI_SHM_SHORTMSGSIZE` may also have to be increased. See the *Sun HPC ClusterTools 4 Performance Guide* for more information.

Numerics

MPI_CANONREDUCE

Prevents reduction operations from using any optimizations that take advantage of the physical location of processors. This may provide more consistent results in the case of floating-point addition, for example. However, the operation may take longer to complete. The default value is 0, meaning optimizations are allowed. To prevent optimizations, set the value to 1.

Tuning Rendezvous

MPI_EAGERONLY

When set to 1, the default, only the eager protocol is used. When set to 0, both eager and rendezvous protocols are used.

MPI_RSM_RENDVSIZE

Messages communicated by remote shared memory that are greater than this size will use the rendezvous protocol unless the environment variable `MPI_EAGERONLY` is set to 1. Default value is 16384 bytes.

MPI_SHM_RENDVSIZE

Messages communicated by shared memory that are greater than this size will use the rendezvous protocol unless the environment variable `MPI_EAGERONLY` is set. The default value is 24576 bytes.

MPI_TCP_RENDVSIZE

Messages communicated by TCP that contain data of this size and greater will use the rendezvous protocol unless the environment variable MPI_EAGERONLY is set. Default value is 49152 bytes.

Miscellaneous

MPI_COSCHED

Specifies the user's preference regarding use of the spind daemon for coscheduling. The value can be 0 (prefer no use) or 1 (prefer use). This preference may be overridden by the system administrator's policy. This policy is set in the hpc.conf file and can be 0 (forbid use), 1 (require use), or 2 (no policy). If no policy is set and no user preference is specified, coscheduling is not used.

Note – If no user preference is specified, the value 2 will be shown when environment variables are printed with MPI_PRINTENV.

MPI_FLOWCONTROL

Limits the number of unexpected messages that can be queued from a particular connection. Once this quantity of unexpected messages has been received, polling the connection for incoming messages stops. The default value, 0, indicates that no limit is set. To limit flow, set the value to some integer greater than zero.

MPI_FULLCONNINIT

Ensures that all connections are established during initialization. By default, connections are established lazily. However, you can override this default by setting the environment variable MPI_FULLCONNINIT to 1, forcing full-connection initialization mode. The default value is 0.

MPI_MAXFHANDLES

The maximum number of Fortran handles for objects other than requests. `MPI_MAXFHANDLES` specifies the upper limit on the number of concurrently allocated Fortran handles for MPI objects other than requests. This variable is ignored in the default 32-bit library. The default value is 1024. Users should take care to free MPI objects that are no longer in use. There is no limit on handle allocation for C codes.

MPI_MAXPROCS

This overrides the value specified by `maxprocs_default` in `hpc.conf`; it cannot exceed the value specified by `maxprocs_limit` in `hpc.conf`. If the value does exceed the `maxprocs_limit` value, the job will abort with an error when the program calls `MPI_Init`. Note that the upper limit of support for RSM communication is 2048 processes.

MPI_MAXREQHANDLES

The maximum number of Fortran request handles. `MPI_MAXREQHANDLES` specifies the upper limit on the number of concurrently allocated MPI request handles. Users must take care to free up request handles by properly completing requests. The default value is 1024. This variable is ignored in the default 32-bit library.

MPI_OPTCOLL

The MPI collectives are implemented using a variety of optimizations. Some of these optimizations can inhibit performance of point-to-point messages for "unsafe" programs. By default, this variable is 1, and optimized collectives are used. The optimizations can be turned off by setting the value to 0.

MPI_RSM_MAXSTRIPE

Specifies the maximum number of interfaces that can be used for striping data during communication over RSM. The value cannot be higher than the number of installed interfaces. The default is 4. The maximum is 64.

MPI_SHM_BCASTSIZE

On SMPs, the implementation of MPI_Bcast() for large messages is done using a double-buffering scheme. The size of each buffer (in bytes) is settable by using this environment variable. The default value is 32768 bytes.

MPI_SHM_GBPOOLSIZE

The amount of memory available, in bytes, to the general buffer pool for use by collective operations. The default value is 20971520 bytes.

MPI_SHM_REDUCESIZE

On SMPs, calling MPI_Reduce() causes all processors to participate in the reduce. Each processor will work on a piece of data equal to the MPI_SHM_REDUCESIZE setting. The default value is 256 bytes. Care must be taken when setting this variable because the system reserves MPI_SHM_REDUCESIZE * *np* * *np* memory to execute the reduce.

MPI_SPINDTIMEOUT

When coscheduling is enabled, limits the length of time (in milliseconds) a message will remain in the poll waiting for the spind daemon to return. If the timeout occurs before the daemon finds any messages, the process re-enters the polling loop. The default value is 1000 ms. A default can also be set by a system administrator in the hpc.conf file.

MPI_TCP_CONNLOOP

Sets the number of times MPI_TCP_CONNTIMEOUT occurs before signaling an error. The default value for this variable is 0, meaning that the program will abort on the first occurrence of MPI_TCP_CONNTIMEOUT.

MPI_TCP_CONNTIMEOUT

Sets the timeout value in seconds that is used for an `accept()` call. The default value for this variable is 600 seconds (10 minutes). This timeout can be triggered in both full- and lazy-connection initialization. After the timeout is reached, a warning message will be printed. If `MPI_TCP_CONNLOOP` is set to 0, then the first timeout will cause the program to abort.

MPI_TCP_SAFEGATHER

Allows use of a congestion-avoidance algorithm for `MPI_Gather()` and `MPI_Gatherv()` over TCP. By default, `MPI_TCP_SAFEGATHER` is set to 1, which means use of this algorithm is on. If you know that your underlying network can handle gathering large amounts of data on a single node, you may want to override this algorithm by setting `MPI_TCP_SAFEGATHER` to 0.

APPENDIX **C**

Troubleshooting

This appendix describes some common problem situations, resulting error messages, and suggestions for fixing the problems. Sun MPI error reporting, including I/O, follows the MPI-2 standard. By default, errors are reported in the form of standard error classes. These classes and their meanings are listed in TABLE C-1 on page 99 (for non-I/O MPI) and TABLE C-2 on page 101 (for MPI I/O) and are also available on the `MPI` man page.

Three predefined error handlers are available in Sun MPI:

- `MPI_ERRORS_RETURN` – The default, returns an error code if an error occurs.
- `MPI_ERRORS_ARE_FATAL` – I/O errors are fatal, and no error code is returned.
- `MPI_THROW_EXCEPTION` – A special error handler to be used only with C^{++}.

MPI Messages

You can make changes to and get information about the error handler using any of the following routines:

- `MPI_Comm_create_errhandler`
- `MPI_Comm_get_errhandler`
- `MPI_Comm_set_errhandler`

Messages resulting from an MPI program fall into two categories:

- *Error messages* – Error messages stem from within MPI. Usually an error message explains why your program cannot complete, and the program aborts.
- *Warning messages* – Warnings stem from the environment in which you are running your MPI program and are usually sent by `MPI_Init()`. They are not associated with an aborted program, that is, programs continue to run despite warning messages.

Error Messages

Sun MPI error messages use a standard format:

[*x y z*] `Error in` *function_name*: *errclass_string*:*intern*(*a*):*description*:*unixerrstring*

where

- [*x y z*] is the *process communication identifier*, and:
  - *x* is the job id (or jid).
  - *y* is the name of the communicator if a name exists; otherwise it is the address of the opaque object.
  - *z* is the rank of the process.

  The process communication identifier is present in every error message.
- *function_name* is the name of the associated MPI function. It is present in every error message.
- *errclass_string* is the string associated with the MPI error class. It is present in every error message.
- *intern* is an internal function. It is optional.
- *a* is a system call if one is the cause of the error. It is optional.
- *description* is a description of the error. It is optional.
- *unixerrstring* is the UNIX error string that describes system call *a*. It is optional.

Warning Messages

Sun MPI warning messages also use a standard format:

[*x y z*] `Warning` *message*

where *message* is a description of the error.

Standard Error Classes

Listed below are the error return classes you may encounter in your MPI programs. Error values may also be found in `mpi.h` (for C), `mpif.h` (for Fortran), and `mpi++.h` (for C^{++}).

TABLE C-1 Sun MPI Standard Error Classes

Error Code	Value	Meaning
`MPI_SUCCESS`	0	Successful return code.
`MPI_ERR_BUFFER`	1	Invalid buffer pointer.
`MPI_ERR_COUNT`	2	Invalid count argument.
`MPI_ERR_TYPE`	3	Invalid datatype argument.
`MPI_ERR_TAG`	4	Invalid tag argument.
`MPI_ERR_COMM`	5	Invalid communicator.
`MPI_ERR_RANK`	6	Invalid rank.
`MPI_ERR_ROOT`	7	Invalid root.
`MPI_ERR_GROUP`	8	Null group passed to function.
`MPI_ERR_OP`	9	Invalid operation.
`MPI_ERR_TOPOLOGY`	10	Invalid topology.
`MPI_ERR_DIMS`	11	Illegal dimension argument.
`MPI_ERR_ARG`	12	Invalid argument.
`MPI_ERR_UNKNOWN`	13	Unknown error.
`MPI_ERR_TRUNCATE`	14	Message truncated on receive.
`MPI_ERR_OTHER`	15	Other error; use `Error_string`.
`MPI_ERR_INTERN`	16	Internal error code.
`MPI_ERR_IN_STATUS`	17	Look in status for error value.
`MPI_ERR_PENDING`	18	Pending request.
`MPI_ERR_REQUEST`	19	Illegal `MPI_Request()` handle.
`MPI_ERR_KEYVAL`	36	Illegal key value.
`MPI_ERR_INFO`	37	Invalid info object.
`MPI_ERR_INFO_KEY`	38	Illegal info key.

TABLE C-1 Sun MPI Standard Error Classes *(Continued)*

Error Code	Value	Meaning
`MPI_ERR_INFO_NOKEY`	39	No such key.
`MPI_ERR_INFO_VALUE`	40	Illegal info value.
`MPI_ERR_TIMEDOUT`	41	Timed out.
`MPI_ERR_RESOURCES`	42	Out of resources.
`MPI_ERR_TRANSPORT`	43	Transport layer error.
`MPI_ERR_HANDSHAKE`	44	Error accepting/connecting.
`MPI_ERR_SPAWN`	45	Error spawning.
`MPI_ERR_WIN`	46	Invalid window.
`MPI_ERR_BASE`	47	Invalid base.
`MPI_ERR_SIZE`	48	Invalid size.
`MPI_ERR_DISP`	49	Invalid displacement.
`MPI_ERR_LOCKTYPE`	50	Invalid locktype.
`MPI_ERR_ASSERT`	51	Invalid assert.
`MPI_ERR_RMA_CONFLICT`	52	Conflicting accesses to window.
`MPI_ERR_RMA_SYNC`	53	Erroneous RMA synchronization.
`MPI_ERR_NO_MEM`	54	Memory exhausted.
`MPI_ERR_LASTCODE`	55	Last error code.

MPI I/O messages are listed separately, in TABLE C-2 on page 101.

MPI I/O Error Handling

Sun MPI I/O error reporting follows the MPI-2 standard. By default, errors are reported in the form of standard error codes (found in `/opt/SUNWhpc/include/mpi.h`). Error classes and their meanings are listed in TABLE C-2 on page 101. They can also be found in `mpif.h` (for Fortran) and `mpi++.h` (for C^{++}).

You can change the default error handler by specifying `MPI_FILE_NULL` as the file handle with the routine `MPI_File_set_errhandler()`, even if no file is currently open. Or, you can use the same routine to change a specific file's error handler.

TABLE C-2 Sun MPI I/O Error Classes

Error Class	Value	Meaning
`MPI_ERR_FILE`	20	Bad file handle.
`MPI_ERR_NOT_SAME`	21	Collective argument not identical on all processes.
`MPI_ERR_AMODE`	22	Unsupported `amode` passed to open.
`MPI_ERR_UNSUPPORTED_DATAREP`	23	Unsupported `datarep` passed to `MPI_File_set_view()`.
`MPI_ERR_UNSUPPORTED_OPERATION`	24	Unsupported operation, such as seeking on a file that supports only sequential access.
`MPI_ERR_NO_SUCH_FILE`	25	File (or directory) does not exist.
`MPI_ERR_FILE_EXISTS`	26	File exists.
`MPI_ERR_BAD_FILE`	27	Invalid file name (for example, path name too long).
`MPI_ERR_ACCESS`	28	Permission denied.
`MPI_ERR_NO_SPACE`	29	Not enough space.
`MPI_ERR_QUOTA`	30	Quota exceeded.
`MPI_ERR_READ_ONLY`	31	Read-only file system.
`MPI_ERR_FILE_IN_USE`	32	File operation could not be completed, as the file is currently open by some process.
`MPI_ERR_DUP_DATAREP`	33	Conversion functions could not be registered because a data representation identifier that was already defined was passed to `MPI_REGISTER_DATAREP`.
`MPI_ERR_CONVERSION`	34	An error occurred in a user-supplied data-conversion function.
`MPI_ERR_IO`	35	I/O error.
`MPI_ERR_INFO`	37	Invalid info object.
`MPI_ERR_INFO_KEY`	38	Illegal info key.

TABLE C-2 Sun MPI I/O Error Classes *(Continued)*

Error Class	Value	Meaning
`MPI_ERR_INFO_NOKEY`	39	No such key.
`MPI_ERR_INFO_VALUE`	40	Illegal info value.
`MPI_ERR_LASTCODE`	55	Last error code.

APPENDIX **D**

TNF Probes

Through Prism, you can use Trace Normal Form (TNF), an extensible system for event-based instrumentation, to aid in debugging and to analyze the performance of your Sun MPI programs. TNF is included with the Solaris operating environment. The TNF-instrumented libraries included with Sun MPI (see "Choosing a Library Path" on page 29) include probes for most of the MPI and MPI I/O routines, including some specific arguments. These probes are also categorized into specific groups, so that you can debug and analyze the performance of particular types of routines. For information about using Prism to take advantage of these probes, see the *Prism User's Guide* and the *Sun HPC ClusterTools 4 Performance Guide*.

This appendix includes all the probes, including their arguments and associated groups, both for MPI (Table D-1 on page 108) and for MPI I/O (Table D-2 on page 123). The following figure depicts the relationships among the various probe groups. (Some probes fall under both the `mpi_request` and the `mpi_pt2pt` groups.)

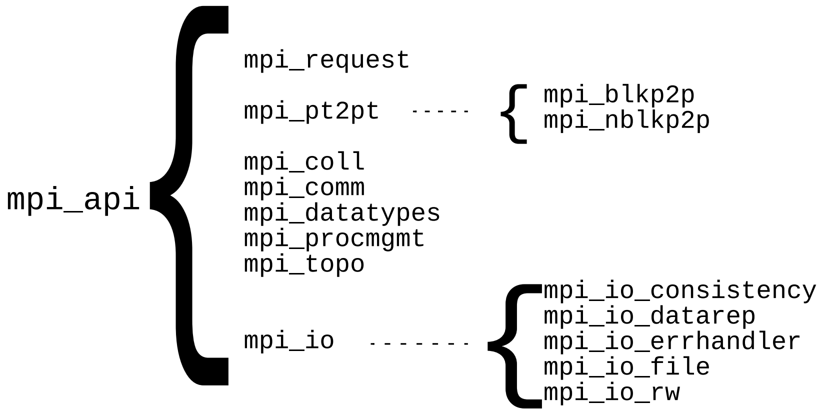


FIGURE D-1 TNF Probe Groups for Sun MPI, Including I/O

TNF Probes for MPI

Each MPI routine is associated with two TNF probes: one ending in `_start` and one ending in `_end`. Probes are also included for some specific arguments, most of which are defined in the MPI standard and described in the man pages included with Sun MPI. Four of the arguments, however, are not mentioned in the standard or man pages:

- `bytes` – The number of bytes sent or received by an MPI process. See the next section for more information about the `bytes` argument.
- `ctxt` – The *context id* is a number assigned to a particular communicator. The processes in a given communicator may be associated with only one context id. You can determine the context id associated with a communicator by using either the `MPI_Comm_set_name_end` or the `MPI_Comm_get_name_end` probe.
- `newctxt` – The context id associated with a communicator that is returned as a `newcomm` or `comm_out` argument.
- `request` – An integer that uniquely refers to a request object. For Fortran calls, this integer is equal to the request-handle argument.

The `bytes` Argument

The meaning of the `bytes` argument varies slightly depending on the situation. Following is some general information for different types of sends and receives, followed by some examples of how the `byte` argument works with them.

- *point-to-point blocking sends* – The `start` probes for routines that initiate point-to-point blocking sends report the number of bytes to be sent. These routines are:
  - `MPI_Bsend()`
  - `MPI_Rsend()`
  - `MPI_Send()`
  - `MPI_Ssend()`
  - `MPI_Sendrecv()`
  - `MPI_Sendrecv_replace()`
- *point-to-point nonblocking sends* – The `end` probes for routines that initiate point-to-point nonblocking sends report the number of bytes to be sent. These routines are:
  - `MPI_Bsend_init()`
  - `MPI_Ibsend()`
  - `MPI_Irsend()`
  - `MPI_Isend()`
  - `MPI_Issend()`

  - `MPI_Rsend_init()`
  - `MPI_Send_init()`
  - `MPI_Ssend_init()`
- *point-to-point receives* – The `end` probes for routines that terminate or could terminate point-to-point nonblocking sends report the number of bytes *actually* received. These routines are:
  - `MPI_Iprobe()`
  - `MPI_Probe()`
  - `MPI_Recv()`
  - `MPI_Test()`
  - `MPI_Testall()`
  - `MPI_Testany()`
  - `MPI_Testsome()`
  - `MPI_Wait()`
  - `MPI_Waitall()`
  - `MPI_Waitany()`
  - `MPI_Waitsome()`
  - `MPI_Sendrecv()`
  - `MPI_Sendrecv_replace()`
- *collectives* – The `start` probes for collective routines report the number of bytes to be sent from an MPI process and the number to be received at the process. Such byte counts are independent of the algorithm used. For example, the number of bytes sent from the root in a broadcast is given as the number of bytes in the broadcast message, regardless of whether the root sends this message multiple times as part of a binary-tree fan-out. These collective routines are:
  - `MPI_Allgather()`, `MPI_Allgatherv()`

    `sendbytes` – Number of bytes to be sent from this process.

    `recvbytes` – Total number of bytes to be received at any process from all processes.
  - `MPI_Allreduce()`, `MPI_Reduce()`, `MPI_Reduce_scatter()`

    `bytes` – Number of bytes on any process to be reduced.
  - `MPI_Alltoall()`, `MPI_Alltoallv()`

    `sendbytes` – Total number of bytes to be sent from this process to all processes.

    `recvbytes` – Total number of bytes to be received at this process from all processes.
  - `MPI_Bcast()`

    `bytes` – Number of bytes to be broadcast.

- `MPI_Gather()`, `MPI_Gatherv()`

  `sendbytes` – Root reports total number of bytes to be sent; other processes report 0.

  `recvbytes` – Root reports total number of bytes to be received; other processes report 0.

- `MPI_Scan`()

  `bytes` – Number of bytes contributed by any process.

- `MPI_Scatter()`, `MPI_Scatterv()`

  `sendbytes` – Root reports total number of bytes to be sent; other processes report 0.

  `recvbytes` – Number of bytes to be received at this process from the root.

- *pack and unpack* – The start probes for these routines report the number of bytes packed or unpacked. These routines are:
  - `MPI_Pack()`
  - `MPI_Unpack()`

Examples:

- `MPI_Send()`

  ```
  call MPI_Send(x,m,MPI_REAL8,...)
  ```

  Probe `mpi_send_start` reports that 8*`m` bytes are to be sent.

- `MPI_Recv()`

  ```
  call MPI_Recv(x,n,MPI_REAL8,...)
  ```

  Probe `mpi_recv_end` reports the number of bytes that were actually received, which must be at most 8*`n`.

- `MPI_Sendrecv()`

  ```
  call MPI_Sendrecv(x,m,MPI_REAL8,...,y,n,MPI_REAL8,...)
  ```

  Probe `mpi_sendrecv_start` reports that 8*`m` bytes are to be sent, and probe `mpi_sendrecv_end` reports the number of bytes that were actually received, which must be at most 8*`n`.

- `MPI_Irecv()`, `MPI_Wait()`

  ```
  integer req
  call MPI_Irecv(x,n,MPI_REAL8,...,req,...)
  call MPI_Wait(req,...)
  ```

Probe `mpi_wait_end` reports the number of bytes that were actually received, which must be at most 8*`n`.

- `MPI_Isend()`, `MPI_Irecv()`, `MPI_Wait()`

```
integer reqs(2)
call MPI_Isend(x,m,MPI_REAL8,...,reqs(1),...)
call MPI_Irecv(Y,N,MPI_REAL8,...,reqs(2),...)
call MPI_Waitany(2,reqs,...)
call MPI_Waitany(2,reqs,...)
```

Probe `mpi_isend_start` reports that 8*`m` bytes are to be sent. The `MPI_Waitany()` call that completes the receive will show the number of bytes that were actually received, which must be at most 8*`n`, in its `mpi_waitany_end` probe. The other `MPI_Waitany()` call, which completes the send, will report 0 bytes received.

- `MPI_Waitall()`

```
integer reqs(8)
call MPI_Isend(x1,m,MPI_REAL8,...,reqs(1),...)
call MPI_Isend(x2,m,MPI_REAL8,...,reqs(2),...)
call MPI_Isend(x3,m,MPI_REAL8,...,reqs(3),...)
call MPI_Isend(x4,m,MPI_REAL8,...,reqs(4),...)
call MPI_Irecv(x5,n,MPI_REAL8,...,reqs(5),...)
call MPI_Irecv(x6,n,MPI_REAL8,...,reqs(6),...)
call MPI_Irecv(x7,n,MPI_REAL8,...,reqs(7),...)
call MPI_Irecv(x8,n,MPI_REAL8,...,reqs(8),...)
call MPI_Waitall(8,reqs,..)
```

Probe `mpi_isend_start` reports that 8*`m` bytes are to be sent in each of the four `MPI_Isend()` cases. Probe `mpi_waitall_end` reports the number of bytes that were actually received, which must be at most 4*8*`n`.

Groups

Every TNF probe for MPI is associated with the `mpi_api` group, so choosing that group allows Prism to probe all the MPI routines for which probes exist (including the I/O routines). Additional groups exist to probe subsets of the MPI routines, as well. Some routines are associated with more than one group. The groups for MPI routine probes are:

- `mpi_api` – All the TNF probes for MPI routines
- `mpi_blkp2p` – Probes for blocking point-to-point routines

- `mpi_coll` – Probes for collective routines
- `mpi_comm` – Probes for communicator-related routines
- `mpi_datatypes` – Probes for data type–related routines
- `mpi_nblkp2p` – Probes for nonblocking point-to-point routines
- `mpi_procmgmt` – Probes for process-management routines
- `mpi_pt2pt` – Probes for all point-to-point routines (blocking and nonblocking)
- `mpi_request` – Probes for functions producing or acting on request(s)
- `mpi_topo` – Probes for topology-related routines
- `mpi-win` - Probes for windows-related routines

Probes for MPI (Non-I/O Routines)

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls

Probe	Argument(s)	Group(s), in Addition to `mpi_api`
`MPI_Accumulate_start`		`mpi_win`
`MPI_Accumulate_end`		`mpi_win`
`MPI_Address_start`		
`MPI_Address_end`		
`MPI_Allgather_start`	`sendbytes recvbytes ctxt`	`mpi_coll`
`MPI_Allgather_end`		`mpi_coll`
`MPI_Allgatherv_start`	`sendbytes recvbytes ctxt`	`mpi_coll`
`MPI_Allgatherv_end`		`mpi_coll`
`MPI_Alloc_mem_start`		`mpi_comm`
`MPI_Alloc_mem_end`		`mpi_comm`
`MPI_Allreduce_start`	`bytes ctxt`	`mpi_coll`
`MPI_Allreduce_end`		`mpi_coll`
`MPI_Alltoall_start`	`sendbytes recvbytes ctxt`	`mpi_coll`
`MPI_Alltoall_end`		`mpi_coll`
`MPI_Alltoallv_start`	`sendbytes recvbytes ctxt`	`mpi_coll`
`MPI_Alltoallv_end`		`mpi_coll`

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Attr_delete_start		
MPI_Attr_delete_end		
MPI_Attr_get_start		
MPI_Attr_get_end		
MPI_Attr_put_start		
MPI_Attr_put_end		
MPI_Barrier_start	ctxt	mpi_coll
MPI_Barrier_end		mpi_coll
MPI_Bcast_start	bytes root ctxt	mpi_coll
MPI_Bcast_end		mpi_coll
MPI_Bsend_start	bytes dest tag	mpi_pt2pt mpi_blkp2p
MPI_Bsend_end		mpi_pt2pt mpi_blkp2p
MPI_Bsend_init_start		mpi_pt2pt mpi_request
MPI_Bsend_init_end	bytes dest tag request	mpi_pt2pt mpi_request
MPI_Buffer_attach_start	buffer size	
MPI_Buffer_attach_end	buffer size	
MPI_Buffer_detach_start	buffer size	
MPI_Buffer_detach_end		
MPI_Cancel_start	request	mpi_request
MPI_Cancel_end		mpi_request
MPI_Cart_coords_start		mpi_topo
MPI_Cart_coords_end		mpi_topo
MPI_Cart_create_start		mpi_topo
MPI_Cart_create_end		mpi_topo
MPI_Cart_get_start		mpi_topo
MPI_Cart_get_end		mpi_topo
MPI_Cart_map_start		mpi_topo

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Cart_map_end		mpi_topo
MPI_Cart_rank_start		mpi_topo
MPI_Cart_rank_end		mpi_topo
MPI_Cart_shift_start		mpi_topo
MPI_Cart_shift_end		mpi_topo
MPI_Cart_sub_start		mpi_topo
MPI_Cart_sub_end		mpi_topo
MPI_Cartdim_get_start		mpi_topo
MPI_Cartdim_get_end		mpi_topo
MPI_Close_port_start	port_name	mpi_procmgmt
MPI_Close_port_end		mpi_procmgmt
MPI_Comm_accept_start	port_name root ctxt	mpi_procmgmt
MPI_Comm_accept_end	port_name root ctxt newctxt	mpi_procmgmt
MPI_Comm_compare_start		mpi_comm
MPI_Comm_compare_end		mpi_comm
MPI_Comm_connect_start	port_name root ctxt	mpi_procmgmt
MPI_Comm_connect_end	port_name root ctxt newctxt	mpi_procmgmt
MPI_Comm_create_start	ctxt group	mpi_comm
MPI_Comm_create_end	ctxt newctxt	mpi_comm
MPI_Comm_create_errhandler_start		
MPI_Comm_create_errhandler_end		
MPI_Comm_create_keyval_start		
MPI_Comm_create_keyval_end		
MPI_Comm_delete_attr_start		
MPI_Comm_delete_attr_end		
MPI_Comm_disconnect_start	ctxt	mpi_procmgmt
MPI_Comm_disconnect_end		mpi_procmgmt

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to `mpi_api`
MPI_Comm_dup_start	ctxt	mpi_comm
MPI_Comm_dup_end	ctxt newctxt	mpi_comm
MPI_Comm_free_start	ctxt	mpi_comm
MPI_Comm_free_end		mpi_comm
MPI_Comm_free_keyval_start		
MPI_Comm_free_keyval_end		
MPI_Comm_get_attr_start		
MPI_Comm_get_attr_end		
MPI_Comm_get_errhandler_start		
MPI_Comm_get_errhandler_end		
MPI_Comm_get_name_start		
MPI_Comm_get_name_end	ctxt comm_name	mpi_comm
MPI_Comm_group_start		
MPI_Comm_group_end		
MPI_Comm_remote_group_start		
MPI_Comm_remote_group_end		
MPI_Comm_set_attr_start		
MPI_Comm_set_attr_end		
MPI_Comm_set_errhandler_start		
MPI_Comm_set_errhandler_end		
MPI_Comm_set_name_start		
MPI_Comm_set_name_end	ctxt comm_name	mpi_comm
MPI_Comm_split_start	ctxt	mpi_comm
MPI_Comm_split_end	ctxt newctxt	mpi_comm
MPI_Comm_test_inter_start		
MPI_Comm_test_inter_end		
MPI_Dims_create_start		mpi_topo
MPI_Dims_create_end		mpi_topo
MPI_Errhandler_create_start		
MPI_Errhandler_create_end		

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Errhandler_free_start		
MPI_Errhandler_free_end		
MPI_Errhandler_get_start		
MPI_Errhandler_get_end		
MPI_Errhandler_set_start		
MPI_Errhandler_set_end		
MPI_Error_class_start		
MPI_Error_class_end		
MPI_Error_string_start		
MPI_Error_string_end		
MPI_Finalize_start		
MPI_Finalize_end		
MPI_Free_mem_start		
MPI_Free_mem_end		
MPI_Gather_start	sendbytes recvbytes root ctxt	mpi_coll
MPI_Gather_end		mpi_coll
MPI_Gatherv_start	sendbytes recvbytes root ctxt	mpi_coll
MPI_Gatherv_end		mpi_coll
MPI_Get_start		mpi_win
MPI_Get_end		mpi_win
MPI_Get_address_start		
MPI_Get_address_end		
MPI_Get_count_start		
MPI_Get_count_end		
MPI_Get_elements_start		
MPI_Get_elements_end		
MPI_Get_processor_name_start		

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Get_processor_name_end		
MPI_Get_version_start		
MPI_Get_version_end		
MPI_Graph_create_start		mpi_topo
MPI_Graph_create_end		mpi_topo
MPI_Graphdims_get_start		mpi_topo
MPI_Graphdims_get_end		mpi_topo
MPI_Graph_get_start		mpi_topo
MPI_Graph_get_end		mpi_topo
MPI_Graph_map_start		mpi_topo
MPI_Graph_map_end		mpi_topo
MPI_Graph_neighbors_start		mpi_topo
MPI_Graph_neighbors_end		mpi_topo
MPI_Graph_neighbors_count_start		mpi_topo
MPI_Graph_neighbors_count_end		mpi_topo
MPI_Grequest_complete_start	request	mpi_request
MPI_Grequest_complete_end		mpi_request
MPI_Grequest_start_start	request	mpi_request
MPI_Grequest_start_end		mpi_request
MPI_Group_compare_start		
MPI_Group_compare_end		
MPI_Group_difference_start		
MPI_Group_difference_end		
MPI_Group_excl_start		
MPI_Group_excl_end		
MPI_Group_free_start		
MPI_Group_free_end		
MPI_Group_incl_start		
MPI_Group_incl_end		
MPI_Group_intersection_start		

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to `mpi_api`
`MPI_Group_intersection_end`		
`MPI_Group_range_excl_start`		
`MPI_Group_range_excl_end`		
`MPI_Group_range_incl_start`		
`MPI_Group_range_incl_end`		
`MPI_Group_translate_ranks_start`		
`MPI_Group_translate_ranks_end`		
`MPI_Group_union_start`		
`MPI_Group_union_end`		
`MPI_Ibsend_start`		`mpi_pt2pt` `mpi_nblkp2p` `mpi_request`
`MPI_Ibsend_end`	`bytes dest tag done request`	`mpi_pt2pt` `mpi_nblkp2p` `mpi_request`
`MPI_Info_create_start`		
`MPI_Info_create_end`		
`MPI_Info_delete_start`		
`MPI_Info_delete_end`		
`MPI_Info_dup_start`		
`MPI_Info_dup_end`		
`MPI_Info_free_start`		
`MPI_Info_free_end`		
`MPI_Info_get_start`		
`MPI_Info_get_end`		
`MPI_Info_get_nkeys_start`		
`MPI_Info_get_nkeys_end`		
`MPI_Info_get_nthkey_start`		
`MPI_Info_get_nthkey_end`		
`MPI_Info_get_valuelen_start`		
`MPI_Info_get_valuelen_end`		

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to `mpi_api`
`MPI_Info_set_start`		
`MPI_Info_set_end`		
`MPI_Intercomm_create_start`		`mpi_comm`
`MPI_Intercomm_create_end`		`mpi_comm`
`MPI_Intercomm_merge_start`		`mpi_comm`
`MPI_Intercomm_merge_end`		`mpi_comm`
`MPI_Iprobe_start`	`source tag ctxt`	
`MPI_Iprobe_end`	`source tag ctxt flag`	
`MPI_Irecv_start`		`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Irecv_end`	`done request`	`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Irsend_start`		`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Irsend_end`	`bytes dest tag done request`	`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Isend_start`		`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Isend_end`	`bytes dest tag done request`	`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Issend_start`		`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Issend_end`	`bytes dest tag done request`	`mpi_pt2pt mpi_nblkp2p mpi_request`
`MPI_Keyval_create_start`		
`MPI_Keyval_create_end`		
`MPI_Keyval_free_start`		

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Keyval_free_end		
MPI_Op_create_start		
MPI_Op_create_end		
MPI_Open_port_start	port_name	mpi_procmgmt
MPI_Open_port_end	port_name	mpi_procmgmt
MPI_Op_free_start		
MPI_Op_free_end		
MPI_Pack_start	bytes	mpi_datatypes
MPI_Pack_end		mpi_datatypes
MPI_Pack_size_start	count datatype	mpi_datatypes
MPI_Pack_size_end	count datatype size	mpi_datatypes
MPI_Pcontrol_start		
MPI_Pcontrol_end		
MPI_Probe_start	source tag ctxt	
MPI_Probe_end	source tag ctxt	
MPI_Put_start		mpi_win
MPI_Put_end		mpi_win
MPI_Query_thread_start		
MPI_Query_thread_end		
MPI_Recv_start		mpi_pt2pt mpi_blkp2p
MPI_Recv_end	bytes source tag	mpi_pt2pt mpi_blkp2p
MPI_Recv_init_start		mpi_pt2pt mpi_request
MPI_Recv_init_end	request	mpi_pt2pt mpi_request
MPI_Reduce_start	bytes root ctxt	mpi_coll
MPI_Reduce_end		mpi_coll
MPI_Reduce_scatter_start	bytes ctxt	mpi_coll
MPI_Reduce_scatter_end		mpi_coll

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Request_free_start	request	mpi_request
MPI_Request_free_end		mpi_request
MPI_Rsend_start	bytes dest tag	mpi_pt2pt mpi_blkp2p
MPI_Rsend_end		mpi_pt2pt mpi_blkp2p
MPI_Rsend_init_start		mpi_pt2pt mpi_request
MPI_Rsend_init_end	bytes dest tag request	mpi_pt2pt mpi_request
MPI_Scan_start	bytes ctxt	mpi_coll
MPI_Scan_end		mpi_coll
MPI_Scatter_start	sendbytes recvbytes root ctxt	mpi_coll
MPI_Scatter_end		mpi_coll
MPI_Scatterv_start	sendbytes recvbytes root ctxt	mpi_coll
MPI_Scatterv_end		mpi_coll
MPI_Send_start	bytes dest tag	mpi_pt2pt mpi_blkp2p
MPI_Send_end		mpi_pt2pt mpi_blkp2p
MPI_Send_init_start		mpi_pt2pt mpi_request
MPI_Send_init_end	bytes dest tag request	mpi_pt2pt mpi_request
MPI_Sendrecv_start	bytes dest sendtag	mpi_pt2pt mpi_blkp2p
MPI_Sendrecv_end	bytes source recvtag	mpi_pt2pt mpi_blkp2p
MPI_Sendrecv_replace_start	bytes dest sendtag	mpi_pt2pt mpi_blkp2p
MPI_Sendrecv_replace_end	bytes source recvtag	mpi_pt2pt mpi_blkp2p

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to `mpi_api`
`MPI_Ssend_start`	`bytes dest tag`	`mpi_pt2pt` `mpi_blkp2p`
`MPI_Ssend_end`		`mpi_pt2pt` `mpi_blkp2p`
`MPI_Ssend_init_start`		`mpi_pt2pt` `mpi_request`
`MPI_Ssend_init_end`	`bytes dest tag request`	`mpi_pt2pt` `mpi_request`
`MPI_Start_start`	`request`	`mpi_pt2pt` `mpi_request`
`MPI_Start_end`		`mpi_pt2pt` `mpi_request`
`MPI_Startall_start`	`count`	`mpi_pt2pt` `mpi_request`
`MPI_Startall_end`		`mpi_pt2pt` `mpi_request`
`MPI_Status_set_cancelled_start`		
`MPI_Status_set_cancelled_end`		
`MPI_Status_set_elements_start`		
`MPI_Status_set_elements_end`		
`MPI_Test_start`	`request`	`mpi_request`
`MPI_Test_end`	`recvbytes source recvtag flag request`	`mpi_request`
`MPI_Testall_start`	`count`	`mpi_request`
`MPI_Testall_end`	`bytes count flag`	`mpi_request`
`MPI_Testany_start`	`count`	`mpi_request`
`MPI_Testany_end`	`bytes index flag`	`mpi_request`
`MPI_Test_cancelled_start`		`mpi_request`
`MPI_Test_cancelled_end`	`flag`	`mpi_request`
`MPI_Testsome_start`	`incount`	`mpi_request`
`MPI_Testsome_end`	`bytes outcount`	`mpi_request`

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to `mpi_api`
`MPI_Topo_test_start`		`mpi_topo`
`MPI_Topo_test_end`		`mpi_topo`
`MPI_Type_contiguous_start`		`mpi_datatypes`
`MPI_Type_contiguous_end`		`mpi_datatypes`
`MPI_Type_create_hindexed_start`		`mpi_datatypes`
`MPI_Type_create_hindexed_end`		`mpi_datatypes`
`MPI_Type_create_f90_integer_start`		`mpi_datatypes`
`MPI_Type_create_f90_integer_end`		`mpi_datatypes`
`MPI_Type_create_keyval_start`		`mpi_datatypes`
`MPI_Type_create_keyval_end`		`mpi_datatypes`
`MPI_Type_create_struct_start`		`mpi_datatypes`
`MPI_Type_create_struct_end`		`mpi_datatypes`
`MPI_Type_delete_attr_start`		`mpi_datatypes`
`MPI_Type_delete_attr_end`		`mpi_datatypes`
`MPI_Type_dup_start`		`mpi_datatypes`
`MPI_Type_dup_end`		`mpi_datatypes`
`MPI_Type_extent_start`		`mpi_datatypes`
`MPI_Type_extent_end`		`mpi_datatypes`
`MPI_Type_free_start`		`mpi_datatypes`
`MPI_Type_free_end`		`mpi_datatypes`
`MPI_Type_free_keyval_start`		`mpi_datatypes`
`MPI_Type_free_keyval_end`		`mpi_datatypes`
`MPI_Type_get_attr_start`		`mpi_datatypes`
`MPI_Type_get_attr_end`		`mpi_datatypes`
`MPI_Type_get_contents_start`		`mpi_datatypes`
`MPI_Type_get_contents_end`		`mpi_datatypes`
`MPI_Type_get_envelope_start`		`mpi_datatypes`
`MPI_Type_get_envelope_end`		`mpi_datatypes`
`MPI_Type_get_extent_start`		`mpi_datatypes`
`MPI_Type_get_extent_end`		`mpi_datatypes`

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Type_get_name_start		mpi_datatypes
MPI_Type_get_name_end		mpi_datatypes
MPI_Type_get_true_extent_start		mpi_datatypes
MPI_Type_get_true_extent_end		mpi_datatypes
MPI_Type_hindexed_start		mpi_datatypes
MPI_Type_hindexed_end		mpi_datatypes
MPI_Type_indexed_start		mpi_datatypes
MPI_Type_indexed_end		mpi_datatypes
MPI_Type_create_indexed_block_start		mpi_datatypes
MPI_Type_create_indexed_block_end		mpi_datatypes
MPI_Type_lb_start		mpi_datatypes
MPI_Type_lb_end		mpi_datatypes
MPI_Type_create_resized_start		mpi_datatypes
MPI_Type_create_resized_end		mpi_datatypes
MPI_Type_set_attr_start		mpi_datatypes
MPI_Type_set_attr_end		mpi_datatypes
MPI_Type_set_name_start		mpi_datatypes
MPI_Type_set_name_end		mpi_datatypes
MPI_Type_size_start		mpi_datatypes
MPI_Type_size_end		mpi_datatypes
MPI_Type_struct_start		mpi_datatypes
MPI_Type_struct_end		mpi_datatypes
MPI_Type_ub_start		mpi_datatypes
MPI_Type_ub_end		mpi_datatypes
MPI_Unpack_start	bytes	mpi_datatypes
MPI_Unpack_end		mpi_datatypes
MPI_Wait_start	request	mpi_request
MPI_Wait_end	recvbytes source recvtag request	mpi_request
MPI_Waitall_start	count	mpi_request

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to mpi_api
MPI_Waitall_end	bytes count	mpi_request
MPI_Waitany_start	count	mpi_request
MPI_Waitany_end	bytes index	mpi_request
MPI_Waitsome_start	incount	mpi_request
MPI_Waitsome_end	bytes outcount	mpi_request
MPI_Win_create_start		mpi_win
MPI_Win_create_end		mpi_win
MPI_Win_create_errhandler_start		
MPI_Win_create_errhandler_end		
MPI_Win_create_keyval_start		
MPI_Win_create_keyval_end		
MPI_Win_delete_attr_start		
MPI_Win_delete_attr_end		
MPI_Win_fence_start		mpi_win
MPI_Win_fence_end		mpi_win
MPI_Win_free_start		mpi_win
MPI_Win_free_end		mpi_win
MPI_Win_free_keyval_start		
MPI_Win_free_keyval_end		
MPI_Win_get_attr_start		
MPI_Win_get_attr_end		
MPI_Win_get_errhandler_start		
MPI_Win_get_errhandler_end		
MPI_Win_get_group_start		
MPI_Win_get_group_end		
MPI_Win_get_name_start		
MPI_Win_get_name_end		
MPI_Win_lock_start		mpi_win
MPI_Win_lock_end		mpi_win
MPI_Win_set_attr_start		

TABLE D-1 TNF Probes, Associated Arguments, and Groups for MPI Calls *(Continued)*

Probe	Argument(s)	Group(s), in Addition to `mpi_api`
`MPI_Win_set_attr_end`		
`MPI_Win_set_errhandler_start`		
`MPI_Win_set_errhandler_end`		
`MPI_Win_set_name_start`		
`MPI_Win_set_name_end`		
`MPI_Win_unlock_start`		`mpi_win`
`MPI_Win_unlock_end`		`mpi_win`

TNF Probes for MPI I/O

Like the MPI routines, each MPI I/O routine is associated with two TNF probes: one ending in `_start` and one ending in `_end`. Probes are also included for some specific arguments, most of which are defined in the MPI standard and described in the man pages included with Sun MPI. The `ctxt` argument for the context id assigned to a particular communicator, however, is not mentioned in the standard or man pages. It is described in "TNF Probes for MPI" on page 104.

Every TNF probe for MPI I/O is associated with both the `mpi_api` and the `mpi_io` groups. Choosing `mpi_api` enables Prism to probe all the MPI routines for which probes exist, whereas choosing `mpi_io` enables you to focus on the I/O routines. Additional groups exist to probe subsets of the I/O routines, as well. The seven groups for MPI I/O routine probes are:

- `mpi_api` – All the TNF probes for MPI routines
- `mpi_io` – MPI I/O routines only
- `mpi_io_consistency` – Atomicity and synchronization routines
- `mpi_io_datarep` – Data representation routines
- `mpi_io_errhandler` – Error-handling routines
- `mpi_io_file` – Group(s), in addition to `mpi_api` and `mpi_io`
- `mpi_io_rw` – Read/write routines

TABLE D-2 TNF Probes, Associated Arguments, and Groups for MPI I/O Calls

Probe	Argument(s)	Group(s), in Addition to `mpi_api` & `mpi_io`
`MPI_File_close_start`	`filename`	`mpi_io_file`
`MPI_File_close_end`		`mpi_io_file`
`MPI_File_create_errhandler_start`		`mpi_io_errhandler`
`MPI_File_create_errhandler_end`		`mpi_io_errhandler`
`MPI_File_delete_start`	`filename`	`mpi_io_file`
`MPI_File_delete_end`	`filename`	`mpi_io_file`
`MPI_File_get_amode_start`	`filename` `amode`	`mpi_io_file`
`MPI_File_get_amode_end`	`filename` `amode`	`mpi_io_file`
`MPI_File_get_atomicity_start`	`filename` `flag`	`mpi_io_consistency`
`MPI_File_get_atomicity_end`	`filename` `flag`	`mpi_io_consistency`
`MPI_File_get_byte_offset_start`	`filename` `offset disp`	`mpi_io_rw`
`MPI_File_get_byte_offset_end`	`filename` `offset disp`	`mpi_io_rw`
`MPI_File_get_errhandler_start`	`filename`	`mpi_io_errhandler`
`MPI_File_get_errhandler_end`	`filename`	`mpi_io_errhandler`
`MPI_File_get_group_start`	`filename`	`mpi_io_file`
`MPI_File_get_group_end`	`filename`	`mpi_io_file`
`MPI_File_get_info_start`	`filename`	`mpi_io_file`
`MPI_File_get_info_end`	`filename`	`mpi_io_file`
`MPI_File_get_position_start`	`filename` `offset`	`mpi_io_rw`
`MPI_File_get_position_end`	`filename` `offset`	`mpi_io_rw`
`MPI_File_get_position_shared_start`	`filename` `offset`	`mpi_io_rw`
`MPI_File_get_position_shared_end`	`filename` `offset`	`mpi_io_rw`

TABLE D-2 TNF Probes, Associated Arguments, and Groups for MPI I/O Calls

Probe	Argument(s)	Group(s), in Addition to `mpi_api` & `mpi_io`
`MPI_File_get_size_start`	`filename` `size`	`mpi_io_file`
`MPI_File_get_size_end`	`filename` `size`	`mpi_io_file`
`MPI_File_get_type_extent_start`	`filename` `datatype` `extent`	`mpi_io_datarep`
`MPI_File_get_type_extent_end`	`filename` `datatype` `extent`	`mpi_io_datarep`
`MPI_File_get_view_start`	`filename` `disp etype` `filetype` `datarep_name`	`mpi_io_file`
`MPI_File_get_view_end`	`filename` `disp etype` `filetype` `datarep_name`	`mpi_io_file`
`MPI_File_iread_start`	`filename` `bytes`	`mpi_io_rw`
`MPI_File_iread_end`	`filename`	`mpi_io_rw`
`MPI_File_iread_at_start`	`filename` `offset`	`mpi_io_rw`
`MPI_File_iread_at_end`	`filename` `offset`	`mpi_io_rw`
`MPI_File_iread_shared_start`	`filename`	`mpi_io_rw`
`MPI_File_iread_shared_end`	`filename`	`mpi_io_rw`
`MPI_File_iwrite_start`	`filename`	`mpi_io_rw`
`MPI_File_iwrite_end`	`filename`	`mpi_io_rw`
`MPI_File_iwrite_at_start`	`filename` `offset`	`mpi_io_rw`
`MPI_File_iwrite_at_end`	`filename` `offset`	`mpi_io_rw`
`MPI_File_iwrite_shared_start`	`filename`	`mpi_io_rw`
`MPI_File_iwrite_shared_end`	`filename`	`mpi_io_rw`

TABLE D-2 TNF Probes, Associated Arguments, and Groups for MPI I/O Calls

Probe	Argument(s)	Group(s), in Addition to `mpi_api` & `mpi_io`
`MPI_File_open_start`	`filename` `amode` `file_handle`	`mpi_io_file`
`MPI_File_open_end`	`filename` `amode` `file_handle`	`mpi_io_file`
`MPI_File_nonblocking_read_actual_end`	`filename` `offset count` `datatype`	`mpi_io_rw`
`MPI_File_nonblocking_write_actual_end`	`filename` `offset count` `datatype`	`mpi_io_rw`
`MPI_File_preallocate_start`	`filename` `size`	`mpi_io_file`
`MPI_File_preallocate_end`	`filename` `size`	`mpi_io_file`
`MPI_File_read_start`	`filename` `count` `datatype`	`mpi_io_rw`
`MPI_File_read_end`	`filename`	`mpi_io_rw`
`MPI_File_read_all_start`	`filename`	`mpi_io_rw`
`MPI_File_read_all_end`	`filename`	`mpi_io_rw`
`MPI_File_read_all_begin_start`	`filename`	`mpi_io_rw`
`MPI_File_read_all_begin_end`	`filename`	`mpi_io_rw`
`MPI_File_read_all_end_start`	`filename`	`mpi_io_rw`
`MPI_File_read_all_end_end`	`filename` `bytes`	`mpi_io_rw`
`MPI_File_read_at_start`	`filename` `offset`	`mpi_io_rw`
`MPI_File_read_at_end`	`filename` `offset`	`mpi_io_rw`
`MPI_File_read_at_all_start`	`filename` `offset`	`mpi_io_rw`
`MPI_File_read_at_all_end`	`filename` `offset`	`mpi_io_rw`

TABLE D-2 TNF Probes, Associated Arguments, and Groups for MPI I/O Calls

Probe	Argument(s)	Group(s), in Addition to `mpi_api` & `mpi_io`
`MPI_File_read_at_all_begin_start`	`filename` `offset`	`mpi_io_rw`
`MPI_File_read_at_all_begin_end`	`filename` `offset`	`mpi_io_rw`
`MPI_File_read_at_all_end_start`	`filename`	`mpi_io_rw`
`MPI_File_read_at_all_end_end`	`filename` `bytes`	`mpi_io_rw`
`MPI_File_read_ordered_start`	`filename`	`mpi_io_rw`
`MPI_File_read_ordered_end`	`filename`	`mpi_io_rw`
`MPI_File_read_ordered_begin_start`	`filename`	`mpi_io_rw`
`MPI_File_read_ordered_begin_end`	`filename`	`mpi_io_rw`
`MPI_File_read_ordered_end_start`	`filename`	`mpi_io_rw`
`MPI_File_read_ordered_end_end`	`filename` `bytes`	`mpi_io_rw`
`MPI_File_read_shared_end`	`filename`	`mpi_io_rw`

`MPI_File_seek_start`	`filename`
	`offset`
	`whence`
		`mpi_io_rw`

`MPI_File_seek_end`	`filename`
	`offset`
	`whence`
		`mpi_io_rw`

`MPI_File_seek_shared_start`	`filename`
	`offset`
	`whence`
		`mpi_io_rw`

`MPI_File_seek_shared_end`	`filename`
	`offset`
	`whence`
		`mpi_io_rw`

`MPI_File_set_atomicity_start`	`filename` `flag`	`mpi_io_consistency`
`MPI_File_set_atomicity_end`	`filename` `flag`	`mpi_io_consistency`
`MPI_File_set_errhandler_start`	`filename`	`mpi_io_errhandler`
`MPI_File_set_errhandler_end`	`filename`	`mpi_io_errhandler`
`MPI_File_set_info_start`	`filename`	`mpi_io_file`
`MPI_File_set_info_end`	`filename`	`mpi_io_file`

TABLE D-2 TNF Probes, Associated Arguments, and Groups for MPI I/O Calls

Probe	Argument(s)	Group(s), in Addition to mpi_api & mpi_io
MPI_File_set_size_start	filename size	mpi_io_file
MPI_File_set_size_end	filename size	mpi_io_file
MPI_File_set_view_start	filename disp etype filetype datarep_name	mpi_io_file
MPI_File_set_view_end	filename disp etype filetype datarep_name	mpi_io_file
MPI_File_sync_start	filename	mpi_io_consistency
MPI_File_sync_end	filename	mpi_io_consistency
MPI_File_write_start	filename	mpi_io_rw
MPI_File_write_end	filename	mpi_io_rw
MPI_File_write_all_start	filename	mpi_io_rw
MPI_File_write_all_end	filename	mpi_io_rw
MPI_File_write_all_begin_start	filename	mpi_io_rw
MPI_File_write_all_begin_end	filename	mpi_io_rw
MPI_File_write_all_end_start	filename	mpi_io_rw
MPI_File_write_all_end_end	filename bytes	mpi_io_rw
MPI_File_write_at_start	filename offset	mpi_io_rw
MPI_File_write_at_end	filename offset	mpi_io_rw
MPI_File_write_at_all_start	filename offset	mpi_io_rw
MPI_File_write_at_all_end	filename offset	mpi_io_rw
MPI_File_write_at_all_begin_start	filename offset	mpi_io_rw
MPI_File_write_at_all_begin_end	filename offset	mpi_io_rw

TABLE D-2 TNF Probes, Associated Arguments, and Groups for MPI I/O Calls

Probe	Argument(s)	Group(s), in Addition to `mpi_api` & `mpi_io`
`MPI_File_write_at_all_end_start`	`filename`	`mpi_io_rw`
`MPI_File_write_at_all_end_end`	`filename` `bytes`	`mpi_io_rw`
`MPI_File_write_ordered_start`	`filename`	`mpi_io_rw`
`MPI_File_write_ordered_end`	`filename`	`mpi_io_rw`
`MPI_File_write_ordered_begin_start`	`filename`	`mpi_io_rw`
`MPI_File_write_ordered_begin_end`	`filename`	`mpi_io_rw`
`MPI_File_write_ordered_end_start`	`filename`	`mpi_io_rw`
`MPI_File_write_ordered_end_end`	`filename` `bytes`	`mpi_io_rw`
`MPI_File_write_shared_start`	`filename`	`mpi_io_rw`
`MPI_File_write_shared_end`	`filename`	`mpi_io_rw`
`MPI_Register_datarep_start`	`datarep_name`	`mpi_io_datarep`
`MPI_Register_datarep_end`	`datarep_name`	`mpi_io_datarep`

Index