Sun™ MPI 3.0 Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300 fax 650 969-9131

Part No.: 805-1556-10
Revision A, November 1997

Contents

Preface

Sun MPI 3.0 Guide describes the Sun™ MPI library of message-passing routines and explains how to develop, execute, and debug an MPI (message-passing interface) program on a Sun HPC System.

For the most part, this guide does not repeat information that is available in detail elsewhere, but rather it focuses on what is specific to the Sun MPI implementation. References to more general source materials are provided in the "Related Publications" section of this preface.

The reader is assumed to be familiar with programming in C or Fortran. Some familiarity with parallel programming and with the message-passing model is also required.

Before You Read This Book

For general information about programming on a Sun HPC System, refer to the *Sun HPC Software User's Guide*. Release notes for Sun MPI are included in the *Sun HPC Software Release Notes*. For general information about writing MPI programs, refer to any of the several MPI source documents cited in the "Related Publications" section later in this preface.

Related Publications

This book focuses on Sun MPI and assumes familiarity with the MPI standard. The following materials provide useful background about using Sun MPI and about the MPI standard.

Books

Among the documents included with Sun HPC Software, you may want to pay particular attention to these:

Application	Title	Part Number
Sun HPC Software	*Sun HPC Software 2.0 Release Notes*	805-2191
Sun HPC Software	*Sun HPC Software 2.0 User's Guide*	805-1554

In addition, you may want to consult the Prism documentation for information on debugging your Sun MPI program, as well as the S3L documentation to learn about using the Sun Scientific Subroutine Library

Prism	*Prism 5.0 User's Guide*	805-1552
Prism	*Prism 5.0 Reference Manual*	805-1553
S3L	*S3L 2.0 Guide*	805-1557

These books, which are not provided by Sun, should be available at your local computer bookstore:

- *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by William Gropp, Ewing Lusk, and Anthony Skjellum (Cambridge: MIT Press, 1994).
- *MPI: The Complete Reference*, by Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra (Cambridge: MIT Press, 1995).
- *Parallel Programming with MPI*, by Peter S. Pacheco (San Francisco: Morgan Kaufmann Publishers, Inc., 1997).

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`. You may need to ask your system administrator for their location at your site.

On the World Wide Web

There is a wealth of documentation on MPI available on the World Wide Web. Here are a few URLs for Web sites (and each of these leads you to others):

- The MPI home page, with links to specifications for MPI-1 and MPI-2 standards:

`http://www.mpi-forum.org`

- *User's Guide for mpich, a Portable Implementation of MPI,* a user's guide for the implementation on which Sun MPI is based:

`http://www.mcs.anl.gov/mpi/mpiuserguide/paper.html`

  A PostScript version is also available at this site:

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

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

`http://www.erc.msstate.edu/mpi/`
`http://www.arc.unm.edu/workshop/mpi/mpi.html`

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- AnswerBook™ online documentation for the Solaris 2.x software environment
- Other software documentation that you received with your system

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface or Symbol	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. Command-line variable; replace with a real name or value.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be `root` to do this. To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Ordering Sun Documents

SunDocs[SM] is a distribution program for Sun Microsystems technical documentation. Contact SunExpress for easy ordering and quick delivery. You can find a listing of available Sun documentation on the World Wide Web.

TABLE P-3 SunExpress Contact Information

Country	Telephone	Fax
Belgium	02-720-09-09	02-725-88-50
Canada	1-800-873-7869	1-800-944-0661
France	0800-90-61-57	0800-90-61-58
Germany	01-30-81-61-91	01-30-81-61-92
Holland	06-022-34-45	06-022-34-46
Japan	0120-33-9096	0120-33-9097
Luxembourg	32-2-720-09-09	32-2-725-88-50
Sweden	020-79-57-26	020-79-57-27
Switzerland	0800-55-19-26	0800-55-19-27
United Kingdom	0800-89-88-88	0800-89-88-87
United States	1-800-873-7869	1-800-944-0661

World Wide Web: `http://www.sun.com/sunexpress/`

Sun Documentation on the Web

The `docs.sun.com` web site enables you to access Sun technical documentation on the World Wide Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email or fax your comments to us. Please include the part number of your document in the subject line of your email or fax message.

- Email: `smcc-docs@sun.com`
- Fax: SMCC Document Feedback
  1-650-786-6443

Information Sources for PVM and PETSc

TABLE P-4 lists organizations and resources for information about the publicly available libraries PVM and PETSc. This information is subject to change.

TABLE P-4 Information Sources for PVM and PETSc

Product	Contact
PVM	Copyright holders: University of Tennessee, Oak Ridge National Laboratory, Emory University Electronic mail: `pvm@msr.epm.ornl.gov` Newsgroup: `comp.parallel.pvm` Web site: `http://www.epm.ornl.gov/pvm/pvm_home.html`
PETSc	Developed and supported by the Mathematics and Computer Science Division of the Argonne National Laboratory.

CHAPTER 1

Introduction to Sun MPI

1.1 What Is Sun MPI?

Sun MPI is Sun Microsystems' implementation of MPI (message-passing interface), the industry-standard specification for writing message-passing programs. Message passing is a programming model that gives the programmer explicit control over interprocess communication.

1.2 Background: The MPI Standard

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 (MPI-1), was first published in 1993; its most recent version was published in June 1995. It has been well received, and there are several implementations available publicly. The MPI-2 specification was published in July, 1997.

1.3 The Sun MPI Library

Sun MPI is a library of message-passing routines based on the MPICH Version 1.0.13 implementation, which was written at Argonne National Laboratory (ANL) and is MPI-1 compliant. It contains all the MPI-1 routines, that is, routines for point-to-point communication, collective (global) communication, and routines for environmental and process management.

Man pages for Sun MPI routines are available online, and the routines are listed in Appendix A, “Sun MPI and Sun MPI I/O Routines.”

Sun MPI provides several advantages over MPICH and other publicly available implementations. Chapter 2, “The Sun MPI Library,” describes the Sun MPI library and its advantages.

1.4 Using Sun MPI

The current release of Sun MPI is optimized to run with Sun HPC Software.

Chapter 3, “Using Sun MPI,” describes developing, executing, and debugging a Sun MPI program.

1.5 MPI I/O

I/O for Sun MPI is based on chapter 9 of the MPI-2 specification. MPI I/O is a library of routines for parallel file I/O that was developed as an extension to the MPI standard. Chapter 4, “Sun MPI I/O,” describes these routines. Their man pages are provided online, and the routines are listed in Appendix A, “Sun MPI and Sun MPI I/O Routines.”

CHAPTER 2

The Sun MPI Library

This chapter describes the Sun MPI library:

- Section 2.1 outlines Sun MPI's advantages and describes the ways in which Sun MPI differs from the MPI standard and the MPICH implementation.
- Section 2.2 describes the Sun MPI routines.
- Section 2.3 describes programming with Sun MPI.
- Section 2.4 describes how to obtain and build MPE.

2.1 Sun MPI Advantages

The functionality of Sun MPI is the same as that of MPICH, but Sun MPI provides the following advantages over publicly available implementations of MPI:

- Multithreaded programming is supported via a thread-safe version of the Sun MPI library.
- Seamless use of different network protocols; for example, if you've compiled your code on a Sun Ultra™ HPC System that has a Scalable Coherent Interface (SCI) network, you can run it without change on a system that has an ATM network.
- Multiprotocol support such that MPI picks the fastest available medium for each type of connection (such as shared memory, SCI, or ATM)
- Improved performance on clusters of SMPs due to communication via shared memory.
- The shared memory is more tunable.
- Full integration with the run-time environment (RTE); that is, Sun MPI is fully integrated with the batch system, load-balancing, etc.
- Prism support; that is, users can develop, run, and debug programs in the Prism programming environment.

- MPI I/O support for file I/O (see Chapter 4, “Sun MPI I/O”).
- Sun MPI is a dynamic library.

2.2 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, “Sun MPI and Sun MPI I/O Routines,” with brief descriptions and their C syntax. For detailed descriptions of individual routines, see the man pages. For more complete information, see the MPI standard and the MPICH user’s guide. (Web sites for these documents are listed in “Related Publications” on page viii.)

2.2.1 Point-to-Point Routines

Point-to-point 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 for or tested for. It is expected that eventually nonblocking send and receive calls will allow the overlap of communication and computation.

MPI’s four modes for point-to-point communication are:

- *Standard*, in which the completion of a send implies that the message either has been received or is buffered internally. Users are free to overwrite the buffer that they passed in with any of the blocking send or receive routines.
- *Buffered*, in which the user guarantees a certain amount of buffering space.
- *Synchronous*, in which rendezvous semantics occur between sender and receiver, that is, a send blocks until the corresponding receive has occurred.
- *Ready*, in which 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.

2.2.2 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 2-1 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.
`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.

2.2.3 Managing Groups, Contexts, and Communicators

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

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, a default communicator is defined, `MPI_COMM_WORLD`, which has as a process group all the processes of the job. For many programs, this is the only communicator needed. The process group that corresponds to `MPI_COMM_WORLD` is not predefined, but can be accessed using `MPI_COMM_GROUP`.

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, “Sun MPI and Sun MPI I/O Routines.”)

2.2.4 Data Types

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

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

The basic data types that can be specified for the data-type argument correspond to the basic data types of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in TABLE 2-2.

TABLE 2-2 Possible Values for the Data Type Argument for Fortran

MPI Data Type	Fortran Data Type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_REAL4	REAL*4
MPI_REAL8	REAL*8
MPI_INTEGER2	INTEGER*2
MPI_INTEGER4	INTEGER*4

Possible values for MPI derived data types in C and the corresponding C types are listed in TABLE 2-3.

TABLE 2-3 Possible Values for the Data Type Argument for C

MPI Data Type	C Data Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

The data types MPI_BYTE and MPI_PACKED have no corresponding Fortran or C data types.

Sun MPI supports the data types listed in TABLE 2-2 and TABLE 2-3; these match the basic data types of Fortran 77 and ANSI C. Sun MPI data types are also provided for the following additional data types:

- MPI_LONG_LONG_INT for 64-bit C integers declared to be of type long long int
- MPI_DOUBLE_COMPLEX for double-precision complex in Fortran declared to be of type DOUBLE COMPLEX
- MPI_REAL4 and MPI_REAL8 for Fortran REALs, declared to be of type REAL*4 and REAL*8 respectively
- MPI_INTEGER2 and MPI_INTEGER4 for Fortran integers declared to be of type INTEGER*2 and INTEGER*4, respectively

2.2.5 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 optimized 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.”

2.2.6 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 communications.

2.2.7 Environmental Inquiry Functions

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

Other than `MPI_Initialized`, no MPI routine may be called before `MPI_Init`. `MPI_Finalize` must be the last MPI routine called on each process, and 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. The Sun MPI implementation uses the error codes and classes implemented in MPICH.

2.2.8 The Thread-Safe Library

Sun MPI comprises two versions of the library:

- Thread-safe – `libmpi_mt.so`
- Not thread-safe (default) – `libmpi.so`

For multithreaded programs, the user must link with the thread-safe library, `libmpi_mt.so`.

For programs that are not multithreaded, the user can link against either library. However, non-multithreaded programs will have better performance using `libmpi.so`, as it does not incur the extra overhead of providing thread-safety. Therefore, you should use `libmpi.so` whenever possible for maximum performance.

2.2.8.1 Stubbing Thread Calls

The `libthread.so` library is automatically linked into `libmpi.so`. This means that any thread-function calls in your program will be resolved by the `libthread.so` library. Simply omitting `libthread.so` from the link line will 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.)

2.2.9 Profiling Interface

Sun's version of the MPI library meets the requirements of the profiling interface described in Chapter 8 of the MPI-1 Standard. For a more detailed description of the profiling interface, please refer to that chapter of the standard. This section covers some specifics of the Sun MPI implementation.

To support the profiling interface, Sun supplies two additional libraries, `libpmpi.so` and `libfmpi.so`.

2.2.9.1 `libpmpi.so`

The `libpmpi.so` library provides a name-shifted interface to all the MPI functions. All calls to `MPI_` are replaced with `PMPI_`.

2.2.9.2 `libfmpi.so`

Sun implements the Fortran binding to MPI as wrappers around the C implementation. Therefore, every Fortran MPI function ultimately calls the corresponding C MPI function. The `libfmpi.so` library provides only the Fortran wrappers, separate from the entire MPI library. This allows a profile library–writer to generate a single profiled library with only the C functional interface. 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 Section 2.4 for more information about obtaining MPE.) The *User's Guide for mpich, a Portable Implementation of MPI*, includes more detailed information about using profiling libraries. For information about this and other MPI- and MPICH-related publications, see the "Related Publications" section on page viii of the preface.

FIGURE 2-1 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()`.

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

`cc ..... -l`*`library–name`* **`-lpmpi -lmpi`**

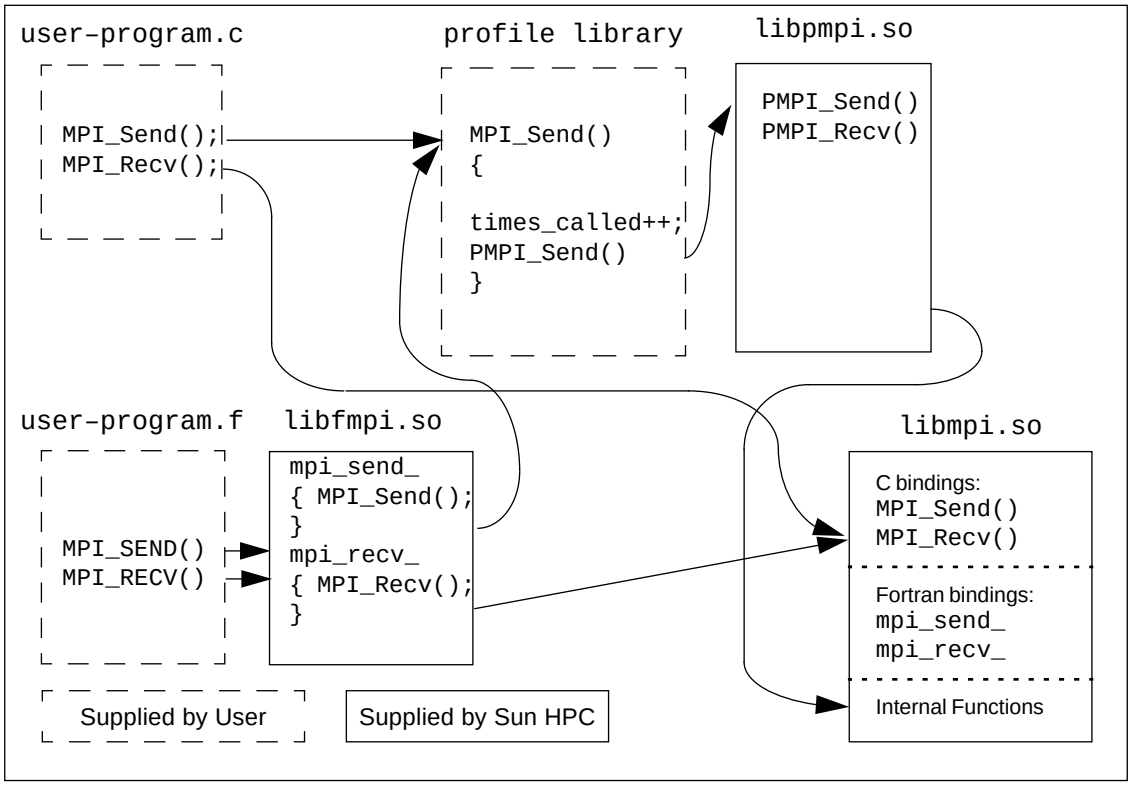

FIGURE 2-1 Sun MPI Profiling Interface

2.3 Programming With Sun MPI

Although there are about 130 routines in the Sun MPI library, you can write programs for a wide range of problems using only six routines:

TABLE 2-4 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 Section 3.6, “Sample MPI Program,” starting on page 3-7, for a sample Sun MPI program. See Section A.1, “Sun MPI Routines,” for a complete list of Sun MPI routines.

2.4 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. In case you would like to use these extensions to the MPI library, we have included some instructions for downloading it from ANL and building it yourself. Note that these procedures may change if ANL makes changes to MPE.

▼ To Obtain and Build MPE

The MPE software is available from Argonne National Laboratory.

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

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

   The `mpe.tar.gz` file is about 240 Kbytes.

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`**

   This will build several libraries.

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.

Please refer to the *User's Guide for mpich, a Portable Implementation of MPI*, for information on how to use MPE. For information about this and other MPI- and MPICH-related publications, see the "Related Publications" section on page viii of the preface.

CHAPTER 3

Using Sun MPI

This chapter describes developing, compiling and linking, and executing a Sun MPI program within the Sun HPC Software environment. It includes information about tuning shared memory allocation, guidelines for multithreaded programming, and some tips for troubleshooting.

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. For complete information about developing MPI programs, see some of the MPI publications listed in the preface. For complete information about executing programs in the Sun HPC Software environment, see the *Sun HPC Software User's Guide.*

3.1 Header Files

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

For C programs, use

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

For Fortran, use

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

These lines allow 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/`. 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 Section 3.4, "Compiling and Linking").

3.2 Developing a Sun MPI Program

Sun MPI routines are called from within a C or Fortran 77 program that you develop and write in the usual way. However, if you are developing a multithreaded program, you must follow the guidelines for thread-safe programming described in Section 3.7. See CODE EXAMPLE 3-1, starting on page 3-8, for an example of a simple Sun MPI program, and CODE EXAMPLE 3-2, starting on page 3-11, for an example of a multithreaded program. Some examples of MPI code and a sample makefile are also available online in the directory `/opt/SUNWhpc/examples/mpi`.

Note – Do not use the characters *TM*, *tm*, *MPI*, or *mpi* as a prefix in any routine name that you create; they are reserved for use by Sun MPI.

Caution – Do not use `mmap` with the `MAP_FIXED` flag. Using this flag could corrupt the Sun MPI shared-memory segment, resulting in unexplained failures in your Sun MPI program.

3.3 Logging In

If you are not already logged in to the Sun HPC System, you'll need to log in when you are ready to compile and link your program.

To log in, issue the `tmlogin` command at your UNIX prompt. For example,

```
% tmlogin –p Mars Planet
```

logs you in to the Sun HPC partition named Mars on the Sun HPC System named Planet. For other login methods, see the *Sun HPC Software User's Guide*.

3.4 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.

If you will be using the Prism debugger, you must compile your program with WorkShop™ Compilers Fortran, either v4.0 or v4.2. (The v4.2 compilers are included in the Sun Performance WorkShop™ Fortran v3.0 suite of tools.) Prism also requires `-xs -g` flags during compilation. If the code is threaded, you will not be able to debug with Prism. (See Section 3.8, "Debugging.")

- To compile a nonthreaded C program `myprog.c` and link with Sun MPI, enter:

```
% cc myprog.c -o myprog -I/opt/SUNWhpc/include \
      -L/opt/SUNWhpc/lib -R/opt/SUNWhpc/lib -lmpi
```

- To compile a nonthreaded Fortran 77 program `myprog.f` and link with MPI, enter:

```
% f77 -dalign myprog.f -o myprog -I/opt/SUNWhpc/include \
      -L/opt/SUNWhpc/lib -R/opt/SUNWhpc/lib -lmpi
```

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

- To compile a C++ program `myprog.cc` and link with Sun MPI, enter:

```
% CC myprog.cc -o myprog -I/opt/SUNWhpc/include \
      -L/opt/SUNWhpc/lib -R/opt/SUNWhpc/lib -mt -lmpi
```

- For *multithreaded programs*, replace `-lmpi` with `-lmpi_mt`.
- For programs that use *MPI I/O*, insert `-lmpi-iof` before `-lmpi`.

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

3.5 Executing

You can run your job *interactively* with `tmrun` (see Section 3.5.3), or you can submit it to a *batch queue* with `tmsub` (see Section 3.5.4). If you'll be using Prism to debug your program, run the job interactively (see the discussion of Prism in Section 3.8.2, "Debugging With Prism").

Before running your program or submitting it to a batch queue, you can use the command `tminfo` to find out how the system is configured (see the *Sun HPC Software User's Guide* for further information).

Before executing your job, you may also need to set an environment variable to adjust the allocation of shared memory. See Section 3.5.1 for information.

3.5.1 Shared Memory Allocation

Sun MPI supports communication via shared memory between MPI processes running in the same task and on the same node. This fast shared-memory communication provides the best performance available when running on an SMP node. During the Sun MPI initialization phase, a file is created for use as the shared memory segment. Eventually all eligible processes attach to this file, and on-node MPI communications take place through it.

The size of the file must be large enough to meet the needs of the task, but it should also avoid needlessly tying up memory resources. Sun MPI specifies its size according to the number of on-node processes participating in the MPI task. In most cases, this default size will accomodate all MPI shared memory use. In other cases, it will be necessary to request a shared-memory file size specific to some tasks' requirements. Three environment variables are supported for custom size specification: `MPI_SHORTMSGSIZE`, `MPI_GLOBMEMSIZE` and `MPI_UNITMEMSIZE`.

In Sun MPI, "short" messages are handled differently from "long" messages. Short messages are passed through memory that is private to a communicating pair; long messages are passed in memory temporarily allocated from an area common to all eligible processes. `MPI_SHORTMSGSIZE` determines the transition point from short to long messages. `MPI_GLOBMEMSIZE` and `MPI_SHORTMSGSIZE` determine the size of the common shared memory area.

Because short messages use memory that is private to a communicating pair, they are more efficient than long messages. Depending on the number of on-node processes and the `MPI_SHORTMSGSIZE` size specified, however, memory demands may be unacceptable. Based on a knowledge of message-size distribution and message traffic patterns, the user can arrive at a compromise between performance and memory use.

The following sections describe how to use these three environment variables. See Section 3.9, "Configuration and Tuning," for more detailed information.

Note – You may not set these environment variables to negative values. Values should be chosen such that the shared-memory file size limits are observed. For compatibility with all supported versions of Solaris, Sun MPI imposes a shared-memory file size limit of 2 Gbytes. When specifying custom shared-memory sizes, some care should be taken if the resultant file size approaches 2 Gbytes. To find out how much memory is available for each node in your local Sun HPC System, use the `tminfo -N` command. (See the *Sun HPC Software User's Guide* for more information about `tminfo`.) See Section 3.9.2, "Shared Memory Configuration," for detailed information about calculating the size of the shared-memory file.

3.5.1.1 Using MPI_SHORTMSGSIZE

If your program will be using many-to-many message patterns, and most of the messages are of a similar size, the MPI_SHORTMSGSIZE variable may be the most effective method of managing memory allocation:

- MPI_SHORTMSGSIZE – A per-process quantity corresponding to the limit on the size of the short-message buffer. The size of the area reserved for short messages is defined as $3N^2$ * MPI_SHORTMSGSIZE, where N is the number of on-node processes in the MPI task. If MPI_SHORTMSGSIZE is not specified, the maximum short message size is 1024 bytes.

3.5.1.2 Using MPI_GLOBMEMSIZE and MPI_UNITMEMSIZE

Shared memory allocated for large messages can be controlled by setting the shell environment variable MPI_GLOBMEMSIZE or MPI_UNITMEMSIZE. If both are in effect in the same shell context, MPI_GLOBMEMSIZE takes precedence. If either is in effect, it overrides the Sun MPI default size.

- MPI_GLOBMEMSIZE – A global value representing the overall quantity of memory allocated to the large-message shared memory area. It is expressed in bytes as either a decimal or as a hexadecimal number.
- MPI_UNITMEMSIZE – A value that can be used to specify the size of the large-message shared memory area based on per-process memory requirements. If MPI_UNITMEMSIZE is set, the amount of memory reserved for large message passing will be equal to N^2 * MPI_UNITMEMSIZE, where N is the number of on-node processes in the MPI task. It is expressed in bytes as either a decimal or as a hexadecimal number.

For example, to set the environment variable MPI_GLOBMEMSIZE to 32 Mbytes (here in a C shell), you must first convert Mbytes to bytes ($32 * 2^{20} = 33554432$). Here is the decimal version:

```
% setenv MPI_GLOBMEMSIZE 33554432
```

and here is the hexadecimal version:

```
% setenv MPI_GLOBMEMSIZE 0x2000000
```

3.5.2 Setting MPI_SPIN_LIMIT

Sun MPI's blocking receive routines poll aggressively for messages. This aggressive polling, also known as "spinning," can use up system resources uselessly, with a detrimental effect on the performance achieved by other jobs in the Sun HPC System. To reduce the effect of this spinning, you should set the MPI_SPIN_LIMIT environment variable before starting your job.

MPI_SPIN_LIMIT limits the number of times a blocking receive routine will poll for its message. Once the number of times the routine has polled for its message equals the value set for MPI_SPIN_LIMIT, it will back off for a period of time, leaving the node free to be used by other jobs during that time. The value of MPI_SPIN_LIMIT is an integer. The default value, 0, allows a blocking receive to poll continually until it receives its message.

To determine an appropriate value for MPI_SPIN_LIMIT, consider the extent to which your program will be using shared memory (intranode) communication and the extent to which it will be using network (internode) communication. In the extreme case of using no network communication, such as you would find with an individual SMP, polling for messages takes a relatively short time, so you can safely set MPI_SPIN_LIMIT to a large value to avoid negative effects on the performance of your own program. At the other extreme, where there is no shared memory communication (such as with networked uniprocessors), polling for messages takes a long time, so you would set MPI_SPIN_LIMIT to a small value. Otherwise, the long spin cycle associated with each blocking receive could result in other jobs stalling for a period of time.

Your situation probably falls between these two extremes. For example, if your program is running on a cluster of eight SMP nodes, you should consider how many of the messages will be communicated over the network and how many will be communicated within a node, by shared memory communication.

3.5.3 tmrun

Use the tmrun command to run your job interactively if resources are available. For example,

```
% tmrun –p Dedicated –np 4 myprog
```

does this:

- tmrun submits the job to run immediately.
- –p Dedicated requests the partition Dedicated.
- –np 4 indicates the number of processes wanted is 4 (meaning that your executable will run on 4 processes and these processes will collectively define MPI_COMM_WORLD).

Note – For Release 3.0, the number of processes in a Sun MPI job is limited to 256.

- myprog is the name of the executable.

If the partition you requested is not available, you receive a message telling you so, and you can try again later. If batch queues are available, you have the option of submitting to one of the batch queues, as shown in the next section.

3.5.4 tmsub

Use the `tmsub` command to submit your job to the batch queue for execution. For example,

```
% tmsub –q Ruffian –np 4 myprog
```

does this:

- `tmsub` submits the job to the batch queue.
- `–q Ruffian` requests the queue Ruffian.
- `–np 4` indicates the number of processes wanted is 4 (meaning that your executable will run on 4 processes and these processes will collectively define `MPI_COMM_WORLD`).

Note – For Release 3.0, the number of processes in a Sun MPI job is limited to 256.

- `myprog` is the name of the executable.

After a successful submission, `tmsub` returns with a unique job ID, or *jid*, that may be used to reference the job in future operations.

```
% tmsub myprog
Job j12 submitted
```

There are many other options to the `tmrun` and `tmsub` commands. For complete information, see the chapter “Executing Programs” in the *Sun HPC Software User’s Guide*.

3.6 Sample MPI Program

Below is a sample MPI program that does two things:

- Produces “Hello from process *x* of *n*” to make sure all processes are there (`MPI_Comm_rank`) and to confirm the total number (`MPI_Comm_size`)
- Sends a simple message from one process to another

Also see CODE EXAMPLE 3-2, starting on page 3-11, for an example of a multithreaded program. Some examples of MPI code and a sample makefile are available online in the directory `/opt/SUNWhpc/examples/mpi`.

CODE EXAMPLE 3-1 Simple Sun MPI Program

```
#include <mpi.h>

main (argc, argv)
  int argc; char *argv[];
{
  int rank,size;
  int send[100],recv[100];
  int msg_len=6;
  int i;
  int source=0, dest=1;
  int tag=0;
  MPI_Status status;

  /* Initialize MPI */
  MPI_Init(&argc,&argv);

  /* Get rank of this process and process group size */
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  printf("Hello from process %d of %d.\n",rank,size);

  /* Set up two arrays to act as message send/recv buffers */
  for ( i=0; i<msg_len; i++ ) { send[i]=i; recv[i]=0; }

  if ( rank == source ) {
    /* Source process sends a message */
    MPI_Send(send,msg_len,MPI_INT,dest,tag,MPI_COMM_WORLD);
    printf("Process %d sent:",rank);
    for ( i=0; i<msg_len; i++ ) printf (" %d", send[i]);
    printf("\n");
  }

  if ( rank == dest ) {
    /* Destination process receives a message */
    MPI_Recv(recv,msg_len,MPI_INT,source,tag,
             MPI_COMM_WORLD,&status);
    printf("Process %d got :",rank);
    for ( i=0; i<msg_len; i++ ) printf(" %d", recv[i]);
    printf(" from process %d.\n",status.MPI_SOURCE);
  }

  /* Wrap up MPI operations */
  MPI_Finalize();
  printf("Process %d is finished.\n",rank);
}
```

3.7 Multithreaded Programming

When you are linked to the thread-safe library `libmpi_mt.so`, Sun MPI calls are thread safe, in accordance with basic tenets of thread safety for MPI mentioned in the MPI-2 specification[1]. This means that:

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

3.7.1 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, meaning 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, you will need to adhere to these guidelines:

- You must not have an operation posted in one thread and then completed in another. Similarly, you must not have a request serviced by 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` has been 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 overrides any stubs.

1. *Document for a Standard Message-Passing Interface*. Please see the preface of this document for more information about this and other recommended reference material.

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 of 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 assure 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.

Please 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. You must not use multiple instances of the same function on the same communicator with other differing parameters.
- 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 via `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.

3.7.2 Sample Threaded MPI Program

CODE EXAMPLE 3-2 is a program that tests multithreaded cshifts in Sun MPI. It is an example of a multithreaded Sun MPI program.

Also see CODE EXAMPLE 3-1, starting on page 3-8, for an example of a nonthreaded Sun MPI program. Some examples of MPI code and a sample makefile are available online in the directory `/opt/SUNWhpc/examples/mpi`.

CODE EXAMPLE 3-2 Program That Tests Multithreaded `cshift`s

```
/*
 * compile:  tmcc –D_REENTRANT mpi_mttest.c –lmpi_mt –lpthread
 * run:      tmrun –np 4 –Ns a.out 7
```

```
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <mpi.h>

#define  MAX_NTHREADS  64
#define  CSHIFT_TAG     9


/*
 * struct containing arguments for the cshift function. we need to do
 * this since the threads creation function accepts the address of
 * only one argument.
 */
typedef struct thrarg_s {
  int       thread_index; /* unique to each set of communicating
threads */
  MPI_Comm  thread_comm;  /* unique to each set of communicating
threads */
  int       nerrors;      /* keep track of the number of errors */
} thrarg_t;


/*
 * do a bunch of cshifts. this involves passing value(s) from one
 * processto the next, much like a bucket brigade. at each point we
 * make sure that we are receiving and sending the correct value(s).
 */
void *
cshifts(void *A)
{
  thrarg_t *B;
  MPI_Status status;
  int mypid, nprocs;
  int msg, new_msg;
  int i, isrc, idest, direction;

  B = (thrarg_t *) A;

  MPI_Comm_rank(B->thread_comm, &mypid);
  MPI_Comm_size(B->thread_comm, &nprocs);

  /*
   * note that the value being passed around is a function of the
   * thread index that we've assigned
   */
  msg = B->thread_index + 1;
```

```
  /* determine if we are shifting forward or backward */
  if (B->thread_index % 2) {
    direction = 1;
  } else {
    direction = -1;
  }

  isrc  = (mypid - direction + nprocs) % nprocs;
  idest = (mypid + direction + nprocs) % nprocs;

  /*
   * cshifts: use an alternate (checkerboard) pattern of sends and
   * recvs
   */
  for (i = 0; i < nprocs; i++) {
    if (mypid % 2) {
     MPI_Recv(&new_msg, 1, MPI_INT, isrc, CSHIFT_TAG, B->thread_comm,
                        &status);
      MPI_Send(&msg, 1, MPI_INT, idest, CSHIFT_TAG, B->thread_comm);

    } else {
      MPI_Send(&msg, 1, MPI_INT, idest, CSHIFT_TAG, B->thread_comm);
     MPI_Recv(&new_msg, 1, MPI_INT, isrc, CSHIFT_TAG, B->thread_comm,
                        &status);
    }

    /* see if the new message has the correct value */
    if (new_msg != B->thread_index + 1) B->nerrors++;
    msg = new_msg;
  }

  return (NULL);
}


/*
 * main program
 *
 * optional command line argument:
 *   nthreads - number of threads per MPI process
 */
int
main(int argc, char *argv[])
{
  pthread_attr_t attr;
  pthread_t      thread_ids[MAX_NTHREADS];
  thrarg_t       mydata[MAX_NTHREADS];
  int            sumerr[MAX_NTHREADS];
```

```
  int  mypid, nprocs, i;
  int  nthreads = 10;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  /*
   * read nthreads argument on the first process and broadcast the
   * value to the other processes
   */
  if (mypid == 0) {
    if (argc > 1) nthreads = atoi(argv [1]);

    printf("********  %d procs, %d threads  ********\n", nprocs,
nthreads);
  }

  if (nprocs > 1)
    MPI_Bcast(&nthreads, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if (nthreads < 1 || nthreads > MAX_NTHREADS) {
    if (mypid == 0) fprintf(stderr, "invalid value of nthreads\n");
    MPI_Finalize();
    exit (1);
  }

  /* fill mydata, which contains argument for the threads */
  for (i = 0; i < nthreads; i++) {
    mydata[i].thread_index = i;
    MPI_Comm_dup(MPI_COMM_WORLD, &(mydata[i].thread_comm));
    mydata[i].nerrors = 0;
  }

  /*
   * create threads and pass the arguments. use OS scheduling.
   */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  for (i = 0; i < nthreads; i++)
    pthread_create(&thread_ids[i], &attr, cshifts,
                        (void *) &(mydata[i].thread_index));

  for (i = 0; i < nthreads; i++) {
    pthread_join(thread_ids[i], NULL);
    MPI_Comm_free(&(mydata[i].thread_comm));
  }
```

```
  /* count errors on process 0 */
  for (i = 0; i < nthreads; i++)
    MPI_Reduce(&mydata[i].nerrors, &sumerr[i], 1, MPI_INT, MPI_SUM,
                      0, MPI_COMM_WORLD);

  if (mypid == 0) {
    int totalerr = 0;

    for (i = 0; i < nthreads; i++) totalerr += sumerr[i];

    if (totalerr <= 0) {
      printf("cshifts passed\n");
    } else {
      printf("cshifts FAILED. number errors = %d\n", totalerr);
    }

    printf("MT-MPI test is done\n");
  }

  MPI_Finalize();

  return (0);
}

/* end file */
```

3.8 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 one (single process, multiple data), each instance may be executing a different line of code at any instant. Prism, part of the optional Sun HPC Parallel Development Environment (PDE) that may be available at your site, eases the debugging process considerably.

Prism is recommended for debugging in the Sun HPC Software environment. However, if you need to debug multithreaded Sun MPI programs at the thread level, you should see Section 3.8.3, “Debugging With dbx.” See also Section 3.8.4, “Debugging With MPE,” if you are using the multiprocessing environment (MPE) from Argonne National Laboratory.

3.8.1 Setting MPI_INIT_TIMEOUT

Sun MPI has timeouts built into the software to help detect when there are network interface problems while starting an MPI task. However, these timeouts can be triggered erroneously when you are debugging programs, such as when using Prism or dbx, and should therefore be disabled prior to using a debugger on a Sun MPI program. The environment variable `MPI_INIT_TIMEOUT` can be used to set or disable the timeout time. When `MPI_INIT_TIMEOUT` is set to a positive integer, the timeout value is set to that time in seconds. When it is set to 0 or a negative integer, the timeout is disabled. The default value is 600 seconds (10 minutes).

For example, to disable timeouts (in a C shell):

```
% setenv MPI_INIT_TIMEOUT -1
```

Again in a C shell, to set timeouts to 5 minutes:

```
% setenv MPI_INIT_TIMEOUT 300
```

3.8.2 Debugging With Prism

The Prism development environment, which is part of the optional Sun HPC PDE, may be available at your site. For complete information on Prism, see the *Prism User's Guide*, especially the chapter on MP Prism. Unless you are using multiple threads in your program, use MP Prism to debug your Sun MPI program. This section is a brief summary of the information in that chapter.

Note – To use MP Prism to debug a Sun MPI program, the program has to be written in the SPMD style — that is, all processes that make up a Sun MPI program must be running the same executable.

It is possible to use Prism to debug multiprocess programs; it requires attaching a Prism debugger to processes on individual nodes. However, organizing such a debugging session is by no means simple.

3.8.2.1 Starting Up MP Prism

Note – To debug a Sun MPI program with MP Prism, you need to have compiled your program using one of the compilers included in the Sun Performance WorkShop Fortran suite of tools.

To use MP Prism, you must be logged in to the Sun HPC System (see Section 3.3).

To start Prism on a Sun MPI program, use the `-np` option to specify how many processes you want to start (which determines how many copies of the program will run). For example,

```
% prism -np 4 myprog
```

starts four copies of `myprog`. You can specify where the processes are to start by setting the `TMRUN_FLAGS` environment variable; Prism also supports a subset of `tmrun` options. For example,

```
% prism -np 4 -p Dedicated myprog
```

starts the processes on the partition Dedicated.

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

Note – To run graphical Prism, you must be running Solaris 2.5.1 or 2.6 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.

One important feature of MP Prism 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*.

3.8.3 Debugging With dbx

To debug your multithreaded program at the thread level, you can use dbx. The following example illustrates this method of debugging.

▼ To Debug Threads With dbx

1. **Add a variable to block the process until you attach with dbx.**

   In this sample program, `simple-comm`, the *wait_for_dbx* variable is set to 1 to create a wait loop. It is placed before the function or functions to be debugged.

CODE EXAMPLE 3-3 Debugging a Multithreaded Sun MPI Program With dbx

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

void
main( int argc, char **argv )
```

```
{
  MPI_Comm comm_dup;
  int error;
  int wait_for_dbx = 1;

  if((error = (MPI_Init(&argc, &argv))) != MPI_SUCCESS) {
    printf("Bad Init\n");
    exit(-1);
  }

  while (wait_for_dbx);

  error = MPI_Comm_dup(MPI_COMM_WORLD, &comm_dup);
  if (error != MPI_SUCCESS) {
    printf("Bad Dup\n");
    exit(-1);
  }

  error = MPI_Comm_free(&comm_dup);
  if (error != MPI_SUCCESS) {
    printf("Bad Comm free\n");
    exit(-1);
  }

  MPI_Finalize();
}
```

2. **Compile the code, then run it.**

   After compiling the program, run it using `tmrun`. (See the *Sun HPC Software User's Guide* for more information.)

   ```
   % tmrun -np 4 simple-comm
   ```

3. **Identify the processes to which you want to attach the debugger.**

   Use `tmps -p` to obtain information about the processes in the task. The first line describes the task, and the following lines describe the processes by rank, process id (or *pid*), state, and the node where the process is running. (See the *Sun HPC Software User's Guide* for more about getting information about processes.)

   ```
   % tmps -p
     TID  NPROC  UID       STATE  AOUT
   t1387     4   tdd       RUN    simple-comm
         0  10838  RUN     dev-node31
         1  10842  RUN     dev-node31
         2  11443  RUN     dev-node30
         3  11449  RUN     dev-node30
   ```

4. **Attach the debugger to the processes that you would like to debug.**

   Attach the debugger to the processes. If you would like to debug only a subset of the processes, you must set up a conditional in such a way that the `while` statement is executed in only the process(es) that you will debug.

```
% dbx simple-comm 10838
Attached to process 10838 with 2 LWPs
t@1 (l@1) stopped in main at line 18 in file "simple-comm.c"
   18     while (wait_for_dbx);
```

5. **Set your variable such that it will allow the process to unblock.**

   At the `dbx` prompt, use `assign` to change the value of the variable (here *wait_for_dbx*) and, hence, unblock the processes.

```
(dbx) assign wait_for_dbx = 0
```

6. **Debug the processes.**

   After you have attached and set the instrumentation code appropriately, you can start debugging the processes as you normally would with `dbx`.

3.8.4 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 Section 2.4, “MPE: Extensions to the Library.”

3.9 Configuration and Tuning

3.9.1 Ratio of Processes to Processors

When running Sun MPI programs, the number of processes should be less than or equal to the number of processors in the partition where you are running. If you run a program with a larger number of processes than the number of processors you have available, using the `-W` tag to `tmrun` or `tmsub` (causing the processes to *wrap*), then more than one process will run on a processor, causing significant performance degradation.

You may find the performance achieved when wrapping processes acceptable in some circumstances, such as in developing and debugging your program. See the *Sun HPC Software User's Guide* for more information about options to `tmrun` and `tmsub`.

3.9.2 Shared Memory Configuration

Sun MPI uses shared memory as a buffer zone for passing MPI messages when two ranks on the same node are communicating. Using shared memory in these circumstances puts system bus and memory at the disposal of communicating processes and results in high data throughput and low communication latencies.

Shared-memory communication is coordinated through the use of *mailboxes*. Each receiver has a mailbox reserved for each sender. Thus, for an *N*-process task, there are N^2 mailboxes. This square layout allows immediate point-to-point communication when a mailbox is empty. The sender writes data to a reserved area of shared memory, then sets a flag in the receivers mailbox. The receiver notices that a message is pending, receives it, and clears the flag.

Messages themselves are passed through one of two different regions of shared memory, depending on their size. Any message up to 1024 bytes is by default a *short message* and passes through a private region bound to a mailbox. A *long message* must be allocated a temporary message buffer from a shared pool of buffer space. The allocated buffer needs to be large enough to contain the entire message in progress. When the message has been received, the temporary buffer is returned to the shared pool.

The size of the shared memory file is a configuration consideration. It needs to be large enough to accommodate the needs of a Sun MPI task's peak message-passing activity, while at the same time recognizing that other users are also making demands on memory resources. By default, Sun MPI creates a shared memory file whose size should satisfy the needs of a broad range of tasks without violating system memory requirements. For those instances where a task requires more (or less) shared memory, a set of shell environment variables is available to tailor the size to a particular need.

The shared memory file is split into three regions: the *mailbox region*, the *short-message region*, and the *long-message region*. Each region is a different size:

- The size of the mailbox region can be determined exactly based on the number *N* of on-node processes participating in the task. It is equal to $N^2 * 384$ bytes.
- The default size of the short-message region is equal to $N^2 * 3072$ bytes.
- The default size of the long-message region is determined by a series of stepwise functions, the slopes of which decrease at selected values of *N*:
  - For $N = 2$, the size is 34,952,448 bytes.
  - For $N = 16$, the size is 83,869,696 bytes.

- For N = 32, the size is 314,507,264.
- For N = 64, the size is 838,598,696 bytes.
- For N = 256, the size is 1,883,242,496 bytes.

For values of N falling between these points, the long-message region size decreases in a linear way with N^2.

When it is necessary to specify a custom shared-memory file size, the sizes of both the short-message region and the long-message region can be varied. Changes to either or both can be made for the same run. For versions of Solaris prior to Solaris 2.6, there is a file size limit of 2 Gbytes. When specifying custom shared-memory sizes, some care should be taken if the resultant file size approaches 2 Gbytes.

The short-message region size is modified by setting the shell environment variable `MPI_SHORTMSGSIZE`. When set, the short message region will be equal to $N^2 * 3 *$ `MPI_SHORTMSGSIZE`, where N is the number of on-node processes in the task.

The long-message region size is modified by setting one of two shell environment variables: `MPI_UNITMEMSIZE` or `MPI_GLOBMEMSIZE`. `MPI_GLOBMEMSIZE` is a global value. Its setting absolutely specifies the size of the long-message region. `MPI_UNITMEMSIZE` is a per-process value. It determines the long-message region size according to the expression $N^2 *$ `MPI_UNITMEMSIZE`, where N is the number of on-node processes in the task. You may set both `MPI_UNITMEMSIZE` and `MPI_GLOBMEMSIZE`, but if you do, only `MPI_GLOBMEMSIZE` will have an effect. See Section 3.5.1.2, “Using MPI_GLOBMEMSIZE and MPI_UNITMEMSIZE,” for more details about setting these environment variables.

Another factor to consider is the amount of swap space configured. Since the shared file is created in `/tmp`, it shares its backing store with swap. If the shared file takes up too large a portion of swap, the task will exit and an error message will appear inthe originating shell (see “Insufficient Swap Space” on page 3-23). This condition can easily be remedied by using the `swap -a` shell command to add swap space. A reasonable rule of thumb is to use no more than 85% of the available swap space. The shell command `swap -l` will report available swap.

3.9.3 Shared Memory Performance

Under light or moderate message-passing loads, short-message transfers and long-message transfers should show about the same nominal performance. However, there is a tradeoff in using these two transfer modes. Because of the inherent message-passing overhead, it is more efficient to send large messages. However, as load increases, using a short-message buffer will provide a performance advantage. This is because short-message buffer accesses are private and, therefore, are not subject to the contention for shared-memory file space that can occur with long-message buffer accesses.

The implication of this short message–long message differential is that overall message-passing performance will depend on message-size distribution and message rate. Message-size distribution will determine how often a short-message path is taken rather than a long-message path. Message-size distribution and message rate will determine the time it takes to service a request for a buffer from the large-message pool. In particular, the number of messages with sizes grouping around the maximum short-message size is important. If message sizes are uniform or tend to cluster around a size, then `MPI_SHORTMSGSIZE` should be set slightly larger than the average size.

When setting `MPI_SHORTMSGSIZE`, overall shared-memory page size should be kept in mind. With this consideration, an attempt should be made to understand the message-size distribution. If the message-size distribution justifies it, `MPI_SHORTMSGSIZE` should be set large enough to take advantage of the efficiencies of large messages. For simple “ping-pong” message exchanges, an `MPI_SHORTMSGSIZE` of 16 Kbytes provides a smooth performance transition between short and long messages.

As described in the previous section and in Section 3.5.1, “Shared Memory Allocation,” two variables are available for setting the long-message region size: `MPI_GLOBMEMSIZE` and `MPI_UNITMEMSIZE`. Each provides a different way of thinking about how much room will be needed for long messages. If no information about message-size distribution is available and the default size is inadequate, using `MPI_GLOBMEMSIZE` is the best way to set the size of the large-message region. An error message (see Section 3.10.1, “Troubleshooting Shared Memory Allocation”) will indicate the available amount of large-message memory and the size of the request that exceeded the limit. You can use that request size to help judge how much to increase the large-message region size.

If some information about message-size distribution and traffic patterns is available, `MPI_UNITMEMSIZE` may provide a more natural way to set the size of the large-message region. In effect, setting `MPI_UNITMEMSIZE` guarantees that every on-node process will be able to send an `MPI_UNITMEMSIZE`-size message simultaneously to every other on-node process without running out of large-message memory.

See Section 3.5.1.2, “Using MPI_GLOBMEMSIZE and MPI_UNITMEMSIZE,” for more details about setting these environment variables.

3.10 Troubleshooting

This section describes some common problem situations, resulting error messages, and suggestions for fixing the problems. TABLE 3-1 lists standard error return values you may encounter in your Sun MPI programs. For the most recent information about Sun MPI, see the *Sun HPC Software 2.0 Release Notes*.

3.10.1 Troubleshooting Shared Memory Allocation

The shared-memory segment used by Sun MPI is a static and finite resource. During the MPI initialization phase, the size of the segment is determined based on user-set environment variables or default settings that scale with the number of processes on a node in the same task. Once allocated, the size of the shared-memory segment does not change. Problems with shared memory use may occur if certain size limits are violated or if a request for shared memory would exceed the amount available.

The 2-Gbyte Limit

In order to be compatible will all supported versions of Solaris, Sun MPI puts an upper bound of 2 Gbytes – 1 byte on the size of a shared-memory file. In the default case, this limit will never be reached. If, however, the user has set `MPI_GLOBMEMSIZE`, `MPI_UNITMEMSIZE`, or `MPI_SHORTMSGSIZE` (or some combination of these environment variables), it is possible to request a file larger than the limit. If the limit is exceeded, an error message similar to the following will be seen at the originating shell and the task will exit:

```
TMTL_shmem_connection_init: [node4–1]
    Sun MPI shared memory limit exceeded.
    Limit:           2147481855 (0x7ffff8ff) bytes.
    Requested:       12034954240 (0x2cd56d400) bytes.
    Adjust environment variable MPI_SHORTMSGSIZE.
    Consult the Sun MPI Guide on shared memory allocation.
```

If this error is seen, environment variable sizes should be reduced so that the requested size will fall within the 2 Gbytes – 1 byte limit. See Section 3.9.2, “Shared Memory Configuration,” for detailed information about calculating the size of the shared-memory file.

Insufficient Swap Space

The shared-memory segment exists as a file in the `tmpfs` area `/tmp`, which is based on non-reserved physical memory and swap space. Space available in `/tmp` for shared memory use will vary dynamically depending on total application demand on physical memory, how much of swap is being used as backing store, and other temporary files resident in `/tmp`. If a request for a shared memory segment exceeds the amount of memory available in `/tmp`, an error message similar to the following will be seen at the originating shell and the task will exit:

```
TMTL_shmem_connection_init: [node4–1]
   Not enough space left for shared memory files
   Free space available on filesystem:   1799831552 bytes
```

```
    Free space available for shmem files: 1609656730 bytes
    Space requested:                      2000014080 bytes
    Consult the Sun MPI Guide on shared memory allocation.
```

If this error is seen, swap space should be increased or more RAM added. Increasing swap space will allow the task to run, but performance may suffer. If performance is a problem here, try adding RAM.

Insufficient Buffer Pool Space

Under stressful conditions of heavy message traffic and large message sizes, it is possible to exhaust the memory available in the shared memory segment. If a request under these circumstances results in this possibility, Sun MPI first backs off from the request to give other processes a chance to free the shared memory they are using. If after the backoff, sufficient memory is still unavailable, an error message similar to the following will be seen at the originating shell, and the task will exit:

```
[node4–1] Sun MPI Error:
    Shared memory buffer pool exhausted.
    Size of the shared memory buffer pool =   10000000 bytes
    Failed for request size               =   16777216 bytes
    Insufficient buffer pool space prevents request completion.
    Use environment variables MPI_GLOBMEMSIZE or MPI_UNITMEMSIZE
    to increase buffer pool size. See the MPI(3SunMPI) man page or
    the Sun MPI Guide for more information.
```

If this error is seen, the shared memory segment size should be increased by setting either `MPI_GLOBMEMSIZE` or `MPI_UNITMEMSIZE`.

Note – In some cases, you may see this error message when the reported request size is smaller than the size of the reported shared-memory buffer pool. This results from congestion caused by other messages still using the shared-memory buffer pool such that there is not enough memory for the requested size. The solution is nonetheless the same: you must increase the shared-meory segment size for your program to run.

3.10.2 Standard Error Values

Listed below are the error return values you may encounter in your MPI programs. Error values may also be found in `mpi_errno.h`.

TABLE 3-1 Sun MPI Standard Error Values

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.

CHAPTER 4

Sun MPI I/O

For Sun MPI, file I/O is a subset of the routines included in the MPI-2 standard. (See the section "On the World Wide Web" on page ix of the preface for more information about MPI-2.) 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 compatible with MPI.

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 in effect accomplish this coordination. Furthermore, MPI I/O allows multiple simultaneous access requests to be made to take advantage of Sun HPC's parallel file system, PFS. It is currently the only application programming interface through which users can access Sun HPC's PFS.

4.1 Other Sun HPC I/O

Sun HPC provides three options for I/O:

- If you use the Sun HPF (High Performance Fortran) compiler, your compiled program will call Sun MPI I/O for you. For more information about how Sun HPF uses MPI I/O, see the *Sun HPF Guide*.
- You can also make file system calls to Solaris or use Solaris raw disk calls. (See your Solaris documentation for more information.) Some file I/O considerations are discussed in the *Sun HPC Software User's Guide*.
- The third option is to use the Sun MPI I/O library.

Note – A direct interface to Sun HPC's PFS (parallel file system) is not available to the user in this release. Currently, the only way to access PFS is through Sun's implementation of MPI I/O, Sun HPF, or the PFS command-line utilities.

4.2 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 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. See Section 4.2.1, "Data Partitioning and Data Types."

4.2.1 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 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 Section 2.2.4, "Data Types.") 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 partitioned 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 file and back again. You create derived data types that describe how the data is arranged in each process's memory and how it should be arranged in that process's part of the disk file. Sun MPI I/O takes care of managing the multiple reads and writes of all the processes.

The Sun MPI I/O routines are described in Section 4.2.3, "Routines." But first, to be able to define data layout, you will need to understand some basic MPI I/O data-layout concepts. Section 4.2.2, "Definitions," explains some of the fundamental terms and concepts.

4.2.2 Definitions

The following terms are used to describe partitioning data among processes. FIGURE 4-1 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 relative to the beginning of a file. The displacement defines the location where a view begins (see 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.

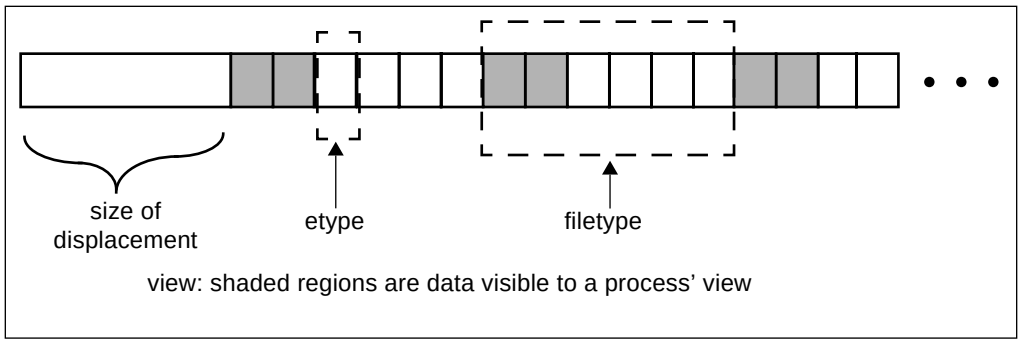

FIGURE 4-1 Displacement, the Elementary Data Type, the File Type, and the View

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

4.2.3 Routines

This release of Sun MPI includes 26 MPI I/O routines, all of which are defined in Chapter 9, "I/O," of the MPI-2 specification. (See the preface for information about this specification.)

Code examples that use many of these routines are provided in Section 4.2.3.5, "Sample Code," starting on page 4-11.

4.2.3.1 File Manipulation

`MPI_File_open`
`MPI_File_close`
`MPI_File_delete`
`MPI_File_set_size`
`MPI_File_get_size`
`MPI_File_get_group`
`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`.

Note – The `MPI_File_open` interface allows the user to pass information to MPI I/O about data layout and usage via the *info* argument. Although passing such information is not supported for this release, the *info* argument must be supplied for compatibility with the MPI-2 specification. Its inclusion will allow support for passing user-supplied information to MPI I/O in future releases.

`MPI_File_delete` deletes a specified file, provided it is not currently open by any process.

Note – For Fortran users: While using the Fortran interface to MPI I/O, if you are working with file names that have trailing spaces, the spaces will be deleted from them when they are acted on by the `MPI_File_open` or `MPI_File_delete` routine. This behavior follows Sun F77 and F90 conventions. (The C interface to MPI I/O does not modify file names.)

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 the size is set to be larger than the current file size, the file size becomes the set size. When using `MPI_File_set_size` on a PFS file, however, you must set the file to length 0. No other values are supported in this release.

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

See all the code examples in Section 4.2.3.5 for sample code using some of these routines.

Note – For Fortran users: Because of the way Sun Performance Workshop F77 handles `INTEGER*8`, for the MPI I/O routines that take an `MPI_Offset(INTEGER*8)`, there is an alternative Fortran interface in addition to the standard interface. The alternative takes a `REAL*8` offset and is distinguished by the presence of a `D` (for double) at the start of the function name:

Conventional Name	Alternative Name
`MPI_File_set_size`	`MPI_DFile_set_size`
`MPI_File_get_size`	`MPI_DFile_get_size`

These `D` routines are available only from Fortran.

4.2.3.2 File Views

`MPI_File_set_view`
`MPI_File_get_view`

The `MPI_File_set_view` routine changes the process's view of the data in the file, specifying its displacement, elementary data type, and file type, as well as setting the independent 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.

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

Note – For Fortran users: Because of the way Sun Performance Workshop F77 handles `INTEGER*8`, for the MPI I/O routines that take an `MPI_Offset(INTEGER*8)`, there is an alternative Fortran interface in addition to the standard interface. The alternative takes a `REAL*8` offset and is distinguished by the presence of a `D` (for double) at the start of the function name:

Conventional Name	Alternative Name
`MPI_File_set_view`	`MPI_DFile_set_view`
`MPI_File_get_view`	`MPI_DFile_get_view`

These `D` routines are available only from Fortran.

See the code examples in Section 4.2.3.5 for sample code using `MPI_File_set_view`.

4.2.3.3 Data Access

The 14 data-access routines in this section are blocking routines. (Nonblocking MPI I/O routines have not been implemented for this release.) The three methods of file positioning used for data access are by:

- Explicit offset
- Individual file pointer
- Shared file pointer

See the following subsections for a more detailed discussion of each of these methods. Sample code illustrating the use of these routines can be found in Section 4.2.3.5, "Sample Code," starting on page 4-11.

Data Access With Explicit Offsets

```
MPI_File_read_at
MPI_File_read_at_all
MPI_File_write_at
MPI_File_write_at_all
```

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 data items 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 via 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_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.

Note – The type signatures of the file type and the buffer type must match (however, this is not checked).

See CODE EXAMPLE 4-1, starting on page 4-12, for sample code using some of these routines.

Note – For Fortran users: Because of the way Sun Performance Workshop F77 handles `INTEGER*8`, for the MPI I/O routines that take an `MPI_Offset(INTEGER*8)`, there is an alternative Fortran interface in addition to the standard interface. The alternative takes a `REAL*8` offset and is distinguished by the presence of a `D` (for double) at the start of the function name:

Conventional Name	Alternative Name
`MPI_File_read_at`	`MPI_DFile_read_at`
`MPI_File_read_at_all`	`MPI_DFile_read_at_all`
`MPI_File_write_at`	`MPI_DFile_write_at`
`MPI_File_write_at_all`	`MPI_DFile_write_at_all`

These `D` routines are available only from Fortran.

Data Access With Individual File Pointers

`MPI_File_read`
`MPI_File_write`
`MPI_File_read_all`
`MPI_File_write_all`
`MPI_File_seek`
`MPI_File_get_position`

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; 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.

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 view 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, even if that location is a hole.

Note – For Fortran users: Because of the way Sun Performance Workshop F77 handles `INTEGER*8`, for the MPI I/O routines that take an `MPI_Offset(INTEGER*8)`, there is an alternative Fortran interface in addition to the standard interface. The alternative takes a `REAL*8` offset and is distinguished by the presence of a `D` (for double) at the start of the function name:

Conventional Name	Alternative Name
`MPI_File_seek`	`MPI_DFile_seek`
`MPI_File_get_position`	`MPI_DFile_get_position`

These `D` routines are available only from Fortran.

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

See CODE EXAMPLE 4-2, starting on page 4-14, for sample code using some of these routines.

Data Access With Shared File Pointers

```
MPI_File_read_ordered
MPI_File_write_ordered
MPI_File_seek_shared
MPI_File_get_position_shared
```

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 shared file pointers, you can also use the current value of the shared file pointer to specify the offset of data accesses implicitly. These four routines use and update only the shared file pointer maintained by the system; 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.
- The multiple calls (one for each process in the communicator group) affect the shared file pointer routines as if the calls were serialized.

After a shared file pointer operation is initiated, the file pointer 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_ordered and MPI_File_write_ordered are collective routines that must be called by all processes in the communicator group associated with the file handle. After all the processes in the group have issued their respective calls, MPI_File_read_ordered attempts to read from the file associated with the file handle a specified total number of data items of a particular data type into a specified user buffer. Similarly, MPI_File_write_ordered attempts to write data from a user buffer to a file. For each process, data is read (or written) at the position where the shared file pointer would be after all processes with ranks within the group lower than this process's rank had read (or written) their data. These routines return the number of elements of the particular data type read or written in the status structure. The shared file pointer is updated by the amount of data requested by all processes of the group.

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 in the communicator group 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.

Note – For Fortran users: Because of the way Sun Performance Workshop F77 handles INTEGER*8, for the MPI I/O routines that take an MPI_Offset(INTEGER*8), there is an alternative Fortran interface in addition to the standard interface. The alternative takes a REAL*8 offset and is distinguished by the presence of a D (for double) at the start of the function name:

Conventional Name	Alternative Name
MPI_File_seek_shared	MPI_DFile_seek_shared
MPI_File_get_position_shared	MPI_DFile_get_position_shared

These D routines are available only from Fortran.

MPI_File_get_position_shared returns the current position of the shared file pointer relative to the current displacement and file type. It also returns the absolute position, in bytes, of the shared file pointer, ignoring the displacement and file type.

The noncollective routines MPI_File_read_shared and MPI_File_write_shared have not been implemented in this release.

See CODE EXAMPLE 4-3, starting on page 4-16, for sample code using some of these routines.

4.2.3.4 File Consistency and Semantics

`MPI_File_set_atomicity`
`MPI_File_get_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. A file is in nonatomic mode by default when it is opened. In nonatomic mode, `MPI_File_sync` can be used to assure weak consistency.

`MPI_File_set_atomicity` is a collective call that sets the consistency semantics for data-access operations, using the set of file handles created by one collective `MPI_File_open`. 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 on the set of file handles created by one collective `MPI_File_open`. Again, a Boolean flag indicates whether the atomic mode is set.

See CODE EXAMPLE 4-1, below, and CODE EXAMPLE 4-2, starting on page 4-14, for sample code using `MPI_File_set_atomicity`.

4.2.3.5 Sample Code

The following code examples demonstrate the use of many of the MPI I/O routines. In each example, each process opens a file using `MPI_File_open`, writes *NUM_INTS* integers to that file, reads back what was just written, then closes the file. Each of the three examples access data using a different type of file positioning: explicit, individual, and shared.

In these examples, since we intend to only outline clearly a program with a typical sequence of MPI I/O calls, we omit error checking for the purpose of clearer illustration of the sequence. However, in normal usage, the program should check whether each MPI I/O routine returns `MPI_SUCCESS`. If the routine does not return `MPI_SUCCESS`, then, based on the error code returned, the program should determine how to proceed.

In the first example, we open the file `pfs:/users/bob/foo` by calling `MPI_File_open`. The `pfs:` prefix specifies that we wish to access a PFS file. After the `pfs:` prefix, we must put the absolute pathname to access a PFS file. By calling `MPI_File_set_view`, we proceed to set the *etype* and *filetype* to `MPI_INT` (the MPI integer type) and the displacement to be 0.

Next, we perform the writes and the reads explicitly. Notice the parameter *offset* in the calls `MPI_File_write_at` and `MPI_File_read_at`. Each process sets *offset* to be *rank* * *NUM_INTS* so that each process in the parallel task will access a unique segment of the file (in this case, a segment with NUM_INTS consecutive integers). Thus, all the processes together write and read the first *comm_size* * *NUM_INT*} integers in the PFS file `pfs:/users/bob/foo`.

At the end of the program, we first check to see that the data read is the same as the data written. After that, we query the file size using `MPI_File_get_size` and check that the file size is what we expect it to be.

CODE EXAMPLE 4-1 Data Access With Explicit File Pointers

```
#include <stdio.h>

#include "mpi.h"
#include "mpio.h"

#define NUM_INTS 100

void
main( int argc, char **argv )
{
  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;

  MPI_Init(&argc, &argv);

  /* get this processor's rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

  /* communicator group MPI_COMM_WORLD opens file "foo"
     for reading and writing (and creating, if necessary) */
  MPI_File_open(MPI_COMM_WORLD, "pfs:/users/bob/foo",
    MPI_MODE_RDWR | MPI_MODE_CREATE, (int)NULL, &fh);

  /* 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;
  ftype = MPI_INT;
  info = (MPI_Info)NULL;
  MPI_File_set_view(fh, disp, etype, ftype, (char *)NULL, info);
```

```
  /* 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;
  MPI_File_write_at(fh, offset, buff1, NUM_INTS, buftype, &status);

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

  /* Check to see that each integer read from each location is
     the same as the integer written to that location. */
  for(i=0; i<NUM_INTS; i++) {
    if(buff1[i] != buff2[i])
      printf("Integer number %d differs\n", i);
  }

  MPI_File_get_size(fh, &file_size);

  if(file_size != (comm_size * NUM_INTS * sizeof(int)))
    printf("File size is not equal to the write size\n");

  MPI_File_close(&fh);

  MPI_Finalize();

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

In the second example, we access data using individual file pointers. First, we call `MPI_File_open` to open the UNIX file `foo` in the current working directory. (Note that for UNIX, we do not need to supply absolute path names. Also, for UNIX files, you could supply an optional `ufs:` prefix, but if no prefix is given, `MPI_File_open` assumes that the file is a UNIX file.) After we open the file, we set atomic mode by calling `MPI_File_set_atomicity`. Once again, we set the *etype* and *ftype* to be `MPI_INT` and the displacement to be 0.

Each process calls `MPI_File_seek` to set the individual file pointer to be *rank* * *NUM_INTS*, then calls `MPI_File_write` to write `NUM_INTS` consecutive integers starting at file position *rank* * *NUM_INTS*. Similarly, we use `MPI_File_seek` and `MPI_File_read` to read back the integers into a separate buffer.

After checking to see if the data read is the same as the data written as in the first example, we query the individual file position by calling `MPI_File_get_position`. Finally, the program concludes by closing the file by calling `MPI_File_close` and deleting the file by calling `MPI_File_delete`.

CODE EXAMPLE 4-2 Data Access With Individual File Pointers

```
#include <stdio.h>

#include "mpi.h"
#include "mpio.h"

#define NUM_INTS 100

void
main( int argc, char **argv )
{
  int i, rank;
  int *buff1, *buff2;
  MPI_File fh;
  MPI_Offset indiv_file_pos;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  /* get this processor's rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* communicator group MPI_COMM_WORLD opens file "foo"
     for reading and writing (and creating, if necessary) */
  MPI_File_open(MPI_COMM_WORLD, "foo", MPI_MODE_RDWR | MPI_MODE_CREATE,
    (MPI_Info)NULL, &fh);

  /* Set the atomicity mode to be cautious so that a process doesn't try
     to read while another writes. */
  MPI_File_set_atomicity(fh, 1);

  /* Set the file view which tiles the file type MPI_INT, starting
     at displacement 0.  In this example, the etype is also MPI_INT.  */
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, (char *)NULL, (MPI_Info)NULL);

  /* 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 individual file pointer to the start of this process's section. */
  MPI_File_seek(fh, rank*NUM_INTS, MPI_SEEK_SET);
```

```
  /* Set the buffer type to also be MPI_INT, then write the buffer (buff1)
     starting at the current value of the individual file pointer. */
  MPI_File_write(fh, buff1, NUM_INTS, MPI_INT, &status);

  /* Reset the individual file pointer to offset 0. */
  MPI_File_seek(fh, rank*NUM_INTS, MPI_SEEK_SET);

  /* Allocate another buffer (buff2) to read into, then read NUM_INTS
     integers into this buffer.  */
  buff2 = (int *)malloc(NUM_INTS*sizeof(int));
  MPI_File_read(fh, buff2, NUM_INTS, MPI_INT, &status);

  /* Check to see that each integer read from each location is
     the same as the integer written to that location. */
  for(i=0; i<NUM_INTS; i++) {
    if(buff1[i] != buff2[i])
      printf("Integer number %d differs: buff1: %d  buff2: %d\n",
        i, buff1[i], buff2[i]);
  }

  MPI_File_get_position(fh, &indiv_file_pos);
  printf("The individual file position for process %d is now %d\n",
     rank, (int)indiv_file_pos);

  MPI_File_close(&fh);

  MPI_File_delete("foo", (MPI_Info)NULL);

  MPI_Finalize();

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

In the third example, each process writes and reads the same amount of data from the file as in the first two examples, but the writes and reads occur collectively using the shared file pointer. Before writing or reading, we call `MPI_File_seek_shared` to set the shared file pointer to be 10. Then, each process calls `MPI_File_write_ordered` to write the data. Since this routine is collective, each process must call it. On the collective write, the write requests are satisfied in the order of rank. That is, process 0 writes the first `NUM_INTS` integers, process 1 writes the next `NUM_INTS` integers, and so on. Similarly, each process calls `MPI_File_seek_shared` and `MPI_File_read_ordered` to read back the same data. Finally, we call `MPI_File_get_position_shared` to examine the position of the shared file position after the read has occurred.

CODE EXAMPLE 4-3 Shared File Pointers and Collective Data Access

```
#include <stdio.h>

#include "mpi.h"
#include "mpio.h"

#define NUM_INTS 100

void
main( int argc, char **argv )
{
  int i, rank;
  int *buff1, *buff2;
  MPI_File fh;
  MPI_Offset shared_file_pos;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  /* get this processor's rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* communicator group MPI_COMM_WORLD opens file "foo"
     for reading and writing (and creating, if necessary) */
  MPI_File_open(MPI_COMM_WORLD, "foo", MPI_MODE_RDWR | MPI_MODE_CREATE,
    (MPI_Info)NULL, &fh);

  /* Set the file view which tiles the file type MPI_INT, starting
     at displacement 0.  In this example, the etype is also MPI_INT.  */
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, (char *)NULL, (MPI_Info)NULL);

  /* 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 shared file pointer to offset 10. */
  MPI_File_seek_shared(fh, 10, MPI_SEEK_SET);

  /* Set the buffer type to also be MPI_INT, then write the buffer (buff1)
     starting at the current value of the shared file pointer.

     Note #1: upon opening or setting the view for a file, the shared file
     pointer is initially set to 0. After the previous call to
     MPI_File_seek_shared, we have set the shared file pointer to be 10.

     Note #2: since the following write and the subsequent read are
     collective operations, each process in the communicator group
```

```
    writes/reads NUM_INTS integers from the file at offset rank * NUM_INTS
    + 10,where rank is the process's rank within the communicator group */
  MPI_File_write_ordered(fh, buff1, NUM_INTS, MPI_INT, &status);

  /* Reset the shared file pointer to offset 0. */
  MPI_File_seek_shared(fh, 10, MPI_SEEK_SET);

  /* Allocate another buffer (buff2) to read into, then read NUM_INTS
     integers into this buffer.  */
  buff2 = (int *)malloc(NUM_INTS*sizeof(int));
  MPI_File_read_ordered(fh, buff2, NUM_INTS, MPI_INT, &status);

  /* Check to see that each integer read from each location is
     the same as the integer written to that location. */
  for(i=0; i<NUM_INTS; i++) {
    if(buff1[i] != buff2[i])
      printf("Integer number %d differs: buff1: %d  buff2: %d\n",
         i, buff1[i], buff2[i]);
  }

  MPI_File_get_position_shared(fh, &shared_file_pos);
  printf("The shared file position for process %d is now %d\n", rank,
     (int)shared_file_pos);

  MPI_File_close(&fh);

  MPI_Finalize();

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

4.2.4 MPI I/O Profiling Interface

Sun MPI and MPI I/O do not provide profiling libraries, but do provide the mechanism that permits users to write their own profiling libraries. Sun's implementation includes a version of each implemented MPI I/O routine with a `PMPI_` prefix in addition to an `MPI_` prefix.

For C, there are both profiling and nonprofiling interfaces, as specified in the MPI standard. For Fortran, there are wrappers. For MPI I/O, both profiling and nonprofiling sets of interfaces are in the library `-lmpi-io` (`libmpi-io.so`).

For more detailed information, see Section 2.2.9, "Profiling Interface," on page 2-8.

4.2.5 Error Handling

For the current release, the error handling of MPI I/O is very simple. If an error occurs, unlike the definition in MPI-2, MPI I/O routines return an error code (found in `/opt/SUNWhpc/include/mpi-io-errno.h` or in `/opt/SUNWhpc/include/mpi-errno.h`). Error classes and their meanings are listed in TABLE 4-1. They can also be found in `mpi-io-errno.h` (for C) and `mpi-iof-errno.h` (for Fortran). Note, however, that although MPI I/O makes extensive use of MPI, its method of handling errors is different from the standard MPI method (that is, on errors, MPI routines abort by default).

Thus, while Sun MPI I/O will not on its own abort your program if an error occurs, your program may abort if you have not used `MPI_Errhandler_set` to specify nondefault error handling for non-I/O MPI routines.

TABLE 4-1 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 (e.g., 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.

TABLE 4-1 Sun MPI I/O Error Classes *(Continued)*

Error Class	Value	Meaning
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_LASTCLASS	35	Last error class — always at end.

4.2.6 For More Information

For more information on MPI I/O, refer to the documents listed in the section “Related Publications,” on page viii of the preface.

APPENDIX **A**

Sun MPI and Sun MPI I/O Routines

The tables in this appendix list the routines and environment variables for 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.

A.1 Sun MPI Routines

TABLE A-1, starting on page A-7, lists the Sun MPI routines in alphabetical order. The following sections list the routines by functional category.

A.1.1 Point-to-Point Communication

A.1.1.1 Blocking Routines

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

A.1.1.2 Nonblocking Routines

```
MPI_Isend
MPI_Ibsend
MPI_Issend
MPI_Irsend
MPI_Irecv
```

A.1.1.3 Communication Buffer Allocation

```
MPI_Buffer_attach
MPI_Buffer_detach
```

A.1.1.4 Status Data Structure

```
MPI_Get_count
MPI_Get_elements
```

A.1.1.5 Persistent (Half-Channel) Communication

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

A.1.1.6 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
```

A.1.1.7 Probing for Messages (Blocking / Nonblocking)

```
MPI_Probe
MPI_Iprobe
```

A.1.1.8 Packing and Unpacking Functions

```
MPI_Pack
MPI_Pack_size
MPI_Unpack
```

A.1.1.9 Derived Data Type Constructors and Functions

```
MPI_Type_commit
MPI_Type_free
MPI_Type_contiguous
MPI_Type_vector
MPI_Type_hvector
MPI_Type_indexed
MPI_Type_hindexed
MPI_Type_struct
MPI_Type_lb
MPI_Type_ub
MPI_Address
MPI_Type_extent
MPI_Type_size
MPI_Type_count
```

A.1.2 Collective Communication

A.1.2.1 Barrier

```
MPI_Barrier
```

A.1.2.2 Broadcast

```
MPI_Bcast
```

A.1.2.3 Processor Gather / Scatter

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

A.1.2.4 Global Reduction / Scan Operations

```
MPI_Reduce
MPI_Allreduce
MPI_Reduce_scatter
MPI_Scan
MPI_Op_create
MPI_Op_free
```

A.1.3 Groups, Contexts, and Communicators

A.1.3.1 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
```

A.1.3.2 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
MPI_Keyval_free
MPI_Attr_put
MPI_Attr_get
MPI_Attr_delete
```

A.1.4 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
```

A.1.5 Environmental Inquiry Functions and Profiling

A.1.5.1 Startup and Shutdown

```
MPI_Init
MPI_Finalize
MPI_Initialized
MPI_Abort
MPI_Get_processor_name
```

A.1.5.2 Error Handler Functions

```
MPI_Errhandler_create
MPI_Errhandler_set
MPI_Errhandler_get
MPI_Errhandler_free
MPI_Error_string
MPI_Error_class
```

A.1.5.3 Timers

```
MPI_Wtime
MPI_Wtick
```

A.1.5.4 Profiling

```
MPI_Pcontrol
```

TABLE A-1 Sun MPI Routines

Routine and C Syntax	Description
MPI_Abort(MPI_Comm *comm*, int *errorcode*)	Terminates MPI execution environment.
MPI_Address(void **location*, MPI_Aint **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_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*)	Deletes attribute value associated with a key.
MPI_Attr_get(MPI_Comm *comm*, int *keyval*, void **attribute_val*, int **flag*)	Retrieves attribute value by key.
MPI_Attr_put(MPI_Comm *comm*, int *keyval*, void **attribute_val*)	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.

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

Routine and C Syntax	Description
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 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.
MPI_Cart_sub(MPI_Comm *comm*, int **remain_dims*, MPI_Comm **comm_new*)	Partitions a communicator into subcommunicators, which form lower-dimensional Cartesian subgrids.
MPI_Cartdim_get(MPI_Comm *comm*, int **ndims*)	Retrieves Cartesian topology information associated with a communicator.
MPI_Comm_compare(MPI_Comm *comm1*, MPI_Comm *comm2*, int **result*)	Compares two communicators.
MPI_Comm_create(MPI_Comm *comm*, MPI_Group *group*, MPI_Comm **newcomm*)	Creates a new communicator from a group.
MPI_Comm_dup(MPI_Comm *comm*, MPI_Comm **newcomm*)	Duplicates an existing communicator with all its cached information.

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_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_size(MPI_Comm _comm_, int *_size_)	Determines the size of the group associated with a communicator.
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_)	Creates an MPI error handler.
MPI_Errhandler_free(MPI_Err-handler *_errhandler_)	Frees an MPI error handler.
MPI_Errhandler_get(MPI_Comm _comm_, MPI_Errhandler *_errhandler_)	Gets the error handler for a communicator.
MPI_Errhandler_set(MPI_Comm _comm_, MPI_Errhandler _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.

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

Routine and C Syntax	Description
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.
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_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_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_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.

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

Routine and C Syntax	Description
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.
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.

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

Routine and C Syntax	Description
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.
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_func-tion *_delete_fn_, int *_keyval_, void *_extra_state_)	Generates a new attribute key.
MPI_Keyval_free(int *_keyval_)	Frees attribute key for communicator cache attribute.
MPI_Op_create(MPI_User_function *_function_, int _commute_, MPI_Op *_op_)	Creates a user-defined combination function handle.
MPI_Op_free(MPI_Op *_op_)	Frees a user-defined combination function handle.
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_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.

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

Routine and C Syntax	Description
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_free(MPI_Request *_request_)	Frees a communication request object.
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 task 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.
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.

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

Routine and C Syntax	Description
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_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 canceled.
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.
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_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_extent(MPI_Datatype *datatype*, MPI_Aint **extent*)	Returns the extent of a data type, the difference between the upper and lower bounds of the data type.

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

Routine and C Syntax	Description
MPI_Type_free(MPI_Ddatatype *_datatype_)	Frees a data type.
MPI_Type_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_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_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_)	Returns the lower bound 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_)	Creates a struct data type.
MPI_Type_ub(MPI_Datatype _datatype_, MPI_Aint *_displacement_)	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.

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

Routine and C Syntax	Description
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.
double **MPI_Wtick**()	Returns the resolution of MPI_Wtime.
double **MPI_Wtime**()	Returns an elapsed time on the calling processor.

A.2 Sun MPI Environment Variables

Five environment variables allow you to fine-tune your Sun MPI environment.

TABLE A-2 Sun MPI Environment Variables

Environment Variable	Description	See Section(s)
MPI_GLOBMEMSIZE	Global value representing the overall quantity of memory allocated to the large-message shared memory area. Expressed in bytes as either a decimal or as a hexadecimal number. Default = unset (Sun MPI default size calculation). When set, overrides MPI_UNITMEMSIZE.	Section 3.5.1, Section 3.5.1.2, Section 3.9.2, Section 3.9.3, Section 3.10.1
MPI_INIT_TIMEOUT	Sets or disables timeout time, in seconds. Default = 600 seconds (10 minutes)	Section 3.8.1

TABLE A-2 Sun MPI Environment Variables *(Continued)*

Environment Variable	Description	See Section(s)
`MPI_SHORTMSGSIZE`	Per-process quantity corresponding to the limit on the size of the short-message buffer. Size of the area reserved for short messages = $3N^2$ * `MPI_SHORTMSGSIZE`, where *N* is the number of on-node processes in the MPI task. Default = 1024 bytes.	Section 3.5.1, Section 3.5.1.1, Section 3.9.2, Section 3.9.3, Section 3.10.1
`MPI_SPIN_LIMIT`	Limits aggressive polling associated with blocking receives. Set to an integer for number of times to poll before backing off. Default = 0, indicating no limit to polling.	Section 3.5.2
`MPI_UNITMEMSIZE`	Specifies size of large-message shared memory area based on per-process memory requirements. If set, the amount of memory reserved for large message passing = N^2 * `MPI_UNITMEMSIZE`, where *N* = number of on-node processes in the MPI task. Expressed in bytes as either a decimal or as a hexadecimal number. Default = unset (Sun MPI default size calculation). Overridden by `MPI_GLOBMEMSIZE` when both are set.	Section 3.5.1, Section 3.5.1.2, Section 3.9.2, Section 3.9.3, Section 3.10.1

A.3 Sun MPI I/O Routines

TABLE A-3, starting on page A-19, lists the Sun MPI I/O routines in alphabetical order. The following sections list the routines by functional category.

A.3.1 File Manipulation

```
MPI_File_open
MPI_File_close
MPI_File_delete
MPI_File_set_size
MPI_File_get_size
MPI_File_get_group
MPI_File_get_amode
```

A.3.2 File Views

```
MPI_File_set_view
MPI_File_get_view
```

A.3.3 Data access

A.3.3.1 Data Access With Explicit Offsets

```
MPI_File_read_at
MPI_File_read_at_all
MPI_File_write_at
MPI_File_write_at_all
```

A.3.3.2 Data Access With Individual File Pointers

```
MPI_File_read
MPI_File_write
MPI_File_read_all
MPI_File_write_all
MPI_File_seek
MPI_File_get_position
```

A.3.3.3 Data Access With Shared File Pointers

```
MPI_File_read_ordered
MPI_File_write_ordered
MPI_File_seek_shared
MPI_File_get_position_shared
```

A.3.4 File Consistency and Semantics

```
MPI_File_set_atomicity
MPI_File_get_atomicity
MPI_File_sync
```

TABLE A-3 Sun MPI I/O Routines

Routine and C Syntax	Description
MPI_File_close(MPI_File *_fh_)	Closes a file (collective).
MPI_File_delete(char *_filename_, MPI_Info _info_)	Deletes a file.
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_group(MPI_File _fh_, MPI_Group *_group_)	Returns process group of file.
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_view(MPI_File _fh_, MPI_Offset *_disp_, MPI_Datatype *_etype_, MPI_Datatype *_filetype_, char *_datarep_)	Returns process's view of data in file.
MPI_File_open(MPI_Comm _comm_, char *_filename_, MPI_Mode _amode_, MPI_Info _info_, MPI_File *_fh_)	Opens 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_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-3 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_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_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_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).
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_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_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).

Index

N

O

P

R

S

T