sgi®

Message Passing Toolkit (MPT) User's
Guide

New Features in This Manual

This update to the *Message Passing Toolkit (MPT) User's Guide* supports the MPT 1.18 release.

Major Documentation Changes

Updated information about multithreaded programming in "Multithreaded Programming" on page 12.

Added information about environment variables relevant to using the InfiniBand interconnect in "Tuning for Running Applications Across Multiple Hosts" on page 35.

Record of Revision

Version	Description
001	March 2004 Original Printing. This manual documents the Message Passing Toolkit implementation of the Message Passing Interface (MPI).
002	November 2004 Supports the MPT 1.11 release.
003	June 2005 Supports the MPT 1.12 release.
004	June 2007 Supports the MPT 1.13 release.
005	October 2007 Supports the MPT 1.17 release.
006	January 2008 Supports the MPT 1.18 release.

Contents

Figures

Tables

About This Manual

This publication documents the SGI implementation of the Message Passing Interface (MPI).

MPI consists of a library, which contains both normal and profiling entry points, and commands that support the MPI interface. MPI is a component of the SGI Message Passing Toolkit (MPT).

MPT is a software package that supports parallel programming on large systems and clusters of computer systems through a technique known as *message passing*. Systems running MPI applications must also be running Array Services software version 3.1 or later.

Related Publications and Other Sources

Material about MPI is available from a variety of sources. Some of these, particularly webpages, include pointers to other resources. Following is a grouped list of these sources.

The MPI standard:

*   As a technical report: University of Tennessee report (reference [24] from *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum).

*   As online PostScript or hypertext on the Web:

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

*   As a journal article in the *International Journal of Supercomputer Applications*, volume 8, number 3/4, 1994. See also *International Journal of Supercomputer Applications*, volume 12, number 1/4, pages 1 to 299, 1998.

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

Newsgroup: `comp.parallel.mpi`

SGI manual: *SpeedShop User's Guide*

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- You can also view man pages by typing man *title* on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
command	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
manpage(*x*)	Man page section identifiers appear in parentheses after man page names.
variable	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the

front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  SGI
  Technical Publications
  1140 East Arques Avenue
  Sunnyvale, CA 94085–4602

SGI values your comments and will respond to them promptly.

Introduction

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

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

- Message Passing Interface (MPI). MPI is a standard specification for a message passing interface, allowing portable message passing programs in Fortran and C languages.

- The SHMEM programming model. The SHMEM programming model is a distributed, shared-memory model that consists of a set of SGI-proprietary message-passing library routines. These routines help distributed applications efficiently transfer data between cooperating processes. The model is based on multiple processes having separate address spaces, with the ability for one process to access data in another process' address space without interrupting the other process. The SHMEM programming model is not a standard like MPI, so SHMEM applications developed on other vendors' hardware might or might not work with the SGI SHMEM implementation.

This chapter provides an overview of the MPI software that is included in the toolkit. This overview includes a description of the MPI-2 Standard features that are provided, a description of the basic components of MPI, and a description of the basic features of MPI. Subsequent chapters address the following topics:

MPI Overview

MPI was created by the Message Passing Interface Forum (MPIF). MPIF is not sanctioned or supported by any official standards organization. Its goal was to develop a widely used standard for writing message passing programs.

SGI supports implementations of MPI that are released as part of the Message Passing Toolkit. The MPI Standard is documented online at the following address:

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

MPI-2 Standard Compliance

The SGI MPI implementation is compliant with the 1.0, 1.1, and 1.2 versions of the MPI Standard specification. In addition, the following MPI-2 features (with section numbers from the MPI-2 Standard specification) are provided:

Feature	Section
MPI-2 parallel I/O	9
A subset of MPI-2 one-sided communication routines (put/get model)	6
MPI spawn functionality	5.3
MPI_Alloc_mem/MPI_Free_mem	4.11
Transfer of handles	4.12.4
MPI-2 replacements for deprecated MPI-1 functions	4.14.1
Extended language bindings for C++ and partial Fortran 90 support	10.1, 10.2.4
Generalized requests	4.5.2
New attribute caching functions	8.8

MPI Components

The MPI library is provided as a dynamic shared object (DSO) (a file with a name that ends in .so). The basic components that are necessary for using MPI are the libmpi.so library, the include files, and the mpirun command.

Profiling support is included in the libmpi.so library. Profiling support replaces all MPI_*Xx* prototypes and function names with PMPI_*Xx* entry points.

MPI Features

The SGI MPI implementation offers a number of significant features that make it the preferred implementation to use on SGI hardware:

- Data transfer optimizations for NUMAlink, including single-copy data transfer

- Use of hardware fetch operations (fetchops), where available, for fast synchronization and lower latency for short messages

- Optimized MPI-2 one-sided commands

- Interoperability with the SHMEM (LIBSMA) programming model

- High performance communication support for partitioned systems via XPMEM

Getting Started

This chapter provides procedures for building MPI applications. It provides examples of the use of the mpirun(1) command to launch MPI jobs. It also provides procedures for building and running SHMEM applications. It covers the following topics:

- "Compiling and Linking MPI Programs" on page 5

- "Compiling and Running SHMEM Applications" on page 8

Compiling and Linking MPI Programs

The default locations for the include files, the .so files, the .a files, and the mpirun command are pulled in automatically. Once the MPT RPM is installed as default, the commands to build an MPI-based application using the .so files are as follows:

- To compile using GNU compilers, choose one of the following commands:

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

- To compile programs with the Intel compiler, use the following commands:

  ```
  % efc -o myprog myprog.f -lmpi        (Fortran - version 7.1)
  % ecc -o myprog myprog.c -lmpi        (C - version 7.1)
  % ifort -o myprog myprog.f -lmpi       (Fortran - version 8)
  % icc -o myprog myprog.c -lmpi        (C - version 8)
  ```

  The libmpi++.so library is compatible with code generated by g++ 3.0 or later compilers, as well as Intel C++ 8.0 or later compilers. If compatibility with previous g++ or C++ compilers is required, the libmpi++.so released with MPT 1.9 (or earlier) must be used.

  Note: You must use the Intel compiler to compile Fortran 90 programs.

- To compile Fortran programs with the Intel compiler, enabling compile-time checking of MPI subroutine calls, insert a `USE MPI` statement near the beginning of each subprogram to be checked and use one of the following commands:

  ```
  % efc -I/usr/include -o myprog myprog.f -lmpi  (version 7.1)
  % ifort -I/usr/include -o myprog myprog.f -lmpi  (version 8)
  ```

  Note: The above command line assumes a default installation; if you have installed MPT into a non-default location, replace `/usr/include` with the name of the relocated directory.

Using `mpirun` to Launch an MPI Application

You must use the `mpirun` command to start MPI applications. For complete specification of the command line syntax, see the `mpirun`(1) man page. This section summarizes the procedures for launching an MPI application.

Launching a Single Program on the Local Host

To run an application on the local host, enter the `mpirun` command with the `-np` argument. Your entry must include the number of processes to run and the name of the MPI executable file.

The following example starts three instances of the `mtest` application, which is passed an argument list (arguments are optional):

```
% mpirun -np 3 mtest 1000 "arg2"
```

Launching a Multiple Program, Multiple Data (MPMD) Application on the Local Host

You are not required to use a different host in each entry that you specify on the `mpirun` command. You can launch a job that has multiple executable files on the same host. In the following example, one copy of `prog1` and five copies of `prog2` are run on the local host. Both executable files use shared memory.

```
% mpirun -np 1 prog1 : 5 prog2
```

Launching a Distributed Application

You can use the `mpirun` command to launch a program that consists of any number of executable files and processes and you can distribute the program to any number of hosts. A host is usually a single machine, or it can be any accessible computer running Array Services software. For available nodes on systems running Array Services software, see the `/usr/lib/array/arrayd.conf` file.

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

The examples in this section show various ways to launch an application that consists of multiple MPI executable files on multiple hosts.

The following example runs ten instances of the `a.out` file on `host_a`:

```
% mpirun host_a -np 10 a.out
```

When specifying multiple hosts, you can omit the `-np` option and list the number of processes directly. The following example launches ten instances of `fred` on three hosts. `fred` has two input arguments.

```
% mpirun host_a, host_b, host_c 10 fred arg1 arg2
```

The following example launches an MPI application on different hosts with different numbers of processes and executable files:

```
% mpirun host_a 6 a.out : host_b 26 b.out
```

Using MPI-2 Spawn Functions to Launch an Application

To use the MPI-2 process creation functions `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple`, you must specify the universe size by specifying the `-up` option on the `mpirun` command line. For example, the following command starts three instances of the `mtest` MPI application in a universe of size 10:

```
% mpirun -up 10 -np 3 mtest
```

By using one of the above MPI spawn functions, `mtest` can start up to seven more MPI processes.

When running MPI applications on partitioned Altix systems which use the MPI-2 `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` functions, it may be necessary to explicitly specify the partitions on which additional MPI processes may be launched. See the section "Launching Spawn Capable Jobs on Altix Partitioned Systems" on the `mpirun`(1) man page.

Compiling and Running SHMEM Applications

To compile SHMEM programs with a GNU compiler, choose one of the following commands:

```
% g++ compute.C -lsma
% gcc compute.c -lsma
% g77 -I/usr/include compute.f -lsma
```

To compile SHMEM programs with the Intel compiler, use the following commands:

```
% ecc compute.C -lsma    (version 7.1)
% ecc compute.c -lsma    (version 7.1)
% efc compute.f -lsma    (version 7.1)
% icc compute.C -lsma    (version 8)
% icc compute.c -lsma    (version 8)
% ifort compute.f -lsma    (version 8)
```

You must use `mpirun` to launch SHMEM applications. The `NPES` variable has no effect on SHMEM programs. To request the desired number of processes to launch, you must set the `-np` option on `mpirun`.

The SHMEM programming model supports single host SHMEM applications, as well as SHMEM applications that span multiple partitions. To launch a SHMEM application on more than one partition, use the multiple host `mpirun` syntax, such as the following:

```
% mpirun hostA, hostB -np 16 ./shmem_app
```

For more information, see the `intro_shmem`(3) man page.

Programming with SGI MPI

Portability is one of the main advantages MPI has over vendor-specific message passing software. Nonetheless, the MPI Standard offers sufficient flexibility for general variations in vendor implementations. In addition, there are often vendor specific programming recommendations for optimal use of the MPI library. This chapter addresses topics that are of interest to those developing or porting MPI applications to SGI systems. It covers the following topics:

- "Job Termination and Error Handling" on page 9

- "Signals" on page 10

- "Buffering" on page 11

- "Multithreaded Programming" on page 12

- "Interoperability with the SHMEM programming model" on page 12

- "Miscellaneous Features of SGI MPI" on page 13

- "Programming Optimizations" on page 13

- "Additional Programming Model Considerations" on page 16

Job Termination and Error Handling

This section describes the behavior of the SGI MPI implementation upon normal job termination. Error handling and characteristics of abnormal job termination are also described.

MPI_Abort

In the SGI MPI implementation, a call to MPI_Abort causes the termination of the entire MPI job, regardless of the communicator argument used. The error code value is returned as the exit status of the mpirun command. A stack traceback is displayed that shows where the program called MPI_Abort.

Error Handling

Section 7.2 of the MPI Standard describes MPI error handling. Although almost all MPI functions return an error status, an error handler is invoked before returning from the function. If the function has an associated communicator, the error handler associated with that communicator is invoked. Otherwise, the error handler associated with MPI_COMM_WORLD is invoked.

The SGI MPI implementation provides the following predefined error handlers:

- MPI_ERRORS_ARE_FATAL. The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_Abort were called by the process that invoked the handler.

- MPI_ERRORS_RETURN. The handler has no effect.

By default, the MPI_ERRORS_ARE_FATAL error handler is associated with MPI_COMM_WORLD and any communicators derived from it. Hence, to handle the error statuses returned from MPI calls, it is necessary to associate either the MPI_ERRORS_RETURN handler or another user defined handler with MPI_COMM_WORLD near the beginning of the application.

MPI_Finalize and Connect Processes

In the SGI implementation of MPI, all pending communications involving an MPI process must be complete before the process calls MPI_Finalize. If there are any pending send or recv requests that are unmatched or not completed, the application will hang in MPI_Finalize. For more details, see section 7.5 of the MPI Standard.

If the application uses the MPI-2 spawn functionality described in Chapter 5 of the MPI-2 Standard, there are additional considerations. In the SGI implementation, all MPI processes are connected. Section 5.5.4 of the MPI-2 Standard defines what is meant by connected processes. When the MPI-2 spawn functionality is used, MPI_Finalize is collective over all connected processes. Thus all MPI processes, both launched on the command line, or subsequently spawned, synchronize in MPI_Finalize.

Signals

In the SGI implementation, MPI processes are UNIX processes. As such, the general rule regarding handling of signals applies as it would to ordinary UNIX processes.

In addition, the SIGURG and SIGUSR1 signals can be propagated from the mpirun process to the other processes in the MPI job, whether they belong to the same process group on a single host, or are running across multiple hosts in a cluster. To make use of this feature, the MPI program must have a signal handler that catches SIGURG or SIGUSR1. When the SIGURG or SIGUSR1 signals are sent to the mpirun process ID, the mpirun process catches the signal and propagates it to all MPI processes.

Buffering

Most MPI implementations use buffering for overall performance reasons and some programs depend on it. However, you should not assume that there is any message buffering between processes because the MPI Standard does not mandate a buffering strategy. Table 3-1 on page 11 illustrates a simple sequence of MPI operations that cannot work unless messages are buffered. If sent messages were not buffered, each process would hang in the initial call, waiting for an MPI_Recv call to take the message.

Because most MPI implementations do buffer messages to some degree, a program like this does not usually hang. The MPI_Send calls return after putting the messages into buffer space, and the MPI_Recv calls get the messages. Nevertheless, program logic like this is not valid by the MPI Standard. Programs that require this sequence of MPI calls should employ one of the buffer MPI send calls, MPI_Bsend or MPI_Ibsend.

Table 3-1 Outline of Improper Dependence on Buffering

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

By default, the SGI implementation of MPI uses buffering under most circumstances. Short messages (64 or fewer bytes) are always buffered. Longer messages are also buffered, although under certain circumstances buffering can be avoided. For performance reasons, it is sometimes desirable to avoid buffering. For further information on unbuffered message delivery, see "Programming Optimizations" on page 13.

Multithreaded Programming

SGI MPI supports hybrid programming models, in which MPI is used to handle one level of parallelism in an application, while POSIX threads or OpenMP processes are used to handle another level. When mixing OpenMP with MPI, for performance reasons it is better to consider invoking MPI functions only outside parallel regions, or only from within master regions. When used in this manner, it is not necessary to initialize MPI for thread safety. You can use `MPI_Init` to initialize MPI. However, to safely invoke MPI functions from any OpenMP process or when using Posix threads, MPI must be initialized with `MPI_Init_thread`.

When using `MPI_Thread_init()` with the threading level `MPI_THREAD_MULTIPLE`, link your program with `-lmpi_mt` instead of `-lmpi`. See the `mpi`(1) man page for more information about compiling and linking MPI programs.

Interoperability with the SHMEM programming model

You can mix SHMEM and MPI message passing in the same program. The application must be linked with both the SHMEM and MPI libraries. Start with an MPI program that calls `MPI_Init` and `MPI_Finalize`.

When you add SHMEM calls, the PE numbers are equal to the MPI rank numbers in `MPI_COMM_WORLD`. Do not call `start_pes()` in a mixed MPI and SHMEM program.

When running the application across a cluster, some MPI processes may not be able to communicate with certain other MPI processes when using SHMEM functions. You can use the `shmem_pe_accessible` and `shmem_addr_accessible` functions to determine whether a SHMEM call can be used to access data residing in another MPI process. Because the SHMEM model functions only with respect to `MPI_COMM_WORLD`, these functions cannot be used to exchange data between MPI processes that are connected via MPI intercommunicators returned from MPI-2 spawn related functions.

SHMEM get and put functions are thread safe. SHMEM collective and synchronization functions are not thread safe unless different threads use different `pSync` and `pWork` arrays.

For more information about the SHMEM programming model, see the `intro_shmem` man page.

Miscellaneous Features of SGI MPI

This section describes other characteristics of the SGI MPI implementation that might be of interest to application developers.

stdin/stdout/stderr

In this implementation, `stdin` is enabled for only those MPI processes with rank 0 in the first `MPI_COMM_WORLD` (which does not need to be located on the same host as `mpirun`). `stdout` and `stderr` results are enabled for all MPI processes in the job, whether launched via `mpirun`, or via one of the MPI-2 spawn functions.

MPI_Get_processor_name

The `MPI_Get_processor_name` function returns the Internet host name of the computer on which the MPI process invoking this subroutine is running.

Programming Optimizations

This section describes ways in which the MPI application developer can best make use of optimized features of SGI's MPI implementation. Following recommendations in this section might require modifications to your MPI application.

Using MPI Point-to-Point Communication Routines

MPI provides for a number of different routines for point-to-point communication. The most efficient ones in terms of latency and bandwidth are the blocking and nonblocking `send`/`receive` functions (`MPI_Send`, `MPI_Isend`, `MPI_Recv`, and `MPI_Irecv`).

Unless required for application semantics, the synchronous send calls (`MPI_Ssend` and `MPI_Issend`) should be avoided. The buffered send calls (`MPI_Bsend` and `MPI_Ibsend`) should also usually be avoided as these double the amount of memory copying on the sender side. The ready send routines (`MPI_Rsend` and `MPI_Irsend`) are treated as standard `MPI_Send` and `MPI_Isend` in this implementation. Persistent requests do not offer any performance advantage over standard requests in this implementation.

Using MPI Collective Communication Routines

The MPI collective calls are frequently layered on top of the point-to-point primitive calls. For small process counts, this can be reasonably effective. However, for higher process counts (32 processes or more) or for clusters, this approach can become less efficient. For this reason, a number of the MPI library collective operations have been optimized to use more complex algorithms.

Some collectives have been optimized for use with clusters. In these cases, steps are taken to reduce the number of messages using the relatively slower interconnect between hosts.

Two of the collective operations have been optimized for use with shared memory. The barrier operation has also been optimized to use hardware fetch operations (`fetchops`). The `MPI_Alltoall` routines also use special techniques to avoid message buffering when using shared memory. For more details, see "Avoiding Message Buffering — Single Copy Methods" on page 15. Table 3-2 on page 14, lists the MPI collective routines optimized in this implementation.

Table 3-2 Optimized MPI Collectives

Routine	Optimized for Clusters	Optimized for Shared Memory
MPI_Alltoall	Yes	Yes
MPI_Barrier	Yes	Yes
MPI_Allreduce	Yes	No
MPI_Bcast	Yes	No

Note: These collectives are optimized across partitions by using the XPMEM driver which is explained in Chapter 6, "Run-time Tuning". These collectives (except `MPI_Barrier`) will try to use single-copy by default for large transfers unless `MPI_DEFAULT_SINGLE_COPY_OFF` is specified.

Using `MPI_Pack`/`MPI_Unpack`

While `MPI_Pack` and `MPI_Unpack` are useful for porting PVM codes to MPI, they essentially double the amount of data to be copied by both the sender and receiver. It

is generally best to avoid the use of these functions by either restructuring your data or using derived data types. Note, however, that use of derived data types may lead to decreased performance in certain cases.

Avoiding Derived Data Types

In general, you should avoid derived data types when possible. In the SGI implementation, use of derived data types does not generally lead to performance gains. Use of derived data types might disable certain types of optimizations (for example, unbuffered or single copy data transfer).

Avoiding Wild Cards

The use of wild cards (MPI_ANY_SOURCE, MPI_ANY_TAG) involves searching multiple queues for messages. While this is not significant for small process counts, for large process counts the cost increases quickly.

Avoiding Message Buffering — Single Copy Methods

One of the most significant optimizations for bandwidth sensitive applications in the MPI library is single copy optimization, avoiding the use of shared memory buffers. However, as discussed in "Buffering" on page 11, some incorrectly coded applications might hang because of buffering assumptions. For this reason, this optimization is not enabled by default for MPI_send, but can be turned on by the user at run time by using the MPI_BUFFER_MAX environment variable. The following steps can be taken by the application developer to increase the opportunities for use of this unbuffered pathway:

- The MPI data type on the send side must be a contiguous type.

- The sender and receiver MPI processes must reside on the same host or, in the case of a partitioned system, the processes may reside on any of the partitions.

- The sender data must be globally accessible by the receiver. The SGI MPI implementation allows data allocated from the static region (common blocks), the private heap, and the stack region to be globally accessible. In addition, memory allocated via the MPI_Alloc_mem function or the SHMEM symmetric heap accessed via the shpalloc or shmalloc functions is globally accessible.

Certain run-time environment variables must be set to enable the unbuffered, single copy method. For more details on how to set the run-time environment, see "Avoiding Message Buffering – Enabling Single Copy" on page 31.

Note: With the Intel 7.1 compiler, ALLOCATABLE arrays are not eligible for single copy, since they do not reside in a globally accessible memory region. This restriction does not apply when using the Intel 8.0/8.1 compilers.

Managing Memory Placement

SGI systems have a ccNUMA memory architecture. For single process and small multiprocess applications, this architecture behaves similarly to flat memory architectures. For more highly parallel applications, memory placement becomes important. MPI takes placement into consideration when laying out shared memory data structures, and the individual MPI processes' address spaces. In general, it is not recommended that the application programmer try to manage memory placement explicitly. There are a number of means to control the placement of the application at run time, however. For more information, see Chapter 6, "Run-time Tuning" on page 29.

Using Global Shared Memory

The MPT software includes the Global Shared Memory (GSM) Feature. This feature allows users to allocate globally accessible shared memory from within an MPI or SHMEM program. The GSM feature can be used to provide shared memory access across partitioned Altix systems and additional memory placement options within a single host configuration.

User-callable functions are provided to allocate a global shared memory segment, free that segment, and provide information about the segment. Once allocated, the application can use this new global shared memory segment via standard loads and stores, just as if it were a System V shared memory segment. For more information, see the GSM_Intro or GSM_Alloc man pages.

Additional Programming Model Considerations

A number of additional programming options might be worth consideration when developing MPI applications for SGI systems. For example, the SHMEM

programming model can provide a means to improve the performance of latency-sensitive sections of an application. Usually, this requires replacing MPI `send/recv` calls with `shmem_put/shmem_get` and `shmem_barrier` calls. The SHMEM programming model can deliver significantly lower latencies for short messages than traditional MPI calls. As an alternative to `shmem_get/shmem_put` calls, you might consider the MPI-2 `MPI_Put`/ `MPI_Get` functions. These provide almost the same performance as the SHMEM calls, while providing a greater degree of portability.

Alternately, you might consider exploiting the shared memory architecture of SGI systems by handling one or more levels of parallelism with OpenMP, with the coarser grained levels of parallelism being handled by MPI. Also, there are special ccNUMA placement considerations to be aware of when running hybrid MPI/OpenMP applications. For further information, see Chapter 6, "Run-time Tuning" on page 29.

Debugging MPI Applications

Debugging MPI applications can be more challenging than debugging sequential applications. This chapter presents methods for debugging MPI applications. It covers the following topics:

- "MPI Routine Argument Checking" on page 19
- "Using TotalView with MPI programs" on page 19
- "Using `idb` and `gdb` with MPI programs" on page 19

MPI Routine Argument Checking

By default, the SGI MPI implementation does not check the arguments to some performance-critical MPI routines such as most of the point-to-point and collective communication routines. You can force MPI to always check the input arguments to MPI functions by setting the `MPI_CHECK_ARGS` environment variable. However, setting this variable might result in some degradation in application performance, so it is not recommended that it be set except when debugging.

Using TotalView with MPI programs

The syntax for running SGI MPI with Etnus' TotalView is as follows:

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

Note that TotalView is not expected to operate with MPI processes started via the `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple` functions.

Using `idb` and `gdb` with MPI programs

Because the `idb` and `gdb` debuggers are designed for sequential, non-parallel applications, they are generally not well suited for use in MPI program debugging and development. However, the use of the `MPI_SLAVE_DEBUG_ATTACH` environment variable makes these debuggers more usable.

If you set the `MPI_SLAVE_DEBUG_ATTACH` environment variable to a global rank number, the MPI process sleeps briefly in startup while you use `idb` or `gdb` to attach to the process. A message is printed to the screen, telling you how to use `idb` or `gdb` to attach to the process.

Similarly, if you want to debug the MPI daemon, setting `MPI_DAEMON_DEBUG_ATTACH` sleeps the daemon briefly while you attach to it.

Profiling MPI Applications

This chapter describes the use of profiling tools to obtain performance information. Compared to the performance analysis of sequential applications, characterizing the performance of parallel applications can be challenging. Often it is most effective to first focus on improving the performance of MPI applications at the single process level.

It may also be important to understand the message traffic generated by an application. A number of tools can be used to analyze this aspect of a message passing application's performance, including Performance Co-Pilot and various third party products. In this chapter, you can learn how to use these various tools with MPI applications. It covers the following topics:

- "Using Profiling Tools with MPI Applications" on page 21
- "Profiling Interface" on page 22
- "MPI Internal Statistics" on page 23
- "Performance Co-Pilot" on page 24

Using Profiling Tools with MPI Applications

Two of the most common SGI profiling tools are `profile.pl` and `histx+`. The following sections describe how to invoke these tools. Performance Co-Pilot tools and tips for writing your own tools are also included.

You can also use the `perfcatch` utility to profile the performance of an MPI program. For more information, see Chapter 7, "MPI Performance Profiling" on page 39.

`profile.pl`

You can use `profile.pl` to obtain procedure level profiling as well as information about the hardware performance monitors. For further information, see the `profile.pl`(1) and `pfmon`(1) man pages.

General format:

% **mpirun** *mpirun_entry_object* [*mpirun_entry_object ...*] **profile.pl** [*profile.pl_options*] *executable*

Example:

```
% mpirun -np 4 profile.pl -s1 -c4,5 -N 1000 ./a.out
```

histx+

histx+ is a small set of tools that can assist with performance analysis and bottlenect identification.

General formats for histx (Histogram) and lipfpm (Linux IPF Performance Monitor):

```
% mpirun -np 4 histx [histx_options] ./a.out
```

```
% lipfpm [lipfpm_options] mmpirun -np 4 ./a.out
```

Examples:

```
% mpirun -np 4 histx -f -o histx.out ./a.out
```

```
% lipfpm -f -e LOADS_RETIRED -e STORES_RETIRED mpirun -np 4 ./a.out
```

Profiling Interface

You can write your own profiling by using the MPI-1 standard PMPI_* calls. In addition, either within your own profiling library or within the application itself you can use the MPI_Wtime function call to time specific calls or sections of your code.

The following example is actual output for a single rank of a program that was run on 128 processors, using a user-created profiling library that performs call counts and timings of common MPI calls. Notice that for this rank most of the MPI time is being spent in MPI_Waitall and MPI_Allreduce.

```
Total job time 2.203333e+02 sec
Total MPI processes 128
Wtime resolution is 8.000000e-07 sec

activity on process rank 0
comm_rank calls 1      time 8.800002e-06
get_count calls 0      time 0.000000e+00
ibsend calls    0      time 0.000000e+00
```

```
probe calls     0       time 0.000000e+00
recv calls      0       time 0.00000e+00   avg datacnt 0   waits 0      wait time 0.00000e+00
irecv calls     22039   time 9.76185e-01   datacnt 23474032 avg datacnt 1065
send calls      0       time 0.000000e+00
ssend calls     0       time 0.000000e+00
isend calls     22039   time 2.950286e+00
wait calls      0       time 0.00000e+00   avg datacnt 0
waitall calls   11045   time 7.73805e+01   # of Reqs 44078   avg data   cnt 137944
barrier calls   680     time 5.133110e+00
alltoall calls  0       time 0.0e+00       avg datacnt 0
alltoallv calls 0       time 0.000000e+00
reduce calls    0       time 0.000000e+00
allreduce calls 4658    time 2.072872e+01
bcast calls     680     time 6.915840e-02
gather calls    0       time 0.000000e+00
gatherv calls   0       time 0.000000e+00
scatter calls   0       time 0.000000e+00
scatterv calls  0       time 0.000000e+00

activity on process rank 1
...
```

MPI Internal Statistics

MPI keeps track of certain resource utilization statistics. These can be used to determine potential performance problems caused by lack of MPI message buffers and other MPI internal resources.

To turn on the displaying of MPI internal statistics, use the MPI_STATS environment variable or the -stats option on the mpirun command. MPI internal statistics are always being gathered, so displaying them does not cause significant additional overhead. In addition, one can sample the MPI statistics counters from within an application, allowing for finer grain measurements. For information about these MPI extensions, see the mpi_stats man page.

These statistics can be very useful in optimizing codes in the following ways:

- To determine if there are enough internal buffers and if processes are waiting (retries) to aquire them

- To determine if single copy optimization is being used for point-to-point or collective calls

For additional information on how to use the MPI statistics counters to help tune the run-time environment for an MPI application, see Chapter 6, "Run-time Tuning" on page 29.

Performance Co-Pilot

In addition to the tools described in the preceding sections, you can also use the MPI agent for Performance Co-Pilot to profile your application. The two additional Performance Co-Pilot tools specifically designed for MPI are `mpivis` and `mpimon`. These tools do not use trace files and can be used live or can be logged for later replay.

Following are examples of the `mpivis` and `mpimon` tools.
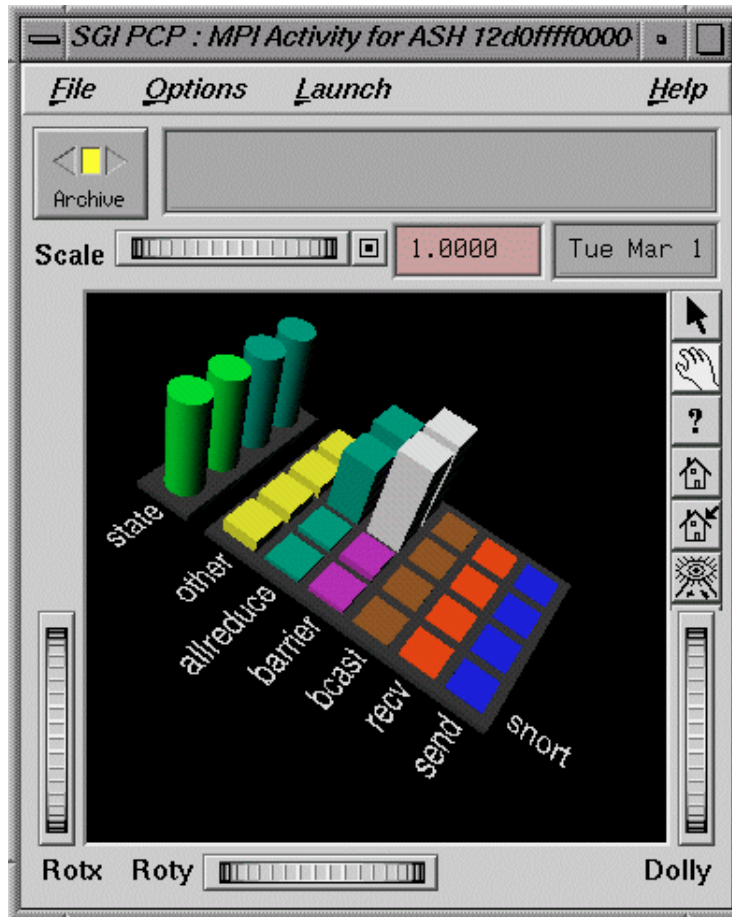
Figure 5-1 mpivis Tool

Figure 5-2 mpimon Tool

Third Party Products

Two third party tools that you can use with the SGI MPI implementation are Vampir from Pallas (`www.pallas.com`) and Jumpshot, which is part of the MPICH distribution. Both of these tools are effective for smaller, short duration MPI jobs. However, the trace files these tools generate can be enormous for longer running or highly parallel jobs. This causes a program to run more slowly, but even more problematic is that the tools to analyze the data are often overwhelmed by the amount of data.

Run-time Tuning

This chapter discusses ways in which the user can tune the run-time environment to improve the performance of an MPI message passing application on SGI computers. None of these ways involve application code changes. This chapter covers the following topics:

- "Reducing Run-time Variability" on page 29
- "Tuning MPI Buffer Resources" on page 30
- "Avoiding Message Buffering – Enabling Single Copy" on page 31
- "Memory Placement and Policies" on page 32
- "Tuning MPI/OpenMP Hybrid Codes" on page 35
- "Tuning for Running Applications Across Multiple Hosts" on page 35
- "Suspending MPI Jobs" on page 37

Reducing Run-time Variability

One of the most common problems with optimizing message passing codes on large shared memory computers is achieving reproducible timings from run to run. To reduce run-time variability, you can take the following precautions:

- Do not oversubscribe the system. In other words, do not request more CPUs than are available and do not request more memory than is available. Oversubscribing causes the system to wait unnecessarily for resources to become available and leads to variations in the results and less than optimal performance.

- Avoid interference from other system activity. The Linux kernel uses more memory on node 0 than on other nodes (node 0 is called the kernel node in the following discussion). If your application uses almost all of the available memory per processor, the memory for processes assigned to the kernel node can unintentionally spill over to nonlocal memory. By keeping user applications off the kernel node, you can avoid this effect.

  Additionally, by restricting system daemons to run on the kernel node, you can also deliver an additional percentage of each application CPU to the user.

- Avoid interference with other applications. You can use cpusets or cpumemsets to address this problem also. You can use cpusets to effectively partition a large, distributed memory host in a fashion that minimizes interactions between jobs running concurrently on the system. See the *Linux Resource Administration Guide* for information about cpusets and cpumemsets.

- On a quiet, dedicated system, you can use `dplace` or the `MPI_DSM_CPULIST` shell variable to improve run-time performance repeatability. These approaches are not as suitable for shared, nondedicated systems.

- Use a batch scheduler; for example, LSF from Platform Computing or PBSpro from Veridan. These batch schedulers use cpusets to avoid oversubscribing the system and possible interference between applications.

Tuning MPI Buffer Resources

By default, the SGI MPI implementation buffers messages whose lengths exceed 64 bytes. Longer messages are buffered in a shared memory region to allow for exchange of data between MPI processes. In the SGI MPI implementation, these buffers are divided into two basic pools.

- For messages exchanged between MPI processes within the same host or between partitioned systems when using the XPMEM driver, buffers from the "per process" pool (called the "per proc" pool) are used. Each MPI process is allocated a fixed portion of this pool when the application is launched. Each of these portions is logically partitioned into 16-KB buffers.

- For MPI jobs running across multiple hosts, a second pool of shared memory is available. Messages exchanged between MPI processes on different hosts use this pool of shared memory, called the "per host" pool. The structure of this pool is somewhat more complex than the "per proc" pool.

For an MPI job running on a single host, messages that exceed 64 bytes are handled as follows. For messages with a length of 16 KB or less, the sender MPI process buffers the entire message. It then delivers a message header (also called a control message) to a mailbox, which is polled by the MPI receiver when an MPI call is made. Upon finding a matching receive request for the sender's control message, the receiver copies the data out of the shared memory buffer into the application buffer indicated in the receive request. The receiver then sends a message header back to the sender process, indicating that the shared memory buffer is available for reuse. Messages whose length exceeds 16 KB are broken down into 16-KB chunks, allowing

the sender and receiver to overlap the copying of data to and from shared memory in a pipeline fashion.

Because there is a finite number of these shared memory buffers, this can be a constraint on the overall application performance for certain communication patterns. You can use the MPI_BUFS_PER_PROC shell variable to adjust the number of buffers available for the "per proc" pool. Similarly, you can use the MPI_BUFS_PER_HOST shell variable to adjust the "per host" pool. You can use the MPI statistics counters to determine if retries for these shared memory buffers are occurring.

For information on the use of these counters, see "MPI Internal Statistics" on page 23. In general, you can avoid excessive numbers of retries for buffers by increasing the number of buffers for the "per proc" pool or "per host" pool. However, you should keep in mind that increasing the number of buffers does consume more memory. Also, increasing the number of "per proc" buffers does potentially increase the probability for cache pollution (that is, the excessive filling of the cache with message buffers). Cache pollution can result in degraded performance during the compute phase of a message passing application.

There are additional buffering considerations to take into account when running an MPI job across multiple hosts. For further discussion of multihost runs, see "Tuning for Running Applications Across Multiple Hosts" on page 35.

For further discussion on programming implications concerning message buffering, see "Buffering" on page 11.

Avoiding Message Buffering – Enabling Single Copy

For message transfers between MPI processes within the same host or transfers between partitions, it is possible under certain conditions to avoid the need to buffer messages. Because many MPI applications are written assuming infinite buffering, the use of this unbuffered approach is not enabled by default for MPI_Send. This section describes how to activate this mechanism by default for MPI_Send.

For MPI_Isend, MPI_Sendrecv, MPI_Alltoall, MPI_Bcast, MPI_Allreduce, and MPI_Reduce, this optimization is enabled by default for large message sizes. To disable this default single copy feature used for the collectives, use the MPI_DEFAULT_SINGLE_COPY_OFF environment variable.

Using the XPMEM Driver for Single Copy Optimization

MPI takes advantage of the XPMEM driver to support single copy message transfers between two processes within the same host or across partitions.

Enabling single copy transfers may result in better performance, since this technique improves MPI's bandwidth. However, single copy transfers may introduce additional synchronization points, which can reduce application performance in some cases.

The threshold for message lengths beyond which MPI attempts to use this single copy method is specified by the `MPI_BUFFER_MAX` shell variable. Its value should be set to the message length in bytes beyond which the single copy method should be tried. In general, a value of 2000 or higher is beneficial for many applications.

During job startup, MPI uses the XPMEM driver (via the xpmem kernel module) to map memory from one MPI process to another. The mapped areas include the static (BSS) region, the private heap, the stack region, and optionally the symmetric heap region of each process.

Memory mapping allows each process to directly access memory from the address space of another process. This technique allows MPI to support single copy transfers for contiguous data types from any of these mapped regions. For these transfers, whether between processes residing on the same host or across partitions, the data is copied using a `bcopy` process. A `bcopy` process is also used to transfer data between two different executable files on the same host or two different executable files across partitions. For data residing outside of a mapped region (a `/dev/zero` region, for example), MPI uses a buffering technique to transfer the data.

Memory mapping is enabled by default. To disable it, set the `MPI_MEMMAP_OFF` environment variable. Memory mapping must be enabled to allow single-copy transfers, MPI-2 one-sided communication, support for the SHMEM model, and certain collective optimizations.

Memory Placement and Policies

The MPI library takes advantage of NUMA placement functions that are available. Usually, the default placement is adequate. Under certain circumstances, however, you might want to modify this default behavior. The easiest way to do this is by setting one or more MPI placement shell variables. Several of the most commonly used of these variables are discribed in the following sections. For a complete listing of memory placement related shell variables, see the `MPI(1)` man page.

MPI_DSM_CPULIST

The `MPI_DSM_CPULIST` shell variable allows you to manually select processors to use for an MPI application. At times, specifying a list of processors on which to run a job can be the best means to insure highly reproducible timings, particularly when running on a dedicated system.

This setting is treated as a comma and/or hyphen delineated ordered list that specifies a mapping of MPI processes to CPUs. If running across multiple hosts, the per host components of the CPU list are delineated by colons. Within hyphen delineated lists CPU striding may be specified by placing "/#" after the list where "#" is the stride distance.

Note: This feature should not be used with MPI applications that use either of the MPI-2 spawn related functions.

Examples of settings are as follows:

Value	CPU Assignment
8,16,32	Place three MPI processes on CPUs 8, 16, and 32.
32,16,8	Place the MPI process rank zero on CPU 32, one on 16, and two on CPU 8.
8-15/2	Place the MPI processes 0 through 3 strided on CPUs 8, 10, 12, and 14
8-15,32-39	Place the MPI processes 0 through 7 on CPUs 8 to 15. Place the MPI processes 8 through 15 on CPUs 32 to 39.
39-32,8-15	Place the MPI processes 0 through 7 on CPUs 39 to 32. Place the MPI processes 8 through 15 on CPUs 8 to 15.
8-15:16-23	Place the MPI processes 0 through 7 on the first host on CPUs 8 through 15. Place MPI processes 8 through 15 on CPUs 16 to 23 on the second host.

Note that the process rank is the `MPI_COMM_WORLD` rank. The interpretation of the CPU values specified in the `MPI_DSM_CPULIST` depends on whether the MPI job is being run within a cpuset. If the job is run outside of a cpuset, the CPUs specify *cpunum* values beginning with 0 and up to the number of CPUs in the system minus one. When running within a cpuset, the default behavior is to interpret the CPU values as relative processor numbers within the cpuset.

The number of processors specified should equal the number of MPI processes that will be used to run the application. The number of colon delineated parts of the list must equal the number of hosts used for the MPI job. If an error occurs in processing the CPU list, the default placement policy is used.

MPI_DSM_DISTRIBUTE

Use the `MPI_DSM_DISTRIBUTE` shell variable to ensure that each MPI process will get a physical CPU and memory on the node to which it was assigned.If this environment variable is used without specifying an `MPI_DSM_CPULIST` variable, it will cause MPI to assign MPI ranks starting at logical CPU 0 and incrementing until all ranks have been placed. Therefore, it is recommended that this variable be used only if running within a cpumemset or on a dedicated system.

MPI_DSM_PPM

The `MPI_DSM_PPM` shell variable allows you to specify the number of MPI processes to be placed on a node. Memory bandwidth intensive applications can benefit from placing fewer MPI processes on each node of a distributed memory host. On SGI Altix 3000 systems, setting `MPI_DSM_PPM` to 1 places one MPI process on each node.

MPI_DSM_VERBOSE

Setting the `MPI_DSM_VERBOSE` shell variable directs MPI to display a synopsis of the NUMA placement options being used at run time.

Using `dplace` for Memory Placement

The `dplace` tool offers another means of specifying the placement of MPI processes within a distributed memory host. The `dplace` tool and MPI interoperate to allow MPI to better manage placement of certain shared memory data structures when `dplace` is used to place the MPI job.

For instructions on how to use `dplace` with MPI, see the `dplace`(1) man page.

Tuning MPI/OpenMP Hybrid Codes

Hybrid MPI/OpenMP applications might require special memory placement features. This section describes a preliminary method for achieving this memory placement.

The basic idea is to space out the MPI processes to accommodate the OpenMP threads associated with each MPI process. In addition, assuming a particular ordering of library init code (see the DSO man page), this method employs procedures to insure that the OpenMP threads remain close to the parent MPI process. This type of placement has been found to improve the performance of some hybrid applications significantly.

To take partial advantage of this placement option, the following requirements must be met:

* When running the application, you must set the MPI_OPENMP_INTEROP shell variable.

* To compile the application, you must use a compiler that supports the -mp compiler option. This hybrid model placement option is not available with other compilers.

MPI reserves nodes for this hybrid placement model based on the number of MPI processes and the number of OpenMP threads per process, rounded up to the nearest multiple of 2. For example, if 6 OpenMP threads per MPI process are going to be used for a 4 MPI process job, MPI will request a placement for 24 (4 X 6) CPUs on the host machine. You should take this into account when requesting resources in a batch environment or when using cpusets. In this implementation, it is assumed that all MPI processes start with the same number of OpenMP threads, as specified by the OMP_NUM_THREADS or equivalent shell variable at job startup.

The OpenMP threads are not actually pinned to a CPU but are free to migrate to any of the CPUs in the OpenMP thread group for each MPI rank. The pinning of the OpenMP thread to a specific CPU will be supported in a future release.

Tuning for Running Applications Across Multiple Hosts

When you are running an MPI application across a cluster of hosts, there are additional run-time environment settings and configurations that you can consider when trying to improve application performance.

Systems can use the XPMEM interconnect to cluster hosts as partitioned systems, or use the Voltaire InfiniBand (IB) interconnect or TCP/IP as the multihost interconnect.

When launched as a distributed application, MPI probes for these interconnects at job startup. For details of launching a distributed application, see "Launching a Distributed Application" on page 7. When a high performance interconnect is detected, MPI attempts to use this interconnect if it is available on every host being used by the MPI job. If the interconnect is not available for use on every host, the library attempts to use the next slower interconnect until this connectivity requirement is met. Table 6-1 on page 36 specifies the order in which MPI probes for available interconnects.

Table 6-1 Inquiry Order for Available Interconnects

Interconnect	Default Order of Selection	Environment Variable to Require Use
XPMEM	1	MPI_USE_XPMEM
InfiniBand	2	MPI_USE_IB
TCP/IP	3	MPI_USE_TCP

The third column of Table 6-1 on page 36 also indicates the environment variable you can set to pick a particular interconnect other than the default.

In general, to insure the best performance of the application, you should allow MPI to pick the fastest available interconnect.

In addition to the choice of interconnect, you should know that multihost jobs may use different buffers from those used by jobs run on a single host. In the SGI implementation of MPI, the XPMEM interconnect uses the "per proc" buffers while the InfiniBand and TCP interconnects use the "per host" buffers. The default setting for the number of buffers per proc or per host might be too low for many applications. You can determine whether this setting is too low by using the MPI statistics described earlier in this section.

When using the TCP/IP interconnect, unless specified otherwise, MPI uses the default IP adapter for each host. To use a nondefault adapter, enter the adapter-specific host name on the mpirun command line.

When using the InfiniBand interconnect, MPT applications may not execute a `fork()` or `system()` call. The InfiniBand driver produces undefined results when an MPT process using InfiniBand forks.

MPI_USE_IB

Requires the MPI library to use the IB (InfiniBand) driver as the interconnect when running across multiple hosts or running with multiple binaries. MPT requires the `ibhost` software stack from Voltaire when the IB interconnect is used. If IB is used, the `MPI_COREDUMP` environment variable is forced to INHIBIT, to comply with the IB driver restriction that no fork()s may occur after IB resources have been allocated. Default: Not set

MPI_IB_RAILS

If the MPI library uses the IB driver as the inter-host interconnect and this is set to equal 1, then the MPI library will use special low latency and higher bandwidth IB code paths. If this is set to 2, the library will try to make use of multiple available separate IB fabrics and split its traffic across them. The `rail-config` utility is required to have been run by the system administrator to enable the library to correctly use the separate fabrics. Default: 0 on Altix systems with Itanium processors. 1 on Altix systems with Xeon processors.

MPI_IB_SINGLE_COPY_BUFFER_MAX

When MPI transfers data over IB, if the size of the cumulative data is greater than this value then MPI will attempt to send the data directly between the processes's buffers and not through intermediate buffers inside the MPI library. Default: 32767

For more information on these environment variables, see the "ENVIRONMENT VARIABLES" section of the `mpi`(1) man page.

Suspending MPI Jobs

MPI software from SGI can internally use the XPMEM kernel module to provide direct access to data on remote partitions and to provide single copy operations to local data. Any pages used by these operations are prevented from paging by the

XPMEM kernel module. As of the SGI ProPack 3 Service Pack 5 and SGI Propack 4 for Linux releases, if an administrator needs to temporarily suspend a MPI application to allow other applications to run, they can unpin these pages so they can be swapped out and made available for other applications.

Each process of a MPI application which is using the XPMEM kernel module will have a /proc/xpmem/*pid* file associated with it. The number of pages owned by this process which are prevented from paging by XPMEM can be displayed by concatenating the /proc/xpmem/*pid* file, for example:

```
# cat  /proc/xpmem/5562
pages pinned by XPMEM: 17
```

To unpin the pages for use by other processes, the administrator must first suspend all the processes in the application. The pages can then be unpinned by echoing any value into the /proc/xpmem/*pid* file, for example:

```
# echo 1 > /proc/xpmem/5562
```

The echo command will not return until that process's pages are unpinned.

When the MPI application is resumed, the XPMEM kernel module will prevent these pages from paging as they are referenced by the application.

MPI Performance Profiling

This chapter describes the `perfcatch` utility used to profile the performance of an MPI program. It covers the following topics:

- "Overview of `perfcatch` Utility" on page 39
- "Using the `perfcatch` Utility" on page 39
- " `MPI_PROFILING_STATS` Results File Example" on page 40
- "MPI Performance Profiling Environment Variables" on page 43
- "MPI Supported Profiled Functions"

For information on additional profiling tools, see "Using Profiling Tools with MPI Applications" on page 21.

Overview of `perfcatch` Utility

The `perfcatch` utility runs an MPI program with a wrapper profiling library that prints MPI call profiling information to a summary file upon MPI program completion. This MPI profiling result file is called `MPI_PROFILING_STATS`, by default (see " `MPI_PROFILING_STATS` Results File Example" on page 40). It is created in the current working directory of the MPI process with rank 0.

Using the `perfcatch` Utility

The syntax of the `perfcatch` utility is, as follows:

`perfcatch [-v | -vofed | -i] cmd args`

The `perfcatch` utility accepts the following options:

No option	Supports MPT
`-v`	Supports Voltaire MPI
`-vofed`	Supports Voltaire OFED MPI

-i   Supports Intel MPI

To use `perfcatch` with an SGI Message Passing Toolkit MPI program, insert the `perfcatch` command in front of the executable name. Here are some examples:

```
mpirun -np 64 perfcatch a.out arg1
```
and

```
mpirun host1 32, host2 64 perfcatch a.out arg1
```

To use `perfcatch` with Intel MPI, add the `-i` options. An example is, as follows:

```
mpiexec -np 64 perfcatch -i a.out arg1
```

For more information, see the `perfcatch`(1) man page.

`MPI_PROFILING_STATS` Results File Example

The MPI profiling result file has a summary statistics section followed by a rank-by-rank profiling information section. The summary statistics section reports some overall statistics, including the percent time each rank spent in MPI functions, and the MPI process that spent the least and the most time in MPI functions. Similar reports are made about system time usage.

The rank-by-rank profiling information section lists every profiled MPI function called by a particular MPI process. The number of calls and the total time consumed by these calls is reported. Some functions report additional information such as average data counts and communication peer lists.

An example `MPI_PROFILING_STATS` results file is, as follows:

```
============================================================
PERFCATCHER version 22
(C) Copyright SGI.  This library may only be used
on SGI hardware platforms. See LICENSE file for
details.
============================================================
MPI program profiling information
Job profile recorded Wed Jan 17 13:05:24 2007
Program command line:                              /home/estes01/michel/sastest/mpi_hello_linux
Total MPI processes                    2

Total MPI job time, avg per rank          0.0054768 sec
Profiled job time, avg per rank           0.0054768 sec
Percent job time profiled, avg per rank   100%

Total user time, avg per rank             0.001 sec
Percent user time, avg per rank           18.2588%
Total system time, avg per rank           0.0045 sec
Percent system time, avg per rank         82.1648%

Time in all profiled MPI routines, avg per rank    5.75004e-07 sec
Percent time in profiled MPI routines, avg per rank 0.0104989%

Rank-by-Rank Summary Statistics
-------------------------------

Rank-by-Rank: Percent in Profiled MPI routines
       Rank:Percent
       0:0.0112245%   1:0.00968502%
  Least:  Rank 1     0.00968502%
  Most:   Rank 0     0.0112245%
  Load Imbalance:  0.000771%

Rank-by-Rank: User Time
       Rank:Percent
       0:17.2683%     1:19.3699%
  Least:  Rank 0     17.2683%
  Most:   Rank 1     19.3699%

Rank-by-Rank: System Time
       Rank:Percent
```

```
      0:86.3416%      1:77.4796%
  Least:  Rank 1      77.4796%
  Most:   Rank 0      86.3416%


Notes
-----


Wtime resolution is                  5e-08 sec


Rank-by-Rank MPI Profiling Results
----------------------------------


Activity on process rank 0

        Single-copy checking was not enabled.
comm_rank          calls:     1  time: 6.50005e-07 s  6.50005e-07 s/call


Activity on process rank 1

        Single-copy checking was not enabled.
comm_rank          calls:     1  time: 5.00004e-07 s  5.00004e-07 s/call


-----------------------------------------------

recv profile

          cnt/sec for all remote ranks
local    ANY_SOURCE        0            1
 rank

-----------------------------------------------

recv wait for data profile

          cnt/sec for all remote ranks
local         0            1
 rank
-----------------------------------------------

recv wait for data profile
```

```
          cnt/sec for all remote ranks
local        0             1
 rank

------------------------------------------------

send profile

          cnt/sec for all destination ranks
  src        0             1
 rank

------------------------------------------------

ssend profile

          cnt/sec for all destination ranks
  src        0             1
 rank

------------------------------------------------

ibsend profile

          cnt/sec for all destination ranks
  src        0             1
 rank
```

MPI Performance Profiling Environment Variables

The MPI performance profiling environment variables are, as follows:

Variable	Description
MPI_PROFILE_AT_INIT	Activates MPI profiling immediately, that is, at the start of MPI program execution.
MPI_PROFILING_STATS_FILE	Specifies the file where MPI profiling results are written. If not

specified, the file
`MPI_PROFILING_STATS` is written.

MPI Supported Profiled Functions

The MPI supported profiled functions are, as follows:

Note: Some functions may not be implemented in all language as indicated below.

Languages	Function
C Fortran	`mpi_allgather`
C Fortran	`mpi_allgatherv`
C Fortran	`mpi_allreduce`
C Fortran	`mpi_alltoall`
C Fortran	`mpi_alltoallv`
C Fortran	`mpi_barrier`
C Fortran	`mpi_bcast`
C Fortran	`mpi_comm_create`
C Fortran	`mpi_comm_free`
C Fortran	`mpi_comm_group`
C Fortran	`mpi_comm_rank`
C Fortran	`mpi_finalize`
C Fortran	`mpi_gather`
C Fortran	`mpi_gatherv`
C	`mpi_get_count`
C Fortran	`mpi_group_difference`
C Fortran	`mpi_group_excl`
C Fortran	`mpi_group_free`
C Fortran	`mpi_group_incl`
C Fortran	`mpi_group_intersection`
C Fortran	`mpi_group_range_excl`

C Fortran	`mpi_group_range_incl`
C Fortran	`mpi_group_union`
C	`mpi_ibsend`
C Fortran	`mpi_init`
C	`mpi_init_thread`
C Fortran	`mpi_irecv`
C Fortran	`mpi_isend`
C	`mpi_probe`
C Fortran	`mpi_recv`
C Fortran	`mpi_reduce`
C Fortran	`mpi_scatter`
C Fortran	`mpi_scatterv`
C Fortran	`mpi_send`
C Fortran	`mpi_sendrecv`
C Fortran	`mpi_ssend`
C Fortran	`mpi_test`
C Fortran	`mpi_testany`
C Fortran	`mpi_wait`
C Fortran	`mpi_wait`

Troubleshooting and Frequently Asked Questions

This chapter provides answers to some common problems users encounter when starting to use SGI MPI, as well as answers to other frequently asked questions. It covers the following topics:

- "What are some things I can try to figure out why `mpirun` is failing? " on page 47

- "My code runs correctly until it reaches `MPI_Finalize()` and then it hangs." on page 49

- "I keep getting error messages about `MPI_REQUEST_MAX` being too small." on page 49

- "I am not seeing `stdout` and/or `stderr` output from my MPI application." on page 50

- "How can I get the MPT software to install on my machine?" on page 50

- "Where can I find more information about the SHMEM programming model? " on page 50

- "The `ps(1)` command says my memory use (`SIZE`) is higher than expected. " on page 50

- "What does `MPI: could not run executable` mean?" on page 51

- "How do I combine MPI with *insert favorite tool here*?" on page 51

- "Must I use `MPIO_Wait()` and `MPIO_Test()`?" on page 52

- "Must I modify my code to replace calls to `MPIO_Wait()` with `MPI_Wait()` and recompile?" on page 52

- "Why do I see "stack traceback" information when my MPI job aborts?" on page 53

What are some things I can try to figure out why `mpirun` is failing?

Here are some things to investigate:

- Look in `/var/log/messages` for any suspicious errors or warnings. For example, if your application tries to pull in a library that it cannot find, a message should appear here. Only the root user can view this file.

- Be sure that you did not misspell the name of your application.

- To find `rld`/dynamic link errors, try to run your program without `mpirun`. You will get the "`mpirun must be used to launch all MPI applications`" message, along with any `rld` link errors that might not be displayed when the program is started with `mpirun`.

  As a last resort, setting the environment variable `LD_DEBUG` to `all` will display a set of messages for each symbol that `rld` resolves. This produces a lot of output, but should help you find the cause of the link arror.

- Be sure that you are setting your remote directory properly. By default, `mpirun` attempts to place your processes on all machines into the directory that has the same name as `$PWD`. This should be the common case, but sometimes different functionality is required. For more information, see the section on `$MPI_DIR` and/or the `-dir` option in the `mpirun` man page.

- If you are using a relative pathname for your application, be sure that it appears in `$PATH`. In particular, `mpirun` will not look in '.' for your application unless '.' appears in `$PATH`.

- Run `/usr/etc/ascheck` to verify that your array is configured correctly.

- Be sure that you can execute `rsh` (or `arshell`) to all of the hosts that you are trying to use without entering a password. This means that either `/etc/hosts.equiv` or `~/.rhosts` must be modified to include the names of every host in the MPI job. Note that using the `-np` syntax (i.e. no hostnames) is equivalent to typing `localhost`, so a *localhost* entry will also be needed in one of the above two files.

- Use the `-verbose` option to verify that you are running the version of MPI that you think you are running.

- Be very careful when setting MPI environment variables from within your `.cshrc` or `.login` files, because these will override any settings that you might later set from within your shell (due to the fact that MPI creates the equivalent of a fresh login session for every job). The safe way to set things up is to test for the existence of `$MPI_ENVIRONMENT` in your scripts and set the other MPI environment variables only if it is undefined.

- If you are running under a Kerberos environment, you may experience unpredictable results because currently, mpirun is unable to pass tokens. For example, in some cases, if you use telnet to connect to a host and then try to run mpirun on that host, it fails. But if you instead use rsh to connect to the host, mpirun succeeds. (This might be because telnet is kerberized but rsh is not.) At any rate, if you are running under such conditions, you will definitely want to talk to the local administrators about the proper way to launch MPI jobs.

- Look in /tmp/.arraysvcs on all machines you are using. In some cases, you might find an errlog file that may be helpful.

My code runs correctly until it reaches `MPI_Finalize()` and then it hangs.

This is almost always caused by send or recv requests that are either unmatched or not completed. An unmatched request is any blocking send for which a corresponding recv is never posted. An incomplete request is any nonblocking send or recv request that was never freed by a call to MPI_Test(), MPI_Wait(), or MPI_Request_free().

Common examples are applications that call MPI_Isend() and then use internal means to determine when it is safe to reuse the send buffer. These applications never call MPI_Wait(). You can fix such codes easily by inserting a call to MPI_Request_free() immediately after all such isend operations, or by adding a call to MPI_Wait() at a later place in the code, prior to the point at which the send buffer must be reused.

I keep getting error messages about `MPI_REQUEST_MAX` being too small.

There are two types of cases in which the MPI library reports an error concerning MPI_REQUEST_MAX. The error reported by the MPI library distinguishes these.

```
MPI has run out of unexpected request entries;
the current allocation level is: XXXXXX
```

The program is sending so many unexpected large messages (greater than 64 bytes) to a process that internal limits in the MPI library have been exceeded. The options here

are to increase the number of allowable requests via the `MPI_REQUEST_MAX` shell variable, or to modify the application.

```
MPI has run out of request entries;
the current allocation level is: MPI_REQUEST_MAX = XXXXX
```

You might have an application problem. You almost certainly are calling `MPI_Isend()` or `MPI_Irecv()` and not completing or freeing your request objects. You need to use `MPI_Request_free()`, as described in the previous section.

I am not seeing `stdout` and/or `stderr` output from my MPI application.

All `stdout` and `stderr` is line-buffered, which means that `mpirun` does not print any partial lines of output. This sometimes causes problems for codes that prompt the user for input parameters but do not end their prompts with a newline character. The only solution for this is to append a newline character to each prompt.

You can set the `MPI_UNBUFFERED_STDIO` environment variable to disable line-buffering. For more information, see the `MPI`(1) and `mpirun`(1) man pages.

How can I get the MPT software to install on my machine?

MPT RPMs are included in ProPack releases. In addition, you can obtain MPT RPMs from the SGI Support website at

```
http://support.sgi.com
```
under "Downloads".

Where can I find more information about the SHMEM programming model?

See the `intro_shmem`(3) man page.

The `ps`(1) command says my memory use (`SIZE`) is higher than expected.

At MPI job start-up, MPI calls the SHMEM library to cross-map all user static memory on all MPI processes to provide optimization opportunities. The result is large virtual memory usage. The `ps`(1) command's `SIZE` statistic is telling you the amount of

virtual address space being used, not the amount of memory being consumed. Even if all of the pages that you could reference were faulted in, most of the virtual address regions point to multiply-mapped (shared) data regions, and even in that case, actual per-process memory usage would be far lower than that indicated by SIZE.

What does `MPI: could not run executable` mean?

This message means that something happened while mpirun was trying to launch your application, which caused it to fail before all of the MPI processes were able to handshake with it.

With Array Services 3.2 or later and MPT 1.3 or later, many scenarios that generated this error message are now improved to be more descriptive.

Prior to Array Services 3.2, no diagnostic information was directly available. This was due to the highly decoupled interface between mpirun and arrayd.

mpirun directs arrayd to launch a master process on each host and listens on a socket for those masters to connect back to it. Since the masters are children of arrayd, arrayd traps SIGCHLD and passes that signal back to mpirun whenever one of the masters terminates. If mpirun receives a signal before it has established connections with every host in the job, it knows that something has gone wrong.

How do I combine MPI with *insert favorite tool here*?

In general, the rule to follow is to run mpirun on your tool and then the tool on your application. Do not try to run the tool on mpirun. Also, because of the way that mpirun sets up stdio, seeing the output from your tool might require a bit of effort. The most ideal case is when the tool directly supports an option to redirect its output to a file. In general, this is the recommended way to mix tools with mpirun. Of course, not all tools (for example, dplace) support such an option. However, it is usually possible to make it work by wrapping a shell script around the tool and having the script do the redirection, as in the following example:

```
> cat myscript
    #!/bin/sh
    setenv MPI_DSM_OFF
    dplace -verbose a.out 2> outfile
    > mpirun -np 4 myscript
    hello world from process 0
```

```
                      hello world from process 1
                      hello world from process 2
                      hello world from process 3
                      > cat outfile
                      there are now 1 threads
                      Setting up policies and initial thread.
                      Migration is off.
                      Data placement policy is PlacementDefault.
                      Creating data PM.
                      Data pagesize is 16k.
                      Setting data PM.
                      Creating stack PM.
                      Stack pagesize is 16k.
                      Stack placement policy is PlacementDefault.
                      Setting stack PM.
                      there are now 2 threads
                      there are now 3 threads
                      there are now 4 threads
                      there are now 5 threads
```

Must I use `MPIO_Wait()` and `MPIO_Test()`?

Beginning with MPT 1.8, MPT has unified the I/O requests generated from nonblocking I/O routines (such as `MPI_File_iwrite()`) and MPI requests from nonblocking message-passing routines (for example, `MPI_Isend()`). Formerly, these were different types of request objects and needed to be kept separate (one was called `MPIO_Request` and the other, `MPI_Request`). Under MPT 1.8 and later, however, this distinction is no longer necessary. You can freely mix request objects returned from I/O and MPI routines in calls to `MPI_Wait()`, `MPI_Test()`, and their variants.

Must I modify my code to replace calls to `MPIO_Wait()` with `MPI_Wait()` and recompile?

No. If you have an application that you compiled prior to MPT 1.8, you can continue to execute that application under MPT 1.8 and beyond without recompiling. Internally, MPT uses the unified requests, and for example, translates calls to `MPIO_Wait()` into calls to `MPI_Wait()`.

Why do I see "stack traceback" information when my MPI job aborts?

This is a new feature beginning with MPT 1.8. More information can be found in the MPI(1) man page in descriptions of the MPI_COREDUMP and MPI_COREDUMP_DEBUGGER environment variables.

Index