LAM/MPI User's Guide
Version 7.1.5b2

The LAM/MPI Team
Open Systems Lab
http://www.lam-mpi.org/

June 9, 2008

Contents

1 Don't Panic! (Who Should Read This Document?) 11

2 Introduction to LAM/MPI 13
2.1 About MPI . . . 13
2.2 About LAM/MPI . . . 13

3 Release Notes 15
3.1 New Feature Overview . . . 15
3.2 Known Issues . . . 17
3.2.1 `mpirun` and MPI Application `cr` Module Disagreement . . . 17
3.2.2 Checkpoint Support Disabled for Spawned Processes . . . 17
3.2.3 BLCR Support Only Works When Compiled Statically . . . 17
3.2.4 Infiniband `rpi` Module . . . 17
3.3 Usage Notes . . . 18
3.3.1 Operating System Bypass Communication: Myrinet and Infiniband . . . 18
3.4 Platform-Specific Notes . . . 19
3.4.1 Provided RPMs . . . 19
3.4.2 Filesystem Issues . . . 20
3.4.3 Dynamic/Embedded Environments . . . 21
3.4.4 Linux . . . 21
3.4.5 Mac OS X (Absoft Fortran Compilers) . . . 21
3.4.6 Microsoft Windows(TM)(Cygwin) . . . 21
3.4.7 Solaris . . . 22

4 Getting Started with LAM/MPI 23
4.1 One-Time Setup . . . 23
4.1.1 Setting the Path . . . 23
4.1.2 Finding the LAM Manual Pages . . . 25
4.2 System Services Interface (SSI) . . . 25
4.3 What Does Your LAM/MPI Installation Support? . . . 26
4.4 Booting the LAM Run-Time Environment . . . 26
4.4.1 The Boot Schema File (a.k.a, "Hostfile", "Machinefile") . . . 26
4.4.2 The `lamboot` Command . . . 27
4.4.3 The `lamnodes` Command . . . 28
4.5 Compiling MPI Programs . . . 28

4.5.1 Sample MPI Program in C . . . 29
4.5.2 Sample MPI Program in C++ . . . 30
4.5.3 Sample MPI Program in Fortran . . . 31
4.6 Running MPI Programs . . . 31
4.6.1 The `mpirun` Command . . . 31
4.6.2 The `mpiexec` Command . . . 32
4.6.3 The `mpitask` Command . . . 33
4.6.4 The `lamclean` Command . . . 34
4.7 Shutting Down the LAM Universe . . . 34

5 Supported MPI Functionality 35
5.1 MPI-1 Support . . . 35
5.1.1 Language Bindings . . . 35
5.1.2 MPI_CANCEL . . . 35
5.2 MPI-2 Support . . . 36
5.2.1 Miscellany . . . 36
5.2.2 Process Creation and Management . . . 38
5.2.3 One-Sided Communication . . . 38
5.2.4 Extended Collective Operations . . . 39
5.2.5 External Interfaces . . . 39
5.2.6 I/O . . . 40
5.2.7 Language Bindings . . . 41

6 System Services Interface (SSI) Overview 43
6.1 Types and Modules . . . 43
6.2 Terminology . . . 43
6.3 SSI Parameters . . . 44
6.3.1 Naming Conventions . . . 44
6.3.2 Setting Parameter Values . . . 45
6.4 Dynamic Shared Object (DSO) Modules . . . 46
6.5 Selecting Modules . . . 47
6.5.1 Specifying Modules . . . 47
6.5.2 Setting Priorities . . . 47
6.5.3 Selection Algorithm . . . 47

7 LAM/MPI Command Quick Reference 49
7.1 The `lamboot` Command . . . 49
7.1.1 Multiple Sessions on the Same Node . . . 50
7.1.2 Avoiding Running on Specific Nodes . . . 51
7.2 The `lamcheckpoint` Command . . . 51
7.3 The `lamclean` Command . . . 52
7.4 The `lamexec` Command . . . 52
7.5 The `lamgrow` Command . . . 52
7.6 The `lamhalt` Command . . . 53
7.7 The `laminfo` Command . . . 53
7.8 The `lamnodes` Command . . . 55

7.9 The `lamrestart` Command . . . 55
7.10 The `lamshrink` Command . . . 56
7.11 The `mpicc`, `mpiCC` / `mpic++`, and `mpif77` Commands . . . 56
7.11.1 Deprecated Names . . . 58
7.12 The `mpiexec` Command . . . 58
7.12.1 General Syntax . . . 58
7.12.2 Launching MPMD Processes . . . 59
7.12.3 Launching MPI Processes with No Established LAM Universe . . . 60
7.13 The `mpimsg` Command (Deprecated) . . . 60
7.14 The `mpirun` Command . . . 60
7.14.1 Simple Examples . . . 60
7.14.2 Controlling Where Processes Are Launched . . . 61
7.14.3 Per-Process Controls . . . 62
7.14.4 Ability to Pass Environment Variables . . . 62
7.14.5 Current Working Directory Behavior . . . 63
7.15 The `mpitask` Command . . . 63
7.16 The `recon` Command . . . 63
7.17 The `tping` Command . . . 64
7.18 The `lamwipe` Command . . . 64

8 Available LAM Modules **65**
8.1 Booting the LAM Run-Time Environment . . . 65
8.1.1 Boot Schema Files (a.k.a., “Hostfiles” or “Machinefiles”) . . . 65
8.1.2 Minimum Requirements . . . 67
8.1.3 Selecting a boot Module . . . 67
8.1.4 boot SSI Parameters . . . 67
8.1.5 The bproc Module . . . 67
8.1.6 The globus Module . . . 69
8.1.7 The rsh Module (including `ssh`) . . . 70
8.1.8 The slurm Module . . . 71
8.1.9 The tm Module (OpenPBS / PBS Pro / Torque) . . . 73

9 Available MPI Modules **75**
9.1 General MPI SSI Parameters . . . 75
9.2 MPI Module Selection Process . . . 75
9.3 MPI Point-to-point Communication (Request Progression Interface / RPI) . . . 76
9.3.1 Two Different Shared Memory RPI Modules . . . 77
9.3.2 The crtcp Module (Checkpoint-able TCP Communication) . . . 77
9.3.3 The gm Module (Myrinet) . . . 78
9.3.4 The ib Module (Infiniband) . . . 81
9.3.5 The lamd Module (Daemon-Based Communication) . . . 85
9.3.6 The sysv Module (Shared Memory Using System V Semaphores) . . . 86
9.3.7 The tcp Module (TCP Communication) . . . 87
9.3.8 The usysv Module (Shared Memory Using Spin Locks) . . . 88
9.4 MPI Collective Communication . . . 89
9.4.1 Selecting a coll Module . . . 89

9.4.2 coll SSI Parameters . . . 90
9.4.3 The lam_basic Module . . . 91
9.4.4 The smp Module . . . 92
9.4.5 The shmem Module . . . 92
9.5 Checkpoint/Restart of MPI Jobs . . . 96
9.5.1 Selecting a cr Module . . . 96
9.5.2 cr SSI Parameters . . . 96
9.5.3 The blcr Module . . . 97
9.5.4 The self Module . . . 99

10 Debugging Parallel Programs **103**
10.1 Naming MPI Objects . . . 103
10.2 TotalView Parallel Debugger . . . 103
10.2.1 Attaching TotalView to MPI Processes . . . 104
10.2.2 Suggested Use . . . 105
10.2.3 Limitations . . . 106
10.2.4 Message Queue Debugging . . . 107
10.3 Serial Debuggers . . . 107
10.3.1 Lauching Debuggers . . . 107
10.3.2 Attaching Debuggers . . . 108
10.4 Memory-Checking Debuggers . . . 108

11 Troubleshooting **111**
11.1 The LAM/MPI Mailing Lists . . . 111
11.1.1 Announcements . . . 111
11.1.2 General Discussion / User Questions . . . 111
11.2 LAM Run-Time Environment Problems . . . 112
11.2.1 Problems with the `lamboot` Command . . . 112
11.3 MPI Problems . . . 113

12 Miscellaneous **115**
12.1 Singleton MPI Processes . . . 115
12.2 MPI-2 I/O Support . . . 115
12.3 Fortran Process Names . . . 115
12.4 MPI Thread Support . . . 116
12.4.1 Thread Level . . . 116
12.5 MPI-2 Name Publishing . . . 117
12.6 Interoperable MPI (IMPI) Support . . . 117
12.6.1 Purpose of IMPI . . . 117
12.6.2 Current IMPI functionality . . . 117
12.6.3 Running an IMPI Job . . . 118
12.6.4 Complex Network Setups . . . 118
12.7 Batch Queuing System Support . . . 119
12.8 Location of LAM's Session Directory . . . 119
12.9 Signal Catching . . . 120
12.10 MPI Attributes . . . 120

Discussion Item 120

List of Tables

3.1 SSI modules that are included in the official LAM/MPI RPMs. . . . 19

4.1 List of common shells and the corresponding environment setup files for interactive shells. . 24
4.2 List of common shells and the corresponding environment setup files for non-interactive shells. 24

5.1 Supported optional fortran datatypes. . . . 35
5.2 Supported MPI-2 info functions. . . . 37
5.3 Supported MPI-2 handle conversion functions. . . . 37
5.4 Supported MPI-2 error handler functions. . . . 37
5.5 Supported MPI-2 new datatype manipulation functions. . . . 38
5.6 Supported MPI-2 dynamic functions. . . . 38
5.7 Supported MPI-2 one-sided functions. . . . 39
5.8 Supported MPI-2 intercommunicator collective functions. . . . 39
5.9 Major topics in the MPI-2 chapter “External Interfaces”, and LAM’s level of support. . . . 39
5.10 Supported MPI-2 external interface functions, grouped by function. . . . 40

6.1 SSI module types and their corresponding scopes. . . . 44

8.1 SSI parameters for the bproc boot module. . . . 69
8.2 SSI parameters for the globus boot module. . . . 70
8.3 SSI parameters for the rsh boot module. . . . 72
8.4 SSI parameters for the slurm boot module. . . . 73
8.5 SSI parameters for the tm boot module. . . . 74

9.1 SSI parameters for the crtcp RPI module. . . . 78
9.2 SSI parameters for the gm RPI module. . . . 79
9.3 SSI parameters for the ib RPI module. . . . 82
9.4 SSI parameters for the lamd RPI module. . . . 85
9.5 SSI parameters for the sysv RPI module. . . . 87
9.6 SSI parameters for the tcp RPI module. . . . 88
9.7 SSI parameters for the usysv RPI module. . . . 89
9.8 Listing of MPI collective functions indicating which have been optimized for SMP environments. . . . 93
9.9 Listing of MPI collective functions indicating which have been implemented using Shared Memory . . . 95
9.10 SSI parameters for the shmem coll module. . . . 95

12.1 Valid values for the LAM_MPI_THREAD_LEVEL environment variable. 116

Chapter 1

Don’t Panic! (Who Should Read This Document?)

This document probably looks huge to new users. But don’t panic! It is divided up into multiple, relatively independent sections that can be read and digested separately. Although this manual covers a lot of relevant material for all users, the following guidelines are suggested for various types of users. If you are:

- **New to MPI**: First, read Chapter 2 for an introduction to MPI and LAM/MPI. A good reference on MPI programming is also strongly recommended; there are several books available as well as excellent on-line tutorials (e.g., [3, 4, 5, 9]).

  When you’re comfortable with the concepts of MPI, move on to **New to LAM/MPI**.

- **New to LAM/MPI**: If you’re familiar with MPI but unfamiliar with LAM/MPI, first read Chapter 4 for a mini-tutorial on getting started with LAM/MPI. You’ll probably be familiar with many of the concepts described, and simply learn the LAM terminology and commands. Glance over and use as a reference Chapter 7 for the rest of the LAM/MPI commands. Chapter 11 contains some quick tips on common problems with LAM/MPI.

  Assuming that you’ve already got MPI codes that you want to run under LAM/MPI, read Chapter 5 to see exactly what MPI-2 features LAM/MPI supports.

  When you’re comfortable with all this, move on to **Previous LAM user**.

- **Previous LAM user**: As a previous LAM user, you’re probably already fairly familiar with all the LAM commands – their basic functionality hasn’t changed much. However, many of them have grown new options and capabilities, particularly in the area of run-time tunable parameters. So be sure to read Chapters 6 to learn about LAM’s System Services Interface (SSI), Chapters 8 and 9 (LAM and MPI SSI modules), and finally Chapter 12 (miscellaneous LAM/MPI information, features, etc.).

  If you’re curious to see a brief listing of new features in this release, see the release notes in Chapter 3. This isn’t really necessary, but when you’re kicking the tires of this version, it’s a good way to ensure that you are aware of all the new features.

  Finally, even for the seasoned MPI and LAM/MPI veteran, be sure to check out Chapter 10 for information about debugging MPI programs in parallel.

- **System administrator**: Unless you’re also a parallel programmer, you’re reading the wrong document. You should be reading the LAM/MPI Installation Guide [14] for detailed information on how to configure, compile, and install LAM/MPI.

Chapter 2

Introduction to LAM/MPI

This chapter provides a summary of the MPI standard and the LAM/MPI implementation of that standard.

2.1 About MPI

The Message Passing Interface (MPI) [2, 7], is a set of API functions enabling programmers to write high-performance parallel programs that pass messages between processes to make up an overall parallel job. MPI is the culmination of decades of research in parallel computing, and was created by the MPI Forum – an open group representing a wide cross-section of industry and academic interests. More information, including the both volumes of the official MPI standard, can be found at the MPI Forum web site.[1]

MPI is suitable for "big iron" parallel machines such as the IBM SP, SGI Origin, etc., but it also works in smaller environments such as a group of workstations. Since clusters of workstations are readily available at many institutions, it has become common to use them as a single parallel computing resource running MPI programs. The MPI standard was designed to support portability and platform independence. As a result, users can enjoy cross-platform development capability as well as transparent heterogenous communication. For example, MPI codes which have been written on the RS-6000 architecture running AIX can be ported to a SPARC architecture running Solaris with little or no modifications.

2.2 About LAM/MPI

LAM/MPI is a high-performance, freely available, open source implementation of the MPI standard that is researched, developed, and maintained at the Open Systems Lab at Indiana University. LAM/MPI supports all of the MPI-1 Standard and much of the MPI-2 standard. More information about LAM/MPI, including all the source code and documentation, is available from the main LAM/MPI web site.[2]

LAM/MPI is not only a library that implements the mandated MPI API, but also the LAM run-time environment: a user-level, daemon-based run-time environment that provides many of the services required by MPI programs. Both major components of the LAM/MPI package are designed as component frameworks – extensible with small modules that are selectable (and configurable) at run-time. This component framework is known as the System Services Interface (SSI). The SSI component architectures are fully documented in [8, 10, 11, 12, 13, 14, 15].

[1] http://www.mpi-forum.org/
[2] http://www.lam-mpi.org/

Chapter 3

Release Notes

This chapter contains release notes as they pertain to the run-time operation of LAM/MPI. The Installation Guide contains additional release notes on the configuration, compilation, and installation of LAM/MPI.

3.1 New Feature Overview

A full, high-level overview of all changes in the 7 series (and previous versions) can be found in the `HISTORY` file that is included in the LAM/MPI distribution.

This docuemntation was originally written for LAM/MPI v7.0. Changebars are used extensively throughout the document to indicate changes, updates, and new features in the versions since 7.0. The change bars indicate a version number in which the change was introduced.

Major new features specific to the 7 series include the following:

- LAM/MPI 7.0 is the first version to feature the System Services Interface (SSI). SSI is a "pluggable" framework that allows for a variety of run-time selectable modules to be used in MPI applications. For example, the selection of which network to use for MPI point-to-point message passing is now a run-time decision, not a compile-time decision. ⊤ (7.1)

  SSI modules can be built as part of the MPI libraries that are linked into user applications or as standalone dynamic shared objects (DSOs). When compiled as DSOs, all SSI modules are installed in `$prefix/lib/lam`; new modules can be added to or removed from an existing LAM installation simply by putting new DSOs in that directory (there is no need to recompile or relink user applications). ⊥ (7.1)

- When used with supported back-end checkpoint/restart systems, LAM/MPI can checkpoint parallel MPI jobs (see Section 9.5, page 96 for more details).

- LAM/MPI supports the following underlying networks for MPI communication, including several run-time tunable-parameters for each (see Section 9.3, page 76 for more details):

  – TCP/IP, using direct peer-to-peer sockets
  – Myrinet, using the native gm message passing library ⊤ (7.1)
  – Infinband, using the Mellanox VAPI (mVAPI) message passing library ⊥ (7.1)
  – Shared memory, using either spin locks or semaphores

  - "LAM Daemon" mode, using LAM's native run-time environment message passing

- LAM's run-time environment can now be "natively" executed in the following environments (see Section 8.1, page 65 for more details):

  - BProc clusters
  - Globus grid environments (beta level support)
  - Traditional `rsh` / `ssh`-based clusters
  - OpenPBS/PBS Pro/Torque batch queue jobs
  - SLURM batch queue systems

- Improvements to collective algorithms:

  - Several collective algorithms have now been made "SMP-aware", exhibiting better performance when enabled and executed on clusters of SMPs (see Section 9.4, page 89 for more details).
  - Several collective now use shared memory collective algorithms (not based on MPI point-to-point communication) when all processes in a communicator are on the same node.
  - Collectives on intercommunicators are now supported.

- Full support of the TotalView parallel debugger (see Section 10.2, page 103 for more details).
- Support for the MPI-2 portable MPI process startup command `mpiexec` (see Section 7.12, page 58 for more details).
- Full documentation for system administrators, users, and developers [8, 10, 11, 12, 13, 14, 15].
- Various MPI enhancements:

  - C++ bindings are provided for all supported MPI functionality.
  - Upgraded the included ROMIO package [16, 17] to version 1.2.5.1 for MPI I/O support.
  - Per MPI-2:4.8 free the MPI_COMM_SELF communicator at the beginning of MPI_FINALIZE, allowing user-specified functions to be automatically invoked.
  - Formal support for MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, and MPI_THREAD_SERIALIZED. MPI_THREAD_MULTIPLE is not supported (see Section 12.4, page 116 for more details).
  - Significantly increased the number of tags and communicators supported in most RPIs.
  - Enhanced scheduling capabilities for MPI_COMM_SPAWN.

- Various LAM run-time environment enhancements:

  - New `laminfo` command that provides detailed information about a given LAM/MPI installation.
  - Use `TMPDIR` environment variable for LAM's session directory.
  - Restore the original `umask` when creating MPI processes.

  - Allow Fortran MPI processes to change how their name shows up in `mpitask`.
  - Better `SIGTERM` support in the LAM daemon; catch the signal and ensure that all sub-processes are killed and resources are released.

- Deprecated functionality (may disappear in future releases of LAM/MPI):

  - `LAMRSH`: The `LAMRSH` environment variable has been deprecated in favor of the `boot_rsh_agent` parameter to the rsh SSI boot module.
  - `LAM_MPI_SOCKET_SUFFIX`: The `LAM_MPI_SOCKET_SUFFIX` has been deprecated in favor of the `LAM_MPI_SESSION_SUFFIX` environment variable.

3.2 Known Issues

3.2.1 `mpirun` and MPI Application cr Module Disagreement

Due to ordering issues in LAM's MPI_INIT startup sequence, it is possible for `mpirun` to believe that it can checkpoint an MPI application when the application knows that it cannot be checkpointed. A common case of this is when an un-checkpointable RPI module is selected for the MPI application, but checkpointing services are available.

In this case, even though there is a mismatch between `mpirun` and the MPI application, there is no actual harm. Regardless of what `mpirun` believes, attempting to checkpoint the MPI application will fail.

3.2.2 Checkpoint Support Disabled for Spawned Processes

Checkpointing support is only enabled for MPI-1 processes – spawned processes will have checkpointing support explicitly disabled (regardless of the SSI parameters passed and the back-end checkpointing support available).

3.2.3 BLCR Support Only Works When Compiled Statically

Due to linker ordering issues, BLCR checkpointing support only works when the blcr modules are compiled statically into LAM. Attempting to use the blcr modules are dynamic shared objects will result in errors when compiling MPI applications (the error will complain that `libpthread` must be listed *after* `libcr`).

3.2.4 Infiniband rpi Module

The Infiniband (ib) module implementation in LAM/MPI is based on the IB send/receive protocol for tiny messages and RDMA protocol for long messages. Future optmizations include allowing tiny messages to use RDMA (for potentialy latency performance improvements for tiny messages).

The ib rpi has been tested with Mellanox VAPI thca-linux-3.2-build-024. Other versions of VAPI, to include OpenIB and versions from other vendors have not been well tested. Whichever Infiniband driver is used, it must include support for shared completion queues. Mellanox VAPI, for example, did not include support for this feature until mVAPI v3.0. **If your Infiniband driver does not support shared completion queues, the LAM/MPI** ib rpi **will not function properly.** Symptoms will include LAM hanging or crashing during MPI_INIT.

Note that the 7.1.x versions of the ib rpi will not scale well to large numbers of nodes because they register a fixed number of buffers (M bytes) for each process peer during MPI_INIT. Hence, for an N-process MPI_COMM_WORLD, the total memory registered by each process during MPI_INIT is $(N-1) \times M$ bytes. This can be prohibitive as N grows large.

This effect can be limited, however, by decreasing the number and size of buffers that the ib rpi module via SSI parameters at run-time. See the Section 9.3.4 (page 81) for more details.

3.3 Usage Notes

3.3.1 Operating System Bypass Communication: Myrinet and Infiniband

The gm and ib RPI modules require an additional memory manager in order to run properly. On most systems, LAM will automatically select the proper memory manager and the system administrator / end user doesn't need to know anything about this. However, on some systems and/or in some applications, extra work is required.

The issue is that OS-bypass networks such as Myrinet and Infiniband require virtual pages to be "pinned" down to specific hardware addresses before they can be used by the Myrinet/Infiniband NIC hardware. This allows the NIC communication processor to operate on memory buffers independent of the main CPU because it knows that the buffers will never be swapped out (or otherwise be relocated in memory) before the operation is complete.[1]

LAM performs the "pinning" operation behind the scenes; for example, if application MPI_SENDs a buffer using the gm or ib RPI modules, LAM will automatically pin the buffer before it is sent. However, since pinning is a relatively expensive operation, LAM usually leaves buffers pinned when the function completes (e.g., MPI_SEND). This typically speeds up future sends and receives because the buffer does not need to be [re-]pinned. However, if the user frees this memory, the buffer *must* be unpinned before it is given back to the operating system. This is where the additional memory manager comes in.

LAM will, by default, intercept calls to `malloc()`, `calloc()`, and `free()` by use of the ptmalloc, ptmalloc2, or Mac OS X dynlib functionality (note that C++ `new` and `delete` are *not* intercepted). However, this is actually only an unfortunate side effect: LAM really only needs to intercept the `sbrk()` function in order to catch memory before it is returned to the operating system. Specifically, an internal LAM routine runs during `sbrk()` to ensure that all memory is properly unpinned before it is given back to the operating system.

There is, sadly, no easy, portable way to intercept `sbrk()` without also intercepting `malloc()` et al. In most cases, however, this is not a problem: the user's application invokes `malloc()` and obtains heap memory, just as expected (and the other memory functions also function as expected). However, there are some applications do their own intercepting of `malloc()` (et al.). These applications will not work properly with a default installation of LAM/MPI.

To fix this problem, LAM allows you to disable all memory management, but only if the top-level application promises to invoke an internal LAM handler function when `sbrk()` is invoked (*before* the memory is returned to the operating system). This is accomplished by configuring LAM with the following switch:

```
shell$ configure --with-memory-manager=external ...
```

[1] Surprisingly, this memory management is unnecessary on Solaris. The details are too lengthy for this document.

"`external`" specifically indicates that if the gm or ib RPI modules are used, the application promises to invoke the internal LAM function for unpinning memory as required. Note that this function is irrelevant (but harmless) when any other RPI module is used. The function that must be invoked is prototyped in `<mpi.h>`:

```
void lam_handle_free(void *buf, size_t length);
```

For applications that must use this functionality, it is probably safest to wrap the call to `lam_handle_free()` in the following preprocessor conditional:

```
#include <mpi.h>

int my_sbrk(...) {
  /* ...sbrk() functionality... */
#if defined(LAM_MPI)
  lam_handle_free(bufer, length);
#endif
  /* ...rest of sbrk() functionality... */
}
```

Note that when LAM is configured this way, *all* MPI applications that use the gm or ib RPI modules must invoke this function as required. Failure to do so will result in undefined behavior. ⊥ (7.1)

3.4 Platform-Specific Notes

3.4.1 Provided RPMs

If you install LAM/MPI via an official RPM from the LAM/MPI web site (or one of its mirrors), you may not have all the SSI modules that are described in Chapters 8 and 9. The modules that are shipped in 7.1.5b2 are listed in Table 3.1. If you need modules that are not provided in the RPMs, you will likely need to download and install the source LAM/MPI tarball.

Boot	Collective	Checkpoint/Restart	RPI
globus	lam_basic	self	crtcp
rsh	smp		lamd
slurm	shmem		sysv
			tcp
			usysv

Table 3.1: SSI modules that are included in the official LAM/MPI RPMs.

This is for multiple reasons:

- If provided as a binary, each SSI module may require a specific configuration (e.g., a specific version of the back-end software that it links to/interacts with). Since each SSI module is orthogonal to other modules, and since the back-end software systems that each SSI module interacts with may release new versions at any time, the number of combinations that would need to be provided is exponential.

The logistics of attempting to provide pre-compiled binaries for all of these configurations is beyond the capability of the LAM Team. As a direct result, significant effort has going into making building LAM/MPI from the source distribution as simple and all-inclusive as possible.

- Although LAM/MPI is free software (and freely distributable), some of the systems that its modules can interact with are not. The LAM Team cannot distribute modules that contain references to non-freely-distributable code.

The `laminfo` command can be used to see which SSI modules are available in your LAM/MPI installation.

3.4.2 Filesystem Issues

Case-insensitive filesystems. On systems with case-insensitive filesystems (such as Mac OS X with HFS+, Linux with NTFS, or Microsoft Windows(TM) (Cygwin)), the `mpicc` and `mpiCC` commands will both refer to the same executable. This obviously makes distinguishing between the `mpicc` and `mpiCC` wrapper compilers impossible. LAM will attempt to determine if you are building on a case-insensitive filesystem. If you are, the C++ wrapper compiler will be called `mpic++`. Otherwise, the C++ compiler will be called `mpiCC` (although `mpic++` will also be available).

NFS-shared `/tmp`. The LAM per-session directory may not work properly when hosted in an NFS directory, and may cause problems when running MPI programs and/or supplementary LAM run-time environment commands. If using a local filesystem is not possible (e.g., on diskless workstations), the use of `tmpfs` or `tinyfs` is recommended. LAM's session directory will not grow large; it contains a small amount of meta data as well as known endpoints for Unix sockets to allow LAM/MPI programs to contact the local LAM run-time environment daemon.

AFS and tokens/permissions. AFS has some peculiarities, especially with file permissions when using `rsh`/`ssh`.

Many sites tend to install the AFS `rsh` replacement that passes tokens to the remote machine as the default `rsh`. Similarly, most modern versions of `ssh` have the ability to pass AFS tokens. Hence, if you are using the rsh boot module with `recon` or `lamboot`, your AFS token will be passed to the remote LAM daemon automatically. If your site does not install the AFS replacement `rsh` as the default, consult the documentation on `--with-rsh` to see how to set the path to the `rsh` that LAM will use.

Once you use the replacement `rsh` or an AFS-capable `ssh`, you should get a token on the target node when using the rsh boot module.[2] This means that your LAM daemons are running with your AFS token, and you should be able to run any program that you wish, including those that are not `system:anyuser` accessible. You will even be able to write into AFS directories where you have write permission (as you would expect).

Keep in mind, however, that AFS tokens have limited lives, and will eventually expire. This means that your LAM daemons (and user MPI programs) will lose their AFS permissions after some specified time unless you renew your token (with the `klog` command, for example) on the originating machine before the token runs out. This can play havoc with long-running MPI programs that periodically write out file results; if you lose your AFS token in the middle of a run, and your program tries to write out to a file, it will not have permission to, which may cause Bad Things to happen.

[2] If you are using a different boot module, you may experience problems with obtaining AFS tokens on remote nodes.

If you need to run long MPI jobs with LAM on AFS, it is usually advisable to ask your AFS administrator to increase your default token life time to a large value, such as 2 weeks.

3.4.3 Dynamic/Embedded Environments

In LAM/MPI version 7.1.5b2, some RPI modules may utilize an additional memory manager mechanism (see Section 3.3.1, page 18 for more details). This can cause problems when running MPI processes as dynamically loaded modules. For example, when running a LAM/MPI program as a MEX function in a Matlab environment, normal Unix linker semantics create situations where both the default Unix and the memory management systems are used. This typically results in process failure.

Note that this *only* occurs when LAM/MPI processes are used in a dynamic environment and an additional memory manager is included in LAM/MPI. This appears to occur because of normal Unix semantics; the only way to avoid it is to use the `--with-memory-manager` parameter to LAM's `configure` script, specifying either "`none`" or "`external`" as its value. See the LAM/MPI Installation Guide for more details.

3.4.4 Linux

LAM/MPI is frequently used on Linux-based machines (IA-32 and otherwise). Although LAM/MPI is generally tested on Red Hat and Mandrake Linux systems using recent kernel versions, it should work on other Linux distributions as well.

Note that kernel versions 2.2.0 through 2.2.9 had some TCP/IP performance problems. It seems that version 2.2.10 fixed these problems; if you are using a Linux version between 2.2.0 and 2.2.9, LAM may exhibit poor TCP performance due to the Linux TCP/IP kernel bugs. We recommend that you upgrade to 2.2.10 (or the latest version). See `http://www.lam-mpi.org/linux/` for a full discussion of the problem.

3.4.5 Mac OS X (Absoft Fortran Compilers)

⊤ (7.1.2)

To use the Absoft Fortran compilers with LAM/MPI on OS X, you must have at least version 9.0 EP (Enhancement Pack). Contact `mailto:support@absoft.com` for details.

⊥ (7.1.2)

3.4.6 Microsoft Windows(TM)(Cygwin)

⊤ (7.1)

LAM/MPI is supported on Microsoft Windows (TM) (Cygwin 1.5.5). Currently tcp, sysv, usysv and tcp RPIs are supported. ROMIO is not suported.

In Microsoft Windows(TM) (Cygwin), IPC services are provided by the CygIPC module. Hence, installation and use of the sysv and usysv RPIs require this module. Specifically, sysv and usysv RPIs are installed if and only if the library `libcygipc.a` is found and `ipc-daemon2.exe` is running when configuring LAM/MPI. Furthermore, to use these RPIs, it is necessary to have `ipc-daemon2.exe` running on all the nodes. For detailed instructions on configuring these RPIs, please refer to the LAM/MPI Installation Guide.

Since there are some issues with the use of the native Cygwin terminal for standard IO redirection, it is advised to run MPI applications on xterm. For more information on getting X services for Cygwin, please see the XFree86 web site.[3]

[3] `http://www.cygwin.com/`

Although we have tried to port the complete functionality of LAM/MPI to Cygwin, because of some
⊥ (7.1) outstanding portability issues, execution of LAM/MPI applications on Cygwin may not always be reliable.

3.4.7 Solaris

⊤ (7.1) The gm RPI will fail to function properly on versions of Solaris older than Solaris 7.

The default amount of shared memory available on Solaris is fairly small. It may need to be increased to allow running more than a small number of processes on a single Solaris node using the sysv or usysv RPI modules.[4] For example, if running the LAM test suite on a single node, some tests run several instances of the executable (e.g., 6) which may cause the system to run out of shared memory and therefore cause the
⊥ (7.1) test to fail. Increasing the shared memory limits on the system will allow the test to pass.

[4] See http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html for a good examplantion of Solaris shared memory.

Chapter 4

Getting Started with LAM/MPI

This chapter provides a summary tutorial describing some of the high points of using LAM/MPI. It is not intended as a comprehensive guide; the finer details of some situations will not be explained. However, it is a good step-by-step guide for users who are new to MPI and/or LAM/MPI.

Using LAM/MPI is conceptually simple:

- Launch the LAM run-time environment (RTE)
- Repeat as necessary:
  - Compile MPI program(s)
  - Run MPI program(s)
- Shut down the LAM run-time environment

The tutorial below will describe each of these steps.

4.1 One-Time Setup

This section describes actions that usually only need to be performed once per user in order to setup LAM to function properly.

4.1.1 Setting the Path

One of the main requirements for LAM/MPI to function properly is for the LAM executables to be in your path. This step may vary from site to site; for example, the LAM executables may already be in your path – consult your local administrator to see if this is the case.

NOTE: If the LAM executables are already in your path, you can skip this step and proceed to Section 4.2.

In many cases, if your system does not already provide the LAM executables in your path, you can add them by editing your "dot" files that are executed automatically by the shell upon login (both interactive and non-interactive logins). Each shell has a different file to edit and corresponding syntax, so you'll need to know which shell you are using. Tables 4.1 and 4.2 list several common shells and the associated files that are typically used. Consult the documentation for your shell for more information.

Shell name	Interactive login startup file
`sh` (or Bash named "`sh`")	`.profile`
`csh`	`.cshrc` followed by `.login`
`tcsh`	`.tcshrc` if it exists, `.cshrc` if it does not, followed by `.login`
`bash`	`.bash_profile` if it exists, or `.bash_login` if it exists, or `.profile` if it exists (in that order). Note that some Linux distributions automatically come with `.bash_profile` scripts for users that automatically execute `.bashrc` as well. Consult the `bash` manual page for more information.

Table 4.1: List of common shells and the corresponding environmental setup files commonly used with each for interactive startups (e.g., normal login). All files listed are assumed to be in the `$HOME` directory.

Shell name	Non-interactive login startup file
`sh` (or Bash named "`sh`")	This shell does not execute any file automatically, so LAM will execute the `.profile` script before invoking LAM executables on remote nodes
`csh`	`.cshrc`
`tcsh`	`.tcshrc` if it exists, `.cshrc` if it does not
`bash`	`.bashrc` if it exists

Table 4.2: List of common shells and the corresponding environmental setup files commonly used with each for non-interactive startups (e.g., normal login). All files listed are assumed to be in the `$HOME` directory.

You'll also need to know the directory where LAM was installed. For the purposes of this tutorial, we'll assume that LAM is installed in `/usr/local/lam`. And to re-emphasize a critical point: these are only guidelines – the specifics may vary depending on your local setup. Consult your local system or network administrator for more details.

Once you have determined all three pieces of information (what shell you are using, what directory LAM was installed to, and what the appropriate "dot" file to edit), open the "dot" file in a text editor and follow the general directions listed below:

- For the Bash, Bourne, and Bourne-related shells, add the following lines:

```
PATH=/usr/local/lam/bin:$PATH
export PATH
```

- For the C shell and related shells (such as `tcsh`), add the following line:

```
set path = (/usr/local/lam/bin $path)
```

4.1.2 Finding the LAM Manual Pages

LAM includes manual pages for all supported MPI functions as well as all of the LAM executables. While this step *is not necessary* for correct MPI functionality, it can be helpful when looking for MPI or LAM-specific information.

Using Tables 4.1 and 4.2, find the right "dot" file to edit. Assuming again that LAM was installed to `/usr/local/lam`, open the appropriate "dot" file in a text editor and follow the general directions listed below:

- For the Bash, Bourne, and Bourne-related shells, add the following lines:

```
MANPATH=/usr/local/lam/man:$MANPATH
export MANPATH
```

- For the C shell and related shells (such as `tcsh`), add the following lines:

```
if ($?MANPATH == 0) then
  setenv MANPATH /usr/local/lam/man
else
  setenv MANPATH /usr/local/lam/man:$MANPATH
endif
```

4.2 System Services Interface (SSI)

LAM/MPI is built around a core of System Services Interface (SSI) plugin modules. SSI allows run-time selection of different underlying services within the LAM/MPI run-time environment, including tunable parameters that can affect the performance of MPI programs.

While this tutorial won't go into much detail about SSI, just be aware that you'll see mention of "SSI" in the text below. In a few places, the tutorial passes parameters to various SSI modules through either environment variables and/or the `-ssi` command line parameter to several LAM commands.

See other sections in this manual for a more complete description of SSI (Chapter 6, page 43), how it works, and what run-time parameters are available (Chapters 8 and 9, pages 65 and 75, respectively). Also, the `lamssi(7)`, `lamssi_boot(7)`, `lamssi_coll(7)`, `lamssi_cr(7)`, and `lamssi_rpi(7)` manual pages each provide additional information on LAM's SSI mechanisms.

4.3 What Does Your LAM/MPI Installation Support?

LAM/MPI can be installed with a large number of configuration options. It depends on what choices your system/network administrator made when configuring and installing LAM/MPI. The `laminfo` command is provided to show the end-user with information about what the installed LAM/MPI supports. Running "`laminfo`" (with no arguments) prints a list of LAM's capabilities, including all of its SSI modules.

Among other things, this shows what language bindings the installed LAM/MPI supports, what underlying network transports it supports, and what directory LAM was installed to. The `-parsable` option prints out all the same information, but in a conveniently machine-parsable format (suitable for using with scripts).

4.4 Booting the LAM Run-Time Environment

Before any MPI programs can be executed, the LAM run-time environment must be launched. This is typically called "booting LAM." A successfully boot process creates an instance of the LAM run-time environment commonly referred to as the "LAM universe."

LAM's run-time environment can be executed in many different environments. For example, it can be run interactively on a cluster of workstations (even on a single workstation, perhaps to simulate parallel execution for debugging and/or development). Or LAM can be run in production batch scheduled systems.

This example will focus on a traditional `rsh` / `ssh`-style workstation cluster (i.e., not under batch systems), where `rsh` or `ssh` is used to launch executables on remote workstations.

4.4.1 The Boot Schema File (a.k.a, "Hostfile", "Machinefile")

When using `rsh` or `ssh` to boot LAM, you will need a text file listing the hosts on which to launch the LAM run-time environment. This file is typically referred to as a "boot schema", "hostfile", or "machinefile." For example:

```
# My boot schema
node1.cluster.example.com
node2.cluster.example.com
node3.cluster.example.com cpu=2
node4.cluster.example.com cpu=2
```

Four nodes are specified in the above example by listing their IP hostnames. Note also the "`cpu=2`" that follows the last two entries. This tells LAM that these machines each have two CPUs available for running MPI programs (e.g., `node3` and `node4` are two-way SMPs). It is important to note that the number of CPUs specified here has *no* correlation to the physicial number of CPUs in the machine. It is simply a

convenience mechanism telling LAM how many MPI processes we will typically launch on that node. The ramifications of the `cpu` key will be discussed later.

The location of this text file is irrelevant; for the purposes of this example, we'll assume that it is named `hostfile` and is located in the current working directory.

4.4.2 The `lamboot` Command

The `lamboot` command is used to launch the LAM run-time environment. For each machine listed in the boot schema, the following conditions must be met for LAM's run-time environment to be booted correctly:

- The machine must be reachable and operational.
- The user must be able to non-interactively execute arbitrary commands on the machine (e.g., without being prompted for a password).
- The LAM executables must be locatable on that machine, using the user's shell search path.
- The user must be able to write to the LAM session directory (usually somewhere under `/tmp`).
- The shell's start-up scripts must not print anything on standard error.
- All machines must be able to resolve the fully-qualified domain name (FQDN) of all the machines being booted (including itself).

Once all of these conditions are met, the `lamboot` command is used to launch the LAM run-time environment. For example:

```
shell$ lamboot -v -ssi boot rsh hostfile

LAM 7.0/MPI 2 C++/ROMIO - Indiana University

n0<1234> ssi:boot:base:linear: booting n0 (node1.cluster.example.com)
n0<1234> ssi:boot:base:linear: booting n1 (node2.cluster.example.com)
n0<1234> ssi:boot:base:linear: booting n2 (node3.cluster.example.com)
n0<1234> ssi:boot:base:linear: booting n3 (node4.cluster.example.com)
n0<1234> ssi:boot:base:linear: finished
```

The parameters passed to `lamboot` in the example above are as follows:

- `-v`: Make `lamboot` be slightly verbose.
- `-ssi boot rsh`: Ensure that LAM uses the `rsh`/`ssh` boot module to boot the LAM universe. Typically, LAM chooses the right boot module automatically (and therefore this parameter is not typically necessary), but to ensure that this tutorial does exactly what we want it to do, we use this parameter to absolutely ensure that LAM uses `rsh` or `ssh` to boot the universe.
- `hostfile`: Name of the boot schema file.

Common causes of failure with the `lamboot` command include (but are not limited to):

- User does not have permission to execute on the remote node. This typically involves setting up a `$HOME/.rhosts` file (if using `rsh`), or properly configured SSH keys (using using `ssh`).

  Setting up `.rhosts` and/or SSH keys for password-less remote logins are beyond the scope of this tutorial; consult local documentation for `rsh` and `ssh`, and/or internet tutorials on setting up SSH keys.[1]

- The first time a user uses `ssh` to execute on a remote node, `ssh` typically prints a warning to the standard error. LAM will interpret this as a failure. If this happens, `lamboot` will complain that ⊤ (7.1) something unexpectedly appeared on `stderr`, and abort. One solution is to manually `ssh` to each node in the boot schema once in order to eliminate the `stderr` warning, and then try `lamboot` again. Another is to use the `boot_rsh_ignore_stderr` SSI parameter. We haven't discussed SSI parameters yet, so it is probably easiest at this point to manually `ssh` to a small number of nodes to ⊥ (7.1) get the warning out of the way.

If you have having problems with `lamboot`, try using the `-d` option to `lamboot`, which will print enormous amounts of debugging output which can be helpful for determining what the problem is. Additionally, check the `lamboot(1)` man page as well as the LAM FAQ on the main LAM web site[2] under the section "Booting LAM" for more information.

4.4.3 The `lamnodes` Command

An easy way to see how many nodes and CPUs are in the current LAM universe is with the `lamnodes` command. For example, with the LAM universe that was created from the boot schema in Section 4.4.1, running the `lamnodes` command would result in the following output:

```
shell$ lamnodes
n0 node1.cluster.example.com:1:origin,this_node
n1 node2.cluster.example.com:1:
n2 node3.cluster.example.com:2:
n3 node4.cluster.example.com:2:
```

The "`n`" number on the far left is the LAM node number. For example, "`n3`" uniquely refers to `node4`. Also note the third column, which indicates how many CPUs are available for running processes on that node. In this example, there are a total of 6 CPUs available for running processes. This information is from the "`cpu`" key that was used in the hostfile, and is helpful for running parallel processes (see below).

Finally, the "`origin`" notation indicates which node `lamboot` was executed from. "`this_node`" obviously indicates which node `lamnodes` is running on.

4.5 Compiling MPI Programs

Note that it is *not* necessary to have LAM booted to compile MPI programs.

Compiling MPI programs can be a complicated process:

[1] As of this writing, a Google search for "ssh keys" turned up several decent tutorials; including any one of them here would significantly increase the length of this already-tremendously-long manual.

[2] http://www.lam-mpi.org/faq/

- The same compilers should be used to compile/link user MPI programs as were used to compile LAM itself.
- Depending on the specific installation configuration of LAM, a variety of `-I`, `-L`, and `-l` flags (and possibly others) may be necessary to compile and/or link a user MPI program.

LAM/MPI provides "wrapper" compilers to hide all of this complexity. These wrapper compilers simply add the correct compiler/linker flags and then invoke the underlying compiler to actually perform the compilation/link. As such, LAM's wrapper compilers can be used just like "real" compilers.

The wrapper compilers are named `mpicc` (for C programs), `mpiCC` and `mpic++` (for C++ programs), and `mpif77` (for Fortran programs). For example:

```
shell$ mpicc -g -c foo.c
shell$ mpicc -g -c bar.c
shell$ mpicc -g foo.o bar.o -o my_mpi_program
```

Note that no additional compiler and linker flags are required for correct MPI compilation or linking. The resulting `my_mpi_program` is ready to run in the LAM run-time environment. Similarly, the other two wrapper compilers can be used to compile MPI programs for their respective languages:

```
shell$ mpiCC -O c++_program.cc -o my_c++_mpi_program
shell$ mpif77 -O f77_program.f -o my_f77_mpi_program
```

Note, too, that any other compiler/linker flags can be passed through the wrapper compilers (such as `-g` and `-O`); they will simply be passed to the back-end compiler.

Finally, note that giving the `-showme` option to any of the wrapper compilers will show both the name of the back-end compiler that will be invoked, and also all the command line options that would have been passed for a given compile command. For example (line breaks added to fit in the documentation):

```
shell$ mpiCC -O c++_program.cc -o my_c++_program -showme
g++ -I/usr/local/lam/include -pthread -O c++_program.cc -o \
my_c++_program -L/usr/local/lam/lib -llammpio -llammpi++ -lpmpi \
-llamf77mpi -lmpi -llam -lutil -pthread
```

⊤ (7.1)

Note that the wrapper compilers only add all the LAM/MPI-specific flags when a command-line argument that does not begin with a dash ("-") is present. For example:

```
shell$ mpicc
gcc: no input files
shell$ mpicc --version
gcc (GCC) 3.2.2 (Mandrake Linux 9.1 3.2.2-3mdk)
Copyright (C) 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

⊥ (7.1)

4.5.1 Sample MPI Program in C

The following is a simple "hello world" C program.

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

int main(int argc, char *argv[]) {
  int rank, size;

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

  printf(``Hello, world! I am %d of %d\n'', rank, size);

  MPI_Finalize();
  return 0;
}
```

This program can be saved in a text file and compiled with the `mpicc` wrapper compiler.

```
shell$ mpicc hello.c -o hello
```

4.5.2 Sample MPI Program in C++

The following is a simple "hello world" C++ program.

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

using namespace std;

int main(int argc, char *argv[]) {
  int rank, size;

  MPI::Init(argc, argv);
  rank = MPI::COMM_WORLD.Get_rank();
  size = MPI::COMM_WORLD.Get_size();

  cout << ``Hello, world! I am '' << rank << `` of '' << size << endl;

  MPI::Finalize();
  return 0;
}
```

This program can be saved in a text file and compiled with the `mpiCC` wrapper compiler (or `mpic++` if on case-insensitive filesystems, such as Mac OS X's HFS+).

```
shell$ mpiCC hello.cc -o hello
```

4.5.3 Sample MPI Program in Fortran

The following is a simple "hello world" Fortran program.

```
program hello
include 'mpif.h'
integer rank, size, ierr

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

print *, "Hello, world! I am ", rank, " of ", size

call MPI_FINALIZE(ierr)
stop
end
```

This program can be saved in a text file and compiled with the `mpif77` wrapper compiler.

```
shell$ mpif77 hello.f -o hello
```

4.6 Running MPI Programs

Once you have successfully established a LAM universe and compiled an MPI program, you can run MPI programs in parallel.

In this section, we will show how to run a Single Program, Multiple Data (SPMD) program. Specifically, we will run the `hello` program (from the previous section) in parallel. The `mpirun` and `mpiexec` commands are used for launching parallel MPI programs, and the `mpitask` commands can be used to provide crude debugging support. The `lamclean` command can be used to completely clean up a failed MPI program (e.g., if an error occurs).

4.6.1 The `mpirun` Command

The `mpirun` command has many different options that can be used to control the execution of a program in parallel. We'll explain only a few of them here.

The simplest way to launch the `hello` program across all CPUs listed in the boot schema is:

```
shell$ mpirun C hello
```

The `C` option means "launch one copy of `hello` on every CPU that was listed in the boot schema." The `C` notation is therefore convenient shorthand notation for launching a set of processes across a group of SMPs.

Another method for running in parallel is:

```
shell$ mpirun N hello
```

The `N` option has a different meaning than `C` – it means "launch one copy of `hello` on every node in the LAM universe." Hence, `N` disregards the CPU count. This can be useful for multi-threaded MPI programs.

Finally, to run an absolute number of processes (regardless of how many CPUs or nodes are in the LAM universe):

```
shell$ mpirun -np 4 hello
```

This runs 4 copies of `hello`. LAM will "schedule" how many copies of `hello` will be run in a round-robin fashion on each node by how many CPUs were listed in the boot schema file.[3] For example, on the LAM universe that we have previously shown in this tutorial, the following would be launched:

- 1 `hello` would be launched on `n0` (named `node1`)
- 1 `hello` would be launched on `n1` (named `node2`)
- 2 `hello`s would be launched on `n2` (named `node3`)

Note that any number can be used – if a number is used that is greater than how many CPUs are in the LAM universe, LAM will "wrap around" and start scheduling starting with the first node again. For example, using `-np 10` would result in the following schedule:

- 2 `hello`s on `n0` (1 from the first pass, and then a second from the "wrap around")
- 2 `hello`s on `n1` (1 from the first pass, and then a second from the "wrap around")
- 4 `hello`s on `n2` (2 from the first pass, and then 2 more from the "wrap around")
- 2 `hello`s on `n3`

The `mpirun(1)` man page contains much more information and `mpirun` and the options available. For example, `mpirun` also supports Multiple Program, Multiple Data (MPMD) programs, although it is not discussed here. Also see Section 7.14 (page 60) in this document.

4.6.2 The `mpiexec` Command

The MPI-2 standard recommends the use of `mpiexec` for portable MPI process startup. In LAM/MPI, `mpiexec` is functionally similar to `mpirun`. Some options that are available to `mpirun` are not available to `mpiexec`, and vice-versa. The end result is typically the same, however – both will launch parallel MPI programs; which you should use is likely simply a personal choice.

That being said, `mpiexec` offers more convenient access in three cases:

- Running MPMD programs
- Running heterogeneous programs
- Running "one-shot" MPI programs (i.e., boot LAM, run the program, then halt LAM)

The general syntax for `mpiexec` is:

```
shell$ mpiexec <global_options> <cmd1> : <cmd2> : ...
```

[3] Note that the use of the word "schedule" does not imply that LAM has ties with the operating system for scheduling purposes (it doesn't). LAM "scheduled" on a per-node basis; so selecting a process to run means that it has been assigned and launched on that node. The operating system is solely responsible for all process and kernel scheduling.

Running MPMD Programs

For example, to run a manager/worker parallel program, where two different executables need to be launched (i.e., `manager` and `worker`, the following can be used:

```
shell$ mpiexec -n 1 manager : worker
```

This runs one copy of `manager` and one copy of `worker` for every CPU in the LAM universe.

Running Heterogeneous Programs

Since LAM is a heterogeneous MPI implementation, it supports running heterogeneous MPI programs. For example, this allows running a parallel job that spans a Sun SPARC machine and an IA-32 Linux machine (even though they are opposite endian machines). Although this can be somewhat complicated to setup (remember that you will first need to `lamboot` successfully, which essentially means that LAM must be correctly installed on both architectures), the `mpiexec` command can be helpful in actually running the resulting MPI job.

Note that you will need to have two MPI executables – one compiled for Solaris (e.g., `hello.solaris`) and one compiled for Linux (e.g., `hello.linux`). Assuming that these executables both reside in the same directory, and that directory is available on both nodes (or the executables can be found in the `PATH` on their respective machines), the following command can be used:

```
shell$ mpiexec -arch solaris hello.solaris : -arch linux hello.linux
```

This runs the `hello.solaris` command on all nodes in the LAM universe that have the string "solaris" anywhere in their architecture string, and `hello.linux` on all nodes that have "linux" in their architecture string. The architecture string of a given LAM installation can be found by running the `laminfo` command.

"One-Shot" MPI Programs

In some cases, it seems like extra work to boot a LAM universe, run a single MPI job, and then shut down the universe. Batch jobs are good examples of this – since only one job is going to be run, why does it take three commands? `mpiexec` provides a convenient way to run "one-shot" MPI jobs.

```
shell$ mpiexec -machinefile hostfile hello
```

This will invoke `lamboot` with the boot schema named "`hostfile`", run the MPI program `hello` on all available CPUs in the resulting universe, and then shut down the universe with the `lamhalt` command (which we'll discuss in Section 4.7, below).

4.6.3 The `mpitask` Command

The `mpitask` command is analogous to the sequential Unix command `ps`. It shows the current status of the MPI program(s) being executed in the LAM universe, and displays primitive information about what MPI function each process is currently executing (if any). Note that in normal practice, the `mpimsg` command only gives a snapshot of what messages are flowing between MPI processes, and therefore is usually only accurate at that single point in time. To really debug message passing traffic, use a tool such as message passing analyzer (e.g., XMPI), or a parallel debugger (e.g., TotalView).

`mpitask` can be run from any node in the LAM universe.

4.6.4 The `lamclean` Command

The `lamclean` command completely removes all running programs from the LAM universe. This can be useful if a parallel job crashes and/or leaves state in the LAM run-time environment (e.g., MPI-2 published names). It is usually run with no parameters:

```
shell$ lamclean
```

`lamclean` is typically only necessary when developing / debugging MPI applications – i.e., programs that hang, messages that are left around, etc. Correct MPI programs should terminate properly, clean up all their messages, unpublish MPI-2 names, etc.

4.7 Shutting Down the LAM Universe

When finished with the LAM universe, it should be shut down with the `lamhalt` command:

```
shell$ lamhalt
```

In most cases, this is sufficient to kill all running MPI processes and shut down the LAM universe.

However, in some rare conditions, `lamhalt` may fail. For example, if any of the nodes in the LAM universe crashed before running `lamhalt`, `lamhalt` will likely timeout and potentially not kill the entire LAM universe. In this case, you will need to use the `lamwipe` command to guarantee that the LAM universe has shut down properly:

```
shell$ lamwipe -v hostfile
```

where `hostfile` is the same boot schema that was used to boot LAM (i.e., all the same nodes are listed). `lamwipe` will forcibly kill all LAM/MPI processes and terminate the LAM universe. This is a slower process than `lamhalt`, and is typically not necessary.

Chapter 5

Supported MPI Functionality

This chapter discusses the exact levels of MPI functionality that is supported by LAM/MPI.

5.1 MPI-1 Support

LAM 7.1.5b2 has support for all MPI-1 functionality.

5.1.1 Language Bindings

LAM provides C, C++, and Fortran 77 bindings for all MPI-1 functions, types, and constants. Profiling support is available in all three languages (if LAM was configured and compiled with profiling support). The `laminfo` command can be used to see if profiling support was included in LAM/MPI. ⊤ (7.1)

Support for optional Fortran types has now been added. Table 5.1 lists the new datatypes. Note that MPI_INTEGER8 and MPI_REAL16 are listed even though they are not defined by the MPI standard. Support for these types is included per request from LAM/MPI users.

Supported Datatypes	
MPI_INTEGER1	MPI_INTEGER2
MPI_INTEGER4	MPI_INTEGER8
MPI_REAL4	MPI_REAL8
MPI_REAL16	

Table 5.1: Supported optional fortran datatypes.

⊥ (7.1)

5.1.2 MPI_CANCEL

MPI_CANCEL works properly for receives, but will almost never work on sends. MPI_CANCEL is most frequently used with unmatched MPI_IRECV's that were made "in case" a matching message arrived. This simply entails removing the receive request from the local queue, and is fairly straightforward to implement.

Actually canceling a send operation is much more difficult because some meta information about a message is usually sent immediately. As such, the message is usually at least partially sent before an MPI_CANCEL is issued. Trying to chase down all the particular cases is a nightmare, to say the least.

As such, the LAM Team decided not to implement MPI_CANCEL on sends, and instead concentrate on other features.

But in true MPI Forum tradition, we would be happy to discuss any code that someone would like to submit that fully implements MPI_CANCEL.

5.2 MPI-2 Support

LAM 7.1.5b2 has support for many MPI-2 features. The main chapters of the MPI-2 standard are listed below, along with a summary of the support provided for each chapter.

5.2.1 Miscellany

Portable MPI Process Startup. The `mpiexec` command is now supported. Common examples include:

```
# Runs 4 copes of the MPI program my_mpi_program
shell$ mpiexec -n 4 my_mpi_program

# Runs my_linux_program on all available Linux machines, and runs
# my_solaris_program on all available Solaris machines
shell$ mpiexec -arch linux my_linux_program : -arch solaris my_solaris_program

# Boot the LAM run-time environment, run my_mpi_program on all
# available CPUs, and then shut down the LAM run-time environment.
shell$ mpiexec -machinefile hostfile my_mpi_program
```

See the `mpiexec(1)` man page for more details on supported options as well as more examples.

Passing NULL to MPI_INIT. Passing NULL as both arguments to MPI_INIT is fully supported.

Version Number. LAM 7.1.5b2 reports its MPI version as 1.2 through the function MPI_GET_VERSION.

Datatype Constructor MPI_TYPE_CREATE_INDEXED_BLOCK. The MPI function MPI_TYPE_CREATE_INDEXED_BLOCK is not supported by LAM/MPI.

Treatment of MPI_Status. Although LAM supports the constants MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE, the function MPI_REQUEST_GET_STATUS is not provided.

Error class for invalid keyval. The error class for invalid keyvals, MPI_ERR_KEYVAL, is fully supported.

Committing committed datatype. Committing a committed datatype is fully supported; its end effect is a no-op.

Allowing user functions at process termination. Attaching attributes to MPI_COMM_SELF that have user-specified delete functions will now trigger these functions to be invoked as the first phase of MPI_FINALIZE. When these functions are run, MPI is still otherwise fully functional.

Determining whether MPI has finished. The function MPI_FINALIZED is fully supported.

The Info object. Full support for MPI_Info objects is provided. See Table 5.2.

Supported Functions		
MPI_INFO_CREATE	MPI_INFO_FREE	MPI_INFO_GET_NTHKEY
MPI_INFO_DELETE	MPI_INFO_GET	MPI_INFO_GET_VALUELEN
MPI_INFO_DUP	MPI_INFO_GET_NKEYS	MPI_INFO_SET

Table 5.2: Supported MPI-2 info functions.

Memory allocation. The MPI_ALLOC_MEM and MPI_FREE_MEM functions will return "special" memory that enable fast memory passing in RPIs that support it. These functions are simply wrappers to `malloc()` and `free()` (respectively) in RPI modules that do not take advantage of "special" memory. These functions can be used portably for potential performance gains.

Language interoperability. Inter-language interoperability is supported. It is possible to initialize LAM/MPI from either C or Fortran and mix MPI calls from both languages. Handle conversions for inter-language interoperability are fully supported. See Table 5.3.

Supported Functions	
MPI_COMM_F2C	MPI_COMM_C2F
MPI_GROUP_F2C	MPI_GROUP_C2F
MPI_TYPE_F2C	MPI_TYPE_C2F
MPI_REQUEST_F2C	MPI_REQUEST_C2F
MPI_INFO_F2C	MPI_INFO_C2F
MPI_WIN_F2C	MPI_WIN_C2F
MPI_STATUS_F2C	MPI_STATUS_C2F

Table 5.3: Supported MPI-2 handle conversion functions.

Error handlers. Communicator and window error handler functions are fully supported; this functionality is not yet supported for MPI_File handles. See Table 5.4

Supported Functions	
MPI_COMM_CREATE_ERRHANDLER	MPI_WIN_CREATE_ERRHANDLER
MPI_COMM_GET_ERRHANDLER	MPI_WIN_GET_ERRHANDLER
MPI_COMM_SET_ERRHANDLER	MPI_WIN_SET_ERRHANDLER

Table 5.4: Supported MPI-2 error handler functions.

New datatype manipulation functions. Several new datatype manipulation functions are provided. Table 5.5 lists the new functions.

Supported Functions	
MPI_GET_ADDRESS	MPI_TYPE_CREATE_SUBARRAY
MPI_TYPE_CREATE_DARRAY	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_CREATE_HINDEXED	MPI_TYPE_GET_EXTENT
MPI_TYPE_CREATE_HVECTOR	MPI_TYPE_GET_TRUE_EXTENT
MPI_TYPE_CREATE_RESIZED	

Table 5.5: Supported MPI-2 new datatype manipulation functions.

New predefined datatypes. Support has been added for the MPI_LONG_LONG_INT, MPI_UNSIGNED_LONG_LONG and MPI_WCHAR basic datatypes.

Canonical MPI_PACK, MPI_UNPACK. Support is not provided for MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, or MPI_PACK_EXTERNAL_SIZE.

5.2.2 Process Creation and Management

LAM/MPI supports all MPI-2 dynamic process management. Table 5.6 lists all the supported functions.

Supported Functions		
MPI_CLOSE_PORT	MPI_COMM_GET_PARENT	MPI_LOOKUP_NAME
MPI_COMM_ACCEPT	MPI_COMM_JOIN	MPI_OPEN_PORT
MPI_COMM_SPAWN	MPI_COMM_CONNECT	MPI_PUBLISH_NAME
MPI_COMM_DISCONNECT	MPI_COMM_SPAWN_MULTIPLE	MPI_UNPUBLISH_NAME

Table 5.6: Supported MPI-2 dynamic functions.

As requested by LAM users, MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE supports some MPI_Info keys for spawning MPMD applications and for more fine-grained control about where children processes are spawned. See the `MPI_Comm_spawn(3)` man page for more details.

These functions supersede the MPIL_COMM_SPAWN function that LAM/MPI introduced in version 6.2b. Hence, MPIL_COMM_SPAWN is no longer available.

5.2.3 One-Sided Communication

Support is provided for get/put/accumulate data transfer operations and for the post/wait/start/complete and fence synchronization operations. No support is provided for window locking.

The datatypes used in the get/put/accumulate operations are restricted to being basic datatypes or single level contiguous/vectors of basic datatypes.

The implementation of the one-sided operations is layered on top of the point-to-point functions, and will thus perform no better than them. Nevertheless it is hoped that providing this support will aid developers in developing and debugging codes using one-sided communication. While LAM provides the required MPI MODE constants, they are ignored by the present implementation.

Table 5.7 lists the functions related to one-sided communication that have been implemented.

Supported Functions		
MPI_ACCUMULATE	MPI_WIN_CREATE	MPI_WIN_POST
MPI_GET	MPI_WIN_FENCE	MPI_WIN_START
MPI_PUT	MPI_WIN_FREE	MPI_WIN_WAIT
MPI_WIN_COMPLETE	MPI_WIN_GET_GROUP	

Table 5.7: Supported MPI-2 one-sided functions.

5.2.4 Extended Collective Operations

⊤ (7.1)

LAM implements the new MPI-2 collective functions MPI_EXSCAN and MPI_ALLTOALLW for intracommunicators.

Intercommunicator collectives are implemented for all the functions listed in Table 5.8. Notably, intercommunicator collectives are *not* defined for MPI_SCAN (because the MPI standard does not define it), MPI_ALLGATHERV, and MPI_EXSCAN.[1]

Supported Functions		
MPI_ALLGATHER	MPI_ALLGATHERV	MPI_ALLTOALL
MPI_ALLTOALLV	MPI_ALLTOALLW	MPI_ALLREDUCE
MPI_REDUCE_SCATTER	MPI_GATHER	MPI_GATHERV
MPI_REDUCE	MPI_BCAST	MPI_SCATTER
MPI_SCATTERV	MPI_BARRIER	

Table 5.8: Supported MPI-2 intercommunicator collective functions.

⊥ (7.1)

5.2.5 External Interfaces

The external interfaces chapter lists several different major topics. LAM's support for these topics is summarized in Table 5.9, and the exact list of functions that are supported is listed in 5.10.

Supported	Description
no	Generalized requests
no	Associating information with MPI_Status
yes	Naming objects
no	Error classes
no	Error codes
yes	Error handlers
yes	Decoding a datatype
yes	MPI and threads
yes	New attribute caching functions
yes	Duplicating a datatype

Table 5.9: Major topics in the MPI-2 chapter "External Interfaces", and LAM's level of support.

[1] These two functions were unfortunately overlooked and forgotten about when LAM/MPI v7.1 was frozen for release.

Supported Functions		
MPI_COMM_SET_NAME MPI_COMM_GET_NAME	MPI_TYPE_SET_NAME MPI_TYPE_GET_NAME	MPI_WIN_SET_NAME MPI_WIN_GET_NAME
MPI_COMM_CREATE_ERRHANDLER MPI_COMM_GET_ERRHANDLER MPI_COMM_SET_ERRHANDLER	MPI_WIN_CREATE_ERRHANDLER MPI_WIN_GET_ERRHANDLER MPI_WIN_SET_ERRHANDLER	
MPI_TYPE_GET_CONTENTS MPI_TYPE_GET_ENVELOPE MPI_TYPE_GET_EXTENT MPI_TYPE_GET_TRUE_EXTENT	MPI_INIT_THREAD MPI_QUERY_THREAD MPI_IS_THREAD_MAIN MPI_TYPE_DUP	
MPI_COMM_CREATE_KEYVAL MPI_COMM_FREE_KEYVAL MPI_COMM_DELETE_ATTR MPI_COMM_GET_ATTR MPI_COMM_SET_ATTR	MPI_TYPE_CREATE_KEYVAL MPI_TYPE_FREE_KEYVAL MPI_TYPE_DELETE_ATTR MPI_TYPE_GET_ATTR MPI_TYPE_SET_ATTR	MPI_WIN_CREATE_KEYVAL MPI_WIN_FREE_KEYVAL MPI_WIN_DELETE_ATTR MPI_WIN_GET_ATTR MPI_WIN_SET_ATTR

Table 5.10: Supported MPI-2 external interface functions, grouped by function.

5.2.6 I/O

MPI-IO support is provided by including the ROMIO package from Argonne National Labs,[2] version 1.2.5.1. The LAM wrapper compilers (`mpicc`, `mpiCC`/`mpic++`, and `mpif77`) will automatically provide all the necessary flags to compile and link programs that use ROMIO function calls.

Although the ROMIO group at Argonne has included support for LAM in their package, there are still a small number of things that the LAM Team had to do to make ROMIO compile and install properly with LAM/MPI. As such, if you try to install the ROMIO package manually with LAM/MPI, you will experience some difficulties.

There are some important limitations to ROMIO that are discussed in the `romio/README` file. One limitation that is not currently listed in the ROMIO README file is that atomic file access will not work with AFS. This is because of file locking problems with AFS (i.e., AFS iteself does not support file locking). The ROMIO test program `atomicity` will fail if you specify an output file on AFS.

Additionally, ROMIO does not support the following LAM functionality:

- LAM MPI-2 datatypes cannot be used with ROMIO; ROMIO makes the fundamental assumption that MPI-2 datatypes are built upon MPI-1 datatypes. LAM builds MPI-2 datatypes natively – ROMIO cannot presently handle this case.

  This will hopefully be fixed in some future release of ROMIO. The ROMIO test programs `coll_test`, `fcoll_test`, `large_array`, and `coll_perf` will fail because they use the MPI-2 datatype MPI_DARRAY.

Please see the sections “ROMIO Users Mailing List” and “Reporting Bugs” in `romio/README` for how to submit questions and bug reports about ROMIO (that do not specifically pertain to LAM).

[2] http://www.mcs.anl.gov/romio/

5.2.7 Language Bindings

LAM provides C, C++, and Fortran 77 bindings for all supported MPI-2 functions, types, and constants. LAM does not provide a Fortran 90 module. However, it is possible to use the Fortran 77 bindings with a Fortran 90 compiler by specifying the F90 compiler as your Fortran compiler when configuring/compiling LAM/MPI. See the LAM Installation Guide [14] for more details.

The C++ bindings include support for the C++ only MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and MPI::LONG_DOUBLE_COMPLEX datatypes.

Note that there are some issues with using MPI and Fortran 90 together. See the F90 / C++ chapter in the MPI-2 standard [2] for more information on using MPI with Fortran 90.

As mentioned in Section 5.1.1, profiling support is available in all three languages (if LAM was compiled with profiling support). The `laminfo` command can be used to see if profiling support was included in LAM/MPI.

Chapter 6

System Services Interface (SSI) Overview

The System Services Interface (SSI) makes up the core of LAM/MPI. It influences how many commands and MPI processes are executed. This chapter provides an overview of what SSI is and what users need to know about how to use it to maximize performance of MPI applications.

6.1 Types and Modules

SSI provides a component framework for the LAM run-time environment (RTE) and the MPI communications layer. Components are selected from each type at run-time and used to effect the LAM RTE and MPI library.

There are currently four types of components used by LAM/MPI:

- boot: Starting the LAM run-time environment, used mainly with the `lamboot` command.
- coll: MPI collective communications, only used within MPI processes.
- cr: Checkpoint/restart functionality, used both within LAM commands and MPI processes.
- rpi: MPI point-to-point communications, only used within MPI processes.

The LAM/MPI distribution includes instances of each component type referred to as modules. Each module is an implementation of the component type which can be selected and used at run-time to provide services to the LAM RTE and MPI communications layer. Chapters 8 and 9 list the modules that are available in the LAM/MPI distribution.

6.2 Terminology

Available The term "available" is used to describe a module that reports (at run-time) that it is able to run in the current environment. For example, an RPI module may check to see if supporting network hardware is present before reporting that it is available or not.

Chapters 8 and 9 list the modules that are included in the LAM/MPI distribution, and detail the requirements for each of them to indicate whether they are available or not.

Selected The term "selected" means that a module has been chosen to be used at run-time. Depending on the module type, zero or more modules may be selected.

Scope Each module selection has a scope depending on the type of the module. "Scope" refers to the duration of the module's selection. Table 6.1 lists the scopes for each module type.

Type	Scope description
boot	A module is selected at the beginning of `lamboot` (or `recon`) and is used for the duration of the LAM universe.
coll	A module is selected every time an MPI communicator is created (including MPI_COMM_WORLD and MPI_COMM_SELF). It remains in use until that communicator has been freed.
cr	Checkpoint/restart modules are selected at the beginning of an MPI job and remain in use until the job completes.
rpi	RPI modules are selected during MPI_INIT and remain in use until MPI_FINALIZE returns.

Table 6.1: SSI module types and their corresponding scopes.

6.3 SSI Parameters

One of the founding principles of SSI is to allow the passing of run-time parameters through the SSI framework. This allows both the selection of which modules will be used at run-time (by passing parameters to the SSI framework itself) as well as tuning run-time performance of individual modules (by passing parameters to each module). Although the specific usage of each SSI module parameter is defined by either the framework or the module that it is passed to, the value of most parameters will be resolved by the following:

1. If a valid value is provided via a run-time SSI parameter, use that.
2. Otherwise, attempt to calculate a meaningful value at run-time or use a compiled-in default value.[1]

As such, it is typically possible to set a parameter's default value when LAM is configured/compiled, but use a different value at run time.

6.3.1 Naming Conventions

SSI parameter names are generally strings containing only letters and underscores, and can typically be broken down into three parts. For example, the parameter `boot_rsh_agent` can be broken into its three components:

- SSI module type: The first string of the name. In this case, it is `boot`.
- SSI module name: The second string of the name, corresponding to a specific SSI module. In this case, it is `rsh`.
- Parameter name: The last string in the name. It may be an arbitrary string, and include multiple underscores. In this case, it is `agent`.

[1] Note that many SSI modules provide configure flags to set compile-time defaults for "tweakable" parameters. See [14].

Although the parameter name is technically only the last part of the string, it is only proper to refer to it within its overall context. Hence, it is correct to say "the `boot_rsh_agent` parameter" as well as "the `agent` parameter to the `rsh` boot module".

Note that the reserved string `base` may appear as a module name, referring to the fact that the parameter applies to all modules of a give type.

6.3.2 Setting Parameter Values

SSI parameters each have a unique name and can take a single string value. The parameter/value pairs can be passed by multiple different mechanisms. Depending on the target module and the specific parameter, mechanisms may include:

- Using command line flags when LAM was configured.
- Setting environment variables before invoking LAM commands.
- Using the `-ssi` command line switch to various LAM commands.
- Setting attributes on MPI communicators.

Users are most likely to utilize the latter three methods. Each is described in detail, below. Listings and explanations of available SSI parameters are provided in Chapters 8 and 9 (pages 65 and 75, respectively), categorized by SSI type and module.

Environment Variables

SSI parameters can be passed via environment variables prefixed with `LAM_MPI_SSI`. For example, selecting which RPI module to use in an MPI job can be accomplished by setting the environment variable `LAM_MPI_SSI_rpi` to a valid RPI module name (e.g., `tcp`).

Note that environment variables must be set *before* invoking the corresponding LAM/MPI commands that will use them.

`-ssi` Command Line Switch

LAM/MPI commands that interact with SSI modules accept the `-ssi` command line switch. This switch expects two parameters to follow: the name of the SSI parameter and its corresponding value. For example:

```
shell$ mpirun C -ssi rpi tcp my_mpi_program
```

runs the `my_mpi_program` on all available CPUs in the LAM universe using the `tcp` RPI module.

Communicator Attributes

Some SSI types accept SSI parameters via MPI communicator attributes (notably the MPI collective communication modules). These parameters follow the same rules and restrictions as normal MPI attributes. Note that for portability between 32 and 64 bit systems, care should be taken when setting and getting attribute values. The following is an example of portable attribute C code:

```
int flag, attribute_val;
void *set_attribute;
void **get_attribute;
MPI_Comm comm = MPI_COMM_WORLD;
int keyval = LAM_MPI_SSI_COLL_BASE_ASSOCIATIVE;

/* Set the value */
set_attribute = (void *) 1;
MPI_Comm_set_attr(comm, keyval, &set_attribute);

/* Get the value */
get_attribute = NULL;
MPI_Comm_get_attr(comm, keyval, &get_attribute, &flag);
if (flag == 1) {
  attribute_val = (int) *get_attribute;
  printf(''Got the attribute value: %d\n'', attribute_val);
}
```

Specifically, the following code is neither correct nor portable:

```
int flag, attribute_val;
MPI_Comm comm = MPI_COMM_WORLD;
int keyval = LAM_MPI_SSI_COLL_BASE_ASSOCIATIVE;

/* Set the value */
attribute_val = 1;
MPI_Comm_set_attr(comm, keyval, &attribute_val);

/* Get the value */
attribute_val = -1;
MPI_Comm_get_attr(comm, keyval, &attribute_val, &flag);
if (flag == 1)
  printf(''Got the attribute value: %d\n'', attribute_val);
```

6.4 Dynamic Shared Object (DSO) Modules

⊤ (7.1)

LAM has the capability of building SSI modules statically as part of the MPI libraries or as dynamic shared objects (DSOs). DSOs are discovered and loaded into LAM processes at run-time. This allows adding (or removing) functionality from an existing LAM installation without the need to recompile or re-link user applications.

The default location for DSO SSI modules is `$prefix/lib/lam`. If otherwise unspecified, this is where LAM will look for DSO SSI modules. However, the SSI parameter `base_module_path` can be used to specify a new colon-delimited path to look for DSO SSI modules. This allows users to specify their own location for modules, if desired.

Note that specifying this parameter overrides the default location. If users wish to augment their search path, they will need to include the default location in the path specification.

```
shell$ mpirun C -ssi base_module_path $prefix/lib/lam:$HOME/my_lam_modules ...
```
⊥ (7.1)

6.5 Selecting Modules

As implied by the previous sections, modules are selected at run-time either by examining (in order) user-specified parameters, run-time calculations, and compiled-in defaults. The selection process involves a flexible negotitation phase which can be both tweaked and arbitrarily overriden by the user and system administrator.

6.5.1 Specifying Modules

Each SSI type has an implicit SSI parameter corresponding to the type name indicating which module(s) to be considered for selection. For example, to specify in that the tcp RPI module should be used, the SSI parameter `rpi` should be set to the value `tcp`. For example:

```
shell$ mpirun C -ssi rpi tcp my_mpi_program
```

The same is true for the other SSI types (boot, cr, and coll), with the exception that the coll type can be used to specify a comma-separated list of modules to be considered as each MPI communicator is created (including MPI_COMM_WORLD). For example:

```
shell$ mpirun C -ssi coll smp,shmem,lam_basic my_mpi_program
```

indicates that the smp and lam_basic modules will potentially both be considered for selection for each MPI communicator.

6.5.2 Setting Priorities

Although typically not useful to individual users, system administrators may use priorities to set system-wide defaults that influence the module selection process in LAM/MPI jobs.

Each module has an associated priority which plays role in whether a module is selected or not. Specifically, if one or more modules of a given type are available for selection, the modules' priorities will be at least one of the factors used to determine which module will finally be selected. Priorities are in the range $[-1, 100]$, with -1 indicating that the module should not be considered for selection, and 100 being the highest priority. Ties will be broken arbitrarily by the SSI framework.

A module's priorty can be set run-time through the normal SSI parameter mechanisms (i.e., environment variables or using the `-ssi` parameter). Every module has an implicit priority SSI parameter in the form `<type>_<module name>_priority`.

For example, a system administrator may set environment variables in system-wide shell setup files (e.g., `/etc/profile`, `/etc/bashrc`, or `/etc/csh.cshrc`) to change the default priorities.

6.5.3 Selection Algorithm

For each component type, the following general selection algorithm is used:

- A list of all available modules is created. If the user specified one or more modules for this type, only those modules are queried to see if they are available. Otherwise, all modules are queried.
- The module with the highest priority (and potentially meeting other selection criteria, depending on the module's type) will be selected.

Each SSI type may define its own additional selection rules. For example, the selection of coll, cr, and rpi modules may be inter-dependant, and depend on the supported MPI thread level. Chapter 9 (page 75) details the selection algorithm for MPI SSI modules.

Chapter 7

LAM/MPI Command Quick Reference

This section is intended to provide a quick reference of the major LAM/MPI commands. Each command also has its own manual page which typically provides more detail than this document.

7.1 The `lamboot` Command

The `lamboot` command is used to start the LAM run-time environment (RTE). `lamboot` is typically the first command used before any other LAM/MPI command (notable exceptions are the wrapper compilers, which do not require the LAM RTE, and `mpiexec` which can launch its own LAM universe). `lamboot` can use any of the available boot SSI modules; Section 8.1 details the requirements and operations of each of the boot SSI modules that are included in the LAM/MPI distribution.

Common arguments that are used with the `lamboot` command are:

- `-b`: When used with the rsh boot module, the "fast" boot algorithm is used which can noticeably speed up the execution time of `lamboot`. It can also be used where remote shell agents cannot provide output from remote nodes (e.g., in a Condor environment). Specifically, the "fast" algorithm assumes that the user's shell on the remote node is the same as the shell on the node where `lamboot` was invoked.

- `-d`: Print debugging output. This will print a *lot* of output, and is typically only necessary if `lamboot` fails for an unknown reason. The output is forwarded to standard out as well as either `/tmp` or syslog facilities. The amount of data produced can fill these filesystems, leading to general system problems.

- `-l`: Use local hostname resolution instead of centralized lookups. This is useful in environments where the same hostname may resolve to different IP addresses on different nodes (e.g., clusters based on Finite Neighborhood Networks[1]). ⊤ (7.1)

- `-prefix` <lam/install/path>: Use the LAM/MPI installation specified in the <lam/install/path> - where <lam/install/path> is the top level directory where LAM/MPI is installed. This is typically used when a user has multiple LAM/MPI installations and want to switch between them without changing the dot files or PATH environment variable.

  This option is not compatible with LAM/MPI versions prior to 7.1. ⊥ (7.1)

[1] See http://www.aggregate.org/ for more details.

- `-s`: Close the `stdout` and `stderr` of the locally-launched LAM daemon (they are normally left open). This is necessary when invoking `lamboot` via a remote agent such as `rsh` or `ssh`.

- `-v`: Print verbose output. This is useful to show progress during `lamboot`'s progress. Unlike `-d`, `-v` does not forward output to a file or syslog.

- `-x`: Run the LAM RTE in fault-tolerant mode.

- <`filename`>: The name of the boot schema file. Boot schemas, while they can be as simple as a list of hostnames, can contain additional information and are discussed in detail in Sections 4.4.1 and 8.1.1, pages 26 and 65, respectively.

Booting the LAM RTE is where most users (particularly first-time users) encounter problems. Each boot module has its own specific requirements and prerequisites for success. Although `lamboot` typically prints detailed messages when errors occur, users are strongly encouraged to read Section 8.1 for the details of the boot module that they will be using. Additionally, the `-d` switch should be used to examine exactly what is happening to determine the actual source of the problem – many problems with `lamboot` come from the operating system or the user's shell setup; not from within LAM itself.

The most common `lamboot` example simply uses a hostfile to launch across an `rsh`/`ssh`-based cluster of nodes (the "`-ssi boot rsh`" is not technically necessary here, but it is specified to make this example correct in all environments):

```
shell$ lamboot -v -ssi boot rsh hostfile

LAM 7.0/MPI 2 C++/ROMIO - Indiana University

n0<1234> ssi:boot:base:linear: booting n0 (node1.cluster.example.com)
n0<1234> ssi:boot:base:linear: booting n1 (node2.cluster.example.com)
n0<1234> ssi:boot:base:linear: booting n2 (node3.cluster.example.com)
n0<1234> ssi:boot:base:linear: booting n3 (node4.cluster.example.com)
n0<1234> ssi:boot:base:linear: finished
```

7.1.1 Multiple Sessions on the Same Node

In some cases (such as in batch-regulated environments), it is desirable to allow multiple universes owned by the same on the same node. The `TMPDIR`, `LAM_MPI_SESSION_PREFIX`, and `LAM_MPI_SESSION_SUFFIX` environment variables can be used to effect this behavior. The main issue is the location of LAM's session directory; each node in a LAM universe has a session directory in a well-known location in the filesystem that identifies how to contact the LAM daemon on that node. Multiple LAM universes can simultaneously co-exist on the same node as long as they have different session directories.

LAM recognizes several batch environments and automatically adapts the session directory to be specific to a batch job. Hence, if the batch scheduler allocates multiple jobs from the same user to the same node, LAM will automatically do the "right thing" and ensure that the LAM universes from each job will not collide. Sections 12.7 and 12.8 (starting on page 119) discuss these issues in detail.

7.1.2 Avoiding Running on Specific Nodes

Once the LAM universe is booted, processes can be launched on any node. The `mpirun`, `mpiexec`, and `lamexec` commands are most commonly used to launch jobs in the universe, and are typically used with the `N` and `C` nomenclatures (see the description of `mpirun` in Section 7.14 for details on the `N` and `C` nomenclature) which launch jobs on all schedulable nodes and CPUs in the LAM universe, respectively. While finer-grained controls are available through `mpirun` (etc.), it can be convenient to simply mark some nodes as "non-schedulable," and therefore avoid having `mpirun` (etc.) launch executables on those nodes when using `N` and `C` nomenclature.

For example, it may be convenient to boot a LAM universe that includes a controller node (e.g., a desktop workstation) and a set of worker nodes. In this case, it is desirable to mark the desktop workstation as "non-scheduable" so that LAM will not launch executables there (by default). Consider the following boot schema:

```
# Mark my_workstation as ''non−schedulable''
my_workstation.office.example.com schedule=no
# All the other nodes are, by default, schedulable
node1.cluster.example.com
node2.cluster.example.com
node3.cluster.example.com
node4.cluster.example.com
```

Booting with this schema allows the convenienve of:

```
shell$ mpirun C my_mpi_program
```

which will only run `my_mpi_program` on the four cluster nodes (i.e., not the workstation). Note that this behavior *only* applies to the `C` and `N` designations; LAM will always allow execution on any node when using the `nX` or `cX` notation:

```
shell$ mpirun c0 C my_mpi_program
```

which will run `my_mpi_program` on all five nodes in the LAM universe.

7.2 The `lamcheckpoint` Command

⊤ (7.1)

The `lamcheckpoint` command is provided to checkpoint a MPI application. One of the arguments to `lamcheckpoint` is the name of the checkpoint/restart module (which can be either one of blcr and self). Additional arguments to `lamcheckpoint` depend of the selected checkpoint/restart module. The name of the module can be specified by passing the cr SSI parameter.

Common arguments that are used with the `lamcheckpoint` command are:

- `-ssi`: Just like with `mpirun`, the `-ssi` flag can be used to pass key=value pairs to LAM. Indeed, it is required to pass at least one SSI parameter: `cr`, indicating which cr module to use for checkpointing.
- `-pid`: Indicate the PID of `mpirun` to checkpoint.

Notes:

- If the blcr cr module is selected, the name of the directory for storing the checkpoint files and the PID of `mpirun` should be passed as SSI parameters to `lamcheckpoint`.
- If the self cr module is selected, the PID of `mpirun` should be passed via the `-pid` parameter.

⊥ (7.1)

See Section 9.5 for more detail about the checkpoint/restart capabilities of LAM/MPI, including details about the blcr and self cr modules.

7.3 The `lamclean` Command

The `lamclean` command is provided to clean up the LAM universe. It is typically only necessary when MPI processes terminate "badly," and potentially leave resources allocated in the LAM universe (such as MPI-2 published names, processes, or shared memory). The `lamclean` command will kill *all* processes running in the LAM universe, and free *all* resources that were associated with them (including unpublishing MPI-2 dynamicly published names).

7.4 The `lamexec` Command

The `lamexec` command is similar to `mpirun` but is used for non-MPI programs. For example:

```
shell$ lamexec N uptime
  5:37pm up 21 days, 23:49, 5 users, load average: 0.31, 0.26, 0.25
  5:37pm up 21 days, 23:49, 2 users, load average: 0.01, 0.00, 0.00
  5:37pm up 21 days, 23:50, 3 users, load average: 0.01, 0.00, 0.00
  5:37pm up 21 days, 23:50, 2 users, load average: 0.87, 0.81, 0.80
```

Most of the parameters and options that are available to `mpirun` are also available to `lamexec`. See the `mpirun` description in Section 7.14 for more details.

7.5 The `lamgrow` Command

The `lamgrow` command adds a single node to the LAM universe. It must use the same boot module that was used to initially boot the LAM universe. `lamgrow` must be run from a node already in the LAM universe. Common parameters include:

- `-v`: Verbose mode.
- `-d`: Debug mode; enables a *lot* of diagnostic output.
- `-n` <`nodeid`>: Assign the new host the node ID `nodeid`. `nodeid` must be an unused node ID. If `-n` is not specified, LAM will find the lowest node ID that is not being used.
- `-no-schedule`: Has the same effect as putting "`no_schedule=yes`" in the boot schema. This means that the `C` and `N` expansion used in `mpirun` and `lamexec` will not include this node.
- `-ssi` <`key`> <`value`>: Pass in SSI parameter `key` with the value `value`.
- <`hostname`>: The name of the host to expand the universe to.

For example, the following adds the node `blinky` to the existing LAM universe using the `rsh` boot module:

```
shell$ lamgrow −ssi boot rsh blinky.cluster.example.com
```

Note that `lamgrow` cannot grow a LAM universe that only contains one node that has an IP address of 127.0.0.1 (e.g., if `lamboot` was run with the default boot schema that only contains the name `localhost`). In this case, `lamgrow` will print an error and abort without adding the new node.

7.6 The `lamhalt` Command

The `lamhalt` command is used to shut down the LAM RTE. Typically, `lamhalt` can simply be run with no command line parameters and it will shut down the LAM RTE. Optionally, the `-v` or `-d` arguments can be used to make `lamhalt` be verbose or extremely verbose, respectively.

There are a small number of cases where `lamhalt` will fail. For example, if a LAM daemon becomes unresponsive (e.g., the daemon was killed), `lamhalt` may fail to shut down the entire LAM universe. It will eventually timeout and therefore complete in finite time, but you may want to use the last-resort `lamwipe` command (see Section 7.18).

7.7 The `laminfo` Command

The `laminfo` command can be used to query the capabilities of the LAM/MPI installation. Running `laminfo` with no parameters shows a prettyprint summary of information. Using the `-parsable` command line switch shows the same summary information, but in a format that should be relatively easy to parse with common unix tools such as `grep`, `cut`, `awk`, etc.

`laminfo` supports a variety of command line options to query for specific information. The `-h` option shows a complete listing of all options. Some of the most common options include:

- `-arch`: Show the architecture that LAM was configured for.
- `-path`: Paired with a second argument, display various paths relevant to the LAM/MPI installation. Valid second arguments include:
  - `prefix`: Main installation prefix
  - `bindir`: Where the LAM/MPI executables are located
  - `libdir`: Where the LAM/MPI libraries are located
  - `incdir`: Where the LAM/MPI include files are located
  - `pkglibdir`: Where dynamic SSI modules are installed[2]
  - `sysconfdir`: Where the LAM/MPI help files are located
- `-version`: Paired with two addition options, display the version of either LAM/MPI or one or more SSI modules. The first argument identifies what to report the version of, and can be any of the following:

[2] Dynamic SSI modules are not supported in LAM/MPI 7.0, but will be supported in future versions.

  - `lam`: Version of LAM/MPI
  - `boot`: Version of all boot modules
  - `boot:module`: Version of a specific boot module
  - `coll`: Version of all coll modules
  - `coll:module`: Version of a specific coll module
  - `cr`: Version of all cr modules
  - `cr:module`: Version of a specific cr module
  - `rpi`: Version of all rpi modules
  - `rpi:module`: Version of a specific rpi module

  The second argument specifies the scope of the version number to display – whether to show the entire version number string, or just one component of it:

  - `full`: Display the entire version number string
  - `major`: Display the major version number
  - `minor`: Display the minor version number
  - `release`: Display the release version number
  - `alpha`: Display the alpha version number
  - `beta`: Display the beta version number
  - `svn`: Display the SVN version number[3]

⊤ (7.1)

- `-param`: Paired with two additional arguments, display the SSI parameters for a given type and/or module. The first argument can be any of the valid SSI types or the special name "base," indicating the SSI framework itself. The second argument can be any valid module name.

  Additionally, either argument can be the wildcard "any" which will match any valid SSI type and/or module.

⊥ (7.1)

Multiple options can be combined to query several attributes at once:

```
shell$ laminfo -parsable -arch -version lam major -version rpi:tcp full -param rpi tcp
version:lam:7
ssi:boot:rsh:version:ssi:1.0
ssi:boot:rsh:version:api:1.0
ssi:boot:rsh:version:module:7.0
arch:i686-pc-linux-gnu
ssi:rpi:tcp:param:rpi_tcp_short:65536
ssi:rpi:tcp:param:rpi_tcp_sockbuf:-1
ssi:rpi:tcp:param:rpi_tcp_priority:20
```

[3] The value will either be 0 (not built from SVN), 1 (built from a Subverstion checkout) or a date encoded in the form YYYYMMDD (built from a nightly tarball on the given date)

Note that three version numbers are returned for the tcp module. The first (ssi) indicates the overall SSI version that the module conforms to, the second (api) indicates what version of the rpi API the module conforms to, and the last (module) indicates the version of the module itself.

Running laminfo with no arguments provides a wealth of information about your LAM/MPI installation (we ask for this output when reporting problems to the LAM/MPI general user's mailing list – see Section 11.1 on page 111). Most of the output fields are self-explanitory; two that are worth explaining are:

- Debug support: This indicates whether your LAM installation was configured with the --with-debug option. It is generally only used by the LAM Team for development and maintenance of LAM itself; it does *not* indicate whether user's MPI applications can be debugged (specifically: user's MPI applications can *always* be debugged, regardless of this setting). This option defaults to "no"; users are discouraged from using this option. See the Install Guide for more information about --with-debug.
- Purify clean: This indicates whether your LAM installation was configured with the --with-purify option. This option is necessary to prevent a number of false positives when using memory-checking debuggers such as Purify, Valgrind, and bcheck. It is off by default because it can cause slight performance degredation in MPI applications. See the Install Guide for more information about --with-purify.

7.8 The lamnodes Command

LAM was specifically designed to abstract away hostnames once lamboot has completed successfully. However, for various reasons (usually related to system-administration concerns, and/or for creating human-readable reports), it can be desirable to retrieve the hostnames of LAM nodes long after lamboot.

The command lamnodes can be used for this purpose. It accepts both the N and C syntax from mpirun, and will return the corresponding names of the specified nodes. For example:

```
shell$ lamnodes N
```

will return the node that each CPU is located on, the hostname of that node, the total number of CPUs on each, and any flags that are set on that node. Specific nodes can also be queried:

```
shell$ lamnodes n0,3
```

will return the node, hostname, number of CPUs, and flags on n0 and n3.

Command line arguments can be used to customize the output of lamnodes. These include:

- -c: Suppress printing CPU counts
- -i: Print IP addresses instead of IP names
- -n: Suppress printing LAM node IDs

7.9 The lamrestart Command

The lamrestart can be used to restart a previously-checkpointed MPI application. The arguments to lamrestart depend on the selected checkpoint/restart module. Regardless of the checkpoint/restart module used, invoking lamrestart results in a new mpirun being launched.

The SSI parameter `cr` must be used to specify which checkpoint/restart module should be used to restart the application. Currently, only two values are possible: `blcr` and `self`.

- If the blcr module is selected, the SSI parameter `cr_blcr_context_file` should be used to pass in the filename of the context file that was created during a pevious successful checkpoint. For example:

```
shell$ lamrestart -ssi cr blcr -ssi cr_blcr_context_file filename
```

- If the self module is selected, the SSI parameter `cr_restart_args` must be passed with the arguments to be passed to `mpirun` to restart the application. For example:

```
shell$ lamrestart -ssi cr self -ssi cr_restart_args "args to mpirun"
```

See Section 9.5 for more detail about the checkpoint/restart capabilities of LAM/MPI, including details about the blcr and self cr modules.

7.10 The `lamshrink` Command

The `lamshrink` command is used to remove a node from a LAM universe:

```
shell$ lamshrink n3
```

removes node n3 from the LAM universe. Note that all nodes with ID's greater than 3 will not have their ID's reduced by one – n3 simply becomes an empty slot in the LAM universe. `mpirun` and `lamexec` will still function correctly, even when used with `C` and `N` notation – they will simply skip the n3 since there is no longer an operational node in that slot.

Note that the `lamgrow` command can optionally be used to fill the empty slot with a new node.

7.11 The `mpicc`, `mpiCC` / `mpic++`, and `mpif77` Commands

Compiling MPI applications can be a complicated process because the list of compiler and linker flags required to successfully compile and link a LAM/MPI application not only can be quite long, it can change depending on the particular configuration that LAM was installed with. For example, if LAM includes native support for Myrinet hardware, the `-lgm` flag needs to be used when linking MPI executables.

To hide all this complexity, "wrapper" compilers are provided that handle all of this automatically. They are called "wrapper" compilers because all they do is add relevant compiler and linker flags to the command line before invoking the real back-end compiler to actually perform the compile/link. Most command line arugments are passed straight through to the back-end compiler without modification.

Therefore, to compile an MPI application, use the wrapper compilers exactly as you would use the real compiler. For example:

```
shell$ mpicc -O -c main.c
shell$ mpicc -O -c foo.c
shell$ mpicc -O -c bar.c
shell$ mpicc -O -o main main.o foo.o bar.o
```

This compiles three C source code files and links them together into a single executable. No additional `-I`, `-L`, or `-l` arguments are required.

The main exceptions to what flags are not passed through to the back-end compiler are:

- `-showme`: Used to show what the wrapper compiler would have executed. This is useful to see the full compile/link line would have been executed. For example (your output may differ from what is shown below, depending on your installed LAM/MPI configuration):

```
shell$ mpicc -O -c main.c -showme
gcc -I/usr/local/lam/include -pthread -O -c foo.c
```

```
# The output line shown below is word wrapped in order to fit nicely in the document margins
shell$ mpicc -O -o main main.o foo.o bar.o -showme
gcc -I/usr/local/lam/include -pthread -O -o main main.o foo.o bar.o \
-L/usr/local/lam/lib -llammpio -lpmpi -llamf77mpi -lmpi -llam -lutil \
-pthread
```

⊤ (7.1)

  Two notable sub-flags are:

  - `-showme:compile`: Show only the compile flags, suitable for substitution into `CFLAGS`.

```
shell$ mpicc -O -c main.c -showme:compile
-I/usr/local/lam/include -pthread
```

  - `-showme:link`: Show only the linker flags (which are actually `LDFLAGS` and `LIBS` mixed together), suitable for substitution into `LIBS`.

```
shell$ mpicc -O -o main main.o foo.o bar.o -showme:link
-L/usr/local/lam/lib -llammpio -lpmpi -llamf77mpi -lmpi -llam -lutil -pthread
```

⊥ (7.1)

- `-lpmpi`: When compiling a user MPI application, the `-lpmpi` argument is used to indicate that MPI profiling support should be included. The wrapper compiler may alter the exact placement of this argument to ensure that proper linker dependency semantics are preserved.

⊤ (7.1)

Neither the compiler nor linker flags can be overridden at run-time. The back-end compiler, however, can be. Environment variables can be used for this purpose:

- `LAMMPICC` (deprecated name: `LAMHCC`): Overrides the default C compiler in the `mpicc` wrapper compiler.
- `LAMMPICXX` (deprecated name: `LAMHCP`): Overrides the default C compiler in the `mpicc` wrapper compiler.
- `LAMMPIF77` (deprecated name: `LAMHF77`): Overrides the default C compiler in the `mpicc` wrapper compiler.

For example (for Bourne-like shells):

```
shell$ LAMPICC=cc
shell$ export LAMMPICC
shell$ mpicc my_application.c −o my_application
```

For csh-like shells:

```
shell% setenv LAMPICC cc
shell% mpicc my_application.c −o my_application
```

All this being said, it is *strongly* recommended to use the wrapper compilers – and their default underlying compilers – for all compiling and linking of MPI applications. Strange behavior can occur in MPI applications if LAM/MPI was configured and compiled with one compiler and then user applications were compiled with a different underlying compiler, to include: failure to compile, failure to link, seg faults and other random bad behavior at run-time.

Finally, note that the wrapper compilers only add all the LAM/MPI-specific flags when a command-line argument that does not begin with a dash ("-") is present. For example:

```
shell$ mpicc
gcc: no input files
shell$ mpicc −−version
gcc (GCC) 3.2.2 (Mandrake Linux 9.1 3.2.2−3mdk)
Copyright (C) 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

⊥ (7.1)

7.11.1 Deprecated Names

Previous versions of LAM/MPI used the names `hcc`, `hcp`, and `hf77` for the wrapper compilers. While these command names still work (they are simply symbolic links to the real wrapper compilers `mpicc`, `mpiCC`/`mpic++`, and `mpif77`, respectively), their use is deprecated.

7.12 The `mpiexec` Command

The `mpiexec` command is used to launch MPI programs. It is similar to, but slightly different than, `mpirun`.[4] Although `mpiexec` is simply a wrapper around other LAM commands (including `lamboot`, `mpirun`, and `lamhalt`), it ties their functionality together and provides a unified interface for launching MPI processes. Specifically, `mpiexec` offers two features from command line flags that require multiple steps when using other LAM commands: launching MPMD MPI processes and launching MPI processes when there is no existing LAM universe.

7.12.1 General Syntax

The general form of `mpiexec` commands is:

[4] The reason that there are two methods to launch MPI executables is because the MPI-2 standard suggests the use of `mpiexec` and provides standardized command line arguments. Hence, even though LAM already had an `mpirun` command to launch MPI executables, `mpiexec` was added to comply with the standard.

```
mpiexec [global_args] local_args1 [: local_args2 [...]]
```

Global arguments are applied to all MPI processes that are launched. They must be specified before any local arguments. Common global arguments include:

- `-boot`: Boot the LAM RTE before launching the MPI processes.
- `-boot-args <args>`: Pass `<args>` to the back-end `lamboot`. Implies `-boot`.
- `-machinefile <filename>`: Specify `<filename>` as the boot schema to use when invoking the back-end `lamboot`. Implies `-boot`. ⊤ (7.1)
- `-prefix <lam/install/path>`: Use the LAM/MPI installation specified in the <lam/install/path> - where <lam/install/path> is the top level directory where LAM/MPI is "installed". This is typically used when a user has multiple LAM/MPI installations and want to switch between them without changing the dot files or PATH environment variable. This option is not compatible with LAM/MPI versions prior to 7.1. ⊥ (7.1)
- `-ssi <key> <value>`: Pass the SSI `<key>` and `<value>` arguments to the back-end `mpirun` command.

Local arguments are specific to an individual MPI process that will be launched. They are specified along with the executable that will be launched. Common local arguments include:

- `-n <numprocs>`: Launch `<numprocs>` number of copies of this executable.
- `-arch <architecture>`: Launch the executable on nodes in the LAM universe that match this architecture. An architecture is determined to be a match if the `<architecture>` matches any subset of the GNU Autoconf architecture string on each of the target nodes (the `laminfo` command shows the GNU Autoconf configure string).
- `<other arguments>`: When `mpiexec` first encounters an argument that it doesn't recognize, the remainder of the arguments will be passed back to `mpirun` to actually start the process.

The following example launches four copies of the `my_mpi_program` executable in the LAM universe, using default scheduling patterns:

```
shell$ mpiexec -n 4 my_mpi_program
```

7.12.2 Launching MPMD Processes

The "`:`" separator can be used to launch multiple executables in the same MPI job. Specifically, each process will share a common MPI_COMM_WORLD. For example, the following launches a single `manager` process as well as a `worker` process for every CPU in the LAM universe:

```
shell$ mpiexec -n 1 manager : C worker
```

Paired with the `-arch` flag, this can be especially helpful in heterogeneous environments:

```
shell$ mpiexec −arch solaris sol_program : −arch linux linux_program
```

Even only "slightly heterogeneous" environments can run into problems with shared libraries, different compilers, etc. The `-arch` flag can be used to differentiate between different versions of the same operating system:

```
shell$ mpiexec −arch solaris2.8 sol2.8_program : −arch solaris2.9 sol2.9_program
```

7.12.3 Launching MPI Processes with No Established LAM Universe

The `-boot`, `-boot-args`, and `-machinefile` global arguments can be used to launch the LAM RTE, run the MPI process(es), and then take down the LAM RTE. This conveniently wraps up several LAM commands and provides "one-shot" execution of MPI processes. For example:

```
shell$ mpiexec −machinefile hostfile C my_mpi_program
```

Some boot SSI modules do not require a hostfile; specifying the `-boot` argument is sufficient in these cases:

```
shell$ mpiexec −boot C my_mpi_program
```

When `mpiexec` is used to boot the LAM RTE, it will do its best to take down the LAM RTE even if errors occur, either during the boot itself, or if an MPI process aborts (or the user hits Control-C).

7.13 The `mpimsg` Command (Deprecated)

The `mpimsg` command is deprecated. It is only useful in a small number of cases (specifically, when the lamd RPI module is used), and may disappear in future LAM/MPI releases.

7.14 The `mpirun` Command

The `mpirun` command is the main mechanism to launch MPI processes in parallel.

7.14.1 Simple Examples

Although `mpirun` supports many different modes of execution, most users will likely only need to use a few of its capabilities. It is common to launch either one process per node or one process per CPU in the LAM universe (CPU counts are established in the boot schema). The following two examples show these two cases:

```
# Launch one copy of my_mpi_program on every schedulable node in the LAM universe
shell$ mpirun N my_mpi_program
```

```
# Launch one copy of my_mpi_program on every schedulable CPU in the LAM universe
shell$ mpirun C my_mpi_program
```

The specific number of processes that are launched can be controlled with the `-np` switch:

```
# Launch four my_mpi_program processes
shell$ mpirun -np 4 my_mpi_program
```

The `-ssi` switch can be used to specify tunable parameters to MPI processes.

```
# Specify to use the usysv RPI module
shell$ mpirun -ssi rpi usysv C my_mpi_program
```

The available modules and their associated parameters are discussed in detail in Chapter 9.

Arbitrary user arguments can also be passed to the user program. `mpirun` will attempt to parse all options (looking for LAM options) until it finds a `--`. All arguments following `--` are directly passed to the MPI application.

```
# Pass three command line arguments to every instance of my_mpi_program
shell$ mpirun -ssi rpi usysv C my_mpi_program arg1 arg2 arg3
# Pass three command line arguments, escaped from parsing
shell$ mpirun -ssi rpi usysv C my_mpi_program -- arg1 arg2 arg3
```

7.14.2 Controlling Where Processes Are Launched

`mpirun` allows for fine-grained control of where to schedule launched processes. Note LAM uses the term "schedule" extensively to indicate which nodes processes are launched on. LAM does *not* influence operating system semantics for prioritizing processes or binding processes to specific CPUs. The boot schema file can be used to indicate how many CPUs are on a node, but this is only used for scheduling purposes. For a fuller description of CPU counts in boot schemas, see Sections 4.4.1 and 8.1.1 on pages 26 and 65, respectively.

LAM offers two main scheduling nomenclatures: by node and by CPU. For example `N` means "all schedulable nodes in the universe" ("schedulable" is defined in Section 7.1.2). Similarly, `C` means "all schedulable CPUs in the universe."

More fine-grained control is also possible – nodes and CPUs can be individually identified, or identified by ranges. The syntax for these concepts is `n`<`range`> and `c`<`range`>, respectively. <`range`> can specify one or more elements by listing integers separated by commas and dashes. For example:

- `n3`: The node with an ID of 3.
- `c2`: The CPU with an ID of 2.
- `n2,4`: The nodes with IDs of 2 and 4.
- `c2,4-7`: The CPUs with IDs of 2, 4, 5, 6, and 7. Note that some of these CPUs may be on the same node(s).

Integers can range from 0 to the highest numbered node/CPU. Note that these nomenclatures can be mixed and matched on the `mpirun` command line:

```
shell$ mpirun n0 C manager-worker
```

will launch the `manager-worker` program on `n0` as well as on every schedulable CPU in the universe (yes, this means that `n0` will likely be over-subscribed).

When running on SMP nodes, it is preferable to use the `C`/`c`<`range`> nomenclature (with appropriate CPU counts in the boot schema) to the `N`/`n`<`range`> nomenclature because of how LAM will order ranks in MPI_COMM_WORLD. For example, consider a LAM universe of two four-way SMPs – `n0` and `n1` both have a CPU count of 4. Using the following:

```
shell$ mpirun C my_mpi_program
```

will launch eight copies of `my_mpi_program`, four on each node. LAM will place as many adjoining MPI_COMM_WORLD ranks on the same node as possible: MPI_COMM_WORLD ranks 0-3 will be scheduled on `n0` and MPI_COMM_WORLD ranks 4-7 will be scheduled on `n1`. Specifically, `C` schedules processes starting with `c0` and incrementing the CPU index number.

Note that unless otherwise specified, LAM schedules processes by CPU (vs. scheduling by node). For example, using `mpirun`'s `-np` switch to specify an absolute number of processes schedules on a per-CPU basis.

7.14.3 Per-Process Controls

`mpirun` allows for arbitrary, per-process controls such as launching MPMD jobs, passing different command line arguments to different MPI_COMM_WORLD ranks, etc. This is accomplished by creating a text file called an application schema that lists, one per line, the location, relevant flags, user executable, and command line arguments for each process. For example (lines beginning with "#" are comments):

```
# Start the manager on c0 with a specific set of command line options
c0 manager manager_arg1 manager_arg2 manager_arg3
# Start the workers on all available CPUs with different arguments
C worker worker_arg1 worker_arg2 worker_arg3
```

Note that the `-ssi` switch is *not* permissible in application schema files; `-ssi` flags are considered to be global to the entire MPI job, not specified per-process. Application schemas are described in more detail in the `appschema(5)` manual page.

7.14.4 Ability to Pass Environment Variables

All environment variables with names that begin with `LAM_MPI_` are automatically passed to remote notes (unless disabled via the `-nx` option to `mpirun`). Additionally, the `-x` option enables exporting of specific environment variables to the remote nodes:

```
shell$ LAM_MPI_FOO=''green eggs and ham''
shell$ export LAM_MPI_FOO
shell$ mpirun C −x DISPLAY,SEUSS=author samIam
```

This will launch the `samIam` application on all available CPUs. The `LAM_MPI_FOO`, `DISPLAY`, and SEUSS environment variables will be created each the process environment before the `smaIam` program is invoked.

Note that the parser for the `-x` option is currently not very sophisticated – it cannot even handle quoted values when defining new environment variables. Users are advised to set variables in the environment

prior to invoking `mpirun`, and only use `-x` to export the variables to the remote nodes (not to define new variables), if possible.

7.14.5 Current Working Directory Behavior

Using the `-wd` option to `mpirun` allows specifying an arbitrary working directory for the launched processes. It can also be used in application schema files to specify working directories on specific nodes and/or for specific applications.

If the `-wd` option appears both in an application schema file and on the command line, the schema file directory will override the command line value. `-wd` is mutually exclusive with `-D`.

If neither `-wd` nor `-D` are specified, the local node will send the present working directory name from the `mpirun` process to each of the remote nodes. The remote nodes will then try to change to that directory. If they fail (e.g., if the directory does not exist on that node), they will start from the user's home directory.

All directory changing occurs before the user's program is invoked; it does not wait until MPI_INIT is called.

7.15 The `mpitask` Command

The `mpitask` command shows a list of the processes running in the LAM universe and a snapshot of their current MPI activity. It is usually invoked with no command line parameters, thereby showing summary details of all processes currently running. Since `mpitask` only provides a snapshot view, it is not advisable to use `mpitask` as a high-resolution debugger (see Chapter 10, page 103, for more details on debugging MPI programs). Instead, `mpitask` can be used to provide answers to high-level questions such as "Where is my program hung?" and "Is my program making progress?"

The following example shows an MPI program running on four nodes, sending a message of 524,288 integers around in a ring pattern. Process 0 is running (i.e., not in an MPI function), while the other three are blocked in MPI_RECV.

```
shell$ mpitask
TASK (G/L) FUNCTION PEER|ROOT TAG COMM COUNT DATATYPE
0 ring <running>
1/1 ring Recv 0/0 201 WORLD 524288 INT
2/2 ring Recv 1/1 201 WORLD 524288 INT
3/3 ring Recv 2/2 201 WORLD 524288 INT
```

7.16 The `recon` Command

The `recon` command is a quick test to see if the user's environment is setup properly to boot the LAM RTE. It takes most of the same parameters as the `lamboot` command.

Although it does not boot the RTE, and does not definitively guarantee that `lamboot` will succeed, it is a good tool for testing while setting up first-time LAM/MPI users. `recon` will display a message when it has completed indicating whether it succeeded or failed.

7.17 The `tping` Command

The `tping` command can be used to verify the functionality of a LAM universe. It is used to send a ping message between the LAM daemons that constitute the LAM RTE.

It commonly takes two arguments: the set of nodes to ping (expressed in `N` notation) and how many times to ping them. Similar to the Unix `ping` command, if the number of times to ping is not specified, `tping` will continue until it is stopped (usually by the user hitting Control-C). The following example pings all nodes in the LAM universe three times:

```
shell$ tping N -c 3
   1 byte from 3 remote nodes and 1 local node: 0.002 secs
   1 byte from 3 remote nodes and 1 local node: 0.001 secs
   1 byte from 3 remote nodes and 1 local node: 0.001 secs

3 messages, 3 bytes (0.003K), 0.005 secs (1.250K/sec)
roundtrip min/avg/max: 0.001/0.002/0.002
```

7.18 The `lamwipe` Command

⊤ (7.1)

The `lamwipe` command used to be called `wipe`. The name `wipe` has now been deprecated and although it still works in this version of LAM/MPI, will be removed in future versions. All users are encouraged to ⊥ (7.1) start using `lamwipe` instead.

The `lamwipe` command is used as a "last resort" command, and is typically only necessary if `lamhalt` fails. This usually only occurs in error conditions, such as if a node fails. The `lamwipe` command takes most of the same parameters as the `lamboot` command – it launches a process on each node in the boot schema to kill the LAM RTE on that node. Hence, it should be used with the same (or an equivalent) boot schema file as was used with `lamboot`.

Chapter 8

Available LAM Modules

There is currently only type of LAM module that is visible to users: boot, which is used to start the LAM run-time environment, most often through the `lamboot` command. The `lamboot` command itself is discussed in Section 7.1 (page 49); the discussion below focuses on the boot modules that make up the "back end" implementation of `lamboot`.

8.1 Booting the LAM Run-Time Environment

LAM provides a number of modules for starting the `lamd` control daemons. In most cases, the `lamd`s are started using the `lamboot` command. In previous versions of LAM/MPI, `lamboot` could only use `rsh` or `ssh` for starting the LAM run-time environment on remote nodes. In LAM/MPI 7.1.5b2, it is possible to use a variety of mechanisms for this process startup. The following mechanisms are available in LAM/MPI 7.1.5b2:

- BProc
- Globus (beta-level support)
- `rsh` / `ssh`
- OpenPBS / PBS Pro / Torque (using the Task Management interface) ⊤ (7.1)
- SLURM (using its native interface) ⊥ (7.1)

These mechanisms are discussed in detail below. Note that the sections below each assume that support for these modules have been compiled into LAM/MPI. The `laminfo` command can be used to determine exactly which modules are supported in your installation (see Section 7.7, page 53).

8.1.1 Boot Schema Files (a.k.a., "Hostfiles" or "Machinefiles")

Before discussing any of the specific boot SSI modules, this section discusses the boot schema file, commonly referred to as a "hostfile" or a "machinefile". Most (but not all) boot SSI modules require a boot schema, and the text below makes frequent mention of them. Hence, it is worth discussing them before getting into the details of each boot SSI.

A boot schema is a text file that, in its simplest form, simply lists every host that the LAM run-time environment will be invoked on. For example:

```
# This is my boot schema
inky.cluster.example.com
pinky.cluster.example.com
blinkly.cluster.example.com
clyde.cluster.example.com
```

Lines beginning with "#" are treated as comments and are ignored. Each non-blank, non-comment line must, at a minimum, list a host. Specifically, the first token on each line must specify a host (although the definition of how that host is specified may vary differ between boot modules).

However, each line can also specify arbitrary "key=value" pairs. A common global key is "`cpu`". This key takes an integer value and indicates to LAM how many CPUs are available for LAM to use. If the key is not present, the value of 1 is assumed. This number does *not* need to reflect the physical number of CPUs – it can be smaller then, equal to, or greater than the number of physical CPUs in the machine. It is solely used as a shorthand notation for `mpirun`'s "C" notation, meaning "launch one process per CPU as specified in the boot schema file." For example, in the following boot schema:

```
inky.cluster.example.com cpu=2
pinky.cluster.example.com cpu=4
blinkly.cluster.example.com cpu=4
# clyde doesn't mention a cpu count, and is therefore implicitly 1
clyde.cluster.example.com
```

issuing the command "`mpirun C foo`" would actually launch 11 copies of `foo`: 2 on `inky`, 4 on `pinky`, 4 on `blinky`, and 1 on `clyde`.

Note that listing a host more than once has the same effect as incrementing the CPU count. The following boot schema has the same effect as the previous example (i.e., CPU counts of 2, 4, 4, and 1, respectively):

```
# inky has a CPU count of 2
inky.cluster.example.com
inky.cluster.example.com
# pinky has a CPU count of 4
pinky.cluster.example.com
pinky.cluster.example.com
pinky.cluster.example.com
pinky.cluster.example.com
# blinky has a CPU count of 4
blinkly.cluster.example.com
blinkly.cluster.example.com
blinkly.cluster.example.com
blinkly.cluster.example.com
# clyde only has 1 CPU
clyde.cluster.example.com
```

Other keys are defined on a per-boot-SSI-module, and are described below.

8.1.2 Minimum Requirements

In order to successfully launch a process on a remote node, several requirements must be met. Although each of the boot modules have different specific requirements, all of them share the following conditions for successful operation:

1. Each target host must be reachable and operational.
2. The user must be able to execute arbitrary processes on the target.
3. The LAM executables must be locatable on that machine. This typically involves using: the shell's search path, the `LAMHOME` environment variable, or a boot-module-specific mechanism.
4. The user must be able to write to the LAM session directory (typically somewhere under `/tmp`; see Section 12.8, page 119).
5. All hosts must be able to resolve the fully-qualified domain name (FQDN) of all the machines being booted (including itself).
6. Unless there is only one host being booted, any host resolving to the IP address 127.0.0.1 cannot be included in the list of hosts.

If all of these conditions are not met, `lamboot` will fail.

8.1.3 Selecting a boot Module

Only one boot module will be selected; it will be used for the life of the LAM universe. As such, module priority values are the only factor used to determine which available module should be selected.

8.1.4 boot SSI Parameters

On many kinds of networks, LAM can know exactly which nodes should be making connections while booting the LAM run-time environment, and promiscuous connections (i.e., allowing any node to connect) are discouraged. However, this is not possible in some complex network configurations and promiscuous connections *must* be enabled.

By default, LAM's base boot SSI startup protocols disable promiscuous connections. However, this behavior can be overridden when LAM is configured and at run-time. If the SSI parameter `boot_base_promisc` set to an empty value, or set to the integer value 1, promiscuous connections will be accepted when than LAM RTE is booted.

8.1.5 The bproc Module

The Beowulf Distributed Process Space (BProc) project[1] is set of kernel modifications, utilities and libraries which allow a user to start processes on other machines in a Beowulf-style cluster. Remote processes started with this mechanism appear in the process table of the front-end machine in a cluster.

LAM/MPI functionality has been tested with BProc version 3.2.5. Prior versions had a bug that affected at least some LAM/MPI functionality. It is strongly recommended to upgrade to at least version 3.2.5 before attempting to use the LAM/MPI native BProc capabilities.

[1] `http://bproc.sourceforge.net/`

Minimum Requirements

Several of the minimum requirements listed in Section 8.1.2 will already be met in a BProc environment because BProc will copy `lamboot`'s entire environment (including the `PATH`) to the remote node. Hence, if `lamboot` is in the user's path on the local node, it will also [automatically] be in the user's path on the remote node.

However, one of the minimum requirements conditions ("The user must be able to execute arbitrary processes on the target") deserves a BProc-specific clarification. BProc has its own internal permission system for determining if users are allowed to execute on specific nodes. The system is similar to the user/-group/other mechanism typically used in many Unix filesystems. Hence, in order for a user to successfully `lamboot` on a BProc cluster, he/she must have BProc execute permissions on each of the target nodes. Consult the BProc documentation for more details.

Usage

In most situations, the `lamboot` command (and related commands) should automatically "know" to use the bproc boot SSI module when running on the BProc head node; no additional command line parameters or environment variables should be required. Specifically, when running in a BProc environment, the bproc module will report that it is available, and artificially inflate its priority relatively high in order to influence the boot module selection process. However, the BProc boot module can be forced by specifying the `boot` SSI parameter with the value of `bproc`.

Running `lamboot` on a BProc cluster is just like running `lamboot` in a "normal" cluster. Specifically, you provide a boot schema file (i.e., a list of nodes to boot on) and run `lamboot` with it. For example:

```
shell$ lamboot hostfile
```

Note that when using the bproc module, `lamboot` will only function properly from the head node. If you launch `lamboot` from a client node, it will likely either fail outright, or fall back to a different boot module (e.g., `rsh`/`ssh`).

It is suggested that the `hostfile` file contain hostnames in the style that BProc prefers – integer numbers. For example, `hostfile` may contain the following:

```
-1
 0
 1
 2
 3
```

which boots on the BProc front end node (-1) and four slave nodes (0, 1, 2, 3). Note that using IP hostnames will also work, but using integer numbers is recommended.

Tunable Parameters

Table 8.1 lists the SSI parameters that are available to the bproc module.

Special Notes

After booting, LAM will, by default, not schedule to run MPI jobs on the BProc front end. Specifically, LAM implicitly sets the "no-schedule" attribute on the -1 node in a BProc cluster. See Section 7.1 (page 49)

SSI parameter name	Default value	Description
boot_bproc_priority	50	Default priority level.

Table 8.1: SSI parameters for the bproc boot module.

for more detail about this attribute and boot schemas in general, and 7.1.2 (page 51).

8.1.6 The globus Module

LAM/MPI 7.1.5b2 includes beta support for Globus. Specifically, only limited types of execution are possible. The LAM Team would appreciate feedback from the Globus community on expanding Globus support in LAM/MPI.

Minimum Requirements

LAM/MPI jobs in Globus environment can only be started on nodes using the "fork" job manager for the Globus gatekeeper. Other job managers are not yet supported.

Usage

Starting the LAM run-time environmetn in Globus environment makes use of the Globus Resource Allocation Manager (GRAM) client `globus-job-run`. The Globus boot SSI module will never run automatically; it must always be specifically requested setting the `boot` SSI parameter to `globus`. Specifically, although the globus module will report itself available if `globus-job-run` can be found in the `PATH`, the default priority will be quite low, effectively ensuring that it will not be selected unless it is the only module available (which will only occur if the `boot` parameter is set to `globus`).

LAM needs to be able to find the Globus executables. This can be accompilshed either by adding the appropriate directory to your path, or by setting the `GLOBUS_LOCATION` environment variable.

Additionally, the `LAM_MPI_SESSION_SUFFIX` environment variable should be set to a unique value. This ensures that this instance of the LAM universe does not conflict with any other, concurrent LAM universes that are running under the same username on nodes in the Globus environment. Although any value can be used for this variable, it is probably best to have some kind of organized format, such as `<your_username>-<some_long_random_number>`.

Next, create a boot schema to use with `lamboot`. Hosts are listed by their Globus contact strings (see the Globus manual for more information about contact strings). In cases where the Globus gatekeeper is running as a `inetd` service on the node, the contact string will simply be the hostname. If the contact string contains whitespace, the *entire* contact string must be enclosed in quotes (i.e., not just the values with whitespaces). For example, if your contact string is:

```
host1:port1:/O=xxx/OU=yyy/CN=aaa bbb ccc
```

Then you will need to have it listed as:

```
"host1:port1:/O=xxx/OU=yyy/CN=aaa bbb ccc"
```

The following will not work:

```
host1:port1:/O=xxx/OU=yyy/CN="aaa bbb ccc"
```

Each host in the boot schema must also have a "`lam_install_path`" key indicating the absolute directory where LAM/MPI is installed. This value is mandatory because you cannot rely on the `PATH`

environment variable in Globus environment because users' "dot" files are not executed in Globus jobs (and therefore the `PATH` environment variable is not provided). Other keys can be used as well; `lam_install_path` is the only mandatory key.

⊤ (7.0.5) Here is a sample Globus boot schema:

```
# Globus boot schema
''inky.mycluster:12853:/O=MegaCorp/OU=Mine/CN=HPC Group'' prefix=/opt/lam cpu=2
''pinky.yourcluster:3245:/O=MegaCorp/OU=Yours/CN=HPC Group'' prefix=/opt/lam cpu=4
''blinky.hiscluster:23452:/O=MegaCorp/OU=His/CN=HPC Group'' prefix=/opt/lam cpu=4
''clyde.hercluster:82342:/O=MegaCorp/OU=Hers/CN=HPC Group'' prefix=/software/lam
```

⊥ (7.0.5)

Once you have this boot schema, the `lamboot` command can be used to launch it. Note, however, that unlike the other boot SSI modules, the Globus boot module will never be automatically selected by LAM – it must be selected manually with the `boot` SSI parameter with the value `globus`.

```
shell$ lamboot -ssi boot globus hostfile
```

Tunable Parameters

Table 8.2 lists the SSI parameters that are available to the globus module.

SSI parameter name	Default value	Description
`boot_globus_priority`	3	Default priority level.

Table 8.2: SSI parameters for the globus boot module.

8.1.7 The rsh Module (including ssh)

The `rsh`/`ssh` boot SSI module is typically the "least common denominator" boot module. When not in an otherwise "special" environment (such as a batch scheduler), the `rsh`/`ssh` boot module is typically used to start the LAM run-time environment.

Minimum Requirements

In addition to the minimum requirements listed in Section 8.1.2, the following additional conditions must also be met for a successful `lamboot` using the `rsh` / `ssh` boot module:

1. The user must be able to execute arbitrary commands on each target host without being prompted for a password.
2. The shell's start-up script must not print anything on standard error. The user can take advantage of the fact that `rsh` / `ssh` will start the shell non-interactively. The start-up script can exit early in this case, before executing many commands relevant only to interactive sessions and likely to generate output.

   ⊤ (7.1) This has now been changed in version 7.1; if the SSI parameter `boot_rsh_ignore_stderr` is nonzero, any output on standard error will *not* be treated as an error. ⊥ (7.1)

Section 4 (page 23) provides a short tutorial on using the `rsh` / `ssh` boot module, including tips on setting up "dot" files, setting up password-less remote execution, etc.

Usage

Using `rsh`, `ssh`, or other remote-execution agent is probably the most common method for starting the LAM run-time execution environment. The boot schema typically lists the hostnames, CPU counts, and an optional username (if the user's name is different on the remote machine). ⊤ (7.1)

The boot schema can also list an optional "prefix", which specifies the LAM/MPI installatation to be used on the particular host listed in the boot schema. This is typically used if the user has mutliple LAM/MPI installations on a host and want to switch between them without changing the dot files or `PATH` environment variables, or if the user has LAM/MPI installed under different paths on different hosts. If the prefix is not specified for a host in the boot schema file, then the LAM/MPI installation which is available in the `PATH` will be used on that host, or if the `-prefix` </lam/install/path> option is specified for `lamboot`, the </lam/install/path> installation will be used. The prefix option in the boot schema file however overrides any prefix option specified on the `lamboot` command line for that host.

For example:

```
# rsh boot schema
inky.cluster.example.com cpu=2
pinky.cluster.example.com cpu=4 prefix=/home/joe/lam7.1/install/
blinky.cluster.example.com cpu=4
clyde.cluster.example.com user=jsmith
```

⊥ (7.1)

The `rsh` / `ssh` boot module will usually run when no other boot module has been selected. It can, however, be manually selected, even when another module would typically [automatically] be selected by specifying the `boot` SSI parameter with the value of `rsh`. For example:

```
shell$ lamboot -ssi boot rsh hostfile
```

Tunable Parameters

⊤ (7.1)

Table 8.3 lists the SSI parameters that are available to the rsh module. ⊥ (7.1)

8.1.8 The slurm Module

⊤ (7.1)

As its name implies, the Simple Linux Utility for Resource Management (SLURM)[2] package is commonly used for managing Linux clusters, typically in high-performance computing environments. SLURM contains a native system for launching applications across the nodes that it manages. When using SLURM, `rsh`/`ssh` is not necessary to launch jobs on remote nodes. Instead, the slurm boot module will automatically use SLURM's native job-launching interface to start LAM daemons.

The advantages of using SLURM's native interface are:

- SLURM can generate proper accounting information for all nodes in a parallel job.
- SLURM can kill entire jobs properly when the job ends.
- `lamboot` executes significantly faster when using SLURM as compared to when it uses `rsh` / `ssh`.

[2] http://www.llnl.gov/linux/slurm/

SSI parameter name	Default value	Description
boot_rsh_agent	From configure	Remote shell agent to use.
boot_rsh_fast	0	If nonzero, assume that the shell on the remote node is the same as on the origin (i.e., do not check).
boot_rsh_ignore_stderr	0	If nonzero, ignore output from stderr when booting; don't treat it as an error.
boot_rsh_priority	10	Default priority level.
boot_rsh_no_n	0	If nonzero, don't use "-n" as an argument to the boot agent
boot_rsh_no_profile	0	If nonzero, don't attempt to run ".profile" for Bourne-type shells.
boot_rsh_username	None	Username to use if different than login name.

Table 8.3: SSI parameters for the rsh boot module.

Usage

SLURM allows running jobs in multiple ways. The slurm boot module is only supported in some of them:

- "Batch" mode: where a script is submitted via the srun command and is executed on the first node from the set that SLURM allocated for the job. The script runs lamboot, mpirun, etc., as is normal for a LAM/MPI job.

  This method is supported, and is perhaps the most common way to run LAM/MPI automated jobs in SLURM environments.

- "Allocate" mode: where the "-A" option is given to srun, meaning that the shell were lamboot runs is likely to *not* be one of the nodes that SLURM has allocated for the job. In this case, LAM daemons will be launched on all nodes that were allocated by SLURM as well as the origin (i.e., the node where lamboot was run. The origin will be marked as "no-schedule," meaning that applications launched by mpirun and lamexec will not be run there unless specifically requested (see See Section 7.1, page 49, for more detail about this attribute and boot schemas in general).

  This method is supported, and is perhaps the most common way to run LAM/MPI interactive jobs in SLURM environments.

- "srun" mode: where a script is submitted via the srun command and is executed on *all* nodes that SLURM allocated for the job. In this case, the commands in the script (e.g., lamboot, mpirun, etc.) will be run on *all* nodes simultaneously, which is most likely not what you want.

  This mode is not supported.

When running in any of the supported SLURM modes, LAM will automatically detect that it should use the slurm boot module – no extra command line parameters or environment variables should be necessary. Specifically, when running in a SLURM job, the slurm module will report that it is available, and artificially inflate its priority relatively high in order to influence the boot module selection process. However, the slurm boot module can be forced by specifying the boot SSI parameter with the value of slurm.

Unlike the `rsh`/`ssh` boot module, you do not need to specify a hostfile for the slurm boot module. Instead, SLURM itself provides a list of nodes (and associated CPU counts) to LAM. Using `lamboot` is therefore as simple as:

```
shell$ lamboot
```

⊤ (7.1)

Note that in environments with multiple TCP networks, SLURM may be configured to use a network that is specifically designated for commodity traffic – another network may exist that is specifically allocated for high-speed MPI traffic. By default, LAM will use the same hostnames that SLURM provides for all of its traffic. This means that LAM will send all of its MPI traffic across the same network that SLURM uses.

However, LAM has the ability to boot using one set of hostnames / addresses and then use a second set of hostnames / addresses for MPI traffic. As such, LAM can redirect its TCP MPI traffic across a secondary network. It is possible that your system administrator has already configured LAM to operate in this manner.

If a secondary TCP network is intended to be used for MPI traffic, see the section entitled "Separating LAM and MPI TCP Traffic" in the LAM/MPI Installation Guide. Note that this functionality has no effect on non-TCP rpi modules (such as Myrinet, Infiniband, etc.). ⊥ (7.1)

Tunable Parameters

Table 8.4 lists the SSI parameters that are available to the slurm module.

SSI parameter name	Default value	Description
`boot_slurm_priority`	50	Default priority level.

Table 8.4: SSI parameters for the slurm boot module.

Special Notes

Since the slurm boot module is designed to work in SLURM jobs, it will fail if the slurm boot module is manually specified and LAM is not currently running in a SLURM job.

The slurm module does not start a shell on the remote node. Instead, the entire environment of `lamboot` is pushed to the remote nodes before starting the LAM run-time environment. ⊥ (7.1)

8.1.9 The tm Module (OpenPBS / PBS Pro / Torque)

Both OpenPBS and PBS Pro (both products of Altair Grid Technologies, LLC), contain support for the Task Management (TM) interface. Torque, the open source fork of the Open MPI product, also contains the TM interface. When using TM, `rsh`/`ssh` is not necessary to launch jobs on remote nodes.

The advantages of using the TM interface are:

- PBS/Torque can generate proper accounting information for all nodes in a parallel job.
- PBS/Torque can kill entire jobs properly when the job ends.
- `lamboot` executes significantly faster when using TM as compared to when it uses `rsh` / `ssh`.

Usage

When running in a PBS/Torque batch job, LAM will automatically detect that it should use the tm boot module – no extra command line parameters or environment variables should be necessary. Specifically, when running in a PBS/Torque job, the tm module will report that it is available, and artificially inflate its priority relatively high in order to influence the boot module selection process. However, the tm boot module can be forced by specifying the `boot` SSI parameter with the value of `tm`.

Unlike the `rsh`/`ssh` boot module, you do not need to specify a hostfile for the tm boot module. Instead, PBS/Torque itself provides a list of nodes (and associated CPU counts) to LAM. Using `lamboot` is therefore as simple as:

```
shell$ lamboot
```

⊤ (7.1) The tm boot modules works in both interactive and non-interactive batch jobs.

Note that in environments with multiple TCP networks, PBS / Torque may be configured to use a network that is specifically designated for commodity traffic – another network may exist that is specifically allocated for high-speed MPI traffic. By default, LAM will use the same hostnames that the TM interface provides for all of its traffic. This means that LAM will send all of its MPI traffic across the same network that PBS / Torque uses.

However, LAM has the ability to boot using one set of hostnames / addresses and then use a second set of hostnames / addresses for MPI traffic. As such, LAM can redirect its TCP MPI traffic across a secondary network. It is possible that your system administrator has already configured LAM to operate in this manner.

If a secondary TCP network is intended to be used for MPI traffic, see the section entitled "Separating LAM and MPI TCP Traffic" in the LAM/MPI Installation Guide. Note that this has no effect on non-TCP
⊥ (7.1) rpi modules (such as Myrinet, Infiniband, etc.).

Tunable Parameters

Table 8.5 lists the SSI parameters that are available to the tm module.

SSI parameter name	Default value	Description
`boot_tm_priority`	50	Default priority level.

Table 8.5: SSI parameters for the tm boot module.

Special Notes

Since the tm boot module is designed to work in PBS/Torque jobs, it will fail if the tm boot module is manually specified and LAM is not currently running in a PBS/Torque job.

The tm module does not start a shell on the remote node. Instead, the entire environment of `lamboot` is pushed to the remote nodes before starting the LAM run-time environment.

Also note that the Altair-provided client RPMs for PBS Pro do not include the `pbs_demux` command, which is necessary for proper execution of TM jobs. The solution is to copy the executable from the server RPMs to the client nodes.

Finally, TM does not provide a mechanism for path searching on the remote nodes, so the `lamd` executable is required to reside in the same location on each node to be booted.

Chapter 9

Available MPI Modules

There are multiple types of MPI modules:

1. rpi: MPI point-to-point communication, also known as the LAM Request Progression Interface (RPI).
2. coll: MPI collective communication.
3. cr: Checkpoint/restart support for MPI programs.

Each of these types, and the modules that are available in the default LAM distribution, are discussed in detail below.

9.1 General MPI SSI Parameters

⊤ (7.1)

The default hostmap file is `$sysconf/lam-hostmap` (typically `$prefix/etc/lam-hostmap.txt`). This file is only useful in environments with multiple TCP networks, and is typically populated by the system administrator (see the LAM/MPI Installation Guide for more details on this file).

The SSI parameter `mpi_hostmap` can be used to specify an alternate hostmap file. For example:

```
shell$ mpirun C -ssi mpi_hostmap my_hostmap.txt my_mpi_application
```

This tells LAM to use the hostmap `my_hostmap.txt` instead of `$sysconf/lam-hostmap.txt`. The special filename "`none`" can also be used to indicate that no address remapping should be performed. ⊥ (7.1)

9.2 MPI Module Selection Process

The modules used in an MPI process may be related or dependent upon external factors. For example, the gm RPI cannot be used for MPI point-to-point communication unless there is Myrinet hardware present in the node. The blcr checkpoint/restart module cannot be used unless thread support was included. And so on. As such, it is important for users to understand the module selection algorithm.

1. Set the thread level to be what was requested, either via MPI_INIT_THREAD or the environment variable LAM_MPI_THREAD_LEVEL.

2. Query relevant modules and make lists of the resulting available modules. "Relevant" means either a specific module (or set of modules) if the user specified them through SSI parameters, or all modules if not specified.

3. Eliminate all modules who do not support the current MPI thread level.

4. If no rpi modules remain, try a lower thread support level until all levels have been tried. If no thread support level can provide an rpi module, abort.

5. Select the highest priority rpi module. Reset the thread level (if necessary) to be at least the lower bound of thread levels that the selected rpi module supports.

6. Eliminate all coll and cr modules that cannot operate at the current thread level.

7. If no coll modules remain, abort. Final selection coll modules is discussed in Section 9.4.1 (page 89).

8. If no cr modules remain and checkpoint/restart support was specifically requested, abort. Otherwise, select the highest priority cr module.

9.3 MPI Point-to-point Communication (Request Progression Interface / RPI)

LAM provides multiple SSI modules for MPI point-to-point communication. Also known as the Request Progression Interface (RPI), these modules are used for all aspects of MPI point-to-point communication in an MPI application. Some of the modules require external hardware and/or software (e.g., the native Myrinet RPI module requires both Myrinet hardware and the GM message passing library). The `laminfo` command can be used to determine which RPI modules are available in a LAM installation.

Although one RPI module will likely be the default, the selection of which RPI module is used can be changed through the SSI parameter `rpi`. For example:

```
shell$ mpirun -ssi rpi tcp C my_mpi_program
```

runs the `my_mpi_program` executable on all available CPUs using the tcp RPI module, while:

```
shell$ mpirun -ssi rpi gm C my_mpi_program
```

runs the `my_mpi_program` executable on all available CPUs using the gm RPI module.

It should be noted that the choice of RPI usually does not affect the boot SSI module – hence, the `lamboot` command requirements on hostnames specified in the boot schema is not dependent upon the RPI. For example, if the gm RPI is selected, `lamboot` may still require TCP/IP hostnames in the boot schema, not Myrinet hostnames. Also note that selecting a particular module does not guarantee that it will be able to be used. For example, selecting the gm RPI module will still cause a run-time failure if there is no Myrinet hardware present.

The available modules are described in the sections below. Note that much of this information (particu- ⊤ (7.0.3) larly the tunable SSI parameters) is also available in the `lamssi_rpi(7)` manual page.

9.3.1 Two Different Shared Memory RPI Modules

The sysv (Section 9.3.6, page 86) and the usysv (Section 9.3.8, page 88) modules differ only in the mechanism used to synchronize the transfer of messages via shared memory. The sysv module uses System V semaphores while the usysv module uses spin locks with back-off. Both modules use a small number of System V semaphores for synchronizing both the deallocation of shared structures and access to the shared pool.

The blocking nature of the sysv module should generally provide better performance than usysv on oversubscribed nodes (i.e., when the number of processes is greater than the number of available processors). System V semaphores will effectively force processes yield to other processes, allowing at least some degree of fair/regular scheduling. In non-oversubscribed environments (i.e., where the number of processes is less than or equal to the number of available processors), the usysv RPI should generally provide better performance than the sysv RPI because spin locks keep processors busy-waiting. This hopefully keeps the operating system from suspending or swapping out the processes, allowing them to react immediately when the lock becomes available. ⊥ (7.0.3)

9.3.2 The crtcp Module (Checkpoint-able TCP Communication)

Module Summary	
Name:	crtcp
Kind:	rpi
Default SSI priority:	25
Checkpoint / restart:	yes

The crtcp RPI module is almost identical to the tcp module, described in Section 9.3.7. TCP sockets are used for communication between MPI processes.

Overview

The following are the main differences between the tcp and crtcp RPI modules:

- The crtcp module can be checkpointed and restarted. It is currently the *only* RPI module in LAM/MPI that supports checkpoint/restart functionality.
- The crtcp module does not have the "fast" message passing optimization that is in the tcp module. As result, there is a small performance loss in certain types of MPI applications.

All other aspects of the crtcp module are the same as the tcp module.

Checkpoint/Restart Functionality

The crtcp module is designed to work in conjunction with a cr module to provide checkpoint/restart functionality. See Section 9.5 for a description of how LAM's overall checkpoint/restart functionality is used.

The crtcp module's checkpoint/restart functionality is invoked when the cr module indicates that it is time to perform a checkpoint. The crtcp then quiesces all "in-flight" MPI messages and then allows the checkpoint to be performed. Upon restart, TCP connections are re-formed, and message passing processing continues. No additional buffers or "rollback" mechanisms are required, nor is any special coding required in the user's MPI application.

Tunable Parameters

The crtcp module has the same tunable parameters as the tcp module (maximum size of a short message and amount of OS socket buffering), although they have different names: `rpi_crtcp_short`, `rpi_crtcp_sockbuf`.

SSI parameter name	Default value	Description
`rpi_crtcp_priority`	25	Default priority level.
`rpi_crtcp_short`	65535	Maximum length (in bytes) of a "short" message.
`rpi_crtcp_sockbuf`	-1	Socket buffering in the OS kernel (-1 means use the short message size).

Table 9.1: SSI parameters for the crtcp RPI module.

9.3.3 The gm Module (Myrinet)

Module Summary	
Name:	gm
Kind:	rpi
Default SSI priority:	50
Checkpoint / restart:	yes (*)

The gm RPI module is for native message passing over Myrinet networking hardware. The gm RPI provides low latency, high bandwidth message passing performance.

Be sure to also read the release notes entitled "Operating System Bypass Communication: Myrinet and Infiniband" in the LAM/MPI Installation Guide for notes about memory management with Myrinet. Specifically, it deals with LAM's automatic overrides of the `malloc()`, `calloc()`, and `free()` functions.

Overview

In general, using the gm RPI module is just like using any other RPI module – MPI functions will simply use native GM message passing for their back-end message transport.

Although it is not required, users are strongly encouraged to use the MPI_ALLOC_MEM and MPI_FREE_MEM functions to allocate and free memory (instead of, for example, `malloc()` and `free()`.

The gm RPI module is marked as "yes" for checkpoint / restart support, but this is only true when the module was configured and compiled with the `--with-rpi-gm-get` configure flag. This enables LAM to use the GM 2.x function `gm_get()`. Note that enabling this feature slightly with the `rpi_gm_cr` SSI parameter decreases the performance of the gm module (which is why it is disabled by default) because of additional bookkeeping that is necessary. The performance difference is actually barely measurable – it is well below one microsecond. It is not the default behavior simply on principle.

At the time of this writing, there still appeared to be problems with `gm_get()`, so this behavior is disabled by default. It is not clear whether the problems with `gm_get()` are due to a problem with Myricom's GM library or a problem in LAM itself; the `--with-rpi-gm-get` option is provided as a "hedging our bets" solution; if the problem does turn out to be with the GM library, LAM users can enable checkpoint support (and slightly lower long message latency) by using this switch.

Tunable Parameters

Table 9.2 shows the SSI parameters that may be changed at run-time; the text below explains each one in detail.

SSI parameter name	Default value	Description
rpi_gm_cr	0	Whether to enable checkpoint / restart support or not.
rpi_gm_fast	0	Whether to enable the "fast" algorithm for sending short messages. This is an unreliable transport and is not recommended for MPI applications that do not continually invoke the MPI progression engine.
rpi_gm_maxport	32	Maximum GM port number to check during MPI_INIT when looking for an available port.
rpi_gm_nopin	0	Whether to let LAM/MPI register ("pin") arbitrary buffers or not.
rpi_gm_port	-1	Specific GM port to use (-1 indicates none).
rpi_gm_priority	50	Default priority level.
rpi_gm_tinymsglen	1024	Maximum length (in bytes) of a "tiny" message.

Table 9.2: SSI parameters for the gm RPI module.

Port Allocation

It is usually unnecessary to specify which Myrinet/GM port to use. LAM/MPI will automatically attempt to acquire ports greater than 1.

By default, LAM will check for any available port between 1 and 8. If your Myrinet hardware has more than 8 possible ports, you can change the upper port number that LAM will check with the rpi_gm_maxport SSI parameter.

However, if you wish LAM to use a specific GM port number (and not check all the ports from $[1, \text{maxport}]$), you can tell LAM which port to use with the rpi_gm_port SSI parameter. Specifying which port to use has precedence over the port range check – if a specific port is indicated, LAM will try to use that and not check a range of ports. Specifying to use port "-1" (or not specifying to use a specific port) will tell LAM to check the range of ports to find any available port.

Note that in all cases, if LAM cannot acquire a valid port for every MPI process in the job, the entire job will be aborted.

Be wary of forcing a specific port to be used, particularly in conjunction with the MPI dynamic process calls (e.g., MPI_COMM_SPAWN). For example, attempting to spawn a child process on a node that already has an MPI process in the same job, LAM will try to use the same specific port, which will result in failure because the MPI process already on that node will have already claimed that port.

Adjusting Message Lengths

The gm RPI uses two different protocols for passing data between MPI processes: tiny and long. Selection of which protocol to use is based solely on the length of the message. Tiny messages are sent (along with tag and communicator information) in one transfer to the receiver. Long messages use a rendezvous protocol – the envelope is sent to the destination, the receiver responds with an ACK (when it is ready), and then the sender sends another envelope followed by the data of the message.

The message lengths at which the different protocols are used can be changed with the SSI parameter `rpi_gm_tinymsglen` , which represent the maximum length of tiny messages. LAM defaults to 1,024 bytes for the maximum lengths of tiny messages.

It may be desirable to adjust these values for different kinds of applications and message passing patterns. The LAM Team would appreciate feedback on the performance of different values for real world applications.

Pinning Memory

The Myrinet native communication library (gm) can only communicate through "registered" (sometimes called "pinned") memory. In most operating systems, LAM/MPI handles this automatically by pinning user-provided buffers when required. This allows for good message passing performance, especially when re-using buffers to send/receive multiple messages.

However, the gm library does not have the ability to pin arbitrary memory on Solaris systems – auxiliary buffers must be used. Although LAM/MPI controls all pinned memory, this has a detrimental effect on performance of large messages: LAM/MPI must copy all messages from the application-provided buffer to an auxiliary buffer before it can be sent (and vice versa for receiving messages). As such, users are strongly encouraged to use the MPI_ALLOC_MEM and MPI_FREE_MEM functions instead of `malloc()` and `free()`. Using these functions will allocate "pinned" memory such that LAM/MPI will not have to use auxiliary buffers and an extra memory copy.

The `rpi_gm_nopin` SSI parameter can be used to force Solaris-like behavior. On Solaris platforms, the default value is "1", specifying to use auxiliary buffers as described above. On non-Solaris platforms, the default value is "0", meaning that LAM/MPI will attempt to pin and send/receive directly from user buffers.

Note that since LAM/MPI manages all pinned memory, LAM/MPI must be aware of memory that is freed so that it can be properly unpinned before it is returned to the operating system. Hence, LAM/MPI must intercept calls to functions such as `sbrk()` and `munmap()` to effect this behavior. Since gm cannot pin arbitrary memory on Solaris, LAM/MPI does not need to intercept these calls on Solaris machines.

To this end, support for additional memory allocation packages are included in LAM/MPI and will automatically be used on platforms that support arbitrary pinning. These memory allocation managers allow LAM/MPI to intercept the relevant functions and ensure that memory is unpinned before returning it to the operating system. Use of these managers will effectively overload all memory allocation functions (e.g., `malloc()`, `calloc()`, `free()`, etc.) for all applications that are linked against the LAM/MPI libraries (potentially regardless of whether they are using the ib RPI module or not).

See Section 3.3.1 (page 18) for more information on LAM's memory allocation managers.

Memory Checking Debuggers

When running LAM's gm RPI through a memory checking debugger (see Section 10.4), a number of "Read from unallocated" (RUA) and/or "Read from uninitialized" (RFU) errors may appear, originating from functions beginning with "`gm_*`" or "`lam_ssi_rpi_gm_*`". These RUA/RFU errors are normal – they are not

actually reads from unallocated sections of memory. The Myrinet hardware and gm kernel device driver handle some aspects of memory allocation, and therefore the operating system/debugging environment is not always aware of all valid memory. As a result, a memory checking debugger will often raise warnings, even though this is valid behavior.

Known Issues

As of LAM 7.1.5b2, the following issues still remain in the gm RPI module:

- Heterogeneity between big and little endian machines is not supported.
- The gm RPI is not supported with IMPI.
- Mixed shared memory / GM message passing is not yet supported; all message passing is through Myrinet / GM.
- XMPI tracing is not yet supported. ⊤ (7.0.3)
- The gm RPI module is designed to run in environments where the number of available processors is greater than or equal to the number of MPI processes on a given node. The gm RPI module will perform poorly (particularly in blocking MPI communication calls) if there are less processors than processes on a node. ⊥ (7.0.3)
- "Fast" support is available and slightly decreases the latency for short gm messages. However, it is unreliable and is subject to timeouts for MPI applications that do not invoke the MPI progression engine often, and is therefore not the default behavior. ⊤ (7.1)
- Support for the `gm_get()` function in the GM 2.x series is available starting with LAM/MPI 7.1, but is disabled by support. See the Installation Guide for more details.
- Checkpoint/restart support is included for the gm module, but is only possible when the gm module was compiled with support for `gm_get()`. ⊥ (7.1)

9.3.4 The ib Module (Infiniband)

⊤ (7.1)

Module Summary	
Name:	ib
Kind:	rpi
Default SSI priority:	50
Checkpoint / restart:	no

The ib RPI module is for native message passing over Infiniband networking hardware. The ib RPI provides low latency, high bandwidth message passing performance.

Be sure to also read the release notes entitled "Operating System Bypass Communication: Myrinet and Infiniband" in the LAM/MPI Installation Guide for notes about memory management with Infiniband. Specifically, it deals with LAM's automatic overrides of the `malloc()`, `calloc()`, and `free()` functions.

Overview

In general, using the ib RPI module is just like using any other RPI module – MPI functions will simply use native Infiniband message passing for their back-end message transport.

Although it is not required, users are strongly encouraged to use the MPI_ALLOC_MEM and MPI_FREE_MEM functions to allocate and free memory used for communication (instead of, for example, `malloc()` and `free()`. This would avoid the need to pin the memory during communication time and hence save on message passsing latency.

Tunable Parameters

Table 9.3 shows the SSI parameters that may be changed at run-time; the text below explains each one in detail.

SSI parameter name	Default value	Description
`rpi_ib_hca_id`	X	The string ID of the Infiniband hardware HCA to be used
`rpi_ib_num_envelopes`	64	Number of envelopes to be preposted per peer process.
`rpi_ib_port`	-1	Specific IB port to use (-1 indicates none).
`rpi_ib_priority`	50	Default priority level.
`rpi_ib_tinymsglen`	1024	Maximum length (in bytes) of a “tiny” message.
`rpi_ib_mtu`	1024	Maximum Transmission Unit (MTU) value to be used for IB.

Table 9.3: SSI parameters for the ib RPI module.

Port Allocation

It is usually unnecessary to specify which Infiniband port to use. LAM/MPI will automatically attempt to acquire ports greater than 1.

However, if you wish LAM to use a specific Infiniband port number, you can tell LAM which port to use with the `rpi_ib_port` SSI parameter. Specifying which port to use has precedence over the port range check – if a specific port is indicated, LAM will try to use that and not check a range of ports. Specifying to use port “-1” (or not specifying to use a specific port) will tell LAM to check the range of ports to find any available port.

Note that in all cases, if LAM cannot acquire a valid port for every MPI process in the job, the entire job will be aborted.

Be wary of forcing a specific port to be used, particularly in conjunction with the MPI dynamic process calls (e.g., MPI_COMM_SPAWN). For example, attempting to spawn a child process on a node that already has an MPI process in the same job, LAM will try to use the same specific port, which will result in failure because the MPI process already on that node will have already claimed that port.

Choosing an HCA ID

The HCA ID is the Mellanox Host Channel Adapter ID. For example: InfiniHost0. It is usually unnecessary to specify which HCA ID to use. LAM/MPI will search for all HCAs available and select the first one which is available. If you want to use a fixed HCA ID, then you can specify that using the `rpi_ib_hca_id` SSI parameter.

Adjusting Message Lengths

The ib RPI uses two different protocols for passing data between MPI processes: tiny and long. Selection of which protocol to use is based solely on the length of the message. Tiny messages are sent (along with tag and communicator information) in one transfer to the receiver. Long messages use a rendezvous protocol – the envelope is sent to the destination, the receiver responds with an ACK (when it is ready), and then the sender sends another envelope followed by the data of the message.

The message lengths at which the different protocols are used can be changed with the SSI parameter `rpi_ib_tinymsglen` , which represent the maximum length of tiny messages. LAM defaults to 1,024 bytes for the maximum lengths of tiny messages.

It may be desirable to adjust these values for different kinds of applications and message passing patterns. The LAM Team would appreciate feedback on the performance of different values for real world applications.

Posting Envelopes to Recieve / Scalability

Receive buffers must be posted to the IB communication hardware/library before any receives can occur. LAM/MPI uses enevelopes that contain MPI signature information, and in the case of tiny messages, they also hold the actual message contents. The size of each envelope is therefore sum of the size of the headers and the maximum size of a tiny message (controlled by `rpi_ib_tinymsglen` SSI parameter). LAM pre-posts 64 evnvelope buffers per peer process by default, but can be overridden at run-time with then `rpi_ib_num_envelopes` SSI parameter. ⊤ (7.1.2)

These two SSI parameters can have a large effect on scalability. Since LAM pre-posts a total of $((num\_processes - 1) \times num\_envelopes \times tinymsglen)$ bytes, this can be prohibitive if $num\_processes$ grows large. However, $num\_envelopes$ and $tinymsglen$ can be adjusted to help keep this number low, although they may have an effect on run-time performance. Changing the number of pre-posted envelopes effectively controls how many messages can be simultaneously flowing across the network; changing the tiny message size affects when LAM switches to use a rendezvous sending protocol instead of an eager send protocol. Relevant values for these parameters are likely to be application-specific; keep this in mind when running large parallel jobs. ⊥ (7.1.2)

Modifying the MTU value

⊤ (7.1.2)

The Maximum Transmission Unit (MTU) values to be used for Infiniband can be configured at runtime using the `rpi_ib_mtu` SSI parameter. It can take in values of 256, 512, 1024, 2048 and 4096 corresponding to MTU256, MTU512, MTU1024, MTU2048 and MTU4096 values of Infiniband MTUs respectively. The default value is 1024 (corresponding to MTU1024). ⊥ (7.1.2)

Pinning Memory

The Infiniband communication library can only communicate through "registered" (sometimes called "pinned") memory. LAM/MPI handles this automatically by pinning user-provided buffers when required. This allows for good message passing performance, especially when re-using buffers to send/receive multiple messages.

Note that since LAM/MPI manages all pinned memory, LAM/MPI must be aware of memory that is freed so that it can be properly unpinned before it is returned to the operating system. Hence, LAM/MPI must intercept calls to functions such as `sbrk()` and `munmap()` to effect this behavior.

To this end, support for additional memory allocation packages are included in LAM/MPI and will automatically be used on platforms that support arbitrary pinning. These memory allocation managers allow LAM/MPI to intercept the relevant functions and ensure that memory is unpinned before returning it to the operating system. Use of these managers will effectively overload all memory allocation functions (e.g., `malloc()`, `calloc()`, `free()`, etc.) for all applications that are linked against the LAM/MPI libraries (potentially regardless of whether they are using the ib RPI module or not).

See Section 3.3.1 (page 18) for more information on LAM's memory allocation managers.

Memory Checking Debuggers

When running LAM's ib RPI through a memory checking debugger (see Section 10.4), a number of "Read from unallocated" (RUA) and/or "Read from uninitialized" (RFU) errors may appear pertaining to VAPI. These RUA/RFU errors are normal – they are not actually reads from unallocated sections of memory. The Infiniband hardware and kernel device driver handle some aspects of memory allocation, and therefore the operating system/debugging environment is not always aware of all valid memory. As a result, a memory checking debugger will often raise warnings, even though this is valid behavior.

Known Issues

⊤ (7.1.2) As of LAM 7.1.5b2, the following issues remain in the ib RPI module:

- The ib rpi will not scale well to large numbers of processes. See the section entitled "Posting Envelopes to Receive / Scalability," above. ⊥ (7.1.2)
- On machines which have IB (VAPI) shared libraries but not the IB hardware, and when LAM is compiled with IB support, you may see some error messages like "can't open device file" when trying to use LAM/MPI, even when you are not using the IB module. This error message pertains to IB (VAPI) shared libraries and is not from within LAM/MPI. It results because when LAM/MPI tries to query the shared libraries, VAPI tries to open the IB device during the shared library init phase, which is not proper.
- Heterogeneity between big and little endian machines is not supported.
- The ib RPI is not supported with IMPI.
- Mixed shared memory / IB message passing is not yet supported; all message passing is through Infiniband.
- XMPI tracing is not yet supported.

- The ib RPI module is designed to run in environments where the number of available processors is greater than or equal to the number of MPI processes on a given node. The ib RPI module will perform poorly (particularly in blocking MPI communication calls) if there are less processors than processes on a node. ⊥ (7.1)

9.3.5 The lamd Module (Daemon-Based Communication)

Module Summary	
Name:	lamd
Kind:	rpi
Default SSI priority:	10
Checkpoint / restart:	no

The lamd RPI module uses the LAM daemons for all interprocess communication. This allows for true asynchronous message passing (i.e., messages can progress even while the user's program is executing), albeit at the cost of a significantly higher latency and lower bandwidth.

Overview

Rather than send messages directly from one MPI process to another, all messages are routed through the local LAM daemon, the remote LAM daemon (if the target process is on a different node), and then finally to the target MPI process. This potentially adds two hops to each MPI message.

Although the latency incurred can be significant, the lamd RPI can actually make message passing progress "in the background." Specifically, since LAM/MPI is an single-threaded MPI implementation, it can typically only make progress passing messages when the user's program is in an MPI function call. With the lamd RPI, since the messages are all routed through separate processes, message passing can actually occur when the user's program is *not* in an MPI function call.

User programs that utilize latency-hiding techniques can exploit this asynchronous message passing behavior, and therefore actually achieve high performance despite of the high overhead associated with the lamd RPI.[1]

Tunable Parameters

The lamd module has only one tunable parameter: its priority.

SSI parameter name	Default value	Description
rpi_lamd_priority	10	Default priority level.

Table 9.4: SSI parameters for the lamd RPI module.

[1] Several users on the LAM/MPI mailing list have mentioned this specifically; even though the lamd RPI is slow, it provides *significantly* better performance because it can provide true asynchronous message passing.

9.3.6 The sysv Module (Shared Memory Using System V Semaphores)

Module Summary	
Name:	sysv
Kind:	rpi
Default SSI priority:	30
Checkpoint / restart:	no

The sysv RPI is the one of two combination shared-memory/TCP message passing modules. Shared memory is used for passing messages to processes on the same node; TCP sockets are used for passing messages to processes on other nodes. System V semaphores are used for synchronization of the shared memory pool.

⊤ (7.0.3)

⊥ (7.0.3) Be sure to read Section 9.3.1 (page 77) on the difference between this module and the usysv module.

Overview

Processes located on the same node communicate via shared memory. One System V shared segment is shared by all processes on the same node. This segment is logically divided into three areas. The total size of the shared segment (in bytes) allocated on each node is:

$$(2 \times C) + (N \times (N - 1) \times (S + C)) + P$$

where C is the cache line size, N is the number of processes on the node, S is the maximum size of short messages, and P is the size of the pool for large messages,

The first area (of size $(2 \times C)$) is for the global pool lock. The sysv module allocates a semaphore set (of size six) for each process pair communicating via shared memory. On some systems, the operating system may need to be reconfigured to allow for more semaphore sets if running tasks with many processes communicating via shared memory.

The second area is for "postboxes," or short message passing. A postbox is used for communication one-way between two processes. Each postbox is the size of a short message plus the length of a cache line. There is enough space allocated for $(N \times (N - 1))$ postboxes. The maximum size of a short message is configurable with the `rpi_ssi_sysv_short` SSI parameter.

The final area in the shared memory area (of size P) is used as a global pool from which space for long message transfers is allocated. Allocation from this pool is locked. The default lock mechanism is a System V semaphore but can be changed to a process-shared pthread mutex lock. The size of this pool is configurable with the `rpi_ssi_sysv_shmpoolsize` SSI parameter. LAM will try to determine P at configuration time if none is explicitly specified. Larger values should improve performance (especially when an application passes large messages) but will also increase the system resources used by each task.

Use of the Global Pool

When a message larger than $(2S)$ is sent, the transport sends S bytes with the first packet. When the acknowledgment is received, it allocates $(\mathrm{messagelength} - S)$ bytes from the global pool to transfer the rest of the message.

To prevent a single large message transfer from monopolizing the global pool, allocations from the pool are actually restricted to a maximum of `rpi_ssi_sysv_shmmaxalloc` bytes. Even with this restriction, it is possible for the global pool to temporarily become exhausted. In this case, the transport will fall back

to using the postbox area to transfer the message. Performance will be degraded, but the application will progress.

Tunable Parameters

Table 9.5 shows the SSI parameters that may be changed at run-time. Each of these parameters were discussed in the previous sections.

SSI parameter name	Default value	Description
`rpi_sysv_priority`	30	Default priority level.
`rpi_sysv_pollyield`	1	Whether or not to force the use of `yield()` to yield the processor.
`rpi_sysv_shmmaxalloc`	From configure	Maximum size of a large message atomic transfer. The default value is calculated when LAM is configured.
`rpi_sysv_shmpoolsize`	From configure	Size of the shared memory pool for large messages. The default value is calculated when LAM is configured.
`rpi_sysv_short`	8192	Maximum length (in bytes) of a "short" message for sending via shared memory (i.e., on-node). Directly affects the size of the allocated "postbox" shared memory area.
`rpi_tcp_short`	65535	Maximum length (in bytes) of a "short" message for sending via TCP sockets (i.e., off-node).
`rpi_tcp_sockbuf`	-1	Socket buffering in the OS kernel (-1 means use the short message size).

Table 9.5: SSI parameters for the sysv RPI module.

9.3.7 The tcp Module (TCP Communication)

Module Summary	
Name:	`tcp`
Kind:	rpi
Default SSI priority:	20
Checkpoint / restart:	no

The tcp RPI module uses TCP sockets for MPI point-to-point communication.

Tunable Parameters

Two different protocols are used to pass messages between processes: short and long. Short messages are sent eagerly and will not block unless the operating system blocks. Long messages use a rendezvous protocol; the body of the message is not sent until a matching MPI receive is posted. The crossover point between the short and long protocol defaults to 64KB, but can be changed with the `rpi_tcp_short` SSI

⊤ (7.1) parameter, an integer specifying the maximum size (in bytes) of a short message. Additionally, the amount of socket buffering requested of the kernel defaults to the size of short messages. It can be altered with the `rpi_tcp_sockbuf` parameter. When this value is -1, the value of the `rpi_tcp_short` parameter is used. Otherwise, its value is passed to the `setsockopt(2)` system call to set the amount of operating ⊥ (7.1) system buffering on every socket that is used for MPI communication.

SSI parameter name	Default value	Description
`rpi_tcp_priority`	20	Default priority level.
`rpi_tcp_short`	65535	Maximum length (in bytes) of a "short" message.
`rpi_tcp_sockbuf`	-1	Socket buffering in the OS kernel (-1 means use the short message size).

Table 9.6: SSI parameters for the tcp RPI module.

9.3.8 The usysv Module (Shared Memory Using Spin Locks)

Module Summary	
Name:	usysv
Kind:	rpi
Default SSI priority:	40
Checkpoint / restart:	no

The usysv RPI is the one of two combination shared-memory/TCP message passing modules. Shared memory is used for passing messages to processes on the same node; TCP sockets are used for passing messages to processes on other nodes. Spin locks with back-off are used for synchronization of the shared memory pool (a System V semaphore or pthread mutex is also used for access to the per-node shared ⊤ (7.0.3) memory pool).

The nature of spin locks means that the usysv RPI will perform poorly when there are more processes than processors (particularly in blocking MPI communication calls). If no higher priority RPI modules are available (e.g., Myrinet/gm) and the user does not select a specific RPI module through the `rpi` SSI parameter, usysv may be selected as the default – even if there are more processes than processors. Users should keep this in mind; in such circumstances, it is probably better to manually select the sysv or tcp RPI ⊥ (7.0.3) modules.

Overview

Aside from synchronization, the usysv RPI module is almost identical to the sysv module. The usysv module uses spin locks with back-off. When a process backs off, it attempts to yield the processor. If the configure script found a system-provided yield function,[2] it is used. If no such function is found, then `select()` on NULL file descriptor sets with a timeout of 10us is used.

[2] Such as `yield()` or `sched_yield()`.

Tunable Parameters

Table 9.7 shows the SSI parameters that may be changed at run-time. Many of these parameters are identical to their sysv counterparts and are not re-described here.

SSI parameter name	Default value	Description
rpi_tcp_short	65535	Maximum length (in bytes) of a "short" message for sending via TCP sockets (i.e., off-node).
rpi_tcp_sockbuf	-1	Socket buffering in the OS kernel (-1 means use the short message size).
rpi_usysv_pollyield	1	Same as sysv counterpart.
rpi_usysv_priority	40	Default priority level.
rpi_usysv_readlockpoll	10,000	Number of iterations to spin before yielding the processing while waiting to read.
rpi_usysv_shmmaxalloc	From configure	Same as sysv counterpart.
rpi_usysv_shmpoolsize	From configure	Same as sysv counterpart.
rpi_usysv_short	8192	Same as sysv counterpart.
rpi_usysv_writelockpoll	10	Number of iterations to spin before yielding the processing while waiting to write.

Table 9.7: SSI parameters for the usysv RPI module.

9.4 MPI Collective Communication

MPI collective communication functions have their basic functionality outlined in the MPI standard. However, the implementation of this functionality can be optimized and/or implemented in different ways. As such, LAM provides modules for implementing the MPI collective routines that are targeted for different environments.

- Basic algorithms
- SMP-optimized algorithms
- Shared Memory algorithms

These modules are discussed in detail below. Note that the sections below each assume that support for these modules have been compiled into LAM/MPI. The laminfo command can be used to determine exactly which modules are supported in your installation (see Section 7.7, page 53).

9.4.1 Selecting a coll Module

coll modules are selected on a per-communicator basis. Most users will not need to override the coll selection mechanisms; the coll modules currently included in LAM/MPI usually select the best module for each communicator. However, mechanisms are provided to override which coll module will be selected on a given communicator.

When each communicator is created (including MPI_COMM_WORLD and MPI_COMM_SELF), all available coll modules are queried to see if they want to be selected. A coll module may therefore be in use by zero or more communicators at any given time. The final selection of which module will be used for a given communicator is based on priority; the module with the highest priority from the set of available modules will be used for all collective calls on that communicator.

Since the selection of which module to use is inherently dynamic and potentially different for each communicator, there are two levels of parameters specifying which modules should be used. The first level specifies the overall set of coll modules that will be available to *all* communicators; the second level is a per-communicator parameter indicating which specific module should be used.

The first level is provided with the `coll` SSI parameter. Its value is a comma-separated list of coll module names. If this parameter is supplied, only these modules will be queried at run time, effectively determining the set of modules available for selection on all communicators. If this parameter is not supplied, all coll modules will be queried.

The second level is provided with the MPI attribute LAM_MPI_SSI_COLL. This attribute can be set to the string name of a specific coll module on a parent communicator before a new communicator is created. If set, the attribute's value indicates the *only* module that will be queried. If this attribute is not set, all available modules are queried.

Note that no coordination is done between the SSI frameworks in each MPI process to ensure that the same modules are available and/or are selected for each communicator. Although `mpirun` allows different environment variables to be exported to each MPI process, and the value of an MPI attribute is local to each process, LAM's behavior is undefined if the same SSI parameters are not available in all MPI processes.

9.4.2 coll SSI Parameters

There are three parameters that apply to all coll modules. Depending on when their values are checked, they may be set by environment variables, command line switches, or attributes on MPI communicators.

- `coll_base_associative`: The MPI standard defines whether reduction operations are commutative or not, but makes no provisions for whether an operator is associative or not. This parameter, if defined to 1, asserts that all reduction operations on a communicator are assumed to be associative. If undefined or defined to 0, all reduction operations are assumed to be non-associative.

  This parameter is examined during every reduction operation. See **Commutative and Associative Reduction Operators**, below.

- `coll_crossover`: If set, define the maximum number of processes that will be used with a linear algorithm. More than this number of processes may use some other kind of algorithm.

  This parameter is only examined during MPI_INIT.

- `coll_reduce_crossover`: For reduction operations, the determination as to whether an algorithm should be linear or not is not based on the number of process, but rather by the number of bytes to be transferred by each process. If this parameter is set, it defines the maximum number of bytes transferred by a single process with a linear algorithm. More than this number of bytes may result in some other kind of algorithm.

  This parameter is only examined during MPI_INIT.

Commutative and Associative Reduction Operators

MPI-1 defines that all built-in reduction operators are commutative. User-defined reduction operators can specify whether they are commutative or not. The MPI standard makes no provisions for whether a reduction operation is associative or not. For some operators and datatypes, this distinction is largely irrelevant (e.g., find the maximum in a set of integers). However, for operations involving the combination of floating point numbers, associativity and commutativity matter. An *Advice to Implementors* note in MPI-1, section 4.9.1, 114:20, states:

> It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors.

Some implementations of the reduction operations may specifically take advantage of data locality, and therefore assume that the reduction operator is associative. As such, LAM will always take the conservative approach to reduction operations and fall back to non-associative algorithms (e.g., lam_basic) for the reduction operations unless specifically told to use associative (SMP-optimized) algorithms by setting the SSI parameter `coll_base_associative` to 1.

9.4.3 The lam_basic Module

Module Summary	
Name:	`lam_basic`
Kind:	coll
Default SSI priority:	0
Checkpoint / restart:	yes

The lam_basic module provides simplistic algorithms for each of the MPI collectives that are layered on top of point-to-point functionality.[3] It can be used in any environment. Its priority is sufficiently low that it will be chosen if no other coll module is available.

Many of the algorithms are twofold: for N or less processes, linear algorithms are used. For more than N processes, binomial algorithms are used. No attempt is made to determine the locality of processes, however – the lam_basic module effectively assumes that there is equal latency between all processes. All reduction operations are performed in a strictly-defined order; associativity is not assumed.

Collectives for Intercommunicators

As of now, only lam_basic module supports intercommunicator collectives according to the MPI-2 standard. These algorithms are built over point-to-point layer and they also make use of an intra-communicator collectives with the help of intra-communicator corresponding to the local group. Mapping among the intercommunicator and corresponding local-intracommunicator is separately managed in the lam_basic module.

[3] The basic algorithms are the same that have been included in LAM/MPI since at least version 6.2.

9.4.4 The smp Module

Module Summary	
Name:	smp
Kind:	coll
Default SSI priority:	50
Checkpoint / restart:	yes

The smp module is geared towards SMP nodes in a LAN. Heavily inspired by the MagPIe algorithms [6], the smp module determines the locality of processes before setting up a dynamic structure in which to perform the collective function. Although all communication is still layered on MPI point-to-point functions, the algorithms attempt to maximize the use of on-node communication before communicating with off-node processes. This results in lower overall latency for the collective operation.

The smp module assumes that there are only two levels of latency between all processes. As such, it will only allow itself to be available for selection when there are at least two nodes in a communicator and there are at least two processes on the same node.[4]

Only some of the collectives have been optimized for SMP environments. Table 9.8 shows which collective functions have been optimized, which were already optimal (from the lam_basic module), and which will eventually be optimized.

Special Notes

Since the goal of the SMP-optimized algorithms attempt to take advantage of data locality, it is strongly recommended to maximize the proximity of MPI_COMM_WORLD rank neighbors on each node. The C nomenclature to mpirun can ensure this automatically.

Also, as a result of the data-locality exploitation, the coll_base_associative parameter is highly relevant – if it is not set to 1, the smp module will fall back to the lam_basic reduction algorithms. ⊤ (7.1)

9.4.5 The shmem Module

Module Summary	
Name:	shmem
Kind:	coll
Default SSI priority:	50
Checkpoint / restart:	yes

The shmem module is developed to facilitate fast collective communication among processes on a single node. Processes on a N-way SMP node can take advantage of the shared memory for message passing. The module will be selected only if the communicator spans over a single node and the all the processes in the communicator can successfully attach the shared memory region to their address space.

The shared memory region consists two disjoint sections. First section of the shared memory is used for synchronization among the processes while the second section is used for message passing (Copying data into and from shared memory).

The second section is known as MESSAGE_POOL and is divided into N equal segments. Default value of N is 8 and is configurable with the coll_base_shmem_num_segments SSI parameter. The size of

[4] As a direct result, smp will never be selected for MPI_COMM_SELF.

MPI function	Status
MPI_ALLGATHER	Optimized for SMP environments.
MPI_ALLGATHERV	Optimized for SMP environments.
MPI_ALLREDUCE	Optimized for SMP environments.
MPI_ALLTOALL	Identical to lam_basic algorithm; already optimized for SMP environments.
MPI_ALLTOALLV	Identical to lam_basic algorithm; already optimized for SMP environments.
MPI_ALLTOALLW	lbid.
MPI_BARRIER	Optimized for SMP environments.
MPI_BCAST	Optimized for SMP environments.
MPI_EXSCAN	lbid.
MPI_GATHER	Identical to lam_basic algorithm; already optimized for SMP environments.
MPI_GATHERV	Identical to lam_basic algorithm; already optimized for SMP environments.
MPI_REDUCE	Optimized for SMP environments.
MPI_REDUCE_SCATTER	Optimized for SMP environments.
MPI_SCAN	Optimized for SMP environments.
MPI_SCATTER	Identical to lam_basic algorithm; already optimized for SMP environments.
MPI_SCATTERV	Identical to lam_basic algorithm; already optimized for SMP environments.

Table 9.8: Listing of MPI collective functions indicating which have been optimized for SMP environments.

the MESSAGE_POOL can be also configured with the `coll_base_shmem_message_pool_size` SSI parameter. Default size of the MESSAGE_POOL is (16384×8).

The first section is known as CONTROL_SECTION and it is logicallu divided into $(2 \times N + 2)$ segments. N is the number of segments in the MESSAGE_POOL section. Total size of this section is:

$$((2 \times N) + 2) \times C \times S$$

Where C is the cache line size, S is the size of the communicator. Shared variabled for synchronization are placed in different CACHELINE for each processes to prevent trashing due to cache invalidation.

General Logic behind Shared Memory Management

Each segment in the MESSAGE_POOL corresponds to *TWO* segments in the CONTROL_SECTION. Whenever a particular segment in MESSAGE_POOL is active, its corresponding segments in the CONTROL_SECTION are used for synchronization. Processes can operate on one segment (Copy the messages), set appropriate synchronizattion variables and can continue with the next message segment. This approach improves performance of the collective algorithms. All the process need to complete a MPI_BARRIER at the last (Default 8th) segment to prevent race conditions. The extra 2 segments in the CONTROL_SECTION are used exclusively for explicit MPI_BARRIER.

Only some of the collectives have been optimized for SMP environments. Table 9.9 shows which collective functions have been optimized, which were already optimal (from the lam_basic module), and which will eventually be optimized.

List of Algorithms

Only some of the collectives have been implemented using shared memory Table 9.9 shows which collective functions have been implemented and which uses lam_basic module)

Tunable Parameters

Table 9.10 shows the SSI parameters that may be changed at run-time. Each of these parameters were discussed in the previous sections.

Special Notes

LAM provides sysv and usysv RPI for the intranode communication. In this case, the collective communication also happens through the shared memory but indirectly in terms of Sends and Recvs. Shared Memory Collective algorithms avoid all the overhead associated with the indirection and provide a minimum blocking way for the collective operations.

The shared memory is created by only one process in the communicator and rest of the processes simply attach the shared memory region to their address space. The process which finalizes last, hands back the shared memory region to the kernel while processes leaving before simply detach the shared memory region from their address space.

$\perp$ (7.1)

MPI function	Status
MPI_ALLGATHER	Implemented using shared memory.
MPI_ALLGATHERV	Uses lam_basic algorithm.
MPI_ALLREDUCE	Implemented using shared memory.
MPI_ALLTOALL	Implemented using shared memory.
MPI_ALLTOALLV	Uses lam_basic algorithm.
MPI_ALLTOALLW	Uses lam_basic algorithm.
MPI_BARRIER	Implemented using shared memory.
MPI_BCAST	Implemented using shared memory.
MPI_EXSCAN	Uses lam_basic algorithm.
MPI_GATHER	Implemented using shared memory.
MPI_GATHERV	Uses lam_basic algorithm.
MPI_REDUCE	Implemented using shared memory.
MPI_REDUCE_SCATTER	Uses lam_basic algorithm.
MPI_SCAN	Uses lam_basic algorithm.
MPI_SCATTER	Implemented using shared memory.
MPI_SCATTERV	Uses lam_basic algorithm.

Table 9.9: Listing of MPI collective functions indicating which have been implemented using Shared Memory

SSI parameter name	Default value	Description
`coll_base_shmem_message_pool_size`	16384×8	Size of the shared memory pool for the messages.
`coll_base_shmem_num_segments`	8	Number of segments in the message pool section.

Table 9.10: SSI parameters for the shmem coll module.

9.5 Checkpoint/Restart of MPI Jobs

LAM supports the ability to involuntarily checkpoint and restart parallel MPI jobs. Due to the asynchronous nature of the checkpoint/restart design, such jobs must run with a thread level of at least MPI_THREAD_SERIALIZED. This allows the checkpoint/restart framework to interrupt the user's job for a checkpoint regardless of whether it is performing message passing functions or not in the MPI communications layer.

LAM does not provide checkpoint/restart functionality itself; cr SSI modules are used to invoke back-end systems that save and restore checkpoints. The following notes apply to checkpointing parallel MPI jobs:

- No special code is required in MPI applications to take advantage of LAM/MPI's checkpoint/restart functionality, although some limitations may be imposed (depending on the back-end checkpointing system that is used).
- LAM's checkpoint/restart functionality *only* involves MPI processes; the LAM universe is not checkpointed. A LAM universe must be independently established before an MPI job can be restored.
- LAM does not yet support checkpointing/restarting MPI-2 applications. In particular, LAM's behavior is undefined when checkpointing MPI processes that invoke any non-local MPI-2 functionality (including dynamic functions and IO).
- Migration of restarted processes is available on a limited basis; the crtcp RPI will start up properly regardless of what nodes the MPI processes are re-started on, but other system-level resources may or ⊤ (7.1) may not be restarted properly (e.g., open files, shared memory, etc.).
- Checkpoint files are saved using a two-phase commit protocol that is coordinated by `mpirun`. `mpirun` initiates a checkpoint request for each process in the MPI job by supplying a temporary context filename. If all the checkpoint requests completed successfully, the saved context files are renamed to ⊥ (7.1) their respective target filenames; otherwise, the checkpoint files are discarded.
- Checkpoints can only be performed after all processes have invoked MPI_INIT and before any process has invoked MPI_FINALIZE.

9.5.1 Selecting a cr Module

The cr framework coordinates with all other SSI modules to ensure that the entire MPI application is ready to be checkpointed before the back-end system is invoked. Specifically, for a parallel job to be able to checkpoint and restart, all the SSI modules that it uses must support checkpoint/restart capabilities.

All coll modules in the LAM/MPI distribution currently support checkpoint/restart capability because they are layered on MPI point-to-point functionality – as long as the RPI module being used supports checkpoint/restart, so do the coll modules. However, only one RPI module currently supports checkpoint/restart: crtcp. Attempting to checkpoint an MPI job when using any other rpi module will result in undefined behavior.

9.5.2 cr SSI Parameters

The `cr` SSI parameter can be used to specify which cr module should be used for an MPI job. An error will occur if a cr module is requested and an rpi or coll module cannot be found that supports checkpoint/restart functionality.

Additionally, the `cr_blcr_base_dir` SSI parameter can be used to specify the directory where checkpoint file(s) will be saved. If it is not set, and no default value was provided when LAM/MPI was configured (with the `--with-cr-file-dir` flag) the user's home directory is used.

9.5.3 The blcr Module

Module Summary	
Name:	`blcr`
Kind:	cr
Default SSI priority:	50
Checkpoint / restart:	yes

Berkeley Lab's Checkpoint/Restart (BLCR) [1] single-node checkpointer provides the capability for checkpointing and restarting processes under Linux. The blcr module, when used with checkpoint/restart SSI modules, will invoke the BLCR system to save and restore checkpoints.

Overview

The blcr module will only automatically be selected when the thread level is MPI_THREAD_SERIALIZED and all selected SSI modules support checkpoint/restart functionality (see the SSI module selection algorithm, Section 9.2, page 75). The blcr module can be specifically selected by setting the `cr` SSI parameter to the value `blcr`. Manually selecting the blcr module will force the MPI thread level to be at least MPI_THREAD_SERIALIZED.

Running a Checkpoint/Restart-Capable MPI Job

There are multiple ways to run a job with checkpoint/restart support:

- Use the crtcp RPI, and invoke MPI_INIT_THREAD with a requested thread level of MPI_THREAD_SERIALIZED. This will automatically make the blcr module available.

```
shell$ mpirun C -ssi rpi crtcp my_mpi_program
```

- Use the crtcp RPI and manually select the blcr module:

```
shell$ mpirun C -ssi rpi crtcp -ssi cr blcr my_mpi_program
```

⊤ (7.0.5)

Depending on the location of the BLCR shared library, it may be necessary to use the `LD_LIBRARY_PATH` environment variable to specify where it can be found. Specifically, if the BLCR library is not in the default path searched by the linker, errors will occur at run time because it cannot be found. In such cases, adding the directory where the `libcr.so*` file(s) can be found to the `LD_LIBRARY_PATH` environment variable *on all nodes where the MPI application will execute* will solve the problem. Note that this may entail editing user's "dot" files to augment the `LD_LIBRARY_PATH` variable.[5] For example:

[5] Ensure to see Section 4.1.1 for details about which shell startup files should be edited. Also note that shell startup files are *only* read when starting the LAM universe. Hence, if you change values in shell startup files, you will likely need to re-invoke the `lamboot` command to put your changes into effect.

```
# ...edit user's shell startup file to augment LD_LIBRARY_PATH...
shell$ lamboot hostfile
shell$ mpirun C -ssi rpi crtcp -ssi cr blcr my_mpi_program
```

Alternatively, the "`-x`" option to `mpirun` can be used to export the `LD_LIBRARY_PATH` environment variable to all MPI processes. For example (Bourne shell and derivates):

```
shell$ LD_LIBRARY_PATH=/location/of/blcr/lib:$LD_LIBRARY_PATH
shell$ export LD_LIBRARY_PATH
shell$ mpirun C -ssi rpi crtcp -ssi cr blcr -x LD_LIBRARY_PATH my_mpi_program
```

For C shell and derivates:

```
shell% setenv LD_LIBRARY_PATH /location/of/blcr/lib:$LD_LIBRARY_PATH
shell% mpirun C -ssi rpi crtcp -ssi cr blcr -x LD_LIBRARY_PATH my_mpi_program
```

⊥ (7.0.5)

Checkpointing and Restarting

Once a checkpoint-capable job is running, the BLCR command `cr_checkpoint` can be used to invoke a checkpoint. Running `cr_checkpoint` with the PID of `mpirun` will cause a context file to be created for `mpirun` as well as a context file for each running MPI process. Before it is checkpointed, `mpirun` will also create an application schema file to assist in restoring the MPI job. These files will all be created in the directory specified by LAM/MPI's configured default, the `cr_blcr_base_dir`, or the user's home directory if no default is specified.

The BLCR `cr_restart` command can then be invoked with the PID and context file generated from `mpirun`, which will restore the entire MPI job.

Tunable Parameters

There are no tunable parameters to the blcr cr module.

Known Issues

- BLCR has its own limitations (e.g., BLCR does not yet support saving and restoring file descriptors); see the documentation included in BLCR for further information. Check the project's main web site[6] to find out more about BLCR.

⊤ (7.1)

- Since a checkpoint request is initiated by invoking `cr_checkpoint` with the PID of `mpirun`, it is not possible to checkpoint MPI jobs that were started using the `-nw` option to `mpirun`, or directly from the command-line without using `mpirun`.
- While the two-phase commit protocol that is used to save checkpoints provides a reasonable guarantee of consistency of saved global state, there is at least one case in which this guarantee fails. For example, the renaming of checkpoint files by `mpirun` is not atomic; if a failure occurs when `mpirun` is in the process of renaming the checkpoint files, the collection of checkpoint files might result in an inconsistent global state.

[6] http://ftg.lbl.gov/

- If the BLCR module(s) are compiled dynamically, the `LD_PRELOAD` environment variable must include the location of the `libcr.so` library. This is to ensure that `libcr.so` is loaded before the PThreads library. ⊥ (7.1) ⊤ (7.1)

9.5.4 The self Module

Module Summary	
Name:	`blcr`
Kind:	cr
Default SSI priority:	25
Checkpoint / restart:	yes

The self module, when used with checkpoint/restart SSI modules, will invoke the user-defined functions to save and restore checkpoints. It is simply a mechanism for user-defined functions to be invoked at LAM's Checkpoint, Continue, and Restart phases. Hence, the *only* data that is saved during the checkpoint is what is written in the user's checkpoint function – *no MPI library state is saved at all*.

As such, the model for the self module is slightly different than, for example, the blcr module. Specifically, the Restart function is not invoked in the same process image of the process that was checkpointed. The Restart phase is invoked during MPI_INIT of a new instance of the application (i.e., it starts over from `main()`). This is described in detail below.

Overview

The self module can be specifically selected by setting the `cr` SSI parameter to the value `self`. Manually selecting the self module will force the MPI thread level to be at least MPI_THREAD_SERIALIZED.

At each of the Checkpoint, Continue, and Restart phases, LAM will make a callback to a user-specified function to do whatever is required for that phase (e.g., save or load application-level data). LAM does this by dynamically looking up functions by name at run time. The following function names are, by default, looked up and invoked at each phase:

- Checkpoint phase: `int lam_cr_self_checkpoint(void)`
- Continue phase: `int lam_cr_self_continue(void)`
- Restart phase: `int lam_cr_self_restart(void)`

To be absolutely clear: these functions are to be provided *by the application* – they are not included in the LAM library. If one of these functions cannot be found at run-time, the self module will skip that phase invocation.

The default function names can be overridden in two ways:

1. Use the `cr_self_user_prefix` to specify a prefix for all three functions. This will cause LAM to assume that the Checkpoint, Restart and Continue functions are named: `${prefix}_checkpoint`, `${prefix}_restart`, and `${prefix}_continue`, respectively, where `${prefix}` is the string value of the `cr_self_user_prefix` SSI parameter.

   For example:

```
shell$ mpirun C -ssi rpi crtcp -ssi cr self \
    -ssi cr_self_user_prefix foo my_mpi_program
```

will look for functions named `foo_checkpoint()`, `foo_continue()`, and `foo_restart()`, respectively.

2. To specify unique names of the Checkpoint, Restart and Continue functions, use the `cr_self_user_checkpoint`, `cr_self_user_restart` and the `cr_self_user_continue` SSI parameters, respectively.

   For example:

```
shell$ mpirun C -ssi rpi crtcp -ssi cr self \
    -ssi cr_self_user_checkpoint save_my_stuff \
    -ssi cr_self_user_continue do_nothing \
    -ssi cr_self_user_restart load_my_stuff \
    my_mpi_program
```

   will look for functions named `save_my_stuff()`, `do_nothing()`, and `load_my_stuff()`, respectively.

   Note that if both the `cr_ssi_user_prefix` and any of the above three parameters are specified, these parameters are given higher preference.

Note that LAM will make no special interpretation for Fortran functions.[7] Hence, if you want to have LAM call fortran functions for any of the three phases, you must specify the "mangled" name to the `cr_self_user_[checkpoint|continue|restart]` SSI parameters.

Compiling self-Checkpointable Applications

It is critically important to compile self-checkpointable applications with the appropriate linker flags to export the symbols for the Checkpoint, Continue, and Restart functions. This allows LAM to look up these symbols at run-time. Each compiler/linker's flags for this are different, but for GCC-based compilers, it is `-export`.

For example, with a GCC-based compiler, when linking the final executable with the appropriate MPI wrapper compiler (e.g., `mpicc`, `mpiCC`, or `mpif77`), use the `-export` swich as follows:

```
shell$ mpicc main.c -c
shell$ mpicc restart_functions.c -c
shell$ mpicc main.o restart_functions.o -o my_mpi_application -export
```

This will result in an MPI application that properly exports its symbols such that LAM can find the Checkpoint, Continue, and Restart functions at run-time.

[7] Fortran compilers typically "mangle" function names in one of four ways: make the name all lower case, make the name all lower case and add one underscore, make the name all lower case and add two underscores, or make the name all uppercase.

Running a Checkpoint/Restart-Capable MPI Job

Even though MPI library state is not used with the self module, a checkpoint-capable RPI must be used for the MPI application. For example, the crtcp RPI module can be selected along with the self module:

```
shell$ mpirun C -ssi rpi crtcp -ssi cr self my_mpi_program
```

Failing to use a checkpoint-capable RPI will result in undefined behavior.

Checkpointing and Restarting

Once a checkpoint-capable job is running, the LAM command `lamcheckpoint` can be used to invoke a checkpoint. Running `lamcheckpoint` with the PID of `mpirun` will cause the user-defined Checkpoint function to be invoked. Although not typically useful in the self module, the Continue function is invoked after the Checkpoint function completes (to be symmetrical with other modules). It is common to either not provide a Continue function or supply a function that does nothing. Once these functions return, process control is returned to the application.

Note that no MPI functions are allowed to be invoked in the Checkpoint or Continue functions.

Although the `lamrestart` command can be used to restart self-checkpointed applications, its invocation is quite bulky and inconvenient; it is frequently simpler to use `mpirun` itself. Remember: with self-checkpointed application, there is no possibility of actually restarting the application because no MPI library state was saved. The application must be completely restarted (i.e., start over from the top of `main()`). The self module does provide some assistance, however, if the `cr_self_do_restart` SSI parameter is set. Specifically, self will invoke the Restart function during MPI_INIT if `cr_self_do_restart` is set to 1. For example:

```
shell$ mpirun C -ssi rpi crtcp -ssi cr self \
    -ssi cr_self_do_restart 1 my_mpi_program
```

The typical model for a Restart function is to load previously-saved data and to set some global variables indicating that a restart is in progress. When MPI_INIT returns, the application can see the global variables and continue performing whatever actions are necessary to effect a restart (e.g., jump to a different point in the application).

Just like with the Checkpoint and Continue functions, no MPI functions can be invoked during the Restart function.

Troubleshooting

The most common cause for incorrect checkpoints using the self module is having LAM look for the wrong symbol names at any of the Checkpoint, Continue, or Restart phases. To verify what function names are being looked up at run time, the `cr_verbose` SSI parameter can be set. For example:

```
shell$ mpirun C -ssi rpi crtcp -ssi cr self \
    -ssi cr_verbose level:1000 my_mpi_program
```

This will output debug-level information that clearly shows the function names that LAM is looking for and whether it is able to find them or not.

If you find that LAM is looking for the right function names but is still somehow not funding the functions at run-time, ensure that you linked your application with the appropriate flag to export symbols (e.g., with GCC-based compilers, use the `-export` flag, as shown in the example above).

Known Issues

- Since a checkpoint request is initiated by invoking `lamcheckpoint` with the PID of `mpirun`, it is not possible to checkpoint MPI jobs that were started using the `-nw` option to `mpirun`, or directly from the command-line without using `mpirun`.

⊥ (7.1)

Chapter 10

Debugging Parallel Programs

LAM/MPI supports multiple methods of debugging parallel programs. The following notes and observations generally apply to debugging in parallel:

- Note that most debuggers require that MPI applications were compiled with debugging support enabled. This typically entails adding `-g` to the compile and link lines when building your MPI application.
- Unless you specifically need it, it is not recommended to compile LAM with `-g`. This will allow you to treat MPI function calls as atomic instructions.
- Even when debugging in parallel, it is possible that not all MPI processes will execute exactly the same code. For example, "if" statements that are based upon a communicator's rank of the calling process, or other location-specific information may cause different execution paths in each MPI process.

10.1 Naming MPI Objects

LAM/MPI supports the MPI-2 functions MPI_<type>_SET_NAME and MPI_<type>_GET_NAME, where <type> can be: COMM, WIN, or TYPE. Hence, you can associate relevant text names with communicators, windows, and datatypes (e.g., "6x13x12 molecule datatype", "Local group reduction intracommunicator", "Spawned worker intercommunicator"). The use of these functions is strongly encouraged while debugging MPI applications. Since they are constant-time, one-time setup functions, using these functions likely does not impact performance, and may be safe to use in production environments, too.

The rationale for using these functions is to allow LAM (and supported debuggers, profilers, and other MPI diagnostic tools) to display accurate information about MPI communicators, windows, and datatypes. For example, whenever a communicator name is available, LAM will use it in relevant error messages; when names are not available, communicators (and windows and types) are identified by index number, which – depending on the application – may vary between successive runs. The TotalView parallel debugger will also show communicator names (if available) when displaying the message queues.

10.2 TotalView Parallel Debugger

TotalView is a commercial debugger from Etnus that supports debugging MPI programs in parallel. That is, with supported MPI implementations, the TotalView debugger can automatically attach to one or more MPI

processes in a parallel application.

LAM now supports basic debugging functionality with the TotalView debugger. Specifically, LAM supports TotalView attaching to one or more MPI processes, as well as viewing the MPI message queues in supported RPI modules.

This section provides some general tips and suggested use of TotalView with LAM/MPI. It is *not* intended to replace the TotalView documentation in any way. **Be sure to consult the TotalView documentation for more information and details than are provided here.**

Note: TotalView is licensed product provided by Etnus. You need to have TotalView installed properly before you can use it with LAM.[1]

10.2.1 Attaching TotalView to MPI Processes

LAM/MPI does not need to be configured or compiled in any special way to allow TotalView to attach to MPI processes.

You can attach TotalView to MPI processes started by `mpirun` / `mpiexec` in following ways:

1. Use the `-tv` convenience argument when running `mpirun` or `mpiexec` (this is the preferred method):

   ```
   shell$ mpirun -tv [...other mpirun arguments...]
   ```

   For example:

   ```
   shell$ mpirun -tv C my_mpi_program arg1 arg2 arg3
   ```

2. Directly launch `mpirun` in TotalView (you *cannot* launch `mpiexec` in TotalView):

   ```
   shell$ totalview mpirun -a [...mpirun arguments...]
   ```

   For example:

   ```
   shell$ totalview mpirun -a C my_mpi_program arg1 arg2 arg3
   ```

   Note the `-a` argument after `mpirun`. This is necessary to tell TotalView that arguments following "`-a`" belong to `mpirun` and not TotalView.

   Also note that the `-tv` convenience argument to `mpirun` simply executes "`totalview mpirun -a ...`"; so both methods are essentially identical.

TotalView can either attach to all MPI processes in MPI_COMM_WORLD or a subset of them. The controls for "partial attach" are in TotalView, not LAM. In TotalView 6.0.0 (analogous methods may work for earlier versions of TotalView – see the TotalView documentation for more details), you need to set the parallel launch preference to "ask." In the root window menu:

1. Select File → Preferences
2. Select the Parallel tab
3. In the "When a job goes parallel" box, select "Ask what to do"
4. Click on OK

[1] Refer to `http://www.etnus.com/` for more information about TotalView.

10.2.2 Suggested Use

Since TotalView support is started with the `mpirun` command, TotalView will, by default, start by debugging `mpirun` itself. While this may seem to be an annoying drawback, there are actually good reasons for this:

- While debugging the parallel program, if you need to re-run the program, you can simply re-run the application from within TotalView itself. There is no need to exit the debugger to run your parallel application again.
- TotalView can be configured to automatically skip displaying the `mpirun` code. Specifically, instead of displaying the `mpirun` code and enabling it for debugging, TotalView will recognize the command named `mpirun` and start executing it immediately upon load. See below for details.

There are two ways to start debugging the MPI application:

1. The preferred method is to have a `$HOME/.tvdrc` file that tells TotalView to skip past the `mpirun` code and automatically start the parallel program. Create or edit your `$HOME/.tvdrc` file to include the following:

```
# Set a variable to say what the MPI ''starter'' program is
set starter_program mpirun

# Check if the newly loaded image is the starter program
# and start it immediately if it is.
proc auto_run_starter {loaded_id} {
    global starter_program
    set executable_name [TV::symbol get $loaded_id full_pathname]
    set file_component [file tail $executable_name]

    if {[string compare $file_component $starter_program] == 0} {
        puts ''Automatically starting $file_component''
        dgo
    }
}

# Append this function to TotalView's image load callbacks so that
# TotalView run this program automatically.
dlappend TV::image_load_callbacks auto_run_starter
```

   Note that when using this method, `mpirun` is actually running in the debugger while you are debugging your parallel application, even though it may not be obvious. Hence, when the MPI job completes, you'll be returned to viewing `mpirun` in the debugger. *This is normal* – all MPI processes have exited; the only process that remains is `mpirun`. If you click "Go" again, `mpirun` will launch the MPI job again.

2. Do not create the `$HOME/.tvdrc` file with the "auto run" functionality described in the previous item, but instead simply click the "go" button when TotalView launches. This runs the `mpirun`

command with the command line arguments, which will eventually launch the MPI programs and allow attachment to the MPI processes.

When TotalView initially attaches to an MPI process, you will see the code for MPI_INIT or one of its sub-functions (which will likely be assembly code, unless LAM itself was compiled with debugging information). You probably want to skip past the rest of MPI_INIT. In the Stack Trace window, click on function which called MPI_INIT (e.g., `main`) and set a breakpoint to line following call to MPI_INIT. Then click "Go".

10.2.3 Limitations

The following limitations are currently imposed when debugging LAM/MPI jobs in TotalView:

1. Cannot attach to scripts: You cannot attach TotalView to MPI processes if they were launched by scripts instead of `mpirun`. Specifically, the following won't work:

   ```
   shell$ mpirun -tv C script_to_launch_foo
   ```

   But this will:

   ```
   shell$ mpirun -tv C foo
   ```

   For that reason, since `mpiexec` is a script, although the `-tv` switch works with `mpiexec` (because it will eventually invoke `mpirun`), you cannot launch `mpiexec` with TotalView.

2. TotalView needs to launch the TotalView server on all remote nodes in order to attach to remote processes.

   The command that TotalView uses to launch remote executables might be different than what LAM/MPI uses. You may have to set this command explicitly and independently of LAM/MPI. For example, if your local environment has `rsh` disabled and only allows `ssh`, then you likely need to set the TotalView remote server launch command to "`ssh`". You can set this internally in TotalView or with the `TVDSVRLAUNCHCMD` environment variable (see the TotalView documentation for more information on this).

3. The TotalView license must be able to be found on all nodes where you expect to attach the debugger.

   Consult with your system administrator to ensure that this is set up properly. You may need to edit your "dot" files (e.g., `.profile`, `.bashrc`, `.cshrc`, etc.) to ensure that relevant environment variable settings exist on all nodes when you `lamboot`.

4. It is always a good idea to let `mpirun` finish before you rerun or exit TotalView.

5. TotalView will not be able to attach to MPI programs when you execute `mpirun` with `-s` option.

   This is because TotalView will not get the source code of your program on nodes other than the source node. We advise you to either use a common filesystem or copy the source code and executable on all nodes when using TotalView with LAM so that you can avoid the use of `mpirun`'s `-s` flag.

10.2.4 Message Queue Debugging

The TotalView debugger can show the sending, receiving, and unexepected message queues for many parallel applications. Note the following:

- The MPI-2 function for naming communicators (MPI_COMM_SET_NAME) is strongly recommended when using the message queue debugging functionality. For example, MPI_COMM_WORLD and MPI_COMM_SELF are automatically named by LAM/MPI. Naming communicators makes it significantly easier to identify communicators of interest in the debugger.

  Any communicator that is not named will be displayed as "`--unnamed--`".

- Message queue debugging of applications is not currently supported for 64 bit executables. If you attempt to use the message queue debugging functionality on a 64 bit executable, TotalView will display a warning before disabling the message queue options.

- The lamd RPI does not support the message queue debugging functionality.

- LAM/MPI does not currently provide debugging support for dynamic processes (e.g., MPI_COMM_SPAWN).

10.3 Serial Debuggers

LAM also allows the use of one or more serial debuggers when debugging a parallel program.

10.3.1 Lauching Debuggers

LAM allows the arbitrary execution of any executable in an MPI context as long as an MPI executable is eventually launched. For example, it is common to `mpirun` a debugger (or a script that launches a debugger on some nodes, and directly runs the application on other nodes) since the debugger will eventually launch the MPI process.

However, one must be careful when running programs on remote nodes that expect the use of `stdin` – `stdin` on remote nodes is redirected to `/dev/null`. For example, it is advantageous to export the `DISPLAY` environment variable, and run a shell script that invokes an `xterm` with "`gdb`" (for example) running in it on each node. For example:

```
shell$ mpirun C -x DISPLAY xterm-gdb.csh
```

Additionally, it may be desirable to only run the debugger on certain ranks in MPI_COMM_WORLD. For example, with parallel jobs that include tens or hundreds of MPI processes, it is really only feasible to attach debuggers to a small number of processes. In this case, a script may be helpful to launch debuggers for some ranks in MPI_COMM_WORLD and directly launch the application in others.

The LAM environment variable LAMRANK can be helpful in this situation. This variable is placed in the environment before the target application is executed. Hence, it is visible to shell scripts as well as the target MPI application. It is erroneous to alter the value of this variable.

Consider the following script:

```
#!/bin/csh -f
```

```
# Which debugger to run
set debugger=gdb

# On MPI_COMM_WORLD rank 0, launch the process in the debugger.
# Elsewhere, just launch the process directly.
if (''$LAMRANK'' == ''0'') then
  echo Launching $debugger on MPI_COMM_WORLD rank $LAMRANK
  $debugger $*
else
  echo Launching MPI executable on MPI_COMM_WORLD rank $LAMRANK
  $*
endif

# All done
exit 0
```

This script can be executed via `mpirun` to launch a debugger on MPI_COMM_WORLD rank 0, and directly launch the MPI process in all other cases.

10.3.2 Attaching Debuggers

In some cases, it is not possible or desirable to start debugging a parallel application immediately. For example, it may only be desirable to attach to certain MPI processes whose identity may not be known until run-time.

In this case, the technique of attaching to a running process can be used (this functionality is supported by many serial debuggers). Specifically, determine which MPI process you want to attach to. Then login to the node where it is running, and use the debugger's "attach" functionality to latch on to the running process.

10.4 Memory-Checking Debuggers

Memory-checking debuggers are an invaluable tool when debugging software (even parallel software). They can provide detailed reports about memory leaks, bad memory accesses, duplicate/bad memory management calls, etc. Some memory-checking debuggers include (but are not limited to): the Solaris Forte debugger (including the `bcheck` command-line memory checker), the Purify software package, and the Valgrind software package.

LAM can be used with memory-checking debuggers. However, LAM should be compiled with special support for such debuggers. This is because in an attempt to optimize performance, there are many structures used internally to LAM that do not always have all memory positions initialized. For example, LAM's internal `struct nmsg` is one of the underlying message constructs used to pass data between LAM processes. But since the `struct nmsg` is used in so many places, it is a generalized structure and contains fields that are not used in every situation.

By default, LAM only initializes relevant struct members before using a structure. Using a structure may involve sending the entire structure (including uninitialized members) to a remote host. This is not a problem for LAM; the remote host will also ignore the irrelevant struct members (depending on the specific function being invoked). More to the point – LAM was designed this way to avoid setting variables that will

not be used; this is a slight optimization in run-time performance. Memory-checking debuggers, however, will flag this behavior with "read from uninitialized" warnings.

The `--with-purify` option can be used with LAM's `configure` script that will force LAM to zero out *all* memory before it is used. This will eliminate the "read from uninitialized" types of warnings that memory-checking debuggers will identify deep inside LAM. This option can only be specified when LAM is configured; it is not possible to enable or disable this behavior at run-time. Since this option invokes a slight overhead penalty in the run-time performance of LAM, it is not the default.

Chapter 11

Troubleshooting

Although LAM is a robust run-time environment, and its MPI layer is a mature software system, errors do occur. Particularly when using LAM/MPI for the first time, some of the initial, per-user setup can be confusing (e.g., setting up `.rhosts` or SSH keys for password-less remote logins). This section aims to identify a few common problems and solutions.

Much more information can be found on the LAM FAQ on the main LAM web site.[1]

11.1 The LAM/MPI Mailing Lists

There are two mailing lists: one for LAM/MPI announcements, and another for questions and user discussion of LAM/MPI.

11.1.1 Announcements

This is a low-volume list that is used to announce new version of LAM/MPI, important patches, etc. To subscribe to the LAM announcement list, visit its list information page (you can also use that page to unsubscribe or change your subscription options):

`http://www.lam-mpi.org/mailman/listinfo.cgi/lam-announce`

NOTE: Users cannot post to this list; all such posts are automatically rejected – only the LAM Team can post to this list.

11.1.2 General Discussion / User Questions

BEFORE YOU POST TO THIS LIST: *Please* check all the other resources listed in this chapter first. Search the mailing list to see if anyone else had a similar problem before you did. Re-read the error message that LAM displayed to you (LAM can sometimes give *incredibly* detailed error messages that tell you *exactly* how to fix the problem). This, unfortunately, does not stop some users from cut-n-pasting the entire error message, verbatim (including the solution to their problem) into a mail message, sending it to the list, and asking "How do I fix this problem?" So please: think (and read) before you post.[2]

[1] `http://www.lam-mpi.org/faq/`

[2] Our deep appologies if some of the information in this section appears to be repetitive and condescending. Believe us when we say that we have tried all other approaches – some users simply either do not read the information provided, or only read the

This list is used for general questions and discussion of LAM/MPI. User can post questions, comments, etc. to this list. **Due to recent increases in spam, only subscribers are allowed to post to the list**. If you are not subscribed to the list, your posts will be discarded.

To subscribe or unsubscribe from the list, visit the list information page:

`http://www.lam-mpi.org/mailman/listinfo.cgi/lam`

After you have subscribed (and received a confirmation e-mail), you can send mail to the list at the following address:

You must be subscribed in order to post to the list
`lam@lam-mpi.org`
You must be subscribed in order to post to the list

Be sure to include the following information in your e-mail:

- The `config.log` file from the top-level LAM directory, if available (**please compress!**).
- The output of "`laminfo -all`".
- A *detailed* description of what is failing. The more details that you provide, the better. E-mails saying "My application doesn't work!" will inevitably be answered with requests for more information about *exactly what doesn't work*; so please include as much detailed information in your initial e-mail as possible.

NOTE: People tend to only reply to the list; if you subscribe, post, and then unsubscribe from the list, you will likely miss replies.

Also please be aware that the list goes to several hundred people around the world – it is not uncommon to move a high-volume exchange off the list, and only post the final resolution of the problem/bug fix to the list. This prevents exchanges like "Did you try X?", "Yes, I tried X, and it did not work.", "Did you try Y?", etc. from cluttering up peoples' inboxes.

11.2 LAM Run-Time Environment Problems

Some common problems with the LAM run-time environment are listed below.

11.2.1 Problems with the `lamboot` Command

Many first-time LAM users do not have their environment properly configured for LAM to boot properly. Refer to Section 4.4.2 for the list of conditions that LAM requires to boot properly. User problems with `lamboot` typically fall into one of the following categories:

- `rsh`/`ssh` is not set up properly for password-less logins to remote nodes.

e-mail address to send "help!" e-mails to. It is our hope that big, bold print will catch some people's eyes and enable them to help themselves rather than having to wait for their post to distribute around the world and then further wait for someone to reply telling them that the solution to their problem was already printed on their screen. Thanks for your time in reading all of this!

Solution: Set up `rsh`/`ssh` properly for password-less remote logins. Consult local documentation or internet tutorials for how to set up `$HOME/.rhosts` and SSH keys. Note that the LAM Team **STRONGLY** discourages the use of `+` in `.rhosts` or `host.equiv` files!

- `rsh`/`ssh` prints something on `stderr`.

  Solution: Clean up system or user "dot" files so that nothing is printed on `stderr` during a remote login.

- A LAM daemon is unable to open a connection back to `lamboot`.

  Solution: Many Linux distributions ship with firewalls enabled. LAM/MPI uses random TCP ports to communicate, and therefore firewall support must be either disabled or opened between machines that will be using LAM/MPI.

- LAM is unable to open a session directory.

  Solution: LAM needs to use a per-user, per-session temporary directory, typically located under `/tmp` (see Section 12.8, page 119). LAM must be able to read/write in this session directory; check permissions in this tree.

- LAM is unable to find the current host in the boot schema.

  Solution: LAM can only boot a universe that includes the current node. If the current node is not listed in the hostfile, or is not listed by a name that can be resolved and identified as the current node, `lamboot` (and friends) will abort.

- LAM is unable to resolve all names in the boot schema.

  Solution: All names in the boot schema must be resolvable by the boot SSI module that is being used. This typically means that there end up being IP hostnames that must be resolved to IP addresses. Resolution can occur by any valid OS mechanism (e.g., through DNS, local file lookup, etc.). Note that the name "`localhost`" (or any address that resolves to 127.0.0.1) cannot be used in a boot schema that includes more than one host – otherwise the other nodes in the resulting LAM universe will not be able to contact that host.

11.3 MPI Problems

For the most part, LAM implements the MPI standard similarly to other MPI implementations. Hence, most MPI programmers are not too surprised by how LAM handles various errors, etc. However, there are some cases that LAM handles in its own unique fashion. In these cases LAM tries to display a helpful message discussing what happened.

Here's some more background on a few of the messages:

- "One of the processes started by mpirun has exited with a nonzero exit code."

  This means that at least one MPI process has exited after invoking MPI_INIT, but before invoking MPI_FINALIZE. This is therefore an error, and LAM will abort the entire MPI application. The last line of the error message indicates the PID, node, and exit status of the failed process.

- "MPI_<function>: process in local group is dead (rank <N>, MPI_COMM_WORLD)"

  This means that some MPI function tried to communicate with a peer MPI process and discovered that the peer process is dead. Common causes of this problem include attempting to communicate with processes that have failed (which, in some cases, won't generate the "One of the processes started by mpirun has exited..." messages), or have already invoked MPI_FINALIZE. Communication should not be initiated that could involve processes that have already invoked MPI_FINALIZE. This may include using MPI_ANY_SOURCE or collectives on communicators that include processes that have already finalized.

Chapter 12

Miscellaneous

This chapter covers a variety of topics that don't conveniently fit into other chapters.

12.1 Singleton MPI Processes

It is possible to run an MPI process without the `mpirun` or `mpiexec` commands – simply run the program as one would normally launch a serial program:

```
shell$ my_mpi_program
```

Doing so will create an MPI_COMM_WORLD with a single process. This process can either run by itself, or spawn or connect to other MPI processes and become part of a larger MPI jobs using the MPI-2 dynamic function calls. A LAM RTE must be running on the local node, as with jobs started with `mpirun`.

12.2 MPI-2 I/O Support

MPI-2 I/O support is provided through the ROMIO package [16, 17]. Since support is provided through a third party package, its integration with LAM/MPI is not "complete." Specifically, everywhere the MPI-2 standard specifies an argument of type MPI_Request, ROMIO's provided functions expect an argument of type MPIO_Request.

Note, too, that the MPIO_Request types cannot be used with LAM's standard MPI_TEST and MPI_WAIT functions – ROMIO's MPIO_TEST and MPIO_WAIT functions must be used instead. There are no array versions of these functions (e.g., MPIO_TESTANY, MPIO_WAITANY, etc., do not exist).

C MPI applications wanting to use MPI-2 I/O functionality can simply include `mpi.h`. Fortran MPI applications, however, must include both `mpif.h` and `mpiof.h`.

Finally, ROMIO includes its own documentation and listings of known issues and limitations. See the `README` file in the ROMIO directory in the LAM distribution.

12.3 Fortran Process Names

Since Fortran does not portably provide the executable name of the process (similar to the way that C programs get an array of `argv`), the `mpitask` command lists the name "LAM MPI Fortran program" by default for MPI programs that used the Fortran binding for MPI_INIT or MPI_INIT_THREAD.

The environment variable LAM_MPI_PROCESS_NAME can be used to override this behavior. Setting this environment variable before invoking `mpirun` will cause `mpitask` to list that name instead of the default title. This environment variable only works for processes that invoke the Fortran binding for MPI_INIT or MPI_INIT_THREAD.

12.4 MPI Thread Support

LAM currently implements support for MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, and MPI_THREAD_SERIALIZED. The constant MPI_THREAD_MULTIPLE is provided, although LAM will never return MPI_THREAD_MULTIPLE in the `provided` argument to MPI_INIT_THREAD.

LAM makes no distinction between MPI_THREAD_SINGLE and MPI_THREAD_FUNNELED. When MPI_THREAD_SERIALIZED is used, a global lock is used to ensure that only one thread is inside any MPI function at any time.

12.4.1 Thread Level

Selecting the thread level for an MPI job is best described in terms of the two parameters passed to MPI_INIT_THREAD: `requested` and `provided`. `requested` is the thread level that the user application requests, while `provided` is the thread level that LAM will run the application with.

- If MPI_INIT is used to initialize the job, `requested` will implicitly be MPI_THREAD_SINGLE. However, if the LAM_MPI_THREAD_LEVEL environment variable is set to one of the values in Table 12.1, the corresponding thread level will be used for `requested`.
- If MPI_INIT_THREAD is used to initialized the job, the `requested` thread level is the first thread level that the job will attempt to use. There is currently no way to specify lower or upper bounds to the thread level that LAM will use.

  The resulting thread level is largely determined by the SSI modules that will be used in an MPI job; each module must be able to support the target thread level. A complex algorithm is used to attempt to find a thread level that is acceptable to all SSI modules. Generally, the algorithm starts at `requested` and works backwards towards MPI_THREAD_SINGLE looking for an acceptable level. However, any module may *increase* the thread level under test if it requires it. At the end of this process, if an acceptable thread level is not found, the MPI job will abort.

Value	Meaning
undefined	MPI_THREAD_SINGLE
0	MPI_THREAD_SINGLE
1	MPI_THREAD_FUNNELED
2	MPI_THREAD_SERIALIZED
3	MPI_THREAD_MULTIPLE

Table 12.1: Valid values for the LAM_MPI_THREAD_LEVEL environment variable.

Also note that certain SSI modules require higher thread support levels than others. For example, any checkpoint/restart SSI module will require a minimum of MPI_THREAD_SERIALIZED, and will attempt to adjust the thread level upwards as necessary (if that CR module will be used during the job).

Hence, using MPI_INIT to initialize an MPI job does not imply that the provided thread level will be MPI_THREAD_SINGLE.

12.5 MPI-2 Name Publishing

LAM supports the MPI-2 functions MPI_PUBLISH_NAME and MPI_UNPUBLISH_NAME for publishing and unpublishing names, respectively. Published names are stored within the LAM daemons, and are therefore persistent, even when the MPI process that published them dies.

As such, it is important for correct MPI programs to unpublish their names before they terminate. However, if stale names are left in the LAM universe when an MPI process terminates, the `lamclean` command can be used to clean *all* names from the LAM RTE.

12.6 Interoperable MPI (IMPI) Support

The IMPI extensions are still considered experimental, and are disabled by default in LAM. They must be enabled when LAM is configured and built (see the Installation Guide file for details).

12.6.1 Purpose of IMPI

The Interoperable Message Passing Interface (IMPI) is a standardized protocol that enables different MPI implementations to communicate with each other. This allows users to run jobs that utilize different hardware, but still use the vendor-tuned MPI implementation on each machine. This would be helpful in situations where the job is too large to fit in one system, or when different portions of code are better suited for different MPI implementations.

IMPI defines only the protocols necessary between MPI implementations; vendors may still use their own high-performance protocols within their own implementations.

Terms that are used throughout the LAM / IMPI documentation include: IMPI clients, IMPI hosts, IMPI processes, and the IMPI server. See the IMPI section of the the LAM FAQ for definitions of these terms on the LAM web site.[1]

For more information about IMPI and the IMPI Standard, see the main IMPI web site.[2].

Note that the IMPI standard only applies to MPI-1 functionality. Using non-local MPI-2 functions on communicators with ranks that live on another MPI implementation will result in undefined behavior (read: kaboom). For example, MPI_COMM_SPAWN will certainly fail, but MPI_COMM_SET_NAME works fine (because it is a local action).

12.6.2 Current IMPI functionality

LAM currently implements a subset of the IMPI functionality:

- Startup and shutdown
- All MPI-1 point-to-point functionality

[1] http://www.lam-mpi.org/faq/
[2] http://impi.nist.gov/

- Some of the data-passing collectives: MPI_ALLREDUCE, MPI_BARRIER, MPI_BCAST, MPI_-REDUCE

LAM does not implement the following on communicators with ranks that reside on another MPI implementation:

- MPI_PROBE and MPI_IPROBE
- MPI_CANCEL
- All data-passing collectives that are not listed above
- All communicator constructor/destructor collectives (e.g., MPI_COMM_SPLIT, etc.)

12.6.3 Running an IMPI Job

Running an IMPI job requires the use of an IMPI server. An open source, freely-available server is available.[3]

As described in the IMPI standard, the first step is to launch the IMPI server with the number of expected clients. The open source server from above requires at least one authentication mechanism to be specified ("none" or "key"). For simplicity, these instructions assume that the "none" mechanism will be used. Only one IMPI server needs to be launched per IMPI job, regardless of how many clients will connect. For this example, assume that there will be 2 IMPI clients; client 0 will be run in LAM/MPI, and client 1 will be run elsewhere.

```
shell$ export IMPI_AUTH_NONE=
shell$ impi_server -server 2 -auth 0
10.0.0.32:9283
```

The IMPI server must be left running for the duration of the IMPI job. The string that the IMPI server gives as output ("10.0.0.32:9283", in this case) must be given to `mpirun` when starting the LAM process that will run in IMPI:

```
shell$ mpirun -client 0 10.0.0.32:9283 C my_mpi_program
```

This will run the MPI program in the local LAM universe and connect it to the IMPI server. From there, the IMPI protocols will take over and join this program to all other IMPI clients.

Note that LAM will launch an auxiliary "helper" MPI program named `impid` that will last for the duration of the IMPI job. It acts as a proxy to the other IMPI processes, and should not be manually killed. It will die on its own accord when the IMPI job is complete. If something goes wrong, it can be killed with the `lamclean` command, just like any other MPI process.

12.6.4 Complex Network Setups

In some complex network configurations – particularly those that span multiple private networking domains – it may necessary to override the hostname that IMPI uses for connectivity (i.e., use something other that what is returned by the `hostname` command). In this case, the `IMPI_HOST_NAME` can be used. If set, this variable is expected to contain a resolvable name (or IP address) that should be used.

[3] http://www.osl.iu.edu/research/impi/

12.7 Batch Queuing System Support

LAM is now aware of some batch queuing systems. Support is currently included for PBS, LSF, and Clubmask-based systems. There is also a generic functionality that allows users of other batch queue systems to take advantages of this functionality.

- When running under a supported batch queue system, LAM will take precautions to isolate itself from other instances of LAM in concurrent batch jobs. That is, the multiple LAM instances from the same user can exist on the same machine when executing in batch. This allows a user to submit as many LAM jobs as necessary, and even if they end up running on the same nodes, a `lamclean` in one job will not kill MPI applications in another job.
- This behavior is *only* exhibited under a batch environment. Other batch systems can easily be supported – let the LAM Team know if you'd like to see support for others included. Manually setting the environment variable `LAM_MPI_SESSION_SUFFIX` on the node where `lamboot` is run achieves the same ends.

12.8 Location of LAM's Session Directory

By default, LAM will create a temporary per-user session directory in the following directory:

```
<tmpdir>/lam-<username>@<hostname>[-<session_suffix>]
```

Each of the components is described below:

`<tmpdir>` : LAM will set the prefix used for the session directory based on the following search order:

1. The value of the `LAM_MPI_SESSION_PREFIX` environment variable
2. The value of the `TMPDIR` environment variable
3. `/tmp/`

It is important to note that (unlike `LAM_MPI_SESSION_SUFFIX`), the environment variables for determining `<tmpdir>` must be set on each node (although they do not necessarily have to be the same value). `<tmpdir>` must exist before `lamboot` is run, or `lamboot` will fail.

`<username>` : The user's name on that host.

`<hostname>` : The hostname.

`<session_suffix>` : LAM will set the suffix (if any) used for the session directory based on the following search order:

1. The value of the `LAM_MPI_SESSION_SUFFIX` environment variable.
2. If running under a supported batch system, a unique session ID (based on information from the batch system) will be used.

`LAM_MPI_SESSION_SUFFIX` and the batch information only need to be available on the node from which `lamboot` is run. `lamboot` will propagate the information to the other nodes.

12.9 Signal Catching

LAM MPI now catches the signals SEGV, BUS, FPE, and ILL. The signal handler terminates the application. This is useful in batch jobs to help ensure that `mpirun` returns if an application process dies. To disable the catching of signals use the `-nsigs` option to `mpirun`.

12.10 MPI Attributes

Discussion item: Need to have discussion of built-in attributes here, such as MPI_UNIVERSE_SIZE, etc. Should specifically mention that MPI_UNIVERSE_SIZE is fixed at MPI_INIT time (at least it is as of this writing – who knows what it will be when we release 7.1? :-).

This whole section is for 7.1. (*End of discussion item.*)

Bibliography

[1] Jason Duell, Paul Hargrove, and Eric Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart, 2002.

[2] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing Interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.

[3] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI — The Complete Reference: Volume 2, the MPI-2 Extensions*. MIT Press, 1998.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[5] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

[6] Thilo Kielmann, Henri E. Bal, and Sergei Gorlatch. Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. In *International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 492–499, Cancun, Mexico, May 2000. IEEE.

[7] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[8] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.

[9] Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[10] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Boot system services interface (SSI) modules for LAM/MPI. Technical Report TR576, Indiana University, Computer Science Department, 2003.

[11] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. MPI collective operations system services interface (SSI) modules for LAM/MPI. Technical Report TR577, Indiana University, Computer Science Department, 2003.

[12] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[13] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department, 2003.

[14] The LAM/MPI Team. *LAM/MPI Installation Guide*. Open Systems Laborator, Pervasive Technology Labs, Indiana University, Bloomington, IN, 7.0 edition, May 2003.

[15] The LAM/MPI Team. *LAM/MPI User's Guide*. Open Systems Laborator, Pervasive Technology Labs, Indiana University, Bloomington, IN, 7.0 edition, May 2003.

[16] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[17] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

Index

.bash_login file, 24
.bash_profile file, 24
.bashrc file, 24
.cshrc file, 24
.login file, 24
.profile file, 24
.rhosts file, 111
.tcshrc file, 24
$HOME/.tvdrc file, 105
$sysconf/lam-hostmap file, 75

Absoft Fortran compilers, 21
AFS filesystem, 20

base_module_path SSI parameter, 46
batch queue systems, 119
    OpenPBS / PBS Pro / Torque (TM) boot SSI module, 73
    SLURM boot SSI module, 71
Berkeley Lab Checkpoint/Restart single-node checkpointer, 97
blcr checkpoint/restart SSI module, 97
boot schema, 65
boot SSI modules, 65–74
    bproc, 67
    globus, 69
    rsh (rsh/ssh), 70
    slurm, 71
    tm (PBS / Torque), 73
boot SSI parameter, 68–72, 74
boot_base_promisc SSI parameter, 67
boot_bproc_priority SSI parameter, 69
boot_globus_priority SSI parameter, 70
boot_rsh_agent SSI parameter, 17, 72
boot_rsh_fast SSI parameter, 72
boot_rsh_ignore_stderr SSI parameter, 70, 72
boot_rsh_no_n SSI parameter, 72
boot_rsh_no_profile SSI parameter, 72
boot_rsh_priority SSI parameter, 72
boot_rsh_username SSI parameter, 72
boot_slurm_priority SSI parameter, 73
boot_tm_priority SSI parameter, 74
booting the LAM run-time environment, 26
bproc boot SSI module, 67

case-insensitive filesystem, 20
checkpoint/restart SSI modules, 96–102
    blcr, 97
    selection process, 96
Clubmask, *see* batch queue systems
coll SSI parameter, 90
coll_base_associative SSI parameter, 90–92
coll_base_shmem_message_pool_size SSI parameter, 95
coll_base_shmem_num_segments SSI parameter, 95
coll_crossover SSI parameter, 90
coll_reduce_crossover SSI parameter, 90
coll_base_shmem_message_pool_size SSI parameter, 94
coll_base_shmem_num_segments SSI parameter, 92
collective SSI modules, 89–92, 94
    lam_basic, 91
    selection process, 89
    shmem, 92
    smp, 92
commands
    cr_checkpoint, 98
    cr_restart, 98
    globus-job-run, 69
    hcc (deprecated), 58
    hcp (deprecated), 58
    hf77 (deprecated), 58

lamboot, 27, 49, 55, 65, 71–74, 97, 112, 119
lamcheckpoint, 51
lamclean, 34, 52, 117
lamexec, 52
lamgrow, 52
lamhalt, 34, 53
laminfo, 16, 20, 26, 35, 41, 53, 65, 76, 89, 112
lamnodes, 28, 55
lamrestart, 55
lamshrink, 56
lamwipe, 34, 64
mpic++, 20, 29, 40, 56
mpiCC, 20, 29, 30, 40, 56
mpicc, 20, 29, 30, 40, 56
mpiexec, 16, 32, 36, 58, 104
mpif77, 29, 31, 40, 56
mpimsg, 60
mpirun, 31, 60, 66, 72, 98, 104, 107, 116, 120
mpitask, 17, 33, 63, 115
pbs_demux, 74
recon, 63
rsh, 65
srun, 72
ssh, 65
tping, 64
wipe (deprecated), 64
compiling MPI programs, 28
configure flags
--with-cr-file-dir, 97
--with-debug, 55
--with-memory-manager, 21
--with-purify, 55, 109
--with-rpi-gm-get, 78
--with-rsh, 20
cr SSI parameter, 96
cr_blcr_base_dir SSI parameter, 97, 98
cr_blcr_context_file SSI parameter, 56
cr_checkpoint command, 98
cr_restart command, 98
cr_restart_args SSI parameter, 56

debuggers, 103–109
attaching, 108
launching, 107
memory-checking, 108
serial, 107
TotalView, 104
DISPLAY environment variable, 107
dynamic environments, 21
dynamic name publishing, *see* published names

e-mail lists, 111
environment variables
DISPLAY, 107
GLOBUS_LOCATION, 69
IMPI_HOST_NAME, 118
LAM_MPI_PROCESS_NAME, 116
LAM_MPI_SESSION_PREFIX, 50, 119
LAM_MPI_SESSION_SUFFIX, 17, 50, 69, 119
LAM_MPI_SOCKET_SUFFIX (deprecated), 17
LAM_MPI_THREAD_LEVEL, 75, 116
LAMHCC (deprecated), 57
LAMHCP (deprecated), 57
LAMHF77 (deprecated), 57
LAMHOME, 67
LAMMPICC, 57
LAMMPICXX, 57
LAMMPIF77, 57
LAMRANK, 107
LAMRSH (deprecated), 17
LD_LIBRARY_PATH, 97, 98
LD_PRELOAD, 99
PATH, 69
TMPDIR, 16, 50, 119
TVDSVRLAUNCHCMD, 106

files
.bash_login, 24
.bash_profile, 24
.bashrc, 24
.cshrc, 24
.login, 24
.profile, 24
.rhosts, 111
.tcshrc, 24
$HOME/.tvdrc, 105
$sysconf/lam-hostmap, 75
libcr.so, 99
filesystem notes

AFS, 20
case-insensitive filesystems, 20
NFS, 20
Fortran compilers
Absoft, 21
fortran process names, 115

globus boot SSI module, 69
`globus-job-run` command, 69
`GLOBUS_LOCATION` environment variable, 69

`hcc` command (deprecated), 58
`hcp` command (deprecated), 58
`hf77` command (deprecated), 58
hostfile, *see* boot schema

I/O support, *see* ROMIO
IMPI, 117
running jobs, 118
server, 118
supported functionality, 117
`IMPI_HOST_NAME` environment variable, 118
Infiniband release notes, 18
Interoperable MPI, *see* IMPI

`LAM_MPI_PROCESS_NAME` environment variable, 116
`LAM_MPI_SESSION_PREFIX` environment variable, 50, 119
`LAM_MPI_SESSION_SUFFIX` environment variable, 17, 50, 69, 119
`LAM_MPI_SOCKET_SUFFIX` environment variable (deprecated), 17
`LAM_MPI_THREAD_LEVEL` environment variable, 75, 116
`lamboot` command, 27, 49, 55, 65, 71–74, 97, 112, 119
boot schema file, 65
common problems and solutions, 27
conditions for success, 27
`lamcheckpoint` command, 51
`lamclean` command, 34, 52, 117
`lamexec` command, 52
`lamgrow` command, 52
`lamhalt` command, 34, 53
`LAMHCC` environment variable (deprecated), 57
`LAMHCP` environment variable (deprecated), 57
`LAMHF77` environment variable (deprecated), 57
`LAMHOME` environment variable, 67
`laminfo` command, 16, 20, 26, 35, 41, 53, 65, 76, 89, 112
`LAMMPICC` environment variable, 57
`LAMMPICXX` environment variable, 57
`LAMMPIF77` environment variable, 57
`lamnodes` command, 28, 55
`LAMRANK` environment variable, 107
`lamrestart` command, 55
`LAMRSH` environment variable (deprecated), 17
`lamshrink` command, 56
`lamssi(7)` manual page, 26
`lamssi_boot(7)` manual page, 26
`lamssi_coll(7)` manual page, 26
`lamssi_cr(7)` manual page, 26
`lamssi_rpi(7)` manual page, 26
`lamwipe` command, 34, 64
`LD_LIBRARY_PATH` environment variable, 97, 98
`LD_PRELOAD` environment variable, 99
`libcr.so` file, 99
listserv mailing lists, 111
Load Sharing Facility, *see* batch queue systems
LSF, *see* batch queue systems

machinefile, *see* boot schema
mailing lists, 111
manual pages, 25
`lamssi(7)`, 26
`lamssi_boot(7)`, 26
`lamssi_coll(7)`, 26
`lamssi_cr(7)`, 26
`lamssi_rpi(7)`, 26
Matlab, 21
Memory management, 18
MEX functions, 21
Microsoft Windows, 21
MPI and threads, *see* threads and MPI
MPI attribute keyvals
LAM_MPI_SSI_COLL, 90
MPI collective modules, *see* collective SSI modules
MPI constants
MPI_ANY_SOURCE, 114
MPI_COMM_SELF, 16, 36, 44, 90, 92, 107

MPI_COMM_WORLD, 18, 44, 47, 59, 62, 90, 92, 104, 107, 108, 115
MPI_ERR_KEYVAL, 36
MPI_STATUS_IGNORE, 36
MPI_STATUSES_IGNORE, 36
MPI_THREAD_FUNNELED, 16, 116
MPI_THREAD_MULTIPLE, 16, 116
MPI_THREAD_SERIALIZED, 16, 96, 97, 99, 116
MPI_THREAD_SINGLE, 16, 116, 117
MPI datatypes
MPI_DARRAY, 40
MPI_INTEGER1, 35
MPI_INTEGER2, 35
MPI_INTEGER4, 35
MPI_INTEGER8, 35
MPI_LONG_LONG_INT, 38
MPI_REAL16, 35
MPI_REAL4, 35
MPI_REAL8, 35
MPI_UNSIGNED_LONG_LONG, 38
MPI_WCHAR, 38
MPI functions
MPI_ACCUMULATE, 39
MPI_ALLGATHER, 39, 93, 95
MPI_ALLGATHERV, 39, 93, 95
MPI_ALLOC_MEM, 37, 78, 80, 82
MPI_ALLREDUCE, 39, 93, 95, 118
MPI_ALLTOALL, 39, 93, 95
MPI_ALLTOALLV, 39, 93, 95
MPI_ALLTOALLW, 39, 93, 95
MPI_BARRIER, 39, 93, 95, 118
MPI_BCAST, 39, 93, 95, 118
MPI_CANCEL, 35, 36, 118
MPI_CLOSE_PORT, 38
MPI_COMM_ACCEPT, 38
MPI_COMM_C2F, 37
MPI_COMM_CONNECT, 38
MPI_COMM_CREATE_ERRHANDLER, 37, 40
MPI_COMM_CREATE_KEYVAL, 40
MPI_COMM_DELETE_ATTR, 40
MPI_COMM_DISCONNECT, 38
MPI_COMM_F2C, 37
MPI_COMM_FREE_KEYVAL, 40
MPI_COMM_GET_ATTR, 40
MPI_COMM_GET_ERRHANDLER, 37, 40
MPI_COMM_GET_NAME, 40
MPI_COMM_GET_PARENT, 38
MPI_COMM_JOIN, 38
MPI_COMM_SET_ATTR, 40
MPI_COMM_SET_ERRHANDLER, 37, 40
MPI_COMM_SET_NAME, 40, 107, 117
MPI_COMM_SPAWN, 16, 38, 79, 82, 107, 117
MPI_COMM_SPAWN_MULTIPLE, 38
MPI_COMM_SPLIT, 118
MPI_EXSCAN, 39, 93, 95
MPI_FINALIZE, 16, 36, 44, 96, 113, 114
MPI_FINALIZED, 37
MPI_FREE_MEM, 37, 78, 80, 82
MPI_GATHER, 39, 93, 95
MPI_GATHERV, 39, 93, 95
MPI_GET, 39
MPI_GET_ADDRESS, 38
MPI_GET_VERSION, 36
MPI_GROUP_C2F, 37
MPI_GROUP_F2C, 37
MPI_INFO_C2F, 37
MPI_INFO_CREATE, 37
MPI_INFO_DELETE, 37
MPI_INFO_DUP, 37
MPI_INFO_F2C, 37
MPI_INFO_FREE, 37
MPI_INFO_GET, 37
MPI_INFO_GET_NKEYS, 37
MPI_INFO_GET_NTHKEY, 37
MPI_INFO_GET_VALUELEN, 37
MPI_INFO_SET, 37
MPI_INIT, 17, 18, 36, 44, 63, 79, 90, 96, 99, 101, 106, 113, 115–117, 120
MPI_INIT_THREAD, 40, 75, 97, 115, 116
MPI_IPROBE, 118
MPI_IRECV, 35
MPI_IS_THREAD_MAIN, 40
MPI_LOOKUP_NAME, 38
MPI_OPEN_PORT, 38
MPI_PACK, 38
MPI_PACK_EXTERNAL, 38
MPI_PACK_EXTERNAL_SIZE, 38

MPI_PROBE, 118
MPI_PUBLISH_NAME, 38, 117
MPI_PUT, 39
MPI_QUERY_THREAD, 40
MPI_RECV, 63
MPI_REDUCE, 39, 91, 93, 95, 118
MPI_REDUCE_SCATTER, 39, 93, 95
MPI_REQUEST_C2F, 37
MPI_REQUEST_F2C, 37
MPI_REQUEST_GET_STATUS, 36
MPI_SCAN, 39, 93, 95
MPI_SCATTER, 39, 93, 95
MPI_SCATTERV, 39, 93, 95
MPI_SEND, 18
MPI_STATUS_C2F, 37
MPI_STATUS_F2C, 37
MPI_TEST, 115
MPI_TYPE_C2F, 37
MPI_TYPE_CREATE_DARRAY, 38
MPI_TYPE_CREATE_HINDEXED, 38
MPI_TYPE_CREATE_HVECTOR, 38
MPI_TYPE_CREATE_INDEXED_BLOCK, 36
MPI_TYPE_CREATE_KEYVAL, 40
MPI_TYPE_CREATE_RESIZED, 38
MPI_TYPE_CREATE_STRUCT, 38
MPI_TYPE_CREATE_SUBARRAY, 38
MPI_TYPE_DELETE_ATTR, 40
MPI_TYPE_DUP, 40
MPI_TYPE_F2C, 37
MPI_TYPE_FREE_KEYVAL, 40
MPI_TYPE_GET_ATTR, 40
MPI_TYPE_GET_CONTENTS, 40
MPI_TYPE_GET_ENVELOPE, 40
MPI_TYPE_GET_EXTENT, 38, 40
MPI_TYPE_GET_NAME, 40
MPI_TYPE_GET_TRUE_EXTENT, 38, 40
MPI_TYPE_SET_ATTR, 40
MPI_TYPE_SET_NAME, 40
MPI_UNPACK, 38
MPI_UNPACK_EXTERNAL, 38
MPI_UNPUBLISH_NAME, 38, 117
MPI_WAIT, 115
MPI_WIN_C2F, 37
MPI_WIN_COMPLETE, 39
MPI_WIN_CREATE, 39
MPI_WIN_CREATE_ERRHANDLER, 37, 40
MPI_WIN_CREATE_KEYVAL, 40
MPI_WIN_DELETE_ATTR, 40
MPI_WIN_F2C, 37
MPI_WIN_FENCE, 39
MPI_WIN_FREE, 39
MPI_WIN_FREE_KEYVAL, 40
MPI_WIN_GET_ATTR, 40
MPI_WIN_GET_ERRHANDLER, 37, 40
MPI_WIN_GET_GROUP, 39
MPI_WIN_GET_NAME, 40
MPI_WIN_POST, 39
MPI_WIN_SET_ATTR, 40
MPI_WIN_SET_ERRHANDLER, 37, 40
MPI_WIN_SET_NAME, 40
MPI_WIN_START, 39
MPI_WIN_WAIT, 39
MPI_BARRIER, 94
MPIL_COMM_SPAWN, 38
MPIO_TEST, 115
MPIO_TESTANY, 115
MPIO_WAIT, 115
MPIO_WAITANY, 115
MPI types
MPI::BOOL, 41
MPI::COMPLEX, 41
MPI::DOUBLE_COMPLEX, 41
MPI::LONG_DOUBLE_COMPLEX, 41
MPI_File, 37
MPI_Info, 37, 38
MPI_Request, 115
MPI_Status, 36, 39
MPIO_Request, 115
MPI-2 I/O support, *see* ROMIO
`mpi_hostmap` SSI parameter, 75
`mpic++` command, 20, 29, 40, 56
`mpiCC` command, 20, 29, 30, 40, 56
`mpicc` command, 20, 29, 30, 40, 56
`mpiexec` command, 16, 32, 36, 58, 104
`mpif77` command, 29, 31, 40, 56
`mpimsg` command, 60
`mpirun` command, 31, 60, 66, 72, 98, 104, 107, 116, 120
`mpitask` command, 17, 33, 63, 115
fortran process names, 115

Myrinet release notes, 18

name publising, *see* published names
NFS filesystem, 20
`no-schedule` boot schema attribute, 51

OpenPBS, *see* batch queue systems

`PATH` environment variable, 69
PBS, *see* batch queue systems
PBS Pro, *see* batch queue systems
`pbs_demux` command, 74
Portable Batch System, *see* batch queue systems
published names, 117

`recon` command, 63
release notes, 15–22
ROMIO, 115
`rpi` SSI parameter, 76
`rpi_crtcp_priority` SSI parameter, 78
`rpi_crtcp_short` SSI parameter, 78
`rpi_crtcp_sockbuf` SSI parameter, 78
`rpi_gm_cr` SSI parameter, 79
`rpi_gm_fast` SSI parameter, 79
`rpi_gm_maxport` SSI parameter, 79
`rpi_gm_nopin` SSI parameter, 79
`rpi_gm_port` SSI parameter, 79
`rpi_gm_priority` SSI parameter, 79
`rpi_gm_tinymsglen` SSI parameter, 79, 80
`rpi_ib_hca_id` SSI parameter, 82
`rpi_ib_mtu` SSI parameter, 82, 83
`rpi_ib_num_envelopes` SSI parameter, 82, 83
`rpi_ib_port` SSI parameter, 82
`rpi_ib_priority` SSI parameter, 82
`rpi_ib_tinymsglen` SSI parameter, 82, 83
`rpi_lamd_priority` SSI parameter, 85
`rpi_ssi_sysv_shmmaxalloc` SSI parameter, 86
`rpi_ssi_sysv_shmpoolsize` SSI parameter, 86
`rpi_ssi_sysv_short` SSI parameter, 86
`rpi_sysv_pollyield` SSI parameter, 87
`rpi_sysv_priority` SSI parameter, 87
`rpi_sysv_shmmaxalloc` SSI parameter, 87
`rpi_sysv_shmpoolsize` SSI parameter, 87
`rpi_sysv_short` SSI parameter, 87
`rpi_tcp_priority` SSI parameter, 88
`rpi_tcp_short` SSI parameter, 87–89
`rpi_tcp_sockbuf` SSI parameter, 87–89
`rpi_usysv_pollyield` SSI parameter, 89
`rpi_usysv_priority` SSI parameter, 89
`rpi_usysv_readlockpoll` SSI parameter, 89
`rpi_usysv_shmmaxalloc` SSI parameter, 89
`rpi_usysv_shmpoolsize` SSI parameter, 89
`rpi_usysv_short` SSI parameter, 89
`rpi_usysv_writelockpoll` SSI parameter, 89
RPMs, 19
rsh (`ssh`) boot SSI module, 70
`rsh` command, 65
running MPI programs, 31

sample MPI program
  C, 29
  C++, 30
  Fortran, 31
serial debuggers, 107
session directory, 119
shell setup
  Bash/Bourne shells, 25
  C shell (and related), 25
signals, 120
slurm boot SSI module, 71
`srun` command, 72
`ssh` command, 65
SSI
  module types, 43
  overview, 43–46
  parameter overview, 44
SSI boot modules, *see* boot SSI modules
SSI collective modules, *see* collective SSI modules
SSI parameters
  `base_module_path`, 46
  `boot`, 68–72, 74
    `bproc` value, 68
    `globus` value, 69, 70
    `rsh` value, 71
    `slurm` value, 72
    `tm` value, 74
  `boot_base_promisc`, 67
  `boot_bproc_priority`, 69
  `boot_globus_priority`, 70
  `boot_rsh_agent`, 17, 72

boot_rsh_fast, 72
boot_rsh_ignore_stderr, 70, 72
boot_rsh_no_n, 72
boot_rsh_no_profile, 72
boot_rsh_priority, 72
boot_rsh_username, 72
boot_slurm_priority, 73
boot_tm_priority, 74
coll, 90
coll_base_associative, 90–92
coll_base_shmem_message_pool_size, 95
coll_base_shmem_num_segments, 95
coll_crossover, 90
coll_reduce_crossover, 90
coll_base_shmem_message_pool_size, 94
coll_base_shmem_num_segments, 92
cr, 96
  blcr value, 97
  self value, 99
cr_blcr_base_dir, 97, 98
cr_blcr_context_file, 56
cr_restart_args, 56
mpi_hostmap, 75
rpi, 76
rpi_crtcp_priority, 78
rpi_crtcp_short, 78
rpi_crtcp_sockbuf, 78
rpi_gm_cr, 79
rpi_gm_fast, 79
rpi_gm_maxport, 79
rpi_gm_nopin, 79
rpi_gm_port, 79
rpi_gm_priority, 79
rpi_gm_tinymsglen, 79, 80
rpi_ib_hca_id, 82
rpi_ib_mtu, 82, 83
rpi_ib_num_envelopes, 82, 83
rpi_ib_port, 82
rpi_ib_priority, 82
rpi_ib_tinymsglen, 82, 83
rpi_lamd_priority, 85
rpi_ssi_sysv_shmmaxalloc, 86
rpi_ssi_sysv_shmpoolsize, 86
rpi_ssi_sysv_short, 86
rpi_sysv_pollyield, 87
rpi_sysv_priority, 87
rpi_sysv_shmmaxalloc, 87
rpi_sysv_shmpoolsize, 87
rpi_sysv_short, 87
rpi_tcp_priority, 88
rpi_tcp_short, 87–89
rpi_tcp_sockbuf, 87–89
rpi_usysv_pollyield, 89
rpi_usysv_priority, 89
rpi_usysv_readlockpoll, 89
rpi_usysv_shmmaxalloc, 89
rpi_usysv_shmpoolsize, 89
rpi_usysv_short, 89
rpi_usysv_writelockpoll, 89
System Services Interface, *see* SSI

threads and MPI, 116
tm boot SSI module, 73
TMPDIR environment variable, 16, 50, 119
TotalView parallel debugger, 104
tping command, 64
TVDSVRLAUNCHCMD environment variable, 106

Windows, *see* Microsoft Windows
wipe command (deprecated), 64
--with-cr-file-dir configure flag, 97
--with-debug configure flag, 55
--with-memory-manager configure flag, 21
--with-purify configure flag, 55, 109
--with-rpi-gm-get configure flag, 78
--with-rsh configure flag, 20
wrapper compilers, 56