

Typeset with  $\text{\LaTeX}$  version 2.17 (2004/04/05)  
on April 12, 2004

for  
Jeffrey M. Squyres  
entitled

## A COMPONENT ARCHITECTURE

FOR THE MESSAGE PASSING INTERFACE (MPI):

THE SYSTEMS SERVICES INTERFACE (SSI) OF LAM/MPI

This class conforms to the University of Notre Dame style guidelines established Spring 2004. However it is still possible to generate a non-conformant document if the published instructions are not followed! Be sure to refer to the published Graduate School guidelines as well.

THIS IS A TEMPORARY VERSION OF THIS CLASSFILE. IT IS ONLY INTENDED TO BE USED FOR DISSERTATIONS IN THE SPRING OF 2004. A new version of this classfile will be available after that, and should be used for all future dissertations.

*This summary page can be disabled by specifying the `nosummary` option to the class invocation. (i.e., `\documentclass[nosummary]{ndthesis}`)*

**THIS PAGE IS NOT PART OF THE THESIS, BUT SHOULD BE  
TURNED IN TO THE PROOFREADER!**

$\text{\LaTeX}$  documentation can be found at these locations:

<http://www.nd.edu/~afsunix/faq/tetexdoc/latex/ndthesis/>  
<http://www.cse.nd.edu/~jsquyres/ndthesis/>

General  $\text{\LaTeX}$  documentation and info:

**On-line docs:**

ND installation <http://www.nd.edu/~afsunix/faq/tetexdoc/>  
 $\text{\TeX}$  User's Group <http://www.tug.org/>

**Books:**

*A Guide...for Beg. & Adv. Users* by Kopka/Daly  
 *$\text{\TeX}$  User's Guide...* by Lamport  
*The  $\text{\TeX}$  Companion* by Goossens/Mittelbach/Samarin

**Packages:** (check on-line docs)

rotating sideways tables and figures  
longtable multi-page tables  
graphicx using Postscript and other figures

A COMPONENT ARCHITECTURE  
FOR THE MESSAGE PASSING INTERFACE (MPI):  
THE SYSTEMS SERVICES INTERFACE (SSI) OF LAM/MPI

A Dissertation

Submitted to the Graduate School  
of the University of Notre Dame  
in Partial Fulfillment of the Requirements  
for the Degree of

Doctor of Philosophy

by

Jeffrey M. Squyres, B.A., B.S., M.S.

---

Andrew Lumsdaine, Director

Graduate Program in Computer Science and Engineering

Notre Dame, Indiana

April 2004

A COMPONENT ARCHITECTURE  
FOR THE MESSAGE PASSING INTERFACE (MPI):  
THE SYSTEMS SERVICES INTERFACE (SSI) OF LAM/MPI

Abstract

by

Jeffrey M. Squyres

This work presents the design and implementation of a component system architecture in LAM/MPI, a production quality, open source implementation of the MPI-1 and MPI-2 standards. Previous versions of LAM/MPI, as well as other MPI implementations, are based on monolithic software architectures that – regardless of how well-abstracted and logically constructed – are highly complex software packages, presenting a steep learning curve for new developers and third parties. As a result, parallel researchers face enormous logistical and technical difficulties when using or adapting existing implementations for their own work. Not only are existing code bases typically locked into highly-specific implementation models (effectively preventing extensions that did not already conform to existing models), but the time investment required to train a researcher in a complex software system can be prohibitive. To address these issues, the current version of LAM/MPI has been re-architected to utilize a component system architecture consisting of four component frameworks and a meta framework that ties them together. Each component framework was designed from analysis of prior monolithic implementations of LAM/MPI and represents a major functional category: run-time environment startup, MPI point-to-point communication, MPI collective communication, and parallel check-

Jeffrey M. Squyres

point/restart. The result is an MPI implementation that is highly modular, has published abstraction and interface boundaries, and is significantly easier to develop, maintain, and use as a vehicle for research. Performance results are shown demonstrating that this component-based approach provides identical (if not better) performance compared to prior monolithic-based implementations.

To all those who waited for me.

## CONTENTS

FIGURES . . . . .	x
TABLES . . . . .	xvi
ACKNOWLEDGMENTS . . . . .	xvii
CHAPTER 1: INTRODUCTION . . . . .	1
1.1 Preventing Code Base Fracturing . . . . .	4
1.2 Contribution . . . . .	8
1.3 Document Organization . . . . .	9
CHAPTER 2: BACKGROUND . . . . .	11
2.1 The Message Passing Interface (MPI) . . . . .	11
2.2 The LAM Implementation of MPI . . . . .	12
2.2.1 Layered Design . . . . .	12
2.2.2 Open Source Model . . . . .	14
2.2.3 Software as a Research Artifact . . . . .	17
2.2.4 Software Engineering . . . . .	18
2.3 The Evolution of LAM/MPI . . . . .	23
2.3.1 Component Programming . . . . .	24
2.3.2 The System Services Interface (SSI) . . . . .	25
2.4 Related Work . . . . .	26
2.4.1 Component Programming . . . . .	26
2.4.2 Run-Time Environment Startup . . . . .	27
2.4.3 MPI Point-to-Point Message Passing . . . . .	29
2.4.4 MPI Collective Message Passing . . . . .	29
2.4.5 Parallel Checkpoint/Restart . . . . .	30
2.4.6 Fault Tolerance . . . . .	31
2.5 Experimental Setup . . . . .	32
CHAPTER 3: THE SYSTEM SERVICES INTERFACE . . . . .	34
3.1 Design . . . . .	35
3.1.1 Multiple Frameworks . . . . .	36

3.1.2	Run-Time Module Selection	36
3.1.3	LAM and MPI Component Types	38
3.1.4	Module Services	39
3.1.5	Static and Dynamic Modules	39
3.2	Implemented Services	40
3.2.1	Configuration and Compilation	40
3.2.2	Installation	40
3.2.3	Versioning	41
3.2.4	Loading / Unloading	41
3.2.5	Parameter Passing	42
3.2.6	The <code>laminfo</code> Command	43
3.2.7	Common Selection Schemes	43
CHAPTER 4: RUN-TIME ENVIRONMENT STARTUP		46
4.1	Design	48
4.1.1	LAM Daemon-Based Run-Time Environment	48
4.1.2	Remote Process Startup Case Study: <code>rsh</code> / <code>ssh</code>	49
4.1.3	Remote Process Startup Case Study: The Portable Batch System (PBS)	51
4.1.4	Action Abstractions	54
4.1.5	Additional Tools	57
4.1.6	Module Selection	58
4.1.7	Rendezvous Algorithms	58
4.2	Implemented Modules	59
4.2.1	The <code>bproc</code> Module	59
4.2.2	The <code>globus</code> Module	60
4.2.3	The <code>rsh</code> Module	62
4.2.4	The PBS <code>tm</code> Module	62
4.3	Results	64
4.3.1	Correctness	65
4.3.2	The <code>bproc</code> Module	66
4.3.3	The <code>globus</code> Module	68
4.3.4	The <code>rsh</code> Module	69
4.3.5	The PBS <code>tm</code> Module	69
CHAPTER 5: MPI POINT-TO-POINT COMMUNICATION		74
5.1	Design	75
5.1.1	Prior Implementation	76
5.1.2	Data Structures	77
5.1.3	Module Selection and Initialization	80
5.1.4	Adding and Removing Processes	81
5.1.5	Request Lifecycle	81
5.1.6	Progression	82
5.1.7	“Fast” Send and Receive	83
5.1.8	Memory Management	84
5.1.9	Checkpoint / Restart Functionality	84
5.1.10	Module Finalization	84

5.2	Implemented Modules	85
5.2.1	The lamd Module	85
5.2.2	The tcp and crtcp Modules	87
5.2.3	The Shared Memory sysv and usysv Modules	89
5.2.4	The Myrinet gm Module	90
5.3	Results	92
5.3.1	Correctness	92
5.3.2	Performance	93
CHAPTER 6: MPI COLLECTIVE ALGORITHMS		102
6.1	Design	104
6.1.1	Layered over Point-to-Point	105
6.1.2	Dedicated Communication Channels	106
6.1.3	Hierarchical coll Modules	106
6.1.4	Module Lifecycle	108
6.2	Implemented Modules	111
6.2.1	The lam_basic Module	111
6.2.2	The smp Module	112
6.3	Results	115
6.3.1	Correctness	115
6.3.2	Performance	118
CHAPTER 7: PARALLEL CHECKPOINT / RESTART		131
7.1	Checkpoint-Based Rollback Recovery	132
7.1.1	Uncoordinated Checkpointing	133
7.1.2	Coordinated Checkpointing	134
7.1.3	Communication-Induced Checkpointing	134
7.1.4	Other Uses of Checkpoint/Restart	135
7.2	Design	136
7.2.1	Overview	136
7.2.2	Prepare For Checkpoint	139
7.2.3	Checkpoint	143
7.2.4	Continue After Checkpoint	143
7.2.5	Restart After Checkpoint	143
7.2.6	Module Selection Mechanism	144
7.2.7	Interaction With Other Modules	145
7.3	Implemented Modules	146
7.3.1	The bcr Module	146
7.3.2	The self Module	149
7.4	Results	150
7.4.1	Correctness	150
7.4.2	Performance	154
CHAPTER 8: CONCLUSIONS		161
8.1	The SSI Meta Framework	163
8.2	The boot Component Framework	164

8.3	The rpi Component Framework . . . . .	165
8.4	The coll Component Framework . . . . .	165
8.5	The cr Component Framework . . . . .	166
8.6	Delivered Software and Documentation . . . . .	166
8.7	Future Work . . . . .	166
APPENDIX A: SSI FRAMEWORK INTERFACE . . . . .		170
A.1	Notation . . . . .	170
A.1.1	The Prefix Rule . . . . .	171
A.1.2	Function Parameters . . . . .	171
A.1.3	Historical Names . . . . .	171
A.1.4	Common LAM Types, Variables, and Functions . . . . .	172
A.2	Directory Layout and Contents . . . . .	172
A.3	Configuring the Module . . . . .	173
A.3.1	Generating configure Scripts . . . . .	173
A.3.2	Running configure Scripts . . . . .	178
A.4	Building the Module . . . . .	184
A.5	Installing the Module . . . . .	185
A.6	Module Source Code . . . . .	185
A.6.1	Header Files . . . . .	187
A.6.2	The Base Module Datatype: lam_ssi_t . . . . .	188
A.6.3	Module Open Function . . . . .	190
A.6.4	Module Close Function . . . . .	191
A.6.5	Example Usage: The tcp rpi Module . . . . .	192
A.6.6	Module Parameters . . . . .	192
APPENDIX B: PARALLEL JOB STARTUP COMPONENT INTERFACE . . . . .		195
B.1	Services Provided by the boot Component Framework . . . . .	195
B.1.1	Header Files . . . . .	195
B.1.2	Internal Type: struct lamnode . . . . .	196
B.1.3	Internal Type: struct psc . . . . .	197
B.1.4	Internal Type: lam_ssi_boot_proc_t . . . . .	198
B.1.5	Global Variable: int lam_ssi_boot_base_server_port . . . . .	198
B.1.6	Global Variable: int lam_ssi_boot_optd . . . . .	199
B.1.7	Utility Function: bhostparse() . . . . .	199
B.1.8	Utility Function: hbootparse() . . . . .	200
B.1.9	Utility Function: lam_deallocate_nodes() . . . . .	200
B.1.10	Utility Function: lam_ssi_boot_base_find_boot_schema() . . . . .	202
B.1.11	Utility Function: lam_ssi_boot_base_find_hostname() . . . . .	203
B.1.12	Utility Function: lam_ssi_boot_base_lamgrow() . . . . .	203
B.1.13	Utility Function: lam_ssi_boot_base_ioexecvp() . . . . .	204
B.1.14	Utility Function: lam_ssi_boot_base_send_lamd_info_-args() . . . . .	205
B.1.15	Utility Function: lam_ssi_boot_build_inet_topo() . . . . .	206
B.1.16	Utility Function: lam_ssi_boot_do_common_args() . . . . .	206
B.1.17	Built-in Algorithms . . . . .	207
B.1.18	TCP-Based Startup Rendezvous Protocols . . . . .	208

B.2	boot Component Framework Module API	210
B.2.1	Data Item: <code>lsb_meta_info</code>	212
B.2.2	Launch Function: <code>lsb_init</code>	212
B.2.3	Launch Function: <code>lsb_finalize</code>	213
B.2.4	Launch Function: <code>lsba_parse_options</code>	213
B.2.5	Launch Function: <code>lsba_allocate_nodes</code>	213
B.2.6	Launch Function: <code>lsba_verify_nodes</code>	214
B.2.7	Launch Function: <code>lsba_prepare_boot</code>	215
B.2.8	Launch Function: <code>lsba_start_rte_procs</code>	215
B.2.9	Launch Function: <code>lsba_deallocate_nodes</code>	216
B.2.10	Algorithm Callback Function: <code>lsba_start_application</code>	216
B.2.11	Algorithm Callback Function: <code>lsba_start_rte_proc</code>	217
B.2.12	Rendezvous Function: <code>lsba_open_srv_connection</code>	218
B.2.13	Rendezvous Function: <code>lsba_send_lamd_info</code>	218
B.2.14	Rendezvous Function: <code>lsba_receive_lamd_info</code>	219
B.2.15	Rendezvous Function: <code>lsba_close_srv_connection</code>	219
B.2.16	Rendezvous Function: <code>lsba_send_universe_info</code>	220
B.2.17	Rendezvous Function: <code>lsba_receive_universe_info</code>	220

## APPENDIX C: MPI POINT-TO-POINT COMMUNICATION COMPONENT INTERFACE

		222
C.1	Services Provided by the rpi SSI	222
C.1.1	Header Files	222
C.1.2	Internal Type: <code>struct _gps</code>	223
C.1.3	Internal Type: <code>struct _proc</code>	223
C.1.4	Internal Type: <code>struct _req</code>	225
C.1.5	Internal Type: <code>struct _comm</code>	231
C.1.6	Internal Type: <code>struct _group</code>	233
C.1.7	Internal Type: <code>struct _status</code>	235
C.1.8	Internal Type: <code>struct lam_ssi_rpi_envl</code>	235
C.1.9	Internal Type: <code>struct lam_ssi_cbuf_msg</code>	236
C.1.10	Global Variable: <code>struct _proc *lam_myproc</code>	238
C.1.11	Utility Function: <code>lam_memcpy()</code>	238
C.1.12	Utility Function: <code>lam_ssi_rpi_base_alloc_mem()</code>	239
C.1.13	Utility Function: <code>lam_ssi_rpi_base_free_mem()</code>	239
C.1.14	Utility Functions: Unexpected Message Buffering	239
C.2	rpi Component Framework Module API	240
C.2.1	Restrictions	241
C.2.2	Data Item: <code>lsr_meta_info</code>	241
C.2.3	Function Call: <code>lsr_query</code>	243
C.2.4	Function Call: <code>lsr_init</code>	243
C.2.5	Function Call: <code>lsra_addprocs</code>	244
C.2.6	Function Call: <code>lsra_finalize</code>	245
C.2.7	Function Call: <code>lsra_build</code>	246
C.2.8	Function Call: <code>lsra_start</code>	246
C.2.9	Function Call: <code>lsra_advance</code>	247
C.2.10	Function Call: <code>lsra_destroy</code>	248
C.2.11	Function Call: <code>lsra_iprobe</code>	248

C.2.12	Function Call: <code>lsra_fastrecv</code>	249
C.2.13	Function Call: <code>lsra_fastsend</code>	250
C.2.14	Function Call: <code>lsra_alloc_mem</code>	250
C.2.15	Function Call: <code>lsra_free_mem</code>	251
C.2.16	Function Call: <code>lsra_interrupt</code>	251
C.2.17	Function Call: <code>lsra_checkpoint</code>	252
C.2.18	Function Call: <code>lsra_continue</code>	253
C.2.19	Function Call: <code>lsra_restart</code>	253
C.2.20	Data Member: <code>lsra_tv_queue_support</code>	253
<b>APPENDIX D: MPI COLLECTIVE ALGORITHMS COMPONENT INTERFACE</b>		<b>255</b>
D.1	Services Provided by the <code>coll</code> Component Framework	255
D.1.1	Header Files	255
D.1.2	Communication During Initialization	256
D.1.3	<b>BLK*</b> Constants	256
D.1.4	Utility Function: <code>lam_mkcoll()</code>	256
D.1.5	Utility Function: <code>lam_mkpt()</code>	256
D.1.6	Utility Function: <code>lam_err_comm()</code>	257
D.1.7	Datatype Accessor Functions	257
D.2	<code>coll</code> Component Framework Module API	260
D.2.1	Layered Point-to-Point Implementations	260
D.2.2	Hierarchical Implementations (Sub-Communicators)	264
D.2.3	Intracommunicators and Intercommunicators	267
D.2.4	Checkpoint / Restart Functionality	267
D.2.5	MPI Exceptions and Return Values	269
D.2.6	Data Member: <code>lsc_meta_info</code>	270
D.2.7	Function Call: <code>lsc_thread_query</code>	270
D.2.8	Function Call: <code>lsc_query</code>	270
D.2.9	Data Member: <code>lsc_has_checkpoint</code>	271
D.2.10	Function Call: <code>lsca_init</code>	271
D.2.11	Function Call: <code>lsca_finalize</code>	272
D.2.12	Function Call: <code>lsca_checkpoint</code>	272
D.2.13	Function Call: <code>lsca_continue</code>	273
D.2.14	Function Call: <code>lsca_restart</code>	273
D.2.15	Function Call: <code>lsca_interrupt</code>	274
D.2.16	Function Call: <code>lsca_allgather</code>	274
D.2.17	Function Call: <code>lsca_allgatherv</code>	275
D.2.18	Function Call: <code>lsca_allreduce</code>	275
D.2.19	Function Call: <code>lsca_alltoall</code>	275
D.2.20	Function Call: <code>lsca_alltoallv</code>	276
D.2.21	Function Call: <code>lsca_alltoallw</code>	276
D.2.22	Function Call: <code>lsca_barrier</code>	277
D.2.23	Data Member: <code>int lsca_bcast_optimization</code>	277
D.2.24	Function Call: <code>lsca_bcast</code>	277
D.2.25	Function Call: <code>lsca_exscan</code>	278
D.2.26	Function Call: <code>lsca_gather</code>	278
D.2.27	Function Call: <code>lsca_gatherv</code>	278
D.2.28	Data Member: <code>int lsca_reduce_optimization</code>	279

D.2.29	Function Call: <code>lsca_reduce</code> . . . . .	279
D.2.30	Function Call: <code>lsca_reduce_scatter</code> . . . . .	279
D.2.31	Function Call: <code>lsca_scan</code> . . . . .	280
D.2.32	Function Call: <code>lsca_scatter</code> . . . . .	280
D.2.33	Function Call: <code>lsca_scatterv</code> . . . . .	280

APPENDIX E: PARALLEL CHECKPOINT / RESTART COMPONENT INTER-  
FACE . . . . . 282

E.1	Services Provided by the <code>cr</code> Component Framework . . . . .	282
E.1.1	Header Files . . . . .	282
E.1.2	Module Selection Mechanism . . . . .	283
E.1.3	Internal Type: <code>lam_ssi_crmpi_base_handler_state_t</code> . . . . .	283
E.1.4	Global Variable: <code>lam_ssi_crmpi_base_handler_state</code> . . . . .	284
E.1.5	<code>crlam</code> Utility Function: <code>lam_ssi_crlam_base_checkpoint()</code> . . . . .	284
E.1.6	<code>crlam</code> Utility Function: <code>lam_ssi_crlam_base_continue()</code> . . . . .	284
E.1.7	<code>crlam</code> Utility Function: <code>lam_ssi_crlam_base_restart()</code> . . . . .	285
E.1.8	<code>crlam</code> Utility Function: <code>lam_ssi_crlam_base_create_restart_-- argv()</code> . . . . .	285
E.1.9	<code>crlam</code> Utility Function: <code>lam_ssi_crlam_base_do_exec()</code> . . . . .	285
E.1.10	<code>crmpi</code> Utility Function: <code>lam_ssi_crmpi_base_checkpoint()</code> . . . . .	286
E.1.11	<code>crmpi</code> Utility Function: <code>lam_ssi_crmpi_base_continue()</code> . . . . .	287
E.1.12	<code>crmpi</code> Utility Function: <code>lam_ssi_crmpi_base_restart()</code> . . . . .	287
E.2	<code>crlam</code> Component Framework Module API . . . . .	288
E.2.1	Data Member: <code>lscrl_meta_info</code> . . . . .	288
E.2.2	Function Call: <code>lscrl_query</code> . . . . .	288
E.2.3	Function Call: <code>lscrla_checkpoint</code> . . . . .	290
E.2.4	Function Call: <code>lscrla_continue</code> . . . . .	291
E.2.5	Function Call: <code>lscrla_disable_checkpoint</code> . . . . .	291
E.2.6	Function Call: <code>lscrla_enable_checkpoint</code> . . . . .	292
E.2.7	Function Call: <code>lscrla_finalize</code> . . . . .	292
E.2.8	Function Call: <code>lscrla_init</code> . . . . .	292
E.2.9	Function Call: <code>lscrla_restart</code> . . . . .	293
E.2.10	Function Call: <code>lscrla_lamcheckpoint</code> . . . . .	294
E.2.11	Function Call: <code>lscrla_lamrestart</code> . . . . .	294
E.3	<code>crmpi</code> Component Framework Module API . . . . .	295
E.3.1	Data Member: <code>lscrm_meta_info</code> . . . . .	296
E.3.2	Function Call: <code>lscrm_query</code> . . . . .	296
E.3.3	Function Call: <code>lscрма_init</code> . . . . .	296
E.3.4	Function Call: <code>lscрма_finalize</code> . . . . .	297
E.3.5	Function Call: <code>lscрма_app_suspend</code> . . . . .	297

BIBLIOGRAPHY . . . . .	299
------------------------	-----

## FIGURES

2.1	High-level architecture of LAM/MPI. . . . .	13
2.2	LAM/MPI as a component system architecture containing component frameworks and modules. . . . .	26
3.1	SSI is the top tier that binds together the individual component frameworks, and indirectly the modules that they contain. . . . .	35
3.2	Sample outputs from the <code>laminfo</code> command. The first command shows all available <code>rpi</code> modules and their associated versioning data; the second command shows the <code>sysv rpi</code> module parameters and their default values. . . . .	44
4.1	Sample parallel applications started with <code>rsh</code> (or <code>ssh</code> ). . . . .	47
4.2	Booting the LAM run-time environment using <code>rsh</code> or <code>ssh</code> . . . . .	51
4.3	Starting a parallel application using PBS's TM interface ( <code>tm_spawn()</code> is the interface function for starting processes under PBS's control). PBS is able to track resource utilization for the entire application. . . . .	52
4.4	Example architecture of PBS over four nodes. There is one instance each of the PBS server and scheduler, and per-node instances of the MOM. . . . .	53
4.5	Booting the LAM run-time environment using the PBS TM interface. . . . .	54
4.6	Sample boot schema for the <code>globus boot</code> component framework. Each line can specify a different <code>prefix</code> and CPU count. . . . .	61
4.7	Execution time for <code>lamboot</code> in seconds using <code>ssh</code> , <code>rsh</code> , and TM on the AVIDD-B cluster. . . . .	72
5.1	An <code>MPI_Comm</code> contains an <code>MPI_Group</code> . The <code>MPI_Group</code> contains a set of references to entries in the process list. Each process entry contains a <code>GPS</code> . . . . .	78
5.2	State progression of an MPI request. After a request is created, it is automatically put in the <code>init</code> state. Note that upon finishing the <code>done</code> state, non-persistent requests are destroyed, but persistent requests are moved back to the <code>init</code> state. . . . .	82
5.3	MPI message passing using the <code>lamd rpi</code> module in two scenarios: (a) when the source process is on node 0 and the destination process is on node 1, and (b) when the source and destination processes are on the same node. . . . .	86
5.4	Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph compares the performance of raw TCP, the <code>tcp rpi</code> module (LAM 7), and the TCP RPI implementation (LAM 6). . . . .	95

5.5	Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph shows the effect of increasing the eager/rendesvous message to increase performance of messages between 128 KB and 1 MB in the tcp rpi module. . . . .	96
5.6	Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph shows the absolute bandwidth difference between the the tcp rpi module (LAM 7) and the TCP RPI implementation (LAM 6). . . . .	97
5.7	Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph shows the percentage bandwidth difference between the the tcp rpi module (LAM 7) and the TCP RPI implementation (LAM 6). . . . .	98
5.8	Ping-pong bandwidth measurements on one AVIDD-B node using shared memory for communication. This graph compares the performance of the sysv rpi module (LAM 7) to the SYSV RPI implementation (LAM 6). . .	99
5.9	Ping-pong bandwidth measurements on one AVIDD-B node using shared memory for communication. This graph compares the performance of the usysv rpi module (LAM 7) to the USYSV RPI implementation (LAM 6). . .	99
5.10	Ping-pong bandwidth measurements between two AVIDD-B nodes using Myrinet for communication. This graph compares the performance of raw GM and the gm rpi module (LAM 7) – there was no GM RPI implementation in LAM 6. . . . .	100
5.11	Ping-pong bandwidth measurements between two AVIDD-B nodes using Myrinet for communication. This graph compares the percentage of bandwidth difference between raw GM and the gm rpi module (LAM 7). . . . .	101
6.1	Four processes are distributed across two nodes. MPI_COMM_WORLD contains all four processes. Two sub-communicators (shown vertically) each contain the two processes local to their respective nodes. One “bridge” communicator (shown horizontally) contains a representative process from each node. . . . .	107
6.2	Phases in the life of a coll module. When a module is created, the selection process determines which coll module will be used. The selected module is then initialized and is ready for normal use (i.e., invoking collectives) and checkpoint/restart services. When the communicator is destroyed, the coll module finalizes itself for that scope. . . . .	108
6.3	MagPIe algorithm for broadcast from process 0. Process 0 sends to its peer on the remote node (process 3). Each then do a local broadcast to the remaining processes on their nodes (processes 1 and 2, and processes 4 and 5, respectively). . . . .	114
6.4	Pseudocode showing the MPI_BCAST implementation using a hierarchical implementation approach. bridge_root, bridge_comm, local_root, and local_comm are all calculated and initialized during the per-communicator initialization and are cached on the communicator. . . . .	114
6.5	Wall-clock execution time for MPI_ALLREDUCE on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the sysv rpi module and the SYSV RPI implementation. . . . .	120

6.6	Wall-clock execution time for <code>MPI_ALLREDUCE</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>tcp rpi</code> module and the TCP RPI implementation. . . . .	121
6.7	Wall-clock execution time for <code>MPI_ALLREDUCE</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>usysv rpi</code> module and the USYSV RPI implementation. . . . .	121
6.8	Wall-clock execution time for <code>MPI_ALLTOALL</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>sysv rpi</code> module and the SYSV RPI implementation. . . . .	122
6.9	Wall-clock execution time for <code>MPI_ALLTOALL</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>tcp rpi</code> module and the TCP RPI implementation. . . . .	122
6.10	Wall-clock execution time for <code>MPI_ALLTOALL</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>usysv rpi</code> module and the USYSV RPI implementation. . . . .	123
6.11	Wall-clock execution time for <code>MPI_BARRIER</code> on the AVIDD-B cluster on varying numbers of processes using the <code>sysv rpi</code> module and the SYSV RPI implementation. . . . .	123
6.12	Wall-clock execution time for <code>MPI_BARRIER</code> on the AVIDD-B cluster on varying numbers of processes using the <code>tcp rpi</code> module and the TCP RPI implementation. . . . .	124
6.13	Wall-clock execution time for <code>MPI_BARRIER</code> on the AVIDD-B cluster on varying numbers of processes using the <code>usysv rpi</code> module and the USYSV RPI implementation. . . . .	124
6.14	Wall-clock execution time for <code>MPI_ALLREDUCE</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>sysv rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smc_coll</code> modules. . . . .	126
6.15	Wall-clock execution time for <code>MPI_ALLREDUCE</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>tcp rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smc_coll</code> modules. . . . .	126
6.16	Wall-clock execution time for <code>MPI_ALLREDUCE</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>usysv rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smc_coll</code> modules. . . . .	127
6.17	Wall-clock execution time for <code>MPI_BARRIER</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>sysv rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smc_coll</code> modules. . . . .	128
6.18	Wall-clock execution time for <code>MPI_BARRIER</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>tcp rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smc_coll</code> modules. . . . .	128
6.19	Wall-clock execution time for <code>MPI_BARRIER</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>usysv rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smc_coll</code> modules. . . . .	129
6.20	Wall-clock execution time for <code>MPI_BCAST</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>sysv rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smc_coll</code> modules. . . . .	129

6.21	Wall-clock execution time for <code>MPI_BCAST</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>tcp rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smp coll</code> modules. . . .	130
6.22	Wall-clock execution time for <code>MPI_BCAST</code> on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the <code>usysv rpi</code> . This graph shows a comparison of the <code>lam_basic</code> and <code>smp coll</code> modules.	130
7.1	A message-passing system consisting of 3 processes. The blocks on each line represent when the checkpoint was taken; the line drawn between them establishes the global state. (a) shows an example of a consistent global state where message $m_1$ is recorded as having been sent by process $P_0$ but not yet received by process $P_1$ , and (b) shows an example of an inconsistent global state in which message $m_2$ is recorded as having been received by $P_2$ but not yet sent by $P_1$ [35]. . . . .	133
7.2	Three phases of checkpoint services in LAM/MPI parallel applications. .	138
7.3	Sequence of events in the checkpoint phase. . . . .	139
7.4	Sequence of events when the application thread is executing outside the MPI library when a checkpoint request arrives. Although the checkpoint/continue phases are shown here, the same sequence occurs in the restart phase (albeit in a different process). . . . .	141
7.5	Sequence of events when the application thread is blocking in the MPI library when a checkpoint request arrives. Although the checkpoint/continue phases are shown here, the same sequence occurs in the restart phase (albeit in a different process). . . . .	142
7.6	Sequence of events in the continue phase. . . . .	143
7.7	Sequence of events in the restart phase. . . . .	144
7.8	Draining TCP sockets before checkpoint: (a) Processes A and B exchange the sent/received byte counter information using LAM's out-of-band communication system. (b) Processes A and B receive data from the in-band channel until their counters match what was received in (a). .	146
7.9	Template for BLCR callback functions. The state of the entire process (including the callback's execution) is saved during the <code>cr_checkpoint()</code> call. The return value from <code>cr_checkpoint()</code> indicates whether the process continued after the checkpoint or was started in a new process. . .	148
7.10	The <code>bcr</code> module in <code>mpirun</code> propagates the checkpoint request by using LAM run-time environment services to launch a <code>cr_checkpoint</code> for every MPI process. . . . .	149
7.11	Wall clock execution time for checkpointing serial processes of varying sizes using BLCR. . . . .	155
7.12	NetPIPE throughput on AVIDD-B cluster of the <code>tcp</code> module compared to the <code>crtcp</code> module, both with and without checkpointing support enabled. .	156
7.13	Depiction of the message passing during a checkpoint, including the artificial "sync message" inserted solely for measuring the magnitude of LAM overhead. The upper bound of LAM overhead is therefore the time interval between (a) and (b). . . . .	159
7.14	Wall clock time of LAM overhead to checkpoint a parallel job with differing numbers of processes. Two outputs are shown; one with individual process sizes of approximately 1 MB, the other with 256 MB processes. .	160

8.1	LAM/MPI is a component system architecture that manages multiple different component frameworks (or “types”); zero or more modules may be available from a given type. This figure depicts a component type with $N$ modules, and shows <code>mpirun</code> passing in run-time parameters to module $B$ .	164
A.1	The <code>configure.params</code> file for the <code>tcp rpi</code> module.	176
A.2	Sample version file for a module. This file will resolve to the full version string “7.1.2cvs1”.	177
A.3	Sample top-level <code>Makefile.am</code> for building the <code>tcp rpi</code> module. Logic based on the <code>LAM_BUILD_LOADABLE_MODULE</code> decides whether to build the module statically or dynamically.	186
A.4	Sample <code>AM_CPPFLAGS</code> required to find the LAM header files. Note that the build directory is explicitly included in order to support <code>VPATH</code> builds properly, even though it will be redundant in non- <code>VPATH</code> builds.	187
A.5	Definition of the <code>lam_ssi_t</code> type.	189
A.6	Definition of the <code>lam_ssi_rpi_t</code> type.	192
A.7	Definition of the <code>lam_ssi_rpi_t</code> type for the <code>tcp rpi</code> module. Note the <code>NULL</code> used for the close function; the <code>tcp</code> module does not have a close function.	193
A.8	Functions for registering and looking up module parameters.	194
B.1	<code>struct lamnode</code> definition.	196
B.2	Example boot schema file.	197
B.3	<code>struct psc</code> definition.	198
B.4	<code>lam_ssi_boot_proc_t</code> enumerated type definition.	198
B.5	Abbreviated sample <code>start_rte_proc()</code> function.	201
B.6	The <code>boot</code> type for exporting the initialization and finalization API function pointers.	210
B.7	The <code>boot</code> type for exporting the main action API function pointers.	211
C.1	<code>struct _gps</code> : GPS type; unique process identification in a LAM universe. It is mainly used for MPI process identification and LAM out-of-band messaging.	223
C.2	<code>struct _proc</code> : Process entry.	224
C.3	<code>struct _req</code> : Underlying structure for <code>MPI_Request</code> , part 1.	226
C.4	<code>struct _req</code> : Underlying structure for <code>MPI_Request</code> , part 2.	227
C.5	<code>struct _com</code> : Underlying structure for <code>MPI_Comm</code> .	232
C.6	<code>struct _group</code> : Underlying structure for <code>MPI_Group</code> .	234
C.7	<code>struct _status</code> : Underlying structure for <code>MPI_Status</code> .	234
C.8	<code>struct _lam_ssi_rpi_envl</code> : General structure for envelopes.	235
C.9	<code>struct _lam_ssi_rpi_cbuf_msg</code> : Unexpected message bodies.	237
C.10	<code>struct lam_ssi_rpi_1.1.0</code> : The <code>rpi</code> basic type for exporting the module meta information and initial query / initialization function pointers.	241
C.11	<code>struct lam_ssi_rpi_actions_1.1.0</code> : The <code>rpi</code> type for exporting API function pointers.	242
D.1	Sample <code>MPI_BARRIER</code> implementation.	258

D.2	Sample MPI_REDUCE implementation. Note the Fortran flag on the <code>op</code> variable than indicates a different calling convention for the reduction function. In this case, the reduction function is in Fortran and therefore the datatype needs to be passed as its Fortran integer handle, not its C pointer handle. . . . .	259
D.3	The <code>coll</code> type for exporting the basic API function pointers and flags. . . .	261
D.4	The <code>coll</code> type for exporting the majority of the collective API function pointers (part 1 of 2). . . . .	262
D.5	The <code>coll</code> type for exporting the majority of the collective API function pointers (part 2 of 2). . . . .	263
D.6	Sample code showing conditional profiling build. . . . .	265
D.7	Sample code creating a hidden communicator. . . . .	266
D.8	Sample MPI exception in a back-end collective implementation. . . . .	269
E.1	The <code>lam_ssi_crmpi_base_handler_state_t</code> type and its possible values. . . . .	283
E.2	<code>lam_ssi_crlam_1_1_0_t</code> : The <code>crlam</code> basic type for exporting the module meta information and initial query function pointer. . . . .	288
E.3	<code>lam_ssi_crlam_actions_1_1_0_t</code> : The <code>crlam</code> type for exporting API function pointers. . . . .	289
E.4	<code>struct lam_ssi_crmpi_1_0_0</code> : The <code>crmpi</code> basic type for exporting the module meta information and function pointers. . . . .	295
E.5	<code>struct lam_ssi_crmpi_actions_1_0_0</code> : The <code>crmpi</code> type for exporting API function pointers. . . . .	295

## TABLES

1.1	Partial listing of MPICH derivatives and related packages . . . . .	6
1.2	Total line counts for each part of the contribution in this work . . . . .	10
2.1	Partial list of Linux distributions, open source BSD distributions, and clustering projects that are known to include LAM/MPI . . . . .	16
2.2	LAM/MPI's formally supported platforms . . . . .	19
2.3	Line counts for various types of files in the LAM/MPI code base . . . . .	20
2.4	LAM/MPI's formally supported compilers . . . . .	23
4.1	Summary of Run-Time Environment Results . . . . .	67
4.2	Description of the University of Pennsylvania Liniac testbed cluster . . . . .	67
4.3	Description of the Indiana University Computer Science <code>bitternut</code> and <code>sawtooth</code> nodes . . . . .	68
4.4	Description of the Indiana University AVIDD-B and Open Systems Lab Thumb clusters . . . . .	70
4.5	Accounting information from PBS on the Thumb cluster . . . . .	73
5.1	Test code coverage of <code>rpi</code> modules . . . . .	94
6.1	Summary of MPI Collective results on 1 node . . . . .	117
6.2	Summary of MPI Collective results for $2^M$ nodes . . . . .	117
6.3	Summary of results for $(2^M + X)$ nodes . . . . .	118
6.4	Test code coverage of <code>coll</code> modules . . . . .	119
7.1	Collective MPI functions tested with checkpoint / restart functionality . . . . .	152
7.2	Point-to-point MPI functions tested with checkpoint / restart functionality . . . . .	152
7.3	Description of the Indiana University Computer Science Thor cluster . . . . .	153
7.4	Checkpoint / restart overhead measurements in LAM/MPI on the Thor cluster . . . . .	157
7.5	Wall-clock execution time of the LU NAS parallel benchmark with checkpoints being taken at different frequencies . . . . .	160
A.1	Component <code>configure</code> script output variables for the LAM library . . . . .	182
A.2	Component <code>configure</code> script output variables for the MPI library . . . . .	182
A.3	Component <code>configure</code> script output variables for user MPI applications . . . . .	183

## ACKNOWLEDGMENTS

This work was supported by a grant from the Lilly Endowment, National Science Foundation grants 0116050, EIA-0202048, and ANI-0330620, the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. This work was performed using computational facilities at Indiana University, the University of Toronto, the University of Pennsylvania, and the College of William and Mary. The College of William and Mary's resources were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

## CHAPTER 1

### INTRODUCTION

The Message Passing Interface (MPI) is the *de facto* standard for message passing parallel programming of large-scale distributed systems [48, 52, 57, 58, 91, 120]. Implementations of MPI comprise the middleware layer for a wide variety of high-performance computing environments. Providing networked, message passing services to parallel applications, MPI implementations therefore play a critical role in overall performance as well as functionality delivered to the application.

MPI implementations are available from a variety of sources, ranging from research projects to production quality products. Research-oriented MPI implementations can be found at many academic institutions and are typically targeted at studying specific issues in parallel computing. Many such implementations are intended only for research purposes; they are neither designed to be utilized by real users, nor are they necessarily complete implementations of the MPI-1 and MPI-2 standards. Two notable open source packages that provide a balance of cutting-edge research and a production-quality implementation are LAM/MPI [19, 127]<sup>1</sup> and MPICH [51, 59]. Other production-quality MPI implementations are distributed by high-speed networking hardware vendors, cluster resellers, and independent software vendors (ISVs) – many of which are derived from LAM/MPI and/or MPICH.

---

<sup>1</sup>The word “LAM” used to be an acronym for “Local Area Multicomputer,” a name which has long since fallen into disuse. “LAM” is now used as the project’s name, not as an acronym. The standalone name “LAM” is frequently used as an abbreviation of “LAM/MPI.”

The MPI standard is so large that any implementation of it necessarily involves an extensive set of design and implementation choices. Parallel researchers therefore need platforms to develop, experiment, test, and deploy their work. Common approaches include either implementing a subset of MPI functionality sufficient to support their research or modifying an existing open-source MPI implementation. Implementing MPI, or even a subset of MPI, is such a large task that it frequently cannot be justified when the desired goal is to develop and experiment with only a small portion of MPI functionality. Similarly, the MPI standard is so large and diverse that modifying an existing implementation, even if it is well-abstracted and logically constructed, is an unwieldy task (at best) for external developers attempting to understand its source code. Both methods available to third party researchers therefore involve enormous logistical and technical difficulties that must be solved before any development can be started. Such issues typically require a large initial time investment to solve, which may be prohibitive.

This is particularly evident when developing emerging high-performance hardware platforms. Since the majority of high-performance computing (HPC) codes are written with MPI, it is vital to establish an MPI implementation for the new platform as quickly as possible. This immediately enables not only real-world applications to utilize the new platform, but also many established MPI benchmarks and testing tools. This has the side-effect of feeding back into the design cycle; user applications and well-known benchmarks can be used as test suites to stress test the new platform.

This work shows that the use of component abstractions within an MPI implementation addresses these issues. The use of components allows small, independent modules to implement discrete, well-defined functionality that are inherently easier to maintain than large, monolithic architectures. For example, third parties need only learn about the components relevant to their work, not the rest of the MPI implementation. Additionally, since modules are designed to be deployable units, not only can third parties independently de-

velop and distribute their own modules, sets of modules can be composed in unique ways by the end-user and/or parallel environment at run-time to form an MPI implementation that is specifically tailored to the environment in which it is running.

This approach shows the value of using component-based programming in scientific computing: complex functionality can be broken down into its most fundamental abstractions and implemented as part of large-scale software systems with no loss in performance. The use of well-designed components in an MPI implementation significantly lowers the overhead required by third party researchers to conduct parallel research. For example, if new algorithms for MPI collective functions could be implemented as a middleware “plug-in,” the focus of the work would be on the collective algorithms – not on the logistical and technical details surrounding the collective algorithms. Additionally, since a complete MPI implementation is used as the framework for the research, real-world MPI applications can be used for testing and verification of the module. Even unmodified legacy MPI applications can be used; its calls to `MPI_BARRIER`, for example, will automatically be routed to the new plug-in.

Simply put, having the ability to easily and selectively replace a variety of small, specific portions of an MPI implementation at run-time empowers third party research in a way that has not previously been available. This work explores these concepts both in theory and in implementation, and demonstrates that abstractions required to support it do not negatively impact performance.

The LAM implementation of MPI has therefore been re-architected to utilize a component system architecture consisting of four component frameworks and a meta framework that ties them together. These component frameworks were designed from analysis of major functionality categories from prior monolithic implementations of LAM/MPI: run-time environment startup, MPI point-to-point communication, and MPI collective communication. The designs of these frameworks were validated by creating multiple

modules for each. Some modules are “component-ized” versions of prior LAM/MPI functionality while others represent new capabilities largely made possible by the component architecture.

Coordinated asynchronous checkpoint/restart of parallel MPI applications is entirely new functionality; it was implemented as the fourth component framework. Its component specifications were designed from an analysis of the actions required by several back-end checkpoint systems. Two different checkpoint/restart modules have been implemented, verifying the approach taken in its framework.

The frameworks were specifically designed to be lightweight and efficient; negligible overhead is introduced by the additional abstractions. A meta framework provides an additional layer of commonality and interaction; it ties the four component frameworks together and allows modules in different frameworks to interact with each other. The meta framework is named the System Services Interface (SSI) of LAM/MPI. The SSI manages the modules in all four component frameworks. It allows multiple shared library modules to be available at run-time, selectively loading the ones that are needed. This has two important side effects: 1) the choice of which module(s) to use is performed at run-time, and 2) since modules exist as standalone shared libraries, an MPI application can use any module without recompiling/relinking.

The end result is an MPI implementation that is inherently modular, has high message-passing performance, publishes abstraction and interface boundaries, and is significantly easier to develop, maintain, and use as a vehicle for research than its monolithic predecessors.

### 1.1 Preventing Code Base Fracturing

The MPICH project from Argonne National Labs was developed as a reference implementation while the MPI-1 standard was being written. It provided valuable insights

and sanity checks on the ideas proposed for the MPI-1 document. By providing stable, usable software immediately available upon the completion of MPI-1, it was also therefore instrumental in advocating the use of MPI to both end users and vendors. As a direct result, MPICH became a standard implementation of MPI; many vendors and ISVs have used MPICH as the basis for their MPI implementation in custom parallel hardware and software environments.

Table 1.1 is a list of several recent MPICH-related and derivative projects; other MPICH-derived projects exist (including vendor MPI implementations) but are not listed for the sake of brevity. The Table lists each package as a “Patch” if it is distributed as a patch to the main MPICH distribution [55], “Full” if it includes a full MPICH distribution, or “Add on” if the software is additional functionality and is not compiled as part of MPICH.

While the number of MPICH-derivative works is a testimony to the success of the MPICH project, it presents practical difficulties for researchers/developers, users and system administrators, and ISVs. Developers need to keep up with MPICH releases in order to keep their software stable. For example, Myricom releases a new version of MPICH-GM (supporting high speed communication over Myrinet networks) to accompany every major MPICH release, costing developer time and resources. More specifically, each project – especially the “full” distributions – are a fork in the MPICH development tree. Significant time and effort is required to merge in the changes from each new MPICH release.

Each package is also mutually exclusive from the others; some cannot be combined without a considerable amount of additional work. For example, the MPICH-GM product cannot be easily extended to include the checkpointing work from the MPICH-V project to produce MPI programs that run natively over Myrinet networks with checkpointing support. Among the packages that can be combined (e.g., packages that add a entirely

TABLE 1.1

Partial listing of MPICH derivatives and related packages

Project Name	Distribution	Description
MP-MPICH	Full	Multi-platform MPICH, providing drivers for the SCALI network interconnect (Theinsch-Westfalische Technische Hochschule, Aachen, Germany)
MPI/GAMMA	Patch	MPI over the Genoa Active Message Machine network (University of Genova, Italy)
MPICH-g2	Patch	MPI for Globus/Grid environments (Northern Illinois University)
MPICH-GM, MPICH-MX	Full	MPICH tuned for Myrinet networks (Myricom)
MPICH NT	Full	Support for Microsoft Windows platforms (Argonne National Labs)
MPICH-SCore	Full	Support for SCore software and hardware environments (PC Cluster Consortium)
MPICH-V [16]	Patch	Three different versions of checkpointing support for MPICH (Paris-Sud University, Paris, France)
mpiexec	Add-on	Support for MPICH to run in Portable Batch Scheduler (PBS) environments (Ohio Supercomputing Center)
MVAPICH, MVAPICH2, MIBAPICH	Patch	Three different versions of patches to MPICH and MPICH2 for support of Infiniband networks (Ohio State University)
MVICH	Patch	Support for VIA networks (Lawrence Berkeley Labs)
Open MOSIX	Patch	Add support for Open MOSIX clusters to MPICH
Quadrics MPI	Full	MPI for the Quadrics network interconnect (Quadrics)
Scyld MPI	Full	Bootstrapping to start MPICH jobs on BProc clusters (the Scyld Computing Corporation)
Unnamed	Patch	Incomplete port of MPICH to Cygwin environments (Institute for High Performance Computing and Databases, St. Petersburg, Russia)

new, non-conflicting directory into the MPICH source tree [55]), the MPICH architecture only allows building one communications device at a time. So even if the differences between MPICH and MPICH-GM could somehow be added to the MPICH-V project, the result would still not yield checkpointable MPI programs running over Myrinet.

Another problem is that users and/or system administrators must install each package separately, and then ensure to use the correct installation when compiling and running their MPI applications. This can be a significant logistical challenge. Additionally, since some of the projects listed in Table 1.1 originally forked from different versions of MPICH, user MPI application behavior can vary depending on which package is used. ISVs need to support every MPI implementation that their customers use (which may include different versions of the same implementation – commercial Quality Assurance procedures typically require separate testing on each). In particular, for ISVs that do not distribute source code, the management of binary applications for potentially multiple versions of each MPI implementation can be cost-prohibitive.

Additional conclusions that can be drawn from Table 1.1 (which is admittedly abbreviated – there are additional MPICH-related and MPICH-derived projects not listed):

- There are researchers, developers, and vendors who are leveraging their work into an MPI implementation.
- By adapting an existing MPI implementation, these researchers, developers, and vendors clearly do not want to write their own MPI implementation.

Developers are therefore *willing to use MPI implementation tools*, but until this point, have only had the option of either using/adapting MPICH or starting a new MPI implementation. Of the two, using the infrastructure in MPICH is clearly a better choice, but is still quite limiting.

MPICH and its derivatives are not the only MPI implementations available; there are many independent implementations available, some of which were either forked or strongly influenced by LAM/MPI. Hence, the current state of the art is users and develop-

ers utilizing multiple MPI implementations, none of which are compatible, all of which require separate installation, and the combination of which present significant logistical challenges for system administrators and software distributors.

This dissertation shows that the use of components as an MPI implementation tool fulfills the requirements of third party developers and solves many of the problems cited above. As will be discussed later in this document, forks from the main LAM code base are unnecessary because modules can be deployed either in conjunction with or separately from LAM's main distribution. Additionally, LAM need only be installed once; all other functionality – even from third parties – can be added to an existing LAM installation. This simultaneous installation allows the composition of available modules (e.g., checkpointing support can be combined with network interconnects), which significantly cuts down on the logistics required for users, system administrators, and ISVs, yet still allows running parallel MPI applications natively in multiple environments.

## 1.2 Contribution

This work comprised of taking a stable version of LAM/MPI, v6.5, transforming it into a component system architecture, converting its existing functionality into a set of modules, and writing additional modules to introduce new functionality. This entailed the following high-level efforts:

- Factor the major functionality of LAM/MPI into individual component architectures. Three component architectures were identified: “boot” (run-time environment startup, Chapter 4), “rpi” (MPI point-to-point communications, Chapter 5), and “coll” (MPI collective communications, Chapter 6).
- Abstract the functionality required and design component interfaces for each architecture.
- Implement each of the component interfaces in their respective architectures and adapt the rest of the LAM code base to call them, as appropriate.
- Convert existing LAM functionality into component modules and write one or more new modules of each component type both to validate the interfaces as well as to add new functionality to LAM.

- Write technical documentation for all component interfaces, enabling third parties to write components (Appendices [A](#), [B](#), [C](#), [D](#), and [E](#)).
- Implement a component system architecture for loading and unloading modules, providing run-time module selection and parameter passing, and allowing interoperation between component architectures.

At the same time, parallel checkpoint/restart capabilities were added to LAM/MPI in a different research effort and were therefore absorbed into the “component-ization” process. The checkpoint restart abstractions resulted into two new component architectures: “`crlam`” and “`crmpi`” (frequently referred to collectively as “`cr`”, Chapter [7](#)).

Since this work was absorbed into an already-existing code base, it is difficult to quantify exactly how much code is specifically attributable to this effort. Table [1.2](#) provides line counts for each part of the contribution (some counts are approximate) to provide a rough order of magnitude of the scale of this work. “Accurate” means that the file and line count is fairly accurate in terms of unique contribution of this work. “Approximate” means that at least some portion of the line count was adapted from LAM/MPI 6.5. Note that the `boot` component architecture was jointly designed with Brian Barrett (who also implemented most of the `tm` module), and the `cr` component architecture was jointly designed with Sriram Sankaran (who also implemented most of the `blcr` modules).

This newly-architected system – including initial checkpoint/restart support – was first released as LAM/MPI v7.0 in the summer of 2003. Research and development has continued since that point; this dissertation contains additional work that is not contained in v7.0. The completed implementation of all ideas presented in this work is expected to be released as v7.1 in May 2004.

### 1.3 Document Organization

Chapter [2](#) provides a more in-depth background of MPI, the LAM implementation of MPI, and how LAM evolved from a monolithic MPI implementation to a component

TABLE 1.2

Total line counts for each part of the contribution in this work

Description	Type	File count	Line count
Component system architecture	Accurate	16	2,929
boot component architecture and modules	Accurate	64	8,709
coll component architecture and modules	Approximate	79	11,711
cr component architecture and modules	Accurate	43	5,158
rpi component architecture and modules	Approximate	189	51,798
$\LaTeX$ component architecture documentation	Accurate	57	11,470
Totals		448	91,775

system architecture. The SSI meta framework is discussed in Chapter 3. Chapters 4, 5, 6, and 7 discuss the four different component frameworks: run-time environment startup (named “boot”), MPI point-to-point communication (called “rpi,” an acronym for MPI Request Progression Interface), MPI collective communication (abbreviated “coll”), and parallel checkpoint/restart (named “cr”), respectively. Chapter 8 presents conclusions and lists future work directions.

Finally, Appendices A, B, C, D, and E provide technical and implementation details for each of the frameworks (the SSI, boot, rpi, coll, and cr, respectively). These Appendices are presented in the same order as their corresponding chapters.

## CHAPTER 2

### BACKGROUND

This chapter expands upon the brief Introduction chapter and provides a background for framing the work described in this dissertation. Concrete motivations are listed as to why this work is both relevant and necessary to advance the field of high-performance computing, specifically in the area of message passing library implementations.

Section 2.1 provides a short background on the Message Passing Interface (MPI), followed by an overview of the LAM implementation of MPI in Section 2.2. The evolution of LAM/MPI and motivation for this dissertation are provided in Section 2.3. Section 2.4 provides a summary of related works to this project. Finally, experimental setup for all results presented in this dissertation is described in Section 2.5.

#### 2.1 The Message Passing Interface (MPI)

There are two documents that specify the MPI application programmer’s interface (API): MPI-1 [91] and MPI-2 [48]. The MPI-1 document, released in 1994, specifies a basic parallel infrastructure as well as multiple modes of sends, receives, and collective operations. The interface is specified in C [76] and Fortran 77. One of its major design goals was to allow application portability between different kinds of parallel hardware; applications written on clusters of commodity hardware should be able to be run on traditional “big iron” machines with little or no change. The MPI-2 document, released

in 1997, specifies extensions to MPI-1, including one-sided communication operations, dynamic process control,<sup>1</sup> and C++ bindings for MPI.

Implementations of the MPI specification are available from a wide variety of sources. Parallel hardware vendors, high-speed networking vendors, and independent software vendors provide MPI implementations fine-tuned for their run-time environments. Government and academic researchers develop MPI implementations to study issues surrounding high-performance computing; such implementations range from research projects to production quality products.

## 2.2 The LAM Implementation of MPI

LAM/MPI is an open source, freely available implementation of the MPI standard [19, 127, 135]. It implements the complete MPI-1 standard and much of the MPI-2 standard, including dynamic process control, one-sided communication, C++ bindings, and MPI I/O (through ROMIO [138, 139]). The project is currently developed by the Open Systems Laboratory at Indiana University and is available under a BSD-like license.

Since its inception in 1989, the LAM project has grown into a mature code base that is both rich with features and efficient in its implementation, delivering both high performance and convenience to MPI users and developers. LAM/MPI is shipped by most Linux and BSD distributions, resold by Linux vendors and ISVs, and downloaded by users around the world. LAM enjoys a wide-ranging user base, active user mailing lists, and community participation.

### 2.2.1 Layered Design

Figure 2.1 shows that LAM/MPI is comprised of two main sub-systems: the LAM run-time environment and the MPI communications layer. For performance reasons, both

---

<sup>1</sup>MPI-1 was criticized for not including dynamic process capabilities, since prior systems such as the Parallel Virtual Machine (PVM) [32, 12, 13, 47] did. As such, dynamic process control was added in MPI-2.

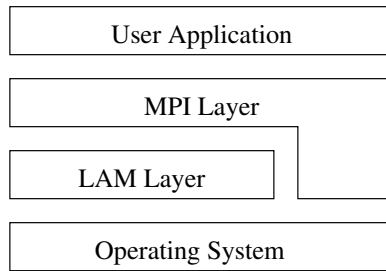


Figure 2.1. High-level architecture showing the user’s MPI application, the MPI layer, the LAM layer, and the underlying operating system.

layers interact directly with the operating system; requiring the MPI layer to utilize the LAM layer for all MPI communications would impose a significant overhead.

#### LAM Run-Time Environment Layer

The LAM layer includes both the LAM run-time environment and a companion library providing C API functions to interact with it. The run-time environment is based on user-level daemons; it provides services such as message passing, process control, remote file access, and I/O forwarding. Many of these services are utilized by MPI processes.

A user starts the LAM run-time environment with the `lamboot` command which, in turn, launches a LAM daemon on each node that will be used. After the run-time environment is established, parallel MPI applications can be run. When the user is finished, the `lamhalt` command is used to kill the LAM daemon on every node, terminating the run-time environment.

#### MPI Communications Layer

The MPI layer provides the MPI API functions as described in the MPI-1 and MPI-2 standards. This includes functions such as `MPI_INIT`, `MPI_SEND`, `MPI_RECV`, etc. Since MPI specifies a large number of interface functions, the logic required to support all the top-level API functions and effect the behavior described in the specification is

fairly complex. The MPI layer uses services from both the LAM layer and the operating system to perform this job.

### 2.2.2 Open Source Model

The LAM/MPI project embraces the open source model for software development [30]. Among the reasons for this is that some of the benefits of open source mirror that of successful research: peer review, wide-spread adoption, and contributions from third parties. This became particularly evident when LAM's source code repository was opened for anonymous, read-only access to the public in 1999; there has been a slow-but-steady stream of contributors – many from academic institutions around the world – who study the internals of LAM and offer critiques, enhancements, and bug fixes.

Active public mailing lists provide not only direct interaction between real-world users and the LAM/MPI research and development team, but also a valuable teaching tool for junior LAM developers. New students on the LAM project are exposed to real-world challenges and have to learn how to research unfamiliar topics and find correct solutions. Requiring junior developers to reply to user e-mail has proven to be an excellent educational device, and helps students bridge the gap from their undergraduate studies to their graduate career.

Although the exact membership change frequently, as of this writing, there are nearly 650 subscribers from around the world on the public LAM mailing lists. Subscribers post all manner of issues about LAM, including:

- Questions about MPI.
- Questions about LAM-specific behavior of MPI functionality.
- Questions about the internals of LAM's implementation.
- Suggestions to other users.
- Requests for features or support for new environments, platforms, and compilers.
- Bug reports.

- Fixes and additions to LAM.
- Challenges and requests for justification.

The last item is perhaps among the most relevant for the research aspects of the LAM project: continual peer review of LAM's methodology and source code. It is not uncommon for a subscriber to ask why a specific functionality was implemented in a given way, or how might a particular new feature be implemented. This forces the LAM group to justify its approach in a manner that satisfies the community.

The LAM developers also use the public lists to announce new versions, discuss new directions, and solicit feedback from real-world users. The availability of the LAM developers and open access to its source code provide the capability for research interaction that used to only be available at physical conferences and meetings. For example, it is not uncommon for users to post questions to the list which lead to lengthy off-list discussions about specific research topics. The two-way interaction ensures that the LAM project stays relevant to users, remains current with leading-edge research, and communicates the project's current status to the community.

For system administrators and end users, stable source code and binary packages are available from LAM's public web site (<http://www.lam-mpi.org/>) under a license similar to the well-known BSD license [96]. This allows LAM/MPI to be included in all major Linux and open source BSD distributions that include parallel and/or clustering [103, 121, 112] software (see Table 2.1), and also allows independent software vendors to resell LAM/MPI as part of their commercial packages. LAM's web site and downloadable software are mirrored at nine sites around the world, spanning Asia, North America, and Europe.

TABLE 2.1

Partial list of Linux distributions, open source BSD distributions, and clustering projects that are known to include LAM/MPI

Package type	Name
Linux distribution	Arch Yellow Dog / Black Lab cAos Debian Gentoo MSC.Linux Red Hat / Fedora Slackware SuSE
BSD distribution	FreeBSD NetBSD OpenBSD
Clustering package	Mandrake CLIC OSCAR [60] (and derivatives) ROCKS [101] Warewulf [82]

### 2.2.3 Software as a Research Artifact

“Success” in academic fields is typically defined by how many peer-reviewed journal and conference publications an individual has. This is certainly a fine measure of achievement, but it overlooks software as a significant artifact of research. Indeed, publishing peer-reviewed software can require just as much work – if not more – than authoring a textual publication.

In conjunction with a textual publication, released software offers significantly more validation of research than would otherwise be possible. Indeed, the software used to generate the results in a textual publication are an integral part of the scientific method and the process of discovery. This software should therefore also be peer-reviewed for applicability, technique, and correctness. This is unfortunately not always possible; for example, intellectual property issues may not permit open distribution of certain source or binary code used in research.

A committee of the National Research Council published a report in 1994 specifically addressing the fact that software can be a significant artifact of research [97]. The report recommends that tenure-track faculty who specialize in experimental computer science and engineering should be given due consideration for research artifacts other than traditional publications, such as significant software artifacts. The following is an excerpt from the report:

Universities should recognize that an experimentalist being considered for tenure or promotion may have ... nonstandard forms of dissemination (e.g., distribution of software artifacts) ... A judgment should be based on the presence of absence of the following:

- One or more computational impact-producing artifacts completed;
- Research results disseminated to and used by the community
- ...

In the spirit of “software is just as important and publications,” this dissertation is accompanied by a full software implementation of the concepts and ideas contained herein.

Version 7.1 of the LAM implementation of MPI is expected be released in May 2004.

#### 2.2.4 Software Engineering

High-quality software clearly must have a solid theoretical basis. Years of research have produced abstractions, models, and algorithms for all manner of different issues in computer science. However, the reduction of those ideas to software is, in itself, research. For example: theory can be reduced to software in many different ways; which is the best one? How is the implemented software certified against the original theory? Few academic software projects manage to project remain coherent and stable yet continue to grow and be a viable platform for new research. Even fewer manage to produce publicly-available software that can be used to replicate results. As such, both the reduction of theory to practice and the management of large software projects are open research questions in themselves.

LAM has confronted these issues and become not only a successful academic research project, but also a community-recognized leader in open source MPI implementations. Indeed, LAM/MPI is included in all major Linux and BSD distributions that include clustering support, is resold by ISVs with parallel products, and is downloaded from its main web site an average of 140 times a day. Two major reasons that LAM is so successful are: 1) great pains are taken to ensure that the software “just works” on as wide a variety of POSIX platforms as possible, and b) those wishing to conduct parallel computing research can reproduce published LAM results. Both of these reasons have wide-reaching effects in the day-to-day research put into the project as well as the management of the LAM code bas.

#### Configuration and Installation

The “it just works” philosophy and the reproducibility requirement demand that LAM be able to run on a wide variety of POSIX platforms. However, every POSIX platform

TABLE 2.2

LAM/MPI's formally supported platforms

Linux 2.4.x	OpenBSD 3.2, 3.3
Solaris 7, 8, 9	AIX 5.1
IRIX 6.5	OS X 10.2, 10.3

is slightly different – the LAM source code must be customized for each particular environment in which it will be compiled. As such, LAM includes a configuration process that has been extensively tested on many modern POSIX platforms. Table 2.2 lists the platforms that LAM is formally tested on.

LAM uses a GNU Autoconf-generated “`configure`” script for this purpose that is comprised of almost 9,400 lines of M4 preprocessor directives and shell script “source code.” Autoconf parses the shell script and M4 preprocessor code to generate the final `configure` script which is run by end-users. The resulting `configure` script customizes the LAM source code for a particular platform. It does this by:

- Searching for C, C++, and Fortran compilers.
- Searching for various characteristics about each compiler, such as datatype sizes and alignments.
- Searching for SSI components to configure, compile, and install.
- Testing for specific characteristics of the underlying operating system.

Note that for every test or action in the `configure` script, there is a corresponding reaction in the LAM source code. For example, if compiling on certain versions of AIX, specific kernel functions must be used for governing socket behavior (different than for all other POSIX platforms). As another example, LAM includes four entirely different implementations of file descriptor passing; which one a given system uses is determined by the `configure` script.

TABLE 2.3

Line counts for various types of files in the LAM/MPI code base

Line count	File type
9,391	M4, shell script, Perl
10,424	Automake source
211,832	C source code
10,487	C++ source code
33,005	C header files
10,963	L <sup>A</sup> T <sub>E</sub> X user-level documentation
11,195	Manual pages
297,297	Total

After LAM has been configured, it must be built and installed. LAM uses the GNU Automake tool to generate over 200 Makefiles that are used to build itself. A single, top-level “make all install” command will build and install all of LAM and its components.

### Coding Practices

LAM is a large software project, consisting of almost 200 directories and nearly 2,000 source code files; total line counts are listed in Table 2.3. While no complex software is ever completely free of bugs, the LAM development team follows standardized coding practices derived from the GNU Coding Standards [36] to both create commonality throughout the code base as well as enforce defensive coding practices. For example, all code in LAM must meet the following requirements:

- Code must compile without warnings on all compilers on all platforms with compiler “pickyness” options enabled. LAM’s `configure` script detects many compiler types automatically, and when being built by developers, automatically enables the correct compiler options to enable “picky” behavior. For example, when

compiling with the GNU compilers, the following flags are used: “-Wall -Wundef -Wno-long-long -Wno-long-double”.

- Code must function properly in 32 and 64 bit environments. LAM is regularly tested in both environments with various memory-checking debuggers and standardized test suites aimed at identifying data size problems.
- Resources must not be leaked. For example, all code must be certified as free of memory leaks and no detectable buffer overflows by memory-checking tools such as the Linux Valgrind package and the Solaris Forte bcheck debugger before it can be included in LAM.

All three conditions are regularly tested, both manually by developers and automatically by test scripts. Every night, a “snapshot” distribution is created from the latest revisions in the LAM source code repository. This snapshot is subjected to a full battery of compilation and run-time regression tests (some of which are described in later chapters). The results of this testing are e-mailed to the LAM development team. This quickly identifies a wide range of coding errors, such as failing to compile, breaking MPI functionality, etc. Developers quickly learn to thoroughly test their code before committing it to the repository for fear of having their bugs broadcast to the group the next morning.

Other development practices include:

- The LAM development team physically meets every week to discuss the status of all current projects. Face-to-face whiteboard discussions have proven invaluable as a team-building concept; no amount of e-mail or instant messenger traffic can replace the high-bandwidth information exchange that can occur in person. Additionally, the weekly meetings allow the group to stay focused on common goals and ensure that all projects are moving in the same general development direction. Even though each developer is typically only working on a small portion of LAM, everyone has generalized knowledge of the entire project.
- Source code is tracked through the Subversion version control system. Code changes, additions, deletions, branches and releases – all are coordinated through through the source code repository. History of code is maintained, and all commits to the repository are accompanied by a meaningful log message. The ability to look at the genealogy of code, particularly when trying to understand a specific segment, can be extraordinarily useful in the comprehension process.
- Commits to the LAM code repository are e-mailed to all developers. This not only allows an informal peer-review process, it actually encourages it. In practice, developers actually *do* read other developers’ code, and when appropriate, make comments, point out mistakes, etc. Peer-reviewed code commits – even though an

information mechanism – has helped LAM’s overall code quality. Additionally, this mechanism keeps all developers generally informed of what changes are being made to the code.

- All bugs and issues are logged to a bug tracking system. Bugs and issues used to be tracked in people’s memories, or when fortunate, in text files. Using a formal bug tracker has proven to be invaluable: long-standing bugs are tracked, new ideas are not forgotten, and bugs are assigned to specific developers. Issue tracking even allows the breakdown of complex new tasks (not necessarily bugs) into small pieces that can be individually tracked. The entire process is available to all developers; every developer can query the status of a bug to see its current status.
- New functionality in LAM must be accompanied with corresponding regressions tests proving that it is working properly. These tests are added to the nightly automated testing harness; any future code changes will be checked against these tests. Hence, the nightly builds use an ever-increasing set of tests to provide correctness checks on all code in the LAM repository.
- Before releasing, a series of “release candidate” tarballs are created and tested thoroughly on every supported platform. All of the above criteria are re-certified (no compiler warnings, memory-debugger clean, 32 and 64 bit clean, etc.) on every platform before it can be released.

These concepts are not revolutionary in themselves; indeed, they are common in commercial circles. LAM simply uses these “best practices” and management tools together in an attempt to keep a coherent and stable code base. This is absolutely critical for both the “it just works” philosophy and the reproducibility requirement. These practices contributed to the successful release of the first version of LAM’s component architecture (version 7.0) in the summer of 2003, supporting all the platforms listed in Table 2.2 and all the compilers listed in Table 2.4. Version 7.0 represented a major overhaul of the LAM code base (orienting it to a component system architecture). Using these coding practices, only minor bugs have been identified (and subsequently fixed) since the original 7.0 release. Most bugs were on platforms or compilers that the LAM team did not have access to, and were therefore not originally tested.

TABLE 2.4

LAM/MPI's formally supported compilers

GNU 2.9x, 3.x	Absoft Fortran 7.5
Portland 5.x	Intel 5.x, 6.x, 7.x, 8.x
Solaris Forte 5, 6	IBM xlc 6.0
OS X xlc 6.0, xlf 8.1	IRIX MIPSPro 7.4

### 2.3 The Evolution of LAM/MPI

LAM/MPI is a large software package, consisting of hundreds of directories and thousands of files. Research, development, and maintenance of this code base – even for the LAM/MPI developers – is a complex task. Even though the LAM/MPI source code is fairly well structured (in terms of file and directory organization), contains many well-abstracted and logically separated functional code designs, and includes several flexible internal APIs, new LAM/MPI developers are inevitably overwhelmed when learning to work in the code base. Third party developers and researchers attempting to extend the LAM/MPI code base – or even to *understand* the code base – are frequently stymied because of the intrinsic complexities of such a large software system. Hence, not only does it take a long time to train new LAM developers, significant external contributions to LAM/MPI are fairly rare; most contributions tend to be small, specific patches.

For these reasons, a natural evolutionary step for LAM was to transform itself into a modular, component-based architecture. This dissertation presents the component system architecture of LAM/MPI: the System Services Interface (SSI).

### 2.3.1 Component Programming

Component programming is not a new concept; it is widely used in industry and is gaining popularity in academic circles. As its name implies, the central concepts in component programming are about composing units (typically at run-time) together to effect functioning systems [132]. A *component* is defined as “an executable unit of independent production, acquisition, and deployment that can be composed into a functioning system.” It is typically comprised of classes, procedures, and/or functions that export a well-defined interface. A *module* is a component paired with resources. A module’s interface is invoked by a higher-level *component framework*. The framework is responsible for discovering, marshaling, and utilizing individual modules. The framework sets policies about how the component interfaces are used and is typically focused on a specific system, architecture, or task.

It is common for an application to have multiple component frameworks, each performing some specific function. It is therefore necessary to have a “component framework framework” for managing them. This “meta framework” is called a *component system*. Similar to a component framework itself, the system defines fixed policies about usage and interaction between its frameworks.

Components are frequently confused with objects. It should therefore be noted that component programming is both distinct from and complementary to the concepts of object-oriented programming and software layering. Component programming can *use* object-oriented code and software layering, but does not necessarily imply either of them. For example, components can range from purely functional implementations with a single software layer to fully object-oriented implementations with deeply nested, layered software abstractions.

One of the main philosophical differences between components and objects is that

components have no *externally* observable state.<sup>2</sup> It follows that modules therefore do not have a unique identity; they cannot be distinguished from copies of themselves. Conversely, objects are units of instantiation; by definition, multiple instances of the same class are distinguishable from each other.

### 2.3.2 The System Services Interface (SSI)

The LAM SSI is a component system that provides run-time services to effect an MPI implementation at run-time. This architecture is divided into three distinct tiers: the component system itself, four component frameworks, and component modules in each of the frameworks. SSI, the component system, manages the four component frameworks. It provides the overall architecture and interaction between the component frameworks, including the ability to accept run-time parameters from higher-level abstractions (e.g., `mpirun`) and pass them down through the component frameworks to individual modules. Each component framework (sometimes referred to as *component types*), in turn, manages zero or more modules. A framework will discover, load, utilize, and unload modules [45, 85, 46]. Modules adhere to the interface prescribed by the component type that they belong to, and provide requested services to higher-level tiers and other parts of LAM/MPI. Figure 2.2 shows this relationship graphically.

A module is defined as a “deployable unit” meaning that each module is a self-contained set of source code. It has its own directory structure and can configure, build, and install itself. The SSI frameworks will automatically detect each component and invoke the corresponding hooks during LAM’s overall configuration, building, installation, and run-time phases.

This system architecture naturally fosters a “plug-n-play” approach to the services that LAM and MPI utilize. Modules can be added to an existing LAM installation where

---

<sup>2</sup>Although modules may have *internal* state, this state may not affect external behavior between subsequent invocations.

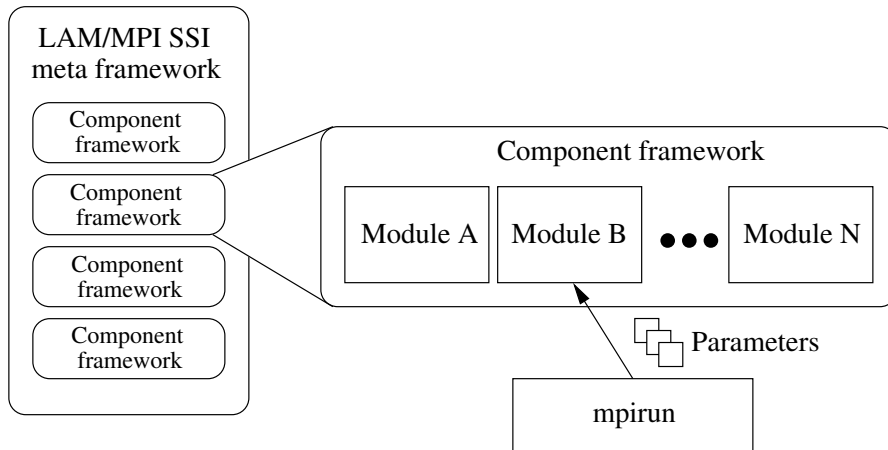


Figure 2.2. LAM/MPI is a component system architecture that manages multiple different component frameworks (or “types”); zero or more modules may be available from a given type. This figure depicts a component type with  $N$  modules, and shows `mpirun` passing in run-time parameters to module  $B$ .

they will be automatically integrated into the run-time environment of MPI processes. At run-time, LAM will compose a system based on the available modules and the run-time environment that it discovers. The result will be an MPI implementation uniquely suited to the environment in which is it running.

## 2.4 Related Work

### 2.4.1 Component Programming

Component systems are used in a wide variety of environments. Common systems include: the Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA) specifications [95, 94], Sun’s Java and its five related component families (JavaBeans, servlets, Enterprise JavaBeans, and J2EE application components), and Microsoft’s component technologies (COM [148] and COM+ [26], OLE/ActiveX [25], .NET [106]). Each of these technologies have varying degrees of industry traction, but taken together, represent a how much of the world’s software is written. The concepts

from these component architectures inspired some of the designs in this work. None of these architectures were directly used because they were too heavyweight, had commercial restrictions, or were generalized solutions mainly intended for web-based and client/server applications.

In the academic/research community, component usage is becoming popular. For example, the Common Component Architecture (CCA) Forum is defining specifications for high-performance component frameworks [2]. The Heterogeneous Adaptable Reconfigurable Networks System (HARNESS) project [11] extensively used the concepts of components and plug-ins to build distributed virtual machines; components are loaded into daemons at run-time to effect specific types of run-time environments. Although many other academic projects – too numerous to mention here – use the concepts of components and modules, the CCA and HARNESS are good representative examples. Both the CCA and HARNESS are closer in scope to what is required than the commercial component technologies listed above, but they are still too general of a solution (e.g., in HARNESS, an entire MPI implementation is a module); the SSI is specifically targeted at the services required for an MPI implementation – and nothing else – thereby allowing it to use abstractions specifically created for MPI-related issues and functionality.

Other, component-related technologies are also widely used. The Netscape web browser popularized the concept of a “plug-in module,” a concept that still exists in all modern web browsers. Users can “plug-in” new software components to provide added functionality to the browser. Indeed, the “plug-in” concept is used in many kinds of architectures; operating systems such as Solaris, Linux, and the BSD flavors all use kernel modules for extending the services that are offered to applications.

### 2.4.2 Run-Time Environment Startup

Parallel applications are required to run in a wide variety of run-time environments. Some of these include: the Portable Batch Scheduling (PBS) system [98, 102, 143], the Load Sharing Facility (LSF) HPC batch system [105], the Simple Linux Utility for Resource Management (SLURM) [70] system, the Sun Grid Engine (SGE) system [130, 131], the Quadrics Resource Management System (RMS) [107, 108], the Beowulf Distributed Process Space (BProc) [64, 65], the Computational Plant (CPlant) [113] Portals project [18], the IBM Parallel Operating Environment (POE) [119], traditional `rsh/ssh`-oriented clusters, and Grid-based environments.

MPICH provides sophisticated scripting in `mpirun` through a combination of configuration-time and run-time selection to decide how to launch a parallel job. The MPICH distribution includes support for several run-time environments. MPI implementations derived from MPICH typically add another `mpirun` that uses a new native run-time environment launch mechanism, if necessary. Although modular in nature, this is not component programming. The scripts are interdependent and cannot simply be “plugged in” at run-time without modifications to the central script libraries.

Other MPI implementations (including the prior generation of LAM/MPI) usually only supported one type of run-time environment, typically either `rsh/ssh` or whatever job-launching service was available in the implementation’s native environment. Native execution in additional environments was typically only provided by vendors to support their products, or system administrators to support their specialized systems.

The closest comparison to a `boot` component framework is grid-related technologies. But grid solutions are much more wide-reaching than what the `boot` framework seeks to accomplish; connecting widely disparate systems with different users and heterogeneous resources is far beyond the scope of `boot`. Although complex in itself, the `boot` is only intended to start LAM executables in relatively heterogeneous systems.

### 2.4.3 MPI Point-to-Point Message Passing

Almost all modern MPI implementations have some form of modular system for different implementations of point-to-point message passing. As the reference MPI implementation that was created alongside the MPI-1 standard, MPICH built on decades of message passing research by creating the Abstract Device Interface (ADI) [55] enable support for different underlying network interconnects. A formal interface was specified; the decision of which network to use was made at configuration / compilation time. Over the years, the ADI has grown and changed (MPICH2 is using ADI-3 [56]); many implementations have been written for different kinds of networks.

Most other MPI implementation have followed this example. Prior generations of LAM had the Request Progression Interface (RPI), which included similar configure / compile-time selection of network protocols, but also included command line options that allowed limited run-time selection capabilities.

In version 4.0 of the Sun HPC Cluster Tools, Sun MPI included support for Loadable Protocol Modules (LPM). This is quite similar to the `rpi` component framework, but tailored for the Sun implementation of MPI. Sun offers three LPMs with their distribution: shared memory, TCP, and remote shared memory. Myricom wrote an LPM for their proprietary network (Myrinet) and distributes it from their web site. The `rpi` framework built on and expanded many of the ideas in Sun MPI.

### 2.4.4 MPI Collective Message Passing

There is a wealth of literature available on parallel collective algorithms. The diversity in research has shown that there is no generalized set of algorithms that apply to all situations. The performance of a collective algorithm depends on multiple factors, including the communication patterns of the application, the underlying network topology, and the amount of data being transferred. Indeed, since MPI has become the predominant

message passing API, renewed interest and research has been cultivated in the collective algorithms used in MPI implementations.

Many modern MPI implementations use function pointers (sometimes cached on a per-communicator basis), such that a developer familiar with the implementation can easily change which algorithms are used. MPI applications, therefore, do not have a portable mechanism to change these algorithms at run-time. There do not appear to be any existing MPI implementations that use a component-based approach to collective algorithms.

#### 2.4.5 Parallel Checkpoint/Restart

Checkpoint/restart for sequential programs has been extensively studied. Libckpt [104] is an open source library for transparent checkpointing of Unix processes. Condor [86, 87, 88, 89, 134] is another system that provides checkpointing services for single process jobs on a number of Unix platforms. The CRAK (Checkpoint/Restart As a Kernel module) project [151, 152] provides a kernel implementation of checkpoint/restart for Linux. CRAK also supports migration of networked processes by adopting a novel approach to socket migration. BLCR (Berkeley Lab's Checkpoint/Restart) [33] is a kernel implementation of checkpoint/restart for multi-threaded applications on Linux [20]. Libtckpt [31] (different than Libckpt) is a user-level checkpoint/restart library that can also checkpoint POSIX threads applications.

In the context of parallel programs, there are vendor implementations of checkpoint/restart for MPI applications running on some commercial parallel computers [27, 10]. Some implementations are also available for checkpointing MPI applications running on commodity hardware. CoCheck [128, 129] is one such tool for PVM and MPI applications. It is built into a native MPI library called tuMPI and layered on top of a portable single-process checkpointing mechanism [49, 86]. One drawback to CoCheck is that a checkpoint request cannot be processed when a send operation is in progress. Conse-

quently, if a matching receive has not been posted by the peer, there is no finite bound on the time taken for the checkpoint request to complete. Also, checkpointing could change the semantics of MPI's synchronous sends in CoCheck: an anticipated receive could cause the return of the send instead of the actual receive by the application.

A checkpoint/restart implementation for MPI at NCCU Taiwan uses a combination of coordinated and uncoordinated strategies for checkpointing MPI applications [84]. It is built on top of the NCCU MPI implementation [24], and uses Libckpt as the back-end checkpointer. Checkpointing of processes running on the same node is coordinated by a local daemon process, while processes on different nodes are checkpointed in an uncoordinated manner using message logging.

Other research-quality checkpointing systems have also been implemented: [1, 15, 110, 111].

A limitation of the existing systems for checkpointing MPI applications on commodity clusters is that they are implemented using MPI libraries that primarily serve as research platforms and are not widely used. Another drawback of some of these checkpoint/restart systems is that they are tightly coupled to a specific single-process checkpointer. Since single-process checkpointers usually support a limited number of platforms, this limits the range of systems on which MPI applications can be checkpointed to those that are supported by the underlying checkpointer.

#### 2.4.6 Fault Tolerance

Checkpoint/restart is only one area of the larger field of fault tolerance. Although not addressed in this work, other areas of fault tolerance have been researched extensively. One of the problems in modern cluster-based systems is the loosely-coupled nature of the environment. Simply detecting that an error has occurred and then bringing all remaining processes into consensus that both a) an error has occurred, and b) where the error

occurred is a complex problem. Research has shown that in a truly asynchronous system, this is impossible [40, 144, 28]. Therefore, to solve this problem in a practical manner, some degree of synchronization must be introduced into the system [22, 29, 62].

Multiple approaches have been proposed for MPI implementations [54], although all are designed with the assumption that the MPI program itself is correct and that failure originated from outside of the application (e.g., hardware or network failures). FT/MPI [10] takes the approach of replicating software and hardware such that if any one element fails, a counterpart can take over and continue the application. FT-MPI [37, 38] (distinct from FT/MPI) focuses on detecting failures within existing MPI semantics (using distributed, semi-synchronous methods) and passing control back to the application for handling. LA-MPI [3, 50] implements transparent data integrity and device failover; if the interconnection network transmits incorrect data or fails during a run, LA-MPI will automatically re-transmit or switch to a different network.

## 2.5 Experimental Setup

The results reported in this document were obtained from a variety of sources, each relevant to the particular experiment being performed. As such, the hardware used for each experiment is described in each results section throughout the document.

The software used for experiments was consisted of several common packages. The Network Protocol Independent Performance Evaluator v3.6 (NetPIPE) [118], the NAS Parallel Benchmarks v2.4 [4, 5, 6], and the Pallas MPI Benchmarks v2.2.1 [100] were used for performance and correctness testing. NetPIPE is a program that performs ping-pong tests, using messages of increasing size between two processes across a network in order to measure communication performance. The NAS Parallel Benchmarks is a suite of application kernels that test several different computational and communication patterns in parallel environments. The Pallas MPI Benchmarks are a collection of bench-

marking routines for point-to-point and collective algorithm performance distributed from Intel GmbH, Software and Solutions Group.

The operating system, hardware, compiler, and compiler flags used for each test is described where the results are given in each chapter.

Two versions of LAM/MPI were used for testing: LAM/MPI v6.5.9 (the previous generation of LAM/MPI, reflecting a monolithic architecture) and a development version of the LAM/MPI code base (what will soon become LAM/MPI v7.1). These two versions are referred to as “version 6” and “version 7” throughout the text, respectively. The user account used for testing used the `tcsh` shell with a `.tcshrc` that contained one statement setting `$PATH`. SSH keys were setup for password-less / passphrase-less login for ease of testing.

## CHAPTER 3

### THE SYSTEM SERVICES INTERFACE

To create the LAM component system architecture, existing abstractions within the LAM/MPI code base were identified and re-factored to formalize their concepts and interfaces. This led to the natural creation of three component frameworks: run-time environment startup, MPI point-to-point communications, and MPI collective communications. Each of these sub-systems represented a major functionality group and had at least some level of abstraction within the LAM and MPI layers; the challenge was to convert their form without changing their function or overall performance.

At the same time, parallel checkpoint/restart functionality was introduced to LAM in a different research effort. This new functionality was originally added in a functional and deeply-integrated manner. But as the benefits and results of the component system architecture became clear, checkpoint/restart functionality was abstracted into a component framework with a corresponding component interface.

LAM therefore ended up with five component-based frameworks: the SSI meta framework, run-time environment startup (named “boot”), MPI point-to-point communication (called “rpi,” an acronym for MPI Request Progression Interface), MPI collective communication (abbreviated “coll”), and parallel application checkpoint/restart (abbreviated “cr”). This chapter describes the SSI meta framework and how it manages the other four frameworks. Appendix [A](#) is an in-depth technical reference for the services provided by the SSI.

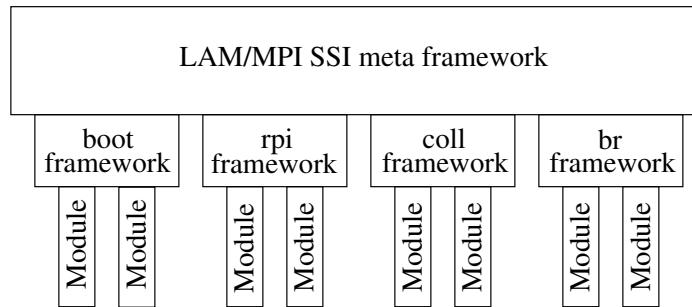


Figure 3.1. SSI is the top tier that binds together the individual component frameworks, and indirectly the modules that they contain.

**Acknowledgements:** All design and implementation of the SSI meta framework discussed in this chapter is new work and was performed by me.

### 3.1 Design

The SSI is the meta framework that ties together the individual component frameworks that comprise LAM/MPI. It was designed to represent the common functionality aspects of the component frameworks as well as provide communication and interaction between them. Policies laid out by the SSI govern much of how each of the component frameworks operate. Hence, although each framework is distinct, SSI manages them and, indirectly, the modules that they contain. See Figure 3.1.

The SSI was created with the following design goals:

- Allow multiple frameworks to co-exist within individual LAM and MPI processes.
- Delay the decisions about which modules to use until run-time. Criteria for such decisions are framework-specific.
- Continue the strict abstraction barrier between the LAM and MPI layers as originally designed in prior versions of LAM.
- Provide common services to component frameworks for acting on modules.
- Allow modules to be statically linked into executables or dynamically loaded into processes at run-time.

The secondary design goals of easing maintenance, creating original configurations by composing software units in unique ways, and enabling third parties to create modules are not explicitly listed because they are inherent to component-based systems. Each of these goals is discussed below.

### 3.1.1 Multiple Frameworks

Since LAM uses four component frameworks, a higher-level architecture system is necessary to have them act as a single, cohesive system. Without this, each component framework would implement much of the same infrastructure as its peers, and all interaction between them would be awkward at best. Thus, the SSI was created specifically to allow multiple frameworks that could interact with each other in a single process.

### 3.1.2 Run-Time Module Selection

Previous monolithic versions of LAM/MPI had only one replaceable sub-system: the underlying transport protocols for MPI point-to-point communication. Implementations existed for TCP, UDP, and shared memory. Although modular in nature, the decision of which transport implementation to use could only be made when LAM was configured and compiled. Additionally, each transport implementation offered a small number of adjustable parameters for changing run-time behavior. These, also, could only be changed when LAM was configured and compiled.

While this mechanism was effective, it forced users who wanted to experiment with multiple network types to not only have multiple installations of LAM/MPI (one for each network transport / parameter settings pair), but also to compile their MPI application with each LAM installation. This typically turned into a logistical nightmare. Separate directory trees needed to be maintained for each LAM installation; users typically needed to modify the `PATH` in their shell startup files (e.g., `.bashrc`, `.profile`, `.cshrc`, etc.) depending on which LAM installation they wanted to use. Further, users needed to

ensure that their MPI application was compiled and run with the same LAM installation.

ISVs who sell MPI-based products were also significantly affected. ISVs are in the unfortunate position of needing to support every MPI implementation that their customers use. This means potentially adapting their software for each MPI implementation, testing on each platform/MPI implementation combination, and shipping different executables to each customer depending on their back-end MPI implementation. Although this can actually be cost-prohibitive, it is the state of today's industry simply because of the nature of MPI implementations that are currently available.

With a component-based approach – allowing multiple MPI point-to-point transport modules to exist in a single application and converting all the adjustable parameters to be sensitive to run-time values – only one installation of LAM is necessary. Similarly, the user's application only needs to be compiled once. Configuration values (to include choosing which transport module to use) can be varied by composing different point-to-point transport modules and passing different configuration parameters at run-time. This not only greatly simplifies the logistics issues for the end user (and the system administrator supporting the end user), it also enables MPI-based ISVs to dramatically reduce the complexity of their binary distributions – one binary distribution can now function in a wide variety of LAM/MPI environments.<sup>1</sup> Finally, this component-based approach makes testing and maintenance significantly easier for the LAM developers.

The benefits of run-time selectable/parameterized modules also apply to the other three component types (run-time environment startup, MPI collective communication, and parallel checkpoint/restart). This represents a giant leap forward in the functionality delivered to system administrators, end users, and LAM developers.

---

<sup>1</sup>Although ISVs would strongly prefer if they could distribute one binary that will work with any MPI implementation on a given platform, even a component-based approach is not enough to create such universal transparency. Component-based solutions are a step in the right direction, but a universal solution will require consensus between MPI implementations, which is as much a social problem as it is a technical problem.

Note that the selection of which module(s) are used at run-time is a framework-level decision. The user may indicate preferences regarding which module(s) are used, but each component framework makes the final decision. For example, if a user indicates they they want to use a Myrinet-based module for point-to-point communication but no Myrinet hardware is found to be available at run-time, the `rpi` framework may either print a suitable error message or choose another module.

### 3.1.3 LAM and MPI Component Types

As shown in Figure 2.1 (page 13), LAM is divided into two layers: the LAM run-time environment and the MPI layer. Previous versions of LAM have maintained a strict abstraction separation between the two layers, thereby preserving logical separations of services and data flow. Although MPI is now the “main product” of LAM, and it is highly unlikely that anyone outside the LAM development team will write LAM-only programs, it was decided that the separation of the LAM and MPI abstractions were important for design and maintenance reasons. The new component system architecture therefore also needed to support the abstraction separation between LAM and MPI.

Hence, the majority of the SSI actually resides in the LAM layer, allowing both the LAM and MPI layers to use it. The component frameworks themselves, however, are split across the two layers. The `boot` framework resides in the LAM layer since it deals with starting LAM processes. The MPI-specific components (`rpi` and `coll`) logically reside in the MPI layer. The checkpoint/restart component is actually split in two parts – `crlam` and `crmpi` – which, as their names imply, reside in the LAM and MPI layers, respectively. This is because checkpoint/restart affects both layers and different actions need to occur in each.

### 3.1.4 Module Services

There are common actions which need to be invoked by the component frameworks for every module with only minor variations. Parameterized versions of these actions are therefore located in the SSI for global access. These services include:

- Configuration and compilation of the module
- Installation of the module (whether static or dynamic)
- Versioning of both the interface used by the module and the module itself
- Loading and unloading of the module at run-time (whether static or dynamic)
- Registering, accepting, and passing module parameters

These services are discussed in more detail in Section [3.2](#).

### 3.1.5 Static and Dynamic Modules

For convenience, modules can be statically linked into the LAM and MPI libraries. Existing configure/build systems, such as the GNU Autoconf, Automake, and Libtool suites, enable this functionality. All modules that are distributed with LAM/MPI currently default to static building.

In order to encourage third-party module development, separate configuration, compilation, deployment, and installation also needs to be possible. Specifically, a module needs to be able to be built in its own source tree (outside of the LAM source tree) and linked into a shared library that LAM can find and open at run-time. This gives third parties the capability of distributing binary-only modules (which is important to ISVs) as well as installing modules into an existing LAM installation. Modules therefore do not need to be distributed with the main LAM/MPI software package; any third party can distribute their module (in source or binary form) and simply install it where LAM expects to find dynamically-loadable modules. This enables the developers to focus on their modules instead of planning how to integrate them into an existing LAM installation, or, worse, forcing the use of multiple LAM installations.

## 3.2 Implemented Services

The SSI provides common services to all modules, regardless of which component framework uses them.

### 3.2.1 Configuration and Compilation

By definition, a module is a deployable unit and therefore needs to be configured and compiled. All the modules included in the LAM/MPI distribution use the GNU tools Autoconf, Automake, and Libtool, and therefore have their own “`configure`” scripts (used to find the compiler, run tests to determine operating system characteristics, etc.) and “`make`” hierarchies.

The SSI provides templates and additional “helper” macros for `configure` scripts and building procedures. In most cases, the SSI significantly reduces the workload for module authors by automatically providing a templated `configure` script suitable for most module configuration and setup.

### 3.2.2 Installation

Once a module has been compiled, it must be installed. For static modules, this means placing the output library in a well-known location in the build tree. This enables the LAM build process to find and incorporate the module into the LAM or MPI library (whichever is appropriate for the module’s component type). The output library does not need to be installed anywhere; its entire contents will be contained in the LAM or MPI library.

For dynamic modules, installation essentially entails copying the shared library containing the component to the location where LAM expects to find run-time loadable modules. The SSI framework in LAM and MPI executables will then be able to find and load the module at run-time.

### 3.2.3 Versioning

Each module contains three pieces of versioning information: the version of the SSI that it conforms to, the name and version of the component framework that it adheres to, and its own name and version number. The SSI uses this information for two purposes. First, it checks the SSI version, component framework name, and component framework version to see how it should interact with the module. This triple corresponds to a specific interface that must be used to communicate with the module.

Second, the SSI ensures that the aggregate set of modules used in a given application (serial or parallel) agree both in name and in version. The SSI defines two identically named modules to have compatible versions if the major and minor version numbers are the same (the release version number is for information only; it is ignored during version number comparisons). For example, two `foo` modules having versions `v1.2.3` and `v4.5.6` would not be considered compatible, but versions `v1.2.3` and `v1.2.8` are considered compatible.

### 3.2.4 Loading / Unloading

When a LAM or MPI process starts, relevant component frameworks will load their respective modules. This is always a two phase operation: 1) finding statically linked modules already loaded in the process space, and 2) finding shared library modules that can be dynamically loaded into the process space.

Static modules are found by traversing a global array of module references (indexed by component framework) that was constructed when LAM was configured and compiled. Dynamic modules are found by looking in a list of directories and eagerly loading all modules of the desired component type.

As each module is found, it is initialized by invoking the module's "open" function. This function allows the module to perform basic queries of the run-time environment

and determine if it can be used in the process. If the module determines that the run-time environment is not suitable, it will return a failure status from the open function causing the framework to close and unload it. Otherwise, the module will return a success status and the framework adds it to the list of *available* modules.

The *scope* of a module is the set of conditions in which it can be used. For example, some component frameworks will only select one module for the life of the process, while other frameworks may select a different module for every MPI communicator. Both the scope and the selection algorithm used are defined by each component framework. These are therefore discussed in later chapters.

### 3.2.5 Parameter Passing

One of the major reasons for using components in LAM/MPI is the idea of customizing the MPI implementation at run-time to the particular environment in which it is operating. A companion concept to this idea is the ability to send arbitrary parameters from higher level abstractions (such as `mpirun`) to modules in LAM and MPI processes. LAM allows such parameters to be passed on the command line or through environment variables.

To this end, the SSI provides a central parameter service where modules can register the parameters that they want to receive (indexed by name). Modules can then request values for the parameters that they have registered. The SSI uses a hierarchical search to find parameter values: the command line is searched first, followed by the environment, and finally a default value is used if no other value is found.

A centralized service for module parameters has several benefits. First, only one generic command line option is necessary for passing all parameters. For example:

```
shell$ mpirun --ssi parameter_name value -np 4 my_mpi_program
```

Here, `mpirun` is launching four instances of the MPI executable `my_mpi_program`

and passing each of them the module parameter named “parameter\_name” with a value of “value”. Similarly, a common form for environment variables can be used to pass module parameters:

```
shell$ LAM_MPI_SSI_parameter_name=value
shell$ export LAM_MPI_SSI_parameter_name
shell$ mpirun -np 4 my_mpi_program
```

This example is the same as the previous; four instances of `my_mpi_program` are launched and each are passed the “parameter\_name” module parameter. Only the mechanism for specifying the parameter is different.

The centralized service handles the bookkeeping of parsing, analysis, and parameter propagation to all parallel processes. This allows construction of master lists (indexed by component type) of all module parameters and default values, enabling utilities such as the `laminfo` command to display them to the user.

### 3.2.6 The `laminfo` Command

The `laminfo` command is one of the tools included in the LAM/MPI software package. `laminfo` provides diagnostic information about all static and dynamic modules that it is able to find. For example, `laminfo` can report the versioning data from each module, the parameters that the module accepts, and the parameters’ corresponding default values. Such diagnostic information can be valuable to end users, system administrators, third party developers, and the LAM developers. Figure 3.2 shows sample output from the `laminfo` command.

### 3.2.7 Common Selection Schemes

Although the selection algorithm of a given component type is specific to its component framework, most use a similar algorithm.

```

shell$ laminfo --version rpi full
  SSI rpi: crtcp (SSI v1.0, API v1.1, Module v1.0.1)
  SSI rpi: lamd (SSI v1.0, API v1.0, Module v7.0)
  SSI rpi: tcp (SSI v1.0, API v1.0, Module v7.0)
  SSI rpi: sysv (SSI v1.0, API v1.0, Module v7.0)
  SSI rpi: usysv (SSI v1.0, API v1.0, Module v7.0)
shell$ laminfo --param rpi sysv
  SSI rpi: parameter "rpi_sysv_pollyield" (default value: "1")
  SSI rpi: parameter "rpi_sysv_poolsize" (default value: "16777216")
  SSI rpi: parameter "rpi_sysv_maxalloc" (default value: "1048576")
  SSI rpi: parameter "rpi_sysv_short" (default value: "8192")
  SSI rpi: parameter "rpi_sysv_priority" (default value: "30")

```

Figure 3.2. Sample outputs from the `laminfo` command. The first command shows all available `rpi` modules and their associated versioning data; the second command shows the `sysv` `rpi` module parameters and their default values.

The selection of which module to use in a given scope is always determined from the available module list. Each component framework mandates both query/initialization and finalization functions. During selection, all the modules in the available list are queried to determine if they can run, and if so, what their *priority* is. Priorities are integers in the range of  $[0, 100]$ , with 100 being the highest. The module with the highest priority will typically be selected as the winner; the losing module(s) will be ignored in the selection scope. If there is a tie, LAM is free to select any of the modules with the highest priority.

Although the exact meanings of priorities are arbitrary, the following guidelines are provided for module authors in assigning priority values:

- A priority of 0 should be interpreted as “if nothing else is available, this one will work.” It is typically reserved for a lowest common denominator type of module.
- A priority of at least 50 should be given when a module detects that it is running in its “native” environment. For example, if `lamboot` is executed in a PBS batch queuing environment, the `tm boot` module will return a priority of at least 50 because it knows that it can run.
- A priority of at least 75 should be given when a module detects that it is running in its “native” environment and a compile-time switch was enabled to make that module the default for its type.

- A priority of 100 should be reserved for modules that *must* be selected.

If a module does not have a parameter named “<type>\_<module>\_priority”, the SSI framework will create one and assign it a default value of zero. It is recommended that module authors create their own priority parameter and assign it a reasonable default value.

## CHAPTER 4

### RUN-TIME ENVIRONMENT STARTUP

MPI applications are run in a variety of parallel environments. Users of traditional Beowulf-style clusters have grown accustomed to using `rsh` or `ssh` to start MPI processes on remote nodes as shown in Figure 4.1. While effective, `rsh` and `ssh` cannot necessarily provide tightly-integrated job control. Although recent versions of `ssh` usually ensure to kill all processes when, for example, control-C is pressed in the launching application, some processes may still be orphaned on remote nodes (and keep running in the background). Additionally, the use of `rsh` and `ssh` does not scale to large clusters consisting of hundreds or thousands of nodes.<sup>1</sup>

For these reasons, prior versions of LAM/MPI did not directly launch MPI processes via `rsh` and `ssh`. It was decided that starting the run-time environment occurs much less frequently than running MPI applications, particularly while developing and debugging. Therefore, the latency of `rsh` and `ssh` should only be incurred once while launching a network of user-level LAM daemons. These daemons are then used to provide definitive job control and monitoring that is not possible through `rsh` and `ssh`. Once the daemon network is established, commands such as `mpirun` send UDP messages to the daemons in order to launch and kill remote MPI processes.

Other environments, such as batch schedulers, provide their own interface for launch-

---

<sup>1</sup>`ssh`, in particular, does not necessarily scale well because of its noticeable latency when utilizing strong authentication mechanisms.

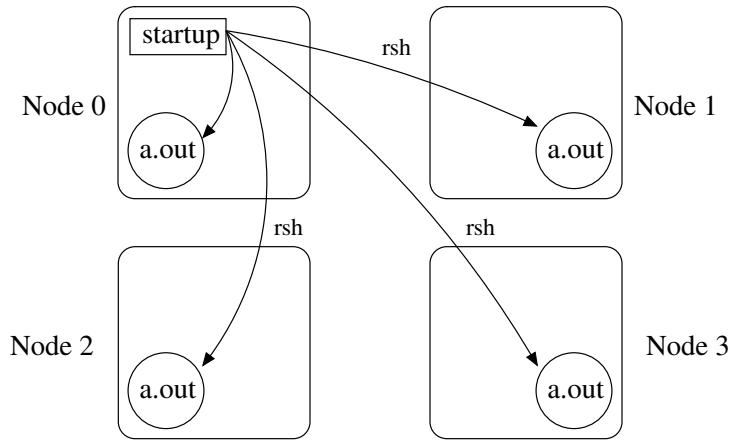


Figure 4.1. Sample parallel applications started with `rsh` (or `ssh`).

ing, monitoring, and killing processes on remote nodes. These interfaces typically include tightly-integrated job control, provide accurate accounting details, and disallow running on resources that are not allocated to a given job. In such environments, it is clear the MPI implementation should use the provided job control mechanisms rather than `rsh` and `ssh`.

Unfortunately, open source MPI implementations – including LAM/MPI – traditionally only directly supported `rsh` and `ssh`. Support for specialized environments was typically provided through patches from third party contributors, workarounds such as `rsh` and `ssh` replacements, or redistribution of customized MPI implementations that were modified to use customized external job control mechanisms.

This chapter discusses the `boot` component framework and how it addresses these problems by providing a dynamic mechanism to add support for arbitrary run-time environments to LAM/MPI. Appendix B provides a detailed technical reference of the `boot` component framework.

**Acknowledgements:** The `boot` component interface was jointly designed with Brian Barrett, who also implemented most of the `tm boot` module. Brian, Nicholas Henke, and

myself jointly wrote the `bproc` module. Vishal Sahay, under my direction, designed and wrote the `globus` module. The `rsh/ssh` functionality was adapted from older LAM/MPI functionality and converted into the `rsh` module by me. The design and implementation of the component architecture was performed by me.

## 4.1 Design

The main design goal of the `boot` component framework is to allow LAM/MPI to use existing run-time environments. `rsh` and `ssh` support (which is just another `boot` module) should only be used only when no other job control mechanism is available.

To create a suitable component interface, it was necessary to study the abstract actions required to start processes remotely. Two run-time launch mechanisms were primarily studied to categorize remote-launch actions: `rsh / ssh` and the Portable Batch System (PBS) [98, 102].

### 4.1.1 LAM Daemon-Based Run-Time Environment

As with previous versions, the `lamboot` command is used to launch a LAM-specific run-time environment. A single LAM daemon is launched on each physical node that will be used. This process is commonly referred to as *booting* the LAM run-time environment (and hence the “`boot`” component framework name). The node that runs `lamboot` is called the *origin* node. During the boot, each LAM daemon is told how many “CPUs” it should use for MPI processes on that node. This CPU count is not bound to an actual number of CPUs; indeed, it is typically provided by the user. The CPU count is simply a management abstraction telling LAM how many processes can typically be launched on a particular node.

Once the LAM daemon run-time environment is established, it is referred to as the LAM *universe*. The LAM universe provides a variety of services to programs that run in

it, such as job control, I/O forwarding, and basic message passing. The message passing services are usually referred to as “out of band” throughout this document; MPI communications are considered “in band.” In fact, all universe-provided services are dependent on the out of band messaging. For example, `mpirun` uses the job control mechanisms to launch MPI processes by sending out of band control messages to the relevant LAM daemons.

The role of the `boot` component framework is therefore to launch LAM daemons, pass startup parameters (such as CPU counts), and make the daemons aware of each other. It is invoked by the `lamboot` command and controls how daemon processes are invoked on all nodes that will be used.

#### 4.1.2 Remote Process Startup Case Study: `rsh` / `ssh`

Using `rsh` and `ssh` to start processes on remote nodes is relatively straightforward and fairly well understood. Using `rsh` / `ssh` has both benefits and drawbacks:

- Benefit: straightforward, well-understood execution model that has been used for years.
- Benefit: automatic standard output and standard error forwarding from remote processes.
- Drawback: potentially poor job control; hitting control-C in the launching application, for example, may or may not reliably kill all processes on remote nodes.
- Drawback: potentially large latency (especially with `ssh` when using its strong authentication mechanisms) that impacts effective scalability.

The interface for `rsh` and `ssh` has not changed significantly in many years. Both commands are fairly interchangeable from an invocation perspective.<sup>2</sup> Hence, the steps required to launch remote processes is virtually identical between the two.

---

<sup>2</sup>Although using some of the more complex `ssh` authentication mechanisms takes considerably more user-level setup than `rsh`'s primitive `.rhost`-based authentication.

When using `rsh` or `ssh` to launch LAM daemons, the user must supply a list of hosts (typically in a host file, commonly referred to as a *boot schema*) on which to start the daemons. With this file, starting the LAM daemons requires the following steps:

1. Parse the command line to find the boot schema filename.
2. Parse the boot schema file and generate a list of nodes. For each node:
  - Identify the node IP name/address to use. If a name, verify that it is able to be resolved into an IP address.
  - Identify the CPU count for this node
3. Check for errors in the node list.
  - Verify that the current node is in the node list.
  - Verify that 127.0.0.1 is not used in conjunction with any other addresses.
4. For each node in the node list:
  - Launch a child process to run an `rsh` or `ssh` command line that launches a LAM daemon on the node. Wait for the child process to complete.
  - Wait for the newly-spawned LAM daemon to “call back” and send its location information (e.g., its IP address, the UDP and TCP ports that it is listening on, etc.). Save this information.
5. Broadcast the full set of LAM daemon location information (including the per-daemon CPU counts) to each newly-spawned LAM daemon.
6. Exit `lamboot`.

Note that the LAM daemon is not launched directly via `rsh` or `ssh`; instead, LAM launches a “helper” executable named `hboot` on remote nodes to facilitate job control issues. Among other reasons, this ensures that `rsh` and `ssh` can terminate successfully even though the LAM daemon has been forked into the background.

The resulting `rsh` / `ssh` boot process is shown in [Figure 4.2](#).

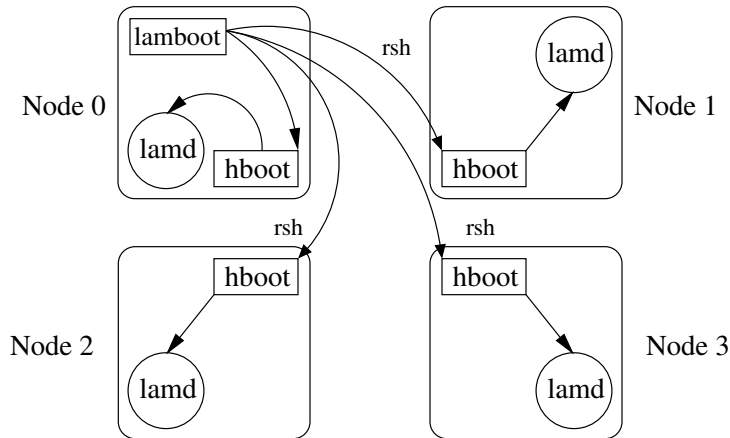


Figure 4.2. Booting the LAM run-time environment using `rsh` or `ssh`.

#### 4.1.3 Remote Process Startup Case Study: The Portable Batch System (PBS)

As its name implies, PBS is a batch scheduling system for workload management; it coordinates the utilization of dedicated and shared resources. PBS was initially developed at NASA but was later commercialized by Veridian software. It is now developed and marketed by Altair Grid Technologies, LLC. There are currently three versions of PBS available:

- OpenPBS [98]: open source, available at no cost. OpenPBS has not undergone any substantial development for several years.
- PBS/Pro [102]: commercialized product (not free). PBS/Pro is marketed, developed, and sold by Altair Grid Technologies, LLC.
- Torque [143]: open source, available at no cost. Torque is effectively a “fork” of the OpenPBS project that is under active development.

All three of these software systems are interchangeable from an MPI implementation’s perspective, and therefore are collectively referred to as “PBS” throughout this document.

PBS provides a C library interface called the Task Management (TM) interface [99] for job control in the PBS environment. PBS’s TM interface is a subset of the PSCHED interface [61]. The PSCHED API aimed to provide a complete interface for parallel job

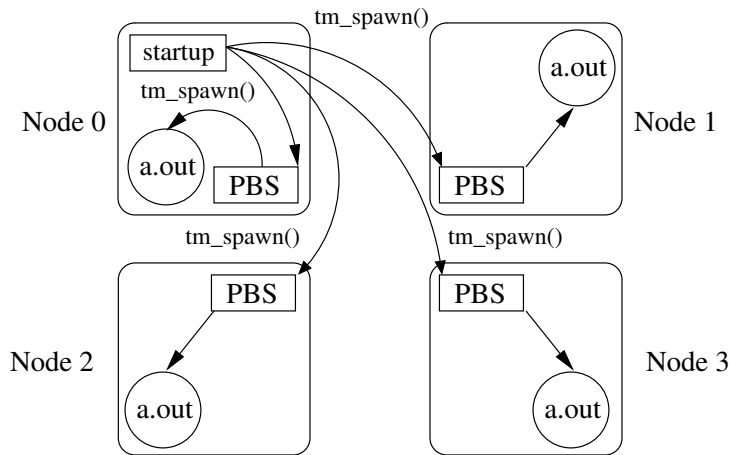


Figure 4.3. Starting a parallel application using PBS's TM interface (`tm_spawn()` is the interface function for starting processes under PBS's control). PBS is able to track resource utilization for the entire application.

and resource management. Utilizing the TM interface results in an application startup similar to that shown in Figure 4.3. All processes started by the TM interface remain under PBS control, allowing both guaranteed resource cleanup and accurate process accounting.

PBS is composed of three main services: a scheduler, a server, and a per-node control daemon (referred to as the Machine Oriented Mini-server, or MOM), shown in Figure 4.4. The scheduler provides job-to-node mappings and handles queue management. The server handles communication between services as well as accounting logs. There is one scheduler and one server per cluster. The PBS MOM executes on every node and provides a number of health and process control features. The MOMs are also used by the server and TM interface for job startup.

PBS is typically utilized by submitting scripts to the batch scheduler, creating a queued job. These scripts (or the command line used to submit them) specify run-time parameters such as how many nodes to use, how long the job is expected to run, etc. PBS allows specification of parallel jobs either by indicating how many nodes to use or how many

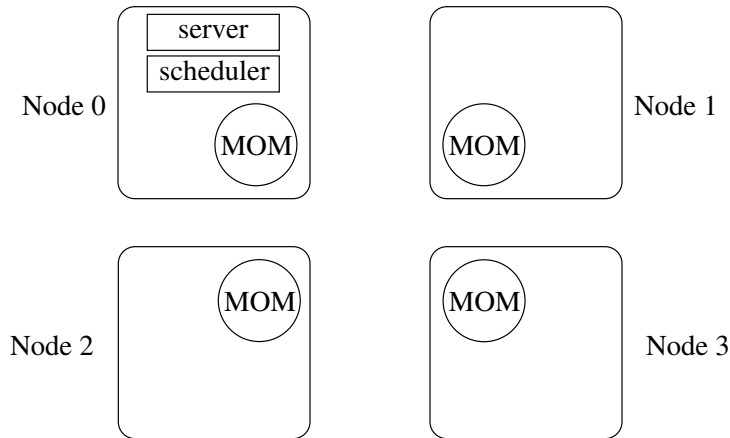


Figure 4.4. Example architecture of PBS over four nodes. There is one instance each of the PBS server and scheduler, and per-node instances of the MOM.

“virtual CPUs” to use. A “virtual CPU” (or VCPU) is PBS’s abstract concept of a CPU, and may actually be located on any physical node. It maps almost exactly to LAM’s concept of “CPU”: a VCPU is not closely bound to a physical CPU, and is only intended as a management abstraction.

When the scheduler finds enough resources to run a job, it allocates a set of VCPUs to the job and starts the script running on the node corresponding to the first allocated VCPU. The script is expected to do everything that is required to run the job, including – if necessary – starting processes in parallel.

Assuming that a PBS job script invokes the `lamboot` command, starting the LAM daemons using the TM interface requires following steps:

1. Ensure that the process is actually running in a PBS job.
2. Invoke the TM initialization function.
3. Generate a list of nodes.
  - Query TM for a list of VCPUs on which the job is allowed to run.
  - Cross-reference the VCPUs with additional information provided by TM to calculate a minimum set of VCPUs that represents each physical node in the job exactly once (recall that `lamboot` starts exactly one LAM daemon on each physical node). Convert this VCPU set to be a list of nodes. Also count how many VCPUs appear on each physical node.

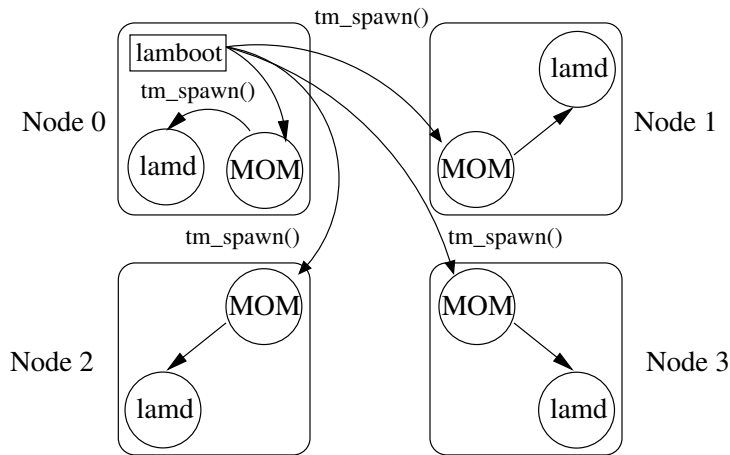


Figure 4.5. Booting the LAM run-time environment using the PBS TM interface.

4. For each node in the node list:
  - Invoke the `tm_spawn()` function to launch a LAM daemon on each node.
  - Wait for the newly-spawned LAM daemon to “call back” and send its location information (e.g., its IP address, the UDP and TCP ports that it is listening on, etc.). Save this information.
5. Broadcast the full set of LAM daemon location information (including the per-daemon CPU counts) to each newly-spawned LAM daemon.
6. Invoke the TM finalization function.
7. Exit `lamboot`.

The resulting TM boot process is shown in Figure 4.5. Note that the `hboot` command is not necessary in a TM environment; `hboot` is designed to overcome specific problems with `rsh` and `ssh`, and is therefore not pictured in Figure 4.5.

#### 4.1.4 Action Abstractions

Examining the steps taken to create LAM universes in the two case studies, the common actions can be reduced down to two main abstractions:

- Generate a list of nodes, count how many CPUs appear on each node, and launch a LAM daemon on each.

- Perform a rendezvous between the newly-launched LAM daemons and `lamboot` such that every daemon becomes aware of its peers.

The specific mechanisms used for the first abstraction are quite different between the two scenarios, but the end result is the same: a LAM daemon is running on each target node. The mechanisms used for the second abstraction are almost identical between the two scenarios. From this analysis, it is clear that there are at least two distinct functionality categorizations: identifying / launching LAM daemons and performing the startup rendezvous. Although there can be overlap between these two categories for ease of implementation or optimizations (e.g., launch a LAM daemon and wait for it to send back its location information), the two categories still perform different functions.

A third category is also possible. In both scenarios, the phrase “for each node in the node list...” was used to indicate an action that needed to be performed on all nodes (particularly with respect to launching a process on a node). There are many algorithms that can be used to implement a “for each item in a list” operation: linear, linear windowed, fixed tree traversal, etc. Therefore, a third functionality category of launching is the algorithm used to traverse all the nodes in the list.

Putting all three of these categories together, along with the similarities and differences of the case study scenarios, above, the list below itemizes the abstract actions that need to be supported by a `boot` module. Each item is marked with “[**Launch**],” “[**Rendezvous**],” or “[**Algorithm**]” to indicate its functionality category.

1. [**Launch**] Parse command line options. If necessary, parse the command line and find the argument specifying the boot schema filename.
2. [**Launch**] Allocate nodes. Construct the node list and count CPUs on each node. Data can be obtained from either the boot schema file or some external source (e.g., the TM interface).
3. [**Launch**] Verify nodes. Ensure that all entries in the node list are valid (e.g., can resolve all names to IP addresses) and that there are no other errors in the list.
4. [**Launch**] Prepare to boot. This is actually a new step that is not required by either `rsh / ssh` or `PBS`, but it is a natural location for a “hook” that support for future

run-time environments may require – a last place for any additional setup work before actually starting remote processes.

5. **[Rendezvous]** Open server port. Open an incoming / server-side port to accept incoming connections (for use in the next step).
6. **[Algorithm]** Start processes. Use some algorithm to perform these actions for each node:
  - **[Launch]** Start a LAM daemon on a specific node.
  - **[Rendezvous]** Receive LAM daemon location information from the newly-started daemon.
7. **[Rendezvous]** Close server port. Disable accepting new connections.
8. **[Rendezvous]** Broadcast information. Loop over the array of started LAM daemons and send the full set of LAM daemon location information obtained in the previous step.
9. **[Launch]** Free resources. Now that all daemons have started, free any resources that were allocated during the boot.

Notably absent from this list are the **[Rendezvous]** client-side actions that are required in the LAM daemon. The following `boot` actions are therefore used by the LAM daemon:

1. **[Rendezvous]** Send the location information for this LAM daemon back to `lamboot`.
2. **[Rendezvous]** Receive the full set of location information about all LAM daemons from `lamboot`.

All of these actions (for both `lamboot` and the LAM daemon) have been mapped to module interface functions and are described in Appendix B.

Note that modules may choose to perform no-ops for some actions. For example, when booting in a PBS job, there is no boot schema to parse. Similarly, there is no need to verify resulting the node list because all the nodes were received directly from the TM interface and therefore all nodes must be valid. Another example is the algorithm; it may either custom-written for a module or use common, pre-packaged algorithms that are provided by the `boot` framework.

#### 4.1.5 Additional Tools

Other LAM commands also use the `boot` component framework. `recon` is essentially a “dry run” of `lamboot`; it tries to launch processes on remote nodes and reports success or failure back to the user. This tool is mainly used to ensure that users have their settings for password-less login via `rsh` or `ssh` setup properly. `wipe` is a carryover from when `rsh` was the main/only mechanism for launching remote nodes, and the LAM run-time environment was not as mature and stable as it is now. If something went wrong with the run-time environment, the `wipe` command would use `rsh` or `ssh` to launch “killer” processes on each node to kill the LAM daemon, release all resources, etc.

Although these two commands need to utilize the **[Launch]** and **[Algorithm]** functional categories of the `boot` module interface, they clearly do not need to utilize the **[Rendezvous]** functionality. As such, the functions described in Appendix B are properly parameterized such that a flag controls whether the **[Rendezvous]** functionality is used or not.

The `lamgrow` command is used to grow an existing LAM universe by one node. It must be run from within a existing universe and have the new node specified on the command line. A `boot` module will be used to launch an additional LAM daemon (including performing the **[Rendezvous]**) and integrate it into the existing universe.

When the `boot` component framework was implemented, the `lamboot`, `recon`, `wipe`, and `lamgrow` commands were transformed into simplistic engines that essentially invoke functions on the selected `boot` module to effect the actions in the sequence shown in Section 4.1.4. Each engine is slightly different; the boot framework provides the flexibility to implement all four commands.

#### 4.1.6 Module Selection

The selection process for `boot` modules uses the simplistic priority-based system described in Section 3.2.7. Essentially: the `boot` component framework queries all available modules for 1) whether they can run or not, and 2) an associated priority. All modules who reply that they can run are ranked by order of their priority. The module with the highest priority is selected; all others are finalized and closed.

The selection of which `boot` SSI module to use persists through the life of the LAM universe. This is not only governed by the fact that the LAM daemons will make a `boot` module selection when initially launched (and keep using that selection until they terminate), it simply does not make sense to change the `boot` module selection after the LAM universe has been established.

Hence, since all LAM processes will only exist within the timeframe of a single LAM universe, the scope of the `boot` module selection is the life of the process (in the case of the LAM daemon, this coincides with the life of the universe). As such, there will only ever be one `boot` module selected during a given process.

No LAM-provided out-of-band communication is available between `boot` modules of peer processes because by definition, there is no LAM run-time environment when the `boot` modules are initialized. Hence, consensus of which `boot` to use must be able to be achieved independently, or utilize communication channels that are provided by the underlying `boot` module's mechanism.

#### 4.1.7 Rendezvous Algorithms

The algorithm used to start execution over a set of nodes is up to each module; a module can provide its own functionality or use one of the generic algorithms provided by the `boot` component framework. The provided algorithms provide all the structure and bookkeeping necessary to launch across a set of nodes (including error detection).

If a module chooses to utilize the built-in algorithms, it provides function pointers for callbacks before invoking the algorithm function. The algorithm function will then iterate over all nodes in the set, invoking the module's callback functions to launch executables on remote nodes. The callback functions required by the provided algorithms are marked as “[Algorithm],” and are described in Section B.2 (page 210). If a module chooses not to use these algorithms, the callback functions do not need to be provided; NULL should be specified for these function pointers.

The following generic algorithms are provided in LAM/MPI:

- **Linear:** A linear approach is used to launch a set of processes: each process is individually started and, if requested, the rendezvous protocols are executed. The next process is not started until the current process has been fully established. This is the simplest algorithm.
- **Linear / windowed:** Well-suited for remote agents that do not need to wait for remote processes to complete launching, this algorithm uses a linear approach with a sliding window for remote process rendezvous callbacks – never allowing more than  $N$  callbacks to be outstanding at any given time. This algorithm is typically only useful for launching LAM daemons (i.e., processes that require callbacks to the booting agent) when hiding the latency involved in remote launching and/or startup protocols is noticeable. If the startup rendezvous protocols are not required, this algorithm is exactly equivalent to the linear algorithm.

## 4.2 Implemented Modules

Several boot modules have been implemented and included in the LAM/MPI distribution, allowing LAM/MPI to be executed in a variety of different run-time environments.

### 4.2.1 The bproc Module

The Beowulf Distributed Process Space (BProc) is set of Linux kernel modifications, utilities and libraries which allow a user to start processes on other machines in a Beowulf-style cluster [64, 65, 117]. Remote processes started with this mechanism appear in the process table of the front-end machine in a cluster. Processes therefore achieve node independence; they can be launched on the front-end cluster machine and moved to compute nodes for execution.

The `bproc` module uses the BProc C interface function `bproc_vexecmove()` to launch LAM daemons. This function executes a new LAM daemon and then moves it out to the target node using internal BProc mechanisms. Although `bproc_vexecmove()` accepts a list of nodes on which to start processes, the LAM daemons are launched in a linear fashion because each one requires slightly different command line arguments. The `wipe` and `recon` commands, however, launch the same executable command on every target node. They can therefore launch across a list of nodes in a single call to `bproc_vexecmove()`.

Similar to a traditional `rsh/ssh`-based cluster, the normal usage of LAM/MPI on a BProc cluster is to launch `lamboot` on the cluster front-end node with a booth schema listing all the hosts that will be in the LAM universe. LAM/MPI launches a LAM daemon on each node (including the front-end node). However, most BProc clusters utilize the front-end node for BProc management, general logins, compilations, etc. It is typically not appropriate to run computationally-intensive codes on the BProc front-end node.

As such, the `bproc` module takes advantage of an abstraction that already existed in the LAM/MPI code base: the ability to “schedule” processes based on attributes in the LAM universe. Specifically, the `bproc` module automatically marks the front-end node as “non-schedulable.” Hence, commands such as the following:

```
shell$ mpirun C my_mpi_application
```

will run `my_mpi_application` on all CPUs in the LAM universe *except* the BProc front-end node.

#### 4.2.2 The `globus` Module

Globus is a software toolkit aimed at running applications on systems that coordinate resources using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service but who are not subject to centralized control [44]. The Globus

```
# Globus boot schema
inky.my_cluster:12853:/O=MegaCorp/OU=Mine/CN=HPC prefix=/opt/lam cpu=2
pinky.your_cluster:3245:/O=MegaCorp/OU=Yours/CN=HPC prefix=/opt/lam cpu=4
blink.your_cluster:23452:/O=MegaCorp/OU=His/CN=HPC prefix=/opt/lam cpu=4
clyde.her_cluster:82342:/O=MegaCorp/OU=Hers/CN=HPC prefix=/software/lam
```

Figure 4.6. Sample boot schema for the `globus boot` component framework. Each line can specify a different `prefix` and CPU count.

Toolkit is an open architecture, open source software distribution containing software tools that make it easier to build Grid infrastructures and Grid applications [42, 43].

The `globus` module for LAM/MPI supports limited Globus functionality; only the Globus “fork” scheduler is supported. Starting the LAM run-time environment in Globus environment makes use of the Globus Resource Allocation Manager (GRAM) client `globus-job-run`. Although the `globus` module will report itself available if `globus-job-run` can be found in locations specified by the `$PATH` or `$GLOBUS_LOCATION` environment variables, the default priority is low enough that it is never selected automatically. The user must manually select to use the `globus` module.

Hosts in the boot schema file are listed by their Globus contact strings [75, 41]. For example, in cases where the Globus gatekeeper is running as a `inetd` service on the node, the contact string will simply be the hostname. Each host in the boot schema must also have a “`prefix`” key indicating the absolute directory where LAM/MPI is installed. This value is mandatory because Globus does not source users’ shell startup files when launching executables remotely, and therefore the `$PATH` environment variable may or may not reflect where the LAM executables are located. Figure 4.6 shows an example boot schema with Globus contact strings.

The `globus` module iterates over the contact strings and essentially invokes `globus-job-run` with each of them. In most other ways, the `globus` module functions just like

the `rsh` module.

#### 4.2.3 The `rsh` Module

The functionality of the `rsh` module is outlined in the `rsh / ssh` case study earlier in this chapter (Section 4.1.2). Although it is still a popular choice among small- to mid-sized clusters, it is, by far, the slowest of the `boot` modules. Since the linear algorithm is used, the time per daemon launch is significantly higher for `rsh` and `ssh` as compared to the other `boot` modules. Although there is no technical restrictions for booting arbitrarily large number of nodes, the wall clock execution time can grow quite large.

`ssh` typically offers better job control than `rsh` (e.g., control-C propagation is more reliable), usually leading to fewer orphaned processes when a user interrupts a parallel job. But `ssh` can be considerably slower than `rsh`, leading to [further] scalability problems.

Benefits of the `rsh` `boot` module include its strong heterogeneity support. There are no central decisions about where the LAM executables are located in the filesystem. Hence, as long as the `$PATH` environment variable is set correctly on each node, the LAM daemon executable will be found and launched.

#### 4.2.4 The PBS `tm` Module

The functionality of the PBS `tm` module is outlined in the PBS case study earlier in this chapter (Section 4.1.3). There are several benefits to using the `TM` interface:

- Start LAM run-time environment in a “native” fashion. `rsh` or `ssh` are not used, and may actually be disabled in some cases (to prevent users from logging on to nodes that are not allocated to them).
- Provide guaranteed cleanup of resources. For example, if a job’s time limit expires, PBS is aware of all processes on all nodes in the job, and can therefore reliably find and kill them. This is in contrast to when `rsh` or `ssh` is used to start jobs on nodes in a PBS job; in such cases, PBS is unaware of the `rsh/ssh`-started processes and is therefore unable to kill them. This has been a significant headache to system administrators as some processes may continue running after PBS thinks that the job has been killed.

- Accurate resource accounting. Since the PBS daemons are ancestors of all processes started using the TM interface, they have access to full accounting information about executed applications. Accurate resource accounting is important for fair scheduling policies, organizations that bill for computer time, error analysis, etc.
- The asynchronous nature of the TM interface function `tm_spawn( )` allows LAM to launch daemons in parallel. Hence, the startup time of the LAM universe under PBS was dramatically reduced (as compared to `rsh` and `ssh`) even for small numbers of nodes.

## Resource Cleanup

When PBS is going to prematurely kill a job, the MOMs in the job will first send a `SIGTERM` signal to the POSIX process group of all processes that they are aware of. When using the TM interface, the MOMs are aware of all job processes (as compared to when using `rsh/ssh`, where PBS is only aware of processes on the first node), and can therefore send a `SIGTERM` to the LAM daemon on each node.

The LAM daemon catches the `SIGTERM` and initiates an orderly shutdown process. It kills all MPI jobs, releases all held resources, and finally exits. Each MOM will wait up to 30 seconds for all process to exit. If not all processes quit, the MOM will use the uncatchable `SIGKILL` to kill them. Once all jobs have finished, the MOM terminates the job and releases the resources back to the scheduler.

## TM Shortcomings

The TM interface has its own shortcomings. For example, for a set of processes launched via `tm_spawn( )`, if any one of them exits, PBS will kill all the rest. This is not technically a problem since the LAM daemons persist for the entire universe, but for daemon-less MPI run-time environments (a potential future direction for LAM/MPI), this could cause unexpected behavior for MPI applications with significant code after `MPI_FINALIZE`.

Other aspects of the TM interface present issues for implementors of middleware but do not appear to exist for any technical reason. For example, additional work must

be performed by the middleware to map VPU to physical nodes. Providing interface functions or data structures to provide this information directly would be beneficial to application writers.

TM requires that the full path to the executable be given to `tm_spawn()`. LAM must therefore be installed in the same location on all nodes in the cluster. Multi-architecture support is therefore more difficult if LAM is installed on a networked filesystem shared across the cluster. This is in contrast to the `rsh` boot module, where the `$PATH` environment variable is used on each node to find the relevant executables. This allows LAM/MPI to be installed in different locations on each node, enabling multi-architecture support through the use of node-specific path settings. However, most PBS installations are homogeneous clusters, so this limitation is not seen as unworkable.

Finally, a `tm_spawn_multiple()`, similar in functionality to `MPI Spawn Multiple`, potentially allowing for a faster, more direct launch mechanism. This was suggested in the PSCHED API, but was not implemented in PBS's TM API.

### 4.3 Results

All the modules listed in Section 4.2 were implemented. LAM 6 could only natively launch processes in `rsh/ssh` environments; a single installation of LAM 7 can natively launch processes in BProc, Globus, `rsh/ssh`, and PBS environments. Without this capability, not only are different installations of MPI required to run in parallel, but also different implementations (this is the current state of the art; see Section 1.1). Although MPICH provides some level of scripting support for choosing a different job-launching mechanisms, it is aimed at support for launching with different communication devices, not necessarily different run-time systems. This is one of the reasons that the `mpiexec` project was implemented as a standalone utility and not part of MPICH; the infrastructure in MPICH did not adapt well to the native PBS run-time environment. Support for a small

number of run-time environments are supported (e.g., Globus), but typically only where the run-time environment is related to the communications device.

#### 4.3.1 Correctness

Each `boot` module was checked for functional correctness. Correctly-functioning `boot` modules start a LAM universe and seed it with meta-data (e.g., peer LAM daemon location information, CPU counts, etc.). `boot` modules can also be used to grow and shrink a LAM universe. Note, however, that `boot` modules are *not* part of the actual run-time of the universe – they are only responsible for *establishing* the universe. Hence, testing the functionality of a `boot` module can be limited to attempting to launch, grow, and shrink a LAM universe, and then testing to see if the universe is functioning correctly after the `boot` module completed its work.

Although each module has different capabilities (corresponding to its back-end run-time environment), the following actions were tested on the modules as relevant:

1. **Test 1:** Boot a LAM universe with the `lamboot` command.

Successful completion of the `lamboot` command means that 1) all LAM daemons were successfully started, b) each communicated their identity back to `lamboot`, and c) each received a broadcast of the union of their peers' location information.

2. **Test 2:** Verify that the universe is functioning.

The `lamnodes`, `tping`, `mpirun`, and `lamexec` commands were used for this test. The `lamnodes` command queries the local LAM daemon for a list of all the LAM daemons (and corresponding daemon attributes) in the universe and displays them. If `lamnodes` correctly displays this information from any node in the universe, then that LAM daemon correctly received all the data originally sent from `lamboot`.

The `tping` command “pings” each LAM daemon, soliciting a response from its central engine. Successful completion of the `tping` command means that every LAM daemon in the universe has entered its main processing loop and is responding to network requests.

The `mpirun` and `lamexec` commands launch MPI and non-MPI processes across the LAM universe, respectively. Successfully running either of these commands (e.g., “`lamexec N uptime`”) means that the `mpirun`/`lamexec` command was able to communicate with the LAM daemon on each node and tell it to launch a command (“`uptime`”). A variety of LAM daemon services are involved

in this action; if it succeeds, the LAM daemons are in their main processing loops and correctly processing several different kinds of network requests.

3. **Test 3:** Remove a node from the universe with the `lamshrink` command.

This action involves editing the peer and routing tables in each LAM daemon. The `lamshrink` command must contact each LAM daemon and tell it to remove an entry from its tables. Once complete, Test 2 is used to verify that the universe is still functioning correctly. If it is, it means that the `boot` module correctly deleted entries from each LAM daemon's tables.

4. **Test 4:** Add a node to the universe with the `lamgrow` command.

The `lamgrow` command performs essentially the same actions as `lamboot`, but after a universe has already been established. If `lamgrow` succeeds, it means that a new LAM daemon has been launched, it has called back to `lamgrow` to identify itself, and has received a list of its peers. Additionally, the `boot` module in `lamgrow` sends an updated table entry to all other LAM daemons in the universe. Upon successful completion, Test 2 is used to check that the universe is functioning correctly.

5. **Test 5:** Verify external data, such as run-time statistics.

Some modules, such as the `tm` module, provide additional services. As described in the testing section below, the `tm` module allows the PBS run-time system to track statistics of the processes that it launches. Upon successful completion of the universe, PBS can be queried to see how much CPU time and memory was used. These values should match the sum of all jobs launched in the LAM universe.

A summary of these tests is listed in Table 4.1. “√” indicates that a test passed; “NA” indicates that a test did not apply to a given module. Each test system is described in detail in the text of its corresponding module's subsection.

#### 4.3.2 The `bproc` Module

The `bproc` module was tested on the University of Pennsylvania's testbed Liniac cluster, described in Table 4.2.

The Clubmask [66] system was used to reserve nodes, and `lamboot` was used with a hostfile consisting of the reserved nodes and the BProc head node. A LAM daemon was successfully launched and moved to each target node, as confirmed by manually examining the process list. The `lamnodes` command confirmed that all nodes in the hostfile were successfully in the universe; `tping` was able to contact each LAM daemon.

TABLE 4.1

Summary of results

	Test System	Test 1	Test 2	Test 3	Test 4	Test 5
bproc	Liniac cluster	✓	✓	✓	✓	NA
globus	IU CS workstations	✓	✓	✓	✓	NA
rsh	AVIDD-B cluster	✓	✓	✓	✓	NA
tm	AVIDD-B, Thumb clusters	✓	✓	NA	NA	✓

TABLE 4.2

Description of the University of Pennsylvania Liniac testbed cluster

Number of nodes	4
Processor type	Pentium III
Processor count	2
Processor speed	1.13 GHz
Cache size	512 KB
RAM	2 GB
Operating system	Red Hat 7.2 (plus updates)
Linux kernel version	2.4.20
Compiler	GNU, v2.96
Compiler flags	-O3 -pthread
Interconnects used	Gigabit Ethernet
Other relevant software	Clubmask 0.6, BProc 3.2.6

TABLE 4.3

Description of the Indiana University Computer Science `bitternut` and `sawtooth` nodes

	<code>bitternut</code> node	<code>sawtooth</code> node
Processor type	Pentium III	Pentium IV
Processor count	2	1
Processor speed	900 MHz	1.7 GHz
Cache size	2.0 GB	512 KB
RAM	2.0 GB	512 MB
Operating system	Red Hat 7.2 (plus updates)	Red Hat 7.3 (plus updates)
Linux kernel version	2.4.9	2.4.18
Compiler	GNU, v2.96	GNU, v2.96
Compiler flags	-O3 -pthread	-O3 -pthread
Interconnects used	Fast Ethernet	Fast Ethernet
Other relevant software	Globus Toolkit v2.2	Globus Toolkit v2.2

Similarly, `lamshrink` was successfully able to remove a node from the universe and `lamgrow` was able to add a node into the universe. The results of each were tested with the `lamnodes` and `tping` commands to verify correct functionality.

#### 4.3.3 The globus Module

The `globus` module can only utilize the “fork” scheduler in Globus systems, and was only tested between two machines each running a Globus Gatekeeper on the Indiana University Computer Science local area network. The descriptions of these machines are listed in Table 4.3

`lamboot` was used with a hostfile consisting of the contact strings for `sawtooth` and `bitternut`. A LAM daemon was successfully launched on each target node via `globus-job-run`, as confirmed by manually examining the process list on each node.

The `lamnodes` and `tping` commands were used to verify that all nodes were in the universe and were responding properly. Similarly, `lamshrink` was successfully able to remove `sawtooth` from the universe, and `lamgrow` was able to add it back into the universe. The results of each were tested with the `lamnodes` and `tping` commands to verify correct functionality.

#### 4.3.4 The `rsh` Module

The `rsh` module was tested on the Indiana University Analysis and Visualization of Instrument-Driven Data / Bloomington (AVIDD-B) cluster. All functions were tested with both the `rsh` and `ssh` command. The AVIDD-B cluster is described in Table 4.4. Resource accounting tests were performed on the Indiana University Open Systems Lab Thumb cluster, also described in Table 4.4.

As described in Section 4.2.3, the `rsh` module is a port of the mature `rsh`-based implementation from LAM/MPI version 6. Testing was comprised of booting different numbers of nodes on AVIDD-B using a hostfile. In all cases, a LAM daemon was successfully launched on each target node via `rsh` (`ssh` was tested similarly), as confirmed by manually examining the process list on each node. `lamnodes` and `tping` verified that all nodes existed in the universe and were responding properly. Similarly, `lamshrink` was successfully able to remove random nodes from the universe; `lamgrow` added them back. `lamnodes` and `tping` were used to verify the stability of the universe after each step.

#### 4.3.5 The PBS `tm` Module

The `tm` module was tested on the same AVIDD-B cluster as the `rsh` module.

Testing comprised of requesting jobs from PBS with varying sizes (from 1 to 96 nodes) and running `lamboot`. No hostfile was specified; the list of nodes was directly obtained from PBS's TM interface. In all cases, a LAM daemon was success-

TABLE 4.4

Description of the Indiana University AVIDD-B and Open Systems Lab Thumb clusters

	AVIDD-B cluster	Thumb cluster
Number of nodes	96	4
Processor type	Xeon	Pentium III
Processor count	2	1
Processor speed	2.4 GHz	730 MHz
Cache size	512 KB	256 KB
RAM	2.5 GB	512 MB
Operating system	Red Hat 7.3 (plus updates)	Red Hat 9 (plus updates)
Linux kernel version	2.4.20-29.7-smp	2.4.20-28.9
Compiler	GNU, v2.96	GNU, v3.2.2-5
Compiler flags	-O3 -pthread	-O3 -pthread
Interconnects used	Gigabit Ethernet, Myrinet (GM library v1.6.5)	Fast Ethernet, Myrinet (GM library v2.0.9)
Other relevant software	Torque v1.0.1p5, Maui Scheduler [67, 68] v3.2.6p6	OSCAR 3.0, OpenPBS v2.3.16 (OSCAR), Maui Scheduler v3.2.5p2 (OS- CAR)

fully launched on all target nodes and a LAM universe was established. `lamnodes` and `typing` were used as with testing the previous `boot` modules; both showed that the LAM universe was functioning properly.

The `tm` module does not support `lamshrink` and `lamgrow` on the rationale that in a job allocated by a static batch environment such as PBS, there is little need to shrink and grow a LAM universe.

### Faster Startup

A side-effect of using the TM interface is that the LAM RTE startup time using the `tm` `boot` module is significantly faster than using the `rsh` module, as shown in Figure 4.7. Faster startup (compared to the traditional `rsh/ssh` method) is beneficial in terms of scalability, and is therefore important to the end user.

There are two reasons for the decrease. First, `rsh` must authenticate every time a remote process is launched, whereas the MOMs only authenticate at startup (`ssh` is particularly slow because of its complex authentication algorithms). Second, the `tm` `boot` module uses the linear windowed algorithm. The `tm` module is capable of sending control messages out serially, but does not wait for the response before sending the next control message.

### Process Accounting

With `rsh` or `ssh`, PBS is only able to log the resources used by the processes on the first node in the job. The use of the TM interface allows PBS to generate accurate process accounting for the entire job. Using the accounting logs provided by the PBS server, CPU time and memory usage can be tracked in addition to wall-clock time. Table 4.5 provides resource information for the execution of a test program that runs a computational kernel for approximately 35 seconds. The accounting information in the Table is from PBS on the Thumb cluster comparing a job run on 1 node and 4 nodes. It shows almost identical

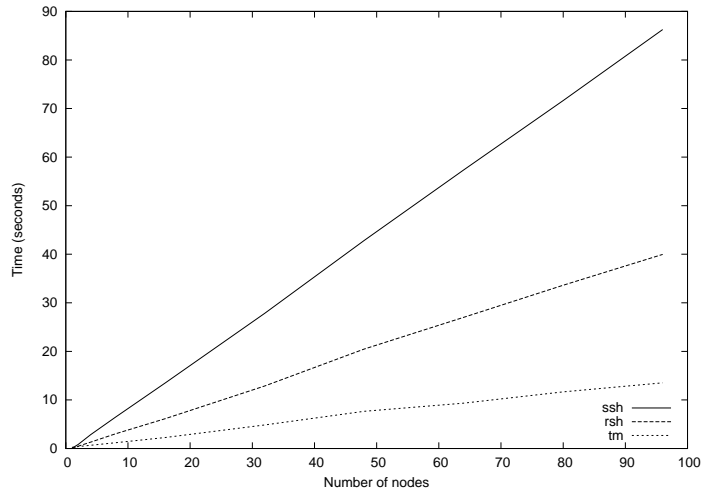


Figure 4.7. Execution time for `lamboot` in seconds using `ssh`, `rsh`, and `TM` on the AVIDD-B cluster.

results between the single node jobs for the `rsh` and `tm` modules. For the four node jobs, the `rsh` module shows results similar to the one node jobs; the main difference is the 6 extra seconds necessary to execute `ssh` to three remote nodes. For the `tm` module, the wall clock time is approximately the same, but the CPU time and memory usage is roughly four times that of the single node `tm` job, showing that PBS was able to accurately measure the CPU time for the entire job.

### Resource Cleanup

Through both the use of the PBS MOMs to start applications and the `SIGTERM` handling in the LAM daemons, LAM is able to properly return resources to the system after job execution. Several scenarios were tested on the Thumb cluster, including the following:

- Allocate a PBS job, run `lamboot` in the PBS job, run an MPI job to completion, run `lamhalt` to take down the LAM universe, and exit the PBS job.

TABLE 4.5

Accounting information from PBS on the Thumb cluster

Module	Number of nodes	Wall clock time	CPU time	Memory
rsh	1	36s	34s	2,228 KB
tm	1	35s	34s	2,992 KB
rsh	4	41s	35s	2,800 KB
tm	4	37s	135s	10,488 KB

- Allocate a PBS job, run `lamboot` in the PBS job, run an MPI job to completion, and exit the PBS job (with the LAM universe still running).
- Allocate a PBS job, run `lamboot` in the PBS job, run an MPI job, and exit the PBS job (with the MPI job and LAM universe still running).
- Allocate a PBS job, run `lamboot` in the PBS job, and exit the PBS job (with the LAM universe still running).

In all cases, the `tm` module is able to catch the `SIGTERM` sent by PBS and kill any MPI jobs and cleanly shut down the LAM universe.

## CHAPTER 5

### MPI POINT-TO-POINT COMMUNICATION

By definition, MPI libraries are intrinsically centered around passing messages between MPI processes. Point-to-point latency and bandwidth measurements are two metrics that are commonly used to rate the efficiency of an MPI implementation. While these are certainly not the only relevant metrics of performance, they are a good indicator of how strongly the high-performance computing community values efficient and robust communication libraries. Communication-intensive parallel applications tend to be sensitive to latency and bandwidth constraints and will perform poorly if the MPI implementation cannot deliver efficient message passing.

Although commodity fast and gigabit Ethernet networks are still commonplace, proprietary, high-speed network interconnects have risen in popularity in parallel computing environments. Among the more popular include Myrinet, Quadrics, and Infiniband. It is not uncommon for organizations to have parallel resources with one or more of these interconnects. Previous monolithic MPI implementations (including LAM/MPI) could only support one underlying network at a time, leading to significant logistical issues for users and ISVs.

In order to achieve high performance, an MPI implementation must utilize low-level interfaces and protocols as close to the underlying interconnection hardware as possible. Modern high-speed interconnects provide companion software libraries that deliver low latency and high bandwidth (among other characteristics). A well-architected MPI imple-

mentation can utilize these libraries in a modular fashion; incorporating new underlying communication transports should require little or no changes in the upper layers of the MPI implementation and cause no performance degradation due to additional abstraction layers.

This chapter describes the `rpi` component type in LAM/MPI, the back-end component framework for LAM's point-to-point MPI functionality. "rpi" is an acronym for Request Progression Interface, reflecting its central philosophy of tracing the life of a point-to-point MPI request from creation to destruction. Similar to the other component frameworks in LAM/MPI, the `rpi` framework supports the dynamic loading of multiple modules at run-time, thereby addressing many of the issues outlined above. Appendix C provides a detailed technical reference of the `rpi` component framework.

**Acknowledgements:** The `rpi` component interface design was strongly influenced by the point-to-point design in LAM/MPI v6.5. The `crtcp` module (described more fully in Chapter 7) is an adaptation of the `tcp` module, and was initially created by Sriram Sankaran. The `tcp`, `lamd`, `sysv`, `usysv` modules were implemented by me based on their corresponding RPI implementations in LAM v6.5. The design and implementation of the component architecture and the implementations of the `gm` module both represented entirely new work that was performed by me.

## 5.1 Design

The core abstractions in the `rpi` component framework revolve around the life of a point-to-point MPI request: its creation, initialization, advancement, and destruction. This design creates a natural mapping between top-level MPI API functions and the back-end implementation – an essentially one-to-one mapping of MPI functionality to the back-end module. This allows great flexibility in module implementation abstractions. The `rpi` also includes other utility functionality such as "special" memory management

and checkpoint/restart support. Although multiple `rpi` modules can be available at run-time, only one `rpi` module will be selected to be used within an MPI process; all MPI processes in a parallel application must select the same module. Hence, the scope of an `rpi` module's selection is the duration between `MPI_INIT` and `MPI_FINALIZE`.

The `rpi` component type is used to perform point-to-point message passing between MPI processes. It accepts MPI requests from the MPI layer and passes the associated messages to the destination process (including, potentially, itself). It also accepts messages from peer processes and passes them up to the MPI layer when a matching receive request is found. Note that the MPI layer does not know (or care) how messages move between processes; all it knows is that a request was created, progressed, and completed. This abstraction barrier gives the `rpi` module complete control over how it effects sending and receiving.

High level operations such as buffer packing/unpacking, handling of buffers for buffered sends, and message data conversion are handled by the MPI layer. The `rpi` module is therefore only responsible for moving messages between MPI processes across its target communications architecture. This philosophy is in accordance with one of the core definitions of components: a module is intended to be focused on a specific purpose and/or architecture. In this case, each `rpi` module targets a specific communications architecture.

### 5.1.1 Prior Implementation

As described in Section 3.1.2, prior versions of LAM/MPI had a compile-time system that supported multiple RPI implementations. The API for RPI implementations was well-defined and therefore was a logical choice to be the first component designed in LAM/MPI. As such, the RPI design was converted into the `rpi` component framework. In addition to using component methodology, the `rpi` framework incorporated new functionality that was not part of the original RPI design: memory management and check-

point/restart support.

### 5.1.2 Data Structures

The MPI layer maintains three data structures that are shared with the selected `rpi` module: the process list, MPI requests, and the active request list. Although `rpi` modules can cache information in these data structures, most of the data members are managed by the MPI layer. The `rpi` module must therefore treat these data members as read-only.

#### Process List

Each process in a LAM universe can be uniquely identified by its *Global Process Space* (GPS) location information. MPI processes identify peer processes by their GPS data in conjunction with additional MPI-specific attributes in a *process entry*. To enable MPI-2 dynamic functionality, MPI processes are therefore always identified by their process entry – not their rank in `MPI_COMM_WORLD` (since there may be multiple `MPI_COMM_WORLD` instances) [53]. Hence, the source and destination rank arguments in top-level MPI functions are always resolved to corresponding process entries. The `rpi` module, therefore, is oriented towards communicating with process entries, not communicators and ranks.

During `MPI_INIT`, each MPI process sends its GPS location information back to `mpirun`. A GPS instance is a unique locator used for identifying LAM processes in the universe, usually in conjunction with LAM's out-of-band communication system. Once `mpirun` receives GPS data from all MPI processes that it launched, it broadcasts the union of data to all MPI processes. Each MPI process uses this GPS data to create the initial entries in its process list.

Sets of process entry references are contained in MPI groups. MPI groups, in turn, are contained in MPI communicators. Specifically, the GPS data from `mpirun` is used for the initial seeding of entries in the process list. This list is then used to create an MPI

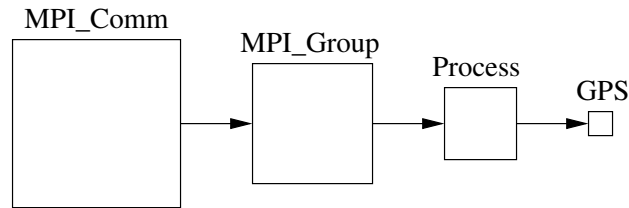


Figure 5.1. An `MPI_Comm` contains an `MPI_Group`. The `MPI_Group` contains a set of references to entries in the process list. Each process entry contains a `GPS`.

group, which is then used to create `MPI_COMM_WORLD`. A similar process is used to create `MPI_COMM_SELF`. This data hierarchy is shown in Figure 5.1.

`MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE` are similar to `mpirun`. Both are invoked collectively over a communicator. The process corresponding to rank 0 in the communicator performs the role of `mpirun`: it launches new processes, collects `GPS` data from them, and then broadcasts the collated set. Note that since the children need to become aware of their parents, the spawning process includes the parent's `GPS` data in the broadcast (`mpirun`, by definition, has no parent processes; it includes a zero-length set of parent `GPS` data in its broadcast). The children create process entries from the `GPS` data (including entries for their parents). Three groups (and communicators) are therefore created from this set: `MPI_COMM_WORLD` and `MPI_COMM_SELF` (as described above), and the communicator that is returned if the children invoke `MPI_COMM_GET_PARENT`. The spawning parent will then broadcast children's `GPS` data to its peers in the spawning communicator. The parents create entries in their process lists for the children, and form a group which is then used to create the intercommunicator that is returned from `MPI_COMM_SPAWN[_MULTIPLE]`. Similar procedures are used for `MPI_COMM_ACCEPT`, `MPI_COMM_CONNECT`, and `MPI_COMM_JOIN`.

Processes are always added to the process list in sorted order. The sorting criteria is unimportant; all that matters is that the same sort algorithm is used in all processes. For

example, at the end of `MPI_INIT` all processes in `MPI_COMM_WORLD` have exactly the same process list. Similarly, at the end of MPI-2 dynamic functions, although all processes may not have the exact same process list, the entries that are common will have the same relative ordering. Since spawning (either via `mpirun` or MPI-2 dynamic functions) is always a synchronous action, and the relative ordering of process lists is guaranteed to be identical in all connected MPI processes, distributed blocking algorithms can be utilized without deadlock. A common example of this is the pairwise setup of point-to-point communication channels; as long as all connected processes follow the same order of process connection, deadlock will not occur.

Each process entry has a generic pointer that can be used by the selected `rpi` module to cache peer-specific information. For example, in a TCP-based `rpi` module, this information may include the IP address and port number that the peer is listening on, and a file descriptor for a socket opened to that peer.

## MPI Requests

An MPI request is created for most send and receive communications. It contains all the parameters from the top-level MPI API call as well as fields for internal tracking, status, flags, a reference to the process entry of the peer, etc. The request also has a generic pointer where `rpi` modules can cache arbitrary data. For example, a TCP-based `rpi` module may need to keep a count of how many bytes have been sent or received so far on this request.

## Request List

The request list is a global list of all currently *active* requests. An active request is defined as one that is in the process of sending, receiving, or waiting to be sent or received. Most requests are immediately moved into the active state, but persistent requests, for example, are not. Persistent requests are initially set in the *start* state; they are moved

into the active state after they have been passed to `MPI_START`. Any request that enters the active state is placed on the request list. It is the `rpi` module's responsibility to identify new requests on the list and start progressing them.

The `rpi` module does not modify the request list itself; it can only modify the data that it has cached on process entries and requests, and the state field on the request. The MPI layer will detect changes in the state field and remove the request from the active list when appropriate.

### 5.1.3 Module Selection and Initialization

An `rpi` module is selected in a manner similar to the algorithm outlined in Section 3.2.7. Each available module's query function is invoked to find out if the module wants to run and its associated priority. Modules that want to run are ranked in priority; the module with the highest priority is selected. All other modules are closed and finalized.

The selected module has its initialization function invoked. It accepts a list of peer process entries (based on the GPS data received from `mpirun` or one of the spawning parent processes). Most modules simply invoke the "addprocs" function with the process list (see below).

The module also returns the maximum communicator ID (CID) and tag value that it allows. These values are propagated to the relevant areas in the MPI layer for error checking purposes. Most modules will be able to support the full range of CID and tag values (i.e., the maximum integer value supported by the type), but at least one of LAM's `rpi` modules packs the CID and tag into a bit-mapped field, and therefore must constrain their values.

#### 5.1.4 Adding and Removing Processes

When a process entry is added to or removed from the process list, the selected `rpi` module must be informed. Two module API functions must be provided for this purpose: “`addprocs`” and “`rmprocs`,” respectively. Modifying the process list is a synchronous action (i.e., it is acceptable to block); it will only happen during `MPI_INIT`, `MPI_FINALIZE`, or the MPI-2 dynamic function calls. Process entries are reference counted; their inclusion in a group increments the reference count. When all groups containing a process are freed (e.g., when connected processes disconnect), the process entry’s reference count will go to zero, naturally triggering cleanup procedures for relevant data structures.

The `addprocs` function does whatever the `rpi` module needs in order to setup point-to-point communication. That may involve exchanging some data using LAM’s out-of-band communication system, opening additional communication channels, or simply saving location information for later use. The `rpi` module typically caches information on the new process entry as a result of the initial setup. Since adding processes is a synchronous action and all processes have the same ordering in their process list, it is known that all peers will be adding processes simultaneously, and blocking actions (such as opening sockets) can proceed in a known, deadlock-free order.

Similarly, the `rmprocs` function frees any resources associated specifically with the process entry and clears any cached data.

#### 5.1.5 Request Lifecycle

As the name “Request Progression Interface” implies, the `rpi` component interface follows the life of an MPI request. Movement of bytes in a message attached to a request is therefore a side-effect; it is not directly part of this API.

A request’s life follows this cycle (shown in Figure 5.2):

- **Building.** Storage is allocated and initialized with request data such as datatype, tag, etc. The request is placed in the *init* state. It is not to be progressed and is

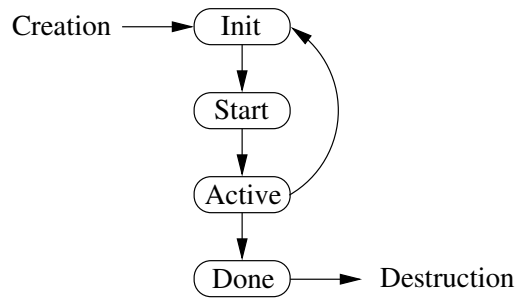


Figure 5.2. State progression of an MPI request. After a request is created, it is automatically put in the *init* state. Note that upon finishing the *done* state, non-persistent requests are destroyed, but persistent requests are moved back to the *init* state.

therefore not linked into the active request list.

- **Starting.** The request is now made a candidate for progression and is linked into the active request list.

It is not necessary at this stage for any data transfer to be done by the *rpi* module, but this is not precluded. All that is required is that the request's progression state be correctly set. Depending on the *rpi* module and the circumstances, the request will be moved into the *start*, *active*, or *done* state.

- **Progression.** The request is progressed in stages until it is complete. The request is moved from the *start* state to the *active* state as soon as any data is transferred. It is moved from the *active* to the *done* state once all data is transferred and all required acknowledgments have been received or sent.
- **Completion.** When completed, the request is either reset to the *init* state ready for restarting (if persistent) or destroyed (if non-persistent).

### 5.1.6 Progression

Each module has an *advance* function that is the main entry point into its progression engine. When the advance function is invoked, the *rpi* module generally performs the following actions:

- Looks for new items in the active request list and starts them.
  - For send requests: Because of MPI's strong message ordering guarantee, messages may not get sent immediately, but instead queued up behind other messages that are awaiting transmission.

- For receive and probe requests: Search through messages that have been received but not yet matched with a request. If a match is found, either complete the request (if the message has already been fully received) or mark it in the active state and let it progress normally (if the message has not yet been fully received).
- Attempts to progress any send requests in the active state. Continue progressing any messages awaiting transmission, and/or check for completion of previously sent messages.
- Checks for and handles incoming traffic to progress receive requests in the active state. Match incoming messages with pending receive requests and receive the message data into the corresponding request's buffer. Incoming messages that do not match any pending requests are *unexpected*, and are queued separately.

Each module will likely perform these actions differently, depending on the nature of the underlying communications architecture.

#### 5.1.7 “Fast” Send and Receive

An optimization is included in the top-level `MPI_SEND` and `MPI_RECV` functions: if the active request list is empty when these functions are invoked, and if the `rpi` module supports it, the corresponding *fast* send or receive function will be invoked on the module.

No request is made for fast sends and receives; the entire queue structure is bypassed. The overriding assumption is that since there is nothing else pending transmission or reception, this message can be sent or received directly, thereby avoiding some latency. Depending on the underlying communications architecture, fast functions may require considerable logic to be able to handle partial sends, interleaved receives, rendezvous protocols, etc. For example, the `tcp rpi` module included in the LAM/MPI distribution has fairly lengthy fast send and receive implementations. This is justified because it noticeably decreases the latency of eager messages, particularly when both sender and receiver use the fast implementations.

### 5.1.8 Memory Management

MPI-2 introduced two memory allocation functions: `MPI_ALLOC_MEM` and `MPI_FREE_MEM`. As described in the MPI-2 standard, message passing and remote memory access operations run faster in some systems when accessing specially allocated memory. These functions allow the MPI implementation to provide the user with “special” memory that can be used for these purposes.

The `rpi` interface has two back-end functions to handle these MPI calls. If the selected `rpi` module supports these functions, they are invoked when the corresponding top-level MPI functions are called. It is the module’s responsibility to perform any tracking necessary for special memory allocated and freed by these functions, as well as recognizing the ability to leverage the use of such special memory in data transfer operations (i.e., distinguish “special” memory from “regular” memory). If the `rpi` module does not provide these functions, the normal C library `malloc()` and `free()` are used.

### 5.1.9 Checkpoint / Restart Functionality

`rpi` modules can insert arbitrary functionality during relevant phases of LAM’s checkpoint/restart procedures. Details of how the `rpi` component framework is used in checkpointing are described in Chapter 7.

### 5.1.10 Module Finalization

The “`rmprocs`” interface function mentioned in Section 5.1.4 is actually a special case of module finalization. The `rpi` module’s `finalize` interface function accepts a pointer to a process entry. If this pointer is non-NULL, the `rpi` module must only finalize its use of that process entry. This will only happen for MPI-2 dynamic processes; entries that represent peer processes in `MPI_COMM_WORLD` will not be finalized until `MPI_FINALIZE`.

During `MPI_FINALIZE`, the finalization function is invoked with a NULL argument,

signifying that all MPI point-to-point communication is now complete and the process is ending. The module can clean up all process entry references that it still has, shut down any network connections, release resources, etc. This is the last function invoked before the module is closed.

## 5.2 Implemented Modules

Six `rpi` modules are included in LAM/MPI version 7: `lamd`, `tcp`, `crtcp`, `sysv`, `usysv`, and `gm`. Several of these modules were ported from previous versions of LAM/MPI (`lamd`, `tcp`, `sysv`, and `usysv`). These previous RPI implementations needed to be adapted to the `rpi` component framework and have their functionality augmented to include new interface functions that were not previously supported. The `crtcp` module is a modification of the `tcp` module and is described below. The `gm` module is new to LAM version 7; it was implemented solely within the `rpi` component framework.

Although all the modules are implemented differently, they each have at least some common characteristics. For example, most make the distinction between short messages (usually sent eagerly) and long messages (usually sent with a rendezvous protocol, potentially using RDMA). These modules serve both as reference message passing algorithms as well as verification of the `rpi` module interface design.

### 5.2.1 The `lamd` Module

The `lamd` `rpi` module uses the LAM out-of-band communication mechanism for MPI communication. In prior, monolithic implementations of LAM/MPI, daemon-based communication was the only mechanism available for MPI messages. It has been ported to all new versions of LAM/MPI since then (including converting it to an `rpi` module) mainly as a reference implementation for the RPI abstraction.

Using the `lamd` module, MPI messages originate in a source process and are sent

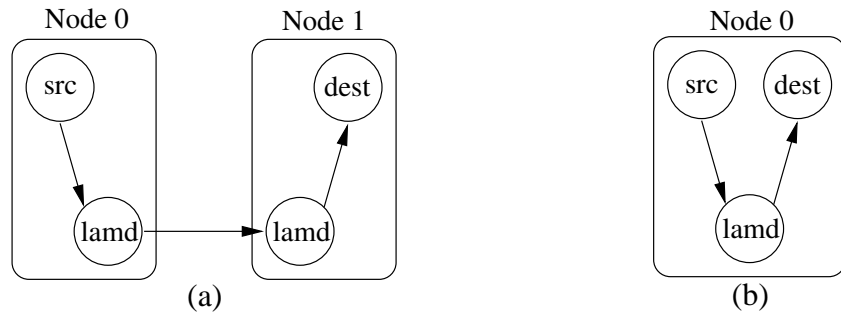


Figure 5.3. MPI message passing using the `lamd rpi` module in two scenarios: (a) when the source process is on node 0 and the destination process is on node 1, and (b) when the source and destination processes are on the same node.

first to the local LAM daemon, then to the destination LAM daemon, and finally to the destination process. When the source and destination processes are on the same machine, then the local and destination LAM daemon are effectively the same – only one LAM daemon is involved. Figure 5.3 shows these scenarios graphically. While this model does create additional latency by adding network hops, it has been maintained in LAM because it offers true asynchronous message passing progress.

All messages sent from process to the local LAM daemon are sent eagerly. Similarly, when the message has been fully transmitted to the destination LAM daemon, the destination process eagerly receives it in a single transfer. This model allows the MPI library to send an entire message to the local LAM daemon and then return to the user application while the LAM daemon progresses sending the message to the remote LAM daemon.

A limitation of the `lamd rpi` module is that due to the nature of LAM’s out-of-band messaging, MPI tags and communicator identifiers are bit-packed into fixed-width fields. These fields are relatively small and are unlikely to be changed due to their wide usage throughout the LAM code base. This directly impacts both the maximum values that can be used for tags and the number of communicators that can be used in an MPI process. Although both values are within the limits specified by the MPI standard, some MPI

applications make implicit assumptions about arbitrarily large integer tag values that will cause run-time errors with the `lamd` module.

Although clearly not suitable for all applications, the `lamd rpi` module can provide high performance for parallel applications that rely on asynchronous message passing progress and are not sensitive to latency.

### 5.2.2 The `tcp` and `crtcp` Modules

Originally written in 1996, the TCP-based MPI point-to-point was the second RPI implementation to be written. It pioneered LAM's use of "client-to-client" (so called "c2c") communication – the concept of sending MPI messages directly from one process to another without an intermediary agent (such as a LAM daemon).

This led to the initial abstractions to allow compile-time switching, which later evolved into limited support for run-time switching between the TCP and LAM daemon implementations. Both the LAM daemon and c2c implementations could be compiled in to a single MPI process and a run-time switch chose which to use. This approach worked well, but the interfaces to the implementations were not identical, forcing many special cases in the MPI layer to decide which RPI implementation to call, different handling for the two different message passing models, and other internal accounting issues. As such, it was not a pure component-based approach.

### Socket Management

LAM's `tcp rpi` module implements MPI point-to-point message passing over TCP sockets. During `MPI_INIT`, the `tcp` module uses LAM's out-of-band communications to exchange TCP port numbers between peer processes and open sockets accordingly. At the end of `MPI_INIT`, therefore, each MPI process has a TCP socket open to every other MPI process. A similar procedure occurs during the MPI-2 dynamic functions (`MPI_COMM_SPAWN`, `MPI_COMM_SPAWN_MULTIPLE`, `MPI_COMM_ACCEPT`, `MPI-`

COMM\_CONNECT, and MPI\_COMM\_JOIN); TCP sockets are opened between newly-connected MPI processes.

Maintaining open TCP sockets to all MPI processes is beneficial for two reasons. First, it is convenient from an implementation standpoint (there is no need to handle asynchronous incoming socket connection requests). Second, all setup overhead is incurred during MPI\_INIT and the MPI-2 dynamic functions; MPI message passing is never delayed by socket rendezvous and initialization. However, this approach does have the notable drawback of creating scalability issues for large parallel applications. For example, most operating systems only support a fixed number of file descriptors per process. In order to scale to thousands of MPI processes, operating system parameters may need to be adjusted to increase the number of file descriptors in each process. The number of file descriptors should not be increased to be arbitrarily large since each descriptor requires memory and resources – a balance needs to be found for a given application, target environment, and operating system.

The main difficulty in a non-blocking TCP message passing system is the fact that TCP supports partial reads and writes. This forces single-threaded message passing systems (such as LAM/MPI) to have a polling-based, re-entrant state machine that maintains the state of messages with relation to the socket that they are traveling across. A single message may take multiple iterations through the state machine before it is actually transferred.

#### The crtcp Module

A second TCP-based module was implemented when checkpoint/restart support was added to LAM/MPI: crtcp. The crtcp module is identical to the tcp module except that it supports checkpointing and restarting. Specifically, it will react to incoming checkpoint signals and drain the network of messages before allowing the checkpoint to continue.

Additionally, upon restart (in a new process), it will mark all sockets as stale and re-invoke the `addprocs` function to create new connections. `crtcp` is distinct from `tcp` because of slight performance degradation due to additional thread-level locking that is required. Hence, `crtcp` should only be used when checkpoint/restart services are required. LAM's checkpoint/restart services are discussed in further detail in Chapter 7.

### 5.2.3 The Shared Memory `sysv` and `usysv` Modules

The `sysv` and `usysv` modules use TCP for communication with MPI processes on different nodes and shared memory for processes on the same node. The only difference between the two modules is the mechanism used for locking access to shared resources: the `sysv` module uses System V semaphores while the `usysv` module uses spin locks. Semaphores will generally provide better performance when there are more MPI processes on a machine than CPUs; a case where processes must yield to each other. Spin locks obtain better performance when the number of MPI processes on a node is less than or equal to the number of CPUs. In this case, each process will spin in a tight loop waiting for access and will never be forced to yield.

For off-node communication, `sysv` and `usysv` directly invoke the `tcp` module by exploiting knowledge of the `tcp` module's internals. This is an abstraction violation that has existed for several years and was not fixed in the recent conversion to a component-based architecture. For on-node communication, one System V shared segment is shared by all MPI 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 set of six semaphores for each process pair communicating via shared memory. In some environments, 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 one-way communication 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 a configurable module 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 by either a System V semaphore or a spin lock, as described above. The size of this pool is a configurable module parameter. LAM will try to determine a reasonable value for  $P$  at configuration time if none is explicitly specified. Larger values should improve performance (especially when an application passes large messages) but will also consume more system resources.

#### 5.2.4 The Myrinet gm Module

The `gm rpi` provides low latency, high bandwidth message passing over Myrinet networks. It uses the Myricom-provided GM message passing library for communication. The GM library interfaces with a companion kernel module to perform data transfer to and from the Myrinet hardware. GM’s semantics are unfortunately quite different than MPI’s semantics; significant bookkeeping and infrastructure had to be implemented in the `gm` module to effect MPI semantics with GM.

The GM kernel module uses operating system bypass mechanisms to interface directly with RAM; data is directly read from and written to target buffers. This requires that buffers shared with the Myrinet hardware must be “pinned” down in memory – they cannot be allowed to be paged out by the operating system. Hence, when a buffer is pro-

vided to the Myrinet hardware, it must be guaranteed not to move elsewhere in physical memory before the data transfer is complete.

The `gm` module must actively pin all buffers before they can be used with the GM library. Operating systems typically have limits as to how much physical memory can be pinned (either with hard-coded values or resource exhaustion constraints), so the `gm` module must keep track of all pinned memory. Additionally, when pinned resources are exhausted (i.e., when the `gm` module attempts to pin new memory and fails), an LRU cache is employed to un-pin memory that is no longer actively being used for data transfers. Both pinning and unpinning memory are slow operations.

Note that freeing memory that is still pinned causes problems within the Myrinet kernel module. Calls to `sbrk(2)`, the function which returns memory to operating system, must be intercepted. When `sbrk()` is invoked, the `gm` module queried to see if any of the memory to be returned is still pinned. If it is, the module unpins it, and then allows `sbrk()` to continue [76].

The GM message passing model is based on event polling [92]. For example, a sender provides a message buffer to the GM library and then polls an event queue. The Myrinet hardware and kernel module progress the data transfer independent of the originating process and place an event in the sender's event queue when it has completed. Similarly, receivers pass buffers to the GM library and are notified via the event queue when a message has been received.

Recent versions of the GM library have included the RDMA “put” and “get” operations that exhibit slightly lower latency than their send and receive counterparts. The `gm` module currently utilizes these operations for long message transfers.

### 5.3 Results

The `rpi` framework is intended to provide an abstraction layer to implement different network transport mechanisms for message passing. Performance is critical; the component framework cannot be responsible for added latency or loss of bandwidth.

Two main types of tests were run on the implemented `rpi` modules – correctness and performance.

#### 5.3.1 Correctness

For correctness, an `rpi` module has to be able to pass blocking, non-blocking, and persistent messages in all modes of MPI point-to-point communication: standard, buffered, synchronous, and ready. This is a complex feature set, and is difficult to implement; heavy testing is required before a `rpi` module can be considered to be functioning properly.

The basis of a correct test is whether a receiver’s message buffer contains the data that it is supposed to contain (as designated by that test). Specific operations are invoked on input data that are expected to generate specific output. If the output does not match the expected result, the test is ruled to have failed. The tests used were derived from the IBM MPI test suite. Over time, tests have been modified and new tests have been added. All tests have been run in LAM/MPI as well as other MPI implementations to verify that they are correct.

The LAM test suite calls every MPI point-to-point function in a wide variety of scenarios. It invokes every mode of MPI point-to-point communication, in all three flavors (blocking, non-blocking, persistent). Additionally, the suite contains tests of collective operation. LAM’s `coll` modules are implemented on top of point-to-point functionality, and therefore effectively add more tests that stress an `rpi` module. Since the fundamental philosophy behind an `rpi` module is the life of an MPI request, the implementation model that it naturally tends to evoke essentially have a one-to-one mapping with the MPI com-

munication modes (and blocking/persistent characteristics).<sup>1</sup> For example, a buffered send will invoke the general “send” code path in an rpi module with a small special case for the buffered mode. Other tests are used to generate specific MPI behavior unrelated to the communication mode such as unexpected messages, “overtaking” messages, complex message matching, etc.

Coverage analysis was performed using the GNU Coverage tool (“gcov”) on Linux. The `gcov` tool was invoked for every test described above and recorded every line in the LAM source code that was accessed. Table 5.1 shows the percentages of code that were accessed. Note that the percentages are deceptively low; a post-mortem analysis of exactly which lines were accessed showed that the majority of lines not accessed were the result of error condition testing. For example, none of the tests pass bad input to MPI functions (e.g., incorrect datatypes, invalid buffers, etc.), and none of the Unix system and library calls failed unexpectedly during the tests. Hence, these code paths (which mainly consisted of passing an error up the call stack) were not exercised, and this accounted for the seemingly-low coverage percentages. The one exception is the `gm` module – it has a fairly low coverage percentage (56%). This is due to the fact that Solaris-specific code was not activated in the Linux tests (the Myrinet GM library performs differently on Solaris than on any other operating system). Some code paths are also not being stressed, indicating that more tests need to be written for the `gm` module.

### 5.3.2 Performance

For performance, the goal is to ensure that the abstractions added by the componentization in LAM 7 did not significantly affect performance as compared to LAM 6. Since the main point-to-point transport engines essentially did not change between LAM 6 and 7, measuring the performance of each RPI implementation in LAM 6 against its corre-

---

<sup>1</sup>This is, of course, not the only way to implement an rpi module. But all the modules included in the LAM distribution use these kinds of models.

TABLE 5.1

Test code coverage of rpi modules

Module	Coverage
gm	56%
lamd	81%
tcp	77%
sysv	82%
usysv	86%

sponding rpi module in LAM 7 will show the difference in infrastructure overhead.

Note, however, that the implementations are *not* exactly identical between LAM 6 and 7 – other changes were made to the infrastructure and to the individual message transport engines. Most were small and only for portability between different flavors of POSIX, but at least one change had a large effect on overall performance, and therefore must be taken into account.

A `memcpy()` optimization was added in LAM 7 which dramatically increases local memory copy transfer rates (indeed, without the optimization, transfer rates are extremely erratic). This makes the performance of LAM 7 inherently better than that of LAM 6 in most cases, particularly for the shared memory transports. As such, in order to compare only the effects of the componentization in LAM 7, the `memcpy()` optimization was added to the LAM 6 version used for these experiments.

The rpi performance comparisons described below were run with the NetPIPE analyzer [118] on the AVIDD-B cluster described in Section 4.3.4. The entire cluster, including the networks used, was otherwise dormant while the experiments were conducted.

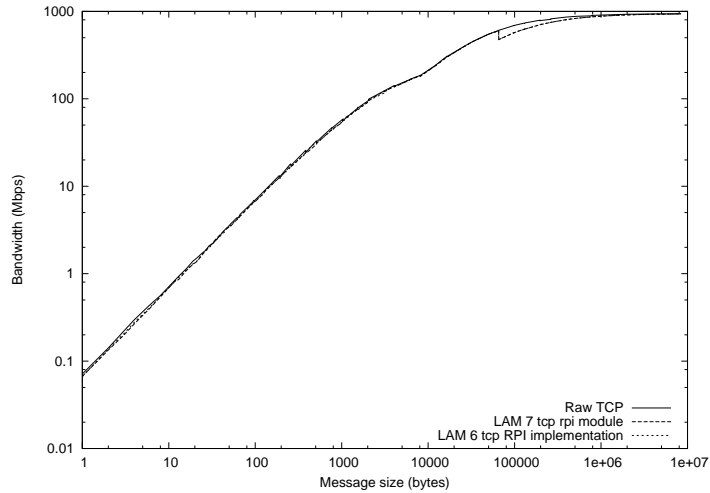


Figure 5.4. Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph compares the performance of raw TCP, the tcp rpi module (LAM 7), and the TCP RPI implementation (LAM 6).

### The tcp and crtcp Modules

The tcp and crtcp modules are identical except for additional bookkeeping required for checkpointing in crtcp. As such, the performance of the tcp module is measured here; the performance of the additional overhead required by the crtcp is measured in Chapter 7.

Figure 5.4 shows ping-pong performance across gigabit Ethernet of raw TCP, the tcp rpi module in LAM 7, and the TCP RPI implementation in LAM 6. The performance is essentially identical across most message sizes. The only noticeable difference is a dip in both the LAM 6 and LAM 7 performance at 64 KB. This corresponds to LAM’s default size to switch between an eager send and a rendezvous protocol, suggesting that the crossover size should be increased to accommodate the high bandwidth and low latency of the gigabit Ethernet.

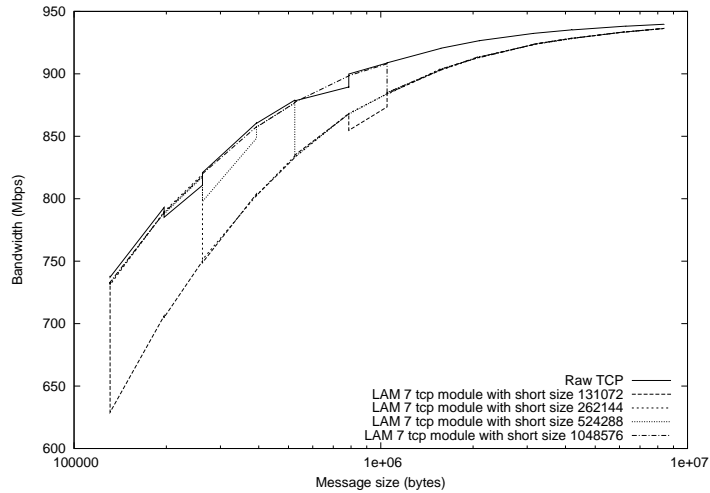


Figure 5.5. Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph shows the effect of increasing the eager/rendezvous message to increase performance of messages between 128 KB and 1 MB in the tcp rpi module.

The tcp rpi module has a run-time parameter allowing the short/long boundary size to be changed. Figure 5.5 shows the effects of increasing this boundary from 128 KB to 1 MB. In an attempt to be easier to read, the graph shows only messages sizes starting with 128 KB. The Figure shows that as the short message boundary increases, the bandwidth also increases to be almost identical to raw TCP.

Figure 5.6 shows the absolute difference between LAM 7 and LAM 6 ping-pong measurements. Figure 5.7 shows the same values, but expressed as a percentage difference. Short messages ( $\leq 10$  KB) exhibit variances of up to  $\pm 6.5\%$ , but these percentages actually reflect small values and are likely due to experimental noise. For example, in Figure 5.7, the largest difference is at message size of 509 bytes: LAM 7 reports 30.0 Mbps while LAM 6 reports 32.1 Mbps – a difference of 2.1 Mbps. Indeed, this experiment was actually run multiple times – similar graphs showing approximately  $\pm 6.5\%$

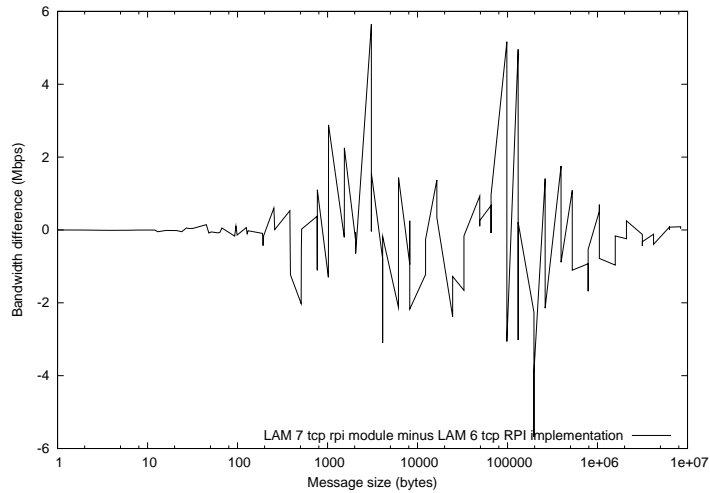


Figure 5.6. Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph shows the absolute bandwidth difference between the the tcp rpi module (LAM 7) and the TCP RPI implementation (LAM 6).

were always generated, but the message sizes where the peaks and valleys were located differed in every run for the small message sizes.

### The sysv and usysv Modules

Figures 5.8 and 5.9 show comparisons of the two shared memory point-to-point transports. The sysv rpi module performs almost identically as the SYSV RPI implementation, and actually outperforms LAM 6 for large message sizes. The usysv rpi module performs slightly worse than the USYSV RPI implementation, but eventually outperforms LAM 6 for large message sizes. Note that the USYSV code uses busy-waiting on dedicated processors and is therefore extremely latency-sensitive. However, the bandwidth difference is actually quite small, and is overtaken for large message sizes. One identifiable cause for the difference is the cost of calling a function through a pointer versus direct addressing. An additional pointer dereference is required, leading to at least

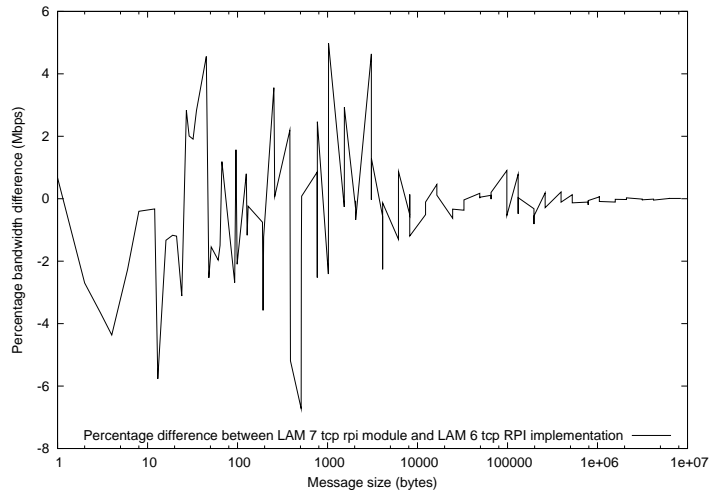


Figure 5.7. Ping-pong bandwidth measurements between two AVIDD-B nodes using gigabit Ethernet for communication. This graph shows the percentage bandwidth difference between the the tcp rpi module (LAM 7) and the TCP RPI implementation (LAM 6).

some of the overhead shown in Figure 5.9.

### The gm Module

The performance of the new gm rpi module cannot be compared to LAM 6 since there was no prior GM RPI implementation. As such, its performance can only be compared to native GM message passing. Figure 5.10 shows that the gm module’s performance is only slightly lower than native GM message passing.

Figure 5.11 shows percentage difference in bandwidth between GM and the first version of the gm rpi module. Small messages show about a 10-12% overhead compared to native GM. This is mainly due to the 36 byte envelope that accompanies all MPI messages. Mid-sized messages jump in overhead, ranging from 15 to 33% as compared to native GM. Further study optimization is clearly required for mid-level messages. Large message sizes perform well, leveling off around 3-4% overhead as compared to native

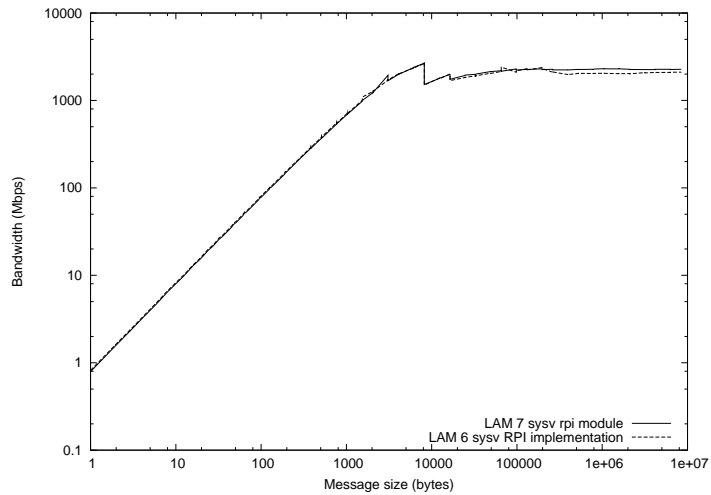


Figure 5.8. Ping-pong bandwidth measurements on one AVIDD-B node using shared memory for communication. This graph compares the performance of the sysv rpi module (LAM 7) to the SYSV RPI implementation (LAM 6).

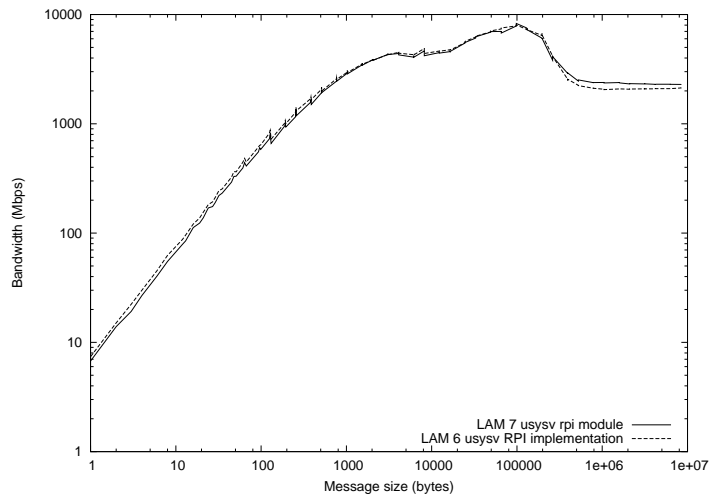


Figure 5.9. Ping-pong bandwidth measurements on one AVIDD-B node using shared memory for communication. This graph compares the performance of the usysv rpi module (LAM 7) to the USYSV RPI implementation (LAM 6).

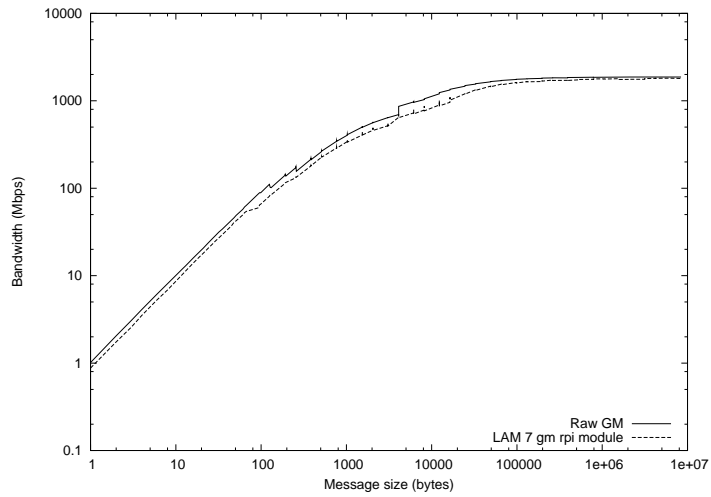


Figure 5.10. Ping-pong bandwidth measurements between two AVIDD-B nodes using Myrinet for communication. This graph compares the performance of raw GM and the gm rpi module (LAM 7) – there was no GM RPI implementation in LAM 6.

GM.

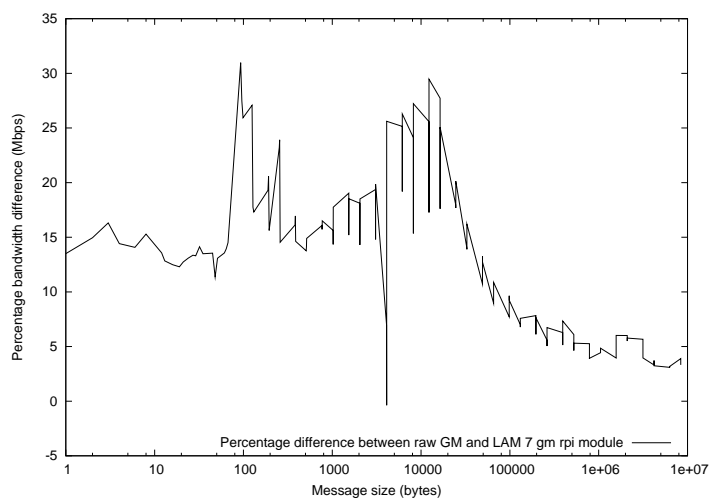


Figure 5.11. Ping-pong bandwidth measurements between two AVIDD-B nodes using Myrinet for communication. This graph compares the percentage of bandwidth difference between raw GM and the gm rpi module (LAM 7).

## CHAPTER 6

### MPI COLLECTIVE ALGORITHMS

Although the performance of the MPI collective operations [48, 52, 57, 58, 91, 120] can be a large factor in the overall run-time of a parallel application, their optimization has not necessarily been a focus in some MPI implementations until fairly recently [137]. Such implementations typically only included linear and/or logarithmic collective algorithms that provided correct answers but not necessarily in an optimized manner. Other MPI implementations (such as vendor-provided MPI implementations) provided at least some algorithms tuned for specific software/hardware environments.

MPI collectives are only a small portion of a compliant, production-quality implementation of MPI; implementors tend to give a higher priority to reliable basic functionality of all parts of MPI before spending time tuning and optimizing the performance of smaller sub-systems. As long as the MPI implementation's collectives returned correct answers, MPI implementors could overlook their performance deficiencies and instead focus on other issues (such as MPI-2 features such as dynamic processing and one-sided communication).

As a direct result, the MPI community has undertaken active research and development of optimized collective algorithms [21, 63, 73, 136, 145]. Although design and theoretical verification is the fundamental basis of a new collective algorithm, it must also be implemented and used in both benchmark and real-world applications (potentially in a variety of different run-time / networking environments) before its performance

can be fully understood. The full cycle of design, development, and experimental testing allows the refinement of algorithms that is not possible when any of the individual steps are skipped.

Much research has been conducted in the area of optimized collective operations resulting in a wide variety of different algorithms and technologies. The solution space is vast; determining which collective algorithms to use in a given application may depend on multiple factors, including the communication patterns of the application, the underlying network topology, and the amount of data being transferred. Hence, one set of collective algorithms is typically not sufficient for all possible application / run-time environment combinations. This is evident in the range of literature available on different algorithms for implementing the MPI collective function semantics.

There are significant barriers to entry for third party researchers when implementing new collective algorithms. Complex practical issues arise when testing new algorithms with a wide variety of MPI applications in a large number of run-time environments. To both ease testing efforts and to make the testing environment as uniform as possible, it is best if MPI test applications can utilize the new algorithms with no source code changes. This will even allow real world MPI applications to be used to for testing purposes; the output and performance from previous runs (using known correct collective algorithms) can be compared against the output when using the collective algorithms under test.

Common approaches for third parties have included the following:

- Use the MPI profiling layer. This is exactly what the MPI profiling layer is for: intercepting MPI functions (and potentially replacing them with something else entirely). Any MPI implementation with a profiling layer and any legacy MPI applications can therefore be used with no changes. But the MPI profiling system only allows one replacement layer at a time; replacing collective functions with alternates would disallow other MPI performance tools that use the profiling layer, for example.
- Edit an existing MPI implementation. To date, this is probably the most common approach; download an open source MPI implementation and edit the back-end implementation. This also allows unmodified MPI applications to use new algorithms. The obvious drawback is that an MPI implementation is a large software

project; finding right data structures and functions to re-implement the collective algorithms, as well as observing any documented (or undocumented) restrictions in the collectives may be a challenge. Some MPI implementations make this easier than others (e.g., replacing sets of function pointers on communicators), but it still requires either modification of the MPI implementation or breaking abstraction barriers to modify opaque MPI data structures. Hence, the learning curve to add or replace functionality in the MPI implementation may be quite large. Additionally, editing an MPI implementation effectively “forks” the source code – all changes will need to be applied to future versions, likely causing recurring maintenance issues.

- Create a new MPI implementation. Entirely new MPI implementations have been created simply to design, test, and implement new MPI collective algorithms [74, 21]. Although all aspects of the MPI collective algorithms can therefore be controlled, and unmodified MPI applications can use the new algorithms, the overhead to produce a working MPI implementation is enormous. The time necessary to create a new MPI implementation is likely to be prohibitive.
- Use alternate function names. This is perhaps the easiest of approaches; implement a new `MPI_BARRIER` algorithm in a C function named `My_MPI_Barrier()`. This method has fewer implementation restrictions than the previous approaches, but has the significant drawback that legacy MPI applications will not use the new algorithms without either a source code change or a re-compilation with header files that remap the MPI collectives to the new functions. While this approach is a fine model for development, it may make testing and deployment difficult.

This chapter describes the `coll` component type in LAM/MPI, and how it avoids the problems of the approaches listed above and provides a per-communicator algorithm selection mechanism from the set of available modules. Appendix D provides a detailed technical reference of the `coll` component framework.

**Acknowledgements:** The `lam_basic` module was strongly influenced by the collective algorithms in LAM/MPI v6.5. The design of the interface and component architecture as well as the implementation of `smp` module were both new work and performed by me.

## 6.1 Design

The main design goal of the `coll` component framework is to allow easy implementation of new MPI collective algorithms. To that end, since every algorithm is different, the simplest interface abstraction is to have a different function for every top-level MPI col-

lective function. This allows `coll` modules to focus on high-performance implementations of algorithms rather than an arbitrarily complicated module interface.

Top-level MPI collective functions have been reduced to thin wrappers that perform error checking before invoking back-end `coll` module interface functions. The scope of a `coll` module is a communicator. Specifically, one `coll` module is assigned to each communicator; this module is used to implement all MPI collectives that are invoked on that communicator. For example, `MPI_BCAST` simply checks the passed parameters for errors and then invokes the back-end broadcast function on the `coll` module assigned to the communicator.

The framework actually specifies two function pointers for each top-level MPI collective; one for intracommunicators and one for intercommunicators. Since the algorithms for each, by definition, will be different, it makes sense to split them into two different functions. The top-level MPI function will invoke the correct function depending on the type of the communicator.

Modules are free to implement the standardized MPI semantics in any way that they choose. Most, however, use one or more of the following models: layered over point-to-point, alternate communication channels, or layered over another `coll` module.

#### 6.1.1 Layered over Point-to-Point

A simple implementation model is to utilize MPI point-to-point functions to send data between processes. For example, using `MPI_SEND` and `MPI_RECV` to exchange data is both natural and easy to understand, freeing the `coll` module author to concentrate on the module's algorithms and remain independent of how the underlying communication occurs. This model has been used extensively by MPI implementations [19, 51] and third party collective algorithm researchers [79].

### 6.1.2 Dedicated Communication Channels

Recently, researchers have been exploring the possibility of avoiding MPI point-to-point functionality, and instead using alternate communication channels for collective communications. Some network interfaces contain native primitives for collective operations and/or streamlined one-sided operations which can lead to significant performance gains as compared to using traditional point-to-point methods. Examples of alternate communication channels that at least partially support collective operations include (but are not limited to): shared memory [90], UDP multicast [74], Myrinet [150], and Infini-band [80].

MPI collectives are synchronous in nature, allowing a considerably simpler implementation model compared to those required to support MPI's point-to-point functionality. Complications arise, however, when supporting multi-threaded MPI applications that allow independent progress in each thread. For example, multiple threads in a process may simultaneously invoke the same `coll` module on different communicators. LAM/MPI does not yet support `MPI_THREAD_MULTIPLE`, so this is not [yet] much of a factor. Hence, `coll` modules utilizing their own communication channels can utilize simplistic models.

### 6.1.3 Hierarchical `coll` Modules

The `coll` framework was carefully designed such that `coll` modules can be re-used at run-time in two ways.

First, the `coll` module “`lam_basic`,” as its name implies, is a basic implementation of all of the MPI collectives. It can be used with any communicator and topology, although typically with less-than-optimal performance. The purpose of this module is to provide a baseline implementation of as many MPI algorithms as possible, allowing other modules to use its routines as necessary. For example, a module that only provides an optimized

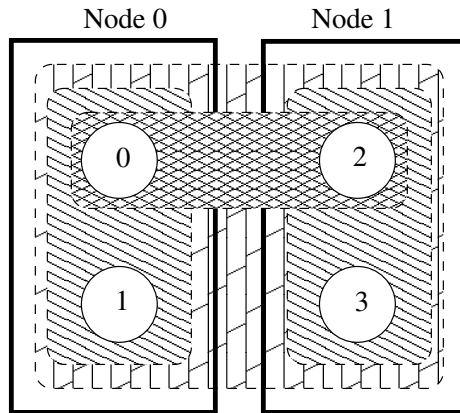


Figure 6.1. Four processes are distributed across two nodes. `MPI_COMM_WORLD` contains all four processes. Two sub-communicators (shown vertically) each contain the two processes local to their respective nodes. One “bridge” communicator (shown horizontally) contains a representative process from each node.

scatter algorithm implementation will automatically use the methods from the `lam_basic` module for all other collective routines. This allows the optimized scatter module to be used in any MPI program even though it only implements a small number of new/optimized routines.

A second, more complex, model involves using a hierarchy of `coll` modules to implement a single, top-level MPI collective. This is useful when a collective is invoked on a communicator that spans multiple kinds of networks. For example, Figure 6.1 shows two SMPs, each running two MPI processes. A single, top-level MPI communicator contains all four processes. The `coll` module used by the top-level communicator creates three sub-communicators: one for each SMP (containing the two processes on each node), and a third “bridge” communicator connecting one representative process from each node.

Note that each sub-communicator will have its own `coll` module. This hierarchical arrangement of communicators effectively allows each network to utilize its own optimized `coll` module, resulting in an efficient movement of data across each medium.

This model will be explained in more detail in Section 6.2.2, where the `smp coll`

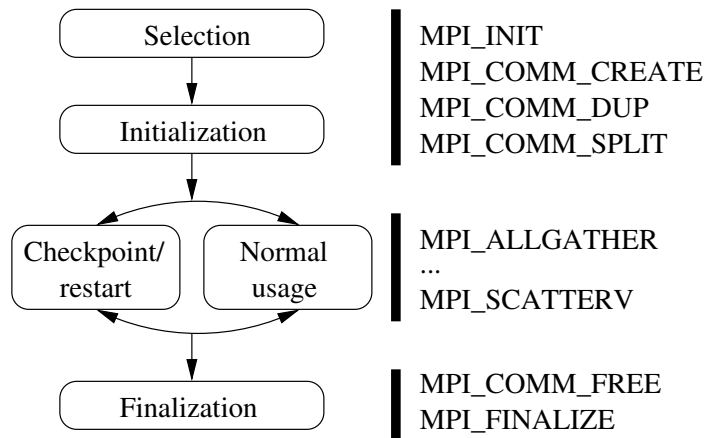


Figure 6.2. Phases in the life of a coll module. When a module is created, the selection process determines which coll module will be used. The selected module is then initialized and is ready for normal use (i.e., invoking collectives) and checkpoint/restart services. When the communicator is destroyed, the coll module finalizes itself for that scope.

module will be discussed as an example implementation.

#### 6.1.4 Module Lifecycle

Since there are potentially many coll scopes per process, modules have a distinct lifecycle in each scope. There are five phases in a coll module’s lifecycle: selection, initialization, checkpoint / restart, normal operation, and finalization. Figure 6.2 shows these phases and the corresponding MPI functions that trigger them. Note that a module may be involved in multiple lifecycles simultaneously; coll modules have a one-to-many relationship with communicators.

##### Selection

As each communicator is created (including MPI\_COMM\_WORLD and MPI\_COMM\_SELF), a coll module is selected from all available modules. The algorithm used is essentially the same as what is described in Section 3.2.7; all available modules are passed

the newly-created communicator and asked if they want to run. If the module wants to run, it provides a priority. The priorities of all modules who want to run are ordered and the module with the highest priority is selected.

Note that no modules are closed as a result of not being selected. Since selection is a per-communicator basis, available modules are not closed until `MPI_FINALIZE`.

### Initialization

Once a `coll` module is selected for a given communicator, it is initialized. Specifically, the module's initialization function is invoked, passing the target communicator as an argument. The initialization function performs any one-time setup required by the module. By definition, a communicator's member processes and ordering are static, allowing a module's initialization routine to pre-compute any data structures that will later be used during collective routines. This design emphasizes the potential run-time optimizations that can be obtained by shifting as much overhead calculation and coordination to the one-time initialization function as possible. This can reduce the amount of computational overhead in the run-time of collective routines. The `coll` framework provides a hook on the communicator for the module to cache its pre-computation results. All subsequent phases in the module's lifecycle are invoked relative to a communicator for which it was selected; the communicator is passed as an argument to all invocation functions. This allows the module to retrieve its communicator-specific pre-computation data when a collective function is invoked.

Note that the pre-computing efforts may include coordination with other MPI processes in the communicator (e.g., via MPI point-to-point functions). Constructing efficient, deadlock-free coordination algorithms is made easier by the fact that all processes in the communicator will be initializing their `coll` module at the same time (since communicator construction is synchronous).

The MPI standard states that only one collective may be invoked on a communicator at a time (including communicator constructors and destructors). Hence, even though the selection or initialization functions of a given module may be executed in multiple threads simultaneously, they will be operating on different communicators. This design encourages modules to only use private data that is cached on the target communicator, and is therefore not likely to cause thread synchronization issues or race conditions.

Once a module has been initialized, it returns a list of function pointers for its algorithm routines to the coll framework which are then assigned to the communicator. These functions are later invoked by the coll framework during the “normal usage” phase in the module’s lifecycle whenever a top-level MPI collective function is invoked. The module is then ready to be checkpointed or used for collective operations.

#### Checkpoint / Restart

coll modules can insert arbitrary functionality during relevant phases of LAM’s checkpoint/restart procedures. Details of how the coll component framework is used in checkpointing are described in Chapter 7.

#### Normal Usage

After a coll module has been initialized with a communicator, that module’s collective routines will be invoked whenever an MPI collective function is invoked on that communicator. Each top-level MPI function is essentially a thin wrapper function that mainly performs error checking on the passed parameters before invoking the corresponding coll module’s collective routine. Communicator-specific pre-computed data (previously cached on the communicator during initialization) may be retrieved and used to optimize the performance of the module’s collective routine.

As mentioned in the Initialization section above, the selected module returns a list of function pointers that are assigned to the communicator. For each top-level MPI collec-

tive function, two module function pointers are provided: one for when the collective is invoked on an intracommunicator, and a second for when the collective is invoked on an intercommunicator.

For example, when the `MPI_Bcast()` function is invoked on `MPI_COMM_WORLD`, it checks all of the parameters that are passed into it. It then invokes the the module's broadcast function pointer (the decision whether to use the intercommunicator broadcast or intracommunicator broadcast was made when the communicator was created) on the `coll` module that was selected for that communicator when it was created.

## Finalization

The final phase in a `coll` module's lifecycle on a communicator occurs when the communicator is destroyed. The module's finalization method is responsible for cleaning up all resources associated with the communicator that is being destroyed. This typically entails freeing any pre-computation data from the communicator's cache.

## 6.2 Implemented Modules

Two `coll` modules are included in LAM/MPI version 7: `lam_basic` and `smf`. These modules serve both as reference algorithms as well as examples of two different implementation models.

### 6.2.1 The `lam_basic` Module

The `lam_basic` module contains a full set of intracommunicator collectives; intercommunicator algorithms have not yet been implemented. Although relatively naive, the `lam_basic` routines can be used on any communicator (regardless of underlying topology), switching between  $O(n)$  and  $O(\log(n))$  algorithms depending on the number of processes in the communicator. The core algorithms used are reliable and mature; they have existed in LAM/MPI production code for several years.

As a mature implementation of collectives in LAM/MPI, its behavioral characteristics were well understood. This made it a natural choice for both influencing the design of the `coll` component interface, but also as a first `coll` module implementation. The successful port of the legacy LAM/MPI collective algorithms to the new framework served as a validation of the overall design.

All of the `lam_basic` algorithms use MPI point-to-point functions for moving data between MPI processes. For example, `MPI_BCAST` has both a linear and a logarithmic implementation. In the linear implementation, the root process loops over `MPI_SEND` while all other processes block on an `MPI_RECV`. The logarithmic implementation uses a traditional binomial tree; parent processes send data with `MPI_SEND` while child processes block in `MPI_RECV`.

The `lam_basic` module is used as a fallback for modules that do not implement a full set of collective algorithms. For example, if a module does not implement its own `MPI_SCATTER`, the `coll` framework will automatically invoke the `lam_basic` `MPI_SCATTER` instead.

### 6.2.2 The `smp` Module

The `smp` module was also instrumental in shaping the design of the `coll` framework. Its algorithms are based on the research from the MagPIe project [77, 78, 79], MagPIe focused on uniprocessors communicating across a WAN; the `smp` module is oriented to SMPs communicating on a LAN. The end effect is the same: two levels of network latency that can be exploited at run-time. Segmenting the communicator into groups of local process peers and electing representatives from each group to communicate with other groups provides a natural segregation of local and global communications.

Similar to the `lam_basic` module, the MagPIe reference implementation software layers its communication over MPI point-to-point functions. However, the MagPIe software

implements much of its own infrastructure: calculating and maintaining groups, using algebraic group operations for membership and maintenance, and determining process rank IDs relative to different groups.

This functionality is also available in the MPI through group and communicator management API functions. Using the standard MPI functions to create local process groups (sub-communicators) and translate rank identifications between groups dramatically simplified the implementation of the MagPIe algorithms in the `smp` module. Using native MPI functionality for the management of MagPIe pre-computation data evolved into the idea of having one `coll` module be able to create sub-communicators – each which have their own `coll` module – to create hierarchically-layered collective modules.

A direct implication of this model is that the `coll` framework must be able to handle recursive communicator creation. During the construction of a communicator, the initialization of a `coll` module may cause the creation of another communicator. This may, in turn, trigger the creation of yet another communicator (and so on). This also applies to the finalization phase: when the top-level communicator is finalized, it will likely call `MPI_COMM_FREE` on any sub-communicators created during initialization.

For example, in the MagPIe broadcast algorithm (adapted for a SMPs-on-a-LAN environment) the root broadcasts the data to the set of representatives from the other process groups. Each representative (including the root) then broadcasts to the members of its local group. This is shown graphically in Figure 6.3.

During the initialization phase of the `smp` module, the three sub-communicators shown in Figure 6.3 are created: two containing local-only processes, and one “bridge” communicator between processes 0 and 3. This allows the implementation of the MagPIe broadcast algorithm to be reduced to the pseudocode shown in Figure 6.4.

Note that there are two calls to `MPI_BCAST`: these broadcasts are effected using whichever module was selected when the sub-communicators were created. Depending

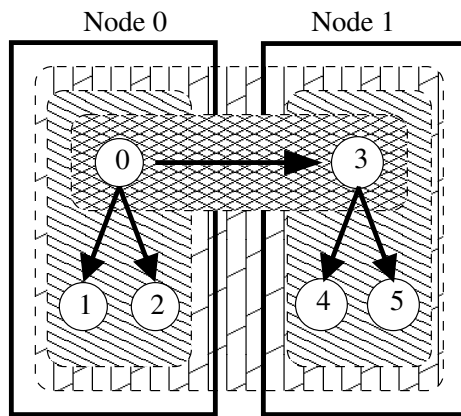


Figure 6.3. MagPie algorithm for broadcast from process 0. Process 0 sends to its peer on the remote node (process 3). Each then do a local broadcast to the remaining processes on their nodes (processes 1 and 2, and processes 4 and 5, respectively).

```

if (i_am_a_representative) {
    MPI_Bcast(buffer, ..., bridge_root, bridge_comm);
}
MPI_Bcast(buffer, ..., local_root, local_comm);

```

Figure 6.4. Pseudocode showing the MPI\_BCAST implementation using a hierarchical implementation approach. `bridge_root`, `bridge_comm`, `local_root`, and `local_comm` are all calculated and initialized during the per-communicator initialization and are cached on the communicator.

on the number of processes and topology involved, the broadcasts may be optimized according to however the selected `coll` module is implemented. For example, if `lam_-basic` is selected, a logarithmic algorithm will be used if the number of processes is large enough.

### 6.3 Results

The `coll` framework is intended to provide an easy to use, high-performance platform for third parties to implement and experiment with collective algorithms in LAM/MPI. As such, it is critical that the framework itself not contribute additional overhead, potentially negating performance gains from optimized collective algorithms.

Two main types of tests were run on the implemented `coll` modules – correctness and performance.

#### 6.3.1 Correctness

Each `coll` module was checked for functional correctness. Correctly functioning `coll` modules perform MPI collective operations in accordance with their specifications in the MPI standard. With the exception of `MPI_BARRIER`, all collectives entails movement and possible combination of user data between MPI processes. `MPI_BARRIER` only performs a synchronization – no user data is required to be transmitted.

Each collective operation was subject to a battery of tests designed to check its basic functionality as well as uncommon cases with unusual input parameters. In all data-movement collectives, the output buffer is used as the basis for determining whether the test case passed or failed. Specific operations are invoked on input data that are expected to generate specific output. If the output does not match the expected result, the test is ruled to have failed. The tests used were derived from the IBM MPI test suite. Over time, tests have been modified and new tests have been added. All tests have been run in

LAM/MPI as well as other MPI implementations to verify that they are correct.

Each test was run in a variety of conditions on both modules in order to trigger as many code paths within the module as possible. The conditions are generated from mixing elements of three domains: *number of processes*, *schedule of processes per node*, and *number of nodes*. The *number of processes* domain is comprised of the following cases:

- **One:** Only one process is run. This may trigger a special case in a collective routine where no loops are performed, and a simply memory copy is used. Or it may be a no-op (e.g., `MPI_BARRIER`).
- **Power of 2:**  $2^N$  processes are run, for some value  $N > 0$ . Several algorithms can be easily optimized for powers of two; running with  $2^N$  processes checks to see if special case algorithms are functioning properly.
- **Non power of 2:**  $(2^N + X)$  processes are run, for some values of  $N$  and  $X$  where the result is not a power of two. While it is frequently easy to optimize for  $2^N$  processes, it can be harder to ensure that algorithms work for the general case where the number of processes is not a power of two. This test ensures that special cases for handling “straggler” processes are handled properly.

The *schedule of processes per node* domain is comprised of the following cases:

- **One process per node:** Exactly one processes per node is launched. This will trigger the selection of the `lam_basic` coll module.
- **$P$  processes per node:** Exactly  $P$  processes are launched on each node, where  $P > 1$ . This will trigger the selection of the `sm` coll module.
- **$P_i$  processes per node:** Each node receives a potentially different number of processes, where  $P_i \neq P_j$  for some values of  $i$  and  $j$ , and  $P_k \neq 1$  for some value of  $k$ . This should also force the selection of the `sm` coll module, and potentially trigger special case code if  $P_n = 1$ .

The *number of nodes* domain is comprised of the following cases:

- **1 node:** All test processes are launched on a single node. This tests single-node message passing, particularly relevant if shared memory is used for optimized message passing.
- **$2^M$  nodes:** The test processes are spread across  $2^M$  nodes, for some  $M > 0$ . Some algorithms use special optimizations for  $2^M$  nodes or processes; this domain case is intended to trigger those code paths.
- **$(2^M + X)$  nodes:** The test processes are spread across  $(2^M + X)$  nodes, for some values of  $M$  and  $X$  where the result is not a power of 2. Some algorithms use special optimizations for  $2^M$  nodes or processes; this domain case is intended to trigger alternate code paths and ensure that the corner cases are handled properly.

TABLE 6.1

Summary of MPI Collective results on 1 node

Schedule	Number of processes		
	1	$2^N$	$2^N + X$
1	✓	NA	NA
$P$	NA	✓	✓
$P_i$	NA	NA	NA

TABLE 6.2

Summary of MPI Collective results for  $2^M$  nodes

Schedule	Number of processes		
	1	$2^N$	$2^N + X$
1	NA	✓	✓
$P$	NA	✓	✓
$P_i$	NA	✓	✓

The three dimensions taken together form a  $3 \times 3 \times 3$  testing cube; each cell in the cube represents a unique combination of number of processes, schedule of processes, and number of nodes. Tables 6.1, 6.2, and 6.3 each show a cross-section of the cube – one for each “slice” of the third dimension (number of nodes). In each of the Tables, “✓” indicates a condition that was tested; “NA” indicates a condition that does not make sense and was not tested.

Each “✓” entry in the tables represents a condition that was tested in the `lam.basic` and `smp` modules, as appropriate (the `smp` module will only allow itself to be used if the

TABLE 6.3

Summary of results for  $(2^M + X)$  nodes

Schedule	Number of processes		
	1	$2^N$	$2^N + X$
1	NA	✓	✓
$P$	NA	✓	✓
$P_i$	NA	✓	✓

communicator spans more than one node and there are multiple processes on at least one node).

Coverage analysis was performed using the GNU Coverage tool (“`gcov`”) on Linux. The `gcov` tool was invoked for every test described above and recorded every line in the LAM source code that was accessed. Table 6.4 shows the percentages of code that were accessed. Just as with the `rpi` modules, the percentages are deceptively low. A post-mortem analysis of exactly which lines were accessed showed that the majority of lines not accessed were the result of error condition testing. For example, none of the tests pass bad input to MPI functions (e.g., incorrect datatypes, invalid buffers, etc.), and none of the Unix system and library calls failed unexpectedly during the tests. Hence, these code paths (which mainly consisted of passing an error up the call stack) were not exercised, and this accounted for the seemingly-low coverage percentages.

### 6.3.2 Performance

The `coll` performance comparisons described below were run with Pallas Benchmarks [100] on the AVIDD-B cluster described in Section 4.3.4. The entire cluster, including the networks used, was otherwise dormant while the experiments were con-

TABLE 6.4

Test code coverage of coll modules

Module	Coverage
lam_basic	75%
gm	87%

ducted. Data-moving collective operations were on 4 MB messages over gigabit Ethernet and shared memory. Since both implemented modules (`lam_basic` and `sm`) use point-to-point MPI functions for communication, results provided in this section are expressed in terms of the three LAM 7 rpi modules that have corresponding RPI implementations in LAM 6: `sysv`, `tcp`, and `usysv`.

#### The `lam_basic` Module

Recall that the `lam_basic` module is based on the same point-to-point algorithms as were in the previous, monolithic architecture of LAM/MPI in version 6. Therefore, measuring the performance of the same algorithms in two different architectures allows the comparison of overhead between the two infrastructures.

Results from a token set of collectives are presented in this section: `MPI_ALLREDUCE`, `MPI_ALLTOALL`, `MPI_BARRIER`. Since `lam_basic`'s implementation of `MPI_ALLREDUCE` is an `MPI_REDUCE` followed by an `MPI_BCAST`, results of these operations are effectively given as well.

Figures 6.5, 6.6, and 6.7 show the performance of `MPI_ALLREDUCE` with the `sysv`, `tcp`, and `usysv` rpi modules, respectively. The reduction performed was a summation across 4 MB of single-precision floating point values. Messages were transferred across all processes in a binomial tree pattern, but no effort was made to optimize the tree (e.g.,

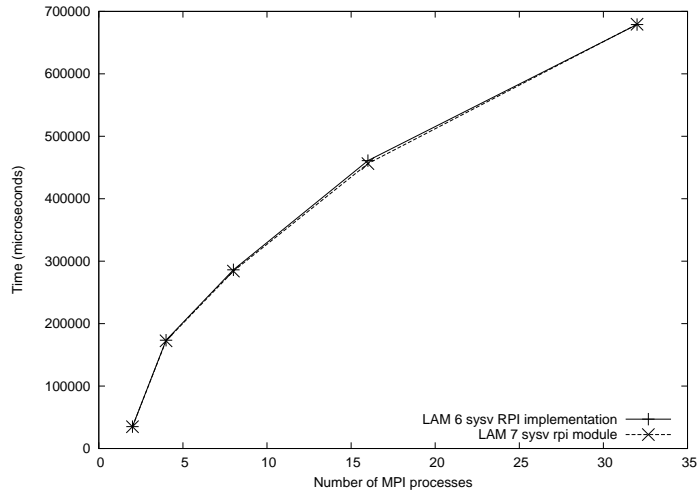


Figure 6.5. Wall-clock execution time for `MPI_ALLREDUCE` on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the `sysv rpi` module and the SYSV RPI implementation.

maximizing on-node communication and minimizing off-node communication). The Figures show identical performance between LAM 6 and LAM 7 – no additional overhead was added by the `coll` abstractions.

Figures 6.8, 6.9, and 6.10 show the performance of `MPI_ALLTOALL` under the same conditions. Each process initiates sends to and receives from all other processes; there is little chance to optimize the collective pattern. This algorithm stresses the point-to-point communications layer. Just as with `MPI_ALLREDUCE`, no additional overhead is evident with the LAM 7 component architecture.

Finally, Figures 6.11, 6.12, and 6.13 show the performance of `MPI_BARRIER`; a zero-byte message scatter and gather across a binomial tree. Similar to the other two tested functions, `MPI_BARRIER` shows no additional overhead in the component architecture implementation.

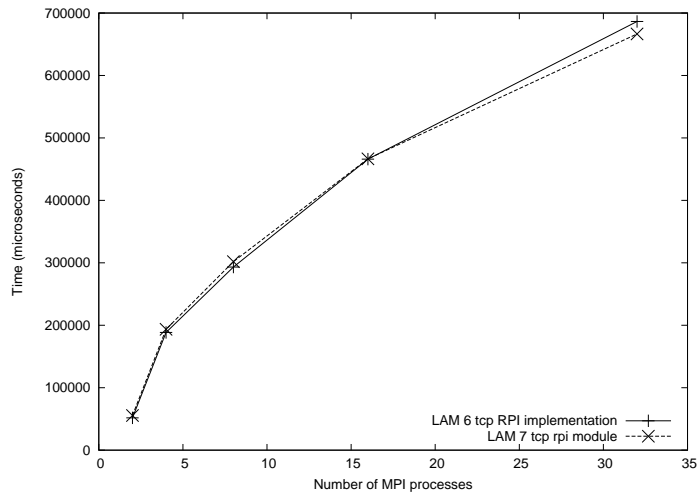


Figure 6.6. Wall-clock execution time for `MPI_ALLREDUCE` on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the `tcp rpi` module and the TCP RPI implementation.

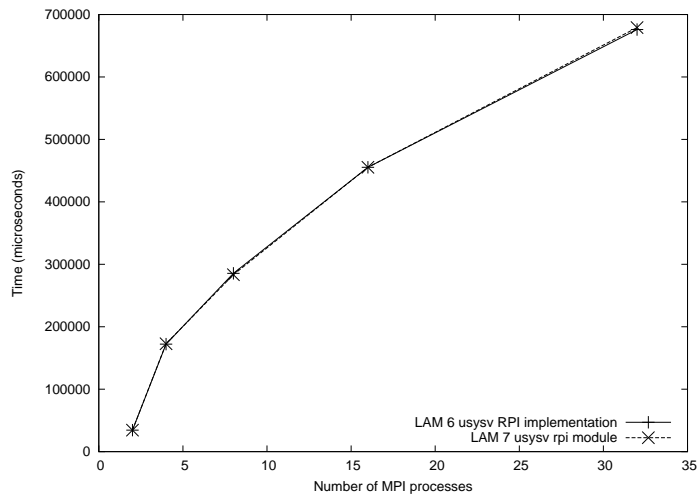


Figure 6.7. Wall-clock execution time for `MPI_ALLREDUCE` on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the `usysv rpi` module and the USYSV RPI implementation.

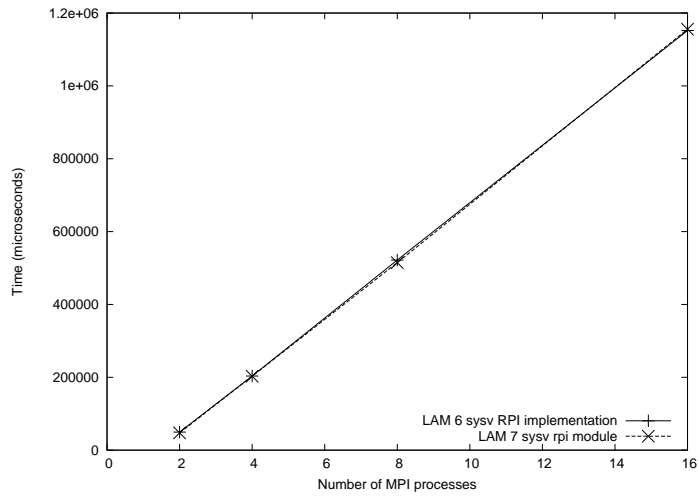


Figure 6.8. Wall-clock execution time for `MPI_ALLTOALL` on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the `sysv rpi` module and the SYSV RPI implementation.

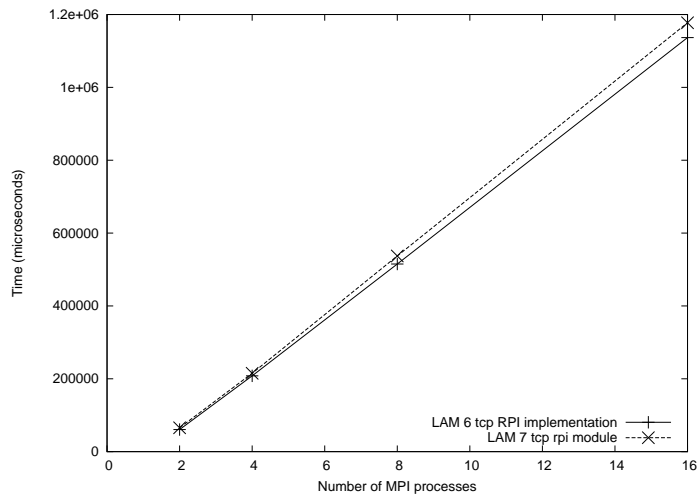


Figure 6.9. Wall-clock execution time for `MPI_ALLTOALL` on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the `tcp rpi` module and the TCP RPI implementation.

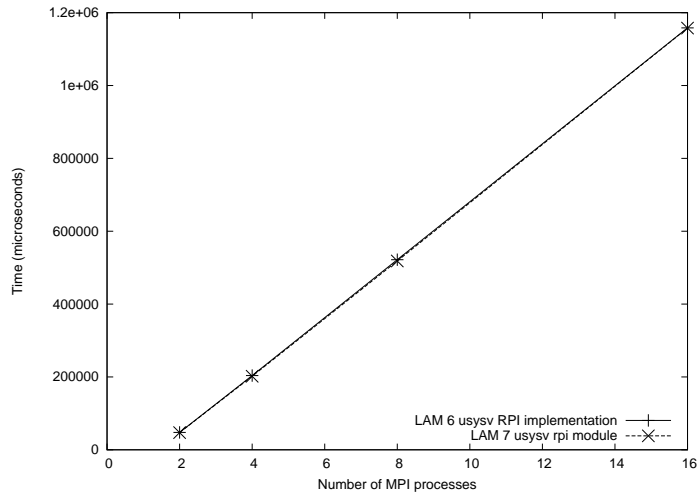


Figure 6.10. Wall-clock execution time for MPI\_ALLTOALL on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the usysv rpi module and the USYSV RPI implementation.

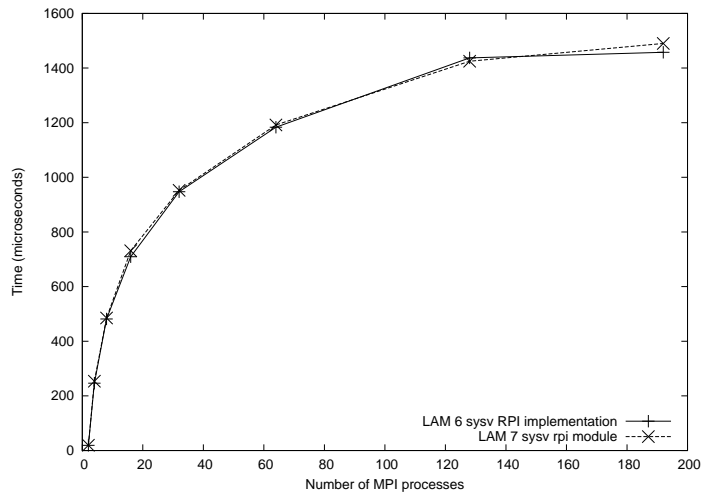


Figure 6.11. Wall-clock execution time for MPI\_BARRIER on the AVIDD-B cluster on varying numbers of processes using the sysv rpi module and the SYSV RPI implementation.

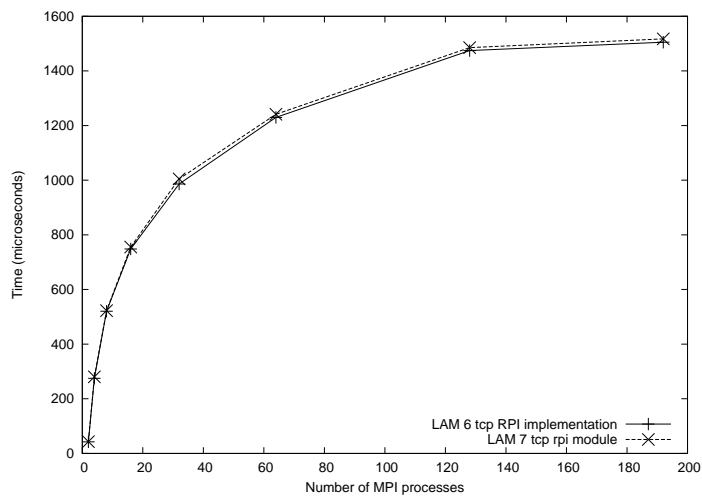


Figure 6.12. Wall-clock execution time for MPI\_BARRIER on the AVIDD-B cluster on varying numbers of processes using the tcp rpi module and the TCP RPI implementation.

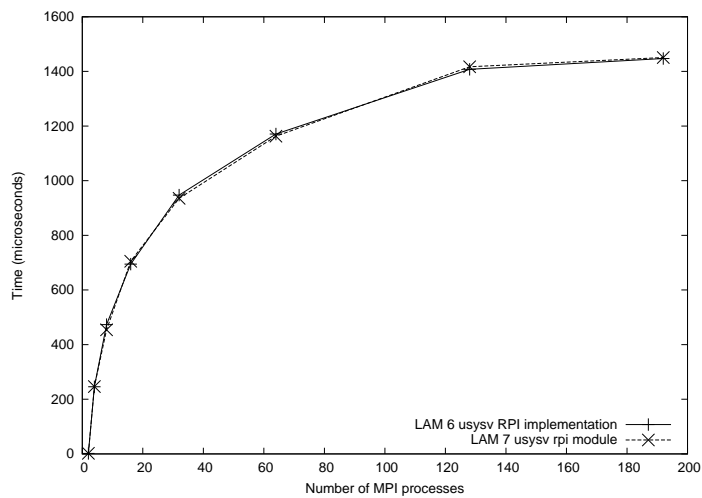


Figure 6.13. Wall-clock execution time for MPI\_BARRIER on the AVIDD-B cluster on varying numbers of processes using the usysv rpi module and the USYSV RPI implementation.

## The smp Module

LAM 6 had no equivalent to the Magpie-like algorithms in the `smp` module. Results presented in this section show comparisons of the LAM 7 `smp` module vs. the `lam_basic` module. Figures 6.14, 6.15, and 6.16 show an `MPI_ALLREDUCE` summation across 4 MB of single-precision floating point values using the `sysv`, `tcp`, and `usysv rpi` modules. Each graph compares the wall-clock execution time of the same operation using the `lam_basic` and `smp coll` modules. Recalling that the AVIDD-B nodes are 2-way SMPs, the results show that the Magpie-based algorithms are able to exploit this fact and achieve substantial performance improvements.

The `MPI_ALLREDUCE` results in Figures 6.14, 6.15, and 6.16 show good speedup at 192 processes: 27.4%, 21.7%, and 25.7%, respectively. As noted above, the `MPI_ALLREDUCE` function is implemented as an `MPI_REDUCE` followed by an `MPI_BCAST`. Both the reduction and broadcasting algorithms were optimized to reduce off-node communication. Note, however, that the MPI standard says that all built-in collective operations are commutative, but makes no guarantees about associativity. The only guidance that it provides is that an MPI implementation should strive to present the same reduction result across successive runs when used with the same input data and number of processes. Since optimizing the reduction algorithm exploits associativity, it is disabled by default (a run-time SSI parameter can be used to enable it). The graphs in Figures 6.14, 6.15, and 6.16 show the behavior with associativity enabled.

Similarly, Figures 6.17, 6.18, and 6.19, and Figures 6.20, 6.21, and 6.22 show the performance of `MPI_BARRIER` and `MPI_BCAST`, respectively, under the same conditions. In all cases, the `smp` module outperforms the `lam_basic` module.

The `MPI_BARRIER` Figures show modest speedups at 192 processes: 9.9%, 8.8%, and 6.5%, respectively. These speedups are smaller than the `MPI_ALLREDUCE` speedups because the total number of bytes sent in a barrier is relatively small, reducing the opti-

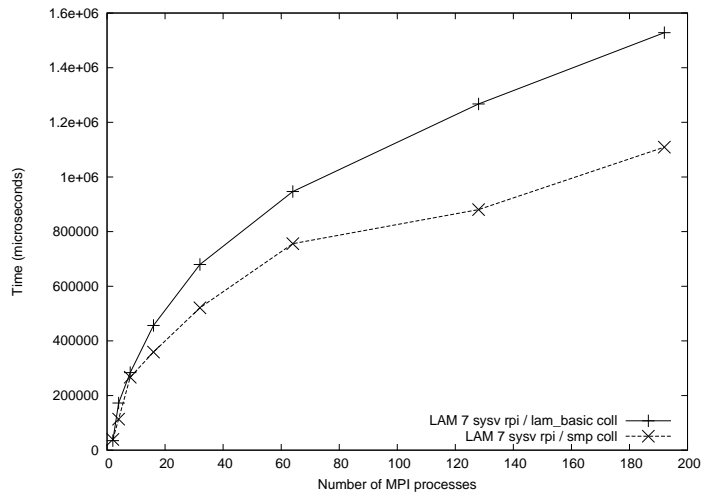


Figure 6.14. Wall-clock execution time for MPI\_ALLREDUCE on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the sysv rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

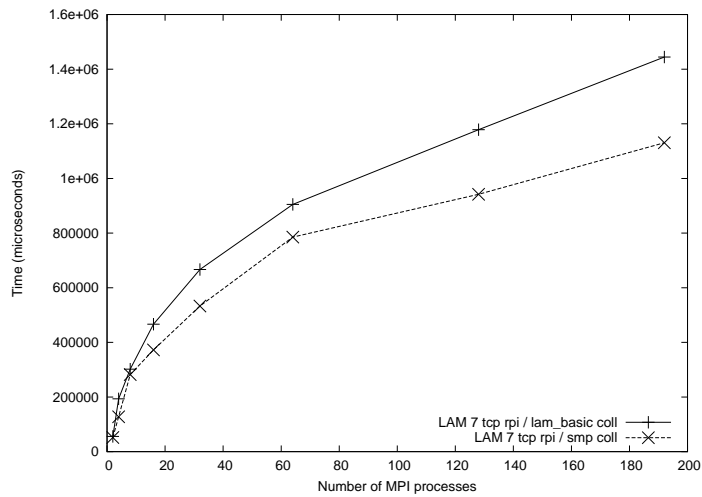


Figure 6.15. Wall-clock execution time for MPI\_ALLREDUCE on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the tcp rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

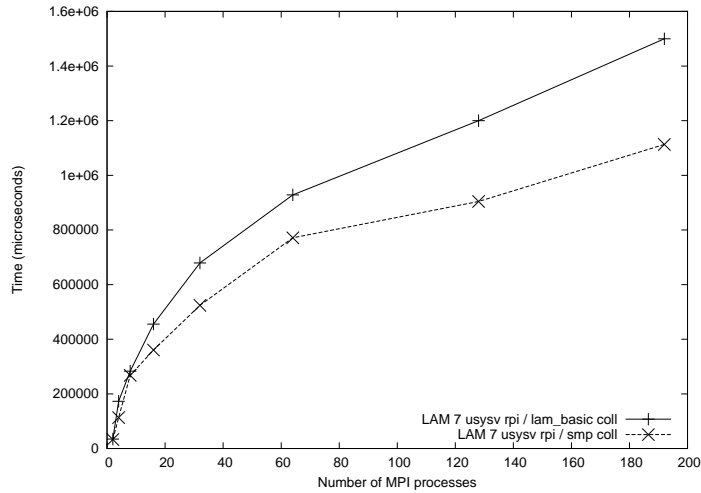


Figure 6.16. Wall-clock execution time for MPI\_ALLREDUCE on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the usysv rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

mization possibilities. The MPI\_BCAST Figures are show tremendous performance improvements at 128 processes – 45.8%, 43.5%, and 45.4% for the sysv, tcp, and usysv rpi modules, respectively. Since large amounts of data is involved in the broadcast, minimizing off-node communication has a large impact on the overall performance.

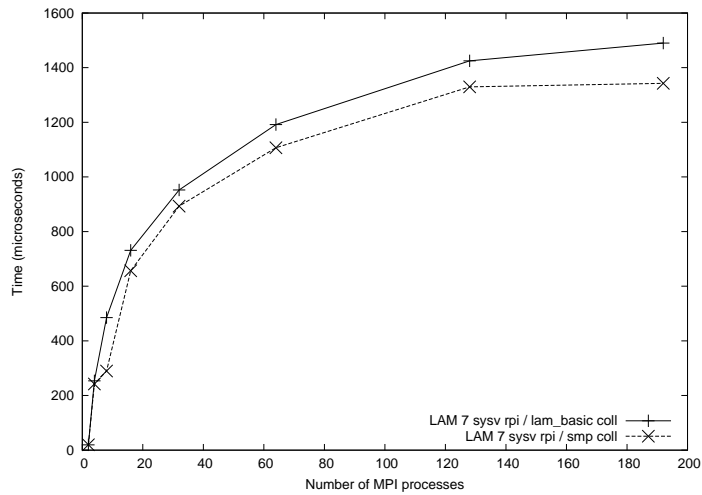


Figure 6.17. Wall-clock execution time for MPI\_BARRIER on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the sysv rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

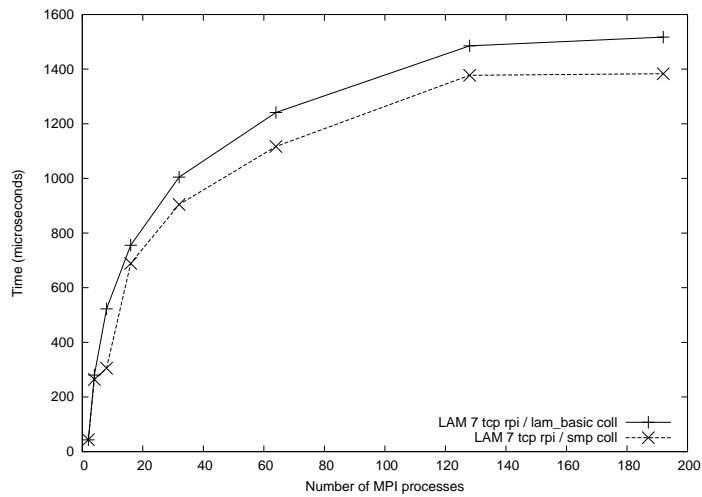


Figure 6.18. Wall-clock execution time for MPI\_BARRIER on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the tcp rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

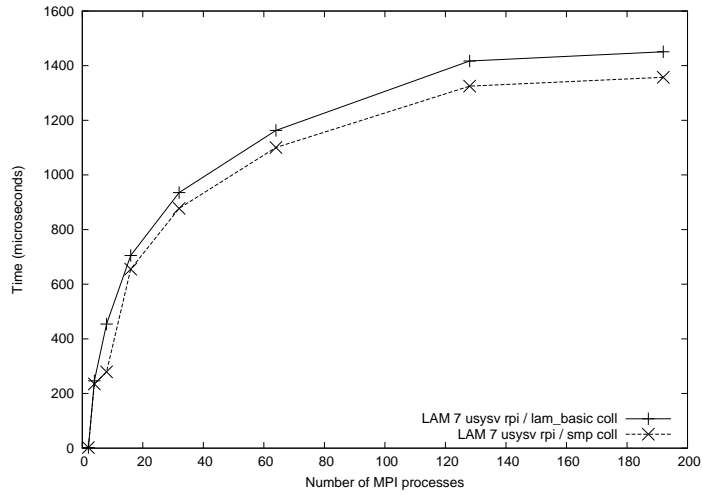


Figure 6.19. Wall-clock execution time for MPI\_BARRIER on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the usysv rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

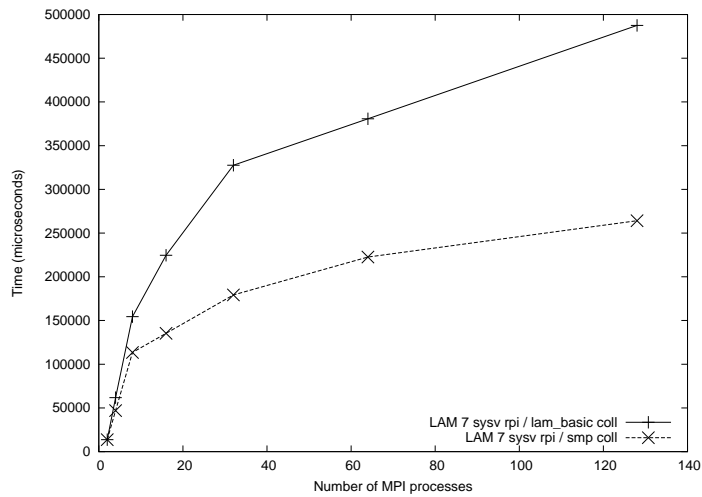


Figure 6.20. Wall-clock execution time for MPI\_BCAST on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the sysv rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

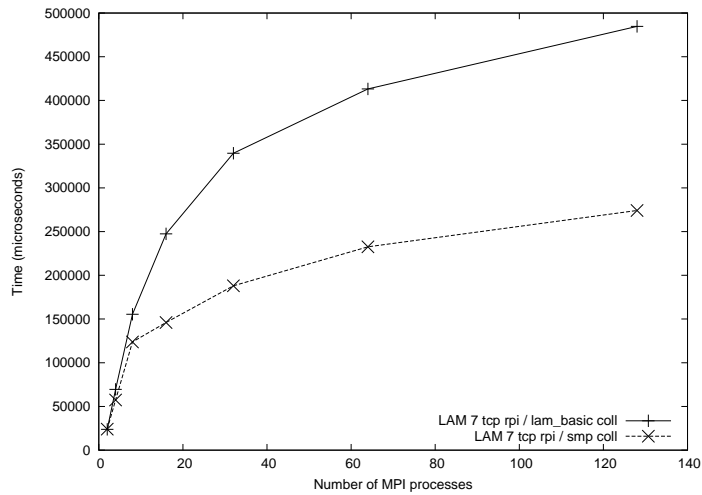


Figure 6.21. Wall-clock execution time for MPI\_BCAST on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the tcp rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

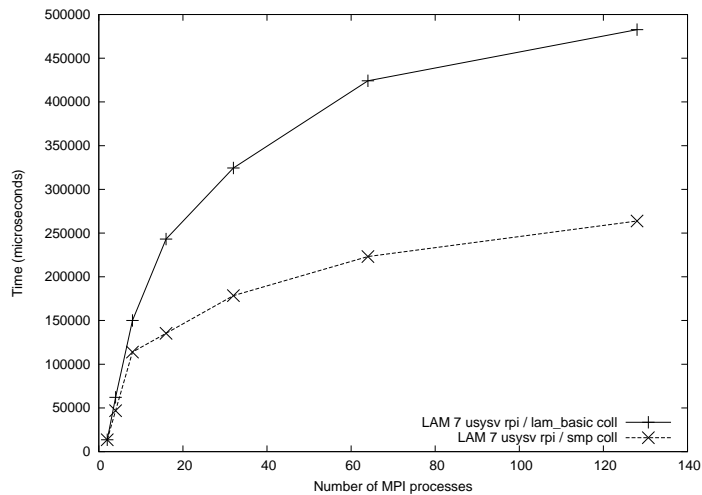


Figure 6.22. Wall-clock execution time for MPI\_BCAST on the AVIDD-B cluster with a 4 MB message on varying numbers of processes using the usysv rpi. This graph shows a comparison of the lam\_basic and smp coll modules.

## CHAPTER 7

### PARALLEL CHECKPOINT / RESTART

In recent years, the supercomputing community has seen a significant increase in the CPU count of large-scale computational resources. Seven of the top ten machines in the November 2002 Top 500 list [142] utilize at least 2000 processors. With machines such as ASCI White, Q, and Red Storm, the processor count for the largest systems is now on the order of 10,000 processors – and this increasing trend will only continue. While the growth in CPU count has provided great increases in computing power, it also presents significant reliability challenges to applications. In particular, since the individual nodes of these large-scale systems are comprised of commodity hardware, the reliability of the individual nodes is targeted for the commodity market. As the node count increases, the reliability of the parallel system decreases (roughly proportional to the node count). Indeed, anecdotal evidence suggests that failures in the computing environment are making it more difficult to complete long-running jobs and that reliability is becoming a limiting factor on scalability.

However, the MPI standard does not specify any particular kind of fault tolerant behavior. In addition, the most widely used MPI implementations have not been designed to be fault-tolerant. To address these issues, a framework has been designed to meet the following criteria:

- **Generality.** The framework must be able to support a wide variety of checkpoint/-restart mechanisms.

- **Transparency.** The framework implementation of coordinated checkpointing and rollback recovery must not require modifications to application source code.
- **Performance.** The framework must not introduce undue overhead to run-time performance.
- **Portability.** Although back-end checkpoint packages may be specific to particular run-time environments, the framework itself must be highly portable.

This chapter describes the `crlam` and `crmpi` component frameworks in LAM/MPI, and how they address the requirements described above [116, 147].

**Acknowledgements:** The `crlam` and `crmpi` component interfaces were designed under my direction by Sriram Sankaran, who also implemented (also under my direction) the first versions of the `bldr` modules. The `self` modules were designed by me and implemented under my direction by Prashanth Charapalli. The implementation of the component architecture and the follow-up `bldr` work after its initial implementation were both performed by me.

## 7.1 Checkpoint-Based Rollback Recovery

In the context of message-passing parallel applications, a *global state* is a collection of the individual states of all participating processes and of the states of the communication channels. A *consistent global state* is one that may occur during a failure-free, correct execution of a distributed computation. Within a consistent global state, if a given process has a local state that indicates a particular message has been received, then the state of the corresponding sender must indicate that the message has been sent [7, 23, 93]. Figure 7.1 shows two examples of global states, one of which is consistent, and the other of which is inconsistent. A *consistent global checkpoint* is a set of *local checkpoints*, one for each process, forming a consistent global state. Each local checkpoint is therefore the state necessary to restart a process and deliver any outstanding messages to it. Any consistent global checkpoint can be used to restart the parallel process upon failure.

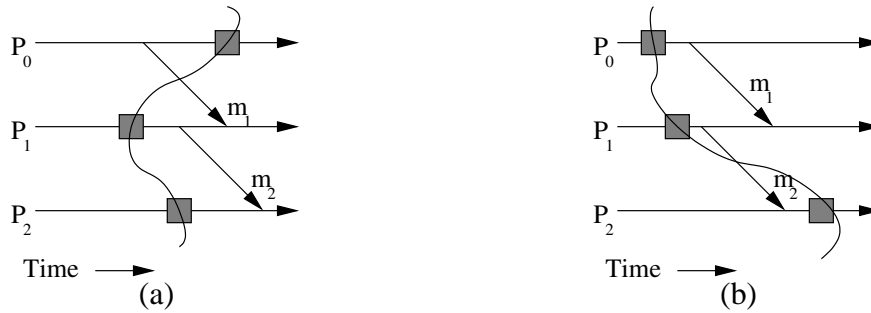


Figure 7.1. A message-passing system consisting of 3 processes. The blocks on each line represent when the checkpoint was taken; the line drawn between them establishes the global state. (a) shows an example of a consistent global state where message  $m_1$  is recorded as having been sent by process  $P_0$  but not yet received by process  $P_1$ , and (b) shows an example of an inconsistent global state in which message  $m_2$  is recorded as having been received by  $P_2$  but not yet sent by  $P_1$  [35].

Checkpoint/restart techniques for parallel jobs can be broadly classified into three categories: uncoordinated, coordinated, and communication-induced (these approaches are analyzed in detail in [35]).

### 7.1.1 Uncoordinated Checkpointing

In the uncoordinated approach, the processes determine their local checkpoints independently. During restart, the set of saved checkpoints is searched for a consistent state from which execution can resume. The main advantage of this autonomy is that each process can take a checkpoint when it is most convenient (without regard to its peers). For efficiency, a process may take checkpoints when the amount of state information to be saved is small [146].

However, this approach has notable disadvantages. First, there is the possibility of the *domino effect* [109] which causes the system to rollback to the beginning of computation, resulting in the loss of potentially large amounts of useful work. Second, a process may take checkpoints that will never be part of a global consistent state. Third, uncoordinated checkpointing requires each process to maintain multiple checkpoints, thereby incurring

a large storage overhead.

### 7.1.2 Coordinated Checkpointing

With the coordinated approach, the determination of local checkpoints by individual processes is orchestrated in such a way that the resulting global checkpoint is guaranteed to be consistent [23, 34, 81, 133, 141]. This typically involves flushing communication channels before taking the checkpoint, guaranteeing that outstanding messages have been received and inconsistent states (such as that shown in Figure 7.1 (b)) cannot happen.

Coordinated checkpointing simplifies recovery from failure and is not susceptible to the domino effect since every process always restarts from its most recent checkpoint. Also, coordinated checkpointing helps reduce storage overhead since only one permanent checkpoint needs to be maintained on stable storage (e.g., storage that is available even in the presence of failures). The main disadvantage of coordinated checkpointing, however, is the potentially large latency involved in saving the checkpoints, since a global checkpoint needs to be determined before the checkpoints can be written to stable storage.

### 7.1.3 Communication-Induced Checkpointing

The communication-induced checkpointing approach forces each process to take checkpoints based on protocol-related information piggybacked on the application messages that it receives from other processes [114]. Checkpoints are taken such that system-wide consistent state always exists on stable storage, thereby avoiding the domino effect [17]. Although processes are allowed to take some of their checkpoints independently, in order to determine a consistent global state, processes may be forced to take additional checkpoints. The checkpoints that a process takes independently are called *local checkpoints*, while those that a process is forced to take are called *forced checkpoints*. The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint. The forced checkpoint must be taken before the application

may process the contents of the message, possibly incurring artificially high latency and overhead. In contrast with coordinated checkpointing, no special coordination messages are exchanged in this approach.

#### 7.1.4 Other Uses of Checkpoint/Restart

The ability to checkpoint and restore applications has a number of uses in a parallel environment besides fault tolerance.

Gang scheduling – checkpointing and restarting all the processes that are part of a single parallel application – allows for more flexible scheduling. For example, jobs with large resource requirements can be intermittently scheduled at off-peak times using the checkpoint/restart capability. Without intermittent scheduling, such large jobs may use all available resources for long periods of time, effectively locking out other jobs for the duration. Hence, the ability to stop and resume large jobs allows scheduling of other available jobs in such a way that the overall system throughput is maximized. Higher total system utilization can then be realized by allowing the available processes to be scheduled in such a way that there is optimal use of computing nodes at all times.

Process migration is another feature that is made possible by the ability to save a process image. If a process needs to be moved from one node to another (perhaps because imminent failure of a node is predicted) it is possible to transfer the state of the processes running on that node to another node by writing the process image directly to a remote node. The process can then resume execution on this new node without having to kill the entire application and start it over again.

Process migration has also proved extremely valuable for systems whose network topology constrains the placement of processes in order to achieve optimal performance. The Cray T3E's interconnect, for example, uses a three-dimensional torus that requires processes that are part of the same parallel application to be placed in contiguous loca-

tions on the torus. This results in fragmentation as jobs of different sizes enter and exit the system. With process migration, jobs can be packed together to eliminate fragmentation, resulting in significantly higher utilization [149]. Networks with such constraining topologies have become less common recently, however IBM's Blue Gene/L project plans to constrain communication among processors [140], and more cluster projects may use them in the future.

On a cluster with a large number of computers, there is a high probability of failure of a long-running application due to the failure of one or more nodes. In the absence of a failure-recovery method, these jobs will need to be restarted from scratch, wasting hours of computation and also bringing down the total system utilization. However, periodically checkpointing the states of such jobs allows resumption of the execution of these jobs from their most recently saved states.

## 7.2 Design

This section presents an overview of the design of the checkpoint/restart system in LAM/MPI. This implementation does not alter the semantics of any of the MPI functions and fully supports all of MPI-1 (MPI-2, in particular, dynamic functions, are not yet supported). The checkpoint/restart system has been designed such that there is a clear separation between the checkpoint/restart functionality and MPI-specific functionality in LAM.

### 7.2.1 Overview

Checkpoint/restart services are implemented in two different component types: `crlam` and `crmpi`. As their names imply, each type is specific to a major architecture layer in LAM/MPI. Throughout this chapter, the name “cr” is used when referring to both types collectively; the individual names “`crlam`” and “`crmpi`” are used to discuss a specific

component type. In general, `crlam` functionality is used by `mpirun` and `crmpi` functionality is used by MPI processes.

The `cr` module types in LAM/MPI perform the following distinct functions:

- Receive the initial checkpoint request in `mpirun`.
- Propagate the checkpoint request out to each MPI process.
- Coordinate all SSI modules in each MPI process to be ready for a checkpoint. This typically entails draining and potentially closing MPI communication channels to create a consistent global state.
- Invoke the back-end checkpoint functionality in each MPI process.
- Coordinate all SSI modules in each MPI process to recover from a checkpoint. This typically entails re-opening MPI communication channels and resuming message passing activities.

Hence, similar to how LAM does not include native message passing on specific types of networks (it includes *interfaces* to companion libraries that perform the native message passing, such as TCP and Myrinet), `cr` modules, themselves, do not perform checkpoints. Instead, `cr` modules interface to back-end checkpointing systems that are capable of saving and restoring the relevant state from a serial process. `cr` modules, therefore, are specific to particular checkpointing systems.

For an MPI application to be checkpointable, it must meet two criteria: 1) a pair of `cr` modules must be selected and 2) all other selected MPI modules must support checkpoint/restart functionality. MPI applications not meeting these criteria are not erroneous; they are simply not checkpointable. The selected `cr` modules will coordinate all other selected modules to prepare them for checkpointing before invoking the back-end checkpointing system. Similarly, upon restart, the `cr` modules will again coordinate all other selected modules to allow them to recover from the restart.

The `cr` framework designs make two other important assumptions. First, although they play an important role during a checkpoint, the LAM daemons are not a logical part of an MPI application (nor do they contain any state specific to MPI-1 applications), and

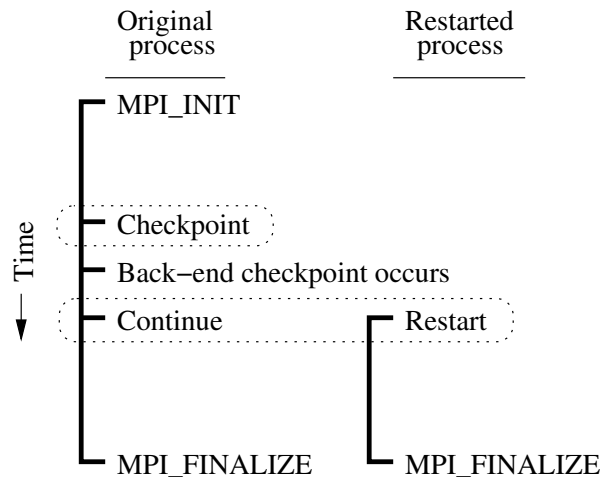


Figure 7.2. Three phases of checkpoint services in LAM/MPI parallel applications.

are themselves not checkpointed. Second, the `cr` design requires the use of a threads package on the target platform.

LAM/MPI uses a coordinated approach to checkpointing MPI jobs: since `mpirun` is the startup coordination point for MPI processes, it was the natural choice to serve as the entry point for a checkpoint request to be sent to a parallel job. The mechanism for initiating and delivering the checkpoint request is defined by each module (two such mechanisms are discussed in Section 7.3).

LAM uses a three-phased approach for checkpoint/restart services (also shown in Figure 7.2):

- **Prepare for checkpoint.** This phase extends from when a checkpoint request arrives in a process to when the back-end checkpoint system is invoked. It is usually described as the “checkpoint” phase.
- **Continue after checkpoint.** Usually referred to as “continue,” this phase starts when the back-end checkpoint system returns from a checkpoint (in the same process that was checkpointed) until the process has resumed normal execution.
- **Restart after checkpoint.** Similar to the “continue” phase, this phase starts when the back-end checkpoint system returns from a checkpoint. The difference is that the “restart” phase occurs in a newly-restarted process; it is not the same process that was initially checkpointed.

Time	<code>mpirun</code>	MPI processes
$t_i$	Receives checkpoint request	
$t_{i+1}$	Propagates checkpoint request to all MPI processes	
$t_{i+2}$	Back-end checkpoint system is invoked	Receives checkpoint request
$t_{i+3}$		Invoke “prepare to checkpoint” interface functions on all selected modules
$t_{i+4}$		Back-end checkpoint system is invoked

Figure 7.3. Sequence of events in the checkpoint phase.

### 7.2.2 Prepare For Checkpoint

Since `mpirun` is the startup coordination point for MPI processes, it was the natural choice to serve as the entry point for a checkpoint request to be sent to a LAM/MPI job. The sequence of events triggered in the checkpoint phase is listed in Figure 7.3.

The `cr` design assumes that a module-specific mechanism will be used to initiate a checkpoint request in `mpirun`. `mpirun` will then propagate the checkpoint request to all MPI processes that it started, also through a module-specific mechanism. One common method is to use the LAM services to invoke a process on every node that will initiate a checkpoint request for each MPI process in the parallel application.

Checkpoint requests arrive and must be handled asynchronously. This implies that the `crlam` propagation mechanism is running in its own thread inside `mpirun`; using a Unix signal alone to initiate a checkpoint request, for example, is not sufficient because propagating the request needs to perform actions that are not safe in a signal handler context. Similarly, a threaded approach is also required in `crmpi` modules. Checkpoint requests are required to be handled independent of the state of the application. Specifically, the

request must be able to proceed regardless of whether the application is in the MPI library (even if it is blocking) or not. As a direct consequence, checkpoint-capable MPI applications will automatically have their MPI thread levels raised to `MPI_THREAD_SERIALIZED`, regardless of whether the user application uses multiple threads or not. The use of `MPI_THREAD_SERIALIZED` forces LAM to employ a global mutex allowing only one thread in the MPI library at a time; the mutex is locked when a thread enters any MPI function and is unlocked when the thread leaves.

This lock also prevents the `crmpi` module from entering the MPI library when an application thread already has the mutex locked. If the application thread is not performing a blocking action, it will finish in a finite time and exit the library, thereby allowing the checkpoint thread to acquire the lock and perform its work. However, if the application thread is blocking (e.g., reading from a socket that has no data available), it must be interrupted and told to yield to the checkpoint thread.

The only locations in LAM/MPI that will block are within MPI SSI modules. Both the `rpi` and `coll` modules have an “interrupt” function that the selected `crmpi` module will invoke to attempt to gain control of the MPI library. This interrupt function is specific to each module; its purpose is to interrupt blocking operations and yield to the checkpoint thread.

The interrupt functions are invoked repeatedly until the `crmpi` module is able to lock the MPI library. The `crmpi` module holds the lock until after both the checkpoint and continue (or restart) phases have completed. This prevents the user application from invoking MPI functions while the checkpoint or restart is occurring, eliminating many types of potential race conditions. Figures 7.4 and 7.5 show two scenarios how this lock is used.

Once the `crmpi` module has acquired the MPI library lock, it will coordinate all other modules to prepare for checkpoint. Specifically, all available `rpi` and `coll` modules will

Time	Checkpoint thread	Application thread
$t_i$	Receive checkpoint request	...executing outside MPI library...
$t_{i+1}$	Acquire MPI library mutex	
$t_{i+2}$	Invoke “prepare to checkpoint” interface functions on all selected modules	Call MPI function, block on acquiring mutex
$t_{i+3}$	Back-end checkpoint system is invoked	
Continue		
$t_{i+4}$	Return from checkpoint	
$t_{i+5}$	Invoke “continue after checkpoint” interface functions on all selected modules	
$t_{i+6}$	Release MPI library mutex	
$t_{i+7}$		Acquire MPI library mutex, proceed with MPI function call

Figure 7.4. Sequence of events when the application thread is executing outside the MPI library when a checkpoint request arrives. Although the checkpoint/continue phases are shown here, the same sequence occurs in the restart phase (albeit in a different process).

have their “prepare for checkpoint” API functions invoked. These modules are responsible for doing whatever is necessary to prepare for a checkpoint. The `crtcp rpi`, for example, coordinates with its peers to drain all of its TCP sockets. This ensures that there are no messages “in flight” on the network; all relevant state is now contained within each process.

Not all modules need to do anything special to prepare for checkpoint. The `lam_-basic` and `smp coll` modules, for example, are implemented on top of MPI point-to-point message passing. All checkpoint preparations and recovery, therefore, are performed by the `rpi` module.

Time	Checkpoint thread	Application thread
$t_i$		Call MPI function, acquire mutex
$t_{i+1}$	Receive checkpoint request	Execute blocking system call in MPI library
$t_{i+2}$	Attempt to acquire MPI library mutex; fail	
$t_{i+3}$	Invoke selected module interrupt functions	
$t_{i+4}$		Receive interruption; release MPI library mutex; yield
$t_{i+5}$	Acquire MPI library mutex	Block on acquiring mutex
$t_{i+6}$	Invoke “prepare to checkpoint” interface functions on all selected modules	
$t_{i+7}$	Back-end checkpoint system is invoked	
Continue		
$t_{i+8}$	Invoke “continue after checkpoint” interface functions on all selected modules	
$t_{i+9}$	Release MPI library mutex	
$t_{i+10}$		Acquire MPI library mutex, proceed with MPI function call

Figure 7.5. Sequence of events when the application thread is blocking in the MPI library when a checkpoint request arrives. Although the checkpoint/continue phases are shown here, the same sequence occurs in the restart phase (albeit in a different process).

Timestep	<code>mpirun</code>	MPI processes
$t_i$	Return from checkpoint	Return from checkpoint
$t_{i+1}$	Resume execution	Invoke “continue after checkpoint” interface functions on all selected modules
$t_{i+2}$		Resume execution

Figure 7.6. Sequence of events in the continue phase.

### 7.2.3 Checkpoint

Once the coordination with other modules is complete, the `cr` modules invoke the back-end checkpointing system to actually perform the checkpoint. This creates a set of process images, one for each MPI process and one for `mpirun`, that can be used to restart the process. This set of images is guaranteed to be globally consistent because the coordination in the “prepare to checkpoint” phase ensured that all sent messages were received – there is no data left on the network. Hence, the state in the union of all processes is consistent in itself, and can be saved and restored in a consistent manner.

### 7.2.4 Continue After Checkpoint

After a globally consistent checkpoint is taken, the MPI processes are allowed to continue execution. Selected modules have their “continue” interface functions invoked to perform any necessary post-checkpoint actions. Afterwards, the checkpoint thread unlocks the MPI library and allows normal execution to resume. Figure 7.6 shows the sequence of events during the continue phase.

### 7.2.5 Restart After Checkpoint

Similar to the continue phase, when an MPI process is restarted, selected modules have their “restart” interface functions invoked to perform any necessary post-checkpoint

Timestep	mpirun	MPI processes
$t_i$	mpirun is re-executed	
$t_{i+1}$	Restart MPI processes	
$t_{i+2}$	Resume execution	Return from checkpoint
$t_{i+3}$		Invoke “restart after checkpoint” interface functions on all selected modules
$t_{i+4}$		Resume execution

Figure 7.7. Sequence of events in the restart phase.

actions. Typical actions in the restart functions include re-allocating resources, discovering the new location of migrated processes, and re-creating communication channels. After the restart functions have completed, the checkpoint thread unlocks the MPI library and allows normal execution to resume. Figure 7.7 shows the sequence of events during the continue phase.

#### 7.2.6 Module Selection Mechanism

The scope of `cr` module selection in the life of an MPI application (i.e., from `MPI_INIT` to `MPI_FINALIZE`). The algorithm used for selection is essentially the same as what is described in Section 3.2.7; all available `crmpi` modules are examined during `MPI_INIT` and asked if they want to run. If the module wants to run, it provides a priority. The priorities of all modules who want to run are sorted and the module with the highest priority is selected.

Once the MPI processes achieve consensus on which `crmpi` module to run, they send the name of the selected module to `mpirun`. `mpirun` then selects the corresponding `crlam` module.

### 7.2.7 Interaction With Other Modules

`rpi` and `coll` modules must be checkpoint-aware in order for an MPI process to be capable of checkpoints. Both of these component types have four interface functions specifically to support checkpoint/restart functionality:

- **Interrupt.** Interrupt the module if it is in a blocking function and yield to the checkpoint thread. For non-blocking modules (e.g., the `gm rpi` module never blocks), this may be a no-op.
- **Checkpoint.** Corresponding to the checkpoint phase, this function performs actions prior to a checkpoint.
- **Continue.** Corresponding to the continue phase, this function performs actions after a checkpoint in the same process that was checkpointed.
- **Restart.** Corresponding to the restart phase, this function performs actions after a checkpoint in a newly-started process.

The actions of each function are module specific. For example, collective modules that are layered on point-to-point MPI functionality need not perform any additional actions; all handling is performed by the `rpi` module. The `lam_basic` and `smp coll` modules in the LAM 7 distribution both conform to this model.

Two `rpi` modules have been adapted to support these four functions: `crtcp` and `gm`. The `crtcp` module uses TCP sockets for MPI communications. During the checkpoint phase, it drains all sockets to generate a global consistent state (i.e., all messages are in processes – the network is empty). The network is drained by using LAM’s out-of-band communication mechanism to exchange sent/received byte counters between each pair of MPI processes. Using this information, each process knows exactly how much data is “in flight” on the network and can safely receive all of it. This process is shown in Figure 7.8. Note that due to `crtcp`’s architecture, normal MPI progression is able to be used; no secondary buffers or message rollback mechanisms need to be utilized [35]. The continue phase is a no-op, but the restart phase re-establishes TCP sockets between all MPI processes. Normal MPI progression then resumes.

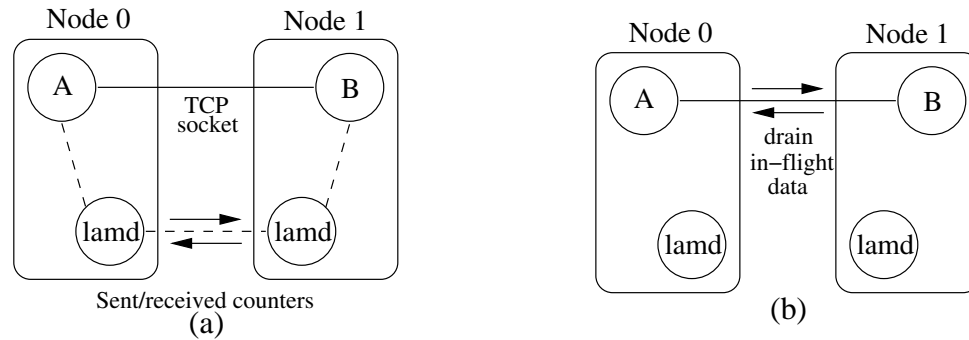


Figure 7.8. Draining TCP sockets before checkpoint: (a) Processes A and B exchange the sent/received byte counter information using LAM’s out-of-band communication system. (b) Processes A and B receive data from the in-band channel until their counters match what was received in (a).

Similar to the `tcp` module, the `gm` module drains the network during the checkpoint phase. However, the `gm` module must close all connections during the checkpoint phase because the GM library reflects state in an operating system kernel module which cannot be checkpointed. Closing the GM connections – and therefore zeroing out the associated process’s state in the GM kernel module – is the only option. Hence, during both continue and restart phases, GM channels must be re-opened.

### 7.3 Implemented Modules

Two `cr` module pairs are included in LAM/MPI version 7: `blcr` and `self`. These modules serve both as reference algorithms as well as examples of two different implementation models.

#### 7.3.1 The `blcr` Module

The Berkeley Lab’s Linux Checkpoint/Restart project (BLCR) [33] is a robust, kernel-level checkpoint/restart implementation. It can be used either as a stand-alone system for checkpointing applications on a single node, or by a scheduling system or parallel communication library for checkpointing and restarting parallel jobs running on multiple

nodes. BLCR is implemented as a Linux kernel module (for recent 2.4 versions of the kernel, such as 2.4.18) and a user-level library. A kernel module implementation has the benefit that it allows BLCR to be easily deployed by new users without requiring them to patch, recompile, and reboot their kernel. While the current implementation of BLCR only supports checkpointing of single processes (including multi-threaded processes), checkpointing of process groups, sessions, and a full range of Unix tools is planned.

BLCR provides a simple user-level interface to libraries/applications that need to interact with checkpoint/restart. It provides a mechanism to register user-level callback functions that are triggered whenever a checkpoint occurs, and that continue when the process restarts (or a periodic checkpoint for backup purposes completes). Two kinds of callbacks can be registered: signal-based callbacks that execute in signal-handler context, and thread-based callbacks that execute in a separate thread. These callbacks allow the application to shutdown its network activity (and perform analogous actions on some other uncheckpointable resource) before a checkpoint is taken, and restore them later. Callbacks are designed to be written as shown in Figure 7.9.

## Implementation

`blcr crlam` and `crmpi` modules have been implemented in LAM 7 that use the BLCR system as a back-end checkpointer. At the start of execution, both of the `blcr` modules register thread-based and signal-based callback functions with BLCR. A checkpoint request is initiated by using the `cr_checkpoint` command with the PID of `mpirun`. This triggers the signal and thread callbacks in `mpirun` that were registered during initialization. The thread-based handler propagates the checkpoint request by using LAM services to launch a `cr_checkpoint` for every MPI process in the application (see Figure 7.10). After some additional bookkeeping, the `crLAM` module invokes the BLCR `cr_checkpoint()` function to checkpoint `mpirun`.

```

void callback(void *data_ptr) {
    struct my_data *pdata = (struct my_data *) data_ptr;
    int did_restart;

    /* ... checkpoint-time shutdown logic ... */

    /* Tell system to take the checkpoint */

    did_restart = cr_checkpoint();

    if (did_restart) {
        /* ... actions to restart from a checkpoint ... */
    } else {
        /* ... actions to continue after a checkpoint ... */
    }
}

```

Figure 7.9. Template for BLCR callback functions. The state of the entire process (including the callback’s execution) is saved during the `cr_checkpoint()` call. The return value from `cr_checkpoint()` indicates whether the process continued after the checkpoint or was started in a new process.

The `cr_checkpoint` commands launched by `mpirun` invokes the thread-based and signal-based callbacks invoked in each of the MPI processes. The locking scheme described in Section 7.2.2 is used; the `crmpi` module eventually gains control of the MPI library. It invokes all available MPI modules’ “prepare for checkpoint” functions and then invokes the BLCR function `cr_checkpoint()` to checkpoint the MPI process.

As shown in Figure 7.9, the return value of the `cr_checkpoint()` function distinguishes between the continue and restart phases. Based on this value, the `bcr` module invokes the proper set of MPI module functions (“continue after checkpoint” or “restart after checkpoint”), unlocks the MPI library, and resumes execution.

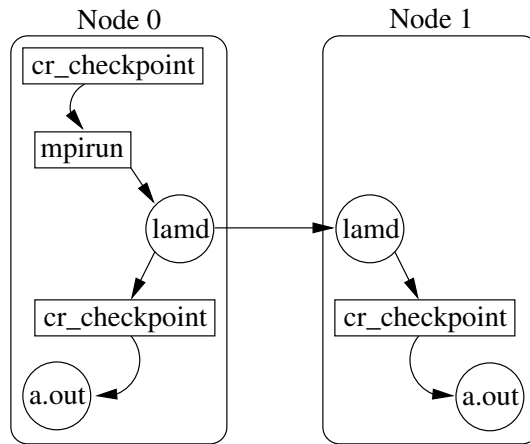


Figure 7.10. The `bcr` module in `mpirun` propagates the checkpoint request by using LAM run-time environment services to launch a `cr_checkpoint` for every MPI process.

### Usage

The `bcr` modules efficiently perform system-level checkpointing and save the state of the parallel application. A process image file is created for each MPI process; this may require significant amounts of disk space. Users and system administrators need to plan accordingly.

Per the requirements of the `cr` component framework discussed in the beginning of this chapter, the `bcr` module requires no source code changes in the MPI application. This allows even legacy MPI applications to be checkpointed.

### 7.3.2 The `self` Module

Since parallel/MPI-aware system-level checkpointers have not existed until recently, many parallel developers have added *application-level checkpointing* into their codes. For example, the application will output a set of “restart” files at regular intervals containing essential variables, state, and any other data necessary to save the state of the computation. If the application fails (e.g., if a node dies in the middle of the run), the application can be

restarted by reading in the last set of restart files. The data read in from the files is used to re-seed the application's state.

Although application-created restart files can be orders of magnitude smaller than their system-created counterparts, it places a tremendous burden on the application developer. Code must be written to marshal essential state, create and manage restart files, read the restart files back in, and re-seed the essential state from the input. Depending on the nature of application, this can be a highly complex task that affects large portions of the code base. This seems to violate one of the `cr` framework's design goals.

However, since many applications have already been developed using this methodology, providing asynchronous access to this functionality is actually completely in the spirit of the `cr` framework. Specifically, the checkpointing code *already* exists in many user applications – the `cr` framework just offers an alternate method to access it. The `self` module is therefore a bridge between application-level checkpointing and formalized checkpoint/restart systems.

## 7.4 Results

Two main sets of tests were run on the implemented `cr` modules – correctness and performance.

### 7.4.1 Correctness

A test suite was written to verify the correctness of the `cr` system through the `blcr` module. A series of parallel MPI test programs were written to check different aspects of the checkpoint/restart implementation. Several tests conducted specific message passing patterns (e.g., all-to-all, ring, pairwise passing, etc.) while others followed the general algorithm listed below. Each test case was designed to test a specific case of checkpoint/restart functionality (e.g., checkpoint while blocking on message passing, checkpoint while mes-

sages are in flight, checkpoint while in collectives, checkpoint while in a specific MPI API function, etc.). All tests checked for correctness of received messages to ensure that any messages interrupted by checkpoints were actually received correctly.

- Invoke `MPI_INIT`
- Even-numbered ranks in `MPI_COMM_WORLD` sleep for a finite number of seconds
- Invoke the function under test
- Checked the received data for correctness
- Invoke `MPI_FINALIZE`

Table 7.1 lists the collective MPI functions that were tested; Table 7.2 lists the point-to-point MPI functions that were tested. Each test was first run without checkpointing to verify that the test itself was correct. Message passing patterns and data checking routines were tested for validity and correctness. Each test was then checkpointed while it was running and allowed to complete. The checkpoint files were then used to restart the application, which was then allowed to run to completion. The tests were run with different numbers of processes in an attempt to generate obscure race conditions and unlikely combinations. Passing the test suite indicated the following:

- Checkpoint requests can be successfully received in `mpirun` and propagated to the MPI processes.
- Processes can be checkpointed and restarted in the MPI framework.
- MPI communication channels can be drained, closed, and re-opened in a new process.
- MPI applications can be interrupted and preempted for checkpoint.
- `MPI_FINALIZE` can successfully shut down an MPI process regardless of whether it was the original application or a restarted process.

TABLE 7.1

Collective MPI functions tested with checkpoint / restart functionality

---

MPI_ALLTOALL	MPI_BARRIER	MPI_BCAST
MPI_GATHER	MPI_GATHERV	MPI_SCATTER
MPI_REDUCE		

---

TABLE 7.2

Point-to-point MPI functions tested with checkpoint / restart functionality

---

MPI_IRECV	MPI_IRSEND	MPI_ISEND
MPI_ISSEND	MPI_RECV	MPI_RSEND
MPI_SEND	MPI_SENDRECV	MPI_SSEND
MPI_WAIT		

---

TABLE 7.3

Description of the Indiana University Computer Science Thor cluster

Number of nodes	8
Processor type	Xeon
Processor count	2
Processor speed	2.8 GHz
Cache size	512 KB
RAM	2 GB
Operating system	Red Hat 8.0 (plus updates)
Linux kernel version	2.4.21-smp
Compiler	GNU, v3.2-7
Compiler flags	-O3 -pthread
Interconnects used	Gigabit Ethernet, Myrinet (GM library v2.0.10)
Other relevant software	None

#### 7.4.2 Performance

Performance tests were conducted on the Thor Computer Science development cluster at Indiana University, described in Table 7.3.

The performance of a checkpointing parallel job can be decomposed into two main parts: the cost of the back-end checkpointing system (BLCR, in this case) and the overhead added by LAM/MPI.

The cost of the back-end checkpointer operating in parallel is directly related to how long it takes to checkpoint a serial process. Intuitively, the time necessary for a checkpoint is dependent on the size of the process because it will be written to stable storage. Figure 7.11 shows the wall clock time to checkpoint a single serial process of varying sizes to a local disk with BLCR. The Figure shows the results of running the checkpoints on each node of the Thor cluster. Each node had been freshly booted; the entire cluster was otherwise dormant. Each time presented in the graph is best checkpoint time out of 10 checkpoints for a given size on a specific node. The Figure shows a clear correlation between process size and how long it takes BLCR to checkpoint it.

However, for sizes above 512 MB, the checkpoint times start to vary. This is due to disk I/O activity and virtual memory overhead; the only difference between the nodes used in this experiment were how much local disk space was available. Figure 7.11 shows that `thor5` had among the worst checkpoint times as process sizes increased; `thor5` had the least amount of space available (although it had an order of magnitude more disk space available than was required).

Measuring the overhead added by LAM is difficult, not only because of the performance variance in the back-end checkpointing system, but also because it comes from multiple sources. The first source of overhead is enabling the ability to generate checkpoints. This entails additional bookkeeping within LAM (described in Section 7.2.7) that is normally unused, and is typically comprised of maintaining peerwise counters that en-

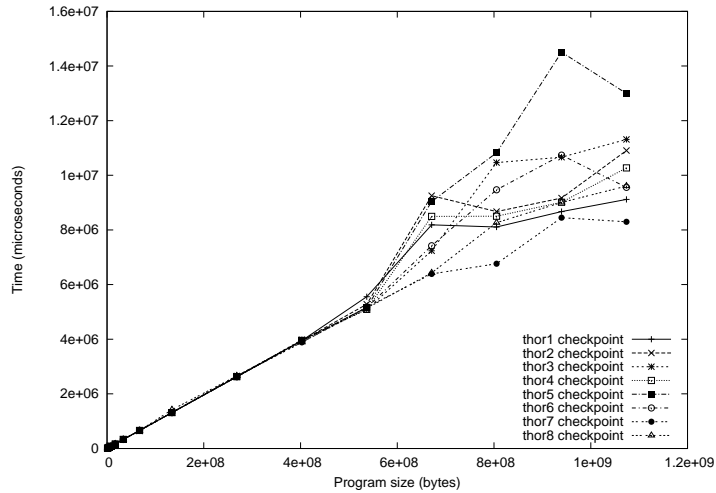


Figure 7.11. Wall clock execution time for checkpointing serial processes of varying sizes using BLCR.

able draining the network at checkpoint time. This functionality has been added to the `crtcp rpi` module. For this reason, the checkpoint/restart-capable `crtcp` module is distinct from the `tcp` module – it was thought that there might be a noticeable loss in performance for applications that did not use checkpointing.

The `crtcp` module has two modes: checkpointing disabled (where it should perform almost exactly like the `tcp` module) and checkpointing enabled (where it performs the additional bookkeeping). Both modes of `crtcp` were measured against the `tcp` module. Table 7.4 shows the NAS parallel benchmarks [6, 4, 5] run on 8 nodes on the Thor cluster using the `tcp` and `crtcp rpi` modules. The Table shows performance of three classes of the NAS parallel benchmarks across eight nodes of the Thor cluster using the `tcp` and `crtcp` modules (with checkpointing enabled and disabled, although no checkpoints were taken). Times shown are wall-clock execution in seconds. The “Difference” column is the `crtcp` (enabled) column minus the `tcp` column, expressed both as time and percentage of the

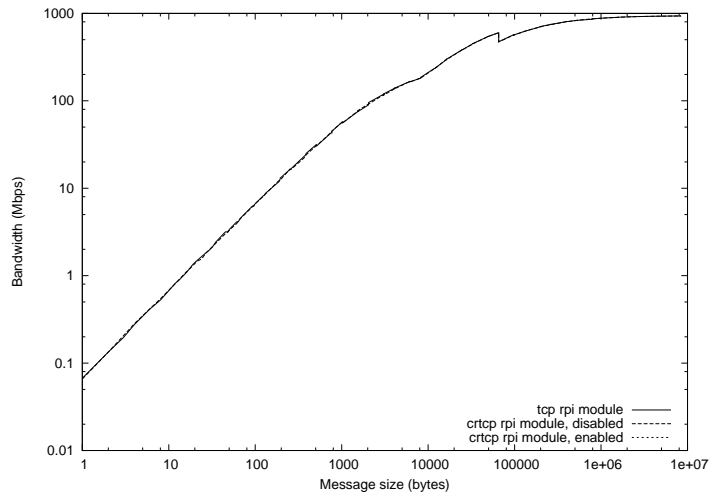


Figure 7.12. NetPIPE throughput on AVIDD-B cluster of the `tcp` module compared to the `crtcp` module, both with and without checkpointing support enabled.

`tcp` time. The difference is negligible.

Additionally, the NetPIPE analyzer [118] was used to compare the bandwidth throughput of the `tcp` and `crtcp` modules (in both modes). Figure 7.12 shows their performance; they are essentially identical (note the default short message size was used, resulting in a “dip” in performance at 64 KB; see Section 5.3.2 for an explanation);

This raises an interesting side issue: since the performance is essentially identical, why not implement the additional bookkeeping in the `tcp` module itself? Indeed, the `crtcp` module started as a direct copy of the `tcp` module. The reason for this is that by developing a separate module, the stability of the existing `tcp` module was never compromised and could still be used for production parallel jobs. The `crtcp` module became a research tool and did not need to conform to the rigorous stability standards required of the `tcp` module until it had been fully developed, understood, and finalized. Having a parallel development structure – especially one that is easily “swappable” at run-time provides an

TABLE 7.4

Checkpoint / restart overhead measurements in LAM/MPI on the Thor cluster

Class	Size	tcp	crtcp (disabled)	crtcp (enabled)	Difference	
cg benchmark						
A	$64 \times 64 \times 64$	1.96	2.32	2.19	0.23	11.7%
B	$102 \times 102 \times 102$	84.58	82.64	82.23	-2.35	-2.8%
C	$162 \times 162 \times 162$	202.50	201.16	201.83	-.67	0.0%
ep benchmark						
A	$64 \times 64 \times 64$	12.02	12.03	11.99	-0.03	-0.2%
B	$102 \times 102 \times 102$	47.93	47.96	47.94	0.01	0.0%
C	$162 \times 162 \times 162$	191.74	191.78	191.77	0.03	0.0%
is benchmark						
A	$64 \times 64 \times 64$	0.99	1.00	1.00	0.01	1.0%
B	$102 \times 102 \times 102$	4.00	3.99	3.98	-0.02	-0.5%
C	$162 \times 162 \times 162$	16.16	16.13	16.11	-0.05	-0.3%
lu benchmark						
A	$64 \times 64 \times 64$	41.22	41.56	41.95	0.73	1.7%
B	$102 \times 102 \times 102$	167.10	168.88	168.55	1.45	0.9%
C	$162 \times 162 \times 162$	671.96	669.18	671.38	-0.58	0.0%
mg benchmark						
A	$64 \times 64 \times 64$	1.78	1.82	1.82	0.04	2.2%
B	$102 \times 102 \times 102$	8.40	8.36	8.47	0.07	0.8%
C	$162 \times 162 \times 162$	52.89	53.28	53.22	0.33	0.6%

extremely useful research capability while simultaneously providing a controlled, known correct baseline to compare with.

The main sources of overhead added by LAM are the propagation of the checkpoint request and the drain of the network. The sum of these overheads should be at least one order of magnitude smaller than performing the checkpoint and writing the process image to disk. Unfortunately, accurately measuring these costs is nearly impossible in an asynchronous cluster environment. As such, additional instrumenting code was inserted into LAM to artificially create a synchronization point in `mpirun` that can be measured. Specifically, the highest ranking process in `MPI_COMM_WORLD` sends a synchronization message back to `mpirun` when it finishes draining its MPI communications network. Since the highest ranking process in `MPI_COMM_WORLD` is always the last one to finish draining its network, this is guaranteed to be end of all LAM-induced overhead. Hence, `mpirun` measures the timespan from when it initiates the first checkpoint request propagation until it receives a synchronization message back from the last MPI process. Figure 7.13 shows this graphically.

The time interval measured in `mpirun` is therefore a crude approximation of the actual overhead. However, it provides an order of magnitude suitable for comparison to the cost of the back-end checkpoint. Figure 7.14 shows the `mpirun` measurements of LAM overhead for parallel applications consisting of two different process sizes – 1 MB and 256 MB. The line for each parallel application increases roughly linearly with the number of processes because `mpirun` must propagate the checkpoint request to more processes and then all processes must participate in draining the MPI communications network. Note that the number of processes is what drives the overhead time, not the size of each application. With one process, the upper bound on LAM-specific overhead is 114 milliseconds, increasing to 187 milliseconds at eight processors – adding approximately 9 milliseconds per process. As shown in Figure 7.11, checkpointing a 400 MB serial

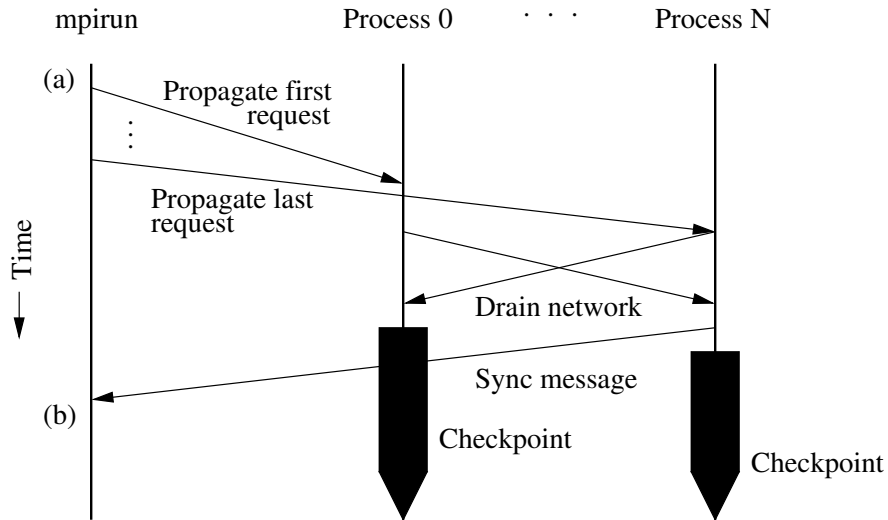


Figure 7.13. Depiction of the message passing during a checkpoint, including the artificial “sync message” inserted solely for measuring the magnitude of LAM overhead. The upper bound of LAM overhead is therefore the time interval between (a) and (b).

process takes approximately 4 seconds. Taking a conservative estimate (using the `thor7` line from Figure 7.11), this time increases approximately 0.7 seconds with every hundred megabytes (until the checkpoint times become erratic).

Putting all the sources of overhead together, the LU NAS parallel benchmark was run with the `b1cr` module’s checkpointing enabled, taking checkpoints at varying frequencies ranging from never (i.e., running with checkpointing enabled but not taking a checkpoint) to every 40 seconds. All checkpoint images were stored to local disk.

Table 7.5 shows the results. The number in parentheses in each entry is the total number of checkpoints taken during the run. Clearly, the number of checkpoints increases the overall wall-clock execution time. However, the majority of the additional time is taken by the BLCR checkpointer. For example, the LU class C checkpoints each generated image files over 200MB. Even in the worst case shown in Table 7.5 (LU, class C, with 19 checkpoints), execution time increased approximately 7.8% as compared to running with no checkpoints [83].

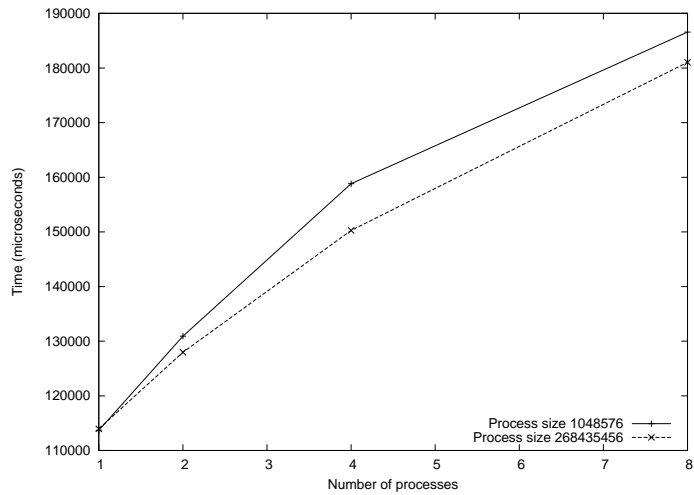


Figure 7.14. Wall clock time of LAM overhead to checkpoint a parallel job with differing numbers of processes. Two outputs are shown; one with individual process sizes of approximately 1 MB, the other with 256 MB processes.

TABLE 7.5

Wall-clock execution time of the LU NAS parallel benchmark with checkpoints being taken at different frequencies

Class	No checkpoints	10 seconds	20 seconds	30 seconds	40 seconds
A	42.70 (0)	44.28 (3)	43.74 (2)	43.28 (1)	43.07 (1)
B	173.84 (0)	181.89 (10)	178.28 (6)	176.96 (4)	175.87 (3)
C	671.38 (0)	724.23 (19)	725.84 (15)	712.50 (12)	716.39 (14)

## CHAPTER 8

### CONCLUSIONS

Research, development, and maintenance of an MPI implementation is a complex task. Even well-abstracted, logically constructed implementations effectively impose a significant learning curve on new developers and make third party development difficult at best. Utilizing a component-based architecture in an MPI implementation forces formal abstractions and library boundaries between different aspects of MPI functionality. These abstractions are realized into multiple independent and logically discreet component architectures, each of which represents one small section of the overall MPI implementation. Modules that implement the component interface therefore only represent a small amount of code compared to the rest of the MPI implementation.

For example, Table 2.3 (page 20) shows that the total lines of code in LAM/MPI (excluding documentation) is 275,139. Table 1.2 (page 10) shows that the line counts for each of the `boot`, `coll`, `cr`, and `rpi` component architectures and all implemented modules are 8,709, 11,711, 5,158, and 51,798, respectively. These represent 3%, 4%, 2%, and 19% of the overall MPI implementation. Recalling that the line counts represent the component architecture *and* all the modules that implement its interface, these numbers show that the code required for a single module is small compared to the overall MPI implementation.

Clearly, the amount of code required to implement a module (and therefore to implement a new piece of MPI functionality) has a direct impact on both the learning curve and

the time required by those wishing to implement small sections of MPI functionality (e.g., researchers developing new collective algorithms). Providing small component architectures for discreet MPI functionality allows rapid development and deployment for both software and hardware researchers and vendors. For example, third parties can perform meaningful experiments in message passing systems with real-world applications on a robust development and research platform instead of extensively modifying a monolithic implementation, or worse, fully implementing their own MPI.

Component concepts also benefit system administrators, end users, and ISVs. System administrators can install only modules that are relevant to their particular systems. Users can compose collections of available modules at run-time to create an MPI implementation that is uniquely suited for their application and the run-time environment in which it is operating. ISVs can dramatically simplify their logistics by only distributing one executable that will work in any LAM/MPI environment. Hardware vendors and ISVs can also create and distribute modules that are tailored for their products (e.g., network interconnects, collective algorithms) that plug-in to an existing LAM/MPI installation.

This dissertation presented a system component architecture for the LAM implementation of MPI and showed how it exhibits the benefits described above with no loss in performance. The four component architectures designed and implemented in this work are formalizations of a portion of an MPI implementation. Each represents years of prior work, theoretical analysis and design, practical software engineering, and real-world requirements. Utilizing these component architectures as an MPI implementation tool – and, more importantly, utilizing and continuing the ideas they represent – represents a fundamental step forward in MPI implementation methodology, functionality delivered to all users (including system administrators and ISVs), and research capabilities provided to third party developers.

The following sections summarize the SSI meta framework and each of the individual

component frameworks.

## 8.1 The SSI Meta Framework

The SSI is the meta framework that ties together the four component frameworks in LAM/MPI. It provides common functionality to all frameworks, allows interaction between them, and helps manage the components in each framework. Two important services that it offers are loading and unloading of components and the module parameter registry.

Static components are found by looking through a list of component handles that was assembled when LAM was configured and compiled. Shared library components are discovered by searching directories at run-time and eagerly loaded into target processes. The capability to load arbitrary modules at run-time allows anyone to distribute components; there is no need to modify LAM/MPI itself or have a custom implementation of LAM/MPI (a fork from the main LAM code base). More importantly, a compiled and linked MPI application is wholly independent of what modules it may find at run-time; ISVs and system administrators can distribute *a single executable for any LAM/MPI environment*. Users can therefore have a single compiled version of their application that will automatically discover and utilize new modules at run-time. Additionally, unrelated modules can be composed together, creating a combinatorial effect for delivered functionality. For example, alternate collective algorithms can be substituted into production MPI codes – with no changes or recompilation of the production MPI codes – for correctness and performance testing.

Module parameter passing empowers the developer to easily provide “tweakable knobs” to the end user. While the implementation of the module registry is not overly complicated, what it represents – the ability to easily expose algorithmic and behavioral characteristics – fundamentally changes how a developer will write MPI functionality. Constants

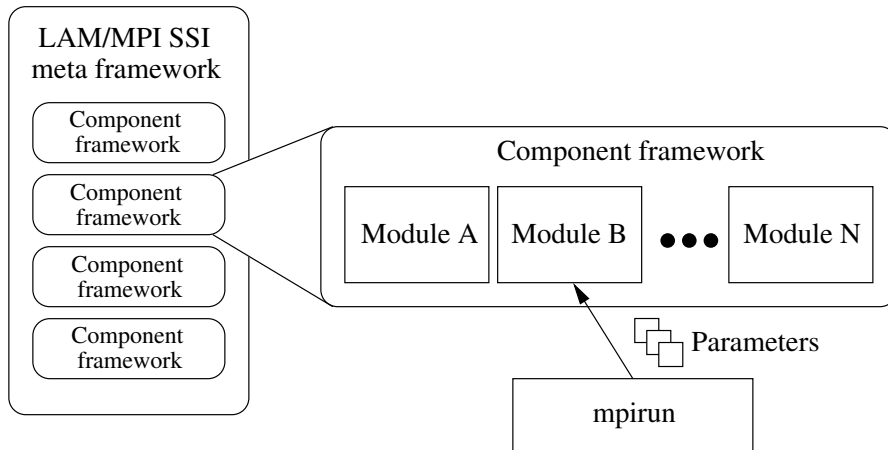


Figure 8.1. LAM/MPI is a component system architecture that manages multiple different component frameworks (or “types”); zero or more modules may be available from a given type. This figure depicts a component type with  $N$  modules, and shows `mpirun` passing in run-time parameters to module  $B$ .

and hard-coded decisions are no longer necessary; providing the user with choices allows arbitrary performance tuning in ways that only the application developer will be able to exploit.

Figure 8.1 shows this relationship graphically (repeated from Chapter 2).

## 8.2 The boot Component Framework

The `boot` component framework expands LAM’s support of run-time environments beyond the traditional `rsh/ssh`-based clusters. Arbitrary run-time environments can now be supported, exploiting native robust job control semantics, detailed accounting statistics, and I/O forwarding.

The design of the `boot` component framework was created by analyzing how parallel jobs are started from two different systems: `rsh/ssh` and PBS. An algorithm was abstracted into a common interface that applied not only to those two run-time environments, but also to BProc and Globus environments. Thus, the design was validated by the

successful creation of three components supporting environments where LAM/MPI had never been used before.

Testing results in the PBS environment show that all of the above claims are true: the use of the native PBS TM interface provides better job control (both in launching and killing jobs) and accurate, detailed accounting data.

### 8.3 The rpi Component Framework

Transforming the existing compile-time RPI interface to the rpi component framework is perhaps one of the most user-visible aspects of the component framework. The end result is that users can change which underlying network their application uses without needing to recompile or re-link their application. Such functionality is tremendously useful for ISVs who now only need to distribute one binary for all LAM/MPI environments.

Performance results comparing the TCP point-to-point component framework to the previous generation of LAM/MPI (a monolithic implementation) show no discernible performance difference. Comparisons of the gm Myrinet module against native GM library message passing showed only modest performance differences.

### 8.4 The coll Component Framework

The coll component framework was one of the major motivations behind the entire SSI system: significantly lowering the learning curve to implement new MPI collective algorithms in the framework of a production-quality MPI implementation.

Performance measurements comparing the LAM “basic” collective algorithms in the coll framework to the same algorithms in the previous, monolithic implementation of LAM/MPI were presented. Not only were the performance results effectively identical, the introduction of the hierarchical implementation model in Section [6.1.3](#) creates inter-

esting future research possibilities for multi-tiered latency environments (e.g., the Grid).

## 8.5 The `cr` Component Framework

Although checkpoint/restart is a well-studied topic and much has been written about both serial and parallel implementations, the `cr` component framework is the first robust, production-quality implementation in an open-source MPI implementation. Although more work remains to be done, having the ability to provide involuntary asynchronous checkpoints is tremendously important to the high-performance computing world. Gang scheduling is now within reach of the commodity market, as is rollback recovery for fail-stop kinds of errors (e.g., a node death in the middle of a run).

## 8.6 Delivered Software and Documentation

All versions of LAM are subjected to rigorous development, maintenance, and testing procedures before release. The open source community – including vendors who either use or must support LAM/MPI in their commercial projects – helps with this process by actively contributing ideas, bug fixes, and peer reviewing the LAM code base.

Many of the concepts discussed in this dissertation are already available in LAM/MPI v7.0. A full set of documentation (approximately 11,470 lines of  $\LaTeX$  code, or 133 printed pages) of documentation was included in the 7.0 release, suitable for developers to write their own modules. The remaining work will be available in LAM/MPI v7.1, expected to be released in May 2004 (for example, dynamic SSI modules are not available in v7.0).

## 8.7 Future Work

Follow-on work to this research can proceed in a variety of directions. First, the module frameworks themselves can be used to explore their respective domains. Point-to-

point and collective message passing, for example, still have many unanswered questions. The design of the MPI specification was specifically intended to allow multi-threaded MPI applications. But the current rpi design is not capable of handling multiple threads, at least not without significant modifications to its design. Significant questions remain about how to implement a multi-threaded-capable MPI implementation where threads can make message passing progress independently [39]. Multi-threaded programming has been common in the TCP sockets-based communication community for years. For example, all modern, high-performance web servers either run in multi-threaded or multi-process mode, where different agents on the same server handle incoming requests. This programming model has proved to be extremely useful for client-server applications; the abstractions afforded by simply dispatching a thread to handle a blocking action can dramatically simplify coding logic. Since MPI applications are essentially collections of independent agents – analogous to client-server models – it follows that they could certainly benefit from the same threaded abstractions and implementation models. Although some vendor MPI implementations allow multi-threaded MPI implementation, it has not been widely researched or accepted in the academic and experimental communities. The question of how to efficiently provide simultaneous message passing progress over multiplexed communication channels still needs much research.

The prevalent use of commodity clusters has raised the importance non-standard network topologies (potentially created for specialized problems). Infiniband, for example, has a highly complex programming model. Many questions remain about how to incorporate Infiniband into high-performance computing environments – both in the point-to-point and collective arenas. For example, emerging research has shown the potential for using the Infiniband interface to natively perform some MPI collective operations. Any performance speedup in collective operations directly benefit user application wall-clock execution time; even if only `MPI_BCAST` is optimized on Infiniband, any application

that heavily uses `MPI_BCAST` will see immediate performance increases. Other proprietary networks have similar potential for optimization, both in the point-to-point and collective domains: Myrinet, Quadrics, and even UDP. Continued research in these areas needs to determine ways of mapping the functionality provided by the interconnect to MPI semantics.

Fault tolerance is also becoming increasingly important. High failure rates can render large computational resources effectively useless unless models can be determined that allow some form of continued operation in the presence of faults. Checkpoint/restart is one such method, but is not necessarily a scalable solution. For example, checkpointing large-memory jobs running on thousands of nodes, particularly if none of the nodes have local disk storage, can cause enormous networking bottlenecks, and potentially take longer than the mean time between failure. In such a scenario, it is possible (and likely) that a failure will occur *during* a checkpoint, raising another set of issues and problems.

Other fault tolerant models have started to be explored, but much more work is required, such as comprehensive studies of what real-world applications *need* for fault tolerance, analysis of exactly what kinds of faults should be detected and how they should be detected, and the exact role of that the middleware will have in the process. Indeed, the MPI standard itself leaves much to be defined in terms of fault tolerance. Preliminary research indicates that aside from system-level checkpointing, an MPI implementation cannot perform all aspects of fault tolerance – the application must be involved. If this is true, programming models must be developed to allow applications to be developed that can utilize middleware fault detection and handling capabilities. Hence, definitions for appropriate MPI behavior need to be created, both for MPI-1 and for more difficult questions such as what exactly fault tolerance means in the context of MPI-2 dynamic processes.

Another topic becoming prevalent is the subject of MPI over wide-area networks

(WANs). Indeed, the Magpie-based collective algorithms implemented in this work were originally developed for WAN scenarios. Interoperable MPI (IMPI) and MPICH-g2 were also designed for WAN and Grid-based execution; MPICH-g2 has topology-aware collectives that extend the Magpie concepts across multiple layers of latency (not just two). Computational grids spread across multiple organizations in different physical locations present significant challenges not only for MPI implementations, but for definitions and expectations of reasonable levels of performance. Globus and other Grid-related projects are exploring execution, security, and resource sharing aspects of such collaborations. Application-level definitions are still lacking, however; the appropriate balance between making an application fully topology-aware and having the middleware transparently hide all such details has not yet been found.

Finally, it has been remarked that message passing is the assembly language of parallel processing. If this is true, it implies that there are “better” abstractions that can be layered on top of message passing, analogous to how modern, high-level languages are implemented on top of assembly language. Indeed, most non-computer scientists only care about generating results. Message passing (and MPI) has enabled them to expand from serial to parallel computations, potentially speeding up execution and making larger problem sizes possible. Message passing is a generalized solution, applying to large classes of computational problems. If message passing really is analogous to assembly language, then suitable generalized parallel abstractions need to be researched that supersede traditional send/receive and put/get models. Such research will need to balance many factors, including user application requirements, efficiency, and scalability.

## APPENDIX A

### SSI FRAMEWORK INTERFACE

The SSI component system architecture is described in Chapter 3. This Appendix describes the technical details and requirements for modules used in any of LAM/MPI's component frameworks [126].

Like any software package, modules need to be configured, compiled, and installed. This can be done either independently (i.e., outside of the LAM/MPI source tree and build/install process) or as part of the LAM source tree's configuration, compilation, and installation structure. Section A.1 introduces some notation that is used throughout this Appendix. Sections A.2 through A.5 deal with the directory layout, configuration, building, and installation of modules that are located in the LAM source tree, respectively. Modules that are not located in the LAM source tree need not abide by the guidelines outlined in these sections. Finally, section A.6 describes the coding conventions that are required in LAM/MPI components.

#### A.1 Notation

The following terms are used throughout this Appendix:

- `<type>`: This string refers to a specific component framework in LAM/MPI; its values are strictly defined by LAM. Valid values are: `boot`, `rpi`, `coll`, `crlam`, and `crmpi`.
- `<module>`: Each module has a string name to distinguish it from other modules in the same component framework. This string is meant to be replaced by the name of a module.

### A.1.1 The Prefix Rule

It is necessary to prevent symbol name clashes between modules, the LAM and MPI libraries, and user programs. Name clashes can occur on the filesystem, in shell variables, in libraries object filenames, and in library public symbol names. Conflict avoidance is necessary regardless of whether the module is loaded statically or dynamically; some platforms have processes with global symbol namespaces.

Modules must therefore effectively create unique namespaces for their variables, functions, and other public named objects. In C, for example, this is effected by adding a prefix onto every public symbol. While guaranteeing to prevent name clashes with anything else, this has the side effect of making variable and function names rather lengthy. Name/symbol prefixes are typically of the form “lam\_ssi\_<type>\_<module>” or “lam\_ssi-<type>-<module>” (depending on the context), but the letters may be all upper case or all lower case (again, depending on context).

This convention is referred to as “the prefix rule” throughout the rest of this document.

### A.1.2 Function Parameters

When describing function prototypes, parameters are marked in one of three ways:

- **IN:** The parameter is read – but not modified – by the function.
- **OUT:** The parameter, or the element pointed to by the parameter, may be modified by the function.
- **IN/OUT:** The parameter, or the element pointed to by the parameter is read and may be modified by the function.

### A.1.3 Historical Names

Note that there are several type, variable, and function names cited in these appendices that do neither begin with a “lam\_” prefix nor adhere to the prefix rule. These names exist solely for historical reasons; they have existed in LAM/MPI for years and therefore

are used widely throughout the code base. All new names, at a minimum, begin with a “lam\_” prefix.

#### A.1.4 Common LAM Types, Variables, and Functions

The following types are used throughout the rest of the Appendices:

- `OPT`: The `OPT` type is for holding command line parameter and parsing results. Its use and function is described in the `all_opt(3)` man page.
- `LIST`: The `LIST` type is a generic linked list. Its use and function is described in the `all_list(3)` man page.

#### A.2 Directory Layout and Contents

This section assumes that a module is being configured, built, and installed from within the LAM source tree.

Each module is essentially a self-contained directory tree. There are two notable exceptions, however:

1. LAM header files for centralized data structures and SSI constructs are contained in the LAM source tree.
2. The module may choose to use helper macros or other reference code from the LAM source tree.

In general, there is no mandated directory structure in a module’s implementation. This allows module authors to use whatever abstractions, file layouts, and directory structures that are appropriate for their code. However, when the module is part of the LAM source tree and is expected to be configured, built, and installed as part of the LAM configuration and build process, certain filename conventions must be followed.

As previously mentioned in Section [A.1](#), each module has a string name to distinguish it from other modules in the same component framework. This name is used in many places: the filesystem, shell scripts, `Makefiles`, and C code. For example, the module needs to have its top-level directory appear in a specific place in the LAM directory tree:

`share/ssi/<type>/<module>`. The LAM provided `tcp rpi` module is rooted at `share/ssi/rpi/tcp`. Since the name of the module is used in many places, it must adhere to the following requirements:

- It must conform to filesystem requirements for directory names. Some filesystems are not case-sensitive, so all-lowercase names are recommended.
- It must conform to C variable name standards (cannot include whitespace, cannot include punctuation, etc.).
- The names “include” and “base” are reserved for use by the component frameworks and should never be used as module names.

### A.3 Configuring the Module

This section assumes that a module is being configured, built, and installed from within the LAM source tree. Note, however, that a module can be setup while in a LAM source tree (to take advantage of helper scripts, configuration templates, etc.) and then distributed, configured, and compiled outside of the LAM source tree.

Since modules should be designed to run on as many systems as possible, they are configured before they are built or installed. There are two steps of configuration:

1. Generating `configure` scripts and related files (e.g., `Makefile.am` or `Makefile.in` files).
2. Running `configure` scripts.

These steps are detailed below.

#### A.3.1 Generating `configure` Scripts

The LAM source tree and modules included in the LAM/MPI software package all use the GNU Autoconf, Automake, and Libtool tools to generate `configure` scripts. These tools require a non-trivial sequence of steps that must be followed to generate these files. As such, the process is automated. The script `autogen.sh` is used to invoke several preprocessing steps, create templates, and finally run the GNU tools. `autogen.sh` can

perform several different actions; its execution is largely dependent on what configuration files are present in the directory where it is executing (see below). Although it is the preferred method of invoking the GNU tools, the `autogen.sh` script is not *necessary*; it is simply a convenience script to run all the right commands in the proper order.

`autogen.sh` can be invoked in a single directory (either the top-level LAM source directory or a module's top-level directory) or in a tree-traversal mode where it will run in both the LAM top-level directory and all valid module directories that it finds of the form `share/ssi/<type>/<module>`.

In each directory, `autogen.sh` will behave differently depending on what it finds:

- If a file named `.lam_no_gnu` exists in a module's top-level directory, `autogen.sh` will not invoke the standard GNU tools to generate `configure` scripts. This may be desirable for third party modules do not need to have any files generated.
- If a file named `.lam_ignore` exists in a module's top-level directory, `autogen.sh` will skip the entire directory tree. The directory will effectively be ignored for the entire LAM configure, build, and install process.
- If an executable script named `autogen.sh` exists in the module's top-level directory (with no corresponding `.lam_no_gnu` or `.lam_ignore` files), this script will be executed in lieu of running the GNU Auto tools.
- If a file named `configure.params` exists in a module's top-level directory, a template `configure.ac` file is generated based on the parameters found in `configure.params`. See below for more details on the `configure.params` file.
- If a file named `configure.in` or `configure.ac` exists in a module's top-level directory (even as a result of a template `configure.ac` being generated from the `configure.params` file), the full suite of GNU tools will be invoked (Autoconf, Libtool, Automake) to generate a `configure` script and any other associated output files.

#### The `configure.params` File

The `configure.params` file is all that many module authors need to provide for their module. The presence of this file indicates that `autogen.sh` should generate a template `configure` script as described above. This template is usually sufficient for

the majority of setup required by modules, and typically only requires two parameters to be specified.

The behavior of this template can be controlled by parameter settings in the `config-params` file:

- `PARAM_INIT_FILE`: Mandatory (this parameter must be present).  
Default value: none.  
Purpose: This parameter specifies a filename that is passed to the Autoconf `AC_INIT` macro. This file serves as a simplistic sanity check for Autoconf; it checks to see if it can find the file in order to ensure that it is operating in the correct directory. Any file in the module's tree is suitable.
- `PARAM_CONFIG_AUX_DIR`: Optional.  
Default value: if a subdirectory named `config` exists, the value "config" is used. Otherwise, "." is used.  
Purpose: Determines where the supplemental GNU tool scripts will be placed (e.g., `config.guess`).
- `PARAM_WANT_C`: Optional.  
Default value: 1.  
Purpose: Indicate whether any files in the module need to be built with the C compiler or not.
- `PARAM_WANT_CXX`: Optional.  
Default value: 0.  
Purpose: Indicate whether any files in the module need to be built with the C++ compiler or not.
- `PARAM_VERSION_FILE`: Optional.  
Default value: if either the file `VERSION` or the file `$config_dir/VERSION` exist, use it as the version file template. Otherwise, empty.  
Purpose: The template `configure` will analyze the version file to find the module's version number. The format of the version file is described below.
- `PARAM_VAR_PREFIX`: Optional.  
Default value: `LAM_SSI_<type>_<module>`  
Purpose: Changes the default prefix used for all output preprocessor macros from the template `configure` script. It is rarely necessary to change this value.
- `PARAM_AM_NAME`: Optional.  
Default value: `<type>_<module>`  
Purpose: Changes the first argument passed to the Automake `AM_INIT_AUTOMAKE` macro. It is rarely necessary to change this value.

```

# This module only needs to provide the two mandatory parameters; the
# default values for all other parameters are sufficient. Hence,
# the other parameters are not even listed here.

# Provide a file for a sanity check to ensure that we're in the right directory
PARAM_INIT_FILE=src/ssi_rpi_tcp.c

# List the Makefiles that need to be generated
PARAM_CONFIG_FILES='Makefile config/Makefile src/Makefile'

```

Figure A.1. The `configure.params` file for the `tcp_rpi` module.

- `PARAM_CONFIG_HEADER_FILE`: Optional.  
 Default value: `src/lam-ssi-<type>-<module>-config.h`  
 Purpose: Change the name of the header file that is output at the end of `configure`.
- `PARAM_CONFIG_FILES`: Mandatory (this parameter must be present).  
 Default value: None.  
 Purpose: Define a list of files (other than the header file) that `configure` will generate (i.e., the arguments to `AC_CONFIG_FILES`). It is typically a list of Makefiles to generate.

Many modules will only need to provide the two mandatory parameters: `PARAM_INIT_FILE` and `PARAM_CONFIG_FILES`. Figure A.1 shows the `configure.params` file for the `tcp_rpi` module included in the LAM/MPI software package.

### The Template `configure` Script

The template `configure` script generated by `autogen.sh` performs the following actions:

- Determine the module's version number by examining the module's version file. A sample version file is shown in Figure A.2. The following C/C++ preprocessor macros will be defined, based on the values from this file:
  - `_${var_prefix}_VERSION`<sup>1</sup>: All the relevant version values put together in a single output string. Components equaling zero will be left out. If the alpha

<sup>1</sup>The value of `$_var_prefix` is obtained from the `PARAM_VAR_PREFIX` in the `configure.params` file.

```
major=7
minor=1
release=2
alpha=0
beta=0
cvs=1
```

Figure A.2. Sample version file for a module. This file will resolve to the full version string “7.1.2cvs1”.

version is non-zero, it will be prefixed with “a”. If the beta version is non-zero, it will be prefixed with “b”. Finally, if the cvs version is non-zero, it will be prefixed with “cvs”.

- $\{\text{var\_prefix}\}$ \_MAJOR\_VERSION: Integer value containing the major version number.
  - $\{\text{var\_prefix}\}$ \_MINOR\_VERSION: Integer value containing the minor version number.
  - $\{\text{var\_prefix}\}$ \_RELEASE\_VERSION: Integer value containing the release version number.
  - $\{\text{var\_prefix}\}$ \_ALPHA\_VERSION: Integer value containing the alpha version number.
  - $\{\text{var\_prefix}\}$ \_BETA\_VERSION: Integer value containing the beta version number.
  - $\{\text{var\_prefix}\}$ \_CVS\_VERSION: Integer value containing the CVS version number.
- Set the installation directory prefix.
  - Set “make clean” to remove common backup files.
  - Check command line parameters and see if the module was selected to be the default for its type. Defines  $\{\text{var\_prefix}\}$ \_DEFAULT to be 0 or 1.
  - Check command line parameters to see if the module was selected to be built as static or dynamic. Sets the Automake conditional LAM\_BUILD\_LOADABLE\_MODULE with the result.
  - Optionally check for the C compiler, based on the value of the PARAM\_WANT\_C parameter value.
  - Optionally check for the C++ compiler, based on the value of the PARAM\_WANT\_CXX parameter value.

- Optionally run module-provided shell code if a `configure.stub` file is found (see below for details on the `configure.stub` file).
- Setup C/C++ preprocessor flags to find the LAM header files.
- Setup Libtool.
- Output a “`config.h`”-style header file with the results of the configure tests.
- Output all other configuration files (e.g., `Makefiles`).

### The `configure.stub` File

If a module has a file named `configure.stub` in its top-level directory, `autogen.sh` may add two calls to two macros into the template `configure` script:

- If `configure.stub` defines a macro named `SSI_CONFIGURE_STUB`, `autogen.sh` will insert a call to it in the generated `configure` script. This macro allows the module author to insert arbitrary Bourne shell code in the `configure` script. Such code is typically used to look for module-specific resources.

For example, the `gm` (Myrinet) `rpi` module looks for the appropriate header files and libraries to determine if it can be built or not. If `<gm.h>` cannot be found, there is no point in trying to build the `gm` module, and the `gm` module’s `SSI_CONFIGURE_STUB` macro will abort. Note that this macro is not invoked if “`--enable-dist`” is specified on the command line.

- If `configure.stub` defines a macro named `SSI_CONFIGURE_STUB_DIST`, `autogen.sh` will insert a call to it in the generated `configure` script. Note that it will only be invoked when “`--enable-dist`” is used on the `configure` command line. This option is used when building a distribution package, and the `configure` *must* succeed (even if, for example, `<gm.h>` cannot be found). See Section [A.3.2](#) for more details.

### A.3.2 Running `configure` Scripts

If a module does not have an executable named `configure`, it will not be configured or built by LAM’s `configure` and build system. Alternatively, if there is a file named `.lam_ignore` in the module’s top-level directory, LAM will ignore that module, even if a corresponding `configure` script exists.

Each module’s `configure` executable will automatically be invoked by LAM’s top-level `configure` script. All the same command line flags and environment variables

that were used to invoke the top-level `configure` script will be passed down to the module's `configure` script.

### Module-Specific `configure` Parameters

It is possible to have module-specific `--with` and/or `--enable` command line switches that enable configure-time parameters (perhaps using the Autoconf-provided `AC_ARG_WITH` or `AC_ARG_ENABLE` macros). In order for these switches to be transparently passed through other `configure` scripts, a standard prefix naming convention must be used to prevent command line parameter collisions. All module parameters must therefore be prefixed with “`ssi-<type>-<module>-`” (this is consistent with the prefix rule described in Section [A.1.1](#)).

Similarly, it is possible to have module-specific environment variables that are used to pass values to the module's `configure` script. These variables should also adhere to the prefix rule described above; environment variables should be of the form “`SSI-<type>-<module>-`”.

### Return Status From `configure`

The exit status of a module's `configure` script determines whether the module will be compiled or ignored by LAM. A zero exit status means that the `configure` script was successful and can be built on this platform. A non-zero exit status means that the `configure` script failed and/or the module cannot be built for some reason.

LAM will only build a module if its `configure` script returns a zero value.

### Returning Flags From `configure`

When linking a module statically into the LAM or MPI libraries, some modules need to return additional flags to the top-level LAM infrastructure to enable LAM and MPI applications to link properly. This occurs in three distinct cases:

1. Linking LAM applications when building LAM/MPI.
2. Linking MPI applications when building LAM/MPI.
3. Linking user MPI applications.

Each of these three cases are linked differently, and therefore need to be addressed individually. For example, an `rpi` module that uses an underlying communications library named `libfoo.a` affects all MPI applications. Hence, “`-L/path/to/libfoo-libfoo`” must be added to the link command for all LAM/MPI applications that link to the MPI library by propagating these flags to the appropriate locations in the LAM/MPI build process. Additionally, the same flags must be added to all user MPI applications. The “wrapper” MPI compilers (`mpicc`, `mpic++`, and `mpif77`) therefore need to be notified of these flags so that they can add them to the command line when users link MPI applications.

A module can return compiler and linker flags by creating a file named `ssi_<type>_<module>_config.sh` in its top-level directory. This file contains Bourne shell variable assignments for the variables listed in Tables [A.1](#), [A.2](#), and [A.3](#). The Bourne shell variables are of the form `<scope>_EXTRA_<flags>`, where `<scope>` can be any of the following values:

- **LIBLAM:** Compiler flags required for all other compilation units that will end up in the LAM library, or linker flags required when linking to the LAM library to create executables within the LAM/MPI software package.
- **LIBMPI:** Compiler flags required for all other compilation units that will end up in the MPI library, or linker flags required when linking to the MPI library to create executables within the LAM/MPI software package.
- **WRAPPER:** Compiler flags required for user compilation units, or linker flags required when linking user MPI executables.

`<flags>` can be any of the following values:

- **CFLAGS:** Any flags that need to be passed to the C compiler in the given scope. This may be compiler warning flags, debugging flags, optimization flags, etc. Note that this specifically does *not* include “`-I`” and “`-D`” arguments. Such arguments are specific to the module and need not be propagated to the rest of the LAM/MPI source tree or user MPI applications.

- CXXFLAGS: Just like CFLAGS, but will be passed to the C++ compiler.<sup>2</sup>
- FFLAGS: Just like CFLAGS, but will be passed to the Fortran compiler.
- LDFLAGS: Typically “-L” flags that need to be passed to the linker to link to a library containing this module.
- LIBS: Typically “-l” flags that need to be passed to the linker to link to a library containing this module.

Note that GNU Libtool automatically takes care of propagating any required LD-FLAGS and LIBS arguments when building the rest of LAM/MPI. Specifically, if Libtool is used to build a module, the module’s `configure` script should simply set appropriate LDFLAGS and LIBS values that can be used to link a final executable. Libtool will then automatically propagate these flags to any executable in the LAM/MPI source tree that needs them. Hence, setting `LIB*_EXTRA_LDFLAGS` and `LIB*_EXTRA_LIBS` in the generated `config.sh` file is frequently not necessary; the `WRAPPERS_EXTRA_*` variables are typically the only values that need to be returned from `configure`.

It is *never* necessary for a module to pass “-I” and “-D” flags back to the upper-level LAM building/configuration environment. This is why Tables [A.1](#), [A.2](#), and [A.3](#) do not have variables for `CPPFLAGS`. Module-specific header files, by definition, will only be needed to compile the module. They will not be needed by the rest of LAM or user MPI programs.

#### When the `configure` Script *Must* Succeed

When building a distribution package of LAM/MPI, it is desirable to include *all* modules, regardless of whether they can configure successfully or not. In this case, modules that normally only allow themselves to be configured successfully when certain conditions are met (e.g., a module that only builds when specific third party libraries and

---

<sup>2</sup>Note that some top-level executables in LAM are written in C++, so if a module sets `CFLAGS`, it should also set `CXXFLAGS`.

TABLE A.1

Available return variables from component module `configure` scripts for building and linking to the LAM library (in addition to what is already propagated by Libtool)

<b>Name</b>	<b>Description</b>
<code>LIBLAM_EXTRA_CFLAGS</code>	Extra flags to be passed to the C compiler
<code>LIBLAM_EXTRA_CXXFLAGS</code>	Extra flags to be passed to the C++ compiler
<code>LIBLAM_EXTRA_FFLAGS</code>	Extra flags to be passed to the Fortran compiler (not currently used anywhere)
<code>LIBLAM_EXTRA_LDFLAGS</code>	Extra “-L” arguments to be passed to the linker
<code>LIBLAM_EXTRA_LIBS</code>	Extra “-l” arguments to be passed to the linker

TABLE A.2

Table of available return variables from component module `configure` scripts for building and linking to the MPI library (in addition to what is already propagated by Libtool)

<b>Name</b>	<b>Description</b>
<code>LIBMPI_EXTRA_CFLAGS</code>	Extra flags to be passed to the C compiler
<code>LIBMPI_EXTRA_CXXFLAGS</code>	Extra flags to be passed to the C++ compiler
<code>LIBMPI_EXTRA_FFLAGS</code>	Extra flags to be passed to the Fortran compiler (not currently used anywhere)
<code>LIBMPI_EXTRA_LDFLAGS</code>	Extra “-L” arguments to be passed to the linker
<code>LIBMPI_EXTRA_LIBS</code>	Extra “-l” arguments to be passed to the linker

TABLE A.3

Table of available return variables from component module `configure` scripts for building and linking user MPI applications

<b>Name</b>	<b>Description</b>
WRAPPER_EXTRA_CFLAGS	Extra flags to be passed through the wrapper compilers to the C compiler
WRAPPER_EXTRA_CXXFLAGS	Extra flags to be passed through the wrapper compilers to the C++ compiler
WRAPPER_EXTRA_FFLAGS	Extra flags to be passed through the wrapper compilers to the Fortran compiler
WRAPPER_EXTRA_LDFLAGS	Extra “-L” arguments to be passed through the wrapper compilers to the linker
WRAPPER_EXTRA_LIBS	Extra “-l” arguments to be passed through the wrapper compilers to the linker

header files are present) should bypass these checks and allow themselves to be configured [pseudo-]successfully. Specifically, the module's `configure` scripts must return an exit status of zero and produce `Makefiles` that, at a bare minimum, have working “`dist`” targets.

LAM requires this behavior when the `--enable-dist` switch is used. Specifically: if LAM (and all of the modules in its source tree) is configured with this switch, all modules must produce a top-level `Makefile` with a working “`dist`” target. Note that no other targets are required to be functional. Hence, all conditional tests can be skipped such that valid `Makefiles` can be generated, even if the module cannot actually be built.

Modules that use the templated `configure` script can define a `SSI_CONFIGURE_-DIST_STUB` macro in the `configure.stub` file that will be automatically invoked when the `--enable-dist` option is used.

Modules that are not built conditionally can ignore this entire section since they always produce fully functional `Makefiles` that include a valid `dist` target.

#### A.4 Building the Module

This section assumes that a module is being configured, built, and installed from within the LAM source tree, and that the module has previously successfully ran its `configure` executable. Modules that failed the configuration phase will not be built or installed.

The top-level LAM building process assumes the presence of the following make targets in the module's directory:

- `all`: build the module. The internals of this target can function however it wants, but at the end of this target, a GNU Libtool library named `liblam_ssi_<type>_<module>.la` must exist in the top-level module library.
- `install`: install the module. See Section [A.5](#) for more details on what happens during installation.
- `uninstall`: uninstall the module. As with the `install` target, this may be a no-op for static libraries.

- `clean`: remove all files generated by the `all` target.
- `dist`: make a distribution package, per the GNU guidelines.
- `distclean`: do everything that the `clean` target does, and additionally remove all other generated files such that the directory tree is the same state that it was in when it was expanded from its distribution package.
- `tags`: use the `etags` program to generate lists of tags.

## A.5 Installing the Module

This section assumes that a module is being configured, built, and installed from within the LAM source tree, and that the module has previously both successfully ran its `configure` executable and been built. Two general kinds of files need to be installed after building a module: the library (or libraries) containing the module code and any additional data / support files.

Library installation requirements depend on whether the module was built statically or dynamically. Statically-built modules need not be installed because the resulting library will be folded into the LAM or MPI library (depending on the module's type). Dynamically-build modules must install the module's shared library into the correct directory where the SSI can find them at run-time: `$prefix/lib/lam`.

If using the templated `configure` file, the decision whether to build the module statically or dynamically is controlled by the Automake conditional `LAM_BUILD_LOADABLE_MODULE`. A sample `Makefile.am` snippet for the `tcp rpi` module is shown in Figure A.3.

A module may also install additional data and support files. These must be installed under the `$sysconf` directory (typically `$prefix/etc`).

## A.6 Module Source Code

Although it is permissible to use any language to implement a module, the top-level interface calls must be able to be resolved with C linkage. Using C++ in the internals

```

# Examine the Automake conditional and decide whether we are building
# the module statically or dynamically

if LAM_BUILD_LOADABLE_MODULE
module_noinst =
module_install = ssi_rpi_tcp.la
else
module_noinst = liblam_ssi_rpi_tcp.la
module_install =
endif

# -----
# If building dynamically, $(module_install) will be non-empty, and
# we'll use this set of Automake macros to build the module.

lamssiexecdir = $(libdir)/lam
lamssiexec_LTLIBRARIES = $(module_install)
ssi_rpi_tcp_la_SOURCES =
# Note that the TCP module source code is in src/libtcp.la. Note also
# that since this is a standalone entity, we must also link to libmpi
# and liblam to resolve all the symbols used in this module.
ssi_rpi_tcp_la_LIBADD = src/libtcp.la \
                        $(top_lam_builddir)/share/libmpi/libmpi.la \
                        $(top_lam_builddir)/share/liblam/liblam.la
ssi_rpi_tcp_la_LDFLAGS = -module -avoid-version

# -----
# If building statically, $(module_noinst) will be non-empty, and
# we'll use this set of Automake macros to build the module. Note
# that LTLIBRARIES *must* be marked "noinst" or Automake will not fold
# this library into the upper level MPI library.

noinst_LTLIBRARIES = $(module_noinst)
liblam_ssi_rpi_tcp_la_SOURCES =
liblam_ssi_rpi_tcp_la_LIBADD = src/libtcp.la
liblam_ssi_rpi_tcp_la_LDFLAGS = -module -avoid-version

```

Figure A.3. Sample top-level `Makefile.am` for building the `tcp rpi` module. Logic based on the `LAM_BUILD_LOADABLE_MODULE` decides whether to build the module statically or dynamically.

```
AM_CPPFLAGS = -DLAM_BUILDING=1 \  
              -I$(top_lam_builddir)/share/include \  
              -I$(top_lam_srcdir)/share/include
```

Figure A.4. Sample `AM_CPPFLAGS` required to find the LAM header files. Note that the build directory is explicitly included in order to support `VPATH` builds properly, even though it will be redundant in non-`VPATH` builds.

of the module is discouraged as building C++ libraries may not be portable. Although recent versions of GNU Libtool now support building C++ libraries with a wide variety of compilers, compatibility problems have arisen in some systems where multiple C++ compilers lay out structures and classes in memory differently, creating problems sharing data structures across object files generated by different compilers.

### A.6.1 Header Files

The top-level SSI include file is (relative to the LAM source tree root) `share/-include/lam-ssi.h`. This file declares some top-level SSI functions and the C type `lam_ssi_t` (a structure defined in Section A.6.2, below).

If the module is built using Automake, the appropriate preprocessor flags can be included by adding an appropriate `-I` flag to Automake's `AM_CPPFLAGS` macro. Additionally, the preprocessor symbol `LAM_BUILDING` *must* be defined and set to the value of 1 if the module is being built statically as part of the LAM or MPI libraries. Figure A.4 shows an example.

Each component framework has its own top-level include file (located in the same directory as `lam-ssi.h`) named `lam-ssi-<type>.h`. Each SSI type's header file will define the datatype `lam_ssi-<type>t` which will be used in all modules of that type.

## A.6.2 The Base Module Datatype: `lam_ssi_t`

Every module must export a public symbol named `lam_ssi_<type>_<module>_<module>` of type `lam_ssi_<type>_t`. While each component framework is responsible for defining exactly what that type is, each contain an instance of type `lam_ssi_t` as its first element. The `lam_ssi_t` type is used to store basic information about a module, and is defined in Figure [A.5](#).

The members of `lam_ssi_t` are:

- The first group of three version fields (`ssi*_version`) refers to the version of SSI that the module conforms to. In this case, they should be hard-coded to 1, 0, and 0, respectively. This is for version control purposes, and is explained in greater detail below.
- The `ssi_kind_name` field identifies the component framework that this module belongs to. Its values are strictly defined by LAM, and are the same as the values allowed for `<type>`.
- The second group of three version fields (`ssi_type*_version`) refers to the API version of the component framework. This is also for version control purposes, and is explained below.
- The `ssi_module_name` field identifies the name of this module. It must agree with the name used in the prefix rule throughout the rest of the module.
- The third group of three version fields (`ssi_module*_version`) refers to the version of the module itself. The contents of these fields are left up to the module.
- The open and close function pointers point to functions as described in Sections [A.6.3](#) and [A.6.4](#), respectively. NULL values may be supplied for these pointers if the module does not have open and/or close functions. Note, however, that a NULL value for the open function implies that the module is always available (i.e., it is as if the open function *did* exist, and returned a success status when invoked).

The base SSI API is currently at version 1.0.0. Although future versions of SSI may alter the `lam_ssi_t` type, the first three `int` values in the struct will always be the SSI API version number. For example, if the layout of `lam_ssi_t` changes in some future SSI API version, the LAM SSI framework will be able to apply the right type to a given module's `lam_ssi_t` by examining the first three `int` values. This enables backward compatibility to be preserved for modules that do not keep up with the newest versions of the SSI API.

```

typedef struct lam_ssi_1_0_0 {

    /* Integer version numbers indicating which SSI API version
       this module conforms to. */

    int ssi_major_version;
    int ssi_minor_version;
    int ssi_release_version;

    /* Information about the type and the version of the type's API that it conforms
       to ('kind' is a historical nomenclature) */

    char ssi_kind_name[LAM_MPI_SSI_BASE_MAX_KIND_NAME_LEN];
    int ssi_kind_major_version;
    int ssi_kind_minor_version;
    int ssi_kind_release_version;

    /* Information about the module itself */

    char ssi_module_name[LAM_MPI_SSI_BASE_MAX_MODULE_NAME_LEN];
    int ssi_module_major_version;
    int ssi_module_minor_version;
    int ssi_module_release_version;

    /* Functions for opening and closing the module */

    lam_ssi_open_module_fn_t ssi_open_module;
    lam_ssi_close_module_fn_t ssi_close_module;
} lam_ssi_1_0_0_t;

/* Set the default type to use version 1.1.0 of the SSI struct */

typedef lam_ssi_1_0_0_t lam_ssi_t;

```

Figure A.5. Definition of the `lam_ssi_t` type.

### A.6.3 Module Open Function

A module may have an *open* function. If it exists, it is invoked exactly once during the life of a process after the module is found (if the module was statically linked) or loaded into the process (if the module was dynamically linked). The open function is used to register module parameters, allocate any one-time, module-specific resources, and make the first determination whether it is possible for the module to run or not. This function is particularly relevant for component frameworks that have multiple scopes in a single process. Modules in such frameworks may need to differentiate between once-per-process and once-per-scope initialization.

Since a pointer to the module's open function is contained in the `lam_ssi_t` instance, it is common to all component frameworks. The function pointer type is defined as follows:

```
typedef int (*lam_ssi_open_module_fn_t)(OPT *args);
```

A typical module open function may be prototyped as follows:

```
int lam_ssi_<type>_<module>_open_module(OPT *args);
```

Although an OPT handle representing the command line that invoked the current executable is passed as an argument, it is expected that most user-specified values and selection criteria will be passed through the module parameters system (described in Section A.6.6). As such, the open function is responsible for registering all parameters (and associated default values) that will be used by the module. It is also worth noting that `laminfo` open opens and closes all modules (it does not invoke any other module functions). Hence, from `laminfo`'s perspective, the open function is the only opportunity for a module to register parameters.

The open function may also check the current run-time environment and determine if it is able to run. This may entail allocating resources and/or checking environmental or

other external factors. If the module determines that it is able to be run, it should return a successful status (and therefore be considered available). If not, it should return a failure status and both the SSI and the component framework will close and ignore the module for the remainder of the process.

If an open function is not provided by the module, it is implied that the open function returned a successful status. The module is therefore considered available.

#### A.6.4 Module Close Function

A module may have a *close* function. If it exists, it is invoked exactly once during the life of the process and is always the last function that is invoked on the module. The main purpose of the close function is to free any resources allocated by the module.

Since a pointer to the module's close function is contained in the `lam_ssi_t` instance, it is common to all component frameworks. The function pointer type is defined as follows:

```
typedef int (*lam_ssi_close_module_fn_t)(void);
```

A typical module close function is prototyped as follows:

```
int lam_ssi_<type>_<module>_close_module(void);
```

The close function will be invoked for every module that either had its open function invoked or had a `NULL` open module function pointer. The timing of when the close function is invoked varies depending on the component framework. For example, the `boot` component framework has only one scope per process; unselected modules will be closed during process initialization. But the `coll` component framework potentially has many scopes in a single process; available `coll` modules will not be closed until `MPI_FINALIZE`.

```

typedef struct lam_ssi_rpi_1_1_0 {
    lam_ssi_1_0_0_t lsr_meta_info;

    /* ...various other members specific to the rpi component framework... */
} lam_ssi_rpi_1_1_0_t;

typedef lam_ssi_rpi_1_1_0_t lam_ssi_rpi_t;

```

Figure A.6. Definition of the `lam_ssi_rpi_t` type.

### A.6.5 Example Usage: The `tcp_rpi` Module

This section provides a module example based on the `tcp_rpi` module included in the LAM/MPI software package. The `rpi` component framework defines a type named `lam_ssi_rpi_1_1_0_t` that contains module interface function pointers. Its first element will always be some version of `lam_ssi_t`, as shown in Figure A.6.

Note that the same version technique is used with the `rpi` component type as with the base SSI API – the `rpi` API show here is version 1.1.0. Future versions may change the layout of `lam_ssi_rpi_t`, but since the first member of it is guaranteed to be some version of `lam_ssi_t`, both the SSI API version and `rpi` API version numbers can be guaranteed to be successfully extracted.

To continue the example, the `tcp_rpi` module exports a global symbol named `lam_ssi_rpi_tcp_module` as shown in Figure A.7.

### A.6.6 Module Parameters

As described in Section 3.2.5, parameters can be passed to modules at run-time by the command line or through the environment. The SSI provides all the necessary registration, bookkeeping, marshaling, and propagation of module parameters to remote processes. A module typically registers module parameters during its open call. The first two functions in Figure A.8 are used by modules to register parameters. Both functions

```

const lam_ssi_rpi_1_1_0_t lam_ssi_rpi_tcp_module = {

    /* First, the lam_ssi_1_0_0_t struct containing meta information
       about the module itself */

    {
        /* SSI API version */

        1, 0, 0,

        /* Module type name and version */

        "rpi",
        1, 1, 0,

        /* Module name and version -- obtained and defined by
           the tcp rpi configure script */

        "tcp",
        LAM_SSI_RPI_TCP_MAJOR_VERSION,
        LAM_SSI_RPI_TCP_MINOR_VERSION,
        LAM_SSI_RPI_TCP_RELEASE_VERSION,

        /* Module open and close function pointers */

        lam_ssi_rpi_tcp_open,
        NULL
    },

    /* ...various other members specific to the rpi component framework... */
};

```

Figure A.7. Definition of the `lam_ssi_rpi_t` type for the `tcp` `rpi` module. Note the `NULL` used for the close function; the `tcp` module does not have a close function.

```

/* Module parameter registration functions */

int lam_ssi_base_param_register_int(char *type_name, char *module_name,
    char *param_name, char *ssi_param_name, int default_value);
int lam_ssi_base_param_register_string(char *type_name, char *module_name,
    char *param_name, char *ssi_param_name, char *default_value);

/* Module parameter lookup functions */

int lam_ssi_base_param_lookup_int(int index);
char *lam_ssi_base_param_lookup_string(int index);

```

Figure A.8. Functions for registering and looking up module parameters.

are similar; one registers an integer parameter, the other registers a string parameter.

The parameters that both functions expect are:

- `type_name`: String name of the component framework for this module.
- `module_name`: String name of this module.
- `param_name`: String name of this parameter.
- `ssi_param_name`: If `NULL`, the default name of the parameter will be used: “`ssi_<type>_<module>_<param>`”. If this parameter is not `NULL`, it will be used as the parameter name instead of the default. `NULL` is the recommended value for this parameter.
- `default_value`: The default value for this parameter.

The return value from the registration functions is an integer handle for the registered parameter. This handle can be used to lookup parameter values through the `lam_ssi_base_param_lookup_int()` and `lam_ssi_base_param_lookup_string()` functions (also shown in Figure A.8).

## APPENDIX B

### PARALLEL JOB STARTUP COMPONENT INTERFACE

The `boot` component framework is described in Chapter 4. This Appendix describes the technical details and requirements for `boot` modules [123]. Section B.1 discusses header files, types, global variables, and utility functions that are provided to all `boot` modules. Section B.2 details the module interface modules and functions.

#### B.1 Services Provided by the `boot` Component Framework

Several services are provided by the `boot` component framework that are available to all `boot` modules.

##### B.1.1 Header Files

The following header files must be included (in order) in all module source files that want to use any of the common `boot` component framework services described in this Appendix:

```
#include <lam-ssi.h>
#include <lam-ssi-boot.h>
```

Both of these files are included in the same location in the LAM source tree: `share/-include`. Section A.6.1 describes how to provide appropriate preprocessor flags to include these files properly.

```

struct lamnode {
    int4 lnd_nodeid;
    int4 lnd_type;
    int4 lnd_ncpus;
    int lnd_bootport;
    char *lnd_hname;
    char *lnd_uname;
    struct sockaddr_in lnd_addr;
    LIST *lnd_keyval;
    struct lnd_ssi_boot_nodeinfo *lnd_ssi;
};

```

Figure B.1. `struct lamnode` definition.

### B.1.2 Internal Type: `struct lamnode`

This type is used to describe nodes in the LAM universe. It is prototyped in `<lam-net.h>` (see Figure B.1).

The individual elements are:

- `lnd_nodeid`: A unique integer identifying a node, from 0 to  $(N - 1)$ , where  $N$  is the total number of nodes in the universe.
- `lnd_type`: A set of bit flags indicating attributes about that node. The most important flags to `BOOT` modules are:
  - `NT_BOOT`: Indicates that a node is supposed to be booted.
  - `NT_ME`: Indicates that this node is the local node.
  - `NT_ORIGIN`: Indicates that this node is the origin node.
  - `NT_WASTE`: Indicates that this node should not be used for default scheduling by `mpirun` and `lamexec`. For example, nodes marked with this attribute will not be used for “`mpirun C a.out`”.
- `lnd_ncpus`: Number of CPUs on that node.
- `lnd_bootport`: TCP port number used in the startup rendezvous protocols.
- `lnd_hname`: String name for the node, usually parsed from the boot schema file.
- `lnd_uname`: String username to be used to login on the remote node, or `NULL` if unnecessary.
- `lnd_addr`: Binary representation of the TCP address of the node.

```
# Sample boot schema file with several "key=value" examples
inky.cluster.example.com cpu=2
pinky.cluster.example.com cpu=4
blinky.cluster.example.com cpu=4
clyde.cluster.example.com cpu=2 user=jsmith
```

Figure B.2. Example boot schema file.

- `lnd_keyval`: List of key-value pairs parsed from the `boot` schema file. These key-value pairs provide an extensible method to obtain module-specific information from the boot schema file. The `bhostparse()` utility function is typically used to parse boot schema files (see Section B.1.7), and will fill the `lnd_keyval` list with every “key=value” pair found in the boot schema. Figure B.2 shows an example boot schema file.

Each `struct lamnode` instance will have its `lnd_keyval` filled with a list of the “key=value” pairs from the boot schema listed above. The `struct lamnode` instance for `clyde` will have two entries which each of the others will have one. All keys and values are represented as strings.

Although all “key=value” pairs will be parsed by `bhostparse()` and placed in the `lnd_keyval` list, commonly used keys include:

- `hostname=<host>`: Specifies the target node’s name or IP address. The first token on each line in the boot schema file is implicitly the hostname.
  - `cpu=<NUM>`: Specifies the number of CPUs that LAM may use on the target node.
  - `user=<username>`: Specifies the login name which can be used to remotely login to the node (if different than the username of process owner).
  - `prefix=<path>`: Specifies the path where LAM binaries are installed on the target node.
  - `schedule=(yes|no)`: Specifies whether this node needs to be scheduled for running jobs or not.
- `lnd_ssi`: “Extra” information that each `boot` module can define. Each module must provide its own definition for the type `lnd_ssi_boot_nodeinfo`.

### B.1.3 Internal Type: `struct psc`

This enum is returned in a `LIST` by the `hbootparse()` function (see Section B.1.8). It contains a list of `argv`-style arrays of processes to start on a target node. See Figure B.3.

```
struct psc {
    char **psc_argv;
    int4 psc_argc;
    int4 psc_delay;
    int4 psc_flags;
};
```

Figure B.3. `struct psc` definition.

```
typedef enum {
    LAM_SSI_BOOT_PROC_LAMD,
    LAM_SSI_BOOT_PROC_RECON,
    LAM_SSI_BOOT_PROC_WIPE,
    LAM_SSI_BOOT_PROC_MAX
} lam_ssi_boot_proc_t;
```

Figure B.4. `lam_ssi_boot_proc_t` enumerated type definition.

The members are:

- `psc_argv`: NULL-terminated array of command line tokens to start on the target node.
- `psc_argc`: Length of `psc_argv`.
- `psc_delay`: Delay this many seconds after starting.
- `psc_flags`: Currently unused; reserved for future expansion.

#### B.1.4 Internal Type: `lam_ssi_boot_proc_t`

This type is used as an argument to boot API functions, indicating which LAM runtime environment process to start. Figure B.4 shows the allowable values.

#### B.1.5 Global Variable: `int lam_ssi_boot_base_server_port`

This `int` is defined and set by the TCP startup rendezvous protocol functions, and is described in Section B.1.18.

### B.1.6 Global Variable: `int lam_ssi_boot_optd`

The `lam_ssi_boot_opt` variable is of type `OPT*`. It contains the parsed arguments from the command line.

Most `boot` modules will not need this variable. One of the API functions (described in Section [B.2.4](#)) receives the `OPT*` variable as a parameter. If later API functions require information from it, the module can save a local copy that can be shared throughout the module. The main purpose of this variable is for utility routines provided by the `boot` component framework that can be used as API functions, but are not part of individual modules.

### B.1.7 Utility Function: `bhostparse()`

```
#include <boot.h>
int bhostparse(char *filename, struct lamnode **nodes, int *nnodes);
```

Arguments:

- IN: `filename` is the name of the file to parse.
- OUT: `lamnodes` is a pointer to an array that will be filled.
- OUT: `nnodes` is a pointer to an `int` that will be filled with the size of the `lamnodes` array.

Parses a boot schema and returns an array of `struct lamnode` instances. `filename` is the filename of the file to parse. `nnodes` will be allocated and filled by this function (it is the caller's responsibility to free the `nodes` array later); `nnodes` is set to the length of `nnodes`.

`bhostparse()` will parse all key-value pairs and place them in the `lnd_keyval` member on the corresponding entries in the `nodes` array. `bhostparse()` also sets/clears `NT_WASTE` depending upon whether the key-value pair `schedule=yes/no` has been defined. See section [B.1.2](#) for information on `NT_WASTE`.

This function is typically invoked in the `allocate_nodes` API call (see Section [B.2.5](#)).

### B.1.8 Utility Function: `hbootparse()`

```
#include <boot.h>

int hbootparse(int debug, OPT *ad, char *inet_topo, char *rtr_topo, LIST **proc_list);
```

Arguments:

- **IN:** `debug` is a flag indicating whether to display debugging information or not.
- **IN:** `ad` is a handler referring to the parsed command line.
- **IN:** `inet_topo` is a string used to set the `$inet_topo` variable in the LAM run-time environment configuration
- **IN:** `rtr_topo` is a string used to set the `$rtr_topo` variable in the LAM run-time environment configuration
- **OUT:** `proc_list` is a `LIST` of processes to start

Find the LAM run-time environment configuration file in the command line parameters and parse it, doing variable substitution (if `inet_topo` or `rtr_topo` are non-NULL), and return a `LIST` of `argv` arrays containing LAM run-time environment processes to start on remote nodes. The default LAM run-time environment configuration file specifies only a single `lamd` (with some associated command line parameters). Other configurations are also possible (such as starting multiple processes on the target node).

The returned `LIST` contains a list of `struct psc` instances. This list can be processed by the `boot` module to actually start the specified process(es) on the target node(s).

If the `boot` module uses one of the built-in booting algorithms, this function is typically invoked by the `start_rte_proc` internal API function (see Section [B.2.11](#)) in conjunction with the `lam_ssi_boot_inet_topo()` function when starting the `lamd` run-time environment process. Figure [B.5](#) shows an example routine (with many details omitted).

### B.1.9 Utility Function: `lam_deallocate_nodes()`

```

int lam_ssi_boot_<module>_start_rte_proc(struct lamnode *node,
    lam_ssi_boot_proc_t which) {
    if (which == LAM_SSI_BOOT_PROC_LAMD) {
        struct psc *p;
        char *inet_buf;
        LIST *bootlist;

        /* Parse the run-time environment configuration file */

        inet_buf = lam_ssi_boot_build_inet_topo(node, origin_lamnode, origin_id)
        hbootparse(lam_ssi_boot_did, lam_ssi_boot_optd, inet_buf, NULL, &bootlist);

        /* Traverse the parsed run-time environment configuration file data
           structures */

        for (p = al_top(bootlist); p != NULL; p = al_next(bootlist, p)) {

            /* Must not modify the contents of the list items; make duplicates to work
               with (see the sfh_argv(3) man page for more details) */

            av_cmd = sfh_argv_dup(p->psc_argv);
            ac_cmd = p->psc_argc;

            /* ... launch an av_cmd command line on node ... */

            /* Free the duplicate argv array */

            sfh_argv_free(av_cmd);
        }
        free(inet_buf);
        al_free(bootlist);
    }
    /* ... handle other "which" values ... */
}

```

Figure B.5. Abbreviated sample `start_rte_proc()` function.

```
#include <boot.h>
```

```
int lam_deallocate_nodes(struct lamnode **nodes, int *nnodes);
```

Arguments:

- IN/OUT: `nodes` is the array to be freed.
- IN/OUT: `nnodes` is the size of the `nodes` array.

Utility function to deallocate an array of `struct lamnodes`. It is *not* sufficient to simply `free()` the `nodes` array; this function is provided because each item in the array may contain additional memory that must be specifically freed (e.g., the key=value pairs). Upon return from this function, `*nodes` will be set to `NULL`, and `*nnodes` will be set to 0.

This function is typically invoked from the `deallocate_nodes()` API function (see Section [B.2.9](#)).

#### B.1.10 Utility Function: `lam_ssi_boot_base_find_boot_schema()`

```
char *lam_ssi_boot_base_find_boot_schema(OPT *args);
```

Arguments:

- IN: `args` is a handle representing the parsed command line.

Analyzes the command line to find the boot schema filename, then check to see if that file exists.

The parameter `args` is not modified by this function. The return value will be a string representing the filename of the found boot schema, or `NULL` if nothing was found (indicating an error). If nothing is found, an appropriate error message will be printed.

If an absolute pathname is found, it is used. If a relative pathname is found, it is checked against the present directory, the `$TROLLIUSHOME/etc` directory, the `$LAM-HOME/etc` directory, and finally the LAM `$sysconf` directory (selected at configura-

tion time). The `$TROLLIUSHOME` and `$LAMHOME` environment variables are historical names.

This function is typically invoked in the `parse_options()` API call (see Section [B.2.4](#)).

#### B.1.11 Utility Function: `lam_ssi_boot_base_find_hostname()`

```
char *lam_ssi_boot_base_find_hostname(OPT *args);
```

Arguments:

- **IN:** `args` is a handle representing the parsed command line.

This function is used when the `lamgrow` command was used to invoke the `boot` module. This is because the `lamgrow` command does not receive a boot schema; instead, a single hostname is passed on the command line for growing the current LAM universe. This function analyzes `argc` and `argv` to find a string hostname or IP address and verify that it exists.

The parameter `args` is not modified by this function. The return value will be a string representing the found hostname/address or `NULL` if nothing was found (indicating an error). If nothing is found, an appropriate error message will be printed.

This function is typically invoked in the `parse_options()` API call (see Section [B.2.4](#)).

#### B.1.12 Utility Function: `lam_ssi_boot_base_lamgrow()`

```
char *lam_ssi_boot_base_lamgrow(char *hostname, struct lamnode **nodes,  
int *nnodes, int *origin);
```

Arguments:

- **IN:** `hostname` is the name of the host to add.
- **OUT:** `nodes` is the output array of nodes in the current universe plus the new node.

- OUT: `nnodes` is a pointer to an integer indicating the size of the `nodes` array.
- OUT: `origin` is a pointer to an integer indicating the index in the `nodes` array that is the origin nodes.

This function is used when the `lamgrow` command was used to invoke the `boot` component framework. This is because the `lamgrow` command does not receive a boot schema file; instead, a single hostname is passed on the command line for growing the current LAM universe. This function analyzes the current LAM universe and generates an array of `struct lamnode` instances based on its contents, to include an instance for the new node to be booted. Only the entry for the new node will have the `NT_BOOT` flag set.

`hostname` is the string host name or IP address of the node to be booted. `nodes` is allocated and filled by this function (it is the caller's responsibility to free the `nodes` array later); `nnodes` is set to the length of `nodes`. The `origin` argument is filled with the origin node's ID. Note that it is not necessarily the same node as the node that is invoking `lamgrow`.

Depending on how `lamgrow` was invoked, it is possible that the array of `struct lamnode` instances may contain entries for "invalid" nodes (see the `lamgrow(1)` man page for more details). Such entries will have a node ID of `NOTNODEID`, and all their other data will be invalid. Although these nodes must be skipped by the booting algorithms (all the provided algorithms properly skip them), space must be allocated for them in all internal arrays and tables.

This function is typically invoked in the `allocate_nodes()` API call (see Section [B.2.5](#)).

#### B.1.13 Utility Function: `lam_ssi_boot_base_ioexecvp()`

```
int lam_ssi_boot_base_ioexecvp(char **cmdv, int showout, char *outbuff,
    int outbuffsize);
```

---

Arguments:

- IN: `cmdv` is a NULL-terminated `argv`-style array of the command to launch.
- IN: `showout` is a flag indicating whether to display the output on the parent's standard output or not.
- IN: `outbuf` is either NULL or a pointer to a buffer where the standard output from the child will be stored.
- IN: If `outbuf` is non-NULL, `outbuffsize` is the length of the `outbuff` buffer.

This function is used to execute a command. It is typically used to execute a command that starts LAM daemons or starts a proxy process which then starts LAM daemons.

`cmdv` contains the command to be executed. The function can direct the new process's standard output to a buffer and/or the parent process's standard output. The `showout` parameter is used to control this. If the child process's standard output is to be directed to an output buffer, then `outbuff` should point to the buffer for the data and `outbuffsize` should contain maximum size of this buffer.

This function is typically indirectly invoked in the `start_application` API call (see Section [B.2.10](#)).

B.1.14 Utility Function: `lam_ssi_boot_base_send_lamd_info_args()`

```
int lam_ssi_boot_base_send_lamd_info_args(OPT *aod, unsigned char agent_haddr[4],  
    int agent_port, int node_id, int dli_port, int num_cpus);
```

This function is part of the built-in TCP-based startup rendezvous protocol set (described in Section [B.1.18](#)); it is a generalized version of the `lam_ssi_boot_base_send_lamd_info()` function. The `lam_ssi_boot_base_send_lamd_info()` function will send the LAM daemon information based on parameters that it finds on the command line (i.e., `lamboot` passes the identity of each LAM daemon on the command line

parameters that it launched the daemon with). This function allows a module to use whatever values it wants rather than what is found on the command line. This is necessary for environments where the identity or location of the of LAM daemons may not be available to `lamboot`.

Two environments that do not yet have `boot` modules yet are the Simple Linux Utility for Resource Management (SLURM) [70] and the Quadrics Resource Management System (RMS). In each of those environments, `lamboot` simply tells the run-time environment “launch N LAM daemons.” The run-time environment does not allocate nodes to the job until `lamboot` makes the request, so there is no way to know what nodes they will be booting on ahead of time. Hence, the identification procedure is left up to the LAM daemon instead of `lamboot`.

This function was added in anticipation of supporting those environments, as well as improving support in BProc environments.

#### B.1.15 Utility Function: `lam_ssi_boot_build_inet_topo()`

```
char *lam_ssi_boot_build_inet_topo(struct lamnode *dest_node,  
    struct lamnode origin_node, int origin);
```

Arguments:

- IN: `dest_node` is the destination node
- IN: `origin_node` is the origin node
- IN: `origin` is the node ID of the origin

Create a string for the `$inet_topo` that is suitable for use in the `hbootparse()` function (see example code in Figure B.5).

#### B.1.16 Utility Function: `lam_ssi_boot_do_common_args()`

```
int lam_ssi_boot_do_common_args(OPT *aod, int *argc, char ***argv);
```

Arguments:

- IN: `aod` is a handle to the parsed command line.
- IN/OUT: `argc` is the length of the `argv` array.
- IN/OUT: `argv` is the command line token array.

Utility function to handle some mundane argument handling (e.g., adding “-v” and/or “-d” to `argv` if they are found in `aod`, as per the description of how `lamboot` and `friends` function in their respective man pages). It is typically invoked with the `argv` of a process to start on a remote node, allowing “-v” and “-d” to propagate to remote processes.

### B.1.17 Built-in Algorithms

The `boot` component framework provides generalized algorithms to launch processes across a set of nodes. These algorithms are generally invoked from within `boot` module API calls. The algorithms, in turn, will make callbacks into the module to perform the actual work (e.g., launch a process). The algorithms perform all the necessary bookkeeping and timing to execute the entire set of tasks as well as exchange all startup rendezvous protocol information (if necessary). Note, however, that these functions will all skip nodes that are either not marked with the `NT_BOOT` flag or have a node ID that is equal to `NOTNODEID`.

Note that the use of these functions is not mandatory. They are simply provided as drop-in algorithms so that modules do not need to write their own algorithms.

Section 4.1.7 generally describes the available algorithms. Their names are long because of the SSI prefix rule. Each of the functions below have the same signature:

```
int algorithm(struct lamnode *nodes, int nnodes, int want_startup_protocol,  
             lam_ssi_boot_proc_t which, int *num_started);
```

The arguments are as follows:

- **IN:** `nodes` is an array of nodes to boot across.
- **IN:** `nnodes` is the length of the `nodes` array.
- **IN:** `want_startup_protocol` is a flag indicating whether the function should invoke the module's startup rendezvous protocol functions at the appropriate times during the boot process. It should only be set to 1 when booting LAM daemons; 0 all other times.
- **IN:** `which` is an enum indicating what kind of process to launch (see Section [B.1.4](#)).
- **OUT:** `num_started` will be filled in by the algorithm indicating how many nodes were actually booted.

The provided algorithm functions are:

- `lam_ssi_boot_base_alg_linear()`: Simple linear process-launching algorithm.
- `lam_ssi_boot_base_alg_linear_windowed()`: Linear algorithm with a sliding window for the startup rendezvous protocols. This is especially well-suited for boot environments where remote process invocation latency can be hidden by not waiting for a remote action to finish before progressing onto the next action. This algorithm guarantees that there will never be more than  $N$  outstanding agents waiting to exchange startup rendezvous protocol information.

This algorithm is especially relevant if the built-in TCP startup rendezvous protocols are used (described in Section [B.1.18](#)), because at least some operating system TCP stacks only allow a limited number of clients to be pending on a listening socket. Hence, using the windowed algorithm will guarantee that that operating system limit is never exceeded.

The default window value is 5, and can be changed by setting the `boot_base_linear_win_size` module parameter.

### B.1.18 TCP-Based Startup Rendezvous Protocols

Most (if not all) `boot` modules will be able to use the generalized TCP startup rendezvous protocol functions since TCP is likely able to be used for such meta-information exchanges regardless of the underlying communication network. If TCP connectivity is not available, the `boot` module will need to provide startup rendezvous protocols itself.

These functions eliminate the need for most `boot` modules to provide their own functions for several of the `boot` API calls. Since these functions can be used for the corresponding API functions, only their names are listed below – their signatures and behavior are described in Section [B.2](#):

- `lam_ssi_boot_base_open_srv_connection()`
- `lam_ssi_boot_base_send_lamd_info()`
- `lam_ssi_boot_base_receive_lamd_info()`
- `lam_ssi_boot_base_close_srv_connection()`
- `lam_ssi_boot_base_send_universe_info()`
- `lam_ssi_boot_base_receive_universe_info()`

Note that these functions are subject to operating system limits such as how many pending clients can be held on a listening socket. Some operating systems have a surprisingly low backlog limit. Modules that utilize booting algorithms that could have multiple clients simultaneously calling the server back should be aware of this limitation, and either use multiple servers or some kind of windowed protocol (e.g., the linear windowed algorithm described in Section [B.1.17](#)).

The information that must be exchanged is:

- From the LAM daemon to `lamboot`, send the following:
  - UDP port number that the LAM daemon will use for normal operations
  - Any other information required to call the LAM daemon back to pass the universe information
- From `lamboot` to the LAM daemon, loop sending the following information to each LAM daemon in the LAM universe:
  - Integer node identifier of that LAM daemon (from 0 to  $N-1$ ), or `NOTNODEID` if it is not a valid node
  - Either the byte-packed IP address of the LAM daemon or the string hostname of the LAM daemon
  - Integer UDP port number that the LAM daemon is listening on
  - Integer (bit flags) for the node that the LAM daemon is running on (see Section [B.1.2](#))
  - Integer number of CPUs that the LAM daemon thinks that the node has

```

typedef struct lam_ssi_boot_1_0_0 {
    lam_ssi_1_0_0_t lsb_meta_info;

    /* Initialize / finalize functions */

    lam_ssi_boot_init_fn_t lsb_init;
    lam_ssi_boot_finalize_fn_t lsb_finalize;
} lam_ssi_boot_1_0_0_t;

```

Figure B.6. The `boot` type for exporting the initialization and finalization API function pointers.

## B.2 boot Component Framework Module API

This is version 1.0.0 of the `boot` component framework module API. Each `boot` module must export a `lam_ssi_boot_1_0_0` named `lam_ssi_boot_<name>.module`. This type is defined in Figure B.6. This `struct` contains a small number of items, one of which is a function pointer that may return a pointer to the `struct` shown in Figure B.7.

The majority of the elements in Figures B.6 and B.7 are function pointer types; each is discussed in detail below.

Function prototypes are also marked as one of three classes:

- **Launch Function:** This function is part of the set of functions that start processes on nodes. This function will be invoked by the LAM infrastructure.
- **Algorithm Callback Function:** This function is invoked as a callback from the LAM-provided boot algorithm function frameworks. Although these functions serve as useful abstractions, they are only required if the LAM-provided boot algorithms are used.
- **Rendezvous Function:** This function is part of the set of functions that exchange startup rendezvous information.

`boot` module writers looking for insight into how the API is used should also look at the source code for `lamboot`, `recon`, and `wipe`.

```

typedef struct lam_ssi_boot_actions_1_0_0 {

    /* Boot API function pointers */

    lam_ssi_boot_parse_options_fn_t lsba_parse_options;
    lam_ssi_boot_allocate_nodes_fn_t lsba_allocate_nodes;
    lam_ssi_boot_verify_nodes_fn_t lsba_verify_nodes;
    lam_ssi_boot_prepare_boot_fn_t lsba_prepare_boot;
    lam_ssi_boot_start_rte_procs_fn_t lsba_start_rte_procs;
    lam_ssi_boot_deallocate_nodes_fn_t lsba_deallocate_nodes;

    /* Algorithm callback functions (optional) */

    lam_ssi_boot_start_application_fn_t lsba_start_application;
    lam_ssi_boot_start_rte_proc_fn_t lsba_start_rte_proc;

    /* Startup protocol: sending individual lamd info */

    lam_ssi_boot_open_srv_connection_fn_t lsba_open_srv_connection;
    lam_ssi_boot_send_lamd_info_fn_t lsba_send_lamd_info;
    lam_ssi_boot_receive_lamd_info_fn_t lsba_receive_lamd_info;
    lam_ssi_boot_close_srv_connection_fn_t lsba_close_srv_connection;

    /* Startup protocol: broadcasting universe info */

    lam_ssi_boot_send_universe_info_fn_t lsba_send_universe_info;
    lam_ssi_boot_receive_universe_info_t lsba_receive_universe_info;
} lam_ssi_boot_actions_1_0_0_t;

```

Figure B.7. The `boot` type for exporting the main action API function pointers.

### B.2.1 Data Item: `lsb_meta_info`

`lsb_meta_info` is the SSI-mandated element contains meta-information about the module. See Section [A.6.2](#) for more information about this element.

### B.2.2 Launch Function: `lsb_init`

- Type: `lam_ssi_boot_init_fn_t`

```
typedef const lam_ssi_boot_actions_t>(*lam_ssi_boot_init_fn_t)
(lam_ssi_boot_location_t where, int *priority);
```

- Arguments:
  - **IN:** `where` is an enum indicating where this module is being initialized. The value will be one of the following:
    - \* `LAM_SSI_BOOT_LOCATION_ROOT`: The module is being invoked on the root of the boot. This typically means a user-level command such as `lamboot`, `recon`, `wipe`, or `lamgrow`.
    - \* `LAM_SSI_BOOT_LOCATION_INTERIOR`: This module is being invoked in an interior node of the boot. This may mean that a hierarchical boot algorithm is being used, and that this process is a “helper” launching application. It directly implies that this node has both a parent and one or more children.  
Note that none of the LAM-provided algorithms currently support this value; it is in anticipation of supporting tree-based algorithms.
    - \* `LAM_SSI_BOOT_LOCATION_LEAF`: All other cases. This includes the LAM daemon itself (see below).
  - **OUT:** `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
- Return value: Either `NULL` or a pointer to the struct shown in Figure [B.7](#).
- Description: If the module wants to be considered for selection, it must return a pointer to the struct shown in Figure [B.7](#) that is filled with relevant data and assign an associated priority to `priority`. See Section [3.2.7](#) for more details on the priority system and how modules are selected at run time.

If the module does not want to be considered during the negotiation for this communicator, it should return `NULL` (the value in `priority` is then ignored).

Note that the LAM daemon itself must also initialize the `boot` component framework and come to the same selection conclusion as its peers. Although the LAM daemon will not use any of `boot` API functions to launch remote processes, it will use the startup rendezvous functions to exchange location information with `lamboot`.

### B.2.3 Launch Function: `lsb_finalize`

- Type: `lam_ssi_boot_finalize_fn_t`

```
typedef int (*lam_ssi_boot_finalize_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, nonzero otherwise.
- Description: Finalize the use of this module. It is the last function to be called in the scope of this module's selection before the module close function. It should release any resources allocated during the life of this scope.

### B.2.4 Launch Function: `lsba_parse_options`

- Type: `lam_ssi_boot_parse_options_fn_t`

```
typedef int (*lam_ssi_boot_parse_options_fn_t)(OPT *args, int bhost_schema_args);
```

- Arguments:
  - IN: `args` contains the command line arguments.
  - IN: `bhost_schema_args` is 1 if the `argc/argv` pair contains a boot schema filename (e.g., from `lamboot`, `recon`, and `wipe`), and 0 if the pair contains a string hostname/IP address (e.g., from `lamgrow`).
- Return value: Zero on success, nonzero otherwise.
- Description: The module can examine the command line parameters.

This API function typically makes use of the two utility functions `lam_ssi_boot_base_find_boot_schema()` and `lam_ssi_boot_base_find_hostname()` (described in Sections [B.1.10](#) and [B.1.11](#), respectively).

### B.2.5 Launch Function: `lsba_allocate_nodes`

- Type: `lam_ssi_boot_allocate_nodes_fn_t`

```
typedef int (*lam_ssi_boot_allocate_nodes_fn_t)  
(struct lamnode **nodes, int *nnodes, int *origin);
```

- Arguments:

- OUT: `nodes` is a pointer to a `struct lamnode` that this function is expected to fill with an array of `struct lamnodes`.
  - OUT: `nnodes` is a pointer to an `int` that this function is expected to fill with the length of `nodes` array.
  - OUT: `origin` is a pointer to an `int` that this function is expected to fill with an index into the `nodes` array representing the element for this node.
- Return value: Zero on success, nonzero otherwise.
  - Description: Create and fill in a `lamnode` structure for each node to be booted. There are few requirements on the completeness of the structure, but all unused fields should be zeroed out before returning. Additionally, the `lnd_type` field for the origin member should have the `NT_ORIGIN` and `NT_ME` flags set. Additionally, the total number of nodes must be correct.

Note that this function's actions may be determined by the value of the `bhost_schema_args` flag to the `parse_options()` API call.

The `deallocate_nodes()` API call should later be used to free the memory associated with the nodes list.

This API function typically makes use of the two utility functions `bhostparse()` and `lam_ssi_boot_base_lamgrow()` (described in Sections [B.1.7](#) and [B.1.12](#), respectively).

#### B.2.6 Launch Function: `lsba_verify_nodes`

- Type: `lam_ssi_boot_verify_nodes_fn_t`

```
typedef int (*lam_ssi_boot_verify_nodes_fn_t)(struct lamnode *nodes, int nnodes);
```

- Arguments:
  - IN: `nodes` is the array of `struct lamnodes` returned by the `allocate_nodes()` API call.
  - IN: `nnodes` length of the `nodes` array.
- Return value: Zero on success, nonzero otherwise.
- Description: Last sanity check on the node list. If possible, check the node list for conditions such as (but not limited to):
  - Existence of node (e.g., try to resolve IP names)
  - Permission to execute on node
  - Ensure that the local node is in the list
  - If the number of nodes is greater than one, ensure that the local address is not 127.0.0.1 if using standard IP-passing scheme

After this call, `lamnodes` should be filled in with enough information for the `boot` component framework to contact each of the target nodes. As such, it needs to determine which entry in the array is the origin (this may have been determined by the `allocate_nodes()` API call, but in some cases, it is not possible to determine it until here in `verify_nodes()`).

### B.2.7 Launch Function: `lsba_prepare_boot`

- Type: `lam_ssi_boot_prepare_boot_fn_t`

```
typedef int (*lam_ssi_boot_prepare_boot_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, nonzero otherwise.
- Description: Perform any setup work that might be needed by the `start_rte_procs()` API call, but that only needs to be done once. For example, on BProc architectures, `boot` modules may generate the `argv` arrays for starting up the LAM daemons.

### B.2.8 Launch Function: `lsba_start_rte_procs`

- Type: `lam_ssi_boot_start_rte_procs_fn_t`

```
typedef int (*lam_ssi_boot_start_rte_procs_fn_t)  
(struct lamnode *nodes, int nnodes, lam_ssi_boot_proc_t which,  
int *num_started);
```

- Arguments:
  - IN: `nodes` is the array of nodes to start processes on
  - IN: `nnodes` is the length of the `nodes` array.
  - IN: `which` is an enum specifying which LAM run-time environment process to start.
  - OUT: `num_started` is a pointer to an `int` indicating how many processes were successfully started.
- Return value: Zero on success, nonzero otherwise.

- Description: Takes a `nodes` array and starts a LAM run-time environment process on each node.

The function must only launch on nodes that have the `NT_BOOT` flag set on their type and do not have a node ID of `NOTNODEID`. All other nodes must be skipped.

Note that there is both a return code from this function (indicating overall success or failure) and a separate count of how many processes were started. This is for the case where *some* processes may start properly, but others fail. The `num_started` argument tells the caller how many processes now need to be cleaned up. The `nodes` array can be examined to find out exactly which nodes were successfully booted (`NT_BOOT` must be reset on the `lnd_type` of nodes that were successfully started).

If any of the LAM-provided boot algorithms are used, this is the function that typically invokes them.

### B.2.9 Launch Function: `lsba_deallocate_nodes`

- Type: `lam_ssi_boot_deallocate_nodes_fn_t`

```
typedef int (*lam_ssi_boot_deallocate_nodes_fn_t)
    (struct lamnode **nodes, int *nnodes);
```

- Arguments:
  - IN/OUT: `nodes` is an array of nodes.
  - IN/OUT: `nnodes` is the length of the `nodes` array.
- Return value: Zero on success, nonzero otherwise.
- Description: Clean up any memory associated with the node allocation step. Although not required, modules are encouraged to reset `nodes` and `nnodes` to `NULL` and `0`, respectively, when the function returns. This function will be called only after the nodes information is no longer needed. This function typically invokes `lam_deallocate_nodes()` (see Section B.1.9).

### B.2.10 Algorithm Callback Function: `lsba_start_application`

- Type: `lam_ssi_boot_start_application_fn_t`

```
typedef int (*lam_ssi_boot_start_application_fn_t)
    (char ***argv, int *argc, int num_procs, struct lamnode *node);
```

- Arguments:
  - IN: `argv` is an array of `argv`-style command line arguments; i.e., an array of commands to start.
  - IN: `argc` is an array of integers indicating how long each array is in the `argv` array.
  - IN: `num_procs` is the length of the first dimension of `argv`.
  - IN: `node` is a pointer to a single `struct lamnode` indicating which node to start on.

- Return value: Number of processes successfully started.

- Description: Launch the specified processes on the specified node. Return the number of processes successfully booted. Hence, if the return value is equal to `num_procs`, the function completed successfully. There is no mandate that processes be started in the order they exist in `argv`.

The return value is explicitly vague so that modules can get “even more parallelism” if they happen to use a remote startup agent that provides a high degree of parallelism.

It is incorrect to use this function directly from a boot algorithm to launch a LAM run-time environment process (e.g., `lamd`, `recon`, `wipe`). Use the `start_rte_proc()` API function instead. This function should only be used by a boot algorithm to start up other instances of the boot algorithm (i.e., “helper” executables).

### B.2.11 Algorithm Callback Function: `lsba_start_rte_proc`

- Type: `lam_ssi_boot_start_rte_proc_fn_t`

```
typedef int (*lam_ssi_boot_start_rte_proc_fn_t)
(struct lamnode *node, lam_ssi_boot_proc_t which);
```

- Arguments:
  - IN: `node` is a pointer to a single `struct lamnode` indicating where the process should be started.
  - IN: `which` is an enum indicating what kind of LAM run-time environment process should be started (see Section [B.1.4](#)).

- Return value: Zero on success, nonzero otherwise.

- Description: Start a LAM run-time environment process on the specified node. This can use the `start_application()` API function (in fact, it is encouraged).

This function exists because the boot algorithm should not need to know any of the details about starting a LAM run-time environment process. This function

provides an upcall to give the boot algorithm an abstract mechanism to launch a LAM run-time environment process.

#### B.2.12 Rendezvous Function: `lsba_open_srv_connection`

- Type: `lam_ssi_boot_open_srv_connection_fn_t`

```
typedef int (*lam_ssi_boot_open_srv_connection_fn_t)
    (struct lamnode *nodes, int nnodes);
```

- Arguments:
  - IN: `nodes` is a pointer to an array of `struct lamnodes` that are expected to connect to this process.
  - IN: `nnodes` is the length of the `nodes` array.
- Return value: Zero on success, nonzero otherwise.
- Description: Open a private, server-side communication endpoint (i.e., the channel will only be used within the boot component framework) that the LAM daemon will connect back to. This function will only be called once in the scope of the module, meaning that if you use the provided linear algorithms, it will only be called once on the origin node. It is conceivable that with other types algorithms, such as tree-based algorithms, this function may be called multiple times on different nodes.

Note that the addresses given in `nodes` may or may not be the actual clients that connect. There are some valid network architectures where connections may seem to come from addresses other than what are listed in the `nodes` array. It is suggested to boot module authors that unless the special “boot promiscuous mode” is enabled in LAM, only accept connections from the addresses listed in the `nodes` array (when possible). However, when “promiscuous mode” is enabled, accept connections from anywhere, and rely on the client to identify themselves in the boot protocol.

#### B.2.13 Rendesvouz Function: `lsba_send_lamd_info`

- Type: `lam_ssi_boot_send_lamd_info_fn_t`

```
typedef int (*lam_ssi_boot_send_lamd_info_fn_t)(OPT *args, int dli_port);
```

- Arguments:
  - IN: `args` contains the command line parameters.

- IN: `dli_port` is the UDP port number that the local LAM daemon is listening on for normal operations.
- Return value: Zero on success, nonzero otherwise.
- Description: Open a connection back to the booting agent, send relevant location information (e.g., the LAM's UDP port number), and then closes the connection. It is assumed that the information necessary to connect back to the invoking agent is either in the command line arguments or available in a module parameter.

#### B.2.14 Rendesvouz Function: `lsba_receive_lamd_info`

- Type: `lam_ssi_boot_receive_lamd_info_fn_t`

```
typedef int (*lam_ssi_boot_receive_lamd_info_fn_t)
(struct lamnode *nodes, int nnodes);
```

- Arguments:
  - IN/OUT: `nodes` is a pointer to an array of `struct lamnodes` that were successfully started.
  - IN: `nnodes` is the length of the `nodes` array.
- Return value: Zero on success, nonzero otherwise.
- Description: Accept a connection from a LAM daemon and receive the information it sends back. The function is provided with an array of `struct lamnode` entries, one of which will correspond to the LAM daemon that will be contacting it. It is up to the function to figure out which one is responding. When finished, close the connection. A new connection is used to broadcast the information at a later time.

In the case that only one LAM daemon can be communicating with the function, (for example, when the boot algorithm is linear), then `nnodes` will one and the job of searching is much easier.

#### B.2.15 Rendesvouz Function: `lsba_close_srv_connection`

- Type: `lam_ssi_boot_close_srv_connection_fn_t`

```
typedef int (*lam_ssi_boot_close_srv_connection_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, nonzero otherwise.

- Description: Close the channel opened during the `open_srv_connection()` API function.

#### B.2.16 Rendesvouz Function: `lsba_send_universe_info`

- Type: `lam_ssi_boot_send_universe_info_fn_t`

```
typedef int (*lam_ssi_boot_send_universe_info_fn_t)
(struct lamnode *nodes, int nnodes, int which);
```

- Arguments:
  - IN: `nodes` is an array of nodes that the information needs to be broadcast to.
  - IN: `nnodes` is the length of the `nodes` array.
  - IN: `which` is an index into the `nodes` array indicating which node to connect and send the information to.
- Return value: Zero on success, nonzero otherwise.
- Description: Connect to LAM daemon and send the union of all the LAM location information (i.e., send information about all the peer LAM daemons that comprise the LAM universe).
 

This function opens a connection to a target LAM daemon, sends the information, and disconnects.

#### B.2.17 Rendesvouz Function: `lsba_receive_universe_info`

- Type: `lam_ssi_boot_receive_universe_info_t`

```
typedef int (*lam_ssi_boot_receive_universe_info_t)
(struct lamnode **universe, int *universe_size);
```

- Arguments:
  - OUT: `universe` is a pointer to an (as yet unallocated) array of information that will be received.
  - OUT: `universe_size` is a pointer to an `int` that will be filled to be the length of the `universe` array.
- Return value: Zero on success, nonzero otherwise.

- Description: After the LAM daemon communicates its port information to the booting process, it waits for information about the entire run-time universe. This function is where it waits for that information. It returns the information returned about all neighbors.

Similar to the `receive_lamd_info()` API function, this function should accept the connection, read the information, and close the connection when finished reading.

## APPENDIX C

### MPI POINT-TO-POINT COMMUNICATION COMPONENT INTERFACE

The `rpi` component framework is described in Chapter 5. This Appendix describes the technical details and requirements for `rpi` modules [125]. Section C.1 discusses header files, types, global variables, and utility functions that are provided to all `rpi` modules. Section C.2 details the module interface modules and functions.

#### C.1 Services Provided by the `rpi` SSI

Several services are provided by the `rpi` component framework that are available to all `rpi` modules.

##### C.1.1 Header Files

The following header files must be included (in order) in all module source files that want to use any of the common `rpi` component services described in this document:

```
#include <lam-ssi.h>  
#include <lam-ssi-rpi.h>
```

Both of these files are included in the same location in the LAM source tree: `share/-include`. Appendix A.6.1 describes how to provide appropriate preprocessor flags to include these files properly.

```

struct _gps {
    /* Unique identification in the LAM universe */

    int4 gps_node;
    int4 gps_pid;
    int4 gps_idx;

    /* Rank in MPI_COMM_WORLD; used for convenience only */

    int4 gps_grank;
};

```

Figure C.1. `struct _gps`: GPS type; unique process identification in a LAM universe. It is mainly used for MPI process identification and LAM out-of-band messaging.

### C.1.2 Internal Type: `struct _gps`

This type is used to identify the location of a process in the LAM universe. It is typically used to fill in attributes required for the LAM out-of-band messaging system.

Figure C.1 shows its definition.

The individual elements are:

- `gps_node`: The node ID in the LAM universe where the process is running. This will be an integer in  $[0, N)$ , where  $N$  is the number of nodes in the LAM universe.
- `gps_pid`: The POSIX PID of the process that invoked `MPI_INIT`.<sup>1</sup>
- `gps_idx`: The index of the process in the local LAM daemon’s process table.
- `gps_grank`: The “global rank” of the process. This is the integer rank of this process in its `MPI_COMM_WORLD`.

### C.1.3 Internal Type: `struct _proc`

This type is used to describe MPI processes. Each process in LAM/MPI maintains a linked list of all the processes that it knows about and can communicate with (including

<sup>1</sup>Note that because of Linux kernel 2.x POSIX thread semantics, this may or may not be the PID of the main thread. However, this *is* the PID of the process (thread) that invoked `MPI_INIT` (and therefore `kinit()`), and the PID that will be returned by `lam_getpid()`.

```

struct _proc {
    /* GPS uniquely identifying this process in the LAM universe */

    struct _gps p_gps;

    /* Various attributes */

    int p_get_nsnd;
    int p_mode;
    int p_refcount;
    int p_num_buf_env;

    /* Hook for the rpi module to hang data */

    struct lam_ssi_rpi_proc *p_rpi;
};

```

Figure C.2. `struct _proc`: Process entry.

itself). See Figure C.2.

It is important to recognize that each of these elements are set from the perspective of the process on which the instance resides. An individual `_proc` instance indicates what process *A* knows about process *B*. Hence, it is possible for multiple processes to have different values for the individual elements, even if they are referring to the same target process.

The individual elements are:

- `p_gps`: The GPS for the process. See Section C.1.2.
- `p_get_nsnd`: The number of messages sent to this process. Used for Guaranteed Envelope Resources (GER) purposes, for `rpi` modules that support them.
- `p_mode`: A bit-mapped flag indicating various things about the process. Valid flags are:
  - `LAM_PFLAG`: Generic flag used for marking. Care needs to be taken to ensure that two portions of code are not using/modifying `LAM_PFLAG` marking simultaneously.
  - `LAM_PDEAD`: Set if the process has died.

- LAM\_PRPIINIT: Set if the rpi module has been initialized on this process.
- LAM\_PCLIENT: Set if the process is in the “client” MPI\_COMM\_WORLD (e.g., if the process was spawned or connected to after MPI\_INIT) in this process’ original MPI\_COMM\_WORLD.
- LAM\_PHOMOG: Set if the process has the same endian as this process.
- LAM\_PRPICONNECT: Set if the rpi module has connected to this process or not.

Note the distinction between whether the rpi module has initialized a process and whether it has connected to it. A process typically only needs to be initialized once, but may be connected multiple times throughout its life (e.g., checkpointing may force disconnects, or an rpi module may choose to only keep processes connected when they are actively communicating).

The rpi module must leave the lower 8 bits of p\_mode alone – they are reserved for the MPI layer.

- p\_refcount: The reference count for this process. Every time a communicator or group includes this process, its reference count is incremented (and vice versa) by the MPI layer.
- p\_num\_buf\_env: Number of buffered envelopes.
- p\_rpi: This member is of type ( lam\_ssi\_rpi\_proc \* ), which must be defined by each rpi module. It is a mechanism for the rpi module to attach rpi-specific state information to a \_proc. This information is typically state information for the connection to/from the process.

For example, the tcp module stores (among others things):

- File descriptor of the socket connection to the remote process
- Pointers to requests (if any) that are currently being read from the socket
- Pointers to requests (if any) that are currently being written to the socket

#### C.1.4 Internal Type: struct \_req

For each process, the LAM/MPI library maintains a linked list of all the requests that need to be progressed. The MPI layer keeps this progression list in order and also removes requests upon completion. rpi authors can thread other lists through the progression list via rpi-specific data. Several of the rpi functions deal with creating requests and moving them along to completion. See below.

```

typedef struct _req *MPI_Request;
struct _req {
    /* For debugging */

    char *rq_name;

    /* State flags and attached buffer (if any) */

    int rq_type;
    int rq_state;
    int rq_marks;
    int rq_flags;
    char *rq_packbuf;
    int rq_packsize;

    /* Arguments to top-level MPI function */

    int rq_count;
    void *rq_buf;
    MPI_Datatype rq_dtype;
    int rq_rank;
    int rq_tag;
    MPI_Comm rq_comm;

```

Figure C.3. `struct _req`: Underlying structure for `MPI_Request`, part 1.

A request is represented by a structure of type (`struct _req`). The type `MPI_Request` is actually a typedef: (`struct _req *`). Hence, when top-level MPI functions pass an `MPI_Request`, they are actually pointing to an underlying `struct _req`. This bears importance, particularly since it implies that the MPI layer must allocate and free individual `struct _req` instances (see the `LAM.RQFDYNREQ` mark, below). See Figures C.3 and C.4.

Each request is typically the result of a call to some kind of MPI send or receive function. The individual members are:

- `rq_name`: String name of the request. This is typically only used for debugging purposes. If not `NULL`, it needs to point to storage on the heap; it will be freed

```
/* Fast lookups, resolved at request creation time */

int rq_cid;
int rq_func;
int rq_seq;
int rq_f77handle;
MPI_Status rq_status;
struct _bsndhdr *rq_bsend;
struct _proc *rq_proc;
struct _req *rq_next;

/* Mainly for IMPI; do not use */

void *rq_extra;
int (*rq_hdlr)();
MPI_Request rq_shadow;

/* Hook for the rpi module to hang data */

struct lam_ssi_rpi_req *rq_rpi;
};
```

Figure C.4. `struct _req`: Underlying structure for `MPI_Request`, part 2.

when the request is destroyed.

- `rq_type`: Flag indicating the type of action specified by this request. Valid values are:
  - `LAM_RQISEND`: Normal mode send
  - `LAM_RQIBSEND`: Buffered mode send
  - `LAM_RQISSEND`: Synchronous mode send
  - `LAM_RQIRSEND`: Ready mode send
  - `LAM_RQIRECV`: Receive
  - `LAM_RQIPROBE`: Probe
  - `LAM_RQIFAKE`: “Fake” request used to indicate a non-blocking buffered send. The `rpi` module should never see a request of this type – the MPI layer should handle it internally. This flag value is only mentioned here for the sake of completeness.

Note that whether the request is blocking is not indicated by `rq_type` – it is indicated by `rq_flags`, described below.

- `rq_state`: The state of this request. Valid values are:
  - `LAM_RQSINIT`: Init state (not active). No communication allowed.
  - `LAM_RQSSTART`: Start state (active, but not yet done). Communication allowed, but not required.
  - `LAM_RQSACTIVE`: Active state (active, but not yet done). Communication allowed, but only required if the destination process is not this process.
  - `LAM_RQSDONE`: Done state (not active). No communication allowed.

It is critical for `rpi` modules to update this member properly. For example, MPI semantics require that `MPI_REQUEST_FREE` may be invoked on a request before the communication associated with it has completed. In such a case, the LAM examines the state of the `rq_state` member to see if it is safe to actually destroy the request or not; the request will only be destroyed if the state is `LAM_RQSINIT` or `LAM_RQSDONE`. Otherwise, the request will be marked as an orphan (see `rq_flags`, below) and LAM will free it when it is actually finished.

- `rq_marks`: Bit-mapped persistent flags on a request. These flags will endure through the entire life of a request, regardless of its state (e.g., flags that only need to be set once on persistent requests). The `rpi` module should not modify these values. Valid values are:
  - `LAM_RQFPERSIST`: This request is persistent.
  - `LAM_RQFDYNBUF`: The buffer associated with this request is dynamic and will be automatically freed when the request is destroyed.
  - `LAM_RQFDYNREQ`: The request itself is dynamic and will be freed when the request is destroyed.

- LAM\_RQFSOURCE: A source request (i.e., indicates direction of message transfer).
  - LAM\_RQFDEST: A destination request (i.e., indicates direction of message transfer).
  - LAM\_RQFBLKTYPE: Indicates that the request is a blocking request.
  - LAM\_RQFOSORIG: Origin of a one-sided request.
  - LAM\_RQFOSTARG: Target of a one-sided request.
  - LAM\_RQFMAND: A mandatory request. This mark is maintained by the MPI layer, and is really only intended for IMPI communications, and will (hopefully) someday be removed.
- `rq_flags`: Bit-mapped active request flags. These will be reset for each iteration through the start state. The `rpi` module should only ever modify the `LAM_RQFTRUNC` value; all other values should not be modified by the `rpi` module. Valid values are:
    - LAM\_RQFCANCEL: The request has been canceled.
    - LAM\_RQFBLOCK: The request is blocking.
    - LAM\_RQFTRUNC: The request was (or is about to be) truncated. If set, this will cause an `MPI_ERR_TRUNCATE` error in the MPI layer.
    - LAM\_RQFHDLDONE: The handler for this request has already been invoked.
    - LAM\_RQFORPHAN: This request is an orphan and needs to be automatically freed when it is done.
    - LAM\_RQFMARK: Arbitrary marking for requests. Care should be taken to ensure that two independent sections of code don't attempt to use/modify the `LAM_RQFMARK` flag at the same time.
    - LAM\_RQFACKDONE: The ACK for this request has completed (e.g., for rendezvous protocols). This flag is only for IMPI support, and should not be used by `rpi` modules.
  - `rq_packbuf`: Pointer to start of contiguous buffer of message data to be sent or area where message data is to be received. Depending on the MPI datatype of data to be sent/received, this may or may not be the same as `rq_buf`, which is a pointer to the buffer given by the user. The MPI layer handles packing and unpacking of this buffer.
  - `rq_packsize`: The size of the data to be sent/received in bytes. This is set by the MPI layer. This is how much message data the `rpi` module must send/receive for the request.
  - `rq_count`: Parameter from the original MPI function call.
  - `rq_buf`: Parameter from the original MPI function call.
  - `rq_dtype`: Parameter from the original MPI function call.

- `rq_rank`: Parameter from the original MPI function call.
- `rq_tag`: Parameter from the original MPI function call.
- `rq_comm`: Parameter from the original MPI function call.
- `rq_cid`: The context ID to use in the message envelope. It corresponds to the communicator member `rq_comm`.
- `rq_func`: A flag indicating which top-level MPI function created this request. See the file `share/include/blktype.h` for a list of valid values.
- `rq_seq`: The message sequence number. If the `rpimodule` is to work with LAM tracing, then this number must be sent with each message (it is set by the MPI layer) and then on the receiving side, the `rpi` module must extract it and set this field in the receiving request with its value.
- `rq_f77handle`: Handle index used by Fortran.
- `rq_status`: The status of a request. In the case of a receive request, the `rpi` module must fill the `MPI_SOURCE` field of this structure with the rank of the sender of the received message, the `MPI_TAG` field with the tag of the received message, and the `st_length` field with the number of bytes in the received message.
- `rq_bsend`: Pointer to the buffer header in the case of a buffered send. Used by MPI layer only.
- `rq_proc`: Pointer to the peer process. This is initially set by the MPI layer. In the case of a receive on `MPI_ANY_SOURCE`, it will be `NULL`. Once the actual source has been determined, the `rpi` module may set it to point to the peer process but is not required to do so.
- `rq_next`: Pointer to the next request in the list. Do not modify. If the `rpi` module needs to maintain its own request lists, it must do so through the `rpi`-specific information handle (`rq_rpi`).
- `rq_extra`: A general place to hang extra state off the request. Used for one-sided and IMPI communications; the `rpimodule` should not modify this field.
- `rq_hdlr`: Function to invoke when a request has moved into the done state. Don't touch this; it is used exclusively in the MPI layer (mostly by one-sided communication).
- `rq_shadow`: "Shadow" requests used by IMPI. Don't touch these; they are handled exclusively in the MPI layer.
- `rq_rpi`: `rpi`-specific data hanging off the request. Its type is `(lam_ssi_rpi_req *)`, and must be defined by the `rpi` module. For example, the `tcp` `rpi` module stores (among others things) the request envelope and a pointer into the data buffer indicating the current read/write position.

### C.1.5 Internal Type: `struct _comm`

The `struct _comm` type is the underlying type for an `MPI_Comm`. The majority of members of this type are probably not useful to `rpi` authors; detailed definitions of this members are skipped for brevity. See Figure C.5.

The individual members are:

- `c_flags`: Bit flags indicating various characteristics of the communicator. The defined flags on this field are:
  - `LAM_CINTER`: If set, this communicator is an inter-communicator. If clear, this communicator is an intra-communicator.
  - `LAM_CLDEAD`: At least one process in the local group is dead.
  - `LAM_CRDEAD`: At least one process in the remote group is dead.
  - `LAM_CFAKE`: This is a “fake” IMPI communicator. `rpi` modules should never see this flag.
  - `LAM_CHOMOG`: All the processes on this communicator are endian-homogeneous. This flag is merely a shortcut for traversing all the procs in a given communicator to see if they are endian-homogeneous or not.
- `c_contextid`: Integer context ID.
- `c_refcount`: Reference count – effectively how many communications are currently using this communicator (since it is possible to `MPI_COMM_FREE` a communicator before all non-blocking communication has completed).
- `c_group`: Local group. Will be a meaningful group for intra-communicators, and `MPI_GROUP_NULL` for inter-communicators,
- `c_rgroup`: Remote group. Will be `MPI_GROUP_NULL` for intra-communicators, and a meaningful group for inter-communicators.
- `c_keys`: MPI attribute key hash table. See [9] for more discussion of the `HASH_LAM` type and accessor functions.
- `c_cube_dim`: Inscribed cube dimension of the communicator. Used for binomial trees in MPI collectives.
- `c_topo_type`: Topology type; either `MPI_GRAPH`, `MPI_CART`, or `MPI_UNDEFINED`.
- `c_topo_nprocs`: Number of processes in topology communicators.
- `c_topo_ndims`: Number of dimensions in Cartesian communicators.
- `c_topo_nedges`: Number of edges in graph communicators.

```

typedef struct _comm *MPI_Comm;
struct _comm {
    /* Identification and reference count information */

    int c_flags;
    int c_contextid;
    int c_refcount;

    /* Groups and attributes */

    MPI_Group c_group;
    MPI_Group c_rgroup;
    HASH *c_keys;

    /* Topology information */

    int c_cube_dim;
    int c_topo_type;
    int c_topo_nprocs;
    int c_topo_ndims;
    int c_topo_nedges;
    int *c_topo_dims;
    int *c_topo_coords;
    int *c_topo_index;
    int *c_topo_edges;

    /* Extra MPI information */

    int c_f77handle;
    MPI_Win c_window;
    MPI_Errhandler c_errhdl;
    char c_name[MPI_MAX_OBJECT_NAME];

    /* Reserved / IMPI -- do not use */

    MPI_Comm c_shadow;
    long c_reserved[4];
};

```

Figure C.5. `struct _com`: Underlying structure for `MPI_Comm`.

- `c_topo_dims`: Array of dimensions for Cartesian communicators.
- `c_topo_coords`: Array of coordinates for Cartesian communicators.
- `c_topo_index`: Array of indices for graph communicators.
- `c_topo_edges`: Array of edges for graph communicators.
- `c_f77handle`: Fortran integer handle for this communicator.
- `c_window`: In LAM/MPI, windows for one-sided message passing are implemented on top of communicators. Abstractly, all windows “have” a communicator that they communicate on (even though it is implemented the other way around – communicators designated for one-sided message passing “have” a window).
- `c_errhdl`: Error handler for this communicator.
- `c_name`: A string name for the communicator.
- `c_shadow`: A “shadow” communicator that is used by IMPI.
- `c_reserved`: Reserved for future expansion.

#### C.1.6 Internal Type: `struct _group`

The `struct _group` type is the underlying type for an `MPI_Group`. This type is probably not useful to `rpi` authors, but it is included here for completeness. See [Figure C.6](#).

The individual members are:

- `g_nprocs`: The size of the group, i.e., the size of the `g_procs` array.
- `g_myrank`: The index of this process in the `g_procs` array. If the process is not in the group, this will be `MPI_UNDEFINED`.
- `g_refcount`: The reference count of this variable. Reference counting is maintained by the MPI layer.
- `g_f77handle`: The Fortran integer handle of this group.
- `g_procs`: Pointer to an array of pointers to the processes in the group. The array is in order of rank in the group. Note that these are simply references to the real, underlying `_proc` instances that represent the peer MPI processes – they are not copies. Be very careful modifying what these pointers refer to.

```

typedef struct _group *MPI_Group;
struct _group {
    /* State and identification */

    int g_nprocs;
    int g_myrank;
    int g_refcount;
    int g_f77handle;

    /* Array of pointers into the process list */

    struct _proc **g_procs;
};

```

Figure C.6. `struct _group`: Underlying structure for `MPI_Group`.

```

typedef struct _status {
    /* Public members (per MPI-1 standard) */

    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;

    /* Private member */

    int st_length;
} MPI_Status;

```

Figure C.7. `struct _status`: Underlying structure for `MPI_Status`.

```

struct lam_ssi_rpi_envl {
    int4 ce_len;
    int4 ce_tag;
    int4 ce_flags;
    int4 ce_rank;
    int4 ce_cid;
    int4 ce_seq;
};

```

Figure C.8. `struct _lam_ssi_rpi_envl`: General structure for envelopes.

### C.1.7 Internal Type: `struct _status`

The `struct _status` type is the underlying type for an `MPI_Status`. See Figure C.7.

Note that by definition in the MPI standard, the first three members listed above (`MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`) are public variables.

The individual members are:

- `MPI_SOURCE`: As described by the MPI standard.
- `MPI_TAG`: As described by the MPI standard.
- `MPI_ERROR`: As described by the MPI standard.
- `st_length`: Private variable for use by the MPI and `rpi` layers. It is the length of the message in bytes.

### C.1.8 Internal Type: `struct lam_ssi_rpi_envl`

The following type is provided as a prototype envelope that can be used by `rpi` modules for prefixing data messages across a communications channel. Although the use of this specific `struct` is not [yet] required, it is strongly encouraged because it will provide compatibility with TotalView debugging, unexpected queue support, and may become required for multi-`rpi` support. See Figure C.8.

The individual members are:

- `ce_len`: Message length (bytes).
- `ce_tag`: Tag of the message (16 bits max).
- `ce_flags`: Flags on this particular envelope. Valid values are:
  - **C2CTINY**: “Tiny” message protocol (usually indicates that the envelope and message payload are included in a single message (i.e., the message payload may have already been received by receiving the envelope)).
  - **C2CSHORT**: “Short” message protocol (usually indicates that the envelope and message payload were sent in separate messages, or, more specifically, must be received in separate buffers, but the message payload directly follows the envelope).
  - **C2CLONG**: “Long” or “rendesvouz” message protocol (usually indicates a three-way handshake to actually transfer the message). This is required for arbitrarily long messages where resources may need to be allocated on the receiver before they can be received properly.
  - **C2CACK**: During a long message protocol handshake, the receiver sends an envelope back to the sender with this bit set indicating that it is ready to receive the main body of the message.
  - **C2C2ND**: During a long message protocol handshake, the sender sends an envelope with this bit set, indicating that the message payload is immediately following.
  - **C2CSSEND**: Indicates a synchronous mode send, which requires the receiver to send back an envelope with **C2CACK** set before the sending request can complete.
  - **C2CBOX**: Long message using the postbox protocol.
  - **C2CBUFFERED**: The envelope has previously been buffered.
- `ce_rank`: Peer rank. This may be the source or the destination, depending on the context of the envelope.
- `ce_cid`: Context ID of the communicator in which this message is being sent/received.
- `ce_seq`: Sequence number.

### C.1.9 Internal Type: `struct lam_ssi_cbuf_msg`

This type may be used for unexpected message buffering. Its use is strongly recommended (see Section [C.1.14](#), page 239) in order to enable external access to unexpected message queues.

The individual members are as follows:

```

struct lam_ssi_rpi_cbuf_msg {
    /* Identification of the message */

    struct _proc *cm_proc;
    struct lam_ssi_rpi_envl cm_env;

    /* Hook for the RPI buffer to hang extra data */

    struct lam_ssi_rpi_cbuf *cm_extra;

    /* Message data buffer */

    char *cm_buf;

    /* Cleanup flag */

    int cm_dont_delete;

    /* Only valid of sending to self */

    MPI_Request cm_req;
};

```

Figure C.9. `struct _lam_ssi_rpi_cbuf_msg`: Unexpected message bodies.

- `cm_proc`: Pointer to the source process.
- `cm_env`: Copy of the incoming envelope.
- `cm_buf`: Message data. This may or may not be `NULL`. For example, in the `tcp_rpi` module, this pointer only ever points to short messages because long message data has not yet been received, since, by definition, an unexpected message receipt means that a corresponding long receive request has not yet been posted.
- `cm_dont_delete`: Flag to indicate whether the buffer should be freed or not when the request has completed.
- `cm_req`: The send request *only* if the sender is this process; `NULL` otherwise.

The use of this `struct` is explained more fully in Section [C.1.14](#).

#### C.1.10 Global Variable: `struct _proc *lam_myproc`

A pointer to the `struct _proc` (described in Section [C.1.3](#)) of this process. It is most commonly used to get the attributes and/or GPS (see Section [C.1.2](#)) of this process.

This variable is `extern`'ed in `<mpisys.h>`.

#### C.1.11 Utility Function: `lam_memcpy()`

`lam_memcpy()`: While technically not an `rpi`-specific call, `lam_memcpy()` is important because some platforms have poor implementations of `memcpy()`. On these platforms, `lam_memcpy()` (particularly with mid- to large-sized copies) may significantly outperform the native `memcpy()`. On platforms with “good” implementations of `memcpy()`, `lam_memcpy()` will actually be a `#define` that maps to `memcpy()` in order to use the native function without any additional function call overhead. Hence, it is always safe to use `lam_memcpy()` instead of `memcpy()`, and ensure portable memory copying performance.

The prototype for this function is the same as for `memcpy()`.

### C.1.12 Utility Function: `lam_ssi_rpi_base_alloc_mem()`

`lam_ssi_rpi_base_alloc_mem()`: A wrapper around `malloc(3)`. This function is provided for `rpi` modules that do not wish to provide their own `lsra_alloc_mem()` functions.

This function fulfills all the requirements (such as prototype) as the `lsra_alloc_mem()` API call needs. See Section [C.2.14](#) (page 250).

### C.1.13 Utility Function: `lam_ssi_rpi_base_free_mem()`

`lam_ssi_rpi_base_free_mem()`: A wrapper around `free(3)`. This function is provided for `rpi` modules that do not wish to provide their own `lsra_free_mem()` functions.

This function fulfills all the requirements (such as prototype) as the `lsra_free_mem()` API call needs. See Section [C.2.15](#) (page 251).

### C.1.14 Utility Functions: Unexpected Message Buffering

It is strongly recommended that `rpi` modules use the `cbuf_*()` functions provided by LAM for unexpected message buffering for the following reasons:

- When the `rpi` design is evolved into multi-`rpi`, having a common buffering for unexpected messages will likely be required to handle unexpected messages in conjunction with `MPI_ANY_SOURCE` in communicators that span multiple `rpi` modules.
- LAM/MPI supports the Etnus TotalView parallel debugger which has the ability to display MPI message queues. LAM exports the unexpected message queues through the standard functions described in this section; if an `rpi` module uses the LAM-provided functions, TotalView will be able to see the unexpected message queue.

The LAM-provided functions for unexpected message buffering are:

- `lam_ssi_rpi_cbuf_init(void)`: This function is required to be called before any unexpected buffering can occur. It is invoked automatically by the `rpi` SSI startup glue after all `rpi` module `open()` functions are invoked, but *before* any `rpi` module `lsr_init()` functions (see Section [C.2.4](#), page 243) are invoked. This function is only mentioned here for completeness.

- `lam_ssi_rpi_cbuf_end(void)`: This function cleans up all the storage and state associated with unexpected message buffering. It is invoked automatically by the rpi SSI shutdown glue after all rpi module `lsra_finalize()` functions are invoked with `(proc == 0)` (see Section C.2.6, page 245), but *before* the rpi module `close()` functions are invoked.
- `lam_ssi_rpi_cbuf_find(struct lam_ssi_rpi_envl *rqenv)`: Given a pointer to an envelope, find if there are any matching messages on the unexpected message queue. If there are no matching messages, `NULL` is returned. Otherwise, a pointer to the first matching `struct lam_ssi_rpi_cbuf_msg` is returned that contains information about the buffered message (see Section C.1.9, page 236).  
Note that because this function may be invoked by a probe, it does *not* remove the message from the unexpected queue.
- `lam_ssi_rpi_cbuf_delete(struct lam_ssi_rpi_cbuf_msg *msg)`: Delete a specific message from the unexpected message queue. The argument is a pointer that was returned from either `lam_ssi_rpi_cbuf_find()` or `lam_ssi_rpi_cbuf_append()`.
- `lam_ssi_rpi_cbuf_append(struct lam_ssi_rpi_cbuf_msg *msg)`: Append a new unexpected message to the end of the queue. `*msg` is copied by value, so there's no need for `msg` to point to stable storage. See Section C.1.9 (page 236) for an explanation of this type. This function returns a pointer to the buffered message upon success, or `NULL` on failure.

## C.2 rpi Component Framework Module API

This is version 1.1.0 of the rpi component framework module API. Each rpi module must export a `lam_ssi_rpi_1_0_0` named `lam_ssi_rpi_<name>.module`. This type is defined in Figure C.10. A second `struct` is used to hold the majority of function pointers and flags for the module. It is only used if the module is selected, and is shown in Figure C.11.

The majority of the elements in Figures C.10 and C.11 are function pointer types; each is discussed in detail below. When describing the function prototypes, the parameters are marked in one of three ways:

- **IN**: The parameter is read – but not modified – by the function.
- **OUT**: The parameter, or the element pointed to by the parameter may be modified by the function.

```

typedef struct lam_ssi_rpi_1_1_0 {
    lam_ssi_1_1_0_t lsr_meta_info;

    /* rpi API function pointers */

    lam_ssi_rpi_query_fn_t lsr_query;
    lam_ssi_rpi_init_fn_t lsr_init;
} lam_ssi_rpi_1_1_0_t;

```

Figure C.10. `struct lam_ssi_rpi_1_1_0`: The `rpi` basic type for exporting the module meta information and initial query / initialization function pointers.

- **IN/OUT**: The parameter, or the element pointed to by the parameter is read by, and may be modified by the function.

`rpi` module writers looking for insight into how the API is used should also look at the source code in `share/mpi/lamreqs.c`. Most MPI functions that involve communication eventually call one or more of the functions in this file.

Unless specifically noted, none of the functions may block. Note that this may make single-threaded implementations arbitrarily complicated. For example, the state machine used in the `tcp` `rpi` module is extremely complicated for this very reason; in non-blocking mode, reads and writes on sockets may return partial completion which will require re-entering the same state at a later time.

### C.2.1 Restrictions

It is illegal for any `rpi` API function to invoke top-level MPI functions.

### C.2.2 Data Item: `lsr_meta_info`

`lsr_meta_info` is the SSI-mandated element contains meta-information about the module. See Section [A.6.2](#) for more information about this element.

```

typedef struct lam_ssi_rpi_actions_1_1_0 {
    /* rpi API function pointers */

    lam_ssi_rpi_addprocs_fn_t lsra_addprocs;
    lam_ssi_rpi_finalize_fn_t lsra_finalize;

    /* Request lifecycle */

    lam_ssi_rpi_build_fn_t lsra_build;
    lam_ssi_rpi_start_fn_t lsra_start;
    lam_ssi_rpi_advance_fn_t lsra_advance;
    lam_ssi_rpi_destroy_fn_t lsra_destroy;

    /* Non-blocking probe; special enough to require its own function */

    lam_ssi_rpi_iprobe_fn_t lsra_iprobe;

    /* "Fast" sending and receiving */

    lam_ssi_rpi_fastrecv_fn_t lsra_fastrecv;
    lam_ssi_rpi_fastsend_fn_t lsra_fastsend;

    /* MPI-2 Memory management */

    lam_ssi_rpi_alloc_mem_fn_t lsra_alloc_mem;
    lam_ssi_rpi_free_mem_fn_t lsra_free_mem;

    /* Checkpoint / restart functionality */

    lam_ssi_rpi_restart_fn_t lsra_interrupt;
    lam_ssi_rpi_checkpoint_fn_t lsra_checkpoint;
    lam_ssi_rpi_continue_fn_t lsra_continue;
    lam_ssi_rpi_restart_fn_t lsra_restart;

    /* Flags */

    int lsra_tv_queue_support;
} lam_ssi_rpi_actions_1_1_0_t;

```

Figure C.11. `struct lam_ssi_rpi_actions_1_1_0`: The `rpi` type for exporting API function pointers.

### C.2.3 Function Call: `lsr_query`

- Type: `lam_ssi_rpi_query_fn_t`

```
typedef int (*lam_ssi_rpi_query_fn_t)
(int *priority, int *thread_min, int *thread_max);
```

- Arguments:
  - OUT: `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
  - OUT: `thread_min` is the minimum MPI thread level that this module supports.
  - OUT: `thread_max` is the maximum MPI thread level that this module supports.
- Return value: 1 if the module wants to be considered for selection, 0 otherwise.

- Description: This function determines whether a module wants to be considered for selection or not. It can invoke whatever initialization functions that it needs to determine whether it can run or not. If this module is not selected, its `lsr_finalize()` function will be invoked shortly after this function.

Additionally, the module must fill in `thread_min` and `thread_max` to be the minimum and maximum MPI thread levels that it supports. `thread_min` must be less than or equal to `thread_max`. See [126] for more details on the priority system and how modules are selected at run time.

If the module does not want to be considered during the negotiation for this application, it should return 0 (the values in `priority`, `thread_min`, and `thread_max` are then ignored).

### C.2.4 Function Call: `lsr_init`

- Type: `lam_ssi_rpi_init_fn_t`

```
typedef lam_ssi_rpi_actions_t (*lam_ssi_rpi_init_fn_t)
(struct _proc **procs, int nprocs, int *maxtag, int *maxcid);
```

- Arguments:
  - IN: `procs` is an array of pointers to the initial set of `struct _proc` instances that this `rpi` module is responsible for. The `procs` array will be freed after the call to `lsr_init()` completes; the `rpi` module is responsible saving its own copy.

- IN: `nprocs` is the length of the `procs` array.
  - OUT: `maxtag` is the maximum MPI tag value that this `rpi` module can handle. `*maxtag` will be the current max tag value when this function is invoked. The `rpi` module may lower this value if necessary, but may *not* raise it!
  - OUT: Similar to `maxtag`, `maxcid` is the maximum number of communicators that this `rpi` module can handle (i.e., the maximum communicator CID). `*maxcid` will be the current maximum CID value when this function is invoked. The `rpi` module may lower this value, but it may *not* raise it!
- Return Value: A pointer to the `struct` shown in Figure C.11. If the module returns `NULL`, an error will occur, because negotiation is over and this module has been selected.
  - Description: Performs primary initialization of the `rpi` module (called from `MPI_INIT`) after the `rpi` module negotiation; it will only be called in the selected module. This function typically performs once-only initialization of the communication sub-layer and initialize all processes with respect to the communication sub-layer. The latter may simply involve a call to `lsra_addprocs()` to initialize the initial set of “new” processes (even though this is the first set of processes that the `rpi` module will receive).

The `tcp` `rpi` module, for example, initializes a hash table for message buffering and then calls `lsra_addprocs()` to save the `procs` array and set up the TCP socket connections between the initial processes.

At the time of this call, the MPI process is also a LAM process, hence all LAM functionality is available to it. In particular the LAM message passing routines `nsend(2)` and `nrecv(2)` (see the LAM documentation and manual pages for more details) are available and can be used to pass out-of-band information between the MPI processes.<sup>2</sup> The `tcp` `rpi` module uses these functions to pass server socket port numbers to clients who must connect. See the function `connect_all()` in `share/ssi/rpi/tcp/src/ssi_rpi_tcp.c`.

Finally, the `rpi` module may lower the current maximum MPI tag and CID values. The final values used will be the maximum over all `rpi` modules that are used in an MPI process. Hence, an `rpi` module may *lower* these values, but the `rpi` module *may not increase them!*

### C.2.5 Function Call: `lsra_addprocs`

- Type: `lam_ssi_rpi_addprocs_fn_t`

```
typedef int (*lam_ssi_rpi_addprocs_fn_t)(struct _proc **procs, int nprocs);
```

- Arguments:

---

<sup>2</sup>Remember that it is illegal for `rpi` modules to invoke MPI functions (e.g., `MPI_SEND`, `MPI_RECV`).

- `procs`: An array of pointers to *new* `struct _proc` instances that this `rpi` module is responsible for. The `procs` array will be freed after the call to `lsr_init()` completes; the `rpi` module is responsible saving its own copy.
- `nprocs`: Length of the `procs` array.

- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: In LAM/MPI, a process can become aware of new processes with which it may communicate. For example, when it spawns MPI children. The MPI layer adds new process entries to the process list and then calls `lsra_addprocs()` to perform `rpi` module initialization with a list of *only* these new processes. The `rpi` module is responsible for augmenting its own internal list with the contents of the `proc` array.

The `tcp rpi` module, for example, adds the contents of the `proc` array to its internal list and then sets up the TCP socket connections between the new processes and the old ones.

This function is called from `MPI_INTERCOM_CREATE`, `MPI_COMM_SPAWN`, `MPI_COMM_SPAWN_MULTIPLE`, `MPI_COMM_ACCEPT`, `MPI_COMM_CONNECT`, and `MPI_COMM_JOIN`.

It is important to allow `lsra_addprocs()` to fail gracefully; do not use network protocols during setup that may deadlock or “hang” in the event of a failure. If so, commands such as `mpirun` and functions such as `MPI_COMM_SPAWN` may never complete.

### C.2.6 Function Call: `lsra_finalize`

- Type: `lam_ssi_rpi_finalize_fn_t`

```
typedef int (*lam_ssi_rpi_finalize_fn_t)(struct _proc *proc);
```

- Arguments:
  - IN/OUT: `proc` is the `_proc` to shut down, or `NULL` if the entire `rpi` module is to be shut down. This function should really only modify the flags in `p_mode` and nothing else in the non-`rpi`-specific portion of `proc`.
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: Performs final cleanup of a given `_proc` instance and/or the over `rpi` module (e.g., clean up all data structures, etc., created by the `rpi` module). This function is called from `MPI_FINALIZE` after all pending communication has completed. It is always called at least once, with (`proc == NULL`).

When `lsra_finalize()` is invoked with (`p != NULL`), it is the `rpi` module’s responsibility to never reference that process again, even when `lsra_finalize()` is invoked with (`p == NULL`).

If MPI-2 dynamic functions were invoked during the program's run, `lsra_finalize()` may be invoked multiple times with (`proc != NULL`) for the `_proc` instances that are not part of `MPI_COMM_WORLD`. Note that this may even happen *before* `MPI_FINALIZE` is invoked. For example, if processes are added by an MPI-2 dynamic function (e.g., `MPI_COMM_SPAWN`), but then later all communicators containing the spawned processes are freed via `MPI_COMM_FREE`, then `lsra_finalize()` will be invoked for each process that is no longer referenced.

The last invocation of `lsra_finalize()` is always with (`proc == NULL`), regardless of whether MPI-2 dynamic functions were used or not.

### C.2.7 Function Call: `lsra_build`

- Type: `lam_ssi_rpi_build_fn_t`

```
typedef int (*lam_ssi_rpi_build_fn_t)(MPI_Request req);
```

- Arguments:
  - IN/OUT: `req` is the request to build.
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: When the MPI layer creates a new request, it initializes general request information and then calls this function to build the `rpi`-specific portion of the request. Certain `rpi` module state, especially that which is unchanged over multiples invocations of a persistent operation, may be initialized here too. This function is called from `_mpi_req_build()`.

This step is separated from the “start” phase in order to optimize persistent MPI communication – “build” only needs to occur once, while “start” may be invoked many times.

### C.2.8 Function Call: `lsra_start`

- Type: `lam_ssi_rpi_start_fn_t`

```
typedef int (*lam_ssi_rpi_start_fn_t)(MPI_Request req_top, MPI_Request req);
```

- Arguments:
  - IN: `req_top` is the top of the active list.
  - IN/OUT: `req` is the request to be started.

- Return Value: Zero on success, LAMERROR otherwise.
- Description: The MPI layer, after adding a request to the progression list, calls `_mpi_req_start()` to make it ready for subsequent progression. Among other things, it moves the request's state to the start state and then calls `lsra_start()` so that the `rpi` module can do any initialization it needs to make the request ready.

This step is separated from the “build” phase in order to optimize persistent MPI communication – “build” only needs to occur once, while “start” may be invoked many times.

This function may also perform some progression past the start state (this is really the only reason that `req_top` is passed in). For example, an `rpi` module needs to also handle the special case of a process sending to or receiving from itself here, and may thus actually advance a request all the way to the done state.

If any further progression is done, the request's state must be updated to reflect this. The possible states after the start state are:

1. The active state: where the data transfer protocol is not yet finished but we have done some transfer and are past the point of cancellation, and
2. The done state: where the data transfer protocol is finished and the request can be completed.

### C.2.9 Function Call: `lsra_advance`

- Type: `lam_ssi_rpi_advance_fn_t`

```
typedef int (*lam_ssi_rpi_advance_fn_t)(MPI_Request req_top, int fl_block);
```

- Arguments:
  - IN/OUT: `req_top` is the first request that is in the active list.
  - IN: `fl_block` is 1 if `lsra_advance()` is allowed to block, or 0 if `lsra_advance()` must not block.
- Return Value: 1 if any requests' state has changed, 0 if none have changed state, or LAMERROR if an error occurred.
- Description: This is where most of the work gets done. Given a pointer to the top of the progression list, advance them where possible. The flag `fl_block` is true if it is permitted to block until progress is made on at least one request.

The MPI layer knows and cares nothing about message transfer protocols and message buffering. This is solely the responsibility of the `rpi` module. The `rpi` module however must update the state of the request as it progresses from `init`, to `start`, to `active`, and finally to `done`, so that the MPI layer can do the Right Things.

Note that a request may be moved from the start to the done state outside of the regular rpi progression by being canceled. The progression function `lsra_advance()` must take this into account. Currently, LAM does not allow the cancellation of requests which are in the active state.

The rpi module must also update other information in requests where appropriate.

#### C.2.10 Function Call: `lsra_destroy`

- Type: `lam_ssi_rpi_destroy_fn_t`

```
typedef int (*lam_ssi_rpi_destroy_fn_t)(MPI_Request req);
```

- Arguments:
  - IN/OUT: `req` is the request to destroy.
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: Destroys only the rpi portion of request. It is called from `_mpi_req_destroy()`. This function should free any dynamic storage created for this request by the rpi module and also perform any other necessary cleanup.

**Note:** it is only necessary to clean up what was created/done in other parts of the rpi module. The rest of the request will be cleaned up by the MPI layer itself.

#### C.2.11 Function Call: `lsra_iprobe`

- Type: `lam_ssi_rpi_iprobe_fn_t`

```
typedef int (*lam_ssi_rpi_iprobe_fn_t)(MPI_Request req);
```

- Arguments:
  - IN/OUT: `req` is the request to check for.
- Return Value: 0 if no match was found, 1 if a match was found, or `LAMERROR` if an error occurred.
- Description: Implements the strange non-blocking probe beast. It is called from `MPI_IPROBE` and is passed a non-blocking probe request which has been built and started. This function should check for matches for the probe in a non-blocking fashion and then return a value of 0 if no match was found, 1 if a match was found or `LAMERROR` if an error occurred.

In the case of a match, the MPI status in the request must also be updated as required by the definition of `MPI_IPROBE`.

This is such a strange function that the generalized code in the `tcp rpi` may be sufficient for other `rpi` modules.

### C.2.12 Function Call: `lsra_fastrecv`

- Type: `lam_ssi_rpi_fastrecv_fn_t`

```
typedef int (*lam_ssi_rpi_fastrecv_fn_t)
(char *buf, int count, MPI_Datatype type, int src, int *tag,
MPI_Comm comm, MPI_Status *status, int *seqnum);
```

- Arguments:
  - IN: `buf` is a pointer to the buffer to receive the incoming message in. It corresponds to the buffer argument in the invoking MPI receive call.
  - IN: `count` is the number of elements to receive. It corresponds to the count argument in invoking the MPI receive call.
  - IN: `type` is the MPI datatype of the element(s) to receive. It corresponds to the datatype argument in the invoking MPI receive call.
  - IN: `src` is the source rank to receive from. As described below, it will not be `MPI_ANY_SOURCE`. `src` corresponds to the source rank argument in the invoking MPI receive call.
  - IN/OUT: `tag` is the tag to use. It corresponds to the tag argument in the invoking MPI receive call. Upon return, it must be set to the tag that was actually used. Note that the tag must be set in the case of `MPI_ANY_TAG` because the `status` argument may be `MPI_STATUS_IGNORE`, and the actual tag would otherwise be lost.
  - IN: `comm` is the communicator to receive in. It corresponds to the communicator argument in the invoking MPI receive call.
  - IN/OUT: `status` is the status that must be filled when this function returns, or the special constant `MPI_STATUS_IGNORE`. It corresponds to the status argument from the invoking MPI receive call.
  - IN/OUT: `seqnum` is the sequence number of the incoming message from the sender's perspective. It is used for matching messages in trace files.
- Return Value: `MPI_SUCCESS` on success, or `LAMERROR` on error.
- Description: Like `lsra_fastsend()`, this function is intended to bypass the normal `rpi` progression mechanism, and is only called from `MPI_RECV` if there are no other active requests and the source of the message is neither `MPI_ANY_SOURCE` nor the destination. If a matching message has already arrived (and assumedly been buffered somewhere), it can just fill in the relevant values and return `MPI_SUCCESS`. If the message has not already arrived, it can block waiting for the message (since no other requests are active).

### C.2.13 Function Call: `lsra_fastsend`

- Type: `lam_ssi_rpi_fastsend_fn_t`

```
typedef int (*lam_ssi_rpi_fastsend_fn_t)
(char *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);
```

- Arguments:
  - **IN:** `buf` is a pointer to the buffer containing bytes to send. It corresponds to the buffer argument in the invoking MPI send call.
  - **IN:** `count` is the number of elements to send. It corresponds to the count argument in invoking the MPI send call.
  - **IN:** `type` is the MPI datatype of the element(s) to send. It corresponds to the datatype argument in the invoking MPI send call.
  - **IN:** `dest` is the destination rank to send to. It corresponds to the destination rank argument in the invoking MPI send call.
  - **IN:** `tag` is the tag to use.
  - **IN:** `comm` is the communicator to send in. It corresponds to the communicator argument in the invoking MPI send call.
- Return Value: `MPI_SUCCESS` on success, or `LAMERROR` on error.
- Description: This is a special case “short circuit” fast send. It was originally an experiment to optimize common sends and receives, but has proved to be a stable and efficient method of bypassing much of the request mechanism (and therefore, avoiding overhead).

This function is a fast blocking send; it takes all the same arguments as `MPI_SEND`. It is only invoked from blocking sends when there are no active requests in the `rpi` module and the destination is not the same as the source. In this case, it is safe to bypass the normal `rpi` progression mechanism and send the message immediately. This function is allowed to block if necessary (since no other requests are active). No request is created, so the send must be completed (in terms of the MPI layer) when the function returns. It must return `MPI_SUCCESS` or an appropriate error code.

### C.2.14 Function Call: `lsra_alloc_mem`

- Type: `lam_ssi_rpi_alloc_mem_fn_t`

```
typedef int (*lam_ssi_rpi_alloc_mem_fn_t)
(MPI_Aint size, MPI_Info info, void *baseptr);
```

- Arguments:
  - IN: `size` is the number of bytes to be allocated.
  - IN: `info` is any “hint” information passed in from `MPI_ALLOC_MEM`.
  - OUT: `baseptr`, as described in MPI-2:4.11, this is an OUTvariable, but is `(void *)` for convenience. Hence, it is actually a pointer to the actual pointer that will be filled. You can think of it as a `(void **)`, even though it really isn’t.
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: This function is used by the `rpi` module to allocate “special” memory that can be used for fast message passing (such as pinned memory for a Myrinet or VIA implementation). This function is invoked as the back-end of `MPI_ALLOC_MEM`.

If the `rpi` module does not need “special” memory for any reason, the function `lam_ssi_rpi_base_alloc_mem()` can be used as the value of this pointer instead (see Section C.1.12), which is mainly a wrapper around `malloc(3)`.

#### C.2.15 Function Call: `lsra_free_mem`

- Type: `lam_ssi_rpi_free_mem_fn_t`

```
typedef int (*lam_ssi_rpi_free_mem_fn_t)(void *baseptr);
```

- Arguments:
  - IN: `baseptr` is a pointer to the memory to be freed. It should be a value that was previously returned from `lsra_alloc_mem()`.
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: This function is used by the `rpi` module to deallocate “special” memory that was previously allocated by `lsra_alloc_mem()`. This function is invoked as the back-end of `MPI_FREE_MEM`.

If the `rpi` module does not need “special” memory for any reason, the function `lam_ssi_rpi_base_free_mem()` can be used as the value of this pointer instead (see Section C.1.13), which is mainly a wrapper around `free(3)`.

#### C.2.16 Function Call: `lsra_interrupt`

- Type: `lam_ssi_rpi_interrupt_fn_t`

```
typedef int (*lam_ssi_rpi_interrupt_fn_t)(void);
```

- Arguments: None
- Return Value: Zero on success, LAMERROR otherwise.
- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected `cr` module when a checkpoint is invoked from a separate, `cr`-specific thread.

The purpose of this function is to do whatever is necessary to let the main application thread know that a checkpoint is about to occur. Some `rpi` module implementations may block, and therefore need to be interrupted. Other implementations may not block, but still need to be asynchronously notified that the `cr` thread is waiting for the application thread to yield.

The mechanism to interrupt the application thread is specific to the `rpi` module. It cannot return until the application thread has yielded to the `cr` thread.

If the `rpi` module does not support checkpoint/restart functionality, it should provide `NULL` for this function pointer.

#### C.2.17 Function Call: `lsra_checkpoint`

- Type: `lam_ssi_rpi_checkpoint_fn_t`

```
typedef int (*lam_ssi_rpi_checkpoint_fn_t)(void);
```

- Arguments: None
- Return Value: Zero on success, LAMERROR otherwise.
- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected `cr` module when a checkpoint is invoked.

The purpose of this function is to do whatever is necessary for the `rpi` module to ready itself for checkpoint. For example, it may drain the network of any “in-flight” messages. This function may use the LAM out-of-band communication mechanism to ensure that all “in-flight” MPI messages are received before the function returns.

Note that it may not be required to close all network connections. LAM’s checkpointing model entails taking a checkpoint and then continuing the job. Hence, one possible model for the checkpoint function is to quiesce the network and then return – leaving all network connections intact. Upon continue, no special actions are required – the network connections are already in place, and normal MPI message passing progression can continue. Upon restart, however, all the network connections will be stale, and will need to be closed or discarded, and then re-opened.

If the `rpi` module does not support checkpoint/restart functionality, it should provide `NULL` for this function pointer.

### C.2.18 Function Call: `lsra_continue`

- Type: `lam_ssi_rpi_continue_fn_t`

```
typedef int (*lam_ssi_rpi_continue_fn_t)(void);
```

- Arguments: None
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected `cr` module when a checkpoint has finished and the parallel application is continuing afterward.

The purpose of this function is to do whatever is necessary for the `rpi` module to be continue the job. Note that if the `rpi` module provides checkpoint/restart support, this function must be provided – even if it does nothing other than return 0.

If the `rpi` module does not support checkpoint/restart functionality, it should provide `NULL` for this function pointer.

### C.2.19 Function Call: `lsra_restart`

- Type: `lam_ssi_rpi_restart_fn_t`

```
typedef int (*lam_ssi_rpi_restart_fn_t)(void);
```

- Arguments: None
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected `C/R` module when a parallel process is restarted.

The purpose of this function is to do whatever is necessary to restart the `rpi` module and ready it for MPI communications. Since the process has just restarted, it is likely to have stale network connections; it is typically safest to close/discard all network connections and re-initiate them.

If the `rpi` module does not support checkpoint/restart functionality, it should provide `NULL` for this function pointer.

### C.2.20 Data Member: `lsra_tv_queue_support`

- Type: `int`

- Description: This flag is used by LAM to determine if the rpi module supports TotalView queue debugging or not. Currently, this means that unexpected messages use the interface described in Section [C.1.14](#) (page [239](#)).

If the rpi module uses the Section [C.1.14](#) interface, it should set this flag to 1. Otherwise, it should set it to 0.

## APPENDIX D

### MPI COLLECTIVE ALGORITHMS COMPONENT INTERFACE

The `coll` component framework is described in Chapter 6. This Appendix describes the technical details and requirements for `coll` modules [124]. Section D.1 discusses header files, types, global variables, and utility functions that are provided to all `coll` modules. Section D.2 details the module interface modules and functions.

#### D.1 Services Provided by the `coll` Component Framework

Several services are provided by the `coll` component framework that are available to all `coll` modules.

##### D.1.1 Header Files

The following header files must be included (in order) in all module source files that want to use any of the common `coll` component services described in this Appendix:

```
#include <lam-ssi.h>  
#include <lam-ssi-coll.h>
```

Both of these files are included in the same location in the LAM source tree: `share/-include`. Appendix A.6.1 describes how to provide appropriate preprocessor flags to include these files properly.

### D.1.2 Communication During Initialization

`coll` modules are initialized near the end of the construction of an MPI communicator. As such, point to point MPI communication *on that communicator* is possible. All point-to-point communication must be fully completed (e.g., no unmatched non-blocking communications) when the initialization is complete.

Limited use of communicator constructors and collectives are permitted during the selection process for a new communicator. See Section [D.2.2](#) for details.

### D.1.3 BLK\* Constants

LAM's BLK\* constants are used to identify which function is being used. These constants are generally in the form of "BLKMPI<FUNCTION>," and are located in the header file `<blktype.h>` The constants can be used as tags for MPI point-to-point functions for layered `coll` module implementations, or used to generate MPI exceptions marked as coming from a specific collective operation.

### D.1.4 Utility Function: `lam_mkcoll()`

```
void lam_mkcoll(MPI_comm comm)
```

Arguments:

- **IN/OUT:** `comm` is the communicator to change to collective context.

This function is used switch a communicator from point-to-point context to collective context. It should be invoked at the beginning of all layered collective implementations before any point-to-point communication is invoked. See Section [D.2.1](#) for more information on layered point-to-point implementations.

### D.1.5 Utility Function: `lam_mkpt()`

```
void lam_mkpt(MPI_comm comm)
```

Arguments:

- IN/OUT: `comm` is the communicator to change to point-to-point context.

This function is used switch a communicator from collective context to point-to-point context. It should be invoked at the end of all layered collective implementations that previously invoked `lam_mkcoll()`. See Section [D.2.1](#) for more information on layered point-to-point implementations.

#### D.1.6 Utility Function: `lam_err_comm()`

```
int lam_err_comm(MPI_comm, int errclass, int error, char *errmsg)
```

Arguments:

- IN: `comm` is the communicator to invoke the exception handler on.
- IN: `errclass` is the error class.
- IN: `error` is the error code.
- IN: `errmsg` is a text error message.

This function is used to invoke MPI exceptions on a communicator. The available MPI error classes are listed in `<mpi.h>`; the integer error codes are defined by the `coll` module. `errmsg` is a string error message that may be printed if LAM's default exception handlers are invoked. Figure [D.1](#) shows an example.

Always return the value from `lam_err_comm()`. If the MPI exception does not abort, its error code will be returned through the function's return value.

#### D.1.7 Datatype Accessor Functions

Although `coll` modules generally need not access individual data members in user-provided buffers, `coll` modules may need to allocate temporary buffers and copy user-provided buffers while performing collective algorithms. LAM provides functions for allocating and copying buffers that are sized and mapped by MPI datatypes.

```

int lam_ssi_coll_<module>_barrier(MPI_Comm comm) {
    /* ... perform the barrier ... */

    if (an_error_occurred) {
        return lam_err_comm(comm, MPI_ERR_COMM, MPI_ERR_OTHER,
            "Error in barrier!");
    }
    return MPI_SUCCESS;
}

```

Figure D.1. Sample MPI\_BARRIER implementation.

Note that reduction operations seem like a notable exception to this policy; a reduction must potentially combine multiple buffers into a final output buffer, and would therefore seem to require access to the individual data members in the buffer. However, a separate function will be utilized for this purpose. Built-in MPI operations such as MPI\_SUM have their own function which will iterate over data members to perform the summation. User-provided operations will have user-specified functions to perform the combination operation. Specifically, the reduction function will be cached on the MPI\_Op passed to the reduction function. Figure D.2 shows an example.

As such, even with reductions, the coll module itself does not need to access the individual data members – it only needs to provide the infrastructure to copy and move buffers between processes. The following functions can be used to

- Allocate a buffer large enough to accommodate a specific MPI datatype.

```

int lam_dtbuffer(MPI_Datatype dtype, int count, char **buffer, char **origin);

```

dtype and count specify how large to make the buffer. Two pointers are returned: buffer is a pointer to the beginning of the allocated buffer (i.e., it can later be used as an argument to free()). origin is a pointer that can be passed as the buffer to MPI functions (e.g., MPI\_SEND, MPI\_RECV, etc.). buffer and origin may be different values if the lower bound of the MPI datatype is artificially set lower than its actual value.

- Copy a buffer mapped by two MPI datatypes.

```

int lam_ssi_coll_<module>_reduce(..., MPI_Op op, ...) {
    /* ... setup ... */

    /* Invoke the reduction function, checking to see if the MPI
       datatype can be passed directly, or whether we need to pass the
       integer handle, as required by Fortran */

    if (op->op_flags & LAM_LANGF77) {
        (op->op_func)(input_buffer, output_buffer, &count, &datatype->dt_f77handle);
    } else {
        (op->op_func)(input_buffer, output_buffer, &count, &datatype);
    }

    /* ... cleanup and final handling ... */
}

```

Figure D.2. Sample MPI\_REDUCE implementation. Note the Fortran flag on the `op` variable than indicates a different calling convention for the reduction function. In this case, the reduction function is in Fortran and therefore the datatype needs to be passed as its Fortran integer handle, not its C pointer handle.

```
int lam_dtsndrcv(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf,
int rcount, MPI_Datatype rdtype, int tag, MPI_Comm comm);
```

Typically, this function is used to copy from a user buffer to a temporary buffer allocated by `lam_dtbuffer` (or vice versa). Different datatypes may be used, as long as they match. If the datatype is relatively simple, the copy is performed directly. If the datatype is not simple, `MPI_SENDRECV` is used.

- Pack a buffer into a contiguous buffer, or unpack a buffer from a contiguous buffer. The functions `MPI_PACK` and `MPI_UNPACK` can be used directly for this purpose.

## D.2 coll Component Framework Module API

This is version 1.1.0 of the `coll` component framework module API. Each `coll` module must export a `struct lam_ssi_coll_1_0_0` named `lam_ssi_coll_<name>_module`. This type is defined in Figure D.3. This `struct` contains only a small number of function pointers used for initialization and querying the module. Figures D.4 and D.5 show a `struct` that contains function pointers to the collective API algorithm functions (it is split across two figures because of its length). A pointer to this `struct` is returned by the module's query function (in Figure D.3) if the module decides that it should be used for a given communicator.

The majority of the elements in Figures D.3, D.4, and D.5 are function pointer types; each is discussed in detail below. A module that returns `NULL` for any of the collective API function pointers will automatically use the corresponding `lam_basic` function pointer.

### D.2.1 Layered Point-to-Point Implementations

Collective algorithms may be layered on top of the MPI point-to-point functions (e.g., `MPI_SEND` and `MPI_RECV`). However, collective implementations must guarantee not to interfere with any pending point-to-point communications on the same communicator.

```

typedef struct lam_ssi_coll_1_1_0 {
    lam_ssi_1_0_0_t lsc_meta_info;

    /* Initialization / querying functions */

    lam_ssi_coll_thread_query_fn_t lsc_thread_query;
    lam_ssi_coll_query_fn_t lsc_query;

    /* Flags */

    int lsc_has_checkpoint;
} lam_ssi_coll_1_1_0_t;

```

Figure D.3. The coll type for exporting the basic API function pointers and flags.

LAM allows for this functionality by having a unique integer context on all communicators. Positive contexts are reserved for point-to-point communications; negative contexts are reserved for collective communications.

As such, layered collective implementations may either use the `lam_mkcoll()` and `lam_mkpt()` functions to change their communicator's context to be collective and then back to point-to-point (respectively), or they may use communicator constructors to make a private communicator with which to communicate. See Sections [D.1.4](#) and [D.1.5](#) for information about `lam_mkcoll()` and `lam_mkpt()`, respectively, and Section [D.2.2](#) for information about using sub-communicators.

It is critical that layered collectives guarantee to not leave any unmatched communications when they complete. Doing so will consume resources and potentially cause deadlock during the communicator destructor (which may be during `MPI_FINALIZE`).

Finally, layered algorithms should use the `PMPI` functions when available. This will guarantee to not create any side-effects from potentially user-intercepted `MPI` functions. However, LAM can be configured without the `MPI` profiling layer. `coll` modules must

```

typedef struct lam_ssi_coll_actions_1_1_0 {

    /* Per-communicator initialization and finalization functions */

    lam_ssi_coll_init_1_0_0_fn_t lsca_init;
    lam_ssi_coll_finalize_fn_t lsca_finalize;

    /* Checkpoint / restart functions */

    lam_ssi_coll_checkpoint_fn_t lsca_checkpoint;
    lam_ssi_coll_continue_fn_t lsca_continue;
    lam_ssi_coll_restart_fn_t lsca_restart;
    lam_ssi_coll_interrupt_fn_t lsca_interrupt;

    /* Collective function pointers */

    lam_ssi_coll_allgather_fn_t lsca_allgather_intra;
    lam_ssi_coll_allgather_fn_t lsca_allgather_inter;

    lam_ssi_coll_allgatherv_fn_t lsca_allgatherv_intra;
    lam_ssi_coll_allgatherv_fn_t lsca_allgatherv_inter;

    lam_ssi_coll_allreduce_fn_t lsca_allreduce_intra;
    lam_ssi_coll_allreduce_fn_t lsca_allreduce_inter;

    lam_ssi_coll_alltoall_fn_t lsca_alltoall_intra;
    lam_ssi_coll_alltoall_fn_t lsca_alltoall_inter;

    lam_ssi_coll_alltoallv_fn_t lsca_alltoallv_intra;
    lam_ssi_coll_alltoallv_fn_t lsca_alltoallv_inter;

    lam_ssi_coll_alltoallw_fn_t lsca_alltoallw_intra;
    lam_ssi_coll_alltoallw_fn_t lsca_alltoallw_inter;

    /* ...continued in next figure */

```

Figure D.4. The coll type for exporting the majority of the collective API function pointers (part 1 of 2).

```
/* ...continued from previous figure */

lam_ssi_coll_barrier_fn_t lsca_barrier_intra;
lam_ssi_coll_barrier_fn_t lsca_barrier_inter;

int lsca_bcast_optimization;
lam_ssi_coll_bcast_fn_t lsca_bcast_intra;
lam_ssi_coll_bcast_fn_t lsca_bcast_inter;

lam_ssi_coll_exscan_fn_t lsca_exscan_intra;
lam_ssi_coll_exscan_fn_t lsca_exscan_inter;

lam_ssi_coll_gather_fn_t lsca_gather_intra;
lam_ssi_coll_gather_fn_t lsca_gather_inter;

lam_ssi_coll_gatherv_fn_t lsca_gatherv_intra;
lam_ssi_coll_gatherv_fn_t lsca_gatherv_inter;

int lsca_reduce_optimization;
lam_ssi_coll_reduce_fn_t lsca_reduce_intra;
lam_ssi_coll_reduce_fn_t lsca_reduce_inter;

lam_ssi_coll_reduce_scatter_fn_t lsca_reduce_scatter_intra;
lam_ssi_coll_reduce_scatter_fn_t lsca_reduce_scatter_inter;

lam_ssi_coll_scan_fn_t lsca_scan_intra;
lam_ssi_coll_scan_fn_t lsca_scan_inter;

lam_ssi_coll_scatter_fn_t lsca_scatter_intra;
lam_ssi_coll_scatter_fn_t lsca_scatter_inter;

lam_ssi_coll_scatterv_fn_t lsca_scatterv_intra;
lam_ssi_coll_scatterv_fn_t lsca_scatterv_inter;
} lam_ssi_coll_actions_1_1_0_t;
```

Figure D.5. The `coll` type for exporting the majority of the collective API function pointers (part 2 of 2).

therefore utilize compile-time macros to switch between MPI and PMPI functions. Figure D.6 is an example of how this works.

### D.2.2 Hierarchical Implementations (Sub-Communicators)

It is permissible for `coll` module to create sub-communicators during the `lsca_init()` API function call (see Section D.2.10). Sub communicators may be useful to create “multi-module” collectives such that a top-level `coll` module is used to partition a communicator into smaller parts, each of which can invoke a dedicated collective algorithm (potentially in a different `coll` module).

For example, the `smp` `coll` module will invoke `MPI_COMM_SPLIT` on the communicator during its `lsca_init()` API function (Section D.2.10) to create a set of sub-communicators. Each sub-communicator will only contain MPI processes on a single node. In this way, the `smp` `coll` module runs a top-level algorithm for each collective, and the sub-communicators perform lower-level algorithms.

However, `MPI_COMM_SPLIT` invokes collectives to create new communications (namely `MPI_ALLREDUCE` and `MPI_ALLGATHER`). This creates a paradox: the `coll` module needs to invoke `MPI_COMM_SPLIT` while it is setting up, but `MPI_COMM_SPLIT` requires a communicator with a fully-functional `coll` module to complete successfully.

In order to enable this kind of behavior, modules can initially supply one set of function pointers for the MPI collective functions and later provide a second set of function pointers. Specifically, the `lam_basic` `coll` module is the “basic” implementation of the MPI collective algorithms. It uses standard linear/logarithmic algorithms which, although they perform well in many environments, are less than optimal in others. However, the `lam_basic` API functions can always be used without any prior setup. As such, in the `smp` module example above, `smp` initially sets up the target communicator with the function

```

/* <lam_config.h> will set LAM_WANT_PROFILE to 1 or 0, depending on
   whether LAM/MPI was built with profiling support or not. */
#include <lam_config.h>
#if LAM_WANT_PROFILE
#define LAM_PROFILELIB 1
#endif

/* If LAM_PROFILELIB is set to 1, <mpi.h> and <mpisys.h> will #define
   all MPI_Foo functions to be PMPI_Foo. */
#include <mpi.h>
#include <mpisys.h>

int lam_ssi_coll_<module>_bcast(..., int root, MPI_Comm comm) {
    /* Simplistic linear algorithm */

    if (rank == root) {
        for (i = 0; i < size; ++i) {
            if (i != root) {
                MPI_Send(..., comm);
            }
        }
    } else {
        MPI_Recv(..., comm, MPI_STATUS_IGNORE);
    }

    return MPI_SUCCESS;
}

```

Figure D.6. Sample code showing conditional profiling build.

```

int lam_ssi_coll_<module>_init(MPI_Comm comm,
    const lam_ssi_coll_actions_t **new_actions) {
    /* ... setup ... */

    MPI_Comm_split(comm, color, key, &sub_comm);
    sub_comm->c_flags |= C_HIDDEN;

    /* ... rest of function ... */
}

```

Figure D.7. Sample code creating a hidden communicator.

pointers from `lam_basic` before invoking `MPI_COMM_SPLIT`. Later, after all setup has been completed, `smp` replaces the `lam_basic` function pointers with its own.

For example:

1. During the creation of a new communicator, the `smp` module (which previously returned valid thread levels from its `lsc_thread_query()` function) has its `lsc_query()` function invoked.
2. It determines that it is eligible to run, and assigns an appropriate priority.
3. It returns a `lam_ssi_coll_actions_t` struct containing (essentially) all `NULL` function pointers (so that the `lam_basic` functions will be used during setup).
4. If the `smp` module is selected, its `lsc_init()` function is invoked.
5. `lsc_init()` invokes `MPI_COMM_SPLIT` to create sub communicators. `MPI_COMM_SPLIT` will use the `lam_basic` function pointers that were previously provided by the `lsc_query()` function to create the sub communicators.
6. Finally, `lsc_init` returns a new struct that contains pointers to all the `smp` module functions.

Also note that all sub-communicators created to implement this kind of behavior should set the `LAM_HIDDEN` flag on the `c_flags` member on the communicator.<sup>1</sup> This will hide the communicator from parallel debuggers that can view message queues. Figure D.7 shows an example.

<sup>1</sup>[125] contains details on the members of the `MPI_Comm` data structure.

### D.2.3 Intracommunicators and Intercommunicators

Figures D.4 and D.5 show two function pointers for each MPI collective function – one for intracommunicators, and one for intercommunicators. The function pointer type is the same for both functions (i.e., the function signature is identical). As such, each function below is only described once; it is implied that there are two pointers (of different names) in the `struct` – one for intracommunicators, and one for intercommunicators. LAM’s MPI layer will automatically invoke the appropriate function depending on the underlying type of `MPI_Comm` that was passed to the top-level MPI function.

### D.2.4 Checkpoint / Restart Functionality

LAM/MPI has the ability to involuntarily checkpoint and restart parallel MPI applications. The signal to checkpoint an MPI function may arrive asynchronously. Applications are typically unaware that checkpoints are occurring. LAM/MPI will ensure that either no user threads are in the MPI library, or that they have been safely interrupted. When an application resumes, the application thread(s) is(are) resumed as if nothing happened.

`coll` modules may or may not support checkpoint/restart functionality. If a module supports checkpoint/restart, it must set the `lsc_has_checkpoint` flag to 1 in its exported basic struct (see Figure D.3) and supply corresponding function pointers for all actions related to the checkpoint/restart functionality (described below). When an MPI application is initialized, the `coll` framework performs a logical AND on the `lsc_has_checkpoint` value from all available `coll` modules. Checkpoint/restart support will only be available if the result of this logical AND is 1.

#### General scheme

LAM’s general mechanism for checkpoint / restart is described in Chapter 7 and Appendix E. `coll` modules can insert arbitrary functions at checkpoint, continue, and restart

times.

Just like `rpi` modules, when a checkpoint begins, each `coll` module must get itself into a state such that it can be later restored with no loss of messages or information. For example, a `coll` module may drain the network such that no messages are “in-flight” on the network when the checkpoint finally occurs. However, this scheme may vary with each `coll` module; draining the network may or may not be necessary. Similarly, at continue and restart times, the `coll` module may need to re-establish network connections and reclaim external resources.

If a collective is implemented as a blocking operation, it may never see that a checkpoint is pending. The checkpoint/restart framework will invoke the module’s “interrupt” function to notify the module that a checkpoint is pending.

When no special action is needed.

Modules that are layered on point-to-point MPI functions probably do not need to do anything (at least in terms of the communication network) for checkpointing, continuing, and restarting since the `rpi` modules will take care of all networking aspects of checkpoint/restart functionality. Modules that fall into this category should either provide empty functions for this API calls, or use the built-in base `coll` empty functions that are designed for this purpose (the signatures of these functions are implied by their corresponding `coll` API functions, and are therefore not listed below):

- `lam_ssi_coll_base_empty_checkpoint()`
- `lam_ssi_coll_base_empty_continue()`
- `lam_ssi_coll_base_empty_restart()`
- `lam_ssi_coll_base_empty_interrupt()`

The no-op interrupt function is helpful for collective modules that never block. Other mechanisms are available to notify the MPI job that a checkpoint is pending; see [Chapter 7](#) and [Appendix E](#) for more information on the checkpoint/restart framework.

```

int lam_ssi_coll_<module>_barrier(MPI_Comm comm) {
    /* ... setup ... */

    if ((err = MPI_Send(...)) != MPI_SUCCESS) {
        return err;
    }

    /* ... rest of function ... */

    return MPI_SUCCESS;
}

```

Figure D.8. Sample MPI exception in a back-end collective implementation.

### D.2.5 MPI Exceptions and Return Values

If a `coll` API function that maps directly to an MPI collective function completes successfully, it must return `MPI_SUCCESS`. Errors that occur during `coll` module API functions should invoke the corresponding MPI exception handler.

With layered `coll` implementations, this is already mostly handled by the underlying MPI function calls. The `coll` module simply needs to check the return value from the underlying MPI function calls to ensure that they were `MPI_SUCCESS` before continuing. Figure D.8 shows an example.

Non-layered `coll` implementations will need to generate their own MPI exceptions (ensuring to return the value of the exception so that handlers such as `MPI_ERRORS_RETURN` can function properly). See Section D.1.6 for details on the `lam_err_comm()` function.

In the presence of errors, `coll` module authors are strongly encouraged to clean up state and leave the module in a restart-able state before invoking corresponding MPI exceptions. This will allow exception handlers such as `MPI_ERRORS_RETURN` to allow MPI programs to continue, even in the presence of non-fatal errors.

## D.2.6 Data Member: `lsc_meta_info`

`lsc_meta_info` is the SSI-mandated element contains meta-information about the module. See Section [A.6.2](#) for more information about this element.

## D.2.7 Function Call: `lsc_thread_query`

- Type: `lam_ssi_coll_thread_query_fn_t`

```
typedef int (*lam_ssi_coll_thread_query_fn_t)
(int *thread_min, int *thread_max);
```

- Arguments:
  - OUT: `thread_min` is the minimum MPI thread level that this module supports. Only meaningful if the function returns zero.
  - OUT: `thread_max` is the maximum MPI thread level that this module supports. Only meaningful if the function returns zero.
- Return value: Zero if the module wants to be considered for negotiation, non-zero otherwise.
- Description: This function is invoked exactly once during the initial module selection process during MPI initialization. It is invoked before all other module API calls.

If the module wants to be considered for negotiation, it must fill in `thread_min` and `thread_max` to be the minimum and maximum MPI thread levels that it supports. `thread_min` must be less than or equal to `thread_max`. This functionality is split from the main query API call (Section [D.2.8](#)) because the thread level of the module is determined only once per process, but the query function will be invoked multiple times.

If the function returns non-zero, the thread levels will be ignored and the module will be ignored for the duration of the process.

## D.2.8 Function Call: `lsc_query`

- Type: `lam_ssi_coll_query_fn_t`

```
typedef const lam_ssi_coll_actions_t (*lam_ssi_coll_query_fn_t)
(MPI_Comm comm, int *priority);
```

- Arguments:
  - IN: `comm` is the communicator that is being setup.
  - OUT: `priority` is the output priority that this module thinks that it should be rated. This value is only meaningful if the function returns a non-NULL value.
- Return value: Either NULL or a pointer to the struct shown in Figures D.4 and D.5. The contents of the struct will be copied – the struct itself will not be freed.
- Description: If the module wants to be considered for the negotiation of collectives on this communicator, it should return a pointer to the struct shown in Figures D.4 and D.5 and assign an associated priority to `priority`. Valid values of `priority` are in the range  $[0, 100]$ , with 0 being the lowest priority, and 100 being the highest.
 

If the module does not want to be considered during the negotiation for this communicator, it should return NULL, and the value in `priority` is ignored.

#### D.2.9 Data Member: `lsc_has_checkpoint`

`lsc_has_checkpoint` is set to 1 if this module supports checkpoint/restart functionality, and 0 otherwise. Its value is checked during `MPI_INIT`.

#### D.2.10 Function Call: `lsca_init`

- Type: `lam_ssi_coll_init_fn_t`

```
typedef int (*lam_ssi_coll_init_fn_t)
(MPI_Comm comm, const lam_ssi_coll_actions_t **new_actions);
```

- Arguments:
  - IN: `comm` is the communicator that this module has been selected for.
  - OUT: `new_actions` is a pointer to a new struct of function pointers for API calls. If the module wishes to change the values that it returned from `lsc_query`, it may fill `new_actions` with the pointer to a new/updated struct that will be used for all future API calls on this communicator. If this argument is set to NULL (or not assigned), the original struct returned from `lsc_query` will be used.
- Return value: Zero on success, nonzero otherwise.

- Description: The `lsca_init` function is invoked on the selected module only. Its purpose is to allow the selected module to set itself up for normal operation on the given communicator. The intent is to setup as much as possible during this function in order to facilitate the operation of all the collective calls at run time (i.e., overhead time is better spent during the initialization phase for each communicator rather than the actual collective call).

Note that this function may make limited use of communicator constructors and collectives. See Section [D.2.2](#) for details.

#### D.2.11 Function Call: `lsca_finalize`

- Type: `lam_ssi_coll_finalize_fn_t`

```
typedef int (*lam_ssi_coll_finalize_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being finalized.
- Return value: Zero on success, nonzero otherwise.
- Description: Finalize the use of this module. It is the last function to be called in the scope of this module's selection. It should release any resources allocated during the life of this scope. This function is only invoked on selected modules.

Note that this function is invoked when the communicator is freed (e.g., during `MPI_COMM_FREE`). Hence, this function must be a *local* action – it cannot involve communication with other MPI processes.

#### D.2.12 Function Call: `lsca_checkpoint`

- Type: `lam_ssi_coll_checkpoint_fn_t`

```
typedef int (*lam_ssi_coll_checkpoint_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being checkpointed.
- Return value: `MPI_SUCCESS` on success, nonzero otherwise.
- Description: Set this communicator to a state where it can be checkpointed. This typically means draining the network such that no collective messages are “in flight”, but the exact definition may vary from module to module.

This function will be invoked once per checkpoint for every communicator that is active in an MPI application. Hence, it may be invoked multiple times in a module for a single checkpoint, but each time with a different communicator argument.

coll modules that require no special action at checkpoint time should supply the value `lam_ssi_coll_base_checkpoint`.

#### D.2.13 Function Call: `lsca_continue`

- Type: `lam_ssi_coll_continue_fn_t`

```
typedef int (*lam_ssi_coll_continue_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being checkpointed.
- Return value: `MPI_SUCCESS` on success, nonzero otherwise.
- Description: After the checkpoint has successfully completed, this function will be invoked if the application continues without interruption. It may be used to do whatever the module needs to continue operation after a successful checkpoint.

This function will be invoked once per checkpoint for every communicator that is active in an MPI application. Hence, it may be invoked multiple times in a module for a single checkpoint, but each time with a different communicator argument.

coll modules that require no special action at continue time should supply the value `lam_ssi_coll_base_continue`.

#### D.2.14 Function Call: `lsca_restart`

- Type: `lam_ssi_coll_restart_fn_t`

```
typedef int (*lam_ssi_coll_restart_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being checkpointed.
- Return value: `MPI_SUCCESS` on success, nonzero otherwise.
- Description: When a parallel application is restored, this function is invoked. It may be used to close now-invalid communication channels (if they were not closed when the checkpoint occurred) and re-open them, or whatever else the coll module needs to resume operation.

This function will be invoked once per restore for every communicator that was active in an MPI application. Hence, it may be invoked multiple times in a module for a single restore, but each time with a different communicator argument.

coll modules that require no special action at continue time should supply the value `lam_ssi_coll_base_restart`.

#### D.2.15 Function Call: `lsca_interrupt`

- Type: `lam_ssi_coll_interrupt_fn_t`

```
typedef int (*lam_ssi_coll_interrupt_fn_t)(void);
```

- Arguments: None
- Return Value: Zero on success, `LAMERROR` otherwise.
- Description: This function is part of the checkpoint/restart functionality. It is invoked by the selected `cr` module when a checkpoint is invoked from a separate, `cr`-specific thread.

The purpose of this function is to do whatever is necessary to let the main application thread know that a checkpoint is about to occur. Some `coll` module implementations may block, and therefore need to be interrupted. Other implementations may not block, but still need to be asynchronously notified that the `cr` thread is waiting for the application thread to yield.

The mechanism to interrupt the application thread is specific to the `coll` module. It cannot return until the application thread has yielded to the `cr` thread.

If the `coll` module does not support checkpoint/restart functionality, it should provide `NULL` for this function pointer.

#### D.2.16 Function Call: `lsca_allgather`

- Type: `lam_ssi_coll_allgather_fn_t`

```
typedef int (*lam_ssi_coll_allgather_fn_t)  
  
(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount,  
  
MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for `MPI_ALLGATHER`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_ALLGATHER` collective.

#### D.2.17 Function Call: `lsca_allgatherv`

- Type: `lam_ssi_coll_allgatherv_fn_t`

```
typedef int (*lam_ssi_coll_allgatherv_fn_t)
    (void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int *rcounts,
     int *disps, MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for `MPI_ALLGATHERV`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_ALLGATHERV` collective.

#### D.2.18 Function Call: `lsca_allreduce`

- Type: `lam_ssi_coll_allreduce_fn_t`

```
typedef int (*lam_ssi_coll_allreduce_fn_t)
    (void *sbuf, void *rbuf, int count, MPI_Datatype dtype, MPI_Op op,
     MPI_Comm comm);
```

- Arguments: Same as for `MPI_ALLREDUCE`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_ALLREDUCE` collective.

#### D.2.19 Function Call: `lsca_alltoall`

- Type: `lam_ssi_coll_alltoall_fn_t`

```
typedef int (*lam_ssi_coll_alltoall_fn_t)
    (void *sbuf, int scount, MPI_Datatype sdtype, void* rbuf, int rcount,
     MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for `MPI_ALLTOALL`

- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_ALLTOALL` collective.

#### D.2.20 Function Call: `lsca_alltoallv`

- Type: `lam_ssi_coll_alltoallv_fn_t`

```
typedef int (*lam_ssi_coll_alltoallv_fn_t)
    (void *sbuf, int *scounts, int *sdisps, MPI_Datatype sdtype, void *rbuf,
     int *rcounts, int *rdisps, MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for `MPI_ALLTOALLV`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_ALLTOALLV` collective.

#### D.2.21 Function Call: `lsca_alltoallw`

- Type: `lam_ssi_coll_alltoallw_fn_t`

```
typedef int (*lam_ssi_coll_alltoallw_fn_t)
    (void *sbuf, int *scounts, int *sdisps, MPI_Datatype *sdtypes,
     void *rbuf, int *rcounts, int *rdisps, MPI_Datatype *rdtypes,
     MPI_Comm comm);
```

- Arguments: Same as for `MPI_ALLTOALLW`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_ALLTOALLW` collective.

LAM does not currently implement this function; this pointer is here for forward compatibility.

#### D.2.22 Function Call: `lsca_barrier`

- Type: `lam_ssi_coll_barrier_fn_t`

```
typedef int (*lam_ssi_coll_barrier_fn_t)(MPI_Comm comm);
```

- Arguments: Same as for `MPI_BARRIER`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_BARRIER` collective.

LAM's `MPI_BARRIER` will return `MPI_SUCCESS` immediately (and not call this API function) if there is only one member process in the communicator.

#### D.2.23 Data Member: `int lsca_bcast_optimization`

This flag should be 1 if `MPI_BCAST` is allowed to return `MPI_SUCCESS` immediately without invoking the underlying module broadcast function when there are zero data bytes to broadcast. A value of 0 means that the underlying module broadcast function will be invoked regardless of how many data bytes there are to broadcast.

#### D.2.24 Function Call: `lsca_bcast`

- Type: `lam_ssi_coll_bcast_fn_t`

```
typedef int (*lam_ssi_coll_bcast_fn_t)  
  
(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

- Arguments: Same as for `MPI_BCAST`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_BCAST` collective.

LAM's `MPI_BCAST` will return `MPI_SUCCESS` immediately (and not call this API function) if there is only one member process in the communicator, or if there are zero bytes to broadcast.

#### D.2.25 Function Call: `lsca_exscan`

- Type: `lam_ssi_coll_exscan_fn_t`

```
typedef int (*lam_ssi_coll_exscan_fn_t)
    (void *sbuf, void *rbuf, int count, MPI_Datatype dtype, MPI_Op op,
    MPI_Comm comm);
```

- Arguments: Same as for `MPI_EXSCAN`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_EXSCAN` collective.  
LAM does not currently implement this function; this pointer is here for forward compatibility.

#### D.2.26 Function Call: `lsca_gather`

- Type: `lam_ssi_coll_gather_fn_t`

```
typedef int (*lam_ssi_coll_gather_fn_t)
    (void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount,
    MPI_Datatype rdtype, int root, MPI_Comm comm);
```

- Arguments: Same as for `MPI_GATHER`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_GATHER` collective.

#### D.2.27 Function Call: `lsca_gatherv`

- Type: `lam_ssi_coll_gatherv_fn_t`

```
typedef int (*lam_ssi_coll_gatherv_fn_t)
    (void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int *rcounts,
    int *disps, MPI_Datatype rdtype, int root, MPI_Comm comm);
```

- Arguments: Same as for `MPI_GATHERV`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_GATHERV` collective.

#### D.2.28 Data Member: `int lsca_reduce_optimization`

This flag should be 1 if `MPI_REDUCE` is allowed to return `MPI_SUCCESS` immediately without invoking the underlying module reduce function when there are zero data bytes to reduce. A value of 0 means that the underlying module reduce function will be invoked regardless of how many data bytes there are to reduce.

#### D.2.29 Function Call: `lsca_reduce`

- Type: `lam_ssi_coll_reduce_fn_t`

```
typedef int (*lam_ssi_coll_reduce_fn_t)
(void *sbuf, void* rbuf, int count, MPI_Datatype dtype, MPI_Op op,
int root, MPI_Comm comm);
```

- Arguments: Same as for `MPI_REDUCE`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_REDUCE` collective.

LAM's `MPI_REDUCE` will return `MPI_SUCCESS` immediately (and not call this API function) if there are zero elements to reduce and `lsca_reduce_optimization` is set to 1.

#### D.2.30 Function Call: `lsca_reduce_scatter`

- Type: `lam_ssi_coll_reduce_scatter_fn_t`

```
typedef int (*lam_ssi_coll_reduce_scatter_fn_t)
(void *sbuf, void *rbuf, int *rcounts, MPI_Datatype dtype, MPI_Op op,
MPI_Comm comm);
```

- Arguments: Same as for `MPI_REDUCE_SCATTER`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_REDUCE_SCATTER` collective.

#### D.2.31 Function Call: `lsca_scan`

- Type: `lam_ssi_coll_scan_fn_t`

```
typedef int (*lam_ssi_coll_scan_fn_t)
    (void *sbuf, void *rbuf, int count, MPI_Datatype dtype, MPI_Op op,
    MPI_Comm comm);
```

- Arguments: Same as for `MPI_SCAN`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_SCAN` collective.

#### D.2.32 Function Call: `lsca_scatter`

- Type: `lam_ssi_coll_scatter_fn_t`

```
typedef int (*lam_ssi_coll_scatter_fn_t)
    (void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount,
    MPI_Datatype rdtype, int root, MPI_Comm comm);
```

- Arguments: Same as for `MPI_SCATTER`
- Return value: `MPI_SUCCESS` on success, or an appropriate error code otherwise.
- Description: Implement the `MPI_SCATTER` collective.

#### D.2.33 Function Call: `lsca_scatterv`

- Type: `lam_ssi_coll_scatterv_fn_t`

```
typedef int (*lam_ssi_coll_scatterv_fn_t)
    (void *sbuf, int *scounts, int *disps, MPI_Datatype sdtype, void* rbuf,
    int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm);
```

- Arguments: Same as for MPI\_SCATTERV
- Return value: MPI\_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI\_SCATTERV collective.

## APPENDIX E

### PARALLEL CHECKPOINT / RESTART COMPONENT INTERFACE

The `cr` component framework is described in Chapter 7. This Appendix describes the technical details and requirements for `cr` modules [115]. Section E.1 discusses header files, types, global variables, and utility functions that are provided to all `cr` modules. Sections E.2 and E.3 detail the module interface modules and functions for `crlam` and `crmpi`, respectively.

#### E.1 Services Provided by the `cr` Component Framework

Several services are provided by the `cr` component framework that are available to all `cr` modules.

##### E.1.1 Header Files

The following header files must be included (in order) in all module source files that want to use any of the common `cr` component services described in this document:

```
#include <lam-ssi.h>
#include <lam-ssi-cr.h>
```

Both of these files are included in the same location in the LAM source tree: `share/-include`. Appendix A.6.1 describes how to provide appropriate preprocessor flags to include these files properly.

```

typedef enum {
    LAM_SSI_CRMPI_BASE_HANDLER_STATE_IDLE,
    LAM_SSI_CRMPI_BASE_HANDLER_STATE_WAITING,
    LAM_SSI_CRMPI_BASE_HANDLER_STATE_RUNNING,

    LAM_SSI_CRMPI_BASE_STATE_MAX
} lam_ssi_crmpi_base_handler_state_t;

```

Figure E.1. The `lam_ssi_crmpi_base_handler_state_t` type and its possible values.

### E.1.2 Module Selection Mechanism

The scope `cr` modules persists through the life of an MPI application. It is selected during `MPI_INIT` and remains selected until `MPI_FINALIZE`. The MPI processes in the application collectively decide on which `crmpi` module to select and then pass its name to `mpirun`. `mpirun` then selects the corresponding `crlam` module (which may be none). Hence, `mpirun` and all of the MPI processes that it launched all share a common `cr` module selection.

### E.1.3 Internal Type: `lam_ssi_crmpi_base_handler_state_t`

This type is an enumerated value used to describe the state of the `crmpi` handler thread in an MPI process. See Figure [E.1](#).

The three possible values are:

- `HANDLER_STATE_IDLE`: The handler thread is inactive (and likely either non-existent or blocking, depending on the `crmpi` module's implementation).
- `HANDLER_STATE_WAITING`: The handler thread is waiting for the main application thread to either leave the MPI library, or yield control.
- `HANDLER_STATE_RUNNING`: The handler thread is actively working on a checkpoint, continue, or restart operation.

#### E.1.4 Global Variable: `lam_ssi_crmpi_base_handler_state`

This variable is available in `crmpi` only. It is of type `lam_ssi_crmpi_base_handler_state_t`, described in Section [E.1.3](#). This variable must be maintained by the `crmpi` handler thread to track its current status in order to notify other SSI modules that they are being interrupted.

```
volatile lam_ssi_crmpi_base_handler_state_t lam_ssi_crmpi_base_handler_state;
```

It may be necessary for the application thread(s) to perform some action based on the execution state of the thread-based checkpoint/restart handler. This variable is marked `volatile` so that it will not be cached by the application thread(s) if the `crmpi` handler thread changes its value.

#### E.1.5 `crlam` Utility Function: `lam_ssi_crlam_base_checkpoint()`

```
int lam_ssi_crlam_base_checkpoint(struct _gps *world, int nprocs);
```

This function must be invoked from the thread-based checkpoint/restart handler when `mpirun` receives a checkpoint request. After preparing `mpirun`<sup>1</sup> to be checkpointed, it will invoke the selected `crlam` module's `lscr_la_checkpoint()` API function (see Section [E.2.3](#)). The `world` array must contain a `struct _gps` element for each MPI process that needs to receive the checkpoint request. `nprocs` is the length of the `world` array.

It returns zero on success, `LAMERROR` otherwise.

#### E.1.6 `crlam` Utility Function: `lam_ssi_crlam_base_continue()`

```
int lam_ssi_crlam_base_continue(void);
```

<sup>1</sup>This function will likely also be used in `lamexec` if checkpointing of non-MPI programs is ever supported.

This function must be invoked from the thread-based checkpoint/restart handler when `mpirun` continues after a successful checkpoint. After preparing `mpirun` to continue after the checkpoint, it will invoke the selected `crlam` module's `lscrla_continue()` API function (see Section [E.2.4](#)).

It returns zero on success, `LAMERROR` otherwise.

E.1.7 `crlam` Utility Function: `lam_ssi_crlam_base_restart()`

```
int lam_ssi_crlam_base_restart(char *executable, char *app_schema);
```

This function must be invoked from the signal-based checkpoint/restart handler when `mpirun` restarts. After preparing `mpirun` to be checkpointed, it will invoke the selected `crlam` module's `lscrla_restart()` API function (see Section [E.2.4](#)). `executable` should be a string path to `mpirun` to re-launch, and `app_schema` should be the string path to the application schema that can be used to re-launch the MPI application.

It returns zero on success, `LAMERROR` otherwise.

E.1.8 `crlam` Utility Function: `lam_ssi_crlam_base_create_restart_argv()`

```
int lam_ssi_crlam_base_create_restart_argv(char **argv, OPT *ad);
```

Based on arguments `argv` and `ad`, this function creates an internal list of arguments that will be given to `mpirun` when it is restarted. This function is typically invoked by the `crlam`'s `lscra_init()` API function with the `argv` and `ad` parameters that it receives as arguments (see Section [E.2.8](#)).

The internal set of arguments that is created is not returned to the caller; it is used by `lam_ssi_crlam_base_do_exec()`.

It returns zero on success, `LAMERROR` otherwise.

E.1.9 `crlam` Utility Function: `lam_ssi_crlam_base_do_exec()`

```
int lam_ssi_crlam_base_do_exec(char *executable, char *app_schema);
```

The use of this function is optional; if it is not used, `cr` modules need to provide equivalent functionality.

This function can be either invoked by the `crlam` module's `lscr_restart()` API function call (see Section [E.2.9](#)), or is used as the `lscr_restart()` API function itself. The utility function `lam_crlam_base_create_restart_argv()` must have been invoked before this function is called.

This utility function launches a new `mpirun` process via `execve(2)` with the command line arguments that were previously setup, and an application schema that can re-launch the MPI job. Since this function `execve()`'s a new process, it never returns.

It returns zero on success, `LAMERROR` otherwise.

#### E.1.10 `crmpi` Utility Function: `lam_ssi_crmpi_base_checkpoint()`

```
int lam_ssi_crmpi_base_checkpoint(void);
```

This function should be invoked from the MPI thread-based checkpoint/restart handler when a checkpoint request arrives. `crmpi_base_checkpoint()` is used to prepare the other SSI modules in use by the MPI process to be checkpointed.

This function invokes the appropriate action function exported by each SSI module that needs pre-checkpoint handling. Currently, this means:

- The function `lsra_checkpoint()` `rpi` API function will be invoked on all active `rpi` SSI modules.
- The function `lsca_checkpoint()` `coll` API function will be invoked on each existing MPI communicator.

`crmpi_base_checkpoint()` will return 0 if all the invoked functions return 0, or `LAMERROR` if any of them returned `LAMERROR`.

#### E.1.11 `crmpi` Utility Function: `lam_ssi_crmpi_base_continue()`

```
int lam_ssi_crmpi_base_continue(void);
```

This function should be invoked from the MPI thread-based checkpoint/restart handler after a checkpoint completes successfully. `crmpi_base_continue()` is used to trigger post-checkpoint actions in other SSI modules in use by the MPI process after a checkpoint.

This function invokes the appropriate action function exported by each SSI module that needs pre-checkpoint handling. Currently, this means:

- The `lsra_continue()` `rpi` API function will be invoked on all active `rpi` SSI modules.
- The `lsca_continue()` `coll` API function will be invoked on each existing MPI communicator.

`crmpi_base_continue()` will return 0 if all the invoked functions return 0, or `LAMERROR` if any of them returned `LAMERROR`.

#### E.1.12 `crmpi` Utility Function: `lam_ssi_crmpi_base_restart()`

```
int lam_ssi_crmpi_base_restart(void);
```

This function should be invoked from the thread-based checkpoint/restart handler when an MPI process is restarted. `crmpi_base_restart()` is used to trigger pre-restart actions in other SSI modules in use by the MPI process when it is restarted.

This function invokes the appropriate action function exported by each SSI module that needs pre-restart handling. Currently, this means:

- The `lsra_restart()` `rpi` API function will be invoked on all active `rpi` SSI modules.
- The `lsca_restart()` `coll` API function will be invoked on each existing MPI communicator.

```

typedef struct lam_ssi_crlam_1_1_0 {
    lam_ssi_1_0_0_t lscrl_meta_info;

    /* crlam API function pointers */

    lam_ssi_crlam_query_fn_t lscrl_query;
} lam_ssi_crlam_1_1_0_t;

```

Figure E.2. `lam_ssi_crlam_1_1_0_t`: The `crlam` basic type for exporting the module meta information and initial query function pointer.

`crmpi_base_restart()` will return 0 if all the invoked functions return 0, or `LAMERROR` if any of them returned `LAMERROR`.

## E.2 `crlam` Component Framework Module API

This is version 1.1.0 of the `crlam` component framework module API. Each `crlam` module must export a `lam_ssi_crlam_1_0_0_t` named `lam_ssi_crlam_<name>_module`. This type is defined in Figure E.2. A second `struct` is used to hold the majority of function pointers and flags for the module. It is only used if the module is selected, and is shown in Figure E.3.

The majority of the elements in Figures E.2, and E.3 are function pointer types; each is discussed in detail below.

### E.2.1 Data Member: `lscrl_meta_info`

`lscrl_meta_info` is the SSI-mandated element and contains meta-information about the module. See Section A.6.2 for more information about this element.

### E.2.2 Function Call: `lscrl_query`

- Type: `lam_ssi_crlam_query_fn_t`

```

typedef struct lam_ssi_crlam_actions_1_1_0 {

    /* crlam action API functions pointers */

    lam_ssi_crlam_checkpoint_fn_t lscr_la_checkpoint;
    lam_ssi_crlam_continue_fn_t lscr_la_continue;
    lam_ssi_crlam_disable_checkpoint_fn_t lscr_la_disable_checkpoint;
    lam_ssi_crlam_enable_checkpoint_fn_t lscr_la_enable_checkpoint;
    lam_ssi_crlam_finalize_fn_t lscr_la_finalize;
    lam_ssi_crlam_init_fn_t lscr_la_init;
    lam_ssi_crlam_restart_fn_t lscr_la_restart;

    /* To be invoked by lamcheckpoint and lamrestart, respectively */

    lam_ssi_crlam_lamcheckpoint_fn_t lscr_la_lamcheckpoint;
    lam_ssi_crlam_lamrestart_fn_t lscr_la_lamrestart;
} lam_ssi_crlam_actions_1_1_0_t;

```

Figure E.3. `lam_ssi_crlam_actions_1_1_0_t`: The `crlam` type for exporting API function pointers.

```
typedef lam_ssi_crlam_actions_t>(*lam_ssi_crlam_query_fn_t)(int *priority);
```

- Arguments:
  - OUT: `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
- Return value: Either `NULL` or a pointer to the `struct` shown in Figure E.3.
- Description: If the module wants to be considered for selection, it should return a pointer to the `struct` shown in Figure E.3 that is filled with relevant data, and assign an associated priority to `priority`.

If the module does not want to be considered during the negotiation for this application, it should return `NULL` (the value in `priority` is then ignored).

### E.2.3 Function Call: `lscrla_checkpoint`

- Type: `lam_ssi_crlam_checkpoint_fn_t`

```
typedef int (*lam_ssi_crlam_checkpoint_fn_t)(void)
```

- Arguments: None.
- Return value: Zero on success, `LAMERROR` otherwise.
- Description: This function is invoked by the `lam_ssi_crlam_base_checkpoint()` utility function (which was, in turn, invoked by the thread-based handler when `mpirun` received the checkpoint request). The stated purpose of this function is to prepare `mpirun` and the MPI processes it launched for checkpoint. This typically means propagating the checkpoint request out to all MPI processes started by `mpirun` and creating an application schema suitable for using to restart the MPI application.

Note that `mpirun` itself will likely be blocking in the LAM function call `rpwait()` (“remote process wait”) while waiting for child processes to complete. The LAM infrastructure is currently not thread-safe, and therefore cannot handle any LAM daemon calls from the same process during this API call. If this function needs to use LAM infrastructure API calls (such as `nsend()/nrecv()`, `rploadgov()`, etc.), the recommended solution is to `fork()` from `mpirun`, register the new process as a LAM process, and invoke the necessary LAM API calls in the new process.

Although the actions performed by this function could be contained within the `mpirun` thread handler function itself, separating it out into a distinct function provided a clean abstraction for this specific functionality. It also provides for future compatibility when/if LAM ever provides the thread-based handler itself (instead of having that provided by the `crlam` module).

#### E.2.4 Function Call: `lscrla_continue`

- Type: `lam_ssi_crlam_continue_fn_t`

```
typedef int (*lam_ssi_crlam_continue_fn_t)(void)
```

- Arguments: None.
- Return value: Zero on success, `LAMERROR` otherwise.
- Description: This function is invoked by the `lam_ssi_crlam_base_continue()` utility function (which was, in turn, invoked by the thread-based handler after a successful checkpoint). The stated purpose of this function is to continue `mpirun` and the MPI processes it launched after a successful checkpoint. This may involve some recovery actions, or it may be an no-op.

The same restrictions apply to this function with regards to LAM API calls as with the `lscrla_checkpoint()` API call.

Although the actions performed by this function could be contained within the `mpirun` thread handler function itself, separating it out into a distinct function provided a clean abstraction for this specific functionality. It also provides for future compatibility when/if LAM ever provides the thread-based handler itself (instead of having that provided by the `crlam` module).

#### E.2.5 Function Call: `lscrla_disable_checkpoint`

- Type: `lam_ssi_crlam_disable_checkpoint_fn_t`

```
typedef void (*lam_ssi_crlam_disable_checkpoint_fn_t)(void)
```

- Arguments: None.
- Return value: None.
- Description: This function is called by `mpirun` to indicate when it is not permissible to allow checkpoints. Hence, if a checkpoint request arrives after `mpirun` has invoked this function, it must be stalled until a corresponding `lscrla_enable_checkpoint()` API call is invoked.

Note that `mpirun` is implicitly in a “disable” state after returning from `lscrla_init()`; checkpoint requests should always be deferred until at least the first invocation of `lscrla_enable_checkpoint()`.

### E.2.6 Function Call: `lscrla_enable_checkpoint`

- Type: `lam_ssi_crlam_enable_checkpoint_fn_t`

```
typedef void (*lam_ssi_crlam_enable_checkpoint_fn_t)(void)
```

- Arguments: None.
- Return value: None.
- Description: This function is called by `mpirun` to indicate when it is permissible to allow checkpoints. Hence, if a checkpoint request arrives after `mpirun` has invoked this function, both `mpirun` and the MPI job can be checkpointed.

Note that `mpirun` is implicitly in a “disable” state after returning from `lscrla_init()`; checkpoint requests should always be deferred until at least the first invocation of this function.

### E.2.7 Function Call: `lscrla_finalize`

- Type: `lam_ssi_crlam_finalize_fn_t`

```
typedef int (*lam_ssi_crlam_finalize_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, `LAMERROR` otherwise.
- Description: Performs the final cleanup of the checkpoint/restart handlers and possibly also invoke routines to detach from the underlying checkpointer.

### E.2.8 Function Call: `lscrla_init`

- Type: `lam_ssi_crlam_init_fn_t`

```
typedef int (*lam_ssi_crlam_init_fn_t)  
  
(char *path, char **argv, OPT *ad, struct _gps *mpiworld, int world_n);
```

- Arguments:
  - IN: `path` is the name of the executable to launch at restart time (e.g., “`mpirun`”).

- IN: `argv` is the list of arguments that were specified at the `mpirun` command-line, and is used to build a list of arguments to be passed to the new `mpirun` at restart.
  - IN: `ad` is used for option-parsing of the `argv` in order to build a restart-time `argv` that will be passed to `mpirun`.
  - IN: `mpiworl` is used to pass the GPS information about all the processes in the MPI job. This information is required to propagate the checkpoint requests from `mpirun` to all the MPI processes.
  - IN: `world_n` is used to pass the count of processes that are part of the MPI job. This information is required to propagate the checkpoint requests from `mpirun` to all the MPI processes.
- Return value: Zero on success, `LAMERROR` otherwise.
  - Description: Performs the primary initialization of the `crlam` sub-layer. This function typically registers the signal-based and thread-based callbacks to perform the actual checkpoint/restart functionality. Any initialization that is specific to the underlying checkpointing system is also performed here.

A minimum requirement for initiating a checkpoint of a LAM/MPI job is the delivery of a signal to `mpirun`. If the underlying checkpointer does not provide a mechanism to create/manage the thread-based checkpoint/restart handler, then a thread has to be explicitly created in this initialization function, and it will have to be blocked from executing (say, by waiting on `read` from a pipe). This thread could then be woken up to start executing when a checkpoint request comes in (say, by writing to the pipe on which the threaded callback is blocked on a `read()`, from signal handler-context).

### E.2.9 Function Call: `lscrla_restart`

- Type: `lam_ssi_crlam_restart_fn_t`

```
typedef int (*lam_ssi_crlam_restart_fn_t)(void)
```

- Arguments: None.
- Return value: Zero on success, `LAMERROR` otherwise.
- Description: This function is invoked by the `lam_ssi_crlam_base_restart()` utility function (which was, in turn, invoked by the signal-based handler upon restart). The stated purpose of this function is to restart `mpirun` with a new application schema so that it can restart the MPI application.

Note that this function runs in signal handler context, and is therefore subject to many restrictions (e.g., it cannot even call `malloc()`). Module authors are reminded to be extremely cautious about what this function does.

Although the actions performed by this function could be contained within the `mpirun` signal handler function itself, separating it out into a distinct function provided a clean abstraction for this specific functionality. It also provides for future compatibility when/if LAM ever provides the signal-based handler itself (instead of having that provided by the `crlam` module).

#### E.2.10 Function Call: `lscrla_lamcheckpoint`

- Type: `lam_ssi_crlam_lamcheckpoint_fn_t`

```
typedef int (*lam_ssi_crlam_lamcheckpoint_fn_t)(pid_t mpirun_pid)
```

- Arguments:
  - IN: `mpirun_pid` PID of `mpirun`.
- Return value: Zero on success, `LAMERROR` otherwise.
- Description: This function is invoked by the `lamcheckpoint` command with the PID of `mpirun`. It should initiate a checkpoint request and send it to `mpirun`, starting the checkpoint process.

The purpose of this function is to hide the mechanism used to checkpoint request mechanism.

#### E.2.11 Function Call: `lscrla_lamrestart`

- Type: `lam_ssi_crlam_lamrestart_fn_t`

```
typedef int (*lam_ssi_crlam_lamrestart_fn_t)(char *mpirun)
```

- Arguments:
  - IN: `mpirun` Location of `mpirun` executable.
- Return value: Zero on success, `LAMERROR` otherwise.
- Description: This function is invoked by the `lamrestart` command. It should initiate a restart of a previously-checkpointed MPI process. It is expected that any necessary information (e.g., restart filename) will be passed through module parameters. The path of `mpirun` is passed as an argument so that the module knows where the “right” `mpirun` command can be found (i.e., one that matches the installation of LAM/MPI).

The purpose of this function is to hide the mechanism used to restart processes.

```

typedef struct lam_ssi_crmpi_1_0_0 {
    lam_ssi_1_0_0_t lscrm_meta_info;

    /* cr mpi API function pointers */

    lam_ssi_crmpi_query_fn_t lscrm_query;
    lam_ssi_crmpi_init_fn_t lscrm_init;
} lam_ssi_crmpi_1_0_0_t;

```

Figure E.4. `struct lam_ssi_crmpi_1_0_0`: The `crmpi` basic type for exporting the module meta information and function pointers.

```

typedef struct lam_ssi_crmpi_actions_1_0_0 {

    /* cr mpi action API functions pointers */

    lam_ssi_crmpi_finalize_fn_t lscрма_finalize;
    lam_ssi_crmpi_app_suspend_fn_t lscрма_app_suspend;
} lam_ssi_crmpi_actions_1_0_0_t;

```

Figure E.5. `struct lam_ssi_crmpi_actions_1_0_0`: The `crmpi` type for exporting API function pointers.

### E.3 `crmpi` Component Framework Module API

This is version 1.0.0 of the `crmpi` component framework module API. Each `crmpi` module must export a `struct lam_ssi_crmpi_1_0_0` named `lam_ssi_crmpi-<name>.module`. This type is defined in Figure E.4. A second `struct` is used to hold the majority of function pointers and flags for the module. It is only used if the module is selected, and is shown in Figure E.5.

### E.3.1 Data Member: `lscrm_meta_info`

`lscrm_meta_info` is the SSI-mandated element and contains meta-information about the module. See Section [A.6.2](#) for more information about this element.

The data in this element should be the same as the data contained in the corresponding `crLAM` meta element (`lscrl_meta_info`, Section [E.2.1](#)).

### E.3.2 Function Call: `lscrm_query`

- Type: `lam_ssi_crmpi_query_fn_t`

```
typedef int (*lam_ssi_crmpi_query_fn_t)
(int *priority, int *thread_min, int *thread_max);
```

- Arguments:
  - OUT: `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
  - OUT: `thread_min` is the minimum MPI thread level that this module supports. Only meaningful if the function returns zero and a non-negative priority.
  - OUT: `thread_max` is the maximum MPI thread level that this module supports. Only meaningful if the function returns zero and a non-negative priority.
- Return value: Zero on success, `LAMERROR` otherwise.  
Either `NULL` or a pointer to the `struct` shown in [Figure E.5](#).
- Description: If the module wants to be considered for selection, it must return zero, assign an associated priority to `priority`, and fill `thread_min` and `thread_max` with the minimum<sup>2</sup> and maximum MPI thread levels that it can operate in.  
If the module does not want to be considered during the negotiation for this application, it should return `LAMERROR` (the values in `priority`, `thread_min`, and `thread_max` are then ignored).

### E.3.3 Function Call: `lscrma_init`

- Type: `lam_ssi_crmpi_init_fn_t`

```
typedef lam_ssi_crmpi_actions_t (*lam_ssi_crmpi_init_fn_t)(void);
```

<sup>2</sup>This value must be at least `MPI_THREAD_SERIALIZED`.

- Arguments: None.
- Return value: A pointer to the struct shown in Figure E.5, or NULL on error.
- Description: Performs the primary initialization of the `crmpi` module and returns a pointer to a `lam_ssi_crmpi_actions_t` filled with function pointers for the actions of this module.

This function typically registers signal-based and thread-based callbacks to perform the actual checkpoint/restart functionality (or launches a thread that sleeps until the appropriate time). Any initialization that is specific to the underlying checkpointing system is also performed here. Typically, one or more mutual exclusion devices might need to be initialized in this function that will be used to synchronize the execution of the main application thread and the checkpoint/restart handler thread within the MPI library.

A minimum requirement for initiating a checkpoint of a LAM/MPI job is the delivery of a signal to each process in the MPI application. If the underlying checkpoint/restart handler does not provide a mechanism to create/manage the thread-based checkpoint/restart handler, then a thread has to be explicitly created in this initialization function, and it will have to be blocked from executing (for example, by blocking on `read()` from a pipe). This thread could then be woken up to start executing when a checkpoint request arrives (for example, by writing to the pipe on which the threaded callback is blocked on a `read()` since `write()` is safe in signal handler context).

#### E.3.4 Function Call: `lscrma_finalize`

- Type: `lam_ssi_crmpi_finalize_fn_t`

```
typedef int (*lam_ssi_crmpi_finalize_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, non-zero otherwise.
- Description: Performs the final cleanup of the checkpoint/restart handlers, and possibly also invoke routines to detach from the underlying checkpointing system.

#### E.3.5 Function Call: `lscrma_app_suspend`

- Type: `lam_ssi_crmpi_app_suspend_fn_t`

```
typedef void (*lam_ssi_crmpi_app_suspend_fn_t)(void);
```

- Arguments: None.
- Return value: None.

- Description: This function is provided to allow the main application thread to yield to the checkpoint/restart thread when it is interrupted in the middle of a blocking MPI call by the threaded handler. This is typically done by using a set of one or more mutual exclusion devices. Once the threaded handler returns from a checkpoint, it will yield control back to the application thread and allow the `lsrma_app_suspend()` call to return.

## BIBLIOGRAPHY

- [1] Adnan Agbaria and Roy Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.
- [2] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings, 1999 Conference on High Performance Distributed Computing*, 1999.
- [3] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mitchel W. Sukalski, Mark A. Taylor, and Timothy S. Woodall. Architecture of la-mpi, a network-fault-tolerant mpi. In *Los Alamos report LA-UR-03-0939, to be published in Proceedings of IPDPS04*, 2004.
- [4] D. Bailey. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63–73, Fall 1991.
- [5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrisnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report TR RNR-094-007, NASA Ames Research Center, Moffett Field, CA, March 1994. 1994.
- [6] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, Aug 1991.
- [7] Roberto Baldoni, Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. Consistent checkpointing in message passing distributed systems. Technical Report P.I. 925, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, FRANCE, mai 1995. <ftp.irisa.fr>.
- [8] Kevin J. Barker. Personal communication, 2001. Personal communication.
- [9] Brian Barrett, Jeff Squyres, and Andrew Lumsdaine. *LAM/MPI Design Document*. Open Systems Laboratory, Pervasive Technology Labs, Indiana University, Bloomington, IN. See <http://www.lam-mpi.org/>.
- [10] Rajanikanth Batchu, Jothi Neelamegam, Zhenqian Cui, Murali Beddhu, Anthony Skjellum, Yoginder Dandass, and Manoj Apte. MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance - Portable Parallel Computing. In Rajkumar Buyya, George Mohay, and Paul Roe, editors, *1st IEEE International Symposium of Cluster Computing and the Grid*, pages 26–33, Los Alamitos, CA, May 2001. IEEE Computer Society.

- [11] Micah Beck, Jack J. Dongarra, Graham E. Fagg, G. Al Geist, Paul Gray, James Kohl, Mauro Migliardi, Keith Moore, Terry Moore, Philip Papadopoulos, Stephen L. Scott, , and Vaidy Sunderam. HARNESS: A next generation distributed virtual machine. *International Journal on Future Generation Computer Systems*, 15(5/6), 1999. Elsevier Publisher.
- [12] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and Debugging in a Heterogeneous Environment. *IEEE Computer*, 26(6):88–95, June 1993.
- [13] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Steve Otto, and Jon Walpole. PVM: Experiences, Current Status and Future Direction. In *Supercomputing'93 Proceedings*, pages 765–6, 1993.
- [14] G. D. Benson, C. Chu, Q. Huang, and S. G. Caglar. A comparison of mpich allgather algorithms on switched networks. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 335–343. Springer Verlag, 2003. The 10th European PVM/MPI User's Group Meeting, Venice Italy September/October 2003 Proceedings.
- [15] Bharat K. Bhargava and Shy-Renn. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems—an optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 3–12, October 1988.
- [16] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *SC'2002 Conference CD*, Baltimore, MD, 2002. IEEE/ACM SIGARCH. pap298,LRI.
- [17] Daniele Briatico, Augusto Ciuffoletti, and Luca Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the Fourth International Symposium on Reliability in Distributed Software and Databases*, pages 207–215, 1984.
- [18] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, , and Rolf Riesen. The portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratory, Sandia, NM, Nov 1999.
- [19] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.
- [20] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA, 1997.
- [21] S. G. Caglar, G. D. Benson, Q. Huang, and C. Chu. USFMPI: A multi-threaded implementation of MPI for linux clusters. In *Proceedings of the 15th International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, December 2003.

- [22] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147–158. ACM Press, 1992.
- [23] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.
- [24] Y. Z. Chang, K. S. Ding, and J. J. Tsay. Efficient Implementation of Message Passing Interface on Local Area Networks. In *1996 International Computer Symposium*, 1996.
- [25] D. Chappell. *Understanding ActiveX and OLE – A Guide for Developers and Managers*. Microsoft, Redmond, WA, 1996.
- [26] D. Chappell. COM+: The future of microsoft’s component object model. In *Notes on Information Technology*. Patricia Seybold Group, Boston, MA, Nov 1998.
- [27] Yuqun Chen, Kai Li, and James S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.
- [28] Flaviu Cristian. Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4:175–188, 1991.
- [29] Flaviu Cristian, Houtan Aghali, Ray Strong, and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206. IEEE Computer Society Press, 1985.
- [30] Chris DiBona, Sam Ockman, and Mark Stone, editors. *Open Sources*. O’Reilly and Associates, 1999.
- [31] William R. Dieter and James E. Lumpp Jr. A user-level checkpointing library for POSIX threads programs. In *Symposium on Fault-Tolerant Computing*, pages 224–227, 1999.
- [32] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–75, April 1993.
- [33] Jason Duell. The Design and Implementation of Berkeley Lab’s Linux Checkpoint/Restart, 2002.
- [34] Elmootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

- [35] Mootaz Elnozahy, Lorenzo Alvisi, Yi Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.
- [36] Richard Stallman et al. *GNU Coding Standards*. Free Software Foundation, October 2001. Included in the GNU Autoconf distribution; see <ftp://ftp.gnu.org/gnu/autoconf/>.
- [37] Graham E. Fagg, Antonin Bukovsky, and Jack J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27:1479–1496, 2001.
- [38] Graham E. Fagg, Edgar Gabriel, Zizhon Chen, Thara Angskun, George Bosilca, Antonin Bukovsky, , and Jack J. Dongarra. Fault tolerant communication library and applications for high performance. In *Proceedings, Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, October 2003.
- [39] E. W. Felton and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Scalable High Performance Computing Conference*, pages 84–89, 1992.
- [40] Michael J. Fisher, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
- [41] I. Foster and N. Karonis. A grid-enabled mpi: Message passing in heterogenous distributed computing systems. In *Proceedings of 1998 Supercomputing Conference*, 1998.
- [42] I. Foster and C. Kesselman. The globus project: A status report. In *Proceedings IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.
- [43] I. Foster and C. Kessleman. Globus: A metacomputing infrastructure toolkit. *International Journal on Supercomputing Applications*, 11(2):115–128, 1997.
- [44] Ian Foster. What is the grid? A three point checklist. In *GRIDToday*. Tabor Communications, July 20 2002.
- [45] Michael Franz. Dynamic Linking of Software Components. *IEEE Computer*, 30(3):74–81, March 1997.
- [46] The FreeBSD Documentation Project, <http://www.freebsd.org/doc/en/books/developers-handbook/>. *FreeBSD Developer's Handbook*, 2003.
- [47] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: A Parallel Virtual Machine*. Scientific and Engineering Computation Series. MIT Press, 1994.
- [48] 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.

- [49] Genias Software GmbH. *CODINE User's Guide*, 1993. <http://www.genias.de/genias/english/codine/>.
- [50] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalksi. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4):285–303, August 2003.
- [51] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [52] 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.
- [53] William Gropp and Ewing Lusk. Dynamic process management in an MPI setting. In *Proceedings / Seventh IEEE Symposium on Parallel and Distributed Processing, October 25–28, 1995, San Antonio, Texas*, pages 530–534. IEEE Computer Society Press, 1995.
- [54] William Gropp and Ewing Lusk. Fault tolerance in MPI programs. *High Performance Computing and Applications*, 2003. Submitted to a special issue.
- [55] William Gropp, Ewing Lusk, David Ashton, Rob Ross, Rajeev Thakur, and Brian Toonen. MPICH Abstract Device Interface. Technical Report ANL/MCS-TM-00, Mathematics and Computer Science Division, Argonne National Laboratory, September 2003. Version 3.4.
- [56] William Gropp, Ewing Lusk, David Ashton, Rob Ross, Rajeev Thakur, and Brian Toonen. Mpich abstract device interface version 3.4 reference manual draft. Technical report, Argonne National Laboratory, Mathematics and Computer Science Division, May 2003. Draft.
- [57] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
- [58] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.
- [59] William D. Gropp and Ewing Lusk. *User's Guide for MPICH, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.
- [60] Open Cluster Group: OSCAR Working Group. OSCAR: A packaged cluster software for High Performance Computing.
- [61] The PSCHED API Working Group. PSCHED: An API for Parallel Job/Resource Management, November 1996.
- [62] Indranil Gupta, Tushar Changra, and Germán Goldszmidt. On scalable and efficient distributed failure detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, pages 170–179. ACM Press, 2001.

- [63] Rinku Gupta, Pavan Balaji, Dhableswar K. Panda, and Jarek Nieplocha. Efficient collective operations using remote memory operations on VIA-based clusters. In *Proceedings, International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.
- [64] E. A. Hendriks. Bproc manual. <http://bproc.sourceforge.net/bproc.html>, 2004.
- [65] Erik A. Hendriks. BProc: The Beowulf distributed process space. *16th Annual ACM International Conference on Supercomputing*, June 2002.
- [66] Nicholas Henke. *The Clubmask Administration Guide*. Liniac Project, University of Pennsylvania, Philadelphia, PA, Dec 2003. <http://www.linac.upenn.edu>.
- [67] David Jackson, Brett Bode, David M. Halstead, Ricky Kendall, and Zhou Lei. The portable batch scheduler and the Maui scheduler on linux clusters. In *Proceedings, 4th Annual Linux Showcase and Conference*, October 2000.
- [68] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the Maui scheduler. In *Proceedings, 7th Workshop on Job Scheduling Strategies for Parallel Processing*, 2001.
- [69] Geoffrey James. *The Tao of Programming*. Info Books, Santa Monica, CA, 1987.
- [70] Morris Jette and Mark Grondona. Simple linux utility for resource management. In *Proceedings of ClusterWorld*, San Jose, CA, USA, June 2003.
- [71] L. V. Kale. A tutorial introduction to CHARM. Technical Report TR 92-6, Department of Computer Science, University of Illinois, 1992.
- [72] V. Kale. *The CHARM++ (4.5) Programming Manual*. Department of Computer Science, University of Illinois, Urbana-Champaign, 1999.
- [73] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel Distributed Processing Symposium (IPDPS)*, pages 377–84, Cancun, Mexico, May 2000.
- [74] Amit Karwande, Xin Yuan, and David Lowenthal. CCMPI: A compiled communication capable MPI prototype for ethernet switched clusters. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, USA, June 2003.
- [75] K. Keahey, V. Welch, S. Lang, B. Liu, and S. Meder. Fine-grain authorization policies in the grid: Design and uimplementation. In *Proceedings of 1st International Workshop on Middleware for Grid Computing*, 2003.
- [76] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series, second edition, 1978.

- [77] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. MPI's reduction operations in clustered wide area systems. In *Proceedings, Message Passing Interface Developer's and User's Conference, Atlanta, March 1999*.
- [78] 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.
- [79] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 34(8):131–140, May 1999.
- [80] S. P. Kini, J. Liu, J. Wu, P. Wyckoff, and D. K. Panda. Fast and Scalable Barrier Using RDMA and Multicast Mechanisms for InfiniBand-Based Cluster. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.
- [81] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. Technical Report TR85-706, Cornell University, Computer Science Department, 1985.
- [82]
- [83] Kai Li, Jeffrey F. Naughton, and James S. Plank. Real-time, concurrent checkpoint for parallel programs. *ACM SIGPLAN Notices*, 25(3):79–88, 1990.
- [84] Wei-Jih Li and Jyh-Jong Tsay. Checkpointing Message-Passing Interface (MPI) Parallel Programs. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, 1997.
- [85] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.
- [86] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.
- [87] Michael Litzkow and Marvin Solomon. The Evolution of Condor Checkpointing, 1998.
- [88] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, April 1997.
- [89] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison computer Sciences, April 1997.

- [90] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
- [91] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.
- [92] Myricom, Inc., Arcadia, CA. *GM: A Message-Passing System for Myrinet Networks*, 2.0.6 edition, 2003. <http://www.myri.com/scs/GM-2/doc/html/>.
- [93] Robert H. B. Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.
- [94] Object Management Group. *CORBA Components*, 3.0 edition, June 2002.
- [95] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, 3.0.3 edition, March 2004.
- [96] Regents of the University of California. BSD license. <http://www.opensource.org/licenses/bsd-license.php>.
- [97] Committee on Academic Careers for Experimental Computer Scientists. *Academic Careers for Experimental Computer Scientists and Engineers*. National Academy Press, Washington D.C., 1994. Computer Science and Telecommunications Board, Commission on Physical Sciences, Mathematics, and Applications, National Research Council.
- [98] Portable Batch System (OpenPBS), <http://www.openpbs.org/>.
- [99] OpenPBS 2.3.16. *TM(3) (manpage)*, May 1997.
- [100] Pallas MPI benchmarks documentation – PMB, part MPI-1, 2003.
- [101] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno. Npaci rocks: Tools and techniques for easily deploying managable linux clusters. In *Proceedings, Cluster 2001*, October 2001.
- [102] Portable Batch System (PBS/Pro), <http://www.pbspro.com/>.
- [103] Gregory F. Pfister. *In Search of Clusters*. Prentice Hall PTR, Upper Saddle River, NJ, 2 edition, 1998.
- [104] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under UNIX. In USENIX, editor, *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA*, pages 213–224, Berkeley, CA, USA, January 1995. USENIX.
- [105] Platform LSF HPC, <http://www.platform.com/products/HPC/>.

- [106] David S. Platt. *Introducing Microsoft .NET*. Microsoft Press, third edition, April 2003. ISBN: 0-7356-1918-2.
- [107] Resource Management System, <http://www.quadrics.com>.
- [108] Quadrics, Ltd. *RMS Reference Manual*, 11 (rms.2.81) edition, Nov 2003.
- [109] Brian Randell. Systems structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.
- [110] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An Extensible Toolkit for Low-Overhead Fault-Tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.
- [111] Sriram S. Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. Technical Report CS-TR-98-29, University of Texas, Austin, June 1, 1999.
- [112] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace*, 1997.
- [113] Rolf Riesen, Ron Brightwell, Lee Ann Fisk, Tramm Hudson, Jim Otto, and Arthur B. Maccabe. Cplant. In *Proceedings of the Second Extreme Linux Workshop at the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [114] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, 6(2):183–194, March 1980.
- [115] 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.
- [116] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.
- [117] Scyld Beowulf Scalable Computing, <http://www.scyld.com/>.
- [118] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. In *Protocol Independent Performance Evaluator. In IASTED International Conference on Intelligent Information Management and Systems*, June 1996.
- [119] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the ibm sp2. In *IBM Systems Journal*, number 2, pages 205–221. IBM, 34 edition, 1995.
- [120] Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

- [121] David HM Spector. *Building Linux Clusters*. O'Reilly and Associates, first edition, July 2000.
- [122] Jeffrey M. Squyres. *A Component Architecture for the Message Passing Interface (MPI): The Systems Services Interface (SSI) of LAM/MPI*. PhD thesis, University of Notre Dame, Notre Dame, IN, May 2004.
- [123] 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.
- [124] 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.
- [125] 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.
- [126] 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.
- [127] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.
- [128] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, HI, 1996.
- [129] Georg Stellner. Cocheck: Checkpointing and Process Migration for MPI. In *Proc. of the Int'l Par. Proc. Symp.*, pages 526–531, Los Alamitos, CA, 1996. IEEE Computer Society Press.
- [130] Sun Grid Engine, <http://gridengine.sunsource.net/>.
- [131] Sun Microsystems, Inc., Santa Clara, CA. *Sun ONE Grid Engine Administration and User's Guide*, October 2002.
- [132] Clemens Szyperski, Domink Druntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, second edition, 2002.
- [133] Yuval Tamir and Carlo H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 32–41, Bellaire, Michigan, August 1984. IEEE.
- [134] Todd Tannenbaum and Michael Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, 1(227):40–48, February 1995.
- [135] The LAM Team. *LAM/MPI User's Guide*. Open Systems Lab, Pervasive Technology Labs, Indiana University, <http://www.lam-mpi.org/>, 2003.

- [136] Rajeev Thakur and William Gropp. Improving the Performance of MPI Collective Communication on Switched Networks. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, November 2002. ref: ANL/MCS-P1007-1102.
- [137] Rajeev Thakur and William Gropp. Improving the Performance of Collective Operations in MPICH. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.
- [138] 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.
- [139] 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.
- [140] The BlueGene/L Team. An Overview of the BlueGene/L Supercomputer, 2002.
- [141] Zhijun Tong, Richard Y. Kain, and W. T. Tsai. Rollback recovery in distributed systems using loosely synchronized clocks. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):246–251, 1992.
- [142] Top500 supercomputer list, November 2002. <http://www.top500.org/>.
- [143] Torque Portable Batch System, <http://www.supercluster.org/projects/torque/>.
- [144] Péter Urbán, Xavier Défago, and André Schiper. Chasing the FLP Impossibility Result in a LAN or How Robust Can a Fault Tolerant Server Be? Technical Report DSC/2001/037, École Polytechnique Fédérale de Lausanne, Switzerland, August 2001.
- [145] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings, SuperComputing*, pages 46–46, November 2000.
- [146] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W. Kent Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.
- [147] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung i, and Chandra M. R. Kintala. Checkpointing and its applications. In *Symposium on Fault-Tolerant Computing*, pages 22–31, 1995.
- [148] Sara Williams and Charlie Kindel. The Component Object Model: A Technical Overview. Technical report, Microsoft Corporation, April 1994.
- [149] Adrian Wong, Leonid Oliker, William Kramer, Teresa Kaltz, and David Bailey. System Utilization Benchmark on the Cray T3E and IBM SP. In *The 6th Workshop on Job Scheduling Strategies for Parallel Processing*, April 2000.

- [150] Qianfeng Zhang. MPI collective operations over Myrinet. Master's thesis, The University of British Columbia, Department of Computer Science, June 2002.
- [151] Hua Zhong and Jason Nieh. CRAK: Linux checkpoint / restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.
- [152] Hua Zhong and Jason Nieh. CRAK: Linux checkpoint / restart as a kernel module. Technical Report CUCS-014-01, Columbia University, 2001.