MPI Primer / Developing With LAM

LAM is a parallel processing environment and development system for a network of independent computers. It features the *Message-Passing Interface (MPI)* programming standard, supported by extensive monitoring and debugging tools.

LAM / MPI Key Features:

- full implementation of the MPI standard
- extensive monitoring and debugging tools, runtime and post-mortem
- heterogeneous computer networks
- add and delete nodes
- node fault detection and recovery
- MPI extensions and LAM programming supplements
- direct communication between application processes
- robust MPI resource management
- MPI-2 dynamic processes
- multi-protocol communication (shared memory and network)

Ohio Supercomputer Center

The Ohio State University

Ohio LAM 6.1 **November 11, 1996**

How to Use This Document

This document is organized into four major chapters. It begins with a tutorial covering the simpler techniques of programming and operation. New users should start with the tutorial. The second chapter is an MPI programming primer emphasizing the commonly used routines. Non-standard extensions to MPI and additional programming capabilities unique to LAM are separated into a third chapter. The last chapter is an operational reference. It describes how to configure and start a LAM multicomputer, and how to monitor processes and messages.

This document is user oriented. It does not give much insight into how the system is implemented. It does not detail every option and capability of every command and routine. An extensive set of manual pages cover all the commands and internal routines in great detail and are meant to supplement this document.

The reader will note a heavy bias towards the C programming language, especially in the code samples. There is no Fortran version of this document. The text attempts to be language insensitive and the appendices contain Fortran code samples and routine prototypes.

We have kept the font and syntax conventions to a minimum.

`code`	This font is used for things you type on the keyboard or see printed on the screen. We use it in code sections and tables but not in the main text.
<symbol>	This is a symbol used to abstract something you would type. We use this convention in commands.
Section	Italics are used to cross reference another section in the document or another document. Italics are also used to distinguish LAM commands.

Table of Contents

LAM Architecture

LAM runs on each computer as a single daemon (server) uniquely structured as a nano-kernel and hand-threaded virtual processes. The nano-kernel component provides a simple message-passing, rendez-vous service to local processes. Some of the in-daemon processes form a network communication subsystem, which transfers messages to and from other LAM daemons on other machines. The network subsystem adds features such as packetization and buffering to the base synchronization. Other in-daemon processes are servers for remote capabilities, such as program execution and parallel file access. The layering is quite distinct: the nano-kernel has no connection with the network subsystem, which has no connection with the servers. Users can configure in or out services as necessary.

The unique software engineering of LAM is transparent to users and system administrators, who only see a conventional daemon. System developers can de-cluster the daemon into a daemon containing only the nano-kernel and several full client processes. This developers' mode is still transparent to users but exposes LAM's highly modular components to simplified individual debugging. It also reveals LAM's evolution from Trollius, which ran natively on scalable multicomputers and joined them to a host network through a uniform programming interface.

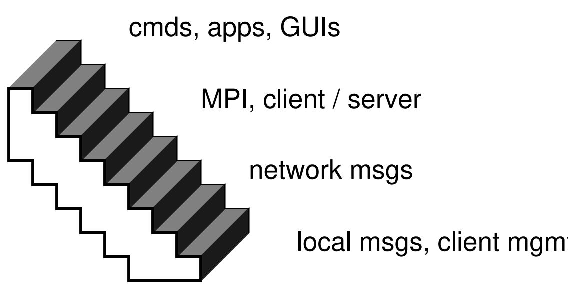

Figure 1: LAM's Layered Design

The network layer in LAM is a documented, primitive and abstract layer on which to implement a more powerful communication standard like MPI (PVM has also been implemented).

Debugging

A most important feature of LAM is hands-on control of the multicomputer. There is very little that cannot be seen or changed at runtime. Programs residing anywhere can be executed anywhere, stopped, resumed, killed, and watched the whole time. Messages can be viewed anywhere on the multicomputer and buffer constraints tuned as experience with the application dictates. If the synchronization of a process and a message can be easily displayed, mismatches resulting in bugs can easily be found. These and other services are available both as a programming library and as utility programs run from any shell.

MPI Implementation

MPI synchronization boils down to four variables: context, tag, source rank, and destination rank. These are mapped to LAM's abstract synchronization at the network layer. MPI debugging tools interpret the LAM information with the knowledge of the LAM / MPI mapping and present detailed information to MPI programmers.

A significant portion of the MPI specification can be and is implemented within the runtime system and independent of the underlying environment.

As with all MPI implementations, LAM must synchronize the launch of MPI applications so that all processes locate each other before user code is entered. The *mpirun* command achieves this after finding and loading the program(s) which constitute the application. A simple SPMD application can be specified on the mpirun command line while a more complex configuration is described in a separate file, called an application schema.

MPI programs developed on LAM can be moved without source code changes to any other platform that supports MPI.

LAM installs anywhere and uses the shell's search path at all times to find LAM and application executables. A multicomputer is specified as a simple list of machine names in a file, which LAM uses to verify access, start the environment, and remove it.

How to Get LAM

LAM is freely available under a GNU license via anonymous ftp from ftp.osc.edu.

LAM / MPI Tutorial Introduction

The example programs in this section illustrate common operations in MPI. You will also see how to run and debug a program with LAM.

Programming Tutorial

For basic applications, MPI is as easy to use as any other message-passing library. The first program is designed to run with exactly two processes. One process sends a message to the other and then both terminate. Enter the following code in trivial.c or obtain the source from the LAM source distribution (examples/trivial/trivial.c).

```
/*
 * Transmit a message in a two process system.
 */
#include <mpi.h>
#define BUFSIZE          64
int                      buf[64];
int
main(argc, argv)
int                      argc;
char                     *argv[];
{
      int                size, rank;
      MPI_Status         status;
/*
 * Initialize MPI.
 */
      MPI_Init(&argc, &argv);
/*
 * Error check the number of processes.
 * Determine my rank in the world group.
```

```
 * The sender will be rank 0 and the receiver, rank 1.
 */
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      if (2 != size) {
            MPI_Finalize();
            return(1);
      }
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/*
 * As rank 0, send a message to rank 1.
 */
      if (0 == rank) {
            MPI_Send(buf, sizeof(buf), MPI_INT, 1, 11,
                        MPI_COMM_WORLD);
      }
/*
 * As rank 1, receive a message from rank 0.
 */
      else {
            MPI_Recv(buf, sizeof(buf), MPI_INT, 0, 11,
                        MPI_COMM_WORLD, &status);
      }
      MPI_Finalize();
      return(0);
}
```

Note that the program uses standard C program structure, statements, variable declarations and types, and functions.

The World of MPI

Processes are represented by a unique “rank” (integer) and ranks are numbered 0, 1, 2, ..., N-1. MPI_COMM_WORLD means “all the processes in the MPI application.” It is called a communicator and it provides all information necessary to do message-passing. Portable libraries do more with communicators to provide synchronization protection that most other message-passing systems cannot handle.

Enter and Exit MPI

As with other systems, two routines are provided to initialize and cleanup an MPI process:

```
MPI_Init(int *argc, char ***argv);
MPI_Finalize(void);
```

Who Am I; Who Are They?

Typically, a process in a parallel application needs to know who it is (its rank) and how many other processes exist. A process finds out its own rank by calling MPI_Comm_rank().

```
MPI_Comm_rank(MPI_Comm comm, int *rank);
```

The total number of processes is returned by MPI_Comm_size().

```
MPI_Comm_size(MPI_Comm comm, int *size);
```

Sending Messages

A message is an array of elements of a given datatype. MPI supports all the basic datatypes and allows a more elaborate application to construct new datatypes at runtime.

A message is sent to a specific process and is marked by a tag (integer) specified by the user. Tags are used to distinguish between different message types a process might send/receive. In the example program above, the additional synchronization offered by the tag is unnecessary. Therefore, any random value is used that matches on both sides.

```
MPI_Send(void *buf, int count, MPI_Datatype
     dtype, int dest, int tag, MPI_Comm comm);
```

Receiving Messages

A receiving process specifies the tag and the rank of the sending process. MPI_ANY_TAG and MPI_ANY_SOURCE may be used to receive a message of any tag and from any sending process.

```
MPI_Recv(void *buf, int count, MPI_Datatype
     dtype, int source, int tag, MPI_Comm comm,
     MPI_Status *status);
```

Information about the received message is returned in a status variable. If wildcards are used, the received message tag is status.MPI_TAG and the rank of the sending process is status.MPI_SOURCE.

Another routine, not used in the example program, returns the number of datatype elements received. It is used when the number of elements received might be smaller than number specified to MPI_Recv(). It is an error to send more elements than the receiving process will accept.

```
MPI_Get_count(MPI_Status, &status,
     MPI_Datatype dtype, int *nelements);
```

Master / Slave Example

The following example program is a communication skeleton for a dynamically load balanced master/slave application. The source can be obtained from the LAM source distribution (examples/trivial/ezstart.c). The program is designed to work with a minimum of two processes: one master and one slave.

```
#include <mpi.h>
#define WORKTAG         1
#define DIETAG          2
#define NUM_WORK_REQS   200
static void             master();
static void             slave();
/*
 * main
 * This program is really MIMD, but is written SPMD for
 * simplicity in launching the application.
 */
int
main(argc, argv)
int                     argc;
char                    *argv[];
{
      int               myrank;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, /* group of everybody */
            &myrank);               /* 0 thru N-1 */
      if (myrank == 0) {
            master();
      } else {
            slave();
      }
      MPI_Finalize();
      return(0);
}
/*
 * master
 * The master process sends work requests to the slaves
 * and collects results.
 */
static void
master()
{
      int               ntasks, rank, work;
      double            result;
      MPI_Status        status;
      MPI_Comm_size(MPI_COMM_WORLD,
            &ntasks);               /* #processes in app */
```

```
/*
 * Seed the slaves.
 */
      work = NUM_WORK_REQS;          /* simulated work */
      for (rank = 1; rank < ntasks; ++rank) {
            MPI_Send(&work,          /* message buffer */
                  1,                 /* one data item */
                  MPI_INT,           /* of this type */
                  rank,              /* to this rank */
                  WORKTAG,           /* a work message */
                  MPI_COMM_WORLD);   /* always use this */
            work--;
      }
/*
 * Receive a result from any slave and dispatch a new work
 * request until work requests have been exhausted.
 */
      while (work > 0) {
            MPI_Recv(&result,        /* message buffer */
                  1,                 /* one data item */
                  MPI_DOUBLE,        /* of this type */
                  MPI_ANY_SOURCE,    /* from anybody */
                  MPI_ANY_TAG,       /* any message */
                  MPI_COMM_WORLD,    /* communicator */
                  &status);          /* recv'd msg info */
            MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
                  WORKTAG, MPI_COMM_WORLD);
            work--;                  /* simulated work */
      }
/*
 * Receive results for outstanding work requests.
 */
      for (rank = 1; rank < ntasks; ++rank) {
            MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      }
/*
 * Tell all the slaves to exit.
 */
      for (rank = 1; rank < ntasks; ++rank) {
            MPI_Send(0, 0, MPI_INT, rank, DIETAG,
                  MPI_COMM_WORLD);
      }
}
```

```
/*
 * slave
 * Each slave process accepts work requests and returns
 * results until a special termination request is received.
 */
static void
slave()
{
      double            result;
      int               work;
      MPI_Status        status;
      for (;;) {
            MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &status);
/*
 * Check the tag of the received message.
 */
            if (status.MPI_TAG == DIETAG) {
                  return;
            }
            sleep(2);
            result = 6.0;            /* simulated result */
            MPI_Send(&result, 1, MPI_DOUBLE, 0, 0,
                  MPI_COMM_WORLD);
      }
}
```

The workings of ranks, tags and message lengths should be mastered before constructing serious MPI applications.

Operation Tutorial

Before running LAM you must establish certain environment variables and search paths for your shell. Add the following commands or equivalent to your shell start-up file (.cshrc, assuming C shell). Do not add these to your .login as they would not be effective on remote machines when *rsh* is used to start LAM.

```
setenv LAMHOME <LAM installation directory>
set path = ($path $LAMHOME/bin)
```

The local system administrator, or the person who installed LAM, will know the location of the LAM installation directory. After editing the shell start-up file, invoke it to establish the new values. This is not necessary on subsequent logins to the UNIX system.

```
% source .cshrc
```

Many LAM commands require one or more nodeids. Nodeids are specified on the command line as n<list>, where <list> is a list of comma separated nodeids or nodeid ranges.

```
n1
n1,3,5-10
```

The mnemonic ‘h’ refers to the local node where the command is typed (as in ‘here’).

Compilation

Any native C compiler is used to translate LAM programs for execution. All LAM runtime routines are found in a few libraries. LAM provides a wrapping command called *hcc* which invokes cc with the proper header and library directories, and is used exactly like the native cc.

```
% hcc -o trivial trivial.c -lmpi
```

The major, internal LAM libraries are automatically linked. The MPI library is explicitly linked. Since LAM supports heterogeneous computing, it is up to the user to compile the source code for each of the various CPUs on their respective machines. After correcting any errors reported by the compiler, proceed to starting the LAM session.

Starting LAM

Before starting LAM, the user specifies the machines that will form the multicomputer. Create a host file listing the machine names, one on each line. An example file is given below for the machines “ohio” and “osc”. Lines starting with the # character are treated as comment lines.

```
# a 2-node LAM
ohio
osc
```

The first machine in the host file will be assigned nodeid 0, the second nodeid 1, etc. Now verify that the multicomputer is ready to run LAM. The *recon* tool checks if the user has access privileges on each machine in the multicomputer and if LAM is installed and accessible.

```
% recon -v <host file>
```

If recon does not report a problem, proceed to start the LAM session with the *lamboot* tool.

```
% lamboot -v <host file>
```

The -v (verbose) option causes lamboot to report on the start-up process as it progresses. You should return to the your own shell's prompt. LAM presents no special shell or interface environment.

Even if all seems well after start-up, verify communication with each node. *tping* is a simple confidence building command for this purpose.

```
% tping n0
```

Repeat this command for all nodes or ping all the nodes at once with the broadcast mnemonic, N. tping responds by sending a message between the local node (where the user invoked tping) and the specified node. Successful execution of tping proves that the target node, nodes along the route from the local node to the target node, and the communication links between them are working properly. If tping fails, press Control-Z, terminate the session with the *wipe* tool and then restart the system. See *Terminating the Session*.

Executing Programs

To execute a program, use the *mpirun* command. The first example program is designed to run with two processes. The -c <#> option runs copies of the given program on nodes selected in a round-robin manner.

```
% mpirun -v -c 2 trivial
```

The example invocation above assumes that the program is locatable on the machine on which it will run. mpirun can also transfer the program to the target node before running it. Assuming your multicomputer for this tutorial is homogeneous, you can use the -s h option to run both processes.

```
% mpirun -v -c 2 -s h trivial
```

If the processes executed correctly, they will terminate and leave no traces. If you want more feedback, try using tprintf() functions within the program.

Monitoring

The first example program runs too quickly to be monitored. Try changing the tag in the call to MPI_Recv() to 12 (from 11). Recompile the program and rerun it as before. Now the receiving process cannot synchronize with the message from the send process because the tags are unequal. Look at the status of all MPI processes with the *mpitask* command.

```
% mpitask
TASK (G/L)     FUNCTION     PEER|ROOT    TAG    COMM   COUNT DATATYPE
0/0 trivial    Finalize
1/1 trivial    Recv         0/0          12     WORLD  64    INT
```

You will notice that the receiving process is blocked in a call to MPI_Recv() - a synchronizing message has not been received. From the code we know this is process rank 1 in the MPI application, which is confirmed in the first column, the MPI task identification. The first number is the rank within the world group. The second number is the rank within the communicator being used by MPI_Recv(), in this case (and in many applications with simple communication structure) also the world group. The specified source of the message is likewise identified. The synchronization tag is 12 and the length of the receive buffer is 64 elements of type MPI_INT.

The message was transferred from the sending process to a system buffer en route to process rank 1. MPI_Send() was able to return and the process has called MPI_Finalize(). System buffers, which can be thought of as message queues for each MPI process, can be examined with the *mpimsg* command.

```
% mpimsg
SRC (G/L)    DEST (G/L)   TAG    COMM   COUNT     DATATYPE     MSG
0/0          1/1          11     WORLD  64        INT          n1,#0
```

The message shows that it originated from process rank 0 using MPI_COMM_WORLD and that it is waiting in the message queue of process rank 1, the destination. The tag is 11 and the message contains 64 elements of type MPI_INT. This information corresponds to the arguments given to MPI_Send(). Since the application is faulty and will never complete, we will kill it with the *lamclean* command.

```
% lamclean -v
```

The LAM session should be in the same state as after invoking lamboot. You can also terminate the session and restart it with lamboot, but this is a much slower operation. You can now correct the program, recompile and rerun.

Terminating the Session

To terminate LAM, use the *wipe* tool. The host file argument must be the same as the one given to lamboot.

```
% wipe -v <host file>
```

MPI Programming Primer

Basic Concepts

Through Message Passing Interface (MPI) an application views its parallel environment as a static group of processes. An MPI process is born into the world with zero or more siblings. This initial collection of processes is called the world group. A unique number, called a rank, is assigned to each member process from the sequence 0 through N-1, where N is the total number of processes in the world group. A member can query its own rank and the size of the world group. Processes may all be running the same program (SPMD) or different programs (MIMD). The world group processes may subdivide, creating additional subgroups with a potentially different rank in each group.

A process sends a message to a destination rank in the desired group. A process may or may not specify a source rank when receiving a message. Messages are further filtered by an arbitrary, user specified, synchronization integer called a tag, which the receiver may also ignore.

An important feature of MPI is the ability to guarantee independent software developers that their choice of tag in a particular library will not conflict with the choice of tag by some other independent developer or by the end user of the library. A further synchronization integer called a context is allocated by MPI and is automatically attached to every message. Thus, the four main synchronization variables in MPI are the source and destination ranks, the tag and the context.

A communicator is an opaque MPI data structure that contains information on one group and that contains one context. A communicator is an argument to all MPI communication routines. After a process is created and initializes MPI, three predefined communicators are available.

MPI_COMM_WORLD	the world group
MPI_COMM_SELF	group with one member, myself
MPI_COMM_PARENT	an intercommunicator between two groups: my world group and my parent group (See *Dynamic Processes*.)

Many applications require no other communicators beyond the world communicator. If new subgroups or new contexts are needed, additional communicators must be created.

MPI constants, templates and prototypes are in the MPI header file, mpi.h.

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

MPI_Init	Initialize MPI state.
MPI_Finalize	Clean up MPI state.
MPI_Abort	Abnormally terminate.
MPI_Comm_size	Get group process count.
MPI_Comm_rank	Get my rank within process group.
MPI_Initialized	Has MPI been initialized?

Initialization

The first MPI routine called by a program must be MPI_Init(). The command line arguments are passed to MPI_Init().

```
MPI_Init(int *argc, char **argv[]);
```

A process ceases MPI operations with MPI_Finalize().

```
MPI_Finalize(void);
```

In response to an error condition, a process can terminate itself and all members of a communicator with MPI_Abort(). The implementation may report the error code argument to the user in a manner consistent with the underlying operation system.

```
MPI_Abort (MPI_Comm comm, int errcode);
```

Basic Parallel Information

Two numbers that are very useful to most parallel applications are the total number of parallel processes and self process identification. This information is learned from the MPI_COMM_WORLD communicator using the routines MPI_Comm_size() and MPI_Comm_rank().

```
MPI_Comm_size (MPI_Comm comm, int *size);
MPI_Comm_rank (MPI_Comm comm, int *rank);
```

Of course, any communicator may be used, but the world information is usually key to decomposing data across the entire parallel application.

MPI_Send	Send a message in standard mode.
MPI_Recv	Receive a message.
MPI_Get_count	Count the elements received.
MPI_Probe	Wait for message arrival.

MPI_Bsend	Send a message in buffered mode.
MPI_Ssend	Send a message in synchronous mode.
MPI_Rsend	Send a message in ready mode.
MPI_Buffer_attach	Attach a buffer for buffered sends.
MPI_Buffer_detach	Detach the current buffer.
MPI_Sendrecv	Send in standard mode, then receive.
MPI_Sendrecv_replace	Send and receive from/to one area.
MPI_Get_elements	Count the basic elements received.

Blocking Point-to-Point

This section focuses on blocking, point-to-point, message-passing routines. The term "blocking" in MPI means that the routine does not return until the associated data buffer may be reused. A point-to-point message is sent by one process and received by one process.

Send Modes

The issues of flow control and buffering present different choices in designing message-passing primitives. MPI does not impose a single choice but instead offers four transmission modes that cover the synchronization, data transfer and performance needs of most applications. The mode is selected by the sender through four different send routines, all with identical argument lists. There is only one receive routine. The four send modes are:

standard	The send completes when the system can buffer the message (it is not obligated to do so) or when the message is received.
buffered	The send completes when the message is buffered in application supplied space, or when the message is received.
synchronous	The send completes when the message is received.
ready	The send must not be started unless a matching receive has been started. The send completes immediately.

Standard Send

Standard mode serves the needs of most applications. A standard mode message is sent with MPI_Send().

```
MPI_Send (void *buf, int count, MPI_Datatype
     dtype, int dest, int tag, MPI_Comm comm);
```

An MPI message is not merely a raw byte array. It is a count of typed elements. The element type may be a simple raw byte or a complex data structure. See *Message Datatypes*.

The four MPI synchronization variables are indicated by the MPI_Send() parameters. The source rank is the caller's. The destination rank and message tag are explicitly given. The context is a property of the communicator.

As a blocking routine, the buffer can be overwritten when MPI_Send() returns. Although most systems will buffer some number of messages, especially short messages, without any receiver, a programmer cannot rely upon MPI_Send() to buffer even one message. Expect that the routine will not return until there is a matching receiver.

Receive

A message in any mode is received with MPI_Recv().

```
MPI_Recv (void *buf, int count, MPI_Datatype
     dtype, int source, int tag, MPI_Comm comm,
     MPI_Status *status);
```

Again the four synchronization variables are indicated, with source and destination swapping places. The source rank and the tag can be ignored with the special values MPI_ANY_SOURCE and MPI_ANY_TAG. If both these wildcards are used, the next message for the given communicator is received.

Status Object

An argument not present in MPI_Send() is the status object pointer. The status object is filled with useful information when MPI_Recv() returns. If the source and/or tag wildcards were used, the actual received source rank and/or message tag are accessible directly from the status object.

`status.MPI_SOURCE`	the sender's rank
`status.MPI_TAG`	the tag given by the sender

Message Lengths

It is erroneous for an MPI program to receive a message longer than the specified receive buffer. The message might be truncated or an error condition might be raised or both. It is completely acceptable to receive a message shorter than the specified receive buffer. If a short message may arrive, the application can query the actual length of the message with MPI_Get_count().

```
MPI_Get_count (MPI_Status *status,
     MPI_Datatype dtype, int *count);
```

The status object and MPI datatype are those provided to MPI_Recv(). The count returned is the number of elements received of the given datatype. See *Message Datatypes*.

Probe Sometimes it is impractical to pre-allocate a receive buffer. MPI_Probe() synchronizes a message and returns information about it without actually receiving it. Only synchronization variables and the status object are provided as arguments. MPI_Probe() does not return until a message is synchronized.

```
MPI_Probe (in source, int tag, MPI_Comm comm,
     MPI_Status *status);
```

After a suitable message buffer has been prepared, the same message reported by MPI_Probe() can be received with MPI_Recv().

`MPI_Isend`	Begin to send a standard message.
`MPI_Irecv`	Begin to receive a message.
`MPI_Wait`	Complete a pending request.
`MPI_Test`	Check or complete a pending request.
`MPI_Iprobe`	Check message arrival.

`MPI_Ibsend`	Begin to send a buffered message.
`MPI_Issend`	Begin to send a synchronous message.
`MPI_Irsend`	Begin to send a ready message.
`MPI_Request_free`	Free a pending request.
`MPI_Waitany`	Complete any one request.
`MPI_Testany`	Check or complete any one request.
`MPI_Waitall`	Complete all requests.
`MPI_Testall`	Check or complete all requests.
`MPI_Waitsome`	Complete one or more requests.
`MPI_Testsome`	Check or complete one or more requests.
`MPI_Cancel`	Cancel a pending request.
`MPI_Test_cancelled`	Check if a pending request was cancelled.

Nonblocking Point-to-Point

The term “nonblocking” in MPI means that the routine returns immediately and may only have started the message transfer operation, not necessarily completed it. The application may not safely reuse the message buffer after a nonblocking routine returns. The four blocking send routines and one blocking receive routine all have nonblocking counterparts. The nonblocking routines have an extra output argument - a request object. The request is later passed to one of a suite of completion routines. Once an operation has completed, its message buffer can be reused.

The intent of nonblocking message-passing is to start a message transfer at the earliest possible moment, continue immediately with important computation, and then insist upon completion at the latest possible moment. When the earliest and latest moment are the same, nonblocking routines are not useful. Otherwise, a non-blocking operation on certain hardware could overlap communication and computation, thus improving performance.

MPI_Isend() begins a standard nonblocking message send.

```
MPI_Isend (void *buf, int count, MPI_Datatype
     dtype, int dest, int tag, MPI_Comm comm,
     MPI_Request *req);
```

Likewise, MPI_Irecv() begins a nonblocking message receive.

```
MPI_Irecv (void *buf, int count, MPI_Datatype
     dtype, int source, int tag, MPI_Comm comm,
     MPI_Request *req);
```

Request Completion

Both routines accept arguments with the same meaning as their blocking counterparts. When the application wishes to complete a nonblocking send or receive, a completion routine is called with the corresponding request. The Test() routine is nonblocking and the Wait() routine is blocking. Other completion routines operate on multiple requests.

```
MPI_Test (MPI_Request *req, int *flag,
     MPI_Status *status);
MPI_Wait (MPI_Request *req, MPI_Status *status);
```

MPI_Test() returns a flag in an output argument that indicates if the request completed. If true, the status object argument is filled with information. If the request was a receive operation, the status object is filled as in MPI_Recv(). Since MPI_Wait() blocks until completion, the status object argument is always filled.

Probe

MPI_Iprobe() is the nonblocking counterpart of MPI_Probe(), but it does not return a request object since it does not begin any message transfer that would need to complete. It sets the flag argument which indicates the presence of a matching message (for a subsequent receive).

```
MPI_Iprobe (int source, int tag, MPI_Comm comm,
     int *flag, MPI_Status *status);
```

Programmers should not consider the nonblocking routines as simply fast versions of the blocking calls and therefore the preferred choice in all applications. Some implementations cannot take advantage of the opportunity to optimize performance offered by the nonblocking routines. In order to preserve the semantics of the message-passing interface, some implementations may even be slower with nonblocking transfers. Programmers should have a clear and substantial computation overlap before considering nonblocking routines.

MPI_Type_vector	Create a strided homogeneous vector.
MPI_Type_struct	Create a heterogeneous structure.
MPI_Address	Get absolute address of memory location.
MPI_Type_commit	Use datatype in message transfers.
MPI_Pack	Pack element into contiguous buffer.
MPI_Unpack	Unpack element from contiguous buffer.
MPI_Pack_size	Get packing buffer size requirement.

MPI_Type_continuous	Create contiguous homogeneous array.
MPI_Type_hvector	Create vector with byte displacement.
MPI_Type_indexed	Create a homogeneous structure.
MPI_Type_hindexed	Create an index with byte displacements.
MPI_Type_extent	Get range of space occupied by a datatype.
MPI_Type_size	Get amount of space occupied by a datatype.
MPI_Type_lb	Get displacement of datatype's lower bound.
MPI_Type_ub	Get displacement of datatype's upper bound.
MPI_Type_free	Free a datatype.

Message Datatypes

Heterogeneous computing requires that message data be typed or described somehow so that its machine representation can be converted as necessary between computer architectures. MPI can thoroughly describe message datatypes, from the simple primitive machine types to complex structures, arrays and indices.

The message-passing routines all accept a datatype argument, whose C typedef is MPI_Datatype. For example, recall MPI_Send(). Message data is specified as a number of elements of a given type.

Several MPI_Datatype values, covering the basic data units on most computer architectures, are predefined:

MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float

MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	a raw byte

The number of bytes occupied by these basic datatypes follows the corresponding C definition. Thus, MPI_INT could occupy four bytes on one machine and eight bytes on another machine. A message count of one MPI_INT specified by both sender and receiver would, in one direction, require padding and always be correct. In the reverse direction, the integer may not be representable in the lesser number of bytes and the communication will fail.

Derived Datatypes

Derived datatypes are built by combining basic datatypes, or previously built derived datatypes. A derived datatype describes a memory layout which consists of multiple arrays of elements. A generalization of this capability is that the four varieties of constructor routines offer more or less control over array length, array element datatype and array displacement.

contiguous	one array length, no displacement, one datatype
vector	one array length, one displacement, one datatype
indexed	multiple array lengths, multiple displacements, one datatype
structure	multiple everything

Strided Vector Datatype

Consider a two dimensional matrix with R rows and C columns stored in row major order. The application wishes to communicate one entire column. A vector derived datatype fits the requirement.

```
MPI_Type_Vector (int count, int blocklength,
     int stride, MPI_Datatype oldtype,
     MPI_Datatype *newtype);
```

Assuming the matrix elements are of MPI_INT, the arguments for the stated requirement would be:

```
int                     R, C;
MPI_Datatype            newtype;
MPI_Type_vector(R, 1, C, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
```

The count of blocks (arrays) is the number of elements in a column (R). Each block contains just one element and the elements are strided (displaced) from each other by the number of elements in a row (C).[1]

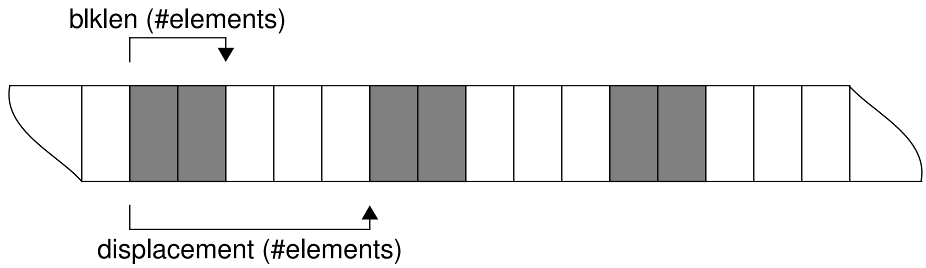

Figure 2: Strided Vector Datatype

Structure Datatype

An arbitrary record whose template is a C structure is a common message form. The most flexible MPI derived datatype, the structure, is required to describe the memory layout.

```
MPI_Type_struct (int count, int blocklengths[],
     MPI_Aint displacements[], MPI_Datatype
     dtypes[], MPI_Datatype *newtype);
```

In the following code fragment, a C struct of diverse fields is described with MPI_Type_struct() in the safest, most portable manner.

```
/*
 * non-trivial structure
 */
struct cell {
      double            energy;
      char              flags;
      float             coord[3];
};
/*
 * We want to be able to send arrays of this datatype.
 */
struct cell             cloud[2];
/*
 * new datatype for cell struct
 */
MPI_Datatype            celltype;
```

1. Note that this datatype is not sufficient to send multiple columns from the matrix, since it does not presume the final displacement between the last element of the first column and the first element of the second column. One solution is to use MPI_Type_struct() and MPI_UB. See *Structure Datatype*.

```
int                     blocklengths[4] = {1, 1, 3, 1};
MPI_Aint                base;
MPI_Aint                displacements[4];
MPI_Datatype            types[4] = {MPI_DOUBLE, MPI_CHAR,
                              MPI_FLOAT, MPI_UB};
MPI_Address(&cloud[0].energy, &displacement[0]);
MPI_Address(&cloud[0].flags, &displacement[1]);
MPI_Address(&cloud[0].coord, &displacement[2]);
MPI_Address(&cloud[1].energy, &displacement[3]);
base = displacement[0];
for (i = 0; i < 4; ++i) displacement[i] -= base;
MPI_Type_struct(4, blocklengths, displacements, types,
            &celltype);
MPI_Type_commit(&celltype);
```

The displacements in a structure datatype are byte offsets from the first storage location of the C structure. Without guessing the compiler's policy for packing and alignment in a C structure, the MPI_Address() routine and some pointer arithmetic are the best way to get the precise values. MPI_Address() simply returns the absolute address of a location in memory. The displacement of the first element within the structure is zero.

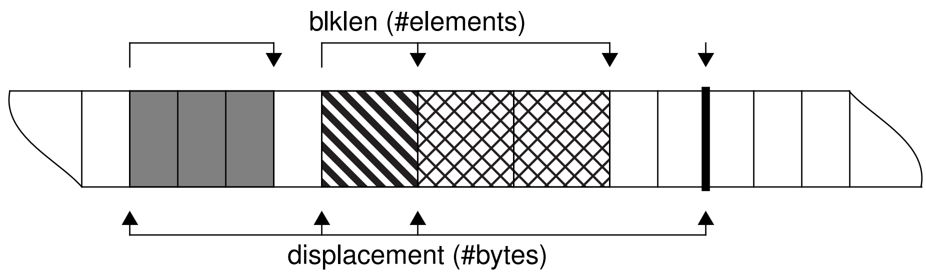

Figure 3: Struct Datatype

When transferring arrays of a given datatype (by specifying a count greater than 1 in MPI_Send(), for example), MPI assumes that the array elements are stored contiguously. If necessary, a gap can be specified at the end of the derived datatype memory layout by adding an artificial element of type MPI_UB, to the datatype description and giving it a displacement that extends to the first byte of the second element in an array.

MPI_Type_Commit() separates the datatypes that will be used to transfer messages from the intermediate ones that are scaffolded on the way to some very complicated datatype. A derived datatype must be committed before being used in communication.

Packed Datatype

The description of a derived datatype is fixed after creation at runtime. If any slight detail changes, such as the blocklength of a particular field in a structure, a new datatype is required. In addition to the tedium of creating many derived datatypes, a receiver may not know in advance which of a nearly identical suite of datatypes will arrive in the next message. MPI's solution is packing and unpacking routines that incrementally assemble and disassemble a contiguous message buffer. The packed message has the special MPI datatype, MPI_PACKED, and is transferred with a count equal to its length in bytes.

```
MPI_Pack_size (int incount, MPI_Datatype dtype,
     MPI_Comm comm, int *size);
```

MPI_Pack_size() returns the packed message buffer size requirement for a given datatype. This may be greater than one would expect from the type description due to hidden, implementation dependent packing overhead.

```
MPI_Pack (void *inbuf, int incount, MPI_Datatype
     dtype, void *outbuf, int outsize,
     int *position, MPI_Comm comm);
```

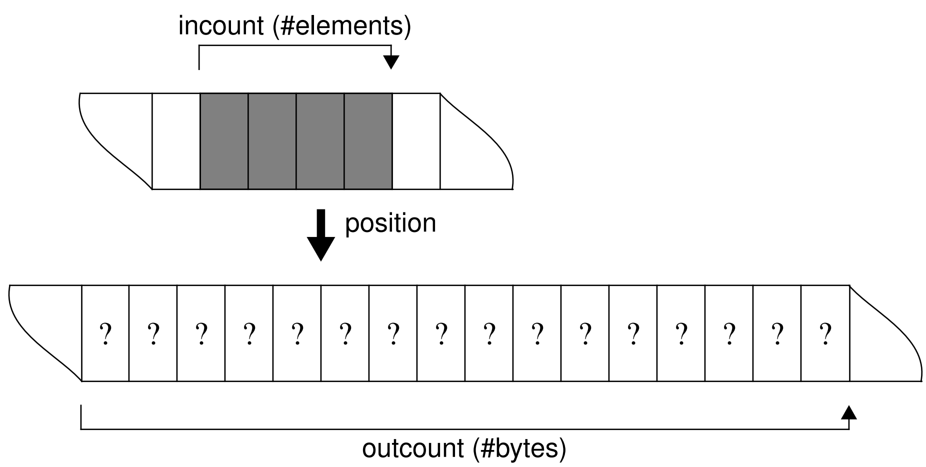

Figure 4: Packed Datatype

Contiguous blocks of homogeneous elements are packed one at a time with MPI_Pack(). After each call, the current location in the packed message buffer is updated. The "in" data are the elements to be packed and the "out" data is the packed message buffer. The outsize is always the maximum size of the packed message buffer, to guard against overflow.

```
MPI_Unpack (void *inbuf, int insize,
      int *position, void *outbuf, int outcount,
      MPI_Datatype datatype, MPI_Comm comm);
```

MPI_Unpack() is the natural reverse of MPI_Pack() where the “in” data is the packed message buffer and the “out” data are the elements to be unpacked.

Consider a networking application that is transferring a variable length message consisting of a count, several (count) Internet addresses as four byte character arrays and an equal number of port numbers as shorts.

```
#define MAXN                100
unsigned char               addrs[MAXN][4];
short                       ports[MAXN];
```

In the following code fragment, a message is packed and sent based on a given count.

```
unsigned int                membersize, maxsize;
int                         position;
int                         nhosts;
int                         dest, tag;
char                        *buffer;
/*
 * Do this once.
 */
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &membersize);
maxsize = membersize;
MPI_Pack_size(MAXN * 4, MPI_UNSIGNED_CHAR, MPI_COMM_WORLD,
            &membersize);
maxsize += membersize;
MPI_Pack_size(MAXN, MPI_SHORT, MPI_COMM_WORLD, &membersize);
maxsize += membersize;
buffer = malloc(maxsize);
/*
 * Do this for every new message.
 */
nhosts = /* some number less than MAXN */ 50;
position = 0;
MPI_Pack(nhosts, 1, MPI_INT, buffer, maxsize, &position,
            MPI_COMM_WORLD);
MPI_Pack(addrs, nhosts * 4, MPI_UNSIGNED_CHAR, buffer,
            maxsize, &position, MPI_COMM_WORLD);
MPI_Pack(ports, nhosts, MPI_SHORT, buffer, maxsize,
            &position, MPI_COMM_WORLD);
MPI_Send(buffer, position, MPI_PACKED, dest, tag,
            MPI_COMM_WORLD);
```

A buffer is allocated once to contain the maximum size of a packed message. In the following code fragment, a message is received and unpacked, based on a count packed into the beginning of the message.

```
int                     src;
int                     msgsize;
MPI_Status              status;
MPI_Recv(buffer, maxsize, MPI_PACKED, src, tag,
            MPI_COMM_WORLD, &status);
position = 0;
MPI_Get_count(&status, MPI_PACKED, &msgsize);
MPI_Unpack(buffer, msgsize, &position, &nhosts, 1, MPI_INT,
            MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, addrs, nhosts * 4,
            MPI_UNSIGNED_CHAR, MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, ports, nhosts,
            MPI_SHORT, MPI_COMM_WORLD);
```

MPI_Bcast	Send one message to all group members.
MPI_Gather	Receive and concatenate from all members.
MPI_Scatter	Separate and distribute data to all members.
MPI_Reduce	Combine messages from all members.

MPI_Barrier	Wait until all group members reach this point.
MPI_Gatherv	Vary counts and buffer displacements.
MPI_Scatterv	Vary counts and buffer displacements.
MPI_Allgather	Gather and then broadcast.
MPI_Allgatherv	Variably gather and then broadcast.
MPI_Alltoall	Gather and then scatter.
MPI_Alltoallv	Variably gather and then scatter.
MPI_Op_create	Create reduction operation.
MPI_Allreduce	Reduce and then broadcast.
MPI_Reduce_scatter	Reduce and then scatter.
MPI_Scan	Perform a prefix reduction.

Collective Message-Passing

Collective operations consist of many point-to-point messages which happen more or less concurrently (depending on the operation and the internal algorithm) and involve all processes in a given communicator. Every process must call the same MPI collective routine. Most of the collective operations are variations and/or combinations of four primitives: broadcast, gather, scatter and reduce.

Broadcast

```
MPI_Bcast (void *buf, int count, MPI_Datatype
     dtype, int root, MPI_Comm comm);
```

In the broadcast operation, all processes specify the same root process, whose buffer contents will be sent. Processes other than the root specify receive buffers. After the operation, all buffers contain the message from the root process.

Scatter

```
MPI_Scatter (void *sendbuf, int sendcount,
     MPI_Datatype sendtype, void *recvbuf,
     int recvcount, MPI_Datatype recvtype,
     int root, MPI_Comm comm);
```

MPI_Scatter() is also a one-to-many collective operation. All processes specify the same receive count. The send arguments are only significant to the root process, whose buffer actually contains sendcount * N elements of the given datatype, where N is the number of processes in the given communicator. The send buffer will be divided equally and dispersed to all pro-

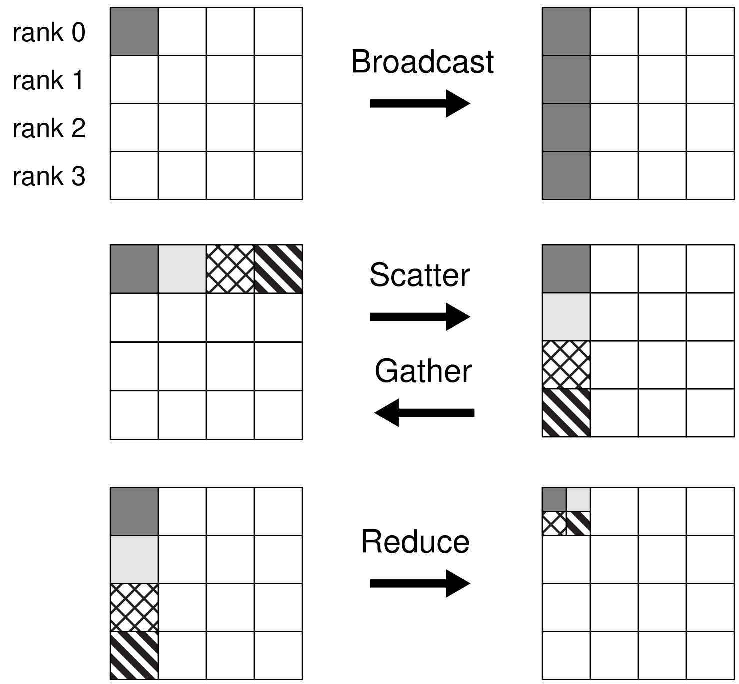

Figure 5: Primitive Collective Operations

cesses (including itself). After the operation, the root has sent sendcount elements to each process in increasing rank order. Rank 0 receives the first sendcount elements from the send buffer. Rank 1 receives the second sendcount elements from the send buffer, and so on.

Gather

```
MPI_Gather (void *sendbuf, int sendcount,
     MPI_Datatype sendtype, void *recvbuf,
     int recvcount, MPI_Datatype recvtype,
     int root, MPI_Comm comm);
```

MPI_Gather() is a many-to-one collective operation and is a complete reverse of the description of MPI_Scatter().

Reduce

```
MPI_Reduce (void *sendbuf, void *recvbuf,
     int count, MPI_Datatype dtype, MPI_Op op,
     int root, MPI_Comm comm);
```

MPI_Reduce() is also a many-to-one collective operation. All processes specify the same count and reduction operation. After the reduction, all processes have sent count elements from their send buffer to the root process.

Elements from corresponding send buffer locations are combined pair-wise to yield a single corresponding element in the root process's receive buffer. The full reduction expression over all processes is always associative and may or may not be commutative. Application specific reduction operations can be defined at runtime. MPI provides several pre-defined operations, all of which are commutative. They can be used only with sensible MPI pre-defined datatypes.

`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_LAND`	logical and
`MPI_BAND`	bitwise and
`MPI_LOR`	logical or
`MPI_BOR`	bitwise or
`MPI_LXOR`	logical exclusive or
`MPI_BXOR`	bitwise exclusive or

The following code fragment illustrates the primitive collective operations together in the context of a statically partitioned regular data domain (e.g., 1-D array). The global domain information is initially obtained by the root process (e.g., rank 0) and is broadcast to all other processes. The initial dataset is also obtained by the root and is scattered to all processes. After the computation phase, a global maximum is returned to the root process followed by the new dataset itself.

```
/*
 * parallel programming with a single control process
 */
      int               root;
      int               rank, size;
      int               i;
      int               full_domain_length;
      int               sub_domain_length;
      double            *full_domain, *sub_domain;
      double            local_max, global_max;
      root = 0;
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
/*
 * Root obtains full domain and broadcasts its length.
 */
      if (rank == root) {
            get_full_domain(&full_domain,
                  &full_domain_length);
      }
      MPI_Bcast(&full_domain_length, 1 MPI_INT, root,
            MPI_COMM_WORLD);
/*
 * Distribute the initial dataset.
 */
      sub_domain_length = full_domain_length / size;
      sub_domain = (double *) malloc(sub_domain_length *
            sizeof(double));
      MPI_Scatter(full_domain, sub_domain_length,
            MPI_DOUBLE, sub_domain, sub_domain_length,
            MPI_DOUBLE, root, MPI_COMM_WORLD);
/*
 * Compute the new dataset.
 */
      compute(sub_domain, sub_domain_length, &local_max);
/*
 * Reduce the local maxima to one global maximum
 * at the root.
 */
      MPI_Reduce(&local_max, &global_max, 1, MPI_DOUBLE,
            MPI_MAX, root, MPI_COMM_WORLD);
/*
 * Collect the new dataset.
 */
      MPI_Gather(sub_domain, sub_domain_length, MPI_DOUBLE,
            full_domain, sub_domain_length, MPI_DOUBLE,
            root, MPI_COMM_WORLD);
```

`MPI_Comm_dup`	Duplicate communicator with new context.
`MPI_Comm_split`	Split into categorized sub-groups.
`MPI_Comm_free`	Release a communicator.
`MPI_Comm_remote_size`	Count intercomm. remote group members.
`MPI_Intercomm_merge`	Create an intracomm. from an intercomm.

`MPI_Comm_compare`	Compare two communicators.
`MPI_Comm_create`	Create a communicator with a given group.
`MPI_Comm_test_inter`	Test for intracommunicator or intercommunicator.
`MPI_Intercomm_create`	Create an intercommunicator.

`MPI_Group_size`	Get number of processes in group.
`MPI_Group_rank`	Get rank of calling process.
`MPI_Group_translate_ranks`	Processes in group A have what ranks in B?
`MPI_Group_compare`	Compare membership of two groups.
`MPI_Comm_group`	Get group from communicator.
`MPI_Group_union`	Create group with all members of 2 others.
`MPI_Group_intersection`	Create with common members of 2 others.
`MPI_Group_difference`	Create with the complement of intersection.
`MPI_Group_incl`	Create with specific members of old group.
`MPI_Group_excl`	Create with the complement of incl.
`MPI_Group_range_incl`	Create with ranges of old group members.
`MPI_Group_range_excl`	Create with the complement of range_incl.
`MPI_Group_free`	Release a group object.

Creating Communicators

A communicator could be described simply as a process group. Its creation is synchronized and its membership is static. There is no period in user code where a communicator is created but not all its members have joined. These qualities make communicators a solid parallel programming foundation. Three communicators are prefabricated before the user code is first called: MPI_COMM_WORLD, MPI_COMM_SELF and MPI_COMM_PARENT. See *Basic Concepts*.

Communicators carry a hidden synchronization variable called the context. If two processes agree on source rank, destination rank and message tag, but use different communicators, they will not synchronize. The extra synchronization means that the global software industry does not have to divide, allocate or reserve tag values. When writing a library or a module of an application, it is a good idea to create new communicators, and hence a private synchronization space. The simplest MPI routine for this purpose is MPI_Comm_dup(), which duplicates everything in a communicator, particularly the group membership, and allocates a new context.

```
MPI_Comm_dup (MPI_comm comm, MPI_comm *newcomm);
```

Applications may wish to split into many subgroups, sometimes for data parallel convenience (i.e. a row of a matrix), sometimes for functional grouping (i.e. multiple distinct programs in a dataflow architecture). The group membership can be extracted from the communicator and manipulated by an entire suite of MPI routines. The new group can then be used to create a new communicator. MPI also provides a powerful routine, MPI_Comm_split(), that starts with a communicator and results in one or more new communicators. It combines group splitting with communicator creation and is sufficient for many common application requirements.

```
MPI_Comm_split (MPI_comm comm, int color,
     int key, MPI_Comm *newcomm);
```

The color and key arguments guide the group splitting. There will be one new communicator for each value of color. Processes providing the same value for color will be grouped in the same communicator. Their ranks in the new communicator are determined by sorting the key arguments. The lowest value of key will become rank 0. Ties are broken by rank in the old communicator. To preserve relative order from the old communicator, simply use the same key everywhere.

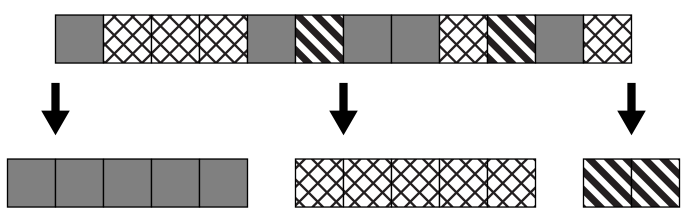

Figure 6: Communicator Split

A communicator is released by MPI_Comm_free(). Underlying system resources may be conserved by releasing unwanted communicators.

```
MPI_Comm_free (MPI_Comm *comm);
```

Inter-communicators

An intercommunicator contains two groups: a local group in which the owning process is a member and a remote group of separate processes. The remote process group has the mirror image intercommunicator - the groups are flipped. Spawning new processes creates an intercommunicator. See *Dynamic Processes*. MPI_Intercomm_merge() creates an intracommunicator (the common form with a single group) from an intercommunicator. This is often done to permit collective operations, which can only be done on intracommunicators.

```
MPI_Intercomm_merge (MPI_Comm intercomm,
     int high, MPI_Comm *newintracomm);
```

The new intracommunicator group contains the union of the two groups of the intercommunicator. The operation is collective over both groups. Rank ordering within the two founding groups is maintained. Ordering between the two founding groups is controlled by the high parameter, a boolean value. The intercommunicator group that sets this parameter true will occupy the higher ranks in the intracommunicator.

The number of members in the remote group of an intercommunicator is obtained by MPI_Comm_remote_size().

```
MPI_Comm_remote_size (MPI_Comm comm, int *size);
```

Fault Tolerance

Some MPI implementations may invalidate a communicator if a member process dies. The MPI library may raise an error condition on any attempt to use a dead communicator, including requests in progress whose communicator suddenly becomes invalid. These faults would then be detectable at the application level by setting a communicator’s error handler to MPI_ERRORS_RETURN (See *Miscellaneous MPI Features*).

A crude but portable fault tolerant master/slave application can be constructed by using the following strategy:

- Spawn processes in groups of one.
- Set the error handler for the parent / child intercommunicators to MPI_ERRORS_RETURN.
- If a communication with a child returns an error, assume it is dead and free the intercommunicator.
- Spawn another process, if desired, to replace the dead process. See *Dynamic Processes*.

`MPI_Cart_create`	Create cartesian topology communicator.
`MPI_Dims_create`	Suggest balanced dimension ranges.
`MPI_Cart_rank`	Get rank from cartesian coordinates.
`MPI_Cart_coords`	Get cartesian coordinates from rank.
`MPI_Cart_shift`	Determine ranks for cartesian shift.

`MPI_Cart_sub`	Split into lower dimensional sub-grids.
`MPI_Graph_create`	Create arbitrary topology communicator.
`MPI_Topo_test`	Get type of communicator topology.
`MPI_Graphdims_get`	Get number of edges and nodes.
`MPI_Graph_get`	Get edges and nodes.
`MPI_Cartdim_get`	Get number of dimensions.
`MPI_Cart_get`	Get dimensions, periodicity and local coordinates.
`MPI_Graph_neighbors_count`	Get number of neighbors in a graph topology.
`MPI_Graph_neighbors`	Get neighbor ranks in a graph topology.
`MPI_Cart_map`	Suggest new ranks in an optimal cartesian mapping.
`MPI_Graph_map`	Suggest new ranks in an optimal graph mapping.

Process Topologies

MPI is a process oriented programming model that is independent of underlying nodes in a parallel computer. Nevertheless, to enhance performance, the data movement patterns in a parallel application should match, as closely as possible, the communication topology of the hardware. Since it is difficult for compilers and message-passing systems to guess at an application's data movement, MPI allows the application to supply a topology to a communicator, in the hope that the MPI implementation will use that information to identify processes in an optimal manner.

For example, if the application is dominated by Cartesian communication and the parallel computer has a cartesian topology, it is preferable to align the distribution of data with the machine, and not blindly place any data coordinate at any node coordinate.

MPI provides two types of topologies, the ubiquitous cartesian grid, and an arbitrary graph. Topology information is attached to a communicator by creating a new communicator. MPI_Cart_create() does this for the cartesian topology.

```
MPI_Cart_create (MPI_Comm oldcomm, int ndims,
     int *dims, int *periods, int reorder,
     MPI_Comm *newcomm);
```

The essential information for a cartesian topology is the number of dimensions, the length of each dimension and a periodicity flag (does the dimension wrap around?) for each dimension. The reorder argument is a flag that indicates if the application will allow a different ranking in the new topology communicator. Reordering may make coordinate calculation easier for the MPI implementation.

With a topology enhanced communicator, the application will use coordinates to decide source and destination ranks. Since MPI communication routines still use ranks, the coordinates must be translated into a rank and vice versa. MPI eases this translation with MPI_Cart_rank() and MPI_Cart_coords().

```
MPI_Cart_rank (MPI_comm comm, int *coords,
     int *rank);
MPI_Cart_coords (MPI_Comm comm, int rank,
     int maxdims, int *coords);
```

To further assist process identification in cartesian topology applications, MPI_Cart_shift() returns the ranks corresponding to common neighbourly shift communication. The direction (dimension) and relative distance are input arguments and two ranks are output arguments, one on each side of the calling process along the given direction. Depending on the periodicity of the cartesian topology associated with the given communicator, one or both ranks may be returned as MPI_PROC_NULL, indicating a shift off the edge of the grid.

```
MPI_Cart_shift (MPI_Comm comm, int direction,
     int distance, int *rank_source,
     *int rank_dest);
```

Consider a two dimensional cartesian dataset. The following code skeleton establishes a corresponding process topology for any number of processes, and then creates a new communicator for collective operations on the first column of processes. Finally, it obtains the ranks which hold the previous and next rows, which would lead to data exchange.

```
int             mycoords[2];
int             dims[2];
int             periods[2] = {1, 0};
int             rank_prev, rank_next;
int             size;
MPI_Comm        comm_cart;
MPI_Comm        comm_col1;
```

```
/*
 * Create communicator with 2D grid topology.
 */
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Dims_create(size, 2, dims);
      MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1,
            &comm_cart);
/*
 * Get local coordinates.
 */
      MPI_Comm_rank(comm_cart, &rank);
      MPI_Cart_coords(comm_cart, rank, 2, mycoords);
/*
 * Build new communicator on first column.
 */
      if (mycoords[1] == 0) {
            MPI_Comm_split(comm_cart, 0, mycoords[0],
                  &comm_col1);
      } else {
            MPI_Comm_split(comm_cart, MPI_UNDEFINED, 0,
                  &comm_col1);
}
/*
 * Get the ranks of the next and previous rows, same column.
 */
      MPI_Cart_shift(comm_cart, 0, 1, &rank_prev,
            &rank_next);
```

MPI_Dims_create() suggests the most balanced ("square") dimension ranges for a given number of nodes and dimensions.

A good reason for building a communicator over a subset of the grid, in this case the first column in a mesh, is to enable the use of collective operations. See *Collective Message-Passing*.

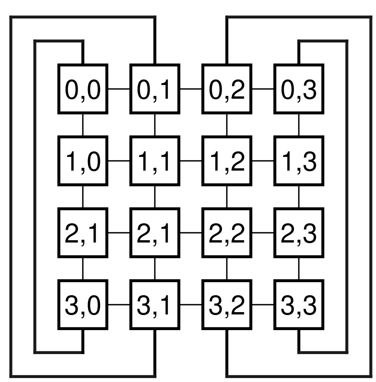

Figure 7: 2D Cartesian Topology

`MPI_Spawn`	Start copies of one program.
`MPI_Spawn_multiple`	Start multiple programs.
`MPI_Port_open`	Obtain a connection point for a server.
`MPI_Port_close`	Release a connection point.
`MPI_Accept`	Accept a connection from a client.
`MPI_Connect`	Make a connection to a server.
`MPI_Name_publish`	Publish a connection point under a service name.
`MPI_Name_unpublish`	Stop publishing a connection point.
`MPI_Name_get`	Get connection point from service name.
`MPI_Info_create`	Create a new info object.
`MPI_Info_set`	Store a key/value pair to an info object.
`MPI_Info_get`	Read the value associated with a stored key.
`MPI_Info_get_valuelen`	Get the length of a key value.
`MPI_Info_get_nkeys`	Get number of keys stored with an info object.
`MPI_Info_get_nthkey`	Get the key name in a sequence position.
`MPI_Info_dup`	Duplicate an info object.
`MPI_Info_free`	Destroy an info object.
`MPI_Info_delete`	Remove a key/value pair from an info object.

Process Creation

Due to the static nature of process groups in MPI (a virtue), process creation must be done carefully. Process creation is a collective operation over a given communicator. A group of processes are created by one call to MPI_Spawn(). The child processes start up, initialize and communicate in the traditional MPI way. They must begin by calling MPI_Init(). The child group has its own MPI_COMM_WORLD which is distinct from the world communicator of the parent group.

```
MPI_Spawn (char program[], char *argv[], int
     maxprocs, MPI_Info info, int root, MPI_Comm,
     parents, MPI_Comm *children, int errs[]);
```

How do the parents communicate with their children? The natural mechanism for communication between two groups is the intercommunicator. An intercommunicator whose remote group contains the children is returned to the parents in the second communicator argument of MPI_Spawn(). The children get the mirror communicator, whose remote group contains the parents, as the pre-defined communicator MPI_COMM_PARENT. In the application's original process world that has no parent, the remote group of MPI_COMM_PARENT is of size 0. See *Creating Communicators*.

The maxprocs parameter is the number of copies of the single program that will be created. Each process will be passed command line arguments consisting of the program name followed by the arguments specified in the argv parameter. (The argv parameter should not contain the program name.) The program name, maxprocs and argv are only significant in the parent process whose rank is given by the root parameter. The result of each individual process spawn is returned through the errs parameter, an array of MPI error codes.

Portable Resource Specification

New processes require resources, beginning with a processor. The specification of resources is a natural area where the MPI abstraction succumbs to the underlying operating system and all its domestic customs and conventions. It is thus difficult if not impossible for an MPI application to make a detailed resource specification and remain portable. The info parameter to MPI_Spawn is an opportunity for the programmer to choose control over portability. MPI implementations are not required to interpret this argument. Thus the only portable value for the info parameter is MPI_INFO_NULL.

Consult each MPI implementation's documentation for (non-portable) features within the info parameter and for the default behaviour with MPI_INFO_NULL.

A common and fairly abstract resource requirement is simply to fill the available processors with processes. MPI makes an attempt, with no guarantees of accuracy, to supply that information through a pre-defined attribute called MPI_UNIVERSE_SIZE, which is cached on MPI_COMM_WORLD. In typical usage, the application would subtract the value associated with MPI_UNIVERSE_SIZE from the current number of processes, often the size of MPI_COMM_WORLD. The difference is the recommended value for the maxprocs parameter of MPI_Spawn(). See *Miscellaneous MPI Features* on how to retrieve the value for MPI_UNIVERSE_SIZE.

`MPI_Errhandler_create`	Create custom error handler.
`MPI_Errhandler_set`	Set error handler for communicator.
`MPI_Error_string`	Get description of error code.
`MPI_Error_class`	Get class of error code.
`MPI_Abort`	Abnormally terminate application.
`MPI_Attr_get`	Get cached attribute value.
`MPI_Wtime`	Get wall clock time.
`MPI_Errhandler_get`	Get error handler from communicator.
`MPI_Errhandler_free`	Release custom error handler.
`MPI_Get_processor_name`	Get the caller's processor name.
`MPI_Wtick`	Get wall clock timer resolution.
`MPI_Get_version`	Get the MPI version numbers.
`MPI_Keyval_create`	Create a new attribute key.
`MPI_Keyval_free`	Release an attribute key.
`MPI_Attr_put`	Cache an attribute in a communicator.
`MPI_Attr_delete`	Remove cached attribute.

Miscellaneous MPI Features

Error Handling

An error handler is a software routine which is called when a error occurs during some MPI operation. One handler is associated with each communicator and is inherited by created communicators which derive from it. When an error occurs in an MPI routine that uses a communicator, that communicator's error handler is called. An application's initial communicator, MPI_COMM_WORLD, gets a default built-in handler, MPI_ERRORS_ARE_FATAL, which aborts all tasks in the communicator.

An application may supply an error handler by first creating an MPI error handler object from a user routine.

```
MPI_Errhandler_create (void (*function)(),
     MPI_Errhandler *errhandler);
```

Error handler routines have two pre-defined parameters followed by implementation dependent parameters using the ANSI C <stdargs.h> mechanism. The first parameter is the handler's communicator and the second is the error code describing the problem.

```
void function (MPI_Comm *comm, int *code, ...);
```

The error handler object is then associated with a communicator by MPI_Errhandler_set().

```
MPI_Errhandler_set (MPI_Comm comm,
     MPI_Errhandler errhandler);
```

A second built-in error handler is MPI_ERRORS_RETURN, which does nothing and allows the error code to be returned by the offending MPI routine where it can be tested and acted upon. In C the error code is the return value of the MPI function. In Fortran the error code is returned through an error parameter to the MPI subroutine.

```
MPI_Error_string (int code, char *errstring,
     int *resultlen);
```

Error codes are converted into descriptive strings by MPI_Error_string(). The user provides space for the string that is a minimum of MPI_MAX_ERROR_STRING characters in length. The actual length of the returned string is returned through the resultlen argument.

MPI defines a list of standard error codes (also called error classes) that can be examined and acted upon by portable applications. All additional error codes, specific to the implementation, can be mapped to one of the standard error codes. The idea is that additional error codes are variations on one of the standard codes, or members of the same error class. Two standard error codes catch any additional error code that does not fit this intent: MPI_ERR_OTHER (doesn't fit but convert to string and learn something) and MPI_ERR_UNKNOWN (no clue). Again, the goal of this design is portable, intelligent applications.

The mapping of error code to standard error code (class) is done by MPI_Error_class().

```
MPI_Error_class (int code, int class);
```

Attribute Caching

MPI provides a mechanism for storing arbitrary information with a communicator. A registered key is associated with each piece of information and is used, like a database record, for storage and retrieval. Several keys and associated values are pre-defined by MPI and stored in MPI_COMM_WORLD.

MPI_TAG_UB	maximum message tag value
MPI_HOST	process rank on user's local processor
MPI_IO	process rank that can fully accomplish I/O
MPI_WTIME_IS_GLOBAL	Are clocks synchronized?
MPI_UNIVERSE_SIZE	#processes to fill machine

All cached information is retrieved by calling MPI_Attr_get() and specifying the desired key.

```
MPI_Attr_get (MPI_Comm comm, int keyval,
     void *attr_val, int *flag);
```

The flag parameter is set to true by MPI_Attr_get() if a value has been stored the specified key, as will be the case for all the pre-defined keys.

Timing Performance measurement is assisted by MPI_Wtime() which returns an elapsed wall clock time from some fixed point in the past.

```
double MPI_Wtime (void);
```

LAM / MPI Extensions

LAM includes several functions beyond the MPI standard that programmers may find useful during the development phase of a software application. They can be used in the final product, though portability would obviously be compromised. One of the extensions is actually an MPI portable library (see *Collective I/O*) which can operate with other MPI implementations. This library is a distinct product from LAM and must be obtained and compiled separately. The other extensions are all intrinsic to LAM.

Some of the extended routines that integrate naturally with MPI have names that begin with MPIL_. Similar functionality will, in certain cases, be found in later versions of the MPI standard. Other routines, which are distinct from MPI concepts and objects, begin with lam_.

`lam_rfopen`	Open a file.
`lam_rfclose`	Close a file.
`lam_rfread`	Read from a file.
`lam_rfwrite`	Write to a file.
`lam_rflseek`	Change position in a file.
`lam_rfaccess`	Check permissions of a file.
`lam_rfmkdir`	Create directory.
`lam_rfchdir`	Change working directory.
`lam_rffstat`	Get status on file descriptor.
`lam_rfstat`	Get status on named file.
`lam_rfdup`	Duplicate file descriptor.
`lam_rfdup2`	Duplicate & place file descriptor.
`lam_rfsystem`	Issue a shell command.
`lam_rfrmdir`	Remove a directory.
`lam_rfunlink`	Remove a file.
`lam_rfgetwd`	Get working directory.
`lam_rfftruncate`	Set length of file descriptor.
`lam_rftruncate`	Set length of named file.

Remote File Access

A node's file system can be accessed via remote file functions having a POSIX-like interface. LAM does not provide a file system, only remote access to a file system from any node.

File pathnames refer to files on the origin node by default. However, a specific nodeid can be attached to a pathname with the following syntax:

```
nodeid:path
```

Each LAM process may have a limited number of simultaneously open LAM file descriptors. All LAM file functions involve message-passing using the same links, buffers and other resources as an application.

LAM prohibits opening of slow devices (such as terminals) for input.

Some LAM specific features of remote file access are controlled by additional flags in the flags argument of the lam_rfopen() routine. These flags are listed below.

`LAM_O_LOCK`	Lock the file descriptor into the remote file server's open descriptor cache. See the manual page lam_rfposix().

`LAM_O_REUSE`	Reuse existing open file descriptor for matching pathname and open flags - if found. This is useful for asynchronous access to one open file with one file pointer.
`LAM_O_1WAY`	Write to the file without waiting for completion or return code. This greatly increases write performance but should only be used on a debugged application.

Portability and Standard I/O

LAM does not conflict with the native operating system's file interface. Thus, open() is a direct UNIX routine (LAM is uninvolved) and operates on the file system of the node on which it is invoked. On remote nodes, a process's pre-opened UNIX standard output (UNIX file handle 1 or stdout) and UNIX standard error (UNIX file handle 2 or stderr) are redirected to LAM as there is no remote terminal. LAM uses the remote file access facility to move data from these two sources to the node and terminal from which the application was launched - the user's local node. It is not possible to read from UNIX standard input (UNIX file handle 0, or stdin) on remote nodes.

Processes on the local node also have access to UNIX standard output and error. Unlike remote processes, local processes can read from UNIX standard input.

The UNIX standard I/O terminations may be redirected by using the normal shell redirections with *mpirun*. See *Executing MPI Programs*.

```
% mpirun my_app > log
```

`CBX_Open`	Open a file for MPI Cubix access.
`CBX_Close`	Close an MPI Cubix file.
`CBX_Read`	Read in either single or multiple mode.
`CBX_Write`	Write in either single or multiple mode.
`CBX_Lseek`	Seek in either single or multiple mode.
`CBX_Order`	Change the order of multiple access.
`CBX_Singl`	Switch file access to single mode.
`CBX_Multi`	Switch file access to multiple mode.
`CBX_Is_singl`	Is the file in single mode?
`CBX_Is_multi`	Is the file in multiple mode?

Collective I/O

MPI Cubix a loosely synchronous, collective I/O library based on a research development of the same name at the California Institute of Technology. This Cubix is integrated with the concepts of MPI communicators and datatypes. The members of a communicator group participate collectively in the I/O operation. Data is transferred as a count of elements of a given datatype, just as in MPI message-passing.

All file access routines eventually translate to POSIX operations on a single file. Only one process in the communicator group invokes the actual POSIX operation. The POSIX file operation bindings are also reflected in the bindings of the MPI Cubix routines, tempered with MPI objects.

There are two different MPI Cubix access methods that solve two common file read/write problems in data parallel programming.

single
: All processes execute the same file routine with the same amount of identical data. The data from only one (arbitrary) process is transferred. This is useful when all processes want to read a global value from a file, or write a global value to a file. It is especially convenient during output to a terminal. All nodes print an error message and it appears once on the terminal.

multiple
: All processes execute the same file routine with different amounts of different data. All the data from all the processes is transferred, but the order of transfer is strictly controlled. By default, process rank 0 will transfer first and the sequence continues until the highest rank transfers last. This is useful in decomposing a data structure during read so that the right nodes get the right subset of

data and in recomposing a data structure during write so that the data structure is not jumbled.

Without Cubix file access, an application often needs a controlling program to manage the parallel processes and filter I/O. Cubix can eliminate the need for a control program. Without synchronization, a message written by N nodes appears N times on the terminal. A decomposed data structure written to a file appears in a random order.

MPI Cubix file descriptors are distinct from LAM remote file descriptors and the file descriptors of the native operating system. An MPI Cubix file descriptor is returned from CBX_Open(). The access method is chosen by one of the special flags, CBX_O_SINGL or CBX_O_MULTI. The owner of the file, the one process that will operate on it at the POSIX level, is chosen in another argument to CBX_Open().

```
#include <fcntl.h>
#include <cbx.h>

int CBX_Open (const char *name, int flags,
     int mode, int owner, MPI_Comm comm);
int CBX_Close (int fd);
```

The access method being used on an open MPI Cubix file can be queried and changed at any time. The change routines are collective. The inquiry routines are not.

```
int CBX_Multi (int fd);
int CBX_Singl (int fd);
int CBX_Is_multi (int fd);
int CBX_Is_singl (int fd);
```

CBX_Read() and CBX_Write() transfer data from and to an open MPI Cubix file. An MPI datatype is among the arguments. The length of the data buffer is a count of elements of the given datatype. Only contiguous data is transferred. If the MPI datatype contains holes, they are also transferred.

```
int CBX_Read (int fd, void *buffer, int count,
     MPI_Datatype dtype);
int CBX_Write (int fd, void *buffer, int count,
     MPI_Datatype dtype);
```

The CBX_Order() routine changes the default order of process data transfer in the MPI Cubix multiple method. Each process specifies a unique sequence number from 0 to N-1, where N is the size of the communicator.

```
int CBX_Order (int fd, int newrank);
```

Cubix Example

```
/*
 * Read and decompose a 1-D array of reals
 * across a 1-D array of processes.
 * First read array size in singl then array in multi.
 * Assume the array length decomposes evenly.
 */
static float            *data;
main(argc, argv)
int                     argc;
char                    *argv[];
{
      int               fd;
      int               glob_len, local_len;
      int               nread;
      int               size;
      MPI_Init(&argc, &argv);
/*
 * Open the file first with Cubix single method.
 * The file will be owned by process rank 0.
 * This is not an error handling tutorial.
 */
      fd = CBX_Open("data", O_RDONLY | CBX_O_SINGL, 0, 0,
            MPI_COMM_WORLD);
/*
 * Read the global (total) length of the array.
 */
      CBX_Read(fd, &glob_len, 1, MPI_INT);
/*
 * Switch to Cubix multiple method.
 */
      CBX_Multi(fd);
/*
 * Calculate the local length, allocate enough
 * space and read the local subset of the data.
 */
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      local_len = glob_len / size;
      data = (float *) malloc(local_len * sizeof(float));
      CBX_Read(fd, data, local_len, MPI_FLOAT);
      CBX_Close(fd);
      MPI_Finalize();
}
```

lam_ksignal	Install a signal handler.
lam_ksigblock	Block selected signals.
lam_ksigsetmask	Set entire blocking mask.
lam_ksigretry	Retry request after selected signals.
lam_ksigsetretry	Set entire retry mask.
lam_ksigmask	Create signal mask.
MPIL_Signal	Deliver a signal to a process.

Signal Handling

LAM provides a UNIX-like signal package. The signals are different and their usage does not conflict with the underlying operating system.

Some signals are used internally by the system. Some have useful default options and others are completely left to the user. The most useful signal is the one that obliges a process to terminate itself. Signals are defined in <lam_ksignal.h>.

LAM_SIGTRACE	unload trace data
LAM_SIGUDIE	terminate
LAM_SIGARREST	suspend execution
LAM_SIGRELEASE	resume execution
LAM_SIGA	user defined (default ignored)
LAM_SIGB	user defined (default ignored)
LAM_SIGFUSE	node about to die
LAM_SIGSHRINK	another node has died

The lam_ksignal(), lam_ksigblock() and lam_ksigsetmask() functions operate identically to their UNIX counterparts. A LAM or MPI routine interrupted by a signal before completion is automatically retried. With the lam_ksigretry() and lam_ksigsetretry() functions, which operate similarly to lam_ksigblock() and lam_ksigsetmask() respectively, the user can disable automatic system call retry and receive an error code instead.

Signal Delivery

MPIL_Signal() delivers a signal to a process identified by a communicator and a rank. The signal number argument is taken from the list defined above.

```
MPIL_Signal (MPI_Comm comm, int rank, int signo);
```

`MPIL_Comm_id`	Get communicator identifier.
`MPIL_Comm_gps`	Get LAM coordinates for an MPI process.
`MPIL_Type_id`	Get datatype identifier.
`MPIL_Trace_on`	Enable trace collection.
`MPIL_Trace_off`	Disable trace collection.

Debugging and Tracing

LAM places great emphasis on debugging through extensive monitoring capabilities. Opaque objects in MPI make it difficult for the user to cross reference the information presented by LAM debugging tools with the values within a running process. If *mpitask* (See *Process Monitoring and Control*) shows a process blocked on a communicator, it prints an identifying number for that communicator. The number is not defined by the MPI standard. It is implementation dependent information internal to the opaque communicator which the program cannot access using the standard API.

MPIL_Comm_id() and MPIL_Type_id() return the internal identifiers for communicators and datatypes, respectively.

```
MPIL_Comm_id (MPI_Comm comm, int *id);
MPIL_Type_id (MPI_Comm comm, int *id);
```

LAM / MPI extensions beginning with the lam_ prefix are LAM-centric. They operate on LAM node and process identifiers, not MPI communicators and ranks. MPIL_Comm_gps() obtains the LAM coordinates from MPI information.

```
MPIL_Comm_gps (MPI_Comm comm, int rank, int *nid,
     int *pid);
```

Execution trace collection for performance visualization and debugging purposes is enabled by *mpirun*. See *Executing MPI Programs*. To avoid information overload and huge trace files, a trace enabled application can toggle on and off actual trace collection so that only interesting phases of the computation are monitored.

```
MPIL_Trace_On (void);
MPIL_Trace_Off (void);
```

LAM Command Reference

Getting Started

Setting Up the UNIX Environment

Before running LAM you must establish certain environment variables and search paths for your shell on each machine in the multicomputer. Add the following commands or equivalent to your shell start-up file (.cshrc, assuming C shell). Do not add these to your .login as they would not be effective on remote machines when *rsh* is used to start LAM.

```
setenv LAMHOME <LAM installation directory>
set path = ($path $LAMHOME/bin)
```

The local system administrator, or the person who installed LAM, will know the location of the LAM installation directory. After editing the shell start-up file, invoke it to establish the new values. This is not necessary on subsequent logins to the UNIX system.

```
% source .cshrc
```

Each remote machine in the multicomputer must be reachable with the UNIX rsh command. rsh does not prompt for passwords and relies on special files on the remote machine (/etc/hosts.equiv and ~/.rhosts) to gain access. One of these files must be prepared to admit the selected user account for the remote machine. See the UNIX manual page for rsh on how to prepare these files.

Node Mnemonics

Many LAM commands require one or more nodeids. Nodeids are specified on the command line as n<list>, where <list> is a list of comma separated nodeids or nodeid ranges.

```
n1
n1,3,5-10
```

In addition to explicit node identification, LAM has special mnemonics that refer to special nodes or a group of nodes.

`h`	the local node where the command is typed (as in ‘here’)
`o`	the origin node where LAM was started with the *lamboot* command
`N`	all nodes
`C`	all nodes intended for application computing

Nodeids are established in the LAM multicomputer plan, called a boot schema (see *Writing a LAM Boot Schema*). LAM nodeids are always numbered consecutively beginning at 0 when the system is first started with lamboot. Thus the number of nodes in the boot schema defines the initial set of nodeids. If nodes are added or subtracted, the contiguous property of nodeids can end. See *Adding and Deleting LAM Nodes*.

Process Identification

LAM processes can be specified in two ways: by process identifier (from the underlying operating system) or by LAM process index. PIDs are specified on the command line as p<list>, where <list> is a list of comma separated PIDs or PID ranges.

```
p5158
p5158,5160,5200-5210
```

Process indices are specified on the command line as i<list>, where <list> is a list of comma separated indices or index ranges.

```
i8
i8-12,14
```

MPI processes are normally labelled by the LAM / MPI status reporting commands, *mpitask* and *mpimsg*, with their global rank in the MPI_COMM_WORLD communicator. With the possibility of multiple concurrent applications, spawned processes, and the need to use LAM node / process identification with LAM process control commands, a supplementary labelling scheme is available. It is known as the GPS, for Global Positioning System, because it absolutely distinguishes a process from all others in a LAM system. The GPS contains the process index and nodeid.

On-line Help

To print a brief summary of the syntax and options of any LAM command, execute the command with the -h option.

```
% recon -h
```

Detailed information on each command (and most programming functions) is available from on-line manual pages. They are an important supplementary reference to this document.

```
% man recon
```

`hcc`	wrapper for local C compiler
`hcp`	wrapper for local C++ compiler
`hf77`	wrapper for local Fortran compiler

Compiling MPI Programs

Objects and binaries are built with the native compiler and linker available, hopefully, on all of the LAM nodes, or at least on one machine of each architecture and operating system type. The *hcc* tool is a wrapper that invokes the native C compiler driver (e.g. cc) and provides the paths to the LAM header files and libraries and implicitly links all LAM libraries. The MPI library is linked explicitly. All options presented to hcc are passed through to the native compiler driver.

```
% hcc -o appl appl.c -lmpi
% hcp -o appl appl.c -lmpi
```

By default, hcc uses the compiler driver that was used to build LAM and specified in the LAM configuration file. A different C compiler can be specified in the LAMHCC environment variable.

In case the C and C++ compilers are different, a separate *hcp* wrapper is provided for C++.

Unlike the C and C++ wrappers, the Fortran wrapper, *hf77*, does not insert an option to search LAM's header file directory. This is because not all Fortran compiler drivers support that option and the Fortran include statement may be required instead to bring in the MPI header file, mpif.h. Note that mpif.h is a Fortran source file, but all other LAM header files intended for Fortran use contain C preprocessor code. The C preprocessor may need to be run explicitly if the Fortran driver does not do so automatically.

Care should be taken not to confuse object files and binaries produced on heterogeneous nodes in the multicomputer. In such situations, it can be a good idea to append the machine or CPU name to the object and executable file names in order to distinguish between them.

```
{sparc}% hcc -o appl.sparc appl.c
{sgi}% hcc -o appl.sgi appl.c
```

`recon`	Verify multicomputer is ready to run LAM.
`lamboot`	Start a LAM multicomputer session.
`tping`	Check communication to given node.
`wipe`	Terminate a LAM session.

Starting LAM

recon

The topology of a multicomputer running LAM is a totally connected graph. Thus it is only necessary for the user to specify the machines to be included in the multicomputer in order to start LAM. The boot schema (ASCII file) serves this purpose. See *Writing a LAM Boot Schema*.

The *recon* tool verifies that LAM can be started on each intended node. Several conditions must exist before a machine can remotely run the software.

- The machine address must be reachable via the network.
- The user must be able to remotely execute on the machine with rsh. Remote host permission must be provided in either /etc/hosts.equiv or the remote user's .rhosts file.
- The remote user's shell must have a search path that will locate LAM executables.
- The remote shell's start-up file must not print anything to standard error when invoked non-interactively.

lamboot

The *lamboot* tool starts a LAM session for the individual user. The -v option prints a message before each start-up step is attempted. The boot schema file is the primary argument to lamboot.

`% lamboot -v` <boot schema>

Fault Tolerance

`-x` Enable fault detection and recovery. Exchange periodic "heartbeat" messages between all nodes.

LAM considers a node to be dead after repeated retransmissions of a message packet go unacknowledged. By default, no action is taken and retransmissions continue indefinitely. With the -x option to lamboot, LAM initiates a procedure to remove the dead node from the multicomputer. The dead nodeid becomes invalid. All other nodeids remain unchanged - holes may develop in the nodeid list. Finally, a signal is set to all application processes on all nodes, notifying them of the failure. The runtime system of each process must, at a minimum, flush a cached table of nodeids, so that it can read updated information from the LAM daemon. Users can trap this signal

(LAM_SIGSHRINK) and take further action particular to the application. See *Signal Handling*.

tping Hands-on control and monitoring are a hallmark of LAM. The simplest command, *tping*, is a confidence building validity check that begins to dispel the black box nature of parallel environments. tping echoes a message to a destination node, or a multicast destination. It is time to restart the session if this command hangs.

```
% tping n0
% tping N
```

wipe To terminate a LAM session, use the *wipe* tool. To restart LAM after a system failure, execute wipe followed by lamboot.

```
% wipe -v <boot schema>
```

`mpirun`	Run an MPI application.
`lamclean`	Terminate and clean up all LAM processes.

Executing MPI Programs

mpirun

An MPI application is started with one invocation of *mpirun*. The programs, number of processes and computing resources are specified in an application schema, a separate file whose name is given to mpirun. Simple SPMD applications can be started from the mpirun command line. MPI processes locate each other through the abstract concepts of communicator and process rank. It is mpirun that provides the hard information on nodeids and process IDs to build the pre-defined MPI_COMM_WORLD communicator.

% mpirun -v my_app_schema

Application Schema

An application schema contains one line for every different program that constitutes the application. For each program, three important directions are given, using options that duplicate the syntax of the mpirun command line:

-s <nodeid>	the node in whose file space the executable program file can be found - Without this option, LAM is directed to look for programs on the same node where they will run.
<nodeids>	the nodes on which the program will run - Without this option, LAM will use all the nodes.
-c <#>	the number of processes to create across the given nodes - Without this option, LAM will create one process on each of the given nodes.

These same options are used on the mpirun command line if the application consists of only one program. The presence of one or more of these options tells mpirun that the filename is an executable program. Otherwise the filename is assumed to be an application schema and is parsed accordingly.

```
#
# sample application schema
#
master h
slave N –s h
```

The above example runs the ‘master’ on the local node (the same node where mpirun is invoked) and ‘slave’ on all the nodes, after taking the ‘slave’ executable file from the local node and shipping it to all nodes. The shipped program is stored in the /tmp directory and deleted when the process dies.

Locating Executable Files

LAM searches for executables files on the source node, as modified by the -s option, by using the list of directories defined by the PATH environment variable. The treatment of a '.' path is special. On the local node invoking mpirun, '.' is the working directory of mpirun. On remote nodes, it is the user's home directory.

Other mpirun options enable powerful functionality within LAM's MPI library.

Direct Communication

`-c2c` Bypass the LAM daemon for MPI communication. Use an optimal protocol to directly connect MPI processes.

The "client to client" feature of the MPI library derives the most speed[1] out of the underlying hardware at the expense of monitoring and control. Messages that have been delivered but not received are buffered with the receiver. It is intended that applications would be debugged first with the daemon and then run in production using direct communication[2].

Guaranteed Envelope Resources

`-nger` Disable GER protocol that protects message envelope queues. Do not detect and report resource overflow errors.

The "Guaranteed Envelope Resources" protocol provides the most robust MPI message delivery system. It protects communication between any process pair from interference from a third process. It prevents the posting of send operations that may not be delivered to the receiver due to lack of system resources (envelope resources) and thus fully respects the spirit of MPI's guarantee of message progress, which in turn reduces confusion in debugging ill-behaved applications.

In addition to protecting process-pair envelope queues, GER publishes the size of the queue so that programmers can know how far they can stress this resource before deadlocking or failing. GER fills a serious portability hole in MPI - knowing the resource limitations directly associated with message-passing. The minimum GER figure for LAM is configured when LAM is

1. The speed is constrained by the quality of the c2c module within the MPI library. Every machine can benefit from a customized solution.
2. Limits on underlying system resources (like file descriptors for a socket implementation) may constrain the scalability of applications using -c2c.

installed. See the manual page on MPI as well as the paper, *Robust MPI Message Delivery Through Guaranteed Resources* for a more detailed discussion.

Trace Collection

`-ton, -toff` Enable trace collection. Trace collection begins immediately after MPI_Init() with -ton, but is deferred until MPIL_Trace_on() with -toff.

The MPI library can generate execution traces detailing message-passing activity. The data can be used for performance tuning or advanced debugging. Actual trace generation is controlled by two switches, both of which must be in the on position to enable trace generation. Both the -ton and -toff options turn the first switch on for the entire run of the application. With -ton, the second switch begins in the on position after the application calls MPI_Init(). With -toff, the second switch begins in the off position and no traces are generated.

The second switch is toggled with runtime functions. See *Debugging and Tracing*. The purpose of beginning with the second switch off is to limit tracing to interesting phases of the computation. The purpose of the first switch is to allow an application to be traced without recompilation and to allow an application littered with trace toggling functions to disable tracing altogether and incur minimal overhead.

See *Collecting Trace Data* for how to assemble a single trace file after running a trace enabled application.

lamclean

An application that goes awry may leave many processes running or blocked, many messages unconsumed, and many resources allocated throughout the multicomputer. Although there are user interface commands to remove a user presence from every individual subsystem, taking care to invoke them all can become tedious. The *lamclean* command can be used when no user presence (processes, message, allocations, registrations) is desired on the multicomputer. The user essentially wants to start over without the longer delay of restarting the LAM session.

```
% lamclean
```

lamclean cannot be used when some or all nodes are not reachable due to catastrophic failure or complete buffer overflow that causes link jamming. If lamclean fails to return, it is time to use the *wipe* tool. See *Starting LAM*. It may be reassuring to use the *mpitask* and *mpimsg* commands to verify that lamclean did the job.

`mpitask`	Print status of MPI processes.
`doom`	Send a signal to a process.

Process Monitoring and Control

mpitask

Monitoring a process's execution state is a major aid in debugging multicomputer applications. This feature helps debug process synchronization and data transfer, the added dimension of parallel programming. The *mpitask* command prints information on MPI processes. With no arguments, all MPI processes on all nodes are reported. The report can be constrained by specifying nodes and LAM processes.

Consider the following example code, whose only purpose is to demonstrate LAM's monitoring capabilities:

```
/*
 * Create an interesting report for mpitask and mpimsg.
 */
#include <mpi.h>
#define ROWS            10
#define COLS            20
struct cell {
      int               code;
      double            coords[3];
};
static struct cell      mat[ROWS][COLS];
static int              blocklengths[3] = {1, 3, 1};
static MPI_Datatype     types[3] =
            {MPI_INT, MPI_DOUBLE, MPI_UB};
main(argc, argv)
int                     argc;
char                    *argv[];
{
      int               rank, size;
      MPI_Comm          newcomm;
      MPI_Datatype      dt_cell, dt_mat;
      MPI_Status        status;
      MPI_Aint          base;
      MPI_Aint          displacements[3];
      int               i, j;
/*
 * Initialize the matrix.
 */
      for (i = 0; i < ROWS; ++i) {
            for (j = 0; j < COLS; ++j) {
                  mat[i][j].code = i;
                  mat[i][j].coords[0] = (double) i;
```

```
                  mat[i][j].coords[1] = (double) j;
                  mat[i][j].coords[2] = (double) i * j
            }
      }
      MPI_Init(&argc, &argv);
/*
 * Create communicators for sub-groups.
 */
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_split(MPI_COMM_WORLD, rank % 3, 0, &newcomm);
/*
 * Build derived datatype for 2 columns of matrix.
 */
      MPI_Address(&mat[0][0].code, &displacements[0]);
      MPI_Address(mat[0][0].coords, &displacements[1]);
      MPI_Address(&mat[0][1].code, &displacements[2]);
      base = displacements[0];
      for (i = 0; i < 3; ++i) displacements[i] -= base;
      MPI_Type_struct(3, blocklengths, displacements,
                  types, &dt_cell);
      MPI_Type_vector(ROWS, 2, COLS, dt_cell, &dt_mat);
      MPI_Type_commit(&dt_mat);
/*
 * Perform a send and a receive that won't be satisfied.
 */
      MPI_Comm_size(newcomm, &size);
      MPI_Comm_rank(newcomm, &rank);
      MPI_Send(&mat[0][0], 1, dt_mat, (rank + 1) % size,
                  0, newcomm);
      MPI_Recv(&mat[0][0], 1, dt_mat, (rank + 1) % size,
                  MPI_ANY_TAG, newcomm, &status);
      MPI_Finalize();
      return(0);
}
```

Let the program be run with a sufficient number of processes. Then examine the state of the application processes with mpitask.

```
% mpirun -v -c 10 demo
```

In its default display mode, mpitask prints information under the following headings.

`TASK (G/L)` an identification of the process - An MPI process is normally identified by its rank in MPI_COMM_WORLD, also referred to as the “global” (G) rank. If the process is blocked on a communicator, a ‘/’ followed by its rank

```
% mpitask
TASK (G/L)      FUNCTION     PEER|ROOT    TAG    COMM   COUNT DATATYPE
0/0 demo        Recv         3/1          ANY    <2>    1     <30>
2/0 demo        Recv         5/1          ANY    <2>    1     <30>
4/1 demo        Recv         7/2          ANY    <2>    1     <30>
6/2 demo        Recv         9/3          ANY    <2>    1     <30>
8/2 demo        Recv         2/0          ANY    <2>    1     <30>
1/0 demo        Recv         4/1          ANY    <2>    1     <30>
3/1 demo        Recv         6/2          ANY    <2>    1     <30>
5/1 demo        Recv         8/2          ANY    <2>    1     <30>
7/2 demo        Recv         1/0          ANY    <2>    1     <30>
9/3 demo        Recv         0/0          ANY    <2>    1     <30>
```

within that communicator is appended. This is also referred to as the “local” (L) rank. The name of the program is also printed.

`FUNCTION`	the MPI routine currently being executed
`PEER|ROOT`	the source or destination process of a communication operation, if one is specified under FUNCTION, or the root process of certain collective operations
`TAG`	the message tag of a point-to-point communication
`COMM`	the communicator ID being used - See *Debugging and Tracing* for how to cross-reference this number with the program’s data.
`COUNT`	the number of elements being transferred
`DATATYPE`	the datatype ID of each element being transferred

Depending on the MPI routine, some fields may not be applicable and will be left blank. If a process is not currently executing an MPI routine, one of the following execution states may be reported:

`<running>`	free to run on the underlying OS
`<paused>`	blocked on lam_kpause()
`<stopped>`	stopped by the LAM signal, LAM_SIGARREST
`<blocked>`	blocked in a LAM routine - In general this should be a transitory state.

GPS Identification

With spawned processes and even multiple MPI applications running concurrently under the same LAM session, MPI_COMM_WORLD rank is not always an unambiguous identification of an MPI process. LAM has an alternative to the global rank, called the GPS (for Global Positioning System).

-gps Identify MPI processes with the GPS instead of the global rank, the rank within MPI_COMM_WORLD.

The GPS is comprised of the nodeid on which the process is running, and the LAM process index within that node.[1]

```
% mpitask -gps n0 i8
TASK (GPS/L)   FUNCTION     PEER|ROOT    TAG    COMM   COUNT DATATYPE
n0,i8/0 demo   Recv         n1,i9/1      ANY    <2>    1     <30>
```

The MPI communicator and datatype are two opaque objects that are shown as unfamiliar identifiers in the format, <#>. Extended library functions can report the same values from within the running application. See *Debugging and Tracing*. Information from within communicators and datatypes can be reported by mpitask.

Communicator Monitoring

-c Instead of the default report, print communicator information on all selected processes.

The communicator report contains an identification of the process, as under the TASK heading in the default report. It also contains the size of the communicator and the global ranks (or GPS, with the -gps option) of all processes in the communicator's process group. If it is an inter-communicator, members of both groups are reported.

```
% mpitask -c n0 i8
TASK (G/L):     0/0 demo
INTRACOMM:      <2>
SIZE:           4
GROUP:          0 3 6 9
```

Datatype Monitoring

-d Instead of the default report, print datatype information on all selected processes.

The datatype report contains an identification of the process, as under the TASK heading in the default report. It also contains a rendering of the datatype's type map, which is not easy to depict with only ASCII characters. The format is hierarchical, with indentation representing a level of datatype derivation. Basic datatypes are written as they are coded. For derived

1. Application process indices do not start at 0 or 1 because LAM system processes occupy the first several positions.

datatypes, the constructor type is shown along with information on displacements, blocklengths and block counts. Compare the sample code with the output of mpitask -d.

```
% mpitask -d n0 i8
TASK (G/L):     0/0 demo
DATATYPE:       <30>
      MPI_VECTOR (10 x 2, 20)
            MPI_STRUCT (3)
                  (1, 0) MPI_INT
                  (3, 8) MPI_DOUBLE
                  (1, 32) MPI_UB
```

The number of MPI processes reported by mpitask can be constrained by specifying nodeids and/or process indices on the mpitask command line. Choosing the right nodeids and process indices is obviously facilitated by the GPS reporting. Selecting a single process is particularly useful for communicator and datatype reporting, when many or all of the processes might have the same communicator or datatype to report.

doom Doom is the command level interface to signal delivery. Node(s) must be specified on the command line. If no processes are specified, all application processes on the selected nodes are signalled. With no other options, doom sends a LAM_SIGUDIE signal. Unfortunately, the user cannot specify signal mnemonics and must give the actual signal number instead. These are listed below.

`-1`	(LAM_SIGTRACE) unload trace data
`-4`	(LAM_SIGUDIE) terminate
`-5`	(LAM_SIGARREST) suspend execution
`-6`	(LAM_SIGRELEASE) resume execution
`-7`	(LAM_SIGA) reserved for user
`-8`	(LAM_SIGB) reserved for user
`-9`	(LAM_SIGFUSE) node about to die
`-10`	(LAM_SIGSHRINK) another node has died

For example, to suspend process index 8 on node 1, use the following form:

```
% doom n1 i8 -5
```

Resume the execution of the same process:

```
% doom n1 i8 -6
```

`mpimsg`	Monitor message buffers.
`bfctl`	Control message buffers.

Message Monitoring and Control

mpimsg

A receiving process is usually debugged with the *mpitask* command, but a sending process transfers a message and returns to a ready state quickly due to the presence of buffers. The *mpimsg* command is provided to examine buffered messages. With no arguments, all MPI messages on all nodes are reported. The report can be constrained by specifying nodes and processes.

See *Process Monitoring and Control* for an example program that can send messages that will not be received. These messages can be examined with mpimsg.

```
% mpimsg
SRC (G/L)   DEST (G/L)  TAG   COMM  COUNT    DATATYPE    MSG
9/3         0/0         0     <2>   1        <30>        n0,#0
8/2         2/0         0     <2>   1        <30>        n0,#1
1/0         4/1         0     <2>   1        <30>        n0,#2
3/1         6/2         0     <2>   1        <30>        n0,#3
5/1         8/2         0     <2>   1        <30>        n0,#4
7/2         1/0         0     <2>   1        <30>        n1,#0
0/0         3/1         0     <2>   1        <30>        n1,#1
2/0         5/1         0     <2>   1        <30>        n1,#2
4/1         7/2         0     <2>   1        <30>        n1,#3
6/2         9/3         0     <2>   1        <30>        n1,#4
```

In its default display mode, mpimsg prints information under the following headings.

`SRC (G/L)`	an identification of the sending process followed by a '/' and the process's communicator rank (the "local" rank)
`DEST (G/L)`	an identification of the receiving process followed by a '/' and the process's communicator rank
`TAG`	the message tag
`COMM`	the communicator ID
`COUNT`	the number of elements in the message
`DATATYPE`	the datatype ID of each element
`MSG`	the message ID to use in a contents query

The same communicator and datatype information that is obtainable from processes with mpitask is also obtainable from messages. The difference is that more precision is needed to specify a message, because one process can generate several messages. Instead of process indices, mpimsg requires a message number as a parameter to -c (communicator) or -d (datatype). In fact the information needed by mpimsg is that exactly printed under the MSG heading in the default report: nodeid and message number.

Message Contents

`-m <#>`	Display the contents of the specified message number on the specified node.

An additional capability unique to message reporting is the display of message contents. The datatype's type map is used to format the data. Offsets at the beginning of each line are from the beginning of the unpacked message. Contiguous blocks of one basic datatype are printed contiguously, with newlines forced between blocks.

```
% mpimsg -gps n0 -m 4
MESSAGE:    n0,i12/2 #4
00000000:   0
00000008:   0     0     0
00000020:   0
00000028:   0     1     0
00000280:   1
00000288:   1     0     0
000002a0:   1
000002a8:   1     1     1
00000500:   2
...
```

bfctl The LAM daemon does not continue to allocate buffer space up until the operating system is out of memory. There is a limit after which no additional messages will be accepted until some are consumed. Processes will block in send operations if the required buffer space is not available. When using the default GER protocol (See *Executing MPI Programs*), *mpirun* will take care of adjusting the buffer limit according to the guaranteed envelope resources. If this protocol is disabled, the user may need to tune the buffer limit manually. The user can control the maximum size a LAM daemon's buffer pool with the *bfctl* command.

`-s`	Adjust the upper limit on buffered messages for the selected nodes.

```
% bfctl N -s 0x100000
```

`lamtrace`	Collect trace data and store in a file.

Collecting Trace Data

After a traced application has completed execution, trace data recording communication activity is stored within the LAM daemon across all nodes on which the application ran. There is a limit on how much trace data one LAM daemon will hold. When that limit is reached, the oldest traces are discarded in favour of the newest traces. See *Debugging and Tracing* for runtime routines that can limit the volume of trace data.

lamtrace

The *lamtrace* command gathers trace data and stores it into a file, which by convention has the suffix .lamtr.

```
% lamtrace -v -mpi
```

`-mpi`	Search for an MPI world trace created by the specified processes.

For the most part, lamtrace and the LAM daemon are ignorant of specific trace formats. In order to extract MPI trace data for a particular world communicator group in the presence of several such groups (due to spawned processes or multiple applications), lamtrace understands the format of an administrative trace record produced by LAM's MPI library. In simple situations with one application and no spawned processes, no node or process focus is required. lamtrace searches all nodes and eventually locates the solitary MPI world trace, which is produced by process rank 0 in MPI_COMM_WORLD. However, if trace data from multiple worlds are present, node and possibly process specification must be given on the command line to get the data for the desired world. The right nodeid and process index can be learned from *mpitask* or inferred from the application schema. For example:

```
% lamtrace -v -mpi n0 i8
```

It is entirely possible to unload trace data before the application has completed, with the obvious caveat that incomplete communication at the moment of the unload will be reflected in the trace data.

Trace data remains in the LAM daemon and awaits an unload after an application terminates. If not unloaded, it should be removed before running the next application. This is one of the actions taken by *lamclean*.

`lamgrow`	Add a node to the current LAM session.
`lamshrink`	Remove a node.

Adding and Deleting LAM Nodes

LAM can be operated in an environment where resource availability is dynamic, perhaps under the control of an external resource manager. LAM is started and an initial set of nodes are established with *lamboot*. If in the future a resource manager (software or human) decides to modify the current set of nodes belonging to a LAM session, the changes are made with two commands, *lamgrow* and *lamshrink*. Both commands must be executed from an existing LAM node.

lamgrow

A new machine is labelled with a nodeid and added to the LAM session with lamgrow. Usage is more restrictive than typical LAM commands.

- The nodeid must not duplicate an existing node.
- Only one node can be added per invocation of lamgrow.
- The machine name must be supplied. LAM will not choose one.
- Only one copy of lamgrow must be running throughout the LAM multicomputer.

```
% lamgrow -v n8 buckeye.osc.edu
```

If a nodeid is not specified, the next highest LAM nodeid is used. With the power to specify a nodeid, lamgrow can remove the initial property guaranteed by lamboot - that nodeids are consecutive starting from zero.

`-x`	Enable fault tolerant detection and recovery. The decision to use this option generally follows the lamboot invocation.
`-c <bhost>`	Update a boot schema by appending the new machine name to the host list. This is a simple convenience feature that updates a boot schema for use by *wipe*.

lamshrink

A single node is removed per invocation of lamshrink. The nodeid and the machine name must be supplied.

% lamshrink -v n8 buckeye.osc.edu

`-w <#secs>`	Signal all application processes on the doomed node (LAM_SIGFUSE) and pause before continuing. See *Signal Handling*.

`fstate`	Get remote filesystem status.
`fctl`	Control remote filesystem.

File Monitoring and Control

There are commands to monitor and control remote file access (See *Remote File Access*). *fstate* prints one line of status information for each open file descriptor.

fstate

FD/COUNT	global file descriptor handle (not the client handle) and reference count
FLAGS	open flags and status flags (see below)
FLOW	total amount of I/O in bytes since opening
CLIENT	nodeid and process ID of last client process
NAME	filename

The open/status flags are single character mnemonics.

R	open for read
W	open for write
L	locked active
A	active, currently open in the underlying filesystem
I	inactive, currently closed in the underlying filesystem

```
% fstate N
NODE        FD/COUNT FLAGS    FLOW     CLIENT      NAME
n0 (o)      0/0      R|L      0        none        /dev/null
n0 (o)      1/0      W|L      0        n0/p25825   /dev/ttya
n0 (o)      2/0      R|W|L    0        none        /dev/ttya
```

fctl

The *fctl* command has two features. The -s option cleans up and closes a specific file descriptor while the -S option does the same thing for all file descriptors. With no options, *fctl* prints the current working directory of the remote filesystem. The working directory is changed by giving a new pathname to *fctl*. In the current release, working directories are kept on a per node basis, not a per process basis.

```
% fctl -s 4
```

`bhost.my3suns` example host file

Writing a LAM Boot Schema

The topology of a multicomputer is established in the boot schema. The boot schema specifies the identifiers and types of nodes, and the physical machines to be used. It may also contain the user account name on a machine in case it is different from the local username. The boot schema is used by *lamboot* when starting the LAM session and by *wipe* when terminating the LAM session. See *Starting LAM*.

A variety of boot schemata describing different multicomputers may already be available for a given installation. These files are generally found in the directory $LAMHOME/boot. LAM users may need to write their own boot schema since the network often affords many choices. This section describes how to write a boot schema for LAM using the host file syntax. The example multicomputer has three nodes, one of which has a different user account name.

Host File Syntax

The host file syntax is an extremely simple way of representing the information required in a LAM boot schema. The machines are listed one on each line with an optional user account name (username) following it. The username is required in case the account name on that machine is different from the one on the local machine where lamboot will be invoked. If the username is not given, the local one will be used. The nodeids are determined by the order in which the machines appear in the file, starting with node 0 and proceeding with consecutive node numbers. A line segment following a # character denotes a comment and is thus skipped.

In the three node example, it is assumed that the machines are named "ohio", "osc" and "faraway.far.edu" and numbered 0, 1, and 2 respectively. It is also assumed that the user is logged on to node 0, and has the same username on node 1, but a different one (guest) on node 2. Since node 1 has the same username as the local node, there is no need to specify it. The example boot schema using the host file syntax is shown below.

```
# a 3 node example
ohio
osc
faraway.far.edu guest
```

`hboot`	Start LAM on one node.

Low Level LAM Start-up

The *lamboot* command runs a lower level program that starts LAM on a specific node. Normally, the user will only need to use lamboot. In some special circumstances, when variations in the normal start-up procedure not controllable with lamboot options are desired, the user may wish to manually start the system. By running the low level *hboot* tool, the user can select options that tailor the start-up to his/her needs and/or bypass some of the complexities of lamboot.

Process Schema

The hboot tool reads a per-node configuration file called a process schema. The process schema contains a list of programs and runtime arguments that will constitute LAM on a node. The default process schema filename for hboot is *conf.otb*. Lamboot invokes hboot using the *conf.lam* process schema. Just as the user can create custom boot schemata, he/she can create custom process schemata. They make it easy to reconfigure LAM at the process level. For a complete description of the process schema grammar, see the *procschema* manual page.

To manually start a LAM session, first consult the boot schema. This file specifies the node identifiers as well as a binding between node identifiers and actual machines. The example boot schema shown below is written with the host file syntax and describes a 3 node multicomputer.

```
# a 3 node example
ohio
osc
faraway.far.edu guest
```

hboot

Each node will be started using the hboot tool, giving each node information about the other nodes in the multicomputer in order to form the fully connected LAM topology. Assuming the user is logged on to machine "ohio", first start LAM locally.

```
{ohio}% hboot -vc conf.lam -I "-n0 -o0
           osc 1 faraway.far.edu 2"
```

Then login to machine "osc" and start LAM on it.

```
{osc}% hboot -vc conf.lam -I "-n1 -o0
           ohio 0 faraway.far.edu 2"
```

Then login to machine “faraway.far.edu” on the account “guest” and start LAM on it.

```
{faraway}% hboot -vc conf.lam -I "-n2 -o0
           ohio.here.edu 0 osc.here.edu 1"
```

Notice that in this last case the full machine names of “ohio” and “osc” are provided since they are in a different domain than “faraway”. The -I option’s parameter becomes the value of the $inet_topo variable in the process schema. This variable is used by LAM to ascertain network information.

-o	the nodeid of the origin node - The origin node is assumed to be the position from where the user would have invoked lamboot. Many LAM features use the origin node as a default nodeid.
-n	the local nodeid

Other than establishing local and remote nodeids, the network information contains machine name / link number pairs for all other nodes. The link number is equivalent to the LAM nodeid.

The same procedure may be done using the rsh UNIX tool instead of logging in to each machine. In this case, use the -s option of hboot in order to allow rsh to return when hboot is done.

Appendix A: Fortran Bindings

This appendix contains Fortran bindings for the library routines described in this document. All bindings are subroutines unless otherwise noted.

from *Initialization*:

```
MPI_INIT (ierror)
     integer ierror
MPI_FINALIZE (ierror)
MPI_ABORT (comm, errcode, ierror)
     integer comm, errcode
MPI_COMM_SIZE (comm, size, ierror)
     integer comm, size
MPI_COMM_RANK (comm, rank, ierror)
     integer comm, rank
```

from *Blocking Point-to-Point*:

```
MPI_SEND (buf, count, dtype, dest, tag, comm,
          ierror)
     <type> buf(*)
     integer count, dtype, dest, tag, comm
MPI_RECV (buf, count, dtype, source, tag, comm,
          status, ierror)
     <type> buf(*)
     integer count, dtype, source, tag, comm
     integer status(MPI_STATUS_SIZE)
MPI_GET_COUNT (status, dtype, count, ierror)
     integer status(MPI_STATUS_SIZE), dtype, count
MPI_PROBE (source, tag, comm, status, ierror)
     integer source, tag, comm
     integer status(MPI_STATUS_SIZE)
```

from *Nonblocking Point-to-Point*:

```
MPI_ISEND (buf, count, dtype, dest, tag, comm,
          request, ierror)
     <type> buf(*)
     integer count, dtype, dest, tag
     integer comm, request
```

```
MPI_IRECV (buf, count, dtype, source, tag, comm,
          request, ierror)
     <type> buf(*)
     integer count, dtype, source, tag
     integer comm, request
MPI_TEST (request, flag, status, ierror)
     logical flag
     integer request, status(MPI_STATUS_SIZE)
MPI_WAIT (request, status, ierror)
     integer request, status(MPI_STATUS_SIZE)
MPI_IPROBE (source, tag, comm, flag, status,
          ierror)
     logical flag
     integer source, tag, comm
     integer status(MPI_STATUS_SIZE)
```

from *Message Datatypes*:

```
MPI_TYPE_VECTOR (count, blocklength, stride,
          oldtype, newtype, ierror)
     integer count, blocklength, stride
     integer oldtype, newtype
MPI_TYPE_STRUCT (count, blocklengths,
          displacements, dtypes, newtype, ierror)
     integer count, blocklengths(*)
     integer displacements(*), dtypes(*), newtype
MPI_ADDRESS (location, address, ierror)
     <type> location(*)
     integer address
MPI_TYPE_COMMIT (dtype, ierror)
     integer dtype
MPI_PACK_SIZE (incount, dtype, comm size, ierror)
     integer incount, dtype, comm, size
MPI_PACK (inbuf, incount, dtype, outbuf, outsize,
          position, comm, ierror)
     <type> inbuf(*), outbuf(*)
     integer incount, dtype, outsize
     integer position, comm
```

```
MPI_UNPACK (inbuf, insize, position, outbuf,
          outcount, dtype, comm, ierror)
     <type> inbuf(*), outbuf(*)
     integer insize, position, outcount
     integer dtype, comm
```

from *Collective Message-Passing*:

```
MPI_BCAST (buf, count, dtype, root, comm, ierror)
     <type> buf(*)
     integer count, dtype, root, comm
MPI_SCATTER (sendbuf, sendcount, sendtype,
          recvbuf, recvcount, recvtype, root,
          comm, ierror)
     <type> sendbuf(*), recvbuf(*)
     integer sendcount, sendtype, recvcount
     integer recvtype, root, comm
MPI_GATHER (sendbuf, sendcount, sendtype,
          recvbuf, recvcount, recvtype, root
          comm, ierror)
     integer sendcount, sendtype, recvcount
     integer recvtype, root, comm
MPI_REDUCE (sendbuf, recvbuf, count, dtype, op,
          root, comm, ierror)
     <type> sendbuf(*), recvbuf(*)
     integer count, dtype, op, root, comm
```

from *Creating Communicators*:

```
MPI_COMM_DUP (comm, newcomm, ierror)
     integer comm, newcomm
MPI_COMM_SPLIT (comm, color, key, newcomm, ierror)
     integer comm, color, key, newcomm
MPI_COMM_FREE (comm, ierror)
     integer comm
MPI_COMM_REMOTE_SIZE (comm, size, ierror)
     integer comm, size
MPI_INTERCOMM_MERGE (intercomm, high, intracomm,
          ierror)
```

```
     integer intercomm, intracomm
     logical high
```

from *Process Topologies*:

```
MPI_CART_CREATE (oldcomm, ndims, dims, periods,
          reorder, newcomm, ierror)
     integer oldcomm, ndims, dims(*), newcomm
     logical periods(*), reorder
MPI_CART_RANK (comm, coords, rank, ierror)
     integer comm, coords(*), rank
MPI_CART_COORDS (comm, rank, maxdims, coords,
          ierror)
     integer comm, rank, maxdims, coords(*)
MPI_CART_SHIFT (comm, direction, distance,
          rank_source, rank_dest, ierror)
     integer comm, direction, distance
     integer rank_source, rank_dest
```

from *Dynamic Processes*:

```
MPI_SPAWN (program, argv, maxprocs, info, root,
          comm, intercomm, ierrors, ierror)
     character*(*) program, argv(*)
     integer info, maxprocs, root, comm
     integer intercomm, ierrors(*)
```

from *Miscellaneous MPI Features*:

```
MPI_ERRHANDLER_CREATE (errfunc, handler, ierror)
     external errfunc
     integer handler
MPI_ERRHANDLER_SET (comm, handler, ierror)
     integer comm, handler
MPI_ERROR_STRING (code, errstring, resultlen,
          ierror)
     integer code, resultlen
     character*(*) errstring
MPI_ERROR_CLASS (code, class, ierror)
     integer code, class
```

```
MPI_ATTR_GET (comm, keyval, attrval, flag, ierror)
     integer comm, keyval, attrval
     logical flag
```

```
double precision MPI_WTIME()
```

from *Remote File Access*:

```
lamf_rfopen (lamfd, file, flags, modes, ierror)
     integer lamfd, flags, modes
     character*(*) file
lamf_rfclose (lamfd, ierror)
     integer lamfd
lamf_rfread (lamfd, buf, length, nread, ierror)
     integer lamfd, length, nread
     <type> buf(*)
lamf_rfwrite (lamfd, buf, length, nwritten,
          ierror)
     integer lamfd, length, nwritten
     <type> buf(*)
```

from *Collective I/O*:

```
CBX_OPEN (file, flags, mode, owner, comm, cbxfd,
          ierror)
     character*(*) file
     integer flags, mode, owner, comm, cbxfd
CBX_CLOSE (cbxfd, ierror)
     integer cbxfd
CBX_READ (cbxfd, buf, count, dtype, nread, ierror)
     integer cbxfd, count, dtype, nread
     <type> buf(*)
CBX_WRITE (cbxfd, buf, count, dtype, nwritten,
          ierror)
     integer cbxfd, count, dtype, nwritten
     <type> buf(*)
CBX_LSEEK (cbxfd, offset, whence, ierror)
     integer cbxfd, offset, whence
CBX_MULTI (cbxfd, ierror)
     integer cbxfd
```

```
CBX_SINGL (cbxfd, ierror)
     integer cbxfd
CBX_IS_MULTI (cbxfd, result, ierror)
     integer cbxfd
     logical result
CBX_IS_SINGL (cbxfd, result, ierror)
     integer cbxfd
     logical result
CBX_ORDER (cbxfd, newrank, ierror)
     integer cbxfd, newrank
```

from *Signal Handling*:

```
MPIL_SIGNAL (comm, rank, signo, ierror)
     integer comm, rank, signo
```

from *Debugging and Tracing*:

```
MPIL_COMM_ID (comm, id, ierror)
     integer comm, id
MPIL_COMM_GPS (comm, rank, nodeid, pid, ierror)
     integer comm, rank, nodeid, pid
MPIL_TYPE_ID (dtype, id, ierror)
     integer dtype, id
MPIL_TRACE_ON (ierror)
MPIL_TRACE_OFF (ierror)
```

Appendix B: Fortran Example Program

The trivial example program from *Programming Tutorial* is shown here in Fortran.

```
c
c Transmit a message in a two process system.
c
      program trivial
#include <mpif.h>
      integer*4         BUFSIZE
      parameter         (BUFSIZE = 64)
      integer*4         buffer(BUFSIZE)
      integer           rank, size
      integer           status(MPI_STATUS_SIZE)
c
c Initialize MPI.
c
      call MPI_INIT(ierror)
c
c Error check the number of processes.
c Determine my rank in the world group.
c The sender will be rank 0 and the receiver, rank 1.
c
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
      if (size .ne. 2) then
            call MPI_FINALIZE(ierror)
            stop
      endif
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
c
c As rank 0, send a message to rank 1.
c
      if (rank .eq. 0) then
            call MPI_SEND(buffer(1), BUFSIZE, MPI_INTEGER,
     +                  1, 11, MPI_COMM_WORLD, ierror)
c
c As rank 1, receive a message from rank 0.
c
      else
            call MPI_RECV(buffer(1), BUFSIZE, MPI_INTEGER,
     +                  0, 11, MPI_COMM_WORLD, status,
     +                  ierror)
      endif
      call MPI_FINALIZE(ierror)
      stop
      end
```

Contact Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
lam@tbag.osc.edu

More Information http://www.osc.edu/lam.html
ftp://ftp.osc.edu/pub/lam

Acknowledgment LAM documentation is supported in part by the National Science Foundation under grant CCR-9510016.

MPI Primer / Developing with LAM