MPI Collective Operations
System Services Interface (SSI) Modules
for LAM/MPI
API Version 1.0.0 / SSI Version 1.0.0

Jeffrey M. Squyres
Brian Barrett
Andrew Lumsdaine
http://www.lam-mpi.org/

Open Systems Laboratory
Pervasive Technologies Labs
Indiana University
CS TR577

August 4, 2003

Contents

1 Overview **4**
- 1.1 General Scheme . . . 4
- 1.2 Thread Safety . . . 4

2 Services Provided by the coll SSI **4**
- 2.1 Header Files . . . 4
- 2.2 Module Selection Mechanism . . . 5
  - 2.2.1 Communication . . . 5
- 2.3 Types . . . 5
- 2.4 Global Variables . . . 5
  - 2.4.1 `int lam_ssi_coll_did` . . . 6
  - 2.4.2 `int lam_ssi_coll_verbose` . . . 6
  - 2.4.3 BLK* Constants . . . 6
- 2.5 Functions . . . 6
  - 2.5.1 `lam_mkcoll()` . . . 6
  - 2.5.2 `lam_mkpt()` . . . 7
  - 2.5.3 `lam_err_comm()` . . . 7
  - 2.5.4 Datatype Accessor Functions . . . 7

3 coll SSI Module API **8**
- 3.1 API Notes . . . 9
  - 3.1.1 Layered Point-to-Point Implementations . . . 9
  - 3.1.2 Hierarchical Implementations (Sub-Communicators) . . . 10
  - 3.1.3 Intracommunicators and Intercommunicators . . . 11
  - 3.1.4 Checkpoint / Restart Functionality . . . 11
  - 3.1.5 MPI Exceptions and Return Values . . . 12
- 3.2 Data Member: `lsc_meta_info` . . . 13
- 3.3 Function Call: `lsc_thread_query` . . . 13
- 3.4 Function Call: `lsc_query` . . . 13
- 3.5 Data Member: `lsc_has_checkpoint` . . . 14
- 3.6 Function Call: `lsca_init` . . . 14
- 3.7 Function Call: `lsca_finalize` . . . 14
- 3.8 Function Call: `lsca_checkpoint` . . . 15
- 3.9 Function Call: `lsca_continue` . . . 15
- 3.10 Function Call: `lsca_restart` . . . 16
- 3.11 Function Call: `lsca_allgather` . . . 16
- 3.12 Function Call: `lsca_allgatherv` . . . 16
- 3.13 Function Call: `lsca_allreduce` . . . 17
- 3.14 Function Call: `lsca_alltoall` . . . 17
- 3.15 Function Call: `lsca_alltoallv` . . . 17
- 3.16 Function Call: `lsca_alltoallw` . . . 17
- 3.17 Function Call: `lsca_barrier` . . . 18
- 3.18 Data Member: `int lsca_bcast_optimization` . . . 18
- 3.19 Function Call: `lsca_bcast` . . . 18
- 3.20 Function Call: `lsca_exscan` . . . 18
- 3.21 Function Call: `lsca_gather` . . . 19

3.22 Function Call: lsca_gatherv 19
3.23 Data Member: int lsca_reduce_optimization 19
3.24 Function Call: lsca_reduce 19
3.25 Function Call: lsca_reduce_scatter 20
3.26 Function Call: lsca_scan 20
3.27 Function Call: lsca_scatter 20
3.28 Function Call: lsca_scatterv 20

4 To Be Determined **21**

5 Acknowledgements **21**

References **21**

1 Overview

Before reading this document, readers are strongly encouraged to read the general LAM/MPI System Services Interface (SSI) overview ([3]). This document uses the terminology and structure defined in that document.

The coll SSI type is used to implement the algorithms in the MPI collective operations in LAM/MPI. Its primary responsibility is to examine the run-time environment of each communicator (as it is created) and decide on an optimal set of collective algorithms to use on that communicator.

The coll SSI is designed for two primary audiences:

1. Algorithm authors, so that they can implement new MPI collective algorithms without needing to know much of the internal details and complexity of an MPI implementation.

2. Hardware providers, so that they can exploit hardware support for collective operations.

1.1 General Scheme

Each coll module is opened and initialized during MPI_INIT. Any module that indicates that it does not want to run will be ignored for the duration of the process. All other modules are eligible for selection on a per-communicator basis for the rest of the process.

As each new communicator is constructed (including MPI_COMM_WORLD and MPI_COMM_SELF), the available coll modules will be queried to see if they want to operate on the new communicator. That is, each new communicator triggers the selection mechanism for coll modules. The selected module will have its function pointers cached on the `MPI_Comm` handle. Whenever a collective function is invoked on the communicator, LAM will perform error checking on the arguments and then invoke the corresponding underlying coll module API function.

When the communicator is destroyed, the coll module's finalize function is invoked. This will happen, at the latest, during MPI_FINALIZE.

1.2 Thread Safety

Note that coll modules are able to indicate the range of MPI thread levels that they support in the `thread_query` API call (Section 3.3).

Before deciding which levels to support, note that MPI semantics guarantee that the same communicator cannot be used for two simultaneous collective operations. Hence, multiple threads will never be executing in a given coll module with the same communicator. This means that most coll modules will likely be able to support [MPI_THREAD_SINGLE,MPI_THREAD_THREAD_SERIALIZED] with little or no effort.

Restricting coll module persistent mutable state to the cached values on the communicator may be sufficient to make most coll modules able to support MPI_THREAD_MULTIPLE with little additional effort.

2 Services Provided by the coll SSI

Several services are provided by the coll SSI that are available to all coll modules.

2.1 Header Files

The following header files must be included (in order) in all module source files that want to use any of the common coll SSI services described in this document:

```
#include <lam-ssi.h>
#include <lam-ssi-coll.h>
```

Both of these files are included in the same location: `share/ssi/include`. If using GNU Automake and the `top_lam_srcdir` macro as recommended in [3], the following can be added to the `AM_CPPFLAGS` macro (remember that `LAM_BUILDING` must be defined to be 1):

```
AM_CPPFLAGS = \
              -I$(top_lam_builddir)/share/include \
              -I$(top_lam_srcdir)/share/include \
              -I$(top_lam_srcdir)/share/ssi/include
```

All three flags are necessary to obtain the necessary header files (note that the build directory is explicitly included in order to support VPATH builds properly, even though it will be redundant in non-VPATH builds).

2.2 Module Selection Mechanism

The coll SSI has potentially many different scopes over the life of an MPI process. Each MPI communicator has its own coll scope. Hence, it is possible for a different coll module to be selected for each MPI communicator. The coll selection process is divided into multiple parts:

1. The module's open function (if it exists). This is invoked exactly once when an MPI process is initialized. If this function exists and returns 0, the module will be ignored for the rest of the process.
2. The module's `lsc_thread_query` API function (see Section 3.3). This function is also invoked exactly once when the application is initialized and is used to query the module to find out what levels of thread support it provides (this occurs during MPI_INIT). If this function returns a nonzero value, the module will be ignored for the rest of the process.
3. The module's `lsc_query` API function (see Section 3.4). This function is invoked once for every new scope (i.e., every new MPI communicator, including MPI_COMM_WORLD and MPI_COMM_SELF). Its return value only determines whether the module will be considered for selection in that communicator.

2.2.1 Communication

coll modules are initialized near the end of the construction of an MPI communicator. As such, point to point communication (e.g., MPI_SEND and MPI_RECV) *on that communicator* is possible. All point-to-point communication conducted on the communicator must be fully completed (e.g., no unmatched non-blocking communications) when the negotiation is complete.

Limited use of communicator constructors and collectives are permitted during the selection process for a new communicator. See Section 3.1.2 for details.

2.3 Types

Some types are used in different API calls and throughout the coll SSI.

2.4 Global Variables

Several global variables are available to all coll modules. These variables are `extern`'ed in the coll header file.

2.4.1 `int lam_ssi_coll_did`

This `int` is set by the coll SSI initialization function, and will therefore be usable by every coll API function (including `open`). It is the debug stream ID specific to the coll SSI modules (see [1] for a description of LAM/MPI debug streams). If coll modules do not create their own debug streams, they should use `lam_ssi_coll_did`.

Debug streams should be used in conjunction with `lam_ssi_coll_verbose`. Note, however, than debug streams should be used with care (and/or "compiled out" with preprocessor directives when not in use) because even if their output is not displayed, they still invoke a function call and may generate overhead at run-time.

2.4.2 `int lam_ssi_coll_verbose`

The `lam_ssi_coll_verbose` variable will be set by the coll SSI initialization function, and will therefore be usable by every coll API function.

`lam_ssi_coll_verbose` is used to indicate how many "status" messages should be displayed. This does not pertain to error messages that the module may need to print – only messages that are superfluous "what's going on" kind of messages.

The value of this variable should be interpreted as following:

- Less than zero: do not output any status messages.
- Zero: print minimal amounts of status messages. This typically corresponds to the "`-v`" flag on various LAM commands.
- Greater than zero: print status messages – potentially more than if the value were zero; exact meaning is left up to the module. A value of 1000 typically corresponds to the "`-d`" flag on various LAM commands.

2.4.3 BLK* Constants

LAM's BLK* constants are used to identify which function is being used. These constants are generally in the form of "`BLKMPI<FUNCTION>`, and are located in the header file `<blktype.h>` The constants can be used as tags for MPI point-to-point functions for layered coll module implementations, or used to generate MPI exceptions marked as coming from a specific collective operation.

2.5 Functions

Several common functions are provided to all coll SSI modules.

2.5.1 `lam_mkcoll()`

void lam_mkcoll(MPI_comm comm)

This function is used switch a communicator from point-to-point context to collective context. It should be invoked at the beginning of all layered collective implementations before any point-to-point communication is invoked. See Section 3.1.1 for more information on layered point-to-point implementations.

2.5.2 `lam_mkpt()`

```
void lam_mkpt(MPI_comm comm)
```

This function is used switch a communicator from collective context to point-to-point context. It should be invoked at the end of all layered collective implementations that previously invoked `lam_mkcoll()`. See Section 3.1.1 for more information on layered point-to-point implementations.

2.5.3 `lam_err_comm()`

```
int lam_err_comm(MPI_comm, int errclass, int error, char *errmsg)
```

This function is used to invoke MPI exceptions on a communicator. The available MPI error classes are listed in `<mpi.h>`; the integer error codes are defined by the coll module. `errmsg` is a string error message that may be printed if LAM's default exception handlers are invoked. For example:

```
int lam_ssi_coll_MODULE_barrier(MPI_Comm comm) {
  /* ... perform the barrier ... */
  if (an_error_occurred)
    return lam_err_comm(comm, MPI_ERR_COMM, MPI_ERR_ARG, "Error in comm");
  return MPI_SUCCESS;
}
```

Always return the value from `lam_err_comm()`. If the MPI exception does not abort, its error code will be returned through the function's return value.

2.5.4 Datatype Accessor Functions

Although coll modules generally need not access individual data members in user-provided buffers, coll modules may need to allocate temporary buffers and copy user-provided buffers while performing collective algorithms. LAM provides functions for allocating and copying buffers that are sized and mapped by MPI datatypes.

Note that reduction operations seem like a notable exception to this policy; a reduction must potentially combine multiple buffers into a final output buffer, and would therefore seem to require access to the individual data members in the buffer. However, a separate function will be utilized for this purpose. Built-in MPI operations such as MPI_SUM have their own function which will iterate over data members to perform the summation. User-provided operations will have user-specified functions to perform the combination operation. Specifically, the reduction function will be cached on the MPI_Op passed to the reduction function. For example:

```
int lam_ssi_coll_MODULE_reduce(..., MPI_Op op, ...) {
  /* ... setup ... */
  /* Invoke the reduction function, checking to see if the MPI
      datatype can be passed directly, or whether we need to pass the
      integer handle, as required by Fortran */

  if (op->op_flags & LAM_LANGF77)
    (op->op_func)(input_buffer, output_buffer, &count, &datatype->dt_f77handle);
  else
```

```
    (op->op_func)(input_buffer, output_buffer, &count, &datatype);

  /* ... cleanup and final handling ... */
}
```

As such, even with reductions, the coll module itself does not need to access the individual data members – it only needs to provide the infrastructure to copy and move buffers between processes. The following functions can be used to

- Allocate a buffer large enough to accomodate a specific MPI datatype.

  ```
  int lam_dtbuffer(MPI_Datatype dtype, int count, char **buffer, char **origin);
  ```

  `dtype` and `count` specify how large to make the buffer. Two pointers are returned: `buffer` is a pointer to the beginning of the allocated buffer (i.e., it can later be used as an argument to `free()`. `origin` is a pointer that can be passed as the buffer to MPI functions (e.g., MPI_SEND, MPI_RECV, etc.). `buffer` and `origin` may be different values if the lower bound of the MPI datatype is artificially set lower than its actual value.

- Copy a buffer mapped by two MPI datatypes.

  ```
  int lam_dtsndrcv(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount
    MPI_Datatype rdtype, int tag, MPI_Comm comm);
  ```

  Typically, this function is used to copy from a user buffer to a temporary buffer allocated by `lam_dtbuffer` (or vice versa). Different datatypes may be used, as long as they match. If the datatype is relatively simple, the copy is performed directly. If the datatype is not simple, MPI_SENDRECV is used.

- Pack a buffer into a contiguous buffer, or unpack a buffer from a contiguous buffer. The functions MPI_PACK and MPI_UNPACK can be used directly for this purpose.

3 coll SSI Module API

This is version 1.0.0 of the coll SSI module API.

Each coll SSI module must export a `struct lam_ssi_coll_1_0_0` named `lam_ssi_coll_<name>_module`. This type is defined in Figure 1. This `struct` contains only a small number of function pointers used for initialization and querying the module. Figures 2 and 3 show a `struct` that contains function pointers to the collective API algorithm functions (it is split across two figures because of its length). A pointer to this `struct` is returned by the module's query function (in Figure 1) if the module decides that it should be used for a given communicator.

The majority of the elements in Figures 1, 2, and and 3 are function pointer types; each is discussed in detail below. When describing the function prototypes, the parameters are marked in one of three ways:

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

```
typedef struct lam_ssi_coll_1_0_0 {
  lam_ssi_1_0_0_t lsc_meta_info;

  /* Initialization / querying functions */

  lam_ssi_coll_thread_query_fn_t lsc_thread_query;
  lam_ssi_coll_query_fn_t lsc_query;

  /* Flags */

  int lsc_has_checkpoint;
} lam_ssi_coll_1_0_0_t;
```

Figure 1: The coll type for exporting the basic API function pointers and flags.

3.1 API Notes

The following sections provide details and clarifications on specific aspects of implementing the collective API function calls.

3.1.1 Layered Point-to-Point Implementations

Collective algorithms may be layered on top of the MPI point-to-point functions (e.g., MPI_SEND and MPI_RECV). However, collective implementations must guarantee not to interfere with any pending point-to-point communications on the same communicator. LAM allows for this functionality by having a unique integer context on all communicators. Positive contexts are reserved for point-to-point communications; negative contexts are reserved for collective communications.

As such, layered collective implementations may either use the `lam_mkcoll()` and `lam_mkpt()` functions to change their communicator's context to be collective and then back to point-to-point (respectively), or they may use communicator constructors to make a private communicator with which to communicate. See Sections 2.5.1 and 2.5.2 for information about `lam_mkcoll()` and `lam_mkpt()`, respectively, and Section 3.1.2 for information about using sub-communicators.

It is critical that layered collectives guarantee to not leave any unmatched communications when they complete. Doing so will consume resources and potentially cause deadlock during the communicator destructor (which may be during MPI_FINALIZE).

Finally, layered algorithms should use the PMPI functions when available. This will guarantee to not create any side-effects from potentially user-intercepted MPI functions. However, LAM can be configured without the MPI profiling layer. coll modules must therefore utilize compile-time macros to switch between MPI and PMPI functions. The following is an example of how this can work:

```
/* <lam_config.h> will set LAM_WANT_PROFILE to 1 or 0, depending on
   whether LAM/MPI was built with profiling support or not. */
#include <lam_config.h>
#if LAM_WANT_PROFILE
#define LAM_PROFILELIB 1
#endif
```

```
/* If LAM_PROFILELIB is set to 1, <mpi.h> and <mpisys.h> will #define
   all MPI_Foo functions to be PMPI_Foo. */
#include <mpi.h>
#include <mpisys.h>

int lam_ssi_coll_MODULE_bcast(..., int root, MPI_Comm comm) {
  /* Simplistic linear algorithm */
  if (rank == root) {
    for (i = 0; i < size; ++i)
      if (i != root)
        MPI_Send(..., comm);
  } else
    MPI_Recv(..., comm, MPI_STATUS_IGNORE);

  return MPI_SUCCESS;
}
```

3.1.2 Hierarchical Implementations (Sub-Communicators)

It is permissible for coll module to create sub-communicators during the `lsca_init` API function call (see Section 3.6). Sub communicators may be useful to create "multi-module" collectives such that a top-level coll module is used to partition a communicator into smaller parts, each of which can invoke a dedicated collective algorithm (potentially in a different coll module).

For example, the smp coll module will invoke MPI_COMM_SPLIT on the communicator during its `lsca_init` function (Section 3.6) to create a set of sub-communicators. Each sub-communicator will only contain MPI processes on a single node. In this way, the smp coll module runs a top-level algorithm for each collective, and the sub-communicators perform lower-level algorithms.

However, MPI_COMM_SPLIT invokes collectives to create new communications (namely MPI_ALLREDUCE and MPI_ALLGATHER). This creates a paradox: the coll module needs to invoke MPI_COMM_SPLIT while it is setting up, but MPI_COMM_SPLIT requires a communicator with a fully-functional coll module to complete successfully.

In order to enable this kind of behavior, modules can initially supply one set of function pointers for the MPI collective functions and later provide a second set of function pointers. Specifically, the lam_basic coll module is the "basic" implementation of the MPI collective algorithms. It uses standard linear/logarithmic algorithms which, although they perform well in many environments, are less than optimal in others. However, the lam_basic API functions can always be used without any prior setup. As such, in the smp module example above, smp initially sets up the target communicator with the function pointers from lam_basic before invoking MPI_COMM_SPLIT. Later, after all setup has been completed, smp replaces the lam_basic function pointers with its own.

For example:

1. During the creation of a new communicator, the smp module (which previously returned valid thread levels from its `lsc_thread_query` function) has its `lsc_query` function invoked.
2. It determines that it is eligible to run, and assigns an appropriate priority.
3. It then invokes `lam_ssi_coll_lam_basic_query()` to return a `struct` of pointers to the lam_basic API functions (inserting its own `lsca_init` and `lcsa_finalize` functions before returning

it to the upper level).

4. If the smp module is selected, its `lsca_init` function is invoked.

5. `lsca_init` invokes MPI_COMM_SPLIT to create sub communicators. MPI_COMM_SPLIT will use the lam_basic function pointers that were previously provided by the `lsc_query` function to create the sub communicators.

6. Finally, `lsca_init` returns a new `struct` that contains pointers to all the smp module functions (effectively replacing the lam_basic function pointers).

Note that `lam_ssi_coll_lam_basic_query()` is intended to be used in exactly this fashion (i.e., the lam_basic module was specifically designed such that it is not necessary to invoke its `init` function). Also, in anticipation of various coll modules needing this function, it is prototyped in `<lam-ssi-coll.h>`.

Also note that all sub-communicators created to implement this kind of behavior should set the LAM_HIDDEN flag on the `c_flags` member on the communicator.[1] This will hide the communicator from parallel debuggers that can view message queues. For example:

```
int lam_ssi_coll_MODULE_init(MPI_Comm comm, const lam_ssi_coll_actions_t **new_actions) {
  /* ... setup ... */
  MPI_Comm_split(comm, color, key, &sub_comm);
  sub_comm->c_flags |= C_HIDDEN;
  /* ... rest of function ... */
}
```

3.1.3 Intracommunicators and Intercommunicators

Figures 2 and 3 show two function pointers for each MPI collective function – one for intracommunicators, and one for intercommunicators. The function pointer type is the same for both functions (i.e., the function signature is identical). As such, each function below is only described once; it is implied that there are two pointers (of different names) in the `struct` – one for intracommunicators, and one for intercommunicators. LAM's MPI layer will automatically invoke the appropriate function depending on the underlying type of `MPI_Comm` that was passed to the top-level MPI function.

It is permissible for a module to provide NULL for any of the collective functions. If a program invokes a collective with a NULL implementation, the MPI exception MPI_ERR_OTHER will be invoked with an error code of ENOT_IMPLEMENTED.

3.1.4 Checkpoint / Restart Functionality

LAM/MPI has the ability to involuntarily checkpoint and restart parallel MPI applications. The signal to checkpoint an MPI function may arrive asynchronously. Applications are typically unaware that checkpoints are occurring. LAM/MPI will ensure that either no user threads are in the MPI library, or that they have been safely interrupted. When an application resumes, the application thread(s) is(are) resumed as if nothing happened.

coll modules may or may not support checkpoint/restart functionality. If a module supports checkpoint/restart, it must set the `lsc_has_checkpoint` flag to 1 in its exported basic struct (see Figure 1)

[1] [2] contains details on the members of the `MPI_Comm` data structure.

and supply corresponding function pointers for all actions related to the checkpoint/restart functionality (described below). When an MPI application is initialized, the coll framework performs a logical AND on the `lsc_has_checkpoint` value from all available coll modules. Checkpoint/restart support will only be available if the result of this logical AND is 1.

General scheme. When a checkpoint begins, each coll module must get itself into a state such that it can be later restored with no loss of messages or information. For example, a coll module may drain the network such that no messages are "in-flight" on the network when the checkpoint finally occurs. However, this scheme may vary with each coll module; draining the network may or may not be necessary.

After a checkpoint, one of two actions are possible: continue and restart. "Continue" occurs if the checkpoint does not stop the parallel application; processing continues as if nothing had happened. "Restart" occurs if a previous checkpoint is later resumed. The difference is only a minor optimization: a "continue" time, the MPI layer may not need to re-build network connections, etc. (i.e., if the network connections were not shut down during the checkpoint). But at restore time, all network connections and external resources likely need to be rebuilt / re-claimed.

A coll module can insert arbitrary code at each of these three points (checkpoint, continue, restart); functions are provided for each step and are described in Sections 3.8, 3.9, and 3.10, respectively.

When no special action is needed. Modules that are layered on point-to-point MPI functions probably do not need to do anything (at least in terms of the communication network) for checkpointing, continuing, and restarting since the RPI modules will take care of all networking aspects of checkpoint/restart functionality. Modules that fall into this category should either provide empty functions for this API calls, or use the built-in base coll empty functions that are designed for this purpose:

- `lam_ssi_coll_base_checkpoint`
- `lam_ssi_coll_base_continue`
- `lam_ssi_coll_base_restart`

3.1.5 MPI Exceptions and Return Values

If a coll API function that maps directly to an MPI collective function completes successfully, it must return MPI_SUCCESS. Errors that occur during coll module API functions should invoke the corresponding MPI exception handler.

With layered coll implementations, this is already mostly handled by the underlying MPI function calls. The coll module simply needs to check the return value from the underlying MPI function calls to ensure that they were MPI_SUCCESS before continuing. For example:

```
int lam_ssi_coll_MODULE_barrier(MPI_Comm comm) {
  /* ... setup ... */
  if ((err = MPI_Send(...)) != MPI_SUCCESS)
    return err;
  /* ... rest of function ... */
  return MPI_SUCCESS;
}
```

Non-layered coll implementations will need to generate their own MPI exceptions (ensuring to return the value of the exception so that handlers such as MPI_ERRORS_RETURN can function properly). See Section 2.5.3 for details on the `lam_err_comm()` function.

In the presence of errors, coll module authors are strongly encouraged to clean up state and leave the module in a restart-able state before invoking corresponding MPI exceptions. This will allow exception handlers such as MPI_ERRORS_RETURN to allow MPI programs to continue, even in the presence of non-fatal errors.

3.2 Data Member: `lsc_meta_info`

`lsc_meta_info` is the SSI-mandated element contains meta-information about the module. See [3] for more information about this element.

3.3 Function Call: `lsc_thread_query`

- Type: `lam_ssi_coll_thread_query_fn_t`

  typedef int (*lam_ssi_coll_thread_query_fn_t)(**int** *thread_min, **int** *thread_max);

- Arguments:
  - OUT: `thread_min` is the minimum MPI thread level that this module supports. Only meaningful if the function returns zero.
  - OUT: `thread_max` is the maximum MPI thread level that this module supports. Only meaningful if the function returns zero.
- Return value: Zero if the module wants to be considered for negotiation, non-zero otherwise.
- Description: This function is invoked exactly once during the initial module selection process during MPI initialization. It is invoked before all other module API calls.

  If the module wants to be considered for negotiation, it must fill in `thread_min` and `thread_max` to be the minimum and maximum MPI thread levels that it supports. `thread_min` must be less than or equal to `thread_max`. This functionality is split from the main query API call (Section 3.4) because the thread level of the module is determined only once per process, but the query function will be invoked multiple times.

  If the function returns non-zero, the thread levels will be ignored and the module will be ignored for the duration of the process.

3.4 Function Call: `lsc_query`

- Type: `lam_ssi_coll_query_fn_t`

  typedef const lam_ssi_coll_actions_t *(*lam_ssi_coll_query_fn_t)(MPI_Comm comm, **int** *priority);

- Arguments:
  - IN: `comm` is the communicator that is being setup.

– OUT: `priority` is the output priority that this SSI module thinks that it should be rated. This value is only meaningful if the function returns a non-`NULL` value.

- Return value: Either NULL or a pointer to the `struct` shown in Figures 2 and 3. The contents of the `struct` will be copied – the `struct` itself will not be freed.

- Description: If the module wants to be considered for the negotiation of collectives on this communicator, it should return a pointer to the `struct` shown in Figures 2 and 3 and assign an associated priority to `priority`. Valid values of `priority` are in the range $[0, 100]$, with 0 being the lowest priority, and 100 being the highest.

  If the module does not want to be considered during the negotiation for this communicator, it should return `NULL`, and the value in `priority` is ignored.

3.5 Data Member: `lsc_has_checkpoint`

`lsc_has_checkpoint` is set to 1 if this module supports checkpoint/restart functionality, and 0 otherwise. Its value is checked during MPI_INIT.

3.6 Function Call: `lsca_init`

- Type: `lam_ssi_coll_init_fn_t`

```
typedef int (*lam_ssi_coll_init_fn_t)(MPI_Comm comm, const lam_ssi_coll_actions_t **new_actions);
```

- Arguments:
  – IN: `comm` is the communicator that this module has been selected for.
  – OUT: `new_actions` is a pointer to a new `struct` of function pointers for API calls. If the module wishes to change the values that it returned from `lsc_query`, it may fill `new_actions` with the pointer to a new/updated `struct` that will be used for all future API calls on this communicator. If this argument is set to NULL (or not assigned), the original `struct` returned from `lsc_query` will be used.

- Return value: Zero on success, nonzero otherwise.

- Description: The `lsca_init` function is invoked on the selected module only. Its purpose is to allow the selected module to set itself up for normal operation on the given communicator. The intent is to setup as much as possible during this function in order to facilitate the operation of all the collective calls at run time (i.e., overhead time is better spent during the initialization phase for each communicator rather than the actual collective call).

  Note that this function may make limited use of communicator constructors and collectives. See Section 3.1.2 for details.

3.7 Function Call: `lsca_finalize`

- Type: `lam_ssi_coll_finalize_fn_t`

```
typedef int (*lam_ssi_coll_finalize_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being finalized.
- Return value: Zero on success, nonzero otherwise.
- Description: Finalize the use of this module. It is the last function to be called in the scope of this module's selection. It should release any resources allocated during the life of this scope. This function is only invoked on selected modules.

3.8 Function Call: `lsca_checkpoint`

- Type: `lam_ssi_coll_checkpoint_fn_t`

```
typedef int (*lam_ssi_coll_checkpoint_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being checkpointed.
- Return value: MPI_SUCCESS on success, nonzero otherwise.
- Description: Set this communicator to a state where it can be checkpointed. This typically means draining the network such that no collective messages are "in flight", but the exact definition may vary from module to module.

  This function will be invoked once per checkpoint for every communicator that is active in an MPI application. Hence, it may be invoked multiple times in a module for a single checkpoint, but each time with a different communicator argument.

  coll modules that require no special action at checkpoint time should supply the value `lam_ssi_coll_base_checkpoint`.

3.9 Function Call: `lsca_continue`

- Type: `lam_ssi_coll_continue_fn_t`

```
typedef int (*lam_ssi_coll_continue_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being checkpointed.
- Return value: MPI_SUCCESS on success, nonzero otherwise.
- Description: After the checkpoint has successfully completed, this function will be invoked if the application continues without interruption. It may be used to do whatever the module needs to continue operation after a successful checkpoint.

  This function will be invoked once per checkpoint for every communicator that is active in an MPI application. Hence, it may be invoked multiple times in a module for a single checkpoint, but each time with a different communicator argument.

  coll modules that require no special action at continue time should supply the value `lam_ssi_coll_base_continue`.

3.10 Function Call: `lsca_restart`

- Type: `lam_ssi_coll_restart_fn_t`

```
typedef int (*lam_ssi_coll_restart_t)(MPI_Comm comm);
```

- Arguments:
  - IN: `comm` is the communicator which is being checkpointed.
- Return value: MPI_SUCCESS on success, nonzero otherwise.
- Description: When a parallel application is restored, this function is invoked. It may be used to close now-invalid communication channels (if they were not closed when the checkpoint occurred) and re-open them, or whatever else the coll module needs to resume operation.

  This function will be invoked once per restore for every communicator that was active in an MPI application. Hence, it may be invoked multiple times in a module for a single restore, but each time with a different communicator argument.

  coll modules that require no special action at continue time should supply the value `lam_ssi_coll_base_restart`.

3.11 Function Call: `lsca_allgather`

- Type: `lam_ssi_coll_allgather_fn_t`

```
typedef int (*lam_ssi_coll_allgather_fn_t)(void *sbuf, int scount, MPI_Datatype sdtype,
    void *rbuf, int rcount, MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for MPI_ALLGATHER
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_ALLGATHER collective.

3.12 Function Call: `lsca_allgatherv`

- Type: `lam_ssi_coll_allgatherv_fn_t`

```
typedef int (*lam_ssi_coll_allgatherv_fn_t)(void *sbuf, int scount, MPI_Datatype sdtype, void * rbuf,
    int *rcounts, int *disps, MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for MPI_ALLGATHERV
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_ALLGATHERV collective.

3.13 Function Call: `lsca_allreduce`

- Type: `lam_ssi_coll_allreduce_fn_t`

```
typedef int (*lam_ssi_coll_allreduce_fn_t)(void *sbuf, void *rbuf, int count, MPI_Datatype dtype,
  MPI_Op op, MPI_Comm comm);
```

- Arguments: Same as for MPI_ALLREDUCE
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_ALLREDUCE collective.

3.14 Function Call: `lsca_alltoall`

- Type: `lam_ssi_coll_alltoall_fn_t`

```
typedef int (*lam_ssi_coll_alltoall_fn_t)(void *sbuf, int scount, MPI_Datatype sdtype, void* rbuf,
  int rcount, MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for MPI_ALLTOALL
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_ALLTOALL collective.

3.15 Function Call: `lsca_alltoallv`

- Type: `lam_ssi_coll_alltoallv_fn_t`

```
typedef int (*lam_ssi_coll_alltoallv_fn_t)(void *sbuf, int *scounts, int *sdisps, MPI_Datatype sdtype,
  void *rbuf, int *rcounts, int *rdisps, MPI_Datatype rdtype, MPI_Comm comm);
```

- Arguments: Same as for MPI_ALLTOALLV
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_ALLTOALLV collective.

3.16 Function Call: `lsca_alltoallw`

- Type: `lam_ssi_coll_alltoallw_fn_t`

```
typedef int (*lam_ssi_coll_alltoallw_fn_t)(void *sbuf, int *scounts, int *sdisps,
  MPI_Datatype *sdtypes, void *rbuf, int *rcounts, int *rdisps, MPI_Datatype *rdtypes,
  MPI_Comm comm);
```

- Arguments: Same as for MPI_ALLTOALLW
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_ALLTOALLW collective.

  LAM does not currently implement this function; this pointer is here for forward compatibility.

3.17 Function Call: `lsca_barrier`

- Type: `lam_ssi_coll_barrier_fn_t`

  typedef int (*lam_ssi_coll_barrier_fn_t)(MPI_Comm comm);

- Arguments: Same as for MPI_BARRIER
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_BARRIER collective.

  LAM's MPI_BARRIER will return MPI_SUCCESS immediately (and not call this API function) if there is only one member process in the communicator.

3.18 Data Member: `int lsca_bcast_optimization`

This flag should be 1 if `MPI_BCAST` is allowed to return MPI_SUCCESS immediately without invoking the underlying module broadcast function when there are zero data bytes to broadcast. A value of 0 means that the underlying module broadcast function will be invoked regardless of how many data bytes there are to broadcast.

3.19 Function Call: `lsca_bcast`

- Type: `lam_ssi_coll_bcast_fn_t`

  typedef int (*lam_ssi_coll_bcast_fn_t)(**void** *buff, **int** count, MPI_Datatype datatype, **int** root, MPI_Comm comm);

- Arguments: Same as for MPI_BCAST
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_BCAST collective.

  LAM's MPI_BCAST will return MPI_SUCCESS immediately (and not call this API function) if there is only one member process in the communicator, or if there are zero bytes to broadcast.

3.20 Function Call: `lsca_exscan`

- Type: `lam_ssi_coll_exscan_fn_t`

  typedef int (*lam_ssi_coll_exscan_fn_t)(**void** *sbuf, **void** *rbuf, **int** count, MPI_Datatype dtype, MPI_Op op, MPI_Comm comm);

- Arguments: Same as for MPI_EXSCAN
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_EXSCAN collective.

  LAM does not currently implement this function; this pointer is here for forward compatibility.

3.21 Function Call: `lsca_gather`

- Type: `lam_ssi_coll_gather_fn_t`

  typedef int (*lam_ssi_coll_gather_fn_t)(**void** *sbuf, **int** scount, MPI_Datatype sdtype, **void** *rbuf, **int** rcount, MPI_Datatype rdtype, **int** root, MPI_Comm comm);

- Arguments: Same as for MPI_GATHER
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_GATHER collective.

3.22 Function Call: `lsca_gatherv`

- Type: `lam_ssi_coll_gatherv_fn_t`

  typedef int (*lam_ssi_coll_gatherv_fn_t)(**void** *sbuf, **int** scount, MPI_Datatype sdtype, **void** *rbuf, **int** *rcounts, **int** *disps, MPI_Datatype rdtype, **int** root, MPI_Comm comm);

- Arguments: Same as for MPI_GATHERV
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_GATHERV collective.

3.23 Data Member: `int lsca_reduce_optimization`

This flag should be 1 if MPI_REDUCE is allowed to return MPI_SUCCESS immediately without invoking the underlying module reduce function when there are zero data bytes to reduce. A value of 0 means that the underlying module reduce function will be invoked regardless of how many data bytes there are to reduce.

3.24 Function Call: `lsca_reduce`

- Type: `lam_ssi_coll_reduce_fn_t`

  typedef int (*lam_ssi_coll_reduce_fn_t)(**void** *sbuf, **void*** rbuf, **int** count, MPI_Datatype dtype, MPI_Op op, **int** root, MPI_Comm comm);

- Arguments: Same as for MPI_REDUCE
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_REDUCE collective.

  LAM's MPI_REDUCE will return MPI_SUCCESS immediately (and not call this API function) if there are zero elements to reduce and `lsca_reduce_optimization` is set to 1.

3.25 Function Call: `lsca_reduce_scatter`

- Type: `lam_ssi_coll_reduce_scatter_fn_t`

```
typedef int (*lam_ssi_coll_reduce_scatter_fn_t)(void *sbuf, void *rbuf, int *rcounts,
  MPI_Datatype dtype, MPI_Op op, MPI_Comm comm);
```

- Arguments: Same as for MPI_REDUCE_SCATTER
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_REDUCE_SCATTER collective.

3.26 Function Call: `lsca_scan`

- Type: `lam_ssi_coll_can_fn_t`

```
typedef int (*lam_ssi_coll_scan_fn_t)(void *sbuf, void *rbuf, int count, MPI_Datatype dtype,
  MPI_Op op, MPI_Comm comm);
```

- Arguments: Same as for MPI_SCAN
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_SCAN collective.

3.27 Function Call: `lsca_scatter`

- Type: `lam_ssi_coll_scatter_fn_t`

```
typedef int (*lam_ssi_coll_scatter_fn_t)(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf,
  int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm);
```

- Arguments: Same as for MPI_SCATTER
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_SCATTER collective.

3.28 Function Call: `lsca_scatterv`

- Type: `lam_ssi_coll_scatterv_fn_t`

```
typedef int (*lam_ssi_coll_scatterv_fn_t)(void *sbuf, int *scounts, int *disps, MPI_Datatype sdtype,
  void* rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm);
```

- Arguments: Same as for MPI_SCATTERV
- Return value: MPI_SUCCESS on success, or an appropriate error code otherwise.
- Description: Implement the MPI_SCATTERV collective.

4 To Be Determined

Things that still need to be addressed:

- During MPI_INIT, LAM should coordinate which coll modules are available (to include version numbers) between all MPI processes (will definitely be required when dynamic modules are real). During each communicator constructor, only modules that exist with the same version on all nodes in the communicator should be checked for availability (i.e., check for module heterogeneity).

  This also needs to occur during MPI_COMM_SPAWN*, MPI_ACCEPT, MPI_CONNECT, and MPI_JOIN.

- The current method of using PMPI functions is clunky and inelegant. It will improved in future versions of the coll API.

- It is likely that in future versions of the coll API, NULL values can be passed in for the checkpoint/continue/restart functions instead of no-op functions. The only reason that this is not true now is because `has_checkpoint` was added late in the release cycle for 7.0. Its presence obviates the need for "must have non-NULL function pointers" for sentinel values indicating checkpoint/restart functionality.

- It is possible that in future versions of the coll SSI, it will be easier to have a module that implements less-than-all of the collectives – have a more straightforward approach to allow coll modules to use functions from other modules.

5 Acknowledgements

This work was supported by a grant from the Lily Endowment and by National Science Foundation grant 0116050.

References

[1] Brian Barrett, Jeff Squyres, and Andrew Lumsdaine. *LAM/MPI Design Document*. Open Systems Laboratory, Pervasive Technology Labs, Indiana University, Bloomington, IN. See `http://www.lam-mpi.org/`.

[2] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[3] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department, 2003.

```
typedef struct lam_ssi_coll_actions_1_0_0 {

  /* Per−communicator initialization and finalization functions */

  lam_ssi_coll_init_fn_t lsca_init;
  lam_ssi_coll_finalize_fn_t lsca_finalize;

  /* Checkpoint / restart functions */

  lam_ssi_coll_checkpoint_fn_t lsca_checkpoint;
  lam_ssi_coll_continue_fn_t lsca_continue;
  lam_ssi_coll_restart_fn_t lsca_restart;

  /* Collective function pointers */

  lam_ssi_coll_allgather_fn_t lsca_allgather_intra;
  lam_ssi_coll_allgather_fn_t lsca_allgather_inter;

  lam_ssi_coll_allgatherv_fn_t lsca_allgatherv_intra;
  lam_ssi_coll_allgatherv_fn_t lsca_allgatherv_inter;

  lam_ssi_coll_allreduce_fn_t lsca_allreduce_intra;
  lam_ssi_coll_allreduce_fn_t lsca_allreduce_inter;

  lam_ssi_coll_alltoall_fn_t lsca_alltoall_intra;
  lam_ssi_coll_alltoall_fn_t lsca_alltoall_inter;

  lam_ssi_coll_alltoallv_fn_t lsca_alltoallv_intra;
  lam_ssi_coll_alltoallv_fn_t lsca_alltoallv_inter;

  lam_ssi_coll_alltoallw_fn_t lsca_alltoallw_intra;
  lam_ssi_coll_alltoallw_fn_t lsca_alltoallw_inter;

  /* ...continued in next figure */
```

Figure 2: The coll type for exporting the majority of the collective API function pointers (part 1 of 2).

```
  /* ...continued from previous figure */

  lam_ssi_coll_barrier_fn_t lsca_barrier_intra;
  lam_ssi_coll_barrier_fn_t lsca_barrier_inter;

  int lsca_bcast_optimization;
  lam_ssi_coll_bcast_fn_t lsca_bcast_intra;
  lam_ssi_coll_bcast_fn_t lsca_bcast_inter;

  lam_ssi_coll_exscan_fn_t lsca_exscan_intra;
  lam_ssi_coll_exscan_fn_t lsca_exscan_inter;

  lam_ssi_coll_gather_fn_t lsca_gather_intra;
  lam_ssi_coll_gather_fn_t lsca_gather_inter;

  lam_ssi_coll_gatherv_fn_t lsca_gatherv_intra;
  lam_ssi_coll_gatherv_fn_t lsca_gatherv_inter;

  int lsca_reduce_optimization;
  lam_ssi_coll_reduce_fn_t lsca_reduce_intra;
  lam_ssi_coll_reduce_fn_t lsca_reduce_inter;

  lam_ssi_coll_reduce_scatter_fn_t lsca_reduce_scatter_intra;
  lam_ssi_coll_reduce_scatter_fn_t lsca_reduce_scatter_inter;

  lam_ssi_coll_scan_fn_t lsca_scan_intra;
  lam_ssi_coll_scan_fn_t lsca_scan_inter;

  lam_ssi_coll_scatter_fn_t lsca_scatter_intra;
  lam_ssi_coll_scatter_fn_t lsca_scatter_inter;

  lam_ssi_coll_scatterv_fn_t lsca_scatterv_intra;
  lam_ssi_coll_scatterv_fn_t lsca_scatterv_inter;
} lam_ssi_coll_actions_1_0_0_t;
```

Figure 3: The coll type for exporting the majority of the collective API function pointers (part 2 of 2).