Checkpoint/Restart System Services Interface (SSI) Modules for LAM/MPI API Version 1.0.0 / SSI Version 1.0.0

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

Open Systems Laboratory
Pervasive Technologies Labs
Indiana University
CS TR578

August 4, 2003

Contents

1 Overview **5**
- 1.1 General Scheme . . . 5
- 1.2 Terminology . . . 5
  - 1.2.1 The cr SSI Type . . . 5
  - 1.2.2 Action Points . . . 6
- 1.3 Restrictions and Limitations . . . 6
- 1.4 The crlam SSI Type . . . 7
- 1.5 The crmpi SSI Type . . . 8

2 Services Provided by the cr SSI **10**
- 2.1 Header Files . . . 10
- 2.2 Module Selection Mechanism . . . 10
- 2.3 Types . . . 10
  - 2.3.1 `lam_ssi_crmpi_base_handler_state_t` . . . 10
- 2.4 Global Variables . . . 11
  - 2.4.1 `int lam_ssi_cr_did` . . . 11
  - 2.4.2 `int lam_ssi_cr_verbose` . . . 11
- 2.5 Global Variables for crmpi . . . 11
  - 2.5.1 `lam_ssi_crmpi_base_handler_state` . . . 12
- 2.6 Utility Functions for crlam . . . 12
  - 2.6.1 `lam_ssi_crlam_base_checkpoint()` . . . 12
  - 2.6.2 `lam_ssi_crlam_base_continue()` . . . 12
  - 2.6.3 `lam_ssi_crlam_base_restart()` . . . 12
  - 2.6.4 `lam_ssi_crlam_base_create_restart_argv()` . . . 13
  - 2.6.5 `lam_ssi_crlam_base_do_exec()` . . . 13
- 2.7 Utility Functions for crmpi . . . 13
  - 2.7.1 `lam_ssi_crmpi_base_checkpoint()` . . . 13
  - 2.7.2 `lam_ssi_crmpi_base_continue()` . . . 14
  - 2.7.3 `lam_ssi_crmpi_base_restart()` . . . 14

3 cr SSI Module API **14**
- 3.1 crlam SSI Module API . . . 14
  - 3.1.1 Data Item: `lscrl_meta_info` . . . 16
  - 3.1.2 Function Call: `lscrl_query` . . . 16
  - 3.1.3 Function Call: `lscrla_checkpoint` . . . 16
  - 3.1.4 Function Call: `lscrla_continue` . . . 17
  - 3.1.5 Function Call: `lscrla_disable_checkpoint` . . . 17
  - 3.1.6 Function Call: `lscrla_enable_checkpoint` . . . 17
  - 3.1.7 Function Call: `lscrla_finalize` . . . 18
  - 3.1.8 Function Call: `lscrla_init` . . . 18
  - 3.1.9 Function Call: `lscrla_restart` . . . 19
- 3.2 crmpi SSI Module API . . . 19
  - 3.2.1 Data Item: `lscrm_meta_info` . . . 20
  - 3.2.2 Function Call: `lscrm_query` . . . 20
  - 3.2.3 Function Call: `lscrma_init` . . . 21
  - 3.2.4 Function Call: `lscrma_finalize` . . . 21

3.2.5 Function Call: `lscrma_app_suspend` 21

4 To Be Determined **22**

5 Acknowledgements **22**

References **22**

List of Figures

1 mpirun uses its crlam module to notify the crmpi module in each of the MPI processes to effect checkpoint/restart functionality. 6
2 Overview of the crmpi module management role. 7
3 lam_ssi_crlam_1_0_0_t: The crlam basic type for exporting the module meta information and initial query function pointer. 15
4 lam_ssi_crlam_actions_1_0_0_t: The crlam type for exporting API function pointers. 15
5 struct lam_ssi_crmpi_1_0_0: The crmpi basic type for exporting the module meta information and function pointers. 19
6 struct lam_ssi_crmpi_actions_1_0_0: The crmpi type for exporting API function pointers. 20

1 Overview

Before reading this document, readers are strongly encouraged to read the general LAM/MPI System Services Interface (SSI) overview ([1]). This document uses the terminology and structure defined in that document.

The cr SSI type is used to provide checkpoint-restart support to MPI-1 jobs.[1] Its primary responsibility is to ensure that all the processes in an MPI job arrive at a consistent global state before their process images are saved to stable storage by interfacing to a back-end checkpointing system.

1.1 General Scheme

LAM's cr framework is modeled on involuntary coordinated checkpoint/restart functionality. cr modules provide the necessary coordination, bookkeeping, and interface to back-end checkpoint systems.

`mpirun` is initial entry point to checkpoint and restart MPI applications. A checkpoint is initiated by the asynchronous delivery of a checkpoint request to `mpirun`.[2] `mpirun` propagates this request out to all MPI processes and then allows itself to be checkpointed. The MPI processes are interrupted and may coordinate amongst themselves to prepare for checkpoint (e.g., by draining all in-flight messages). When ready, the MPI processes allow themselves to be checkpointed. After a successful checkpoint, both `mpirun` and the MPI processes continue as if nothing had happened.

An MPI job can be restarted by restoring the `mpirun` process image that was saved at checkpoint time.[3] The resumed `mpirun` process will `exec()` a new copy of itself with an application schema suitable for restarting the MPI application. The MPI processes will be restarted and may coordinate amongst each other to re-create network connections, etc. When the MPI layer has fully re-established itself, control will be returned to the MPI application as if nothing had happened. `mpirun` also resumes waiting for the MPI applications to terminate (i.e., its normal function).

1.2 Terminology

The following sections define terminology that is used throughout this document.

1.2.1 The cr SSI Type

The cr SSI type is composed of two SSI subtypes: crmpi and crlam. Hence, a given cr component is actually comprised of *two* modules – one of each subtype. These two modules work together at run-time to effect the overall checkpoint/restart functionality. Specifically, the two modules will both have the same name and will therefore be considered together during the selection process.

crmpi modules are used to invoke checkpoint/restart functionality in MPI processes (i.e., processes that invoke MPI_INIT and MPI_FINALIZE). Coordination between running MPI processes at checkpoint and restart time is accomplished, in part, by `mpirun`.[4] `mpirun` uses crlam modules to perform this coordination and bookkeeping. See Figure 1.

These two SSI kinds have much common functionality, but are unique in tangible ways. Throughout the rest of the document, the two SSI types will be referred to collectively as the cr SSI when describing common features data structures, APIs, etc., and individually as crlam or crmpi when functionality that is unique to that type is discussed.

[1] MPI-2 jobs – particularly dynamic jobs – are not yet supported.

[2] Currently, the checkpoint/restart request delivery utilizes module-specific mechanisms.

[3] Similar to the checkpoint request, this is through a cr module-specific mechanism.

[4] Checkpointing and restarting non-MPI programs (and therefore `lamexec`) is not currently supported.

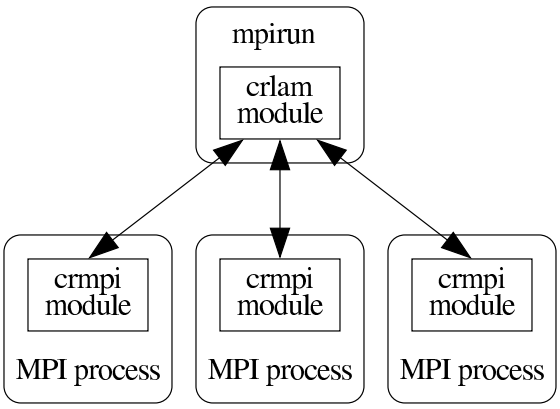

Figure 1: `mpirun` uses its crlam module to notify the crmpi module in each of the MPI processes to effect checkpoint/restart functionality.

1.2.2 Action Points

Three main action points are defined in the cr SSI model:

- Checkpoint: this action is invoked when a checkpoint request is received.
- Continue: this action is invoked immediately after a successful checkpoint. No processes will have migrated.
- Restart: this action is invoked upon restart, and is effectively the same as continue, except that some processes may have migrated to new locations.

1.3 Restrictions and Limitations

Each back-end checkpointing system may impose restrictions on exactly what the MPI process may have checkpointed and restarted. Although other MPI SSI types (e.g., rpi and coll) are responsible for ensuring that their resources can be reliably checkpointed and restarted, all non-MPI resources (e.g., file descriptors, shared memory, etc.) are the responsibility of the MPI application. Such resources may or may not be automatically saved/restored by the back-end checkpoint system. cr module authors are strongly encouraged to clearly document exactly what system-level resources will automatically be saved and restored.

MPI applications may only be checkpointed after all MPI processes have invoked MPI_INIT. Once any MPI process in the overall application invokes MPI_FINALIZE, the job is no longer checkpointable. Application authors are advised to invoke MPI_INIT as close to the beginning of `main()` as possible, and invoke MPI_BARRIER immediately before MPI_FINALIZE in order to maximize the window of checkpoint availability.

An MPI process is checkpointable when the following conditions are met:

- The MPI process is operating at MPI_THREAD_SERIALIZED or higher.[5]
- A valid pair of cr modules have been selected (crlam and crmpi). This may imply a specific MPI thread level.
- The selected rpi module is cr-aware.

[5]This condition may be relaxed in future versions of this API. See Section 4 on page 22.

- The available coll module(s) is(are) cr-aware.

"cr-aware" means that rpi and coll modules are able to coordinate with the crmpi module in order to prepare for, continue after, and restart after a checkpoint. Hence, the crmpi module type is a manager of the other MPI SSI module types; when a checkpoint request arrives, the crmpi module simply invokes "prepare to checkpoint" methods on the rpi and coll modules (see Figure 2). These modules then do whatever is required to prepare themselves for checkpointing. A similar pattern is used after a successful checkpoint and when restarting from a prior checkpoint. This preserves an abstraction barrier between the crmpi module and system resources used by the underlying network(s) used by the MPI process.

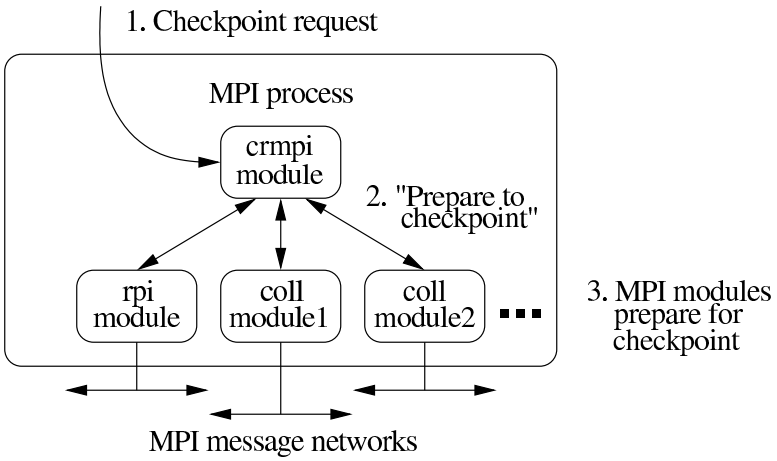

Figure 2: The crmpi module receives an asynchronous checkpoint request. It notifies the other MPI modules to prepare themselves for checkpoint. The MPI modules, each of which may be using its own private network for MPI messages, is responsible for all aspects of checkpointing and restarting itself. The crmpi module, therefore, is not involved in the details of closing/re-opening network devices, etc.

Since `mpirun` is used as the entry point for checkpoint requests, the use of the "`-nw`" (no wait) switch is not supported. Specifically, if an MPI application is launched with "`-nw`", it cannot be checkpointed.

Also note that the LAM universe is *not* checkpointed or restarted. The cr SSI type only saves and restarts `mpirun` and MPI processes. The LAM run-time environment is considered transient without permanent state that needs to be saved. It is an exercise outside the scope of this document to reliably kill and restart the LAM universe for the checkpointed MPI jobs to run in. This may involve some kind of interaction with a resource scheduler; for example, it is possible to checkpoint an MPI job on one set of nodes, kill the LAM universe that it was running in, start a new LAM universe on a different set of nodes, and restart the MPI application in the new universe.

1.4 The crlam SSI Type

Section 3.1 details the API for crlam modules. It is relatively small and assumes asynchronous handling created by the crlam initialization function. The following is a typical chronology of events in `mpirun` during a checkpointed MPI application (items marked with $^{(\text{API})}$ represent events involved with crlam API function calls):

1. The user invokes `mpirun C mpi_application`.
2. `mpirun` launches `mpi_application` across the LAM universe.

3. `mpirun` is told which crlam module to use.[6] This indicates that all MPI processes have invoked MPI_INIT.

4. `mpirun` opens, selects, and initializes the specified crlam module.(API)

5. `mpirun` indicates that it is able to checkpoint the MPI application.(API)

6. `mpirun` goes into a blocking loop waiting for any MPI process to die.

7. **A checkpoint request arrives.**

8. Appropriate handlers must be active in `mpirun` (the exact mechanisms are crlam-module specific; signal handlers and separate threads that were previously created by the crlam module initialization function are typical approaches) that can receive asynchronous checkpoint requests. Note that these requests will likely interrupt `mpirun` in a blocking LAM library call. Since the LAM library is not currently thread-safe, some kinds of actions must be performed either outside of the LAM library or in a new [child] process that can freely utilize the LAM library.

   The following must occur at each action point:

   - Checkpoint:
     - Propagate the checkpoint request to the MPI processes.(API)
     - Allow `mpirun` to be checkpointed.
   - Continue:
     - No additional processing is required.(API)
   - Restart:
     - Re-exec `mpirun` with an application schema suitable for restarting the entire MPI application. Re-execing `mpirun` avoids the problem of breaking out of the blocking LAM library call by effectively re-initializing `mpirun` with a "new" parallel application.(API)

9. Resume the blocking loop waiting for MPI processes to terminate.

10. When one MPI process invokes MPI_FINALIZE, `mpirun` indicates that it is no longer able to checkpoint the MPI application.(API)

11. `mpirun` completes the loop, exiting after all MPI processes have invoked MPI_FINALIZE.(API)

1.5 The crmpi SSI Type

LAM's checkpoint/restart model is built upon involuntary checkpoints; checkpoint requests should be able to be received and handled at any time. Similar to crlam, the crmpi SSI type allows for each module to define its own mechanisms for asynchronous interruptions, although signals and separate threads are commonly used.

As discussed in Section 1.3, the crmpi SSI type is only responsibly for the high-level coordination required in MPI processes for receiving checkpoint requests, coordinating the checkpoint, continue, and restart action points, and interfacing to the back-end checkpointing system. crmpi modules are *not* responsible for closing/re-establishing network connections, draining "in flight" MPI messages, replaying message streams, or any other prepare-to-checkpoint/restart-from-checkpoint actions. crmpi modules invoke

[6] The MPI processes collectively decide which crmpi module to use, and then send the name back to `mpirun`.

methods on other MPI modules for "prepare to checkpoint," "continue after checkpoint," and "restart after checkpoint."

Section 3.2 details the API for crmpi modules. It is even smaller than the crlam API and assumes asynchronous handling created by the crmpi initialization function. The following is a typical chronology of events in an MPI process that is checkpointed (items marked with (API) represent events involved with crlam API function calls):

1. The MPI process invokes MPI_INIT.
2. A set of SSI modules are opened, selected, and initialized for the process, including a crmpi module.(API)
3. The name of the selected crmpi module is sent back to `mpirun` (so that it can select the corresponding crlam module).
4. MPI_INIT returns and the MPI process progresses.
5. **A checkpoint request arrives.**
6. Appropriate handlers must be active in the MPI process (the exact mechanisms are crmpi-module specific; signal handlers and separate threads that were previously created by the crmpi module initialization function are typical approaches) that can receive asynchronous checkpoint requests. When the checkpoint request arrives, the MPI process may or may not be in an MPI function call.
   - If the MPI process is not in an MPI function call, the crmpi module can continue with the checkpoint procedure.
   - If the MPI process is in an MPI function call, it must either wait for the function call to finish or interrupt it[7] (deadlocks are possible, for example, if the MPI process is blocking while receiving an MPI message that was not sent before the checkpoint).
7. The crmpi module gains control of and locks the MPI library[8] to prevent the MPI process from [re-]entering the MPI library during the checkpoint/continue/restart action points). The following must occur at each action point:
   - Checkpoint:
     - Coordinate with the other active MPI modules to prepare them for checkpoint.
     - Allow the MPI process to be checkpointed.
   - Continue:
     - Coordinate with the other active MPI modules to continue after the successful checkpoint.
   - Restart:
     - Coordinate with the other active MPI modules to restart after a prior successful checkpoint.
8. Unlock the MPI library and return control to the main MPI process.
9. The MPI process invokes MPI_FINALIZE. The crmpi module is finalized and closed.(API)

[7] The interrupt mechanism is not yet standardized. See Section 4 on page 22.
[8] The locking mechanism is standardized but not yet well abstracted to the cr type. See Section 4 on page 22.

2 Services Provided by the cr SSI

Several services are provided by the cr SSI that are available to all cr 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 cr SSI services described in this document:

```
#include <lam-ssi.h>
#include <lam-ssi-cr.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 [1], 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 selection of a cr SSI module persists through the life of an MPI job. It is selected during MPI_INIT and remains selected until MPI_FINALIZE. The MPI processes in the job collectively decide on which crmpi module to select and then pass its name to `mpirun`. `mpirun` then selects the corresponding crlam module (which may be none). Hence, `mpirun` and all of the MPI processes that it launched all share a common cr module selection.

2.3 Types

Some types are used in different API calls and throughout the checkpoint/restart-aware RPIs.

2.3.1 `lam_ssi_crmpi_base_handler_state_t`

This type is an enumerated value used to describe the state of the crmpi handler thread in an MPI process:

```
typedef enum {
  LAM_SSI_CRMPI_BASE_HANDLER_STATE_IDLE,
  LAM_SSI_CRMPI_BASE_HANDLER_STATE_WAITING,
  LAM_SSI_CRMPI_BASE_HANDLER_STATE_RUNNING,

  LAM_SSI_CRMPI_BASE_STATE_MAX
} lam_ssi_crmpi_base_handler_state_t;
```

The three possible values are:

- HANDLER_STATE_IDLE: The handler thread is inactive (and likely either non-existent or blocking, depending on the crmpi module's implementation).
- HANDLER_STATE_WAITING: The handler thread is waiting for the main application thread to either leave the MPI library, or yield control.
- HANDLER_STATE_RUNNING: The handler thread is actively working on a checkpoint, continue, or restart operation.

2.4 Global Variables

Several global variables are available to all cr modules. These variables are extern'ed in <lam-ssi-cr.h>.

2.4.1 int lam_ssi_cr_did

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

Debug streams should be used in conjunction with lam_ssi_cr_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_cr_verbose

The lam_ssi_cr_verbose variable will be set by the cr SSI initialization function, and will therefore be usable by every cr API function.

lam_ssi_cr_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.

2.5 Global Variables for crmpi

The global variables listed below are available to crmpi modules. These variables are extern'ed in <lam-ssi-cr.h>.

2.5.1 lam_ssi_crmpi_base_handler_state

This variable is available in crmpi only. It is of type lam_ssi_crmpi_base_handler_state_t, described in Section 2.3.1. This variable must be maintained by the crmpi handler thread to track its current status in order notify other SSI modules that they are being interrupted.

```
volatile lam_ssi_crmpi_base_handler_state_t lam_ssi_crmpi_base_handler_state;
```

It may be necessary for the application thread(s) to perform some action based on the execution state of the thread-based checkpoint/restart handler. This variable is marked volatile so that it will not be cached by the application thread(s) if the crmpi handler thread changes its value.

2.6 Utility Functions for crlam

The general utility functions listed below are provided to all modules by the crlam SSI.

2.6.1 lam_ssi_crlam_base_checkpoint()

```
int lam_ssi_crlam_base_checkpoint(struct _gps *world, int nprocs);
```

This function must be invoked from the thread-based checkpoint/restart handler when mpirun receives a checkpoint request. After preparing mpirun[9] to be checkpointed, it will invoke the selected crlam module's lscrla_checkpoint() API function (see Section 3.1.3). The world array must contain a struct _gps element for each MPI process that needs to receive the checkpoint request. nprocs is the length of the world array.

It returns zero on success, LAMERROR otherwise.

2.6.2 lam_ssi_crlam_base_continue()

```
int lam_ssi_crlam_base_continue(void);
```

This function must be invoked from the thread-based checkpoint/restart handler when mpirun continues after a successful checkpoint. After preparing mpirun to continue after the checkpoint, it will invoke the selected crlam module's lscrla_continue() API function (see Section 3.1.4).

It returns zero on success, LAMERROR otherwise.

2.6.3 lam_ssi_crlam_base_restart()

```
int lam_ssi_crlam_base_restart(char *executable, char *app_schema);
```

This function must be invoked from the signal-based checkpoint/restart handler when mpirun restarts. After preparing mpirun to be checkpointed, it will invoke the selected crlam module's lscrla_restart() API function (see Section 3.1.4). executable should be a string path to mpirun to re-launch, and app_schema should be the string path to the application schema that can be used to re-launch the MPI application.

It returns zero on success, LAMERROR otherwise.

[9] This function will likely also be used in lamexec if checkpointing of non-MPI programs is ever supported.

2.6.4 `lam_ssi_crlam_base_create_restart_argv()`

int lam_ssi_crlam_base_create_restart_argv(**char** **argv, OPT *ad);

Based on arguments `argv` and `ad`, this function creates an internal list of arguments that will be given to `mpirun` when it is restarted. This function is typically invoked by the crlam's `lscra_init()` API function with the `argv` and `ad` parameters that it receives as arguments (see Section 3.1.8).

The internal set of arguments that is created is not returned to the caller; it is used by `lam_ssi_crlam_base_do_exec()`. Note that `OPT *` is an internal LAM type for holding command line parameters. Its use is documented in the `all_opt(3)` manual page.

It returns zero on success, LAMERROR otherwise.

2.6.5 `lam_ssi_crlam_base_do_exec()`

int lam_ssi_crlam_base_do_exec(**char** *executable, **char** *app_schema);

The use of this function is optional; if it is not used, cr modules need to provide equivalent functionality.

This function can be either invoked by the crlam module's `lscr_restart()` API function call (see Section 3.1.9), or is used as the `lscr_restart()` API function itself. The utility function `lam_crlam_base_create_restart_argv()` must have been invoked before this function is called.

This utility function launches a new `mpirun` process via `execve(2)` with the command line arguments that were previously setup, and an application schema that can re-launch the MPI job. Since this function `execve()`'s a new process, it never returns.

It returns zero on success, LAMERROR otherwise.

2.7 Utility Functions for crmpi

The general utility functions listed below are provided to all modules by the crmpi SSI.

2.7.1 `lam_ssi_crmpi_base_checkpoint()`

int lam_ssi_crmpi_base_checkpoint(**void**);

This function should be invoked from the MPI thread-based checkpoint/restart handler when a checkpoint request arrives. `crmpi_base_checkpoint()` is used to prepare the other SSI modules in use by the MPI process to be checkpointed.

This function invokes the appropriate action function exported by each SSI module that needs pre-checkpoint handling. Currently, this means:

- The function `lsra_checkpoint()` rpi API function will be invoked on all active `rpi` SSI modules.
- The function `lsca_checkpoint()` coll API function will be invoked on each existing MPI communicator.

`crmpi_base_checkpoint()` will return 0 if all the invoked functions return 0, or LAMERROR if any of them returned LAMERROR.

2.7.2 `lam_ssi_crmpi_base_continue()`

int lam_ssi_crmpi_base_continue(**void**);

This function should be invoked from the MPI thread-based checkpoint/restart handler after a checkpoint completes successfully. `crmpi_base_continue()` is used to trigger post-checkpoint actions in other SSI modules in use by the MPI process after a checkpoint.

This function invokes the appropriate action function exported by each SSI module that needs pre-checkpoint handling. Currently, this means:

- The `lsra_continue()` rpi API function will be invoked on all active `rpi` SSI modules.
- The `lsca_continue()` coll API function will be invoked on each existing MPI communicator.

`crmpi_base_continue()` will return 0 if all the invoked functions return 0, or LAMERROR if any of them returned LAMERROR.

2.7.3 `lam_ssi_crmpi_base_restart()`

int lam_ssi_crmpi_base_restart(**void**);

This function should be invoked from the thread-based checkpoint/restart handler when an MPI process is restarted. `crmpi_base_restart()` is used to trigger pre-restart actions in other SSI modules in use by the MPI process when it is restarted.

This function invokes the appropriate action function exported by each SSI module that needs pre-restart handling. Currently, this means:

- The `lsra_restart()` rpi API function will be invoked on all active `rpi` SSI modules.
- The `lsca_restart()` coll API function will be invoked on each existing MPI communicator.

`crmpi_base_restart()` will return 0 if all the invoked functions return 0, or LAMERROR if any of them returned LAMERROR.

3 cr SSI Module API

3.1 crlam SSI Module API

This is version 1.0.0 of the crlam SSI module API.

Each crlam SSI module must export a `lam_ssi_crlam_1_0_0_t` named `lam_ssi_crlam_<name>_module`. This type is defined in Figure 3. A second `struct` is used to hold the majority of function pointers and flags for the module. It is only used if the module is selected, and is shown in Figure 4.

The majority of the elements in Figures 3, and 4 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_crlam_1_0_0 {
  lam_ssi_1_0_0_t lscrl_meta_info;

  /* cr lam API function pointers */

  lam_ssi_crlam_query_fn_t lscrl_query;
} lam_ssi_crlam_1_0_0_t;
```

Figure 3: `lam_ssi_crlam_1_0_0_t`: The crlam basic type for exporting the module meta information and initial query function pointer.

```
typedef struct lam_ssi_crlam_actions_1_0_0 {

  /* cr lam action API functions pointers */

  lam_ssi_crlam_checkpoint_fn_t lscrla_checkpoint;
  lam_ssi_crlam_continue_fn_t lscrla_continue;
  lam_ssi_crlam_disable_checkpoint_fn_t lscrla_disable_checkpoint;
  lam_ssi_crlam_enable_checkpoint_fn_t lscrla_enable_checkpoint;
  lam_ssi_crlam_finalize_fn_t lscrla_finalize;
  lam_ssi_crlam_init_fn_t lscrla_init;
  lam_ssi_crlam_restart_fn_t lscrla_restart;
} lam_ssi_crlam_actions_1_0_0_t;
```

Figure 4: `lam_ssi_crlam_actions_1_0_0_t`: The crlam type for exporting API function pointers.

3.1.1 Data Item: `lscrl_meta_info`

`lscrl_meta_info` is the SSI-mandated element and contains meta-information about the module. See [1] for more information about this element.

3.1.2 Function Call: `lscrl_query`

- Type: `lam_ssi_crlam_query_fn_t`

  typedef lam_ssi_crlam_actions_t *(*lam_ssi_crlam_query_fn_t)(**int** *priority);

- Arguments:
  - OUT: `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
- Return value: Either `NULL` or a pointer to the `struct` shown in Figure 4.
- Description: If the module wants to be considered for selection, it should return a pointer to the `struct` shown in Figure 4 that is filled with relevant data, and assign an associated priority to `priority`. See [1] for more details on the priority system and how modules are selected at run time.

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

3.1.3 Function Call: `lscrla_checkpoint`

- Type: `lam_ssi_crlam_checkpoint_fn_t`

  typedef int (*lam_ssi_crlam_checkpoint_fn_t)(**void**)

- Arguments: None.
- Return value: Zero on success, LAMERROR otherwise.
- Description: This function is invoked by the `lam_ssi_crlam_base_checkpoint()` utility function (which was, in turn, invoked by the thread-based handler when `mpirun` received the checkpoint request). The stated purpose of this function is to prepare `mpirun` and the MPI processes it launched for checkpoint. This typically means propagating the checkpoint request out to all MPI processes started by `mpirun` and creating an application schema suitable for using to restart the MPI application.

  Note that `mpirun` itself will likely be blocking in the LAM function call `rpwait()` ("remote process wait") while waiting for child processes to complete. The LAM infrastructure is currently not thread-safe, and therefore cannot handle any LAM daemon calls from the same process during this API call. If this function needs to use LAM infrastructure API calls (such as `nsend()`/`nrecv()`, `rploadgov()`, etc.), the recommended solution is to `fork()` from `mpirun`, register the new process as a LAM process, and invoke the necessary LAM API calls in the new process.

  Although the actions performed by this function could be contained within the `mpirun` thread handler function itself, separating it out into a distinct function provided a clean abstraction for this specific

functionality. It also provides for future compatibility when/if LAM ever provides the thread-based handler itself (instead of having that provided by the crlam module).

3.1.4 Function Call: `lscrla_continue`

- Type: `lam_ssi_crlam_continue_fn_t`

  typedef int (*lam_ssi_crlam_continue_fn_t)(**void**)

- Arguments: None.
- Return value: Zero on success, LAMERROR otherwise.
- Description: This function is invoked by the `lam_ssi_crlam_base_continue()` utility function (which was, in turn, invoked by the thread-based handler after a successful checkpoint). The stated purpose of this function is to continue `mpirun` and the MPI processes it launched after a successful checkpoint. This may involve some recovery actions, or it may be an no-op.

  The same restrictions apply to this function with regards to LAM API calls as with the `lscrla_checkpoint()` API call.

  Although the actions performed by this function could be contained within the `mpirun` thread handler function itself, separating it out into a distinct function provided a clean abstraction for this specific functionality. It also provides for future compatibility when/if LAM ever provides the thread-based handler itself (instead of having that provided by the crlam module).

3.1.5 Function Call: `lscrla_disable_checkpoint`

- Type: `lam_ssi_crlam_disable_checkpoint_fn_t`

  typedef void (*lam_ssi_crlam_disable_checkpoint_fn_t)(**void**)

- Arguments: None.
- Return value: None.
- Description: This function is called by `mpirun` to indicate when it is not permissible to allow checkpoints. Hence, if a checkpoint request arrives after `mpirun` has invoked this function, it must be stalled until a corresponding `lscrla_enable_checkpoint()` API call is invoked.

  Note that `mpirun` is implicitly in a "disable" state after returning from `lscrla_init()`; checkpoint requests should always be deferred until at least the first invocation of `lscrla_enable_checkpoint()`.

3.1.6 Function Call: `lscrla_enable_checkpoint`

- Type: `lam_ssi_crlam_enable_checkpoint_fn_t`

  typedef void (*lam_ssi_crlam_enable_checkpoint_fn_t)(**void**)

- Arguments: None.

- Return value: None.
- Description: This function is called by `mpirun` to indicate when it is permissible to allow checkpoints. Hence, if a checkpoint request arrives after `mpirun` has invoked this function, both `mpirun` and the MPI job can be checkpointed.

  Note that `mpirun` is implicitly in a "disable" state after returning from `lscrla_init()`; checkpoint requests should always be deferred until at least the first invocation of this function.

3.1.7 Function Call: `lscrla_finalize`

- Type: `lam_ssi_crlam_finalize_fn_t`

```
typedef int (*lam_ssi_crlam_finalize_fn_t)(void);
```

- Arguments: None.
- Return value: Zero on success, LAMERROR otherwise.
- Description: Performs the final cleanup of the checkpoint/restart handlers and possibly also invoke routines to detach from the underlying checkpointer.

3.1.8 Function Call: `lscrla_init`

- Type: `lam_ssi_crlam_init_fn_t`

```
typedef int (*lam_ssi_crlam_init_fn_t)(char *path, char **argv, OPT *ad,
  struct _gps *mpiworld, int world_n);
```

- Arguments:
  - IN: `path` is the name of the executable to launch at restart time (e.g., "`mpirun`").
  - IN: `argv` is the list of arguments that were specified at the `mpirun` command-line, and is used to build a list of arguments to be passed to the new `mpirun` at restart.
  - IN: `ad` is used for option-parsing of the argv in order to build a restart-time argv that will be passed to `mpirun`.
  - IN: `mpiworld` is used to pass the GPS information about all the processes in the MPI job. This information is required to propagate the checkpoint requests from `mpirun` to all the MPI processes.
  - IN: `world_n` is used to pass the count of processes that are part of the MPI job. This information is required to propagate the checkpoint requests from `mpirun` to all the MPI processes.
- Return value: Zero on success, LAMERROR otherwise.
- Description: Performs the primary initialization of the crlam sub-layer. This function typically registers the signal-based and thread-based callbacks to perform the actual checkpoint/restart functionality. Any initialization that is specific to the underlying checkpointing system is also performed here.

  A minimum requirement for initiating a checkpoint of a LAM/MPI job is the delivery of a signal to `mpirun`. If the underlying checkpointer does not provide a mechanism to create/manage the

thread-based checkpoint/restart handler, then a thread has to be explicitly created in this initialization function, and it will have to be blocked from executing (say, by waiting on `read` from a pipe). This thread could then be woken up to start executing when a checkpoint request comes in (say, by writing to the pipe on which the threaded callback is blocked on a `read`, from signal handler-context).

3.1.9 Function Call: `lscrla_restart`

- Type: `lam_ssi_crlam_restart_fn_t`

```
typedef int (*lam_ssi_crlam_restart_fn_t)(void)
```

- Arguments: None.
- Return value: Zero on success, LAMERROR otherwise.
- Description: This function is invoked by the `lam_ssi_crlam_base_restart()` utility function (which was, in turn, invoked by the signal-based handler upon restart). The stated purpose of this function is to restart `mpirun` with a new application schema so that it can restart the MPI application.

  Note that this function runs in signal handler context, and is therefore subject to many restrictions (e.g., it can't even call `malloc()`. Module authors are reminded to be extremely cautious about what this function does.

  Although the actions performed by this function could be contained within the `mpirun` signal handler function itself, separating it out into a distinct function provided a clean abstraction for this specific functionality. It also provides for future compatibility when/if LAM ever provides the signal-based handler itself (instead of having that provided by the crlam module).

3.2 crmpi SSI Module API

This is version 1.0.0 of the crmpi SSI module API.

Each crmpi SSI module must export a `struct lam_ssi_crmpi_1_0_0` named `lam_ssi_crmpi_<name>_module`. This type is defined in Figure 5. A second `struct` is used to hold the majority of function pointers and flags for the module. It is only used if the module is selected, and is shown in Figure 6.

```
typedef struct lam_ssi_crmpi_1_0_0 {
  lam_ssi_1_0_0_t lscrm_meta_info;

  /* cr mpi API function pointers */

  lam_ssi_crmpi_query_fn_t lscrm_query;
  lam_ssi_crmpi_init_fn_t lscrm_init;
} lam_ssi_crmpi_1_0_0_t;
```

Figure 5: `struct lam_ssi_crmpi_1_0_0`: The crmpi basic type for exporting the module meta information and function pointers.

```
typedef struct lam_ssi_crmpi_actions_1_0_0 {

  /* cr mpi action API functions pointers */

  lam_ssi_crmpi_finalize_fn_t lscrma_finalize;
  lam_ssi_crmpi_app_suspend_fn_t lscrma_app_suspend;
} lam_ssi_crmpi_actions_1_0_0_t;
```

Figure 6: `struct lam_ssi_crmpi_actions_1_0_0`: The crmpi type for exporting API function pointers.

3.2.1 Data Item: `lscrm_meta_info`

`lscrm_meta_info` is the SSI-mandated element and contains meta-information about the module. See [1] for more information about this element.

The data in this element should be the same as the data contained in the corresponding crlam meta element (`lscrl_meta_info`, Section 3.1.1).

3.2.2 Function Call: `lscrm_query`

- Type: `lam_ssi_crmpi_query_fn_t`

  ```
  typedef int (*lam_ssi_crmpi_query_fn_t)(int *priority, int *thread_min, int *thread_max);
  ```

- Arguments:
  - OUT: `priority` is the priority of this module, and is used to choose which module will be selected from the set of available modules at run time.
  - OUT: `thread_min` is the minimum MPI thread level that this module supports. Only meaningful if the function returns zero and a non-negative priority.
  - OUT: `thread_max` is the maximum MPI thread level that this module supports. Only meaningful if the function returns zero and a non-negative priority.

- Return value: Zero on success, LAMERROR otherwise.

  Either NULL or a pointer to the `struct` shown in Figure 6.

- Description: If the module wants to be considered for selection, it must return zero, assign an associated priority to `priority`, and fill `thread_min` and `thread_max` with the minimum[10] and maximum MPI thread levels that it can operate in.

  See [1] for more details on the priority system and how modules are selected at run time.

  If the module does not want to be considered during the negotiation for this application, it should return LAMERROR (the values in `priority`, `thread_min`, and `thread_max` are then ignored).

[10] This value must currently be at least MPI_THREAD_SERIALIZED, although this condition may be relaxed in future versions of this API. See Section 4.

3.2.3 Function Call: `lscrma_init`

- Type: `lam_ssi_crmpi_init_fn_t`

  typedef lam_ssi_crmpi_actions_t *(*lam_ssi_crmpi_init_fn_t)(**void**);

- Arguments: None.
- Return value: A pointer to the `struct` shown in Figure 6, or NULL on error.
- Description: Performs the primary initialization of the crmpi SSI module and returns a pointer to a `lam_ssi_crmpi_actions_t` filled with function pointers for the actions of this module.

  This function typically registers signal-based and thread-based callbacks to perform the actual checkpoint/restart functionality (or launches a thread that sleeps until the appropriate time). Any initialization that is specific to the underlying checkpointing system is also performed here. Typically, one or more mutual exclusion devices might need to be initialized in this function that will be used to synchronize the execution of the main application thread and the checkpoint/restart handler thread within the MPI library.

  A minimum requirement for initiating a checkpoint of a LAM/MPI job is the delivery of a signal to each process in the MPI application. If the underlying checkpointer does not provide a mechanism to create/manage the thread-based checkpoint/restart handler, then a thread has to be explicitly created in this initialization function, and it will have to be blocked from executing (for example, by blocking on `read()` from a pipe). This thread could then be woken up to start executing when a checkpoint request arrives (for example, by writing to the pipe on which the threaded callback is blocked on a `read()` since `write()` is safe in signal handler context).

3.2.4 Function Call: `lscrma_finalize`

- Type: `lam_ssi_crmpi_finalize_fn_t`

  typedef int (*lam_ssi_crmpi_finalize_fn_t)(**void**);

- Arguments: None.
- Return value: Zero on success, non-zero otherwise.
- Description: Performs the final cleanup of the checkpoint/restart handlers, and possibly also invoke routines to detach from the underlying checkpointing system.

3.2.5 Function Call: `lscrma_app_suspend`

- Type: `lam_ssi_crmpi_app_suspend_fn_t`

```
typedef void (*lam_ssi_crmpi_app_suspend_fn_t)(void);
```

- Arguments: None.
- Return value: None.

- Description: This function is provided to allow the main application thread to yield to the checkpoint/restart thread when it is interrupted in the middle of a blocking MPI call by the threaded handler. This is typically done by using a set of one or more mutual exclusion devices. Once the threaded handler returns from a checkpoint, it will yield control back to the application thread and allow the `lscrma_app_suspend()` call to return.

4 To Be Determined

Things that still need to be addressed:

- It is possible that MPI_THREAD_SERIALIZED may not be required. User-level checkpointers, for example, may not require this. Future versions of this API may relax this constraint.
- The mechanism for a crmpi handler thread to interrupt the application thread in a blocking MPI call is not yet established. The rpiblcr crmpi module uses SIGUSR1 to interrupt the user application in the rpicrtcp rpi. However, this does not necessarily work well with other rpi modules (e.g., Myrinet should *not* be signaled). This needs to be fixed; a standard interrupt mechanism needs to be defined.
- The locking mechanism utilizes a "one big lock" approach that protects the entire MPI library. Although the mechanisms is currently in place, it needs to be abstracted better such that it will both be workable and not incur undue overhead in an MPI_THREAD_MULTIPLE environment.

5 Acknowledgements

This work was supported by a grant from the Lily Endowment National Science Foundation grant 0116050, and by the Director, Office of Science, Office of Advanced Scientific Computing Research, U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

[1] 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.