Title

Parallel Runtime Interface for Fortran (PRIF) Design Document, Revision 0.2



Authors

Rouson, Damian

Richardson, Brad

Bonachea, Dan

et al.

Publication Date

2023-12-20

DOI

10.25344/S4DG6S

# Parallel Runtime Interface for Fortran (PRIF) Design Document Revision 0.2

Damian Rouson, Brad Richardson, Dan Bonachea, Katherine Rasmussen
*Lawrence Berkeley National Laboratory, USA*
lbl-flang@lbl.gov

Lawrence Berkeley National Laboratory Technical Report (LBNL-2001563)
doi:10.25344/S4DG6S

December 20, 2023

**Abstract**

This design document proposes an interface to support the parallel features of Fortran, named the Parallel Runtime Interface for Fortran (PRIF). PRIF is a proposed solution in which the runtime library is responsible for coarray allocation, deallocation and accesses, image synchronization, atomic operations, events, and teams. In this interface, the compiler is responsible for transforming the invocation of Fortran-level parallel features into procedure calls to the necessary PRIF procedures. The interface is designed for portability across shared- and distributed-memory machines, different operating systems, and multiple architectures. Implementations of this interface are intended as an augmentation for the compiler's own runtime library. With an implementation-agnostic interface, alternative parallel runtime libraries may be developed that support the same interface. One benefit of this approach is the ability to vary the communication substrate. A central aim of this document is to define a parallel runtime interface in standard Fortran syntax, which enables us to leverage Fortran to succinctly express various properties of the procedure interfaces, including argument attributes.

**WORK IN PROGRESS** This is still a draft a may continue to evolve. Feedback and questions should be directed to lbl-flang@lbl.gov.

## Changelog

### Revision 0.1

- Identify parallel features
- Sketch out high-level design
- Decide on compiler vs PRIF responsibilities

### Revision 0.2 (Dec. 2023)

- Change name to PRIF
- Fill out interfaces to all PRIF provided procedures
- Write descriptions, discussions and overviews of various features, arguments, etc.

# Contents

# 1 Problem description

In order to be fully Fortran 2023 compliant, a Fortran compiler needs support for what is commonly referred to as coarray fortran, which includes features related to parallelism. These features include the following statements, subroutines, functions, types, and kind type parameters:

- **Statements:**
  - *Synchronization:* `sync all`, `sync images`, `sync memory`, `sync team`
  - *Events:* `event post`, `event wait`
  - *Notify:* `notify wait`
  - *Error termination:* `error stop`
  - *Locks:* `lock`, `unlock`
  - *Failed images:* `fail image`
  - *Teams:* `form team`, `change team`
  - *Critical sections:* `critical`, `end critical`
- **Intrinsic functions:** `num_images`, `this_image`, `lcobound`, `ucobound`, `team_number`, `get_team`, `failed_images`, `stopped_images`, `image_status`, `coshape`, `image_index`
- **Intrinsic subroutines:**
  - *Collective subroutines:* `co_sum`, `co_max`, `co_min`, `co_reduce`, `co_broadcast`
  - *Atomic subroutines:* `atomic_add`, `atomic_and`, `atomic_cas`, `atomic_define`, `atomic_fetch_add`, `atomic_fetch_and`, `atomic_fetch_or`, `atomic_fetch_xor`, `atomic_or`, `atomic_ref`, `atomic_xor`
  - *Other subroutines:* `event_query`
- **Types, kind type parameters, and values:**
  - *Intrinsic derived types:* `event_type`, `team_type`, `lock_type`, `notify_type`
  - *Atomic kind type parameters:* `atomic_int_kind` and `atomic_logical_kind`
  - *Values:* `stat_failed_image`, `stat_locked`, `stat_locked_other_image`, `stat_stopped_image`, `stat_unlocked`, `stat_unlocked_failed_image`

In addition to being able to support syntax related to the above features, compilers will also need to be able to handle new execution concepts such as image control. The image control concept affects the behaviors of some statements that were introduced in Fortran expressly for supporting parallel programming, but image control also affects the behavior of some statements that pre-existed parallelism in standard Fortran:

- **Image control statements:**
  - *Pre-existing statements*: `allocate`, `deallocate`, `stop`, `end`, a `call` referencing `move_alloc` with coarray arguments
  - *New statements:* `sync all`, `sync images`, `sync memory`, `sync team`, `change team`, `end team`, `critical`, `end critical`, `event post`, `event wait`, `form team`, `lock`, `unlock`, `notify wait`

One consequence of the statements being categorized as image control statements will be the need to restrict code movement by optimizing compilers.

# 2 Proposed solution

This design document proposes an interface to support the above features, named Parallel Runtime Interface for Fortran (PRIF). By defining an implementation-agnostic interface, we envision facilitating the development of alternative parallel runtime libraries that support the same interface. One benefit of this approach is the ability to vary the communication substrate. A central aim of this document is to use a parallel runtime interface in standard Fortran syntax, which enables us to leverage Fortran to succinctly express various properties of the procedure interfaces, including argument attributes. See Rouson and Bonachea (2022) for additional details.

## 2.1 Parallel Runtime Interface for Fortran (PRIF)

The Parallel Runtime Interface for Fortran is a proposed interface in which the PRIF implementation is responsible for coarray allocation, deallocation and accesses, image synchronization, atomic operations, events,

and teams. In this interface, the compiler is responsible for transforming the invocation of Fortran-level parallel features to add procedure calls to the necessary PRIF procedures. Below you can find a table showing the delegation of tasks between the compiler and the PRIF implementation. The interface is designed for portability across shared and distributed memory machines, different operating systems, and multiple architectures. The Caffeine implementation, see below, of the Parallel Runtime Interface for Fortran plans to support the following architectures: x86_64, PowerPC64, AArch64, with the possibility of supporting more as requested. Implementations of this interface are intended as an augmentation for the compiler's own runtime library. While the interface can support multiple implementations, we envision needing to build the PRIF implementation as part of installing the compiler. The procedures and types provided for direct invocation as part of the PRIF implementation shall be defined in a Fortran module with the name `prif`.

## 2.2 Delegation of tasks between the Fortran compiler and the PRIF implementation

The following table outlines which tasks will be the responsibility of the Fortran compiler and which tasks will be the responsibility of the PRIF implementation. A 'X' in the "Fortran compiler" column indicates that the compiler has the primary responsibility for that task, while a 'X' in the "PRIF implementation" column indicates that the compiler will invoke the PRIF implementation to perform the task and the PRIF implementation has primary responsibility for the task's implementation. See the Procedure descriptions for the list of PRIF implementation procedures that the compiler will invoke.

| Tasks | Fortran compiler | Runtime library |
|---|---|---|
| Establish and initialize static coarrays prior to `main` | X | |
| Track corank of coarrays | X | |
| Track local coarrays for implicit deallocation when exiting a scope | X | |
| Initialize a coarray with `SOURCE=` as part of allocate-stmt | X | |
| Provide `lock_type` coarrays for `critical-constructs` | X | |
| Provide final subroutine for all derived types that are finalizable or that have allocatable components that appear in a coarray | X | |
| Track variable allocation status, including resulting from use of `move_alloc` | X | |
| Track coarrays for implicit deallocation at `end-team-stmt` | | X |
| Allocate and deallocate a coarray | | X |
| Reference a coindexed-object | | X |
| Team stack abstraction | | X |
| `form-team-stmt`, `change-team-stmt`, `end-team-stmt` | | X |
| Intrinsic functions related to Coarray Fortran, like `num_images`, etc | | X |
| Atomic subroutines | | X |
| Collective subroutines | | X |
| Synchronization statements | | X |
| Events | | X |
| Locks | | X |
| `critical-construct` | | X |

### 2.2.1 Caffeine - LBL's Implementation of the Parallel Runtime Interface for Fortran

Implementations of some parts of the Parallel Runtime Interface for Fortran exist in Caffeine, a parallel runtime library targeting coarray Fortran compilers. Caffeine will continue to be developed in order to fully implement the proposed Parallel Runtime Interface for Fortran. Caffeine uses the GASNet-EX exascale networking middleware but with the implementation-agnostic interface and the ability to vary the communication substrate, it might also be possible to develop wrappers that would support the proposed interface with OpenCoarrays, which uses the Message Passing Interface (MPI).

## 2.3 Types Descriptions

### 2.3.1 Fortran Intrinsic Derived types

These types will be defined in the PRIF implementation and it is proposed that the compiler will use a rename to use the PRIF implementation definitions for these types in the compiler's implementation of the `ISO_Fortran_Env` module. This enables the internal structure of each given type to be tailored as needed for a given implementation.

#### 2.3.1.1 `prif_team_type`

- implementation for `team_type` from `ISO_Fortran_Env`

#### 2.3.1.2 `prif_event_type`

- implementation for `event_type` from `ISO_Fortran_Env`

#### 2.3.1.3 `prif_lock_type`

- implementation for `lock_type` from `ISO_Fortran_Env`

#### 2.3.1.4 `prif_notify_type`

- implementation for `notify_type` from `ISO_Fortran_Env`

### 2.3.2 Constants in `ISO_FORTRAN_ENV`

These values will be defined in the PRIF implementation and it is proposed that the compiler will use a rename to use the PRIF implementation definitions for these values in the compiler's implementation of the `ISO_Fortran_Env` module.

**2.3.2.1** `PRIF_ATOMIC_INT_KIND` This shall be set to an implementation defined value from the `INTEGER_KINDS` array.

**2.3.2.2** `PRIF_ATOMIC_LOGICAL_KIND` This shall be set to an implementation defined value from the `LOGICAL_KINDS` array.

**2.3.2.3** `PRIF_CURRENT_TEAM` This shall be a value of type `integer(c_int)` that is defined by the implementation and shall be distinct from the values `PRIF_INITIAL_TEAM` and `PRIF_PARENT_TEAM`

**2.3.2.4** `PRIF_INITIAL_TEAM` This shall be a value of type `integer(c_int)` that is defined by the implementation and shall be distinct from the values `PRIF_CURRENT_TEAM` and `PRIF_PARENT_TEAM`

**2.3.2.5** `PRIF_PARENT_TEAM` This shall be a value of type `integer(c_int)` that is defined by the implementation and shall be distinct from the values `PRIF_CURRENT_TEAM` and `PRIF_INITIAL_TEAM`

**2.3.2.6** `PRIF_STAT_FAILED_IMAGE` This shall be a value of type `integer(c_int)` that is defined by the implementation to be negative if the implementation cannot detect failed images and positive otherwise and shall be distinct from `PRIF_STAT_LOCKED`, `PRIF_STAT_LOCKED_OTHER_IMAGE`, `PRIF_STAT_STOPPED_IMAGE`, `PRIF_STAT_UNLOCKED` and `PRIF_STAT_UNLOCKED_FAILED_IMAGE`.

**2.3.2.7** `PRIF_STAT_LOCKED` This shall be a value of type `integer(c_int)` that is defined by the implementation and shall be distinct from `PRIF_STAT_FAILED_IMAGE`, `PRIF_STAT_LOCKED_OTHER_IMAGE`, `PRIF_STAT_STOPPED_IMAGE`, `PRIF_STAT_UNLOCKED` and `PRIF_STAT_UNLOCKED_FAILED_IMAGE`.

**2.3.2.8** `PRIF_STAT_LOCKED_OTHER_IMAGE` This shall be a value of type `integer(c_int)` that is defined by the implementation and shall be distinct from `PRIF_STAT_FAILED_IMAGE`, `PRIF_STAT_LOCKED`, `PRIF_STAT_STOPPED_IMAGE`, `PRIF_STAT_UNLOCKED` and `PRIF_STAT_UNLOCKED_FAILED_IMAGE`.

**2.3.2.9** `PRIF_STAT_STOPPED_IMAGE` This shall be a positive value of type `integer(c_int)` that is defined by the implementation and shall be distinct from `PRIF_STAT_FAILED_IMAGE`, `PRIF_STAT_LOCKED`, `PRIF_STAT_LOCKED_OTHER_IMAGE`, `PRIF_STAT_UNLOCKED` and `PRIF_STAT_UNLOCKED_FAILED_IMAGE`.

**2.3.2.10** `PRIF_STAT_UNLOCKED` This shall be a value of type `integer(c_int)` that is defined by the implementation and shall be distinct from `PRIF_STAT_FAILED_IMAGE`, `PRIF_STAT_LOCKED`, `PRIF_STAT_LOCKED_OTHER_IMAGE`, `PRIF_STAT_STOPPED_IMAGE` and `PRIF_STAT_UNLOCKED_FAILED_IMAGE`.

**2.3.2.11** `PRIF_STAT_UNLOCKED_FAILED_IMAGE` This shall be a value of type `integer(c_int)` that is defined by the implementation and shall be distinct from `PRIF_STAT_FAILED_IMAGE`, `PRIF_STAT_LOCKED`, `PRIF_STAT_LOCKED_OTHER_IMAGE`, `PRIF_STAT_STOPPED_IMAGE` and `PRIF_STAT_UNLOCKED`.

### 2.3.3 PRIF specific types

These types are used to represent opaque "descriptors" that can be passed to and from the PRIF implementation between operations.

#### 2.3.3.1 `prif_coarray_handle`

- a derived type provided by the PRIF implementation and that will be opaque to the compiler that represents a reference to a coarray variable is used for coarray operations.
- It maintains some "context data" on a per-image basis, which the compiler may use to support proper implementation of coarray arguments, especially with respect to automatic deallocation of coarrays at an `end team` statement. This is accessed/set with the provided procedures `prif_get_context_handle` and `prif_set_context_handle`. PRIF does not interpret the contents of this context data in any way, and it is only accessible on the current image. The context data is a property of the allocated coarray object, and is thus shared between all handles and aliases that refer to the same coarray allocation (i.e. those created from a call to `prif_alias_create`).

#### 2.3.3.2 `prif_critical_type`

- a derived type provided by the PRIF implementation that will be opaque to the compiler that will be used for implementing `critical` blocks

## 2.4 Procedure descriptions

The PRIF API provides implementations of parallel Fortran features, as specified in Fortran 2023. For any given `prif_*` procedure that corresponds to a Fortran procedure or statement of similar name, the constraints and semantics associated with each argument to the `prif_` procedure match those of the analogous argument to the parallel Fortran feature, except where this document explicitly specifies otherwise. For any given `prif_*` procedure that corresponds to a Fortran procedure or statement of similar name, the constraints and semantics match those of the analogous parallel Fortran feature. Specifically, any required synchronization is performed by the PRIF implementation unless otherwise specified.

Where possible, optional arguments are used for optional parts or different forms of statements or procedures. In some cases the different forms or presence of certain options change the return type or rank, and in those cases a generic interface with different specific procedures is used.

### 2.4.1 Common arguments

- `team`

  - a value of type `prif_team_type` that identifies a team that the current image is a member of
  - shall not be present with `team_number` except in a call to `prif_form_team`
- `team_number`
  - a value of type `integer(c_intmax_t)` that identifies a sibling team or in a call to `prif_form_team`, which team to become a member of
  - shall not be present with `team` except in a call to `prif_form_team`
- **`image_num`, any argument identifying an image**
  - May identify the current image

#### 2.4.2 Integer and Pointer Arguments

There are several categories of arguments where the PRIF implementation will need pointers and/or integers. These fall broadly into the following categories.

1. `integer(c_intptr_t)`: Anything containing a pointer representation where the compiler might be expected to perform pointer arithmetic
2. `type(c_ptr)` and `type(c_funptr)`: Anything containing a pointer to an object/function where the compiler is expected only to pass it (back) to the PRIF implementation
3. `integer(c_size_t)`: Anything containing an object size, in units of bytes or elements, i.e. shape, element_size, etc.
4. `integer(c_ptrdiff_t)`: strides between elements for non-contiguous coarray accesses
5. `integer(c_int)`: Integer arguments corresponding to image index and stat arguments. It is expected that the most common arguments appearing in Fortran code will be of default integer, it is expected that this will correspond with that kind, and there is no reason to expect these arguments to have values that would not be representable in this kind.
6. `integer(c_intmax_t)`: Bounds, cobounds, indices, coindices, and any other argument to an intrinsic procedure that accepts or returns an arbitrary integer.

The compiler is responsible for generating values and temporary variables as necessary to pass arguments of the correct type/size, and perform conversions when needed.

##### 2.4.2.1 sync-stat-list

- **`stat`** : This argument is `intent(out)` representing the presence and type of any error that occurs. A value of zero, indicates no error occurred. It is of type `integer(c_int)`, to minimize the frequency that integer conversions will be needed. If a different kind of integer is used as the argument, it is the compiler's responsibility to use an intermediate variable as the argument to the PRIF implementation procedure and provide conversion to the actual argument.
- **`errmsg` or `errmsg_alloc`** : There are two optional arguments for this, one which is allocatable and one which is not. It is the compiler's responsibility to ensure the appropriate optional argument is passed. If no error occurs, the definition status of the actual argument is unchanged.

#### 2.4.3 Program startup and shutdown

For a program that uses parallel Fortran features, the compiler shall insert calls to `prif_init` and `prif_stop`. These procedures will initialize and terminate the parallel runtime. `prif_init` shall be called prior to any other calls to the PRIF implementation.

##### 2.4.3.1 `prif_init`

- **Description**: This procedure will initialize the parallel environment.
- **Procedure Interface**:

```
subroutine prif_init(exit_code)
  integer(c_int), intent(out) :: exit_code
end subroutine
```

- **Further argument descriptions**:
  - `exit_code`: a non-zero value indicates an error occurred during initialization.

#### 2.4.3.2 `prif_stop`

- **Description**: This procedure synchronizes all executing images, cleans up the parallel runtime environment, and terminates the program. Calls to this procedure do not return.
- **Procedure Interface**:

```
subroutine prif_stop(quiet, stop_code_int, stop_code_char)
  logical(c_bool), intent(in) :: quiet
  integer(c_int), intent(in), optional :: stop_code_int
  character(len=*), intent(in), optional :: stop_code_char
end subroutine
```

- **Further argument descriptions**: At most one of the arguments `stop_code_int` or `stop_code_char` shall be supplied.
  - `quiet`: if this argument has the value `.true.`, no output of signaling exceptions or stop code will be produced. Note that in the case the statement does not contain this optional part, the compiler should provide the value `.false.`.
  - `stop_code_int`: is used as the process exit code if it is provided. Otherwise, the process exit code is `0`.
  - `stop_code_char`: is written to the unit identified by the named constant `OUTPUT_UNIT` from the intrinsic module `ISO_FORTRAN_ENV` if provided.

#### 2.4.3.3 `prif_error_stop`

- **Description**: This procedure terminates all executing images Calls to this procedure do not return.
- **Procedure Interface**:

```
subroutine prif_error_stop(quiet, stop_code_int, stop_code_char)
  logical(c_bool), intent(in) :: quiet
  integer(c_int), intent(in), optional :: stop_code_int
  character(len=*), intent(in), optional :: stop_code_char
end subroutine
```

- **Further argument descriptions**: At most one of the arguments `stop_code_int` or `stop_code_char` shall be supplied.
  - `quiet`: if this argument has the value `.true.`, no output of signaling exceptions or stop code will be produced. Note that in the case the statement does not contain this optional part, the compiler should provide the value `.false.`.
  - `stop_code_int`: is used as the process exit code if it is provided. Otherwise, the process exit code is a non-zero value.
  - `stop_code_char`: is written to the unit identified by the named constant `ERROR_UNIT` from the intrinsic module `ISO_FORTRAN_ENV` if provided.

#### 2.4.3.4 `prif_fail_image`

- **Description**: causes the executing image to cease participating in program execution without initiating termination. Calls to this procedure do not return.
- **Procedure Interface**:

```
subroutine prif_fail_image()
end subroutine
```

### 2.4.4 Image Queries

#### 2.4.4.1 `prif_num_images`

- **Description**: Query the number of images in the specified or current team.
- **Procedure Interface**:

```
subroutine prif_num_images(team, team_number, image_count)
  type(prif_team_type), intent(in), optional :: team
  integer(c_intmax_t), intent(in), optional :: team_number
  integer(c_int), intent(out) :: image_count
end subroutine
```

- **Further argument descriptions**:
  - `team` **and** `team_number`: optional arguments that specify a team. They shall not both be present in the same call.

#### 2.4.4.2 `prif_this_image`

- **Description**: Determine the image index or cosubscripts with respect to a given coarray of the current image in a given team or the current team. team, or the cosubscripts
- **Procedure Interface**:

```
interface prif_this_image
  subroutine prif_this_image_no_coarray(team, image_index)
    type(prif_team_type), intent(in), optional :: team
    integer(c_int), intent(out) :: image_index
  end subroutine

  subroutine prif_this_image_with_coarray( &
      coarray_handle, team, cosubscripts)
    type(prif_coarray_handle), intent(in) :: coarray_handle
    type(prif_team_type), intent(in), optional :: team
    integer(c_intmax_t), intent(out) :: cosubscripts(:)
  end subroutine

  subroutine prif_this_image_with_dim( &
      coarray_handle, dim, team, cosubscript)
    type(prif_coarray_handle), intent(in) :: coarray_handle
    integer(c_int), intent(in) :: dim
    type(prif_team_type), intent(in), optional :: team
    integer(c_intmax_t), intent(out) :: cosubscript
  end subroutine
end interface
```

- **Further argument descriptions**:
  - `cosubscripts`: the cosubcripts that would identify the current image in the specified team when used as coindices for the specified coarray
  - `dim`: identify which of the elements from `cosubscripts` should be returned as the `cosubscript` value
  - `cosubscript`: the element identified by `dim` or the array `cosubscripts` that would have been returned without the `dim` argument present

#### 2.4.4.3 `prif_failed_images`

- **Description**: Determine the image indices of known failed images, if any.
- **Procedure Interface**:

```
subroutine prif_failed_images(team, failed_images)
  type(prif_team_type), intent(in), optional :: team
  integer(c_int), allocatable, intent(out) :: failed_images(:)
```

```
end subroutine
```

#### 2.4.4.4 `prif_stopped_images`

- **Description**: Determine the image indices of images known to have initiated normal termination, if any.
- **Procedure Interface**:

```
subroutine prif_stopped_images(team, stopped_images)
  type(prif_team_type), intent(in), optional :: team
  integer(c_int), allocatable, intent(out) :: stopped_images(:)
end subroutine
```

#### 2.4.4.5 `prif_image_status`

- **Description**: Determine the image execution state of an image
- **Procedure Interface**:

```
impure elemental subroutine prif_image_status(image, team, image_status)
  integer(c_int), intent(in) :: image
  type(prif_team_type), intent(in), optional :: team
  integer(c_int), intent(out) :: image_status
end subroutine
```

- **Further argument descriptions**:
  - `image`: the image index of the image in the given or current team for which to return the execution status
  - `team`: if provided, the team from which to identify the image
  - `image_status`: has the value `PRIF_STAT_FAILED_IMAGE` if the identified image has failed, `PRIF_STAT_STOPPED_IMAGE` if the identified image has initiated normal termination, or zero.

### 2.4.5 Coarrays

#### 2.4.5.1 Common arguments

- `coarray_handle`
  - Argument for many of the coarray access procedures
  - scalar of type `prif_coarray_handle`
  - is a handle for the established coarray
  - represents the distributed object of the coarray in the team in which it was established
- `coindices`
  - Argument for many of the coarray access procedures
  - 1d assumed-shape array of type `integer`
  - correspond to the coindices appearing in a coindexed object
- `value` or `local_buffer`
  - Argument for `put` and `get` operations
  - assumed-rank array of `type(*)` or `type(c_ptr)`
  - It is the value to be sent in a `put` operation, and is assigned the value retrieved in the case of a `get` operation
- `image_num`
  - identifies the image to be communicated with
  - is the image index in the initial team
  - may be the current image

**2.4.5.2 Allocation and deallocation** Calls to `prif_allocate` and `prif_deallocate` are collective operations, while other allocation/deallocation operations are not. Note that a call to `move_alloc` with coarray arguments is also a collective operation, as described in the section below.

**2.4.5.2.1 Static coarray allocation** The compiler is responsible to generate code that collectively runs `prif_allocate` once for each static coarray and initializes them where applicable.

#### 2.4.5.2.2 `prif_allocate`

- **Description**: This procedure allocates memory for a coarray. This call is collective over the current team. Calls to `prif_allocate` will be inserted by the compiler when there is an explicit coarray allocation or at the beginning of a program to allocate space for statically declared coarrays in the source code. The PRIF implementation will store the coshape information in order to internally track it during the lifetime of the coarray.
- **Procedure Interface**:

```
subroutine prif_allocate( &
    lcobounds, ucobounds, lbounds, ubounds, element_length, &
    final_func, coarray_handle, allocated_memory, &
    stat, errmsg, errmsg_alloc)
  integer(kind=c_intmax_t), intent(in) :: lcobounds(:), ucobounds(:)
  integer(kind=c_intmax_t), intent(in) :: lbounds(:), ubounds(:)
  integer(kind=c_size_t), intent(in) :: element_length
  type(c_funptr), intent(in) :: final_func
  type(prif_coarray_handle), intent(out) :: coarray_handle
  type(c_ptr), intent(out) :: allocated_memory
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - **`lcobounds and ucobounds`**: Shall be the lower and upper bounds of the codimensions of the coarray being allocated. Shall be 1d arrays with the same dimensions as each other. The cobounds shall be sufficient to have a unique index for every image in the current team. I.e. `product(coshape(coarray)) >= num_images`.
  - **`lbounds and ubounds`**: Shall be the the lower and upper bounds of the local portion of the array. Shall be 1d arrays with the same dimensions as each other.
  - **`element_length`**: size of a single element of the array in bytes
  - **`final_func`**: Shall be a function pointer to the final subroutine, if any, for derived types. It is the responsibility of the compiler to generate such a subroutine if necessary to clean up allocatable components, typically with calls to `prif_deallocate_non_symmetric`. It may also be necessary to modify the allocation status of the coarray variable, especially in the case that it was allocated through a dummy argument. Its interface should be equivalent to the following Fortran interface
    ```
    subroutine coarray_cleanup(handle, stat, errmsg) bind(C)
      type(prif_coarray_handle), intent(in) :: handle
      integer(c_int), intent(out) :: stat
      character(len=:), intent(out), allocatable :: errmsg
    end subroutine
    ```
    or to the following equivalent C prototype
    ```
    void coarray_cleanup(
        prif_handle_t* handle, int* stat, CFI_cdesc_t* errmsg)
    ```
    The coarray handle can then be interrogated to determine the memory address and size of the data in order to orchestrate calling any necessary final subroutines or deallocation of any allocatable components, or the context data to orchestrate modifying the allocation status of a local variable portion of the coarray. It will be invoked once on each image, upon deallocation of the coarray.
  - **`coarray_handle`**: Represents the distributed object of the coarray on the corresponding team. The handle is created by the PRIF implementation and the compiler uses it for subsequent coindexed-object references of the associated coarray and for deallocation of the associated coarray.

- `allocated_memory`: A pointer to the local block of allocated memory for the Fortran object. The compiler is responsible for associating the local Fortran object with this memory, and initializing it if necessary.

#### 2.4.5.2.3 `prif_allocate_non_symmetric`

- **Description**: This procedure is used to allocate components of coarray objects.
- **Procedure Interface**:

```
subroutine prif_allocate_non_symmetric( &
    size_in_bytes, allocated_memory, stat, errmsg, errmsg_alloc)
  integer(kind=c_size_t) :: size_in_bytes
  type(c_ptr), intent(out) :: allocated_memory
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `size_in_bytes`: The size, in bytes, of the object to be allocated.
  - `allocated_memory`: A pointer to the block of allocated memory for the Fortran object. The compiler is responsible for associating the Fortran object with this memory, and initializing it if necessary.

#### 2.4.5.2.4 `prif_deallocate`

- **Description**: This procedure releases memory previously allocated for all of the coarrays associated with the handles in `coarray_handles`. This means that any local objects associated with this memory become invalid. The compiler will insert calls to this procedure when exiting a local scope where implicit deallocation of a coarray is mandated by the standard and when a coarray is explicitly deallocated through a `deallocate-stmt` in the source code. This call is collective over the current team, and the provided list of handles must denote corresponding coarrays (in the same order on every image) that were allocated by the current team using `prif_allocate` and not yet deallocated. It will start with a synchronization over the current team, and then the final subroutine for each coarray (if any) will be called. A synchronization will also occur before control is returned from this procedure, after all deallocation has been completed.
- **Procedure Interface**:

```
subroutine prif_deallocate( &
    coarray_handles, stat, errmsg, errmsg_alloc)
  type(prif_coarray_handle), intent(in) :: coarray_handles(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Argument descriptions**:
  - `coarray_handles`: Is an array of all of the handles for the coarrays that shall be deallocated.

#### 2.4.5.2.5 `prif_deallocate_non_symmetric`

- **Description**: This procedure releases memory previously allocated by a call to `prif_allocate_non_symmetric`.
- **Procedure Interface**:

```
subroutine prif_deallocate_non_symmetric( &
    mem, stat, errmsg, errmsg_alloc)
  type(c_ptr), intent(in) :: mem
```

```
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `mem`: Pointer to the block of memory to be released.

#### 2.4.5.2.6 `prif_alias_create`

- **Description**: Create a new coarray handle for an existing coarray, such as in a `prif_change_team` or to pass to a coarray dummy argument (especially in the case that the cobounds are different)
- **Procedure Interface**:

```
subroutine prif_alias_create( &
    source_handle, alias_co_lbounds, alias_co_ubounds, alias_handle)
  type(prif_coarray_handle), intent(in) :: source_handle
  integer(c_intmax_t), intent(in) :: alias_co_lbounds(:)
  integer(c_intmax_t), intent(in) :: alias_co_ubounds(:)
  type(prif_coarray_handle), intent(out) :: alias_handle
end subroutine
```

- **Further argument descriptions**:
  - `source_handle`: a handle (which may itself be an alias) to the existing coarray for which an alias is to be created
  - `alias_co_lbounds and alias_co_ubounds`: the cobounds to be used for the new alias
  - `alias_handle`: a new alias to the existing coarray

#### 2.4.5.2.7 `prif_alias_destroy`

- **Description**: Delete an alias to a coarray
- **Procedure Interface**:

```
subroutine prif_alias_destroy(alias_handle)
  type(prif_coarray_handle), intent(in) :: alias_handle
end subroutine
```

- **Further argument descriptions**:
  - `alias_handle`: the alias to be destroyed

**2.4.5.2.8 `move_alloc`** This is not provided by PRIF, but should be easily implemented through manipulation of `prif_coarray_handle`s. Note that calls to `prif_set_context_data` will likely be required as part of the operation. Note that `move_alloc` with coarray arguments is an image control statement that requires synchronization, so the compiler should likely insert call(s) to `prif_sync_all` as part of the implementation.

### 2.4.5.3 Queries

#### 2.4.5.3.1 `prif_set_context_data`

- **Description**: This procedure stores a `c_ptr` associated with a coarray handle for future retrieval. A typical usage would be to store a reference to the actual variable whose allocation status must be changed in the case that the coarray is deallocated.
- **Procedure Interface**:

```
subroutine prif_set_context_data(coarray_handle, context_data)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  type(c_ptr), intent(in) :: context_data
end subroutine
```

#### 2.4.5.3.2 `prif_get_context_data`

- **Description**: This procedure returns the `c_ptr` provided in the most recent call to `prif_set_context_data` with the same coarray handle
- **Procedure Interface**:

```
subroutine prif_get_context_data(coarray_handle, context_data)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  type(c_ptr), intent(out) :: context_data
end subroutine
```

#### 2.4.5.3.3 `prif_base_pointer`

- **Description**: This procedure returns a C pointer value referencing the base of the coarray elements on a given image and may be used in conjunction with various communication operations. Pointer arithmetic operations may be performed with the value and the results provided as input to the `get/put_*raw` or atomic procedures (none of which are guaranteed to perform validity checks, e.g., to detect out-of-bounds access violations). It is not valid to dereference the produced pointer value or the result of any operations performed with it on any image except for the identified image.
- **Procedure Interface**:

```
subroutine prif_base_pointer( &
    coarray_handle, coindices, team, team_number, ptr)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_intmax_t), intent(in) :: coindices(:)
  type(prif_team_type), optional, intent(in) :: team
  integer(c_intmax_t), optional, intent(in) :: team_number
  integer(c_intptr_t), intent(out) :: ptr
end subroutine
```

#### 2.4.5.3.4 `prif_local_data_size`

- **Description**: This procedure returns the size of the coarray data associated with the current image. This will be equal to the following expression of the arguments provided to `prif_allocate` at the time that the coarray was allocated; `element_length * product(ubounds-lbounds+1)`
- **Procedure Interface**:

```
subroutine prif_local_data_size(coarray_handle, data_size)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(out) :: data_size
end subroutine
```

#### 2.4.5.3.5 `prif_lcobound`

- **Description**: returns the lower cobound(s) of the coarray referred to by the coarray_handle. It is the compiler's responsibility to convert to a different kind if the `kind` argument appears.
- **Procedure Interface**:

```
interface prif_lcobound
  subroutine prif_lcobound_with_dim(coarray_handle, dim, lcobound)
    type(prif_coarray_handle), intent(in) :: coarray_handle
    integer(c_int), intent(in) :: dim
    integer(c_intmax_t), intent(out):: lcobound
  end subroutine
  subroutine prif_lcobound_no_dim(coarray_handle, lcobounds)
    type(prif_coarray_handle), intent(in) :: coarray_handle
    integer(c_intmax_t), intent(out) :: lcobounds(:)
```

```
  end subroutine
end interface
```

- **Further argument descriptions**:
  - `dim`: which codimension of the coarray to report the lower cobound of
  - `lcobound`: the lower cobound of the given dimension
  - `lcobounds`: an array of the size of the corank of the coarray, returns the lower cobounds of the given coarray

#### 2.4.5.3.6 `prif_ucobound`

- **Description**: returns the upper cobound(s) of the coarray referred to by the coarray_handle. It is the compiler's responsibility to convert to a different kind if the `kind` argument appears.
- **Procedure Interface**:

```
interface prif_ucobound
  subroutine prif_ucobound_with_dim(coarray_handle, dim, ucobound)
    type(prif_coarray_handle), intent(in) :: coarray_handle
    integer(c_int), intent(in) :: dim
    integer(c_intmax_t), intent(out):: ucobound
  end subroutine
  subroutine prif_ucobound_no_dim(coarray_handle, ucobounds)
    type(prif_coarray_handle), intent(in) :: coarray_handle
    integer(c_intmax_t), intent(out) :: ucobounds(:)
  end subroutine
end interface
```

- **Further argument descriptions**:
  - `dim`: which codimension of the coarray to report the upper cobound of
  - `ucobound`: the upper cobound of the given dimension
  - `ucobounds`: an array of the size of the corank of the coarray, returns the upper cobounds of the given coarray

#### 2.4.5.3.7 `prif_coshape`

- **Description**:
- **Procedure Interface**:

```
subroutine prif_coshape(coarray_handle, sizes)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(out) :: sizes(:)
end subroutine
```

- **Further argument descriptions**:
  - `sizes`: an array of the size of the corank of the coarray, returns the difference between the upper and lower cobounds + 1

#### 2.4.5.3.8 `prif_image_index`

- **Description**: returns the index of the image identified by the coindices provided in the `sub` argument with the given coarray on the identified team or the current team if no team is identified
- **Procedure Interface**:

```
subroutine prif_image_index( &
    coarray_handle, sub, team, team_number, image_index)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_intmax_t), intent(in) :: sub(:)
  type(prif_team_type), intent(in), optional :: team
```

```
  integer(c_int), intent(in), optional :: team_number
  integer(c_int), intent(out) :: image_index
end subroutine
```

- **Further argument descriptions**:
  - `team and team_number`: optional arguments that specify a team. They shall not both be present in the same call.
  - `sub`: A list of integers that identify a specific image in the identified or current team when interpreted as coindices for the provided coarray.

**2.4.5.4 Access** Coarray accesses will maintain serial dependencies for the issuing image. Any data access ordering between images is defined only with respect to ordered segments. Note that for put operations, "local completion" means that the provided arguments are no longer needed (e.g. their memory can be freed once the procedure has returned).

#### 2.4.5.4.1 Common Arguments

- `notify_ptr`: optional pointer on the identified image to the notify variable that should be updated on completion of the put operation. The referenced variable shall be of type `prif_notify_type`. If this argument is not present, no notification is performed.

#### 2.4.5.4.2 `prif_put`

- **Description**: This procedure assigns to the elements of a coarray, when the elements to be assigned to are contiguous in linear memory on both sides. The compiler can use this to implement assignment to a `coindexed-object`. It need not call this procedure when the coarray reference is not a `coindexed-object`. This procedure blocks on local completion.
- **Procedure Interface**:

```
subroutine prif_put( &
    coarray_handle, coindices, value, first_element_addr, &
    team, team_number, notify_ptr, stat, errmsg, errmsg_alloc)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_intmax_t), intent(in) :: coindices(:)
  type(*), dimension(..), intent(in), contiguous :: value
  type(c_ptr), intent(in) :: first_element_addr
  type(prif_team_type), optional, intent(in) :: team
  integer(c_intmax_t), optional, intent(in) :: team_number
  integer(c_intptr_t), optional, intent(in) :: notify_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `first_element_addr`: The address of the local data in the coarray corresponding to the first element to be assigned to on the identified image

#### 2.4.5.4.3 `prif_put_raw`

- **Description**: Assign to `size` number of bytes on given image, starting at remote pointer, copying from local_buffer.
- **Procedure Interface**:

```
subroutine prif_put_raw( &
    image_num, local_buffer, remote_ptr, notify_ptr, size, &
    stat, errmsg, errmsg_alloc)
```

```
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: local_buffer
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_intptr_t), optional, intent(in) :: notify_ptr
  integer(c_size_t), intent(in) :: size
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `image_num`: identifies the image to be written to in the initial team
  - `local_buffer`: pointer to the contiguous local data which should be copied to the identified image.
  - `remote_ptr`: pointer to where on the identified image the data should be written
  - `size`: how much data is to be transferred in bytes

#### 2.4.5.4.4 `prif_put_raw_strided`

- **Description**: Assign to memory on given image, starting at remote pointer, copying from local_buffer, progressing through local_buffer in local_buffer_stride increments and through remote memory in remote_ptr_stride increments, transferring extent number of elements in each dimension.
- **Procedure Interface**:

```
subroutine prif_put_raw_strided( &
    image_num, local_buffer, remote_ptr, element_size, extent, &
    remote_ptr_stride, local_buffer_stride, notify_ptr, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: local_buffer
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_ptrdiff_t), intent(in) :: remote_ptr_stride(:)
  integer(c_ptrdiff_t), intent(in) :: local_buffer_stride(:)
  integer(c_intptr_t), optional, intent(in) :: notify_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `remote_ptr_stride`, `local_buffer_stride` and `extent` must each have size equal to the rank of the referenced coarray.
  - `image_num`: identifies the image to be written to in the initial team
  - `local_buffer`: pointer to the local data which should be copied to the identified image.
  - `remote_ptr`: pointer to where on the identified image the data should be written
  - `element_size`: The size of each element in bytes
  - `extent`: How many elements in each dimension should be transferred
  - `remote_ptr_stride`: The stride (in units of bytes) between elements in each dimension on the specified image. Each component of stride may independently be positive or negative, but (together with `extent`) must specify a region of distinct (non-overlapping) elements. The striding starts at the `remote_ptr`.
  - `local_buffer_stride`: The stride between elements in each dimension in the local buffer. Each component of stride may independently be positive or negative, but (together with `extent`) must specify a region of distinct (non-overlapping) elements. The striding starts at the `local_buffer`.

#### 2.4.5.4.5 `prif_get`

- **Description**: This procedure fetches data in a coarray from a specified image, when the elements are contiguous in linear memory on both sides. The compiler can use this to implement reads from a `coindexed-object`. It need not call this procedure when the coarray reference is not a `coindexed-object`. This procedure blocks until the requested data has been successfully assigned to the `value` argument.
- **Procedure Interface**:

```
subroutine prif_get( &
    coarray_handle, coindices, first_element_addr, value, team, team_number, &
    stat, errmsg, errmsg_alloc)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_intmax_t), intent(in) :: coindices(:)
  type(c_ptr), intent(in) :: first_element_addr
  type(*), dimension(..), intent(out), contiguous :: value
  type(prif_team_type), optional, intent(in) :: team
  integer(c_intmax_t), optional, intent(in) :: team_number
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `first_element_addr`: The address of the local data in the coarray corresponding to the first element to be fetched from the identified image

#### 2.4.5.4.6 `prif_get_raw`

- **Description**: Fetch `size` number of contiguous bytes from given image, starting at remote pointer, copying into local_buffer.
- **Procedure Interface**:

```
subroutine prif_get_raw( &
    image_num, local_buffer, remote_ptr, size, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: local_buffer
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_size_t), intent(in) :: size
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `image_num`: identifies the image from which the data should be fetched in the initial team
  - `local_buffer`: pointer to the contiguous local memory into which the retrieved data should be written
  - `remote_ptr`: pointer to where on the identified image the data begins
  - `size`: how much data is to be transferred in bytes

#### 2.4.5.4.7 `prif_get_raw_strided`

- **Description**: Copy from given image, starting at remote pointer, writing into local_buffer, progressing through local_buffer in local_buffer_stride increments and through remote memory in remote_ptr_stride increments, transferring extent number of elements in each dimension.

- **Procedure Interface**:

```
subroutine prif_get_raw_strided( &
    image_num, local_buffer, remote_ptr, element_size, extent, &
    remote_ptr_stride, local_buffer_stride, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: local_buffer
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_ptrdiff_t), intent(in) :: remote_ptr_stride(:)
  integer(c_ptrdiff_t), intent(in) :: local_buffer_stride(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `remote_ptr_stride`, `local_buffer_stride` and `extent` must each have size equal to the rank of the referenced coarray.
  - **`image_num`**: identifies the image from which the data should be fetched in the initial team
  - **`local_buffer`**: pointer to the local memory into which the retrieved data should be written
  - **`remote_ptr`**: pointer to where on the identified image the data begins
  - **`element_size`**: The size of each element in bytes
  - **`extent`**: How many elements in each dimension should be transferred
  - **`remote_ptr_stride`**: The stride (in units of bytes) between elements in each dimension on the specified image. Each component of stride may independently be positive or negative, but (together with `extent`) must specify a region of distinct (non-overlapping) elements. The striding starts at the `remote_ptr`.
  - **`local_buffer_stride`**: The stride between elements in each dimension in the local buffer. Each component of stride may independently be positive or negative, but (together with `extent`) must specify a region of distinct (non-overlapping) elements. The striding starts at the `local_buffer`.

### 2.4.6 Synchronization

#### 2.4.6.1 `prif_sync_memory`

- **Description**: Ends one segment and begins another, waiting on pending communication operations with other images.
- **Procedure Interface**:

```
subroutine prif_sync_memory(stat, errmsg, errmsg_alloc)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

#### 2.4.6.2 `prif_sync_all`

- **Description**: Performs a synchronization of all images in the current team.
- **Procedure Interface**:

```
subroutine prif_sync_all(stat, errmsg, errmsg_alloc)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
```

```
end subroutine
```

#### 2.4.6.3 `prif_sync_images`

- **Description**: Performs a synchronization with the listed images.
- **Procedure Interface**:

```
subroutine prif_sync_images(image_set, stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in), optional :: image_set(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `image_set`: The image inidices of the images in the current team with which to synchronize. Note, if a scalar appears, the compiler should pass its value as a size 1 array, and if an asterisk (`*`) appears, the compiler should not pass `image_set`.

#### 2.4.6.4 `prif_sync_team`

- **Description**: Performs a synchronization with the images of the identified team.
- **Procedure Interface**:

```
subroutine prif_sync_team(team, stat, errmsg, errmsg_alloc)
  type(prif_team_type), intent(in) :: team
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `team`: Identifies the team to synchronize.

#### 2.4.6.5 `prif_lock`

- **Description**: Waits until the identified lock variable is unlocked and then locks it if the `acquired_lock` argument is not present. Otherwise it sets the `acquired_lock` argument to `.false.` if the identified lock variable was locked, or locks the identified lock variable and sets the `acquired_lock` argument to `.true.`. Note that if the identified lock variable was already locked by the current image an error condition occurs.
- **Procedure Interface**:

```
subroutine prif_lock( &
    image_num, lock_var_ptr, acquired_lock, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: lock_var_ptr
  logical(c_bool), intent(out), optional :: acquired_lock
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `image_num`: the image index in the initial team for the lock variable to be locked
  - `lock_var_ptr`: a pointer to the base address of the lock variable to be locked on the identified image, typically obtained from a call to `prif_base_pointer`

- `acquired_lock`: if present is set to `.true.` if the lock was locked by the current image, or set to `.false.` otherwise

#### 2.4.6.6 `prif_unlock`

- **Description**: Unlocks the identified lock variable. Note that if the identified lock variable was not locked by the current image an error condition occurs.
- **Procedure Interface**:

```
subroutine prif_unlock( &
    image_num, lock_var_ptr, stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: lock_var_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `image_num`: the image index in the initial team for the lock variable to be unlocked
  - `lock_var_ptr`: a pointer to the base address of the lock variable to be unlocked on the identified image, typically obtained from a call to `prif_base_pointer`

#### 2.4.6.7 `prif_critical`

- **Description**: The compiler shall define a coarray, and establish (allocate) it in the initial team, that shall only be used to begin and end the critical block. An efficient implementation will likely define one for each critical block. The coarray shall be a scalar coarray of type `prif_critical_type` and the associated coarray handle shall be passed to this procedure. This procedure waits until any other image which has executed this procedure with a corresponding coarray handle has subsequently executed `prif_end_critical` with the same coarray handle an identical number of times.
- **Procedure Interface**:

```
subroutine prif_critical( &
    critical_coarray, stat, errmsg, errmsg_alloc)
  type(prif_coarray_handle), intent(in) :: critical_coarray
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `critical_coarray`: the handle for the `prif_critical_type` coarray associated with a given critical construct

#### 2.4.6.8 `prif_end_critical`

- **Description**: Completes execution of the critical construct associated with the provided coarray handle.
- **Procedure Interface**:

```
subroutine prif_end_critical(critical_coarray)
  type(prif_coarray_handle), intent(in) :: critical_coarray
end subroutine
```

- **Further argument descriptions**:
  - `critical_coarray`: the handle for the `prif_critical_type` coarray associated with a given critical construct

### 2.4.7 Events and Notifications

#### 2.4.7.1 `prif_event_post`

- **Description**: Atomically increment the count of the event variable by one.
- **Procedure Interface**:

```
subroutine prif_event_post( &
    image_num, event_var_ptr, stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: event_var_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `image_num`: the image index in the initial team for the event variable to be incremented
  - `event_var_ptr`: a pointer to the base address of the event variable to be incremented on the identified image, typically obtained from a call to `prif_base_pointer`

#### 2.4.7.2 `prif_event_wait`

- **Description**: Wait until the count of the provided event variable is greater than or equal to `until_count`, and then atomically decrement the count by that value. If `until_count` is not present it has the value 1.
- **Procedure Interface**:

```
subroutine prif_event_wait( &
    event_var_ptr, until_count, stat, errmsg, errmsg_alloc)
  integer(c_ptr), intent(in) :: event_var_ptr
  integer(c_intmax_t), intent(in), optional :: until_count
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `event_var_ptr`: a pointer to the event variable to be waited on
  - `until_count`: the count of the given event variable to be waited for. Has the value 1 if not provided.

#### 2.4.7.3 `prif_event_query`

- **Description**: Query the count of an event.
- **Procedure Interface**:

```
subroutine prif_event_query(event_var_ptr, count, stat)
  integer(c_ptr), intent(in) :: event_var_ptr
  integer(c_intmax_t), intent(out) :: count
  integer(c_int), intent(out), optional :: stat
end subroutine
```

- **Further argument descriptions**:
  - `event_var_ptr`: a pointer to the event variable to be queried
  - `count`: the current count of the given event variable.

#### 2.4.7.4 `prif_notify_wait`

- **Description**: Wait on notification of a put operation
- **Procedure Interface**:

```
subroutine prif_notify_wait( &
    notify_var_ptr, until_count, stat, errmsg, errmsg_alloc)
  integer(c_ptr), intent(in) :: notify_var_ptr
  integer(c_intmax_t), intent(in), optional :: until_count
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `notify_var_ptr`: a pointer to the notify variable to be waited on. The referenced variable shall be of type `prif_notify_type`.
  - `until_count`: the count of the given notify variable to be waited for. Has the value 1 if not provided.

### 2.4.8 Teams

Team creation forms a tree structure, where a given team may create multiple child teams. The initial team is created by the `prif_init` procedure. Each subsequently created team's parent team is then the current team. Team membership is thus strictly hierarchical, following a single path along the tree formed by team creation.

#### 2.4.8.1 `prif_form_team`

- **Description**: Create teams. Each image receives a team value denoting the newly created team containing all images in the current team which specify the same value for `team_number`.
- **Procedure Interface**:

```
subroutine prif_form_team( &
    team_number, team, new_index, stat, errmsg, errmsg_alloc)
  integer(c_intmax_t), intent(in) :: team_number
  type(prif_team_type), intent(out) :: team
  integer(c_int), intent(in), optional :: new_index
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

- **Further argument descriptions**:
  - `new_index`: the index that the current image will have in its new team

#### 2.4.8.2 `prif_get_team`

- **Description**: Get the team value for the current or an ancestor team. It returns the current team if `level` is not present or has the value `PRIF_CURRENT_TEAM`, the parent team if `level` is present with the value `PRIF_PARENT_TEAM`, or the initial team if `level` is present with the value `PRIF_INITIAL_TEAM`
- **Procedure Interface**:

```
subroutine prif_get_team(level, team)
  integer(c_int), intent(in), optional :: level
  type(prif_team_type), intent(out) :: team
end subroutine
```

- **Further argument descriptions**:
  - `level`: identify which team value to be returned

#### 2.4.8.3 `prif_team_number`

- **Description**: Return the `team_number` that was specified in the call to `prif_form_team` for the specified team, or -1 if the team is the initial team. If `team` is not present, the current team is used.
- **Procedure Interface**:

```
subroutine prif_team_number(team, team_number)
  type(prif_team_type), intent(in), optional :: team
  integer(c_intmax_t), intent(out) :: team_number
end subroutine
```

#### 2.4.8.4 `prif_change_team`

- **Description**: changes the current team to the specified team. For any associate names specified in the `CHANGE TEAM` statement the compiler should follow a call to this procedure with calls to `prif_alias_create` to create the alias coarray handle, and associate any non-coindexed references to the associate name within the `CHANGE TEAM` construct with the selector.
- **Procedure Interface**:

```
subroutine prif_change_team(team, stat, errmsg, errmsg_alloc)
  type(prif_team_type), intent(in) :: team
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

#### 2.4.8.5 `prif_end_team`

- **Description**: Changes the current team to the parent team. During the execution of `prif_end_team`, the PRIF implementation will deallocate any coarrays allocated during the change team construct. Prior to invoking `prif_end_team`, the compiler is responsible for invoking `prif_alias_destroy` for any `prif_coarray_handle` handles created as part of the `change team` statement.
- **Procedure Interface**:

```
subroutine prif_end_team(stat, errmsg, errmsg_alloc)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

### 2.4.9 Collectives

#### 2.4.9.1 Common arguments

- `a`
  - Argument for all the collective subroutines: `prif_co_broadcast`, `prif_co_max`, `prif_co_min`, `prif_co_reduce`, `prif_co_sum`,
  - may be any type for `co_broadcast` or `co_reduce`, any numeric for `co_sum`, and integer, real, or character for `co_min` or `co_max`
  - is always `intent(inout)`
  - for `co_max`, `co_min`, `co_reduce`, `co_sum` it is assigned the value computed by the collective operation, if no error conditions occurs and if `result_image` is absent, or the executing image is the one identified by `result_image`, otherwise `a` becomes undefined

  - for `co_broadcast`, the value of the argument on the `source_image` is assigned to the `a` argument on all other images
- **`source_image` or `result_image`**
  - These arguments are of type `integer(c_int)`, to minimize the frequency that integer conversions will be needed.

#### 2.4.9.2 `prif_co_broadcast`

- **Description**: Broadcast value to images
- **Procedure Interface**:

```
subroutine prif_co_broadcast( &
    a, source_image, stat, errmsg, errmsg_alloc)
  type(*), intent(inout), contiguous, target :: a(..)
  integer(c_int), intent(in) :: source_image
  integer(c_int), optional, intent(out) :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

#### 2.4.9.3 `prif_co_max`

- **Description**: Compute maximum value across images
- **Procedure Interface**:

```
subroutine prif_co_max( &
    a, result_image, stat, errmsg, errmsg_alloc)
  type(*), intent(inout), contiguous, target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

#### 2.4.9.4 `prif_co_min`

- **Description**: Compute minimum value across images
- **Procedure Interface**:

```
subroutine prif_co_min( &
    a, result_image, stat, errmsg, errmsg_alloc)
  type(*), intent(inout), contiguous, target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

#### 2.4.9.5 `prif_co_reduce`

- **Description**: Generalized reduction across images
- **Procedure Interface**:

```
subroutine prif_co_reduce( &
    a, operation, result_image, stat, errmsg, errmsg_alloc)
  type(*), intent(inout), contiguous, target :: a(..)
  type(c_funptr), value :: operation
```

```
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

#### 2.4.9.6 `prif_co_sum`

- **Description**: Compute sum across images
- **Procedure Interface**:

```
subroutine prif_co_sum( &
    a, result_image, stat, errmsg, errmsg_alloc)
  type(*), intent(inout), contiguous, target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

### 2.4.10 Atomic Memory Operation

All atomic operations are blocking operations.

#### 2.4.10.1 Common arguments

- `atom_remote_ptr`
  - Argument for all of the atomic subroutines
  - is type `integer(c_intptr_t)`
  - is the location of the atomic variable on the identified image to be operated on
  - it is the responsibility of the compiler to perform the necessary operations on the coarray or coindexed actual argument to get the relevant remote pointer
- `image_num`
  - identifies the image on which the atomic operation is to be performed
  - is the image index in the initial team

**2.4.10.2 Non-fetching Atomic Operations** Perform specified operation on a variable in a coarray atomically.

##### 2.4.10.2.1 Common argument

- `value`: value to perform the operation with

##### 2.4.10.2.2 `prif_atomic_add`, Addition

```
subroutine prif_atomic_add(atom_remote_ptr, image_num, value, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

##### 2.4.10.2.3 `prif_atomic_and`, Bitwise And

```
subroutine prif_atomic_and(atom_remote_ptr, image_num, value, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
```

```
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

#### 2.4.10.2.4 `prif_atomic_or`, Bitwise Or

```
subroutine prif_atomic_or(atom_remote_ptr, image_num, value, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

#### 2.4.10.2.5 `prif_atomic_xor`, Bitwise Xor

```
subroutine prif_atomic_xor(atom_remote_ptr, image_num, value, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

**2.4.10.3 Atomic Fetch Operations** Perform specified operation on a variable in a coarray atomically and save its original value.

#### 2.4.10.3.1 Common arguments

- `value`: value to perform the operation with
- `old`: is set to the initial value of the atomic variable

#### 2.4.10.3.2 `prif_atomic_fetch_add`, Addition

```
subroutine prif_atomic_fetch_add( &
    atom_remote_ptr, image_num, value, old, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(atomic_int_kind), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

#### 2.4.10.3.3 `prif_atomic_fetch_and`, Bitwise And

```
subroutine prif_atomic_fetch_and( &
    atom_remote_ptr, image_num, value, old, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(atomic_int_kind), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

#### 2.4.10.3.4 `prif_atomic_fetch_or`, Bitwise Or

```
subroutine prif_atomic_fetch_or( &
    atom_remote_ptr, image_num, value, old, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(atomic_int_kind), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

#### 2.4.10.3.5 `prif_atomic_fetch_xor`, Bitwise Xor

```
subroutine prif_atomic_fetch_xor( &
    atom_remote_ptr, image_num, value, old, stat)
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(c_int), intent(in) :: image_num
  integer(atomic_int_kind), intent(in) :: value
  integer(atomic_int_kind), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

### 2.4.10.4 Atomic Access

Atomically set or retrieve the value of an atomic variable in a coarray.

#### 2.4.10.4.1 Common argument

- `value`: value to which the variable shall be set, or retrieved from the variable

#### 2.4.10.4.2 `prif_atomic_define`, set variable's value

```
interface prif_atomic_define
  subroutine prif_atomic_define_int( &
      atom_remote_ptr, image_num, value, stat)
    integer(c_intptr_t), intent(in) :: atom_remote_ptr
    integer(c_int), intent(in) :: image_num
    integer(atomic_int_kind), intent(in) :: value
    integer(c_int), intent(out), optional :: stat
  end subroutine

  subroutine prif_atomic_define_logical( &
      atom_remote_ptr, image_num, value, stat)
    integer(c_intptr_t), intent(in) :: atom_remote_ptr
    integer(c_int), intent(in) :: image_num
    logical(atomic_logical_kind), intent(in) :: value
    integer(c_int), intent(out), optional :: stat
  end subroutine
end interface
```

#### 2.4.10.4.3 `prif_atomic_ref`, retrieve variable's value

```
interface prif_atomic_ref
  subroutine prif_atomic_ref_int( &
      value, atom_remote_ptr, image_num, stat)
    integer(atomic_int_kind), intent(out) :: value
    integer(c_intptr_t), intent(in) :: atom_remote_ptr
    integer(c_int), intent(in) :: image_num
```

```
    integer(c_int), intent(out), optional :: stat
  end subroutine

  subroutine prif_atomic_ref_logical( &
      value, atom_remote_ptr, image_num, stat)
    logical(atomic_logical_kind), intent(out) :: value
    integer(c_intptr_t), intent(in) :: atom_remote_ptr
    integer(c_int), intent(in) :: image_num
    integer(c_int), intent(out), optional :: stat
  end subroutine
end interface
```

**2.4.10.4.4 `prif_atomic_cas`, Compare and Swap** If the value of the atomic variable is equal to the value of the `compare` argument, set it to the value of the `new` argument. The `old` argument is set to the initial value of the atomic variable.

```
interface prif_atomic_cas
  subroutine prif_atomic_cas_int( &
      atom_remote_ptr, image_num, old, compare, new, stat)
    integer(c_intptr_t), intent(in) :: atom_remote_ptr
    integer(c_int), intent(in) :: image_num
    integer(atomic_int_kind), intent(out) :: old
    integer(atomic_int_kind), intent(in) :: compare
    integer(atomic_int_kind), intent(in) :: new
    integer(c_int), intent(out), optional :: stat
  end subroutine

  subroutine prif_atomic_cas_logical( &
      atom_remote_ptr, image_num, old, compare, new, stat)
    integer(c_intptr_t), intent(in) :: atom_remote_ptr
    integer(c_int), intent(in) :: image_num
    logical(atomic_logical_kind), intent(out) :: old
    logical(atomic_logical_kind), intent(in) :: compare
    logical(atomic_logical_kind), intent(in) :: new
    integer(c_int), intent(out), optional :: stat
  end subroutine
end interface
```

- **Further argument descriptions**:
  - `old`: is set to the initial value of the atomic variable
  - `compare`: the value with which to compare the atomic variable
  - `new`: the value to set the atomic variable too if it is initially equal to the `compare` argument

# 3 Future Work

At present all communication operations are semantically blocking on at least local completion. We acknowledge that this prohibits certain types of static optimization, namely the explicit overlap of communication with computation. In the future we intend to develop split-phased/asynchronous versions of various communication operations to enable more opportunities for static optimization of communication.

# 4 Acknowledgments

This research is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231

# 5 Copyright



# 6 Legal Disclaimer