Parallel Runtime Interface for Fortran (PRIF) Specification, Revision 0.3

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

Lawrence Berkeley National Laboratory Technical Report (LBNL-2001590)
doi:10.25344/S4501W

May 3, 2024

Abstract

This document specifies 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 document is still a draft a may continue to evolve.
Feedback and questions should be directed to: fortran@lbl.gov

Contents

1 Change Log

1.1 Revision 0.1

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

1.2 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.

1.3 Revision 0.3 (May 2024)

- `prif_(de)allocate` are renamed to `prif_(de)allocate_coarray`
- `prif_(de)allocate_non_symmetric` are renamed to `prif_(de)allocate`
- `prif_local_data_size` renamed to `prif_size_bytes` and add a client note about the procedure
- Update interface to `prif_base_pointer` by replacing three arguments, `coindices`, `team`, and `team_number`, with one argument `image_num`. Update the semantics of `prif_base_pointer`, as it is no longer responsible for resolving the coindices and team information into a number that represents the image on the initial team before returning the address. That is now expected to occur before the `prif_base_pointer` call and passed into the `image_num` argument.
- Add target attribute on `coarray_handles` argument to `prif_deallocate_coarray`
- Add pointer attribute on `handle` argument to `coarray_cleanup` callback for `prif_allocate_coarray`
- Add target attribute on `value` argument to `prif_put` and `prif_get`
- Add new PRIF-specific constant `PRIF_STAT_OUT_OF_MEMORY`
- Clarify that remote pointers passed to various procedures must reference storage allocated using `prif_allocate_coarray` or `prif_allocate`
- Clarify description of the `allocated_memory` argument for the procedures `prif_allocate_coarray` and `prif_allocate`
- Clarify descriptions of `event_var_ptr`, `lock_var_ptr`, and `notify_ptr`
- Clarify descriptions for `prif_stop`, `prif_put`, `prif_get`, intrinsic derived types, sections about `MOVE_ALLOC` and coarray accesses
- Replace the phrase "local completion" with the phrase "source completion", and add the new phrase to the glossary
- Clarify that `prif_stop` should be used to initiate normal termination
- Describe the `operation` argument to `prif_co_reduce`
- Rename and clarify the cobounds arguments to `prif_alias_create`
- Clarify the descriptions of `source_image`/`result_image` arguments to collective calls
- Clarify completion semantics for atomic operations
- Rename `coindices` argument names to `cosubscripts` to more closely correspond with the terms used in the Fortran standard
- Rename `local_buffer` and `local_buffer_stride` arg names to `current_image_buffer` and `current_image_buffer_stride`
- Update `coindexed-object` references to *coindexed-named-object* to match the term change in the most recent Fortran 2023 standard
- Convert several explanatory sections to "Notes"
- Add implementation note about the PRIF API being defined in Fortran
- Add section "How to read the PRIF specification"
- Add section "Glossary"
- Improve description of the `final_func` arg to `prif_allocate_coarray` and move some of previous description to a client note.

2 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 supporting 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` to `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.

3 Proposed Solution

This specification 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 specify 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.

3.1 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.

Implementations of PRIF 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`.

3.2 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	PRIF implementation
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`	X	
Provide `prif_critical_type` coarrays for `CRITICAL`	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	
Intrinsics related to parallelism, eg. `NUM_IMAGES`, `COSHAPE`, `IMAGE_INDEX`		X
Allocate and deallocate a coarray		X
Reference a *coindexed-named-object*		X
Team statements/constructs: `FORM TEAM`, `CHANGE TEAM`, `END TEAM`		X
Team stack abstraction		X
Track coarrays for implicit deallocation at `END TEAM`		X
Atomic subroutines, e.g. `ATOMIC_FETCH_ADD`		X
Collective subroutines, e.g. `CO_BROADCAST`, `CO_SUM`		X
Synchronization statements, e.g. `SYNC ALL`, `SYNC TEAM`		X
Events: `EVENT POST`, `EVENT WAIT`		X
Locks: `LOCK`, `UNLOCK`		X
`CRITICAL` construct		X
`NOTIFY WAIT` statement		X

NOTE: Caffeine - LBNL's Implementation of the Parallel Runtime Interface for Fortran

Implementations for much of the Parallel Runtime Interface for Fortran exist in Caffeine, a parallel runtime library supporting coarray Fortran compilers. Caffeine will continue to be developed in order to fully implement PRIF. Caffeine targets the GASNet-EX exascale networking middleware, however PRIF is deliberately agnostic to details of the communication substrate. As such it should be possible to develop PRIF implementations targeting other substrates including the Message Passing Interface (MPI).

3.3 How to read the PRIF specification

The following types and procedures align with corresponding types and procedures from the Fortran standard. In many cases, the correspondence is clear from the identifiers. For example, the PRIF procedure `prif_num_images` corresponds to the intrinsic function `NUM_IMAGES` that is defined in the Fortran standard. In other cases, the correspondence may be less clear and is stated explicitly.

In order to avoid redundancy, some details are not included below as the corresponding descriptions in the Fortran standard contain the detailed descriptions of what is required by the language. For example, this document references the term *coindexed-named-object* multiple times, but does not define it since it is part of the language and the Fortran standard defines it. As such, in order to fully understand the PRIF specification, it is critical to read and reference the Fortran standard alongside it. Additionally, the descriptions in the PRIF specification use similar language to the language used in the Fortran standard, such as terms like 'shall'." Where PRIF uses terms not defined in the standard, their definitions may be found in the `Glossary`.

4 PRIF Types and Constants

4.1 Fortran Intrinsic Derived Types

These types will be defined by the PRIF implementation. The compiler will use these PRIF-provided implementation definitions for the corresponding 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 PRIF implementation.

4.1.1 `prif_team_type`

- implementation for `TEAM_TYPE` from `ISO_FORTRAN_ENV`

4.1.2 `prif_event_type`

- implementation for `EVENT_TYPE` from `ISO_FORTRAN_ENV`

4.1.3 `prif_lock_type`

- implementation for `LOCK_TYPE` from `ISO_FORTRAN_ENV`

4.1.4 `prif_notify_type`

- implementation for `NOTIFY_TYPE` from `ISO_FORTRAN_ENV`

4.2 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.

4.2.1 `PRIF_ATOMIC_INT_KIND`

This shall be set to an implementation-defined value from the compiler-provided `INTEGER_KINDS` array.

4.2.2 `PRIF_ATOMIC_LOGICAL_KIND`

This shall be set to an implementation-defined value from the compiler-provided `LOGICAL_KINDS` array.

4.2.3 `PRIF_CURRENT_TEAM`

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

4.2.4 `PRIF_INITIAL_TEAM`

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

4.2.5 `PRIF_PARENT_TEAM`

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

4.2.6 `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. It shall be distinct from all other stat constants defined by this specification.

4.2.7 `PRIF_STAT_LOCKED`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat constants defined by this specification.

4.2.8 `PRIF_STAT_LOCKED_OTHER_IMAGE`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat constants defined by this specification.

4.2.9 `PRIF_STAT_STOPPED_IMAGE`

This shall be a positive value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat constants defined by this specification.

4.2.10 `PRIF_STAT_UNLOCKED`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat constants defined by this specification.

4.2.11 `PRIF_STAT_UNLOCKED_FAILED_IMAGE`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat constants defined by this specification.

4.3 PRIF-Specific Constants

This constant is not defined by the Fortran standard.

4.3.1 `PRIF_STAT_OUT_OF_MEMORY`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat constants defined by this specification. It shall indicate a low-memory condition and may be returned by `prif_allocate_coarray` or `prif_allocate`.

4.4 PRIF-Specific Types

These derived types are defined by the PRIF implementation and the contents are opaque to the compiler. They don't correspond directly to types mandated by the Fortran specification, but rather are helper types used in PRIF to provide the parallel Fortran features.

4.4.1 `prif_coarray_handle`

- a derived type provided by the PRIF implementation whose contents are opaque to the compiler. It represents a reference to a coarray descriptor and is passed back and forth across PRIF for coarray operations.
- Each coarray descriptor 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 `MOVE_ALLOC` operations on allocatable coarrays. This is accessed/set with the 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`).

4.4.2 `prif_critical_type`

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

5 PRIF Procedures

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. In particular, any required synchronization is performed by the PRIF implementation unless otherwise specified.

IMPLEMENTATION NOTE:

The PRIF API is defined as a set of Fortran language procedures and supporting types, and as such an implementation of PRIF cannot be expressed solely in C/C++. However C/C++ can be used to implement portions of the PRIF procedures via calls to BIND(C) procedures.

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.

5.1 Common Arguments

There are multiple Common Arguments sections throughout the specification that outline details of the arguments that are common for the following sections of procedure interfaces.

5.1.1 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 integer arguments appearing in Fortran code will be of default integer kind, 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, cosubscripts, 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.

5.1.2 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 join
  - shall not be present with `team` except in a call to `prif_form_team`
- `image_num`
  - an argument identifying the image to be communicated with
  - is permitted to identify the current image
  - the image index is always relative to the initial team, unless otherwise specified

5.1.3 `stat` and `errmsg` Arguments

- `stat` : This argument is `intent(out)` and represents 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 `intent(out)` 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, and at most one shall be provided in any given call. If no error occurs, the definition status of the actual argument is unchanged.

5.2 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. `prif_stop` shall be called to initiate normal termination if the program reaches normal termination at the end of the main program.

5.2.1 `prif_init`

Description: This procedure will initialize the parallel environment.

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

Further argument descriptions:

- `stat`: a non-zero value indicates an error occurred during initialization.

5.2.2 `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. This procedure supports both normal termination at the end of a program, as well as any `STOP` statements from the user source code.

```
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. If a `STOP` 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.

5.2.3 `prif_error_stop`

Description: This procedure terminates all executing images. Calls to this procedure do not return.

```
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. If an `ERROR STOP` 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.

5.2.4 `prif_fail_image`

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

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

5.3 Image Queries

5.3.1 `prif_num_images`

Description: Query the number of images in the specified or current team.

```
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.

5.3.2 `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.

```
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:

- **`coarray_handle`**: is described in the Common Arguments section under Storage Management
- **`cosubscripts`**: the cosubscripts that would identify the current image in the specified team when used as cosubscripts 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` of the array `cosubscripts` that would have been returned without the `dim` argument present

5.3.3 `prif_failed_images`

Description: Determine the image indices of any images known to have failed.

```
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
```

5.3.4 `prif_stopped_images`

Description: Determine the image indices of any images known to have initiated normal termination.

```
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
```

5.3.5 `prif_image_status`

Description: Determine the image execution state of an image

```
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`: defined to the value `PRIF_STAT_FAILED_IMAGE` if the identified image has failed, `PRIF_STAT_STOPPED_IMAGE` if the identified image has initiated normal termination, otherwise zero.

5.4 Storage Management

5.4.1 Common Arguments

- `coarray_handle`
  - Argument for many of the coarray access procedures
  - scalar of type `prif_coarray_handle`
  - is a handle for the descriptor of an established coarray
- `cosubscripts`
  - Argument for many of the coarray access procedures
  - 1d assumed-shape array of type `integer(c_intmax_t)`
  - correspond to the cosubscripts appearing in a *coindexed-named-object* reference
- `value` or `current_image_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`
  - an argument identifying the image to be communicated with
  - is permitted to identify the current image
  - the image index is always relative to the initial team, unless otherwise specified

5.4.2 `prif_allocate_coarray`

Description: This procedure allocates memory for a coarray. This call is collective over the current team. Calls to `prif_allocate_coarray` 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.

```
subroutine prif_allocate_coarray( &
    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(ucobounds - lcobounds + 1) >= num_images()`.
- **`lbounds and ubounds`**: Shall be the the lower and upper bounds of the current image's 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 the C address of a procedure that is interoperable, or `C_NULL_FUNPTR`. If not null, this procedure will be invoked by the PRIF implementation once by each image at deallocation of this coarray, before the storage is released. The procedure's interface shall be equivalent to the following Fortran interface

  ```
  subroutine coarray_cleanup(handle, stat, errmsg) bind(C)
    type(prif_coarray_handle), pointer, 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(
      CFI_cdesc_t* handle, int* stat, CFI_cdesc_t* errmsg)
  ```

- **`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-named-object* references of the associated coarray and for deallocation of the associated coarray.
- **`allocated_memory`**: A pointer to the block of allocated but uninitialized memory that provides the storage for the current image's coarray. The compiler is responsible for associating the Fortran-level coarray object with this storage, and initializing the storage if necessary. The returned pointer value

may differ across images in the team. `prif_base_pointer` should be used to locate corresponding coarrays on other images.

CLIENT NOTE:

`final_func` is used by the compiler to support various clean-up operations at coarray deallocation, whether it happens explicitly (i.e. via `prif_deallocate_coarray`) or implicitly (e.g. via `prif_end_team`). First, `final_func` may be used to support the user-defined final subroutine for derived types. Second, it may be necessary for the compiler to generate such a subroutine to clean up allocatable components, typically with calls to `prif_deallocate`. Third, it may also be necessary to modify the allocation status of an allocatable coarray variable, especially in the case that it was allocated through a dummy argument.
The coarray handle can be interrogated by the procedure callback using PRIF queries 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. The `pointer` attribute for the `handle` argument is to permit `prif_coarray_handle` definitions which are not C interoperable.

5.4.3 `prif_allocate`

Description: This procedure is used to non-collectively allocate remotely accessible storage, such as needed for an allocatable component of a coarray.

```
subroutine prif_allocate( &
    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 but uninitialized memory that provides the requested storage. The compiler is responsible for associating the Fortran object with this storage, and initializing the storage if necessary.

5.4.4 `prif_deallocate_coarray`

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` statement. 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_coarray` and not yet deallocated. The implementation starts 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.

```
subroutine prif_deallocate_coarray( &
    coarray_handles, stat, errmsg, errmsg_alloc)
  type(prif_coarray_handle), target, 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
```

Further argument descriptions:

- `coarray_handles`: Is an array of all of the handles for the coarrays that shall be deallocated. Note that the target attribute is not required for the actual argument to this procedure. It is only to allow the implementation to call the `final_func` procedures with each handle.

5.4.5 `prif_deallocate`

Description: This non-collective procedure releases memory previously allocated by a call to `prif_allocate`.

```
subroutine prif_deallocate( &
    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.

CLIENT NOTE:

Calls to `prif_allocate_coarray` and `prif_deallocate_coarray` are collective operations, while calls to `prif_allocate` and `prif_deallocate` are not. Note that a call to `MOVE_ALLOC` with coarray arguments is also a collective operation, as described in the section below.

CLIENT NOTE:

The compiler is responsible to generate code that collectively runs `prif_allocate_coarray` once for each static coarray and initializes them where applicable.

5.4.6 `prif_alias_create`

Description: Create a new coarray handle for an existing coarray, such as part of `CHANGE TEAM` after `prif_change_team`, or to pass to a coarray dummy argument (especially in the case that the cobounds are different)

```
subroutine prif_alias_create( &
    source_handle, alias_lcobounds, alias_ucobounds, alias_handle)
  type(prif_coarray_handle), intent(in) :: source_handle
  integer(c_intmax_t), intent(in) :: alias_lcobounds(:)
  integer(c_intmax_t), intent(in) :: alias_ucobounds(:)
  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_lcobounds and alias_ucobounds`: the cobounds to be used for the new alias. Both arguments must have the same size, but it need not match the corank associated with `source_handle`
- `alias_handle`: a new alias to the existing coarray

5.4.7 `prif_alias_destroy`

Description: Delete an alias to a coarray. Does not deallocate the original coarray.

```
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

5.4.8 `MOVE_ALLOC`

This is not provided by PRIF because it depends on unspecified details of the compiler's `allocatable` attribute. It is the compiler's responsibility to implement `MOVE_ALLOC` using PRIF-provided operations. For example, according to the Fortran standard, `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.

CLIENT NOTE:

It is envisioned that the use of `prif_set_context_data` and `prif_get_context_data` will allow for an efficient implementation of `MOVE_ALLOC` that maintains tracking of allocation status

5.5 Coarray Queries

5.5.1 `prif_set_context_data`

Description: This procedure stores a `c_ptr` associated with a coarray 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.

```
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
```

5.5.2 `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 (possibly via an alias coarray handle).

```
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
```

5.5.3 `prif_base_pointer`

Description: This procedure returns a C pointer value referencing the base of the corresponding 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. If the `image_num` argument is zero, then `coarray_handle` is ignored and `ptr` becomes defined with the value zero. It is an error to pass a number less than `0` or greater than the number of images to the `image_num` argument, in which case `ptr` becomes undefined.

```
subroutine prif_base_pointer(coarray_handle, image_num, ptr)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(out) :: ptr
end subroutine
```

Further argument descriptions:

- `image_num`: identifies the image number in the initial team on which the address is being requested
- `ptr`: returns a pointer to the beginning of the data elements for the corresponding coarray on the identified image

5.5.4 `prif_size_bytes`

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

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

CLIENT NOTE:

`prif_size_bytes` can be used to calculate the number of elements in an array coarray given only the handle and element size

5.5.5 `prif_lcobound`

Description: returns the lower cobound(s) associated with a coarray handle. It is the compiler's responsibility to convert to a different kind if the `kind` argument to `LCOBOUND` appears.

```
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 handle, returns the lower cobounds of the given coarray handle

5.5.6 `prif_ucobound`

Description: returns the upper cobound(s) associated with a coarray handle. It is the compiler's responsibility to convert to a different kind if the `kind` argument to `UCOBOUND` appears.

```
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 handle, returns the upper cobounds of the given coarray handle

5.5.7 `prif_coshape`

Description: returns the sizes of codimensions of a coarray

```
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 handle, returns the difference between the upper and lower cobounds + 1

5.5.8 `prif_image_index`

Description: returns the index of the image, on the identified team or the current team if no team is provided, identified by the cosubscripts provided in the `sub` argument with the given coarray handle

```
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 cosubscripts for the provided coarray handle.

5.6 Coarray Access

The memory consistency semantics of coarray accesses follow those defined by the Image Execution Control section of the Fortran standard. In particular, 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, "source completion" means that the provided source locations are no longer needed (e.g. their memory can be freed once the procedure has returned).

5.6.1 Common Arguments

- **`notify_ptr`**: 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`, and the storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`. If this optional argument is omitted, then no notification is performed.
- **`remote_ptr`**: pointer to where on the identified image the data begins. The referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.

5.6.2 `prif_put`

Description: This procedure assigns to the elements of a coarray, when the elements to be assigned are contiguous in linear memory on both sides. The compiler can use this to implement assignment to a *coindexed-named-object*. It need not call this procedure when the coarray reference is not a *coindexed-named-object*. This procedure blocks on source completion. This procedure corresponds to a contiguous coarray reference on the left hand side of an *assignment-stmt*.

```
subroutine prif_put( &
    coarray_handle, cosubscripts, 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) :: cosubscripts(:)
  type(*), intent(in), contiguous, target :: 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 current image's data in the coarray corresponding to the first element to be assigned to on the identified image

5.6.3 `prif_put_raw`

Description: Assign to `size` number of contiguous bytes on given image, starting at `remote_ptr`, copying from `current_image_buffer`.

```
subroutine prif_put_raw( &
    image_num, current_image_buffer, remote_ptr, notify_ptr, size, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: current_image_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
- **`current_image_buffer`**: pointer to the contiguous data which should be copied to the identified image.
- **`size`**: how much data is to be transferred in bytes

5.6.4 `prif_put_raw_strided`

Description: Assign to memory on given image, starting at `remote_ptr`, copying from `current_image_buffer`, progressing through `current_image_buffer` in `current_image_buffer_stride` increments and through remote memory in `remote_ptr_stride` increments, transferring `extent` number of elements in each dimension.

```
subroutine prif_put_raw_strided( &
    image_num, current_image_buffer, remote_ptr, element_size, extent, &
    remote_ptr_stride, current_image_buffer_stride, notify_ptr, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: current_image_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) :: current_image_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`, `current_image_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
- **`current_image_buffer`**: pointer to the data which should be copied to the identified image.
- **`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`.
- `current_image_buffer_stride`: The stride (in units of bytes) between elements in each dimension in the current image 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 `current_image_buffer`.

5.6.5 `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-named-object*. It need not call this procedure when the coarray reference is not a *coindexed-named-object*. This procedure blocks until the requested data has been successfully assigned to the `value` argument. This procedure corresponds to any *coindexed-named-object* reference that reads contiguous coarray data.

```
subroutine prif_get( &
    coarray_handle, cosubscripts, 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) :: cosubscripts(:)
  type(c_ptr), intent(in) :: first_element_addr
  type(*), intent(inout), contiguous, target :: 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 data in the coarray corresponding to the first element to be fetched from the identified image

5.6.6 `prif_get_raw`

Description: Fetch `size` number of contiguous bytes from given image, starting at `remote_ptr`, copying into `current_image_buffer`.

```
subroutine prif_get_raw( &
    image_num, current_image_buffer, remote_ptr, size, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: current_image_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
- `current_image_buffer`: pointer to the contiguous memory into which the retrieved data should be written
- `size`: how much data is to be transferred in bytes

5.6.7 `prif_get_raw_strided`

Description: Copy from given image, starting at `remote_ptr`, writing into `current_image_buffer`, progressing through `current_image_buffer` in `current_image_buffer_stride` increments and through remote memory in `remote_ptr_stride` increments, transferring `extent` number of elements in each dimension.

```
subroutine prif_get_raw_strided( &
    image_num, current_image_buffer, remote_ptr, element_size, extent, &
    remote_ptr_stride, current_image_buffer_stride, &
    stat, errmsg, errmsg_alloc)
  integer(c_int), intent(in) :: image_num
  type(c_ptr), intent(in) :: current_image_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) :: current_image_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`, `current_image_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
- `current_image_buffer`: pointer to the memory into which the retrieved 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`.
- `current_image_buffer_stride`: The stride (in units of bytes) between elements in each dimension in the current image 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 `current_image_buffer`.

5.7 Synchronization

5.7.1 `prif_sync_memory`

Description: Ends one segment and begins another, waiting on any pending communication operations with other images.

```
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
```

5.7.2 `prif_sync_all`

Description: Performs a synchronization of all images in the current team.

```
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
```

5.7.3 `prif_sync_images`

Description: Performs a synchronization with the listed images.

```
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. Given a scalar argument to SYNC IMAGES, the compiler should pass its value in an array of size 1. Given an asterisk (`*`) argument to SYNC IMAGES, the compiler should omit the `image_set` argument.

5.7.4 `prif_sync_team`

Description: Performs a synchronization with the images of the identified team.

```
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.

5.7.5 `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.`. If the identified lock variable was already locked by the current image, then an error condition occurs.

```
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`. The referenced variable shall be of type `prif_lock_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `acquired_lock`: if present is set to `.true.` if the lock was locked by the current image, or set to `.false.` otherwise

5.7.6 `prif_unlock`

Description: Unlocks the identified lock variable. If the identified lock variable was not locked by the current image, then an error condition occurs.

```
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`. The referenced variable shall be of type `prif_lock_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.

5.7.7 `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.

```
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

5.7.8 `prif_end_critical`

Description: Completes execution of the critical construct associated with the provided coarray handle.

```
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

5.8 Events and Notifications

5.8.1 `prif_event_post`

Description: Atomically increment the count of the event variable by one.

```
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`. The referenced variable shall be of type `prif_event_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.

5.8.2 `prif_event_wait`

Description: Wait until the count of the provided event variable on the calling image 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.

```
subroutine prif_event_wait( &
    event_var_ptr, until_count, stat, errmsg, errmsg_alloc)
  type(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. The referenced variable shall be of type `prif_event_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `until_count`: the count of the given event variable to be waited for. Has the value 1 if not provided.

5.8.3 `prif_event_query`

Description: Query the count of an event variable on the calling image.

```
subroutine prif_event_query(event_var_ptr, count, stat)
  type(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. The referenced variable shall be of type `prif_event_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `count`: the current count of the given event variable.

5.8.4 `prif_notify_wait`

Description: Wait on notification of an incoming put operation

```
  subroutine prif_notify_wait( &
      notify_var_ptr, until_count, stat, errmsg, errmsg_alloc)
    type(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 on the calling image to be waited on. The referenced variable shall be of type `prif_notify_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `until_count`: the count of the given notify variable to be waited for. Has the value 1 if not provided.

5.9 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 is the then-current team. Team membership is thus strictly hierarchical, following a single path along the tree formed by team creation.

5.9.1 `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`.

```
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

5.9.2 `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`

```
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

5.9.3 `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.

```
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
```

5.9.4 `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.

```
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
```

5.9.5 `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 that became allocated during the change team construct. Prior to invoking `prif_end_team`, the compiler is responsible for invoking `prif_alias_destroy` to delete any `prif_coarray_handle` aliases created as part of the `CHANGE TEAM` construct.

```
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
```

5.10 Collectives

5.10.1 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 `prif_co_broadcast` or `prif_co_reduce`, any numeric for `prif_co_sum`, and integer, real, or character for `prif_co_min` or `prif_co_max`
  - is always `intent(inout)`
  - for `prif_co_max`, `prif_co_min`, `prif_co_reduce`, `prif_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 `prif_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`
  - Identifies the image in the current team that is the root of the collective operation.
  - If `result_image` is omitted, then all participating images receive the resulting value.

5.10.2 `prif_co_broadcast`

Description: Broadcast value to images

```
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
```

5.10.3 `prif_co_max`

Description: Compute maximum value across images

```
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
```

5.10.4 `prif_co_min`

Description: Compute minimum value across images

```
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
```

5.10.5 `prif_co_reduce`

Description: Generalized reduction across images

```
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
```

Further argument descriptions:

- `operation`: the result of `C_FUNLOC` on a reduction operation procedure that meets the requirements outlined in the Fortran standard for the corresponding argument to CO_REDUCE. Note the procedure itself need NOT be interoperable (i.e. `BIND(C)`) nor are the arguments required to have interoperable types.

5.10.6 `prif_co_sum`

Description: Compute sum across images

```
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
```

5.11 Atomic Memory Operations

All atomic operations are fully blocking operations, meaning they do not return to the caller until after all semantics involving the atomic variable are fully committed with respect to all images.

5.11.1 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 compute the relevant remote pointer
  - The referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `image_num`
  - an argument identifying the image to be communicated with
  - is permitted to identify the current image
  - the image index is always relative to the initial team, unless otherwise specified

5.11.2 Non-Fetching Atomic Operations

Description: Each of the following procedures atomically performs the specified operation on a variable in a coarray.

5.11.2.1 Common Argument

- `value`: value to perform the operation with

5.11.2.2 `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
```

5.11.2.3 `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
```

5.11.2.4 `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
```

5.11.2.5 `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
```

5.11.3 Fetching Atomic Operations

Description: Each of the following procedures atomically performs the specified operation on a variable in a coarray, and retrieves the original value.

5.11.3.1 Common Arguments

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

5.11.3.2 `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
```

5.11.3.3 `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
```

5.11.3.4 `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
```

5.11.3.5 `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
```

5.11.4 Atomic Access

Description: The following procedures atomically set or retrieve the value of a variable in a coarray.

5.11.4.1 Common Argument

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

5.11.4.2 `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
```

5.11.4.3 `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
```

5.11.5 `prif_atomic_cas`

Description: Performs an atomic compare-and-swap operation. 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 assign into the atomic variable, if it is initially equal to the `compare` argument

6 Glossary

- **Client Note**: a note that is relevant information for compiler developers who are clients of the PRIF interface
- **Implementation Note**: a note that is relevant information for runtime library developers who are implementing the PRIF interface
- **Source Completion**: The source-side resources provided to a communication operation by this image are no longer in use by the PRIF implementation, and the client is now permitted to modify or reclaim them.

7 Future Work

At present all communication operations are semantically blocking on at least source 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.

8 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

The authors would like to thank Etienne Renault and Jean-Didier Pailleux of SiPearl for providing helpful comments and suggestions regarding an earlier revision of this specification.

9 Copyright

10 Legal Disclaimer