Parallel Runtime Interface for Fortran (PRIF): A Multi-Image Solution for LLVM Flang

Dan Bonachea, Katherine Rasmussen, Brad Richardson, Damian Rouson
Computer Languages and Systems Software Group and NERSC
Lawrence Berkeley National Laboratory, USA
fortran.lbl.gov
fortran@lbl.gov

***Abstract*—Fortran compilers that provide support for Fortran's native parallel features often do so with a runtime library that depends on details of both the compiler implementation and the communication library, while others provide limited or no support at all. This paper introduces a new generalized interface that is both compiler- and runtime-library-agnostic, providing flexibility while fully supporting all of Fortran's parallel features. The Parallel Runtime Interface for Fortran (PRIF) was developed to be portable across shared- and distributed-memory systems, with varying operating systems, toolchains and architectures. It achieves this by defining a set of Fortran procedures corresponding to each of the parallel features defined in the Fortran standard that may be invoked by a Fortran compiler and implemented by a runtime library. PRIF aims to be used as the solution for LLVM Flang to provide parallel Fortran support. This paper also briefly describes our PRIF prototype implementation: Caffeine.**

***Index Terms*—Fortran, Parallel Fortran, HPC, PGAS, RMA, LLVM Flang, Runtime Libraries, Caffeine, GASNet-EX**

I. Introduction

A. Parallel Fortran

Fortran plays important roles in fields ranging from weather [1] and climate [2] to nuclear energy [3], aerospace engineering [4], and fire protection engineering [5]. If you looked at a weather forecast today, received electricity from a power plant licensed by the U. S. Nuclear Regulatory Commission, rode in any one of numerous car or aircraft models, or live in one of 195 countries that signed the Paris climate accord, then Fortran codes impacted your life in one or more ways today even before you encountered this paper. Teams are also writing Fortran software in emerging disciplines such as deep learning [6, 7] and to develop tools in areas where Fortran has not been typically employed, such as package management [8]. Fortran thus remains a key component in the High-Performance Computing (HPC) software ecosystem [9].

As of the 2008 standard [10], Fortran is a natively parallel language, offering Single-Program, Multiple-Data (SPMD) parallelism in the form of multi-image (multi-process) execution and a Partitioned Global Address Space (PGAS) shared memory abstraction that seamlessly supports both single-node deployments and large-scale distributed-memory architectures. The term Coarray Fortran has been used broadly to refer to the features in the language that are related to parallelism. However, there are multiple features, such as the collective subroutines, that do not have any coarray involvement and as such, we prefer the term "Parallel Fortran". We use this term to refer to the entire set of features in Fortran that are provided to achieve multi-image parallelism. This feature set includes coarrays, collective subroutines, teams, synchronization, atomics, locks, events, critical, and notifications. For the purposes of this paper, the term does not include language features focused on single-image parallelism, such as `DO CONCURRENT`, `ELEMENTAL` procedures, and array expressions. Table 1 defines some of the other Fortran terms used throughout this paper.

The full Parallel Fortran feature set is extensive, non-trivial, and there are a large number of inter-related parallel features that are required to be fully Fortran 2023 compliant [11]. By defining a standard interface to a parallel runtime library, we can lower the burden on compiler teams to support the parallel features, by allowing a separate team to take responsibility for their implementation.

B. Motivation and Objectives

As stated in our paper about the Caffeine parallel runtime library, published in LLVM-HPC2022 [12], LLVM Flang is an important addition to LLVM and to the Fortran ecosystem. As expressed in said paper, we have used test-driven development techniques to contribute both compile-time, static semantics tests related to Parallel Fortran and compile-time error checking to the compiler [13]. The Fortran feature support in LLVM Flang has increased significantly since

TABLE 1
Fortran standard terms and definitions (adapted from [11])

Term	Definition
intrinsic	entity or operation defined in the Fortran standard and accessible without further definition or specification
image	instance of a Fortran program, usually corresponding to an OS-level process
team	ordered image set created by a `FORM TEAM` statement, or the initial ordered set of all images
coarray	data structure partitioned across a team's images and accessible by each image in the corresponding team
rank	number of array dimensions of a data entity (zero for a scalar entity)
corank	number of codimensions of a data entity (zero for entities that are not coarrays)
coindexed object	named scalar coarray variable followed by an image selector (an expression including square brackets)

 doi:10.25344/S4N017

our reporting in the Caffeine paper [12] and the compiler now may be used to compile many single-image Fortran programs. However, we believe that the addition of support for Parallel Fortran will greatly expand LLVM Flang's utility and provide more options to Fortran developers wishing to use Parallel Fortran. In exploring solutions for Parallel Fortran support in LLVM Flang, we considered OpenCoarrays [14], but it was deemed an unsuitable solution, partially because it requires a gfortran-specific object descriptor. Instead, we have developed a generalized interface expressed in Fortran called Parallel Runtime Interface for Fortran (PRIF). The semantics of PRIF procedures make no assumptions about the exact implementation of a compiler's object descriptor. Instead PRIF empowers the compiler to make the necessary arrangements and procedure calls which provide any necessary details in a uniform way through the interface.[1] A standardized, generalized interface designed to avoid baked-in assumptions related to specific compilers or runtime libraries frees the compiler from depending on a single runtime library, and vice versa.

An analogous approach was deployed by our research group in the design of the Berkeley UPC Runtime Library (UPCR) [15], a standardized runtime interface for implementing the communication operations in UPC. UPC [16, 17] is a PGAS extension to ISO C which was developed contemporaneously with Coarray Fortran [18] and shares many of the same characteristic parallel language features. UPCR successfully exposed a portable, high-performance and network-independent interface implementing the parallel features of UPC over the GASNet [19] communication system. UPCR was eventually targeted by four different UPC compilers developed by several institutions [20–23]. UPCR enables efficient UPC language execution across dozens of OS, architecture and network combinations [24], encompassing platforms from laptops to supercomputers. This work demonstrated the value of having a standardized interface to an underlying runtime system. Other types of projects which have demonstrated value in analogous situations include the Java bytecode interface [25], and processor instruction set architectures (ISA) [26].

[1]It is worth noting that many PRIF implementations are likely to make internal use of the Fortran-specified C interoperability features, and especially requiring the use of the `CFI_c_desc_t`.

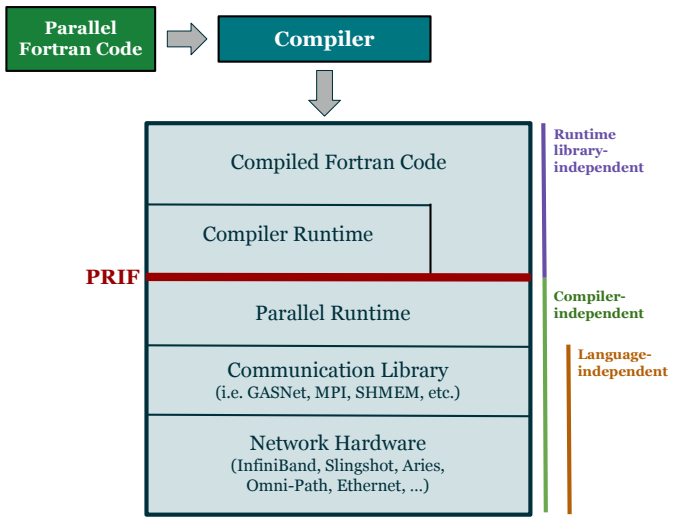

Fig. 1. PRIF's role in a Parallel Fortran software stack

Based on the experiences outlined in [12], we propose that an interface for supporting Fortran's parallel features should be specified **in Fortran** because:

1) We posit that a subset of modern Fortran's non-parallel features are sufficient to write a runtime library, mostly in Fortran, that provides the parallel Fortran features.
2) Furthermore, we maintain that the use of type-agnostic procedure arguments liberates the runtime from directly referencing compiler-specific data structures, enabling the parallel runtime to be portable across compilers.
3) We also suggest that using type- and rank-agnostic arguments (features that first appeared in Fortran's 2018 standard [27]) reduces the complexity of the implementation of certain PRIF procedures.
4) Finally, a prototype implementation exists, Caffeine [28], that deploys these techniques to deliver a portable implementation of Fortran's parallel features, suitable for eventual adoption into compilers such as LLVM Flang.

Writing a parallel runtime library in Fortran opens the door to contributions from any Fortran developer and use by any Fortran compiler, which further broadens the potential community of contributors and adopters. To the best of our knowledge, our implementation-language choice and compiler-independent design contrast starkly with every other parallel runtime library developed for Fortran to date.

Since the Caffeine paper previously presented in this workshop [12], we began development of PRIF, in response to requests from the LLVM and Fortran community for an open runtime standard that could support multiple independent implementations. In the course of that work, a number of discoveries were made that have informed and updated our approach relative to the original paper. One notable change is replacing type- and rank-agnostic arguments in many procedures with the passing of pointers to type-erased storage. This change largely arose from a requirement to support coarray communication of user-defined and non-interoperable types, as well as metadata associated with unspecified aspects of Fortran object representation (e.g., the compiler-specific bit-level representation of Fortran's allocatable components). The PRIF procedures that provide support for Fortran's collective subroutines are notable exceptions where some type- and rank-agnostic arguments remain.

C. Organization

The rest of this paper is organized as follows: Section II describes the PRIF interface itself and discusses the role of the compiler. Section III presents significant design challenges that were encountered during the design of PRIF, and how they were resolved. Section IV presents the current status of Caffeine, our prototype implementation of PRIF, along with future work, and we conclude in Section V.

TABLE 2
DELEGATION OF RESPONSIBILITIES BETWEEN THE FORTRAN COMPILER AND THE PRIF IMPLEMENTATION

Responsibility	Fortran compiler	PRIF 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`	X	
Provide `prif_critical_type` coarrays for `CRITICAL`	X	
Track variable allocation status, including resulting from use of `MOVE_ALLOC`	X	
Provide a finalization subroutine for each coarray allocation of derived type that is finalizable or has allocatable components	X	
Intrinsics related to parallelism, e.g., `NUM_IMAGES`, `COSHAPE`, `IMAGE_INDEX`		X
Allocate and deallocate a coarray		X
Reference a coindexed 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

II. THE INTERFACE

A. The Specification

The PRIF Specification [29] provides a clear definition of the interface for both compiler-writers and runtime library developers. While the latest revision of the PRIF specification provides the technical interface details required to implement or target PRIF, this paper supplies rationale underlying many of the design decisions that have shaped PRIF and explains why PRIF is an important addition to LLVM Flang and the Fortran ecosystem. This section presents a sampling of representative interfaces specified by PRIF, in order to provide a concrete basis for illustrating the interface design decisions.

The PRIF Specification defines procedure interfaces with the required number and ordering of arguments, and the required types, kinds and intent of those arguments. The document also outlines the required derived types and named constants that are needed for the interface. Additionally, the specification includes some rationale, notes for the developers of the compiler or the runtime library, a meticulous change log, and a section that discusses potential future additions to the interface. Additional PRIF design rationale is presented in this paper's §III. An important goal of the PRIF Specification is delineating the responsibilities for implementing the various parallel Fortran features, as outlined in Table 2. An X in the "Fortran compiler" column indicates that the compiler has the primary responsibility for that task, while an X in the "PRIF library" column indicates that the compiler will invoke PRIF to perform the task and the PRIF-compliant library has primary responsibility for the task's implementation.

B. The Compiler

As mentioned earlier, PRIF is designed as a standardized interface to any conformant runtime library supporting the parallel features. It is the compiler's responsibility to orchestrate procedure calls to the PRIF library, as dictated by the invocations of parallel features in the Fortran source program. While this could potentially be achieved through source-to-source transformation, we expect most compilers will use later phases of processing to accomplish this. As shown in Figure 1, the implementation of PRIF will supplement the compiler's own runtime library that is used to support non-parallel features.

C. The `prif` Module

PRIF-compliant runtime libraries define a module named `prif`, which contains publicly accessible definitions of all of the types, named constants and procedure interfaces outlined in the PRIF Specification. The compiler will compile Fortran portions of the PRIF library implementation and hence have access to their definitions in the same manner as it would for other Fortran modules. PRIF deliberately does not mandate that different PRIF implementations be ABI-interchangeable, maximizing flexibility for library implementations.

D. ISO_Fortran_Env Types and Named Constants

In the Fortran language, the `ISO_FORTRAN_ENV` module defines a number of intrinsic derived types and named constants that are involved in the parallel feature-set. These include the derived types `TEAM_TYPE`, `EVENT_TYPE`, `LOCK_TYPE`, and `NOTIFY_TYPE`, which are used to support teams, events, locks, and notifications, respectively. Named constants from the same module which are involved in Parallel Fortran include `ATOMIC_INT_KIND`,

```
module subroutine prif_allocate_coarray( &
    lcobounds, ucobounds, lbounds, ubounds, element_size, final_func, &
    coarray_handle, allocated_memory, stat, errmsg, errmsg_alloc)
  implicit none
  integer(c_intmax_t), dimension(:), intent(in)  :: lcobounds, ucobounds
  integer(c_intmax_t), dimension(:), intent(in)  :: lbounds, ubounds
  integer(c_size_t),                 intent(in)  :: element_size
  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=:), allocatable,     intent(inout), optional :: errmsg_alloc
end subroutine
```

Fig. 2. The interface for `prif_allocate_coarray`

ATOMIC_LOGICAL_KIND, CURRENT_TEAM and many others. The Fortran standard mandates that compilers provide the ISO_FORTRAN_ENV module to users, and the PRIF Specification in turn delineates that the runtime library implementation provides the definitions for these relevant intrinsic derived types and named constants through equivalents in the `prif` module.

The components comprising the PRIF definitions of the Fortran intrinsic derived types are deliberately unspecified by PRIF, and to ensure portability the compiler should not hard-code reliance on those details. However, at compile-time the detailed representation corresponding to a given PRIF implementation will be visible to the compiler in the interface declarations of the `prif` module.

E. PRIF-specific types

The interface also defines two PRIF-specific helper types, which are required to support the PRIF procedures: `prif_coarray_handle` and `prif_critical_type`. These types should not appear in normal Fortran code, but provide runtime implementations a place to keep relevant metadata and enable the compiler to orchestrate passing it back where appropriate.

F. PRIF-specific Named Constants

PRIF also specifies additional named constants, not required by the Fortran standard, that provide important information to the compiler when invoking certain PRIF procedures. For example, PRIF_STAT_OUT_OF_MEMORY may be returned from calls to `prif_allocate_coarray` or `prif_allocate` when a low-memory condition has occurred.

G. PRIF Procedures

The remainder of the PRIF Specification deals with the interfaces for the PRIF procedures. For each intrinsic Fortran procedure related to multi-image parallelism, there are corresponding `prif` module procedures. For example, in order to support the Fortran intrinsic function NUM_IMAGES, PRIF specifies the `prif_num_images` module procedure. The remaining features of Parallel Fortran, such as coarray declarations and references, or synchronization statements, also have corresponding PRIF procedures, such as `prif_allocate_coarray` or `prif_sync_all`. The constraints and semantics associated with the arguments and call to a given `prif` module procedure match those of the analogous argument to the Parallel Fortran feature, except where explicitly specified otherwise. Below we describe the interfaces and semantics for some of the key PRIF procedures to illustrate the design and conventions of PRIF.

```
! Example coarray declaration in Fortran:
integer :: coarr(10)[*]

! Equivalent compiler-generated PRIF code:
integer(c_int) :: ni
call prif_num_images(ni)
call prif_allocate_coarray(                &
  lcobounds=[1_c_intmax_t],                &
  ucobounds=[ni],                          &
  lbounds=[1_c_intmax_t)],                 &
  ubounds=[10_c_intmax_t],                 &
  element_size=int_size_in_bytes,          &
  final_func=c_null_funptr,                &
  coarray_handle=coarr_coarray_handle,     &
  allocated_memory=coarr_mem)
```

Fig. 3. Fortran coarray declaration and equivalent PRIF procedure call

1) `prif_allocate_coarray`: Figure 2 demonstrates the interface for `prif_allocate_coarray`. This call is invoked collectively by compiler-generated code to construct every coarray in the program, regardless of how the coarray is constructed at the Fortran syntax level. So for example, every ALLOCATE statement that allocates a coarray object will invoke `prif_allocate_coarray` once for each coarray.

```
module subroutine prif_put( &
    image_num, coarray_handle, offset, current_image_buffer, &
    size_in_bytes, stat, errmsg, errmsg_alloc)
  implicit none
  integer(c_int),                intent(in) :: image_num
  type(prif_coarray_handle),     intent(in) :: coarray_handle
  integer(c_size_t),             intent(in) :: offset
  type(c_ptr),                   intent(in) :: current_image_buffer
  integer(c_size_t),             intent(in) :: size_in_bytes
  integer(c_int),                intent(out),   optional :: stat
  character(len=*),              intent(inout), optional :: errmsg
  character(len=:), allocatable, intent(inout), optional :: errmsg_alloc
end subroutine
```

Fig. 4. The interface for `prif_put`

Fortran also allows coarrays to be constructed statically (e.g., as module or common-block variables), and the compiler targeting PRIF is also responsible for inserting appropriate calls to `prif_allocate_coarray` for each such coarray as part of program initialization. Figure 3 shows an example of a coarray declaration in Fortran, and some corresponding code the compiler could generate that invokes PRIF to construct that coarray.

The salient details of the `prif_allocate_coarray` arguments and calling convention are as follows:

- The call is collective over all the images in the current team, who must all pass compatible arguments for constructing corresponding coarrays.
- The call constructs a coarray descriptor and returns a `prif_coarray_handle` value that acts as a handle to that descriptor. Many PRIF calls accept a `prif_coarray_handle` as a reference to a given coarray descriptor.
- The `lcobounds, ucobounds` arguments specify the cobounds for the coarray being constructed. The PRIF implementation tracks this information in the coarray descriptor, and uses it to answer future queries on the coarray (e.g., the `COSHAPE` intrinsic which is supported using `prif_coshape`).
- The `prif_allocate_coarray` call notably receives deliberately limited information regarding the static *type* of the coarray. This static information is never directly passed to PRIF, because in general coarrays may be defined over user-defined derived types that are program-specific and whose details are unavailable to the PRIF implementation. Instead, the `element_size` and `lbounds, ubounds` arguments respectively pass the type-erased storage size and array bounds of the opaque memory that should be allocated to back the coarray elements.
- The `allocated_memory` argument returns a `c_ptr` to the caller referencing the uninitialized storage that serves as the coarray element storage for the calling image. The compiler is responsible for initializing this storage (as appropriate) and associating the element memory with the relevant coarray variable.

2) `prif_put`: Once a coarray has been allocated, it can be used in Remote Memory Access (RMA) operations that read or write data stored in the coarray on any image. For example, a Fortran-level assignment into a coindexed object may be translated into a call to the `prif_put` subroutine, the simplest in a family of procedures that can be used to copy local data into a remote coarray.

Figure 4 presents the interface for the `prif_put` subroutine. The pertinent details to notice are as follows:

- The target image is selected by the scalar integer `image_num` argument, which identifies the image index in the initial team of all images. This notably differs from the richer Fortran-level image selectors, where user-provided cosubscripts are interpreted relative to the cobounds of the coarray, and also optionally a provided team specifier, which defaults to the current team. PRIF instead provides the `prif_image_index` subroutine that mirrors the `IMAGE_INDEX` intrinsic and performs translation of cosubscripts for a given coarray into a scalar image index in a selected team. PRIF deliberately factors this image translation work out of the RMA interface, enabling compiler optimizations such as common subexpression elimination to hoist redundant image translation computation out of loops.
- The selected data to be updated in the coarray is identified to `prif_put` through the combination of a `prif_coarray_handle` which names a coarray descriptor, an `offset` in bytes from the beginning of the coarray elements, and a `size_in_bytes` indicating the contiguous amount of data to be copied. This is deliberately a type-less interface, which allows it to operate on coarrays of any type without knowledge of the underlying type declarations and unspecified/compiler-specific layout information. The compiler is responsible for using the static type information associated with the

coindexed object to compute the byte offset and length in bytes of the access operation.
- The `current_image_buffer` argument is a `c_ptr` referencing the start of the source data on the calling image. All put operations in PRIF semantically block on *source completion*, meaning the source data is fully "consumed" before the call returns, allowing the caller to overwrite or reclaim that memory. In practice this generally means the PRIF implementation has either copied-out the source data (e.g., into a temporary staging location) or injected it into a reliable network transmission. Note that return from the `prif_put` call does NOT imply the data transfer has been committed into remote coarray storage (i.e., *remote completion*). PRIF follows the Fortran memory consistency model, which specifies that image control statements (such as `SYNC ALL`, i.e., `prif_sync_all`) are used to order parallel execution segments and establish remote completion of RMA accesses.

The arguments to `prif_get` (used for reads of coindexed objects) are identical to those of `prif_put` (and hence omitted for brevity); the only salient semantic difference is the direction of data copy is reversed. PRIF specifies an entire family of `prif_put` and `prif_get` subroutines, which capture a variety of Fortran-level use cases: contiguous versus non-contiguous (i.e., strided) data, direct versus indirect access (§III-B), and use of the Fortran 2023 `NOTIFY=` specifier for coindexed assignment.

III. Technical Considerations for the PRIF Design

A. Communication library

One of the design goals of PRIF is to insulate the Fortran compiler from details of the communication libraries used to implement multi-image communication across a variety of parallel system architectures. As such, much of the PRIF design was influenced by considering how different communication libraries could be used to implement the runtime library. Caffeine [28], our prototype implementation of PRIF, targets the GASNet-EX [30] communication library. GASNet-EX is a language-independent, networking middleware layer that provides network-independent, high-performance communication primitives for HPC, including one-sided RMA and Active Messages. We also envision an implementation of PRIF that targets the widely deployed Message Passing Interface (MPI) communication library, in particular utilizing the one-sided passive-target MPI RMA interfaces to implement PRIF coarray access routines and atomic memory operations. In particular, the design of PRIF's direct versus indirect accesses described in the next section was directly influenced by the desire to enable efficient implementation of important common cases using the MPI RMA window abstraction.

B. Symmetric and Non-symmetric Shared Objects

Fortran coarrays are shared objects, meaning that any image in a parallel execution can perform RMA operations (puts and gets) to access the elements of a coarray. These accesses are one-sided, meaning that only the image initiating the operation is explicitly involved in the RMA.

Fortran specifies that coarray allocation and deallocation are always collective operations, meaning that all images in the current team participate in (and synchronize during) the allocation and deallocation of a coarray, and that all images agree on the identity and size of the corresponding coarrays. This design philosophy enables an implementation strategy known as *symmetric shared heap management*, whereby coarray objects are positioned at memory locations across all images in a manner that maintains a symmetric property.

This technique imparts several implementation benefits. Most fundamentally, it reduces the amount of metadata required to perform one-sided RMA by ensuring that a memory reference for a coarray element on any remote image can be computed mathematically from the address of the corresponding local element, without additional communication. A second important benefit arising from the collective allocation of shared memory objects is that it provides the runtime library the opportunity to efficiently exchange information (e.g., network-level memory registration keys) to accelerate later RMA operations involving the coarray.

If coarray elements were the only shared objects, then the entire shared memory heap could be managed in a symmetric manner, and the design of PRIF would be greatly simplified. However, Fortran also allows remote access to objects that were not allocated during any coarray allocation, but are reachable via indirection through a coarray. One specific example of this is a coarray of derived type with an `allocatable` or `pointer` component. The target memory of such a component is allocated non-collectively by the image with affinity, and the allocation of the target object referenced by that component can happen long after the coarray itself is allocated, or even before in the case of a `pointer` component. Allocation of such target objects is non-collective and hence non-symmetric, as there is no guaranteed correspondence in the size (or existence) of such objects across images. Further, there is no guaranteed "binding" between the storage backing the coarray and the storage backing these indirected components (in particular, they need not reside at a fixed offset from the base of the coarray storage) yet they are nevertheless remotely accessible by Fortran semantics. We refer to such objects as non-symmetric shared objects.

This language semantic imposes a requirement on compliant runtime implementations, and in turn also on the PRIF Specification to expose capabilities fulfilling this requirement. PRIF deals with this situation by providing two flavors of shared heap allocation: symmetric/collective allocation of coarrays (via `prif_allocate_coarray` and `prif_deallocate_coarray`), and non-symmetric/non-collective allocation of other potentially shared objects (via `prif_allocate` and `prif_deallocate`).

Correspondingly, many communication operations in PRIF have both a "direct" procedure variant that operates on symmetric shared objects (those residing directly in coarray element storage), and an "indirect" procedure variant that

operates on non-symmetric shared objects (those reachable via one or more `allocatable` or `pointer` components of a derived-type coarray). The key interface distinction is that direct access variants represent the target memory location using a combination of `prif_coarray_handle` and an offset in bytes from the start of the coarray data (as shown in Figure 4), whereas indirect access variants represent the target memory location using its address in the virtual memory space of the target process. This design enables implementations to reap the benefits of symmetric heap management for RMA involving symmetric coarray objects, while preserving the semantically required capability of RMA on non-symmetric objects.

C. Teams

The use of Fortran teams, and specifically the `CHANGE TEAM` construct, implicitly modifies the semantics of various parallel features. For example, collective operations such as `CO_SUM` are only performed over members of the current team. Additionally, synchronization and allocation of coarrays only occur with members of the current team. Several other parallel features accept a team number or `TEAM_TYPE` variable as an argument, but use the current team by default when one is not provided. While this doesn't significantly impact the design of PRIF, it is worth noting for potential implementations.

The PRIF implementation is responsible for constructing teams (via `prif_form_team`), answering team state queries (`prif_get_team`, `prif_team_number`) and tracking the current team, which is updated in a stack-like discipline using `prif_change_team` and `prif_end_team`.

D. Clean-up operations at coarray deallocation

For coarrays of derived type, coarray deallocation in general requires execution of compiler-generated code for some clean-up actions. One example of such actions is running user-provided final subroutines when the coarray contains objects of a finalizable derived type. Another example is when a coarray of derived type contains `allocatable` components, any referenced objects need to be transitively deallocated. The PRIF implementation has no knowledge of derived types defined by the user at the Fortran language level, so the compiler must assume responsibility for generating code to perform these cleanup actions upon deallocation.

Coarray deallocation can occur either explicitly (e.g., via a `DEALLOCATE` statement or a local variable scope exit, both of which are translated into a call to `prif_deallocate_coarray`) or implicitly (e.g., via an `END TEAM` statement closing a `CHANGE TEAM` construct that allocated one or more coarrays). PRIF unifies both use cases under one mechanism; the `prif_allocate_coarray` call includes a `final_func` argument, which accepts an optional function pointer to a cleanup callback that will be invoked by PRIF immediately before coarray deallocation. For any coarray allocation that potentially requires cleanup actions, the compiler is expected to generate a cleanup function that accepts the `prif_coarray_handle` and implements appropriate finalization actions.

E. Reassociation of Coarrays

Fortran's `CHANGE TEAM` construct includes an optional coarray association list, which enables the programmer to re-associate existing coarrays with a new name (and optionally altered corank and cobounds) for the duration of the `CHANGE TEAM` construct. Similar re-association can also be performed when calling procedures that accept a coarray dummy argument. PRIF specifies the `prif_alias_create` and `prif_alias_destroy` procedures that enable this functionality. `prif_alias_create` creates a new coarray descriptor, with possibly altered corank and cobounds, that aliases the same (unmodified) coarray element data.

F. Allocatable Coarray Tracking and Move Alloc

Fortran allows declaration of an `allocatable` coarray variable, which represents a coarray that can be allocated and deallocated dynamically (and collectively) during program execution. Once allocated, such coarrays behave similarly to any other, and in particular can be passed down to called procedures via a coarray dummy argument (which may or may not include an `allocatable` attribute). The `ALLOCATED` intrinsic inquiry function allows the programmer to query whether any given `allocatable` variable is currently allocated. The `MOVE_ALLOC` intrinsic subroutine enables the Fortran programmer to effectively transfer ownership of an unmodified target object from one `allocatable` variable to another.

The combination of these features presents some potential challenges to implementers of parallel Fortran. For example, a procedure that receives an `allocatable` coarray as an `intent(inout)` dummy argument is in general permitted to allocate or deallocate that coarray. Upon return from the callee procedure, the corresponding `allocatable` variable in the caller's scope must reflect such operations; for example, subsequent calls to the `ALLOCATED` query must reflect any change in state. Similarly, such procedures are also in general permitted to call `MOVE_ALLOC` to transfer ownership of a coarray object from one `allocatable` coarray variable to another, again potentially impacting outer scopes in the call chain. One consequence of these semantics is that upon coarray deallocation, the compiler may be required to update the allocation state of `allocatable` variables that currently own the target object, and the identity of such `allocatable` variables may be non-local and statically unknowable.

PRIF is designed to avoid compiler-specific assumptions about object representations, and in particular never directly manipulates `allocatable` variables (as that would require detailed knowledge of the compiler-specific internal representations used to support the `allocatable` attribute). One direct consequence of this design is that PRIF does not include a `MOVE_ALLOC` operation; it is the compiler's responsibility to implement `MOVE_ALLOC` via a combination of appropriate manipulation of `allocatable` variable representation and

```
module subroutine prif_co_reduce( &
    a, operation, result_image, stat, errmsg, errmsg_alloc)
  implicit none
  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
```

Fig. 5. The interface for `prif_co_reduce`

calls to PRIF procedures. For example, according to the Fortran standard, `MOVE_ALLOC` with coarray arguments is an image control statement that requires synchronization, thus the compiler should likely insert calls to `prif_sync_all` as part of the implementation.

As explained in §III-D, PRIF enlists a compiler-generated code callback at coarray deallocation time to perform any necessary cleanup actions. One important responsibility of such a cleanup callback is to update the allocation state of any `allocatable` variables referencing the coarray being deallocated. Because the identity of such variables cannot be statically known in general, particularly for implicit deallocation at `END TEAM`, the compiler needs the ability to maintain some dynamic information associated with any `allocatable` coarray, and later retrieve that information (e.g., inside the cleanup callback) to perform appropriate actions. PRIF enables this capability using the `prif_set_context_data` and `prif_get_context_data` subroutines, which respectively store and retrieve a pointer to compiler-owned metadata that is associated with the coarray object. This metadata pointer is also shared between any aliased coarray descriptors created using coarray re-association (§III-E), because all aliases share the same underlying target object and allocation status.

G. Collective Subroutines

The interfaces to the collective subroutines are not particularly complicated, but some requirements on `CO_BROADCAST` and `CO_REDUCE` have implications for both PRIF clients and implementers.

`CO_BROADCAST` is required to operate on derived types, including those with `allocatable` or polymorphic components. Implementing this directly in a runtime would require knowledge of the internal representation of object descriptors for a given compiler. To avoid that dependency, `prif_co_broadcast` instead specifies it is the compiler's responsibility to generate code that performs any necessary communication through additional calls to PRIF procedures, for example to arrange for each image to execute the necessary allocations of the components and broadcast their data separately.

`CO_REDUCE` is also required to operate on derived types, and it additionally accepts a user-provided function for the reduction operation. It is the compiler's job to enforce that the provided function matches the type of the data argument. The data argument is prohibited from having `allocatable` or `pointer` components, but the arguments are not guaranteed to be C-interoperable and could have length type parameters, which could complicate matters for certain compilers. In particular, a compiler's representation of the `type(*)` argument to `prif_co_reduce` (see Figure 5), may not match the representation expected by a user-provided operation function for the type of the actual argument if it is passed directly as the `operation` argument. We are still exploring whether this detail (which is specific to `CO_REDUCE`) can be transparently accommodated by PRIF implementations, or whether it will need to be explicitly addressed by compilers when calling `prif_co_reduce`.

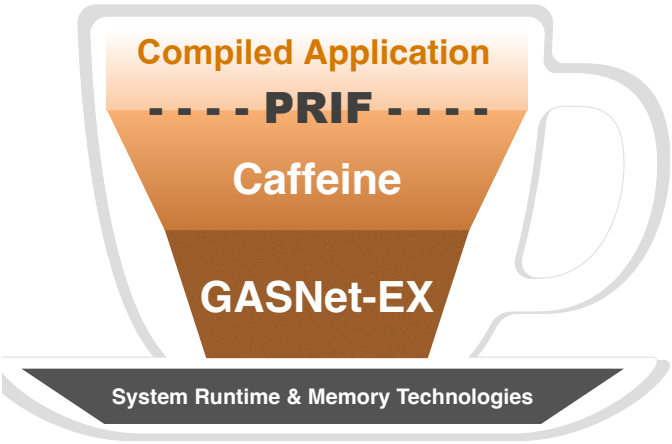

Fig. 6. Caffeine software stack

IV. Caffeine and Future Work

A. Caffeine Implementation Status

As mentioned earlier, a prototype implementation of a runtime conforming to the PRIF Specification is in-progress. Figure 6 illustrates the Caffeine system software stack, where

TABLE 3
STATUS OF CAFFEINE'S SUPPORT FOR THE PARALLEL FEATURES OF FORTRAN

Feature	**Status**
Program launch	yes
Normal termination: `STOP` and `END PROGRAM` statements	yes
Error termination: `ERROR STOP` statement	yes
Collective subroutines: `CO_{BROADCAST,SUM,MIN,MAX,REDUCE}`	yes
Image enumeration: `THIS_IMAGE`, `NUM_IMAGES`, `IMAGE_INDEX` intrinsic functions	partial
Synchronization: `SYNC {ALL,IMAGES,MEMORY,TEAM}` statements	partial
Coarrays: declaration, access, (de)allocation, inquiry functions	partial
Teams: `TEAM_TYPE` intrinsic type and `{FORM,CHANGE,END} TEAM` statements	partial
Critical construct: `CRITICAL` and `END CRITICAL`	no
Atomics: `ATOMIC_{INT,LOGICAL}_KIND` kind parameters and `ATOMIC_{DEFINE,REF,...}` subroutines	no
Locks: `LOCK` and `UNLOCK` constructs	no
Events: `EVENT_TYPE` intrinsic type, `EVENT_QUERY` subroutine and `EVENT {POST,WAIT}` statements	no
Notifications: `NOTIFY_TYPE` intrinsic type and `NOTIFY WAIT` statements	no
Failed/stopped images: `FAIL IMAGE` statement, `{FAILED,STOPPED}_IMAGES` intrinsic functions, related constants	no

the Caffeine library implements PRIF using the GASNet-EX library for network-level communication services. Table 3 reports on the current implementation status of various PRIF features in Caffeine at the time of writing. We believe the progress so far is already sufficient to enable support for a significant number of real parallel Fortran applications. We intend to continue work on Caffeine to enable complete support for the parallel Fortran feature set.

B. Future Work

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

At present PRIF does not expose a capability for an image to *directly* access shared objects associated with another image (i.e., via simple load/store instructions). We acknowledge that in some cases an image may be co-located in the same physical memory domain with the image whose coarray data it needs to access, but we don't currently expose this capability to PRIF clients. In the future we intend to expose shared-memory bypass for coarray access to PRIF clients.

V. CONCLUSIONS

While Fortran has been supporting scientific applications for many decades, the introduction of multi-image SPMD/PGAS parallelism in Fortran 2008 enriched the functionality available to its users through features defined in the language itself. The Parallel Fortran feature set greatly expanded in subsequent specification revisions, and a fully Fortran 2023 compliant compiler must support all of these features. While LLVM Flang has made great progress in the Fortran features that it supports for compilation, it remains a goal to add support for Parallel Fortran features to the compiler. We have shown our work towards achieving that goal which includes developing PRIF, a new, standardized interface that is both compiler and runtime library agnostic to reduce the burden on compiler teams when adding support for Parallel Fortran.

PRIF is an interface written in Fortran, which provides significant benefits, such as type-agnostic and type-erased arguments allowing a runtime implementation to be portable across compilers. We detailed the structure of the PRIF Specification and presented an overview of the contents it mandates for the `prif` module, which includes derived types, named constants, and procedure interfaces. Throughout the development of PRIF, we encountered a variety of complex, technical considerations required by the design of multi-image parallelism in Fortran. We have described the most critical of these considerations and the ways in which PRIF addresses the design of the language. Additionally, we provided an introduction to Caffeine, a PRIF implementation, which currently supports a partial but significant and frequently used set of Parallel Fortran features and expressed our plans for future work in PRIF and in Caffeine.

ACKNOWLEDGMENTS

Selected portions of this paper are reproduced with permission from [12, 29].

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research.

This research was supported in part 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.

Acronyms

HPC High-Performance Computing

PGAS Partitioned Global Address Space

RMA Remote Memory Access

SPMD Single-Program, Multiple-Data

PRIF Parallel Runtime Interface for Fortran

References

[1] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill *et al.*, "A description of the advanced research WRF model version 4," *National Center for Atmospheric Research: Boulder, CO, USA*, vol. 145, p. 145, 2019, doi:10.5065/1dfh-6p97.

[2] G. Danabasoglu, J.-F. Lamarque, J. Bacmeister, D. Bailey, A. DuVivier, J. Edwards, L. Emmons *et al.*, "The community earth system model version 2 (CESM2)," *Journal of Advances in Modeling Earth Systems*, vol. 12, no. 2, 2020, doi:10.1029/2019MS001916.

[3] M. Ding, X. Zhou, H. Zhang, H. Bian, and Q. Yan, "A review of the development of nuclear fuel performance analysis and codes for PWRs," *Annals of Nuclear Energy*, vol. 163, p. 108542, 2021, doi:10.1016/j.anucene.2021.108542.

[4] R. T. Biedron, J.-R. Carlson, J. M. Derlaga, P. A. Gnoffo, D. P. Hammond, W. T. Jones, B. Kleb, E. M. Lee-Rausch, E. J. Nielsen, M. A. Park *et al.*, "FUN3D Manual: 13.2," *NASA TM*, vol. 219661, 2017, https://fun3d.larc.nasa.gov/papers/FUN3D_Manual-13.2.pdf.

[5] K. B. McGrattan, R. J. McDermott, C. G. Weinschenk, and G. P. Forney, "Fire dynamics simulator user's guide," *NIST special publication*, vol. 1019, 2013, doi:10.6028/NIST.sp.1019.

[6] J. Ott, M. Pritchard, N. Best, E. Linstead, M. Curcic, and P. Baldi, "A fortran-keras deep learning bridge for scientific computing," *Scientific Programming*, vol. 2020, no. 1, 2020, doi:10.1155/2020/8888811.

[7] *Inference-Engine*, https://go.lbl.gov/inference-engine.

[8] *Fortran Package Manager*, https://fpm.fortran-lang.org.

[9] B. Austin *et al.*, *NERSC-10 Workload Analysis*, 2020, doi:10.25344/S4N30W.

[10] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:2010*. International Organization for Standardization (ISO), Oct 2010, https://www.iso.org/standard/50459.html.

[11] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — ISO/IEC 1539-1:2023*. International Organization for Standardization (ISO), Nov 2023, https://www.iso.org/standard/82170.html.

[12] D. Rouson and D. Bonachea, "Caffeine: CoArray Fortran Framework of Efficient Interfaces to Network Environments," in *Proceedings of the Eighth Annual Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC2022)*, November 2022, doi:10.25344/S4459B.

[13] K. Rasmussen, D. Rouson, N. George, D. Bonachea, H. Kadhem, and B. Friesen, "Agile Acceleration of LLVM Flang Support for Fortran 2018 Parallel Programming," in *Research Poster at the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC22)*, Nov 2022, doi:10.25344/S4CP4S.

[14] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson, "OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers," in *Partitioned Global Address Space Programming Models (PGAS)*, 2014, doi:10.1145/2676870.2676876.

[15] *Berkeley UPC Runtime Library*, Lawrence Berkeley National Laboratory, https://upc.lbl.gov/docs/system/index.shtml.

[16] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[17] UPC Consortium, "UPC Language and Library Specifications, v1.3," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-6623E, Nov. 2013, doi:10.2172/1134233.

[18] R. W. Numrich and J. Reid, "Co-Array Fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2, 1998, pp. 1–31, doi:10.1145/289918.289920.

[19] D. Bonachea and P. H. Hargrove, "GASNet specification, v1.8.1," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001064, August 2017, doi:10.2172/1398512.

[20] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick, "A Performance Analysis of the Berkeley UPC Compiler," in *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003, doi:10.1145/782814.782825. [Online]. Available: https://escholarship.org/uc/item/91v1j2jw

[21] *GNU Unified Parallel C (GCC/UPC) Compiler*, Intrepid Technology, Inc., https://github.com/Intrepid/GUPC.

[22] *Clang UPC Compiler*, https://clangupc.github.io/clang-upc/.

[23] *Clang UPC2C Translator*, https://clangupc.github.io/clang-upc2c/.

[24] *Berkeley UPC Runtime Downloads*, Lawrence Berkeley National Laboratory, https://upc.lbl.gov/download/.

[25] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Addison-wesley, 2013.

[26] D. Chisnall, "How to Design an ISA: The popularity of RISC-V has led many to try designing instruction sets," *Queue*, vol. 21, no. 6, pp. 27–46, 2023, doi:10.1145/3639445.

[27] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:2018*. International Organization for Standardization (ISO), Nov 2018, https://www.iso.org/standard/72320.html.

[28] *Caffeine: CoArray Fortran Framework of Efficient Interfaces to Network Environments*, https://go.lbl.gov/caffeine.

[29] D. Bonachea, K. Rasmussen, B. Richardson, and D. Rouson, "Parallel Runtime Interface for Fortran (PRIF) Specification, Revision 0.4," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001604, July 2024, doi:10.25344/S4WG64.

[30] D. Bonachea and P. H. Hargrove, "GASNet-EX: A High-Performance, Portable Communication Library for Exascale," in *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, ser. LNCS, vol. 11882. Springer, October 2018, doi:10.25344/S4QP4W.