Title

Caffeine: CoArray Fortran Framework of Efficient Interfaces to Network Environments

ISBN

9781665475587

Authors

Rouson, Damian

Bonachea, Dan

Publication Date

2022-11-13

DOI

10.25344/S4459B

Peer reviewed

Caffeine: CoArray Fortran Framework of Efficient Interfaces to Network Environments

Damian Rouson, Dan Bonachea
Computer Languages and Systems Software Group
Lawrence Berkeley National Laboratory, USA
{Rouson, DOBonachea}@lbl.gov

***Abstract*—This paper provides an introduction to the CoArray Fortran Framework of Efficient Interfaces to Network Environments (Caffeine), a parallel runtime library built atop the GASNet-EX exascale networking library. Caffeine leverages several non-parallel Fortran features to write type- and rank-agnostic interfaces and corresponding procedure definitions that support parallel Fortran 2018 features, including communication, collective operations, and related services. One major goal is to develop a runtime library that can eventually be considered for adoption by LLVM Flang, enabling that compiler to support the parallel features of Fortran.**

The paper describes the motivations behind Caffeine's design and implementation decisions, details the current state of Caffeine's development, and previews future work. We explain how the design and implementation offer benefits related to software sustainability by lowering the barrier to user contributions, reducing complexity through the use of Fortran 2018 C-interoperability features, and high performance through the use of a lightweight communication substrate.

***Index Terms*—HPC, PGAS, RMA, LLVM Flang, Exascale Computing, Runtime Libraries, GASNet-EX**

I. Introduction

A. Why Fortran matters

Rumors of Fortran's demise are greatly exaggerated. Sixty-five years after the publication of the language's seminal description [1], Fortran has reached Medicare age and survived longstanding calls for its retirement [2]. Despite published descriptions of Fortran as an "infantile disorder," [3] the world's first widely used high-level programming language remains relevant. User surveys and system monitoring at the National Energy Research Scientific Computing Center (NERSC) [4] over the past several years reveal that Fortran remains very popular in the workload of this production supercomputing center (Fig. 1). Fortran plays important roles in fields ranging from weather [5] and climate [6] to nuclear energy [7], aerospace engineering [8], and fire protection engineering [9]. 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.

To ensure a sustainable path for future Fortran code development, a vibrant community of developers at varying educational and career stages has undertaken an effort to grow and modernize the Fortran ecosystem [11], including extending the application of the language into non-traditional domains such as software package management [12]. Among the other many signs of new growth in the Fortran world is the increase in the number of production Fortran compiler projects over the past 5 years. These projects include new, open-source compilers, such as LFortran [13] and LLVM Flang, along with proprietary compilers from vendors who either did not previously produce a Fortran compiler or vendors who have undertaken the replacement of their legacy Fortran compiler with a new compiler. The LLVM compiler infrastructure [14] plays a central role in many such efforts. Recent versions of the Intel [15] and IBM [16] Fortran compiler front ends, for example, now use an LLVM back end. Recent versions of the the NVIDIA, Arm, AMD, and Huawei compilers are essentially private forks of "Classic Flang" [17], which also targets LLVM but with plans for eventual replacement by LLVM Flang, presumably sometime after LLVM Flang reaches feature parity with Classic Flang. All of these developments portend potentially broad impact for work that advances LLVM Flang.

B. Motivation and Objectives

Because of the paramount importance of parallelism in High-Performance Computing (HPC), our work centers around the Fortran 2018 parallel programming feature subset that is commonly called "Coarray Fortran". This feature subset adds Single-Program, Multiple-Data (SPMD) multi-process support to Fortran. Coarrays provide a Partitioned Global Address Space (PGAS) memory model; every coarray represents a

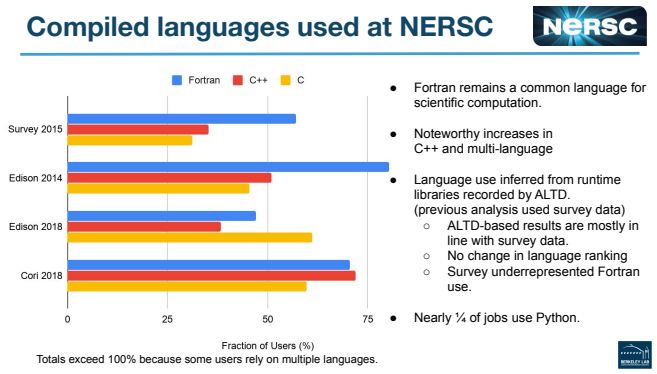

Fig. 1. Programming languages used at the National Energy Research Scientific Computing Center (reproduced with permission from [10])

 doi:10.25344/S4459B

distributed data structure, where the array elements are spread across all the processes (images), and any element can be directly accessed by any process using one-sided Remote Memory Access (RMA) communication. The Fortran 2018 parallel feature subset also includes various collective communication operations (such as broadcasts and reductions), interprocess synchronization operations (such as event semaphores and barriers), and a process grouping abstraction (teams). Tbl. 1 defines several Fortran terms used throughout this paper.

LLVM Flang currently provides an experimental capability for compiling Fortran 95 programs [18]. However, the LLVM Flang front end can parse standard Fortran 2018 [19] and can perform compile-time checks of static semantics. The Berkeley Lab Flang project [20] focuses on testing. Following Test-Driven Development (TDD) practices, we aim to develop a comprehensive set of unit tests and to use the resulting test suite as a specification for functionality that Flang and Caffeine must support to comply with the 2018 standard. Our tests fall into two categories: (1) compile-time, static semantics tests for Flang [21] and (2) runtime behavioral tests for Caffeine. Consistent with TDD, we add features to the Flang frontend when our tests expose missing features such as Flang not recognizing a procedure as intrinsic.

As of this writing, we have pushed 30 commits with static semantics tests to the main branch of the LLVM-Project repository, and we have merged runtime tests with 44 assertions exercising ten parallel features into the main branch of the Caffeine repository [22]. Section III-C of this paper describes the runtime tests. Ultimately, we aim to either merge Caffeine into LLVM Flang or establish it as an external dependency, providing Flang users with access to Fortran's parallel features.

C. Contributions

The remainder of this paper is organized as follows: Section II provides an overview of the Caffeine software stack and the compiler development workflow that Caffeine's design implicitly proposes; Section III presents the status of Caffeine development, Caffeine's compiler-facing interface, and the runtime unit tests that serve as the specification driving Caffeine development; Section IV describes some future work required for Caffeine to cover all of Fortran's parallel features.

The main contributions of this paper are the following:

- 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.
- 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.
- We also suggest that use of 2018 type- and rank-agnostic arguments greatly reduces the complexity of the resulting runtime relative to using earlier Fortran standards.
- Finally, we present initial work on Caffeine that deploys these techniques to deliver a portable implementation of Fortran's parallel features, suitable for eventual adoption into compilers such as LLVM Flang.

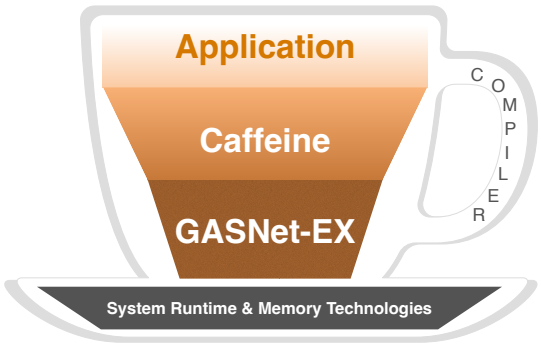

Fig. 2. Caffeine system stack

II. Methodology: Structure and Workflow

A. System Stack

Fig. 2 depicts our envisioned system stack, in which an application developer writes standard Fortran, including some Coarray Fortran syntax. The compiler translates that syntax into Caffeine [22] procedure invocations and data structures, which Caffeine supports using GASNet-EX procedures and data structures. GASNet-EX supports Caffeine's procedure invocations using lower-level, platform-specific communication protocols. For demonstration purposes, the first author has integrated direct references to Caffeine procedures into an application, Matcha [23], using the Fortran `use` statement's renaming capability to replace standard Fortran syntax with Caffeine procedure invocations. Such direct references can be removed when using a compiler that has adopted Caffeine as its parallel runtime library.

B. GASNet-EX

GASNet-EX [24] is a language-independent, networking middleware layer that provides network-independent, high-performance communication primitives for HPC, including one-sided RMA and Active Messages. Unlike the dominant Message Passing Interface (MPI) communication standard, the GASNet-EX interface and implementation are designed specifically to meet the needs of alternative programming

TABLE 1
Fortran standard terms and definitions (adapted from [19])

Term	Definition
assumed -type	unlimited polymorphic data object declared with `type(*)`, described informally here as "type-agnostic"
assumed -rank	data-object dummy argument that assumes the rank of its effective argument, described here as "rank-agnostic"
coarray	data structure partitioned across a team's images and accessible by each image in the corresponding team
effective argument	entity that is argument-associated with a dummy argument in a procedure call
image	instance of a Fortran program
intrinsic	entity or operation defined in the Fortran standard and accessible without further definition or specification
rank	number of array dimensions of a data entity (zero for a scalar entity)
team	ordered image set created by executing a `form team` statement, or the initial ordered set of all images
unlimited polymorphic	able to have any dynamic type during program execution

models on emerging exascale systems. GASNet-EX is implemented directly over the native/proprietary APIs of many networks, including all of those in use at the HPC centers of the U. S. Department of Energy's Office of Science [25]. GASNet-EX's interface is primarily intended as a compilation target and for use by runtime library writers (as opposed to domain scientists), and the primary goals are high performance, interface portability, and expressiveness. GASNet-EX provides communication services for many projects, including both programming models and other parallel libraries and frameworks. Examples of alternative HPC programming models using GASNet-EX include: UPC++ [26–28], the Legion programming system [29], HPE's Chapel language [30], the Omni Xcalable Compiler [31], and many UPC [32–34] and CAF/Fortran [22, 35–37] compiler runtimes. GASNet-EX has also been adopted for communication services by a number of parallel libraries and frameworks, including [38, 39]. See [40] for full details on current client software, and Fig. 3 for an overview of the GASNet-EX software ecosystem.

GASNet-EX's long-established track record of providing robust and portable communication services to a wide range alternative programming models, including several parallel Fortran implementations, made it a natural choice as the initial communication backend for Caffeine. In addition, GASNet-EX delivers very competitive RMA performance on most network hardware of interest to HPC, often rivaling vendor-provided MPI implementations. Fig. 4 is reproduced with permission from [24] and compares the flood bandwidth performance of GASNet-EX RMA operations with the vendor MPI implementation on the Haswell partition of NERSC's Cray XC40 Cori [41] supercomputer (using a Cray Aries [42, 43] network). The main observation is that GASNet-EX RMA often reaches higher bandwidths at smaller transfer sizes than the equivalent operation in MPI. Similarly for fine-grained/small-payload transfers, GASNet-EX RMA latency performance is routinely over 2x better than MPI [44]. These benefits are attributed primarily to GASNet-EX's lighter-weight interfaces providing a closer semantic match to network hardware capabilities. See [24, 44] for detailed methodology and additional, qualitatively comparable results on other supercomputers of interest. GASNet-EX's proven ability to deliver very efficient one-sided RMA will enable Caffeine to deploy very efficient coarray access operations.

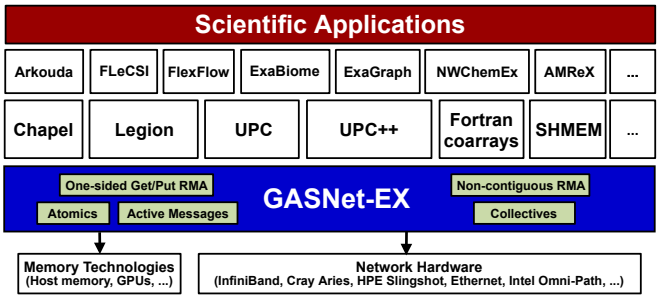

Fig. 3. GASNet-EX software ecosystem

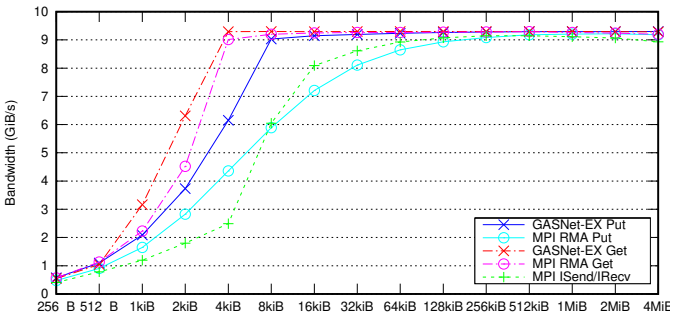

Fig. 4. Microbenchmark performance comparison of GASNet-EX RMA on NERSC Cori, versus Cray MPI's RMA and message-passing (from [24])

C. Caffeine

1) A Sustainable Workflow: Four actively developed compilers support parallel Fortran 2018 features: the HPE Cray, Intel, GNU Compiler Collection (GCC), and Numerical Algorithms Group (NAG) compilers. These compilers have each been under development for roughly 20-40 years concurrently with the publication of several Fortran standards. Hence, the compilers have progressed through the standards chronologically. The dormant g95 project [45] followed a more nonlinear development timeline by layering parallel features from the Fortran 2008 standard [46] atop a Fortran 95 compiler that only partially supported the Fortran 2003 standard [47].

The choice between a chronological or a nonlinear development timeline harbors several subtleties at the intersection of history, technology, and sociology. Historically, coarray features formally entered the language in the 2008 standard, but Coarray Fortran [48] was initially developed as an extension to Fortran 95 and thus has very little explicit interaction with Fortran 2003 features. This makes it feasible to leapfrog the 2003 standard and support 2008 coarrays without impacting ongoing 2003 development in any significant way. Technologically, the chronological approach defers one of the most compelling HPC language properties, parallelism, until later in the timeline than is necessary. Sociologically, the chronological approach usually leads to writing the parallel runtime library in the compiler's primary implementation language, typically C/C++, which raises the barrier to user contributions because the users are Fortran programmers and not necessarily C/C++ programmers. In an open-source project, lowering barriers to user contributions improves a software project's long-term sustainability.

The key insight that inspired Caffeine follows:

Insight 1. A subset of Fortran's *non-parallel* 2003, 2008, and 2018 features collectively provide a compelling platform for writing a runtime library, mostly in Fortran, that supports the *parallel* features of Fortran 2008 and 2018.

Fig. 5 details the proposed compiler development timeline. In the envisioned scenario, early in implementing the 2003 standard, a Fortran 95 compiler's developers would prioritize certain 2003 C-interoperability features that are used in Caffeine. The compiler developers would then leapfrog the 2008

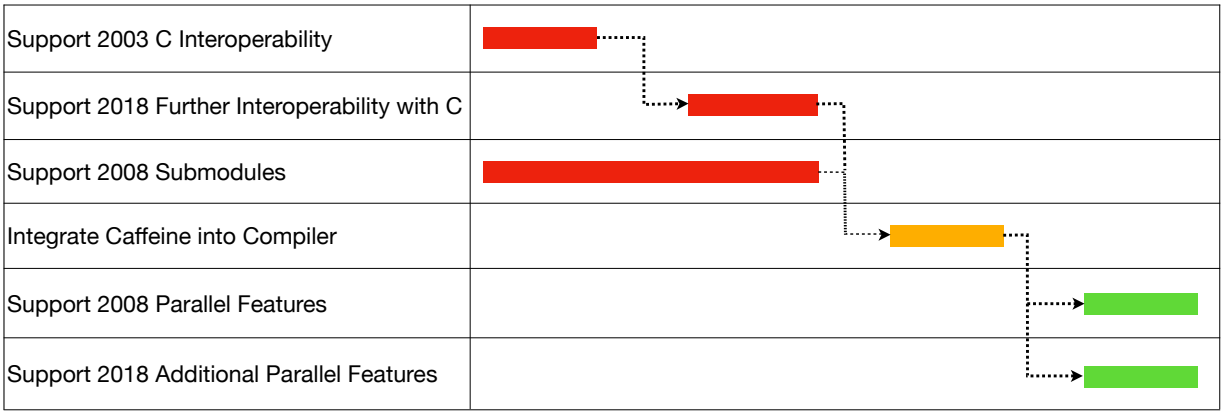

Fig. 5. A possible Coarray Fortran support Gantt chart color-coded for compiler prerequisites for building Caffeine (red), Caffeine integration process (yellow), and resulting 2008/2018 parallel feature availability (green)

standard to implement certain 2018 C-interoperability features used in Caffeine. Along the way, if the compiler developers can implement 2008 submodules, then such a compiler could compile Caffeine in its current form. If necessary, however, it would be straightforward to refactor Caffeine to remove submodules, which are used to facilitate modest reductions in compile time.

The result of integrating Caffeine into a compiler with the described workflow could be early support for all 2008 and 2018 parallel programming features *before* even reaching full 2003 compliance. Such an outcome could positively influence parallel programming practice and performance as well as prospects for user adoption and the likelihood of user contributions.

Writing a parallel runtime library in Fortran has the additional portability benefit of facilitating use by *any* Fortran compiler, which further broadens the potential community of adopters and contributors. This contrasts starkly with every parallel runtime library developed for Fortran compilers to date. Each of the aforementioned commercial compiler vendors developed one or more proprietary runtimes that are usable by only the corresponding commercial compiler. The one non-commercial, parallel Fortran compiler, GCC `gfortran`, uses the OpenCoarrays [37] parallel runtime library developed by the first author and his collaborators. Although open-source and therefore hypothetically usable by other compilers, 14 of 37 OpenCoarrays procedures require a `gfortran`-specific data descriptor as one argument. Section III-B details how Proposition 1 obviates the need for such compiler specificity.

III. Discussion of Results

A. Status of parallel feature support

Tbl. 2 details the status of Caffeine's support for Fortran's parallel features. The first column denotes the standard version in which a feature first appeared. In some cases, a feature introduced in 2008 has expanded capabilities in 2018. For example, in 2008, the intrinsic function `num_images` accepted no arguments, but in 2018, it accepts an optional argument of type `team_type`, an intrinsic derived type that entered the language in the 2018 standard. Our `num_images` intrinsic function support is currently partial because we have not yet added support for `team_type`.

B. Compiler-facing interface

Fig. 6 depicts the interface body for the Caffeine subroutine supporting Fortran's `co_sum` collective subroutine. One salient feature of this interface body was the second key insight that inspired Caffeine's design:

Insight 2. The use of Fortran 2018 assumed-type (`type(*)`) and assumed-rank (e.g., `a(..)`) dummy arguments obviates the need for passing compiler-specific data descriptors to the runtime library.

Assumed-type, assumed-rank procedure dummy arguments are part of the 2018 standard's expansion of the 2003 standard's C-interoperability features. These 2018 features add type- and rank-agnostic capabilities to the language. Their introduction was largely inspired by a desire to facilitate a more modern Fortran interface to implementations of the MPI standard [49] through MPI's unfortunately misnamed `mpi_f08` module. Assumed-type entities are unlimited polymorphic (i.e. type-agnostic) data objects, so the effective argument passed in the procedure call may have any type. When a Fortran 2018 compiler generates code for invoking a procedure via an interface body with an assumed-type dummy argument, what gets passed is a Fortran-defined, C-language structure,

```
module subroutine caf_co_sum(a, result_image, stat, errmsg)
  implicit none
  type(*), intent(inout), contiguous, target :: a(..)
  integer, intent(in), target, optional :: result_image
  integer, intent(out), target, optional :: stat
  character(len=*), intent(inout), target, optional :: errmsg
end subroutine
```

Fig. 6. A sample interface body for `co_sum`.

TABLE 2
STATUS OF CAFFEINE'S SUPPORT FOR THE PARALLEL FEATURES OF FORTRAN 2008 AND 2018.

Standard	Feature	Status
2008	Program launch	yes
2008	Normal termination: `stop` and `end program` statements	yes
2008	Error termination: `error stop` statement	yes
2008	Image enumeration: `this_image` and `num_images` intrinsic functions	partial
2008	Synchronization: `sync {all,images,memory,team}` statements	partial
2008	Coarrays: declaration, access, (de)allocation, inquiry functions	no
2008	Critical construct: `critical` and `end critical`	no
2008	Atomics: `atomic_{int,logical}_kind` kind parameters and `atomic_{define,ref,...}` subroutines	no
2008	Locks: `lock` and `unlock` constructs	no
2018	Collective subroutines: `co_{broadcast,sum,min,max,reduce}`	yes
2018	Events: `event_type` intrinsic type, `event_query` subroutine and `event {post,wait}` statements	no
2018	Teams: `team_type` intrinsic type and `{form,change,end} team` statements	no
2018	Failed/stopped images: `fail image` statement, `{failed,stopped}_image` intrinsic functions, related constants	no

`CFI_cdesc_t`, that serves as a descriptor of the effective argument, including its base address, storage size, type and other information. The ability to ensure that information gets passed into the runtime in a compiler-independent, standard-conforming way liberates the runtime from being "hard-wired" to one compiler. The fact the enabling mechanism is C-interoperable is a bonus, given that the communication software employed by any parallel runtime library is likely to be C/C++ code such as GASNet-EX or an MPI implementation.

Fortran 2003 provided a different form of unlimited polymorphic declaration, `class(*)`, that we employed in the early stages of writing Caffeine. However, passing such an argument to C necessitated first inserting type-guarding blocks of the form `select type(a)` followed by a proverbial combinatorial explosion of type-selection branches of the form `type is(real)`, `type is(integer)`, etc., each of which must precede a rank-guarding block `select rank` containing a list of all supported ranks such as `rank is(0)` for scalars, `rank is(1)` for one-dimensional arrays, etc. Switching to 2018-style assumed-type, assumed-rank dummy arguments led to an order-of-magnitude reduction in code complexity with several files shrinking from over 400 lines to approximately 40 lines. This observation yielded the third key insight that enabled Caffeine:

Insight 3. The use of Fortran 2018 assumed-type, assumed-rank dummy arguments offers considerable complexity reduction relative to using the Fortran 2003 C-interoperability feature set.

Such complexity reduction implies more concise and maintainable code. It also likely yields increased code robustness: earlier attempts to use `class(*)` instead of `type(*)` yielded massively replicated code, which is harder to manually audit for correctness.

A final important feature of Caffeine is that the compiler-facing interface makes no reference to a specific communication substrate, leaving the option to swap the communication backend between GASNet-EX, MPI, or OpenSHMEM [50], for example. The choice of which backend to use could even be delayed until link-time so that a parallel Fortran software package could be compiled into object files and subsequently linked to either of the aforementioned communication libraries without rewriting or even recompiling a single line of Fortran source code. If this plan comes to fruition, then we envision delivering on the one of the original aims of OpenCoarrays [37], which developed experimental GASNet and OpenSHMEM options before settling on MPI as the only maintained solution.

C. Unit tests

The Caffeine test suite uses the open-source Veggies [51] unit testing framework and its companion program Cart [52]. Given a test suite comprised of test files, `test/*_test.f90` containing modules that use Veggies derived types and functions, Cart writes a driver program that executes the entire test suite. Because the construction of the driver is automated, there is no need to consider its internal details in presenting the test suite. Each file with a name of the form `*_test.f90` tests one Caffeine feature. For example, `caf_co_sum_test.f90` verifies the behavior of the `co_sum` collective subroutine.

Fig. 7 shows the beginning of the `co_sum` test module. At line 10, the `caf_co_sum_test` module's first function, `test_caf_co_sum`, produces a `tests` object containing the results of each of the module's tests. Lines 13–24 define the `tests` object by invoking the Veggies `describe` function with two arguments:

1) a string, "The `caf_co_sum` subroutine", describing what is being tested, and
2) an array constructor with elements intended to be read as sentences detailing what `it` (the subroutine being tested) must do to satisfy the tests.

For example, "it sums default integer scalars with no optional arguments present." Following each such description is the name of the function that performs the described check: `sums_default_integer_scalars` in this case. Lines 27-34 define that function. In this manner, the Caffeine test suite reads as a hybrid natural-language/formal specification of the required behavior. The natural language aspects derive from the juxtaposition of Veggies procedure names with re-

```
module caf_co_sum_test
    use caffeine_m, only : caf_co_sum, caf_num_images, caf_this_image
    use vegetables, only: result_t, test_item_t, assert_equals, describe, it, assert_that, assert_equals, succeed

    implicit none
    private
    public :: test_caf_co_sum

contains
    function test_caf_co_sum() result(tests)
        type(test_item_t) tests

        tests = describe( &
          "The caf_co_sum subroutine", &
          [ it("sums default integer scalars with no optional arguments present", sum_default_integer_scalars) &
           ,it("sums default integer scalars with all arguments present", sum_integers_all_arguments) &
           ,it("sums integer(c_int64_t) scalars with stat argument present", sum_c_int64_scalars) &
           ,it("sums default integer 1D arrays with no optional arguments present", sum_default_integer_1D_array) &
           ,it("sums default integer 15D arrays with stat argument present", sum_default_integer_15D_array) &
           ,it("sums default real scalars with result_image argument present", sum_default_real_scalars) &
           ,it("sums double precision 2D arrays with no optional arguments present", sum_double_precision_2D_array) &
           ,it("sums default complex scalars with stat argument present", sum_default_complex_scalars) &
           ,it("sums double precision 1D complex arrays with no optional arguments present", sum_dble_complex_1D_arrays) &
          ])
    end function

    function sum_default_integer_scalars() result(result_)
        type(result_t) result_
        integer i

        i = 1
        call caf_co_sum(i)
        result_ = assert_equals(caf_num_images(), i)
    end function
```

Fig. 7. A partial listing of the `co_sum` unit test, from `test/caf_co_sum_test.f90` in the Caffeine repository.

lated strings. The formal aspect comes from expressing the specification in a language with a standard grammar. When compiled and executed, the specification also serves as the verification system.

IV. Future Work

Although the project began only recently, Caffeine already implements many of the parallel Fortran features (shown in Tbl. 2). However several important features remain to reach Caffeine's goals of providing a complete set of parallel features for use with a serial Fortran compiler.

The most significant remaining work for Caffeine is supporting coarray allocation and access. We expect to follow the general coarray design used in the OpenUH Coarray Fortran compiler [36], which was implemented using the GASNet-1 [53] communication layer (the predecessor to GASNet-EX). We expect to directly utilize GASNet-EX's powerful and efficient RMA capabilities to implement coarray access operations as a lightweight pass-through. The trickiest part is designing coarray memory management to provide the heap symmetry and team-entangled semantics required by the Fortran standard. Ge [54] describes the algorithm used to provide standard-compliant coarray memory management in the OpenUH runtime. We expect to implement a similar algorithm in Caffeine, with some simplifications enabled by the robust support for subset teams and team-aware hierarchical collective operations introduced in GASNet-EX.

Another notable remaining feature is Fortran image teams (`team_type` and associated functions and statements), which should naturally map to a combination of GASNet-EX teams and some Caffeine logic and local metadata used to implement Fortran-specific semantics in `form team`, `change team` and coarray management.

Atomics (`atomic_{int,logical}_kind` and `atomic_*` subroutines) should map naturally to GASNet-EX remote atomic memory operations, which automatically leverage any available NIC offload capabilities to accelerate these operations. We also expect that events (`event_type`, the `event post` and `event wait` statements, and the `event_query` subroutine), locks (the `lock` and `unlock` statements), `critical` blocks and `sync images` will be straightforward to implement using GASNet-EX Active Messages and traditional distributed synchronization algorithms.

V. Conclusions

Fortran remains relevant through its impact on daily life and its common use in HPC. Due to the importance of parallelism in HPC, Berkeley Lab's Flang testing project focuses on advancing frontend support for modern Fortran's parallel features. Our current work in Flang is timely because it begins the transition to supporting serial Fortran features beyond the 95 standard.

Modern serial Fortran provides a viable path for writing a parallel runtime library that supports Fortran's parallel features. Writing a parallel runtime library primarily in Fortran offers several key benefits, including software sustainability through lowering the barrier to user contributions, reduced complexity through a type- and rank-agnostic compiler-facing

interface, and portability across compilers through liberation from compiler-specific data structure descriptors.

We detailed the current status of parallel Fortran-standard feature support in Caffeine. We gave an overview of the Caffeine system stack, including the expected performance benefits associated with choosing GASNet-EX as the initial communication substrate. We also gave a description of Caffeine's test suite as a compilable specification that describes and verifies Caffeine's intended behaviors. Lastly, we presented our plans for future work to complete Caffeine's support for Fortran's parallel feature set.

Acknowledgments

This research was 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, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Acronyms

GCC GNU Compiler Collection

HPC High-Performance Computing

MPI Message Passing Interface

NAG Numerical Algorithms Group

NERSC National Energy Research Scientific Computing Center

PGAS Partitioned Global Address Space

RMA Remote Memory Access

SPMD Single-Program, Multiple-Data

TDD Test-Driven Development

References

[1] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern *et al.*, "The FORTRAN automatic coding system," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, 1957, pp. 188–198, doi:10.1145/1455567.1455599.

[2] D. Cann, "Retire Fortran? a debate rekindled," *Communications of the ACM*, vol. 35, no. 8, pp. 81–89, 1992, doi:10.1145/135226.135231.

[3] E. W. Dijkstra, "How do we tell truths that might hurt?" *ACM Sigplan Notices*, vol. 17, no. 5, pp. 13–15, 1982, doi:10.1145/947923.947924.

[4] *National Energy Research Scientific Computing Center (NERSC)*, https://www.nersc.gov.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] L. J. Kedward, B. Aradi, O. Čertík, M. Curcic, S. Ehlert, P. Engel, R. Goswami, M. Hirsch, A. Lozada-Blanco, V. Magnin *et al.*, "The state of Fortran," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 63–72, 2022, doi:10.1109/MCSE.2022.3159862.

[12] *Fortran Package Manager (fpm)*, https://github.com/fortran-lang/fpm.

[13] *LFortran*, https://lfortran.org.

[14] *LLVM Compiler Infrastructure project*, https://github.com/llvm/llvm-project.

[15] Intel Corporation, *Porting Guide for ifort Users to ifx*, https://www.intel.com/content/www/us/en/developer/articles/guide/porting-guide-for-ifort-to-ifx.html.

[16] IBM Developer Blog, *IBM C/C++ and Fortran compilers to adopt LLVM open source infrastructure*, https://developer.ibm.com/blogs/c-and-fortran-adopt-llvm-open-source/.

[17] *Classic Flang*, https://github.com/flang-compiler/flang.

[18] *Experimental Flang support for executable generation*, https://go.lbl.gov/flang-new-experimental.

[19] 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.

[20] Lawrence Berkeley National Lab, *Flang Testing project*, https://go.lbl.gov/flang-testing.

[21] 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.

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

[23] *Motility Analysis of T-Cell Histories in Activation (Matcha)*, https://go.lbl.gov/matcha.

[24] 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.

[25] *DOE Advanced Scientific Computing Research (ASCR) Facilities*, https://science.energy.gov/ascr/facilities.

[26] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "UPC++: A High-Performance Communication Framework for Asynchronous Computation," in *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS)*, 2019, doi:10.25344/S4V88H.

[27] D. Bonachea and A. Kamil, "UPC++ v1.0 Specification, Revision 2022.3.0," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001452, March 2022, doi:10.25344/S4530J.

[28] J. Bachan, S. B. Baden, D. Bonachea, M. Grossman, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, B. van Straalen, and D. Waters, "UPC++ v1.0 Programmer's Guide, Revision 2022.3.0," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001453, March 2022, doi:10.25344/S41C7Q.

[29] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, 2012, doi:10.1109/SC.2012.71.

[30] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," in *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 21, no. 3, August 2007, pp. 291–312, doi:10.1177/1094342007078442.

[31] H. Murai, M. Nakao, H. Iwashita, and M. Sato, "Preliminary Performance Evaluation of Coarray-based Implementation of Fiber Miniapp Suite Using XcalableMP PGAS Language," in *Proceedings of the Second Annual PGAS Applications Workshop (PAW'17)*, 2017, doi:10.1145/3144779.3144780.

[32] 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.

[33] *GCC/UPC Compiler*, Intrepid Technology, Inc., https://github.com/Intrepid/GUPC.

[34] *Clang UPC Compiler*, http://clangupc.github.io.

[35] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey, "A Multi-platform Co-Array Fortran Compiler," in *Parallel Architecture and Compilation Techniques (PACT)*, 2004, doi:10.1109/PACT.2004.1342539.

[36] D. Eachempati, H. J. Jun, and B. Chapman, "An Open-source Compiler and Runtime Implementation for Coarray Fortran," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Models (PGAS'10)*, 2010, doi:10.1145/2020373.2020386.

[37] 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.

[38] B. Brock, A. Buluç, and K. A. Yelick, "BCL: A cross-platform distributed container library," *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, 2019, doi:10.1145/3337821.3337912.

[39] C. Chan, B. Wang, J. Bachan, and J. Macfarlane, "Mobiliti: Scalable Transportation Simulation Using High-Performance Parallel Computing," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, Nov 2018, doi:10.1109/ITSC.2018.8569397.

[40] *GASNet*, http://gasnet.lbl.gov.

[41] NERSC, "Cori Haswell Nodes," doi:10.25344/S4859K.

[42] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC Series Network," Cray Inc., White Paper WP-Aries01-1112, November 2012, doi:10.25344/S4RW2H.

[43] P. H. Hargrove and D. Bonachea, "GASNet-EX performance improvements due to specialization for the Cray Aries network," in *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, November 2018, pp. 23–33, doi:10.25344/S44S38.

[44] P. H. Hargrove and D. Bonachea, "GASNet-EX RMA Communication Performance on Recent Supercomputing Systems," in *2022 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, November 2022, doi:https://doi.org/10.25344/S40C7D.

[45] *The G95 Fortran compiler*, http://g95.sourceforge.net.

[46] 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.

[47] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:2004*. International Organization for Standardization (ISO), Nov 2004, https://www.iso.org/standard/39691.html.

[48] 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.

[49] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[50] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, doi:10.1145/2020373.2020375.

[51] *Veggies Fortran unit testing framework*, https://www.archaeologic.codes/veggies.

[52] *Cart test suite tool*, https://www.archaeologic.codes/cart-repo.

[53] 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.

[54] S. Ge, "Implementation and evaluation of additional parallel features of Coarray Fortran," Master's thesis, University of Houston, May 2016, http://hdl.handle.net/10657/3268.