Lawrence Berkeley National Laboratory

LBL Publications

Title

Agile Acceleration of LLVM Flang Support for Fortran 2018 Parallel Programming

Permalink

https://escholarship.org/uc/item/01h204x9

Authors

Rasmussen, Katherine
Rouson, Damian
George, Najé
et al.

Publication Date

2022-11-15

DOI

10.25344/S4CP4S

Peer reviewed

eScholarship.org Powered by the California Digital Library University of California

Agile Acceleration of LLVM Flang Support for Fortran 2018 Parallel Programming

Katherine Rasmussen, Damian Rouson,
Dan Bonachea, Brian Friesen, Hussain Kadhem
Lawrence Berkeley National Laboratory, USA
{krasmussen,rouson,dobonachea,bfriesen,hmk}@lbl.gov

Najé George
Computational Science Research Center
San Diego State University, USA
ngeorge8641@sdsu.edu

***Abstract*—The LLVM Flang compiler (”Flang”) is currently Fortran 95 compliant, and the frontend can parse Fortran 2018. However, Flang does not have a comprehensive 2018 test suite and does not fully implement the static semantics of the 2018 standard. We are investigating whether agile software development techniques, such as pair programming and test-driven development (TDD), can help Flang to rapidly progress to Fortran 2018 compliance. Because of the paramount importance of parallelism in high-performance computing, we are focusing on Fortran's parallel features, commonly denoted "Coarray Fortran". We are developing what we believe are the first comprehensive, open-source tests for the static semantics of Fortran 2018 parallel features, and contributing them to the LLVM project. A related effort involves writing runtime tests for parallel 2018 features and supporting those tests by developing a new parallel runtime library: the CoArray Fortran Framework of Efficient Interfaces to Network Environments (Caffeine).**

***Index Terms*—HPC, Exascale Computing, Compiler Testing**

I. Introduction

The LLVM Compiler Infrastructure project [1] provides open-source frontends for several languages, including C and C++. Because of the essential role that Fortran plays in high-performance computing, the U.S. Department of Energy (DOE) has led a multi-year effort, through the Exascale Computing Project (ECP), to develop a Fortran frontend for LLVM [2] named "Flang". At the time of writing, over 150 developers have authored 4895 commits to the Flang subdirectory on the main branch of the LLVM repository [1] (hereafter "upstream"); that work has brought Flang up to Fortran 95 [3] compliance. In addition to full support for Fortran 95, Flang can parse Fortran 2018 [4]. However, the Flang test suite does not exhaustively check Fortran standards-compliance, and the authors are unaware of a publicly available, comprehensive Fortran 2018 compliance test suite. Moreover, Fortran's evolution through three subsequent standards, commonly denoted Fortran 2003, 2008, and 2018 [4–6], implies significant remaining work to reach up-to-date compliance. This situation makes it attractive to explore ways to accelerate Flang's progression through Fortran language standards.

Berkeley Lab's ECP Flang project [7] focuses on writing static semantic tests for Fortran's parallel features, which are commonly referred to as "Coarray Fortran." We contribute these tests to the main branch of the upstream LLVM repository. We are also writing a comprehensive runtime test suite for parallel features. Fortran 2008 introduced parallel features but Flang lacks complete 2008 support, so we are also developing the Caffeine library to support runtime test execution.

Agile software development [8] encourages early release of working software and subsequent iteration toward complete solutions. To the extent that the concept of agility is associated with deliberate speed, agile practices might provide helpful avenues along which to attempt rapid development. This abstract and our poster focus primarily on the outcomes of approximately one person-year of effort, during which we reached 56% completion of static semantics test coverage. Although a direct comparison of speed with other development approaches would require a great deal more information and analysis, our subjective experience indicates that the agile practices described herein save us development time through frequent, rich developer interactions and through the unambiguous specification of feature requirements in the form of unit tests.

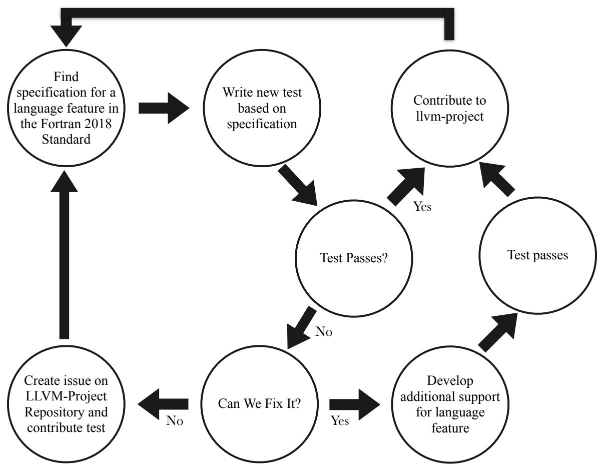

Find specification for a language feature in the Fortran 2018 Standard
Write new test based on specification
Contribute to llvm-project
Test Passes?
Yes
No
Test passes
Create issue on LLVM-Project Repository and contribute test
No
Can We Fix It?
Yes
Develop additional support for language feature

Fig. 1. Agile Test-Driven Development

II. Methodology

A. Agile Practices

The agile technique that we believe to be novel in its application to an open-source Fortran compiler is test-driven development (TDD) [9]. Our approach is illustrated in Fig. 1.

TDD starts with writing software tests in lieu of other forms of requirements documents and specifications. Given the resulting tests, developers add features to the subject software specifically to, and only to, support the tests.

Another agile practice we employ is pair programming: interactive sessions in which coders receive immediate feedback from a live observer. Pair programming keeps team members abreast of work progression in real time and enables synchronous communication of useful input. Our experience indicates that two minds are better than one. For example, we find and fix mistakes more quickly when one member focuses on writing the test while the other focuses on critiquing and contextualizing it within the broader aims of the project.

Once a new test is ready for wider dissemination and additional feedback, we post it to the LLVM community's code-review tool, Phabricator, which also records the developer dialogue and any resulting code updates. Throughout this process, we leverage an agile practice embraced by the wider LLVM community: continuous integration (CI) testing. LLVM's CI infrastructure uses automation to verify that contributed changes do not break pre-existing code.

B. Static Semantic Tests

We register a static semantic test for each parallel feature in a GitHub issue associated with a GitHub project [10] that captures the test's status. For each feature, such as the intrinsic function `num_images`, we write a test program containing a comprehensive set of standard-conforming and non-conforming statements exercising the given feature. Constructing each program requires consulting the formal definition of the feature in the Fortran 2018 standard [4]. The list of conforming invocations of `num_images`, for example, covers each of the function signatures defined in the 2018 standard: `num_images()`, `num_images(team)`, and `num_images(team_number)`, where `team` is of intrinsic type `team_type` and `team_number` is an integer. For each allowable form of invocation, different lines test keyword and non-keyword arguments; different keyword argument ordering; and variable, constant, or literal-constant arguments. We also test each allowable argument type. For example, the collective subroutine `co_sum` accepts a first argument of any numeric type, so we test each of `integer`, `real` and `complex` arguments in separate invocations.

The non-conforming statements include various disallowed statement forms and procedure invocations with arguments rendered invalid by their type, type parameters, array dimensions (rank), (im)mutability, or other attributes. For example, if the standard requires a dummy argument to have the `intent(out)` attribute, we intentionally violate the argument's intent specification by passing a literal constant. We also test procedure invocations with incorrect numbers of arguments, invalid or repeated keyword names, and other static semantics violations. We check constraints the standard requires compilers to report as errors as well as statically verifiable non-conformance that the standard specifies but does not require compilers to diagnose.

We use LLVM's `llvm-lit` testing tool [11], where tests pass if and only if all lines of code marked by an `ERROR` directive generate the error message provided in the directive and no unmarked lines generate an error. The `XFAIL` directive is used to mark a test that is expected to fail.

Our static semantics test suite reveals Flang's current level of parallel feature support. Many of our tests for intrinsic procedures, for example, have revealed that the compiler interpreted some intrinsic procedure references as user-defined procedures with missing interfaces. In such cases, we mark the corresponding tests with `XFAIL`, thus directing attention to features for which additional work will be needed for the test to pass. In other cases, the interface for the intrinsic procedure is already available, but our tests have revealed potential programmer mistakes that can be caught at compile time but are not currently caught. Where feasible, our team has added either the interface for the intrinsic procedures or the additional static semantic analysis.

III. Runtime Work

Because Flang cannot yet produce executable programs from Fortran 2018 source code, we develop runtime tests in a separate repository: Caffeine [12], a compiler-agnostic Fortran 2018 parallel runtime library that executes atop the GASNet-EX [13, 14] networking middleware – see Fig. 2. We are also developing Caffeine itself. We report elsewhere [15] on the design and implementation of Caffeine.

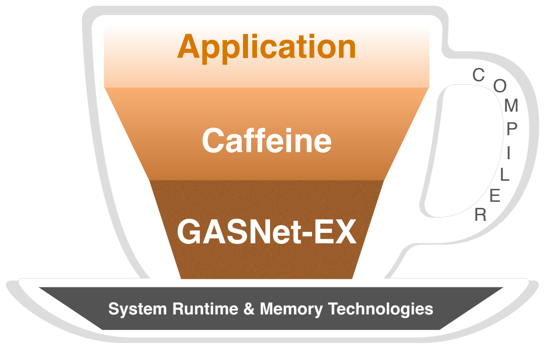


Fig. 2. Caffeine system stack

IV. Outcomes

We have deployed static semantics tests for 32 of 41 parallel statements and intrinsic procedures, including tests for intrinsic functions that support coarrays, collective subroutines, synchronization statements, and event statements. All 32 tests have been pushed to the main branch of the upstream LLVM Compiler Infrastructure project. Additional tests remain in-development or in-review. Based on missing features identified by our tests, the authors have contributed upstream additional static semantic analysis and error checking for 11 parallel features. We have also developed 44 runtime tests that we exercise by developing Caffeine.

Acknowledgment

This research was supported in part by the Sustainable Research Pathways for High Performance Computing (SRP-HPC), a project of the Sustainable Horizons Institute. 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.

References

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

[2] *Flang Project in the Exascale Computing Project*, https://www.exascaleproject.org/research-project/flang/.

[3] Fortran Standards Committee JTC1/SC22/WG5, *Information technology — Programming languages — Fortran, ISO/IEC 1539-1:1997*. International Organization for Standardization (ISO), Dec 1997, https://www.iso.org/standard/26933.html.

[4] 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.

[5] 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.

[6] 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.

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

[8] M. Beedle and et al., *Manifesto for Agile Software Development*, https://agilemanifesto.org/.

[9] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] Lawrence Berkeley National Lab, *Semantics Tests for Parallel Features in LLVM Flang*, https://github.com/BerkeleyLab/flang-testing-project/projects/1.

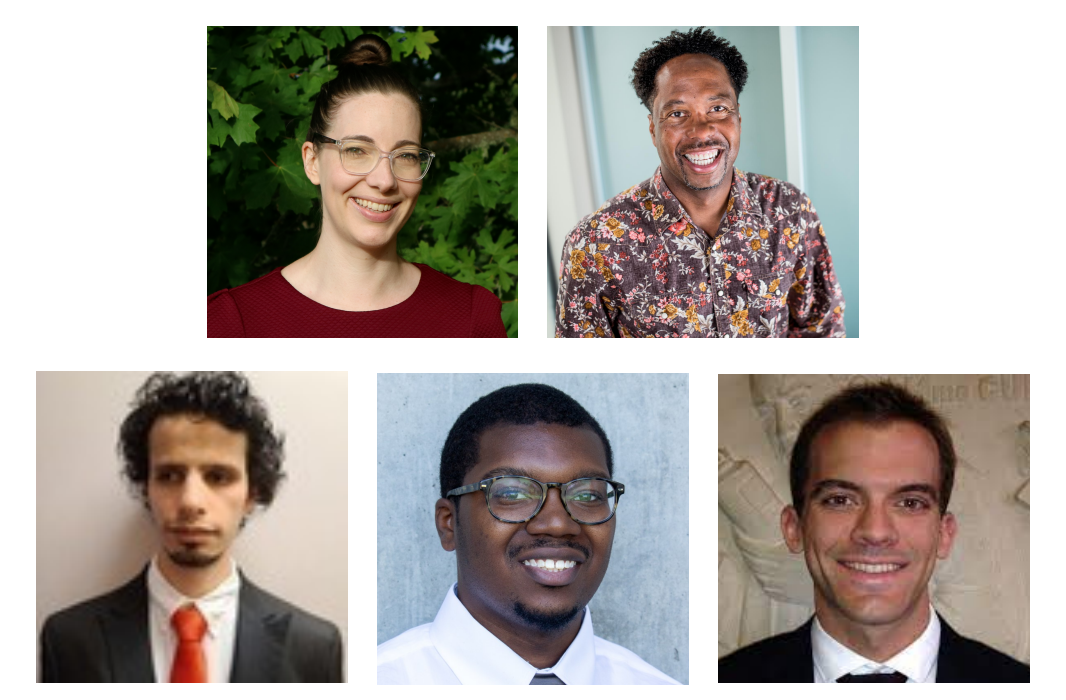
[11] *lit - LLVM Integrated Tester*, https://llvm.org/docs/CommandGuide/lit.html.

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

[13] 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.

[14] *GASNet*, https://gasnet.lbl.gov.

[15] 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.



Video Walkthrough

go.lbl.gov/sc22-flang-testing

Agile Acceleration of LLVM Flang Support for Fortran 2018 Parallel Programming

Katherine Rasmussen[1], Damian Rouson[1], Najé George[2], Dan Bonachea[1], Hussain Kadhem[1], Brian Friesen[1]
[1]Lawrence Berkeley National Laboratory, [2]San Diego State University

BERKELEY LAB

SAN DIEGO STATE UNIVERSITY

Introduction

Problem

LLVM's Flang Fortran compiler is currently Fortran 95 compliant, and the frontend can parse Fortran 2018. However, Flang does not have a comprehensive 2018 test suite and does not fully implement the static semantics of the 2018 standard.

Solution

Agile software encourages early delivery of working software subject to continual improvement. We are investigating whether agile techniques centered around pair programming and test-driven development (TDD) can help Flang to rapidly progress to Fortran 2018 compliance. Because of the paramount importance of parallelism in high-performance computing, we are focusing on Fortran's parallel features, commonly denoted "Coarray Fortran." We are developing what we believe are the first comprehensive, open-source tests for Fortran 2018 parallel features. We push our compile-time behavior tests to the main LLVM-Project repository. We push our runtime tests for parallel Fortran features to the repository of the Caffeine parallel runtime library that we are concurrently developing.

Objectives

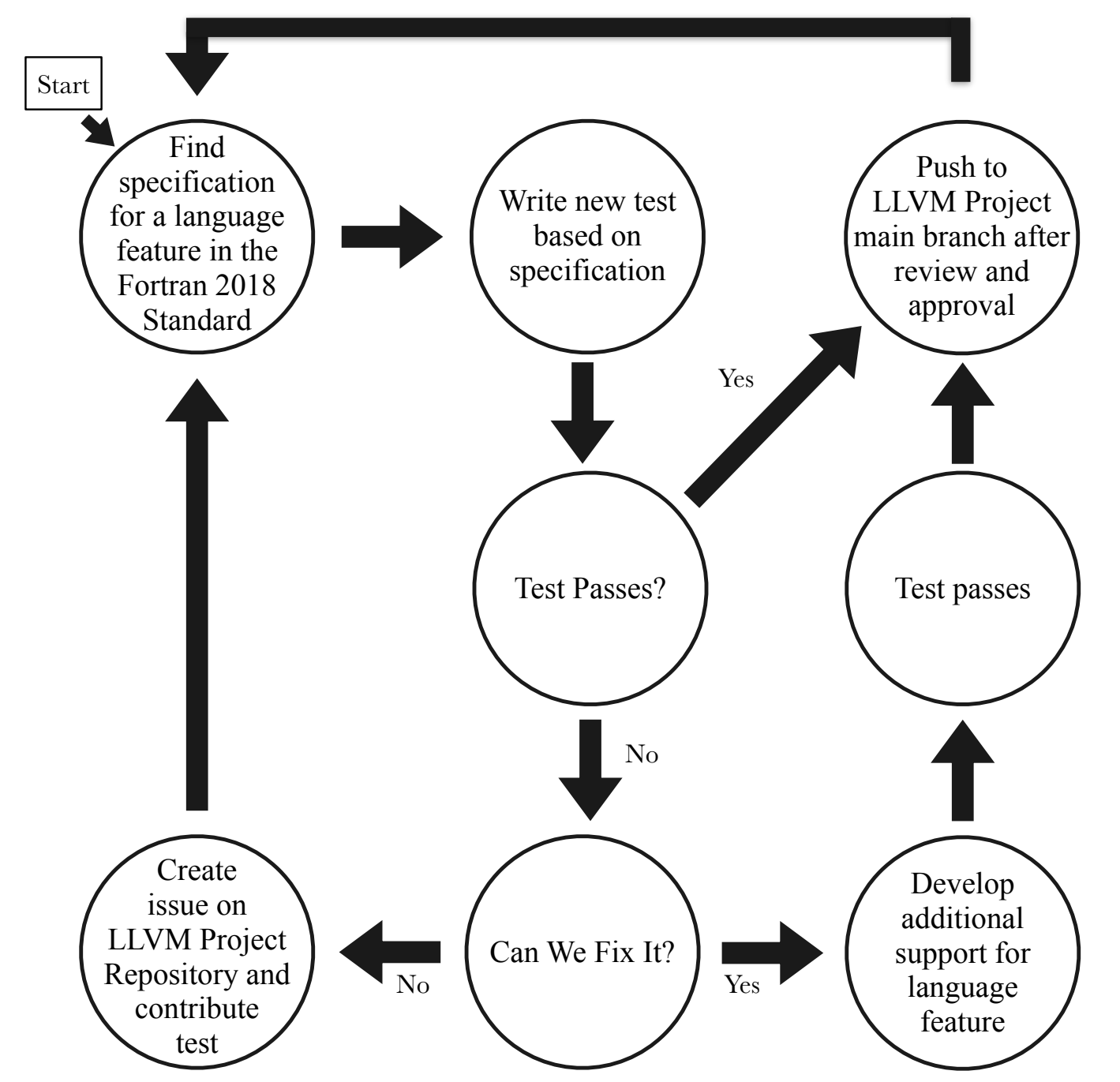
- Exhaustively delineate all of the parallel programming features in Fortran 2018
- Develop semantics tests for LLVM Flang covering statically checkable program errors that the Fortran standard obligates the compiler to detect
- Expand frontend support, including additional error checking, when tests identify missing capabilities

Approach

- Employ agile software development practices
- Test a comprehensive range of standard-conforming and non-conforming Fortran 2018 syntax
- Test-driven development: any contributed tests that fail provide a specification for new features to add to Flang

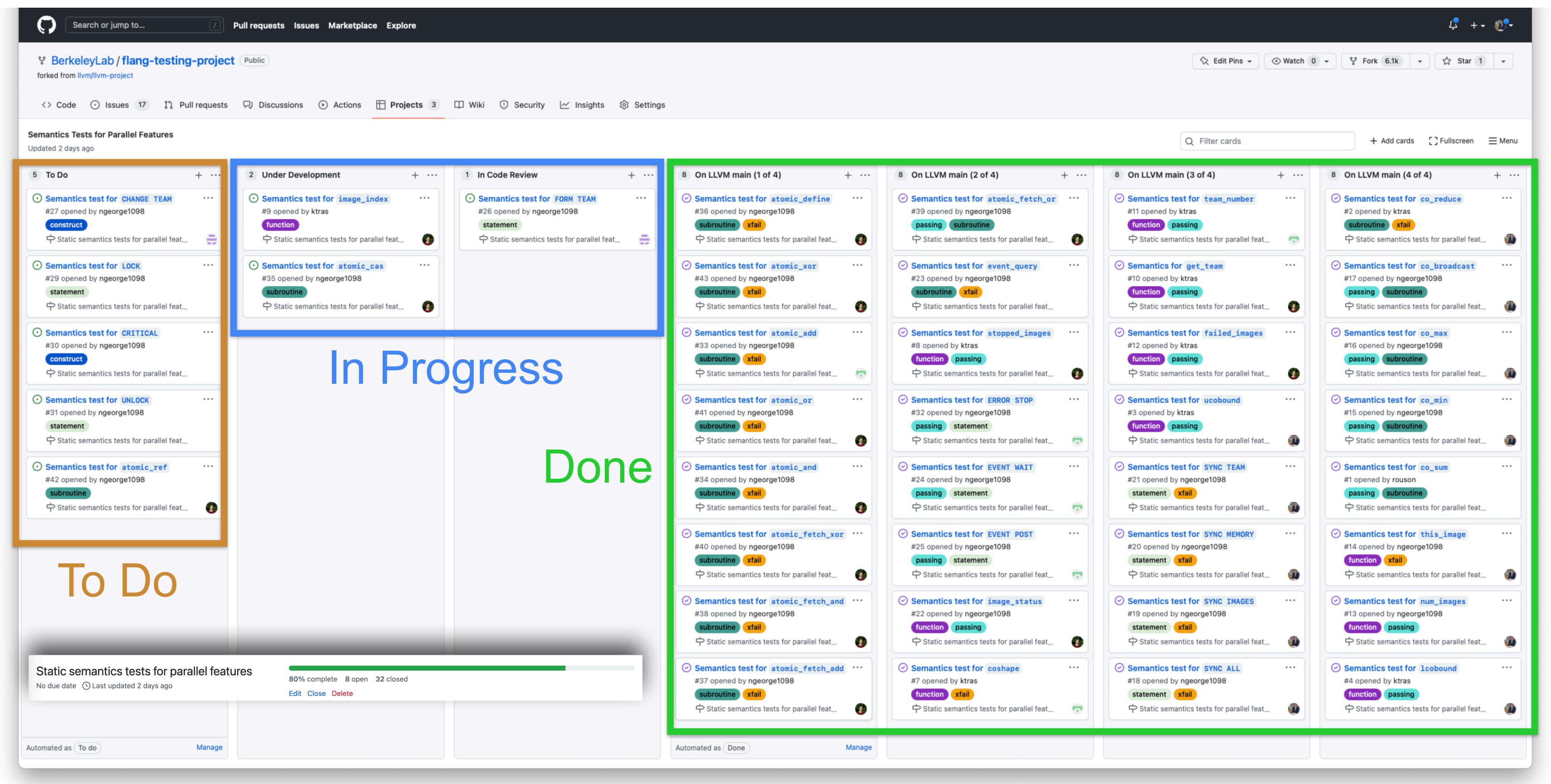
Agile Development

- Test-driven development


Start
Find specification for a language feature in the Fortran 2018 Standard
Write new test based on specification
Push to LLVM Project main branch after review and approval
Yes
Test Passes?
Test passes
No
Create issue on LLVM Project Repository and contribute test
No
Can We Fix It?
Yes
Develop additional support for language feature

- Pair programming sessions
- Valuable team member interactions
- Get feedback early and frequently
- Leverage existing git and Github tools
- Leverage existing agile practices of the LLVM developer community:
  - Use LLVM's continuous integration (CI) test infrastructure to quickly fix CI failures.
  - Code reviews on Phabricator for feedback, edits, and approvals

GitHub Project Board



Search or jump to... Pull requests Issues Marketplace Explore

BerkeleyLab / flang-testing-project Public
forked from llvm/llvm-project

Edit Pins · Watch 0 · Fork 6.1k · Star 1

Code · Issues 17 · Pull requests · Discussions · Actions · Projects 3 · Wiki · Security · Insights · Settings

Semantics Tests for Parallel Features
Updated 2 days ago

Filter cards · Add cards · Fullscreen · Menu

5 To Do
- Semantics test for CHANGE TEAM — #27 opened by ngeorge1098 — construct — Static semantics tests for parallel feat...
- Semantics test for LOCK — #29 opened by ngeorge1098 — statement — Static semantics tests for parallel feat...
- Semantics test for CRITICAL — #30 opened by ngeorge1098 — construct — Static semantics tests for parallel feat...
- Semantics test for UNLOCK — #31 opened by ngeorge1098 — statement — Static semantics tests for parallel feat...
- Semantics test for atomic_ref — #42 opened by ngeorge1098 — subroutine — Static semantics tests for parallel feat...

Automated as To do · Manage

2 Under Development
- Semantics test for image_index — #9 opened by ktras — function — Static semantics tests for parallel feat...
- Semantics test for atomic_cas — #35 opened by ngeorge1098 — subroutine — Static semantics tests for parallel feat...

1 In Code Review
- Semantics test for FORM TEAM — #26 opened by ngeorge1098 — statement — Static semantics tests for parallel feat...

8 On LLVM main (1 of 4)
- Semantics test for atomic_define — #36 opened by ngeorge1098 — subroutine xfail
- Semantics test for atomic_xor — #43 opened by ngeorge1098 — subroutine xfail
- Semantics test for atomic_add — #33 opened by ngeorge1098 — subroutine xfail
- Semantics test for atomic_or — #41 opened by ngeorge1098 — subroutine xfail
- Semantics test for atomic_and — #34 opened by ngeorge1098 — subroutine xfail
- Semantics test for atomic_fetch_xor — #40 opened by ngeorge1098 — subroutine xfail
- Semantics test for atomic_fetch_and — #38 opened by ngeorge1098 — subroutine xfail
- Semantics test for atomic_fetch_add — #37 opened by ngeorge1098 — subroutine xfail

Automated as Done · Manage

8 On LLVM main (2 of 4)
- Semantics test for atomic_fetch_or — #39 opened by ngeorge1098 — passing subroutine
- Semantics test for event_query — #23 opened by ngeorge1098 — subroutine xfail
- Semantics test for stopped_images — #8 opened by ktras — function passing
- Semantics test for ERROR STOP — #32 opened by ngeorge1098 — passing statement
- Semantics test for EVENT WAIT — #24 opened by ngeorge1098 — passing statement
- Semantics test for EVENT POST — #25 opened by ngeorge1098 — passing statement
- Semantics test for image_status — #22 opened by ngeorge1098 — function passing
- Semantics test for coshape — #7 opened by ktras — function xfail

8 On LLVM main (3 of 4)
- Semantics test for team_number — #11 opened by ktras — function passing
- Semantics for get_team — #10 opened by ktras — function passing
- Semantics test for failed_images — #12 opened by ktras — function passing
- Semantics test for ucobound — #3 opened by ktras — function passing
- Semantics test for SYNC TEAM — #21 opened by ngeorge1098 — statement xfail
- Semantics test for SYNC MEMORY — #20 opened by ngeorge1098 — statement xfail
- Semantics test for SYNC IMAGES — #19 opened by ngeorge1098 — statement xfail
- Semantics test for SYNC ALL — #18 opened by ngeorge1098 — statement xfail

8 On LLVM main (4 of 4)
- Semantics test for co_reduce — #2 opened by ktras — subroutine xfail
- Semantics test for co_broadcast — #17 opened by ngeorge1098 — passing subroutine
- Semantics test for co_max — #16 opened by ngeorge1098 — passing subroutine
- Semantics test for co_min — #15 opened by ngeorge1098 — passing subroutine
- Semantics test for co_sum — #1 opened by rouson — passing subroutine
- Semantics test for this_image — #14 opened by ngeorge1098 — function xfail
- Semantics test for num_images — #13 opened by ngeorge1098 — function passing
- Semantics test for lcobound — #4 opened by ktras — function passing

Static semantics tests for parallel features
No due date · Last updated 2 days ago
80% complete · 8 open · 32 closed
Edit Close Delete

Figure 1: Exhaustive list of Fortran 2018 parallel programming features to test
https://go.lbl.gov/flang-testing

Compile-Time Test Coverage

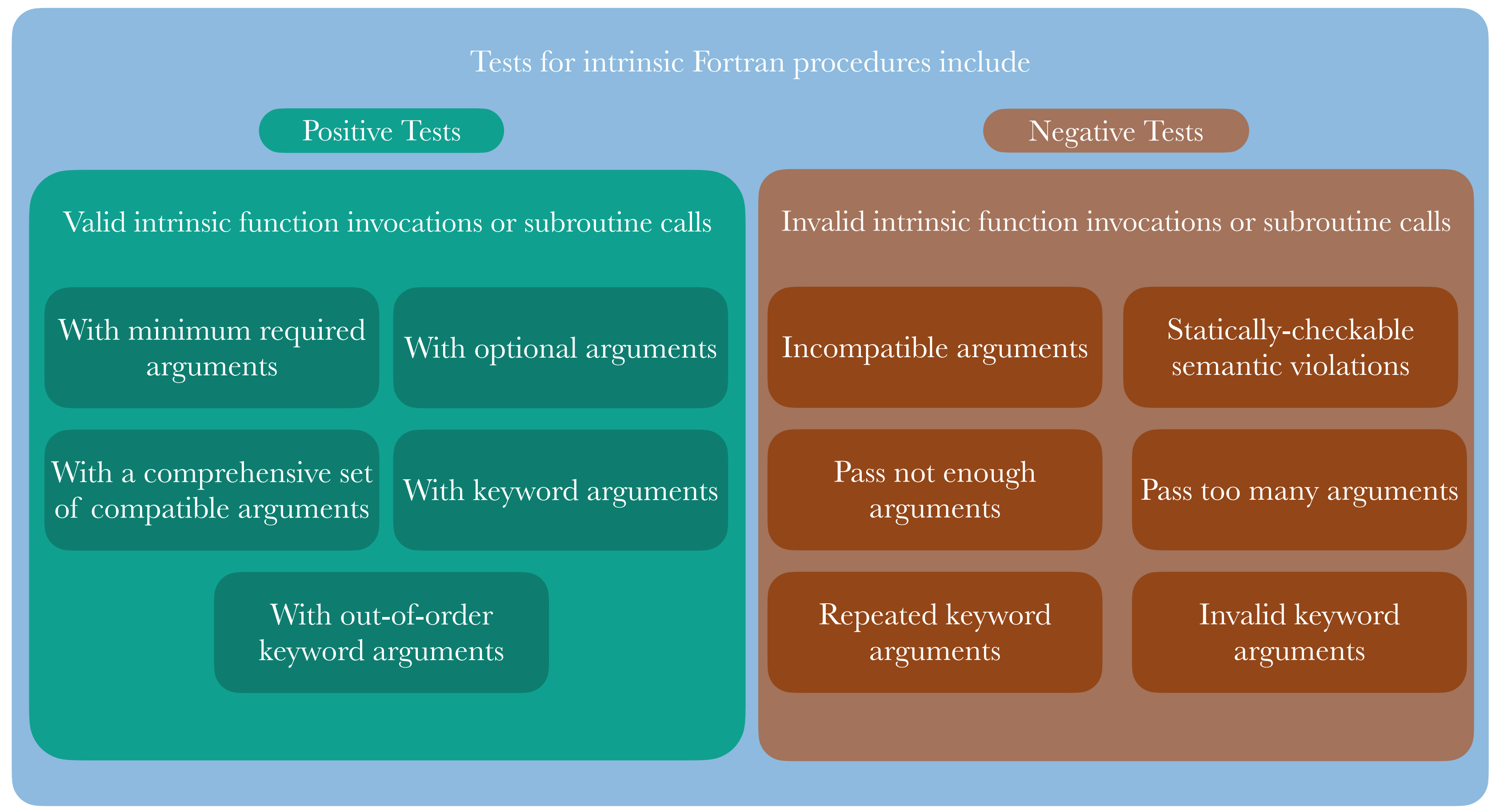

Tests for intrinsic Fortran procedures include
Positive Tests
Negative Tests
Valid intrinsic function invocations or subroutine calls
With minimum required arguments
With optional arguments
With a comprehensive set of compatible arguments
With keyword arguments
With out-of-order keyword arguments
Invalid intrinsic function invocations or subroutine calls
Incompatible arguments
Statically-checkable semantic violations
Pass not enough arguments
Pass too many arguments
Repeated keyword arguments
Invalid keyword arguments

Figure 2: Diagram outlining the components of the static semantic tests for intrinsic functions and intrinsic subroutines

Test-Driven Development Example

CO_SUM (A [, RESULT_IMAGE, STAT, ERRMSG])

Figure 3: Signature for the intrinsic collective subroutine, ***co_sum***, as defined by the Fortran 2018 standard. 'a' is the only required argument and the rest of the arguments are optional.

```
! RUN: %python %S/test_errors.py %s %flang_fc1
! XFAIL: *
! This test checks for
! the co_reduce interf

program test_co_sum
  implicit none

  integer i, status, integer_array(1), coindexed_integer[*], coindexed_result_image[*]
  complex c, complex_array(1,1,1, 1,1,1, 1,1,1, 1,1,1, 1,1,1)
  double precision d, double_precision_array(1)
  real r, real_array(1), coindexed_real[*]
  character(len=1) message, coindexed_character[*], character_array(1), repeated_message
  logical bool

  !___ standard-conforming calls with no keyword arguments ___

  ! Calls with only the required argument
  ! Argument 'a', shall be of a numeric type, test all Fortran numeric types
  call co_sum(i)
  call co_sum(c)
  call co_sum(d)
  call co_sum(r)
  ! Calls with one optional argument, 'result_image'
  call co_sum(i, 1)
  ! Calls with two optional arguments, 'result_image' and 'stat'
  call co_sum(c, 1, status)
  ! Calls with three optional arguments, 'result_image', and 'stat', and 'errmsg'
  call co_sum(d, 1, status, message)
  ! Argument 'a' may be an array
  call co_sum(integer_array)

  !___ standard-conforming calls with keyword arguments ___

  ! all arguments present
  call co_sum(a=i, result_image=1, stat=status, errmsg=message)
  call co_sum(a = i, result_image = 1, stat = status, errmsg = message)
  ! change order of keyword arguments
  call co_sum(result_image=1, a=i, errmsg=message, stat=status)

  ! one optional argument not present
  call co_sum(a=i,                 stat=status, errmsg=message)
  call co_sum(a=i, result_image=1,              errmsg=message)
  call co_sum(a=i, result_image=1, stat=status                )

  ! two optional arguments not present
  call co_sum(a=i, result_image=1                             )
  call co_sum(a=i,                 stat=status                )
  call co_sum(a=i,                              errmsg=message)

  ! no optional arguments present
  call co_sum(a=i                                             )

  !___ non-standard-conforming calls ___

  ! missing mandatory arguments
  call co_sum()
  call co_sum(result_image=1, stat=status, errmsg=message)

  ! repeated keyword arguments
  call co_sum(a=i, a=c)

  ! keyword arguments supplied positionally by an earlier actual argument
  call co_sum(i, 1, a=c)

  ! incompatible actual arguments for argument 'a'
  call co_sum(bool)                                          ! type mismatch
  call co_sum(a=1+1)                                         ! 'a' argument is intent(inout) and must be definable
  call co_sum(a=coindexed_real[1])                           ! 'a' argument to 'co_sum' may not be a coindexed object

  ! incompatible actual arguments for argument 'result_image'
  call co_sum(i, result_image=bool)                          ! type mismatch
  call co_sum(c, result_image=integer_array)                 ! rank mismatch

  ! incompatible actual arguments for argument 'stat'
  call co_sum(a=i, result_image=1, stat=1+1, errmsg=message) ! argument is intent(out) and must be definable
  call co_sum(stat=coindexed_integer[1], a=d)                ! argument may not be a coindexed object
  call co_sum(r, stat=message)                               ! type mismatch
  call co_sum(i, stat=integer_array)                         ! rank mismatch

  ! incompatible actual arguments for argument 'errmsg'
  call co_sum(a=i, result_image=1, stat=status, errmsg='c')  ! argument is intent(inout) and must be definable
  call co_sum(c, errmsg=coindexed_character[1])              ! argument may not be a coindexed object
  call co_sum(c, errmsg=i)                                   ! type mismatch
  call co_sum(d, errmsg=character_array)                     ! rank mismatch

  ! too many arguments
  call co_sum(r, result_image=1, stat=status, errmsg=message, 3.4)

  ! keyword argument with incorrect name
  call co_sum(fake=3.4)

  ! coindexed objects not allowed for 'a', 'stat', 'errmsg', but is allowed 'result_image'
  call co_sum(result_image=coindexed_result_image[1], a=coindexed_real[1], &
              errmsg=coindexed_character[1], stat=coindexed_integer[1])

end program test_co_sum
```

XFAIL directive to inform LLVM lit testing framework that test should expectedly fail. The tests fails since compiler doesn't know the interface to the intrinsic subroutine ***co_sum***.

No errors produced for invalid code.

Figure 4: Static semantics test excerpt for the ***co_sum*** subroutine, this test expectedly fails.

```
! RUN: %python %S/test_errors.py %s %flang_fc1
! This test
! the co_reduce inte

program test_co_sum
  implicit none

  integer i, status, integer_array(1), coindexed_integer[*], coindexed_result_image[*]
  complex c, complex_array(1,1,1, 1,1,1, 1,1,1, 1,1,1, 1,1,1)
  double precision d, double_precision_array(1)
  real r, real_array(1), coindexed_real[*]
  character(len=1) message, coindexed_character[*], character_array(1), repeated_message
  logical bool

  !___ standard-conforming calls with no keyword arguments ___

  ! Calls with only the required argument
  ! Argument 'a', shall be of a numeric type, test all Fortran numeric types
  call co_sum(i)
  call co_sum(c)
  call co_sum(d)
  call co_sum(r)
  ! Calls with one optional argument, 'result_image'
  call co_sum(i, 1)
  ! Calls with two optional arguments, 'result_image' and 'stat'
  call co_sum(c, 1, status)
  ! Calls with three optional arguments, 'result_image', and 'stat', and 'errmsg'
  call co_sum(d, 1, status, message)
  ! Argument 'a' may be an array
  call co_sum(integer_array)

  !___ standard-conforming calls with keyword arguments ___

  ! all arguments present
  call co_sum(a=i, result_image=1, stat=status, errmsg=message)
  call co_sum(a = i, result_image = 1, stat = status, errmsg = message)
  ! change order of keyword arguments
  call co_sum(result_image=1, a=i, errmsg=message, stat=status)

  ! one optional argument not present
  call co_sum(a=i,                 stat=status, errmsg=message)
  call co_sum(a=i, result_image=1,              errmsg=message)
  call co_sum(a=i, result_image=1, stat=status                )

  ! two optional arguments not present
  call co_sum(a=i, result_image=1                             )
  call co_sum(a=i,                 stat=status                )
  call co_sum(a=i,                              errmsg=message)

  ! no optional arguments present
  call co_sum(a=i                                             )

  !___ non-standard-conforming calls ___

  !ERROR: missing mandatory 'a=' argument
  call co_sum()
  !ERROR: missing mandatory 'a=' argument
  call co_sum(result_image=1, stat=status, errmsg=message)

  !ERROR: repeated keyword argument to intrinsic 'co_sum'
  call co_sum(a=i, a=c)

  !ERROR: keyword argument to intrinsic 'co_sum' was supplied positionally by an earlier actual argument
  call co_sum(i, 1, a=c)

  !ERROR: Actual argument for 'a=' has bad type 'LOGICAL(4)'
  call co_sum(bool)
  !ERROR: Actual argument associated with INTENT(IN OUT) dummy argument 'a=' must be definable
  call co_sum(a=1+1)
  !ERROR: 'a' argument to 'co_sum' may not be a coindexed object
  call co_sum(a=coindexed_real[1])

  !ERROR: Actual argument for 'result_image=' has bad type 'LOGICAL(4)'
  call co_sum(i, result_image=bool)
  !ERROR: 'result_image=' argument has unacceptable rank 1
  call co_sum(c, result_image=integer_array)

  !ERROR: Actual argument associated with INTENT(OUT) dummy argument 'stat=' must be definable
  call co_sum(a=i, result_image=1, stat=1+1, errmsg=message)
  !ERROR: 'stat' argument to 'co_sum' may not be a coindexed object
  call co_sum(stat=coindexed_integer[1], a=d)
  !ERROR: Actual argument for 'stat=' has bad type 'CHARACTER(KIND=1,LEN=1_8)'
  call co_sum(r, stat=message)
  !ERROR: 'stat=' argument has unacceptable rank 1
  call co_sum(i, stat=integer_array)

  !ERROR: Actual argument associated with INTENT(IN OUT) dummy argument 'errmsg=' must be definable
  call co_sum(a=i, result_image=1, stat=status, errmsg='c')
  !ERROR: 'errmsg' argument to 'co_sum' may not be a coindexed object
  call co_sum(c, errmsg=coindexed_character[1])
  !ERROR: Actual argument for 'errmsg=' has bad type 'INTEGER(4)'
  call co_sum(c, errmsg=i)
  !ERROR: 'errmsg=' argument has unacceptable rank 1
  call co_sum(d, errmsg=character_array)

  !ERROR: too many actual arguments for intrinsic 'co_sum'
  call co_sum(r, result_image=1, stat=status, errmsg=message, 3.4)

  !ERROR: unknown keyword argument to intrinsic 'co_sum'
  call co_sum(fake=3.4)

  !ERROR: 'a' argument to 'co_sum' may not be a coindexed object
  !ERROR: 'errmsg' argument to 'co_sum' may not be a coindexed object
  !ERROR: 'stat' argument to 'co_sum' may not be a coindexed object
  call co_sum(result_image=coindexed_result_image[1], a=coindexed_real[1], &
              errmsg=coindexed_character[1], stat=coindexed_integer[1])

end program test_co_sum
```

XFAIL directive removed. The test passes now that the compiler knows that ***co_sum*** is an intrinsic subroutine and knows its interface.

ERROR directives now produced for invalid code that match the error the compiler produces.

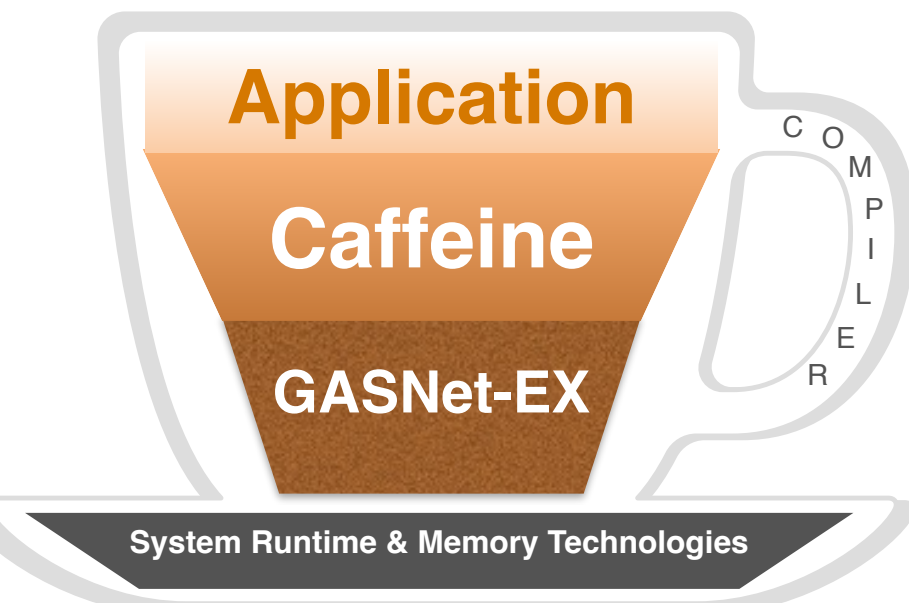
Figure 5: Updated static semantics test excerpt for the ***co_sum*** subroutine that passes after interface is added

```
{"co_sum",
    {{"a",            AnyNumeric,  Rank::anyOrAssumedRank, Optionality::required, common::Intent::InOut},
     {"result_image", AnyInt,      Rank::scalar,           Optionality::optional, common::Intent::In},
     {"stat",         AnyInt,      Rank::scalar,           Optionality::optional, common::Intent::Out},
     {"errmsg",       DefaultChar, Rank::scalar,           Optionality::optional, common::Intent::InOut}},
    {}, Rank::elemental, IntrinsicClass::collectiveSubroutine},
```

Figure 6: Interface to compiler for ***co_sum*** allows the test to pass when combined with a static semantic check for coindexed objects (not shown).

Runtime Tests

- Because Flang cannot yet produce executable files from Fortran 2018 source code, we are developing runtime tests in a separate repository: Caffeine.
- Caffeine is a runtime library that supports parallel Fortran 2018 features.
- Caffeine runs atop the GASNet-EX exascale networking middleware.



go.lbl.gov/caffeine

- For more on Caffeine, see: Rouson & Bonachea (2022) "*Caffeine: CoArray Fortran Framework of Efficient Interfaces to Network Environments*" SC22 Workshop on the LLVM Infrastructure in HPC doi:10.25344/S4459B

Outcomes

- The Berkeley Lab fork of the LLVM-Project GitHub repository includes a project board capturing an exhaustive list of 41 parallel features to test.
- We have pushed static semantics tests for 32 such features upstream to LLVM-Project: intrinsic functions supporting parallelism, collective subroutines, atomic subroutines, synchronization statements, and more.
- We have contributed additional static semantic analysis and error checking for 11 missing parallel features exposed by our tests. More contributions are under development or in code review.
- We contributed error checks for 2 non-parallel features.
- We have developed 44 runtime tests that we exercise by developing Caffeine.

U.S. DEPARTMENT OF ENERGY Office of Science

This research was supported in part by the Sustainable Research Pathways for High Performance Computing (SRP-HPC) a project of the Sustainable Horizons Institute.
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.

ECP EXASCALE COMPUTING PROJECT