Proceedings of SCS Spring Simulation Multi-Conference (SpringSim'17), 23 - 26 April 2017, Virginia Beach, VA, USA
Available at: http://scs.org/wp-content/uploads/2017/06/14_Final_Manuscript-3.pdf

A FRAMEWORK FOR UNIT TESTING WITH COARRAY FORTRAN

Ambra Abdullahi Hassan
University of Rome Tor Vergata
Rome, Italy
ambra.abdullahi@uniroma2.it

Valeria Cardellini
University of Rome Tor Vergata
Rome, Italy
cardellini@ing.uniroma2.it

Salvatore Filippone
Cranfield University
Cranfield, UK
salvatore.filippone@cranfield.ac.uk

ABSTRACT

Parallelism is a ubiquitous feature of modern computing architectures; indeed, we might even say that serial code is now automatically legacy code. Writing parallel code poses significant challenges to programs, and is often error-prone. Partitioned Global Address Space (PGAS) languages, such as Coarray Fortran (CAF), represent a promising development direction in the quest for a trade-off between simplicity and performance. CAF is a parallel programming model that allows a smooth migration from serial to parallel code. However, despite CAF simplicity, refactoring serial code and migrating it to parallel versions is still error-prone, especially in complex softwares. The combination of unit testing, which drastically reduces defect injection, and CAF is therefore a very appealing prospect; however, it requires appropriate tools to realize its potential. In this paper, we present the first CAF-compatible framework for unit tests, developed as an extension to the Parallel Fortran Unit Test framework (pFUnit).

Keywords: Coarray Fortran; Test-Driven Development; Unit tests; pFUnit; Refactoring

1 INTRODUCTION

Scientific software tends to have rather peculiar characteristics (Carver, Kendall, Squires, and Post 2007): its requirements gathering process is unique in that nature often plays a prominent role, it is often the outcome of research projects with little or no development planning, it is often written by scientists whose primary research interest is not software-related, and scientific software packages tend to have very long lifetimes.

Scientific software development then is rarely accompanied by techniques such as Test-Driven Development (TDD) that are very frequent in other application areas, nor is it often the case that (semi-)automated tools are employed. Many authors have advocated the use of disciplined design strategies in this context, see e.g., (Rouson, Xia, and Xu 2011).

The introduction of parallel features into existing serial codes is a necessary step in today's world; unfortunately, parallel programming techniques open many opportunities to introduce subtle bugs and puzzling

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA
©2017 Society for Modeling & Simulation International (SCS)

©2017 The Society for Modeling and Simulation International. This is the Author Accepted Manuscript.
Please refer to any applicable publisher terms of use.

behavior in the software under consideration. Therefore, we deem appropriate to also introduce development strategies to keep these risks under control and to increase our confidence in the numerical results produced by the software.

1.1 Parallelization and PGAS Languages

With the proliferation of multicore processors and many-core accelerators, any serial code is automatically "legacy", and ought to be parallelized in order to perform effectively (Radhakrishnan, Rouson, Morris, Shende, and Kassinos 2013). The Message Passing Interface (MPI) offers a rich set of functionalities for parallel applications on distributed systems, but parallelization must be handled manually by the programmer, and taking care of very low-level data transfer details. The Partitioned Global Address Space (PGAS) parallel programming model is an effective and interesting alternative that combines the advantages of the Single Program Multiple Data (SPMD) programming style for distributed memory systems with the data referencing semantics of shared memory systems. Unified Parallel C (UPC) (UPC Consortium 2005), Coarray Fortran (CAF) (Numrich and Reid 1998), Chapel (Chamberlain, B.L. 2015), X10 (Saraswat, Bloom, Peshansky, Tardieu, and Grove 2012), SHMEM are all based on the PGAS model. In the Fortran case, this model has been integrated in the Fortran 2008 standard with the introduction of coarrays.

CAF communications are largely abstracted at a much higher level than in MPI; moreover, code written in CAF can easily work on both shared and distributed memory architectures. Additionally, CAF syntax allows to express a parallel algorithm in a simpler style. CAF has been tested on different scientific applications (Ashby and Reid 2008), (Hasert, Klimach, and Roller 2011), (Cardellini, Fanfarillo, and Filippone 2016) and it has been shown that CAF code efficiency is comparable to that of MPI code. In the trade-off between readability and efficiency, we think that CAF is a winning choice when parallelizing a scientific legacy code.

1.2 Paper Contributions

However, CAF currently lacks the support of unit testing tools and this makes the developer task harder. The goal of our paper is to fill this gap and to provide a contribution toward the test-driven development of scientific applications written in CAF, in such a way to make the development schedule much more predictable. To this end, we extend pFUnit, the parallel Fortran Unit testing framework, with the support for CAF, thus contributing with a framework for unit testing with Coarray Fortran.

The rest of the paper is organized as follows. In Section 2 we provide some background on CAF and testing of scientific software, including TDD and pFUnit. In Section 3 we describe the proposed extension of pFUnit that supports CAF, also providing an example of testing a simple CAF code with pFUnit. In Section 4 we describe a case study of unit testing related to the migration from MPI to CAF of PSBLAS, a library of Basic Linear Algebra Subroutines for parallel sparse applications. We conclude with Section 5 providing some hints for future work.

2 BACKGROUND

2.1 Coarray Fortran

Coarray Fortran achieves a good trade-off between readability and performance: it introduces minimal syntactic extensions, it allows for very clear specification of data movement, and it allows a smooth path for code migration and incremental parallelization. Compared with the parallelization of application under

MPI, the parallel features are easier to follow, the code tends to be far shorter, and the impact of the parallel statements is much smaller.

CAF started as an extension of Fortran for parallel processing and is now part of the Fortran 2008 standard. CAF adopts the PGAS model for SPMD parallel programming, where multiple "images" share their address space; the shared space is partitioned, and each image has a local portion. The standard is very careful not to constrain what an "image" should be, so as to allow different implementations to make use of underlying threads and/or processes. From a logical, user-centered point of view, each CAF program is replicated across images, and the images execute independently until they reach a synchronization point, either explicit or implicit. The number of images is not specified in the source code, and can be chosen at compile, link or run-time, depending on the particular implementation of the language. Images are assigned unique indices through which the user can control the flow of the program by conditional statements, similar to common usage in MPI. Coherently with the language default rules, image indices range from 1 to a maximum index which can be retrieved at runtime through an appropriate intrinsic function `num_images()`. Each image has its own set of data objects; some of these objects may also be declared as coarray objects, meaning that they can be accessed by other images as well. Coarrays are declared with a so-called codimension, indicated by square brackets. The codimension spans the space of all the images:

```
integer :: i[*]
real :: a(10)[*]
real :: b(0:9)[0:4,*]
```

As already mentioned, one of the main advantages of CAF over MPI is its simplicity. Much of the parallel bookkeeping is handled behind the scenes by the compiler, and the resulting parallel code is shorter and more readable; this reduces the chances of injecting defects in the code. As an example, let us consider the burden of passing a non-contiguous array (for example, a matrix row in Fortran) using `MPI_DATATYPE`:

```
! Define vector
call MPI_Type_vector(n,nb,n,MPI_REAL, myvector,ierr)
call MPI_Type_commit(myvector, ierr)
! Communicate
if(n_rank == 0) then
   call MPI_Send(a,1,myvector,1,100, MPI_COMM_WORLD, ierr)
endif
if(n_rank == 1) then
   call MPI_Recv(a,1,myvector,0,100, MPI_COMM_WORLD, mystatus, ierr)
endif

call MPI_Type_free(myvector, ierr)
```

All of the above code is equivalent to just a single CAF statement:

```
a(1,:)[2]=a(1,:)[1]
```

with no need to write separate code for sending and receiving a message.

Currently, the number of compilers implementing CAF has increased: the Intel and Cray compilers support it. The GCC compiler provides CAF support via a communication library; the base GCC distribution only handles a single image, but the OpenCoarrays project provides an MPI-based and a GASNet-based communication library (Fanfarillo, Burnus, Cardellini, Filippone, Nagle, and Rouson 2014).

2.2 Unit Test and Test-Driven Development

Software testing can be applied at different levels: *unit tests* are fine-grained tests, focusing on small portions of the code (a single module, function or subroutine) (Osherove 2015), while *regression tests* are coarse-

grained tests, that encompass a large portion of the implementation and are used to verify that the software still performs correctly after a modification.

Many scientific softwares rely on regression, disregarding unit tests. This choice presents several disadvantages, spanning from the impossibility to perform verification in the early stages of software's lifecycle, to difficulties in locating the error responsible for failure, and to a long time required to run tests.

Additionally, the search for performance through parallelism increases complexity of the code and may cause new classes of errors: race conditions and deadlocks are two such examples.

Race conditions occur when multiple images try to access concurrently the same resource. For example, in the code fragment:

```
count[1]=0
do i=1 , n
  count[1] = count[1] + 1
enddo
```

the final value of the variable `count[1]` is not uniquely determined by the code, and depends upon the order in which the various images execute the statements.

A deadlock occurs when an image is waiting for an event that cannot possibly occur, e.g., when two images wait for each other; as a consequence, the program makes no further progress. With CAF a deadlock occurs in the following example:

```
if (this_image == 1) then
   sync images(num_images())
endif
```

because image 1 is waiting for synchronization with the last image, but the last image is not executing the matching `sync` statement.

These new classes of errors enforce the need for systematic unit testing.

Test-Driven-Development (TDD) is a software development practice relying strongly on unit tests. TDD combines a test-first approach with refactoring: before actually implementing the code, the programmer writes automated unit tests for the functionality to be implemented.

Writing unit tests is a time-consuming process, and if the associated effort is perceived to be excessive, the programmer may be inclined to reduce or skip it altogether. Unit testing frameworks are tools intended to help the developers to write, execute and review tests and results more efficiently. Many testing frameworks exist; they are often called "xUnit" frameworks, where "x" stands for the (initials of the) name of the language they are developed for.Usually, a unit testing framework provides code libraries with basic classes, attributes and assert methods; it also includes a test runner used to automatically identify and run tests, and provides information about their number, failures, raised exceptions, location of failures. Good testing frameworks are critical for the acceptance of TDD.

2.3 pFUnit: A Unit Testing Framework for Parallel Fortran

Given the above discussion, it should by now be clear that a unit testing framework for Coarray Fortran applications is a desirable tool. A good basis for a CAF compatible unit testing framework ought to have certain characteristics. First, it should be conceived for the Computational Science & Engineering (CSE) and HPC development communities. The ideal framework should also handle parallelism and should be able to detect the peculiar errors caused by concurrently execution of different images. Additionally, it must be easy to extend, for example through object-oriented (OO) features.

pFUnit from NASA (Clune and Rood 2011) is a tool that satisfies all of these requirements. pFUnit is a unit testing framework implemented by keeping in mind the open source xUnit family, and is developed in Fortran 2003, using OO design techniques. While patterned after Junit, pFUnit is tailored to the CSE/HPC environment and supports comparison of single/double precision quantities with optional tolerance and test for infinity and *NaN* (thus permitting to check for subtle numerical issues that can affect result quality). Moreover, the available support for parallelism and its OO structure, make it an ideal starting point to start for the creation of a CAF-compliance unit testing framework.

The simplest way to write tests using pFUnit is through a preprocessor input file. The preprocessor input file is not written in standard Fortran and has extension "*.pf*". It is a Fortran free format file with preprocessor directives added. The parser automatically generates one test suite per file/module, and suite names are derived from the file or module containing the tests. Once the preprocessor is invoked, it generates a Fortran file that when compiled and linked with pFUnit will provide tests subroutine. Finally, pFUnit provides a driver, that is, a short program that bundles all of the test suites, runs the tests and produces a short summary.

3 EXTENDING PFUNIT WITH CAF SUPPORT

We now present how we provide support for Coarray Fortran in pFUnit. To this end, we created a set of CAF classes, extending those already available inside pFUnit. They are shown in Figure 1 and include:

CafTestCase This class allows a single test procedure to be executed multiple times with different input values. By analogy with MpiTestCase, it is a special subclass of the more general ParameterizedTestCase and allows to run tests using pFUnit custom support for parameterized tests.

CafTestParameter This class provides procedures for setting/getting the number of images for the running test and ensures that image causing the failure is correctly reported.

CafTestMethod This class permits test fixture by specifying setUp() and tearDown() methods.

CafContext This class provides some communication routines such as gather and reduce.

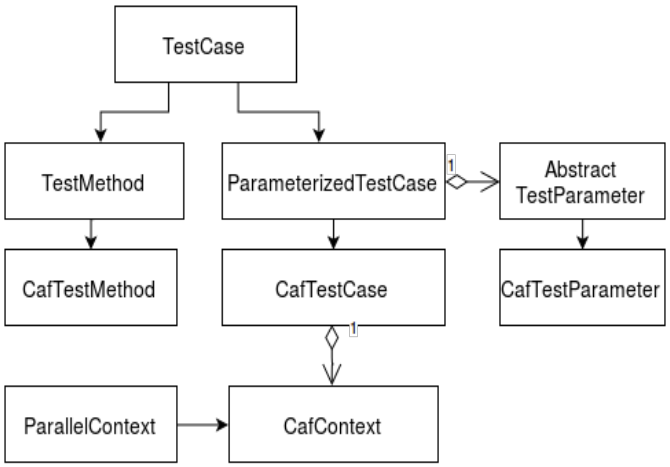


Figure 1: CAF classes inside pFUnit.

In the `CAF_context` class, some collective operations need to be defined. Collectives for coarrays have been proposed and scheduled for inclusion in a future revision of the Fortran standard, but are not currently available in many compilers. For this reason a simple `co_sum` function and a logical reduction, as well as some gather subroutines are provided in this class.

The implementation of these collectives is not fully optimized, since they are intended to be superseded by the runtime of the compiler, but this is not too restrictive since they will will run only once for each test suite.

From the user perspective, CAF tests have a single, mandatory argument of type CafTestCase and with `intent(INOUT)`. It provides two methods `getNumImages()` and `getImageRank()`, returning the total number of images running for the test and the rank of the current image. In CAF, test failing assertions of any types provide information about the rank of the process (or the ranks of the processes) that detects the failure. This permits to recover not only on the test that has failed, but on the specific image that causes the failure.

3.1 An Example of CAF Unit Test Using pFUnit

Let us now consider a simple example of testing a CAF code with pFUnit. The example has been run on a Linux laptop, using GCC 6.1.0, MPICH 3.2, OpenCoarrays 1.6.2 and our development version of pFUnit. Let us assume that we have an application where the CAF images are organized in a linear array, and our computations need to deal with elements of an index space. Let us also assume that the amount of computation per point in the index space is constant; naturally, we want our workload to be distributed as evenly as possible. A possible solution would be to use the following routine which determines, for each image, the size of the local number of indices and the first index assigned to it:

```
subroutine data_distribution(n,il,nl)
  integer, intent(in)  :: n
  integer, intent(out) :: il, nl
  integer :: nmi
  ! Compute first local index IL and number of local indices NL. We want the data to
  ! be evenly spread, i.e., for all images! NL must be within 1 of n/num_images()
  ! For all images we should have  that IL[ME+1]-IL[ME]=NL
  associate(me=>this_image(),images=>num_images())
    nmi = mod(n,images)
    nl = n/images  + merge(1,0,me<=nmi)
    il = min(n+1,merge((1+(me-1)*nl), &
         & (1+ nmi*(nl+1) + (me-nmi-1)*nl),&
         & me <= nmi))
  end associate
end subroutine data_distribution
```

It is easy to see that on `NP` images, this routine will assign to each image either `N/NP` indices or `(N/NP)+1`, as necessary to make sure the indices add up to `N`. To check that our routine is working properly with pFUnit we need to specify clearly which properties our data distribution is supposed to have; in our case we have:

1. The local sizes must add up to the global size `N`;
2. The local sizes must differ by at most 1 from `N/NP`;
3. For each image `me<num_images()`, `il[me+1]-il[me]` must be equal to `nl`.

Checking the first property is very simple when the underlying compiler (like GNU Fortran 6.1.0) supports the collective intrinsics:

```
  @test(nimgs=[std])
  subroutine test_distribution_1(this)
    implicit none
    Class(CafTestMethod), intent(inout) :: this
    integer, parameter :: gsz=27
    integer :: ngl, info
    integer, allocatable  :: il[:], nl[:]
```

```
    allocate(il[*],nl[*],stat=info)
    @assertEqual(info,0,"Failed allocation!")
    ! Set up checks
    associate(me=>this_image(),images=>num_images())
      call data_distribution(gsz,il,nl)
      ! Build reference data to check against
      ngl = nl
      call co_sum(ngl)
      @assertEqual(ngl,gsz,"Sizes do not add up!")
    end associate
  end subroutine test_distribution_1
```

The macro `@test(nimgs=[std])` indicates that the test we are running is a CAF test that runs on all available images. The `this` argument is mandatory and must have `intent(inout)`; the assertion `@assertEqual` is used to verify that the actual result matches the expected output.

The coarrays remote access facilities make it very easy to check for consistency across process boundaries. Since the OpenCoarrays installation we are using is built on top of MPICH, we execute the tests with the `mpirun` command, as shown in Figure 2. The output gives us confidence that the data distribution is computed correctly; the run with 15 processes tells us that the border case where we have more processes than indices (in our case 13) is being handled correctly. Since the test has been run on an quad-core laptop, running 15 MPI processes has a substantial overhead, which is entirely normal; the run with 4 processes was very fast, as expected.

```
[localhost CAF_pFUnit] mpirun -np 4 ./testCAF
...
Time:          0.004 seconds
 OK
 (3 tests)
[localhost CAF_pFUnit] mpirun -np 15 ./testCAF
...
Time:          6.207 seconds
 OK
 (3 tests)
```

Figure 2: Test output

What happens if there is an error? To demonstrate this, we wrapped the `data_distribution` routine injecting two errors on image 2: we alter both the local number of indices as well as the starting index. The test output is shown in Figure 3. We get a fairly precise indication of what went wrong and where:

- An error on all images because the total size does not match the expected value;
- An error on image 2 because the local number of indices is not in the expected range `(N/NP):(N/NP)+1`;
- An error on images 1 and 2, because having injected an error in the start index on image 2 affects the checks on both of these images.

3.2 Limitations: Team Support

It is often desirable to be able to run tests using varying number of processes/images; this is because some bugs reveal themselves only when running on a certain number of processes. In pFUnit, the user is al-

```
[localhost CAF_pFUnit] mpirun -np 4 ./testCAF
.F.F.F
Time:          0.013 seconds

 Failure in: CAF_distribution_test_mod_suite.test_distribution_1[nimgs=4][nimgs=4]
   Location: [testCAF.pf:25]
Distribution does not add up! expected 13 but found: 17;  difference: |4|. (IMG=1)

..........

 Failure in: CAF_distribution_test_mod_suite.test_distribution_2[nimgs=4][nimgs=4]
   Location: [testCAF.pf:50]
One image is getting too many entries expected 4 to be less than or equal to: 1. (IMG=2)

 Failure in: CAF_distribution_test_mod_suite.test_distribution_3[nimgs=4][nimgs=4]
   Location: [testCAF.pf:81]
Start indices not consistent expected 4 but found: -4;  difference: |8|. (IMG=1)

 Failure in: CAF_distribution_test_mod_suite.test_distribution_3[nimgs=4][nimgs=4]
   Location: [testCAF.pf:81]
Start indices not consistent expected 7 but found: 11;  difference: |4|. (IMG=2)

 FAILURES!!!
Tests run: 3, Failures: 3, Errors: 0
 there is an error
```

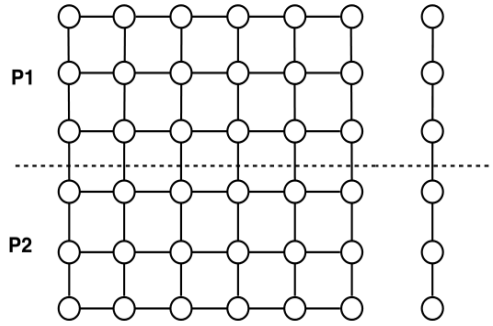
Figure 3: Test output with errors

lowed to control this aspect by passing an optional argument `nimgs=[<list>]` (in the case of CAF) or `nprocs=[<list>]` (in the case of MPI): in this way the test procedure will execute once for each item in `<list>`. MPI makes this possible by constructing a sub-communicator of the appropriate size for each execution. In principle, CAF allows for clustering of images in teams, which are meant to be similar to MPI communicators; at any time an image executes as a member of a team (the current team). Constructs are available to create and synchronize teams.

Unfortunately, at the time of this writing, the only compiler supporting teams is the OpenUH one (Khaldi, Eachempati, Ge, Jouvelot, and Chapman 2015). However, this compiler does not support other standard Fortran features; most importantly, it does not support the OO programming features, and therefore cannot be used with pFUnit. As a result, the current version of pFUnit only allows the CAF user-defined tests to be run with all images, and the only accepted values for the optional argument `nimgs` is `[std]`.

4 CASE STUDY: PSBLAS

We illustrate a case study that shows how pFUnit can be used to detect errors. We applied pFUnit to perform unit tests on the PSBLAS library (Filippone and Colajanni 2000) (Filippone and Buttari 2012) during the library migration from MPI to CAF. At the time of writing, we have written a total of 12 test suites and 241 unit tests. PSBLAS is a library of Basic Linear Algebra Subroutines that implements iterative solvers for sparse linear systems and includes subroutines for multiplying sparse matrices by dense matrices, solving sparse triangular systems, and preprocessing sparse matrices, as well as additional routines for dense matrix operations. It is implemented in Fortran 2003 and the current version uses message passing to address a distributed memory execution model. We converted the code gradually from MPI to coarrays, thus having MPI and CAF coexisting in the same code. In PSBLAS, we detected three communication patterns that had to be modified: 1) the halo exchange, 2) the collective subroutines, and 3) the point-to-point communication in data distribution. In PSBLAS, data allocation on the distributed-memory architecture is driven by the



Before Halo Exchange

P1			P2		
loc(i)	glob(i)	val(i)	loc(i)	glob(i)	val(i)
1	1	1	1	4	2
2	2	1	2	5	2
3	3	1	3	6	2
4	4	4	4	3	3

After Halo Exchange

P1			P2		
loc(i)	glob(i)	val(i)	loc(i)	glob(i)	val(i)
1	1	1	1	4	2
2	2	1	2	5	2
3	3	1	3	6	2
4	4	2	4	3	1

Figure 4: Example of halo exchange on 2 images.

discretization mesh of the original PDEs. Each variable is associated to one point of the discretization mesh. If $a_{ij} \neq 0$ we say that point `i` depends on point `j`. After the partition of the discretization mesh into subdomains, each point is assigned to a parallel process. An halo point is a point which belongs to another domain, but there is at least one point in this domain that depends on it. Halo points are requested by other domains when performing a computational step (for example a matrix-vector product) : every time this happens we perform an halo exchange operation that can be considered as a sparse all-to-all communication.

The subroutine `psb_halo` performs such an exchange and gathers values of the halo elements: we used our framework to unit test this procedure. We need to run the whole suite of tests multiple times if we want to change the number of images participating in the test. To avoid that, in each test we distributed variables not among all the available images, but only on a subset of them. In this way, while all tests run on the same number of images, communication actually takes place only between some of the images. Of course this has to be taken into account when asserting equality on all images (we have to do that, otherwise a deadlock can occur). In the following example, we test the halo exchange of a vector. The variables are distributed only between two images. When calling the `assert` statement, the expected solution check and the obtained result `v` are multiplied by `1` if the image takes part to the communication, `0` otherwise.

```
@test(nimgs=[std])
subroutine test_psb_dhalo_2imgs_v(this)
  implicit none
  Class(CafTestMethod), intent(inout) :: this
  integer :: me, true
  real(psb_dpk_), allocatable :: v(:), check(:)
  type(psb_desc_type):: desc_a
  !Distributing point, Creating input vector v and expected solution check.
  ...
  !Calling the halo subroutine
  call psb_halo(v, desc_a, info)
  @assertEqual(0,info, "ERROR in psb_halo")

  if ((me==1).or.(me==2)) then
    true = 1
  else
    true=0
  endif

  @assertEqual(true*check,true*v)

  !Deallocate free and exit psblas
  ...
end subroutine test_psb_dhalo_2imgs_v
```

When writing a test for a given functionality, the programmer should ensure to reach the maximum code coverage. This means to test all the implementations of a given interface and to consider all the possible branches. Let us consider, for example, the collective subroutine `psb_amx` in PSBLAS which implements a maximum absolute value reduction. By looking at its interface, we can see that for its testing we need at least 15 different unit tests.

```
interface psb_amx
  module procedure psb_iamxs, psb_iamxv, psb_iamxm, psb_samxs, psb_samxv, psb_samxm,&
       & psb_camxs, psb_camxv, psb_camxm,&
       & psb_damxs, psb_damxv, psb_damxm,&
       & psb_zamxs, psb_zamxv, psb_zamxm
end interface
```

Additionally, it admits one optional argument `root` indicating which image holds the final value. If `root = -1`, then the final result is shared among images, thus performing an all-reduce operation. This parameter leads to a branch in the code, thus doubling the number of unit tests needed.

```
if (root_ == -1) then
  ! All reduce
     ...
else
  ! Reduce, root_ is the root process
     ...
endif
```

Finally, we test the utility subroutine `psb_matdist` to distribute a matrix among images according to a user defined data distribution.

```
subroutine test_psb_dmatdist1(this)
  implicit none
  Class(CafTestMethod), intent(inout) :: this
  integer :: me, np, info, iunit=12, nv, i,&
       & nz, last, j, irow, icontxt
  type(psb_desc_type):: desc_a
  type(psb_dspmat_type) :: a, a_out
  integer, parameter :: m_problem = 10
  integer, allocatable :: ipv(:), ivg(:), ia(:),&
       & ja(:), ia_exp(:), ja_exp(:)
  real(psb_dpk_), allocatable :: val(:), val_exp(:),&
       & a_exp(:,:), a_aux(:,:)
  me = this_image()
  np = num_images()
  call psb_init(icontxt,np,MPI_COMM_WORLD)
  call mm_mat_read(a,info,iunit=iunit,&
       & filename="matrix1.mtx")
  allocate(ivg(m_problem),ipv(np))
  do i=1,m_problem
    call part_block(i,m_problem,np,ipv,nv)
    ivg(i) = ipv(1)
  enddo

  !Getting the expected solution
  call a%csgetrow(1,10,nz,ia,ja,val,info)
  allocate(ia_exp(nz),ja_exp(nz), val_exp(nz))
  last = 0
  do i=1, m_problem
    if (me == ivg(i) + 1) then
      irow=i
      do j=1, nz
        if (ia(j) == irow) then
          last = last + 1
          ia_exp(last)=ia(j)
          ja_exp(last)=ja(j)
          val_exp(last)=val(j)
        endif
      enddo
    endif
  enddo

  if (allocated(a_exp)) deallocate(a_exp)
  allocate(a_exp(m_problem,m_problem))
  a_exp = 0.0d0
  do i=1,last
    a_exp(ia_exp(i),ja_exp(i))=val_exp(i)
  enddo

  !Test subroutine
  call psb_matdist(a, a_out, icontxt, &
       & desc_a,info, v=ivg)
  call a_out%csgetrow(1,m_problem,nz,ia,ja,val,info)
  !Convert to global indices
  call psb_loc_to_glob(ia, desc_a, info)
  call psb_loc_to_glob(ja, desc_a, info)
  if (allocated(a_aux)) deallocate(a_aux)
  allocate(a_aux(m_problem,m_problem))
  a_aux = 0.0d0
  do i=1,last
    a_aux(ia(i),ja(i))=val(i)
  enddo
  @assertEqual(a_aux,a_exp)

  !Free
  deallocate(a_aux, a_exp, ia, ja, val)
  deallocate(ipv, ivg, ia_exp, ja_exp, val_exp)
  call psb_spfree(a, desc_a, info)
  call psb_cdfree(desc_a, info)
  call psb_exit(icontxt)
end subroutine test_psb_dmatdist1
```

We use an auxiliary input file containing a matrix of size `m_problem` in the Matrix Market format. The test runs on all images and with all input matrices as long as the parameter `m_problem` is changed accordingly. We use a block partition distribution, through a call to `part_block`. It returns the vector `vg` of size

`m_problem`: variable `i` belongs to process `j` if `vg(i)=j`. We use this vector to manually create the partition, building the matrix `a_exp` that represents the expected solution. We then create the partition through a call to the `psb_mat` subroutine, and we build an auxiliary local matrix `a_aux`. Finally, we check the correctness of the solution by asserting the equality of the two matrices.

5 CONCLUSIONS

Coarray Fortran (CAF) makes parallelism syntactically simpler than using MPI, and CAF code is much easier to write and maintain. We believe that CAF is particularly suitable for parallelization of legacy Fortran codes, where the trade-off between readability and performance is an issue.

Support tools for CAF programmers are rare; the inadequacy of common unit testing frameworks, for example, is a major difficulty in applying TDD techniques in the case of CAF code. To improve this situation, we have extended the existing Parallel Fortran Unit Testing framework (pFUnit) to support CAF. pFUnit has been proved to be well suited to this task thanks to its object-oriented architecture and its design tailored to the needs of the HPC and CSE communities.

Future work will include the enablement of the `team` support as soon as it becomes available in the underlying compiler(s), and the collection of more usage data in the context of new application development. We are currently in contact with the authors of pFUnit to arrange for general availability of our extensions.

ACKNOWLEDGMENTS

We wish to thank Dr. Tom Clune of NASA for giving us full access to the latest pFUnit source code, as well as for starting the pFUnit project in the first place.

REFERENCES

Ashby, J. V., and J. K. Reid. 2008. “Migrating a scientific application from MPI to coarrays”. In *Proc. of 2008 Cray User Group Conf.*, CUG ’08.

Cardellini, V., A. Fanfarillo, and S. Filippone. 2016. “Heterogeneous CAF-based load balancing on Intel Xeon Phi”. In *Proc. of 2016 IEEE Int’l Parallel and Distributed Processing Symposium Workshops*, IPDPSW ’16, pp. 702–711.

Carver, J. C., R. P. Kendall, S. E. Squires, and D. E. Post. 2007. “Software development environments for scientific and engineering software: A series of case studies”. In *Proc. of 29th Int’l Conf. on Software Engineering*, ICSE ’07, pp. 550–559, IEEE.

Chamberlain, B.L. 2015. “Chapel”. In *Programming Models for Parallel Computing*. MIT Press.

Clune, T. L., and R. B. Rood. 2011. “Software testing and verification in climate model development”. *IEEE Software* vol. 28 (6), pp. 49–55.

Fanfarillo, A., T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. 2014. “OpenCoarrays: Open-source transport layers supporting Coarray Fortran compilers”. In *Proc. of 8th Int’l Conf. on Partitioned Global Address Space Programming Models*, PGAS ’14, pp. 4:1–4:11, ACM.

Filippone, S., and A. Buttari. 2012. “Object-oriented techniques for sparse matrix computations in Fortran 2003”. *ACM Transactions on Mathematical Software* vol. 38 (4), pp. 23:1–23:20.

Filippone, S., and M. Colajanni. 2000. "PSBLAS: A library for parallel linear algebra computation on sparse matrices". *ACM Transactions on Mathematical Software* vol. 26 (4), pp. 527–550.

Hasert, M., H. Klimach, and S. Roller. 2011. "CAF versus MPI-applicability of coarray Fortran to a flow solver". In *Recent Advances in the Message Passing Interface*, EuroMPI '11, pp. 228–236, Springer.

Khaldi, D., D. Eachempati, S. Ge, P. Jouvelot, and B. Chapman. 2015. "A team-based methodology of memory hierarchy-aware runtime support in Coarray Fortran". In *Proc. of 2015 IEEE Int'l Conf. on Cluster Computing*, CLUSTER '15, pp. 448–451.

Numrich, R. W., and J. Reid. 1998. "Co-Array Fortran for parallel programming". *ACM Sigplan Fortran Forum* vol. 17 (2), pp. 1–31.

Osherove, R. 2015. *The Art of Unit Testing*. MITP-Verlags GmbH & Co. KG.

Radhakrishnan, H., D. W. Rouson, K. Morris, S. Shende, and S. C. Kassinos. 2013. "Test-driven coarray parallelization of a legacy Fortran application". In *Proc. of 1st Int'l Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pp. 33–40. ACM.

Rouson, D., J. Xia, and X. Xu. 2011. *Scientific Software Design*. Cambridge University Press.

Saraswat, V. and Bloom, B. and Peshansky, I. and Tardieu, O. and Grove, D. 2012. "The X10 language specification, v2.2.3".

UPC Consortium 2005. "UPC language specifications, v1.2". Technical Report LBNL-59208, Lawrence Berkeley National Lab.

AUTHOR BIOGRAPHIES

AMBRA ABDULLAHI HASSAN is a PhD student at the University of Rome Tor Vergata, Italy. Her research interests include high performance computing and software for computational linear algebra.

VALERIA CARDELLINI, PhD, is Associate Professor at the University of Rome Tor Vergata, Italy. Her research interests are in the field of distributed and parallel computing systems. She has published more than 80 papers in international conferences and journals, has served as TPC member of conferences and co-chair of workshops and participated in EU projects on IT topics, including EoCoE and Cost Action ACROSS.

SALVATORE FILIPPONE, PhD, is a Lecturer at Cranfield University, UK. His main research interests are in software development for High Performance Computing; he has served as TPC member of a number of international conferences, as a reviewer and evaluator of EU projects, and is Associate Editor for the ACM Transactions on Mathematical Software.

https://dspace.org/

A framework for unit testing with coarray Fortran

Abdullahi Hassan, Ambra; Cardellini, Valeria; Filippone, Salvatore

2017-04-26
The Society for Modeling and Simulation International
https://dspace.lib.cranfield.ac.uk/handle/1826/12180
Hassan AA, Cardellini V, Filippone S, A framework for unit testing with coarray Fortran, Proceedings of SCS Spring Simulation Multi-Conference (SpringSim'17), 23 - 26 April 2017, Virginia Beach, VA, USA

Downloaded from a DSpace repository. https://dspace.org/