Reinventing Two-Sided Communication in Fortran to Transmit Polymorphic Objects Between Images

Brad Richardson

1 Motivation

As of the latest version of the Fortran standard ("Programming languages Fortran" 2023), there are a handful of constraints that prevent the effective use of polymorphic objects in coarrays. Specifically, it is not allowed to coindex an object which has any polymorphic components. There are reasons to want these restrictions for performance and memory management, but there are problems for which polymorphism is highly desirable. However, it is valid to use `co_broadcast` on objects with allocatable, polymorphic components. This paper describes a method of combining this feature with a novel use of the teams feature to re-derive a two-sided communication mechanism that is able to communicate polymorphic objects between images.

2 Usage

A variable of the derived type `communicator_t` defined in the `communicator` module is declared. The `init` type-bound procedure of this variable must be called by every image in the current team prior to initiating any communication. An image may then initiate transfer of any data object by calling the `send_to` type-bound procedure of that object and indicating the image to which the data should be sent. The identified image must execute a call to `receive_from` of the corresponding `communicator_t` variable identifying the image which is executing the `send_to` procedure. The `send_to` and `receive_from` procedures constitute synchronous, blocking operations. Thus, care must be taken to ensure that a deadlock does not occur. For instance, if all images initiate a send operation, then no image will be available to execute a corresponding receive, and all the images will wait forever.

An example program making use of the `communicator` library is shown below. The image with the largest index creates a message and sends it to its lower neighbor, i.e. image with one lower index. Each other image receives the message from its upper neighbor, i.e. image one higher index, and then sends it to its lower neighbor. Each image then prints the message.

```
program simple
  use communicator, only: communicator_t

  implicit none

  type(communicator_t) :: comm
  character(len=20) :: message[*]
  integer :: me, ni
  class(*), allocatable :: payload

  call comm%init()
  me = this_image()
  ni = num_images()
  if (me == ni) then
    write(message, "(A,I0)") "Hello from image ", me
  else
    call comm%receive_from(me+1, payload)
    select type (payload)
    type is (character(len=*))
      message = payload
```

```
    class default
      message = "Didn't get a string message"
    end select
  end if
  if (me > 1) call comm%send_to(me-1, message)
  critical
    print *, "Received message '" // trim(message) // "' on image ", me
  end critical
end program
```

Executing this program then results in output like the following.

```
 Received message 'Hello from image 4' on image  4
 Received message 'Hello from image 4' on image  3
 Received message 'Hello from image 4' on image  1
 Received message 'Hello from image 4' on image  2
```

3 Design

The responsibility of the `communicator_t` derived type is to create and keep track of a set of teams for each pair of images. When a pair of images initiates communication, the communicator then selects the appropriate team for that pair, arranges for them to join that team, and then perform a `co_broadcast` operation. By using a derived type, the internal data structure for storing the team information can be encapsulated, and individual communicators can be created in the presence of other uses of teams. There is a caveat however, that a communicator can only be used by images whose current team is the team that was current when the communicator was initialized. Take the following, more complicated program as an example.

```
program complex
  use communicator, only: communicator_t
  use iso_fortran_env, only: team_type

  implicit none

  integer, parameter :: RED = 1, BLUE = 2

  type(communicator_t) :: comm
  character(len=20) :: message[*]
  integer :: me, ni, my_team_num
  class(*), allocatable :: payload
  type(team_type) :: my_team

  call comm%init()
  me = this_image()
  ni = num_images()
  if (me == ni) then
    write(message, "(A,I0,A)") "Hello initial team"
  else
    call comm%receive_from(me+1, payload)
    select type (payload)
    type is (character(len=*))
      message = payload
    class default
      message = "Didn't get a string message"
    end select
```

```
  end if
  if (me > 1) call comm%send_to(me-1, message)
  critical
    print *, "Received message '" // trim(message) &
        // "' on image ", me, " of the initial team"
  end critical
  my_team_num = which_team(me)
  form team (my_team_num, my_team)
  change team (my_team)
  block
    integer :: me_now, ni_now
    type(communicator_t) :: inner_comm

    call inner_comm%init()
    me_now = this_image()
    ni_now = num_images()
    if (me_now == 1) then
      message = team_message(my_team_num)
    else
      call inner_comm%receive_from(me_now-1, payload)
      select type (payload)
      type is(character(len=*))
        message = payload
      class default
        message = "Didn't get a string message"
      end select
    end if
    if (me_now < ni_now) call inner_comm%send_to(me_now+1, message)
    critical
      print *, "Received message '" // trim(message) &
          // "' on image ", me_now, " of team " &
          // team_string(my_team_num)
    end critical
  end block
  end team
contains
  pure function which_team(im_num)
    integer, intent(in) :: im_num
    integer :: which_team

    which_team = merge(RED, BLUE, mod(im_num, 2) == 0)
  end function

  pure function team_message(team_num)
    integer, intent(in) :: team_num
    character(len=:), allocatable :: team_message

    if (team_num == RED) then
        team_message = "Red Team Rules!"
    else
        team_message = "Go Team Blue!"
    end if
  end function
```

```
  pure function team_string(team_num)
    integer, intent(in) :: team_num
    character(len=:), allocatable :: team_string

    if (team_num == RED) then
        team_string = "Red"
    else
        team_string = "Blue"
    end if
  end function
end program
```

Executing this program then results in output like the following.

```
 Received message 'Hello initial team' on image  4  of the initial team
 Received message 'Hello initial team' on image  3  of the initial team
 Received message 'Hello initial team' on image  2  of the initial team
 Received message 'Hello initial team' on image  1  of the initial team
 Received message 'Red Team Rules!' on image  1  of team Red
 Received message 'Red Team Rules!' on image  2  of team Red
 Received message 'Go Team Blue!' on image  2  of team Blue
 Received message 'Go Team Blue!' on image  1  of team Blue
```

The current implementation performs this by creating 2 teams for every pair of images in the current team. One where each image is identified as the sender, and one where it is the receiver. It stores these teams in a square matrix with dimensions being the number of images in the current team. This makes it simple to look up which team to join when initiating a communication. The sender looks up using its image index as one index, and the identified receiver as the other. The receiver looks up using its image index in the opposite position as the sender, and uses the identified sender as the other index. The `co_broadcast` operation can then always be performed specifying image 1 as the `source_image`.

For the communication, a `payload_t` derived type, with a single `class(*)` component is used. The sender copies the data into the payload, and it is then broadcast to the receiver. The receiver then calls `move_alloc` to move the now allocated data from the payload and into the output argument.

The implementation described above can be found at the GitLab repository `https://gitlab.com/everythingfunctional/communicator`.

4 Conclusion

Given the features of teams and a broadcast operation, it is possible to implement two sided communication. In Fortran this enables us to perform communication of polymorphic objects between images. However, this mechanism still does not allow communication of polymorphic objects between images in different teams, where it is possible to perform one-sided communication of non-polymorphic objects between images in different teams. It is expected that uses cases for such communication are likely rare at least.

References

"Programming languages Fortran." 2023. Standard. Vol. 2023. Geneva, CH: International Organization for Standardization.