MPMD with Coarray Fortran (2008): an Example Program

by Michael Siehl

www.mpmd-with-coarray-fortran.de

April 2016 (160327)

1 Introduction

MPMD (Multiple-Program Multiple-Data) might be a great way to express parallelism more naturally, since most real-world processes may consist of multiple *distinct* tasks which are executed in parallel. With that, MPMD could be regarded as a kind of general purpose parallelism, in contrast to pure SPMD (Single-Program Multiple-Data) which might be much more limited to some specific algorithm.

We developed some example program to briefly demonstrate how we use Fortran 2008 coarrays to do MPMD-like parallel programming. While the coding itself is not that difficult, it is also required to make the code accessible to others and thus, to identify and explain the key coding techniques. Therefore, chapter 2 may serve as a first preparation to understand our MPMD example program, whereas chapter 3 explains our example program in more detail. Chapter 4 brings some modifications to our example program to illustrate a simple PGAS load balancing technique.

In its core, we do not much more than extending a traditional sequential object-based programming style with remote communication through PGAS memory (coarrays). To do so, we

- encapsulate access to PGAS memory (coarrays) into thin and thus easy to maintain *coarray wrapper Abstract Data Types (ADT)*,
- use *purely local (non-coarray) Abstract Data Types (ADT)* to implement parallel program logic code with sequential syntax style.

Hopefully, this approach to parallel programming might be especially appealing to the many sequential (Fortran) programmers out there. Within our approach, we use coarrays just as what we believe they are: Remote communication channels that can be reused at many times during program execution.

2 General Coding Requirements

Our programming style is object-based Fortran 90/95 using *Abstract Data Types* (ADT), since we have long-term experiences with that programming style. Nevertheless, we are fairly confident that a (limited) use of Fortran 2003 OOP, namely the use of F2003 type-bound procedures, should do equally well for MPMD-like parallel programming with Coarray Fortran. Thus, others should be encouraged to try it out that way.

2.1 Encapsulate access to PGAS memory into Coarray Wrapper Objects

Initially, we started to encapsulate access to PGAS memory (coarrays) into separate wrapper ADT modules, because we wanted to avoid coding errors in our coarray programming. Subsequently, these coarray wrappers turned out to be the simple key to other required coding techniques for doing MPMD-like parallel programming with Coarray Fortran 2008 already. We use these coarray wrappers to establish and use remote communication channels between purely local (non-coarray) objects.

2.2 Coarray Correspondence

Coarray correspondence is explained in chapter 2 of Aleksandar Donev's paper *'Rationale for Co-Arrays in Fortran 2008'* [1]: ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1702.pdf .

Basically, coarray correspondence is the mechanism to establish a *remote communication channel* between coarray images. To allow coarray correspondence with MPMD-like programming, we declare all our coarrays PUBLIC in the specification part of our coarray wrapper modules. This allows to access corresponding coarrays by USE association. With that, we can easily access the remote communication channels just by placing a USE statement for a coarray wrapper module, wherever required within our purely local (non-coarray) coding.
Moreover, due to our programming style (i.e. encapsulate all access to coarrays into wrapper modules), it happens that the concept of coarray correspondence does extend to certain of our purely local (non coarray) declarations, sometimes. We call these *'corresponding local object declarations'* and declare them (according to their coarray counterparts) in the specification part of our purely local ADT modules. (These are not used within our example program.)

2.3 Circular Module Dependencies through PGAS memory

We use a fork/join-like programming style to hide Coarray Fortran's underlying SPMD model in favor of an extremely flexible MPMD-like programming style with Fortran 2008 coarrays.
(There is some information that this may not differ from how the runtime of APGAS languages might be implemented. See chapter 2.4 of the paper 'An Implementation of the Coarray Programming Model in C++' by Don M. J. Browne [2]: http://static.ph.ed.ac.uk/dissertations/hpc-msc/2012-2013/An%20implementation%20of%20the%20Coarray%20Programming%20Model%20in%20C++.pdf).

That fork/join-style implies very different code execution, and thus calling sequences, on the distinct coarray images. Furthermore, it requires a kind of (light-weight) circular module dependencies at many times. The following **example** may show this basically:

Assume two coarray images:
On the executing image 1 assume this simple (sequential) calling sequence between (purely local, non-coarray) objects: Type A → Type B.
On the remote image 2 assume that we have just 'started' an object of Type A, which is now idle waiting.

Now, being on the executing image 1, assume we want our Type B to 'tell' our remote Type A (on the remote image 2) to start execution of an object of Type C on that remote image 2:
calling sequence on image 1 (as stated above): Type A → Type B
calling sequence on image 2: Type A → Type C
(Here, Type B on image 1 'tells' Type A on image 2 to start execution of an object of Type C on image 2).

The (kind of) circular module dependency here is: On image 1, Type A does USE Type B (sequentially), but between the both images, Type B (on image 1) needs to 'use' Type A (on image 2). Thus, the parallel calling sequence between the types is:
Type A (image 1) → Type B (image 1) → Type A (image 2) → Type C (image 2).

(Of course, with Coarray Fortran we cannot call a method remotely but instead can only transfer data remotely. Still, it is fairly simple to implement program logic code that calls certain methods, dependent on the remotely transfered data.)

Nevertheless, circular module dependencies are not directly allowed in Fortran and they are not at all required here, because we can easily achieve the task by accessing (remote) PGAS memory of the remote Type (A) through its coarray wrapper. We use this simple technique within our example program to group the coarray images into teams with Fortran 2008 already.

2.4 Communicate with Types that are in 'USE' on remote images exclusively

Apart from that circular dependency scenario, there is another extreme which frequently occurs in our MPMD-like programming: We need to 'distribute' most of the (purely local, non-coarray) ADT objects to a limited number of images only and not to others. This leads to completely different 'USE' of ADTs on the distinct images. Nevertheless, even if an ADT module is only in 'USE' on a remote image, we still may need a way to communicate with that remote ADT from our executing image.

Taking our simple example above, it's easy to see: Type B is executed on image 1 exclusively, whereas Type C is executed on image 2 exclusively. Thus, both Types B and C do exist on distinct images only. With other words, we never come to any USE statement of these Types on the other image respectively, but still may need a remote communication channel between these both different remote Types.

In practice, the solution is fairly simple and not any different from the one for circular module dependencies: Since all we can remotely access is PGAS memory, and since we've already encapsulated access to the Types PGAS memory into separate coarray wrappers, all we need, to establish a communication channel to a remote Type, is to 'USE' the coarray wrapper of that remote Type. The remote Type itself is a purely local object and in our scenario only used on a remote image. (By the way, that is why we can't use Submodules to wrap access to PGAS memory, since a Submodule can't be used independently from its parent Module).

2.5 APGAS Features

With that already, we possibly did open a door to implement our own APGAS features within our Coarray Fortran programming. To understand the possibilities of coarrays, it might be helpfull to take a look at APGAS languages. One APGAS language is Chapel from Cray and it may serve as a source of good ideas for our Fortran programming too. In Chapel, for an example, they have an empty/full protocol which might serve as a substitute for send/receive in MPI. (Well, that is what we believe, we've never used Chapel and are no MPI experts). While this may require more remote communication than a simple one-sided communication, it might be helpful in certain situations. (Example: using only one coarray object to remotely transfer a collection of local objects). We should be able to implement something quite similar in our Fortran code, using not much more than an additional logical object member and just a small piece of program logic code to accomplish our own empty/full protocol in Fortran. (Nevertheless, we did not implement such a feature ourselves yet.)

2.6 Coarray Teams: MPMD and Scalability

MPMD itself does not seem to be the ideal tool to achieve high levels of scalability. Nevertheless, our MPMD programming style does already allow to group images into teams. With that, scalability can be achieved easily with two obvious strategies (also shown within our example program): Firstly, we can change the number of teams from outside our source code. And secondly, we can size each team independently, i.e. the number of images per team, also from outside our source code. (A better strategy might be to dynamically size the teams from within our program code).
(We took the idea of coarray teams from the upcoming Fortran 2015 standard proposal and do consider this, as well as other ideas from that upcoming Fortran standard, to be essential for doing anything useful with MPMD-like PGAS programming.)

2.7 Avoid built-in Synchronization Methods

Our MPMD-like programming does seem to prohibit any use of Coarray Fortran 2008 built-in synchronizations, except maybe at early stages of program execution, as we do use SYNC ALL and SYNC IMAGES in our example program. (We also tried SYNC IMAGES at a slightly later stage of program execution, but without success). It prohibits allocatable coarrays too, since allocation and deallocation would imply barrier synchronizations between all images with Fortran 2008.

Moreover, built-in methods for synchronizations are rarely required any more within our MPMD-like programming, because more powerful synchronization methods can easily be implemented by the programmer him/herself using simple serial Fortran syntax. That way, the synchronization methods are coded as being between objects rather than being between images, as our example program already shows.

2.8 Error Handling

Error handling is a major issue with parallel programming. With Coarray Fortran, some coding mistakes, that we possibly would expect to generate compile-time errors or at least run-time errors with helpful messages, may turn out to generate hard to resolve run-time failures instead. (For an example: Trying to access a coarray image that does not exist.)
Further, we need stable algorithms and all the good habits from sequential error handling plus some additions. We can't easily restart images, which excludes failed images from further computation in most cases. Therefore we need a global coarray image error management, likely on a distinct image dedicated for that purpose. Error messages and tracing

information on the worker images should be copied into the PGAS memory of these images to allow remote access to such error messages. Also, we possibly need to end program execution always with a controlled ERROR STOP in Coarray Fortran 2008.
Debugging of a MPMD-like parallel application might be a challenge. That's why we added a mechanism to our routines that should help to maintain a stack trace on every image. (These are our *OOOGglob_subSetProcedures* and *OOOGglob_subResetProcedures* routines, that are called from within every routine of our example program).

Nevertheless, we have not implemented a parallel error handler nor a parallel stack trace yet.

3 Description of the Example Program

(This chapter contains text passages colored with blue (like this one) which should be ignored by the reader, since we do not use these features within our example program.)

The intention of our example program is to show, very basically, a programming style that allows MPMD-like parallel programming with Coarray Fortran 2008 already.

3.1 Brief Program Description

The program uses 13 coarray images which it groups into 3 teams, each consisting of 4 images. The remaining image is used independently from the teams for initial management and therefore executes an InitialManager object. Within each team, one of the 4 images is required to execute a TeamManager object whereas the remaining 3 (worker) images do execute a TeamMember object. The number of teams, and thus the number of TeamManager objects, as well as the number of TeamMembers per team, is determined by text files which are provided in advance of program execution. The *TeamManagers.txt* file gives the number of teams, the image number where each TeamManager object shall be executed, as well as an additional file name for each team, for example *TeamMembers_1.txt* for the first team. Further, these *TeamMembers_X.txt* files give the number of TeamMember objects (and thus images) for each team, together with an image number for each of its TeamMember to execute on.
With that, it is easy to change the number of teams as well as the size of each team, using the files. Nevertheless, if you do so, you must also adjust the number of images at runtime. (With ifort you may use the -coarray-num-images option at compile time, and with OpenCoarrays/Gfortran you use the -np option when you start program execution). If the number of coarray images does not match with what the files actually require, program execution may fail.
More details about program execution are given in the *Use Case* section, below in this document.

3.2 Compile and Run the Example Program

To compile with OpenCoarrays/gfortran [3]:
mpifort -fcoarray=lib -L*/home/ms/OpenCoarrays/opencoarrays-1.0.1/src/mpi* OOOGglob_Globals.f90 OOOEerro_admError.f90 OOOPstpa_admStartPath.f90 OOOPimsc_admImageStatus_CA.f90 OOOPtmec_admTeamMember_CA.f90 OOOPtemc_admTeamManager_CA.f90 OOOPimmc_admImageManager_CA.f90 OOOPinmc_admInitialManager_CA.f90 OOOPtmem_admTeamMember.f90 OOOPtema_admTeamManager.f90 OOOPinma_admInitialManager.f90 OOOPimma_admImageManager.f90 Main_Sub.f90 Main.f90 -lcaf_mpi -o a_gfortran.out
(Here, *-L/home/ms/OpenCoarrays/opencoarrays-1.0.1/src/mpi* must be replaced by the actual path to your OpenCoarrays installation).

And to run the OpenCoarrays/gfortran compiled program:
mpirun -np 13 ./a_gfortran.out

To compile the example program using ifort:
ifort -coarray -coarray-num-images=13 OOOGglob_Globals.f90 OOOEerro_admError.f90 OOOPstpa_admStartPath.f90 OOOPimsc_admImageStatus_CA.f90 OOOPtmec_admTeamMember_CA.f90 OOOPtemc_admTeamManager_CA.f90 OOOPimmc_admImageManager_CA.f90 OOOPinmc_admInitialManager_CA.f90 OOOPtmem_admTeamMember.f90 OOOPtema_admTeamManager.f90 OOOPinma_admInitialManager.f90 OOOPimma_admImageManager.f90 Main_Sub.f90 Main.f90 -o a.out

To run the ifort compiled program:
a.out

(By the way, ifort and OpenCoarrays/gfortran can't be used together within the same (Linux Ubuntu) terminal window to compile the example program, possibly due to the use of different MPI implementations. If you want to use both compilers simultaneously on your system, you must open another terminal window for use of each compiler separately).

Another requirement for our example program to execute is a *start.txt* file which contains just the path to the *TeamManagers.txt* and the *TeamMembers_X.txt* files. This path, or working directory, must be placed into quotation marks within the *start.txt* file, and the *start.txt* file itself must be placed in the execution directory of our example program. Further, due to its simplistic design, the program may only execute on shared memory machines.

3.3 Naming Conventions

We use a simple but yet efficient naming convention for the public interface (methods and data declarations) of our Fortran 9x style modules. For an example take the '*OOOPimma_admImageManager.f90'* source file:

The leading four letters (*'OOOP'*, capitalized) are used as placeholders for the namespace (here: *P* stands for the 'Parallel' namespace, other namespaces are *E* for 'Error Handling' and *G* for the 'Globals' namespace; the *O*'s are placeholders, meaning outer scope).

The following four letters (*'imma'*, small letters) are an unique abbreviation (within the namespace) of the Abstract Data Type name (here: *'imma'* stands for *Im*age*Ma*nager). For our coarray wrapper ADT modules, we preserve the last letter to be a '*c*'. Thus the abbreviation of the ImageManager's coarray wrapper is *immc.* With that, we have effectively only the leading three letters for an unique abbreviation of our Abstact Data Type names. (In the case of our *imma* abbreviation, it is the *imm* letters that must be unique).

This simple naming convention already allows to easily identify the membership of every public method or public data declaration. It is also the key to bind a procedure to a specific type with Fortran 9x style. And, more importantly, it turns all our Fortran 9x style code into template code and thus is our main means for code reuse with Fortran 9x. (Nevertheless, our aim here is to show MPMD with Coarray Fortran, thus we did shorten the source code massively. Therefore it possibly can't be used as template code directly.)

The '*_adm*' abbreviation is short for *A*bstract *D*ata Type *M*odule and thus identifies the *.f90* source code file to embrace an Abstract Data Type (ADT).

Routinely, we massively indent error handling and tracing related code, to improve the readability of our program logic code. Our error handling related code starts from column 65.

Our further naming conventions are briefly explained in the header of the source code files, as well as at the end of this document.

3.4 Source Code Files

Our example program consists of 14 source code files:

Main.f90 (the entry point of the application, of course)
Main_Sub.f90 (we use that routinely to avoid program logic code in our *Main.f90*)
OOOGglob_Globals.f90 (contains global definitions of parameters and some routines for tracing)
OOOEerro_admError.f90 (a very primitive sequential error handler)

purely local (non-coarray) ADT modules:
OOOPstpa_admStartPath.f90 (to get the working directory from our start.txt file)
OOOPimma_admImageManager.f90 (the virtual start point of our app, executed by all images)
OOOPinma_admInitialManager.f90 (the InitialManager object, executed on image 1 only)
OOOPtema_admTeamManager.f90 (executed only on the images of the TeamManagers.txt file)
OOOPtmem_admTeamMember.f90 (each TeamManager starts its own set of TeamMember images as stated by its TeamMembers_X.txt file, where X is the TeamManager number)

coarray wrapper ADT modules:
OOOPimsc_admImageStatus_CA.f90 (the only standalone coarray wrapper of our application)
OOOPimmc_admImageManager_CA.f90
OOOPinmc_admInitialManager_CA.f90
OOOPtemc_admTeamManager_CA.f90
OOOPtmec_admTeamMember_CA.f90

Coarray wrappers are only necessary for those purely local ADT modules that actually require remote communication. Thus the *OOOPstpa_admStartPath.f90* source code file does not have a companion coarray wrapper.
On the other hand, the *OOOPimsc_admImageStatus_CA.f90* coarray wrapper is a standalone coarray wrapper (without any purely local ADT module), since it does not require any additional program logic code. Its only aim is to keep the working status of every image within its PGAS memory.

To understand the MPMD-like programming techniques, it is not necessary to few the complete source code, since most of it is repeated framework code. The most relevant program logic code is contained in the *OOOPimma_admImageManager.f90, OOOPinma_admInitialManager.f90, and OOOPtema_admTeamManager.f90* files. That is in a sequential syntax style, with the exception of using THIS_IMAGE() and NUM_IMAGES() there. (Initially, we did also encapsulate access to these into our coarray wrappers. Nevertheless, since the editor (we use the Code::Blocks IDE) is able to highlight THIS_IMAGE() and NUM_IMAGES(), leaving them in the purely local ADT modules makes the code more readable.) As another exception, at startup in our *OOOPimma_Start* routine, we use SYNC ALL and SYNC IMAGES, but not at any later point of program execution.

3.5 Use Case

The following *Use Case* describes basically the parallel execution path of the example program:

1. Initially, on all images the program starts the **ImageManager** [OOOPimma_Start].

2. On image 1 only, the ImageManager starts the InitialManager [OOOPinma_Start]. On all other images, the ImageManager calls its [IIimma_SYNC_CheckActivityFlag] subroutine for synchronization, which initially does nothing on its image except using a do loop to check the value of an **activity flag** [OOOPimscG_intImageActivityFlag_CA] (the 'G' stands for property get) in its image (local) PGAS memory, which is initially set to the state [OOOPimscEnum_ImageActivityFlag % InitialWaiting].

('*_SYNC_'* is short for synchronization routine, '*Enum*' is short for enumeration, and '*II*' stands for private (inner) scope, meaning the routine is not part of the public interface).

3. The **InitialManager** (image 1 only) loads initial data for the TeamManager images [IIinma_LoadTeamManagers] from file [TeamManagers.txt] which must be provided by the user in advance.

4. The InitialManager (image 1 only) activates the **TeamManager images** [IIinma_ActivateTeamManagerImage] as given by the file [TeamManagers.txt]: Firstly, it sets the TeamMembersFileName-property of the (remote) ImageManager on the TeamManager's image [OOOPimmcS_chrTeamMembersFileName_CA] (the 'S' stands for property set). Then it sets the (remote) ImageActivityFlag [OOOPimmcS_intImageActivityFlag_CA] to value [OOOPimscEnum_ImageActivityFlag % TeamManager]. (Here, we use these both property set routines of the coarray wrapper to do remote writes. For an explanation see the *'The Structure of our Coarray Wrapper ADT Modules'* section below.)

4a. At the same time (due to parallelism), the **ImageManager** (on all other images, but not on image 1) still checks the value of the **activity flag** in its image (local) PGAS memory [IIimma_SYNC_CheckActivityFlag] (see 2.). Since that value is now set to [OOOPimscEnum_ImageActivityFlag % TeamManager] on the TeamManager images (see 4.), the ImageManager now starts the **TeamManager** on those Images [OOOPtema_Start].

5. The **TeamManagers** load their TeamMembers data [IItema_LoadTeamMembers] from file [m_chrTeamMembersFileName], [TeamMembers_X.txt] (where X is the TeamManager number, but not its image number). The files must be provided by the user in advance.

6. The **TeamManagers** do activate their **TeamMember images** [IItema_ActivateTeamMemberImage] as given by file [TeamMembers_X.txt]: They set the (remote) ImageActivityFlag [OOOPimscS_intImageActivityFlag_CA] (again, the property set is used to do remote write, see 4.) to value [OOOPimscEnum_ImageActivityFlag % TeamMember].

6a. At the same time, the **ImageManager** (on the remaining idle images) checks the value of the **activity flag** in its image (local) PGAS memory [IIimma_SYNC_CheckActivityFlag] (see 2.). Since that value is now set to [OOOPimscEnum_ImageActivityFlag % TeamMember] on the TeamMember images (see 6.), the ImageManagers now start the **TeamMembers** on those Images [OOOPtmem_Start].

3.6 The Structure of our Coarray Wrapper ADT Modules

(The reader may ignore the blue colored text passages in the following, since we do not use these features within our example program.)

Every purely local (non-coarray) ADT module that requires remote communication, gets its own coarray wrapper ADT module with name ending '*_CA'*. The last letter of the ADT name abbreviation turns into '*c*' (as mentioned above).

The following explanations are a brief overview:

Derived Data Type Definition
The *derived data type* definitions of our coarray wrapper modules are mainly just a clone from their corresponding purely local ADT modules, at least for those object members that are required for remote communication. (That's not always true in our example program, since we did shorten the code to make it easier to survey). Their names are ending with a '*_CA*' and we use them to declare all our coarrays.
(We also use these d*erived data type* definition*s* to declare purely local (non-coarray) objects from it, in cases where we want to use a remote communication channel between two (or more) purely local objects of same type on distinct images. Such a case is possibly not shown in our example program, since we do focus on MPMD, but it is one reason for the current design of our coarray wrappers, namely its (empty) '*Local ADT Routines'* sections. See below for a rationale.)

Corresponding Coarray Declaration
In order to allow easy access to corresponding coarrays, and thus to the remote communication channels, through USE association, we declare all our coarrays in the specification part of our coarray wrappers and call it '*Corresponding Coarray Declaration'*.

Local ADT Routines
As mentioned above, the *derived data type* definitions of our coarray wrappers may also be used to declare purely local (non-coarray) objects in cases where we want to establish a remote communication channel between two purely local objects of same type on distinct images. (That's not shown in our example program since we do focus on MPMD and thus communication with a different type on a remote image. Therefore, the '*Local ADT Routines*' sections are emtpy in our example program.)
The advantage of using such purely local (non-coarray) declarations, instead of directly using the coarray declarations (though an USE statement), is that they provide another layer and thus allow to avoid any direct use of coarrays within our program logic code. That leads to a clean remote interface for our *purely local ADT modules* but works only for remote communication between purely local objects of same type.
(These *local ADT routines* did also prove to be helpful when using derived type coarrays with allocatable components.)

Coarray ADT Routines
The names of our '*Coarray ADT Routines'* do end with *_CA*. This code section does consist of two parts mainly:

- (a) access routines for the coarray type members (property set, property get) and
- (b) management routines for the coarray type as a whole.

(a)
Compared to sequential programming, the access routines for the properties (property set, property get) have an extension for remote communication, the ImageNumber argument, that allows remote read (property get) and remote write (property set) for each single member, in case the actual ImageNumber argument is a remote image number.
An additional property access routine called *CopyImgToImg*, short for copy-image-to-image, encapsulates **three kinds of remote communication** between corresponding coarrays into one single place (see its simple interface to understand):

1. sending data from a local image to a remote image (remote write)
2. getting data from a remote image to the local image (remote read)
3. sending data between two remote images

(b)
The only management routine for the coarray type as a whole is routine CopyCoarrayObjImgToImg, short for copy-coarray-object-image-to-image, which also encapsulates remote write, remote read, as well as sending data between two remote images into one single place.

Error Handling Routines
Our coarray wrappers contain two routines for error handling: The general *ErrorHandler* subroutine is to be used for both, purely local (non coarray) and coarray objects, whereas the *ImageNumberBoundError_CA* function is intended to be used for coarray objects only, to check if an actual image number is valid.

3.7 The Structure of our Purely Local (Non-Coarray) ADT Modules

Our purely local (non-coarray) ADT modules are basically not much different from what we use with sequential programming. Nevertheless, there are some additions to handle parallel programming:

USE statement for the coarray wrapper
As already mentioned above, every purely local (non-coarray) ADT that requires remote communication gets its own coarray wrapper ADT module, associated by a USE statement in the purely local module header.

Corresponding Local Object Declaration
Due to our programming style (i.e. encapsulate all access to coarrays into wrapper objects), it happens that the concept of coarray correspondence does extend to certain of our purely local (non coarray) declarations. We call these '*corresponding local object declarations*' and declare them (according to their coarray counterparts) in the specification part of our local ADT modules. Nevertheless, these are not used in our example program since they are not much useful within our MPMD-like example program. The explanations of our above *Local ADT Routines* section does apply here as well.

Accessing Corresponding Coarrays
We access corresponding coarrays by USE association exclusively.

To establish a remote communication channel between two purely local objects of same type on distinct images, all we need is the USE statement for the type's coarray wrapper at the header of that purely local ADT module. Then, to access that remote communication channel, we may use the *corresponding local object declaration* together with the *local ADT routines* of the coarray wrapper (see above). That is not shown in our example program since we focus on MPMD.

Instead, because we frequently need to establish a remote communication channel between purely local (non-coarray) objects of different type within our MPMD-like programming, we place an USE statement for the remote type's coarray wrapper wherever we need the remote communication channel, and directly use its declared (public) coarray together with its *_CA* routines. For an example see the *IIinma_ActivateTeamManagerImage* subroutine within the *OOOPinma_admInitialManager.f90* module: To access a remote image it uses the coarray wrappers of the ImageManager (immc) and ImageStatus (imsc) types and then calls methods of these coarray wrappers to access the remote type's PGAS memory.

Synchronization Routines
Our synchronization routines are marked with a *'_SYNC_'* naming convention.

The use of *purely local (non-coarray) ADT modules* does allow a sequential-like programming style, even for the parallel logic code. That is also true for our synchronization routines. Nevertheless, these synchronization routines are still the 'parallel heart' of our *purely local ADT modules* and therefore do require a certain degree of 'parallel thinking' while coding them, even if the synchronizations are coded as being between (remote) objects instead of being between images.

A simple example is our *IIimma_SYNC_CheckActivityFlag* synchronization routine of the ImageManager object: Every synchronization routine requires one or more **counterpart routines** on a distinct image (in our case: within a remote object). For the ImageManager's *IIimma_SYNC_CheckActivityFlag* synchronization routine, we already have two counterpart routines: The *IIinma_ActivateTeamManagerImage* routine of the *InitalManager* object, and the *IItema_ActivateTeamMemberImage* routine of the *TeamManager* object. The **Use Case** section above explains how these work in conjunction.

4 A Simple PGAS Load Balancing Technique

We did modify our original MPMD example program slightly, to illustrate a simple PGAS load balancing technique.

Our original MPMD example program did use 13 coarray images to execute:

- an InitialManager object on image 1
- 3 TeamManager objects on images 2, 6, and 10 respectively
- 9 TeamMember objects on images 3, 4, 5, 7, 8, 9, 11, 12, and 13 respectively.

This could easily be altered from outside the source code by modifying the TeamManagers.txt and the TeamMembersX.txt files. Nevertheless, the original example program did require that any of these objects got executed on its own coarray image.

By now, we want to illustrate (very basically) the ease of using coarrays (or PGAS in general) to develop parallel code that might be used equally well for both purposes, even at the same time:
(1) initiate distributed execution on several distinct coarray images (in parallel), as well as
(2) initiate local execution on a single coarray image (sequentially, so to speak).

To achieve this, we did modify few subroutines of the original example program. The modifications are in the following subroutines (as well as in one emulated enumeration) and are marked with the date-stamp *160414* therein:

- Subroutine IIimma_SYNC_CheckActivityFlag (of the ImageManager ADT)
- Enumeration (emulated) OOOPimscEnum_ImageActivityFlag (of the ImageStatus coarray wrapper)
- Subroutine OOOPtmem_Start (of the TeamMember ADT)
- Subroutine OOOPimma_Start (of the InitialManager ADT)
- Subroutine OOOPtema_Start (of the TeamManager ADT)
- Subroutine OOOPimma_Start (of the ImageManager ADT)
- Subroutine IItema_ActivateTeamMember Image (of the TeamManager ADT).

Running the modified example program with the original TeamManagers.txt and TeamMembersX.txt files does lead to the same runtime behavior as with the original code (with the addition that the images will give a 'execution finished' message by now).

For illustration purposes, instead of the 13 coarray images of the original program, assume that we have only 11 coarray images available for doing the same computations. Our Load Balancing example program does allow to reuse the TeamManager images to execute a TeamMember object afterwards. It does also allow to pick up only a fraction of the TeamManager images for that purpose. Since we have 2 coarray images less available, we choose to reuse 2 TeamManager images. Thus, our Load Balancing example program uses 11 coarray images to execute:

- an InitialManager object on image 1
- 3 TeamManager objects on images 2, 6, and 9 respectively
- 9 TeamMember objects on images 3, 4, 5, 6, 7, 8, 9, 10, and 11 respectively.

Effectively, the Load Balancing example program does execute the same computations, no matter if it uses 11 or 13 coarray images. To do so with only 11 images, it does reuse two TeamManager images (6 and 9) to execute TeamMember objects afterwards.

Nevertheless, the crucial point here is not just the simple reuse of coarray images. More importantly, the Load Balancing example program does use the same PGAS parallel programming code to initiate both, the remote execution as well as the local execution of TeamMember objects. To achieve that it uses the ImageStatus_CA coarray wrapper for both, remote and local data transfer through PGAS memory (see subroutine IItema_ActivateTeamMemberImage of the TeamManager ADT).

Of course, this example program is still very primitive, but yet using such a PGAS coding style routinely might be very helpful:

- to free the programmer (algorithm design as well) largely from (early) decisions between purely local or remotely distributed program execution, and thus
- for easy development of dynamic load balancing code later on.

5 Conclusion: Additional Coding Effort for Parallel Programming

The most obvious part of additional coding effort for parallel programming (compared to sequential programming) is the development and maintenance of the coarray wrapper ADT modules. We made the coarray wrappers thin to leave them easy to maintain. Further, the development and maintenance/adjustment of the coarray wrappers is a merely mechanical process since it should not introduce new program logic code regularly.

Our simple example program does only show the very basics of a MPMD-like programming style using Coarray Fortran (2008). Therefore, using our coarray team approach in real-world development will require more framework development, namely a parallel error handler as well as a much more sophisticated coarray image management. Nevertheless, framework development is not uncommon with sequential programming either, possibly just somewhat more is required for MPMD-like parallel programming. Still, we are pretty much confident that's well worth the effort, since we may gain so much more from it, namely solutions for algorithms that might be unthinkable with sequential programming or with pure SPMD either.

Our Further Source Code Naming Conventions and Abbreviations

for scalar members:
- m: ADT member
- S: property set, G: property get,
- CopyImgToImg: copy an ADT member image to image

for array members:
- A: array
- mA: ADT array member
- SA: set array property, GA: get array property,
- CopyAImgToImg: copy an ADT array member image to image

for elements of array members:
- SAElement: set only one array element property
- GAElement: get only one array element property
- CopyAElementImgToImg: copy only one element of an ADT array member image to image

- 99: signals a static array member which has an upper array bound larger than necessary; the upper bound is given by a global parameter

other naming conventions:
- _CA: coarray routine / coarray declaration
- _SYNC_: synchronization routine
- CopyCoarrayObjImgToImg: copy a coarray ADT object image to image
- CopyCoarrayObjFromImg: copy a coarray ADT object from a remote image to the executing image

- DC: deep copy routine
- Enum: enumeration

- OO: public (outer) scope (the two leading namespace letters)
- II: private (inner) scope
- UU: sub-object, object declaration of a nested type

References

[1] Aleksandar Donev. *Rationale for Co-Arrays in Fortran 2008.* ISO/IEC JTC1/SC22/WG5 N1702 (2007).

[2] Don M. J. Browne. *An Implementation of the Coarray Programming Model in C++.* Dissertation: The University of Edinburgh (2013).

[3] Fanfarillo, A., Burnus, T., Cardellini, V., Filippone, S., Nagle, D., & Rouson, D. (2014, October). OpenCoarrays: open-source transport layers supporting coarray Fortran compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (p. 4). ACM.