MPMD and Load Balancing with Fortran 2008 Coarrays (Extended Abstract)

Michael Siehl
https://github.com/MichaelSiehl
miesiehl@gmail.com

ABSTRACT
This paper describes some programming techniques that we use to apply MPMD-style parallel programming with Fortran 2008 coarrays (PGAS). The paper also describes a simple PGAS load balancing technique: we use the exactly same source code for both, sequential as well as parallel execution, even at the same time.

CCS
• Computing methodologies → Parallel computing methodologies → Parallel programming languages • Software and its engineering → Software notations and tools → General programming languages → Language types → Parallel programming languages.

Keywords
Fortran 2008; Coarray Fortran; PGAS; MPMD; load balancing.

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, section 2 may serve as a first preparation to understand our MPMD example program, whereas section 3 explains its code structure in some detail. Section 4 brings some modifications to our example program to illustrate a simple PGAS load balancing technique. The source codes are available through the GitHub repositories at https://github.com/MichaelSiehl

At its core, we do not much more than extending a traditional sequential object-based programming style (Fortran 95 derived data types) with remote communication through PGAS memory (Fortran 2008 coarrays). To do so, we

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

We only need a small fraction of the coarray related language features that Fortran 2008 offers. (Most of these features are tailored to the underlying SPMD programming model and are not useful for MPMD-style programming.) With that, and also due to our coarray wrappers, we can nearly completely retain Fortran's sequential syntax style, even for the implementation of the parallel program logic codes. 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. An important exception to this rule is shown in section 4, where we use the same coarray also as a kind of local saved/static variable to achieve load balancing.

2. CODING REQUIREMENTS FOR MPMD
Our programming style is object-based using Fortran 90/95 style *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 Fortran 2003 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 Using Coarray Wrapper Modules
Initially, we started to encapsulate access to PGAS memory (coarrays) into separate wrapper ADT modules because we wanted to avoid coding errors within our coarray programming. Subsequently, these coarray wrappers turned out to be the simple key to further required coding techniques for doing MPMD-like parallel programming with Fortran 2008. We use these coarray wrappers to establish and use remote communication channels between our 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 *remote communication channels* 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.

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 transferred data.)

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 coarray images only but not to others. This leads to a completely different 'USE' of ADT modules 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 object 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 for 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 Fortran submodules to wrap access to PGAS memory, since a submodule can't be used independently from its parent module).

2.5 Avoid Built-In Blocking (Barrier) Synchronization Methods

Our MPMD-like programming does seem to prohibit any use of Fortran 2008 built-in synchronization methods, except 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 [3]. (On the other hand, we may use coarrays of derived type with allocatable components, but do not use them within our example program yet).

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

Within our synchronization methods, we use DO loops to check values in PGAS memory and believe this to be the simple key for implementing non-blocking (wait-free?) synchronization methods on a regular basis. We use such DO loop synchronization also within our example program, but only for blocking (wait) synchronizations: our example program does nothing else than just checking a flag's value in PGAS memory within a DO loop. Nevertheless, in our real-world MPMD-style programming we are not prevented from executing further tasks within that DO loop, instead of just waiting. The goal is to avoid blocking synchronizations as much as possible.

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).

3. CODE STRUCTURE

This section explains the code structure of our MPMD-style example program.

Coarray wrappers are only necessary for those purely local (non-coarray) ADT modules that actually require remote communication.

On the other hand, we use standalone coarray wrappers (without any purely local ADT module) if no additional (parallel) program logic code is required. That is useful if the only aim is to maintain some status variable within the PGAS memory of coarray images.

3.1 The Code Structure of our Coarray Wrappers

Every purely local (non-coarray) ADT module that requires remote communication, gets its own coarray wrapper ADT module. The following explanations are a brief overview of the coarray wrapper's code sections:

3.1.1 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. We use them to declare all our (derived type) coarrays.

3.1.2 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 this code section '*Corresponding Coarray Declaration*'.

3.1.3 Local ADT Routines

(Readers may ignore this section since we do not use this within our example program yet).

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 on remote communication between objects of distinct types. Therefore, the '*Local ADT Routines*' sections are empty in our example program.)

The advantage of using such purely local (non-coarray) declarations, instead of directly using the coarray declarations (through an USE statement), is that they provide another layer and thus allow to avoid any direct usage 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.

3.1.4 Coarray ADT Routines

This code section consists of two parts mainly:

(1) access routines for the coarray type members (property set, property get) and

(2) (one) management routine(s) for the coarray type as a whole.

(1)

Compared to sequential programming, the access routines for the derived type coarray members (property set, property get) have an extension for remote communication, the ImageNumber argument. This allows remote read (property get) and remote write (property set) for each single member.

An additional member 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

(2)

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.

3.2 The Code 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.

3.2.1 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 ADT module header.

3.2.2 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. (This is not used in our example program since we do 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 access routines.

3.2.3 Synchronization Routines

The use of *purely local (non-coarray) ADT modules* does allow a sequential-like syntax, even for the parallel codes. 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 coarray images.

Every synchronization routine requires one or more *counterpart routines* on a distinct image (in our case: within a remote object).

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 memory in general) to develop parallel code that might be used equally well for the following both purposes, even at the same time:

1. initiate distributed object execution on several distinct coarray images (in truly parallel), as well as
2. initiate purely local (non-distributed) object execution on a single coarray image (sequentially, so to speak).

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. To do so, we did modify our TeamManagers.txt and TeamMembersX.txt files accordingly. Thus, the 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 our TeamMember objects. To achieve that it uses the ImageStatus_CA coarray wrapper for both, remote and local data transfer through PGAS memory. In the first case we use the (derived type) coarray as a remote communication channel between distinct coarray images, whereas in the second case we merely use the same coarray (together with the same PGAS parallel programming code) as a kind of local saved/static variable.

The key is also our simple synchronization technique: we use a simple flag in PGAS memory to signal (and check for) synchronizations between objects but not necessarily between distinct coarray images.

Using such a PGAS coding style routinely might be very helpful:

- to free the programmer (algorithm design as well) greatly from (early) decisions between purely local (non-distributed) object execution versus remotely distributed object execution, and thus
- for easier development of dynamic load balancing code later on.

5. CONCLUSION AND FUTURE WORK

We are (still) developing a MPMD-like programming approach using Fortran 2008 coarrays. Fortran modules play a major role within our approach. We use them to:

- encapsulate access to PGAS memory into coarray wrappers,
- establish and access (by USE association) remote communication channels (corresponding coarrays),
- establish (a kind of) circular module dependencies through PGAS memory to set up a fork/join-like programming style,
- communicate with objects of derived types that are in 'USE' on remote coarray images exclusively.

Besides, the real power of that approach lies in the encapsulation of coarray functionality into the wrapper modules. With that, the programmer can merely focus on development using ordinary sequential language features, even for the parallel program logic codes. Thus, it is possible to turn a sequential programming language into a parallel programming language and at the same time retain the sequential syntax style.

The coarray wrappers are the key: we still need to define some base functionality (e.g. further access routines for remote communication) and thus a kind of standard interface for the wrappers. With that, the coarray wrappers get derived from the purely local types. This process is a merely mechanical one and could even be assigned to a kind of code generator.

In principle, our approach may not be limited to Fortran, provided that another programming language does offer coarrays and modules.

6. REFERENCES

[1] Aleksandar Donev. Rationale for Co-Arrays in Fortran 2008. ISO/IEC JTC1/SC22/WG5 N1702 (2007). ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1702.pdf.

[2] Don M. J. Browne. An Implementation of the Coarray Programming Model in C++. Dissertation: The University of Edinburgh (2013). http://static.ph.ed.ac.uk/dissertations/hpc-msc/2012-2013/An%20implementation%20of%20the%20Coarray%20Programming%20Model%20in%20C++.pdf.

[3] Michael Metcalf, John Reid, and Malcolm Cohen. Modern Fortran Explained. Oxford University Press, Oxford, UK, 2011.