Towards an implementation of Coarrays in GNU Fortran

Toon Moene
The GNU Fortran Team
toon@moene.indiv.nluug.nl

Abstract

Coarrays are an extension of Fortran that has been defined around a decade ago to allow for asynchronous parallel programming within the Fortran language, as opposed to methods outside the language (via the OpenMP standard and the MPI library). It will be part of the upcoming Fortran 2008 Standard.

As the description by one of its authors mentions:

> A Fortran program containing coarrays is interpreted as if it were replicated a fixed number of times and all copies were executed asynchronously. Each copy has its own set of data objects and is called an image. The array syntax of Fortran is extended with additional trailing subscripts in square brackets to give a clear and straightforward representation of access to data on other images. [5]

This paper describes the changes in the parser, intermediate representation and run-time library of GNU Fortran required to implement coarrays.

In addition, it necessarily discusses various paradigms in parallel computing.

1 Introduction

The number of cores in commodity processors is increasing and their clock rates are decreasing. Currently, two cores are the practical minimum available, and four cores are readily available. So, regardless of language support, people will use parallel programming whenever the opportunity arises. They will do it by running multiple jobs in parallel, faster.

However, it would be useful to have programming languages that enable parallel programming; it would even be more useful to augment existing languages to make them suitable for writing parallel programs.

Coarrays (which will be part of the upcoming Fortran 2008 Standard) are intended to be the answer to the above request for the Fortran language.

To put a context to this answer, we provide a limited overview of the current tools for parallellisation that are available to the Fortran programmer, but remain an add-on feature, existing outside of the language.

In the rest of this article we use the concept of *computational domain*. This comprises the whole set of structures containing the data that the computations are being performed on (e.g., the set of arrays that form the model domain in case of a computational fluid dynamics solver).

We also use the concept of *direct addressability*; with this we mean the capability of a CPU to access or store a data item using an address in a register or the instruction, or a combination of both numbers.

2 Existing methods of parallellization for Fortran programmers

2.1 OpenMP and Fortran

The OpenMP model [2] is based on the concept of multiple *threads* each executing on one CPU that is able to directly address all memory (the precise definition of the memory model supported by OpenMP is outside the scope of this article).

The main purpose of OpenMP is to enable distributing iterations of a loop over threads.

A typical example is:

```
      SUBROUTINE A1(N, A, B)
      INTEGER I, N
      REAL B(N), A(N)
!$OMP PARALLEL DO
      DO I=2,N
          B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END PARALLEL DO
      END SUBROUTINE A1
```

The work in the do loop is divided between the processors that are available to the program. All CPUs then perform their part in parallel. The assumption is that each CPU involved can access any element of `A` and `B`. The loop variable `I` is *private* by default, i.e., local to each thread.

2.2 Using the Message Passing Interface Library in Fortran

The MPI model [1] assumes that the computational domain is distributed over a collection of computing *nodes*, in such a configuration that the memory of one node is not directly addressable from another.

The consequence is that, in case data from one node is needed on another node, that data has to be *communicated* using the underlying inter-node communication layer (via *message passing*) before being accessed.

A typical example is:

```
include "mpif.h"      ! MPI definitions
integer myid, ierr
real :: p(10,10)      ! A 10x10 array
  ...                 ! on each node
call MPI_INIT(ierr)   ! Initialize MPI
call MPI_COMM_RANK(MPI_COMM_WORLD,&
 & myid, ierr)        ! Which node am I?
if (myid==0) then
   read(*,*) p        ! Read on node 0
endif
call MPI_BCAST(p,100,MPI_REAL,0,&
 & MPI_COMM_WORLD,ierr) ! Broadcast to
  ...                   ! other nodes
call MPI_FINALIZE(ierr) ! Terminate MPI
```

This shows how to read data on one image and distribute it to the rest.

3 Coarrays

Coarrays are an effort to abstract "non-local"-ness. They provide a syntactical way to specify that certain parts of the computational domain are not "local", i.e., not directly addressable.

3.1 Images

This is done by defining the notion of an *image*. An image is an abstraction of what is commonly known as a node in a cluster. Inside the node there can be multiple processors (for instance, a quad-core CPU), but the defining characteristic is that an image is a unit of direct addressability: All items (array elements, scalars) "local" to it are addressable directly, while all items "remote" (those with *codimensions*, denoted by *cosubscripts* in square brackets) might have to be accessed via an indirect method of exchange (left unspecified by the Standard) between images.

The mapping between images and hardware is unspecified, so multiple images may be implemented on shared memory hardware using multithreading, turning the codimensions into normal trailing dimensions and using the synchronization primitives for enforcing the order of computation.

Execution on all images is started before the first executable Fortran statement is reached; there is no synchronisation between images unless explicitly requested by the programmer via one of the synchronization primitives, or caused by the other events mentioned in paragraph 3.4.

3.2 Coarray notation

A typical example is:

```
integer i
real :: p(10,10)[*]    ! A 10x10 array
  ...                  ! on each image
sync all               ! Synchronize
                       ! all images
if (this_image()==1) then
   read (*,*) p        ! Read on image 1
   do i = 2, num_images()
     p(:,:)[i] = p     ! Then broadcast
   end do              ! to the others
end if
sync all
  ...
```

This shows how to read data on one image and distribute it to the rest. Note the implicit "exchange" when broadcasting.

In the declaration of a coarray, the last codimension should be "*". If it is the only codimension, it denotes "on each image"; if it is not, the array elements are distributed over the images as outlined in paragraph 3.3.

Coarrays can be declared as above, or *allocated* - in the following manner:

```
real, allocatable :: a(:)[:], s[:,:]
  ...
allocate ( a(10)[*], s[-1:34,0:*] )
```

Note that the upper bound of the last codimension must be "*", because it depends on the number of images, which is specified outside the program.

Note also that `s` is a *coscalar*.

3.3 Cosubscripts and image index

The relation between the *image index* and a set of cosubscripts can be complicated. To quote [5], paragraph 2:

> Data on other images are normally referenced by cosubscripts enclosed in square brackets. Each set of cosubscripts maps to an image index, which is an integer between one and the number of images, in the same way as a set of array subscripts maps to a position in array element order.
>
> The number of images may be retrieved through the intrinsic function `num_images()`. On each image, the image index is available from the intrinsic function `this_image()` with no arguments. The set of subscript indices that correspond to the current image for a coarray `z` are available as `this_image(z)`. The image index that corresponds to a set of cosubscript indices `sub` for a coarray `z` is available as `image_index(z,sub)`. For example, on image 5, for the array declared as
>
> ```
>   real :: z(10,20)[10,0:9,0:*]
> ```
>
> `this_image()` has the value 5 and `this_image(z)` has the value `(/5,0,0/)`. For the same example on image 213, `this_image(z)` has the value `(/3,1,2/)`. On any image, the value of `image_index(z,(/5,0,0/))` is 5 and the value of `image_index(z,(/3,1,2/))` is 213.

3.4 Synchronization

In addition to the new non-local array syntax, coarray Fortran also adds the following synchronization primitives:

- **`all stop`**. Stops computation on all images as soon as practical. This denotes an error condition.
- **`sync all`**. Synchronizes all images.
- **`sync images`**. Synchronizes a given set of images.
- **`sync memory`**. Causes cache, registers and communication buffers to be flushed to be able to make their content available to other images.
- **`critical ... end critical`**. Ensures that only one image is executing this block of code at any one time.

Synchronization is also caused by:

- `ALLOCATE` or `DEALLOCATE` statements that allocate or deallocate a coarray.
- `END`, `END BLOCK` (a Fortran 2008 construct), or `RETURN` statement that involves an implicit deallocation of a coarray.
- `END PROGRAM` or `STOP` statement.

3.5 Auxiliary intrinsics

The following additional intrinsics deal with the number of images and the index of same:

- **`num_images()`**. Gives the number of images.

- **`this_image(a)`**. Gives the set of cosubscript indices corresponding to the current image for coarray `a`. Without argument, it gives the image index of the current image.
- **`image_index(a,sub)`**. Gives the image index for coarray `a` corresponding to the set of cosubscript indices `sub`.

In the next sections we will discuss the changes to the Fortran front end and its run time library necessary to implement coarray Fortran.

4 Additions to the Fortran Internal Representation

The internal representation of arrays in the Fortran front end is given by the type `gfc_array_spec` (defined in file `gfortran.h`). It has to be augmented to support codimensions; this could be accomplished as follows (with the new field `corank` added).

```
/* Array specification.  */
typedef struct
{
  int rank;
  int corank;
  array_type type;
  struct gfc_expr
    *lower[GFC_MAX_DIMENSIONS],
    *upper[GFC_MAX_DIMENSIONS],
  ...
}
gfc_array_spec;
```

An array with corank 0 is a local array.

The codimensions could just be added at the end of the regular dimensions in the arrays `lower` ane `upper`. As the sum of rank and corank cannot be larger than 15, `GFC_MAX_DIMENSIONS` has to be changed from 7 to 15 in file `libgfortran.h`.

5 Additions to the Fortran Parser

In file `array.c` the various forms of array access are parsed. These routines have to be augmented to recognize the codimensions in square brackets ([]).

In addition, the following new statements have to be parsed; this all starts with "matching" them in file `match.c`:

```
all stop, sync all, sync images,
sync memory, critical ... end critical
```

The first four can be treated like STOP (routine `gfc_match_stop`). The fifth one looks (syntactically) like a simple form of a `do ... end do` block (see routine `gfc_match_do`).

6 The translation into GIMPLE

6.1 The format of array descriptors

During execution of a Fortran program, array descriptors give the code a unified view on the layout of arrays.

These descriptors are built by the function `gfc_get_array_type_bounds` in file `trans-array.c`.

As discussed before (for the case of the internal representation of arrays) the codimensions could just be added after the regular dimensions. A new macro `GFC_TYPE_COARRAY_RANK` whould have to be added to store the corank. Rank and corank determine the interpretation of the following lower and upper bounds (whether they refer to regular dimensions or codimensions).

6.2 The translation of array expressions

The main translation work for arrays (scalarization, allocation, initialization) is performed by the code in file `trans-array.c`.

6.2.1 Scalarization

The *scalarizer* is the part of the Fortran front end that breaks down array expressions and assignments to their constituent loops.

To describe how, it is best to first quote the comment in file `trans-array.c` that gives an overview of its functionality:

> The function gfc_conv_loop_setup generates the scalarization setup code. It determines the range of the scalarizing loop variables. If a temporary is required, this is created and initialized. Code for scalar expressions taken outside the loop is also generated at this time. Next the offset and scaling required to translate from loop variables to array indices for each term is calculated.
>
> A call to gfc_start_scalarized_body marks the start of the scalarized expression. This creates a scope and declares the loop variables. [...]
>
> The scalar gfc_conv_* functions are then used to build the main body of the scalarization loop. Scalarization loop variables and precalculated scalar values are automatically substituted. Note that gfc_advance_se_ss_chain must be used, rather than changing the se->ss directly.
>
> For assignment expressions requiring a temporary two sub loops are generated. The first stores the result of the expression in the temporary, the second copies it to the result. A call to gfc_trans_scalarized_loop_boundary marks the end of the main loop code and the start of the copying loop. The temporary may be less than full rank.
>
> Finally gfc_trans_scalarizing_loops is called to generate the implicit do loops. The loops are added to the pre chain of the loopinfo. The post chain may still contain cleanup code.

So function gfc_conv_loop_setup has to be augmented to deal with any communication of array sections, if necessary, to gather the constituent parts of the array expression.

For example:

```
real :: a(10)[*], b(10)[*]
integer :: n
   ...
print*,a(:) + b(:)[n]
```

This will cause, on every image, to be printed: the ten numbers that are the sum of the corresponding elements of a (locally) and b (on the image with index n).

To be able to compute this on a particular image, array b has to be transferred from image n to that other image (except - of course - on image n).

For assignment expressions, fanning out of the results has to be added to the copying loops generated after the call to function gfc_trans_scalarized_loop_boundary.

For example:

```
real :: a(10)[*], b(10)[*]
integer :: n
   ...
a(1:5)[1:n-1] = sin( b(1:10:2)[n] )
```

The sine of the five odd-numbered elements of array b on image n are transferred to the first five elements of array a on the images 1 .. n-1.

The computation takes place on image n. The results are then broadcast to images 1 .. n-1. Nothing happens on images n+1 .. *.

The rules that govern the distribution of a coarray over images are outlined in paragraph 3.3.

6.2.2 Allocation

The code for allocation of arrays is generated by function gfc_array_allocate. It has to be augmented to construct the correct array descriptor with corank and codimensions for coarrays.

The rules that govern the distribution of a coarray over images are outlined in paragraph 3.3.

6.2.3 Initialization

Initialization must be local. This means that the compiler doesn't have to be augmented for performing non-local initialization, but it must capture new, illegal, forms of initialization:

```
real :: a(10)[*]
data a(1)    /0.0/     ! Permitted
data a(1)[2] /0.0/     ! Not permitted
```

and

```
real :: a(10)[*] = 0.0 ! Not permitted
```

6.3 The translation of the synchronization primitives

Translation of these statements (which should result in a run time library call) will be performed by code in file `trans-stmt.c`. Except for `critical ... end critical` the pattern to be followed is that of function `gfc_trans_stop`, which translates the STOP statement.

The latter statement can be treated as a block construct like a simple `do ... end do` without a while condition (see function `gfc_trans_do_while`).

6.4 The translation of the auxiliary intrinsics

The translation of `num_images`, `this_image`, and `image_index` is straightforward and will most probably be done inline.

7 Additions to the Fortran Run Time Library

The run time library must be enhanced by

1. A routine to get the desired number of images and one to set the local image index.
2. A routine to start processing on all images. This routine must be called before the processing of Fortran executable statements begins.
3. A routine to perform **`all stop`**. (Note that changes to the handling of **`stop`** are also necessary).
4. Routines to perform communication between images.
5. Routines to perform the synchronization primitives among images. (for simplicity, initially **`sync all`** can be coded as a special case of **`sync images`**).

8 Support of multiple memory models

Perhaps an initial implementation of coarrays in GNU Fortran could be made just for shared memory architectures; this would greatly simplify the first implementation. Later extensions could then deal with distributed memory systems, either by using GASnet routines [3] (like Berkeley Unified Parallel C [4]) to implement the communication and synchronization, or by supplying specialized routines.

Alternatively, we could aim for a complete implementation based on GASnet and develop the optimized-for-shared-memory version later.

Finally, the choice between memory models could be based on a compiler switch, or ideally, an environment variable.

9 Acknowledgements

The author thanks the following persons for suggestions to improve this paper: Dan Nagle, John Reid, Bill Long, Tobias Burnus, and Bernhard Fischer.

For the description of coarray Fortran, the author relied - next to a draft of the upcoming Fortran 2008 Standard - heavily on a tutorial [5] by John Reid, one of the authors of the original coarray proposal.

The author is also indebted to the other members of the GNU Fortran Team, for providing Fortran-using professionals with an outstanding compiler and run time library.

References

[1] Various authors. The Message Passing Interface Standard, version 2.0, 1997. `http://www.mpi-forum.org/docs/mpi-20.ps`.

[2] Various authors. The OpenMP Standard, version 2.5, 2005. `http://www.openmp.org/mp-documents/spec25.pdf`.

[3] Various authors. Global Address Space Networking, 2007. `http://gasnet.cs.berkeley.edu/`.

[4] Various authors. Berkeley Unified Parallel C, 2008. `http://upc.lbl.gov/`.

[5] John K. Reid. Coarrays in the next Fortran Standard, 2008. `ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1724.pdf`.