New Parallel Programming Features in

Intel® (Visual) Fortran Composer XE

by Steve Lionel,
Developer Products Division
Fortran programmers have been doing parallel processing for many years using methods outside the Fortran standard such as auto-parallelization, OpenMP, and MPI. Fortran 2008, approved as an international standard in late 2010, brought parallel programming into the language for the first time with not one, but two language features. (Of course, you can’t be parallel with just one.)

This article provides a brief overview of these two new features, DO CONCURRENT and coarrays. The former is pretty easy to get one’s head around; the latter is not.

**DO CONCURRENT**

Back in the early 1990s, an attempt was made at extending the Fortran 90 language for high performance computing and parallel processing. Called High Performance Fortran or HPF, it attempted to build on Fortran 90’s array syntax in a way that permitted array operations to be done in parallel. HPF introduced the FORALL and WHERE constructs, PURE procedures with no side effects, and a number of intrinsic procedures for operations such as scatter/gather. While HPF was not widely adopted, some pieces of it were incorporated into the Fortran 95 standard, approved in 1997.

Of the various HPF features that persisted, none was perhaps more misunderstood than FORALL. Here’s an example of a FORALL construct:

```fortran
REAL :: A(10, 10), B(10, 10) = 1.0
…
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0)
A(I, J) = REAL(I + J + 2)
B(I, J) = A(I, J) + B(I, J) * REAL(I * J)
END FORALL
```

The parenthesized list after the FORALL keyword is called the forall-header. It has one or more forall-triplets that specify the range of values taken on by the index-name. In this example we have two forall index names, I and J, which are each specified to take values from 1 to 10. The increment, if not specified by the third element in the triplet, is 1, just as in Fortran 90 array notation. The last part is the mask expression that determines the conditions under which the FORALL construct body (the two assignments) is executed.

Many Fortran programmers looked at FORALL and saw a loop, or in this case, two nested loops, perhaps with an IF at the top that skips to the next iteration, if the expression is false. But FORALL is not a loop construct, it is a “masked array assignment.” If you try to think of this as a loop, you might expect each iteration to execute both assignments, and that these could be done in parallel. But that’s not how FORALL was defined. Instead, the first assignment is executed completely, across all combinations of all the index names, filtered by the mask. Then, the second assignment is executed completely, again across all combinations and filtered by the mask. Inside a FORALL construct, an assignment statement may reference functions if they are PURE, but the only statement types allowed in a FORALL are assignment statements, WHERE constructs, or other FORALLs.

FORALL was a noble experiment, but the rules were too restrictive to be amenable to doing the assignments in parallel and it did not meet the needs of the Fortran community. So, Fortran 2008 brings what I call “FORALL Done Right”: DO CONCURRENT.

A DO CONCURRENT construct looks like a blend of traditional DO and FORALL. In fact, the beginning of a DO CONCURRENT uses the FORALL header syntax. For example:

```fortran
DO CONCURRENT (I=1:N)
T = A(I) + B(I)
C(I) = T + SQRT(T)
END DO
```

As with FORALL, the mask is optional. If present, it reduces the set of active combinations of the index names to those where the mask expression is true. Unlike FORALL, each range of a DO CONCURRENT is an iteration and is executed independently for all the active index combinations.

There are some restrictions on what you can have in a DO CONCURRENT. For example, you can’t RETURN or GO TO out of the construct, and you can’t reference a variable that is defined or made to be undefined by another iteration. You can even do I/O in a DO CONCURRENT, so long as a record written by one iteration is not read by another. As with FORALL, any procedure called from within the construct must be PURE (which guarantees that it has no side effects). Note that it is the programmer’s responsibility to ensure that there are no dependencies between loop iterations—the compiler is not required to check these for you.

DO CONCURRENT is supported as of Intel® [Visual] Fortran Composer XE 2011 and the compiler will attempt to execute the construct in parallel if you have enabled auto-parallelization (/Qparallel or -parallel). However, there is no guarantee that any particular DO CONCURRENT will be run in parallel, and, of course, the order in which the iterations run is unpredictable. As a side effect, use of DO CONCURRENT can also help with automatic vectorization, as you are guaranteeing that there are no loop-carried dependencies.
Coarrays

If you are an MPI programmer, you know the basic drill: collect some data, call MPI_SEND to send it to a copy of your program running on another "node," and then use MPI_RECV to get results back. (This is a simplification, of course.) Wouldn't it be nice to be able to "reach out and touch" the other copies of your program using normal Fortran syntax, and not have to worry about adding calls to move data around?

Coarray Fortran, first proposed in the 1990s as an extension of Fortran 90 called F- -F minus minus), provides simple syntax for adding parallelism to a Fortran program. (The syntax is simple, though the definition and implementation is not.) It was implemented by Cray for its T3E and X1 supercomputers in the early 2000s, and was added, in a modified and somewhat reduced form, into the Fortran 2008 standard. Intel released the first full implementation of Fortran 2008's coarrays for mainstream computers in the Intel (Visual) Composer XE 2011 release for Linux* and Windows.*

The fundamental concepts of Coarray Fortran are these:

> Image: Multiple copies of your application run in parallel; each is called an image.
> Coarray: Variables become coarrays when they are given the CODIMENSION attribute. Somewhat confusingly, scalars can also be coarrays – the standard defines a coarray as any entity with a non-zero corank, and these can be scalars or arrays. Codimensions (and coindices) are denoted with square brackets [ ].

Coarrays are split up across all the images of your application, so that a portion of each coarray resides in the local memory of an individual image. This property is associated with the Partitioned Global Address Space (PGAS) parallel programming concept. Here, coarrays exist in a shared "address space," but image-specific segments are individually addressable. Let's look at a simple example.

We will declare an array A with dimension 10x20 and with one codimension:

```
real, dimension(10,20), codimension[*] :: A
```

It helps if you think of codimensions as additional dimensions, and indeed the Fortran standard limits the sum of the number of dimensions and codimensions to fifteen. (Intel® Fortran supports 31 as an extension.) The last upper cobound in the codimension must be *, at runtime this takes on the value of the number of images. If when run there are eight images, the cobounds of A are 1:8.

As with dimensions, you can have multiple codimensions with lower and upper bounds, and as with dimensions, only the last one may have * as an upper bound. So we might have:

```
integer, codimension[4,2:6,3:*] :: B
```

When you reference a coarray, you can do so with or without the coindices, which are enclosed in square brackets. If no coindices are present, you are referencing your image's piece of the coarray. If the coindices are present, you are specifying the coindex of the image you want.

Now, at this point you might be asking what happens if there aren't enough images to fill up the coindices, just as you would with a regular array that's an error. Unlike a regular array, the "shape" of a coarray may be ragged. Using the B example above, 20 images are needed to fill in each "layer" of the coarray. If there are, say, 39 images, there is a coindex [3,6,4], but not [4,6,4]. (Remember that Fortran does things in column-major order where the left subscript varies the fastest.)

Intrinsic procedures are provided to allow you to find the number of images, index of your own image, and the cobounds of any coarray.

What makes coarrays so nice is that they are integrated thoroughly into the Fortran language. You can use a coarray in most places where a regular variable is allowed, such as:

- Expressions and assignments
- Arguments to procedure calls
- I/O statements

This makes a program using coarrays look clean. For example, consider a Jacobian solver that breaks up the problem into blocks of data. Most of the calculation involves an image's local block, but at the edges of each block it needs to consider values from "halo cells," those on the edge of adjacent image's chunks. Here's what such code might look like using coarrays: (Figure 1)

Fortran defines additional coarray behaviors that ease programming. For example:

- You can have ALLOCATABLE coarrays (and in fact this is the most common usage), where every allocation is a synchronization point, to make sure that all images have allocated their coarrays consistently and completely.
- All images can do I/O. Normally, each has its own set of unit numbers, but the language says that "standard output" (unit 6) is preconnected on all images. While an implementation is not required to "merge the streams," Intel Fortran does, so all standard output writes get displayed on the console where the image is run. "Standard input" (unit 5) is preconnected on image 1 only.
- Every image has an implicit synchronization point at its start and again at its end.

```
my_subgrid( 0, 1:my_M) = my_subgrid( my_N, 1:my_M)[my_north_P,me_Q]
my_subgrid( my_N+1, 1:my_M) = my_subgrid( 1, 1:my_M)[my_south_P,me_Q]
my_subgrid( 1:my_N, my_M+1) = my_subgrid( 1:my_N, 1 )[me_P, my_east_Q]
my_subgrid( 1:my_N, 0 ) = my_subgrid( 1:my_N, my_M)[me_P, my_west_Q]
```

Figure 1
“With Intel® [Visual] Fortran Composer XE you get coarray support in a “shared memory” mode, running on a single system. To build a coarray program just add the -coarray (or /Qcoarray) compiler option and then run the executable as normal. No special configuration is required.”

The language provides several methods of synchronization among images. The SYNC ALL statement causes all images to wait until all of them have executed that SYNC ALL the same number of times. SYNC MEMORY makes sure that all memory updates have completed before continuing. SYNC IMAGES is like SYNC ALL, but you restrict the synchronization to a specified set of images.

There are also locks, declared using the LOCK_TYPE defined in intrinsic module ISO_FORTRAN_ENV, and LOCK and UNLOCK operations on these. Lastly, there is the ability to do atomic (uninterrupted) reads and writes of integer and logical variables through the ATOMIC_DEFINE and ATOMIC_REF intrinsic procedures (these last are newly supported as of Intel® Fortran Composer XE 2013).

With Intel® [Visual] Fortran Composer XE you get coarray support in a “shared memory” mode, running on a single system. To build a coarray program just add the -coarray (or /Qcoarray) compiler option and then run the executable as normal. No special configuration is required. To add support for a “distributed memory” model across a cluster requires that you also have a license for Intel® Cluster Studio (in addition to having a cluster). Yes, this applies to Windows clusters too. (Support for distributed-memory coarray applications on Windows was added in Update 6 of Intel® Visual Fortran Composer XE 2011).

For further reading about Fortran 2008, including coarrays and DO CONCURRENT, you can refer to the following documents from the Fortran standards committee:

- COARRAYS IN THE NEXT FORTRAN STANDARD
- THE NEW FEATURES OF FORTRAN 2008
- FORTRAN 2008 STANDARD