Appendix D

Fortran quick reference

Fortran [1] is the oldest high-level programming language, and was developed to facilitate the translation of math formulae into machine code. Fortran was the first major language to use a compiler to translate from a high-level program representation to assembly language. Due to its age, it carries certain arcane baggage that later programming languages have evolved away from. As a result, Fortran has fallen into disfavor in certain programming circles.

However, with the introduction of Fortran 90 (and later revisions to the standard), Fortran is a truly modern programming language. Much of the arcane baggage associated with it is actually no longer part of the language used by modern programmers, and exists in a deprecated form simply to support legacy codes. It is now modular and has many object-oriented features. Furthermore, Fortran should be of interest to those studying parallel programming because of its functional and data parallel constructs and because of the coarray notation introduced after Fortran 2003. Unlike languages like C and C++, which are considered to be more modern, Fortran has become a truly parallel language with features added to recent language standards. With coarrays, Fortran provides a mechanism to run replicas (called images) of a program on many different processors (or cores) at the same time. It also provides language constructs that allow the program to both read from and write to data residing on *remote* processors and to synchronize the different program images so that access to the data by the multiple images can be done atomically.

Considering the data parallel features introduced in Fortran 90 and 95, with the coarray syntax adopted in the post-2003 standards, Fortran is an important modern language to look at in the context of concurrent programs.

[1] Originally named FORTRAN for FORmula TRANslation language.

D.1 Fortran syntax

Programmers will find that Fortran is similar to C at the statement and expression level, though the syntax is different, especially for declaring functions and variables. Fortran is a rather large language and here we consider only those language elements that are specifically related to parallel programming. We will focus on two aspects of Fortran in this context: coarray notation, which is *explicitly* parallel, and data-parallel constructs which are *implicitly* parallel. Data parallelism is a high-level abstraction that is, at the same time, both easier to program and gives the compiler more leeway (if fully exploited) in retargeting a program to different computer architectures, including those with multiple cores.

D.1.1 Data-parallel and functional features

Arrays are first class citizens in Fortran as they are part of the language type system rather than just a pointer to raw memory as in C. A Fortran array contains metadata associated with it that describes the array's shape and size. This metadata is lacking in other languages, such as C, C++, and Java. Even in the case where multidimensional array data structures are available (such as in C++ or Java), they exist outside the language standard, limiting the ability of the compiler to analyze them fully. The existence of this metadata allows the use of array notation whereby explicit indexing of arrays is not required. For example, if `A`, `B`, and `C` are all arrays of the same rank and shape, then the statement `C = A + B` results in the sum of elements of `A` and `B` being stored in the corresponding elements of `C`. The first element of `C` will contain the value of the first element of `A` added to the first element of `B`.

The use of array notation allows one to program in a data-parallel subset of Fortran. This style of programming makes use of array notation and pure and elemental functions to operate on array elements. A "pure" function is one without side effects (though it may change the value of array elements in an array argument). An elemental function is a pure function with the additional constraint that it takes only scalar arguments and must return a scalar. Though defined in terms of scalars, surprisingly, an elemental function may take an array as an actual argument, with the compiler applying the function to each element of the array, and returning an array result. The plus operator as described in the previous paragraph can be seen as an elemental function, as the plus operator is defined in terms of scalars but can be applied to whole arrays and can return an array result.

Because elemental functions return scalar values and are free from side effects, the compiler is free to distribute the computation over any hardware processing elements available to it, such as the multiple cores and vector units

Listing D.1: An example of an elemental function.

```
elemental real function third(x)
  real, intent(in) :: x
  third = x/3.0
end function halve
```

Listing D.2: Data parallel solution to Laplace's equation in one dimension.

```
pure subroutine laplace(T, lb, rb)
  include "one_half.h"
  real, intent(in) :: lb, rb
  real, dimension(:), intent(inout) :: T

  T = one_half(  EOSHIFT(T, shift=-1, boundary=lb) &
               + EOSHIFT(T, shift= 1, boundary=rb) &
              )

end subroutine laplace
```

on an Intel or AMD processor or the Synergistic Processing Units on an IBM Cell processor.

An example of an elemental function is shown in Listing D.1. This example simply returns its input divided by 2. While this may seem like a trivial example, such simple functions may be composed with other elemental functions to perform powerful computations, especially when applied to arrays.

The use of an elemental function is shown in the pure procedure `laplace` in Listing D.2. This function (for example) can be used to solve the steady-state temperature along a one-dimensional surface. The temperature at any point along the surface is equal to the average of the temperature at neighboring points. Rather than writing explicit loops and using indexing to reference neighboring points, one may simply shift the array `T` to the left and right, add the results and divide by 2. The intrinsic function `EOSHIFT` (end-off shift) semantically returns an array copy with the value at any element shifted by the amount given by the `shift` argument. The value shifted in at the boundary is supplied by the `boundary` argument. The elemental function `one_half` is used to divide by 2.

Note that semantically the elemental functions return their array results by value. This is important because had loops been used to express the `laplace` procedure, temporary variables would have been required. Otherwise, `T(i)` would be updated with new values of `T(i-1)`.

There are several advantages to this style of programming:

- There are no loops or index variables to keep track of. Off by one index errors are a common programming mistake.

- The written code is closer to the algorithm, easier to understand, and is usually substantially shorter.

- Semantically the intrinsic functions return arrays by value. This is usually what the algorithm requires.

- Because pure and elemental function are free from side effects, it is easier for a compiler to schedule the work to be done in parallel.

Complete and very concise and elegant programs can be built with procedures similar to the `laplace` example shown above. To aid this effort, Fortran supplies intrinsic functions like the array constructors (`CSHIFT`, `EOSHIFT`, `MERGE`, `TRANSPOSE`, ...), the array location functions (`MAXLOC` and `MINLOC`), and the array reduction functions (`ANY`, `COUNT`, `MINVAL`, `SUM`, `PRODUCT`, ...).

D.2 Coarrays

Coarray Fortran is a small extension to the Fortran language introduced by Numrich and Reid in the late-1990s [77]. It was designed as a minimal set of extensions to Fortran to provide for concurrency and parallel computation. It extends the type system so that specially declared scalar, array, and user-defined types may be shared between communicating processes using a new syntax.

Coarray Fortran is a member of the Partitioned Global Address Space (PGAS) family of languages. As such, memory is not shared amongst processes; rather, the global address space of a coarray program is partitioned amongst the set of collaborating processes making up the running program. The array syntax of Fortran was extended so that coarray variables residing in the address space of any one process, can be referenced by any other process using square bracket notation.

The minimalist philosophy of Coarray Fortran is from the perspective of the language and compiler, not from the perspective of the writer of a program. Coarray Fortran places a heavy responsibility on the programmer to write a correct program free of race conditions. The programmer must take care to ensure that any interactions between processes are carefully synchronized by hand in order to avoid correctness problems. The compiler is free to treat a program as if it were executing in a single process.

Listing D.3: An example of a simple coarray program.

```
PROGRAM COARRAY_EXAMPLE

   real, allocatable :: T(:)[*]

   ALLOCATE( T(100)[*] )

   T = 3.0

   if (THIS_IMAGE() == 2)
      T(1)[2] = 2.0
   end if

   SYNC ALL

   T(100) = T(1)[2]

END PROGRAM
```

D.2.1 Coarray notation

The data-parallel notation discussed above is *implicitly* parallel. No additional and explicit parallel syntax is required for this style of programming. All of the burden of parallelization is placed on the compiler, not on the programmer. With the adoption of coarray syntax, Fortran has also become an *explicit* parallel language. Coarrays extend the Fortran language in the following ways:

- Introduction of a small set of keywords to annotate program code with information related to parallelism, particularly related to synchronization.
- New syntax using square bracket notation `[ ]` to reference remote memory.
- Additional intrinsic library routines that take coarray arguments.
- Runtime support so that a program image can be run on separate processing elements.

Listing D.3 shows a small (and essentially useless) coarray program. Coarray T is declared on line 3 and memory for the coarray allocated on line 5. In Fortran the use of parentheses associated with an array reference indicates the local portion of the array; the square brackets make reference to the remote portion of the coarray. Thus there are 100 array elements allocated on

every program image on line 5 as denoted by the `(100)`. One can't explicitly declare the number of processors to use within a program because this is a runtime dependent parameter. Thus the symbol `*` is used in place of an explicit number to denote the size of the codimension, much the way that `:` is used to denote an unknown (as yet) number of array elements in the array declaration on line 3.

Coarrays can be used as if they were local arrays if the square bracket notation is left off of the array reference, as in the assignment statement at line 7. This statement assigns the value 3.0 to every element of the local array. Since this statement is executed on every image, every element of the array *on every image* will be initialized with the value 3.0.

A key feature of coarrays is the ability to reference memory that is nonlocal. The assignment statement at line 10 states that the first element of array `T` on image 2 should be assigned with the value of 2.0. Note that one must be careful to indicate which image is to perform the assignment (as is done with the if construct beginning at line 9, otherwise all images will attempt to perform this work concurrently. This is not only inefficient but the attempt to do so is forbidden by the language standard (though it may not be caught by the compiler).

"Then", at line 15, every image assigns the last element of `T` with the value "previously" stored in the first element of `T` on image 2. Then and previously are used with quotes here to emphasize that without the `SYNC ALL` statement at line 13, there would be a race condition and `T(100)` would be 3.0 on some images and 2.0 on others. The synchronization ensures that *every* image reaches line 13 before *any* image executes line 15.

An example of a coarray program is shown in Listing D.4. The coarray variable `T` is declared in line 2 as a real coarray with one local dimension, denoted by `(:)` and one codimension, denoted by `[*]`. By using square brackets to represent the codimension, it is always easy to tell when remote memory is being referenced.

This example is quite a bit more complicated that the earlier `laplace` example. There is an explicit loop (lines 9–12) to update individual elements of the interior of the array and a temporary variable (`tmp`) must be used to ensure that the update is done using only previous values. In addition, boundary values must be explicitly obtained from neighboring processors (lines 16–19) and care must be taken to synchronize the processors to avoid race conditions.

D.2.2 Concurrency model

The concurrency model in Coarray Fortran is that of multiple processes with no shared state that communicate asynchronously via message passing. Each process (called an image) runs the same program and all points of communication between processes are made explicit using square bracket notation `[ ]`.

Listing D.4: Coarray program solving a simple heat flow equation in one dimension.

```
subroutine co_advance(T, lb, rb)
  real :: T(:)[*], lb, rb, tmp
  integer :: i, me

  integer :: left_boundary  = lb
  integer :: right_boundary = rb

  tmp = T( LBOUND(T) )
  do i = LBOUND(T)+1, UBOUND(T)-1
    T(i) = ( tmp + T(i+1) ) / 2.0
    tmp = T(i)
  end do

  me = THIS_IMAGE()

  if (me /= CO_LBOUND(T)) &
    left_boundary  = T(UBOUND(T))[me-1]
  if (me /= CO_UBOUND(T)) &
    right_boundary = T(LBOUND(T))[me+1]

  SYNC ALL
  T(lower) = (left_boundary + T(lower) &
                            + T(lower+1))  / 3.0
  T(upper) = (T(upper-1) + T(upper)    &
                         + right_boundary) / 3.0
  SYNC ALL

end subroutine co_advance
```

The Fortran runtime is responsible for replicating the program on each processor (or core) and starting up the program. The runtime is also responsible for file IO, communication, and synchronization of processes as specified explicitly by the programmer.

D.2.3 Memory model

Coarray Fortran memory is partitioned by image number and is globally addressable. It is the responsibility of the programmer to ensure that memory accesses are sequentially consistent, so that all memory accesses appear as if they occur in the sequential order specified by the program. A memory conflict occurs if two images access the same memory location concurrently and at least one of them is a write.

To avoid memory conflicts, image control statements are provided by the language to synchronize memory access. These include, the `SYNC ALL`, `SYNC MEMORY` and `CRITICAL` statements. In addition, the `ALLOCATE` and `DEALLOCATE` statements (when used with a coarray variable), and the `STOP` and `END PROGRAM` statements also cause synchronization across participating processes. The Fortran language provides no guarantees of memory consistency other than through the use of these image control (synchronization) statements.

D.2.4 Memory Allocation

Coarray memory allocation is provided via the `ALLOCATE` statement as shown earlier in Listing D.3. Coarray memory allocation and deallocation statements imply synchronization across all images and therefore must be called concurrently by all processes. In addition, the coarray allocation size (bounds) must be the same on all images.

While a common, coarray allocation size allows for certain compiler optimizations, it is somewhat limiting for algorithms exhibiting task parallelism. All concurrent tasks must allocate the same coarray data structures even though they may not ultimately need them. To get around this limitation, all tasks need to allocate a coarray of user-defined type, but only those tasks needing access to the data need allocate *local* memory contained within the data type. Ultimately, however, this limitation makes Fortran better suited for algorithms exhibiting some form of data parallelism.

D.3 Fortran intrisic functions

In addition to the coarray extensions that Fortran provides beyond the base language, a set of intrinsic libraries have been proposed (though not yet

officially adopted as of 2008) are provided to operate on coarrays. These include (for example) the array location functions (`CO_MAXLOC` and `CO_MINLOC`), and the array reduction functions (`CO_ANY`, `CO_COUNT`, `CO_MINVAL`, `CO_SUM`, `CO_PRODUCT`).

For example, the following function `global_sum` returns the sum of all of the elements of the input array `array`:

```
REAL FUNCTION global_sum(array)
  REAL, INTENT(IN) :: array(:,:)[*]
  REAL, SAVE       :: temp[*]
  temp = SUM(array)    ! local sum on the executing image
  CALL CO_SUM(temp, global_sum)
END FUNCTION
```

A benefit to Fortran users is that these coarray intrinsic functions are specified by the language unlike similar functionality in the MPI standard. This means that compilers are able to perform certain compiler optimizations (like code motion) that would otherwise not be allowed for functions whose semantics are not specified by the language.

D.3.1 Locking

As discussed in the main text of this book, it is not unusual for programmers to come upon algorithmic instances where it is necessary to require a mechanism to protect access to some data structure that is visible to multiple processes (Fortran images). As often this cannot easily be achieved solely by the synchronization operations discussed above, Fortran provides a new data type for lock variables and a set of operations to act upon them.

A lock variable is a scalar variable of intrinsic type `TYPE(LOCK_TYPE)`. These lock variables behave like typical locks, although unlike in some other languages, lock variables do not need to be explicitly initialized. Standard locking and unlocking operations are provided by the `LOCK` and `UNLOCK` statements. The semantics of these operations are precisely what would be expected. If the lock is available when the `LOCK` statement is executed, the caller acquires the lock and is allowed to proceed. Otherwise, the statement blocks until the image that holds it releases it via the `UNLOCK` statement.

In addition, Fortran provides an easy mechanism to limit the execution of a block of code one image *at a time.* This is accomplished via a critical section, as demonstrated below:

```
CRITICAL
  ! block of code to be executed serially
END CRITICAL
```

D.4 History

Coarray Fortran evolved out of work on the F−− programming language developed for the CRAY-T3D computer in the early 1990s. F−− was developed as a language-based alternative to the SHMEM, one-sided message-passing library developed by CRAY. Coarray notation can therefore be thought of as syntactic sugar for a one-sided get/put messaging layer like SHMEM or MPI-2. However, since the coarray syntax is incorporated into the Fortran language, it fits within the type system of the language, is known by the compiler, and can be more efficient that pure library implementations of message passing.

Coarray Fortran has been standardized by the Fortran J3 standards body. A basic subset of coarrays will appear in the Fortran 2008 standard, while other features, such as collective operations on coarrays will appear shortly thereafter.

D.5 Further information

This appendix was written based on the Fortran 2008 standard. Please check the Fortran documentation if you are using a more recent versions of the standard. New standards usually appear every 5–10 years.

- Official home of Fortran standards: `http://www.nag.co.uk/SC22WG5/`
- Fortran J3 Web Page: `http://j3-fortran.org/`