Using Arrays Efficiently

This topic discusses how to efficiently access arrays and pass array arguments.

Accessing Arrays Efficiently

Many of the array access efficiency techniques described in this section are applied automatically by the Intel Fortran loop transformation optimizations. Several aspects of array use can improve run-time performance; the following sections discuss the most important aspects.

Perform the fewest operations necessary

The fastest array access occurs when contiguous access to the whole array or most of an array occurs. Perform one or a few array operations that access all of the array or major parts of an array instead of numerous operations on scattered array elements. Rather than use explicit loops for array access, use elemental array operations, such as the following line that increments all elements of array variable `A`:

Example

```
A = A + 1
```

When reading or writing an array, use the array name and not a DO loop or an implied DO-loop that specifies each element number. Fortran 95/90 array syntax allows you to reference a whole array by using its name in an expression.

For example:

Example

```
REAL ::  A(100,100)
A = 0.0
A = A + 1            ! Increment all elements
                     ! of A by 1
...
WRITE (8) A          ! Fast whole array use
```

Similarly, you can use derived-type array structure components, such as:

Example

```
TYPE X
  INTEGER A(5)
```

```
END TYPE X
...
TYPE (X) Z
WRITE (8)Z%A        ! Fast array structure
                    !  component use
```

Access arrays using the proper array syntax

Make sure multidimensional arrays are referenced using proper array syntax and are traversed in the natural ascending storage order, which is column-major order for Fortran. With column-major order, the leftmost subscript varies most rapidly with a stride of one. Whole array access uses column-major order.

Avoid row-major order, as is done by C, where the rightmost subscript varies most rapidly. For example, consider the nested `DO` loops that access a two-dimension array with the `J` loop as the innermost loop:

Example

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO I=1,3                   ! I outer loop varies slowest
  DO J=1,5                 ! J inner loop varies fastest
    X (I,J) = Y(I,J) + 1   ! Inefficient row-major storage order
  END DO                   ! (rightmost subscript varies fastest)
END DO
...
END PROGRAM
```

Since `J` varies the fastest and is the second array subscript in the expression `X (I,J)`, the array is accessed in row-major order. To make the array accessed in natural column-major order, examine the array algorithm and data being modified. Using arrays `X` and `Y`, the array can be accessed in natural column-major order by changing the nesting order of the `DO` loops so the innermost loop variable corresponds to the leftmost array dimension:

Example

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO J=1,5                   ! J outer loop varies slowest
  DO I=1,3                 ! I inner loop varies fastest
    X (I,J) = Y(I,J) + 1   ! Efficient column-major storage order
  END DO                   ! (leftmost subscript varies fastest)
END DO
...
```

```
END PROGRAM
```

The Intel Fortran whole array access (`X = Y + 1`) uses efficient column major order. However, if the application requires that `J` vary the fastest or if you cannot modify the loop order without changing the results, consider modifying the application to use a rearranged order of array dimensions. Program modifications include rearranging the order of:

- Dimensions in the declaration of the arrays `X(5,3)` and `Y(5,3)`
- The assignment of `X(J,I)` and `Y(J,I)` within the `DO` loops
- All other references to arrays `X` and `Y`

In this case, the original `DO` loop nesting is used where `J` is the innermost loop:

Example

```
INTEGER  X(5,3), Y(5,3), I, J
Y = 0
DO I=1,3                  ! I outer loop varies slowest
  DO J=1,5                ! J inner loop varies fastest
    X (J,I) = Y(J,I) + 1  ! Efficient column-major storage order
  END DO                  ! (leftmost subscript varies fastest)
END DO
...
END PROGRAM
```

Code written to access multidimensional arrays in row-major order (like C) or random order can often make inefficient use of the CPU memory cache. For more information on using natural storage order during record, see Improving I/O Performance.

Use available intrinsics

Use the available Fortran 95/90 array intrinsic procedures rather than create your own.

Whenever possible, use Fortran 95/90 array intrinsic procedures instead of creating your own routines to accomplish the same task. Fortran 95/90 array intrinsic procedures are designed for efficient use with the various Intel Fortran run-time components.

Using the standard-conforming array intrinsics can also make your program more portable.

Avoid leftmost array dimensions

With multidimensional arrays where access to array elements will be noncontiguous, avoid leftmost array dimensions that are a power of two (such as 256, 512).

Since the cache sizes are a power of 2, array dimensions that are also a power of 2 may make inefficient use of cache when array access is noncontiguous. If the cache size is an exact multiple of the leftmost dimension, your program will probably make use of the cache less efficient. This does not apply to contiguous sequential access or whole array access.

One work-around is to increase the dimension to allow some unused elements, making the leftmost dimension larger than actually needed. For example, increasing the leftmost dimension of A from 512 to 520 would make better use of cache:

Example

```
REAL A (512,100)
DO I = 2,511
  DO J = 2,99
    A(I,J)=(A(I+1,J-1) + A(I-1, J+1)) * 0.5
  END DO
END DO
```

In this code, array A has a leftmost dimension of 512, a power of two. The innermost loop accesses the rightmost dimension (row major), causing inefficient access. Increasing the leftmost dimension of `A to 520 (REAL A (520,100))` allows the loop to provide better performance, but at the expense of some unused elements.

Because loop index variables `I` and `J` are used in the calculation, changing the nesting order of the `DO` loops changes the results.

For more information on arrays and their data declaration statements, see the *Intel® Fortran Language Reference*.

Passing Array Arguments Efficiently

In Fortran, there are two general types of array arguments:

- Explicit-shape arrays (introduced with Fortran 77); for example, `A(3,4)` and `B(0:*)`

  These arrays have a fixed rank and extent that is known at compile time. Other dummy argument (receiving) arrays that are not deferred-shape (such as assumed-size arrays) can be grouped with explicit-shape array arguments.

- Deferred-shape arrays (introduced with Fortran 95/90); for example, `C(:.:)`

  Types of deferred-shape arrays include array pointers and allocatable arrays. Assumed-shape array arguments generally follow the rules about passing deferred-shape array arguments.

When passing arrays as arguments, either the starting (base) address of the array or the address of an array descriptor is passed:

- When using explicit-shape (or assumed-size) arrays to receive an array, the starting address of the array is passed.

- When using deferred-shape or assumed-shape arrays to receive an array, the address of the array descriptor is passed (the compiler creates the array descriptor).

Passing an assumed-shape array or array pointer to an explicit-shape array can slow run-time performance. This is because the compiler needs to create an array temporary for the entire array. The array temporary is created because the passed array may not be contiguous and the receiving (explicit-shape) array requires a contiguous array. When an array temporary is created, the size of the passed array determines whether the impact on slowing run-time performance is slight or severe.

The following table summarizes what happens with the various combinations of array types. The amount of run-time performance inefficiency depends on the size of the array.

	Dummy Argument Array Types (choose one)	
Actual Argument Array Types (choose one)	**Explicit-Shape Arrays**	**Deferred-Shape and Assumed-Shape Arrays**
Explicit-Shape Arrays	Result when using this combination: Very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional.	Result when using this combination: Efficient. Only allowed for assumed-shape arrays (not deferred-shape arrays). Does not use an array temporary. Passes an array descriptor. Requires an interface block.
Deferred-Shape and Assumed-Shape Arrays	Result when using this combination: When passing an allocatable array, very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional. When not passing an allocatable array, not efficient. Instead use allocatable arrays whenever possible. Uses an array temporary. Does not pass an array descriptor. Interface block optional.	Result when using this combination: Efficient. Requires an assumed-shape or array pointer as dummy argument. Does not use an array temporary. Passes an array descriptor. Requires an interface block.