Introduction to Modern Fortran

Advanced Array Concepts

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

November 2007

Summary

This will describe some advanced array features
Use them only when you need their facilities

It will also cover some aspects of array use
Important for correctness and performance

There is a lot more on both

- Please ask if you need any help

Higher Rank Constructors

Constructors create only rank one arrays
We shall now see how to construct higher ranks

It is done by constructing a rank one array
And then mapped using the RESHAPE function

This is very easy, but looks a bit messy

The RESHAPE Intrinsic (1)

This allows arbitrary restructuring of arrays
The following is only its very simplest use

```
RESHAPE (source, shape)
```

source provides the data in array element order
shape specifies the shape of array to deliver

The RESHAPE Intrinsic (2)

```
REAL, DIMENSION(3, 4) :: array

array = RESHAPE( (/ 1.1, 2.1, 3.1, 1.2, 2.2, &
      3.2, 1.3, 2.3, 3.3, 1.4, 2.4, 3.4 /), (/ 3, 4 /) )
```

Is functionally equivalent to:

```
DO m = 1, 3
      DO n = 1, 4
            array(m, n) = m+0.1*n
      END DO
END DO
```

The RESHAPE Intrinsic (3)

It can be used in initialisation expressions

```
REAL, DIMENSION(3, 4) :: array = &
    RESHAPE( (/ 1.1, 2.1, 3.1, 1.2, 2.2, &
        3.2, 1.3, 2.3, 3.3, 1.4, 2.4, 3.4 /), (/ 3, 4 /) )
```

It also allows arbitrary reordering
And padding with copies of an array

See the references for more details

Example

Create the zero vector, and the three unit vectors

```
REAL, DIMENSION(1:3), PARAMETER :: &
     vec_0 = (/ 0.0, 0.0, 0.0 /), &
     vec_i = (/ 1.0, 0.0, 0.0 /), &
     vec_j = (/ 0.0, 1.0, 0.0 /), &
     vec_k = (/ 0.0, 0.0, 1.0 /)
```

Create the identity matrix

```
REAL, DIMENSION(1:3, 1:3), PARAMETER :: &
   identity = RESHAPE( (/ vec_i, vec_j, vec_k /), (/ 3, 3 /) )
```

RESHAPE More Generally

It isn't restricted to multi–dim. constants
You can use it for fancy array restructuring

- Study the specification before doing that

Restructuring arrays is dangerous territory

- And there are several other such intrinsics

I.e. ones with important uses but no simple uses

Vector Indexing (1)

Vectors may be used as indices

```
INTEGER, DIMENSION(1:5) :: &
        j = (/ 3, 1, 5, 2, 4 /), k = (/ 2, 3, 2, 1, 3 /)
REAL, DIMENSION(1:5) :: x, &
        y = (/ 1.2, 2.3, 3.4, 4.5, 5.6 /)
x(j) = y(k)
PRINT *, y(k)
PRINT *, x
```

```
2.3000000  3.4000001  2.3000000  1.2000000  3.4000001
3.4000001  1.2000000  2.3000000  3.4000001  2.3000000
```

Vector Indexing (2)

Using vector indices is a bit like sections
There are important differences – be careful

You can them for reading arrays quite safely
Elements must be distinct if updating

- NOT recommended for use in arguments

If used in arguments, those must not be updated
And it forces the compiler to copy the array

Masked Assignment (1)

Set all negative values in an array A to zero

```
REAL, DIMENSION(20, 30) :: array

DO j = 1,30
    DO k = 1,20
        IF (array(k,j) < 0.0) array(k,j) = 0.0
    END DO
END DO
```

But the WHERE statement is more convenient

```
WHERE (array < 0.0) array = 0.0
```

Masked Assignment (2)

It has a statement construct form, too

```
WHERE (array < 0.0)
      array = 0.0
ELSE WHERE
      array = 0.01*array
END WHERE
```

Masking expressions are LOGICAL arrays

You can use an actual array there, if you want

Masks and assignments need the same shape

Masked Assignment (3)

Fortran 2003 extends it considerably

Don’t use LHS arrays in non–elemental functions
The following is asking for trouble:

```
WHERE (arr1 < arr2)
    arr1 = 1.0
ELSE WHERE
    arr2 = sum(arr1)
END WHERE
```

- Don’t bother with the FORALL statement

Memory Efficiency (1)

Local arrays can be implemented in many ways
Only a few Ada compilers handle them properly

You can exhaust your program’s stack with them
Too big, or too many due to deep recursion

- It will usually cause a truly horrible crash

Allocatable arrays always go on the ‘heap’
Automatic arrays often go on the ‘heap’
That is less efficient, but is handled much better

- Making all big arrays allocatable isn’t stupid

Memory Efficiency (2)

As always, every solution has its own problems
Lots of allocation and deallocation isn't ideal

- Each (de)allocation costs some CPU time

Not generally a problem for Fortran programs

- Poor compilers may have memory leaks

Most Fortran compilers don't have them badly

Both because of the language's restrictions

Memory Efficiency (3)

- The big problem is memory fragmentation
Describing how and why is beyond this course
Luckily, in AD 2007, there is a simple solution

- Best one is to use 64–bit addressing
Gets rid of the worst of the problems, painlessly
I do that, even on systems with 2 GB of memory

- Please ask if you want to know more

Order of Evaluation (1)

Array assignments etc. are like implicit loops
But, except in I/O, no order of evaluation implied
Also the behaviour is different when modifying

- Each pass of a loop is executed in order
- Array assignments do it all “in parallel”

- You should avoid code where it matters

The compiler may have to copy the array
It risks confusion when tuning your code

Order of Evaluation (2)

```
INTEGER, DIMENSION(5) :: array = (/ 1, 2, 3, 4, 5 /)
array(2:5) = array(1:4)
PRINT *, array

array = (/ 1, 2, 3, 4, 5 /)
DO k = 1,4
    array(k+1) = array(k)
END DO
PRINT *, array
```

```
1    1    2    3    4
1    1    1    1    1
```

Performance (1)

- Efficient use of arrays is critical

This course has NOT taught any of that
It covers quite enough without adding it!

- Generally, follow this procedure:

Start by writing clean and clear code
Get it working, and test it fairly thoroughly
If too slow, use a profiler to see where
And only then tune only those aspects

Performance (2)

You get most gain by using faster methods
Followed by the following aspects:

- Improve the layout and access patterns

This is locality (improved cache usage etc.)

- Avoid unnecessary array copying

Compilers often have to do that for some codes
Some compilers copy when they don't need to

- Improve the actual CPU efficiency

This is getting into advanced tuning

Memory Locality (1)

Things used together should be stored together
Remember that ‘‘first index varies fastest’’

```
REAL, DIMENSION(3000, 5000) :: array
DO n = 1, 5000
    DO m = 1, 3000
        array(m, n) = m+0.1*n
    END DO
END DO
```

- Note that the first index varies fastest

Memory Locality (2)

Sections and masking can cause trouble

```
REAL, DIMENSION(1000, 1000) :: array
CALL FRED( array(123, :) )
```

The elements of the vector are a long way apart
A problem if FRED accesses it a lot

- Consider making a temporary copy of it

Access Patterns

- Sequential access is generally efficient

Avoid non–sequential access whereever possible

- This can be much slower than sequential

```
REAL, DIMENSION(1000) :: arr1, arr2
INTEGER, DIMENSION(1000) :: random
arr1(random) = arr2(random)
```

Unnecessary Copying (1)

It is hard to describe when this may occur
It helps if you can mentally compile the code

- Avoiding using the LHS array on the RHS
Except when the uses are purely elemental

- Generally, sections do not need a copy
Unlike arguments with vector indexed arrays

- Compilers often do unnecessary copying
In a very bad case, even for `CALL Fred(data(:))`

Example

```
INTEGER :: arr1(1:50), arr2(1:100), arr3(1:100)
REAL, DIMENSION(20, 20) :: mat1, mat2, mat3
```

These shouldn’t require a copy

```
arr1 = arr1+arr2(1:50)+arr3(arr2(51:100))
mat1 = MATMUL(mat2, mat3)
```

But these almost certainly will

```
arr1 = arr1(::–1)+arr2(1:50)
mat1 = MATMUL(mat1, mat2)
```

Unnecessary Copying (2)

And, while this shouldn't, ...

```
mat1 = mat1 + MATMUL(mat2, mat3)
```

There is more on this under procedures

- Generally, don't worry unless you have to
If your program runs fast enough, who cares?

- If not, time and profile it first
Ask for advice if you have problems

High-Performance Problems

There are some other problems some people hit
Too complicated to even describe here

- Ignore them until you have problems

Then ask for help with tackling them

Buzzwords and phrases include:

TLB thrashing
Cache conflicts
False sharing
Memory banking

Reminder

- You don’t have to remember all of this
- Start by using the simplest features only
- Use the fancy ones only when you need them

If you know they exist, you can look them up