

LANGUAGE SUPPORT FOR PERFORMANCE - FORTRAN 90*

Charles Severance

Kevin Dowd

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0[†]

The previous American National Standards Institute (ANSI) FORTRAN standard release, FORTRAN 77 (X3.9-1978), was written to promote portability of FORTRAN programs between different platforms. It didn't invent new language components, but instead incorporated good features that were already available in production compilers. Unlike FORTRAN 77, FORTRAN 90 (ANSI X3.198-1992) brings new extensions and features to the language. Some of these just bring FORTRAN up to date with newer languages like C (dynamic memory allocation, scoping rules) and C++ (generic function interfaces). But some of the new features are unique to FORTRAN (array operations). Interestingly, while the FORTRAN 90 specification was being developed, the dominant high performance computer architectures were scalable SIMD systems such as the Connection Machine and shared-memory vector-parallel processor systems from companies like Cray Research.

FORTRAN 90 does a surprisingly good job of meeting the needs of these very different architectures. Its features also map reasonably well onto the new shared uniform memory multiprocessors. However, as we will see later, FORTRAN 90 alone is not yet sufficient to meet the needs of the scalable distributed and nonuniform access memory systems that are becoming dominant at the high end of computing.

The FORTRAN 90 extensions to FORTRAN 77 include:

- Array constructs
- Dynamic memory allocation and automatic variables
- Pointers
- New data types, structures
- New intrinsic functions, including many that operate on vectors or matrices
- New control structures, such as a WHERE statement
- Enhanced procedure interfaces

1 FORTRAN 90 Array Constructs

With FORTRAN 90 array constructs, you can specify whole arrays or array sections as the participants in unary and binary operations. These constructs are a key feature for "unserializing" applications so that they are better suited to vector computers and parallel processors. For example, say you wish to add two vectors, A and B. In FORTRAN 90, you can express this as a simple addition operation, rather than a traditional loop. That is, you can write:

*Version 1.3: Aug 25, 2010 10:39 am -0500

[†]<http://creativecommons.org/licenses/by/3.0/>

```
A = A + B
```

instead of the traditional FORTRAN 77 loop:

```
DO I=1,N
  A(I) = A(I) + B(I)
ENDDO
```

The code generated by the compiler on your workstation may not look any different, but for some of the parallel machines available now and workstations just around the corner, the difference are significant. The FORTRAN 90 version states explicitly that the computations can be performed in any order, including all in parallel at the same time.

One important effect of this is that if the FORTRAN 90 version experienced a floating-point fault adding element 17, and you were to look at the memory in a debugger, it would be perfectly legal for element 27 to be already computed.

You are not limited to one-dimensional arrays. For instance, the element-wise addition of two two-dimensional arrays could be stated like this:¹

```
A = A + B
```

in lieu of:

```
DO J=1,M
  DO I=1,N
    A(I,J) = A(I,J) + B(I,J)
  END DO
END DO
```

Naturally, when you want to combine two arrays in an operation, their shapes have to be compatible. Adding a seven-element vector to an eight-element vector doesn't make sense. Neither would multiplying a 2×4 array by a 3×4 array. When the two arrays have compatible shapes, relative to the operation being performed upon them, we say they are in *shape conformance*, as in the following code:

¹Just in case you are wondering, $A*B$ gives you an element-wise multiplication of array members— not matrix multiplication. That is covered by a FORTRAN 90 intrinsic function.

```
DOUBLE PRECISION A(8), B(8)
...
A = A + B
```

Scalars are always considered to be in shape conformance with arrays (and other scalars). In a binary operation with an array, a scalar is treated as an array of the same size with a single element duplicated throughout.

Still, we are limited. When you reference a particular array, *A*, for example, you reference the whole thing, from the first element to the last. You can imagine cases where you might be interested in specifying a subset of an array. This could be either a group of consecutive elements or something like "every eighth element" (i.e., a non-unit stride through the array). Parts of arrays, possibly noncontiguous, are called *array sections*.

FORTRAN 90 array sections can be specified by replacing traditional subscripts with triplets of the form *a:b:c*, meaning "elements *a* through *b*, taken with an increment of *c*." You can omit parts of the triplet, provided the meaning remains clear. For example, *a:b* means "elements *a* through *b*"; *a:* means "elements from *a* to the upper bound with an increment of 1." Remember that a triplet replaces a single subscript, so an *n*-dimension array can have *n* triplets.

You can use triplets in expressions, again making sure that the parts of the expression are in conformance. Consider these statements:

```
REAL X(10,10), Y(100)
...
X(10,1:10) = Y(91:100)
X(10,:)   = Y(91:100)
```

The first statement above assigns the last 10 elements of *Y* to the 10th row of *X*. The second statement expresses the same thing slightly differently. The lone " : " tells the compiler that the whole range (1 through 10) is implied.

2 FORTRAN 90 Intrinsic

FORTRAN 90 extends the functionality of FORTRAN 77 intrinsics, and adds many new ones as well, including some intrinsic subroutines. Most can be *array-valued*: they can return arrays sections or scalars, depending on how they are invoked. For example, here's a new, array-valued use of the *SIN* intrinsic:

```
REAL A(100,10,2)
...
A = SIN(A)
```

Each element of array *A* is replaced with its sine. FORTRAN 90 intrinsics work with array sections too, as long as the variable receiving the result is in shape conformance with the one passed:

```

REAL A(100,10,2)
REAL B(10,10,100)
...
B(:, :, 1) = COS(A(1:100:10, :, 1))

```

Other intrinsics, such as `SQRT`, `LOG`, etc., have been extended as well. Among the new intrinsics are:

Reductions: FORTRAN 90 has vector reductions such as `MAXVAL`, `MINVAL`, and `SUM`. For higher-order arrays (anything more than a vector) these functions can perform a reduction along a particular dimension. Additionally, there is a `DOT_PRODUCT` function for the vectors.

Matrix manipulation: Intrinsics `MATMUL` and `TRANPOSE` can manipulate whole matrices.

Constructing or reshaping arrays: `RESHAPE` allows you to create a new array from elements of an old one with a different shape. `SPREAD` replicates an array along a new dimension. `MERGE` copies portions of one array into another under control of a mask. `CSHIFT` allows an array to be shifted in one or more dimensions.

Inquiry functions: `SHAPE`, `SIZE`, `LBOUND`, and `UBOUND` let you ask questions about how an array is constructed.

Parallel tests: Two other new reduction intrinsics, `ANY` and `ALL`, are for testing many array elements in parallel.

3 New Control Features

FORTRAN 90 includes some new control features, including a conditional *assignment primitive* called `WHERE`, that puts shape-conforming array assignments under control of a mask as in the following example. Here's an example of the `WHERE` primitive:

```

REAL A(2,2), B(2,2), C(2,2)
DATA B/1,2,3,4/, C/1,1,5,5/
...
WHERE (B .EQ. C)
  A = 1.0
  C = B + 1.0
ELSEWHERE
  A = -1.0
ENDWHERE

```

In places where the logical expression is `TRUE`, `A` gets `1.0` and `C` gets `B+1.0`. In the `ELSEWHERE` clause, `A` gets `-1.0`. The result of the operation above would be arrays `A` and `C` with the elements:

```

A =  1.0  -1.0          C =  2.0   5.0
    -1.0  -1.0          1.0   5.0

```

Again, no order is implied in these conditional assignments, meaning they can be done in parallel. This lack of implied order is critical to allowing SIMD computer systems and SPMD environments to have flexibility in performing these computations.

4 Automatic and Allocatable Arrays

Every program needs temporary variables or work space. In the past, FORTRAN programmers have often managed their own scratch space by declaring an array large enough to handle any temporary requirements. This practice gobbles up memory (albeit virtual memory, usually), and can even have an effect on performance. With the ability to allocate memory dynamically, programmers can wait until later to decide how much scratch space to set aside. FORTRAN 90 supports dynamic memory allocation with two new language features: automatic arrays and allocatable arrays.

Like the local variables of a C program, FORTRAN 90's automatic arrays are assigned storage only for the life of the subroutine or function that contains them. This is different from traditional local storage for FORTRAN arrays, where some space was set aside at compile or link time. The size and shape of automatic arrays can be sculpted from a combination of constants and arguments. For instance, here's a declaration of an automatic array, B, using FORTRAN 90's new specification syntax:

```
SUBROUTINE RELAX(N,A)
  INTEGER N
  REAL, DIMENSION (N) :: A, B
```

Two arrays are declared: A, the dummy argument, and B, an automatic, explicit shape array. When the subroutine returns, B ceases to exist. Notice that the size of B is taken from one of the arguments, N.

Allocatable arrays give you the ability to choose the size of an array after examining other variables in the program. For example, you might want to determine the amount of input data before allocating the arrays. This little program asks the user for the matrix's size before allocating storage:

```
INTEGER M,N
REAL, ALLOCATABLE, DIMENSION (:,:) :: X
...
WRITE (*,*) 'ENTER THE DIMENSIONS OF X'
READ (*,*) M,N
ALLOCATE (X(M,N))
...
do something with X
...
DEALLOCATE (X)
...
```

The ALLOCATE statement creates an $M \times N$ array that is later freed by the DEALLOCATE statement. As with C programs, it's important to give back allocated memory when you are done with it; otherwise, your program might consume all the virtual storage available.

5 Heat Flow in FORTRAN 90

The heat flow problem is an ideal program to use to demonstrate how nicely FORTRAN 90 can express regular array programs:

```

PROGRAM HEATROD
PARAMETER(MAXTIME=200)
INTEGER TICKS,I,MAXTIME
REAL*4 ROD(10)
ROD(1) = 100.0
DO I=2,9
  ROD(I) = 0.0
ENDDO
ROD(10) = 0.0
DO TICKS=1,MAXTIME
  IF ( MOD(TICKS,20) .EQ. 1 ) PRINT 100,TICKS,(ROD(I),I=1,10)
  ROD(2:9) = (ROD(1:8) + ROD(3:10) ) / 2
ENDDO
100 FORMAT(I4,10F7.2)
END

```

The program is identical, except the inner loop is now replaced by a single statement that computes the "new" section by averaging a strip of the "left" elements and a strip of the "right" elements.

The output of this program is as follows:

```

E6000: f90 heat90.f
E6000:a.out
  1 100.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 21 100.00 82.38 66.34 50.30 38.18 26.06 18.20 10.35  5.18  0.00
 41 100.00 87.04 74.52 61.99 50.56 39.13 28.94 18.75  9.38  0.00
 61 100.00 88.36 76.84 65.32 54.12 42.91 32.07 21.22 10.61  0.00
 81 100.00 88.74 77.51 66.28 55.14 44.00 32.97 21.93 10.97  0.00
101 100.00 88.84 77.70 66.55 55.44 44.32 33.23 22.14 11.07  0.00
121 100.00 88.88 77.76 66.63 55.52 44.41 33.30 22.20 11.10  0.00
141 100.00 88.89 77.77 66.66 55.55 44.43 33.32 22.22 11.11  0.00
161 100.00 88.89 77.78 66.66 55.55 44.44 33.33 22.22 11.11  0.00
181 100.00 88.89 77.78 66.67 55.55 44.44 33.33 22.22 11.11  0.00
E6000:

```

If you look closely, this output is the same as the red-black implementation. That is because in FORTRAN 90:

$$\text{ROD}(2:9) = (\text{ROD}(1:8) + \text{ROD}(3:10)) / 2$$

is a *single* assignment statement. As shown in Figure 1 (Data alignment and computations), the right side is completely evaluated before the resulting array section is assigned into `ROD(2:9)`. For a moment, that might seem unnatural, but consider the following statement:

$$I = I + 1$$

We know that if I starts with 5, it's incremented up to six by this statement. That happens because the right side ($5+1$) is evaluated before the assignment of 6 into I is performed. In FORTRAN 90, a variable can be an entire array. So, this is a red-black operation. There is an "old" ROD on the right side and a "new" ROD on the left side!

To really "think" FORTRAN 90, it's good to pretend you are on an SIMD system with millions of little CPUs. First we carefully align the data, sliding it around, and then— wham— in a single instruction, we add all the aligned values in an instant. Figure 1 (Data alignment and computations) shows graphically this act of "aligning" the values and then adding them. The data flow graph is extremely simple. The top two rows are read-only, and the data flows from top to bottom. Using the temporary space eliminates the seeming dependency. This approach of "thinking SIMD" is one of the ways to force ourselves to focus our thoughts on the data rather than the control. SIMD may not be a good architecture for your problem but if you can express it so that SIMD could work, a good SPMD environment can take advantage of the data parallelism that you have identified.

The above example actually highlights one of the challenges in producing an efficient implementation of FORTRAN 90. If these arrays contained 10 million elements, and the compiler used a simple approach, it would need 30 million elements for the old "left" values, the old "right" values, and for the new values. Data flow optimization is needed to determine just how much extra data must be maintained to give the proper results. If the compiler is clever, the extra memory can be quite small:

Data alignment and computations

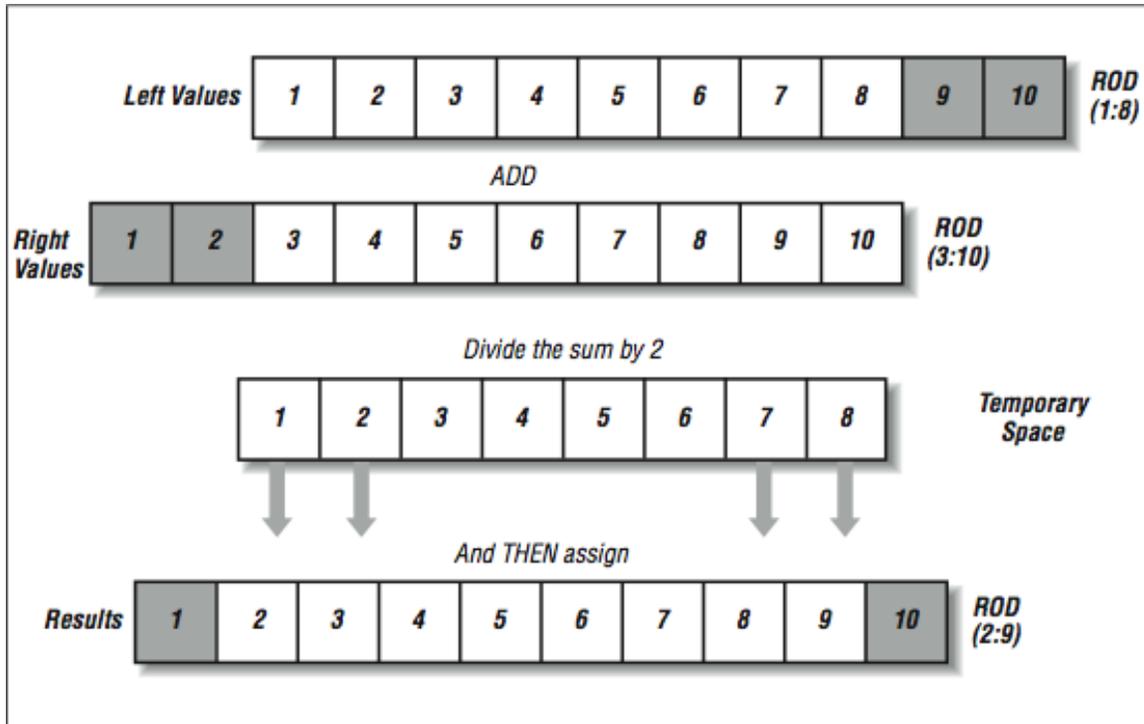


Figure 1

```

SAVE1 = ROD(1)
DO I=2,9
  SAVE2 = ROD(I)
  ROD(I) = (SAVE1 + ROD(I+1) ) / 2
  SAVE1 = SAVE2
ENDDO

```

This does not have the parallelism that the full red-black implementation has, but it does produce the correct results with only two extra data elements. The trick is to save the old "left" value just before you wipe it out. A good FORTRAN 90 compiler uses data flow analysis, looking at a template of how the computation moves across the data to see if it can save a few elements for a short period of time to alleviate the need for a complete extra copy of the data.

The advantage of the FORTRAN 90 language is that it's up to the compiler whether it uses a complete copy of the array or a few data elements to insure that the program executes properly. Most importantly, it can change its approach as you move from one architecture to another.

6 FORTRAN 90 Versus FORTRAN 77

Interestingly, FORTRAN 90 has never been fully embraced by the high performance community. There are a few reasons why:

- There is a concern that the use of pointers and dynamic data structures would ruin performance and lose the optimization advantages of FORTRAN over C. Some people would say that FORTRAN 90 is trying to be a better C than C. Others would say, "who wants to become more like the slower language!" Whatever the reason, there was some controversy when FORTRAN 90 was implemented, leading to some reluctance in adoption by programmers. Some vendors said, "You can use FORTRAN 90, but FORTRAN 77 will always be faster."
- Because vendors often implemented different subsets of FORTRAN 90, it was not as portable as FORTRAN 77. Because of this, users who needed maximum portability stuck with FORTRAN 77.
- Sometimes vendors purchased their fully compliant FORTRAN 90 compilers from a third party who demanded high license fees. So, you could get the free (and faster according to the vendor) FORTRAN 77 or pay for the slower (wink wink) FORTRAN 90 compiler.
- Because of these factors, the number of serious applications developed in FORTRAN 90 was small. So the benchmarks used to purchase new systems were almost exclusively FORTRAN 77. This further motivated the vendors to improve their FORTRAN 77 compilers instead of their FORTRAN 90 compilers.
- As the FORTRAN 77 compilers became more sophisticated using data flow analysis, it became relatively easy to write portable "parallel" code in FORTRAN 77, using the techniques we have discussed in this book.
- One of the greatest potential benefits to FORTRAN 90 was portability between SIMD and the parallel/vector supercomputers. As both of these architectures were replaced with the shared uniform memory multiprocessors, FORTRAN 77 became the language that afforded the maximum portability across the computers typically used by high performance computing programmers.
- The FORTRAN 77 compilers supported directives that allowed programmers to fine-tune the performance of their applications by taking full control of the parallelism. Certain dialects of FORTRAN 77 essentially became parallel programming "assembly language." Even highly tuned versions of these codes were relatively portable across the different vendor shared uniform memory multiprocessors.

So, events conspired against FORTRAN 90 in the short run. However, FORTRAN 77 is not well suited for the distributed memory systems because it does not lend itself well to data layout directives. As we need to partition and distribute the data carefully on these new systems, we must give the compiler *lots* of flexibility. FORTRAN 90 is the language best suited to this purpose.

7 FORTRAN 90 Summary

Well, that's the whirlwind tour of FORTRAN 90. We have probably done the language a disservice by covering it so briefly, but we wanted to give you a feel for it. There are many features that were not discussed. If you would like to learn more, we recommend *FORTRAN 90 Explained*, by Michael Metcalf and John Reid (Oxford University Press).

FORTRAN 90 by itself is not sufficient to give us scalable performance on distributed memory systems. So far, compilers are not yet capable of performing enough data flow analysis to decide where to store the data and when to retrieve the memory. So, for now, we programmers must get involved with the data layout. We must decompose the problem into parallel chunks that can be individually processed. We have several options. We can use High Performance FORTRAN and leave some of the details to the compiler, or we can use explicit message-passing and take care of *all* of the details ourselves.