Fortran: A Modern Standard Programming Language For Parallel Scalable High Performance Technical Computing

David B. Loveman
Digital Equipment Corporation
129 Parker Street, PKO3-1/A14
Maynard MA 01754
loveman@msbcs.enet.dec.com

Abstract -- *Fortran is often thought of as an old, archaic programming language that used to be adequate for technical computing but is rapidly being replaced by more modern languages such as C and, especially, C++.*

No perception could be less accurate.

Fortran has been modernized by the standardization process that lead to Fortran 90, and further enhanced with features developed by the High Performance Fortran Forum, many of which have been incorporated in the Fortran 95 draft standard. This modernization makes Fortran the ideal programming language for the development of new technical computing applications or the modernization of legacy codes written in FORTRAN 77. Indeed, since Fortran 90 provides all of the features of FORTRAN 77, initial conversion of a legacy application typically only requires recompilation.

This paper provides a quick overview of "modern" Fortran for the development of numerically intensive technical computing applications and looks at some simple examples. These examples are chosen to be illustrative of a data parallel coding style that is readable and understandable, performs well on a single processor system, and scales well on multiple processor shared memory and distributed memory systems. Compilers supporting this style of programming are available from a number of vendors.

BACKGROUND

Technical application developers must program in a "lowest common denominator" open systems programming language, in order to achieve platform independent portability with acceptable performance across multiple vendor's platforms. Modern hardware with multiple levels of memory and multiple communicating processors (SMP and farm clusters) makes development of efficient and portable applications difficult.

Earlier languages such as FORTRAN 77[1] [3], C and C++ assume that memory access is uniform and linear, making the development of scalable applications capable of exploiting multiple processors very difficult. Thus, efficiency requires programming with knowledge of the hardware making the application low-level and not portable.

A parallel scalable application must be implemented using language features that do not presume characteristics of a computer architecture. FORTRAN 77, C, C++, and other languages presume that the underlying computer has a linear memory with uniform access costs to consecutive addresses. This memory is visible in such language constructs as Fortran COMMON and C pointers. Clearly, distributed memory computers such as workstation farms do not have such a memory. Not so obviously, neither do shared memory SMP systems.

Fortran 90 ([1], [7]) , the new ANSI and ISO standard, provides developers of technical and scientific applications with significant new capabilities. Although Fortran 90 preserves, for compatibility, the linear memory aspects of FORTRAN 77, it provides new features to allow one to:

- operate on entire arrays or array sections using array syntax, the powerful forall statement, and array intrinsic and inquiry functions;
- pass arrays and array sections to procedures without depending on the array element order of storage and use explicit procedure interfaces to provide compile-time call checking and improved optimization;
- allocate and free data objects without requiring memory addresses and without inhibiting optimization;

[1] Note that the correct spellings are "FORTRAN 77" and "Fortran 90."

0190-3918/96 $5.00 © 1996 IEEE

- use modules to provide global data without using COMMON and INCLUDE, implement new libraries and interfaces to existing libraries, and support data abstraction with module private data;
- implement data structures with named heterogeneous components, define user-defined data types, and overload built-in operators to operate on those data types;
- implement dynamic data structures such as lists and trees without requiring memory addresses or complicated indexing schemes;
- use "convenience features" such as lower case letters, long names, free source form, and modern control structures.

Product Fortran 90 compilers are currently available from major vendors such as Digital ([4], [5], [11]), IBM, and Sun, and from third parties including Microsoft.

The High Performance Fortran (HPF) Forum ([6]), a consortium of vendors, users, research laboratories, and universities, has defined an industry consensus on a small set of extensions to the Fortran 90 standard, some of which are expected to be included in a revision to the Fortran standard scheduled for 1995. These extensions support portable and scalable programming on a wide variety of parallel multiple processor hardware architectures and include data placement directives, the `FORALL` construct, a collection of standard library procedures, and a procedure call mechanism supporting an interface between HPF's data parallel computing model and other models, such as message passing.

WHAT IS FORTRAN?

> "I don't know what the technical characteristics of the standard language for scientific and engineering computation in the year 2000 will be . . . but I know it will be called Fortran."
>
> paraphrase attributed to John Backus

Fortran was one of the first "high-level" programming languages, and remains widely used for programming scientific and engineering computations. The Fortran definition was standardized in the 1960s as FORTRAN 66 (ANS X3.9-1966); revised in the 1970s as FORTRAN 77 (ANSI X3.9-1978) [3]; augmented by a set of extensions known as MIL-STD-1753; and has been modernized as Fortran 90 (ISO/IEC 1539 : 1991 (E) and ANSI X3.198-1992) [7].

Fortran was developed as an easy to learn language for which a compiler could generate very efficient code for a wide variety of computer architectures. Upward compatible additions to FORTRAN 77 will make Fortran 90 the premiere programming language for high performance scientific and engineering computing well into the next decade.

FORTRAN 77

FORTRAN 77 provides many features helpful for the development of technical applications, including:

- Array reference and arithmetic computation notation
- Iteration (`DO`) loops
- Conditional (`IF .. THEN .. ELSE`) statements
- Subroutines and functions
- Global (`COMMON`) variables, but see the next section
- Independent compilation of program units
- Complex numbers
- Character data type and operations
- Formatted input and output
- Unformatted input and output
- Direct-access file input and output

It also provides a number of old style and politically incorrect features such as:

- statement number terminated `DO` loops
- computed and assigned `GO TO` statements
- Hollerith data (for compatibility with FORTRAN 66)
- arithmetic `IF` statements [2]
- EQUIVALENCE
- Fixed form source input
- Implicit data typing and IMPLICIT statement

Problems with Fortran

Unfortunately, as was mentioned earlier, the presumption of a linear memory model is directly visible in Fortran in two ways:

- *Sequence association* is the definition in Fortran of the mapping of multi-dimensional arrays to a linear sequence ordering. The programmer knows that arrays are stored in column-major order.
- *Storage association* is the definition in Fortran of the mapping of Fortran data objects to underlying storage units.

Typical uses in Fortran of sequence and storage association include:

- assumed size array dummies

[2] Originally in FORTRAN I for efficient execution on the IBM 704.

- COMMON reshaping
- EQUIVALENCE reshaping
- procedure argument reshaping

Some linear memory examples[3] to puzzle out are given below. There is an error in the example, left as an exercise to find.

```
real, dimension(10,10) :: a, b
complex, dimension(100) :: c
common /cm/ c
real, dimension(2,3,6) :: d
equivalence (a(9,6),d(1,2,3))
call subr (a,b(5,5),d)
...
subroutine subr(x,y,z)
  real, dimension(100) :: x
  real, dimension(10) :: y
  real, dimension(3:8,3:*) :: z
  real, dimension(100) :: w
  real, dimension(10,10) :: t
  common /cm/ w, t
  ...
```

What does Fortran 90 add to Fortran?

Lexical improvements include a free form source format; long names of up to 31 characters; and the use of a larger character set including the use of symbols such as "<" and ">=".

Array facilities allow "... for processing whole arrays and subarrays ... [and] provide a more concise and higher level language that will allow programmers more quickly and reliably to develop scientific/engineering applications, and ... [will] facilitate optimization of array operations on many computer architectures" [7] and include:

- array operations, subsections, expressions, masked and unmasked assignment, constructors, and elemental and transformational intrinsic functions
- automatic arrays and assumed-shape array dummy arguments
- dynamic storage via allocatable arrays and pointers

Data object and specification features include IMPLICIT NONE, declaration factoring by object or attribute, data object initialization, and multiple data types with user-specified precision and range, and numeric inquiry functions.

User-defined derived types permit the definition and use of structured objects; structure constructors, initialization, assignment, user-defined operators and generic overloading of operators and intrinsic functions; and pointers to data structures, and their use in expressions and assignments.

The module capability supports packages of global data definitions, procedure libraries, and the encapsulation of derived data types.

New procedural capabilities allow internal procedures, recursive procedures, procedures with optional or keyword arguments, specification of the data passage direction (IN, OUT, or INOUT) of arguments, many new intrinsic procedures, and explicit interface definitions for compiler checking and optimization.

Control constructs new to Fortran 90 include new forms of the DO statement and a CASE statement.

Improved input/output facilities include partial records or non-advancing input/output, NAMELIST, and new FORMAT statement edit descriptors.

Finally, Fortran 90 identified certain obsolescent Fortran features, which may inhibit portability or optimization, and which may be removed in future standards.

What is High Performance Fortran?

High Performance Fortran (HPF) ([6], [8], [9]) is Fortran 90, augmented by a small number of de facto industry standard extensions. It provides language features allowing application programmers flexibility in developing or migrating computationally intensive applications to high performance computing environments while at the same time allowing efficient implementation on a wide variety of platforms.

High Performance Fortran supports the *data parallel* programming style for massively parallel MIMD and SIMD supercomputers as well as for pipelined multiple instruction issue RISC processors. It also supports inter-operation with other programming styles such as SPMD with explicit message passing and pre-existing scalar "dusty deck" codes. A target-independent language, it provides a standard base language to which can be added target-dependent features as required.

HPF was developed in a spirit of cooperation and a belief that agreement on a de facto standard language will increase the size of addressable markets and that there are more interesting topics to compete on than language syntax. Critical to HPF's success was the emphasis on agreement rather than promulgation of particular vendor-specific language features. As a result, HPF provides a platform-independent programming model that matches application problems rather than target machines, is understandable by both people and by tools, is capable of being compiled into very efficient

[3] "stupid memory tricks" A. Gaber Mohamed, Syracuse University

code for a variety of platform architectures, and is de facto standard, widely used, open, and interoperable. It promises users that they will have to "recode only one more time" and their code investment will be preserved.

HPF extends Fortran 90 in four areas:

Data layout and placement directives - HPF directives describe the collocation of data (ALIGN) and the partitioning of data among memory regions (DISTRIBUTE). Compilers may interpret these directives to improve storage allocation for data. Users should expect the compiler to arrange the computation to minimize communication while retaining parallelism, without changing the meaning of the original program. The NOSEQUENCE directive tells the compiler it may ignore Fortran's usual rules regarding column-wise array layout in consecutive memory locations.

Parallel statements and directive - The FORALL construct ([2], [10]) is an element-at-a-time generalization of data parallel array operations. The INDEPENDENT directive asserts that the statements in a particular section of code do not exhibit any dependencies requiring sequential execution; when properly used it does not change the semantics of the construct, but may provide more information to the compiler to allow optimizations. PURE procedures are procedures restricted sufficiently so that they may be invoked within a FORALL.

Intrinsic and library procedures - Fortran 90 anticipated some, but not all, of the basic operations that are valuable for parallel algorithm design. HPF adds system inquiry intrinsics, new computational intrinsics and extensions of existing intrinsics, distribution inquiry intrinsics, and a standard library of computational functions.

The EXTRINSIC capability - HPF's global address space and single logical thread of control does not easily allow the expression of certain programming paradigms. The extrinsic[4] procedure interface is an escape mechanism for calling non-HPF code, written in other paradigms such as SPMD with explicit message-passing, from an HPF program. This allows the programmer to descend to a lower level of abstraction to handle problems that are not efficiently addressed by HPF, and to hand-tune critical kernels, or call optimized SPMD libraries. This interface can also be used to interface HPF to other languages, such as C.

The FORALL construct and related features

The FORALL construct can be viewed as an extension to the Fortran 90 array assignment and WHERE construct which is intended to be more suggestive of local operations on each element of an array. and able to specify more general array sections than are allowed by the basic array triplet notation.

The HPF FORALL construct had its origins in the original Fortran 8x definition, in the Fortran implementations of COMPASS, Digital Equipment, MasPar, and Thinking Machines and in research at Rice, Syracuse, Vienna, and other universities. Precursors to the FORALL statement can be found as early as the DO FOR ALL statement implemented in the ILLIAC IV IVTRAN compiler.

The FORALL construct is used to specify array assignments in terms of array elements or groups of array sections, optionally masked with a scalar logical expression. In functionality, it is similar to array assignment statements; however, more general array sections can be assigned in FORALL.

In many cases, compiler optimizations such as copy propagation can eliminate the requirement for temporaries for lower bounds, upper bounds and strides. Similarly, dependence analysis may eliminate the requirement for array temporaries. Thus a FORALL statement such as:

```
forall (i=1:m, j=1:n:2) a(i,j) = i * b(j)
```

can be implemented on a scalar machine as:

```
do i=1,m
  do j=1,n,2
    a(i,j) = i * b(j)
  end do
end do
```

On a parallel SIMD or MIMD machine it can, of course, be implemented in parallel.

Any Fortran 90 array assignment can be written as a FORALL, but not all FORALL statements have natural expressions in array syntax.

For example:

```
forall (i=1:n, j=1:n, a(i,j) /= 0.0) &
     b(i,j) = 1.0 / a(i,j)
```

expresses the same computation as

```
where (a /= 0.0) b = 1.0 / a
```

and

```
forall (i=1:l) r(i) = s(i)
```

expresses the same computation as

```
r(1:l) = s(1:l)
```

or

```
r = s
```

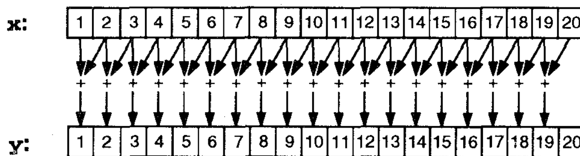
[4] Extrinsic: Originating from the outside; external (American Heritage Dictionary, 1983)

Examples of element array assignments without natural Fortran 90 equivalents are:

```
forall (i=1:100) a(i, i) = b(i, i)
forall (i=1:100) a(i, i) = b(i+1, i-1)
forall (i=1:100) a(u(i), v(i)) = s(i)
forall (i=2:100) r(i) = r(i/2)
  ! array representing a binary tree
! parallel prefix operations
forall(i=1:100) r(i) = sum(s(1:i))
  ! equivalent to hpf function
  r = sum_reduce(s)
```

EXAMPLE - A SIMPLE CALCULATION

Let's look at a very simple example showing some benefits of Fortran 90 and HPF. Assume we have a vector x of n elements and wish to calculate a new vector y of n elements in which the ith element of y is assigned the sum of the ith and $(i+1)^{st}$ elements of x (and the nth element of y is unchanged):



Given the declarations:

```
integer, parameter :: n = 20
real, dimension(n) :: x, y
  !hpf$ align x(:) with y(:)
  !hpf$ distribute y(block)
```

the computation can be written in three different ways, whole array data parallel:

```
y(1:n-1) = x(1:n-1) + x(2:n)
```

element-by-element data parallel:

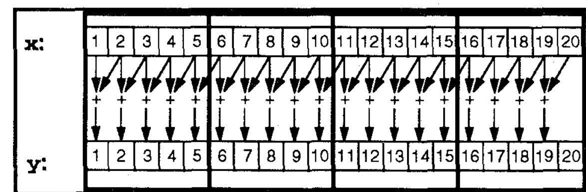
```
forall (i = 1:n-1) y(i) = x(i)+x(i+1)
```

and iteration parallel:

```
  !hpf$ independent
do i = 1, n-1
  y(i) = x(i) + x(i+1)
end do
```

The `align` and `distribute` directives inform the compiler that the developer believes that corresponding elements of x and y tend to be accessed "at about the same time," that elements close to a given element in its array's index space also tend to be accessed "at about the same time," and that performance will be better if these elements are located "near" each other on systems with non-uniform memory access. Specifically, the `align` directive requests that a given element of x be allocated "on the same processor" as the corresponding element of y, and that the elements of y should be assigned to processors in roughly equal sized blocks, depending on the size of y and the number of processors available. HPF leaves up to an implementation the precise definition of phrases such as "at about the same time," "near," and "on the same processor." The `independent` directive asserts that the `do` loop iterations are independent and may be executed in any order.

If this example is to be executed on four processors, the computation can be illustrated by the following picture. Note the "edge" problem requiring communication and that the last processor does a smaller number of calculations than the other three:



Lets look at what we would have to do in order to implement this simple calculation using a Single Program Multiple Data (SPMD) style with explicit message passing. First we would have to manually do the data decomposition. With arrays of length twenty distributed in blocks across four processors, there would be five elements per processor. Because of the communication requirement, however, we have to add a last element to the per-processor x as space to hold the value communicated from the processor one to the right, as follows:

```
real, dimension(6) :: x
real, dimension(5) :: y
```

In order to "set up" for the computation, the "first" element of x on all but the first processor is sent to the processor one to the left, and all processors except the last receive a value into the last element of x, as follows:

```
if (mynode() > 0) then
  call send(x(1), to=mynode()-1)
end if
if (mynode() < 3) then
  call receive(x(6), from=mynode()+1)
end if
```

In order to perform the per-processor computation, we must remember to compute one less value on the last processor:

```
if (mynode() < 3) then
  y(1:5) = x(1:5) + x(2:6)
else
  y(1:4) = x(1:4) + x(2:5)
end if
```

There are two problems with coding in this style. First it is cumbersome, time consuming, and error prone. Second, since details about the machine architecture, number of processors, and data layout are "hard wired" into the code, changes are difficult. Suppose for example

we wanted to make some modifications such as scaling the problem array size to 100, increasing the number of processors to 8 (n.b., an uneven distribution), and changing the distribution from block to cyclic. In our explicit SPMD with message passing style, we would have to begin anew with a new data distribution. In HPF we simply have to change the array dimension parameter and the distribution directive, as follows:

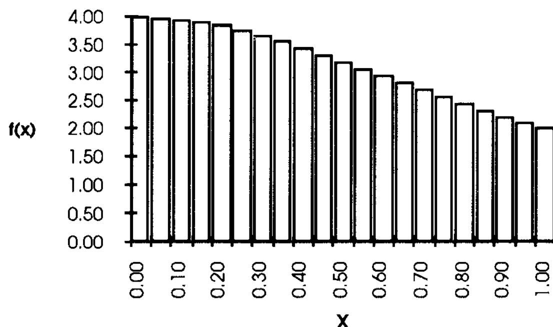
```
integer, parameter :: n = 100
real, dimension(n) :: x, y
  !hpf$ align x(:) with y(:)
  !hpf$ distribute y(cyclic)
y(1:n-1) = x(1:n-1) + x(2:n)
```

EXAMPLE - THE COMPUTATION OF PI

As a first example, lets consider how we might compute the value of pi. One way is to numerically integrate the following formula, as described in [12]:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

The usual way to do this is to plot the curve defined by the function and to divide the area under the curve into a number of evenly spaced rectangles. The sum of the areas of the rectangles is an approximation to the value of the integral; the more rectangles, the more accurate the approximation.



In the development of this example we shall pay particular attention to the style issues of scalable data parallel coding and to illustrate some of the features of Fortran 90.

The height of each rectangle is determined by the function. We could choose to have the function pass through the upper left corner of each rectangle and have the sum of the areas approach pi from above as we increase the number of rectangles, or we could choose to have the function pass through the upper right corner of each rectangle and have the sum of the areas approach pi from below as we increase the number of rectangles. In order to have a more accurate approximation, however, we choose to have the function pass through the midpoint of the top of the rectangle.

Code Design Issues

There are a number of decisions to be made:

- Shall we read in the number of rectangles or shall it be a parameter?
- Shall we do any manual "optimizations" on our code or should we assume that we have a clever compiler?
- How shall we represent the function to be integrated?
- Shall we proceed sequentially, calculating rectangle areas one after another and keeping a running sum or shall we proceed in a data parallel style and compute all of the rectangle areas and then sum them?

Input Values

The first decision is straightforward: If we read in the number, we will have something like the following in our code, using list directed input:

```
integer :: n ! number of rectangles
. . .
read *, n
```

while if we use a parameter we will have:

```
                  ! number of rectangles
integer, parameter :: n = 1000
```

In this simple example, we have chosen to use a single letter variable name `n` and remind ourselves of its usage with comments rather than use a longer name such as `number_of_rectangles`, often better practice.

Optimization

The question of hand optimization of code is difficult. On the one hand we want the best performance we can get. On the other hand we want our code to be readable and maintainable over time. An example of a hand optimization would be to sum all of the heights and then multiply the heights by the (constant) rectangle width rather than summing the areas directly. Our recommendation is to program in as readable manner as possible. If this does not provide acceptable performance

do hand optimization only as required and communicate with your compiler developers so that compiler optimization will improve over time.

Function Representation

The function to be integrated can be represented implicitly or explicitly. An implicit representation uses the body of the function in line in the computation, perhaps as a hand optimization to avoid the (suspected) cost of a function call. This example in particular calls out for an explicit representation as a Fortran function.

There are 4 different ways we can explicitly represent the function to be integrated:

- arithmetic statement function
- internal function
- module function
- external function

Fortran 77 style arithmetic statement functions are an obsolescent form which allow you to define one-line functions as the last item in the declaration part of a compilation unit. For example, the function to be integrated for pi is given below, complete with the type declaration needed to avoid warnings from the use of `implicit none`, which should always be used to force declarations for all names in a compilation unit.

```
  real f
  f(x) = 4 / (1.0 + x**2)
```

An explicit function definition, as below, is more general than an arithmetic statement function, and should be used. Note that it is a `pure` function, with no side effects, and a dummy argument for which a value is passed in and not changed.

```
  pure real function f(x)
    real, intent(in) :: x
    f = 4 / (1.0 + x**2)
  end function f
```

Where can this function be placed? It can be an internal procedure within the main program, as in

```
program pi_example
  . . .
contains
  pure real function f(x)
    real, intent(in) :: x
    f = 4 / (1.0 + x**2)
  end function f
end program pi_example
```

Such internal procedures are the Fortran 90 replacement for arithmetic statement functions.

Alternatively, the function can be packaged in a module. Modules can be used to contain a mixture of data and procedure definitions. A module containing just procedure definitions behaves like a library; the main program can access the procedures in the module via a `use` statement. For example,

```
module function_to_be_integrated
contains
  pure real function f(x)
    real, intent(in) :: x
    f = 4 / (1.0 + x**2)
  end function f
end module function_to_be_integrated

program pi_example
  use function_to_be_integrated
  . . .
end program pi_example
```

If the function were a previously compiled external function, its interface should be made explicit by means of an interface block describing the arguments of the function and the value to be returned:

```
  interface
    pure real function f(x)
      real, intent(in) :: x
    end function f
  end interface
```

The interface block can either appear directly in the main program or can be in a module of such interface blocks used to provide explicit interfaces to preexisting libraries.

There are still other possibilities involving the use of include files or Fortran's default behavior (something that looks like an array reference to an undeclared array is interpreted as a reference to an external procedure of default data type) that we shall not present because they are styles that are strongly recommended against.

Sequential or Data Parallel

In a sequential style we calculate rectangle areas one after another and keep a running partial sum, as in:

```
  partial_sum = 0.0
  do i = 1, n     ! in the ith rectangle
    mid_point = w * (i-0.5)
    height = f(mid_point)
    area = w * height
    partial_sum = partial_sum + area
  end do
  pi = partial_sum
```

whereas in a data parallel style we would compute all of the rectangle areas and then sum them. This would be awkward in FORTRAN 77:

```
real, dimension(n) :: rect_area

do i = 1, n       ! in the ith rectangle
  mid_point = w * (i-0.5)
  height = f(mid_point)
  rect_area(i) = w * height
end do
partial_sum = 0.0
do i = 1, n
  partial_sum =      &
    partial_sum+rect_area(i)
end do
pi = partial_sum
```

The second loop performs a reduction operation, specifically a sum reduction. Fortran 90 provides a number of intrinsic reduction functions including sum. In the first loop, the iterations are all independent of one another and need not be carried out in any particular order. The scalars mid_point and height are each set and used only within a single loop iteration and are said to be private to that iteration. The compiler can be told this to potentially allow optimization by preceding the loop with and HPF independent directive. Since the iterations are independent, we can also given a data distribution for the array rect_area. This says that, should we compile for multiple processes, the data, and consequently the array iterations, should be spread across them in roughly equal sized blocks.

```
real, dimension(n) :: rect_area
  !hpf$ distribute (block) :: rect_area

!hpf$ independent(mid_point, height)
do i = 1, n       ! in the ith rectangle
  mid_point = w * (i-0.5)
  height = f(mid_point)
  rect_area(i) = w * height
end do
pi = sum(rect_area)
```

The HPF independent directive depends on correct information being given by the programmer about the lack of dependencies between loop iterations and the private nature of scalars used in loop iterations, information that may not be checked by the compiler and which, if incorrect, will lead to erroneous behavior. In this case, of course, we could eliminate the requirement for iteration private variables in two ways. We could promote them to be arrays, as in

```
real, dimension(n) ::         &
    rect_area, mid_point, height
  !hpf$ distribute (block) :: &
    rect_area, mid_point, height

!hpf$ independent
do i = 1, n        ! in the ith rectangle
  mid_point(i) = w * (i-0.5)
  height(i) = f(mid_point(i))
  rect_area(i) = w * height(i)
end do
pi = sum(rect_area)
```

or we could propagate the expressions, eliminating the need for the variables completely, as in

```
real, dimension(n) :: rect_area
  !hpf$ distribute (block) :: rect_area

!hpf$ independent
do i = 1, n        ! in the ith rectangle
  rect_area(i) = w * f(w * (i-0.5))
end do
pi = sum(rect_area)
```

Fortran 90 provides two different data parallel approaches in addition to the do loop iteration style just discussed.

The array syntax approach involves operations on entire arrays. We can use an array constructor to make an array with n values, each of which is the area of one of the rectangles:

```
rect_area =          &
    (/ ( w * f(w * (i-0.5)), i=1, n ) /)
pi = sum(rect_area)
```

Alternatively, and preferably, we can take an element-at-a-time-view. This uses a forall statement to let us describe the computation to be performed at a single data point, and the set of data points for which it is to be done, as in

```
forall (i=1:n)       &
  rect_area(i) = w * f(w * (i-0.5))
pi = sum(rect_area)
```

Note that, since we specified f to be a pure function, it can be called in a forall statement.

The Code For Pi

The following program shows the complete program to calculate the value of pi. Notice that as good practice we have included the nosequence directive to indicate that we are not depending on the Fortran rules about array element order or storage association and

implicit none to indicate that lack of a variable declaration should indicate an error rather than the traditional Fortran implied data typing.

```
program pi_example
! compute pi by numeric integration
  !hpf$ nosequence
  implicit none
                       ! number of rectangles
  integer, parameter :: n = 1000
                       ! width of a rectangle
  real, parameter    :: w = 1.0 / n

  integer            :: i          ! index
  real               :: pi
  real, dimension(n) :: rect_area
    !hpf$ distribute (block) :: rect_area


  forall (i=1:n)        &
    rect_area(i) = w * f(w * (i-0.5))
  pi = sum(rect_area)
  print *, 'value computed = ', pi

contains

  pure real function f(x)
    ! to be integrated
    real, intent(in) :: x
    f = 4 / (1.0 + x**2)
  end function f

end program pi_example
```

ACKNOWLEDGMENTS

This paper would obviously have been impossible without the efforts of my colleagues on the High Performance Fortran Forum and a number of people at Digital who specified, designed, and implemented the DEC Fortran 90 compiler and Parallel Software Environment. Any mistakes in this paper are, of course, mine.

REFERENCES

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T, Martin, Brian T. Smith, Jerrold L. Wagener, *Fortran 90 Handbook,* Intertext Publications, McGraw-Hill Book Company, New York, NY, 1992.

[2] Eugene Albert, Joan D. Lukas, and Guy L. Steele, Jr., "Data Parallel Computers and the FORALL Statement," *Journal of Parallel and Distributed Computing*, pp. 185-192, October 1991.

[3] American National Standards Institute, Inc., 1430 Broadway, New York, NY., *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute, Inc., 1978.

[4] Marco Annaratone, David B. Loveman, and Carl D. Offner, "High Performance Fortran on Workstation Farms," Proceedings, *Eighth International Parallel Processing Symposium,* Cancun, Mexico, IEEE Computer Society Press, pp. 664-669, April 1994.

[5] Digital Equipment Corporation, Maynard, Massachusetts, *DEC Fortran 90 Language Reference Manual*, June 1994, [AA-Q66SA-TK].

[6] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.0,* CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised May. 1993). Also appeared in a special issue of *Scientific Programming*, Volume 2 / Numbers 1 and 2, John Wiley & Sons, Spring and Summer 1993.

[7] ISO, *Fortran 90*, May 1991, [ISO/IEC 1539: 1991 (E) and ANSI X3.198-1992].

[8] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1993.

[9] David B. Loveman, "High Performance Fortran," *IEEE Parallel & Distributed Technology,* Vol. 1, No. 1, February 1993.

[10] David B. Loveman, "Element Array Assignment the FORALL Statement," *Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 6-9, 1992.

[11] David B. Loveman, "The DEC High Performance Fortran 90 Compiler Front End," *Proceedings of Frontiers'95, The Fifth Symposium on the Frontiers of Massively Parallel Computation,* McLean, VA, IEEE Computer Society Press, pp. 46-53, February 1995.

[12] Susan Ragsdale (ed.) *Parallel Programming*, McGraw-Hill, 1991, pp. 24-30.