PL

Fortran 90 and Computational Science

Copyright (C) 1991, 1992, 1993, 1994, 1995 by the Computational Science Education Project

This electronic book is copyrighted, and protected by the copyright laws of the United States. This (and all associated documents in the system) must contain the above copyright notice. If this electronic book is used anywhere other than the project's original system, CSEP must be notified in writing (email is acceptable) and the copyright notice must remain intact.

1 Overview of Fortran 90

The diagram in Figure 1 shows the major components of Fortran 90 [1],[2]. The size of each slice of this "pie" is roughly proportional to the number of syntax rules needed to describe the features associated with that slice, and hence is a measure of the structural complexity of those features. (These measures should not, however, be taken as an indication of conceptual or semantic complexity nor of implementation effort-syntactic complexity may or may not be related to these other forms of complexity.)

Fortran 77 Fortran 90 is a super-set of Fortran 77-all standard Fortran 77 programs are standard Fortran 90 programs. Fortran 90 therefore encompasses and is completely compatible with the existing Fortran 77 computational science infrastructure.

Source Form To Fortran 77's 'fixed' source form Fortran 90 adds another source form, called 'free' source form, in which there are no column dependencies. In free source form comments need not start in column 1 and column 6 is not reserved for continuation; continuation in free source form is indicated by a trailing ampersand, on the 'first' line. In both source forms an exclamation point , '!', may be used to initiate end-of-line comments (e.g., following a statement on that line) and a semicolon may be used to separate two statements on the same line. As in Fortran 77, Fortran 90 names (of variables, procedures, etc.) begin with a letter and contain letters and digits; in addition, names may have up to 31 characters, may contain underscore, '_', characters, and may contain both upper and lower case letters.

Control Structures Missing from Fortran 77 is a complete set of modern control structures (it has only IF - THEN - ELSE - END IF); this is remedied in Fortran 90 with the

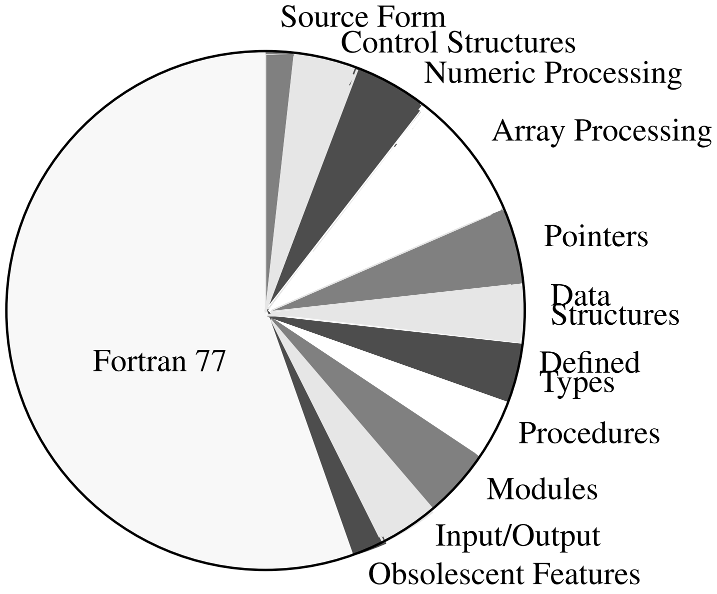

Figure 1: Fortran 90

addition of DO - END DO and SELECT CASE - END SELECT control structures. The CASE construct, which has the form:

```
SELECT   CASE   (expression)
         CASE   (value-list)
         ...
         CASE   (value-list)
         ...
         ...
END SELECT
```

provides "parallel" selection control. It offers no increase in semantic power over the IF ("sequential") selection control, but in situations where it fits the problem CASE offers computational advantages over IF because only one expression is evaluated.

The new DO construct is likely to be extremely useful in computational science applications. The reason is that, in order to realize the computational benefits of data parallelism, a great many indexed do loops will be replaced by array operations. The emphasis in the use of loops will therefore shift from processing arrays to indefinite repetition operations such as "read-test-process" and "while" repetition. The two new forms of DO construct added in Fortran 90 therefore are:

```
DO                                  DO WHILE (logical-expr)
```

```
          ...                                   ...
          IF (logical-expr) EXIT        END DO
          ...
     END DO
```

Indexed loops may also be terminated with END DO, in which case the loop label is not needed. (Loop labels are permitted, however, in all three forms of DO construct.)

Numeric Processing This will be discussed in detail in section 3. The main features in this category are the numeric representation model, many intrinsic functions that return useful model values, the numeric kind system, intrinsic functions that return kind values, and generic operations.

Array Processing This will be discussed in detail in section 4. Array operations are one of the most significant aspects of Fortran 90.

Pointers Pointers provide two important capabilities in Fortran 90- dynamic data structures and dynamic arrays. The latter is especially important for computational science because it allows arrays to be dynamically allocated (and reallocated) of the proper size and provides a means to minimize data transfers when performing operations on variable array sections.

Because of the adverse impact that pointers have on optimization, a Fortran 90 pointer may point only to (a) a data object explicitly declared as a pointer target, (b) a dynamically created object, or (c) another pointer. This makes it possible for static storage optimization technology to be be applied to data that has neither the pointer nor target attributes.

Such pointers may be used for arrays whose sizes are determined at run time, such as dynamically allocated arrays; such arrays are declared as "deferred shape" arrays, in which the rank is specified, and thereby fixed, but the extent in each dimension is unspecified and dynamic. These dimensions will be dynamically established later. The following examples illustrate such dynamic behavior. (The "=¿" is the Fortran 90 syntax for pointer assignment, and the ALLOCATE statement is used for dynamic allocation of storage.)

```
REAL, TARGET  :: B(100,100)          ! Array B has the target attribute.
REAL, POINTER :: U(:,:),V(:),W(:,:)  ! Declaration of 3 pointer arrays
 ...
U => B(I:I+2,J:J+2)                  ! U points to a 3x3 section of B.
ALLOCATE ( W(M,N) )                  ! Dynamically allocate W, size MxN
V => B(:,J)                          ! V points to the Jth column of B.
V => W(I-1,1:N:2)                    ! V changed to point to (part of)
                                     !     the I-1st row of W.
```

Data Structures Arrays are probably the most important composite data objects for computational science, but more heterogeneous objects are necessary, including dynamically linked structures. In the jargon of Fortran 90, structures are objects of user-defined type, discussed briefly next and in more detail in section 5 on page 50 and section 6 on page 50. Dynamically linked structures are provided by essentially recursive user-defined types:

This is an example of a type that could be used for a doubly-liked list structure. Recursive components (PREVIOUS and NEXT in this example) must be pointers. In general, a defined type may have any number and types of components.

```
TYPE LIST
   REAL                   :: DATA
   TYPE(LIST),POINTER     :: PREVIOUS, NEXT
END TYPE LIST
```

User-Defined Types and Operators User-defined types and operators, together with modules, give Fortran 90 an outstanding data abstraction facility and corresponding support for this aspect of object oriented programming. A user-defined type is defined with the TYPE - END TYPE construct, and objects are declared in a manner analogous to declaration of objects of intrinsic type. A simple example of a type definition and corresponding object declaration is as follows.

```
TYPE RATIONAL                           ! This defines the type RATIONAL.
   INTEGER :: NUMERATOR
   INTEGER :: DENOMINATOR
END TYPE RATIONAL
  ...
TYPE (RATIONAL) :: X, Y(100,100)        ! X and Y are variables of
      ! type RATIONAL.
```

This might be the type defined in connection with a complete rational arithmetic data abstraction. Such an abstraction requires, in addition to the type definition, an appropriate set of operator definitions. These are done by specifying operator interfaces for user-defined functions. The following example illustrates extending the "+" operator to addition between objects of type RATIONAL.

```
INTERFACE OPERATOR (+)
 FUNCTION RAT_ADD(X,Y)
  TYPE (RATIONAL) :: RAT_ADD
  TYPE (RATIONAL) :: X, Y
 END FUNCTION RAT_ADD
END INTERFACE
```

More details on user-defined types and operators will be discussed in sections of the next release of this chapter.

Procedures To the Fortran 77 procedure facilities Fortran 90 adds the following new features: function results may be array-valued or structure-valued (or both), procedures may be recursive, arguments may be optional or intent-in (can't be changed in the procedure), procedures may be internal to other program units, and others.

An important concept new to Fortran 90 is that of an explicit procedure interface. This means that the interface of a procedure-that is, the data types and other characteristics of the dummy arguments and (if the procedure is a function) function result-is known at the point of call. Explicit interfaces provide a long list of benefits and capabilities; for example, a function result can be array valued if (and only if) the function interface is explicit where the function is called. Procedure interfaces may be made explicit by providing an interface block for the procedure or by placing the procedure definition internal to the calling program or in a module used by the calling program.

One of the most onerous sources of programming errors historically has been the mismatching of argument data types across procedure boundaries. Such errors, guaranteed to corrupt results, are among the hardest to debug. Use of explicit interfaces is simple and prevents this problem completely; this will likely be one of the most important practical applications of interface blocks.

Modules Fortran 90 has a new kind of program unit, the module, that is neither part of Fortran 77 nor included in most pre-90 Fortran compilers. Because of this lack of implementation history, and because modules impose more sophisticated name management requirements on implementations, modules have triggered some controversy. When this dust finally settles, however, modules are likely to be generally accepted as one of the most important and useful features of Fortran 90.

Unlike main programs and external subprograms, modules are not themselves executable program units. Rather, they contain definitions that can be conveniently accessed and used by executable program units. For example, a module might contain interface blocks for a library of external procedures and be used to make the interfaces of these library procedures explicit in an application using that library. A likely growing trend is to repackage the entire library in a module-that is, to place all of the procedure definitions in a module (assuming the procedures can be written in Fortran)-which has the twin benefits to an application of making the procedure interfaces explicit without the need for interface blocks and giving the application developer a powerful tool for namespace management.

Data related uses of modules are just as important as procedure related uses. User-defined type definitions can be placed in a module and thus made available to the other program units of an application , and indeed this is the preferred way of packaging most type definitions. Data objects, of any type, kind, and shape can be declared in a module and thus become global data objects for an application using that module. This provides a non- storage-associated global data alternative to COMMON, which can be used where storage association is a problem (e.g., in distributed memory environments). Arrays in modules can be allocatable, thus providing a means of having dynamic arrays conveniently accessible to any of the program units of the application.

A module can simultaneously contain both data related and procedure related entities. One typical such application of a module is to package a data abstraction- that is, to contain a type definition, interfaces defining operators to be used in conjunction with operands of that type, and possibly even the procedures defining operations on that type. For example the type definition for RATIONAL and the extension of "+" to RATIONAL addition illustrated above could be packaged in a module as follows:

```
MODULE RATIONAL_ARITHMETIC

 TYPE RATIONAL
  INTEGER :: NUMERATOR
  INTEGER :: DENOMINATOR
 END TYPE RATIONAL

 INTERFACE OPERATOR (+)
  FUNCTION RAT_ADD(X,Y)
   TYPE (RATIONAL) :: RAT_ADD
   TYPE (RATIONAL) :: X, Y
  END FUNCTION RAT_ADD
 END INTERFACE

 ...   ! and other stuff for a complete rational arithmetic facility

END MODULE RATIONAL_ARITHMETIC
```

This sort of comprehensive use of modules for will be described in more detail in the next release.

Input and Output Fortran 90 adds a few additional options to the comprehensive file connection capabilities of Fortran 77. One important example is the ACTION= open specifier, which allows a file to be connected as READ or WRITE or READWRITE. The major I/O additions in Fortran 90, however, are NAMELIST and nonadvancing I/O. NAME-LIST has long been included in many Fortran compilers; the standard has finally caught up to this common practice and has identified a standard form for NAMELIST from among the various existing versions. In addition to specifying a standard NAMELIST, its inclusion in Fortran 90 will insure that NAMELIST is available in all Fortran 90 compilers.

The other major I/O addition in Fortran 90 is nonadvancing I/O, which provides the functionality of reading or writing only part of a record. Recall that each formatted sequential READ or WRITE, the only form of I/O to which nonadvancing applies, in Fortran 77 causes (at least) one entire record to be read or written; that is, the file is always positioned between records after execution of a READ or WRITE statement. Non advancing I/O is different in that the file remains positioned after the last character transferred (effectively between characters in a record rather than between records)- a nonadvancing READ does not "skip" to the end of the record and a nonadvancing WRITE does not terminate the record (i.e., does not write an end-of-record mark). The next READ or WRITE on that file starts where the last one left off. Nonadvancing I/O is specified by including ADVANCE='NO' in the control list of the READ or WRITE statement. For example, the following nonadvancing READ statement could be used to obtain the next character from the input terminal:

```
CHARACTER :: C
  ...
  READ(*,'(A)',ADVANCE='NO',IOSTAT=IOS) C
  ...
```

Other forms of input/output considered for Fortran 90, but then not adopted, were asynchronous and parallel I/O and a variation of keyed access. Nor is there any direct database support in Fortran 90, though Fortran bindings exist to modern database facilities such as the structured query language (SQL) standard.

Obsolescent Features Fortran 90 includes a first step toward a model for planned language evolution as opposed to unbalanced growth or ad hoc removal of features. As with most first steps, this one is small, tentative, and with an as-yet unsure outcome. The idea is to ultimately remove those features that become obsolete as the language evolves, but to officially identify such candidates as "obsolescent" (in the process of becoming obsolete) well in advance of actual removal. This is intended to give the Fortran community (a) a chance to review the recommendations and prevent mistakes from being made and (b) time to prepare for the change in an orderly way. According to the current model, a feature listed as obsolescent in one version of the Fortran standard is a candidate for removal from the next version.

The following ten features of Fortran 90 are listed as obsolescent:

1. the arithmetic IF
2. real (and double precision) DO index variables and expressions
3. shared DO termination (i.e., two loops terminating on the same statement)
4. DO loop termination on other than END DO or CONTINUE
5. branching to an END IF statement from outside that IF construct
6. alternate return (use a return code variable instead)
7. the PAUSE statement (use READ instead)

Table 1:

functionality	F77	C	C^{++}	F90
numerical robustness	2	4	3	1
data parallelism	3	3	3	1
data abstraction	4	3	2	1
object oriented programming	4	3	1	2
functional programming	4	3	2	1
average	3.4	3.2	2.2	1.2

8. ASSIGN and assigned GOTO statements (use internal procedures instead)
9. assigned FORMAT specifiers (use character strings instead)
10. the H edit descriptor

2 Comparison of Fortran 77, C, C++, and Fortran 90

For thirty years, from its inception through Fortran 77, Fortran has been the principal language of computational science. During this time Fortran's numerical capabilities have been remarkably stable and superior to that of other computer languages; the biggest changes have come in the form of increasingly diverse and reliable libraries of numerical routines. The union of Fortran, techniques for its use, and the extensive numerical libraries characterize the predominant infrastructure for computational science.

In the past decade, however, the increasing importance of dynamic data structures (particularly dynamic arrays), unix workstations, sophisticated interactive visualization facilities, and, more recently, parallel architectures-none of which Fortran 77 accommodates well-has spurred interest in the use of other languages for computation languages, most notably C. Recently C++ has also garnered considerable interest, and Fortran has attempted to address its deficiencies for modern computational science by evolving to Fortran 90. A general attempt is made in this section to compare the relative suitability to computational science of these four languages, two flavors of C (C and C++) and two flavors of Fortran (Fortran 77 and Fortran 90). Table 1 summarizes this comparison, and the following subsections attempt to rationalize these rankings from best (1) to worst (4)..

2.1 Numerical Robustness

In section 3.3 below, Numeric Polymorphism, is an example of several versions of a picture-smoothing routine that are given one generic name. This generic capability is described there as one of the features that provide Fortran 90 with additional numeric robustness over Fortran 77 (and C). Fortran 77, Fortran 90, and C versions of subroutine SMOOTH are given here for comparison purposes. (Note that the Fortran 90 version makes use of the data parallelism described in section 4.)

Numeric polymorphism, plus real kind type parameterization, decimal precision selection, and numeric environmental inquiry, justify ranking Fortran 90 first among the four languages. The reason for ranking Fortran 77 second is its support for complex variables, important in many computational science applications. C++ nudges out C for third place due to its capabilities in the general area of polymorphism.

```
***************************************************************
*   Fortran 77 subroutine to compute a 3x3 average for each   *
*   element of an input matrix, except for the edges of the   *
*   matrix. This is a simple version of a common technique    *
*   for refining/enhancing a picture represented by a matrix. *
***************************************************************
*234567
      SUBROUTINE smooth(output, input, n, m)
      INTEGER i, j, n, m
      REAL upper, mid, lower
      REAL output(n,m), input(n,m)

      do 10, i = 2, n - 1
        do 20, j = 2, m - 1
           upper = input(i-1,j-1)+input(i-1,j)+input(i-1,j+1)
           mid   = input(i  ,j-1)+input(i  ,j)+input(i  ,j+1)
           lower = input(i+1,j-1)+input(i+1,j)+input(i+1,j+1)

           output(i,j) = (upper + mid + lower) / 9.0
 20     continue
 10   continue

      do 30, i = 1, n
           output(i,1) = input(i,1)
           output(i,m) = input(i,m)
 30   continue

      do 40, j = 2, m-1
           output(1,j) = input(1,j)
```

```
           output(n,j) = input(n,j)
 40   continue

      end



****************************************************************
*   Fortran 90 version of subroutine SMOOTH.                   *
****************************************************************
*234567
      SUBROUTINE smooth(output, input)
          REAL     input(:,:), output(size(input,1),size(input,2))
          INTEGER  N, M
          N = size(input,1)
          M = size(input,2)

      output(2:n-1,2:m-1) =                                       &
       ( input(1:n-2,1:m-2)+input(1:n-2,2:m-1)+input(1:n-2,3:m)   &
       + input(2:n-1,1:m-2)+input(2:n-1,2:m-1)+input(2:n-1,3:m)   &
       + input(3:n,1:m-2) + input(3:n,2:m-1) + input(3:n,3:m) / 9.

      output((/1,n/,:) = input((/1,n/),:)
      output(2:n-1,(/1,m/) = input(2:n-1,(/1,m/))

      end
/****************************************************************
 *   C version of subroutine SMOOTH.                            *
 *   C++ version would be similar, though classes would be used *
 *   if polymorphism were important as described in section 3.3.*
****************************************************************/
void smooth(void)
{
         int i, j;
         float upper, mid, lower;

         for ( i = 1; i < (IMAX - 1); i++) {
             for ( j = 1; j < (JMAX - 1); j++) {
          upper = input[i-1][j-1] + input[i-1][j] + input[i-
1][j+1];
                 mid   = input[i  ][j-1] + input[i  ][j] + input[i
][j+1];
```

```
                 lower = input[i+1][j-1] + input[i+1][j] +
input[i+1][j+1];

                 output[i][j] = ( upper + mid + lower ) / 9.0;
             }
          }
          for ( i = 0; i < IMAX; i++) {
              output[i][0     ] = input[i][0     ];
              output[i][JMAX-1] = input[i][JMAX-1];
          }
          for ( j = 0; j < JMAX; j++) {
       output[0     ][j] = input[0     ][j];
       output[IMAX-1][j] = input[IMAX-1][j];
          }
}

void wrt_input(void)
{
   int i, j;
          (void)printf("The values of the input matrix:\n");

(void)printf("************************************************\n")
;
   for ( i = 0; i < IMAX; i++) {
              for ( j = 0; j < (JMAX - 1); j++) {
                  (void)printf(" %f",  input[i][j]);
              }
              (void)printf(" %f\n", input[i][JMAX-1]);
          }

(void)printf("************************************************\n")
;
}

void wrt_output(void)
{
   int i, j;

          (void)printf("The values of the input matrix after
smoothing:\n");

(void)printf("************************************************\n")
```

```
;
   for ( i = 0; i < IMAX; i++) {
              for ( j = 0; j < (JMAX -1); j++) {
                  (void)printf(" %f", output[i][j]);
              }
              (void)printf(" %f\n", output[i][JMAX-1]);   }

(void)printf("********************************************\n")
;}
```

2.2 Data Parallelism Section

Section 4.5 below contains two versions of the Gaussian elimination algorithm for solving systems of linear equations, with and without using the maximum pivot strategy. That example was included to illustrate the data parallel features of Fortran 90-it is a relatively simple, practical problem that makes use of many of the Fortran 90 data parallel capabilities. For comparison with the programs in section 4.5, Fortran 77 and C versions are supplied here.

Of the four languages, only Fortran 90 has data parallel capabilities meaningful for computational science; the nature of the other three languages in this regard are essential the same, namely missing altogether. This explains the reason for the rankings of the four languages along this dimension.

Here are the set of Fortran 77 and C routines that perform the Gaussian elimination computation:

```
****************************************************************
*     Program to determine the correct processing of the       *
*     subroutines:  pivot.f, triang.f, and back.f. The         *
*     subroutines determine the solution to a series of        *
*     simultaneous equations.                                  *
****************************************************************
*234567
      PROGRAM testg
      INTEGER IMAX, JMAX
      PARAMETER (IMAX = 3, JMAX = 4)
      REAL  matrix(IMAX, JMAX)
      REAL  solvec(IMAX)
      INTEGER i, j, n

      DATA ( (matrix(i,j), j = 1, JMAX), i = 1, IMAX)
     +     /-1.0, 1.0, 2.0, 2.0, 3.0, -1.0, 1.0, 6.0,
     +      -1.0, 3.0, 4.0, 4.0/
```

```
      n = IMAX

      write(*,*) \"The original matrix,\",n,\" by \",n+1,\" :\"
      call wrtmat(matrix, n, n + 1)
      call pivot(matrix, n)
      write(*,*) \"The matrix after pivoting:\"
      call wrtmat(matrix, n, n + 1)
      call triang(matrix ,n)
      write(*,*) \"The matrix after lower triangulation:\"
      call wrtmat(matrix, n, n + 1)
      call back(solvec, matrix, n)
      write(*,*) \"The solution vector after back substitution:\"
      write(*,*) \"******************************************\"
      write(*,*) (solvec(i), i = 1, n)
      write(*,*) \"******************************************\"

      end


****************************************************************
*     Subroutine to determine the largest value in the         *
*     first column of an augmented matrix and move the         *
*     row with the largest value in the first column to        *
*     first row.  The process is then repeated for the         *
*     successive rows and columns, and for each                *
*     iteration, the column position and the row position      *
*     are decremented by 1 (That is, 1st Column-1st Row        *
*     then 2nd Column-2nd Row, then 3rd Column-3rd Row,        *
*     etc.                                                     *
****************************************************************
*234567
      SUBROUTINE pivot(matrix, n)
      INTEGER i, j, k, n
      REAL  matrix(n, n + 1), maxval, tempval

      do 10, j = 1, n
        maxval = matrix(j,j)
        do 20, i = j + 1, n
          if ( maxval .lt. matrix(i,j) ) then
             maxval = matrix(i,j)
             do 30, k = 1, n + 1
                tempval = matrix(i,k)
```

```
               matrix(i,k) = matrix(j, k)
               matrix(j, k) = tempval
 30         continue
          endif
 20     continue
 10   continue

      end



***************************************************************
*     Subroutine that performs the lower decomposition of an   *
*     input matrix.                                           *
***************************************************************
*234567
      SUBROUTINE triang(matrix, n)
      INTEGER i, j, k, n
      REAL  matrix(n, n + 1),  pivot, pcelem

      do 10, j = 1, n
        pivot = matrix(j,j)
        do 20, k = j + 1, n + 1
           matrix(j,k) = matrix(j,k) / pivot
 20     continue

        do 30, i = j + 1, n
           pcelem = matrix(i,j)
           do 40, k = j + 1, n + 1
              matrix(i,k) = matrix(i,k)  - pcelem * matrix(j,k)
 40       continue

 30     continue
 10   continue

      end
***************************************************************
*     Subroutine to compute a solution vector from an         *
*     augmented matrix that has already undergone lower       *
*     decomposition.                                          *
***************************************************************
*234567
```

```
      SUBROUTINE back(solvec, matrix, n)
      INTEGER n
      REAL solvec(n), matrix(n, n + 1), sum

      solvec(n) = matrix(n, n + 1)

      do 10, i = n - 1, 1, -1
           sum = 0.0
           do 20, j =  i + 1, n
              sum = sum + matrix(i, j) * solvec(j)
 20        continue
           solvec(i) = matrix(i, n + 1) - sum
 10   continue

      end


*************************************************************
*     Program to test the subroutine bisec.f, which          *
*     determines the root of an equation (declared in f.f).  *
*     However, the function does assume that the function-f  *
*     is bracketed by the two values. That is, there is not  *
*     more than one root between the two endpoints supplied by*
*     the user.                                              *
*************************************************************
*234567
      PROGRAM  testbs
      REAL xleft, xright
      REAL f
      EXTERNAL f

      write(*,*) \"Please enter an initial left and right value:\"
      read(*,*) xleft, xright
      call bisec(f, xleft, xright)

      end
```

And here are the C routines for the same algorithm:

```
/************************************************************* *
 * Program to determine the correct processing of the three        *
 * functions pivot.c, triang.c, and back.c. The functions          *
 * determine the solution to a series of simultaneous equations.   *
************************************************************** **/
#include <stdio.h>
#define IMAX 3
#define JMAX 4

float matrix[IMAX][JMAX] = {
                         {-1.0, 1.0, 2.0, 2.0 },
                         { 3.0,-1.0, 1.0, 6.0 },
                         {-1.0, 3.0, 4.0, 4.0 }
                           };

float solvec[IMAX]       = { 0.0, 0.0, 0.0 };

main()
{
        void wrt_output(void);
        void pivot(void);
        void triang(void);
        void back(void);
        void wrt_vector(void);

        (void)printf("The original matrix %d by %d :\n", IMAX, JMAX);
        (void)wrt_output();
        (void)pivot();
        (void)printf("The matrix after pivoting:\n");
        (void)wrt_output();
        (void)triang();
        (void)printf("The matrix after lower decomposition:\n");
        (void)wrt_output();
        (void)back();
        (void)printf("The solution vector after back
substitution:\n");
        (void)wrt_vector();
}

/************************************************************* ***
 *  Function to determine the largest value in the first column    *
 *  of an augmented matrix and move the row with the largest       *
```

```
 *  value in the first column to the first row.  The process is    *
 *  then repeated for the successive rows and columns, and for     *
 *  each iteration, the column position and the row position that  *
 *  are tested are incremented by 1 (That is, 1st Column-1st Row,  *
 *  2nd Column-2nd Row, 3rd Column-3rd Row, etc.                   *

****************************************************************** **/

void pivot()
{
 int i, j, k;
 float maxval, tempval;

 for ( j = 0; j < IMAX; j++) {
  maxval = matrix[j][j];
  for ( i = ( j + 1 ); i < IMAX; i++) {
   if ( maxval < matrix[i][j] ) {
    maxval = matrix[i][j];
    for( k = 0; k <= IMAX; k++) {
     tempval = matrix[i][k];
     matrix[i][k] = matrix[j][k];
     matrix[j][k] = tempval;
    }
    }
  }
 }
}

/***************************************************************** ***
 *  Function that performs the lower decomposition of an input     *
 *  matrix.                                                        *
****************************************************************** **/

void triang(void)
{
 int i, j, k;
 float pivot, pcelem;

 for ( j = 0; j < IMAX; j++) {
                pivot  = matrix[j][j];
  for ( k = ( j + 1 ); k <= IMAX; k++) {
   matrix[j][k] = matrix[j][k] / pivot;
```

```
  }
  for ( i = ( j + 1 ); i < IMAX; i++) {
   pcelem = matrix[i][j];
   for ( k = ( j + 1 ); k <= IMAX; k++) {
                 matrix[i][k] = matrix[i][k] - ( pcelem * matrix[j][k] );
   }
  }
 }
}

/************************************************************** ***
 *  Function to compute a solution vector from an augmented        *
 *  matrix that has already undergone lower decomposition.         *
*************************************************************** **/

void back(void)
{
 int i, j;
        float sum;

        solvec[IMAX - 1] = matrix[IMAX - 1][JMAX - 1];

 for ( i = (IMAX - 1); i > -1; i--) {
  sum = 0.0;
  for ( j = (i + 1); j < IMAX; j++) {
                    sum = sum + matrix[i][j] * solvec[j];
  }
  solvec[i] = matrix[i][IMAX] - sum;
 }
}

void wrt_output(void)
{
 int i, j;


(void)printf("****************************************\n");
        for (i = 0; i < IMAX; i++) {
            for ( j = 0; j < (JMAX - 1); j++) {
  (void)printf(" %f", matrix[i][j]);
            }
            (void)printf(" %f\n", matrix[i][JMAX - 1]);
```

```c
 }

(void)printf("******************************************\n");
}
void wrt_vector(void)
{

(void)printf("******************************************\n");
 (void)printf(" %f", solvec[0]);
        (void)printf(" %f", solvec[1]);
        (void)printf(" %f\n", solvec[2]);

(void)printf("******************************************\n");
}

/**************************************************************** ******
 * Program to test the function bisec.c, which determines the         *
 * root of an equation (declared in f)..  However, the function does  *
 * assume that the function-f is bracketed by the two values.  That   *
 * is, there is not more than one root between the two endpoints      *
 * supplied by the user.                                              *

***************************************************************** *****/

#include <stdio.h>
#include <math.h>

main()
{
 void bisec(float init_left_val, float init_right_val);
 float f(float value);
        float xleft, xright;
        char line[100];

        (void)printf("Please enter an initial left and right
value:");
        (void)fgets(line, sizeof(line), stdin);
        (void)sscanf(line, "%f %f", &xleft, &xright );

        (void)bisec(xleft, xright);
        return(0);
}
```

2.3 Data Abstraction

Fortran 90 has a very practical, easy-to-use data abstraction capability. C++, as an important part of object-oriented programming, also has significant data abstraction capabilities. For computational science much advantage can be obtained from data abstraction without the additional complexities of object-oriented programming, and therefore a slight edge is given to Fortran 90 in this area. Both Fortran 77 and C fall far short of both Fortran 90 and C++ here, though C is given the nod over Fortran 77 because of C's support of data structures.

2.4 Object Oriented Programming

Because Fortran 90 does not support automatic inheritance, C++ is clearly the superior language along this dimension. Fortran 90's polymorphic (generic) features give it a manual (rather than automatic) inheritance capability, which places it ahead of both C and Fortran 77. Again the data structuring capabilities of C place it ahead of Fortran 77 in this general area.

2.5 Functional Programming

Because of the lack of recursion and data structures, Fortran 77 is clearly last in this category also. The other three languages all have these essential aspects for functional programming. Of the three, however, only Fortran 90 allows lazy evaluation; standard C (and hence C++) specifies a "sequence point" between function argument evaluation and evaluation of the function itself, precluding lazy evaluation of function arguments. Thus Fortran 90 must be ranked first of the four in this category. Polymorphism is also important in functional programming, and C++ is superior to C in this regard.

3 Numerical Robustness

In many respects numerical computation is the heart of computational science, as much of computational science involves the numerical simulation of mathematical models. Good numerical facilities are therefore key. Traditionally available numerical facilities have consisted primarily of single and double precision real data types, complex data type (single precision only), and rich libraries of functions providing a wide range of specific numerical computations. Occasionally additional capabilities, such as double precision complex or quadruple precision real numeric data types, are encountered. These traditional numeric facilities are adequate for a great many computational science applications.

In some cases, however, better selection of the numeric precision of an operand, or more information about the current numeric implementation environment than is available from the traditional tools, is needed to guarantee convergence, yield the most accurate result, or provide some other form of robustness of the numerical computation. The first of these- better selection of numeric precision-is provided by the Fortran 90 "type kind" mechanism and the ability to make user and implementation defined functions generic over different kinds of arguments. The second- information about the numeric environment-is provided by the numeric approximation model and the corresponding inquiry intrinsic functions that return environmental information related to this model. The next three subsections describe the first of these two enhancements to numerical robustness and the last two subsections describe the second one.

3.1 Numeric Kind Parameterization

The Fortran 90 KIND mechanism is a standardization, regularization, and generalization of the common "*size" extension to Fortran 77. For example, though they are not standard, it is common for REAL*4 to be the same as single precision REAL and REAL*8 to be equivalent to DOUBLE PRECISION; the "*size" syntax is a form of parameterization of the different kinds of real data types. Fortran 90 formalizes this concept of a type kind and associates an integer kind value with each intrinsic data type. These kind values are left implementation dependent (more about that in a minute), but if an implementation chooses the value 4 for single precision REAL and 8 for DOUBLE PRECISION, then alternative ways of specifying REAL are REAL(4) and REAL(KIND=4), and alternative ways of specifying DOUBLE PRECISION are REAL(8) and REAL(KIND=8). The complete formal syntax for the REAL type specifier in declaration statements is therefore:

```
REAL [([KIND= ] kind-value )]
```

Though some implementations may choose kind values 4 for single precision real and 8 for double precision real, so that these numbers are similar to the common "*4" and "*8" extensions, or to represent the number of bytes in an object of this kind (the original motivation for the 4 and 8), this is not appropriate for all architectures. Some machines, for example (e.g., the Cray vector supercomputers), use a different (from 4) number of bytes for single precision real objects. Possibly more important, an implementation might support more than one form of single precision, one the default REAL and the other(s) using the same amount of storage but a different representation method. For example, on a machine whose native arithmetic is not IEEE, a Fortran implementation might support a second form of (single precision) real based upon IEEE arithmetic. In Fortran 90 terms, this would merely be another kind of real, with a different type kind value. There is therefore no "obvious" set of kind values optimal for every implementation, which is the reason the kind values are left implementation dependent.

Fortunately there is a way to both isolate implementation kind value dependencies from application code and make the code more readable at the same time. That technique is to use the KIND intrinsic function to establish the correct values for (appropriately named) integer constants. (The KIND intrinsic returns the integer kind value for the type of its argument for that implementation.) For example, suppose that SINGLE and DOUBLE are the names of integer constants that have (somehow acquired) the proper kind value for single and double precision real, respectively. Then

```
REAL(SINGLE)    ...single precision variables...
REAL(DOUBLE)    ...double precision variables...
```

may be used to declare single and double precision real variables, as indicated. Alternatively, the "full blown" syntax for such declarations is:

```
REAL(KIND=SINGLE)    ...single precision variables...
REAL(KIND=DOUBLE)    ...double precision variables...
```

Prior to their use in this way, SINGLE and DOUBLE have to be given the proper values, which may be accomplished with the following declaration:

```
INTEGER, PARAMETER ::  SINGLE = KIND(1.0),                   &
                       DOUBLE = KIND(1D0)
```

(This uses the "entity oriented" declaration style of Fortran 90 which, in this case, condenses an INTEGER statement and a PARAMETER statement into one combined statement.) The first use of the KIND intrinsic function in the preceding example has a single precision real argument(1.0), so that the value it returns is the integer kind value for single precision real. The second instance of the KIND intrinsic has a double precision real argument (1D0), so it returns the integer kind value for double precision real. Programs that use this technique are portable, regardless of the kind values chosen by the implementation

Type COMPLEX uses the same KIND values as does type REAL-the complex kind is really the kind of the real/imaginary parts-and there is a complex kind for each real kind supported by the implementation. Thus

```
COMPLEX(SINGLE)     ...single precision complex variables...
COMPLEX(DOUBLE)     ...double precision complex variables...
```

is the best manner in which to declare complex objects.

Further, if the implementation's native arithmetic is not IEEE, but it supports an IEEE data type, the integer constant IEEE could be assigned the kind value specified by the implementation for IEEE arithmetic. Then IEEE variables could be declared with:

```
REAL(IEEE)          ...real variables of  type  IEEE ...
COMPLEX(IEEE)       ...complex variables  of type IEEE...
```

3.2 Precision Selection

This paves the way for discussion of that numerically useful feature of Fortran 90, the SELECTED_REAL_KIND intrinsic function, which allows the programmer to specify minimum decimal precision and/or exponent range properties. The SELECTED_REAL_KIND function has two optional integer arguments, at least one of which must be supplied, one for the desired decimal precision and the other for the desired (decimal) exponent range; SELECTED_REAL_KIND then returns the kind value for the "smallest" kind of real data type the implementation supports that meets or exceeds these specified conditions. An error condition exists if there is no such data type. Thus, for example,

```
INTEGER, PARAMETER :: P9 = SELECTED_REAL_KIND(9)
```

declares the integer constant P9 to have the real kind value appropriate to real objects with at least 9 decimal digits of precision. Variables meeting this requirement may be declared with the statement

```
REAL(P9)      ...9 digit (at least) precision real variable...
```

On a Sun workstation the value of P9 would be the same as DOUBLE (as defined above); on a Cray supercomputer the value of P9 would be the same as SINGLE. That is, REAL(P9) declares double precision variables on the Sun and single precision variables on the Cray. With this technique entire programs may be portably written based upon desired precision/range properties of the variables rather than upon implementation defaults for single and double real precision. This in itself is a powerful tool for numerical robustness.

Real constants of any kind can be formed by appending the kind value with an underscore to the default single precision real constants provided by the language. Thus, using the kind values defined above:

```
1.41_SINGLE and 1.41 are the same,
1.41_DOUBLE and 1.41D0 are the same,
1.41_IEEE is the IEEE version of 1.41,
1.41_P9 is the appropriate 9+ digit representation of 1.41, and
 typically will be equivalent to 1.41_SINGLE or 1.41_DOUBLE
```

3.3 Numeric Polymorphism

All of the computational intrinsic functions are generic over all of the type kinds provided by the implementation. Thus, for example, the result returned by COS(X) is the appropriate value of kind SINGLE, DOUBLE, IEEE, or P9, depending on whether X is of kind SINGLE, DOUBLE, IEEE, or P9, respectively. This generic property aids significantly in the development of portable robust application code. Intrinsic functions are similarly generic in Fortran 77, but a robustness deficiency of Fortran 77 is that user and implementation (and third party software vendor) supplied procedures cannot be made generic over argument types. Fortran 90 remedies this deficiency. (In this chapter "polymorphism" can be assumed to mean that generic properties, familiar in regard to the Fortran 77 intrinsic functions, may be specified for and are therefore extended to user-defined procedures.)

The interface block can be used to specify a generic name for a set of user supplied procedures, or to add procedures to an existing generic name. In the following two examples, the first interface block defines a new generic name (SMOOTH) associated with four specific procedures, and the second interface block extends the COS intrinsic functions to arguments of type RATIONAL.

```
INTERFACE SMOOTH                       ! SMOOTH is the generic name

 INTEGER FUNCTION SMOOTH_INT(AA)       ! for procedures   SMOOTH_INT
  INTEGER :: AA(:,:)                   !    SMOOTH_SINGLE
 END FUNCTION SMOOTH_INT               !  SMOOTH_DOUBLE
                                       !   SMOOTH_RATIONAL
 INTEGER FUNCTION SMOOTH_SINGLE(AA)
  REAL(SINGLE) :: AA(:,:)              ! AA is an assumed shape two-
 END FUNCTION SMOOTH_SINGLE            ! dimensional array in each case.

 INTEGER FUNCTION SMOOTH_DOUBLE(AA)
  REAL(DOUBLE) :: AA(:,:)
 END FUNCTION SMOOTH_DOUBLE

 INTEGER FUNCTION SMOOTH_RATIONAL(AA)
  TYPE(RATIONAL) :: AA(:,:)
 END FUNCTION SMOOTH_RATIONAL

END INTERFACE

INTERFACE COS                         ! Extends the generic properties
 FUNCTION RATIONAL_COS(X)             !  of COS to return results of
  TYPE(RATIONAL) :: RATIONAL_COS      !  type RATIONAL, assuming the
  TYPE(RATIONAL) :: X                 !  argument is of type RATIONAL.
 END FUNCTION RATIONAL_COS
END INTERFACE
```

In the SMOOTH example notice that the argument (AA) is a different type in each case but the function result is the same type (INTEGER in each case). The function result could also have been different in some or all cases. The only requirement for such a generic set is that the type/kind/rank pattern for the set of dummy arguments be unique in each specific case. In the COS example, note the conceptual similarity with the extension of the "+" operator in section 1 on page 1, User Defined Types and Operators.

These generic definition capabilities make it practical for the programmer to specify precision with the SELECTED_REAL_KIND function. The program can then be geared toward the optimal precision for the application rather than focussing on the specific precisions provided by the implementation. This not only is a more natural way to develop numerical software, but it contributes to numerical robustness as well. Fortran 77, C, and Fortran 90 versions of (one variation of) the routine SMOOTH are given in section 2.1.

3.4 The Numeric Approximation Model

Dynamic access to the numerical properties of the implementation can enable the development of portable numerically robust software. Fortran 90 provides such access through a combination of a model of the methods for approximating real values and a set of corresponding intrinsic functions for extracting model values. These intrinsic functions, of which there are a total of 16, are called the environmental inquiry (nine) and numeric manipulation (seven) intrinsic functions.

Each kind of real number is modeled by

$$x = sb^e \sum f_i b^{-i} \qquad\qquad i = 1, \ldots, p \tag{1}$$

where

x is the real value

s is (the sign of the value)

b is the radix (base) and is usually 2; b is constant for a given real kind

p is the base b precision; p is constant for a given real kind

e is the base b exponent of the value

f_i is the ith digit, base b, of the value; $0 < f_i < b$;

f_1 may be 0 only if all f_i are 0

The principal characteristics of a given real kind are its values for b and p and its range for e.

IEEE arithmetic is based upon a binary (b=2) representation in which p=24 (single precision), p=56 (double precision), and $-127 < e < 127$; IEEE uses what would be an exponent of -127 to represent zero and NaNs (illegal or out-of-range values). A nonbinary example is that of IBM 370 real arithmetic, in which b=16, p=6 (single precision), p=14 (double precision), and -127e127. In most implementations the main difference in the representation of different real kinds is the value of p, though it is possible (and occasionally happens) for the value of b or the range of e to vary between kinds.

3.5 Environmental Inquiry

There are nine important model-related values that are fixed for and characteristic of a given real kind; these values constitute a sort of "fingerprint" for the kind and are useful in various computational contexts. The environmental inquiry intrinsic functions allow the programmer to access these values at any time, in (a) numerical expressions in declaration of data objects and(b) any computation. Each of these environmental intrinsics takes a single argument, which may be a constant, scalar variable, or array variable. If it is a variable, its value need not be defined because only the type kind of the argument is used by these functions, not the value of the argument. These nine characteristic values and related environmental intrinsic functions shown in Table 2:

characteristic values of a real kind	intrinsic function name
the decimal precision	PRECISION
the decimal exponent range	RANGE
the largest value	HUGE
the smallest value	TINY
a small value compared to 1; $b^1 - p$	EPSILON
the base b	RADIX
the value of p	DIGITS
the minimum value of e	MINEXPONENT
the maximum value of e	MAXEXPONENT

Table 2:

Of these nine values, HUGE, TINY, and EPSILON are especially useful in writing robust portable numerical software, though in specialized contexts the others are tantamount to indispensable.

The seven numeric manipulation functions allow the programmer to access important model values related to a given specific value of that kind. The given value is the value of the (first) argument, which may be any expression that has a (defined) value of that kind. (Three of these seven functions also have a second argument.) These seven useful values and the related numeric manipulation intrinsic functions are shown in Table 3:

values related to the argument value	intrinsic function name
exponent value, e	EXPONENT
fractional part, $s \sum f_i b^{-i}$	FRACTION
nearest value (second argument specifies direction)	NEAREST
reciprocal of the relative spacing near the argument	RRSPACING
change e by value of the second argument	SCALE
set e to value of second argument	SET_EXPONENT
absolute spacing near the argument	SPACING

Table 3:

An (idealized) example will illustrate the use of these functions for writing robust portable code. In this example Newton's method is used to find the root to maximum accuracy (for the real kind being used) of a function F, in the minimum number of iterations. This example assumes the function F and its derivative function DF are available, and that the value X is already established in a region in which Newton's method will converge to the root.

```
...
do
 DX = F(X)/DF(X)                  ! Compute the next delta-X.
 X = X-DX
 if (DX<2*spacing(X)) exit        ! Stop if near the spacing
end do                            !  limits of that kind
...                               !      in this region.
```

4 Data Parallelism

Fundamental physical phenomena, such as thermal generation/dissipation properties and electronic signal speeds, place theoretical and practical limits on the computation speeds of single processor systems. Though these limits are currently roughly in the "gigaflop" (a billion numerical operations per second) range, some contemporary applications of computational science require substantially greater speeds, as will most applications on the scale of grand challenge problems. It is becoming more feasible to scale up computational capacity by adding processors than by increasing single processor speed. It is likely that all future applications involving massive amounts of computation will make significant use of parallelism.

Applications may employ (either or both of) two principal forms of parallelism, which here will be termed "data parallelism" and "process parallelism". Data parallelism involves performing a similar computation on many data objects simultaneously. The prototypical such situation, especially for computational science applications, is simultaneous operations on all the elements of an array-for example, dividing each element of the array by a given value (e.g., normalizing the pivot row in matrix reduction).

For the purposes here data parallelism will mean concurrent operations on array elements. Process parallelism involves performing different processes in parallel, where a process is an arbitrary sequence of computations. A subroutine, for example, is a process, as is any block of statements. An increasingly important form of process parallelism is evaluating two expressions simultaneously-two actual arguments, for example, in a procedure call. A number of approaches to specifying process parallelism have been developed, and process parallelism standards are beginning to appear [3]. This type of parallelism will be discussed in the next release of this chapter.

Process parallelism may be important in many applications of computational science, but because of the ubiquitousness of arrays in such applications, data parallelism is likely to be useful in most and critical in many. This section therefore focuses on parallel array operations, with particular emphasis on the extraordinarily rich set of such operations in Fortran 90.

4.1 Array Operations

APL was perhaps in many respects the computer language that pioneered the concept of an array as a data object in its own right and not just a cartesian collection of data objects. Operations can be performed on whole arrays, such as

$$C \leftarrow A + B \tag{2}$$

where A, B, and C may all be arrays; for example A, B, and C could all be two dimensional arrays of size 200x300. The meaning of such an operation is $C_{i,j}$ $A_{i,j}$ + $B_{i,j}$ for all 200 values of i and 300 values of j, for a total in this case of 60,000 individual computations involving the array elements. The APL model, therefore, is concurrent element- by-corresponding-element computation for all the elements in the array(s). (Though few implementations of APL capitalized on this conceptual parallelism.)

In addition to supporting all the usual mathematical operations, in a manner analogous to that shown above for addition, APL defined other whole array operations necessary for a reasonably complete paradigm in which arrays are objects in their own right. These include reduction operations (e.g., +/A means sum all the elements of array A), construction operations (e.g., i4 constructs a vector of four elements whose values are 1,2,3,4), and inquiry operations (e.g., if ρB returns the shape of array B-a vector whose size is the rank of B and whose element values are the corresponding dimension sizes of B-then */ρB computes the total number of elements in array B). All such operations can be combined into more arbitrarily complex array-valued expressions (e.g., for a one-dimensional array Q, */Q + i*/ρQ is evaluated right to left (that's APL!) to generate a 1,2,3,... vector the size of Q which is added to Q and the resulting elements multiplied together; if Q is the vector (3,2,4), then the value of */Q+i*/$\Phi\rho$Q = */((3,2,4)+(1,2,3))=*/(4,4,7)=56). APL's introduction well before data parallelism became physically practical and when the processing of dynamic languages was inefficient, coupled with (what turned out to be) unpopular characteristics such as arcane notation and unnatural right-to-left evaluation, resulted in it not being used for many practical applications.

The Fortran 90 array operations provide virtually all of the element-by-element data parallel features of APL, without its disadvantages. These operations are provided as natural extensions of scalar operations, functions, and expressions, using familiar Fortran rules for expression evaluation. Reduction, construction, and inquiry operations are provided with the addition of a number of meaningfully-named intrinsic functions. Care was taken to ensure that these operations can be efficiently implemented on contemporary parallel processing systems.

Generally speaking, except in a few contexts in which an expression is restricted to be scalar, any Fortran 90 expression may have array operands and the result is array valued. Fortran 77 allowed only scalar expressions; (almost) all such expressions in Fortran 90 may be data parallel array valued as well. (Scalar expressions are required in control contexts such as IF statement control conditions (scalar logical expression), DO loop indexing expressions, and I/O specifiers such as unit number, file names, open statement specifiers, etc.) Examples of array operations are as follows (any or all of the variables may be arrays):

```
C = A+B
print*, P*Q-R, S
call T3(X,Q,Z-V)
```

Note that in these cases the array expressions are indistinguishable from scalar expressions-you need to know from other contexts that these variables have been declared as arrays-but each potentially represents millions of parallel computations. If A, B, C, P, Q, and R are two-dimensional arrays, and Z and V are one-dimensional arrays, these three statements could be written in the following equivalent form which clearly identifies the array operations.

```
C(:,:) = A(:,:)+B(:,:)
print*, P(:,:)*Q(:,:)-R(:,:), S
call T3(X,Q(:,:),Z(:)-V(:))
```

Functions may be defined as array valued and hence be operands in array-valued expressions. These are described in section 4.4, along with array-related intrinsic functions.

Thus, element-by-element data-parallel array- valued expressions are a straight forward natural extension/generalization of scalar expressions, with arrays replacing scalars as operands.

Conformability Requirement The principal requirement in forming an array expression is conformability of the operands. This means that each operand of an array operation must have the same rank and the same number of elements along each dimension-that is, conformable arrays have exactly the same shape. The result of such an operation is, of course, conformable with the operands, and the value of each element of the array result is the scalar computation involving the corresponding elements of the array operands.

Thus if A and B are the following 2x3 arrays:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 5 \\ 1 & 7 & 4 \end{bmatrix} \mathbf{B} = \begin{bmatrix} 5 & 4 & 1 \\ 2 & 2 & 3 \end{bmatrix} \tag{3}$$

the result of $\mathbf{A} + \mathbf{B}$ is

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} 7 & 7 & 6 \\ 3 & 9 & 7 \end{bmatrix} \tag{4}$$

and the result of $\mathbf{A} \times \mathbf{B}$ is

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} 10 & 12 & 5 \\ 2 & 14 & 12 \end{bmatrix} \tag{5}$$

If there is more than one operation in an expression, the (array-valued) result of the first subexpression is an operand for the second operation, and so on. For example, for $\mathbf{A}$ and $\mathbf{B}$ as given above, in the expression $\mathbf{A} + \mathbf{B} \times \mathbf{A}$ the result of $\mathbf{B} \times \mathbf{A}$ is added to $\mathbf{A}$; thus the result of $\mathbf{A} + \mathbf{B} \times \mathbf{A}$ is

$$\begin{bmatrix} 2 & 3 & 5 \\ 1 & 7 & 4 \end{bmatrix} + \begin{bmatrix} 10 & 12 & 5 \\ 2 & 14 & 12 \end{bmatrix} = \begin{bmatrix} 12 & 15 & 10 \\ 3 & 21 & 16 \end{bmatrix} \tag{6}$$

Note that, for example, a 3x2 array is not conformable with a 2x3 array-they have the same rank and total number of elements, but corresponding dimensions don't have the same size-and thus two such arrays cannot be the operands in the same array operation. The only exception to this basic conformability rule is in the event that one of the operands is a scalar. In this case the scalar is "broadcast" into an array conformable with the other operand, the value of each element of this broadcast array being that of the scalar. For example, $\mathbf{B} + \mathbf{2}$ is a valid array operation and (assuming $\mathbf{B}$ is as given above) the result of $\mathbf{B} + \mathbf{2}$ is

$$\begin{bmatrix} 5 & 4 & 1 \\ 2 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 7 & 6 & 3 \\ 4 & 4 & 5 \end{bmatrix} \tag{7}$$

Common uses of (broadcast) scalars in array operations are to initialize and scale arrays:

$\mathbf{A} = \mathbf{0}$! sets each element of $\mathbf{A}$ to zero
$\mathbf{B} = (\mathbf{B} + \mathbf{1})/\mathbf{2}$! add 1 to each element of $\mathbf{B}$ then take half the result

This last example illustrates a key aspect of the Fortran 90 array operations: in an array-valued assignment the effect is as if the right-hand side array value is fully evaluated before any assignment takes place. Otherwise it is possible (though not in this simple example) for the right-hand-side array value to be affected before its evaluation is complete. Thus the Fortran 90 conceptual model is that all elements of the right-hand-side array value are computed in parallel (or in any order) before any assignment takes place, and any implementation is allowed that guarantees this behavior.

An example where this rule is important is in the pivoting step in section 4.5. There the pivot row is normalized with the array operation

```
G(P,:) = G(P,:)/G(P,K)
```

This operation uses an array section, G(P,:); array sections are described in section 4.2 below. In this case G(P,:) is the Pth row (pivot row) of matrix G and G(P,K) is the pivot element (K is the pivot column). The normalization scales the row so that the pivot element value is one. Note that if the value of this element is changed to one before the evaluation of the right-hand side is complete, then the row is not properly normalized (this is typical of a common error in sequential programming). Therefore, array operations should not be thought of as "loops" over the array elements, which implies a sequentially of the operations; in general, thinking of array operations as loops gives incorrect results when assignment is involved. Array operations should be thought of as integral/parallel computations.

Array Constructors Array values may be explicitly constructed using the array constructor and, if the desired resultant array has dimension higher than one, the RESHAPE intrinsic function; an array constructor forms a one-dimensional array. An array constructor is simply a list of the element values of the result, separated by commas and enclosed in (/... /) delimiters. The above APL $\iota 4$ example may be written as the Fortran 90 array constructor

```
(/1,2,3,4/)
```

(though there is a more general way of specifying such an "iota" sequence). Each element in this example is a simple scalar constant. In general, any element in an array constructor can be any scalar expression. If they are all constants, however, then such a constructor (possibly combined with the RESHAPE function - see below) represents an array constant and may appear in a PARAMETER declaration. Array constructors (combined with RESHAPE) are, therefore, the Fortran 90 means of representing array constants.

If each element had to be explicitly listed, the array constructor would not be very practical for specifying very large array values. Therefore two forms for array constructor list items are provided in addition to scalar expressions. These are implied-do constructs and array expressions. The first of these has the form

```
(expression-list, index-variable = first-value, last-value[,increment])
```

The index-variable is a scalar integer variable serving as an iterative index in exactly the same manner as implied-do loops in Fortran 77 I/O statements. One simple application of an implied-do in an array constructor is to generate any iota sequence. For example, the array constructor

```
(/(k, k=1,n)/)
```

generates the vector whose element values are 1,2,3,4,5,...n; if n is 4 the result is identical to (/1,2,3,4/). As another example, a vector of a million alternating ones and zeros,

```
(/1,0,1,0,1,0,1,$\ldots$/), can be specified with (/ (1,0, j=1,500000) /).
```

The implied-do simply replicates the list the specified number of times, and if the index-variable is an operand in an expression in the expression-list, each replication of that item uses the corresponding value of the index-variable. The items in the implied-do expression-list may be any of the three forms allowed in the array constructor itself-scalar expressions, implied-do constructs, and array expressions. The two examples above used only simple scalar expressions in the implied-do lists.

An array expression of any dimension may appear in an array constructor. For example, if **A** is a 1000×1000 array then

```
(/A+1.3/)
```

is an array constructor of one million elements, each having a value of 1.3 more than the corresponding element value of **A**. The elements of **A**+1.3 are placed in the array constructor in the familiar Fortran "column-major" order, that is column by column by running down the first dimension and then the second. Thus (/ A+1.3 /) is equivalent to

```
(/((A(j,k)+1.3,j=1,1000),k=1,1000)/)
```

Implied-do constructs may be used for a different order. For example, if a row by row vector of elements of **A**+1.3 is desired, rather than column by column, the following array constructor would do the job:

```
(/((A(j,k)+1.3,k=1,1000),j=1,1000)/)
```

Finally, a simple form of the RESHAPE intrinsic function can be used to reshape the (one-dimensional) result of an array constructor into the desired shape. The form that this takes is

```
RESHAPE (array-constructor, shape-vector)
```

where the shape-vector has one element for each dimension of the desired array shape, and the value of each shape-vector element is the number of elements in that dimension in the target array. For example, a 1000×1000 identity matrix of type real can be specified as the constant named Ident_1000 by the declaration

```
real, parameter :: Ident_1000 =                                          &
                   RESHAPE((/(1.0,(0.0,k=1,1000),j=1,999),1.0/),(/1000,1000/))
```

Thus the array constructor, coupled with the RESHAPE intrinsic, is an extremely powerful tool for constructing array values, including array constants.

Masked Array Assignment A "mask" is an array of type logical. A masked array operation is one in which a mask conformable to the result of the operation is used to specify that only a subset of the parallel element operations are to be performed. This functionality is available in some of the intrinsic functions and for array assignment. In the latter case an array-valued assignment is placed under mask control in a WHERE statement, the general form of which is:

```
WHERE (mask) array-assignment-statement
```

The WHERE mask must be conformable with the array on the left of the assignment (which must be conformable with the expression on the right of the assignment). For all those mask elements that have the value .TRUE. the corresponding element assignments take place; where the mask is .FALSE. the assignment is not made. A typical example of the use of masked array assignment is

```
WHERE (C.gt.0)  A = B/C
```

which suppresses the division and assignment for those elements of **C** that have value zero (or negative, in this case). By the rules of conformability, arrays **A**, **B**, and **C** are all conformable and the (array-valued) logical expression C.gt.0 is therefore a mask conformable with these arrays. It is often the case that, as in this example, a WHERE mask is the result of a logical expression involving one or more of the assignment operation operands.

Several assignments can be placed under the control of a single mask, in which case the WHERE takes a block form:

```
WHERE (mask)
     array-assignment-1
     array-assignment-2
     ...
END WHERE
```

Any number of array assignments can be grouped in this manner; of course, they all have to be conformable with the mask.

The forms of WHERE described above leave unassigned some elements of the array on the left hand side of the assignment statement. An extension of the block form of WHERE, the ELSEWHERE option, allows a value to be given to the left-hand-side array elements where the mask is .FALSE. This takes the form:

```
WHERE (mask)
     array-assignment-1
     array-assignment-2
```

```
     ...
ELSEWHERE
     array-assignment-n+1
     ...
END WHERE
```

A simple example of this last form of WHERE is

```
WHERE (C.gt.0)
     H = B/C
ELSEWHERE
     H = B
END WHERE
```

In this case those elements of **H** for which **C** is less than or equal to zero are simply assigned the corresponding value of **B**. This is an important form of WHERE, because it results in a fully defined array **H** that can be used in subsequent array operations. Without the ELSEWHERE the array H might end up not being fully defined, in which case it cannot be used in other array expressions.

Assumed-Shape Dummy Arguments One place that Fortran 77 permitted the appearance of an (unsubscripted) array name was as a procedure argument. Given the limited Fortran 77 concept of an array, this was sufficient to be considered as passing the array object to the procedure. However, a Fortran 77 array always occupied a block of contiguous storage, and therefore only the location of this block needed to be passed to the procedure. The procedure could treat this as the location of a (contiguous) array of the same shape, as an array of a different shape, or even as a scalar. This is not sufficient for Fortran 90, where arrays are full-fledged objects, the conformability rules apply, and array objects need not occupy contiguous storage (as is the case with many array sections-see section 4.2).

In Fortran 90, arbitrary array expressions may be used as actual arguments in procedure calls. The called procedure must handle these arguments properly as array-valued objects, which is not always possible if just a single location is passed. Assumed-shape dummy arguments solve this problem. They accommodate the passing of array "descriptors", which contain descriptive information about the array in addition to its location. This additional information includes the rank (number of dimensions) of the array object being passed, the type and size of each element, the number of elements in each dimension, and the "stride" in each dimension; the stride represents the spread between elements in a dimension and hence accounts for any departure from contiguity. Thus any array expression can be passed to an assumed-shape dummy argument. (Any array expression can be passed to an "old fashioned" dummy argument as well, but that might result in expensive behind-the-scenes packing into and unpacking from contiguous temporary storage.)

An assumed-shape dummy argument is declared with a colon for each dimension, as in the following example in which T is a scalar, **U** is a two- dimensional assumed-shape array, and **V** is a one-dimensional assumed-shape array.

```
SUBROUTINE CALC3(T,U,V)
     REAL   T,U(:,:),V(:)
     ...
END  SUBROUTINE CALC3
```

In a call to CALC3, any two-dimensional array expression (of type real) may be passed to **U** and any one-dimensional array expression may be passed to **V**; conversely, a two-dimensional real array must be passed to **U** and a one-dimensional real array must be passed to **V**. In effect the colons in the declarations for **U** and **V** instruct CALC3 to accept the descriptor information supplied by the calling program. **U** and **V** then exactly represent the corresponding array objects in the actual argument list and may be used in array operations in the body of CALC3. If CALC3 is an internal procedure in the calling program, or is a module procedure in a module being used by the calling program, the proper association between the assumed-shape dummy arguments and the corresponding actual arguments is transparently accomplished. If CALC3 is an external procedure, however, an interface block for CALC3 must be provided in the calling program so that it knows that assumed-shape dummy arguments are the receivers and therefore efficient descriptors can be passed; otherwise the calling program cannot assume the dummy arguments are assumed-shape and must therefore provide a contiguous actual argument, packing and unpacking the array(s) if necessary. An adequate interface block for CALC3 is:

```
INTERFACE
   SUBROUTINE CALC3(T,U,V)
        REAL T,U(:,:),V(:)
   END SUBROUTINE CALC3
END INTERFACE
```

4.2 Array Sections

A portion of an array containing more than one element is called an array section. Often an array operation is needed on an array section, not the entire array. The earlier example of normalizing the pivot row of a matrix is a case in point. In this example it was exactly one row of the matrix that was of interest in the computation, not the whole array. In this case the array section is one row of a two-dimensional array; in general virtually any rectilinear subset of an array can be an array section and hence an object that can be used in array operations. An array section may be of any dimensionality up to and including the dimensionality of the array on which the section is being defined.

An array element, which is a scalar, is of course specified by the array name and a subscript value for each dimension. The general form for this is the familiar

```
array-name ( subscript-1, subscript-2, subscript-3, ...)
```

where the number of subscripts is the dimensionality of the array and each subscript is a scalar integer expression, or scalar subscript for short. An array section is specified by replacing at least one scalar subscript by a "section subscript". A section subscript is a sequence of scalar subscript values for that dimension, and thus a section subscript may be thought of (and constructed as) a one-dimensional array of subscript values, called a vector subscript. If (only) one scalar subscript is replaced by a vector subscript the result is a one-dimensional array section; if two scalar subscripts are replaced by vector subscripts the result is a two-dimensional array section, and so on. An array section has dimensionality equal to the number of vector subscripts it has.

It's time for an example. Consider the 5x6 array **Q** as shown.Three sections of **Q** are shown in bold: the entire second column (a one- dimensional section), the 2x2 upper right hand corner of **Q** (a two-dimensional section), and the last half of the fifth row of **Q** (a one-dimensional section).

$$\mathbf{Q} = \begin{bmatrix} 13 & \mathbf{11} & 25 & 2 & \mathbf{1} & \mathbf{9} \\ 9 & \mathbf{3} & 31 & 14 & \mathbf{52} & \mathbf{27} \\ 16 & \mathbf{45} & 54 & 36 & 15 & 20 \\ 7 & \mathbf{20} & 18 & 19 & 8 & 19 \\ 37 & \mathbf{56} & 54 & \mathbf{66} & \mathbf{77} & \mathbf{90} \end{bmatrix} \tag{8}$$

$$\begin{array}{ll} \mathrm{Q}((/1,2,3,4,5/),2) = (/11,3,45,20,56/) & \text{! the second column} \\ \mathrm{Q}((/1,2/),(/5,6/)) = \begin{bmatrix} 1 & 9 \\ 52 & 27 \end{bmatrix} & \text{! upper right corner} \\ \mathrm{Q}(5,(/4,5,6/)) = (/66,77,90/) & \text{! last part of 5th row} \end{array}$$

Note that all of the vector subscripts in these examples could be written with implied-do constructs:

$$\begin{array}{ll} \mathrm{Q}((/(\mathrm{k},\mathrm{k}=1,5)/),2) & \text{! the second column} \\ \mathrm{Q}((/(\mathrm{k},\ \mathrm{k}=1,2)/),(/(\mathrm{k},\mathrm{k}=5,6)/)) & \text{! the upper right corner} \\ \mathrm{Q}(5,(/(\mathrm{k},\mathrm{k}=4,6)/)) & \text{! last part of 5th row} \end{array}$$

The implied-do form is more extensible and, for large sections, considerable more compact than explicit lists. Implied-do constructs are also useful for regularly-spaced but noncontiguous vector subscripts. For example,

```
Q((/(k,k=1,5,2)/),2) = Q((/1,3,5/),2) = (/11,45,56/)
```

The implied-do form is common enough that a more readable shorthand notation, called a "triplet subscript", is also provided for the indexed-do control triplet.

A triplet subscript is just the indexed-do control values, separated by colons rather than commas, with the last one (the increment or stride value) optional. Thus using triplet notation the above four examples may be written (much more clearly!) as:

```
Q(1:5,2)     ! the second column
Q(1:2,5:6)   ! the upper right corner
Q(5, 4:6 )   ! last part of 5th row
Q(1:5:2,2)   ! every other element of 2nd col.
```

A further simplification is provided in that if the initial triplet value is omitted the lower bound of that dimension is assumed, and if the second triplet value is omitted the upper bound of that dimension is assumed. Thus the most compact way of expressing these four sections is:

```
Q(:,2)     ! the second column
Q(:2,5:)   ! upper right corner
Q(5,4:)    ! last part of the 5th row
Q(::2,2)   ! every other element of 2nd col.
```

The form Q(:,:) is a section that is in fact the entire array **Q**, which explains an example early in this section. (Note that Q(:,:) also has the form of an assumed-shape dummy argument declaration; there is no ambiguity, however, because if this notation appears in an array expression it always represents an array section.)

Returning to the more general form of vector subscripts, though the above examples employ array constructors, any one-dimensional array expression is permitted. The only requirement is that the value of each element of the vector subscript be a valid subscript value for that dimension. A common form for vector subscripts is a one-dimensional integer array name (or section), whose element values have been previously established. This form is extremely useful for indirect access, such as indexing into a table; e.g., table elements may be retrieved (or set) by subscripting the table array with an array containing the desired table index values.

A couple of final examples will complete this introduction to array sections. Again, for purposes of explicitness, array constructors will be used, but in practice more concise one-dimensional array names or sections are more likely. First, an array section need not be as easily depictable graphically as the examples above. For the array **Q** defined above, consider the section

```
Q((/2,5,3/),(/6,4/))
```

This represents the array section

$$\begin{bmatrix} Q_{2,6}, Q_{2,4} \\ Q_{5,6}, Q_{5,4} \\ Q_{3,6}, Q_{3,4} \end{bmatrix} = \begin{bmatrix} 27 & 14 \\ 90 & 66 \\ 20 & 36 \end{bmatrix} \tag{9}$$

This section can be used in any array expression in which a 3x2 array object is valid. It may also appear on the left hand side of an array assignment, in which case the (1,1) element of the right hand side expression value gets assigned to $Q_{2,6}$, the (3,2) value of the right hand side gets assigned to $Q_{3,4}$, and so on.

A vector subscript may contain more elements than the size of that array dimension. In this case there are duplicate values, since all of the values must be within the array dimension range. Indeed, subscript values may be duplicated in a vector subscript even if the size of the vector is less than the array dimension (the only requirement is that the subscript values must be within range). Both of these cases are illustrated in the following example, which specifies a 7x4 section from the elements of **Q**.

$$Q((/4,1,2,3,4,2,5/),(/1,4,4,3)) = \begin{bmatrix} Q_{4,1} & Q_{4,4} & Q_{4,4} & Q_{4,3} \\ Q_{1,1} & Q_{1,4} & Q_{1,4} & Q_{1,3} \\ Q_{2,1} & Q_{2,4} & Q_{2,4} & Q_{2,3} \\ Q_{3,1} & Q_{3,4} & Q_{3,4} & Q_{3,3} \\ Q_{4,1} & Q_{4,4} & Q_{4,4} & Q_{4,3} \\ Q_{2,1} & Q_{2,4} & Q_{2,4} & Q_{2,3} \\ Q_{5,1} & Q_{5,4} & Q_{5,4} & Q_{5,3} \end{bmatrix} \tag{10}$$

Note that rows one and five of this section are identical, as are rows three and six and columns two and three. Many elements of **Q** therefore appear twice in this array section and two elements, $Q_{2,4}$ and $Q_{4,4}$, each appear four times. Array sections with multiple appearances of a given parent array element are perfectly legitimate array operands in array expressions, but such sections must not appear on the left hand side of array assignments.

4.3 Dynamic Arrays

Fortran 90 has three varieties of dynamic arrays. All three allow array creation at run time with sizes determined by computed (or input) values. These three varieties of dynamic arrays are:

- automatic arrays
- allocatable arrays
- pointer arrays

Automatic Arrays Automatic arrays are local arrays whose sizes depend upon values associated with dummy arguments. Automatic arrays are automatically created (allocated) upon entry to the procedure and automatically deallocated upon exit from the procedure. The size of an automatic array typically is different in different activations of the procedure. Examples of automatic arrays are:

```
function F18(A,N)
```

```
 integer  N                     ! A scalar
 real A(:,:)                    ! An assumed shape array
 real F18(size(A,1)  )          ! The function result itself is
                                !  an automatic array.

 complex  Local_1(N,2*N+3)      ! Local_1 is an automatic array
                                !  whose size is based on N.

 real Local_2(size(A,1),size(A,2))     ! Local_2 is an automatic array
                                !  exactly the same size as A.

 real Local_3(4*size(A,2))      ! Local_3 is a one-dimensional
                                !  array 4 times the size of
                                ! the second dimension of A.
 ...                            !
end function F18
```

Note the importance of the intrinsic inquiry functions, such as SIZE (which returns the argument array size in a specified dimension) in declaring automatic arrays. Fortran 90 provides a number of inquiry functions that are allowed to appear in declarations. Array bounds and sizes, character lengths, and type kinds may all be specified with expressions involving these inquiry functions. Roughly, a specification expression, as such expressions are called, is a scalar integer expression that has operands whose values are determinable upon entry to the procedure. Such operands include constants, references to intrinsic procedures, and variables accessible through dummy arguments, modules, common, and (in the case of module and internal procedures) the host procedure.

Allocatable Arrays Allocatable arrays are those explicitly declared ALLOCATABLE. An allocatable array may be local to a procedure or may be placed in a module and effectively be global to all procedures of the application. An allocatable array is explicitly allocated with the ALLOCATE statement, and deallocated either explicitly with the DEALLOCATE statement or, if it is a local array for which SAVE has not been specified, automatically upon exit from the procedure. (If SAVE has been specified, local allocatable arrays can persist from one execution of the procedure to the next - they must be explicitly deallocated with a DEALLOCATE statement.) A global allocatable array persists until it is explicitly deallocated, which may occur in a procedure different from the one in which it was allocated. Use an allocatable (or pointer) array if its size depends on a computed value other than a dummy argument or variable in a module, common, or the host. The allocation status (allocated or not allocated) of an allocatable array may be tested with the ALLOCATED intrinsic function. Examples of allocatable arrays are:

```
subroutine Peach
```

```
 use Recipe                     ! Accesses global allocatable array, Jam.

 real, allocable :: Pie(:,:)    ! Pie is a 2-dimensional allocatable array.
 ...
 allocate ( Pie(N,2*N )   )     ! Allocate a local allocatable array.

 if (.not.allocated(Jam)) allocate ( Jam(4*M) )
                                ! Allocate a global allocable array if
                                ! it is not already allocated.
 ... deallocate ( Pie )
 ...
end subroutine Peach

module Recipe                   ! Jam is a global allocatable array, and
 real, allocable :: Jam(:)      ! can be allocated and deallocated in
 ...                            ! any procedure(s) using this module.
end module Recipe
```

Note that the declared bounds for allocatable arrays are simply colons, indicating that these will be provided later, at the time of allocation. This makes allocatable array declaration appear similar to assumed-shape dummy argument declaration, appropriate because the "deferred" nature of the sizes of the dimensions is conceptually similar.

Pointer Arrays Pointer arrays are similar to allocatable arrays in that they are explicitly allocated with the ALLOCATE statement to have arbitrary computed sizes and are explicitly deallocated with the DEALLOCATE statement. Examples of pointer arrays are given in section 1, in the subsection entitled Pointers. These examples also illustrate target arrays and the use of pointer assignment, the latter of which cannot be used with allocatable arrays. Additional, very simple, examples of pointer arrays result by replacing "allocatable" with "pointer" in the preceding examples of allocatable arrays.

In addition, pointer arrays can be used as aliases for ("point to") other arrays and array sections; the pointer assignment statement is used to establish such aliases. The target for pointer associations (as such aliasing is called) may be other explicitly allocated arrays, or static or automatic arrays that have been explicitly identified as allowable targets for pointers. The association status of a pointer array may be tested with the ASSOCIATED intrinsic function. Finally, pointer arrays may be dummy arguments and structure components, neither of which are allowed for allocatable arrays. Given this apparent similarity between allocatable arrays and pointer arrays, what is the fundamental distinction between these two forms of dynamic arrays, and when should allocatable arrays be used rather than pointer arrays? Pointer arrays subsume all of the functionality of allocatable arrays, and in this sense allocatable arrays are never needed-pointer arrays could always suffice. The problem with pointer arrays is efficiency.

Though pointer arrays must always point to explicit targets, which makes optimization practical that would otherwise be infeasible, pointer assignment makes optimization of pointer arrays much more difficult than for allocatable arrays. Because of their more limited nature and functionality, allocatable arrays are just "simpler" and can be expected to be more efficient than pointer arrays.

Therefore, when all that is needed is simple dynamic allocation and deallocation of arrays, and automatic arrays are not sufficient, use allocatable arrays. A common example of this is if a "work array" is needed of a size dependent upon the results of a local computation. If, on the other hand, the algorithm calls for a dynamic alias, of for example a "moving" section of a host array, then a pointer array is probably indicated.

4.4 Array-valued Functions

Fortran 90 functions can return array-valued results. A number of intrinsic functions always return array values and most intrinsic functions can return array values. In addition, user-written functions may be array valued. Array-valued functions provide two (related) tremendous benefits. First, array-valued functions may be used as operands in array-valued expressions, allowing data-parallel computations to be expressed in the most natural forms. Second, this facilitates composing a computation from array-valued subexpressions, which often can be evaluated in parallel. Thus array-valued functions provide increased opportunities to combine process parallelism with data parallelism in a natural way.

For example, where **g** and **u** are large conformable two-dimensional arrays, **g** + cshift(u,1,2) - cshift(u,-1,2) is an expression similar to some commonly found in seismic modeling computations. (CSHIFT is the intrinsic function that circularly shifts the first argument - an array - the amount of the second argument along the dimension specified by the third argument.) This expression very clearly expresses the nature of the computation, which the following diagram stylizes for a single element of the result of the expression,
once one is thinking data parallel, and offers the compiler the opportunity to evaluate any subexpressions in parallel. This might be particularly appropriate, for example, if g were itself a function reference, in which case it probably would be advantageous to evaluate the subexpression cshift(u,1,2)-cshift(u,-1,2) in parallel with the evaluation of **g**.

The two categories of array-valued intrinsic functions are known as "transformational" and "elemental" functions. A transformational function accepts an input array and produces a different array as the result-it "transforms" the input array into something else, possibly even a differently shaped array, or even a scalar. (A transformational function can even transform a scalar into an array.) CSHIFT is an example of a transformational function, albeit a very simple one with a result that is conformable with its (first) argument. Intrinsic function MATMUL (matrix multiplication) is an example of a transformational function that returns an array result of different shape than (either of) its arguments. The reduction functions, SUM, PRODUCT, COUNT, etc., are examples of transformational functions that "reduce" array arguments to scalar results. The Fortran 90 array transformational intrinsic functions (42 in all) are listed in Table 4.

transformational intrinsic function	comment
environmental inquiry functions (9)	see Section sec3.5
array functions (21)	see below
ASSOCIATED	check association status of pointer
BIT_SIZE	number of bits in an integer
DOT_PRODUCT	mathematical dot product of two vectors
KIND	see Section 3.1 and Section 3.2
LEN	length of a character string
MATMUL	mathematical matrix product
PRESENT	check presence of an optional argument
REPEAT	replicate a character string
SELECTED_INT_KIND	see Section 3.1 and Section 3.2
SELECTED_REAL_KIND	see Section 3.1 and Section 3.2
TRIM	remove trailing blanks from a string
TRANSFER	transfer bit pattern to a different type

Table 4:

The elemental intrinsic functions are (most of) those defined with scalar dummy arguments. Such functions may be called with array actual arguments, and return an array result conformable with the actual argument. Each element of the result is what would have been obtained if the function had been called with just the corresponding (scalar) element of the actual argument. Thus an elemental function is automatically (and conceptually in parallel) applied to each element of the actual argument. Any of the usual computational intrinsic functions can be called elementally. For example, in

```
COS(X)
```

X may be scalar, in which case COS returns a scalar result, or X may be an array (any dimension), in which case COS returns an array-valued result conformable with X. If the seismic-like example above were modified to

```
exp(g) + cshift(u,1,2) - cshift(u,-1,2)
```

then each term in the expression becomes an intrinsic function call that returns a result conformable with g and u. The first term is an elemental call and the other two are transformational. All of the 108 Fortran 90 intrinsic functions may be called elementally except for the 42 listed above as transformational.

Note that whereas elemental function calls may be considered to be a number of independent scalar function calls, a transformational function is considered as in integral self-contained computation, delivering the result "all together, all at once".

Fortran 90 provides 21 intrinsic array functions, some (such as SIZE) that allow inquiries to be made about array properties and others that either construct arrays or extract information from arrays. These functions, all of which are transformational, are listed in Table 5

Users may define array-valued functions; all such functions are transformational. Function F18 in the previous section is an example of a user-defined array-valued function. In this case the shape of the array is determined (dynamically) from arguments, such as the shape properties of an array argument; this is probably the most useful form of array- valued functions. See also the examples in sections 4.5.

Note that function results are declared to be array- valued with ordinary declaration statements, as if the function name is an ordinary variable (as indeed it is within the body of the function). Though automatic arrays may be the most useful form for user-defined array-valued functions, any other form is also valid: explicit-shape array, allocatable array, pointer array. These are also declared and used in the procedure as if the function name were just another variable. The main additional requirement is that the array value must be fully defined before returning from an execution of the function. On the other end of things, the interfaces of array-valued functions must be explicit where such functions are used, so that the caller knows that it's dealing with a function that is array-valued.

A simple example of an array-valued function definition will complete this section. Suppose that the partial sums of a one-dimensional array of n elements are needed in an array expression-that is, the kth value needed is sum(P(1:k)). An array-valued function is ideal for delivering the requisite set of values (although in this simple case it might be almost as good to use the array constructor in the expression rather than the call to Partial_sums):

```
function Partial_sums(P)
 real   P(:)                     ! Assumed-shape dummy array
 real  Partial_sums(size(P))     ! The partial sums to be returned
 integer  k

 Partial_sums = (/(sum(P(1:k),k=1,size(P))/)

                                 ! This is functionally equivalent to
                                 ! do k=1,size(P)
                                 !  Partial_sums(k) = sum(P(1:k))
                                 ! end do
                                 ! but the do loop specifies a set of sequential
                                 ! computations rather than parallel computations

end function Partial_sums
```

The following more complicated examples of data parallel computations are also configured to deliver results as user-defined array-valued functions.

array intrinsic function	comment
ALL	true if all of the element values are true
ANY	true if any of the element values are true
ALLOCATED	check if array is allocated
COUNT	number of elements having the value true
CSHIFT	circularly shift an array along a dimension
EOSHIFT	end-off shift an array along a dimension
LBOUND	lower bound of an array
MAXLOC	location of maximum element in an array
MAXVAL	maximum element value in an array
MERGE	merge two arrays, under a mask
MINLOC	location of minimum element in an array
MINVAL	minimum element value in an array
PACK	gather an array into a vector, under a mask
PRODUCT	product of all the elements of an array
SHAPE	shape of an array
SIZE	total size of an array
SPREAD	spread an array by adding a dimension
SUM	sum of all of the elements of an array
TRANSPOSE	matrix transpose of a two-dimensional array
UBOUND	upper bound of an array
UNPACK	scatter a vector into an array, under a mask

Table 5:

	num. scalar operations	num. parallel operations	execution times
Simple_Gauss, sequential	$4N^3$	-	$4N^3$
Pivot_Gauss, sequential	$(N+7)N^3$	-	$(N+7)N^3$
Simple_Gauss, parallel	$5N^3$	10	10
Pivot_Gauss, parallel	$(N \ln N + 8)N^3$	$\ln N + 14$	$\ln N + 14$

Table 6:

4.5 Example: Gaussian Elimination

To illustrate realistic uses of data parallelism, this example presents two forms of the classic Gauss elimination algorithm for solving systems of linear equations. This particular example is chosen because of the near-universal familiarity with Gaussian elimination, so that maximum attention can be paid to the data parallel techniques with a minimum of distraction from becoming familiar with the problem. One form of the example, called Simple_Gauss, marches the pivot down the main diagonal of the matrix (called Grid or G in the code below); the other form, called Pivot_Gauss, implements the more complicated but more robust maximum pivot strategy for Gaussian elimination.

Both Simple_Gauss and Pivot_Gauss have two versions - a scalar sequential version and a data-parallel version. As presented, the data-parallel version would compile and run; the sequential version is commented out with "!!" at the beginning of these lines. If these comment characters are removed, and the lines ending with "!!!!" comments are commented out, the sequential version would compile and run.

The sequential versions contain no array operations (except for the initialization of G) and are characterized by the familiar scalar do-loops over the matrix. The data-parallel replacements for these loops immediately follow so that the sequential and parallel versions can be conveniently compared. Some liberty (though not much) has been taken with the presentation of these examples in an attempt to make these comparisons easy and most useful. Also in order to facilitate these comparisons, the entire matrix is reduced for each pivot, rather than just those columns needing reduction, so that each version of each algorithm involves about twice as many (scalar element) operations as are really necessary; with some loss of clarity the algorithms can easily be adjusted to limit the number of operations accordingly.

An analysis of these algorithms shows that the sequential version of Simple_Gauss has about $4N^3+7N^2+9N+1$ scalar operations, and the parallel version has about $5N^3+8N^2+4N+1$ scalar operations in 10 parallel operations. The sequential version of Pivot_Gauss has about $N^4+7N^3+4N^2+5N+1$ scalar operations, and the parallel version has about $(\ln N)N^4+8N^3+9N^2+5N+1$ scalar operations in $\ln N+14$ parallel operations. For reasonably large values of N, these results are summarized in Table 6. The last column of the table (idealistically) assumes that a data-parallel operation takes the same time as a scalar operation.

Thus in both cases the parallel version involves more scalar operations than does the sequential version, but the number of parallel operations is astoundingly low in comparison.

The effective cost of a parallel operation, in terms of a scalar operation, currently varies widely from system to system, but the trend appears to be (and certainly this is not inconsistent with theoretical possibility and the inexorable march of technology) asymtotic toward scalar operation costs. Viewed in these terms, the data-parallel version of Gaussian elimination is indeed attractive.

Finally, a word on the Fortran 90 intrinsic function SPREAD, used in the primary reduction operation in both Simple_Gauss and Pivot_Gauss. SPREAD replicates (spreads) a scalar into a one-dimensional array, or replicates an n-dimensional array into an n+1-dimensional array. The scalar-to-one- dimensional array form is that used here, and is just what the doctor ordered to convert the scalar operation G(i,j)=G(i,L)*G(L,j) into a whole-array operation on G. L is "constant" in this expression, in the loops over i and j, and thus must be "spread" in these places to fill out the array for the whole-array operation. Understanding this is key to, and the most difficult part of, assimilating a good feel for the data-parallel versions of this algorithm. SPREAD has three arguments: the first is the scalar or array to be spread, the second is the dimension over which the spreading occurs (and must be one for spreading a scalar), and the third is the number of replications (N or N+1 is these cases).

```
function  Simple_Gauss(Grid)       ! Gauss elimination - not max pivot.
  real ::  Grid(:,:)               ! The matrix to be reduced.
  real ::  Simple_Gauss(size(Grid,1))   ! Returns the solution vector.
  real ::  G(size(Grid,1),size(Grid,2))   ! G is a local work array.
  logical :: Not_pivot_row(size(Grid,1),size(Grid,2  )) ! Pivot row mask.
 if (size(Grid,2).ne.size(Grid,1)+1) stop \"bad Grid shape\"
 N = size(Grid,1)
 G = Grid      ! Work on G, not Grid.

 do L=1,N      ! G(L,L) is next pivot element.

  if (abs(G(L,L)).lt.1E-4)  stop \"zero encountered in pivot\"
!!  G_pivot = G(L,L)     !!
!!  do j=1,N+1     !! Normalize pivot row.
!!   G(L,j) = G(L,j)/G_pivot    !!
!!  end do     !!
  G(L,:) = G(L,:)/G(L,L)     !!!!   Data-parallel version.

!!  do i=1,N     !!
!!   do j=1,N+1    !!
!! if ( i.ne.L.and.j.ne.L )  then      !! Then reduce matrix with
!! G(i,j) = G(i,j)-G(i,L)*G(L,j) !     !   this pivot element.
!! end if  !!
!!   end do    !!
!!  end do     !!
  Not_pivot_row = .true.; Not_pivot_row(L,:) = .false.
```

```
                                                       !!!!   Data-parallel version.
  where ( Not_pivot_row )   &                          !!!!   Data-parallel version.
   G = G-G(:,spread(L,1,N+1))*G(spread(L,1,N),:)      !!!!   Data-parallel version.

 end do                            ! Repeat for all pivots.

!! do i=1,N                        !! Finally, extract the
!!  Simple_Gauss(i) = G(i,N+1)     !!   solution vector from
!! end do      !!   the last column of G.
 Simple_Gauss = G(:,N+1)           !!!!   Data-parallel version.

end function Simple_Gauss

function  Pivot_Gauss(Grid)       ! Gauss elimination, max pivot.
  real ::  Grid(:,:)   ! The matrix to be reduced.
  real ::  Pivot_Gauss(size(Grid,1)  ) ! Returns the solution vector.
  real ::  G(size(Grid,1),size(Grid,2))   ! G is a local work array.
  integer ::  P(size(Grid,1),2)   ! P is array of pivots.
  logical :: Not_pivot_row(size(Grid,1),size(Grid,2))
                                  ! Mask current pivot row only.
  logical :: Not_pivot_rows_or_cols(size(Grid,1),size(Grid,1))
        ! Mask out all
        ! previous pivots
        ! rows and columns.
 if (size(Grid,2).ne.size(Grid,1)+1) stop \"bad Grid shape\"
 N = size(Grid,1)
 G = Grid      ! Work on G, not Grid.

 do L=1,N      ! L is next pivot number.

!!  G_pivot = -1      !!
!!  do i=1,N      !! First, find next pivot.
!!   Inner_pivot_search:    &  !!
!!   do j=1,N     !!
!!    do k=1,L-1    !!
!!     if (i.eq.P(k,1).or.j.eq.P(k,2))   cycle Inner_pivot_search
!!        !! Skip this element; it's in a
!!    end do    !!   previous pivot row or col.
!!    if (abs(G(i,j)).gt.G_pivot) then      !!
!!     G_pivot = abs(G(i,j))   !!
!!     P(L,1) = i   !!
!!     P(L,2) = j   !!
```

```
!!    end if    !!
!!   end do    Inner_pivot_search    !!
!!  end do      !!
  Not_pivot_rows_or_cols = .true.      !!!!   Data-parallel version.
  Not_pivot_rows_or_cols(P(1:L-1,1),:) = .false.   !!!!   Data-parallel version.
  Not_pivot_rows_or_cols(:,P(1:L-1,2)) = .false.   !!!!   Data-parallel version.
 P(L,:) = maxloc(abs(G(:,1:N)),mask=Not_pivot_rows_or_cols)
                                                   !!!!   Data-parallel version.
  if (abs(G(P(L,1),P(L,2))).lt.1E-4)  stop \"ill-conditioned matrix\"

!!  G_pivot = G(P(L,1),P(L,2))      !!
!!  do j=1,N+1      !! Then, normalize pivot row,
!!   G(P(L,1),j) = G(P(L,1),j)/G_pivot          !!   establish pivot row flag.
!!  end do    !!
  G(P(L,1),:) = G(P(L,1),:)/G(P(L,1),P(L,2))    !!!!   Data-parallel version.


!!  do i=1,N      !! Then reduce matrix with
!!   do j=1,N  +1   !!   this pivot element.
!!    if ( i.ne.P(L,1).and.j.ne.P(L,2) )  then     !!
!! G(i,j) = G(i,j)-G(i,P(L,2))*G(P(L,1),j)       !!
!!    end if    !!
!!   end do     !!
!!  end do      !!
  Not_pivot_row = .true.; Not_pivot_row(P(L,1),:) = .false.
                                          !!!! Data-parallel ver
 where ( Not_pivot_row ) &                !!!!   Data-parallel version.
 G = G-G(:,spread(P(L,2),1,N+1))*G(spread(P(L,1),1,N),:)
                                          !!!!   Data-parallel version.

 end do       ! Repeat for all pivots.

!! do i=1,N                                !! Finally, unscramble the
!! Pivot_Gauss(P(i,2)) = G(P(i,1),N+1)      !!   solution vector from
!! end do                                  !!   the last column of G.
 Pivot_Gauss(P(:,2)) = G(P(:,1),N+1)       !!!!   Data-parallel version.

end function Pivot_Gauss
```

References

[1] *Fortran 90 Handbook*, Adams, Brainerd, Martin, Smith, Wagener, McGraw-Hill, 1992.

[2] Programming Language Fortran, ANSI standard X3.198-1992

[3] DRAFT - Process Parallelism Standard, ANSI committee X3H5