Cray Fortran Pointers vs. Fortran 90 Pointers and Porting from the CRAY C90 to the SGI Origin2000

Mark R. Fahey*
Dan Nagle
U.S. Army Corps of Engineers Waterways Experiment Station
Major Shared Resource Center
Vicksburg, MS 39180

April 19, 1999

1 Fortran Pointers

Pointers can be used to access user-managed storage by dynamically associating variables and arrays to particular locations in a block of storage. Fortran 90 introduced to standard Fortran the concept of the pointer. Cray Fortran has had pointers for a long time. However, Fortran 90 pointers and Cray pointers are very different.

Cray's (nonstandard) *pointer* is a variable whose value is the address of another entity, which is called a *pointee*. The Cray Fortran pointer type statement declares both the pointer and its pointee, as follows:

```
POINTER (ptr, ptee)
```

The pointee may be of any type, but the pointer is always typed a `POINTER`. A Cray pointer is a "true" pointer; that is, it is a variable that contains the address of the pointee. The size of the area pointed to is determined by the size of the pointee. The nonstandard intrinsic function `LOC` generates the address of a variable to be used with a Cray pointer, as follows:

```
ptr = LOC( variable )
```

The Fortran 90 standard does not provide for the Cray Fortran pointer data-type. Fortran 90 has the *pointer* and *target* attributes, for example

```
integer, POINTER :: iptr
integer, TARGET :: itrgt
```

*Computational Migration Group, Nichols Research, mfahey@nrcmail.wes.hpc.mil

The pointer and target must be the same type, kind and rank. A Fortran 90 pointer is an alias for the target; that is, it is a second (or subsequent) name that refers to the variable. A Fortran 90 pointer may refer to an array section, which need not be contiguous storage.

With pointers (and allocatable arrays) standardized in Fortran 90, the Fortran `EQUIVALENCE` statement has become a deprecated feature. The `EQUIVALENCE` statement specifies that a given storage area may be shared by two or more objects.

Cray and Fortran 90 pointers can appear in the same program, and point to the same location in memory. For example, the program

```
real :: YP
pointer ( P, YP )
real, target :: A
real, pointer :: Aptr
data A/1.0/
Aptr => A
P = loc(A)
print*,Aptr,' ',A,' ',YP
```

has the output

```
 1.   1.   1.
```

However, a Cray pointer cannot have the target attribute and a Fortran 90 pointer cannot have the Cray pointer type. Therefore, a Cray pointer cannot point to a Fortran 90 pointer and vice versa. Furthermore, a Fortran 90 target cannot be a Cray pointee and a Cray pointee cannot have the Fortran 90 target attribute. For a more detailed description of Cray pointers (and pointees) and their restrictions, see [1].

The remainder of this note is organized as follows. Section 2 describes issues relating to porting Cray pointers from the CRAY C90 to the SGI Origin2000. Section 3 finishes this note with an example demonstrating use of pointers.

2 Porting Cray Pointers from the CRAY C90 to the SGI Origin2000

The `f77` and `f90` compilers on the SGI Origin2000[1] support the Cray pointer type and the associated intrinsic function `LOC`. Thus, a code that has Cray pointers will compile and run on the SGI Origin2000 in its native 32-bit binary application interface.

If a 64-bit compilation is desired, the code should be compiled with the `-64 -i8 -r8 -d16`[2] options when using `f90` or `f77`. If the `-i8` option cannot be used[3] and if a Cray pointer is passed as an argument to a subprogram, then the user must be sure to specify that the argument in the receiving subprogram is of Cray pointer type. If this is not specified,

[1]SGI Origin2000 at CEWES MSRC.

[2]This is the `-default64` option for the `f90` compiler.

[3]A 64-bit precompiled library subprogram may expect integers of `KIND=4` (in a common block for instance), but `-i8` will force the calling subprograms integers to be of `KIND=8`.

then the program will crash even though it will compile without warning messages. This is because upon creation (with `-64`), a Cray pointer is “like” an `INTEGER(KIND=8)` variable, but a receiving subprogram will assume it is an `INTEGER(KIND=4)`. The following program demonstrates the problem mentioned above and shows how to solve it.

```
      integer :: I
      real :: YP
      pointer ( P, YP )
      real :: A
      data A/1.0/
      P = loc(A)
      I = P
      print*,'The variables P =',P,' and I =',I,' in main'
      call subprog1( P )
      call subprog2( P, I )
!
      subroutine subprog1( P )
      print*,'The variable P =',P,' in subprog1'
      end
!
      subroutine subprog2( P, I )
      pointer ( P, ydummy )
      integer I
      print*,'The variables P =',P,' and I =',I,' in subprog2'
      end
```

This code produces the following output when compiled with the `-64` option.

```
  The variables P = 268509264  and I = 268509264  in main
  The variable P = 0  in subprog1
  The variables P = 268509264  and I = 268509264  in subprog2
```

This shows that one must be sure to type subprogram arguments or pass true integers as arguments if the `-i8` option is not used. The safest and clearest choice is to use explicit typing for subprogram arguments.

3 A Fast Fourier Transform Example

The rest of this note discusses Fortran array storage order, followed by a discussion to illustrate the different pointers via the example of the two-dimensional Fast Fourier Transform (FFT). We will view the FFT without pointers, with Cray pointers, and with Fortran 90 pointers. The last example is done in C to clarify the issues raised here for C programmers. (The interested reader is referred to, e.g., [2], for more information on FFT’s.)

3.1 Fortran Storage Order

In the case of a one-dimensional array, storage order is intuitive and in fact every language supporting arrays handles the situation similarly. If `A` is an array, then `A(I+1)` occupies the next storage location after `A(I)`. A complication arises when discussing multidimensional arrays. In the case of the two-dimensional array, one has a choice of whether to count down the columns or count across the rows. Fortran counts down the columns, thus Fortran is a *column-major* programming language (most other programming languages, e.g., C, are *row-major* languages). Suppose `A` is declared as

```
   integer, parameter :: N = 1000
   complex, dimension( N, N ) :: A
```

Then Fortran storage order is as follows:

```
  A(1,1),A(2,1),...,A(N,1),A(1,2),A(2,2),...,A(N,2),...,A(1,N),...,A(N,N)
```

3.2 The Basic Example

The rest of this paper describes, using an example of the two-dimensional FFT, how to program the solution using no pointers, Cray pointers, and Fortran 90 pointers. The examples assume that the "dirty work" is done in a subroutine `FOUR1D()`, which will compute a one-dimensional FFT. `FOUR1D()` has the following interface

```
interface
   subroutine four1d( CDATA, N, INC, ISIGN )
   complex, intent( in out ), dimension( N*N ) :: CDATA
   integer, intent( in ) :: N, INC, ISIGN
   end subroutine
end interface
```

and computes an in-place one-dimensional FFT of length `N` with `INC` words between elements. `ISIGN` is the forward/reverse ± 1 switch.

Note that, in `FOUR1D()`, the array is declared to be a one-dimensional array. The routine `FOUR1D()` will address this one-dimensional array using the size (`N`) and increment (`INC`) provided. As discussed above, addressing columns is done by setting `INC = 1`, while addressing rows is done by setting `INC = N`.

Two things should be noted about the two-dimensional FFT. First, the two-dimensional FFT will transform each column and each row of the array. Second, the transform of each column involves consecutive storage locations, while the transform of each row involves storage locations separated by the size of a column. Thus, the row transform step must include a gather and scatter of the data in the array. These gather/scatter steps may be implicit in the call to `FOUR1D()`, or they may be explicitly done by the program. Note also, that a row-major language has the same problem in computing the two-dimensional FFT, except the step which requires the gather/scatter is the column transform rather than the row transform.

3.3 Two-Dimensional FFT without Pointers

Here is a straight-forward implementation of the two-dimensional FFT that does not use pointers.

```
   integer, parameter :: N = 1024
   complex, dimension( N, N ) :: A

! first, transform the columns
! note that INC = 1, because column elements are consecutive storage locations

   do i = 1, N
      call four1d( A( 1, i ), N, 1, 1 )
   enddo

! now, transform the rows
! note that INC = N, because row elements are N units apart

   do i = 1, N
      call four1d( A( i, 1 ), N, N, 1 )
   enddo
```

The gather/scatter step is done implicitly by passing the value `N` to the `FOUR1D()` argument `INC` in the row transform step. The value of `INC` was 1 in the column transform. Note that the array indices change on each call to `FOUR1D()`.

3.4 Two-Dimensional FFT with Cray Pointers

Now, let us re-do the previous example using Cray pointers. Here, we set the pointer to the location of the row or column to be transformed, and pass the pointee array to `FOUR1D()`.

```
   integer, parameter :: N = 1024
   complex, dimension( N, N ) :: A

! declare the Cray pointer
! note that *NO* storage is allocated for PA (the pointee array)
! PA won't refer to memory until P is set to the address of some
! memory by the LOC() function

   complex, dimension( N*N ) :: PA
   pointer( P, PA )

! first, transform the columns
! note that inc = 1, because column elements are consecutive storage locations

   do i = 1, N
```

```
      P = loc( A( 1, i ) )
      call four1d( PA, N, 1, 1 )
   enddo

! now, transform the rows
! note that inc = n, because row elements are n units apart

   do i = 1, N
      P = loc( A( i, 1 ) )
      call four1d( PA, N, N, 1 )
   enddo
```

A number of points should be made clear. First, the size of the pointee array `PA` really doesn't matter. The pointee `PA` just needs to be declared a one-dimensional array. The data addressed are controlled by `N` and `INC`. Second, the setting of `P` to a location within `A` by the `P = LOC( A( ...)  )` statement involves no data movement. It's simply a run-time equivalence of `A` and `PA`. Third, where the example in Section 3.3 passed the array A with indices (and in fact the indices vary with each call to `FOUR1D()`), the Cray pointers example passed `PA` without indices. This is because the location of `PA` is changed by resetting `P` at each iteration (i.e., the run-time equivalencing changes as the program executes).

3.5 Two-Dimensional FFT with Fortran 90 Pointers

Now, let us re-do the previous example using Fortran pointers. Here, the pointee array must be declared with the TARGET attribute. This enables the compiler to better optimize the routine because the compiler assumes that *only* variables declared with the POINTER or TARGET attribute may be aliased.

```
   integer, parameter :: N = 1024
   complex, dimension( N, N ), target :: A

! declare the Fortran pointer
! note that *NO* storage is allocated for PA (the pointer array)
! PA won't refer to memory until it is assigned ( => ) to a target

   complex, dimension( N ), pointer :: PA

! first, transform the columns
! PA points to a different column on each iteration

   do i = 1, N
      PA => A( :, i )
      call four1d( PA, N, 1, 1 )
   enddo

! now, transform the rows
! PA points to a different row on each iteration
! INC is set to 1 rather than N because the gather/scatter
! is done explicitly by the pointer assignment

   do i = 1, N
      PA => A( i, : )
      call four1d( PA, N, 1, 1 )
   enddo
```

The biggest difference between the Fortran 90 pointer example and the others is that the gather/scatter is done explicitly by the pointer assignment, not implicitly by changing `INC`. If the programmer wants to use the array features of Fortran 90, an interface block should be provided so the compiler knows to pass an array descriptor rather than an array address. If the programmer does use an interface block for `FOUR1D()`, the chances of a copy being done are lower, because the compiler can tell how the `FOUR1D()` subroutine is declared. The interface also enables usage checking, so the compiler may catch additional errors at compile time.

3.6 Two-Dimensional FFT with C Pointers

Here we have the FFT example in C. Note that this is closest to the Cray Fortran example. We define a type `complex` and the function prototype analogous to `FOUR1D()`

```
 void four1d( complex* a, int n, int inc, int sign)
```

The differences with the Cray Fortran example are: C is row-major where Fortran is column-major, a pointer in C need not be declared with its pointee (a fact that causes some difficulties with optimization), the `DO`-loops are replaced by for-loops, the arrays are constrained to have lower limits in each dimension of zero, and type `complex` must be defined by the programmer.

```
#define N 1024

typedef struct complex {
   float r;
   float i;
} complex;

complex a[ N ][ N ];

/* define the pointer to a */
complex* pa;

/* first, transform the columns
  note that inc = N, because column elements are N storage locations apart */
   for ( int i = 0; i < N; i++ )
   {
      pa = &a[ 0 ][ i ];
      four1d( pa, N, N, 1 );
   }
/* now, transform the rows
  note that inc = 1, because row elements are consecutive storage units */
   for ( i = 0; i < N; i++ )
   {
      pa = &a[ i ][ 0 ];
      four1d( pa, N, 1, 1 );
   }
```

In this example, the pointer `pa` does not refer to memory until it is assigned the address of an array element via the `pa = &a[ ... ]` statement. This example could also have been written so that the array `a` was declared as a pointer to complex. In fact, since all this example shows is `pa` being passed to a function, `a[ ... ]` could have been passed directly.

4 Conclusion

Cray Fortran pointers and Fortran 90 pointers are two entirely different ways of dynamically associating variables to particular blocks of storage. Codes written for Cray machines like the CRAY C90 before the Fortran 90 standard may contain Cray pointers. When migrating these codes to the SGI Origin2000, one should be aware that the Fortran compilers on the SGI Origin2000 support Cray pointers and will not warn the user of their presence. If a Cray pointer is passed as an argument and the receiving subroutine does not specify the variable as a pointer, then the code may crash. Such bugs are difficult to find and can be easily averted by either explicitly specifying the pointer type in the subprogram, using the `-i8 option`, passing true integers instead, or, as a last resort, rewriting the code (possibly using Fortran 90 pointers.)

References

[1] *Fortran Language Reference Manual, Volume 1.* Silicon Graphics, Document Number 007-3692-002 (`http://techpubs.sgi.com/library`).

[2] *Numerical Recipes in Fortran: The Art of Scientific Computing*, Second edition. William H. Press, et al, Cambridge University Press, 1992.