ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-187

ADIFOR Working Note No. 11

ADIFOR Strategies Related to `POINTER` Usage in MM5

Christian Bischof **Peyvand Khademi** **Timothy Knauff**

Mathematics and Computer Science Division

Technical Memorandum No. 187

March 1994

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Aerospace Agency under Purchase Order L25935D.

Contents

ADIFOR Working Note No. 11

ADIFOR Strategies Related to `POINTER` Usage in MM5

Christian Bischof **Peyvand Khademi** **Timothy Knauff**

Abstract
`POINTER`s are nonstandard Fortran statements which cannot be processed by ADIFOR. We are interested in generating derivative code for MM5, a mesoscale model code which uses `POINTER`s extensively and in a particular structured manner. We briefly report on `POINTER`s and their role in MM5 and, for their particular usage in MM5, describe the three-step code transformation scheme consisting of pre-ADIFOR, ADIFOR, and post-ADIFOR transformations that result in the generation of correct derivative code for MM5.

Introduction

In our attempt to generate derivative code for MM5 (the fifth-generation Penn State/NCAR Mesoscale Model) [5], with ADIFOR (Automatic DIfferentiation of FORtran) [2], [3], we encountered the nonstandard Fortran statement `POINTER`. The purpose of this note is to

1. document our understanding of the role of `POINTER`s as they are used in MM5, and
2. describe our workaround strategy for "masking" `POINTER`s for processing of the code with ADIFOR, and then "unmasking" them.

`POINTER`s in Fortran

`POINTER` is a Cray extension to the Fortran 77 standard which has been standardized in Fortran 90. `POINTER`s are admissible parts of many Fortran compilers, including the Cray CFT77 [4], RS/6000 xlf [1], and Sun f77 [7] compilers, and to the best of our knowledge have identical syntax and semantics in these compilers. The following information about `POINTER`s is worth noting:

- The syntax is as follows:

  `POINTER` (*pointer*,*pointee*) [,(*pointer*,*pointee*)]...

- The `POINTER` statement allows one to specify that the value of the variable *pointer* should be used as the base address for any reference to *pointee*.

```
      REAL A(10,10), CURCOL(10), NORMS(10)
      POINTER (PTR,CURCOL)

C     2 Alternative Declarations for the pointer and pointee:
C     REAL CURCOL                      |     REAL CURCOL
C     POINTER (PTR,CURCOL(10))         |     POINTER (PTR,CURCOL)
C                                      |     DIMENSION CURCOL(10)
      DO I=1,10
        PTR = LOC(A(1,I))
        NORMS(I) = SNORM2(10,CURCOL)
      ENDDO

      REAL FUNCTION SNORM2(N,X)
      REAL X(N)
      SNORM2 = 0.0
      DO I=1,N
        SNORM2 = SNORM2 + X(I)**2
      ENDDO
      SNORM2 = SQRT(SNORM2)
      RETURN
```

Figure 1: A simple example of the usage of the POINTER statement

- A pointer can appear in a `COMMON` statement but cannot appear in a type statement.[1] A pointer occupies storage adequate for an address. The compilers mentioned above assign storage equivalent to an `INTEGER` to a `POINTER` variable.

- A pointee cannot appear in a `COMMON` statement, but can be declared as a variable and can be dimensioned. The compiler does not allocate storage for a pointee, even if it appears in a type or dimension statement.

- The `LOC` function returns the address of a variable and can be used to define a pointer (example: `PTR = LOC(ARR(I,J))`).

Figure 1 shows a simple code making use of the `POINTER` statement. We compute the Euclidean norm of the columns of a matrix by using a `POINTER` to point to a column at a time.

[1]The Cray compiler is less restrictive, stating that a pointer cannot appear in a *preceding* type statement.

Why and How POINTERs Are Used in MM5

In studying the MM5 code and documentation [6] we have learned that MM5 uses POINTERs to associate model parameters with values for a given nest. There appear to be two principal reasons for using such a scheme:

1. to allow for nest shifting without the need for passing long parameter lists, and
2. to simplify dumping a state and subsequently restarting from the same.

To better explain the above reasons, we describe some aspects of the code:

- Two COMMONs play key roles in the overall MM5 pointer scheme:

```
REAL ALLARR
COMMON /HUGE/ ALLARR(IHUGE,MAXNES)

INTEGER IAXALL
COMMON /ADDRO/ IAXALL(NUMVAR,MAXNES)
```

- ALLARR is the array where the values for all model variables for all nests reside. IHUGE is the sum of the product of the dimensions of all pointees (i.e., the maximum number of all variables describing the state of a particular nest), and MAXNES is the maximum number of nests during a simulation. Each column of ALLARR contains the full set of model parameters for a given nest.

- IAXALL is an array containing address values for all the POINTERs in MM5. NUMVAR (approximately 300) is the number of pointers. For a given nest, IAXALL maps each pointer to a location in ALLARR corresponding to the start of the image of the pointee. Each column of IAXALL is the full mapping for a given nest.

- The mapping for *all* nests is created once at the beginning of a new run or a restart by calling the addressing subroutine, ADDALL, and subsequently never changed. ALLARR addresses are assigned to corresponding IAXALL entries via the LOC intrinsic. Figure 2 contains code fragments from SUBROUTINE ADDALL, showing instances of this mapping. (Note in this example that each two successive IAXALL entries will be different in value by an amount equal to MIX*MJX*MKX, which is equal to the "extent" of a particular pointee array. We shall come back to this point later.)

- A second addressing subroutine, ADDRX1C (Figure 3), is called once at the beginning and subsequently for every nest shift, to effectively assign the appropriate column of IAXALL to the set of actual pointers. This process involves

```
      SUBROUTINE ADDALL

      IX3D = MIX*MJX*MKX
      ...
      DO 100 K = 1,MAXNES
          IAXALL(1,K) = LOC(ALLARR(1,K))
          NCOUNT = 1
          NCOU = 1
          DO 15 N = 2,NVARX + 1
              NCOUNT = NCOUNT + 1
              IAXALL(NCOUNT,K) = LOC(ALLARR(1+ (N-1)*IX3D,K))
 15       CONTINUE
          NCOU = NCOU + (NVARX)*IX3D
          ...
100   CONTINUE
```

Figure 2: Excerpts from the file "addall.f"

passing the appropriate column of `IAXALL` to the dummy argument `IARR` of `ADDRX1C`, then `EQUIVALENCE`ing the first pointer listed in each `COMMON` to the first entry in a local array (e.g., `IDUDU(1)`) (thus also `EQUIVALENCE`ing subsequent entries in the `COMMON` to successive entries in the local array), and finally copying the values of `IARR` into `IDUDU(1)`, `IDUDU(2)`, etc. As a result of the `EQUIVALENCE` statement, `IAUA`, `IAUB`, ..., and `IAPA`, `IAPB`, ... are assigned (address) values stored in a particular column of `IAXALL`.

- Figure 4 depicts the addressing scheme for a given nest value, `NUMNES`. We have shown only one column (drawn as a row) of `ALLARR` and `IAXALL`, and only two of the many `COMMON`s in MM5. The arrows from entries of `IAXALL` to starting addresses of blocks of `ALLARR` depict the pointer assignments, which are made once by calling `ADDALL` and are never changed. The links between the pointers in the `COMMON`s and entries of `IAXALL` (drawn with three horizontal bars in the middle) depict the assignment of pointer values to a given nest. Each time there is a nest shift, these links are redrawn to reassign the pointers to the appropriate column of `IAXALL`.

- `ADDRX1N` is called to concurrently define pointers for a coarser and a finer nest. We note that the sets of pointers accessed in the two nests will always be disjoint.

```
      SUBROUTINE ADDRX1C(IARR)
      INTEGER IARR(NUMVAR)

      INTEGER IAUA,IAUB,...
      COMMON /ADDR1/IAUA,IAUB,...
      INTEGER IAPA,IAPB,IAZ0,IAH0,...
      COMMON /ADDR2/IAPA,IAPB,IAZ0,IAH0,...

      INTEGER IDUDU1(NVARX+NVARMX+4),IDUDU2(NVARSX),...

      EQUIVALENCE (IDUDU1(1),IAUA)
      EQUIVALENCE (IDUDU2(1),IAPA)
      ...
      DO 10 N = 1,NVARX + NVARMX + 4
          NN = N
          IDUDU1(N) = IARR(NN)
   10 CONTINUE
      NM = NN
      DO 20 N = 1,NVARSX
          NN = NM + N
          IDUDU2(N) = IARR(NN)
   20 CONTINUE
      ...
```

```
      CALL ADDRX1C(IAXALL(1,NUMNES))
```

Figure 3: Excerpts from the file "addrx1c.f" and an example of a call to ADDRX1C

- A pointee array (e.g., `UA` in Figure 6), wherever it appears in the computation, will refer to the values of some quantity some model values for some nest, but the nest information will not appear explicitly in its dimensions.

Thus, in using `POINTER`s, to shift from one nest to another, one simply reassigns all pointers by calling `ADDRX1C`. Shifting nests has been achieved without the need for potentially very large subroutine interfaces. Also, dumping a state now merely requires saving `ALLARR`.

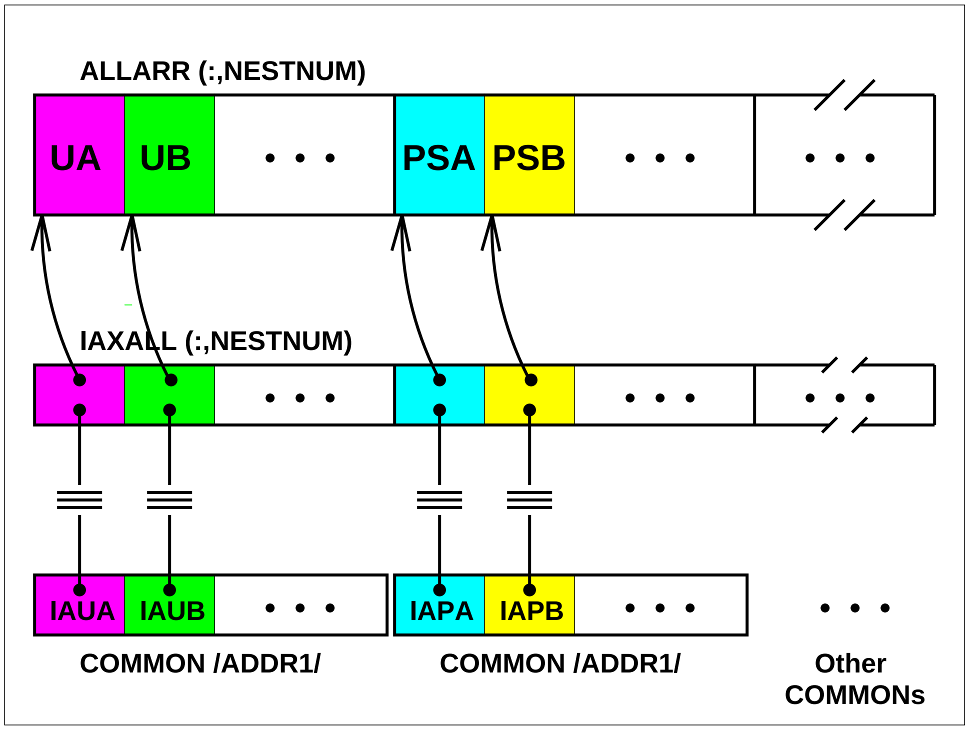

Figure 4: Schematic illustrating the MM5 POINTER addressing scheme

`POINTER`s and ADIFOR

Our first challenge in the development of a sensitivity-augmented version of MM5 is that the `POINTER` extension is not supported by ADIFOR. This necessitates modifying the original code in such a way as to make it admissible for ADIFOR processing, while maintaining the dependence profile of the code (which is used by ADIFOR in constructing the derivative code) as well as the intended semantics of the program. While one could avoid the use of `POINTER`s, the effort seems prohibitive given the pervasive and structured use of `POINTER`s in MM5.

We have devised a method for solving this problem by systematically "masking" `POINTER`s prior to processing the code with ADIFOR, and reintroducing in the ADIFOR-generated code the same `POINTER` statements and, additionally, the corresponding `POINTER` statements for the derivative objects. To this end, we have developed a set of Perl scripts tailored to the particular structured use of `POINTER`s in MM5.

A critical aspect of the usage of `POINTER`s in MM5 is that there is no "aliasing"; that is, every address in `ALLARR` is pointed to by exactly one pointer. As a result, a dependence analysis at one particular nest level will accurately capture the dependence profile of the code, since, outside of pointer shifting in `ADDRX1C`, there is no "hidden" dependence between nest levels. This fact is significant from the point of

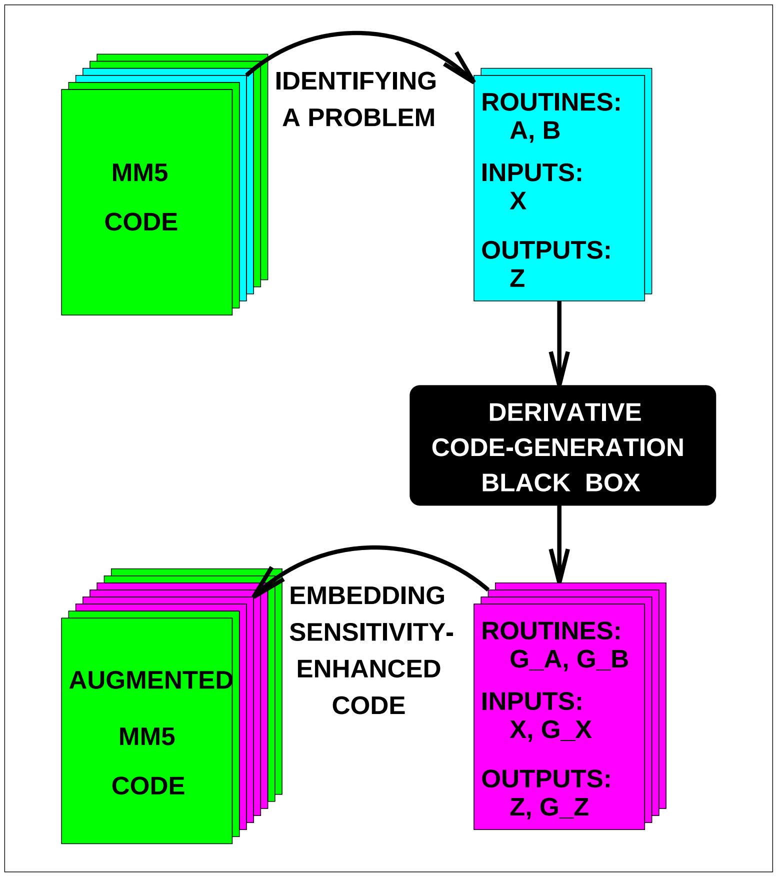

Figure 5: Schematic illustrating the augmentation of MM5 with sensitivity code

view of dependence analysis in ADIFOR, since there is never a danger of an entry in `ALLARR` having multiple dependency profiles.

Stepping through the Transformation of `HIRPBL`

ADIFOR operates on a subroutine or a suite of subroutines. Our initial goal is to augment some subset of MM5 with sensitivity code.

Figure 5 is a schematic of this process. Subroutine A is identified as the top-level subroutine, and B represents other subroutines in the calling sequence of A. X and Z have been nominated as the independent and dependent variables, respectively. The "Derivative Code-Generation Black Box," which we will describe in the following paragraphs, includes ADIFOR as well as the pre- and post-ADIFOR transformations. The black box generates the derivative code, contained in the subroutines G_A and G_B. Finally, a sensitivity driver code will embed the sensitivity-enhanced code in the original calling context by properly initializing the seed matrix, G_X.

```
ccc   The POINTERs originally listed in a COMMON statement:
      COMMON /ADDR1/ IAUA, IAUB, ...

ccc   The POINTEEs are originally declared as REAL:
      REAL UA, UB, ...

ccc   The non-standard POINTER statement:
      POINTER (IAUA, UA(MIX,MJX,MKX)), (IAUB, UB(MIX,MJX,MKX)), ...
```

```
      COMMON /ADDR2/ IAPA, IAPB, IAZO, IAHO, ...
      REAL PSA, PSB, ZOL, HOL, ...
      POINTER (IAPA,   PSA(MIX,MJX)), (IAPB,   PSB(MIX,MJX)),
     +        (IAZO,   ZOL(MIX,MJX)), (IAHO,   HOL(MIX,MJX)), ...
```

Figure 6: Excerpts from the code for HIRPBL

We have chosen the high-resolution planetary boundary layer (HIRPBL) module, consisting of three subroutines (`HIRPBL`, `SFCRAD`, and `SLAB`) as the first MM5 submodel to be augmented with sensitivity computations. This module was chosen because it is the interface along which one would incorporate the BATS code into MM5, by (in essence) interchanging the surface interaction model in BATS for `SLAB`. The sensitivities of interest, for example, derivatives of TTNP (tendencies of temperature) with respect to DZQ (layer thickness), could provide insight into how MM5 is impacted by different approaches to the modeling of this phenomenon.

We will now step through Figures 6–10 which exemplify the transformations yielding the derivative code. First a few general remarks about the code segments in these figures:

- The code segments were extracted verbatim from actual files and massaged slightly and only for readability purposes.
- Wherever "..." appears, it signifies that in the actual code more variables followed which we have omitted since they behave similarly to the variables we've shown.
- There are two kinds of comment lines in the code: those starting with "C" are part of the transformations themselves and were inserted by the Perl tools; those starting with "ccc" were subsequently inserted manually for readability.

- Each figure is divided into two sections. The upper section is an example of a `COMMON` that ADIFOR will not nominate as active, and the lower section is an example of an active `COMMON`. We will follow the progression of these two `COMMON`s through the transformations.

- We make use of the functions *unify* and *flow* of the *fortran-manipulate.pl* package in /home/derivs/share/lib/perl, which, respectively, construct a long line from a set of Fortran continuation lines and split up a long line into Fortran continuation lines.

- The main Perl scripts, *de_pointer.pl*, *re_pointer.pl*, and *gradient_ptr.pl*, along with a few other subsidiary ones, reside in /home/derivs/share/MM5. We will not go into the internal details of these scripts here; rather, we'll discuss the main functionalities and refer the reader to the *README* file in the same directory for the details.

Figure 6, an excerpt from the code for HIRPBL, is the starting point of our transformation scheme. Each `POINTER` statement is coupled with the appearance of its pointer in a `COMMON` statement and the declaration of its pointee in a `REAL` statement. We note that each pointee appears with a dimension in the `POINTER` statement, which specifies its "extent" and is used to infer the proper offset from the base address.

By masking the `POINTER` statement, *de_pointer.pl* transforms the code for HIRPBL to the code fragment we denote as *de_pointered_hirpbl.f* (Figure 7). Here "masking" means that though the `POINTER` statement no longer appears as code, the pointer and pointee appear in a context visible to ADIFOR, namely, in `INTEGER` and `COMMON` statements, respectively. We do echo the `POINTER` statement in *de_pointered_hirpbl.f* as a comment to facilitate its unmasking later.

The key idea is this: *de_pointered_hirpbl.f* will never get executed; therefore, at this stage of the transformation we are interested not in code that will run correctly but rather in code that will meet the ADIFOR restriction requirements and will cause the correct dependency propagation in ADIFOR.

The `INTEGER` declaration of the pointer will simply ensure that we do not rely upon default implicit typing of Fortran and that the former `POINTER` variables are properly typed. This declaration is in fact extraneous, but it does serve a documentation purpose without complicating the transformation or ADIFOR steps. The `COMMON` statements inserted by *de_pointer.pl* (e.g., `COMMON /UA_CMN/`) do play a necessary role (one that could also be played by `DIMENSION` statements). Since the pointee arrays appear dimensionless in the `REAL` declaration, without the dimension information in the `COMMON`, ADIFOR would not know how to dimension the `G_` variables corresponding to those variables that it determines to be active.

A somewhat unwanted side effect of the `COMMON` declaration is that all variables in the `COMMON` will be activated if ADIFOR recognizes a subset of them to be active.

```
ccc   We leave the COMMON statement listing POINTERs as is:
      COMMON /ADDR1/ IAUA, IAUB, ...

ccc   The POINTERs are now declared as INTEGERs:
      INTEGER IAUA, IAUB, ...

ccc   We leave the REAL declaration of the POINTEEs as is:
      REAL UA, UB, ...

ccc   The POINTER statement is commented out:
C     POINTER (IAUA, UA(MIX,MJX,MKX)), (IAUB, UB(MIX,MJX,MKX)), ...

ccc   We insert a COMMON statement listing the POINTEEs. The
ccc   name for the COMMON is constructed using the name of the
ccc   first POINTEE in the POINTER list:
      COMMON /UA_CMN/ UA(MIX,MJX,MKX), UB(MIX,MJX,MKX), ...
```

```
      COMMON /ADDR2/ IAPA, IAPB, IAZO, IAHO, ...
      INTEGER IAPA, IAPB, IAZO, IAHO, ...
      REAL PSA, PSB, ZOL, HOL, ...
C     POINTER (IAPA,   PSA(MIX,MJX)), (IAPB,   PSB(MIX,MJX)),
C    +          (IAZO,   ZOL(MIX,MJX)), (IAHO,   HOL(MIX,MJX))
      COMMON /PSA_CMN/ PSA(MIX, MJX), PSB(MIX, MJX), ZOL(MIX, MJX),
     +                  HOL(MIX, MJX), ...
```

Figure 7: Excerpts from the file "depointered_hirpbl.f"

Since the corresponding gradient arrays will also be pointee arrays, however, no storage cost will be associated with this side effect.

ADIFORed_hirpbl.f is the result of processing *de_pointered_hirpbl.f* through ADIFOR. The only changes caused by this step are the appearance, in the lower section of Figure 8, of `REAL` and `COMMON` declarations of the gradient variables created by ADIFOR. It turns out that none of the variables in `COMMON /ADDR1/` are active; hence, ADIFOR does not create gradient objects corresponding to any of these variables. On the other hand, in `COMMON /ADDR2/`, variable `ZOL` is active, and hence, ADIFOR creates gradient objects corresponding to all of these variables.

re_pointer.pl transforms *ADIFORed_hirpbl.f* to *re_pointered_ADIFORed_hirpbl.f* (Figure 9) by unmasking the `POINTER` statement. In the upper section of Figure 9,

```
      COMMON /ADDR1/ IAUA, IAUB, ...
      INTEGER IAUA, IAUB, ...
      REAL UA, UB, ...
C     POINTER (IAUA, UA(MIX,MJX,MKX)), (IAUB, UB(MIX,MJX,MKX)), ...
      COMMON /UA_CMN/ UA(MIX, MJX, MKX), UB(MIX, MJX, MKX), ...

ccc   Note: ADIFOR does not create gradient object variable decla-
ccc         rations here, since the COMMON /UA_CMN/ is not active.
```

```
      COMMON /ADDR2/ IAPA, IAPB, IAZO, IAHO, ...
      INTEGER IAPA, IAPB, IAZO, IAHO, ...
      REAL PSA, PSB, ZOL, HOL, ...
C     POINTER (IAPA,   PSA(MIX,MJX)), (IAPB,   PSB(MIX,MJX)),
C    +        (IAZO,   ZOL(MIX,MJX)), (IAHO,   HOL(MIX,MJX)), ...
      COMMON /PSA_CMN/ PSA(MIX, MJX), PSB(MIX, MJX), ZOL(MIX, MJX),
     +HOL(MIX, MJX), ...

ccc   ADIFOR inserts REAL gradient object variables in corres-
ccc   pondance to active variables, and puts these in a COMMON:
      REAL ...
      REAL G_HOL(G_PMAX_, MIX, MJX)
      REAL G_ZOL(G_PMAX_, MIX, MJX)
      REAL G_PSB(G_PMAX_, MIX, MJX)
      REAL G_PSA(G_PMAX_, MIX, MJX)

      COMMON /G_PSA_CMN/ G_PSA, G_PSB, G_ZOL, G_HOL, ...
```

Figure 8: Excerpts from the file "ADIFORed_hirpbl.f"

this unmasking entails the reintroduction of the `POINTER` statement and the removal of both the `INTEGER` statement for the pointers and the `COMMON` statement for the pointees. It is worth noting that the upper section is now identical to what it looked like originally in Figure 6, as it should since there were no active variables present. By contrast, in the lower section of Figure 9, we note the continued presence of the `REAL` declarations of the gradient variables; however, the `COMMON` declaration for the gradient variables is deleted in anticipation of last step of the transformation.

gradient_ptr.pl performs the last step in our transformation scheme, resulting in *augmented_ADIFORed_hirpbl.f* (Figure 10). For every `POINTER` statement in

```
      COMMON /ADDR1/ IAUA, IAUB, ...
      REAL UA, UB, ...

ccc   We uncomment the original POINTER statement, and remove the
ccc   INTEGER and COMMON statements which were introduced earlier.
      POINTER (IAUA, UA(MIX,MJX,MKX)), (IAUB, UB(MIX,MJX,MKX)), ...
```

```
ccc   The pointers in the COMMOM /ADDR2/ will point to active
ccc   variables.
      COMMON /ADDR2/ IAPA, IAPB, IAZO, IAHO, ...

      REAL PSA, PSB, ZOL, HOL, ...
      POINTER (IAPA,   PSA(MIX,MJX)), (IAPB,   PSB(MIX,MJX)),
     +         (IAZO,   ZOL(MIX,MJX)), (IAHO,   HOL(MIX,MJX)), ...
      REAL ...
      REAL G_HOL(G_PMAX_, MIX, MJX)
      REAL G_ZOL(G_PMAX_, MIX, MJX)
      REAL G_PSB(G_PMAX_, MIX, MJX)
      REAL G_PSA(G_PMAX_, MIX, MJX)
```

Figure 9: Excerpts from the file "repointered_ADIFORed_hirpbl.f"

re_pointered_ADIFORed_hirpbl.f, *augment_ptr.pl* generates a corresponding POINTER statement for the gradient variables and also a COMMON containing those gradient pointers.

It should be clear why these are precisely the transformations needed to complete our scheme for the objects in the lower (active) section of Figure 10. For the upper section, though the gradient pointers do not enter into the computation of derivatives (because they correspond to inactive variables), both the gradient COMMON and POINTER statements are needed for the proper implementation of the gradient addressing scheme. As we shall see in the next section, COMMON /G_ADDR1/ is accessed in SUBROUTINE G_ADDRX1C; the gradient POINTER statement is needed so that the compiler knows the sizes of the items in the COMMON. We also note that in the upper section, the pointee variables are *not* dimensioned anywhere.

Mapping Gradient POINTERs to Addresses

The sole remaining issue to be resolved is the above-mentioned gradient POINTER addressing scheme. Earlier, we discussed the addressing subroutines, in particular,

```
ccc   We insert a COMMON statement, listing the original pointers
ccc   prepended by 'G_':
      COMMON /G_ADDR1/ G_IAUA, G_IAUB, ...
      COMMON /ADDR1/ IAUA, IAUB, ...

      REAL UA, UB, ...

ccc   We insert a POINTER statement for the gradient objects in
ccc   correspondance to the original POINTER statement:
      POINTER (g_IAUA, g_UA), (g_IAUB, g_UB), ...
      POINTER (IAUA, UA(MIX,MJX,MKX)), (IAUB, UB(MIX,MJX,MKX)), ...
```

```
      COMMON /G_ADDR2/ G_IAPA, G_IAPB, G_IAZO, G_IAHO, ...
      COMMON /ADDR2/ IAPA, IAPB, IAZO, IAHO, ...
      REAL PSA, PSB, ZOL, HOL, ...
      POINTER (g_IAPA, g_PSA), (g_IAPB, g_PSB), (g_IAZO, g_ZOL),
     +        (g_IAHO, g_HOL)
      POINTER (IAPA,   PSA(MIX,MJX)), (IAPB,   PSB(MIX,MJX)),
     +        (IAZO,   ZOL(MIX,MJX)), (IAHO,   HOL(MIX,MJX)), ...
      REAL ...
      REAL G_HOL(G_PMAX_, MIX, MJX)
      REAL G_ZOL(G_PMAX_, MIX, MJX)
      REAL G_PSB(G_PMAX_, MIX, MJX)
      REAL G_PSA(G_PMAX_, MIX, MJX)
```

Figure 10: Excerpts from the file "augmented_ADIFORed_hirpbl.f"

ADDALL and ADDRX1C and the arrays ALLARR and IAXALL. What is now needed are subroutines G_ADDALL and G_ADDRX1C (and G_ADDRX1N) to implement the corresponding mapping for the gradient scheme involving G_ALLARR and G_IAXALL. Fortunately, it turns out that we can do this quite simply. Having included all gradient pointers (corresponding to active and inactive variables) in COMMONs (see Figure 10), we can now exploit the inherent structural commonality between the original and the gradient addressing schemes.

We first declare the arrays G_ALLARR and G_IAXALL (Figure 11). Note that G_ALLARR has the added leading dimension for the gradient vectors, but G_IAXALL has the same dimension as IAXALL, since there is a one-to-one correspondence between the original POINTERs and their gradient counterpart.

```
      REAL G_ALLARR
      COMMON /G_HUGE/ G_ALLARR(G_PMAX,IHUGE,MAXNES)
      INTEGER G_IAXALL
      COMMON /G_ADDR0/ G_IAXALL(NUMVAR,MAXNES)

      SUBROUTINE G_ADDALL

      G_IAXALL(NCOUNT,K) = LOC(G_ALLARR(1,1+ (N-1)*IX3D,K))
```

```
      SUBROUTINE G_ADDRX1C(IARR)

      COMMON /G_ADDR1/ G_IAUA, G_IAUB, ...
```

Figure 11: Excerpts from the files "g_addall.f and g_addrx1c.f"

Given the of Fortran array A, declared as "DIMENSION A(X,Y,Z)",
and the formula for linearizing the array offset for the entry A(i,j,k):

i + (j-1)*X + (k-1)*X*Y,

we can compute the difference between two consecutive `G_IAXALL` entries,
`G_IAXALL(NCOUNT,K)` and `G_IAXALL(NCOUNT+1,K)`:

```
    LOC(G_ALLARR(1,1+ ((N+1)-1)*IX3D,K))  -
    LOC(G_ALLARR(1,1+ (N-1)*IX3D,K))

 =  [ 1 + ((1 + N*IX3D)-1)*G_PMAX + (K-1)*G_PMAX*IHUGE ] -
    [ 1 + ((1 + N*IX3D - IX3D)-1)*G_PMAX + (K-1)*G_PMAX*IHUGE ]

 =  (N*IX3D)*G_PMAX - (N*IX3D - IX3D)*G_PMAX
 =  IX3D*G_PMAX
```

Figure 12: Computation of a gradient pointer offset

We then copy each addressing subroutine to its gradient counterpart and perform a few changes, as shown by example in Figure 11 (compare these with Figures 2 and 3). Thus `G_ADDRX1C/N` differ from `ADDRX1C/N` only in the names used in the pointer `COMMON`s. And `G_ADDALL` differs from `ADDALL` in that "`ALLARR(`" and "`IAXALL`" are replaced by "`G_ALLARR(1,`" and "`G_IAXALL`", respectively. Figure 12 is the address computation showing the correctness of the resulting gradient pointer offset calculations.

Acknowledgments

The authors express their thanks to Thomas Canfield (ANL/MCS), John Michalakes (ANL/MCS), and Kathy Simunich (ANL/EID) for helping us understand MM5.

References

[1] *AIX XL FORTRAN Compiler/6000 Language Reference.* International Business Machines Corporation, 1992.

[2] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank and Paul Hovland. ADIFOR: Generating Derivative Codes from Fortran Programs. *Scientific Programming*, 1(1), pp. 1-29, 1992

[3] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank and Paul Hovland. *Getting Started with ADIFOR.* Argonne Technical Memorandum MCS-TM-164, 1992.

[4] *Cray Computer Systems CFT77 Reference Manual.* Cray Research, Inc., 1986.

[5] Georg A. Grell, Jimy Dudhia and David R. Stauffer. *A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5).* National Center for Atmospheric Research, NCAR/TN-398+IA, 1993.

[6] Phillip L. Haagenson and Jimy Dudhia. *The Penn State/NCAR Mesoscale Model (MM5) Source Code Documentation.* National Center for Atmospheric Research, NCAR/TN-392, 1993.

[7] *Sun FORTRAN Reference Guide.* Sun Microsystems, Inc. 1991.