Predicting the future with Linux - Numerical Weather Forecasting

Toon Moene
toon@moene.indiv.nluug.nl

22nd of May, 1999

- The Atmospheric Equations of Motion
- Discretization $\rightarrow$ Fortran code
- Enters ... GNU Fortran (g77)
- Fixing it up
- Alias analysis
- Future improvements

The Atmospheric Equations of Motion

- The conservation of momentum (Newtons "Force Law")

  $\frac{\mathbf{F}}{m} = \frac{d\mathbf{v}}{dt}$

- The conservation of mass (continuity equation)

- The conservation of energy

$$\frac{\mathbf{F}}{m} = \frac{d\mathbf{v}}{dt}$$

in a curvilinear, rotating, coordinate system:

$$\frac{du}{dt} - \frac{uvtan\phi}{a} + \frac{uw}{a} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + 2\Omega vsin\phi + 2\Omega wcos\phi$$

$$\frac{dv}{dt} + \frac{u^2 tan\phi}{a} + \frac{vw}{a} = -\frac{1}{\rho}\frac{\partial p}{\partial y} - 2\Omega usin\phi$$

$$\frac{dw}{dt} - \frac{u^2 + v^2}{a} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - g + 2\Omega ucos\phi$$

Scale analysis:

$U \sim 10ms^{-1}$	horizontal velocity scale
$W \sim 0.01ms^{-1}$	vertical velocity scale
$L \sim 10^6 m$	length scale
$D \sim 10^4 m$	depth scale
$\frac{L}{U} \sim 10^5 s$	time scale
$\Delta p \sim 10^3 Nm^{-2}$	typical pressure difference
$\rho \sim 1 kgm^{-3}$	density of air
$a \sim 10^7 m$	earth radius
$2\Omega \sin\phi \sim 10^{-4} s^{-1}$	rate of rotation

leads to (within 10 %):

$$-fv = \frac{1}{\rho}\frac{\partial p}{\partial x}$$

$$fu = \frac{1}{\rho}\frac{\partial p}{\partial y}$$

with $f = 2\Omega sin\phi$, and:

$$\frac{1}{\rho}\frac{\partial p}{\partial z} = -g$$

From continuous equations to finite differences:

$$\frac{du}{dt} = lim_{\Delta t \to 0} \frac{\Delta u}{\Delta t}$$

gives rise to the following approximation:

$$\frac{du}{dt} \simeq \frac{u_{i+1} - u_i}{t_{i+1} - t_i}$$

and

$$x_{i+\frac{1}{2}} \simeq \frac{1}{2}(x_{i+1} + x_i)$$

Hence the following expresses, in Fortran, the computation of kinetic energy:

```
      DO 810 I=ILONP2,ILNLT
         ZEK(I) = 0.25 *
     +         ( ( PUZ(I-1   ,K)*PUZ(I-1   ,K)
     +                   *HYU(I-1   )
     +           + PUZ(I     ,K)*PUZ(I     ,K)
     +                   *HYU(I     ))*RHYV (I)
     +         + ( PVZ(I-ILON,K)*PVZ(I-ILON,K)
     +                   *HXV(I-ILON)
     +           + PVZ(I     ,K)*PVZ(I     ,K)
     +                   *HXV(I     ))*RHXU (I) )
 810  CONTINUE
```

If we now again look at the Fortran code:

```
      DO 810 I=ILONP2,ILNLT
         ZEK(I) = 0.25 *
     +        ( ( PUZ(I-1   ,K)*PUZ(I-1   ,K)
     +                *HYU(I-1   )
     +          + PUZ(I     ,K)*PUZ(I     ,K)
     +                *HYU(I     ))*RHYV (I)
     +        + ( PVZ(I-ILON,K)*PVZ(I-ILON,K)
     +                *HXV(I-ILON)
     +          + PVZ(I     ,K)*PVZ(I     ,K)
     +                *HXV(I     ))*RHXU (I) )
 810  CONTINUE
```

let's see at what's involved in computing just one array access PUZ(I,K):

```
T1 = (K - 1) * S1  S1: Size of 1 column of PUZ
T1 = T1 + I - 1    Note: Fortran array indices
T1 = T1 * 4        start at 1
addr(EL) = addr(PUZ) + T1
```

So to access one element of PUZ we already have performed 6 integer operations, before even being able to do some floating point math.

Clearly, there must be a better way to get at PUZ(I,K) than this one:

```
T1 = (K - 1) * S1
T1 = T1 + I - 1
T1 = T1 * 4
addr(EL) = addr(PUZ) + T1
```

The first thing to note is that:

```
T1 = (K - 1) * S1
```

is a *loop invariant* - it doesn't change in the loop over I. Secondly,

```
T1 = T1 + I - 1
T1 = T1 * 4
```

just means that T1 gets incremented by 4 on every loop iteration. So, with the right initialisation of T1

```
T1 = T1 + 4
```

suffices. This is called "strength reduction of *induction variables*" because you turn a multiplication into an addition.

Unfortunately, *gcc* tended to compute `S1` in a way that led to huge inefficiencies. So the following routine

```
subroutine copy3(a, b, n)
real a(n,n,n), b(n,n,n)
do i = 1, n
   do j = 1, n
      do k = 1, n
         b(k,j,i) = a(k,j,i)
      enddo
   enddo
enddo
```

didn't generate the following for the inner loop:

```
movel (a0)+,(a1)+
```

but you got something like *this*:

```
movel a6@(-8),d1
lsrl #3,d1
tstl a3
sne d0
extbl d0
negl d0
addl d0,d1
movel a6@(-16),d2
mulsl d1,d2
movel a6@(-4),d1
lsrl #3,d1
...
negl d0
addl d0,d1
movel a6@(-16),d0
mulsl d1,d0
movel a6@(-12),d1
lsrl #3,d1
mulsl d4,d1
addl a6@(8),d0
movel d0,a0
addl d1,a0
movel a0@(d3:l:4),a1@(d3:l:4)
```

Alias analysis: Which variables / arrays are an alias ("another name") for the same memory location ?

```
      SUBROUTINE SAXPY(N,SA,SX,SY)
      REAL SX(N),SY(N),SA
      DO 10 I = 1,N
         SY(I) = SY(I) + SA*SX(I)
   10 CONTINUE
      END
```

In the example above, four pointers are passed to routine `SAXPY`. The Fortran rules state that none of the variables and arrays may overlap each other. Hence:

- Arrays `SX` and `SY` may be copied to a vector register prior to the loop.

- Variable `SA` is a *loop invariant*.

John Carr implemented a form of alias analysis that (among other features) honors the Fortran alias rules. It was first used in *egcs*.

Let's turn our attention to the kinetic energy loop again:

```
      DO 810 I=ILONP2,ILNLT
         ZEK(I) = 0.25 *
     +        ( ( PUZ(I-1   ,K)*PUZ(I-1   ,K)
     +                *HYU(I-1   )
     +          + PUZ(I     ,K)*PUZ(I     ,K)
     +                *HYU(I     ))*RHYV (I)
     +        + ( PVZ(I-ILON,K)*PVZ(I-ILON,K)
     +                *HXV(I-ILON)
     +          + PVZ(I     ,K)*PVZ(I     ,K)
     +                *HXV(I     ))*RHXU (I) )
 810  CONTINUE
```

If we strength reduce all induction variables and move all loop invariant code out of the loop, we need 11 registers to hold the addresses needed to step through the arrays.

We can do better, by noting that

```
{ PUZ(I-1   ,K), PUZ(I     ,K) }
{ PVZ(I-ILON,K), PVZ(I     ,K) }
{ HYU(I-1   )  , HYU(I     )   }
{ HXV(I-ILON)  , HXV(I     )   }
```

form 4 *equivalence classes* of induction variables that differ only by a constant - which means they can be written in the form of address-register-with-offset.

In doing so, we save 4 registers and need only 7 registers for addressing.

Richard Henderson implemented this optimization, which will be part of egcs-1.2 ... I mean gcc-2.95.

Let's now turn to some optimizations that are not done *yet* but are offered by other Fortran compilers.

If we turn our attention to the kinetic energy loop for a last time:

```
      DO 810 I=ILONP2,ILNLT
         ZEK(I) = 0.25 *
     +        ( ( PUZ(I-1   ,K)*PUZ(I-1   ,K)
     +                  *HYU(I-1   )
     +          + PUZ(I     ,K)*PUZ(I     ,K)
     +                  *HYU(I     ))*RHYV (I)
     +        + ( PVZ(I-ILON,K)*PVZ(I-ILON,K)
     +                  *HXV(I-ILON)
     +          + PVZ(I     ,K)*PVZ(I     ,K)
     +                  *HXV(I     ))*RHXU (I) )
 810  CONTINUE
```

and note that we need 7 address registers and 1 loop count register, we see that we *still* run out of registers when coding this loop for the i386.

A solution is to *split* the loop:

```
      DO 811 I=ILONP2,ILNLT
         ZEK(I) = 0.25 *
     +        ( ( PUZ(I-1   ,K)*PUZ(I-1   ,K)
     +                   *HYU(I-1   )
     +          + PUZ(I     ,K)*PUZ(I     ,K)
     +                   *HYU(I     ))*RHYV (I) )
 811  CONTINUE

      DO 812 I=ILONP2,ILNLT
         ZEK(I) = ZEK(I) + 0.25 *
     +        ( ( PVZ(I-ILON,K)*PVZ(I-ILON,K)
     +                   *HXV(I-ILON)
     +          + PVZ(I     ,K)*PVZ(I     ,K)
     +                   *HXV(I     ))*RHXU (I) )
 812  CONTINUE
```

This works, because the array ZEK which is *set* in this loop, is not *used* in the loop. Now both loops use 4 address registers and a loop count register.

The opposite action is to *fuse* two loops with the same number of iterations:

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

can be combined:

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

if `A` and `Y` do not overlap and neither `B` and `X`. Just as with loop unrolling, this has the effect of increasing the work done per loop iteration, but without the overhead.

One optimization that is especially suited for finite difference code is the combining of array accesses after loop unrolling. A typical loop:

```
DO I = 2, N-1
   X(I) = Y(I-1) + Y(I) + Y(I+1)
ENDDO
```

Unroll this loop (assume for simplicity that its iteration count is even):

```
DO I = 2, N-1, 2
   X(I  ) = Y(I-1) + Y(I  ) + Y(I+1)
   X(I+1) = Y(I  ) + Y(I+1) + Y(I+2)
ENDDO
```

Note that `Y(I)` and `Y(I+1)` occur twice in the loop - it would be beneficial to only read them from memory once.

Another loop optimization that suggests itself is the following, trying to eliminate unnecessary loop constructs:

```
SUBROUTINE SUB(A, B, C, N, M)
REAL A(N, M), B(N, M), C(N, M)
DO J = 1, M
   DO I = 1, N
      C(I, J) = A(I, J) + B(I, J)
   ENDDO
ENDDO
END
```

If we could generate code that looks like what is written below, we would have nothing but *one* loop.

```
DO K = 1, M*N
   C(K, 1) = A(K, 1) + B(K, 1)
ENDDO
```

In this case, this transformation can be done because the tuple (i, j) forms an induction variable $i + n * (j - 1)$ in its own right (replaced by k in the *collapsed* loop).

Conclusions:

- In the four years since *g77* went public several Fortran specific optimizations have found their way into *gcc* or rather *egcs*:

  1. Optimal treatment of multi-rank arrays.
  2. Alias analysis with Fortran rules.
  3. Optimal treatment of strength reduction and elimination of induction variables.

- Missing still are optimizations common in other Fortran compilers, which are closer to transformations on the input source.

Input to the Fortran Birds-of-a-Feather Session (the first column lists the number of times the item in the second column has been brought up):

6	Full Fortran 90 support.
3	Cray POINTER support.
2	ALLOCATE/DEALLOCATE.
2	Run-time array bound checking.
2	Intrinsics in parameter statements.
2	Improved performance for complex code.
2	Improved debugging.
1	Full compatibility with f2c.
1	Support for OpenMP parallellization.
1	Select Case for character variables.
1	Generation of thread safe code.