Predicting the future with Linux - Numerical Weather Forecasting

Toon Moene

toon@moene.indiv.nluug.nl

ABSTRACT

This paper describes the equations used in numerical weather forecasting with an eye towards their transcription in Fortran and subsequent compilation and optimization by the GNU compiling system. Discussed are trade-offs in optimizations and optimizations especially relevant for this kind of code.

1 Introduction

Ever since L.F. Richardson described a method for solving the atmospheric equations of motion in a way suitable for weather *forecasting* ([4]), it was clear that weather forecasting once would become a computational science, even though his initial effort wasn't really successful.

Nowadays, we know how to overcome the numerical problems he encountered; furthermore, the observation network has reached a density that enables us to determine the "initial state" of the atmosphere, from which a forecast is made, with sufficient accuracy.

On the other side of the equation electronic computers (as opposed to human ones) have progressed far enough that I can now make simple forecasts on my laptop.

2 The Atmospheric Equations of Motion

These equations can be derived from the following basic conservation laws:

- The conservation of momentum (Newtons "Force Law").
- The conservation of mass (continuity equation).
- The conservation of thermodynamic energy.

From the conservation of momentum we get the following equations for the three components of the change in wind velocity with time (following [3]):

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

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

$$\frac{dw}{dt} - \frac{u^2+v^2}{a} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - g + 2\Omega ucos\phi \quad (3)$$

x, y, z, t	three space coordinates plus time
u, v, w	wind velocity components
ρ	density of the air
p	pressure of the air
ϕ	latitude
a	radius of the earth
g	acceleration of gravity
Ω	angular velocity of the earth rotation

The complexity in these formula's derived from Newtons simple $F = ma$ is caused by the fact that the movement of atmospheric air masses takes place in a rotating, curvi-linear coordinate system.

Applying scale analysis (for depression-sized systems; meteorologists call these "synoptic scale systems") to these equations shows that of the first two only the following remains (within an error of about 10 %):

$$-fv = \frac{1}{\rho}\frac{\partial p}{\partial x} \tag{4}$$

$$fu = \frac{1}{\rho}\frac{\partial p}{\partial y} \tag{5}$$

with $f = 2\Omega sin\phi$

due to the fact that vertical velocity is small and the length scale is large with respect to (horizontal) velocity. These equations express that to a first approximation, the atmosphere is in a balance which cancels the pressure gradient force by the Coriolis force, or explained differently, that to first order, the deflection of wind by the earth's rotation is in balance with the force exerted by pressure differences.

This, however, already shows one of the difficulties of weather *forecasting*: The approximate equations we derived do not contain a time derivative, i.e. they can't be used for computing atmospheric development !

A similar scale argument will derive the "hydrostatic equation" from the vertical equation of motion:

$$\frac{1}{\rho}\frac{\partial p}{\partial z} = -g \tag{6}$$

which basically tells you that the weight on a horizontal plane equals the mass of air above it; on these scales there is no significant vertical acceleration.

Subsequently, from the continuity equation we learn that atmospheric motion on synoptic scale is predominantly horizontal.

3 Fortran Code

How does this translate to Fortran code, after discretizing using finite differences ? (for a treatise on finite difference approximations to partial differential equations, see [2], Chapter 19.1 Flux-Conservative Initial Value Problems)

Figure 1 gives a snippet from the routine in our code responsible for the advection computations.

What we see here is a part of the computation of kinetic energy `ZEK`, in a loop over horizontal grid points (`DO I = ...`). One recognizes the contributions of the u- and v-component of the wind (`PUZ` and `PVZ`, respectively), multiplied by map factors (due to the fact that the model works in SI units, but the grid is defined in a latitude / longitude coordinate system). There is an outer loop - using index `K` - over layers in the vertical.

If we now shift our focus to that of the compiler, we see a loop accessing 7 arrays in memory storage order (remember that Fortran stores arrays in column major order, which means that for accessing in memory order one has to vary the first index in the inner loop).

What optimizations can a compiler perform on this code ? What are the transformations it has to make to remove all redundant computations ? This is the issue of the next few paragraphs.

4 History of g77

The first public release of g77 was on the 17th of February, 1995, version 0.5.8. At that time the most recent gcc release was 2.6.3.

Very soon after going public we received the first "complaints" about slow code that was generated (especially with respect to f2c, the free Fortran-to-C compiler). After about two months I was able to pinpoint the first culprit: For a rank-two array like `PUZ` and `PVZ` above, the compiler must generate code to compute the final offset into it by computing the size of the rank-1 subarray "in" it. It turned out that gcc does all these computations in sizes of bits and computes the byte-size of it by performing a ceiling division (i.e. a division that computes the first integral number larger than the real quotient). Needless to say, these divisions led to atrocious code in loops like the above.

This could quickly be solved by checking, at the spot where the division was generated, whether the sub-array size was an integral number of bytes and substituting an exact division in that case. Constant folding (the optimization that combines expressions containing only constants) would then take care of the (exact) division by eight.

However, this was only the beginning. If one looks again at the rank-2 arrays above (and forgetting the bit to byte size conversion for the moment), then the offset into the array `PUZ(I,K)` - assuming that `PUZ` contains 4-byte REALs - is computed as follows (see Figure 2; `S1` is the extent of the first index).

So before we even have accessed an element of `PUZ`, we have already performed 6 integer instructions (one of which is an integer multiplication !); and this for code that is well known to be "floating point intensive" !

Clearly, there must be a better way - and indeed there is. If we turn to [1], better known as the "Dragon Book" because of its cover) Chapter 13, Paragraphs 13.4 "Loop-Invariant Computations" and 13.5 "Induction Variable Elimination", we see that we can get rid of all of this: The first computation is loop-invariant (`K` and `S1` are independent of the value of `I`) and `addr(EL)` simply increases by 4 with every loop round.

Interestingly enough, both optimizations mentioned were already part of gcc when g77 was released. However, they didn't work "optimally". One of the reasons is that both were "capped" by a heuristic to prevent them to take up too many registers. For instance, if you don't compute `addr(EL)` the way it is done above, you have to reserve a register to hold its (changing) value throughout the loop traversal. Although this heuristic seemed reasonable for loops in C programs, which are rank-1 by design, it turned out to be suboptimal for Fortran loops with rank-2, -3 or even higher arrays. To return to the example above, if one completely removes all invariant code from the loop and eliminates all induction variables, one only needs a register for `addr(EL)` and not for `K, S1, T1` or even `addr(PUZ)` any more !

A further complication was that the routines that performed said optimizations were not able to recognise `T1 + I - 1` as an induction variable (which is of the form $invariant + iv + constant$). Because Fortran array indices start with 1 by default, this meant that a lot of the induction variables weren't even noticed. This was solved by running loop optimization twice, to get them recognized in two stages.

These improvements went into g77-0.5.18 (as an optional patch to gcc-2.7.2), which was released on April 1st, 1996.

5 Alias analysis

By now, at least on these aspects, we were on level with run-of-the-mill Fortran compilers. However, a typical Fortran class of optimizations was still missing: Alias (variable overlap) analysis based on the Fortran rules of aliasing:

- Actual arguments to routines cannot overlap items in common blocks (static storage).
- Actual arguments cannot overlap each other.

[In fact, the Fortran rules can be stated more concisely as: If you, the programmer, don't tell the compiler two items can overlap, by using the construct EQUIVALENCE, they shouldn't].

Unfortunately, the example I chose doesn't contain opportunities for this type of optimization, but it *is* very common in Fortran. Consider a well-known example from the LAPACK library (Figure 3)

Clearly, scheduling of the code in the loop (especially after loop unrolling) is impaired if the compiler cannot use the fact that `SX` and `SY` are guaranteed not to overlap. Furthermore, because `SY` cannot overlap `SA` (the scalar multiplier), `SA` is *therefore* a loop invariant.

An alias analysis that includes the Fortran rules written by John Carr was offered to the GCC community several times during 1996 and rejected. We backstitched his code (via a patch distributed with g77-0.5.20, March 1st, 1997) into gcc-2.7.2, because of its obvious benefit to Fortran optimizations. It is part of egcs since its very beginning in August 1997.

6 The present

If we return to the example in Figure 1 above, it is easy to see that the standard loop optimizations I described there will use more registers than is strictly necessary. The reason for this is that induction variable elimination will identify each of the array accesses as an induction variable and reserve a register for it. This means that in that particular loop, 11 registers are necessary for holding addresses. However, as can easily be seen, if we categorize the array accesses in classes that differ by a constant, we're left with only 7 (assuming `ILON` is determined a constant by constant propagation). This classification is useful, because most if not all computer architectures support offset addressing, which means that for every class only one register is needed.

Interestingly, gcc originally already contained a routine to combine these induction variables; it didn't seem to ever recognise the opportunities, though. Work by Richard Henderson, which will show up in egcs-1.2, will remedy this shortcoming.

7 The future

Lets turn to some optimizations that other Fortran compilers perform that g77/gcc/egcs does not, yet.

7.1 Loop splitting

In the loop in Figure 1 above, even after all the mentioned code transformations are performed, we still need 8 integer registers, which means that suboptimal code will result on the Intel (IA32) architecture, which doesn't offer that many free integer registers. One way around this is loop splitting (see Figure 4).

Now you only need 5 integer registers (4 addresses and a count) per loop. The reason that this transformation can be made lies in the fact that the store into `ZEK` does not change any of the other arrays accessed in the loop.

```
      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
```

Figure 1: Computation of kinetic energy

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

Figure 2: Computation of array offset

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

Figure 3: Alias analysis example

```
      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
```

Figure 4: Loop splitting

7.2 Combining array accesses after loop unrolling

In finite difference code often several adjacent elements of the same array are accessed in a loop body (see Figure 5).

Obviously, `Y(I)` in one loop iteration is `Y(I-1)` in the next, and mutatis mutandis for `Y(I+1)`. This means that, after loop unrolling, several accesses of the same array element will be disguised as being different. It would be nice if these values were kept in registers instead of being (re-)fetched from memory.

7.3 Loop fusion

Given two loops with the same number of iterations, one can often combine them (see Figure 6).

This has the same beneficial effect as loop unrolling (getting more work done per iteration) without the extra code overhead. Of course this only works if the store into `A` does not change `B` or `Y` and vice versa.

7.4 Loop interchange

In Fortran, to address array elements in storage order, one has to have the left-most index varying fastest. If this isn't the case, sometimes the inner and the outer loop can be interchanged (see Figure 7).

Although normally Fortran users won't write code like the first loop nest, it might result from other transformations.

7.5 Loop blocking/tiling

In loop nests where the depth is greater than the rank of the arrays being worked on, it is beneficial to split up the work into "blocks" instead of sweeping across entire arrays, because in that way, the parts of the matrices being worked on can be contained in cache. The obvious candidate is matrix multiplication (see Figure 8 where `C` is assumed being zeroed beforehand):

(Note that the first optimization to be performed here is to move the i-loop inside the k-loop).

The resulting code is complicated and messy and hence not shown here - moreover, the optimal arrangement depends on cache size and replacement policy.

7.6 Collapsing "perfect" loop nests

See the nested loop in Figure 9. In this loop nest it is possible to collapse the inner and outer loop because the tuple (i, j) forms an induction variable $i + n * (j - 1)$ in its own right (replaced by k in the "collapsed" loop).

The reason this is possible is that the extent of the arrays along the first rank is equal to the end value of the loop index i - hence after a complete i-loop, incrementing j by one and starting i with 1 will update the "collapsed" loop index $k = i + n * (j - 1)$ by one.

After I explained this on the egcs mailing list, I learned from a Cray employee that this transformation is discussed in on page 256 of [5], and is already present in Cray compilers for years.

As can be seen from the loop in Figure 1, in our code we did something similar "by hand", because the I index runs over the whole horizontal domain, instead of having two indices running over latitudes and longitudes.

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

Figure 5: Combining array element accesses in unrolled loops

```
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
```

Figure 6: Loop fusion

```
Transform:

   DO I = 1, N
      DO J = 1, M
         A(I, J) = B(I, J) + C(I, J)
      ENDDO
   ENDDO

into:

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

Figure 7: Loop interchange

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

Figure 8: Loop candidate for blocking / tiling

```
   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

Collapse the loop into:

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

Figure 9: Collapsing perfect loop nests

8 Conclusions

Although egcs/g77 now performs a reasonable set of optimizations on Fortran code, it is shown that more is possible. However, the concensus at present is that it is next to impossible to perform these ”higher order” code restructuring optimizations on the current intermediate code (RTL); it will probably have to wait until the Fortran Front End can compile entire routines into the intermediate tree code format for these optimizations to become practical.

References

[1] A.V Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Co., 1978.

[2] W.H. Press et al. *Numerical Recipes in FORTRAN*. Cambridge Univ. Press, 1992.

[3] J.R. Holton. *An Introduction to Dynamic Meteorology*. Academic Press, Inc, 1979.

[4] L.F. Richardson. *Weather Prediction by Numerical Process*. Cambridge Univ. Press, 1922.

[5] H Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. (Publisher unknown), 1990.