Gfortran & HIRLAM: From non-compile to test platform in 6 months

Toon Moene

The GNU Fortran Team

`toon@moene.indiv.nluug.nl`

April 5, 2006

Abstract

At last year's GCC summit it was shown how close gfortran was to being able to completely compile HIRLAM, a state of the art limited area weather forecasting model, consisting of 1,200,000+ lines of Fortran and some tens of thousands of lines of C code (see *http://hirlam.knmi.nl*).

Six months later a prerelease version of gfortran 4.1 was able to fully compile and link the main executables of the HIRLAM system, permitting real forecasts to be made.

At present, gfortran is used as one of the preferred platforms to test new features of the forecasting system.

1   Introduction

To enable widespread use of HIRLAM for research purposes, it is paramount that the forecasting system can be used on platforms based on free software, e.g., GNU/Linux, using the GNU compilers.

As of mid-December 2005, it is possible to compile the major components of the system using gfortran 4.1 (then a prerelease version).

This paper discusses how well this compiler - or rather, subsequent versions - fare in compiling HIRLAM, how to determine whether the generated code is correct, and how well the code is optimized.

2   The HIRLAM weather forecasting system

Like all numerical weather forecasting systems, HIRLAM consists of the following main components:

- The forecasting code, used to compute later weather developments from an *initial state* of the atmosphere.

- The data asimilation code, that determines this initial state from a previous forecast and observations (so called because it *assimilates* the observations in the model state).

- Code to compute properties of the surface from satellite information and transform it to the HIRLAM grid, to be used as the lower boundary condition during the forecast.

- Code to interpolate global weather forecasts to the HIRLAM grid to be able to use them as lateral boundary conditions during the limited area forecast.

- Code to compare forecasts with observations for verification purposes.

- Various file format conversion routines.

Except for the 3rd set of programs, all code has been used in the tests for this paper (the files that contain the surface properties have been retrieved from another platform HIRLAM runs on).

3 The test platform

All tests were performed on an HP zv6025EA laptop, featuring 512 Mbyte of memory and a 2 Ghz Athlon 64 processor, running Debian AMD64 *testing* (dated April 15th) and using gcc version 4.2.0 20060415 (experimental).

The HIRLAM system was configured to use a grid of 166 squares in the east-west direction, 130 squares in the north-south direction and 31 layers in the vertical (roughly 55x55 km horizontal and 100 m (lower atmosphere) to 2000 m (upper atmosphere) vertical resolution). This is essentially the layout KNMI (the Dutch Meteorological Service) used operationally until March, 2002.

4 How to check whether the code is correct

All models are wrong, but some are useful

How does one proceed convincing oneself that a new compiler actually correctly compiles the weather forecasting code ?

This is not an easy question. First of all, there is no hope that two correct compilers even on the same system, using the same floating point instruction set, will generate bit-identical forecasts.

The reason for this is twofold:

- It is very hard to write a Fortran program with a single, unambiguous, meaning.

- Even given an unambiguous Fortran program, any compiler is free to provide its own approximation to the computations specified by the program.

An example of the first issue:

```
READ*,X,Y,Z
PRINT*,X+Y+Z
END
```

This program has three distinct meanings: it can print the value of (X+Y)+Z, X+(Y+Z) or even (X+Z)+Y.

An example of the second issue:

```
X = 1 000 000.
PRINT*,SIN(X)
END
```

The value printed will depend on the thoroughness of the run time library writer, and/or the willingness of the programmer to sacrifice accuracy for running time (-"use fast math lib").

So bitwise comparison of the same model run, compiled by two different compilers on the same platform, will not give us an answer to the question: Is the code generated by the new compiler correct ?

A better approach is to check whether the code compiled by the second compiler gives the same forecast (i.e., within human discernability) as the one compiled by the first.

- Does it give the same overall weather development (fronts, depressions, large scale precipitation) ?

- When there are differences, are they known to be caused by hard-to-forecast phenomena, like the exact path and strength of thunderstorm complexes ?

gfortran passes this test when comparing with the same HIRLAM run at ECMWF (the European Centre for Medium Range Weather Forecasts in Reading, England) using their IBM Power cluster and xlf compiler.

[This is not an absolute guarantee - in the 13 years of running HIRLAM operationally at KNMI, the author has seen spectacular compiler bugs go unnoticed for some time because their effects were hidden by later computations.]

5 Run times of the components

The main time consuming parts of a HIRLAM run are:

- Determining the "best known" initial state of the model atmosphere, using a previous forecast and the new observations of the state of the atmosphere.

- The actual forecast computation itself.

Both components have a different run time dependence on grid size, so the results given here are for the grid layout mentioned above (run times obtained using gfortran 4.2-20060415 with -O2 as optimization option).

Determining the initial state of the model atmosphere takes about 5 minutes on the test system when using only gfortran and its run time library, and about 2 minutes when using the Fast Fourier Transform library *fftw3* instead of the FFT routines that come with the HIRLAM code.

Performing the forecast computations takes about 25 minutes (for a 24 hour forecast).

Because the latter component takes the most time, the following sections will deal only with that component.

6 Hot spots

The *gprof* breakdown of time spent in the forecast computations looks like this (slightly edited to take out non-essential information, and retaining only those routines using more than 2 % of the total time):

```
Flat profile:

% time   calls   name
 18.34   85684   verint_
  9.34    1380   invlo4_
  7.84   85684   bixint_
  6.76     133   sl2tim_
  5.30   14950   condcv_
  4.74   14950   radia_
  4.65   14950   vcbr_
  3.25     133   sldyn_
  2.98   14950   phtask_
  2.42     133   sldynm_
  2.29   14950   phys_
  2.19   14950   prevap_
```

This profile was obtained by compiling with gcc-4.2-20060415 using -O2 as the only optimization option.

To understand better what this breakdown means, a short explanation of the inner workings of the forecast code is in order. Numerical weather forecasting consists of two complementary computational processes, at every time step during the forecast:

- Computing the large scale motion of the atmosphere over the grid boxes, using a discretized version of the (simplified) fluid dynamics differential equations.

- Computing the subgrid effects of this large scale air movement (resulting evaporation/condensation, cloud formation, precipitation, long and short wave radiation, turbulent kinetic energy and momentum, heat and humidity fluxes at the surface - in every grid box).

Traditionally, the first set of computations is collectively known as *dynamics*, whereas the second set is known as *physics* (this naming convention is a bit odd, as all of it is physics in one way or another).

Of the twelve routines above, six belong to "dynamics" (D) and six to "physics" (P):

- `verint`: Compute the 3D interpolation of physical quantities in the departure point of an air parcel that ends up in the grid box under consideration (D).

- `invlo4`: Main computational routine of a low pass filter to damp high frequency numerical noise (D).

- `bixint`: Determine the 3D departure point of an air parcel (D).

- `sl2tim`: Main time stepping routine (D).

- `condcv`: Compute all effects of convective condensation (P).

- `radia`: Compute all effects of incoming (short wave) and outgoing (long wave) radiation (P).

- `vcbr`: Compute net turbulent kinetic energy (production minus dissipation) (P).

- `sldyn`: Half of the main time stepping routine (D).

- `phtask`: Divide the "physics" routines over independent parts of the computational domain (P).

- `sldynm`: Other half of the main time stepping routine (D).

- `phys`: Main "physics" routine (P).

- `prevap`: Compute all effects of precipitation release (P).

Apparently, there is no single *hot spot*. However, the top three routines are obvious candidates for closer scrutiny. This paper will only address `verint`.

7 Various optimizations

The above gprof breakdown of run time spent in various components of the forecast code was based on compilation with -O2 as its only optimization option.

What are the effects of other optimizations ?

7.1 Function inlining

Compiling with -O3 has no discernable effect over compiling with -O2. This is not really surprising, as no routine is small enough to be beneficially inlined.

4

7.2 Loop unrolling

Real programmers are not afraid of five page long DO loops

Loop unrolling has a negligible effect on the run time of HIRLAM, probably because most loops are large (the content of the main loop in `verint` is a single expression spread over 60+ lines).

7.3 Modulo scheduling

Perhaps for the same reason, modulo scheduling has no discernable effect on the run time.

7.4 Fast-and-loose math

The combined effect of the options gathered under *-ffast-math* leads to a reduction in run time by approximately 3 %, with a negligible effect on the results of the forecast.

7.5 Autovectorization

As HIRLAM was originally developed in the mid-80s on traditional vector supercomputers, it is reasonable to assume that GCC's vectorization will have a beneficial effect on the run time of the code.

However, as will be made clear below, most of the loops in HIRLAM are not vectorized yet, so the effect of vectorization currently is negligible.

7.5.1 Maximum obtainable speedup

Using gcc-4.2-20060415, the following code:

```
REAL A(5 000)
REAL B(5 000)
REAL C(5 000)
READ*,X,Y
A = X
B = Y
DO I = 1, 1 000 000
    C = A * B
ENDDO
PRINT*,C(2 000)
END
```

runs in 7.6 seconds on the test system when compiled with option -O2 and in 2.5 seconds when compiled with option -O2 -ftree-vectorize.

So it is safe to assume that the *maximum* speed up from vectorization of the HIRLAM code (with 4 byte REALs in inner loops of 166 elements) will be a factor of 3.

7.5.2 Efficacy of the vectorization pass

Using gcc-4.2-20060415, the effects of using the *-ftree-vectorize* flag on the HIRLAM code are as follows:

vectorized: 2995 loops

not vectorized: 8040 loops

of those 8040, the following lists the numbers by most important cause:

- unhandled data-ref: 3043

- unable to determine dependency: 1846

- complicated access pattern: 1738

- unsupported use in stmt: 685

5

Examples of the first category are mostly loops "in the wrong order", e.g.:

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

Message:

```
vect6.f:4: note: not vectorized:
 unhandled data-ref
vect6.f:4: note: vectorized 0
 loops in function.
```

This will be solved once more general loop restructuring algorithms are implemented in the middle end.

An example of the second problem:

```
 SUBROUTINE S(N, M)
 DIMENSION A(N, M), B(N, M)
 DO J = 1, M
  DO I = 1, N
   A(I, J) = A(I, J) + B(I, J)
  ENDDO
 ENDDO
 END
```

Message:

```
vect1.f:4: note: not vectorized:
 can't determine dependence
  between (*a_35)[D.1306_47]
  and (*a_35)[D.1306_47]
vect1.f:4: note: vectorized 0
 loops in function.
```

Note how the compiler cannot seem to get the dependence between the array references A(I,J) and A(I,J) !

It is not clear yet what causes this problem.

An example of the third problem:

```
 SUBROUTINE S(N)
 DIMENSION A(N), B(N)
 DO I = ISTART, N
  A(I) = B(I)
 ENDDO
 END
```

Message:

```
vect4.f:4: note: not vectorized:
 complicated access pattern.
vect4.f:4: note: vectorized 0
 loops in function.
```

If the variable ISTART is replace by a constant, the loop will vectorize. This is only a problem on targets with 64-bit addressing.

An example of the fourth problem:

```
 FUNCTION SUM(A, N)
 DIMENSION A(N)
 SUM = 0.0
 DO I = 1, N
  SUM = SUM + A(I)
 ENDDO
 END
```

Message:

```
vect11.f:4: note: not vectorized:
 unsupported use in stmt.
vect11.f:4: note: vectorized 0
 loops in function.
```

This is a reduction - this should have been supported when all stage 2 projects of version 4.2 were included in mainline.

6

7.5.3 Indirect addressing

`verint`, the top routine in the gprof break-down, has loops to do linear, quadratic and cubic interpolation of atmospheric quantities to the departure point of air parcels arriving in the grid box under consideration. For simplicity, (a slightly edited version of) the linear interpolation loop is shown:

```
C  LINEAR INTERPOLATION

 DO JY = KLAT1,KLAT2
 DO JX = KLON1,KLON2
  IDX  = KP(JX,JY)
  IDY  = KQ(JX,JY)
  ILEV = KR(JX,JY)

  PRES(JX,JY) = PGAMA(JX,JY,1)*(

+  PBETA(JX,JY,1)*(PALFA(JX,JY,1)
+     *PARG(IDX-1,IDY-1,ILEV-1)
+                 +PALFA(JX,JY,2)
+     *PARG(IDX  ,IDY-1,ILEV-1))
+ +PBETA(JX,JY,2)*(PALFA(JX,JY,1)
+     *PARG(IDX-1,IDY ,ILEV-1)
+                 +PALFA(JX,JY,2)
+     *PARG(IDX  ,IDY  ,ILEV-1)))

+             + PGAMA(JX,JY,2)*(

+  PBETA(JX,JY,1)*(PALFA(JX,JY,1)
+     *PARG(IDX-1,IDY-1,ILEV  )
+                 +PALFA(JX,JY,2)
+     *PARG(IDX  ,IDY-1,ILEV  ))
+ +PBETA(JX,JY,2)*(PALFA(JX,JY,1)
+     *PARG(IDX-1,IDY ,ILEV  )
+                 +PALFA(JX,JY,2)
+     *PARG(IDX  ,IDY  ,ILEV  )))
 ENDDO
 ENDDO
```

`KP, KQ, KR` are the indices of the departure box and `PALFA, PBETA, PGAMA` are the relative distances in east-west, north-south and top-bottom in this box. This loop cannot be vectorized, because the array indices `IDX, IDY, ILEV` are not induction variables of the loop.

9 Conclusions

HIRLAM can be run when compiled with the released gfortran 4.1. The additional code reorganization and optimization features of version 4.2 do not as yet lead to significantly shorter run times.

Run time of one component (data assimilation) is significantly reduced when using the Fast Fourier Transform library fftw3, as compared to the "built in" FFT code.