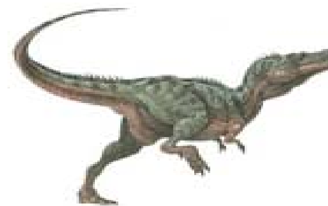
IFS Vectorisation Improvements Using Cray Supercomputer

John Hague. Cray Consultant, Sept 2014



Current System

- **2 clusters of approx 3500 Intel Ivybridge nodes each**
  - **24 cores/node**
  - **64 GB/node**
- **ECMWF's Integrated Forecast System (IFS) cycle 40r1 –> 41r1**
  - **T1279 Forecast**
  - **4D-Var T399 Minimisation**
  - **Cray Fortran Version 8.2.7 -> 8.3.1**

Intel Vector Capabilities

- AVX (available in Ivybridge) for:
  - 4-way vector - previous (IBM) system was 2-way vector
- AVX2 (available in Haswell) for:
  - Vector gather
  - FMA
- AVX512F (available in KNL and future mainstream Xeon cpus) for
  - Vector scatter
  - Vector stride > 1
  - Vector predicated division etc
  - 8-way vector

IFS Initial Vectorisation Status

- IFS T1279 forecast model is 6% faster when compiling with vectorisation (i.e. with vector2 or vector3 compilation flags)
- So approximately 6% of Fortran code running in vector mode
- Also: 15% of code is using vector version of DGEMM
- if vector operations are twice as fast in future hardware, expect a speedup of approximately (6+15)/2 =10%
- Need to get more out of vectorisation to take advantage of improved hardware vector capabilities

Will try to improve vectorisation and look at what may vectorise in the future that does not vectorise now

What can Tools do for Us

- Profilers
  - ECMWF’s Drhook: elapsed times for routines
  - Cray’s PAT (Performance Analysis Tool) CPU profiler
  - Drhook is more accurate, but PAT gives routine and statement level profiling

- The Compiler
  - Shows how code has been optimised with ”loopmark” output
  - User can interact with the compiler using flags for a whole routine, or in-code directives

- Experiments made with IFS T1279 on 100 nodes
  - 1 Day Forecast: 400 tasks with 12-way OpenMP (and hyperthreading)
  - 4D-Var minimisation; 400 tasks with 6-way OpenMP (w/o hyperthreading)

Drhook Elapsed time Profile for T1279 Forecast

% Time (self)	Self (sec)	calls	Self ms/call	Routine
4.07	12.412	6088	2.04	*CLOUDSC
3.09	9.419	1826	5.16	*LE_DGEMM
1.54	4.708	3053	1.54	*CPG
1.52	4.639	55448	0.08	*UPDATE_STATE
1.50	4.587	35183	0.13	*VERINT_DGEMM_1
1.34	4.103	172	23.86	*RADLSWR
1.30	3.972	18300	0.22	*LAITRI
1.14	3.481	2029410	0.00	*CUADJTQ
0.99	3.036	6096	0.50	*VDFMAIN
0.97	2.969	60958	0.05	*LAITLI
0.88	2.694	175	15.40	*RRTM_RTRN1A_140GP_MCICA
0.85	2.599	9138	0.28	*LASCAW
0.81	2.481	3044	0.82	*VDFOUTER
0.79	2.415	174	13.88	*SRTM_SPCVRT_MCICA
0.66	2.024	174	11.63	*SRTM_SRTM_224GP_MCICA
0.65	1.971	38976	0.05	*SRTM_REFTRA
0.60	1.820	9147	0.20	*LARCHE
0.58	1.761	504	3.49	*LTDIR_MOD
0.58	1.756	39200	0.04	*SRTM_VRTQDR
0.56	1.715	6056	0.28	*VDFEXCU
0.55	1.682	3044	0.55	*LOCAL_STATE_INI
0.54	1.654	496	3.34	*LTINV_MOD
0.53	1.604	171	9.38	*RRTM_ECRT_140GP_MCICA

15.5% → (at row 1.14 *CUADJTQ)

26.1% → (at row 0.53 *RRTM_ECRT_140GP_MCICA)

Forecast model uses approx 1200 routines out of total of about 10000

PAT cpu profile for T1279 Forecast

```
 Samp%    |     Samp  | Imb.    |   Imb.   |Group
          |           | Samp    | Samp%    | Function
 100.0%   | 39644.0   |     --  |      --  |Total
|-----------------------------------------------------------
|  27.4%  | 10864.3   |     --  |      --  |USER
||----------------------------------------------------------
||    2.6%  |   1012.7  |   26.3  |     3.8%  |cloudsc_
||    1.1%  |    439.7  |   26.3  |     8.5%  |update_state_
||    1.0%  |    398.0  |   15.0  |     5.4%  |laitri_
||    1.0%  |    386.3  |    4.7  |     1.8%  |cpg_
||    1.0%  |    383.7  |   10.3  |     3.9%  |radlswr_
||    0.8%  |    313.7  |   11.3  |     5.2%  |laitli_
||    0.7%  |    279.0  |   42.0  |    19.6%  |lascaw_
||    0.6%  |    237.0  |   17.0  |    10.0%  |srtm_spcvrt_mcica_
||    0.6%  |    225.0  |    7.0  |     4.5%  |vdfmain_
||    0.5%  |    218.0  |    7.0  |     4.7%  |srtm_reftra_
||    0.5%  |    215.7  |   56.3  |    31.1%  |cuadjtq_
||    0.5%  |    214.0  |   16.0  |    10.4%  |rrtm_rtrn1a_140gp_mcica_
||    0.5%  |    195.7  |   12.3  |     8.9%  |srtm_vrtqdr_
||    0.4%  |    154.3  |   14.7  |    13.0%  |propags2_
||    0.4%  |    149.7  |    8.3  |     7.9%  |vdfexcu_
```

- All routines compiled with module perftools loaded
- Similar to Drhook

PAT profile for CLOUDSC

```
||      2.7% |  1134.0 |    -- |      -- |cloudsc_
||||-----------------------------------------
4|||    0.0% |     8.0 |   3.0 |   40.9% |line.2657
4|||    0.1% |    47.7 |   4.3 |   12.5% |line.2658
4|||    0.0% |    21.0 |   1.0 |    6.8% |line.2659
4|||    0.0% |     5.3 |   1.7 |   35.7% |line.2660
```

```
1042. + 1-------< DO JK=NCLDTOP,KLEV

2654. + 1 f-----<   DO JM=1,NCLV
2655. + 1 f 3---<     DO JN=1,NCLV
2656. + 1 f 3 4-<       DO JL=KIDIA,KFDIA
2657.   1 f 3 4           JO=IORDER(JL,JM)
2658.   1 f 3 4           LLINDEX3(JL,JO,JN)
                            =ZSOLQA(JL,JO,JN)<0.0_JPRB
2659.   1 f 3 4           ZSINKSUM(JL,JO)=ZSINKSUM(JL,JO)
                            -ZSOLQA(JL,JO,JN)
2660.   1 f 3 4->       ENDDO
2661.   1 f 3--->     ENDDO

2683.   1 f----->   ENDDO
```

Simple Examples

- **Look at simple loops to investigate current Cray Fortran compiler (cce/8.2.7) vectorisation capabilities**
- **In order to achieve bit reproducibility of results, Cray and ECMWF have determined that all routines should be compiled with:**

  -hflex_mp=conservative –hfp1 –haddparen

- **Bit reproduciblity feature developed over serveral years in discussion with interested parties including ECMWF.**
- **"–hfp1" also sets "–hfp_trap", which inhibits the vectorisation of conditional code when that code contains operations that can raise floating point errors when given bad data (like division) or bad addresses**
- **Note**
  - "-hfp2" gives better optimisation, and disables floating point trapping
  - "–hnofp_trap" also disables floating point trapping

Simple Examples

INDIRECT ADDRESSING

Vectorisation with indirect addressing gives a "partially vectorised" message.

```
 7.              !DIR$ PREFERVECTOR
 8.   + Vpr2--< do ii=1,N
 9.     Vpr2       a(ii)=b(ix(ii))
10.     Vpr2--> enddo

ftn-6209 A loop starting at line 8 was partially vectorized
```

- **Pre-AVX2 X86 machines do not support the vectorisation of gather operations**
- **The Cray compiler loads b(ix(ii)) one scalar element at a time, packing them into a vector, and then stores the resulting vector as a single operation**

Simple Examples

INDIRECT ADDRESSING

Vectorisation with indirect addressing gives a "partially vectorised" message.

```
11.              !DIR$ IVDEP
12.              !DIR$ PREFERVECTOR
14.   + Vpr2--< do ii=1,N
15.     Vpr2      a(ix(ii))=b(ii)
16.     Vpr2--> enddo

ftn-6209 A loop starting at line 14 was partially vectorized
```

- **pre-AVX512F X86 machines do not support the vectorisation of scatter operations**
- **Tthe Cray compiler loads a vector of values, and then stores them one at a time into memory.**

Simple Examples

ASIN and ACOS

Loops containing ASIN or ACOS are only “partially vectorised”

```
    7.     Vpr2--< do ii=1,N
    8.     Vpr2       zz=sqrt(a(ii))
    9.     Vpr2       b(ii)=asin(zz)
   10.     Vpr2--> enddo

ftn-6209 A loop starting at line 7 was partially vectorized
```

- **Currently, the functions ASIN and ACOS cannot be vectorised by the compiler**
- **If the loop is split, the part(s) without ASIN or ACOS are fully vectorised**
- **Partial vectorisation may not be as good**

Simple Examples

DERIVED TYPE POINTER ARRAYS

Loops containing pointer arrays in Derived Types are only "partially vectorised"

```
 2.            use zmod
 5.            type (state) :: yom
 8. + Vpr2-< do ii=1,N
 9.   Vpr2       yom%u(ii)=zz(ii)
10.   Vpr2-> enddo
11.             end

ftn-6209 A loop starting at line
8 was partially vectorized.
```

```
1. module zmod
3. type state
4.   REAL*8, dimension(:),
                 pointer :: u
6. end type state
```

- **yom%u(1) may be more than one word away from yom%u(2)**
- **pre-AVX512F X86 hardware only supports "stride-1" stores in vector mode**
- **Full vectorisation if the "contiguous" flag is used**

Simple Examples

CONDITIONAL PROCESSING

- **Loops with conditional statements will vectorise if mathematical operations are not involved**

```
12.     Vr2---<      do jj=1,N
13.     Vr2            if(ll(jj)) then
14.     Vr2              z1(jj) = z2(jj)
15.     Vr2            endif
16.     Vr2--->      enddo
```

Simple Examples

CONDITIONAL PROCESSING

If mathematical functions are included, the loop will not vectorise because of a "potential hazard"

```
10.                !DIR$     IVDEP
11.                !DIR$     PREFERVECTOR
12.   + 1-----<        do jj=1,N
13.     1                if(ll(jj)) then
14.     1                  z1(jj) = 1.d0/z2(jj)
15.     1                endif
16.     1----->        enddo

ftn-6339 A loop starting at line 12 was not vectorized because
of a potential hazard in conditional code on line 14.
```

- Pre-AVX512F X86 hardware does not support predicated division
- Vectorisation would cause the division to be evaluated for all occurrences
  - This may cause an exception since specification of –hfp1 (default for IFS compilation) also sets fp_trap.
  - The loop does vectorise if compiled with "-hfp2" or "-hnofp_trap"

Simple Examples

CONDITIONAL PROCESSING: METHOD(1)

If divide is specifically performed for all iterations (but the result only stored conditionally), the loop vectorises

```
19.     Vr2---<       do jj=1,N
20.     Vr2             zx=1.d0/z2(jj)
22.     Vr2             if(l1(jj)) z1(jj)=zx
24.     Vr2--->       enddo
```

The code basically states that z2(jj) will not be 0 for any element – which allows full vectorisation

This is likely to be efficient if the divide is required most of the time

Simple Examples

CONDITIONAL PROCESSING: METHOD(2)

Alternatively, it is possible to create gather and scatter loops before and after a vectorisable inner loop

```
26.                 !DIR$     IVDEP
27.                 !DIR$     PREFERVECTOR
28.                       jjj=0
29.   + 1-----<           do jj=1,N
30.     1                   if(ll(jj)) then
31.     1                     jjj=jjj+1
32.     1                     ix(jjj)=jj
33.     1                     zz(jjj)=z2(jj)
34.     1                   endif
35.     1----->           enddo
36.                 !DIR$     NOFUSION
37.     Vr2---<           do jj=1,jjj
38.     Vr2                 zz(jj)=1.d0/zz(jj)
39.     Vr2--->           enddo
41.                 !DIR$     IVDEP
42.                 !DIR$     PREFERVECTOR
43.   + Vpr2--<           do jj=1,jjj
44.     Vpr2                z1(ix(jj))=zz(jj)
45.     Vpr2-->           enddo
```

Simple Examples

CONDITIONAL PROCESSING

•METHOD(2) maybe more efficient than METHOD(1) since the central loop performs only the computations (divides) which are necessary

- Scatter/Gather facilities will be available in hardware with AVX512

•When predicated computation is available, maybe neither “hand coded” method will be required

- AVX512F hardware will support predicated division etc.
- METHOD(2) maybe still useful if multiple loops use the same “gathered” data

- But note that if the original loop count is small (typically NPROMA=16 in IFS), inner loop may have very small loop count

IFS Subroutines

Compiler flags for low hanging fruit

All routines compiled with the following flags and drhook times compared for all routines

NOVECTOR

The “-O vector0” flag initially improved times for one routine (radlswr), but with the latest compiler vectorisation was OK

NOPATTERN

The “-h nopattern” flag improved times for a few routines

CONTIGUOUS

The “-h contiguous” flag was added to all subroutines in 3 subdirectories - accounting for about 70% of cpu time in the forecast model

Only two showed an improvement greater than 0.05 sec. Adding the contiguous flag to 2 additional subdirectories caused the model to crash with a SEGV

IFS Subroutines

Changing crayftn options

crayftn_wrapper used to compile specific routines with different options

```
OPTDEFAULT='-hflex_mp=conservative -hfp1 -hadd_paren'

OPT=$OPTDEFAULT

if [ $file = ’larmes.F90' ] ; then
  OPT="$OPTDEFAULT -hcontiguous”
fi

if [ $file = ’srtm_srtm_224gp_mcica_.F90' ] ; then
  OPT="$OPTDEFAULT –O nopattern”
fi

if [ $file = 'radlswr.F90' ] ; then
  OPT="$OPTDEFAULT -O vector0"
fi
```

Some IFS Subroutine Enhancements

All issues described in the Simple Examples were encountered in the IFS Subroutines

SRTM_SRTM_224GP_MCICA, RRTM_ECRT_140GP_MCICA

“nopattern” Flag prevented “pattern substitution”, which enabled vectorisation

LARMES, PHYS_ARRAYS_INI:

”contiguous” Flag improved vectorisation

LARCHE:

Splitting ASIN and ACOS loop by hand enabled all code (except ACOS and ASIN) to to be vectorised. Compiler appreciated assistance!

Some IFS Subroutine Enhancements

CUADJTQ: Conditional processing:

- Contains loops with several exponential functions.
- Loops without conditional processing vectorise well.
- For Loops with conditional processing,
  - Scatter/Gather coded by hand,
  - Central part of loop vectorised.
  - Effective because central loop contains high compute content

- For T159 60 steps, 24 tasks, 1 thread/task:

IBM		Cray		
No Vmass	Vmass	Vector0	Vector2	Hand Coded
4.496	3.848	5.808	3.270	2.838

Some IFS Subroutine Enhancements

SRTM_VRTQDR: Indexed Addressing

Loops added to remove indexed addressing where not necessary

PROPAGS2: Indexed Addressing

Gather/Scatter performed by hand, enabling central loops to be vectorised

LASCAW: Indexed Addressing

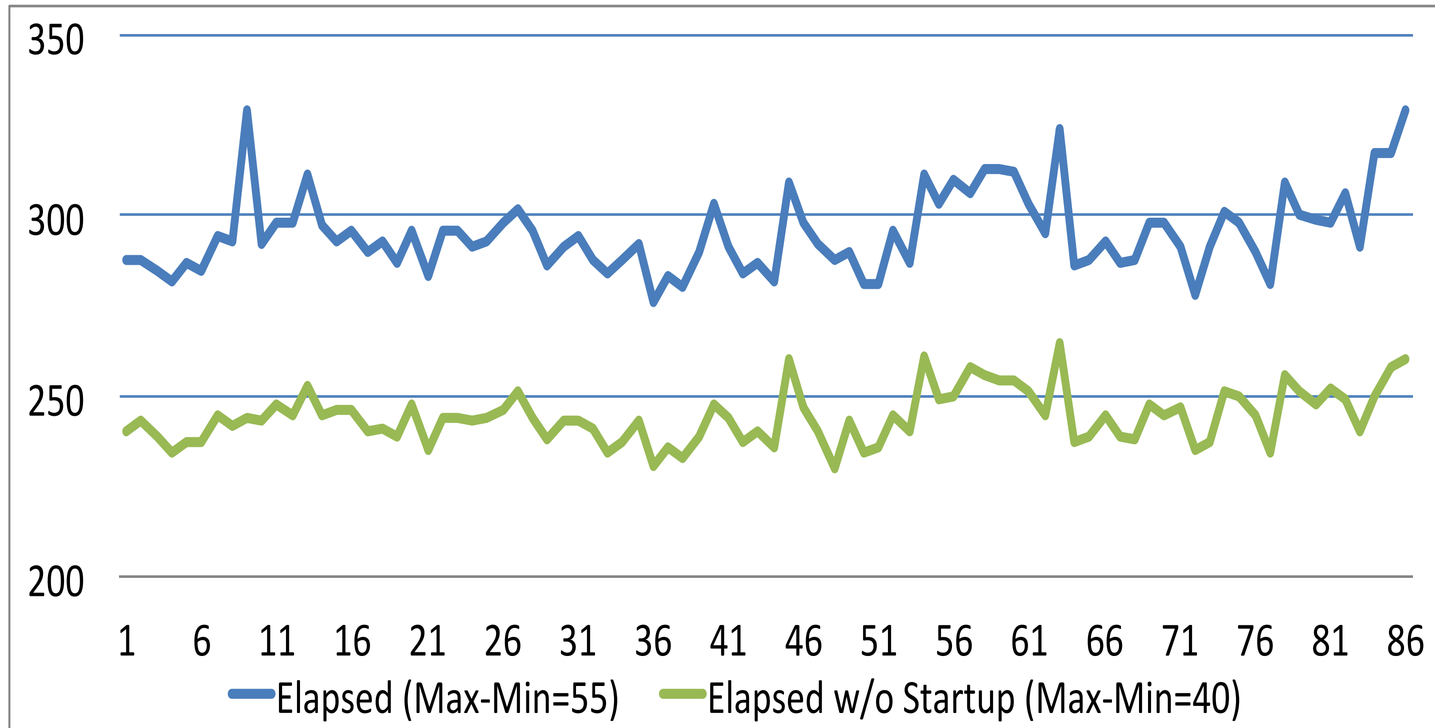
PREFERVECTOR Directive added to several loops, causing compiler to “partially vectorise” those loops

LWVDRAD: Indexed Addressing

Loops added to remove indexed addressing where not necessary

Improved fragment in test harness – but not whole routine in actual program

Elapsed times for Forecast



Large time variations, so drhook used to measure improvements

Improved Routine Times (Forecast)

	Non-tuned	Tuned	Speedup
SRTM_SRTM_224GP_MCICA	2.53	1.01	1.53
UPDATE_STATE	4.86	3.37	1.49
RRTM_ECRT_140GP_MCICA	1.98	0.68	1.30
CLOUDSC	12.90	11.70	1.20
LAITRI	4.13	3.23	0.89
LASCAW	2.71	1.91	0.80
SRTM_VRTQDR	1.87	1.24	0.63
VDFOUTER	2.60	1.84	0.76
PROPAGS2	1.71	0.96	0.75
SECSPOM	0.99	0.45	0.53
CUADJTQ	3.67	3.19	0.49
LAITLI	3.20	2.79	0.41
CPG	4.98	4.61	0.37
VDFTOFDC	0.65	0.35	0.30
LOCAL_STATE_INI	1.78	1.56	0.22
SLTEND	1.45	1.25	0.20
PHYS_ARRAYS_INI	0.70	0.58	0.12
LARCHE	1.89	1.78	0.11
LARMES	0.49	0.40	0.10

Sum of speedups > 0.05s =			12.19

Approx WALLCLOCK Time 280s

What next

Where do we go from here?

- **Only 4% extra from optimised vectorisation**
- **But more to come from improved hardware**
  - **gather/scatter capabilities**
  - **Predicated divide etc**
  - **stride > 1**
- **Would like:**
  - **Compiler to implement predicated computation**
  - **Compiler to implement scatter/gather techniques**
  - **ECMWF to avoid conditional processing wherever possible**
- **Should enable considerably more vectorisation for**
  - **Conditional Processing**
  - **indexed addressing**
- **There is a lot of scope for improvement**
  - **But beware memory access limitations**

Extra

IFS Subroutine Example

CLOUDSC: Key loop

Loops of the following type account for 30% of CLOUDSC and 3% of the total time.

Difficult to vectorise because the first and second indices of the inner arrays both depend on JL

```
2655. + 1 f 3---<      DO JN=1,NCLV
2656. + 1 f 3 4-<        DO JL=KIDIA,KFDIA
2657.   1 f 3 4            JO=IORDER(JL,JM)
2658.   1 f 3 4            LLINDEX3(JL,JO,JN)
                             =ZSOLQA(JL,JO,JN)<0.0_JPRB
2659.   1 f 3 4            ZSINKSUM(JL,JO)=ZSINKSUM(JL,JO)
                             -ZSOLQA(JL,JO,JN)
2660.   1 f 3 4->        ENDDO
2661.   1 f 3--->      ENDDO
```

IFS Subroutine Examples

CLOUDSC: Key loop

Interchanging loops removes this restriction

Loop will still not vectorise because stride > 1

But loop will “unwind” which makes it faster

```
2738. + 1 2 3---<      DO JL=KIDIA,KFDIA
2739.   1 2 3            JO=IORDER(JL,JM)
2741.   1 2 3        !DIR$ IVDEP
2742.   1 2 3        !DIR$ PREFERVECTOR
2743.   1 2 3        !DIR$ LOOP_INFO EST_TRIPS(5)
2744. + 1 2 3 w-<        DO JN=1,NCLV
2745.   1 2 3 w              LLINDEX3(JL,JO,JN)=
                               ZSOLQA(JL,JO,JN)<0.0_JPRB
2748.   1 2 3 w->        ENDDO
2749. + 1 2 3 w->        ZSINKSUM(JL,JO)=ZSINKSUM(JL,JO)
                          -SUM(ZSOLQA(JL,JO,1:NCLV))
2750.   1 2 3--->      ENDDO
```

IFS Subroutine Example

LWVDRAD: Indexed Addressing: key fragment

Inner loop uses indirect addressing: compiler vectorises on outer loop !

```
34. 1 2 Vp---< DO JA = 1, 8
35. 1 2 Vp         IJA = (JA-1)*KLEV+JKJ
36. 1 2 Vp 4-<     DO JJ=1,JN
37. 1 2 Vp 4         JL=JX(JJ)
38. 1 2 Vp 4         IJL= JL - ILOOP
39. 1 2 Vp 4         ZXN= PXDIVA5(IJL,IJA,JK)*ZTTP(JL,JA,IKJ1+1)
40. 1 2 Vp 4         ZXD= -PXNA5(IJL,IJA,JK)
                           *PXDIVA5(IJL,IJA,JK)* ZXN
41. 1 2 Vp 4         ZZ = 0.5*(PGA(JL,JA,2,JKJ)*ZXN
42. 1 2 Vp 4            +(PGB(JL,JA,2,JKJ)
                        +2.0D0*PZZA5(IJL,IJA,JK))*ZXD)
43. 1 2 Vp 4         PABCU(JL,JA,IKN)
                        = PABCU(JL,JA,IKN)+ZZ/PZZA5(IJL,IJA,JK)
44. 1 2 Vp 4         PABCU(JL,JA,IKD2)
                        = PABCU(JL,JA,IKD2)-ZZ/PZZA5(IJL,IJA,JK)
45. 1 2 Vp 4->     ENDDO
46. 1 2 Vp---> ENDDO
```

IFS Subroutine Examples

LWVDRAD: Indexed Addressing: key fragment

In many cases Indexed Addressing not necessary

Add alternative loop (Inner loop has loop count of 4)

```
31. 1 2 3----< DO JA = 1, 8
32. 1 2 3          IJA = (JA-1)*KLEV+JKJ
33. 1 2 3 Vs-<     DO JL=KIDIA,KIDIA+3
34. 1 2 3 Vs         IJL= JL - ILOOP
35. 1 2 3 Vs         ZXN= PXDIVA5(IJL,IJA,JK)*ZTTP(JL,JA,IKJ1+1)
36. 1 2 3 Vs         ZXD= -PXNA5(IJL,IJA,JK)
                           *PXDIVA5(IJL,IJA,JK)* ZXN
37. 1 2 3 Vs         ZZ = 0.5*(PGA(JL,JA,2,JKJ)*ZXN
38. 1 2 3 Vs            +(PGB(JL,JA,2,JKJ)
                        +2.0D0*PZZA5(IJL,IJA,JK))*ZXD)
39. 1 2 3 Vs         PABCU(JL,JA,IKN)
                        = PABCU(JL,JA,IKN)+ZZ/PZZA5(IJL,IJA,JK)
40. 1 2 3 Vs         PABCU(JL,JA,IKD2)
                        = PABCU(JL,JA,IKD2)-ZZ/PZZA5(IJL,IJA,JK)
41. 1 2 3 Vs->     ENDDO
42. 1 2 3----> ENDDO
```

IFS Subroutine Examples

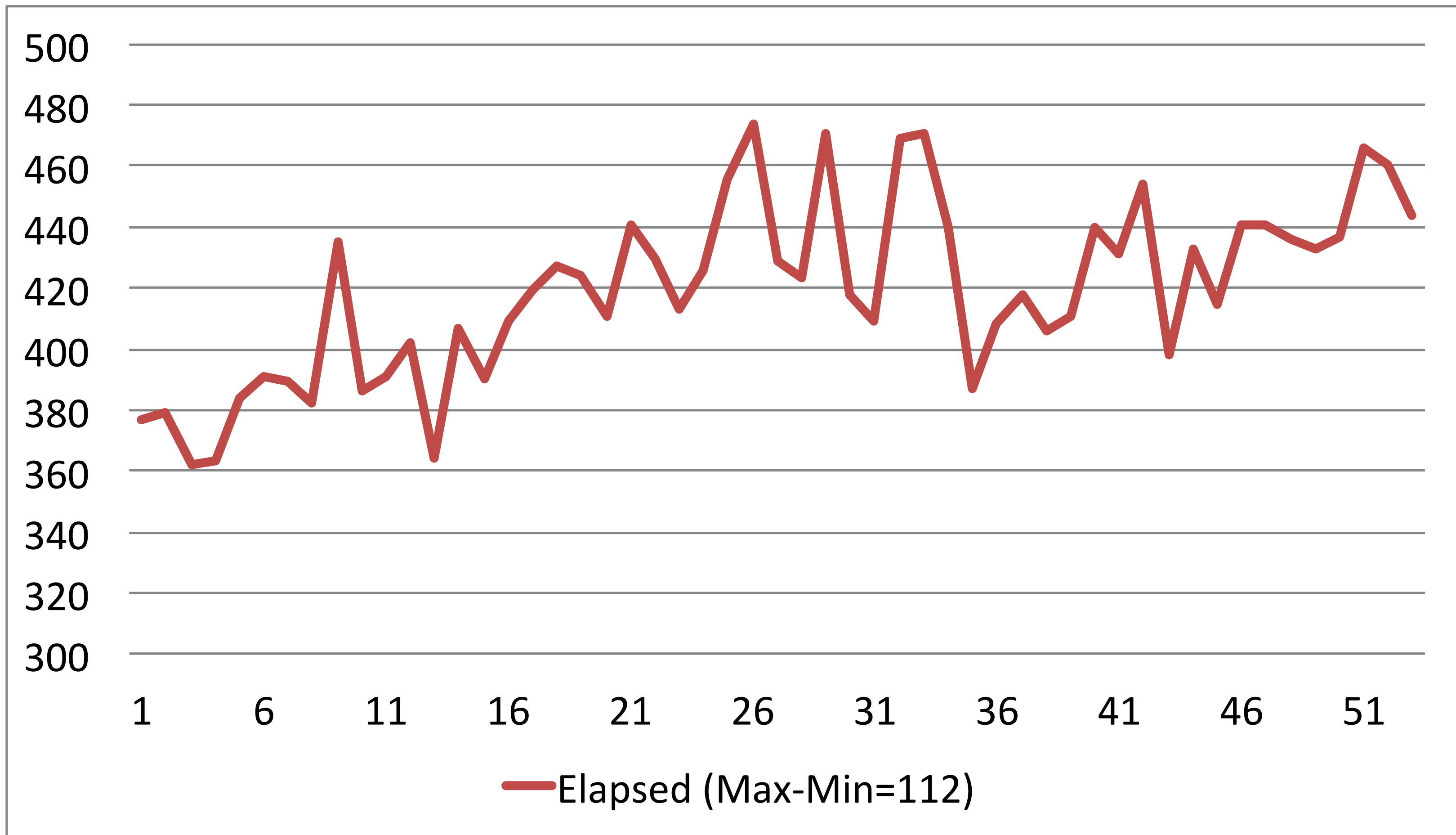
LWVDRAD: **Indexed Addressing:** Timing

Key fragment was timed with and without reallocation of arrays

```
Time for 300 calls of LWVDRAD fragment
   Indexed without reallocation:                 1790ms
   Non-indexed (vector) without reallocation: 1022ms
   Indexed with reallocation:                    2100ms
   Non-indexed (vector) with reallocation:     1478ms
```

- But time for LWVDRAD in Minimisation did not improve
- Note the storage allocation was as important as the vectorisation
- Arrays were large
  - PXDIVA5 was 4x8x137x138 = 590K double words = 4.6MB
  - More than 20 such arrays involved in LWVDRAD.
  - Arrays allocated and deallocated in a routine several calls above LWVDRAD, and occupied a different area of storage each time

Elapsed Time for 4D_Var Minimisation



Improved Routine Times (Minimisation)

	Non-tuned	Tuned	Speedup
LWVDR	20.31	13.84	6.47
BRPTOB	3.25	1.44	1.81
LWVDRTL	8.81	7.70	1.11
RTTOV_INTAVG_CHAN	1.47	0.66	0.81
LWAD	1.37	1.11	0.26
LASCAW	1.54	1.42	0.12
LAITRI	0.32	0.25	0.07

Sum of speedups > 0.5s =			10.70

Approx WALLCLOCK Time 400s