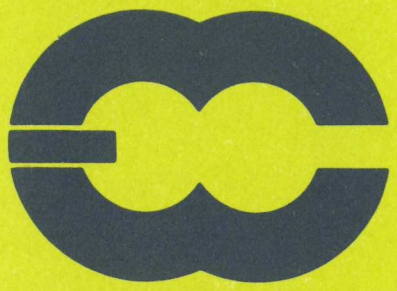

European Centre for Medium Range Weather Forecasts

Fast Fourier Transforms on Cray-1

Internal Report 21
Research Dept.

January 1979

Centre Européen pour les Prévisions Météorologiques à Moyen Terme

Europäisches Zentrum für mittelfristige Wettervorhersage

Fast Fourier Transforms on Cray-1

By

C. Temperton

European Centre for Medium Range Weather Forecasts

Shinfield Park

Internal Report No. 21

Research Department January 1979

N O T E

All timings in this report were performed on the Cray-1 Serial #9 at ECMWF. The FORTRAN compiler used was CFT 1.03 dated 3rd November, 1978.

1. Introduction

The Fast Fourier Transform (FFT) algorithm has a number of important applications in numerical weather prediction. Among these we may list the following :

(1) In gridpoint models based on a regular latitude-longitude grid, the convergence of meridians towards the poles leads to a severe restriction on the length of the timestep in order to maintain computational stability. This difficulty can be removed by Fourier filtering the higher zonal wavenumbers in the fields (or their time tendencies) near the poles (Holloway et al., 1973; Williamson, 1976; Burridge and Haseler,1977). The operational model currently envisaged for ECMWF, with 15 levels and $1\frac{1}{2}^{o}$ horizontal resolution, will require ~ 7500 real Fourier transforms of length 240 at each timestep using an explicit time integration scheme.

(2) The FFT may be used as a basis for very fast direct methods of solving discrete elliptic equations. In particular it may be used to solve the Helmholtz equations which arise at each timestep in a semi-implicit gridpoint model (Burridge and Haseler,1978) in either spherical or channel geometry. The semi-implicit version of the operational model referred to above will require ~ 15000 real Fourier transforms per timestep.

(3) The most important application is in spectral prediction models. At each timestep, the nonlinear interactions are computed by transforming certain fields to gridpoint space, carrying out the required multiplications, and then transforming back to wave-number space (e.g. Orszag,1971). To obtain a grid-

The implementation of specific problems on the vector-processing Cray-1 computer has been considered by Buzbee et al. (1977) (point relaxation methods for solving elliptic partial differential equations), and Jordan and Fong (1977) (various methods for matrix inversion and the solution of linear systems).

This report is concerned with the efficient implementation of Fast Fourier Transforms on the Cray-1. Section 2 discusses relevant features of Cray-1 architecture and vector processing. Section 3 describes the FFT algorithm. In Sections 4-6, various approaches to the problem are evaluated using Fortran FFT routines. Finally, Section 7 discusses the implementation of the FFT algorithm using CAL (Cray Assembly Language) routines.

2. The Cray-1 : a vector processing computer

This section introduces some relevant features of the Cray-1, and some of the concepts and jargon of vector processing. For further details see e.g. Johnson (1977).

The architectural feature which distinguishes the Cray-1 from other computers is the provision of eight vector registers, each capable of holding 64 words (floating-point numbers in the present application). In addition to the usual scalar capability, these provide scope for logical and arithmetic operations on vectors. (A vector in this context is just a string of numbers, not necessarily having geometrical significance).

All vector arithmetic operations are register-to-register, so the vectors must be loaded from memory into the registers before the arithmetic is performed, and the results stored back in memory afterwards. To transfer a vector between memory and a register, one first

By changing the details of the addressing, the sequence of instructions above could also handle a loop of the form:

```
  NN = 2 * N
  DO  10  I = 1,NN,2
  D(I) = A(I) + B * C (I + 1)
 10 CONTINUE
```

or even:

```
  J = 1
  K = N
  DO  10  I = 1,N
  D(I) = A(J) + B * C (K)
  J = J+3
  K=K-1
 10 CONTINUE
```

but a loop of the form :

```
  DO  10  I = 1,N
  D(INDEX (I)) = A(I) + B * C (I)
 10 CONTINUE
```

could not be handled by vector processing techniques, since the array D no longer satisfies the definition of a vector.

It is essential that the operations on each element of the vector are logically independent of those on the other elements; thus a recursive calculation of the form :

```
  DO  10  I = 2,N
  A(I) = B(I) + A(I-1)
 10 CONTINUE
```

initially free, the functional units will operate in parallel during the sequence :

```
V0  ← load from memory
V1  ← V2 + V0
V4  ← V5 * V1.
```

The second instruction will issue 9 cycles after the first (at chain slot time for a vector load), and the third will issue 8 cycles after the second (at chain slot time for a vector floating-point addition). The three instructions will then be executing in parallel.

By using the functional units either in parallel or in sequences of chained instructions, very high processing rates may be achieved. Speeds on Cray-1 are generally quoted in megaflops (millions of floating-point operations per second) rather than mips (millions of instructions per second), since one machine instruction may generate up to 64 floating-point operations. With a basic machine cycle time of 12.5 nsec (=80 million cycles per second), if two functional units can be kept continuously busy a rate of 160 megaflops will be achieved. Speeds of this order can be realised for computations such as matrix multiplication.

Execution speeds are slowed down for short vectors, since the overheads (e.g. vector start-up times) remain the same as for long vectors; however, this problem is nowhere near as serious as for other vector machines, e.g. the CDC STAR-100.

In some computations, memory references are the main bottleneck; this is particularly the case if the vectors are stored with 16 words (or less seriously, 8 words) between consecutive elements. Memory is divided into 16

(e) use functional units in parallel or in chained sequences. Requirements (a), (b), (c) and to a large extent (d) can be met with FORTRAN programs; in order to satisfy (e) it may (depending on the level of sophistication of the compiler) be necessary to use assembly language (CAL).

3. The Fast Fourier Transform

In this report we will consider only complex periodic Fourier transforms of the form :

$$z_j = \sum_{k=0}^{N-1} c_k \exp(2\ ijk\pi/N), \qquad 0 \le j \le N-1,$$

where z_j and c_k are complex numbers, and $i = \sqrt{-1}$. In normal meteorological applications we use real/"half-complex" transforms of the form

$$x_j = \sum_{k=0}^{N-1} c_k \exp(2\ ijk\pi/N), \qquad 0 \le j \le N-1, \qquad (1)$$

where x_j is real, and $c_{N-k} = c_k^*$ (* denotes complex conjugate). Eq. (1) is often rewritten as

$$x_j = \sum_{k=0}^{N/2} \{ a_k \cos \frac{2jk\pi}{N} + b_k \sin \frac{2jk\pi}{N} \} \qquad (2)$$

where the real coefficients a_k, b_k are simply related to the real and imaginary parts of c_k. The transforms (1) and (2) are usually handled by adding a pre- or post-processing step to a complex transform of length N/2 (Cooley, Lewis and Welch, 1970). This extra step is readily vectorized. An alternative procedure suggested by Bergland (1968) leads to an algorithm whose structure is very similar to that of the complex FFT.

For simplicity we assume that the complex data in A are stored contiguously, with real and imaginary parts occupying alternate words of memory. C is a work space of the same size as A. Then the passes through the data are made as follows :

```
      COMPLEX A(N), C(N), W(N)
      INTEGER IFAC(NFAX)
      LA = 1
      DO  10  I = 1,NFAX
      CALL PASS (A,C,W,IFAC(I),N,LA)
      LA = LA * IFAC (I)
      exchange rôles of arrays A & C
   10 CONTINUE
```

The variable LA is equal to the product of all factors of N used in previous passes, and controls the indexing within each pass.

The subroutine PASS contains a nested loop for each permitted value of IFAC(I); on p. 14 we outline the code required for IFAC(I)=2.

For any value of IFAC, each pass through the inner loop references IFAC complex values in each of the arrays A and C. If K > 1, then (IFAC-1) of the values in C are multiplied by complex numbers (phase factors) from the array W.

The Cooley-Tukey/Gentleman-Sande versions of the FFT have the same loop structure, but the indexing is different. Notice that the IF statement within the inner loop removes redundant but harmless multiplications, since W(1)=1. Also, some Fortran compilers may generate more efficient code if the complex arithmetic statements are expanded into their real equivalents. For a typical scalar

(b) the increment for the storage of vectors in memory is 1 complex = 2 real words, so there will be no memory bank conflicts;

(c) redundant multiplications by W(1) have been removed by providing the extra (DO 10) loop;

(d) the vector length for the inner loops is LA,which will be 1 for the first pass, increasing with successive passes.

With reference to point (d), it can be seen that the order in which the factors of N are used will influence the vector length LA. For example, for N=120=2.3.4.5, if the factors are used in ascending order then the vector length in successive passes will be 1,2,6,24; while if they are used in descending order the vector lengths will be 1,5, 20,60.

Times and megaflop rates are given in Table 2 for complex FFT's using vectorization scheme A, for ascending and descending orderings of the factors of N. With the exception of $N=100=4.5^2$, where other considerations are evidently at work, there is clearly an advantage to be gained by using the factors in descending order.

Table 1 : Scalar FFT

N	time (μs)	megaflops
$32=4^2.2$	157	3.3
$36=4.3^2$	180	4.3
$48=4^2.3$	235	4.2
$50=5^2.2$	275	5.1
$64=4^3$	302	4.2
$96=4^2.3.2$	557	4.4
$100=5^2.4$	552	5.7
$120=5.4.3.2$	743	5.1
$128=4^3.2$	736	4.3
$1024=4^5$	7477	5.0

Vectorization scheme A

```
      DO   10   L=1,LA
      C(JA) = A(IA) + A(IB)
      C(JB) = A(IA) - A(IB)
      IA = IA + 1
      IB = IB + 1
      JA = JA + 1
      JB = JB + 1
10    CONTINUE
      IF (LA. EQ.M)  RETURN
      LA1 = LA + 1
      JA = JA + JUMP
      JB = JB + JUMP
      DO   30   K = LA1, M, LA
      DO   20   L = 1,LA
      C(JA) =  A(IA) + A(IB)
      C(JB) =  W(K)* (A(IA) - A(IB))
      IA = IA + 1
      IB = IB + 1
      JA = JA + 1
      JB = JB + 1
20    CONTINUE
      JA = JA + JUMP
      JB = JB + JUMP
30    CONTINUE
```

(b) -continued:
For many frequently used combinations of factors of N these increments will become multiples of 8, leading to memory bank conflicts during the later stages of the transforms;

(c) redundant multiplications by W(1)=1 have to be carried out;

(d) the vector length for the inner loops is (M/LA) = = N/(IFAC*LA), which decreases on successive passes, becoming 1 on the last pass. For Scheme B the vector lengths are maximized by using the factors of N in ascending order.

Times and megaflop rates for Scheme B are given in Table 3. Redundant complex multiplications have been ignored in the operation count, so that we have slightly redefined megaflops as "millions of useful floating-point operations per second". Comparing Tables 2 and 3, we see that Scheme B is slower than Scheme A, a consequence of the extra arithmetic, additional vector loads, and in some cases memory bank conflicts. Moreover, comparison with Table 1 shows that for the shorter transforms ($N \leq 64$) Scheme B is slower than the original scalar implementation.

Scheme B + A

Since Schemes A and B produce identical results, there is nothing to prevent us from choosing the more appropriate scheme at each stage of the transform. The optimum combination is clearly to use Scheme B initially, and to switch to Scheme A for the later stages of the transform. This combined scheme has the following properties :

(a) W(K) is only a 'vector' during the initial passes, while Scheme B is being used;

(b) memory bank conflicts will not occur provided that the switch to Scheme A is made sufficiently early;

(c) there will only be a few redundant multiplications by W(1);

(d) the vector length will be N/IFAC(1) in the first pass, falling with subsequent passes using Scheme B and then rising again after the switch to Scheme A. For example, if $N = 2^{2p}$ and the switch is made at the halfway point (i.e. p passes with Scheme B followed by p passes with Scheme A), then the minimum vector length will be $2^p = \sqrt{N}$.

The optimum ordering of the factors of N is no longer so easy to choose. In preparing Table 4, both ascending and descending orders were tried, and the faster time is reported; other orderings (where possible) might prove even faster. Again, redundant complex multiplications do not contribute towards the operation count.

With the combined vectorization scheme, the benefits of vector processing are at last becoming apparent. For N = 32 the vectorized code runs twice as fast as the scalar code, and for N = 1024 nearly seven times faster.

Scheme P has the following properties :

(a) as in Scheme B, W(K) is a'vector' within the inner loop, requiring extra vector loads;

(b) the storage increment for vector loads is 2 real words; and for vector stores is 2*IFAC real words; memory bank conflicts will only arise during vector stores when IFAC=4 (these too could be eliminated if all real parts were stored in one array, and all imaginary parts in another);

(c) redundant complex multiplications (when W(K)=1) have to be carried out. Since W(K)=1 for $1 \leq K \leq LA$, a refinement is possible (at the expense of decreased vector lengths) in which the loop is split into two separate loops, the first for $1 \leq K \leq LA$ without the multiplication and the second for $LA + 1 \leq K \leq M$ including it. In practice it was usually found worthwhile only to provide a special loop for the last pass, in which W(K)=1 for <u>all</u> K. For the very long transform N=1024 it was worth providing a special loop for the last two passes.

(d) apart from this special case, the vector length is M=N/IFAC throughout.

The major disadvantage of this form of the FFT is that the results emerge in scrambled order; in some circumstances this may be acceptable, but if for example the complex transform is used as part of a <u>real</u> transform and is to be followed by a (vectorizable) postprocessing step, then the results must be unscrambled by a <u>scalar</u> loop of the form:

```
      DO   10   I = 1,N
      A(I) = B(INDEX(I))
   10 CONTINUE
```

Table 5 : Vectorization Scheme P

N	without reordering time (μs)	without reordering Mflops	reordering time (μs)	total time (μs)	overall Mflops (including reordering)
$32=2.4^2$	68	8	15	82	6
$36=3^2.4$	75	10	17	92	8
$48=3.4^2$	81	12	22	103	10
$50=2.5^2$	92	15	23	115	12
$64=4^3$	90	14	29	119	11
$96=2.3.4^2$	124	20	43	167	15
$100=4.5^2$	132	24	45	177	18
$120=2.3.4.5$	149	25	53	203	19
$128=2.4^3$	142	22	57	199	16
$1024=4^5$	921	41	449	1371	27

6. Vectorization of multiple FFT

So far we have considered vectorization of the FFT algorithm applied to a single set of data. Although this approach is currently fashionable among theorists in the field of parallel computation (e.g. Heller,1978), it ignores one important aspect of the problem; as mentioned in the Introduction, we usually want to perform many FFT's simultaneously. Provided only that the sets of data to be transformed are separated in memory by a constant increment, the multiple FFT problem is readily vectorized in the form shown on p.25.

Here LOT is the number of transforms being performed simultaneously, and INC is the increment between them. The inner loop simply steps from one transform to the

Scheme M (multiple)

```
      M = N/IFAC
      JUMP = (IFAC-1) * LA
      IABASE = 1
      IBBASE = IABASE + M
      JABASE = 1
      JBBASE = JABASE + LA
C -----------------------------------------

      DO    30    K = 1, M,   LA
      DO    20    L = 1, LA
      IA = IABASE
      IB = IBBASE
      JA = JABASE
      JB = JBBASE
      DO    10    IJK = 1, LOT
      C(JA) = A(IA) + A(IB)
      C(JB)= W(K)*(A(IA) -A(IB))
      IA = IA + INC
      IB = IB + INC
      JA = JA + INC
      JB = JB + INC
   10 CONTINUE
      IABASE = IABASE + 1
      IBBASE = IBBASE + 1
      JABASE = JABASE + 1
      JBBASE = JBBASE + 1

   20 CONTINUE
      JABASE = JABASE + JUMP
      JBBASE = JBBASE + JUMP

   30 CONTINUE
```

of Table 6 are 10-15 times the scalar speeds given in Table 1. However, we will show in the next section that a further doubling in speed can be achieved by writing a multiple FFT routine in the assembly language CAL.

7. FFT's in CAL

Despite the development of sophisticated Fortran compilers, it is commonly found on scalar machines that a programmer who knows what is actually going on in a program, and has some knowledge of the machine's architecture, can write assembly language programs which execute considerably faster than the compiler-produced code - typically by a factor of 2. In this section it is demonstrated that the same is (currently) true on the Cray-1.

Following the experiments described in previous sections, a CAL version was written of the multiple FFT routine. Here we need only consider the innermost loops, where all the significant computation is done. Also, since the relatively small amount of addressing arithmetic can be overlapped with the vector floating-point computation, we need only consider the latter.

On p.30 we show the innermost loops in Fortran (now using real arithmetic) and CAL for the case IFAC=2, both with and without the multiplication by the phase factor W(K). Corresponding timing analyses, in terms of machine cycles, are shown on p.31 & 32 for a vector length VL=64, i.e. 64 simultaneous transforms.

In case (a) there are eight memory references (four loads and four stores), and four arithmetic instructions (all additions/subtractions). In case (b) there are again eight memory references, but ten arithmetic instructions

intermediate results are always ready by the time they are needed.

For IFAC greater than 2 the rate could in fact be improved slightly by reorganizing the loops so the first vector load is performed near the end of the preceding loop,and the last vector store near the beginning of the following loop.

The last two columns show the megaflop rates achieved by the corresponding FORTRAN loops, and the CAL: FORTRAN speed ratio, which in most cases is not far from 2.

Table 8 shows the times and megaflop rates for multiple complex FFT's implemented in CAL for M transforms performed in parallel, for M=4,16,64 and 128. Comparison with Table 6 shows that the CAL code runs ~ 2.2 times faster than FORTRAN for M=4, 2.05 times faster for M=16, and 1.9 times faster for M=64.

Timing analysis for W(K)=1

	issue	chain slot	operands free	functional units free	result free
V0 ← A(IA)	0	9		68	73
V1 ← A(IB)	68	77		136	141
V2 ← B(IA)	136	145		204	209
V4 ← V0+V1	141	149	205	209	213
V3 ← B(IB)	204	213		272	277
V5 ← V0-V1	209	217	273	277	281
V4 → C(JA)	272		336	341	
V6 ← V2+V3	277	285	341	345	349
V5 → C(JB)	341		405	410	
V7 ← V2-V3	345	353	409	413	417
V6 → D(JA)	410		474	479	
V7 → D(JB)	479		543	548	

Table 8 : FFT's in CAL

N	M = 4		M = 16		M = 64		M = 128	
	µs	Mflops	µs	Mflops	µs	Mflops	µs	Mflops
$32=2.4^2$	30	17	11	45	7	75	7	78
$36=3^2.4$	36	21	14	56	9	90	8	93
$48=3.4^2$	45	22	18	55	11	86	11	88
$50=2.5^2$	56	25	22	63	14	97	14	100
$64=4^3$	56	23	23	54	15	82	15	84
$96=2.3.4^2$	109	22	43	56	28	87	28	87
$100=4.5^2$	106	30	45	70	31	100	31	102
$120=2.3.4.5$	148	25	59	64	39	97	38	98
$128=2.4^3$	139	23	58	55	37	84	37	85
$1024=4^5$	1357	28	601	62	416	90	415	90

8. Conclusions

Fast Fourier Transforms are most efficiently implemented on Cray-1 by performing them in parallel, provided a (quite small) number of transforms can be done simultaneously. Processing rates of ~ 50 megaflops can be achieved using FORTRAN routines, and 80 - 100 megaflops in CAL.

In the classification of Jordan & Fong (1977), FFT's can be performed at "supervector" speed, i.e. the processing rate is limited only by the number of registers and/or functional units available; however, to achieve a really significant increase in speed a "super-Cray" would need more ports to memory as well as more registers and functional units.

9. Acknowledgements

The author wishes to thank David Burridge and David Dent of ECMWF for many helpful discussions on vectorization and the Cray-1, and for reviewing this report.

Appendix 2 : Radix-4 v. radix-2 transforms

FFT routines designed for scalar machines often allow 4 as a factor of N, since this reduces the operation count to 80 % - 85 % of the work required in a simple radix-2 formulation (e.g. Singleton, 1969). On the Cray-1 the radix-4 scheme has an additional advantage, namely that the number of passes through the data (and hence the number of vector loads and stores) is reduced by a factor of two. As shown in Table 9, times for the radix-4 transform are ~ 75 % of the times for the radix-2 transform. (One pass with IFAC=2 will still of course be required if N is an odd power of 2).

Table 9 : Times per transform for N a power of 2
(multiple CAL routine, 64 transforms in parallel)

N	time (μs)	
	radix-4	radix-2
16	3	4
32	7	10
64	15	22
128	37	50
256	82	114
512	193	255
1024	428	571

```
   15 CONTINUE
      IBASE=IBASE+INC1
      JBASE=JBASE+INC2
   20 CONTINUE
      IF (LA.EQ.M) RETURN
      LA1=LA+1
      JBASE=JBASE+JUMP
      DO 40 K=LA1,M,LA
      KB=K+K-2
      C1=TRIGS(KB+1)
      S1=TRIGS(KB+2)
      DO 30 L=1,LA
      I=IBASE
      J=JBASE
CDIR$ IVDEP
      DO 25 IJK=1,LOT
      C(JA+J)=A(IA+I)+A(IB+I)
      D(JA+J)=B(IA+I)+B(IB+I)
      C(JB+J)=C1*(A(IA+I)-A(IB+I))-S1*(B(IA+I)-B(IB+I))
      D(JB+J)=S1*(A(IA+1)-A(IB+I))+C1*(B(IA+I)-B(IB+I))
      I=I+INC3
      J=J+INC4
   25 CONTINUE
      IBASE=IBASE+INC1
      JBASE=JBASE+INC2
   30 CONTINUE
      JBASE=JBASE+JUMP
   40 CONTINUE
      RETURN
C
C     CODING FOR FACTOR 3
C
   50 IA=1
      JA=1
      IB=IA+IINK
      JB=JA+JINK
      IC=IB+IINK
      JC=JB+JINK
      DO 60 L=1,LA
      I=IBASE
      J=JBASE
CDIR$ IVDEP
      DO 55 IJK=1,LOT
      C(JA+J)=A(IA+I)+(A(IB+I)+A(IC+I))
      D(JA+J)=B(IA+I)+(B(IB+I)+B(IC+I))
      C(JB+J)=(A(IA+I)-0.5*(A(IB+I)+A(IC+I)))-(SIN60*(B(IB+I)-B(IC+I)))
      C(JC+J)=(A(IA+I)-0.5*(A(IB+I)+A(IC+I)))+(SIN60*(B(IB+I)-B(IC+I)))
      D(JB+J)=(B(IA+I)-0.5*(B(IB+I)+B(IC+I)))+(SIN60*(A(IB+I)-A(IC+I)))
      D(JC+J)=(B(IA+I)-0.5*(B(IB+I)+B(IC+I)))-(SIN60*(A(IB+I)-A(IC+I)))
      I=I+INC3
      J=J+INC4
   55 CONTINUE
      IBASE=IBASE+INC1
      JBASE=JBASE+INC2
   60 CONTINUE
      IF (LA.EQ.M) RETURN
      LA1=LA+1
```

```
      C(JD+J)=(A(IA+I)-A(IC+I))+(B(IB+I)-B(ID+I))
      D(JB+J)=(B(IA+I)-B(IC+I))+(A(IB+I)-A(ID+I))
      D(JD+J)=(B(IA+I)-B(IC+I))-(A(IB+I)-A(ID+I))
      I=I+INC3
      J=J+INC4
   95 CONTINUE
      IBASE=IBASE+INC1
      JBASE=JBASE+INC2
  100 CONTINUE
      IF (LA.EQ.M) RETURN
      LA1=LA+1
      JBASE=JBASE+JUMP
      DO 120 K=LA1,M,LA
      KB=K+K-2
      KC=KB+KB
      KD=KC+KB
      C1=TRIGS(KB+1)
      S1=TRIGS(KB+2)
      C2=TRIGS(KC+1)
      S2=TRIGS(KC+2)
      C3=TRIGS(KD+1)
      S3=TRIGS(KD+2)
      DO 110 L=1,LA
      I=IBASE
      J=JBASE
CDIR$ IVDEP
      DO 105 IJK=1,LOT
      C(JA+J)=(A(IA+I)+A(IC+I))+(A(IB+I)+A(ID+I))
      D(JA+J)=(B(IA+I)+B(IC+I))+(B(IB+I)+B(ID+I))
      C(JC+J)=
     *     C2*((A(IA+I)+A(IC+I))-(A(IB+I)+A(ID+I)))
     *    -S2*((B(IA+I)+B(IC+I))-(B(IB+I)+B(ID+I)))
      D(JC+J)=
     *     S2*((A(IA+I)+A(IC+I))-(A(IB+I)+A(ID+I)))
     *    +C2*((B(IA+I)+B(IC+I))-(B(IB+I)+B(ID+I)))
      C(JB+J)=
     *     C1*((A(IA+I)-A(IC+I))-(B(IB+I)-B(ID+I)))
     *    -S1*((B(IA+I)-B(IC+I))+(A(IB+I)-A(ID+I)))
      D(JB+J)=
     *     S1*((A(IA+I)-A(IC+I))-(B(IB+I)-B(ID+I)))
     *    +C1*((B(IA+I)-B(IC+I))+(A(IB+I)-A(ID+I)))
      C(JD+J)=
     *     C3*((A(IA+I)-A(IC+I))+(B(IB+I)-B(ID+I)))
     *    -S3*((B(IA+I)-B(IC+I))-(A(IB+I)-A(ID+I)))
      D(JD+J)=
     *     S3*((A(IA+I)-A(IC+I))+(B(IB+I)-B(ID+I)))
     *    +C3*((B(IA+I)-B(IC+I))-(A(IB+I)-A(ID+I)))
      I=I+INC3
      J=J+INC4
  105 CONTINUE
      IBASE=IBASE+INC1
      JBASE=JBASE+INC2
  110 CONTINUE
      JBASE=JBASE+JUMP
  120 CONTINUE
      RETURN
C
```

```
      DO 150 L=1,LA
      I=IBASE
      J=JBASE
CDIR$ IVDEP
      DO 145 IJK=1,LOT
      C(JA+J)=A(IA+I)+(A(IB+I)+A(IE+I))+(A(IC+I)+A(ID+I))
      D(JA+J)=B(IA+I)+(B(IB+I)+B(IE+I))+(B(IC+I)+B(ID+I))
      C(JB+J)=
     *     C1*((A(IA+I)+COS72*(A(IB+I)+A(IE+I))-COS36*(A(IC+I)+A(ID+I)))
     *       -(SIN72*(B(IB+I)-B(IE+I))+SIN36*(B(IC+I)-B(ID+I))))
     *    -S1*((B(IA+I)+COS72*(B(IB+I)+B(IE+I))-COS36*(B(IC+I)+B(ID+I)))
     *       +(SIN72*(A(IB+I)-A(IE+I))+SIN36*(A(IC+I)-A(ID+I))))
      D(JB+J)=
     *     S1*((A(IA+I)+COS72*(A(IB+I)+A(IE+I))-COS36*(A(IC+I)+A(ID+I)))
     *       -(SIN72*(B(IB+I)-B(IE+I))+SIN36*(B(IC+I)-B(ID+I))))
     *    +C1*((B(IA+I)+COS72*(B(IB+I)+B(IE+I))-COS36*(B(IC+I)+B(ID+I)))
     *       +(SIN72*(A(IB+I)-A(IE+I))+SIN36*(A(IC+I)-A(ID+I))))
      C(JE+J)=
     *     C4*((A(IA+I)+COS72*(A(IB+I)+A(IE+I))-COS36*(A(IC+I)+A(ID+I)))
     *       +(SIN72*(B(IB+I)-B(IE+I))+SIN36*(B(IC+I)-B(ID+I))))
     *    -S4*((B(IA+I)+COS72*(B(IB+I)+B(IE+I))-COS36*(B(IC+I)+B(ID+I)))
     *       -(SIN72*(A(IB+I)-A(IE+I))+SIN36*(A(IC+I)-A(ID+I))))
      D(JE+J)=
     *     S4*((A(IA+I)+COS72*(A(IB+I)+A(IE+I))-COS36*(A(IC+I)+A(ID+I)))
     *       +(SIN72*(B(IB+I)-B(IE+I))+SIN36*(B(IC+I)-B(ID+I))))
     *    +C4*((B(IA+I)+COS72*(B(IB+I)+B(IE+I))-COS36*(B(IC+I)+B(ID+I)))
     *       -(SIN72*(A(IB+I)-A(IE+I))+SIN36*(A(IC+I)-A(ID+I))))
      C(JC+J)=
     *     C2*((A(IA+I)-COS36*(A(IB+I)+A(IE+I))+COS72*(A(IC+I)+A(ID+I)))
     *       -(SIN36*(B(IB+I)-B(IE+I))-SIN72*(B(IC+I)-B(ID+I))))
     *    -S2*((B(IA+I)-COS36*(B(IB+I)+B(IE+I))+COS72*(B(IC+I)+B(ID+I)))
     *       +(SIN36*(A(IB+I)-A(IE+I))-SIN72*(A(IC+I)-A(ID+I))))
      D(JC+J)=
     *     S2*((A(IA+I)-COS36*(A(IB+I)+A(IE+I))+COS72*(A(IC+I)+A(ID+I)))
     *       -(SIN36*(B(IB+I)-B(IE+I))-SIN72*(B(IC+I)-B(ID+I))))
     *    +C2*((B(IA+I)-COS36*(B(IB+I)+B(IE+I))+COS72*(B(IC+I)+B(ID+I)))
     *       +(SIN36*(A(IB+I)-A(IE+I))-SIN72*(A(IC+I)-A(ID+I))))
      C(JD+J)=
     *     C3*((A(IA+I)-COS36*(A(IB+I)+A(IE+I))+COS72*(A(IC+I)+A(ID+I)))
     *       +(SIN36*(B(IB+I)-B(IE+I))-SIN72*(B(IC+I)-B(ID+I))))
     *    -S3*((B(IA+I)-COS36*(B(IB+I)+B(IE+I))+COS72*(B(IC+I)+B(ID+I)))
     *       -(SIN36*(A(IB+I)-A(IE+I))-SIN72*(A(IC+I)-A(ID+I))))
      D(JD+J)=
     *     S3*((A(IA+I)-COS36*(A(IB+I)+A(IE+I))+COS72*(A(IC+I)+A(ID+I)))
     *       +(SIN36*(B(IB+I)-B(IE+I))-SIN72*(B(IC+I)-B(ID+I))))
     *    +C3*((B(IA+I)-COS36*(B(IB+I)+B(IE+I))+COS72*(B(IC+I)+B(ID+I)))
     *       -(SIN36*(A(IB+I)-A(IE+I))-SIN72*(A(IC+I)-A(ID+I))))
      I=I+INC3
      J=J+INC4
  145 CONTINUE
      IBASE=IBASE+INC1
      JBASE=JBASE+INC2
  150 CONTINUE
      JBASE=JBASE+JUMP
  160 CONTINUE
      RETURN
      END
```

References:

P.M. Flanders, D.J.Hunt,S.F.Reddaway & D.Parkinson 1977 "Efficient high-speed computing with the Distributed Array Processor" in Kuck et al. (1977), pp. 113-128.

D. Heller 1978 " A survey of parallel algorithms in numerical linear algebra", SIAM Review 20, pp.740-777.

J.L. Holloway, M.J. Spelman & S. Manabe 1973 " Latitude-Longitude grid suitable for numerical time integration of a global atmospheric model", Mon. Wea. Rev. 101, pp.69-78.

P.M. Johnson 1977 " An introduction to vector processing", Cray Research Inc. Publication 2240002 A

T.L. Jordan & K. Fong 1977 " Some linear algebraic algorithms and their performance on Cray-1", in Kuck et al. (1977), pp. 313-316.

D.J. Kuck, D.H. Lawrie & A.H. Sameh(editors) 1977 " High Speed Computer and Algorithm Organization", Academic Press.

S.A. Orszag 1971 " Numerical simulation of incompressible flows within simple boundaries, I: Galerkin (spectral) representations", Studies in Applied Mathematics 50, pp. 293-327.

J.M. Ortega & R.G. Voigt 1977 " Solution of partial differential equations on vector computers", Proc.1977 Army Numerical Analysis and Computers Conference (ARO Report 77-3).

EUROPEAN CENTRE FOR MEDIUM RANGE WEATHER FORECASTS

Research Department (RD)

Internal Report No. 21

No. 1 Users Guide for the GFDL Model (November 1976)

No. 2 The effect of Replacing Southern Hemispheric Analyses by Climatology on Medium Range Weather Forecasts (January 1977)

No. 3 Test of a Lateral Boundary Relaxation Scheme in a Barotropic Model (February 1977)

No. 4 Parameterization of the Surface Fluxes (February 1977)

No. 5 An Improved Algorithm for the Direct Solution of Poisson's Equation over Irregular Regions (February 1977)

No. 6 Comparative Extended Range Numerical Integrations with the ECMWF Global Forecasting Model 1 : The N24, Non-Adiabatic Experiment (March 1977)

No. 7 The ECMWF Limited Area Model (March 1977)

No. 8 A Comprehensive Radiation Scheme designed for Fast Computation (May 1977)

No. 9 Documentation for the ECMWF Grid-Point Model (May 1977)

No. 10 Numerical Tests of Parameterization Schemes at an Actual Case of Transformation of Arctic Air (June 1977)

No. 11 Analysis Error Calculations for the FGGE (June 1977)

No. 12 Normal Modes of a Barotropic Version of the ECMWF Grid-Point Model (July 1977)

No. 13 Direct Methods for the Solution of the Discrete Poisson Equation : Some Comparisons (July 1977)

No. 14 On the FACR (ℓ) Algorithm for the Discrete Poisson Equation (September 1977)

No. 15 A Routine for Normal Mode Initialization with Non-Linear Correction for a Multi-Level Spectral Model with Triangular Truncation (August 1977)

No. 16 A Channel Version of the ECMWF Grid-Point Model (December 1977)

No. 17 A Comparative Study of Some Low Resolution Explicit and Semi-Implicit Spectral Integrations (August 1978)

EUROPEAN CENTRE FOR MEDIUM RANGE WEATHER FORECASTS
Research Department (RD)

Internal Report No. 21

No. 18 Verification and storing with empirical orthogonal functions

No. 19 Documentation of the ECMWF Spectral Model

No. 20 A study of the effect of an interactive radiation scheme on a medium range forecast

No. 21 Fast Fourier Transforms on Cray-1