JOURNAL OF COMPUTATIONAL PHYSICS 75, 199–216 (1988)

A Self-sorting In-place Prime Factor Real/Half-Complex FFT Algorithm

CLIVE TEMPERTON

Division de Recherche en Prévision Numérique, Service de l'Environnement Atmosphérique, 2121 Trans-Canada Highway, Dorval, Québec, Canada H9P 1J3

Received October 24, 1986; revised April 1, 1987

A new fast Fourier transform algorithm for real or half-complex (conjugate-symmetric) input data is described. Based on the decomposition of N (the length of the transform) into mutually prime factors, the algorithm performs transforms in-place and without pre- or post-reordering of the data. With large-scale scientific computing in mind, the emphasis is on reducing the number of additions required. Compared with the best available algorithm based on specializing the conventional FFT to the real/half-complex case, the number of multiplications is also reduced by about 50%. On the Cray X-MP, a transform package based on the new algorithm runs up to 20% faster than the previous fastest available routines, besides halving the storage requirements.

1. INTRODUCTION

Recently, a form of the complex FFT (fast Fourier transform) algorithm has been introduced which is not only self-sorting (both input and output data are naturally ordered) but also in-place (no additional work space is required), besides having a lower operation count than the conventional FFT procedure. The new algorithm is a descendant of that due to Good [4], via Kolba and Parks [6], and is referred to as the PFA (prime factor algorithm) since it depends on the decomposition of N (the length of the transform) into mutually prime factors.

The self-sorting in-place form was suggested by Burrus and Eschenbacher [2] and implemented by Rothweiler [7]. Temperton [13] modified the algorithm to simplify the indexing procedure and to reduce the number of additions rather than the number of multiplications. The implementation of the new algorithm on the Cray-1 was described in [14], where it was shown to save up to 32% in CPU time compared with a conventional FFT routine (for multiple complex transforms), besides requiring only half the memory.

Most applications of the FFT in computational fluid dynamics require transforms between real data in physical space and half-complex (conjugate-symmetric) data in transform space. Thus, we need to compute both

$$z_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j \omega_N^{-jk}, \qquad 0 \leqslant k \leqslant N-1, \tag{1}$$

0021-9991/88 $3.00

and its inverse,

$$x_j = \sum_{k=0}^{N-1} z_k \omega_N^{jk}, \qquad 0 \leqslant j \leqslant N-1, \tag{2}$$

where $\omega_N = \exp(2i\pi/N)$, the data x_j are real, and the complex Fourier coefficients z_k satisfy the relationship $z_{N-k} = z_k^*$.

In an earlier paper [11], the author showed how the self-sorting mixed-radix "conventional" complex FFT [9] could be specialized to the real/half-complex case. The present paper shows how the complex FFT algorithm of [13] may be specialized in a somewhat similar way.

The development of the real/half-complex form of the new algorithm is described in Section 2. Details of the implementation are discussed in Sections 3 and 4. Section 5 presents detailed operation counts and comparisons with other real/half-complex transform algorithms. Finally, Section 6 presents the results of timing experiments on the Cray-1 and Cray X-MP.

2. Development of the New Algorithm

A simple way of specializing the prime factor algorithm to the real/half-complex case would be to use the pre- or post-processing techniques of Cooley, Lewis, and Welch [3]. For example, to implement Eq. (1) for x_j real (and N even) we could form the complex sequence

$$c_j = x_{2j} + ix_{2j+1}, \qquad 0 \leqslant j \leqslant \frac{N}{2} - 1$$

perform a complex FFT of length $N/2$ (using the PFA), and recover the coefficients z_k through a post-processing step which requires $(2.5N-6)$ real additions and $(N-4)$ real multiplications. Alternatively, to transform two independent sets of real data x_j and y_j, we could form the complex sequence

$$c_j = x_j + iy_j, \qquad 0 \leqslant j \leqslant N-1,$$

use the PFA to perform a complex FFT of length N and obtain the corresponding Fourier coefficients through another post-processing step requiring $(2N-4)$ additions. If the conventional FFT algorithm is used for the complex transforms,

the operation count per real transform is very similar for these two procedures [11]. If the PFA is used for the complex transforms, the second alternative is slightly more efficient.

Suppose then that we need to perform real to half-complex transforms of length $N = 180$ (for example, this is a typical value for current global numerical weather prediction models). Combining these transforms in pairs, using the PFA for complex transforms of length $N = 180$ together with the Cooley–Lewis–Welch post-processing step, would cost 1916 real additions and 576 real multiplications per real transform. For comparison, the conventional mixed-radix FFT specialized to the real/half-complex case by pruning redundant operations [11] takes 1714 real additions and 928 real multiplications. Thus on a computer such as the Cray-1 or Cray X-MP, where all the multiplications within the FFT can be overlapped with additions and the CPU time depends only on the number of additions and memory references [10], combining the complex PFA with post-processing will take longer than the specialized real/half-complex transform routines already available [11]. Similar reasoning would apply on a Cyber 205 [12] and on the IBM 3090 Vector Facility [1]. Within the present context, this option is therefore not worth pursuing further.

The specialization of the conventional complex FFT algorithm to the real/half-complex case [11] proceeded from the following observation: if the algorithm is applied to real input data, then about half the computation is redundant. At each stage, every complex result is either purely real or accompanied by its complex conjugate. The algorithm can thus be "pruned" to remove the redundant operations. Since the self-sorting algorithm of [11] used a work array of the same size as the input data, it was possible to store the mixture of real and complex intermediate results in an orderly manner.

If we apply the self-sorting in-place complex PFA of [13] to real input data, we again find that about half the computation is redundant, and in principle a similar pruning could be carried out. In practice however, three considerations make this option difficult. First, we would like to retain the in-place property, which reduces our freedom in designing a convenient storage pattern for the mixture of real and complex intermediate results. Second, the indexing, which depends on the Chinese Remainder Theorem [13], does not lend itself readily to this special case; it becomes very difficult to keep track of the respective locations of the real and imaginary parts of each complex number. Third, a "pruned" version of the algorithm would require a lot of additional code: each stage of the algorithm (corresponding to a particular factor N_i of N) would require code both for real transforms of length N_i and for complex transforms of length N_i. This last consideration weighed heavily since, in order to outperform the routines already available on the Cray, it would clearly be necessary to program the new algorithm in CAL (Cray Assembly Language).

Thus, pruning the algorithm in this way was rejected as a viable option even though it can be shown that the operation count is favorable. Fortunately there is another way of pruning the algorithm, with the same operation count, which

retains the self-sorting in-place properties of the complex PFA together with most of its simplicity and elegance.

To motivate the following discussion, it will be helpful first to reexpress the algorithm of [13] in terms of a matrix factorization. So, let $\bar{W}_N$ be the DFT (discrete Fourier transform) matrix of order N; element (j, k) of $\bar{W}_N$ is ω_N^{-jk}, where the rows and columns of $\bar{W}_N$ are indexed from 0 to $N-1$. Omitting the scaling factor, Eq. (1) can then be written as

$$\mathbf{z} = \bar{W}_N \mathbf{x},$$

where for the time being $\mathbf{z}$ and $\mathbf{x}$ are both assumed to be complex.

The algorithm of [13] made use of the following idea: define $\bar{W}_n^{[r]}$ to be the matrix with element (j, k) given by ω_n^{-jkr}, i.e., each element of $\bar{W}_n$ is raised to the power r. If r is mutually prime to n then $\bar{W}_n^{[r]}$ is just $\bar{W}_n$ with the rows permuted. In this case the transform defined by the matrix $\bar{W}_n^{[r]}$ is referred to in [13] as a *rotated* transform. The use of these rotated transforms permits the algorithm of [13] to be self-sorting as well as in-place, using a very simple indexing scheme. (The same indexing scheme could be used without the rotations to give an algorithm which would be in-place but not self-sorting.)

Now, suppose $N = N_1 N_2 \cdots N_k$ where the factors N_i are mutually prime. Then the algorithm of [13] is equivalent to the factorization

$$\bar{W}_N = P_N^{-1} (\bar{W}_{N_k}^{[r_k]} \times \cdots \times \bar{W}_{N_2}^{[r_2]} \times \bar{W}_{N_1}^{[r_1]}) P_N, \tag{3}$$

where $\times$ denotes the Kronecker product. P_N is a permutation matrix which maps the one-dimensional data array of length N into a k-dimensional array via the Chinese Remainder Theorem mapping [13]. The Kronecker product of rotated DFT matrices simply represents a k-dimensional DFT of this data, except that the transform in each dimension includes the effect of the rotation r_i. Finally, P_N^{-1} maps the k-dimensional array back into a one-dimensional array of results. There is no need to apply the permutations P_N and P_N^{-1} explicitly to the data; the mapping is achieved implicitly by the simple indexing scheme developed in [13].

In order to perform the short ("small-n") rotated transforms of length N_i in Eq. (3), a set of DFT modules is provided in the code for implementing the transform algorithm. The set of allowable values of N_i was $\{2, 3, 4, 5, 7, 8, 9, 16\}$, as in most published work on PFA algorithms. The required rotations can be achieved by appropriately setting the values of certain multiplier constants in the small-n DFT algorithms [13, 15].

Now, it turns out that each of the rotated DFT modules can be factorized as

$$\bar{W}_n^{[r]} = X_n V_n^{[r]}, \tag{4}$$

where the elements of $V_n^{[r]}$ are *all real numbers* and X_n is a "folding" matrix,

$$X_n = \begin{bmatrix}
1 & & & & & & & & & & \\
& 1 & & & & & & & & & i \\
& & 1 & & & & & & & i & \\
& & & \cdot & & & & & \cdot & & \\
& & & & \cdot & & & \cdot & & & \\
& & & & & 1 & i & & & & \\
& & & & & 1 & -i & & & & \\
& & & & \cdot & & & \cdot & & & \\
& & & \cdot & & & & & \cdot & & \\
& & 1 & & & & & & & -i & \\
& 1 & & & & & & & & & -i
\end{bmatrix} \qquad (n \text{ odd})$$

or

$$X_n = \begin{bmatrix}
1 & & & & & & & & & & & \\
& 1 & & & & & & & & & & i \\
& & 1 & & & & & & & & i & \\
& & & \cdot & & & & & & \cdot & & \\
& & & & \cdot & & & & \cdot & & & \\
& & & & & 1 & & i & & & & \\
& & & & & & 1 & & & & & \\
& & & & & 1 & & -i & & & & \\
& & & & \cdot & & & & \cdot & & & \\
& & & \cdot & & & & & & \cdot & & \\
& & \cdot & & & & & & & & \cdot & \\
& 1 & & & & & & & & & & -i
\end{bmatrix} \qquad (n \text{ even}).$$

Thus the "hard" part of the rotated *complex* transform of length n is contained in the *real* matrix $V_n^{[r]}$. In fact, not all the DFT modules as described in [13] were of this form; some had to be redesigned for the purposes of specializing the algorithm to the real/half-complex case. In the process, the algorithms for $N_i = 7, 8, 9, 16$ were slightly improved.

An account of the redesign procedure is given in a separate paper [15], together with a complete specification of the new DFT modules. Although the modules given in [15] are for multiplication by $W_n^{[r]}$, where element (j, k) of $W_n^{[r]}$ is ω_n^{jkr}, they are easily applied to the present case by noting that

$$\bar{W}_n^{[r]} = W_n^{[n-r]}.$$

The significance of Eq. (4) will now become apparent. If we substitute this factorization into Eq. (3), we obtain

$$\bar{W}_N = P_N^{-1}((X_{N_k} V_{N_k}^{[r_k]}) \times \cdots \times (X_{N_2} V_{N_2}^{[r_2]}) \times (X_{N_1} V_{N_1}^{[r_1]}))\, P_N. \tag{5}$$

Using the algebra of Kronecker products, Eq. (5) becomes

$$\bar{W}_N = P_N^{-1}(X_{N_k} \times \cdots \times X_{N_2} \times X_{N_1})(V_{N_k}^{[r_k]} \times \cdots \times V_{N_2}^{[r_2]} \times V_{N_1}^{[r_1]})\, P_N. \tag{6}$$

Since $P_N P_N^{-1}$ is just the identity matrix of order N, we can finally write Eq. (6) as

$$\bar{W}_N = \tilde{X}_N \tilde{V}_N \tag{7}$$

where

$$\tilde{X}_N = P_N^{-1}(X_{N_k} \times \cdots \times X_{N_2} \times X_{N_1})\, P_N \tag{8}$$

and

$$\tilde{V}_N = P_N^{-1}(V_{N_k}^{[r_k]} \times \cdots \times V_{N_2}^{[r_2]} \times V_{N_1}^{[r_1]})\, P_N. \tag{9}$$

Suppose we wish to compute $\mathbf{z} = \bar{W}_N \mathbf{x}$. Using Eq. (7), we can first calculate

$$\mathbf{y} = \tilde{V}_N \mathbf{x} \tag{10}$$

and then

$$\mathbf{z} = \tilde{X}_N \mathbf{y}. \tag{11}$$

This is true whether the input data $\mathbf{x}$ is complex or real. Now, since the V-matrices in Eq. (9) are all real, $\tilde{V}_N$ will also be real. If the input data vector $\mathbf{x}$ is complex, then the partial transform in Eq. (10) will amount to a separate transform of the real and imaginary parts of $\mathbf{x}$, which will not start to interact until Eq. (11) is implemented. If on the other hand the input data vector $\mathbf{x}$ is real, then the intermediate vector $\mathbf{y}$ will also be real and the partial transform (10) will take just half the work of the complex case. Nevertheless the "hard" part of the transform will already have been done, and it remains only to apply the Kronecker product $\tilde{X}_N$ of the folding matrices to produce the final results.

In the following section, Eqs. (10) and (11) will be called respectively the "V-stage" and the "X-stage" of the algorithm.

3. Implementation

(a) *The V-stage*

In the V-stage of the algorithm, we multiply the real input data vector $\mathbf{x}$ by the matrix $\tilde{V}_N$ to produce the real intermediate result vector $\mathbf{y}$. The factorization of the

matrix $\tilde{V}_N$ is given by Eq. (9), and we see that it has the same form as the factorization of the matrix $\bar{W}_N$ given in Eq. (3), except that the complex matrices $\bar{W}_{N_i}^{[r]}$ have been replaced by the real matrices $V_{N_i}^{[r]}$. P_N is the same permutation matrix as before, mapping the one-dimensional input data array (now real) into a k-dimensional array via the CRT. The Kronecker product of V-matrices in Eq. (9) represents a k-dimensional partial transform with real outputs. Again there is no need to apply the permutations P_N and P_N^{-1} explicitly to the data; the required mapping is achieved implicitly by the indexing scheme of [13]. Finally, algorithms for each of the "small-n" partial transforms $V_n^{[r]}$ can be obtained from the corresponding algorithms for $\bar{W}_n^{[r]} = X_n V_n^{[r]}$ developed in [15], simply by deleting the "X"-part of the algorithm specification.

The V-stage thus consists of k substages corresponding to the k factors N_i $(1 \leqslant i \leqslant k)$ of N. Each of these substages consists of N/N_i "small-n" partial transforms of length $n = N_i$. The numbers of real additions $A(N_i)$, real multiplications $M(N_i)$ and logical operations (sign bit manipulations when N_i is a power of 2) for these transforms are summarized in Table I for each of the values of N_i catered for in the prime factor algorithm. These operation counts include the latest improvements described in [15]. The output from each of the short partial transforms can overwrite the corresponding input, and as explained previously the permutation is accounted for by the indexing logic, so no extra physical data transfer is necessary. Consequently, the entire V-stage of the algorithm can be done in-place.

The total operation count for the V-stage may be obtained from the information given in Table I. If $N = N_1 N_2 \cdots N_k$, then the number of real additions is

$$A(N) = \sum_{i=1}^{k} (N/N_i)\, A(N_i)$$

while the number of real multiplications is

$$M(N) = \sum_{i=1}^{k} (N/N_i)\, M(N_i).$$

TABLE I

Real Operation Counts for Multiplication by $V_{N_i}^{[r]}$

N_i	Additions $A(N_i)$	Multiplications $M(N_i)$	Logical operations
2	2	0	0
3	4	2	—
4	6	0	1
5	12	6	—
7	24	16	—
8	20	2	2
9	32	14	—
16	58	12	3

(b) *The X-stage*

In the X-stage of the algorithm, we multiply the real intermediate result vector **y** by the matrix $\tilde{X}_N$ to produce the "half-complex" final result vector **z**. The elements of **z** satisfy the complex conjugate symmetry relationship

$$z_{N-j} = z_j^*, \qquad 0 \leqslant j \leqslant N-1, \tag{12}$$

so that it is only necessary to compute and store z_j for $0 \leqslant j \leqslant N/2$. Moreover z_0 is real, as is $z_{N/2}$ if N is even. It will be shown that a convenient storage pattern for the results, which permits in-place computation, is the following. If $z_j = a_j + ib_j$ (a_j, $b_j$real) then the coefficients will be stored in the order

$$a_0, a_1, a_2, \ldots, a_{M-1}, a_M, b_{M-1}, b_{M-2}, \ldots, b_2, b_1,$$

where $M = N/2$ is even, while $M = (N+1)/2$ and a_M is absent if N is odd. Hence each imaginary part b_j will occupy the space vacated by the corresponding redundant coefficient z_{N-j}.

As defined by Eq. (8), $\tilde{X}_N$ is the Kronecker product of the "folding" matrices X_{N_i}, $1 \leqslant i \leqslant k$ (where $N = N_1 N_2 \cdots N_k$), applied to the k-dimensional array obtained via the Chinese Remainder Theorem (CRT) mapping. To explain the implementation of this stage of the algorithm, it will be helpful to take a "geometrical" viewpoint and to consider a specific example.

Figure 1 shows in a graphical form the CRT mapping for $N = 60$ where $N_1 = 5$, $N_2 = 4$, and $N_3 = 3$. Each integer n ($0 \leqslant n \leqslant N-1$) is mapped into a triplet (n_1, n_2, n_3) ($0 \leqslant n_i \leqslant N_i - 1$, $1 \leqslant i \leqslant 3$) giving a set of coordinates in the three-dimensional array. The mapping is defined [13] by

$$n_i = n \text{ modulo } N_i, \qquad 1 \leqslant i \leqslant 3. \tag{13}$$

Notice that this mapping has a useful symmetry property. Let the coordinates of n be (n_1, n_2, n_3) and define $\bar{n} = N - n$ with coordinates $(\bar{n}_1, \bar{n}_2, \bar{n}_3)$. Then from Eq. (13) it follows that

$$\begin{aligned} &\bar{n}_i = 0 && \text{if} \quad n_i = 0 \\ &\bar{n}_i = N_i - n_i && \text{if} \quad n_i > 0 \end{aligned} \qquad 1 \leqslant i \leqslant 3. \tag{14}$$

This symmetry property, coupled with the storage pattern described above for the real and imaginary parts of the results, is the key to retaining the self-sorting and in-place properties of the complex PFA [13] when specializing it to the real/half-complex case.

In the present example, multiplication of the intermediate real vector **y** by the matrix $\tilde{X}_{60}$ is equivalent to (logically) arranging the elements of **y** in the three-dimensional array depicted in Fig. 1, then applying the "folding" matrices X_3, X_4, and X_5 along each of the coordinate directions appropriately; these three operations can be done in any order.

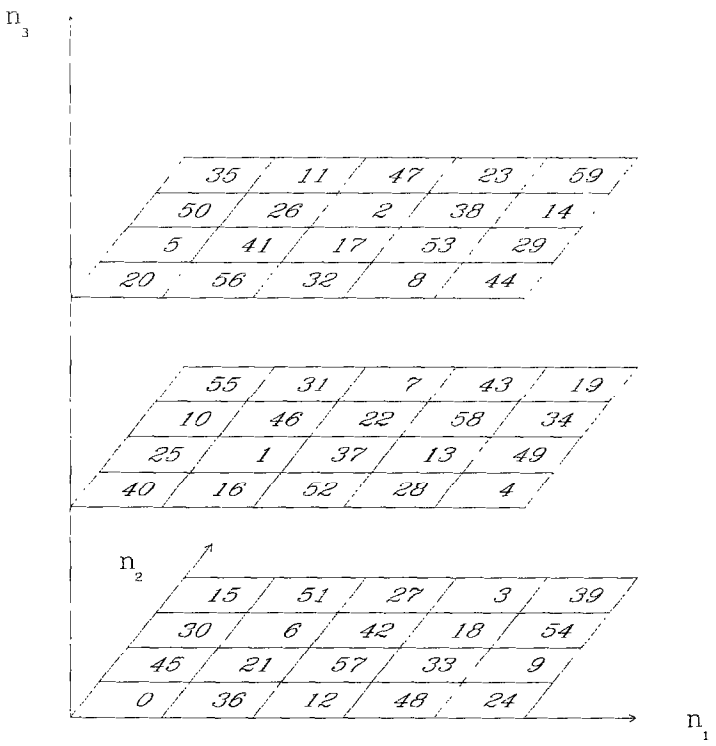

FIG. 1. The Chinese Remainder Theorem mapping for $N = 60$ ($N_1 = 5$, $N_2 = 4$, $N_3 = 3$).

We now consider the effect of each of these folding operations. This is most succinctly described by means of a diagram, as in Fig. 2. To describe the effect of each folding, we slice the three-dimensional array of Fig. 1 into planes perpendicular to the folding direction. Thus, X_3 couples the "horizontal" planes Q_1 and Q_2, leaving Q_0 unchanged. X_4 couples the planes R_1 and R_3, leaving R_0 and R_2 unchanged. X_5 couples S_1 with S_4 and S_2 with S_3, leaving S_0 unchanged.

If we applied these folding operations one after the other, starting with an array of N real numbers, we would produce a final array of N complex numbers of which half would be redundant because of conjugate-symmetry. To maintain the in-place property of the algorithm and to eliminate the redundant computation, we adopt a more subtle strategy.

First, notice that there are two points which are not changed by any of the folding operations, namely the intersections $Q_0 \cap R_0 \cap S_0$ ($n = 0$) and $Q_0 \cap R_2 \cap S_0$ ($n = 30 = N/2$). As might be anticipated, these are precisely the two elements of the final result vector $\mathbf{z}$ which are purely real, and they are already in their correct locations.

Next, consider those points which are only changed by one of the folding

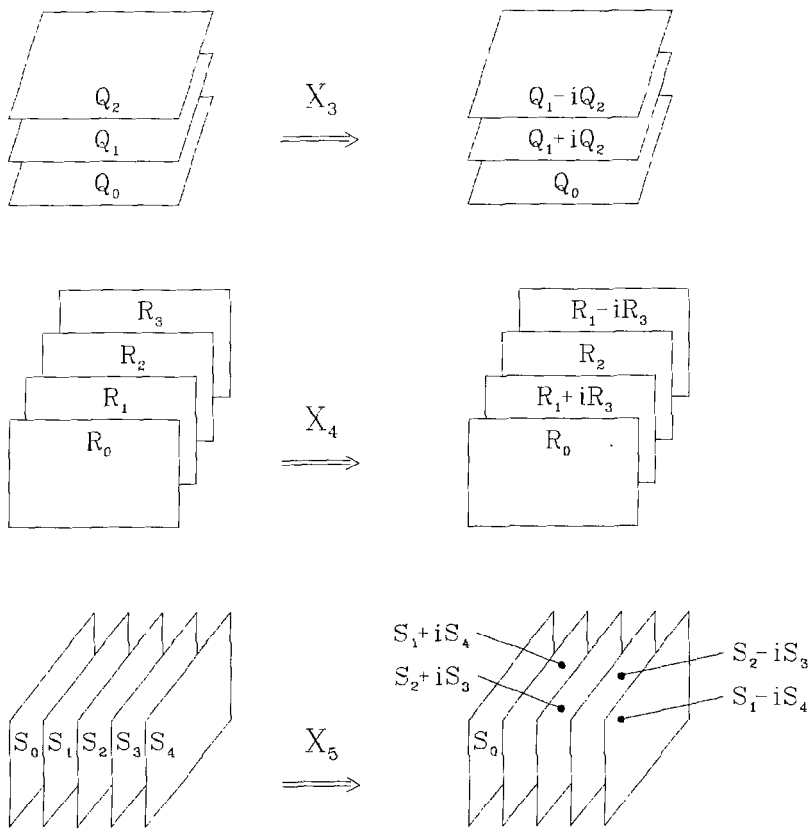

FIG. 2. The effect of applying the folding operations X_3, X_4, and X_5 in the X-stage of the algorithm for $N = 60$.

operations, for example X_5. Such points will come in pairs, e.g. (12, 48) and (36, 24) (see Fig. 1). In the first case, after the folding we will have

$$\begin{aligned} z_{12} &= y_{12} + iy_{48} \\ z_{48} &= y_{12} - iy_{48}. \end{aligned} \tag{15}$$

Since y_{12} and y_{48} are both real, $z_{48} = z_{12}^*$ as expected from Eq. (12). Hence z_{48} is redundant; instead we store the real part of z_{12} in location 12 and the imaginary part in location 48, and from Eq. (15) we see that they are already correctly located. In the second case, the folding gives

$$\begin{aligned} z_{36} &= y_{36} + iy_{24} \\ z_{24} &= y_{36} - iy_{24}. \end{aligned}$$

This case is slightly different in that z_{36} is the redundant member of the pair: to obtain the specified ordering of the results it is necessary to exchange the contents of locations 24 and 36, changing the sign of the imaginary part in the process.

The three-dimensional array of Fig. 1 contains a number of such pairs of points which are changed by only one of the folding operations. In a corresponding k-dimensional array, any such point must have $(k-1)$ of its coordinates zero, except that if one of the N_i's is even (e.g., $N_2=4$ in the present example), the corresponding coordinate may be either zero or $N_i/2$. (Note that since the N_i's must be mutually prime, at most one of them can be even.) To complete the list for $N=60$, the remaining pairs are (45, 15), (40, 20), (6, 54), (42, 18), and (10, 50). The fact that all these pairs are of the form $(n, N-n)$, which permits the computation to be done in-place, is a result of the symmetry property (14).

Now consider those points which are changed by only two of the three folding operations. Such points will come in groups of 4; e.g., in the present example the group (21, 9, 39, 51) are unchanged by X_3 but interact with each other through X_4 and X_5. Thanks to the symmetry property (14), each such group of points, forming the vertices of a rectangle in Fig. 1, is of the form $(n, n', N-n, N-n')$. By considering each such group separately, it is easy to keep track of the redundancies due to conjugate-symmetry, and to overwrite the corresponding elements of the intermediate real vector **y** with the appropriately stored elements of the half-complex result vector **z**.

Finally, consider those points which are changed by all three of the folding operations. Such points will come in groups of eight, forming the vertices of a cuboid in Fig. 1. Again the symmetry property (14) guarantees that if the point n is a vertex of the cuboid, then the opposite vertex corresponds to the point $N-n$. By treating the complete three-dimensional folding operation on each such cuboid in turn, the computation can again be done in place.

The example $N=60=5\times4\times3$ requires a three-dimensional representation of the X-stage of the algorithm. The set of DFT modules provided for the V-stage, corresponding to allowable factors N_i of N, permits up to 4 mutually prime factors. Therefore, the implementation of the X-stage also includes provision for four-dimensional folding on hypercuboids with 16 vertices.

As described earlier, the indexing for the V-stage of the algorithm is quite

TABLE II

Numbers of Vertices and Real Additions for Folding Operations

Dimension	Vertices	Additions
1 (pair)	2	0
2 (rectangle)	4	4
3 (cuboid)	8	16
4 (hypercuboid)	16	48

straightforward. No such simple solution was found for the X-stage, and it must be admitted that this is the least elegant part of the implementation. Code is provided for folding on lines, rectangles, cuboids, and hypercuboids, and for indexing purposes it was found necessary to use a precomputed address list containing the locations of the "vertices." By supplying separate input and output address lists, the same code can be used either to compute the results in-place as described above, or alternatively to write the results into another array with any chosen storage pattern.

For each category of folding, the numbers of vertices and real additions is given in Table II. To determine the contribution of the X-stage to the total operation count for a given value of N, using the information in Table II, we need to know the number of p-dimensional folding operations for each p. Explicit formulae for these numbers are given in the Appendix.

4. Inverse Transforms

In Sections 2 and 3, we have developed an algorithm to implement Eq. (1), with real input and conjugate-symmetric output. We will now sketch briefly the development of the corresponding inverse algorithm to implement Eq. (2), which may be written as

$$\mathbf{x} = W_N \mathbf{z}, \tag{16}$$

where element (j, k) of W_N is ω_N^{jk}. The algorithm of [13] is equivalent to

$$W_N = P_N^{-1}(W_{N_k}^{[r_k]} \times \cdots \times W_{N_2}^{[r_2]} \times W_{N_1}^{[r_1]})\, P_N. \tag{17}$$

As shown in [15], we can write

$$W_n^{[r]} = U_n^{[r]} X_n^T, \tag{18}$$

where the elements of $U_n^{[r]}$ are all real.

Substituting (18) into (17), we obtain the following analog of Eq. (7):

$$W_N = \tilde{U}_N \tilde{X}_N^T, \tag{19}$$

where

$$\tilde{U}_N = P_N^{-1}(U_{N_k}^{[r_k]} \times \cdots \times U_{N_2}^{[r_2]} \times U_{N_1}^{[r_1]})\, P_N. \tag{20}$$

Thus Eq. (16) is implemented by a two-stage process,

$$\mathbf{y} = \tilde{X}_N^T \mathbf{z}, \tag{21}$$

followed by

$$\mathbf{x} = \tilde{U}_N \mathbf{y}. \tag{22}$$

If the input data **z** is conjugate-symmetric, then the intermediate vector **y** is real. Implementing Eq. (21) is equivalent to inverting all the operations of the X-stage described in Section 3; the required program structure is very similar and uses the same address list for the "unfolding" operations which are now to be carried out. Implementing Eq. (22), using the decomposition of $\tilde{U}_N$ given in Eq. (20), is analogous to the V-stage of Section 3. The same indexing scheme can be used, together with the "small-n" partial transforms $U_n^{[r]}$ defined by Eq. (18). The algorithms for these partial transforms may be derived either as outlined in [15], or by "inverting" the corresponding algorithms for multiplication by $V_n^{[r]}$ used in Section 3.

For each stage of the inverse transform, the operation count is the same as for the corresponding stage of the forward transform.

5. Comparison of Operation Counts

In Table III we show the number of real additions and multiplications required to compute a real/half-complex transform of length N, for a range of values of N and for four different algorithms.

The first column of operation counts is for a "traditional" implementation in which a complex transform of length $N/2$ is followed by the post-processing step of Cooley, Lewis, and Welch [3]. The complex transform is assumed to be of the

TABLE III

Real Operation Counts (Additions/Multiplications) for Real/Half-complex Transforms of Length N

N	Conventional: complex $N/2$ + postprocessing	Conventional: specialized to real case	Winograd	PFA
60	576/288	434/232	386/68	374/112
64	540/192	408/152	—	—
120	1306/636	1016/522	920/138	896/254
128	1244/448	974/366	—	—
180	2216/1224	1714/928	1778/260	1558/496
192	2140/960	1654/694	—	—
240	2976/1512	2364/1176	2246/316	2090/628
256	2876/1152	2328/984	—	—
336	4552/2520	3692/2048	3800/478	3290/1244
360	4886/2628	3964/2212	4004/522	3564/1082
504	7418/4284	6196/3844	6672/786	5536/2062
512	6396/2560	5294/2222	—	—
1008	16352/9576	13650/8190	15892/1774	12330/4628
1024	14332/6144	12120/5464	—	—

conventional mixed-radix form as described in [9], with provision for radices (factors) 2, 3, 4, 5, and 7.

The second column is for the algorithm described in [11], obtained by taking a *complex* mixed-radix transform of length N and pruning the redundant operations. In keeping with the strategy proposed in [11], N is now allowed to have one factor of 8 and any number of factors of 6 (e.g., $180 = 5 \times 6 \times 6$ and $192 = 4 \times 6 \times 8$). This provision of factor 6 allowed some of the gains due to the "prime factor" approach to be anticipated within the context of an otherwise more conventional algorithm. Comparing the first two columns shows that in moving from a "complex $N/2$ plus postprocessing" approach to a specialized real/half-complex transform algorithm, a saving of the order of 20% is realized in the numbers of both additions and multiplications, as previously documented in [11].

While the algorithms in the first two columns can handle any N factorizable as $N = 2^p 3^q 5^r 7^s$, those in the third and fourth columns can only handle a selection of values of N with $p \leqslant 4$, $q \leqslant 2$, $r \leqslant 1$ and $s \leqslant 1$. (In principle there is no such limitation, but in practice it would require either the design and coding of very large individual DFT modules—e.g., for factors 25, 27, 32, ...— or a judicious mixture of the conventional and prime factor algorithms. The latter approach might well be worth exploring.)

The operation counts in the third column relate to a specialization of Winograd's FFT algorithm [16] to real input data. Silverman [8] provides operation counts for this case, but close inspection reveals that not all the redundant additions have been "pruned" from the complex algorithm. The operation counts have thus been recalculated on the basis of an implementation similar to that briefly suggested by Johnson and Burrus [5]. The same starting point is used as for the prime factor algorithm described in the present paper, namely the factorization of the DFT matrix $\bar{W}_N$ given by Eq. (7). The V-stage of the algorithm is then implemented by further factorizing each of the V-matrices in Eq. (9) in the form

$$V_{N_i}^{[r_i]} = A_{N_i} M_{N_i}^{[r_i]} B_{N_i}, \tag{23}$$

where the A, M, and B matrices are all real. Furthermore, all the additions are contained in the matrices A and B, while all the multiplications are contained in the diagonal matrix M (which in general will be of order greater than N_i, with A and B rectangular). Using the factorization (23), all the multiplications in the V-stage may be "nested" by rearranging the Kronecker product in Eq. (9), just as in the Winograd algorithm applied to complex data [8, 10]. The real/half-complex transform is then completed using the same X-stage as for the prime factor algorithm described in Section 3.

Comparing the operation counts in the third column of Table III with those of the first two columns, we see a dramatic decrease in the number of multiplications required (down to between 1 and 2 real multiplications per point for the whole transform). However, for some values of N this is achieved at the expense of requiring more additions than the specialized conventional algorithm of the second

column. For FFTs on most large modern scientific computers, reducing the number of *additions* is the most important consideration [10], and for this reason the Winograd approach was not selected for implementation. (Other good reasons include greater program complexity, lack of an in-place capability, and considerations of traffic between memory and registers on the Cray-1 [10], which hold equally for the complex and real/half-complex cases.)

The operation counts in the fourth colmn of Table III are for the real/half-complex prime factor algorithm as described in this paper. Comparing with the second column (which might be taken as representing the previous "state-of-the-art"), we see a large saving (over 50% in some cases) in the number of multiplications, together with a smaller but nevertheless useful saving (of order 10%) in the number of additions. In the following section we shall see how this theoretical improvement translates into a saving of CPU time in practice.

6. Timing Results

The algorithm described in this paper has been implemented in CAL (Cray Assembler Language), and run on both the Cray-1 and the Cray X-MP. It is intended for use in numerical weather prediction models, where the requirement is usually to perform many transforms of the same length simultaneously. The simplest approach to vectorization is therefore appropriate, namely to perform the transforms "in parallel" with each vector consisting of one element from each transform, and the vector length being equal to the number of transforms. (For the more difficult problem of vectorizing a *single* transform, the V-stage of Section 3 may be vectorized in the same way as described for the corresponding complex transform in [14]. For the X-stage, it would be necessary to make use of the hardware scatter/gather feature on the Cray X-MP).

The package was first developed and tested on the Cray-1. On this machine, a few parts of the code run at less than maximum efficiency because of the single path between memory and vector registers. This was regarded as only a temporary disadvantage, since the real target machine was the Cray X-MP on which the code should run at close to maximum speed.

Timing comparisons were made on both machines between the new code (RPFA) and the existing package known as FFT77, which was based on the conventional FFT specialized to the real/half-complex case by pruning redundant operations (operation counts as in column 2 of Table III). FFT77 is also implemented in CAL and vectorized in the same way as the new package. Note however that while FFT77 could handle transforms of any length of the form $N = 2^p 3^q 5^r$, no provision was included for factors of 7.

In Table IV we present times per transform for the case in which 64 transforms are performed simultaneously, and for the same set of values of N (the length of the transforms) as in Table III. The times shown are averages between those for forward and inverse transforms. On the X-MP, the times are for a single processor.

TABLE IV

Time in Microseconds for a Real/Half-complex Transform of Length N

	Cray-1		Cray X-MP	
N	FFT77	RPFA	FFT77	RPFA
60	7.64	7.31	5.71	4.76
64	7.43	—	5.35	—
120	16.2	15.0	12.3	10.4
128	16.0	—	11.8	—
180	26.8	24.8	20.5	17.3
192	26.0	—	19.4	—
240	36.6	33.5	27.7	23.3
256	38.3	—	28.2	—
336	—	52.2	—	36.4
360	62.2	51.6	46.8	37.7
504	—	80.4	—	58.5
512	84.2	—	62.2	—
1008	—	176.7	—	128.7
1024	195.8	—	142.6	—

On the Cray-1, the improvement is generally only rather modest, as expected from the considerations of memory-to-register traffic noted above. On the X-MP, the gain is consistently between 15% and 20%. In particular, if the freedom is available for a particular application, it may be worth changing from a power-of-two transform length (e.g., $N=256$) to a nearby value which can be handled by the new package (e.g., $N=240$). The ability of the new package to perform transforms in-place should also prove useful in situations where there is a shortage of available work space.

Finally, the gain on the Cray-1 and Cray X-MP is a consequence primarily of the reduction in the number of additions. Since the number of multiplications is also reduced by 50%, the algorithm should also prove useful on a wide range of other machines, from mainframes down to PCs, where the number of floating-point multiplications contributes significantly to the cost.

APPENDIX

The following formulae specify the number of folding operations in each category for the X-stage of the algorithm.

Suppose $N=N_1N_2\cdots N_m$, and let the number of p-dimensional folds be $F(p)$. We have to consider two cases.

If N is odd, then:

$$\text{(pairs):} \qquad F(1) = \sum_{i=1}^{m} (N_i - 1)/2$$

$$\text{(rectangles):} \qquad F(2) = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} (N_i - 1)(N_j - 1)/4$$

$$\text{(cuboids):} \qquad F(3) = \sum_{i=1}^{m-2} \sum_{j=i+1}^{m-1} \sum_{k=j+1}^{m} (N_i - 1)(N_j - 1)(N_k - 1)/8$$

$$\text{(hypercuboids):} \qquad F(4) = \sum_{i=1}^{m-3} \sum_{j=i+1}^{m-2} \sum_{k=j+1}^{m-1} \sum_{l=k+1}^{m} (N_i - 1)(N_j - 1) \times (N_k - 1)(N_l - 1)/16.$$

If N is even, let N_1 be the even factor. Then:

$$\text{(pairs):} \qquad F(1) = (N_1 - 2)/2 + \sum_{i=2}^{m} (N_i - 1)$$

$$\text{(rectangles):} \qquad F(2) = (N_1 - 2) \sum_{j=2}^{m} (N_j - 1)/4 + \sum_{i=2}^{m-1} \sum_{j=i+1}^{m} (N_i - 1)(N_j - 1)/2$$

$$\text{(cuboids):} \qquad F(3) = (N_1 - 2) \sum_{i=2}^{m-1} \sum_{k=j+1}^{m} (N_j - 1)(N_k - 1)/8 + \sum_{i=2}^{m-2} \sum_{j=i+1}^{m-1} \sum_{k=j+1}^{m} (N_i - 1)(N_j - 1)(N_k - 1)/4$$

$$\text{(hypercuboids):} \qquad F(4) = (N_1 - 2) \sum_{j=2}^{m-2} \sum_{k=j+1}^{m-1} \sum_{l=k+1}^{m} (N_j - 1)(N_k - 1)(N_l - 1)/16 + \sum_{i=2}^{m-3} \sum_{j=i+1}^{m-2} \sum_{k=j+1}^{m-1} \sum_{l=k+1}^{m} (N_i - 1)(N_j - 1) \times (N_k - 1)(N_l - 1)/8.$$

Acknowledgments

The author wishes to thank David Parks of Cray Research Inc. for running the tests on the Cray X-MP, Evhen Yakimiw for reviewing a first draft of this article, and Maryse Ferland for typing the manuscript.

References

1. R. C. Agarwal and J. W. Cooley, *IBM J. Res. Dev.* **30**, 145 (1986).
2. C. S. Burrus and P. W. Eschenbacher, *IEEE Trans. Acoust. Speech Signal Process.* **29**, 806 (1981).

3. J. W. COOLEY, P. A. W. LEWIS, AND P. D. WELCH, *J. Sound Vib.* **12**, 315 (1970).
4. I. J. GOOD, *J. R. Statist. Soc. Ser. B* **20**, 361 (1958).
5. H. W. JOHNSON AND C. S. BURRUS, *IEEE Trans. Acoust. Speech Signal Processing* **31**, 378 (1983).
6. D. P. KOLBA AND T. W. PARKS, *IEEE Trans. Acoustics, Speech and Signal Process.* **25**, 281 (1977).
7. J. H. ROTHWEILER, *IEEE Trans. Acoust. Speech Signal Process.* **30**, 105 (1982).
8. H. F. SILVERMAN, *IEEE Trans. Acoust. Speech Signal Process.* **25**, 152 (1977).
9. C. TEMPERTON, *J. Comput. Phys.* **52**, 1 (1983).
10. C. TEMPERTON, *J. Comput. Phys.* **52**, 198 (1983).
11. C. TEMPERTON, *J. Comput. Phys.* **52**, 340 (1983).
12. C. TEMPERTON, "Fast Fourier Transforms on the Cyber 205," in *High-Speed Computation*, edited by J. S. Kowalik (Springer-Verlag, Berlin, 1984), p. 403.
13. C. TEMPERTON, *J. Comput. Phys.* **58**, 283 (1985).
14. C. TEMPERTON, *Parallel Comput.*, in press.
15. C. TEMPERTON, *J. Comput. Phys.*, in press.
16. S. WINOGRAD, *Math. Comput.* **32**, 175 (1978).