MATHEMATICS OF COMPUTATION, VOLUME 33, NUMBER 147
JULY 1979, PAGES 977–992

Computing the Fast Fourier Transform on a Vector Computer

By David G. Korn* and Jules J. Lambiotte, Jr.

Abstract. Two algorithms are presented for performing a Fast Fourier Transform on a vector computer and are compared on the Control Data Corporation STAR-100. The relative merits of the two algorithms are shown to depend upon whether only a few or many independent transforms are desired.

A theorem is proved which shows that a set of independent transforms can be computed by performing a partial transformation on a single vector. The results of this theorem also apply to nonvector machines and have reduced the average time per transform by a factor of two on the CDC 6600 computer.

I. Introduction. The Discrete Fourier Transform of a set of N complex numbers X_k, $k = 0, 1, \ldots, N-1$, is the set of N complex numbers,

$$\alpha_j = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{-i(2\pi jk/N)}, \qquad j = 0, 1, \ldots, N-1. \tag{1.1}$$

With the definition $W_N = e^{2\pi i/N}$, Eq. (1.1) becomes

$$\alpha = \frac{1}{N}\, TX, \tag{1.2}$$

where $\alpha, X \in C^N$ and $T \in C^N \times C^N$ with $T_{jk} = W_N^{-jk}$. Also,

$$X_k = \sum_{j=0}^{N-1} \alpha_j e^{i(2\pi jk/N)}, \qquad k = 0, 1, \ldots, N-1. \tag{1.3}$$

Hence, $X = T^*\alpha$ where T^* is the conjugate transpose of T.

A straightforward evaluation of (1.2) requires $O(N^2)$ multiplications and additions. However, Cooley and Tukey [1] proposed an algorithm, now called the Fast Fourier Transform (FFT), which considerably reduces the operation count when N is a highly composite integer. In fact, when $N = 2^q$ the FFT requires $O(Nq)$ operations. There are many variants of the Cooley-Tukey FFT, but they all basically take advantage of the periodicity properties of the powers of W to factor out common quantities in the evaluation of α. Many of these algorithms do not compute α, but $R\alpha$ where, when N is a power of two, R is a permutation matrix representing the permutation mapping

Received June 5, 1978.
AMS (MOS) subject classifications (1970). Primary 68A10, Secondary 42A68.
Key words and phrases. Fast Fourier Transform, parallel computation.
*The contributions to this paper by the first author were a result of work performed under NASA Contract Number NAS1-14101 at ICASE, NASA Langley Research Center, Hampton, Virginia 23665.

σ_R such that σ_R: $j \longrightarrow i$, and i is the integer obtained by reversing the binary representation of j. In this case a bit reversal permutation is performed after the FFT has been applied to the data. It is, however, possible to maintain the natural ordering throughout the algorithm at the expense of requiring an extra storage array.

This paper considers the implementation of the FFT on a vector computer. The STAR-100 is used as the model. The requirement for efficient computation on a vector computer (see Section 2) is such that existing programs to perform the FFT cannot be used in their usual form. Two algorithms are presented. One is an adaptation and extension of an algorithm proposed by Pease [2] for transforming a single data set on a parallel computer which he describes. The other algorithm is a variant of the Cooley-Tukey algorithm specifically designed for the case when one desires to transform M independent sets of data.

Section 2 describes the characteristics of the STAR-100 computer and gives the timing for the pertinent vector instructions. Section 3 discusses the timing considerations with regard to the calculation of the FFT. The implementation based on Pease's algorithm is presented in Section 4. The algorithm designed for many transforms is described in Section 5. Timing comparisons on the STAR-100 are given in Section 6, and Section 7 contains concluding remarks.

II. Timing Considerations for the STAR-100. A vector pipeline computer such as the Control Data Corporation STAR-100 obtains its computational advantage over conventional computers when it is required to perform some arithmetical or logical operation on a vector; that is, a large set of data stored consecutively in memory. When any central processing unit performs an arithmetic operation on a pair of source operands, numerous subtasks such as exponent comparison, coefficient alignment, normalization, etc., must be carried out. On a vector computer, these subtasks are performed in an assembly line fashion on the vectors of source operands. Each segment of the pipeline performs its particular subtask on one operand pair, passes the result to the next segment, and receives the next pair of operands from the segment preceding it. The overall effect is that there is a fairly substantial time to complete the first result, called startup or prime time, but each succeeding result quickly follows since it is only one segment behind. The time to complete a vector operation on vectors of length N is given by

$$T = \rho + \Upsilon N, \tag{2.1}$$

where

ρ is the prime time (minor cycles),

Υ is the result rate (minor cycles/result),

T is the total time (minor cycles).

The values for ρ and Υ depend upon the particular vector instruction involved. Some of the more frequently used operations are listed in Table 1. The value for Υ is exact whereas the value for ρ is a lower bound. The STAR-100 minor cycle time is 40 nanoseconds.

TABLE I
STAR-100 *vector times*

Operation	ρ (Minor Cycles)	$\Upsilon(\frac{\text{Minor Cycles}}{\text{Result}})$
MULTIPLY	159	1
DIVIDE	167	2
ADD, SUBSTRACT	71	1/2
MERGE	123	3
COMPRESS	92	1
SCATTER	100	19

Equation (2.1) shows that the overall result rate (T/N) decreases from approximately ρ cycles/result when N is small to Υ cycles/result when N is very large. It is apparent, therefore, that the most efficient use of STAR occurs when N is large. However, not every algorithm expresses its computation so that it naturally involves long vector operations. Frequently, it is possible to modify an existing algorithm, which is efficient on conventional computers, in such a way that long vector operations can be used on STAR. However, sometimes it may be necessary to use an alternate or even a new algorithm which is not as efficient on a conventional computer as the algorithm it replaces but which does allow for long vector operations. The added complexity of the vector algorithm may degrade it to the point that the less complex algorithm, even with its short vector lengths, can outperform it for some problem sizes. The two algorithms discussed here exhibit these characteristics to a degree.

III. Timing Considerations for the Fast Fourier Transform. The class of algorithms we will consider for implementing the FFT consists of factoring N into factors $r_1, r_2, \ldots, r_q$ and performing q passes over the data (for base 2 algorithms $q = \log_2 N$). Each of the q steps requires $K_i N$ operations, where K_i depends on r_i and on the algorithm selected.

For a scalar machine the time for an algorithm of the type just described can be expressed as

$$T = S_0 + \sum_{i=1}^{q} (T_i N + S_i), \tag{3.1}$$

where T_i is the time required to perform the K_i operations at the ith step and the S_i are startup times—S_0 for the subroutine call and S_i, $i > 0$, for loop initializations. Typically S_i and T_i are the same order of magnitude so that we can approximate the timing for $N \gg 1$ as

$$T \sim N \sum_{i=1}^{q} T_i. \tag{3.2}$$

Finding a good algorithm means minimizing the above sum and since the T_i are monotonic functions of K_i, this means minimizing the number of operations.

For a vector machine the timing becomes more complicated. Each vector operation consists of a priming time and a rate time. It is important to perform operations with long vectors in order not to be dominated by the large priming times needed to initiate each vector operation. It is convenient to look at the average length vector used in the algorithm as a measure of the efficiency of the method. By efficiency we mean the percentage of time spent streaming vector results. The length at which a vector operation achieves a particular efficiency is larger than would be predicted from the values in Table I. This is primarily due to the scalar code which is generated by the compiler to describe the length and address of the vectors involved. The additional scalar time can be thought of as an increase in the priming time. The efficiency increases with the length of the vectors since the priming time is amortized over a large number.

Efficiency is a guide for selecting a good vector algorithm, but the total amount of computation involved must also be considered. For example, if algorithm A has longer vectors than does algorithm B but also requires more total computation than algorithm B, then the latter algorithm becomes relatively more efficient as N, the size of the problem (and vector lengths), increases. There typically is some value N_0, of N, for which algorithm A is superior for $N < N_0$ and inferior for $N > N_0$. In fact, as pointed out in Lambiotte and Voigt [3], there are examples of parallel algorithms which, in an effort to maximize vector lengths, require a higher order of computation than the serial algorithms they replace. For large enough N, such a parallel algorithm is inferior even to a scalar implementation of the serial algorithm it replaced.

For a single FFT of size N we can express the execution time as

$$T = S_0 + S_1 q + S_2 N + S_3 N q + N \sum_{i=1}^{q} T_i, \tag{3.3}$$

where S_i are times consisting of scalar and priming times and T_i are rate times. We consider only algorithms for which $S_3 = 0$, since virtually any known FFT algorithm can be coded that way. We also assume $S_0 = 0$ so that

$$T \approx S_1 q + S_2 N + N \sum_{i=1}^{q} T_i. \tag{3.4}$$

For N very large, the last term will dominate but for the range of N usually considered, $32 < N < 50{,}000$, the other terms can be important also. The Pease algorithm, discussed in the next section, has $O(q)$ vector operations of length N. Thus, S_1 reflects the priming times and is large while S_2 reflects a modest amount of scalar computation and is effectively zero. However, the requirement of the STAR-100 that vectors be in contiguous locations requires the use of vector compresses and merges. These and other extra operations increase the value of the T_i. The classical algorithms (Cooley-Tukey and Sande variants) can be programmed with average vector length $\log_2 N$ (that is, S_2 is large and S_1 is effectively zero) and as a result are inefficient for N in the range of practical interest. However, there is no need to compress or expand data sets so the T_i are smaller than in the Pease implementation.

In the majority of applications for which vector computers are appropriate, M independent transforms of size N can be taken simultaneously. If the data is stored so that the M corresponding data elements are in contiguous locations, then any scalar algorithm could be encoded using vector operations of length M in place of the corresponding scalar operations. In this case, the average vector length is M, and an algorithm which minimizes the values of T_i can be chosen. The timing for such an algorithm would be

$$T = S_0 + S_1 q + S_2 N + S_3 N q + NM \sum_{i=1}^{q} T_i. \tag{3.5}$$

However, a theorem is presented in Section 5 which shows that the FFT of M independent transforms of size N is related to the transform of one set of data of size MN in such a way that average vector lengths of size $M \log_2 N$ can be achieved without increasing the T_i in (3.5). The time for this algorithm is then

$$T \approx S_2 N + NM \sum_{i=1}^{q} T_i. \tag{3.6}$$

IV. Vector Implementation of the Pease Algorithm. We consider the evaluation of (1.3) under the assumption that only one or a few independent sets of data are to be transformed.

Pease demonstrated that when $N = 2^q$, and recalling that R is the matrix representation of the bit reversal permutation, then RX can be evaluated by factoring the $N \times N$ matrix $T'(N)$, defined by $T'(N) = RT^*(N)$, as

$$T'(N) = \prod_{k=0}^{q-1} C(N)P(N)F(N, k-1). \tag{4.1}$$

Then

$$Y = T'(N)\alpha, \tag{4.2}$$

and

$$X = RY. \tag{4.3}$$

Prior to defining the operators in (4.1), the following notation is introduced. The Kronecker product of the $N \times N$ matrix A and the $M \times M$ identity matrix, denoted $I(M)$, is the block diagonal matrix which has A as each of its M diagonal entries. It is denoted here as $A \otimes I(M)$. With these definitions, then

$$C(N) = D \otimes I(N/2),$$

where $D = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ and

$$F(N, k) = \prod_{j=0}^{k} G(N, q-j-2), \tag{4.4}$$

where for $l \geqslant 0$, $G(N, l)$ is a diagonal matrix of size N which for $i = 0$ to $N-1$ has W^{2^l} in the (i, i) position if both the first and $l+2$ bits in the binary representation of i are 1. Otherwise, a 1 goes in that position. Consequently, $G(N, l)$ is of the form

diag($I(N/2)$, $S(N/2, l)$), where $S(N/2, l)$ is also a diagonal matrix. For example,

$$G(8, 0) = \text{diag}(I(4), \text{diag}(1, 1, W, W)),$$

$$G(8, 1) = \text{diag}(I(4), \text{diag}(1, W^2, 1, W^2)).$$

For simplicity in notation, $F(N, -1) = I(N)$. Also, $P(N)$ is a permutation matrix which performs a perfect shuffle of the upper and lower halves of the vector on which it operates. Formally, the permutation σ_p, which $P(N)$ represents, is defined as σ_P: $i \longrightarrow j$ where

$$j = 2i \qquad \text{if } i < N/2,$$

$$j = 2i - N + 1 \quad \text{if } i \geqslant N/2.$$

We note for later use that $P(N)^{-1} = P^T(N)$ and σ_P^T is defined by σ_P^T: $j \longrightarrow i$ where

$$i = j/2 \qquad \text{if } j \text{ is even},$$

$$i = (N + j - 1)/2 \quad \text{if } j \text{ is odd}.$$

4.1. *Vector Implementation of the Factorization.* We consider the implementation of (4.1) and (4.2) on the STAR-100 computer. The reference to system size, N, will be dropped for convenience until later. β will refer to a vector with complex components $\beta_0, \beta_1, \ldots, \beta_{N-1}$.

(a) *Formation of $P\beta$.* As has been shown previously in Lambiotte and Voigt [3], this operation can be performed easily on STAR using the MERGE instruction on the two vectors $\beta 1 = [\beta_0, \beta_1, \ldots, \beta_{N/2-1}]$ and $\beta 2 = [\beta_{N/2}, \beta_{N/2+1}, \ldots, \beta_{N-1}]$ with a bit pattern in the order vector given by

$$[1, 0, 1, 0, \ldots, 1, 0].$$

(b) *Form $F(k)$ and Compute $F(k)\beta$.* Once $F(k)$ is formed, $F(k)\beta$ is merely a complex multiplication between the last $N/2$ diagonal elements of $F(k)$ and the corresponding elements in β. From (4.1) we see that we need $F(l)$ prior to $F(l + 1)$, and from (4.4),

$$F(l + 1) = F(l)G(q - l - 3).$$

The diagonal matrix $G(q - l - 3)$ can be formed easily by evaluating $W^{2^{q-l-3}}$ and overstoring this value into a vector of ones at the appropriate position.

(c) *Compute $C\beta$.* Because of the form of C, the computation $C\beta$ cannot be performed very efficiently on a vector computer. However, we observe that if we define $\bar{C} = P^T C P$, then

$$\bar{C} = \begin{bmatrix} I(N/2) & I(N/2) \\ I(N/2) & -I(N/2) \end{bmatrix}.$$

Therefore, letting $\beta 1$ and $\beta 2$ be the upper and lower halves of β as before, we have

$$\bar{C}\beta = [\beta 1 + \beta 2, \beta 1 - \beta 2]^T.$$

This computation can be performed with vectors of length $N/2$. Therefore, we rewrite

(4.1) as

$$T'(N) = \prod_{k=0}^{q-1} P(N)\bar{C}(N)F(N, k-1). \tag{4.5}$$

Thus, with $T'(N)$ in the form (4.5), $T'(N)\alpha$ can be evaluated using vectors which are of size N or $N/2$.

4.2. *Computation of RY.* We present two approaches to performing the bit reversal on a vector Y. The first method requires q applications of the MERGE instruction. We first define the generalized shuffle $Q(2^k, N)$. $Q(2^k, N)$ operates on the vector β by forming a new vector which consists of the first 2^k elements of $\beta 1$ followed by the first 2^k elements of $\beta 2$, followed by the second 2^k elements of $\beta 1$, etc. For example,

$$Q(2, 8)\beta = [\beta_0, \beta_1, \beta_4, \beta_5, \beta_2, \beta_3, \beta_6, \beta_7]^T$$

and can be obtained on STAR by doing a MERGE of $\beta 1$ and $\beta 2$ using the bit order vector given by $[1, 1, 0, 0, 1, 1, 0, 0]$.

We formally define $Q(2^k, N)$ by expressing i as

$$i = (r-1)2^k + t \quad \text{where } 0 \leqslant t \leqslant 2^k - 1,\ 1 \leqslant r \leqslant 2^{q-k},$$

then $Q(2^k, N)$ is the matrix representing the permutation $\sigma_{Q(k)}$: $i \longrightarrow j$ where

$$\begin{aligned} j &= 2(r-1)2^k + t && \text{if } i < N/2, \\ j &= (2r-1)2^k - N + t && \text{if } i \geqslant N/2. \end{aligned} \tag{4.6}$$

We note that $Q(1, N) = P(N)$. For later use, we observe that $Q^{-1}(2^k, N)\alpha$ is a vector composed first of all the odd groups of size 2^k from α, followed by all the even groups. On STAR this can be implemented by a COMPRESS with the bit pattern just described to obtain the odd groups followed by another COMPRESS using the logical "not" of that bit pattern to obtain the even groups.

With $Q(2^k, N)$ defined, the following theorem can be proved:

THEOREM. *The bit reversal permutation $X = RY$ can be expressed as*

$$X = \prod_{k=0}^{q-2} Q(2^k, N)Y. \tag{4.7}$$

Proof. Let $a_{q-1}2^{q-1} + a_{q-2}2^{q-2} + \cdots + a_1 2 + a_0$ be the binary expansion of i and represented by the q-tuple $[a_{q-1}, a_{q-2}, \ldots, a_1, a_0]$. It is easy to see that one could obtain the bit reversed q-tuple $j = [a_0, a_1, \ldots, a_{q-2}, a_{q-1}]$ by the following simple algorithm:

For $j = 0, 1, \ldots, q-2$

  Replace present q-tuple by doing a left end around shift involving the $q-j$ leftmost bits.

The proof of the theorem then is to show that $\sigma_{Q(k)}$: $i \longrightarrow j$ where j is obtained from i by doing a left end around shift of the leftmost $q-k$ components of the q-tuple

representing i. To show this, let

$$i = (r-1)2^k + t, \quad \text{where } 0 \leqslant t < 2^k, 1 \leqslant r < 2^{q-k},$$

so that r tells which group of size 2^k that i is in and t gives its position within that group. Then let the q-tuple for i be

$$i = [a_{q-k-1}, a_{q-k-2}, \cdots, a_1, a_0, b_{k-1}, b_{k-2}, \cdots, b_1, b_0],$$

so that the leftmost $q-k$ components are the $q-k$ tuple for $(r-1)2^k$ and the rightmost k-tuple represents t. Then

$$(r-1)2^k = a_{q-k-1}2^{q-1} + a_{q-k-2}2^{q-2} + \cdots + a_0 2^k.$$

Now, if $i < N/2$, by definition of $\sigma_{Q(k)}$, $j = 2(r-1)2^k + t$. Then,

$$2(r-1)2^k = a_{q-k-1}2^q + a_{q-k-2}2^{q-1} + \cdots + a_0 2^{k+1}. \tag{4.8}$$

However, since $i < N/2$, we have $a_{q-k-1} = 0$ so we can write

$$j = a_{q-k-2}2^{q-1} + a_{q-k-3}2^{q-2} + \cdots + a_0 2^{k+1} + a_{q-k-1}2^k + t.$$

Thus, j is represented by the q-tuple

$$j = [a_{q-k-2}, a_{q-k-3}, \cdots, a_0, a_{q-k-1}, b_{k-1}, \cdots, b_1, b_0], \tag{4.9}$$

which is the stated end around shift. Now, if $i \geqslant N/2$

$$j = (2r-1)2^k - N + t = 2(r-1)2^k + 2^k - N + t.$$

Then, as in (4.8),

$$j = a_{q-k-1}2^q + a_{q-k-2}2^{q-1} + \cdots + a_0 2^{k+1} + 2^k - N + t.$$

Since $I \geqslant N/2$, $a_{q-k-1} = 1$ so j can be rewritten

$$j = a_{q-k-2}2^{q-1} + \cdots + a_0 2^{k+1} + a_{q-k-1}2^k + t,$$

which again is represented as the q-tuple in (4.9).

The implementation of (4.7) then involves $q-1$ MERGE instructions of length N.

4.3. *Alternate Bit Reversal Implementation.* Another approach to performing the bit reversal on the vector Y is to form a vector of indices which define the permutation and then to use the STAR SCATTER instruction to disperse the elements of Y to their appropriate positions in X. One must, however, be able to generate the indexing vector, called I'_N here, for any value of N. To do this, we define $\overline{1}$ as a vector of size L containing the integer 1 in each element. Then, if one knows I'_L, the vector I'_{2L} can be generated by

$$I'_{2L} = \left[\frac{2*I'_L}{2*I'_L + \overline{1}}\right]$$

A data set such as I'_{16} can be preset initially and I'_N generated using the recursion shown.

4.4. *The FFT on M Vectors.* The algorithm defined by Eqs. (4.5) and (4.7) can be performed on STAR with vector operations of length N or $N/2$. We now generalize

the algorithm to transform M vectors at a time with vectors of length NM or $NM/2$. Let $Y^{(i)} = T'(N)\alpha^{(i)}$ for $i = 1, 2, \ldots, M$. Now, let α be the NM vector $\alpha = [\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(M)}]^T$. The M transforms can be expressed as

$$(T'(N) \otimes I(M))\alpha$$

$$= \left\{ \prod_{k=0}^{q-1} [Q(1, N) \otimes I(M)]\, [C(N) \otimes I(M)]\, [F(N, k-1) \otimes I(M)] \right\} \alpha. \tag{4.10}$$

Now, inserting $Q(N/2, NM)Q(N/2, NM)^{-1}$ between each factor in (4.10) and regrouping yields

$$\begin{aligned}(T'(N) \otimes I(M))\alpha = \Bigg\{ \prod_{k=0}^{q-1} & [(Q(1, N) \otimes I(M))Q(N/2, NM)] \\ & [Q^{-1}(N/2, NM)\,(C(N) \otimes I(M))Q(N/2, NM)] \\ & [Q^{-1}(N/2, NM)\,(F(N, k-1) \otimes I(M))Q(N/2, NM)] \\ & Q^{-1}(N/2, NM) \Bigg\} \alpha.\end{aligned} \tag{4.11}$$

Equation (4.11) can be simplified through the following identities which can be easily, though tediously, verified for N a power of 2:

Identity 1.

$$(Q(j, N) \otimes I(M))Q(N/2, NM) = Q(j, NM).$$

Identity 2.

$$Q^{-1}(N/2, NM)\,(\bar{C}(N) \otimes I(M))Q(N/2, NM) = \bar{C}(NM).$$

Identity 3.

$$Q^{-1}(N/2, NM)\,(F(N, l) \otimes I(M))Q(N/2, NM) = \bar{F}(NM, l),$$

where if we denote

$$F(N, l) = \operatorname{diag}(I(N/2), D(N/2, l)),$$

then

$$\bar{F}(NM, l) = \operatorname{diag}(I(NM/2), D(N/2, l) \otimes I(M)).$$

Inserting these three identities into (4.11) yields

$$(T'(N) \otimes I(M))\alpha = \left[\prod_{k=0}^{q-1} Q(1, NM)\bar{C}(NM)\bar{F}(NM, k)Q^{-1}(N/2, NM) \right] \alpha. \tag{4.12}$$

Similarly, (4.7) can be modified to yield

$$(R \otimes I(M)) = \prod_{k=0}^{q-2} Q(2^k, NM)Q^{-1}(N/2, NM). \tag{4.13}$$

Equations (4.12) and (4.13) are similar to (4.5) and (4.7) except that from an im-

plementation point of view, the vector lengths are of size NM and the logic must be included to apply the Q^{-1} operator q times when $M > 1$. The advantage of using vectors of length NM comes, however, at the expense of doing the extra work associated with Q^{-1}. This advantage becomes smaller as N gets larger since little is gained by increasing already large vector lengths.

V. Vector Implementation of the Stockham Algorithm. As discussed in Section 3, a straightforward implementation of a standard FFT algorithm on a vector computer would yield vectors with average length only $\log_2 N$ and, thus, be less efficient than the Pease algorithm discussed in Section 4. However, many scientific applications require the computation of more than one transform of size N. In this section a theorem is proved which indicates that M independent transforms can be computed with vectors of average length $M \log_2 N$. The particular algorithm discussed in this section avoids the time consuming bit reversal at the expense of extra storage. This idea has been attributed to Stockham in a paper by Cochran et al. [4].

The key to generating an efficient algorithm is contained in the following theorem.

THEOREM. *Let the MN complex elements, X_{jk}, $j = 0, \ldots, M-1$; $k = 0, \ldots, N-1$ for M Fourier transforms each of size N be stored by column. Let $N = \Pi_{i=1}^{q} r_i$. Then, applying the FFT algorithm to an array of size MN using M as the first factor and stopping the algorithm after q steps yields the M Fourier transforms of size N to within a scale factor.*

Proof. Let Y_μ, $\mu = 0, \ldots, MN-1$, be the one-dimensional array of x_{jk} where $\mu = kM + j$ and

$$B_\partial = \frac{1}{MN} \sum_{\mu=0}^{MN-1} Y_\mu W_{MN}^{-\mu\partial}, \qquad \partial = 0, 1, \ldots, MN-1,$$

be its transform. Now let

$$\partial = \gamma N + \nu, \quad \gamma = 0, \ldots, M-1, \ \nu = 0, \ldots, N-1.$$

Then

$$B_\partial = \frac{1}{MN} \sum_{j=0}^{M-1} \sum_{k=0}^{N-1} Y_{kM+j} W_{MN}^{-\partial(kM+j)}. \tag{5.1}$$

But

$$W_{MN}^{-kM\partial} = W_{MN}^{-kM\nu} = W_N^{-k\nu}. \tag{5.2}$$

So

$$B_\partial = \frac{1}{MN} \sum_{j=0}^{M-1} \left[\sum_{k=0}^{N-1} Y_{kM+j} W_N^{-k\nu} \right] W_{MN}^{-\partial j} = \frac{1}{MN} \sum_{j=0}^{M-1} A'_{\nu M+j} W_{MN}^{-\partial j}, \tag{5.3}$$

where

$$(5.4)\quad A'_{\nu M+j} = \sum_{k=0}^{N-1} Y_{kM+j}\, W_N^{-k\nu}, \qquad j = 0, \ldots, M-1, \nu = 0, \ldots, N-1.$$

The quantities $A'_{\nu M+j}$ are the result of the first q steps of a standard FFT algorithm and for a fixed j the N values $A'_{\nu M+j}$, $\nu = 0, \ldots, N-1$, comprise the FFT of the $j+1$st set of data to within a scale factor. Q.E.D.

This theorem contributes to an efficient implementation in two ways. First, as we now show, the average vector length is proportional to the size of the system. Let $MN = M \prod_{j=1}^{q} r_j$ and consider step j, $1 \leqslant j \leqslant q$, of the FFT process. Define $l_j = \prod_{i=1}^{j} r_i$. The vectorizable operations for a typical FFT algorithm on the STAR-100 are of the form

$$(5.5)\quad Z_p = \sum_{k=0}^{r_{j-1}} Y_k W_{l_j}^{kp}, \qquad p = 1, \ldots, l_j,$$

where Z_p and Y_k are complex vectors of length MN/l_j and W_{l_j} is a complex scalar. There are l_j such vectors at each step j. Vectors range in length from M at the last step up to MN/r_1 at the first. For $N = a^q$ the average length complex vector is $qM(a-1)/a$.

The second important factor resulting from viewing the M systems as one large system is that the FFT software for computing one FFT requires only a minimal amount of additional logic to compute the M transforms.

If we total the number of operations required to evaluate the expressions (5.5), we find that for each value of p, $r_j - 1$ complex vector additions and $r_j - 1$ vector multiplications are needed. Summing up over the q steps we find that a total of

$$(5.6)\quad \sum_{j=1}^{q} l_j(r_j - 1)$$

are required. For $N = a^q$ this yields approximately Na operations. However, certain symmetries of the sines and cosines can be exploited to decrease the number of multiplications as described by Bergland [5]. For $r_j = 2$, half of the multiplications can be eliminated, while for $r_j = 4$ only 3/8 of the multiplications need be performed. Additional multiplications can be eliminated at the first step when $r_1 = 2$ and the first two steps when $r_1 = r_2 = 2$ since the required powers of W are ± 1 and $\pm i$.

Multiplication of a complex vector by a complex scalar can be implemented with little overhead on the STAR-100 computer. Let $\alpha = a + ib$ be a complex scalar and let F be a complex vector of length L stored with real and imaginary parts alternating. To compute the complex vector αF do the following:

1. Compute aF and bF. (Two vector multiplications of length $2L$.)
2. Subtract even elements of bF from odd elements of aF to form $\mathrm{Re}(\alpha F)$. (Vector subtract of length $2L$ using a control vector.)
3. Add odd elements of bF to even elements of aF to form $\mathrm{Img}(\alpha F)$. (Vector add of length $2L$ using a control vector.)

The total number of cycles for this multiply, obtained from the timings in Section 2, is $460 + 6L$. Since four multiplies and two additions are required for each complex multiply the time would be at least $5L$ cycles even if vectors were not required to be contiguous.

For $N = 2^q$ we have $2N$ vector adds and N complex vector multiplies. Since the average length complex vector is $qM/2$, the timing in cycles can be approximated by

$$T = 620N + 4qMN. \tag{5.7}$$

The coefficient of N is a lower bound and can be expected to be significantly higher in practice. The 4 in (5.7) is an upper bound since fewer multiplications are performed the first two steps of the process.

The permutation or reverse binary ordering can be eliminated if a second array is used. At each of the q steps the array is moved from one array to the other in such a way that natural ordering is preserved at each step. This idea has been attributed to Stockham in a paper by Cochran et al. [4].

This investigation has also brought about the realization that even on a serial computer there are advantages to computing the M transforms in one pass through the subroutine. The overhead activity associated with each entry into an FFT subroutine is significant and needs to be done only once when all M transforms are performed in one call. The results of this approach on a CDC 6600 are presented in Section 6.

VI. Timing Results. Subroutines for the Pease algorithm, described in Section 4, and the Stockham algorithm, described in Section 5, were written for the STAR-100 in STAR FORTRAN. STAR FORTAN is a superset of FORTRAN with extensions to allow the programmer to use vector instructions. Tables II and III contain the STAR-100 times for the two codes for several values of M and N. As expected, the Pease algorithm is superior only for small values of M. In general, if $M > 5$ the Stockham algorithm is preferred. Table II is not extended beyond $M = 30$ since by that time the average time per transform for the Pease algorithm is essentially at its minimum and little could be gained from choosing M larger. For instance, Table II shows that the time for $N = 128$ and $M = 30$ is only 1.1 times faster than doing the transforms in three groups of $M = 10$. Several entries have been omitted from Table III because for those size problems, paging results and the I/O time dominates the CPU time.

The average time per transform for the Stockham algorithm is plotted in Figure 2. The plot indicates that the most dramatic improvements come from increases in M when M is small. In fact, when M reaches 30 to 40 the startups have been sufficiently amortized so that the method is efficient although important gains are still made as M increases.

A similar plot is contained in Figure 3 for an unvectorized version of this algorithm on the CDC-6600. It shows that the advantage of an implementation based on the theorem in Section 5 carries over to a serial computer. In fact, the overhead of the first transform, approximately 50%, can be spread over the M transforms bringing down the average time per transform significantly. The asymptotic value is reached

more quickly on the 6600 than on the STAR-100 since the overhead contribution on the serial computer is not as significant as the priming time contribution on the vector computer. The serial program was coded in FORTRAN and compiled using the FTN optimizing compiler.

TABLE II

Time (seconds) for the Pease algorithm on the STAR-100

NUMBERS OF TRANSFORMS M	SYSTEM SIZE N					
	64	128	256	512	1024	2048
1	.0011	.0016	.0027	.0050	.0094	.0185
5	.0029	.0054	.0108	.0228	.0492	.1050
10	.0048	.0095	.0194	.0440	.0961	.2080
20	.0085	.0176	.0375	.0861	.1900	.4150
30	.0120	.0257	.0552	.1280	.2840	.6200

TABLE III

Time (seconds) for the Stockham algorithm on the STAR-100

NUMBER OF TRANSFORMS M	SYSTEM SIZE N					
	64	128	256	512	1024	2048
1	.0031	.0057	.0114	.0215	.0455	.0905
5	.0033	.0062	.0126	.0250	.0502	.1013
10	.0035	.0071	.0140	.0283	.0573	.1177
30	.0045	.0094	.0194	.0410	.0855	.1501
50	.0054	.0118	.0248	.0536	.1141	—
100	.0079	.0177	.0383	.0855	—	—

When $N = 2^q$, Eq. (3.6) becomes

$$T \approx S_2 N + \gamma MNq. \tag{6.1}$$

Figure 4 plots T/MNq as a function of M. For M large enough, this value approaches γ. The upper bound on γ from Section 5 is 160 nanoseconds. But because several operations are avoided as described in that section, the actual value is lower. For the values of N of interest γ is approximately 3.5 minor cycles (140 nanoseconds) and S_2 is 950 minor cycles. The value of S_2 is higher than predicted in Section 5 primarily due to scalar code required by the algorithm and by the compiler.

A base 4 algorithm was coded and found to be slower for most values of M, $M < 100$. Savings at larger values of M were not sufficient to justify its use.

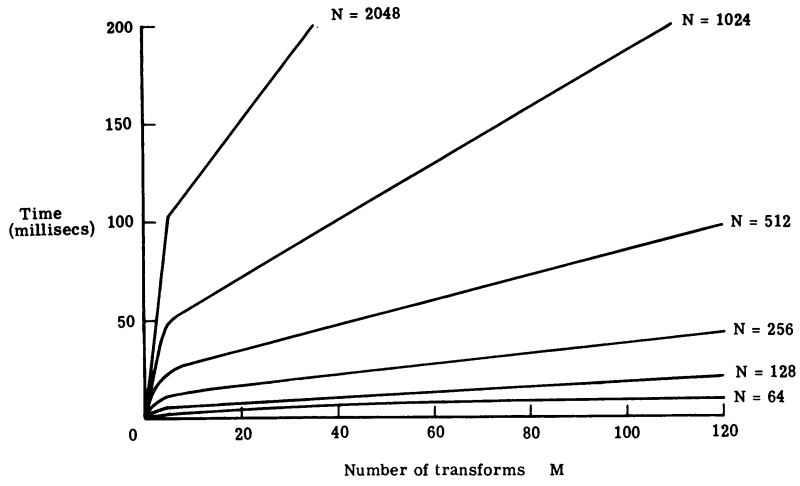

FIGURE 1

STAR-100 *complex FFT times using the minimum of the two algorithm times*

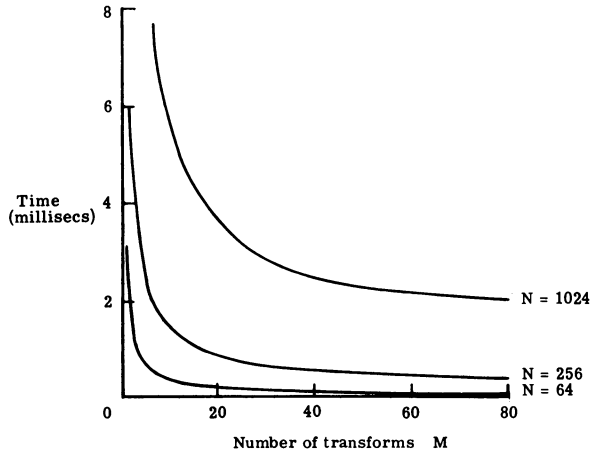

FIGURE 2

STAR-100 *time per transform for the Stockham algorithm*

In summary, the timing for the FFT for the STAR-100 can be expressed as approximately 640 $N \log_2 N$ nanoseconds for a single large transform using the Pease algorithm and 140 $N \log_2 N$ nanoseconds per transform when enough transforms are taken simultaneously. The latter figure is about 35 times faster than on a CDC 6600 and 9 times faster than the CDC CYBER 175. Since these figures are reached only asymptotically as the effect of the priming time becomes negligible, the timing for problems of practical size is greater. It is found, however, that for $N \geqslant 512$ the Pease algorithm requires less than 1100 $N \log_2 N$ nanoseconds, and for $M \geqslant 100$ and $N \geqslant 64$, the Stockham algorithm requires less than 200 $N \log_2 N$ nanoseconds per transform.

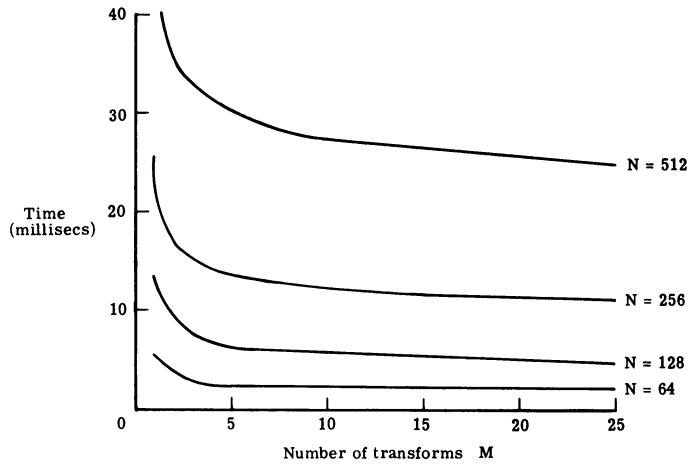

FIGURE 3

CDC-6600 *time per transform for the Stockham algorithm*

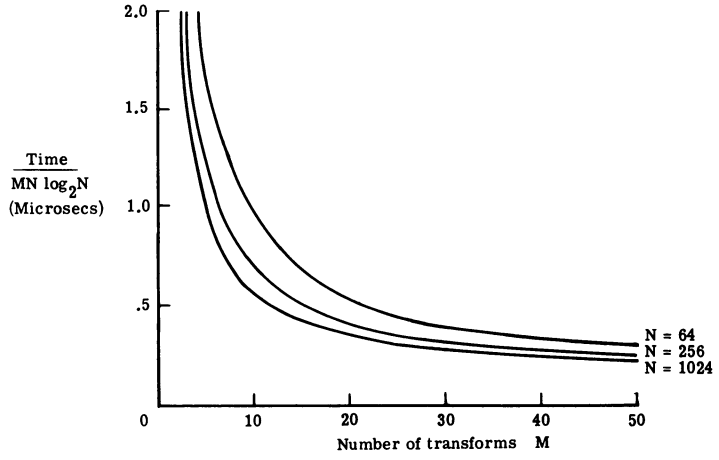

FIGURE 4

STAR-100 *Stockham algorithm timings per MN* $\log_2 N$

VII. Concluding Remarks. Two algorithms have been presented for performing the complex FFT on the STAR-100. The Pease algorithm is most attractive for a single large transform whereas the Stockham algorithm can be as much as four times faster when many independent transforms are to be computed. In this paper the Pease algorithm is extended to perform M transforms but the gain from increasing the vector lengths from N to MN is not as important as in the Stockham algorithm where the extended algorithm increases the vector lengths from $\log_2 N$ to $M \log_2 N$.

The implementation of the Stockham algorithm is based on a theorem in Section 5. This report shows that a similar implementation on a serial computer can cut running time by nearly a factor of two when twenty or more transforms are required.

It is evident that it is desirable to use the Stockham algorithm when many independent transforms are required as might occur, for instance, in finite difference solutions to partial differential equations where each column of grid data must be transformed. In such a case, the data should be stored consecutively by rows of the grid.

If the transform of M columns of real data is required, then no rearrangement of the data is necessary assuming M is even (if not, a dummy row can be added). The even columns can be taken as the imaginary part of the $M/2$ complex transforms. The M real transforms can be easily obtained from the $M/2$ complex transforms.

In some applications it may be required to transform data stored both by columns and by rows of a grid, as in the two-dimensional FFT. If the grid is stored by rows, then the Stockham algorithm can be used to transform the column data and the Pease algorithm to transform the row data.

Bell Telephone Laboratories
Holmdel, New Jersey 07733

Analysis and Computation Division
NASA Langley Research Center
Hampton, Virginia 23665

1. J. W. COOLEY & J. W. TUKEY, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, v. 19, 1965, pp.297–301.

2. M. C. PEASE, "An adaptation of the fast Fourier transform for parallel processing," *J. Assoc. Comput. Mach.*, v. 15, 1968, pp. 253–264.

3. JULES J. LAMBIOTTE, JR. & ROBERT G. VOIGT, "The solution of tridiagonal linear systems on the CDC STAR-100 computer," *ACM Trans. Math. Software*, v. 1, 1975, pp. 308–329.

4. W. T. COCHRAN ET AL., "What is the fast Fourier transform?," *IEEE Trans. Audio Electroacoustics*, v. Au-15, 1967, pp. 45–55.

5. G. D. BERGLAND, "A fast Fourier transform using base 8 iterations," *Math. Comp.*, v. 22, 1968, pp. 275–279.