Parallel Computing 5 (1987) 197–210
North-Holland

Multiprocessor FFTs

Paul N. SWARZTRAUBER *

National Center for Atmospheric Research, Boulder, CO 80307, U.S.A.

Abstract. Several multiprocessor FFTs are developed in this paper for both vector multiprocessors with shared memory and the hypercube. Two FFTs for vector multiprocessors are given that compute an ordered transform and have a stride of one except for a single 'link' step. Since multiple FFTs provide additional options for both vectorization and distribution we show that a single FFT can be performed in terms of two multiple FFTs and implement the algorithm on an ALLIANT FX/8 multiprocessor. For the hypercube the focus is on the development of algorithms that minimize interprocessor communication. On a hypercube of dimension d the unordered FFT requires $d+1$ parallel transmissions. The ordered FFT requires from $1.5d+2$ to $2d+1$ parallel transmissions depending on the length of the sequence. It is also shown that a class of orderings called index-digit permutations which includes matrix transposition, the perfect shuffle, and digit reversal can be performed with less than or equal to $1.5d$ parallel transmissions.

Keywords. Fast Fourier Transform (FFT), vector multiprocessor, distributed FFTs, multiprocessor systems, hypercube.

1. Introduction

Efficient FFTs for vector-concurrent and hypercube multiprocessors are presented in this paper. A review of eight existing FFTs is presented in the next section that differ only in the order that intermediate computations are stored in memory. Nevertheless, it is this difference that makes one or the other more suitable for a particular application or architecture. For example, the Pease FFT is efficient for pipeline or vector architectures and applications that do not require an ordered transform. The Stockham FFT is somewhat less efficient on a vector processor, but it computes an ordered transform. Like the existing FFTs, all of the multiprocessor FFTs that are developed in this paper are strictly reorderings of the Cooley–Tukey FFT. These orderings are members of a particular class called index-digit permutations that play a fundamental role in the development of multiprocessor FFTs.

There is a substantial amount of literature on the implementation of the FFT on a vector-processor, including [2,6,11,12,13]. Much of this material is applicable to the work on vector-concurrent FFTs that is presented in Section 3 and is the reason for the review that is given in Section 2. However, vectorization in the context of parallel computing is somewhat more complex. These two concepts can be at odds since the distribution of a computation may shorten the vectors in each processor. This tradeoff between vectorization and distribution is dependent on the specific architecture under consideration, and attempts to achieve a satisfactory balance may lead to the selection of quite different algorithms.

In Section 3 we develop two FFTs for a single long sequence that are efficient on a shared-memory vector-processor. They both compute an ordered transform and have vector lengths that are comparable to the Stockham FFT. However, they differ from the Stockham FFT since they access the data with a stride of one except for a single 'link' pass in which the

* This work was completed while the author was visiting the Center for Supercomputer Research and Development, University of Illinois, Urbana, IL 61801, U.S.A.

data is implicitly transposed. They differ primarily in the way that the computations are distributed. Although the second FFT is for a single sequence, it is performed in terms of two coupled multiple FFTs. This approach is motivated by the fact that multiple transforms provide options for both vectorization and distribution that are not available for the transform of a single sequence. Only transforms of long sequences or multiple sequences are considered simply because these are the only problems that consume a noticeable amount of computing resource. Section 3 ends with the implementation of the second vector-concurrent FFT on the ALLIANT FX/8 multiprocessor.

In Section 4, the focus is on hypercube FFTs. The development of several commercially available hypercubes has renewed interest in algorithms for the hypercube. Recent papers include [7,8,10] which do not specifically discuss the the FFT but are nevertheless applicable since they focus on the communication complexity of algorithms, which is also a central issue for the FFT on the hypercube. Gannon [5] also addresses this issue but in a more general setting. Fraser [4] discusses the FFT using external storage and introduces the concept of an index-digit permutation. Flanders [3] uses a similar concept in the implementation of several algorithms, including the FFT, on the ICL DAP. Literature that specifically addresses the FFT on the hypercube is sparse but does include an earlier paper by Pease [9].

Section 4 begins with several definitions including the definition of a class of orderings called index-digit permutations. This class includes all of the orderings that occur in the FFTs. In particular it includes digit reversal, the perfect shuffle and matrix transposition. The i-cycle is also defined which is central to the implementation of the FFT as well as any index-digit permutation on the hypercube. The unordered FFT requires $d+1$ parallel transmissions on a hypercube cube of dimension d. The communication complexity of the ordered FFT also depends on the length N of the sequence. If $N = 2^r$ and r is an even integer that satisfies $r/2 \leqslant d$, then $r/2 + d + 1$ parallel transmissions are required. If r is odd and $(r+1)/2 \leqslant d$, then $(r+1)/2 + d + 1$ transmissions are required. All other cases require $2d+1$ parallel transmissions. At the end of Section 4 it is shown that any index-digit permutation can be performed in less than or equal to $1.5d$ parallel transmissions.

2. Existing FFTs

Eight different FFTs are presented in this section for use later in developing FFTs that are suitable for multiprocessors. One such FFT will be a composite of two FFTs that are described in this section. Given a sequence x_n, we begin with a definition of the discrete Fourier transform

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-ink2\pi/N}. \tag{2.1}$$

If N has factors $N = N_0 N_1$, then by using the standard index maps

$$n = i + jN_0, \qquad k = l + mN_1 \tag{2.2a, b}$$

we can define the two-dimensional arrays

$$x(i, j) = x_n, \quad i = 0, \dots, N_0 - 1, \; j = 0, \dots, N_1 - 1, \tag{2.3a}$$

$$X(l, m) = X_k, \quad l = 0, \dots, N_1 - 1, \; m = 0, \dots, N_0 - 1, \tag{2.3b}$$

Substituting (2.2) and (2.3) into (2.1) we obtain

$$X(l, m) = \sum_{i=0}^{N_0 - 1} \omega_{N_0}^{im} \omega_N^{il} \sum_{j=0}^{N_1 - 1} x(i, j) \omega_{N_1}^{jl} \tag{2.4}$$

where $\omega_N = e^{-i2\pi/N}$. Therefore $X(l, m)$, and hence X_k, can be computed as two multiple transforms. First, N_0 transforms of length N_1 (multiplied by ω_N^{il}) are computed from

$$X^{(1)}(i, l) = \omega_N^{il} \sum_{j=0}^{N_1-1} x(i, j)\omega_{N_1}^{jl}. \tag{2.5}$$

Next, N_1 transforms of length N_0 are computed from

$$X^{(2)}(m, l) = \sum_{i=0}^{N_0-1} X^{(1)}(i, l)\omega_{N_0}^{im}. \tag{2.6}$$

Finally from (2.4)

$$X(l, m) = X^{(2)}(m, l) \tag{2.7}$$

which implies the results from (2.6) be transposed or digit reversed to obtain the ordered transform.

If N_1 is selected as a small prime factor of N, then (2.5) can be computed quickly as N_0 short transforms of length N_1. Then, instead of computing $X^{(2)}(m, l)$ from (2.6), we can first factor the transforms in the same way the transform (2.1) was factored. If $N = N_0 N_1 \cdots N_{r-1}$, then r such factorizations are possible which yield the Cooley–Tukey FFT. To generalize the factorization (2.5) we must first generalize the index map (2.2a) by

$$n = I(i_0, \ldots, i_{r-1}) \tag{2.8}$$

where the index function is defined by

$$I(i_0, \ldots, i_{r-1}) = i_0 + i_1 N_0 + \cdots + i_{r-1} N_0 \cdots N_{r-2} \tag{2.9}$$

Similarly we generalize (2.2b) by

$$k = I(k_{r-1}, \ldots, k_0). \tag{2.10}$$

Proceeding in a manner similar to the case of two factors we define the multi-dimensional arrays:

$$x(i_0, \ldots, i_{r-1}) = x_n, \qquad X(k_{r-1}, \ldots, k_0) = X_k. \tag{2.11a, b}$$

Repeated factorization yields the following recurrence, which is a generalization of (2.5):

$$X^{(s+1)}(i_0, \ldots, i_{r-s-1}, k_{r-s}, \ldots, k_{r-1}) = \omega_{N(s)}^{ik_{r-s}} \sum_{i_{r-s}=0}^{N_{r-s}-1} X^{(s)}(i_0, \ldots, i_{r-s}, k_{r-s+1}, \ldots, k_{r-1})\omega_{N_{r-s}}^{i_{r-s}k_{r-s}} \tag{2.12}$$

where i is defined below in equation (2.15a) and $N(s) = N_0 \cdots N_{r-s}$. The recurrence is initialized with

$$X^{(1)}(i_0, \ldots, i_{r-1}) = x(i_0, \ldots, i_{r-1}). \tag{2.13}$$

The FFT is then given by a digit-reversed ordering, which is the generalization of (2.7):

$$X_k = X(k_{r-1}, \ldots, k_0) = X^{(r)}(k_0, \ldots, k_{r-1}). \tag{2.14}$$

In practice (2.12) can be performed with three subscripts. First define

$$i = I(i_0, \ldots, i_{r-s-1}), \qquad k = I(k_{r-s+1}, \ldots, k_{r-1}), \tag{2.15a, b}$$

then (2.11) has the form

$$X^{(s+1)}(i, k_{r-s}, k) = \omega_{N(s)}^{ik_{r-s}} \sum_{i_{r-s}=0}^{N_{r-s}-1} X^{(s)}(i, i_{r-s}, k)\omega_{N_{r-s}}^{i_{r-s}k_{r-s}}. \tag{2.16}$$

The recurrence (2.16) is in a form that is convenient to program. It can be included in a quadruple loop in a subprogram that is called r times. The array $X^{(s)}(i, i_{r-s}, k)$ has variable dimensions that depend on s and are determined by the index ranges $i = 0, \ldots, N_0 \cdots N_{r-s-1} - 1$ and $k = 0, \ldots, N_{r-s+1} \cdots N_{r-1}$.

Sample programs for the case $N_i = 2$ are given in [12]. The recurrence (2.16) defines a single step of the Cooley–Tukey FFT which has the attribute that it can be performed in place. However, as mentioned above, $X^{(r)}(i, k_{r-s}, k)$ must be digit reversed if an ordered transform is desired. This explicit reordering can be eliminated by a partial reordering at each step s of the FFT. To this end, consider the following reorderings of the intermediate sequences:

$$Y^{(s)}(k_{r-1}, \ldots, k_{r-s+1}, i_0, \ldots, i_{r-s}) = X^{(s)}(i_0, \ldots, i_{r-s}, k_{r-s+1}, \ldots, k_{r-1}) \qquad (2.17)$$

which implies that

$$Y^{(s+1)}(k_{r-1}, \ldots, k_{r-s}, i_0, \ldots, i_{r-s-1}) = X^{(s+1)}(i_0, \ldots, i_{r-s-1}, k_{r-s}, \ldots, k_{r-1}). \qquad (2.18)$$

If we then define i as in (2.15a) but k as

$$k = I(k_{r-1}, \ldots, k_{r-s+1}) \qquad (2.19)$$

and substitute (2.17) and (2.18) into (2.12), we obtain the following recurrence for the reordered sequences:

$$Y^{(s+1)}(k, k_{r-s}, i) = \omega_{N(s)}^{ik_{r-s}} \sum_{i_{r-s}=0}^{N_{r-s}-1} Y^{(s)}(k, i, i_{r-s}) \omega_{N_{r-s}}^{i_{r-s}k_{r-s}}. \qquad (2.20)$$

From (2.14) and (2.17)

$$X_k = X(k_{r-1}, \ldots, k_0) = X^{(r)}(k_0, \ldots, k_{r-1}) = Y(k_{r-1}, \ldots, k_0) \qquad (2.21)$$

and therefore $Y(k_{r-1}, \ldots, k_0)$ contains the ordered transform. The recurrence (2.20) defines a single step of the Stockham auto-sort FFT [11–13]. It has the advantage that the transform is ordered, but it cannot be performed in place. However, this disadvantage does not seem too important since it is a popular algorithm, particularly for general purpose software.

A variant of the Stockham algorithm which is also auto-sorting, can be obtained by using a different ordering of the intermediate sequences given by

$$Y^{(s)}(i_0, \ldots, i_{r-s}, k_{r-1}, \ldots, k_{r-s+1}) = X^{(s)}(i_0, \ldots, i_{r-s}, k_{r-s+1}, \ldots, k_{r-1}). \qquad (2.22)$$

In terms of this ordering the recurrence (2.12) takes the form

$$Y^{(s+1)}(i, k, k_{r-s}) = \omega_{N(s)}^{ik_{r-s}} \sum_{i_{r-s}=0}^{N_{r-s}-1} Y^{(s)}(i, i_{r-s}, k) \omega_{N_{r-s}}^{i_{r-s}k_{r-s}}. \qquad (2.23)$$

Since (2.21) is applicable to this case as well, (2.23) is also auto-sorting.

The last FFT to be developed in this section is the Pease FFT, which is not auto-sorting but uses long vectors. It is obtained from yet another ordering:

$$Y^{(s)}(k_{r-s+1}, \ldots, k_{r-1}, i_0, \ldots, i_{r-s}) = X^{(s)}(i_0, \ldots, i_{r-s}, k_{r-s+1}, \ldots, k_{r-1}). \qquad (2.24)$$

With i and k as defined in (2.15a, b) the recurrence (2.12) takes the form

$$Y^{(s+1)}(k_{r-s}, k, i) = \omega_{N(s)}^{ik_{r-s}} \sum_{i_{r-s}=0}^{N_{r-s}-1} Y^{(s)}(k, i, i_{r-s}) \omega_{N_{r-s}}^{i_{r-s}k_{r-s}}. \qquad (2.25)$$

From (2.14) and (2.24)

$$X_k = X(k_{r-1}, \ldots, k_0) = X^{(r)}(k_0, \ldots, k_{r-1}) = Y(k_0, \ldots, k_{r-1}) \qquad (2.26)$$

and therefore a digit reversal is required to obtain an ordered transform. The interest in the Pease FFT is due to the fact that the index pair k, i occurs in the same order on both sides of (2.25), and hence it can be replaced by a single index with a larger range. That is, (2.25) can be written as

$$Y^{(s+1)}(k_{r-s}, m) = \omega_{N(s)}^{i(m)k_{r-s}} \sum_{i_{r-s}=0}^{N_{r-s}-1} Y^{(s)}(m, i_{r-s}) \omega_{N_{r-s}}^{i_{r-s}k_{r-s}}. \tag{2.27}$$

The index m has the range $m = 0, \ldots, N/N_{r-s} - 1$, which is large compared with the other FFTs, and hence (2.27) is efficient on vector processors. However, the Pease FFT is not competitive if the transform must be ordered. The Pease FFT also requires $O(N \log N)$ storage for the $\omega_{N(s)}^{i(m)k_{r-s}}$ that are replications of $\omega_{N(s)}^{ik_{r-s}}$ that are necessary to construct a vector with N/N_{r-s} elements.

The inverse of the transforms presented in this section can be obtained in two different ways. One can either replace ω by ω^{-1}, or the order of the operations in any of the recurrences can be formally reversed. If both of these inversions are implemented, a new forward transform is obtained. This yields an additional four FFTs that are distinct from those presented above but identical to those presented in [12]. Thus we have a total of eight FFTs from which to develop FFTs that are suitable for multiprocessors.

3. Vector-concurrent FFTs

Two vector-concurrent FFTs are developed in this section that are designed for vector multi-processors with shared memory such as the CRAY-XMP or ALLIANT FX/8. The FFTs are related in the sense that computations are performed on long vectors whose elements are stored consecutively (have a stride of one) except for a single pass in which the data movement is similar to a matrix transpose. They differ in their development and implementation. The first FFT is based on a combination of FFTs presented in the previous section. The second FFT is based on equation (2.4), which shows that the transform of a single sequence can be computed in terms of two multiple transforms. This approach is attractive, since the multiple transform problem offers options for distribution and vectorization that are not offered by the transform of a single sequence. The result is an algorithm that is efficient for both single and multiple transforms. The implementation of this FFT on the ALLIANT FX/8 vector-multiprocessor is described at the end of the section.

In theory, all the computations in (2.12), can be computed in parallel. The computations for the 'block' of indices, $i = 0, \ldots, N_0 \cdots N_{r-s-1} - 1$, $k_{r-s} = 0, \ldots, N_{r-s} - 1$, and $k = 0, \ldots, N_{r-s+1} \cdots N_{r-1} - 1$ can be performed simultaneously. On a multiprocessor with p processors, it would be seem reasonable to partition this block into p sections with a resulting speedup of about p. However, in the context of a vector processor and a desire to reduce the total number of computations, the situation becomes somewhat more complex.

It is known [1] that the total number of computations can be significantly reduced by unrolling the loop with index i_{r-s}. The bulk of the computations in most implementations of the FFT is performed in subprograms where N_{r-s} is fixed and the index $i_{r-s} = 0, \ldots, N_{r-s} - 1$ is unrolled and does not explicitly appear as an index. This is referred to as a fixed-radix subprogram, and most implementations have at least subprograms for $N_{r-s} = 2$ and 4. The motivation for this unrolling is simply that one can reduce the number of operations by eliminating multiplications by ± 1 and $\pm i$ and by various other computational techniques.

Hence the 'block' of indices is reduced to the 'rectangle' $i = 0, \ldots, N_0 \cdots N_{r-s-1} - 1$, and $k = 0, \ldots, N_{r-s+1} \cdots N_{r-1} - 1$ or, for the Pease FFT, the 'line' $m = 0, \ldots, N/N_{r-s} - 1$. Therefore the Pease FFT does not have a nuisance attribute shared by the other FFTs, namely, that the shape of the 'rectangle' changes as a function of s. This change results in a degradation of performance because the vectors vary substantially in length including vectors of length one. This degradation can be minimized by loop inversion which is discussed in [11–13]. The idea is simply to choose i or k as the index for the inner loop, depending on which has the greater range. This requires a duplicate copy of the code in which the order of the loops is interchanged. The effect is to lengthen the shortest vector to $\sqrt{N/N_i}$ (for some i) which can significantly improve performance.

However, there is an unpleasant side effect to loop inversion. Either before or after the inversion the stride of the vector is no longer unity, which tends to degrade somewhat the performance of a vector processor. This effect can also be minimized. The index i corresponds to the inner loop of the Stockham variant FFT (2.23), whereas the index k corresponds to the inner loop of the Stockham FFT (2.20). Therefore, if at the crossover point where the ranges of i and k are about equal, we convert from the order (2.22) to (2.17) and proceed with (2.20), then the stride is always be equal to one except for the reordering. The reordering can be combined with a step of the FFT by using the following 'link' for only one value of s at the crossover point:

$$Y^{(s+1)}(k, k_{r-s}, i) = \omega_{N(s)}^{ik_{r-s}} \sum_{i_{r-s}=0}^{N_{r-s}-1} Y^{(s)}(i, i_{r-s}, k)\,\omega_{N_{r-s}}^{i_{r-s}k_{r-s}}. \tag{3.1}$$

Although the stride in (3.1) is not unity, it is executed only once during the FFT. The movement of data is similar to a matrix transposition which can be performed in a reasonably efficient way.

An alternative way to carry out the orderings that are implicit in each pass of the FFT is to perform a gather (or several gathers) at each pass in the FFT. This method is due to Fornberg [2] and is particularly useful on the CYBER 205, where a gather is more economical than a scatter and long vectors are particularly important. The idea is to perform N_{r-s} gathers at each pass instead of using a stride that is not equal to one. However, this method does not work for all the FFTs given Section 2. The indices i and k must appear as the first two subscripts in the recurrence as they do in the Stockham variant (2.23) or in the double inverted form (3.2) of the Stockham FFT (2.20).

Although the vectors are reasonably long and the stride is one, there is still the question of how to distribute the computations. It would be attractive to simply distribute the outer loop. For example, the computations begin with (2.23) in which i is the inner loop and k is the outer loop. The difficulty with distributing the outer loop is that its range begins at one. Therefore it is necessary to break up the inner loop and distribute the pieces among the processors. As the computation proceeds it is necessary to keep track of the current shape of the 'rectangle' and distribute the computations accordingly.

An approach that avoids this complication is given in a second vector-concurrent FFT which follows. From Section 2 it is evident that the transform of a single sequence can be performed as two or more multiple transforms. The computation of (2.1) was performed in terms of the two multiple transforms (2.5) and (2.6). This result was then extended to additional multiple transforms in the discussion that follows (2.7). Thus it seems reasonable to focus some effort on the multiple transform problem, particularly in the context of vector-multiprocessors where it provides additional flexibility for distribution and vectorization. It also provides options from an algorithmic point of view. For example, it is conceivable that the two multiple-transform problems would be solved on different computers each running FFTs that were developed for the particular architecture.

The multiple transform problem would seem particularly suited to the multiprocessor since one could simply distribute the transforms across the processors. Indeed this approach seems reasonable for scalar processors without shared memory. However for vector processors with a shared memory system it is more attractive to distribute the individual operations in the FFT. That is, each processor is assigned a certain number of computations which it performs on all of the sequences. The number of operations depends on N, which is usually sufficient for distribution in a manner that is independent of the step s. If there are M sequences of length N, then all of the vectors in each processor have length M.

To maintain a stride of one it is necessary to store the sequences rowwise. That is, the first element of the first sequence is followed by the the first element of the second sequence and so forth. From (2.5) it can be seen that this is the natural order if the multiple sequences have occurred in the course of transforming a single sequence. A stride of one can be maintained in the second multiple transform (2.6) if $X^{(1)}(i, l)$ is transposed prior to the computations in (2.6). Then, following the second multiple transform, the results are in the correct order without the digit reversal (2.7). Like the previous algorithm, this approach has a separate 'link' step in which $X^{(1)}(i, l)$ is transposed. This transposition could be combined with either the last step of (2.5) or the first step of (2.6). This algorithm was implemented on the ALLIANT FX/8 using the the double inverted form of the Stockham FFT (2.20)

$$Y^{(s)}(m, k, i, i_{r-s}) = \sum_{k_{r-s}=0}^{N_{r-s}-1} \omega_{N(s)}^{ik_{r-s}} Y^{(s+1)}(m, k, k_{r-s}, i)\omega_{N_{r-s}}^{i_{r-s}k_{r-s}}. \tag{3.2}$$

The additional index m corresponds to the multiple-transform index with range $m = 0, \ldots, M - 1$. If it is chosen as the inner loop, then all multiplies are scalar-vector and the stride is one.

The double inverted form (3.2) was motivated by the architecture of the FX/8. The difference between (3.2) and (2.20) is simply that the first operation in (3.2) is a scalar-vector multiply, whereas the first operation in (2.20) is a vector-vector add. This observation is not evident until one examines the computations for a fixed radix (say 2 or 4) in which the k_{r-s} loop is unrolled. On the FX/8 it is possible to combine a vector load with a scalar-vector mutliply, however it is not possible to combine a vector load with a vector-vector add. By placing scalar-vector multiplies at the beginning of the computations, it is possible to eliminate vector loads. Also, since all the multiplies are scalar-vector, one can make extensive use of the linked triad instruction on the FX/8.

Using these techniques, it is possible to code a radix 2 pass with 12 vector instructions that perform 10 floating-point operations and a radix 4 pass with 36 vector instructions that perform 34 floating-point operations. As described in Section 2, the index k_{r-s} is unrolled in the radix 2 and 4 subprograms, leaving m as the inner loop and both k and i as outer loops that must be distributed across the ALLIANT's eight processors. Since the compiler distributes only the loop that is next to the inner loop and since the range of both k and i is, for some pass, less than eight, it was necessary to combine the two outer loops into a single loop (say ik) whose range was always greater than eight (for reasonable values of N). This permits the compiler to distribute the single ik loop in a way that keeps all eight processors active.

4. Hypercube FFTs

In this section FFTs are developed for the hypercube in which the memory is distributed across the processors and interprocessor communication is necessary if the processors share the computation. It is assumed that the hypercube has dimension d and hence $P = 2^d$ processors. It is also assumed that a single sequence, of length $N = 2^r$, will be transformed and that each processor contains 2^{r-d} elements. It is known that the time required by interprocessor communication is substantial and hence the focus is primarily on reducing communication. To this end, we develop efficient communication algorithms and observe that the amount of communication depends significantly on whether or not the FFT is ordered.

All the FFTs in Section 2 were developed from the Cooley–Tukey FFT simply by reordering the intermediate computations. Each FFT was attractive for a particular application, but the emphasis was primarily on vectorization and shared-memory multiprocessors. In this section we will use the same procedure to develop FFTs that are particularly suited to the hypercube by again reordering the intermediate computations in the Cooley–Tukey FFT.

These orderings are somewhat more complex than those developed in Section 2, but they all belong to a class of orderings called index-digit permutations [4]. This class also includes matrix transposition, digit reversal, and the perfect shuffle. The index-digit permutation is defined below together with several other concepts that are fundamental to subsequent discussions.

An *index-digit permutation* of the sequence $x_n = x(i_0, \dots, i_{r-1})$ is an ordering in which the element x_n in location $n = i_{r-1}i_{r-2}\dots i_0$ (binary) is moved to location $n' = i'_{r-1}i'_{r-2}\dots i'_0$ (binary) where the digits i'_j are a permutation of the digits i_j. For example, if $r = 4$ and $n = i_3i_2i_1i_0$, then an ordering, in which the element x_n is moved to location $n' = i_1i_0i_2i_3$, is an index-digit permutation which is designated by $x(i_3, i_2, i_0, i_1)$. Equations (2.17), (2.22), and (2.24) also provide examples of index-digit permutations.

The *position of an index* is the number of indices to the left of the index. Any index-digit permutation can be defined by a cycle whose elements are positions of an index. For example, the permutation given above can be defined by the cycle (0, 2, 1, 3) in which the index at position 0 moves to position 2, position 2 moves to position 1, and so forth. Since the order of indices is the opposite of digits, the *position of a digit* is defined as the number of digits to the right of the digit.

A *standard sequence to processor map* is one in which the element x_n with $n = i_{r-1}i_{r-2}\dots i_0$ (binary) has *address* $i_{r-d-1}i_{r-d-2}\dots i_0$ in *processor number* $i_{r-1}i_{r-2}\dots i_{r-d}$. Therefore both a processor number and address are required to identify a particular element in the sequence. Sequence to processor maps are designated by $x(i_0, \dots, i_{r-d-1} \mid i_{r-d}, \dots, i_{r-1})$ where the partition $\mid$ is introduced for expository purposes to separate the address on the left from the processor number on the right. For example if $r = 4$ and $d = 2$, then the element $x(i_0, i_1 \mid i_2, i_3)$ has address i_1i_0 and is located in processor number i_3i_2. An *index permuted sequence to processor map* is one in which the indices i_j are permuted.

Definition. An *i-cycle* is an index-digit permutation of x_n in which the most significant digit of the address (called the *pivot*) is exchanged with any other digit, either in the address or the processor number.

Alternatively, an i-cycle is a reordering that exchanges the index in position $r - d - 1$ with any other index. Two examples of i-cycles are given in Table 1. The second entry in Table 1 is obtained from the first by an i-cycle that exchanges indices in the second (pivot) and first position. The third entry is obtained from the second by an i-cycle that exchanges indices in the second and the fourth position. The i-cycle has three properties that make it particularly useful for algorithmic design on the hypercube.

Table 1
Sample i-cycles for the case $d = 2$ and $r = 5$

$X(i_0, i_1, i_2 \mid i_3, i_4)$
$X(i_0, i_2, i_1 \mid i_3, i_4)$
$X(i_0, i_2, i_4 \mid i_3, i_1)$

I-cycle property A. An i-cycle may or may not require interprocessor communication, depending on whether or not the index is in the processor number.

For example, the first i-cycle in Table 1 does not require interprocessor communication because the processor number is unchanged. However the second i-cycle does require interprocessor communication because the processor number is changed. When interprocessor communication is required, then it is efficient since it occurs between neighboring processors.

To see this, let the processor number be fixed and b_p be the bit that is exchanged with the pivot. If b_p and the pivot are the same for some element in that processor, then that element remains in the processor since the processor number is unchanged following the i-cycle. However, if the bits are different the word must be transmitted to a processor whose number (in binary) differs in only one digit, and hence the transmission is to a neighboring processor.

I-cycle property B. Since the pivot is the most significant digit in the address field, the elements to be transmitted are located consecutively.

It can be shown that this packet has length $2^{r-d-1} = N/(2P)$ elements. Also, since the digits are binary, there is at most one destination processor.

I-cycle property C. Any index-digit permutation can be implemented as a sequence of i-cycles.

To see this, first decompose the permutation into disjoint cycles. Next decompose each cycle into i-cycles by interchanging the first position with the pivot position and restore it following the completion of the cycle. For example, if the cycle is (2, 8, 7, 5) and the pivot is in position 3, then this cycle is equivalent to the i-cycles (3, 2)(3, 8)(3, 7)(3, 5)(3, 2) applied from left to right.

Consider now the implementation of the FFT on the hypercube. We begin with a review of the intermediate orderings in the Cooley–Tukey FFT which will be index-digit permuted to obtain the orderings for the hypercube FFTs. The orderings for the particular case $N = 16$ or $r = 4$ are given in Table 2. The original sequence is at the top of the table and the remaining entries correspond to the intermediate sequences in the FFT. At each step i_j is replaced by k_j using the recurrence (2.12). The final entry in Table 2 is the ordered transform which is obtained from the previous entry by a digit reversal. The letter a in the superscript indicates an ordering rather than computational step.

Now assume that we wish to perform this computation on a hypercube with eight processors. If the sequence is mapped onto the processors in standard order, each processor contains two elements. This distribution is denoted as $x(i_0 \,\vdots\, i_1, i_2, i_3)$.

The difficulty with performing the FFT is that the first step ($s = 1$) requires the computation of a sum on the index i_3 which is located in the processor number. For $s = 1$, the computation of $X^{(2)}(i_0, i_1, i_2, k_3)$ using (2.12) requires values that are different processors. To make these values available in the same processor we must first perform an i-cycle to bring i_3 into the address. The result of the i-cycle and the computations is listed as the second entry in Table 3.

Table 2
Intermediate orderings for Cooley–Tukey FFT, $N = 16$

$x(i_0, i_1, i_2, i_3)$	$X^{(4)}(i_0, k_1, k_2, k_3)$
$X^{(2)}(i_0, i_1, i_2, k_3)$	$X^{(5)}(k_0, k_1, k_2, k_3)$
$X^{(3)}(i_0, i_1, k_2, k_3)$	$X^{(5a)}(k_3, k_2, k_1, k_0)$

Table 3
Intermediate orderings for the hypercube FFT, $d = 3$, $r = 4$

$x(i_0 \mid i_1, i_2, i_3)$	$X^{(4)}(k_1 \mid k_2, k_3, i_0)$
$X^{(2)}(k_3 \mid i_1, i_2, i_0)$	$X^{(5)}(k_0 \mid k_2, k_3, k_1)$
$X^{(3)}(k_2 \mid i_1, k_3, i_0)$	

The remaining entries define the intermediate orderings and i-cycles that are necessary to compute the FFT.

The last entry in Table 3 is the unordered transform which was obtained with four parallel transmissions. In general the unordered transform can be obtained with $d + 1$ parallel transmissions for any $1 < d < r$. This is demonstrated by applying an extension of the algorithm in Table 3 to the right $d + 1$ indices $i_{r-d-1} \mid i_{r-d}, \ldots, i_{r-1}$. In this way the k_j's for $j \geqslant r - d - 1$ are computed with $d + 1$ parallel transmissions. The remaining k_j's can be computed without interprocessor communication.

It will be shown that the ordered transform requires at least $1.5d + 2$ parallel transmissions. Hence, where applicable, the unordered transform is preferred. It can be used for any problem in which the transform remains invisible to the user such as the solution of partial differential equations using the Fourier method [11,13]. Following the development in Section 2, we can obtain the inverse transform by the formal inversion of the operations given in Table 3. The result is the original ordered sequence. Thus transforms can be performed between physical and Fourier space without introducing new orderings and with the optimal number of parallel transmissions $d + 1$. This concludes the comments on the unordered transform. The fundamental result for the ordered transform is given in the following theorem.

Theorem 4.1. *An ordered FFT of length $N = 2^r$ can be implemented on a hypercube of dimension d with $r/2 + d + 1$ parallel transmissions if $d > r/2$ and r is even. If r is odd and $d > (r + 1)/2$, then $(r + 1)/2 + d + 1$ parallel transmissions are required. For the remaining cases the ordered FFT can be implemented with $2d + 1$ parallel transmissions.*

Proof. Only the proof for even r is given since the proof for odd r is quite similar. The proof is separated into four cases: $d = r/2$, $d < r/2$, $d = r - 1$, and $r/2 < d < r - 1$.

(a) $d = r/2$. The i-cycles for the special case $r = 8$ and $d = 4$ are given in Table 4. The size of r and d have been increased compared with previous examples so that the pattern of i-cycles that define the algorithm is evident.

The algorithm for the general case $d = r/2$ has two parts. In the first part the i-cycles alternate between indices in the processor number and the address. The result of this part is $X^{(d+1)}(k_{r-1}, \ldots, k_{r-d} \mid i_{r-d-2}, \ldots, i_0, i_{r-d-1})$. In the second part the i-cycles reference only the processor number much like the i-cycles for the unordered FFT. The result is an ordered FFT

Table 4
Intermediate orderings for the hypercube FFT, $d = 4$, $r = 8$

$x(i_0, i_1, i_2, i_3 \mid i_4, i_5, i_6, i_7)$	$X^{(5)}(k_7, k_6, k_5, k_4 \mid i_2, i_1, i_0, i_3)$
$X^{(2)}(i_0, i_1, i_2, k_7 \mid i_4, i_5, i_6, i_3)$	$X^{(6)}(k_7, k_6, k_5, k_3 \mid i_2, i_1, i_0, k_4)$
$X^{(2a)}(k_7, i_1, i_2, i_0 \mid i_4, i_5, i_6, i_3)$	$X^{(7)}(k_7, k_6, k_5, k_2 \mid k_3, i_1, i_0, k_4)$
$X^{(3)}(k_7, i_1, i_2, k_6 \mid i_4, i_5, i_0, i_3)$	$X^{(8)}(k_7, k_6, k_5, k_1 \mid k_3, k_2, i_0, k_4)$
$X^{(3a)}(k_7, k_6, i_2, i_1 \mid i_4, i_5, i_0, i_3)$	$X^{(9)}(k_7, k_6, k_5, k_0 \mid k_3, k_2, k_1, k_4)$
$X^{(4)}(k_7, k_6, i_2, k_5 \mid i_4, i_1, i_0, i_3)$	$X^{(9a)}(k_7, k_6, k_5, k_4 \mid k_3, k_2, k_1, k_0)$
$X^{(4a)}(k_7, k_6, k_5, i_2 \mid i_4, i_1, i_0, i_3)$	

Table 5
Intermediate orderings for the hypercube FFT, $d = 3$, $r = 8$

$x(i_0, i_1, i_2, i_3, i_4 \mid i_5, i_6, i_7)$	$X^{(5)}(k_7, k_6, k_5, i_3, k_4 \mid i_1, i_0, i_2)$
$X^{(1a)}(i_0, i_1, i_4, i_3, i_2 \mid i_5, i_6, i_7)$	$X^{(6)}(k_7, k_6, k_5, k_4, k_3 \mid i_1, i_0, i_2)$
$X^{(2)}(i_0, i_1, i_4, i_3, k_7 \mid i_5, i_6, i_2)$	$X^{(7)}(k_7, k_6, k_5, k_4, k_2 \mid i_1, i_0, k_3)$
$X^{(2a)}(k_7, i_1, i_4, i_3, i_0 \mid i_5, i_6, i_2)$	$X^{(8)}(k_7, k_6, k_5, k_4, k_1 \mid k_2, i_0, k_3)$
$X^{(3)}(k_7, i_1, i_4, i_3, k_6 \mid i_5, i_0, i_2)$	$X^{(9)}(k_7, k_6, k_5, k_4, k_0 \mid k_2, k_1, k_3)$
$X^{(3a)}(k_7, k_6, i_4, i_3, i_1 \mid i_5, i_0, i_2)$	$X^{(9a)}(k_7, k_6, k_5, k_4, k_3 \mid k_2, k_1, k_0)$
$X^{(4)}(k_7, k_6, i_4, i_3, k_5 \mid i_1, i_0, i_2)$	

at step $((r+1)a)$. The number of parallel transmission in Table 4 is nine. A total of $2d+1$ are required for the case $d = r/2$.

(b) $d < r/2$. The i-cycles for the special case $r = 8$ and $d = 3$ are given in Table 5.

The i-cycles for the general case $d < r/2$ can be separated into three parts. The first and last parts make use of the algorithm developed above for case (a).

(1) Interchange i_{d-1} and the pivot i_{r-d-1} as in step (1a) in Table 5. Then apply the first half of the algorithm for case (a) to the left $d-1$ and right $d+1$ indices $i_0, \ldots, i_{d-1} \mid i_{r-d}, \ldots, i_{r-1}$. The result of this part is the ordering:

$$X^{(d+1)}(k_{r-1}, \ldots, k_{r-d+1}, i_{d+1}, \ldots, i_{r-d-2}, k_{r-d} \mid i_{d-2}, \ldots, i_0, i_{d-1}).$$

This part requires d parallel transmissions.

(2) Compute k_{r-d-1} through k_d. The result of this part is the ordering $X^{(r-d+1)}$ $(k_{r-1}, \ldots, k_d \mid i_{d-2}, \ldots, i_0, i_{d-1})$. In Table 5 this result is in step (6). This part does not require any interprocessor communication since the processor number remains unchanged throughout.

(3) Complete the last half of the algorithm for case (a). The result is an ordered transform at step $((r+1)a)$. This part requires $d+1$ parallel transformations and hence the total number of parallel transmissions for case (b) is $2d+1$.

(c) $d = r-1$. The i-cycles for the special case $r = 8$ and $d = 7$ are given in Table 6.

The algorithm for the general case $d = r-1$ has two parts. The first part of the i-cycles in Table 6 are determined by following the pattern established in Table 3 in which the k_j's are computed in the order of descending j. The result of this part is $X^{(r/2+2)}$ $(k_{r/2-1} \mid i_1, \ldots, i_{r/2-2}, k_{r/2}, \ldots, k_{r-1}, i_0)$. This part requires $r/2+1$ parallel transmissions.

From Table 3 it is evident that if the pattern established in the first half is continued, then the resulting transform would not be ordered. Therefore in the second part we switch to i-cycles that guarantee the indices will be reversed. Starting with step (6), the permutation that produces the the indices in reverse order is given by the cycle (0, 4, 2, 5, 1, 6). Notice that every other position, starting with 2, decreases by 1 whereas every other position, starting with 4, increases by 1.

Table 6
Intermediate orderings for the hypercube FFT, $d = 7$, $r = 8$

$x(i_0 \mid i_1, i_2, i_3, i_4, i_5, i_6, i_7)$	$X^{(7)}(k_2 \mid i_1, k_5, k_4, k_3, k_6, k_7, i_0)$
$X^{(2)}(k_7 \mid i_1, i_2, i_3, i_4, i_5, i_6, i_0)$	$X^{(7a)}(k_6 \mid i_1, k_5, k_4, k_3, k_2, k_7, i_0)$
$X^{(3)}(k_6 \mid i_1, i_2, i_3, i_4, i_5, k_7, i_0)$	$X^{(8)}(k_1 \mid k_6, k_5, k_4, k_3, k_2, i_7, i_0)$
$X^{(4)}(k_5 \mid i_1, i_2, i_3, i_4, k_6, k_7, i_0)$	$X^{(8a)}(k_7 \mid k_6, k_5, k_4, k_3, k_2, k_1, i_0)$
$X^{(5)}(k_4 \mid i_1, i_2, i_3, k_5, k_6, k_7, i_0)$	$X^{(9)}(k_0 \mid K_6, k_5, k_4, k_3, k_2, k_1, k_7)$
$X^{(6)}(k_3 \mid i_1, i_2, k_4, k_5, k_6, k_7, i_0)$	$X^{(9a)}(k_7 \mid k_6, k_5, k_4, k_3, k_2, k_1, k_0)$
$X^{(6a)}(k_5 \mid i_1, i_2, k_4, k_3, k_6, k_7, i_0)$	

Table 7
Intermediate orderings for the hypercube FFT, $d = 5$, $r = 8$

$x(i_0, i_1, i_2 \,\vdots\, i_3, i_4, i_5, i_6, i_7)$	$X^{(6)}(k_7, k_6, k_3 \,\vdots\, k_4, k_5, i_1, i_0, i_2)$
$X^{(2)}(i_0, i_1, k_7 \,\vdots\, i_3, i_4, i_5, i_6, i_2)$	$X^{(6a)}(k_7, k_6, k_5 \,\vdots\, k_4, k_3, i_1, i_0, i_2)$
$X^{(2a)}(k_7, i_1, i_0 \,\vdots\, i_3, i_4, i_5, i_6, i_2)$	$X^{(7)}(k_7, k_6, k_2 \,\vdots\, k_4, k_3, i_1, i_0, k_5)$
$X^{(3)}(k_7, i_1, k_6 \,\vdots\, i_3, i_4, i_5, i_0, i_2)$	$X^{(8)}(k_7, k_6, k_1 \,\vdots\, k_4, k_3, k_2, i_0, k_5)$
$X^{(3a)}(k_7, k_6, i_1 \,\vdots\, i_3, i_4, i_5, i_0, i_2)$	$X^{(9)}(k_7, k_6, k_0 \,\vdots\, k_4, k_3, k_2, k_1, k_5)$
$X^{(4)}(k_7, k_6, k_5 \,\vdots\, i_3, i_4, i_1, i_0, i_2)$	$X^{(9a)}(k_7, k_6, k_5 \,\vdots\, k_4, k_3, k_2, k_1, k_0)$
$X^{(5)}(k_7, k_6, k_4 \,\vdots\, i_3, k_5, i_1, i_0, i_2)$	

Following the development in the definition of the i-cycle, the cycle can be implemented by the i-cycles (0, 4)(0, 2)(0, 5)(0, 1)(0, 6) which are given in steps. (6a) through (8a) in Table 6. Finally k_0 is computed in step (9) and placed in its proper position in step (9a). The number of parallel transmissions in Table 6 is twelve. A total of $3r/2$ are required for the case $d = r - 1$. A total of $3(r+1)/2 - 1$ parallel transmissions are required if r is odd. In terms of d, the number of parallel transmissions, is $3(d+1)/2$ which has a noticeably smaller constant of proportionality than $2d + 1$ which is required for $d < r/2$.

(d) $r/2 < d < r - 1$. This case is treated by combining the algorithms used for the cases (a) $d = r/2$ and (c) $d = r - 1$. The i-cycles are given in Table 7 for the case $r = 8$ and $d = 5$.

The i-cycles for the general case $r/2 < d < r - 1$ are constructed in three parts.

(1) Perform the first part of the case (a), $d = r/2$ algorithm, except for the last step, on the left $r - d$ and the right $r - d$ indices while ignoring the center indices: $i_{r-d}, \ldots, i_{d-1}$. The result of this part is

$$X^{((r-d)a)}\big(k_{r-1}, \ldots, k_{d+1}, i_{r-d-2} \,\vdots\, i_{r-d}, \ldots, i_d, i_{r-d-3}, \ldots, i_0, i_{r-d-1}\big).$$

This part requires $r - d - 1$ parallel transmissions.

(2) Perform the case (c), $d = r - 1$ algorithm, except for the last two steps, on the center indices $i_{r-d-2} \,\vdots\, i_{r-d}, \ldots, i_d$. Since the length of this sequence is $2d - r + 2$, the total number of parallel transmissions is $3(2d - r + 2)/2 - 2$. The result of this part is

$$X^{((d+1)a)}\big(k_{r-1}, \ldots, k_d \,\vdots\, k_{d-1}, \ldots, k_{r-d}, i_{r-d-2}, \ldots, i_0, i_{r-d-1}\big).$$

(3) Perform the last half of the case (a), $d = r/2$, algorithm on the left $r - d$ and right $r - d$ indices. This part requires $r - d + 1$ parallel transmissions. The total number of transmissions for parts (1), (2), and (3) is $r/2 + d + 1$. If r is odd, then it can be determined that $(r+1)/2 + d + 1$ parallel transmissions are required for $(r+1)/2 < d < r$. This completes the algorithm for the case (d) and the proof of Theorem 4.1. □

A second theorem is given below. Its proof includes an efficient algorithm for implementing the general index-digit permutation which includes matrix transposition, digit reversal, and the perfect shuffle.

Theorem 4.2. *Any index-digit permutation can be performed on a hypercube of dimension d in* $3d/2$ *parallel transmissions if d is even and* $3(d-1)/2 + 1$ *parallel transmissions of d is odd.*

Proof. (a) Decompose the permutation into disjoint cycles.

(b) If the cycle contains a position in the address, then write the cycle with this position first. For example, if the cycle is (7, 5, 2, 8) and 2 is a position in the address, then rewrite the cycle as (2, 8, 7, 5). If two or more positions are in the address, then either one can be chosen.

(c) As in Property C of the i-cycle, decompose the disjoint cycles into i-cycles. For example, if the cycle is (2, 8, 7, 5) and the pivot digit is in position 3, then this cycle is equivalent to the i-cycles (3, 2)(3, 8)(3, 7)(3, 5)(3, 2) applied from left to right.

The position of each digit in the processor number appears in at most one i-cycle unless it is the first element of a disjoint cycle whose positions are all in the processor number. In the latter case, the position appears in two i-cycles. Therefore the total number of i-cycles that contain positions in the processor number and hence the total number of interprocessor communications is

$$n_t \leqslant d + n_n \tag{4.1}$$

where n_n is the total number of disjoint cycles that do not contain a position in the address field. But $n_n \leqslant (d-1)/2$ if d is odd or $n_n \leqslant d/2$ if d is even. This completes the proof of Theorem 4.2, which also provides an algorithm for implementing index-digit permutations on a hypercube. □

We now turn our attention to the computations that must be performed in each processor following the transmissions that have been defined above. Consider the case in which the sequence is mapped onto the processors in standard order and $r = 8$, $d = 4$. The intermediate orderings are given in Table 4. It can be seen that the computations depend on the step s as well as the processor number p. For example, consider the computation of $X^{(3)}(k_7, i_1, i_2, k_6 \,\vdots\, i_4, i_5, i_0, i_3)$ from $X^{(2b)}(k_7, i_1, i_2, i_6 \,\vdots\, i_4, i_5, i_0, i_3)$ in processor $p = b_3 b_2 b_1 b_0$ (binary). Since the indices to the right of the partition also determine the processor number, $i_4 = b_0$, $i_5 = b_1$, $i_0 = b_2$, and $i_3 = b_3$. Hence from (2.12), the computations in processor p are

$$\begin{aligned} &X^{(3)}(k_7, i_1, i_2, k_6 \,\vdots\, i_4, i_5, i_0, i_3) \\ &\quad = \omega_{128}^{ik_6} \sum_{i_6=0}^{1} X^{(2b)}(k_7, i_1, i_2, i_6 \,\vdots\, i_4, i_5, i_0, i_3)\, \omega_2^{i_6 k_6} \end{aligned} \tag{4.2}$$

where $i = b_1 b_0 b_3 i_2 i_1 b_2$ (binary). Alternatively

$$X^{(3)}(k_7, i_1, i_2, 0, p) = X^{(2b)}(k_7, i_1, i_2, 0, p) + X^{(2b)}(k_7, i_1, i_2, 1, p), \tag{4.3}$$

$$X^{(3)}(k_7, i_1, i_2, 1, p) = \omega_{128}^{i}\left[X^{(2b)}(k_7, i_1, i_2, 0, p) - X^{(2b)}(k_7, i_1, i_2, 1, p)\right]. \tag{4.4}$$

The complex exponentials ω_{128}^{i} can be precomputed and used repeatedly. Also, as in Section 2, the indices k_7, i_1, and i_2 can be combined into a single index.

5. Summary

Several vector-concurrent FFTs were developed for both shared and nonshared multiprocesors. The focus throughout the paper was on the efficient movement of data. Eight FFTs were reviewed in Section 2 including the Cooley–Tukey, Pease, Stockham, and the Stockham variant, together with FFTs that were derived from these by the method of double inversion. In Section 3 these FFTs were used to develop two FFTs for vector multiprocessors with shared memory. The emphasis was on algorithms that use long arrays with elements that are stored consecutively since they utilize the full potential of the vector processor.

On the hypercube the emphasis was on algorithms that minimize interprocessor communication which is known to make a significant contribution to the total computing time. Although the hypercube interconnections facilitate the development of FFTs, their implementation is nevertheless a nontrivial task. Several hypercube FFTs were given for different cases that depend on the length of the sequence as well as the number of processors. It was shown that the number of parallel transmissions for the FFT on a hypercube of dimension d could range from $1.5d + 2$ to $2d + 1$ depending on the case. It was also shown that any index-digit permutation could be performed in less than or equal to $1.5d$ parallel transmissions including a matrix transposition, digit reversal, or perfect shuffle.

Acknowledgment

This work was completed while the author was visiting the Center for Supercomputer Research and Development at the University of Illinois. The author wishes to acknowledge many helpful discussions with individuals at the Center, particularly with Professors Dennis Gannon, Ahmed Sameh, and William Jalby.

References

[1] E.O. Brigham, *The Fast Fourier Transform* (Prentice-Hall, Englewood Cliffs, NJ, 1974).
[2] B. Fornberg, A vector implementation of the fast Fourier transform, *Math. Comput.* **36** (1981) 189–191.
[3] P.M. Flanders, A unified approach to a class of data movements on an array processor, *IEEE Trans. Comput.* **31** (1982) 809–819.
[4] D. Fraser, Array permutation by index-digit permutation, *J. ACM* **22** (1976) 298–308.
[5] D. Gannon and J. Rosendale, On the impact of communication complexity on the design of parallel numerical algorithms, *IEEE Trans. Comput.* **33** (1984) 1180–1194.
[6] D.G. Korn and J.J. Lamboitte, Jr., Computing the fast Fourier transform on a vector computer, *Math. Comput.* **33** (1979) 997–992.
[7] S.L. Johnsson, Communication efficient basic linear algebra computations on hypercube architectures, Yale University, Report No. YALEU/DCS/RR-361, 1985.
[8] O.A. McBryan and E.F. Van de Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Statist. Comput.*, to appear.
[9] M.C. Pease, The indirect binary n-cube microprocessor array, *IEEE Trans. Comput.* **26** (1977) 458–473.
[10] Y. Saad and M.H. Schultz, Data communication in hypercubes, Yale University, Report No. YALEU/DCS/RR-428, Oct. 1985.
[11] P.N. Swarztrauber, Vectorizing the FFTs, in: G. Rodrigue, ed., *Parallel Computations* (Academic Press, New York, 1982) 490–501.
[12] P.N. Swarztrauber, FFT algorithms for vector computers, *Parallel Comput.* **1** (1984) 45–63.
[13] C. Temperton, Fast Fourier transforms and Poisson solvers on the Cray-1, in: C. Jessope, ed., *Infotech State of the Art Report: Supercomputers: 2* (Infotech, 1979) 359–379.