FFT algorithms and their adaptation to parallel processing

Eleanor Chu a,*, Alan George b,1

a Department of Computing and Information Science and Department of Mathematics and Statistics, University of Guelph, Guelph, Ont., Canada N1G 2W1
b Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1

Received 22 September 1997; accepted 5 May 1998
Submitted by D.P. O’Leary

Abstract

The development of the fast Fourier transform (FFT) and its numerous variants in the past 30 years has led to very efficient software and hardware implementations of the transform on uniprocessor computers. In recent years, many researchers have recognized the practical importance of minimizing computing time by parallelizing sequential FFT algorithms in various ways for today’s high-performance multiprocessor computers. This paper presents many FFT variants already proposed by others in a common framework, which illuminates the progress made in parallelizing them to this date. In addition, three new parallel FFT algorithms along with communication complexity results are presented. The proposed algorithms show alternative ways of designing parallel FFT algorithms which feature reduced communication cost and further flexibility in the choices of data mappings. © 1998 Elsevier Science Inc. All rights reserved.

1. Introduction

The discrete Fourier transform (DFT) has played an important role in control, signals and image processing over a long period of time, and the practical importance of fast Fourier transform algorithms (FFTs) has also been well recognized. In the past 30 years, the development of the FFT and its numerous var...
iants has led to very efficient software and hardware implementations of the transform on uniprocessor computers. Since multiprocessor computers became commercially available a decade ago, the parallelization of sequential FFTs on high-performance multiprocessor computers has received the attention of numerous researchers. The variations of sequential FFTs, together with different parallelizing techniques, different multiprocessor architectures, and different network topologies, has led to a large number of parallel FFT variants, each with its own "twist" on things. A survey of the FFT literature suggests that a useful contribution would be to put the various ideas in a common framework. To this end, the concept of index-digit permutation [5,6] and the associated binary address based notation [6,15,18] have been borrowed, modified, and applied in a uniform manner to explain the various ideas which have appeared in the FFT literature. Throughout this paper, examples from the radix-2 FFTs are used to introduce and demonstrate how this notation can bring all FFT variants, sequential or parallel, into a common framework which facilitates the understanding, implementation, and design of the various FFT algorithms. The notation extends to the entire class of radix-2^{k} FFTs, including mixed-radix and split-radix FFTs, in a straightforward way, but is omitted to save space. Three new distributed-memory parallel FFTs along with communication complexity results are also presented. The proposed algorithms show alternative ways of designing parallel FFT algorithms which feature reduced communication cost and further flexibility in the choices of data mappings.

1.1. The discrete Fourier transform (DFT)

The DFT refers to the transformation of $N$ discrete samples $x_0, x_1, \ldots, x_{N-1}$ according to the formula

$$X_r = \sum_{l=0}^{N-1} x_l \omega_N^{rl}, \quad r = 0, 1, \ldots, N - 1,$$

(1)

where the $x_l$'s are sampled at equally spaced intervals in one period from a complex time series, the $X_r$'s are complex numbers which represent a spectrum of the time series resulting from the transform, and $\omega_N$ is the $N$th primitive root of unity; i.e., $\omega_N = e^{-2\pi j/N}$ with $j \equiv \sqrt{-1}$. Although Eq. (1) defines $X_r$ for all $r$, one only needs to consider values for $r = 0$ to $r = N - 1$, since $\omega_N^r = 1$ for $\lambda \geq 1$, which implies that $X_{r+\lambda N} = X_r$. When $N=4$, the DFT of $x_0, x_1, x_2$ and $x_3$ is the product of the matrix $\Omega$ and the vector $x$ in the equation below.

$$
\begin{pmatrix}
X_0 \\
X_1 \\
X_2 \\
X_3
\end{pmatrix}
= 
\begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & \omega_4 & \omega_4^2 & \omega_4^3 \\
1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\
1 & \omega_4^3 & \omega_4^6 & \omega_4^9
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3
\end{pmatrix}
$$

(2)
A straightforward implementation of the DFT defined by Eq. (1) requires $N^2$ complex multiplications and $(N-1)^2$ complex additions in addition to the cost of generating the matrix $\Omega$. The efficient computation of the $N \times N$ matrix $\Omega$ is an important problem in its own right, but beyond the scope of this paper; we shall assume that all $\omega_n^j$ values, which are commonly referred to as twiddle factors, are pre-computed and available to the computer programs implementing the various transform algorithms. One could argue that this is a reasonable assumption to make, since FFT codes are usually applied sequentially to large numbers of vectors, and thus pre-computation of the twiddle factors is an efficient strategy. The argument appears to be valid for single processor machines, and for shared-memory multiprocessors, since access to the twiddle factors is straightforward. As pointed out by Chamberlain [2], this is also an efficient strategy for distributed-memory machines if multiple transforms are performed by each processor all at once.

However, for transforming a single large vector on distributed-memory machines, the best strategy is not at all obvious. If every processor has a copy of all the twiddle factors, then they may consume more store than the data being transformed – a somewhat incongruous circumstance. If this represents an unacceptable amount of storage, then the FFT algorithm must arrange that the (pre-computed) twiddle factors are conveyed among the processors so that they are available when needed. A final option is to compute the twiddle factors "on the fly" when they are needed, which relieves the storage and communication burdens at the expense of additional computation.

The choice of strategy depends on a number of factors: the relative speeds of communication and computation, the amount of memory available compared to the size of the problem being solved, and the algorithm itself. Although specific strategies were considered and compared under special circumstances in Refs. [2,18], their results do not seem to generalize. Due to the large number of algorithms surveyed in this paper, consideration of the twiddle factor distribution for all of them would exceed the space permitted; however, the issue is important, and will be dealt with elsewhere.

1.2. FFT variants and the divide-and-conquer paradigm

The divide-and-conquer paradigm is fundamental to the FFT algorithm. The three major steps of the divide-and-conquer paradigm are

Step 1. Divide the problem into two or more subproblems of smaller size.

Step 2. Solve each subproblem recursively by the same algorithm. Apply the boundary condition to terminate the recursion when the sizes of the subproblems are small enough.

Step 3. Obtain the solution for the original problem by combining the solutions to the subproblems.
The radix-2 FFT is traditionally introduced as a recursive algorithm obtained from dividing the given problem (and each subproblem) into two subproblems of half the size. Within this framework, there are two commonly used FFT variants which differ in the way the two half-size subproblems are defined. They are referred to as the DIF (decimation in frequency) FFT and the DIT (decimation in time) FFT. To conserve space, only the DIF version is considered here. However, the notation and the approach in this paper apply equally well to the DIT case.

The remainder of this article consists of the following sections. The derivation of the radix-2 DIF FFT algorithm is presented in Section 2. A binary address based notation to assist in describing algorithms is introduced in Section 3. Various data mapping strategies for parallelizing FFTs on multiprocessor computers are discussed in Section 4. Specific parallel FFTs in the literature are explained and presented in a unified manner in Section 5. Three new distributed-memory parallel FFTs along with communication complexity results are presented in Section 6. Section 7 contains concluding remarks.

2. The radix-2 decimation-in-frequency (DIF) FFT

As its name implies, the radix-2 DIF FFT algorithm is obtained by decimating the output frequency series into an even-indexed set \( \{X_{2k}\} | k = 0, \ldots, N/2 - 1 \} \), and an odd-indexed set \( \{X_{2k+1}\} | k = 0, \ldots, N/2 - 1 \} \). To define the two half-size subproblems, Eq. (1) is rewritten as

\[
X_r = \sum_{l=0}^{N/2-1} x_l \omega_N^{rl} + \sum_{l=N/2}^{N-1} x_l \omega_N^{rl} \\
- \sum_{l=0}^{N/2-1} x_l \omega_N^{rl} + \sum_{l=0}^{N/2-1} x_{l+N/2} \omega_N^{rl(l+N/2)} \\
= \sum_{l=0}^{N/2-1} (x_l + x_{l+N/2} \omega_N^{N/2}) \omega_N^{rl}, \quad r = 0, 1, \ldots, N - 1
\]

(3)

The following identities involving the twiddle factors are used in what follows.

\[
(\omega_N)^{N/2} = -1, \quad \omega_{N/2} = \omega_N^2, \quad \omega_N^N = 1.
\]

(4)

For \( r \) even, using Eq. (4) in Eq. (3) yields
\[ X_{2k} = \sum_{l=0}^{N/2-1} \left( x_l + x_{l+N/2} \omega_N^{2kl} \right) \omega_N^{2kl} \]
\[ = \sum_{l=0}^{N/2-1} (x_l + x_{l+N/2}) \omega_N^{kl}, \quad k = 0, 1, \ldots, N/2 - 1. \]  

(5)

Defining \( Y_k = X_{2k} \) and \( y_l = x_l + x_{l+N/2} \) yields the half-size subproblem

\[ Y_k = \sum_{l=0}^{N/2-1} y_l \omega_N^{kl}, \quad k = 0, 1, \ldots, N/2 - 1. \]  

(6)

Similarly, for \( r \) odd, using (4) in (3) yields

\[ X_{2k+1} = \sum_{l=0}^{N/2-1} \left( x_l + x_{l+N/2} \omega_N^{(2k+1)N/2} \right) \omega_N^{(2k+1)l} \]
\[ = \sum_{l=0}^{N/2-1} \left( (x_l - x_{l+N/2}) \omega_N^{l} \right) \omega_N^{kl}, \quad k = 0, 1, \ldots, N/2 - 1. \]  

(7)

Defining \( Z_k = X_{2k+1} \) and \( z_l = (x_l - x_{l+N/2}) \omega_N^l \) yields the second half-size problem

\[ Z_k = \sum_{l=0}^{N/2-1} z_l \omega_N^{kl}, \quad k = 0, 1, \ldots, N/2 - 1. \]  

(8)

Note that because \( X_{2k} = Y_k \) in (6) and \( X_{2k+1} = Z_k \) in (8), no more computation is needed to obtain the solution for the original problems after the two subproblems are solved. Therefore, in the implementation of the DIF FFT, the bulk of the work is done during the subdivision step; i.e., the set-up of appropriate subproblems, and there is no combination step. Consequently, the computation of \( y_l = x_l + x_{l+N/2} \), and \( z_l = (x_l - x_{l+N/2}) \omega_N^l \), completes the first (subdivision) step.

The computation of \( y_l \) and \( z_l \) in the subdivision step as defined above is referred to as the Gentleman-Sande butterfly in the literature, and is depicted by the annotated butterfly symbol in Fig. 1.

Let \( T(N) \) be the arithmetic cost of computing the radix-2 DIF FFT of size \( N \), and, for some constant \( c \), let \( cN \) be the cost of computing \( y_l \)‘s and \( z_l \)’s in the

\[ y_l = \left( x_l + x_{l+N/2} \right) \omega_N^l \]
\[ z_l = \left( x_l - x_{l+N/2} \right) \omega_N^l \]

Fig. 1. The Gentleman-Sande butterfly.
subdivision step. Then \( T(N) \) is the sum of the subdivision cost and the cost of computing (6) and (8), which implies

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1, \\
2T(N/2) + cN & \text{if } N = 2^n > 1.
\end{cases}
\]

Assuming that \( N = 2^n \) for the radix-2 FFT, the solution to the equation above is

\[ T(N) = cN \log_2 N. \quad (9) \]

3. A binary address based notation for butterfly operations

In the abstract, the equations introduced in the previous sections completely describe the computations. However, in practise, the computations are normally performed in place in a one-dimensional array, with new values overwriting old values as implied by the butterfly in Fig. 1. Throughout this paper, it is assumed that the input data \( x_l, 0 \leq l < N \) are stored in the one-dimensional array \( a \) in natural order. \(^2\) Since these computations are performed repeatedly on subproblems of various sizes, determining the location of \( X_r \) in the array at the end of the computation is not immediately obvious. The butterfly notations do not explicitly relate the locations of the input data to the locations of the output data; for example, the Gentleman-Sande butterfly represents only the first subdivision step of the recursive DIF FFT algorithm. Thus, there is a gap between the elegant (but somewhat implicit) butterfly notations and the detailed specification of the positions of the outputs \( X_r \), contained in the repeatedly modified array. The purpose of this section is to describe a simple notation to assist in this specification. It will also facilitate the adaptation of these FFT algorithms to parallel processing, which is the focus of this paper. The notation is not entirely new, but rather a modest embellishment and extension of the ideas that have been used in the past [6,15,18]. The radix-2 DIF FFT introduced in Section 2 is used as a vehicle for introducing the notation.

3.1. Iterative form of the radix-2 DIF FFT

Recall that the Gentleman-Sande butterfly represents the first subdivision step. Because there is no combination step, it is straightforward to derive an iterative algorithm by simply subdividing each resulting subproblem iteratively

\(^2\) The input data \( x_0, x_1, \ldots, x_{N-1} \) are said to be in “natural order” if \( x_l \) and \( x_{l+1} \) are stored in consecutive locations in \( a \) for all \( 0 \leq l \leq N-2 \). Similarly, the output data \( X_0, X_1, \ldots, X_{N-1} \) are said to be in natural order if \( X_r \) and \( X_{r+1} \) are stored in consecutive locations in \( a \) for all \( 0 \leq r \leq N-2 \).
until the problem size is one, as shown in the pseudo-code algorithm below. It is assumed that the twiddle factors are pre-computed and stored in an array \( w \), with \( w[i] = \omega^i_N \), \( 0 \leq i < N/2 \).

**Algorithm 3.1. The iterative DIF FFT algorithm in pseudo-code**

```plaintext
begin
  NumOfSubProblems := 1
  SubProblemSize := N
  while SubProblemSize > 1 do
    Subdivide each subproblem
    HalfSize := SubProblemSize/2
    for K := 0 to NumOfSubProblems - 1 do
      JFirst := K * SubProblemSize
      JLast := JFirst + HalfSize - 1
      Jtwiddle := 0
      for J := JFirst to JLast do
        W := w[Jtwiddle]; Temp := a[J]
        a[J] := Temp + a[J + HalfSize]
        a[J + HalfSize] := W * (Temp - a[J + HalfSize])
        Jtwiddle := Jtwiddle + NumOfSubProblems
      end for
    end for
    NumOfSubProblems := 2 * NumOfSubProblems
    SubProblemSize := HalfSize
  end while
end
```

Since the initial \( SubproblemSize = N = 2^n \), the variable \( HalfSize \) takes on the values \( 2^{n-1}, 2^{n-2}, \ldots, 2, 1 \); the distance between ado \( a[J] \) and \( a[J + HalfSize] \) is always a power of 2. Thus, denoted as binary numbers \( i_{n-1} \cdots i_1 i_0 \), \( J \) and \( J + HalfSize \) differ only in bit \( i_k \) when \( HalfSize = 2^k \) and the algorithm can be expressed in terms of binary addresses as shown below.

**Algorithm 3.2. The iterative DIF FFT algorithm in terms of binary addresses**

```plaintext
begin
  k := n - 1
  while k \geq 0 do
    Subdivide each subproblem
    Apply Gentleman-Sande butterfly computation
    to all pairs of array elements whose
    binary addresses differ in bit \( i_k \)
    k := k - 1
  end while
end
```
For $N = 32 = 2^5$, the DIF FFT algorithm as described above consists of five stages involving the sequence $k = 4, 3, 2, 1, 0$, with butterflies being applied to all pairs of array elements whose binary addresses differ in bit $i_k$. Since the algorithm can be expressed in terms of binary addresses, it is convenient to adopt binary notation for the subscripts of $a$ and $x$. Thus, in what follows the notation $x_{i_4i_3i_2i_1i_0}$ will mean $x_r$, where $i_4i_3i_2i_1i_0$ is the binary representation of $r$. Similarly, $a[i_4i_3i_2i_1i_0]$ refers to the element $a[m]$, where the binary representation of $m$ is $i_4i_3i_2i_1i_0$. Then a shorthand notation for describing this 5-step process being applied to $a[m]$ is the sequence

$$i_4i_3i_2i_1i_0 \quad \tau_4 i_3i_2i_1i_0 \quad \tau_4 \tau_3i_2i_1i_0 \quad \tau_4 \tau_3 \tau_2i_1i_0 \quad \tau_4 \tau_3 \tau_2 \tau_1i_0,$$

where $i_k$ indicates that the butterfly computation involving the pairs in $a$ different in bit $i_k$ is being performed in the current stage, and $\tau_k$ indicates that the corresponding butterfly operations were performed in a previous stage. The transformation is completed when butterflies in all stages have been performed.

Which element of $X$ will be found in $a[i_4i_3i_2i_1i_0]$? If Eqs. (6) and (8) are interpreted as implied by the butterfly in Fig. 1, the top half of $a$ will contain the even-numbered $X$'s after all stages are completed, and the bottom half will contain the odd-numbered $X$'s. Thus, if $i_4 = 0$, then $a[i_4i_3i_2i_1i_0]$ is in the top half of $a$ and will ultimately contain an even-numbered $X$, and if $i_4 = 1$, then $a[i_4i_3i_2i_1i_0]$ is in the bottom half of $a$ and will ultimately contain an odd-numbered $X$. This in turn means that after the butterfly step identified by operation $i_4i_3i_2i_1i_0$ is completed, it is evident that $a[i_4i_3i_2i_1i_0]$ will ultimately contain output $X_r$, where the rightmost bit of the binary representation of $r$ is $i_4$. By exactly the same argument, applied recursively to both the top and bottom halves of the array, we can conclude that the second from the rightmost bit of $r$ will be $i_3$. Repeating the argument three more times on successively smaller portions of the array $a$ yields the conclusion that $a[i_4i_3i_2i_1i_0]$ will finally contain $X_{i_0i_1i_2i_3i_4}$. In other words, $X_{i_0i_1i_2i_3i_4}$ will occupy the position originally occupied by $x_{i_4i_3i_2i_1i_0}$. That is, the output $X$ is in bit-reversed order.

Fig. 2 displays the initial contents of $a$, the five stages of butterfly computation together with the associated twiddle factors, and the resulting bit-reversed sequence. To help identify the pairs of subproblems resulting from every stage of butterfly computation, the subproblems involving $a[0]$, which initially contains the input element $x_0$, are highlighted. Note that for an input sequence of $N$ elements, there are exactly $N/2$ butterflies in each of the $\log_2 N$ stages of butterfly computations.
3.2. The effect of permutations

The algorithm described in the previous section is correct only because $x_{i_4i_3i_2i_1i_0}$ is initially contained in $a[i_4i_3i_2i_1i_0]$. However, the following program is correct, regardless of where $x_{i_4i_3i_2i_1i_0}$ is found in $a$.  

![Diagram of DIF FFT with naturally ordered input and bit-reversed output.](image)

Fig. 2. The DIF FFT with naturally ordered input and bit-reversed output.
Algorithm 3.3. The iterative DIF FFT applied to the \( x \) elements

begin

\[ k := n - 1 \]

Initial problem size \( N = 2^n \)

while \( k \geq 0 \) do

Subdivide each subproblem

Apply Gentleman-Sande butterfly

computation to all pairs of elements of

\( x \) whose (binary) subscripts differ in bit \( i_k \)

\[ k := k - 1 \]

end while

end

Thus, the input data could be permuted arbitrarily, and as long as the butterflies are applied correctly to the data, the correct answers would be obtained. Moreover, the element of \( a \) that initially contained \( x_{i_4i_3i_2i_1i_0} \), would contain \( X_{i_4i_3i_2i_1i_0} \) at the end of the computation.

For example, suppose the objective is to find an initial ordering of the input so that the resulting output is in natural order. The observation above implies that initially, \( a[i_0i_1i_2i_3i_4] \) should contain \( x_{i_4i_3i_2i_1i_0} \), since at the end of the computation one wants \( a[i_0i_1i_2i_3i_4] \) to contain \( X_{i_4i_3i_2i_1i_0} \). That is, the input should be placed in bit-reversed order before the computation begins.

Again for the case \( N = 32 \), a shorthand notation describing the 5-step process, together with the initial permutation to bit-reversed order, is the sequence

\[ i_4i_3i_2i_1i_0 \quad i_0i_1i_2i_3i_4 \quad i_0i_1i_2i_3i_4 \quad i_0i_1i_2i_3i_4 \quad i_0i_1i_2i_3i_4. \]

Here the sequence begins with \( i_4i_3i_2i_1i_0 \), which is intended to imply that \( x_{i_4i_3i_2i_1i_0} \) is assumed to be in \( a[i_4i_3i_2i_1i_0] \) as before; the notation \( i_0i_1i_2i_3i_4 \) is intended to imply that \( x_{i_4i_3i_2i_1i_0} \) has been permuted to \( a[i_0i_1i_2i_3i_4] \) before the first butterfly computation is performed. That is, \( i_4i_3i_2i_1i_0 \) always represents the binary representation of the subscripts of \( x \); the order in which the bits appear, or are permuted during the computation, refer to movements that \( x_{i_4i_3i_2i_1i_0} \) or its derivatives undergo in \( a \) during the computation.

3.3. Arranging that input and output are both in natural order

When input \( x \) and output \( X \) are both in natural order, the algorithm is referred to as an “ordered” FFT in the literature. The key to understanding what is required is to view each butterfly computation as consisting of one permutation step followed by one in-place computation step. These permutation steps record the initial input as well as the input to each subsequent subproblem, and the notation introduced above can be used to describe this process in a natural way. Again using the example above, with the input in the natural order,
the first in-place butterfly is denoted by $i_{4}i_{3}i_{2}i_{1}i_{0}$. If this butterfly operation is preceded by permuting the data in $a[i_{4}i_{3}i_{2}i_{1}i_{0}]$ to $a[i_{3}i_{2}i_{1}i_{0}i_{4}]$, it is natural to use $i_{3}i_{2}i_{1}i_{0}i_{4}$ to denote the combined effect. Since the permutation step cannot be done in-place, two arrays of size $N$ are used to alternately store the data; this doubles the storage requirement. As usual, assuming that $x$ is initially contained in $a$ in the natural order, a second array $b$ would alternately contain the data. The entire computation process, along with the use of two arrays, is depicted below.

Since the Gentleman-Sande butterfly does not permute the input when the problem size is 2, no permutation step precedes the last butterfly operation, so there is no relocation of data from $a$ to $b$ in the last step of the process.

Another class of ordered FFTs perform in-place permutation and consequently do not need a second array; they are the so-called “self-sorting in-place” algorithms. This class contains variants of the prime-factor algorithms [1,12,16] and a radix-2 FFT [8]. Recently, this class has been further extended to include self-sorting in-place radix-3, radix-4, and radix-5 FFTs [17]. To keep our discussion brief, only the radix-2 algorithm is reviewed here. Using the notation developed earlier, the process of applying the self-sorting in-place radix-2 FFT to array $a$ is depicted below for $N = 32$.

Observe that the permutation always involves bits in symmetric positions: e.g., in step 1, the leftmost bit $i_{4}$ switches with the rightmost bit $i_{0}$ and in step 2, bit $i_{3}$, the second bit from the left end, switches with bit $i_{1}$, the second bit from the right end. Accordingly, the ordering of the bits is “reversed” after only two steps, and the permutation can be implemented using “pairwise” interchanges. The contents in $a[i_{4}i_{3}i_{2}i_{1}1]$ and $a[i_{4}i_{3}i_{2}i_{1}0]$ are switched in step 1 and the contents in $a[i_{4}i_{0}i_{2}i_{1}1]$ and $a[i_{4}i_{0}i_{2}i_{1}0]$ are switched in step 2. Since each pairwise interchange can be done using a single temporary location, the array $b$ is not needed.

4. Mapping data to processors

For purposes of this paper, it is assumed that the multiprocessor available has $P = 2^{d}$ processors, and each processor has its own local memory. That is, the machine in question is a distributed-memory multiprocessor, where each processor is connected to the others via a communication network with a prescribed topology. A common topology is a hypercube, but others such as a regular grid or a ring are also commonly used.
A key step in parallelizing the FFT on such multiprocessor computers is the mapping of the elements of \( x \) to the processors. Since \( N = 2^n \) and \( P = 2^d \), it is natural to select \( d \) of the \( n \) bits in the binary representation of the subscripts of \( x \) to create a unique \( d \)-bit binary processor ID number specifying the data-to-processor mapping. While any set of \( d \) bits could be chosen, there seems to be no compelling reason not to choose consecutive bits. Different choices of bits yield different mappings. Then \( n - d + 1 \) cyclic block mappings (CBMs) are shown in Fig. 3, where \( n = 5 \) and \( d = 2 \), and the array locations mapped to pro-

Fig. 3. The \( n - d + 1 \) cyclic block mappings for \( N = 2^n = 32 \) and \( P = 2^d = 4 \).
cessor $P_0$ are shaded to highlight the cyclic nature with various block sizes. When the block size is $N/P$ or one, the mappings are respectively referred to as block or cyclic mappings in the literature.

In what follows the notation $i_{n-1} \ldots i_{k+1} | i_k \ldots i_{k-d+1} | i_{k-d} \ldots i_0$ is used to denote that bits $i_k \ldots i_{k-d+1}$ have been chosen to specify the data-to-processor allocation. Each processor is always assigned $N/P$ data elements; i.e., this class of mappings ensures even data distribution. Each processor can always compute the butterflies involving the $N/P$ local data elements independently, because these data have the same $d$ processor address bits. Thus, no inter-processor communication is required to compute the butterflies involving the bits specified by the braces below.

In general, since any $d$ bits can be used to form the processor ID number, it is easier to recognize the generic communication pattern if one concatenates the bits representing the ID into one group called “PID”, and refers to the remaining $n - d$ bits, which are concatenated to form the local array address, as “Local $M$”. For the class of CBM mappings, one can use the following equivalent notation, where the leading $d$ bits are always used to identify the processor ID number.

$$|\text{PID}| \text{ Local } M = |i_k \ldots i_{k-d+1} | \overline{i_{n-1} \ldots i_{k+1} i_{k-d} \ldots i_0}.$$

In either case, the $d$ consecutive bits are marked by the symbol “$|$” at both ends. The two notations are equivalent and both are used in the text.

Computing the butterflies involving the address bits used to define the processor ID number will involve exchange of data between processors whose ID numbers differ in exactly one bit. Of course these processors may or may not be physically adjacent, depending on the network topology. For example, if the $p$ processors form a hypercube network, data communication between such a pair is always between neighbouring processors. If the $p$ processors form a linear array or a two-dimensional grid, such a pair of processors can be physically many hops apart, and the simultaneous data exchange between all pairs can cause traffic congestion in the network.

CBM mappings have received considerable study in the literature dealing with parallelizing FFTs [2-4,7,9,14,15,18,19]. These works vary in the choice of the blocksize, whether DIF or DIT transforms are used, and whether the input and/or output is in unordered (reverse-binary) or in natural order, and so on. The purpose of this paper is to bring all of these treatments into a common framework.

5. Parallel FFTs

Recall from Section 3 that one may apply the algorithm to the naturally ordered input and obtain the output in bit-reversed order, or one may permute
the input into bit-reversed order in advance and obtain the \( X \) values in natural order, or one may apply permutations during the computation so that naturally ordered output is obtained from naturally ordered input. A fourth option that was not considered is to use naturally ordered input, and then bit-reverse the output after the computation is complete in order to obtain naturally ordered output. The task of bit-reversing a vector is an interesting problem in its own right, with many variations and intricacies [10]. However, the problem is outside the scope of this paper. The second and last choices are simply special cases of the third, where all the permutations are applied either in advance of the computation or after all computation has been completed. Moreover, only one of a large number of permutation variations was considered. Indeed, given naturally ordered input, any binary permutation of \( X \) can be obtained, and in numerous different ways. For example, suppose after the computation one requires that \( a[i_2 i_3 i_0 i_4 i_1] \) contain \( X_{i_0 i_1 i_2 i_3 i_4} \). Two of several possible schemes are shown below, using the notation developed earlier.

Of course it should also be evident that assuming that the input is in natural order is simply a matter of expositional convenience. Any binary permutation of the input can be handled using the same shorthand notation. As noted earlier, the term "ordered" FFT in the literature refers to an algorithm which transforms naturally ordered \( x \) to naturally ordered \( X \); the algorithms resulting from any other permutation of either input elements or output elements are all referred to as "unordered" FFTs. Therefore, the FFT with bit-reversed input or bit-reversed output is also an unordered FFT.

5.1. Parallel FFTs without inter-processor permutations

In general, in the parallel context, if permutations are allowed, it may turn out that part of a processor's complement of data may migrate to another machine. This subsection deals with the case where no permutations are performed. Using the notation and the example of the previous section, computations involving the four mappings are depicted below. The notation \( \leftrightarrow \) indicates that \( N/P \) data elements must be exchanged between processors in advance of the butterfly computation. The symbol \( i_k \) identifies two things: first, it indicates that the incoming data from another processor are the elements whose subscripts differ from a processor's own data in bit \( i_k \); second, it indicates that all pairs of processors whose binary ID number differ in bit \( i_k \) send each other a copy of their own data.
Since the input sequence is in natural order and the output is in bit-reversed order, the processor initially allocated \(x_{i_1i_0} \Delta \) will finally have \(X_{i_1i_0} \Delta \) if interprocessor permutation is not allowed. For example, if the initial mapping is \(i_4i_3i_2i_1i_0 \), processor \(P_0 \) initially contains \(x_0, x_1, x_2, \ldots, x_7 \) in their natural order as depicted in Fig. 3; when the parallel FFT ends, processor \(P_0 \) contains \(X_0, X_1, X_2, \ldots, X_7 \), which are the first eight elements in the output array in Fig. 2. In this case, \(x \) and \(X \) are said to be comparably mapped to the processors [11], p.160. Note that the initial mapping is a CBM mapping of naturally ordered \(x \), but the final mapping is not a CBM mapping of naturally ordered \(X \).

Butterfly computations will cause communication between processors if the two input elements are stored in different processors. Since both input elements are needed to update each of them, the two processors involved must exchange the \(N/P \) data elements for each other to update their local data. As shown above, butterflies require data to be exchanged in exactly \(d = \log P \) stages, regardless of the blocksize used in the mapping. (In the example, \(d = 2 \).) Algorithms of this type are described in Refs [2,4,9]. This is also version 1 of the distributed-memory FFTs in Ref. [11, pp. 156–162.]

A consequence of this scheme is that one half of the processors update their local data according to a formula not involving the twiddle factor; i.e., they each update \(N/P \) elements according to \(y_i = (x_i + x_{i+N/2}) \). The other half of the processors update their local data according to a formula involving the multiplication of a pre-computed twiddle factor; i.e., they each update \(N/P \) elements according to \(z_i = (x_i - x_{i+N/2}) \omega_i \). Thus, the arithmetic workload is not evenly divided among all processors unless each processor computes both \(y_i \) and \(z_i \).

To balance the arithmetic workload, Walton [19] proposed a variant of this type of communication which allows each pair of processors to exchange \(\frac{1}{2} (N/P) \) elements twice: once before the butterfly computation, and once after the
butterfly computation. The second communication is needed for each processor to get back the \( \frac{1}{2}(N/P) \) elements updated by the other processor. The same idea is employed in version 2 of the two-processor distributed-memory FFT in Ref. [11], p. 158. Compared to the scheme above, the amount of data exchanged between each pair of processors remains the same, but the number of message exchanges is doubled and the notation \( \leftrightarrow \) now denotes two exchanges of \( \frac{1}{2}(N/P) \) elements between all pairs of processors. Unfortunately, the gain from balancing the arithmetic workload may be offset by doubling the number of exchanges if the startup cost for each message exchange is high. The arithmetic workload can be balanced, and the amount of data exchanged can be halved without doubling the number of exchanges if inter-processor permutations are allowed, as discussed in the next section.

5.2. Parallel FFTs with inter-processor permutations

As suggested in Refs. [3, 7, 15, 18], if processors are not required to keep and update only the same data throughout the entire computation, then every other concurrent message exchange can be eliminated from Walton's scheme [19]. To introduce this concept, consider the butterfly computations depicted by

\[
\begin{align*}
|i_4i_3i_2i_1i_0|_\Delta & |i_2i_3i_4i_1i_0|_\Delta & |i_2\tau_4i_3i_1i_0|_\Delta & |\tau_3\tau_4i_2i_1i_0|_\Delta & |\tau_3\tau_4i_2i_1i_0|_\Delta & |\tau_3\tau_4i_2i_1i_0|_\Delta
\end{align*}
\]

The notation \( \leftrightarrow \) denotes one concurrent message exchange of \( \frac{1}{2}(N/P) \) data elements between all pairs of processors, which is shown to occur in the butterfly stages involving bits which form the processor ID number. After data are distributed to individual processors according to the initial mapping \( |i_4i_3i_2i_1i_0| \), the element \( x_{i_4i_3i_2i_1i_0} \) in \( A[i_4i_3i_2i_1i_0] \) can be found in \( A[i_2i_1i_0] \) in processor \( P_{i_1} \). When bit \( i_4 \) in the PID and bit \( i_2 \) in the local \( M \) switch their positions, the mapping is changed to \( |i_2i_3i_4i_1i_0| \), which means that the data in \( a[i_4i_3i_2i_1i_0] \) can now be found in \( A[i_4i_1i_0] \) in \( P_{i_1} \). To identify the one half of data each processor must send out, the symbol \( \Delta \) is used to label two different bits: the bit \( i_3 \), which has just been permuted from PID into Local \( M \), and the bit \( i_1 \), which has just been permuted from Local \( M \) to the PID. In the example above, \( i_4 \) and \( i_2 \) have switched their respective positions in the PID and the Local \( M \), \( i_3 \) and \( \tau_4 \) have switched their respective positions in the PID and the Local \( M \), and \( i_2 \) and \( \tau_3 \) have switched their respective positions in the PID and the Local \( M \). Because \( i_k \)

\( ^3 \) While the symbol \( i_k \) in the PID still identifies the pair of processors, this symbol alone is not sufficient to identify the one half of data each processor needs to send out. To avoid redundancy, this notation will be modified in the next section when this idea is further exploited.
was in PID before the switch, $i_k = 1$ in one processor, and $i_k = 0$ in the other processor. On the other hand, because $i_l$ was in Local $M$ before the switch, $i_l = 0$ for half of the data, and $i_l = 1$ for another half of the data. Consequently, the value of $i_k$, the PID bit, is equal to $i_l$, the local $M$ bit, for half of the data elements in each processor, and the notation which represents the switch of these two bits identifies both the PID of the other processor as well as the data to be sent out or received. To depict exactly what happens, the data exchange between two processors and the butterfly computation represented by

$$|i_2i_3| |i_4i_5i_0\rangle$$

(discussed above) is shown in Fig. 4. While such a detailed schematic diagram can be drawn to trace the entire parallel FFT process stage by stage, there is no need to do so because all information can be deciphered from the simple notation introduced above.

Note that in the example above, element $x_{i_0i_1i_2i_3i_4i_5i_6}$ is initially contained in $A[i_2i_3i_0]$ in processor $P_{ik}$, and it is finally located in processor $P_{ik}$ at the same relative location. Note also that the number of concurrent exchanges $d + 1$. A second example, using the "|PID| Local $M$" notation with a different mapping, suggests the general situation.

$$|i_2i_1| |i_4i_5i_0\rangle |i_2i_1| |i_4i_5i_0\rangle |i_0i_1| |i_4i_5i_0\rangle |i_0i_1| |i_4i_5i_0\rangle$$

Note that in this example, the order of the data within each processor again has been preserved, but the data in the aggregate has migrated from processor $P_{ik}$ to processor $P_{ik}$, and the number of concurrent exchanges remains $d + 1$.

It is straightforward to develop a communication scheme so that this is true for all CBM mappings: the data within each processor remains in the same relative order, but its processor location is changed from a processor with ID = $i_{k-1} \ldots i_{k-d+1}$ to another processor with ID = $i_{k-d+1}i_{k-1} \ldots i_{k-d+2}$; i.e., the $d$ bits are cyclic-shifted to the right by one position. In all cases, the number of concurrent exchanges remains $d + 1$. The communication schemes described in Ref. [3,7,15,18] may be viewed as variations of this basic scheme, and can all be treated within the same framework.

---

4 They are also called parallel transmissions in the literature [15,18]. In Ref. [3] the authors caution that the actual protocol for message exchange may vary among machines. For example, on the Intel iPSC/860 [13], the duplex bandwidth of a pair-wise data exchange can be utilized only if both nodes start sending the data simultaneously. If one node starts before a second one, the second node must wait until the first one completes its transfer. Thus the communication is reduced to simplex mode. In Ref. [11] the communication overhead is modeled assuming that all exchanges are asynchronous and they are thus performed in the simplex mode.
Fig. 4. Butterfly computation (Stage 1) with data migration between P0 and P2.

Note also that in the first example, the PID bit in question is always exchanged with the leftmost bit in the Local M, which is referred to as the "pivot" in Refs. [15,18]. However, in the second example, the PID bit in question is always exchanged with the rightmost bit in the Local M. Clearly, the so-called "pivot" could be arbitrarily chosen, if one so desires, from the bits of the Local M. Since the ID number is formed by consecutive bits, whenever a PID bit is permuted into the local pivot position, it will be exchanged with the next PID bit and occupy the latter's position back in the PID field. After d exchanges, we have the following scenario: the rightmost PID bit is in the Local M, and the pivot \( i_p \) or \( \tau_p \) from Local M is still in the leftmost position in PID. Therefore, one more permutation involving these two bits will get \( i_p \) or \( \tau_p \) back into its original position in the Local M, and the rightmost PID bit would be cyclic-shifted into the leftmost position in the PID as shown below.

\[
\begin{align*}
|\underbrace{\tau_{k-d} \tau_k \ldots \tau_{k-d+2}}_\Delta | & \underbrace{|\tau_n-1 \ldots \tau_{k+2} \tau_{k+1}}_\Delta | i_{k-d} \ldots i_p+1 i_p i_{p-1} i_{p-1} \ldots i_1 i_0 \end{align*}
\]
Observe that as long as the PID is not formed by the leftmost \( d \) bits, there would be at least one \( \tau_p \) bit available when the butterfly computation reaches any PID bit. One thus has the option of using a \( \tau_p \) (instead of \( i_p \)) bit as the pivot. In this case, the \( \tau_p \) bit may stay as the leftmost bit in the PID if the local data is not required to remain together in one processor, and one concurrent message exchange can be saved, with the final mapping determined by

\[
\begin{bmatrix}
\tau_k \ldots \tau_{k-d+2} \\
\tau_{n-1} \ldots \tau_{p+1} \tau_p \tau_{p-1} \ldots \tau_{k+2} \tau_{k+1} \ \iota_{k-d} \ldots \iota_0
\end{bmatrix}
\]

The PID in the mapping above is no longer formed by consecutive bits.

In all cases, the final \( |\text{PID}| \text{ Local } M \) is a permutation of the \( n \) bits, which uniquely determines the data mapping of \( X_{k_1 k_2 \ldots k_{n-1}} \). In certain contexts, it is important to have the output array \( X \) mapped to the processors in a specific way to facilitate subsequent computations. For example, the final distribution of \( X \) is required to be identical to the initial one for \( x \) in the solution of partial differential equations using spectral techniques [3]. This has motivated the development of a number of algorithms which are now reviewed.

5.3. Parallel FFTs in the literature

In this section some parallel FFTs in the literature are reviewed. To facilitate the discussion, the same example (when it is possible) is used to provide a summary of some distributed-memory parallel FFT literature in Table 1. The basic parallel FFT without inter-processor permutation underlies the work in Refs. [2,4,9,19,20], and the communication cost is either \( d \) concurrent exchanges of \( N/P \) elements or \( 2d \) concurrent exchanges of \( \frac{N}{2}(N/P) \) elements. The final mapping is identical to the initial mapping in all of the algorithms in Refs. [2,4,9,19]. However, because the output element \( X_{k_1 k_2 \ldots k_{n-1}} \) overwrites \( x_{k_1 k_2 \ldots k_{n-1}} \) in its initial location in the same processor, the resulting mapping is not a CBM of \( X \). These algorithms parallelize an unordered sequential FFT, and thus are referred to as unordered parallel FFTs in the literature, and they are so labelled in Table 1. The re-coding of PID (by Gray code) in Ref. [2] and the use of radix-4 FFT in Ref. [4] do not affect the communication scheme, because all processors are identical and the data are accessed in a similar pattern by all radix-\( 2^d \) FFTs. Furthermore, the choice of a different local FFT in Ref. [4] is independent of the choice of the communication algorithm.

In Ref. [20], the PID part of the initial mapping is preserved in the final mapping as dictated by the communication scheme, but the bits which form the Local \( M \) are bit-reversed in the final mapping because each processor applies an
Table 1
Parallel FFTs in the literature:
Some one-dimensional (1-D) parallel FFTs using $P = 2^d = 4, 8, \text{ or } 16$ processors and input data: $N = 2^n = 32$

<table>
<thead>
<tr>
<th>References for the examples</th>
<th>Initial mapping of $x_{i_1 i_2 i_3 i_4}$</th>
<th>Final mapping of $X_{i_1 i_2 i_3 i_4}$</th>
<th># Concurrent exchanges</th>
<th>Message length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walton [19]</td>
<td>$</td>
<td>i_4 i_3 i_2 i_1 i_0</td>
<td>$ (block)</td>
<td>$</td>
</tr>
<tr>
<td>Jamieson et al. [7]</td>
<td>$</td>
<td>i_3 i_2 i_1 i_0 i_4</td>
<td>$ (cyclic)</td>
<td>$</td>
</tr>
<tr>
<td>Swarztrauber [15]</td>
<td>$</td>
<td>i_4 i_3 i_2 i_1 i_0</td>
<td>$ (block)</td>
<td>$</td>
</tr>
<tr>
<td>Chamberlain [2]</td>
<td>$</td>
<td>\text{Gray} [i_4 i_3 i_2]</td>
<td>i_1 i_0</td>
<td>$ (block w. PID in Gray code)</td>
</tr>
<tr>
<td>Tong and Swarztrauber [18]</td>
<td>$</td>
<td>i_2 i_1 i_0 i_4 i_3</td>
<td>$ (cyclic)</td>
<td>$</td>
</tr>
<tr>
<td>Johansson and Krawitz [9]</td>
<td>$</td>
<td>i_4 i_3 i_2 i_1 i_0</td>
<td>$ (block)</td>
<td>$</td>
</tr>
<tr>
<td>Dubey et al. [3]</td>
<td>$</td>
<td>i_4 i_3 i_2 i_1 i_0</td>
<td>$ (block)</td>
<td>$</td>
</tr>
<tr>
<td>Yang [20]</td>
<td>$</td>
<td>i_1 i_0</td>
<td>i_4 i_3 i_2</td>
<td>$ (cyclic)</td>
</tr>
<tr>
<td>Fabbretti et al. [4] (radix-4 and local split-radix)</td>
<td>$</td>
<td>i_4 i_3 i_2 i_1 i_0</td>
<td>$ (block)</td>
<td>$</td>
</tr>
</tbody>
</table>
ordered (sequential) FFT to its local data. Since the initial mapping is restricted
to be cyclic, the resulting mapping is, in general, given by

$$|d\text{ PID bits}|i_{d+1}i_{d+2} \ldots i_{n-2}i_{n-1},$$

where the PID preserves its initial arrangement: $i_{d-1}i_{d-2} \ldots i_0$. Since the Local
$M$ is formed by the rightmost $n - d$ bits of $X$'s subscript, $X_0, X_1 \ldots, X_{N/P}$ are
stored in natural order in one processor, which is determined to be $P_0$; and
the next $N/P$ elements, namely $X_{N/P}, X_{N/P+1}, \ldots, X_{2(N/P)-1}$, are stored in natural
order in one processor, which is determined to be $P_{P/2}$ by simply reversing the
leftmost $d$ bits of $X$'s subscript; and so on. Since the mapping is equivalent to a
CBM block mapping of the naturally ordered $X$, we use the term “block-equiva-
 lent” in the relevant table entries. To use this method, one simply distributes $x$
before the computation using the cyclic CBM, and apply this algorithm to ob-
tain naturally ordered $X$. Note that the input data must be distributed among
the processors one way or the other, and the communication cost for data dis-
btribution is the same regardless of the initial mapping. Therefore, the cyclic ini-
tial mapping used by this method does not cause extra communication, and
this is one way to obtain the block-equivalent mapping of naturally ordered
$X$. It will be shown later that the communication cost can be halved in an im-
proved algorithm which achieves equivalent results.

The FFTs in Refs. [3,7,15,18] all contain inter-processor-permutations
which may be viewed as switching a PID bit with an address bit, the pivot.
In Ref. [7] the PID is formed using the rightmost $d$ bits as shown in the two
initial mappings given in Table 1. To ensure that the stride for data in butterfly
operations is always one (for good locality), the current rightmost bit in the Local
$M$ is used as the Pivot for either a local or an inter-processor permutation,
which precedes each butterfly operation, as depicted by the following example.

$$|i_1i_0i_2| \quad |i_1i_0i_2| \quad |i_1i_0\tau_2i_2| \quad |i_1i_0\tau_3i_3| \quad |\tau_2\tau_1\tau_3i_0|$$

Here the sequence begins with $|i_1i_0i_3i_2$, which is the result of a cyclic mapping
of naturally ordered $x$; the following $|i_1i_0\tau_2i_3$ is intended to imply that $i_3$
has been permuted into the pivot location before the butterfly computation begins.
Since the initial mapping $|i_1i_0\tau_2i_3$ may be obtained by permuting the local data
resulting from a cyclic mapping, it is referred to as a cyclic-variant in Table 1.
Given the initial mapping and the fixed-position of the pivot, which is a $\tau_\rho$ (in-
stead of $i_\rho$) bit, the communication cost and the final mapping can be easily
predicted for given $N$ and $P$ as explained in the previous section. However,
the resulting mapping is neither a CBM nor its equivalent for $X$.

The work in Ref. [3] deals with all possible initial CBM mappings for given
$N$ and $P$ on a hypercube. The initial CBM for $x$ and the final CBM for $X$ are
required to be identical. Under the condition, the authors show that in addition
to \( d \) concurrent exchanges between all pairs of processors with IDs different in one bit, their generalized subroutine could, in the worst case, require each processor to send data to all the other processors. This requirement may cause severe data contention depending on the network topology. A new general algorithm is proposed in Section 6 which requires only \( 1.5d \) more concurrent exchanges in the worst case. Thus the algorithm proposed in Section 6 appears to deal with the question in Ref. [3] concerning the communications requirement for solving the data rearrangement problems arising in FFT or other similar algorithms.

The work in Refs. [15,18] concerns two specific CBM mappings; in each case, the initial mapping for input \( x_{i_1,i_2,i_3,i_4} \) is required to be maintained for output \( X_{i_1,i_2,i_3,i_4} \). These two parallel algorithms are depicted in Table 2 using given \( N \) and \( P \) values. For the natural-order parallel FFT in Ref. [15], the communication cost of \( 2d + 1 = 5 \) exchanges for \( N = 2^n = 32 \) and \( P = 2^d = 4 \) are marked by five occurrences of \( \leftarrow \rightarrow \) in the first column. The algorithm differs depending on whether \( d = n/2, \ d < n/2, \ d = n - 1, \) or \( n/2 < d < n - 1 \). Consequently, the communication costs range from \( 1.5d + 2 \) to \( 2d + 1 \) concurrent exchanges as indicated by the following theorem.

**Theorem [15].** An ordered FFT of length \( N = 2^n \) can be implemented on a hypercube of dimension \( d \) with \( n/2 + d + 1 \) parallel transmissions if \( n/2 < d \) and \( n \) is even. If \( n \) is odd and \( (n + 1)/2 < d \), then \( (n + 1)/2 + d + 1 \) parallel transmissions are required. For the remaining cases the ordered FFT can be implemented with \( 2d + 1 \) parallel transmissions.

Note that a hypercube of dimension \( d \) has \( P = 2^d \) processors, and that the condition \( n/2 < d \) is equivalent to \( N/P < P \), and \( (n + 1)/2 < d \) is equivalent to \( N/P < P/2 \), so they represent fine-grain cases. Consequently, for more common medium-grain and large-grain cases, \( 2d + 1 \) parallel exchanges will be required.

The cyclic-order parallel FFT in Ref. [18] refers to an FFT with naturally ordered input \( x \) and output \( X \) both CBM mapped to the processors using one element per block. That is, the initial cyclic mapping of \( x \) is maintained for \( X \). The case \( N/P < P \) was considered in Ref. [18] for the massively parallel Connection Machine (CM). The communication cost was derived in proving the following lemma.

**Lemma [18].** A cyclic-order FFT of length \( N = 2^n \) can be implemented on a hypercube of dimension \( d \) (where \( n/2 < d \)) with \( 2d - n/2 \) parallel transmissions if \( n \) is even and \( 2d - (n - 1)/2 \) parallel transmissions if \( n \) is odd.

Since the condition \( n/2 < d \) is equivalent to \( N/P < P \), this lemma again applies to the fine-grain cases, and the number of parallel message exchanges is
Table 2
Some ordered parallel FFTs in the literature

<table>
<thead>
<tr>
<th>Swarztrauber [15]</th>
<th>Tong and Swarztrauber [18]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example: $N = 2^8 = 32$, $P = 2^d = 4$ ($\frac{N}{P} \geq P$)</td>
<td>Example [18]: $N = 2^8 = 256$, $P = 2^d = 32$ ($\frac{N}{P} &lt; P$)</td>
</tr>
<tr>
<td>$[i_0 i_1 i_2 i_3 i_4]$ (block map of $x$)</td>
<td>$[i_0 i_1 i_2 i_3 i_4]$ (cyclic map of $x$)</td>
</tr>
<tr>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_1 i_2 i_3 i_4]$ (block map of $X$)</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
<tr>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
<td>$[i_4 i_3 i_2 i_1 i_0]$</td>
</tr>
</tbody>
</table>

$2d - n/2 > d$ (if $n$ is even) or $2d - (n - 1)/2 > d$ (if $n$ is odd). In the finest-grain case, $N/P = 1$ if $n$ is even: i.e., $n = d$, and the number of parallel transmissions is given by $2d - n/2 = 1.5d$. If $n$ is odd, $N/P = 2$ or $n - 1 = d$ for the finest-grain case, and the number of parallel message exchanges is given by $2d - (n - 1)/2 = 1.5d$. In the example cited in Table 2, the condition
In both algorithms, the pivot is always the current leftmost bit in the Local M, but reordering of the address bits in Local M is also performed by exchanging the pivot with another bit in the Local M. This implies that other bits from Local M can effectively serve as the pivot bit, although they must first be permuted into the fixed pivot location. In Refs. [15,18], the permutation of any other bit with the fixed pivot bit is defined as a single i-cycle. The algorithms were developed by first decomposing the required permutation (or final mapping) into disjoint cycles, and each disjoint cycle can then be implemented by a sequence of i-cycles. Some of the i-cycles are followed by butterfly computations, and other i-cycles are used only for the purpose of rearranging local data or permuting data between processors. The pseudocode (similar to CM Fortran) FFT algorithm given in Ref. [18] uses an i-cycle subroutine assuming that \( N/\tilde{P} = 2 \), where \( \tilde{P} \) could represent the number of virtual processors when \( N/P > 2 \). In the latter case, \( \tilde{P} > P \), and the cost of 1.5\( d \) concurrent exchanges, where \( d = \log_2 \tilde{P} \), includes the communication between virtual processors. In the next section alternative ways are proposed having the same or lower communication cost and without the restriction to fixed-pivot i-cycles. In addition, the "\( N/P \)" ratio is not restricted to a specific value.

6. New parallel FFTs and a generalization

In this section three new parallel FFT algorithms are proposed, including a generalized algorithm and communication complexity results for other CBM mappings which are believed to be new. For easy comparison with the algorithms reviewed in the last section, the first two algorithms are depicted in Table 3 and a general algorithm is depicted in Table 4 below. Since the result of bit-reversing two bits is the same as the result of cyclic-shifting them one position to the right, examples with 8 or 32 processors are now used to demonstrate the general cases better.

6.1. Algorithm I

The elaboration of algorithm I is as follows. Consider a general case, in which the initial cyclic mapping of naturally ordered \( x_{i_n-1 \ldots i_0} \) is transformed to the final mapping of \( X_{i_{d-1} \ldots i_0} \) given by

\[
d\text{PID bits} | i_d i_{d+1} \ldots i_{n-2} i_{n-1},
\]

where \( \text{PID} = i_0 i_{d-1} i_{d-2} \ldots i_{2} i_1 \); i.e., all bits in the initial PID are cyclic-shifted to the right by one position. Since the rightmost \( n - d \) bits of \( X \)'s subscript form
the local $M$, the final mapping is equivalent to a block CBM of naturally ordered $X$. To obtain the final local $M$ which bit-reverses the initial local $M$, all that is required is to apply an ordered FFT to the local data. As explained in Section 5, the final PID is obtained from permuting each PID bit into the pivot position and back into the PID field in sequence. The combination of a specific initial mapping, inter-processor permutations, and an ordered local FFT produces an ordered parallel FFT. Compared to other parallel FFTs reviewed in Section 5, the communication cost is halved because only $d + 1$ concurrent exchanges of $\frac{4}{P}N$ elements are needed. Of course, the same algorithm can also transform block mapped $x$ to cyclic-equivalent mapped $X$, and vice versa.

6.2. Algorithm II

Observe first that the initial and final PIDs use the rightmost $d$ bits from the subscripts of $x$ and $X$, respectively. Because $X$'s subscript bit-reverses $x$'s subscript, the final PID is formed by bit-reversing the leftmost $d$ bits from $x$'s subscript. The algorithm is designed to perform the first $d$ stages of local computation without reordering the local data, followed by $n - 2d$ more stages of local computation using an ordered sequential FFT. The initial mapping is thus changed to

$$\begin{align*}
|\text{Initial rightmost } d \text{ bits}| \quad |\text{Initial leftmost } d \text{ bits}| \quad |\text{Reversed } n - 2d \text{ bits}|
\end{align*}$$

$$\begin{align*}
= [i_{d-1} i_{d-2} \ldots i_1 i_0 | \tau_{n-1} \tau_{n-2} \ldots \tau_{n-d+1} \tau_{n-d} \tau_d \tau_{d+1} \ldots \tau_{n-d-1}].
\end{align*}$$

The objective is to apply inter-processor permutations in the remaining $d$ stages so that the final mapping becomes
For simplicity, assume \( n - d \geq d \) (which implies \( N/P \geq P \)) so that the rightmost \( d \) bits do not overlap the leftmost \( d \) bits; the case \( n - d < d \) will be dealt with later. Under this assumption, the remaining \( d \) inter-processor permutations are performed to switch \( i_d-1 \) with \( \tau_{n-d} \), \( i_{d-2} \) with \( \tau_{n-d+1} \), \( \ldots \), \( i_1 \) with \( \tau_{n-2} \), and \( i_0 \) with \( \tau_{n-1} \). That is, the PID bit is switched with a different pivot each time, and the pivots are simply chosen in the order they appear in the final PID. Since all potential pivot bits are \( \tau_p \) instead of \( i_p \), only \( d \) concurrent exchanges of \( \frac{1}{2}(N/P) \) elements are required.

For the case \( n - d < d \) (which implies \( N/P < P \)), local reordering of data may be added to Algorithm II to produce a cyclic-equivalent mapping for elements of \( X \). That is, the final mapping of \( X_{i_0, i_1, \ldots, i_{d-1}} \) is given by

\[
| \tau_{n-1} \tau_{n-2} \ldots \tau_{d+1} \tau_d \tau_{d-1} \ldots \tau_{n-d} | \tau_0 \tau_1 \ldots \tau_{n-d-1} |
\]

Note that the final PID bit-reverses the rightmost \( d \) bits of \( X \)'s subscript. In order to describe how Algorithm II is generalized to include this case, the condition \( n - d < d \) is reflected in the initial mapping given by

\[
| \text{Initial rightmost } d \text{ bits } | n - d < d \text{ bits } |
\]

\[
\equiv | i_{d-1} i_{d-2} \ldots i_{n-d} i_{n-d-1} \ldots i_1 i_0 | i_{n-1} \ldots i_{d+1} i_d |
\]

After the first \( n - d \) stages of local computations (without reordering of the local data), the mapping becomes

\[
| i_{d-1} i_{d-2} \ldots i_{n-d} i_{n-d-1} \ldots i_1 i_0 | \tau_{n-1} \ldots \tau_{d+1} \tau_d |\]

The next \( d - (n - d) = 2d - n \) inter-processor permutations switch the leftmost \( 2d - n \) bits in PID with the bits in the Local \( M \) in the order from left to right; if it reaches the end of Local \( M \), begin from the leftmost bit in Local \( M \) until all \( 2d - n \) bits in PID are processed. The following are mapping results:

\[
| \tau_n \tau_{n-2} \ldots \tau_{k+1} \tau_k i_{n-1} i_0 \ldots \tau_{m-2} \tau_{m-1} \tau_m \tau_{k-2} \tau_k \ldots |\]

Now the local reordering can be applied to rearrange the bits in Local \( M \) to obtain

\[
| \tau_{n-1} \tau_{n-2} \ldots \tau_{k+1} \tau_k i_{n-d} i_0 \ldots \tau_{m-2} \tau_{m-1} \tau_m \tau_{k-2} \tau_k \ldots |\]

(Note that the bits braced from \( \tau_{m-1} \) to \( \tau_{n-d} \) are reversed, and the bits braced from \( \tau_{k-1} \) to \( \tau_m \) are reversed. Since the bit-reversing operations can be incorporated into the computation, the internal reordering does not necessarily increase processing time in an efficient implementation.) The final \( n \) \( d \) inter-processor permutations in Algorithm II switches the remaining \( n - d \) bits in PID into the local \( M \), resulting in the final mapping
This is a cyclic-equivalent mapping for $X_{i_0i_1\ldots i_{n-1}}$ because the Local $M$ is formed by the leftmost $n - d$ bits of $X$. Note that the rightmost $d$ bits of $X$'s subscript can be obtained by reversing the PID bits, and vice versa.

6.3. A general algorithm and communication complexity results

This algorithm allows the choice of any initial CBM for naturally ordered $x$ and any final CBM for naturally ordered $X$. For $N = 128$ and $P = 32$, the four possible initial CBMs of $x$ are shown in the first four entries in column 1 in Table 4; in each case, the initial CBM is required to be maintained for $X$ as shown in column 2. In column 3, both actual and the maximum (worse-case) communication complexities are presented for $N = 2^n$ and $P = 2^d$, assuming only that $N/P \geq 2$. That is, the specific relationships between $n$ and $d$ in different cases are not exploited in the algorithm.

The algorithm uses the ideas developed in Section 5. Recall that by using $d + 1$ inter-processor permutations, a parallel FFT can be completed with the initial PID bits cyclic-shifted-to-the-right by one position. Apparently more inter-processor permutations may be added afterwards to switch the desired $\tau_p$ bits into the PID and arrange them in final order, and that can be followed by local reordering so that the bits in the Local $M$ are also arranged as desired. To determine the maximum communication cost, consider the two possibilities.

Case (i). If the initial PID does not overlap the final PID, then exactly $d$ more inter-processor permutations are needed to switch in the final PID bits. In total, $2d + 1$ concurrent exchanges are needed.

Case (ii). If the initial PID overlaps the final PID, then one may first switch in the $m \tau_p$ bits currently in Local $M$ so that these $m$ PID bits are put in their final positions. Repeat this process until no desired PID bits exist in the local $M$. Now the only remaining task is to reorder the $v = d - k$ potentially out-of-order bits in the PID. Because the final PID will no longer involve bits in the Local $M$, one need only consider a fixed (arbitrary) pivot. How many inter-processor permutations are needed to reorder $v$ PID bits via a single pivot? Observe that in the worst case, the non-PID pivot bit is permuted back in Local $M$ after two PID bits happen to switch their positions; i.e., three permutations could be required for placing every two bits. Therefore, at most $1.5v$ inter-processor permutations are required to reorder the $v$ PID bits. Since $v$ can be as large as $d$, in total $2.5d + 1$ concurrent exchanges are needed.

Although the two cases above were considered separately to help explain the algorithm, note that case (ii) reduces to case (i) if $m = d$. The actual communication cost for given $N$ and $P$ can also be predicted for the entire class of CBM mappings as shown in Table 4. Since the reordering phase requires only
Table 4
General communication results for CBM mappings: A general algorithm for arbitrary initial and final CBM mappings. Assumption: \( N = 2^a, P = 2^b, \) and \( N/P \geq 2 \)

<table>
<thead>
<tr>
<th>Any initial CBM of ( X_{(i_1,i_2,i_3,i_4)} )</th>
<th>Any final CBM of ( X_{(j_1,j_2,j_3,j_4)} )</th>
<th># Concurrent exchanges</th>
<th>Message length</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_1 i_2 i_3 i_4 ) (block)</td>
<td>( \tau_0 \tau_1 \tau_2 \tau_3 \tau_4 \tau_5 ) (block)</td>
<td>( 2d + 1 \leq 2.5d + 1 )</td>
<td>( \frac{1}{2} \frac{N}{P} )</td>
</tr>
<tr>
<td>( i_0 i_1 i_2 i_3 )</td>
<td>( \tau_0 \tau_1 \tau_2 \tau_3 \tau_4 \tau_5 )</td>
<td>( 2d + 2 \leq 2.5d + 1 )</td>
<td>( \frac{1}{2} \frac{N}{P} )</td>
</tr>
<tr>
<td>( i_1 i_2 i_3 i_4 ) (cyclic)</td>
<td>( \tau_0 \tau_1 \tau_2 \tau_3 \tau_4 \tau_5 ) (cyclic)</td>
<td>( 2d + 1 \leq 2.5d + 1 )</td>
<td>( \frac{1}{2} \frac{N}{P} )</td>
</tr>
<tr>
<td>( i_0 i_2 i_3 i_4 )</td>
<td>( \tau_0 \tau_1 \tau_2 \tau_3 \tau_4 \tau_6 )</td>
<td>( 2d \leq 2.5d + 1 )</td>
<td>( \frac{1}{2} \frac{N}{P} )</td>
</tr>
<tr>
<td>( i_0 i_2 i_4 i_6 )</td>
<td>( \tau_0 \tau_1 \tau_2 \tau_3 \tau_4 \tau_6 )</td>
<td>( 2d + 2 \leq 2.5d + 1 )</td>
<td>( \frac{1}{2} \frac{N}{P} )</td>
</tr>
<tr>
<td>( i_0 i_2 i_4 i_6 ) (block)</td>
<td>( \tau_0 \tau_1 \tau_2 \tau_3 \tau_4 \tau_6 ) (block)</td>
<td>( 2d \leq 2.5d + 1 )</td>
<td>( \frac{1}{2} \frac{N}{P} )</td>
</tr>
</tbody>
</table>

1.5d more concurrent exchanges in the worst case, this algorithm appears to have dealt with the question in Ref. [3] concerning the communication requirement for solving the data rearrangement problems arising in FFT or other similar algorithms. In addition, since the reordering phase is independent of the computation phase, the algorithm and the communication complexity results are applicable even if the initial and final CBM maps are different. The last three entries in Table 4 are such examples. Thus, the proposed algorithm deals with cases that do not appear to be handled by the algorithm proposed in Ref. [3].

7. Conclusion

In this paper the various ideas which have appeared in the FFT literature were presented in a common framework. To this end, the concept of index-digit permutation and the associated binary address based notation were borrowed, modified, and applied in a uniform manner to explain and further explore the various algorithm design and implementation issues related to both sequential and parallel FFTs. Although the study is limited to variants of a radix-2 DIF FFT algorithm, the notations and results may be generalized to cover the entire family of radix-2 \( a \) DIF and DIT FFTs, including the mixed-radix and the split-radix variants. These possible extensions as well as the extension to two-dimen-
sional (2-D) FFTs are omitted for lack of space and will be presented elsewhere. In addition, three new parallel FFT algorithms along with communication results were presented. The proposed algorithms show alternative ways of designing parallel FFT algorithms which feature reduced communication cost and further flexibility in the choices of initial and final data mappings.

Acknowledgements

The author are grateful to the anonymous referee who obviously read the original manuscript with great care. In particular, the referee provided valuable commentary about issues related to accessing twiddle factors on distributed-memory computers; those points have been included. The referee also drew the authors' attention to the literature on self-sorting in-place FFT algorithms, which are briefly reviewed at the end of Section 3. The work of first author was supported in part by Canadian Natural Sciences and Engineering Research Council grant OGP0121352. The work of second author was supported in part by Canadian Natural Sciences and Engineering Research Council grant OGP000811.

References