Contents

CHAPTER 11

Fast Fourier Transform 1

CHAPTER 11

Fast Fourier Transform

The discrete Fourier transform (DFT) plays an important role in many scientific and technical applications, including time series and wave analysis, solutions to linear partial differential equations, convolution, digital signal processing, and image filtering. The DFT is a linear transformation that maps n regularly sampled points from a cycle of a periodic signal, like a sine wave, onto an equal number of points representing the frequency spectrum of the signal. In 1965, Cooley and Tukey devised an algorithm to compute the DFT of an n-point series in $\Theta(n \log n)$ operations. Their new algorithm was a significant improvement over previously known methods for computing the DFT, which required $\Theta(n^2)$ operations. The revolutionary algorithm by Cooley and Tukey and its variations are referred to as the ***fast Fourier transform*** (FFT). Due to its wide application in scientific and engineering fields, there has been a lot of interest in implementing FFT on parallel computers.

Several different forms of the FFT algorithm exist. This chapter discusses its simplest form, the one-dimensional, unordered, radix-2 FFT. Parallel formulations of higher-radix and multidimensional FFTs are similar to the simple algorithm discussed in this chapter because the underlying ideas behind all sequential FFT algorithms are the same. An ordered FFT is obtained by performing bit reversal (Section 11.4) on the output sequence of an unordered FFT. Bit reversal does not affect the overall complexity of a parallel implementation of FFT.

In this chapter we discuss two parallel formulations of the basic algorithm: the ***binary-exchange algorithm*** and the ***transpose algorithm***. Depending on the size of the input n, the number of processes p, and the memory or network bandwidth, one of these may run faster than the other.

11.1 The Serial Algorithm

Consider a sequence $X = \langle X[0], X[1], \ldots, X[n-1] \rangle$ of length n. The discrete Fourier transform of the sequence X is the sequence $Y = \langle Y[0], Y[1], \ldots, Y[n-1] \rangle$, where

$$Y[i] = \sum_{k=0}^{n-1} X[k]\omega^{ki}, \quad 0 \le i < n. \tag{11.1}$$

In Equation 11.1, ω is the primitive n^{th} root of unity in the complex plane; that is, $\omega = e^{2\pi\sqrt{-1}/n}$, where e is the base of the natural logarithm. More generally, the powers of ω in the equation can be thought of as elements of the finite commutative ring of integers modulo n. The powers of ω used in an FFT computation are also known as ***twiddle factors***.

The computation of each $Y[i]$ according to Equation 11.1 requires n complex multiplications. Therefore, the sequential complexity of computing the entire sequence Y of length n is $\Theta(n^2)$. The fast Fourier transform algorithm described below reduces this complexity to $\Theta(n \log n)$.

Assume that n is a power of two. The FFT algorithm is based on the following step that permits an n-point DFT computation to be split into two $(n/2)$-point DFT computations:

$$\begin{aligned}
Y[i] &= \sum_{k=0}^{(n/2)-1} X[2k]\omega^{2ki} + \sum_{k=0}^{(n/2)-1} X[2k+1]\omega^{(2k+1)i} \\
&= \sum_{k=0}^{(n/2)-1} X[2k]e^{2(2\pi\sqrt{-1}/n)ki} + \sum_{k=0}^{(n/2)-1} X[2k+1]\omega^{i} e^{2(2\pi\sqrt{-1}/n)ki} \\
&= \sum_{k=0}^{(n/2)-1} X[2k]e^{2\pi\sqrt{-1}ki/(n/2)} + \omega^{i} \sum_{k=0}^{(n/2)-1} X[2k+1]e^{2\pi\sqrt{-1}ki/(n/2)}
\end{aligned} \tag{11.2}$$

Let $\tilde{\omega} = e^{2\pi\sqrt{-1}/(n/2)} = \omega^2$; that is, $\tilde{\omega}$ is the primitive $(n/2)^{nd}$ root of unity. Then, we can rewrite Equation 11.2 as follows:

$$Y[i] = \sum_{k=0}^{(n/2)-1} X[2k]\tilde{\omega}^{ki} + \omega^{i} \sum_{k=0}^{(n/2)-1} X[2k+1]\tilde{\omega}^{ki} \tag{11.3}$$

---

procedure R_FFT(X, Y, n, ω)
if $(n = 1)$ **then** $Y[0] := X[0]$ **else**
begin
    R_FFT($\langle X[0], X[2], \ldots, X[n-2] \rangle, \langle Q[0], Q[1], \ldots, Q[n/2] \rangle, n/2, \omega^2$);
    R_FFT($\langle X[1], X[3], \ldots, X[n-1] \rangle, \langle T[0], T[1], \ldots, T[n/2] \rangle, n/2, \omega^2$);
    for $i := 0$ **to** $n-1$ **do**
        $Y[i] := Q[i \bmod (n/2)] + \omega^{i} T[i \bmod (n/2)]$;
end R_FFT

Algorithm 11.1 The recursive, one-dimensional, unordered, radix-2 FFT algorithm. Here $\omega = e^{2\pi\sqrt{-1}/n}$.

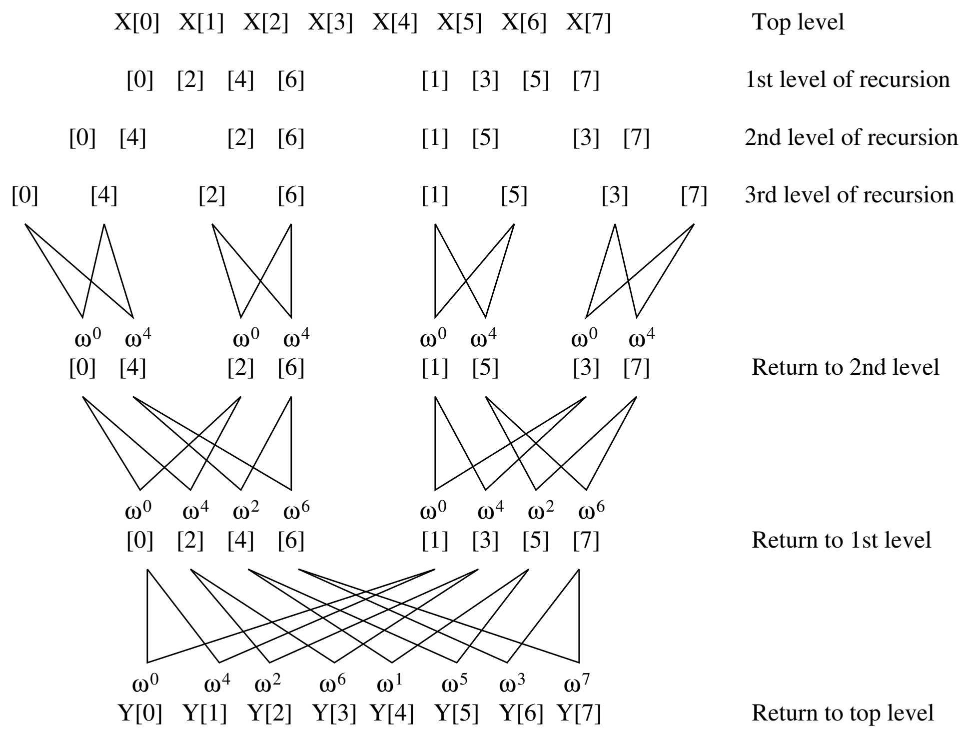


Figure 11.1 A recursive 8-point unordered FFT computation.

In Equation 11.3, each of the two summations on the right-hand side is an $(n/2)$-point DFT computation. If n is a power of two, each of these DFT computations can be divided similarly into smaller computations in a recursive manner. This leads to the recursive FFT algorithm given in Algorithm 11.1. This FFT algorithm is called the radix-2 algorithm because at each level of recursion, the input sequence is split into two equal halves.

Figure 11.1 illustrates how the recursive algorithm works on an 8-point sequence. As the figure shows, the first set of computations corresponding to line 7 of Algorithm 11.1 takes place at the deepest level of recursion. At this level, the elements of the sequence whose indices differ by $n/2$ are used in the computation. In each subsequent level, the difference between the indices of the elements used together in a computation decreases by a factor of two. The figure also shows the powers of ω used in each computation.

The size of the input sequence over which an FFT is computed recursively decreases by a factor of two at each level of recursion (lines 4 and 5 of Algorithm 11.1). Hence, the maximum number of levels of recursion is $\log n$ for an initial sequence of length n. At the m^{th} level of recursion, 2^m FFTs of size $n/2^m$ each are computed. Thus, the total number of arithmetic operations (line 7) at each level is $\Theta(n)$ and the overall sequential complexity of Algorithm 11.1 is $\Theta(n \log n)$.

The serial FFT algorithm can also be cast in an iterative form. The parallel implemen-

1. **procedure** ITERATIVE_FFT(X, Y, n)
2. **begin**
3. $r := \log n$;
4. **for** $i := 0$ **to** $n-1$ **do** $R[i] := X[i]$;
5. **for** $m := 0$ **to** $r-1$ **do** /* Outer loop */
6. **begin**
7. **for** $i := 0$ **to** $n-1$ **do** $S[i] := R[i]$;
8. **for** $i := 0$ **to** $n-1$ **do** /* Inner loop */
9. **begin**

/* Let $(b_0 b_1 \cdots b_{r-1})$ be the binary representation of i */

10. $j := (b_0 \ldots b_{m-1} 0 b_{m+1} \cdots b_{r-1})$;
11. $k := (b_0 \ldots b_{m-1} 1 b_{m+1} \cdots b_{r-1})$;
12. $R[i] := S[j] + S[k] \times \omega^{(b_m b_{m-1} \cdots b_0 0 \cdots 0)}$;
13. **endfor**; /* Inner loop */
14. **endfor**; /* Outer loop */
15. **for** $i := 0$ **to** $n-1$ **do** $Y[i] := R[i]$;
16. **end** ITERATIVE_FFT

Algorithm 11.2 The Cooley-Tukey algorithm for one-dimensional, unordered, radix-2 FFT. Here $\omega = e^{2\pi\sqrt{-1}/n}$.

tations of the iterative form are easier to illustrate. Therefore, before describing parallel FFT algorithms, we give the iterative form of the serial algorithm. An iterative FFT algorithm is derived by casting each level of recursion, starting with the deepest level, as an iteration. Algorithm 11.2 gives the classic iterative Cooley-Tukey algorithm for an n-point, one-dimensional, unordered, radix-2 FFT. The program performs $\log n$ iterations of the outer loop starting on line 5. The value of the loop index m in the iterative version of the algorithm corresponds to the $(\log n - m)^{th}$ level of recursion in the recursive version (Figure 11.1). Just as in each level of recursion, each iteration performs n complex multiplications and additions.

Algorithm 11.2 has two main loops. The outer loop starting at line 5 is executed $\log n$ times for an n-point FFT, and the inner loop starting at line 8 is executed n times during each iteration of the outer loop. All operations of the inner loop are constant-time arithmetic operations. Thus, the sequential time complexity of the algorithm is $\Theta(n \log n)$. In every iteration of the outer loop, the sequence R is updated using the elements that were stored in the sequence S during the previous iteration. For the first iteration, the input sequence X serves as the initial sequence R. The updated sequence X from the final iteration is the desired Fourier transform and is copied to the output sequence Y.

Line 12 in Algorithm 11.2 performs a crucial step in the FFT algorithm. This step updates $R[i]$ by using $S[j]$ and $S[k]$. The indices j and k are derived from the index i as

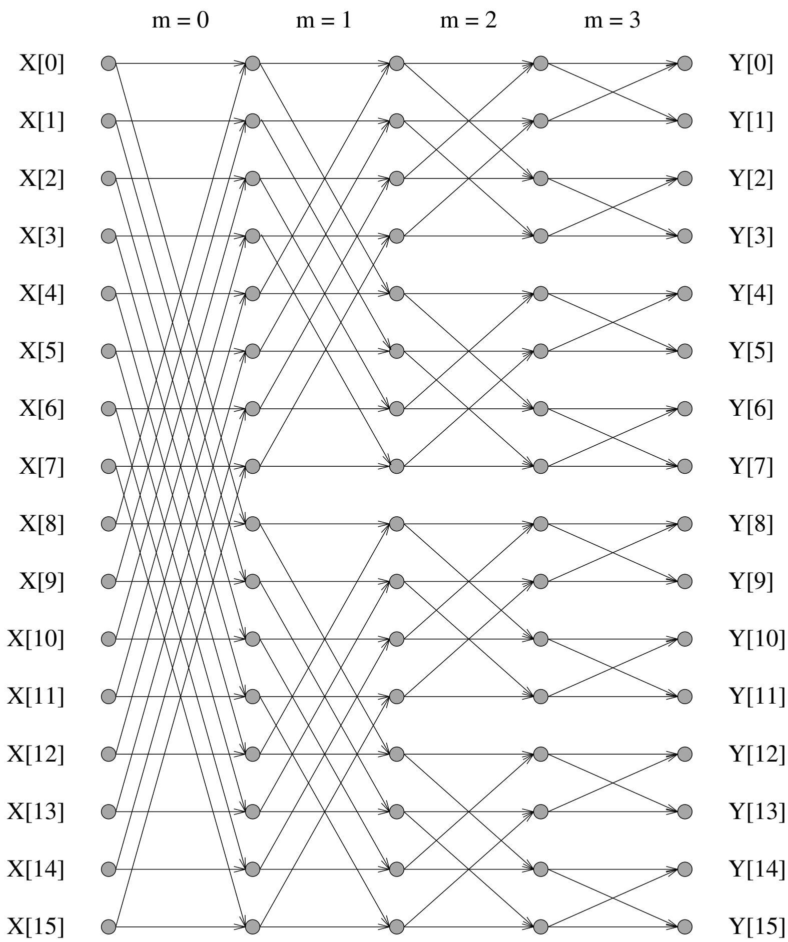


Figure 11.2 The pattern of combination of elements of the input and the intermediate sequences during a 16-point unordered FFT computation.

follows. Assume that $n = 2^r$. Since $0 \le i < n$, the binary representation of i contains r bits. Let $(b_0 b_1 \cdots b_{r-1})$ be the binary representation of index i. In the m^{th} iteration of the outer loop $(0 \le m < r)$, index j is derived by forcing the m^{th} most significant bit of i (that is, b_m) to zero. Index k is derived by forcing b_m to 1. Thus, the binary representations of j and k differ only in their m^{th} most significant bits. In the binary representation of i, b_m is either 0 or 1. Hence, of the two indices j and k, one is the same as index i, depending on whether $b_m = 0$ or $b_m = 1$. In the m^{th} iteration of the outer loop, for each i between 0 and $n-1$, $R[i]$ is generated by executing line 12 of Algorithm 11.2 on $S[i]$ and on another element of S whose index differs from i only in the m^{th} most significant bit. Figure 11.2 shows the pattern in which these elements are paired for the case in which $n = 16$.

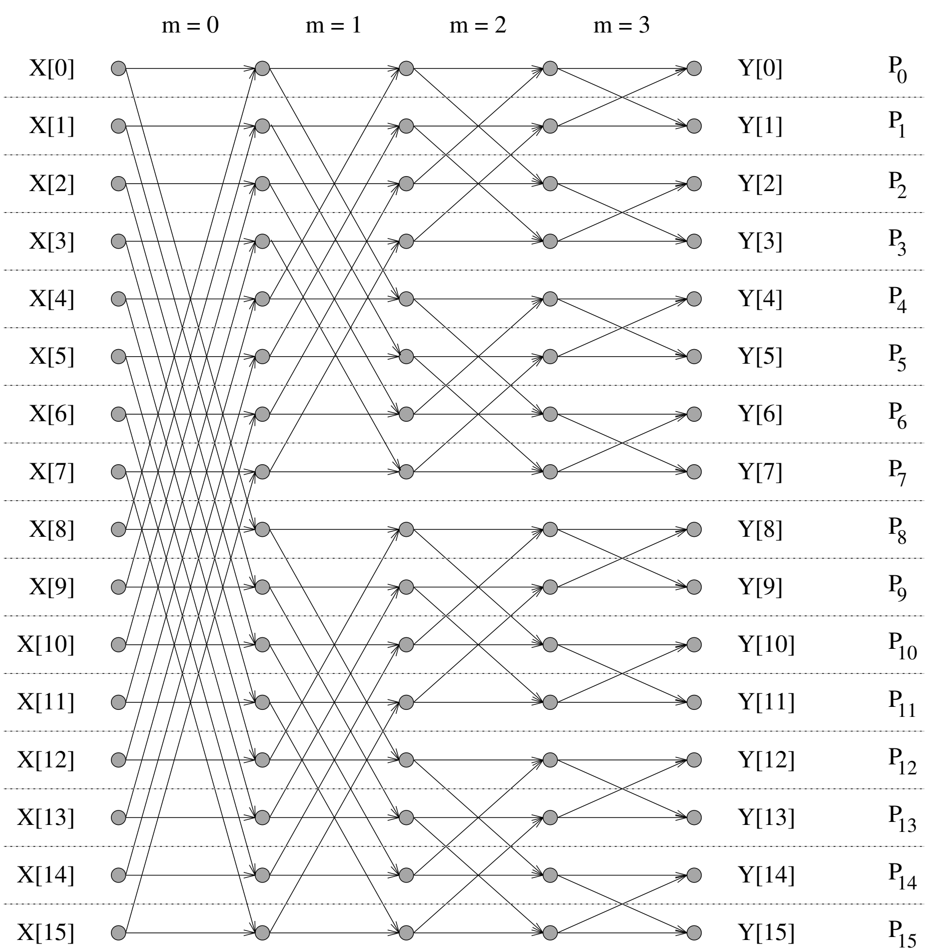


Figure 11.3 A 16-point unordered FFT on 16 processes. P_i denotes the process labeled i.

11.2 The Binary-Exchange Algorithm

This section discusses the ***binary-exchange algorithm*** for performing FFT on a parallel computer. First, a decomposition is induced by partitioning the input or the output vector. Therefore, each task starts with one element of the input vector and computes the corresponding element of the output. If each task is assigned the same label as the index of its input or output element, then in each of the $\log n$ iterations of the algorithm, exchange of data takes place between pairs of tasks with labels differing in one bit position.

11.2.1 A Full Bandwidth Network

In this subsection, we describe the implementation of the binary-exchange algorithm on a parallel computer on which a bisection width (Section **??**) of $\Theta(p)$ is available to p parallel processes. Since the pattern of interaction among the tasks of parallel FFT matches that of a hypercube network, we describe the algorithm assuming such an interconnection network. However, the performance and scalability analysis would be valid for any parallel computer with an overall simultaneous data-transfer capacity of $O(p)$.

One Task Per Process

We first onsider a simple mapping in which one task is assigned to each process. Figure 11.3 illustrates the interaction pattern induced by this mapping of the binary-exchange algorithm for $n = 16$. As the figure shows, process i $(0 \leq i < n)$ initially stores $X[i]$ and finally generates $Y[i]$. In each of the $\log n$ iterations of the outer loop, process P_i updates the value of $R[i]$ by executing line 12 of Algorithm 11.2. All n updates are performed in parallel.

To perform the updates, process P_i requires an element of S from a process whose label differs from i in only one bit. Recall that in a hypercube, a process is connected to all those processes whose labels differ from its own in only one bit position. Thus, the parallel FFT computation maps naturally onto a hypercube because every pair of communicating processes is directly connected. In the first iteration of the outer loop, the labels of each pair of communicating processes differ only in their most significant bits. For instance, processes P_0 to P_7 communicate with P_8 to P_{15}, respectively. Similarly, in the second iteration, the labels of processes communicating with each other differ in the second most significant bit, and so on.

In each of the $\log n$ iterations of this algorithm, every process performs one complex multiplication and addition, and exchanges one complex number with a directly-connected process. Thus, there is a constant amount of work per iteration. Hence, it takes $\Theta(\log n)$ time to execute the algorithm in parallel by using a hypercube with n nodes. This hypercube formulation of FFT is cost-optimal because its process-time product is $\Theta(n \log n)$, the same as the complexity of a serial n-point FFT.

Multiple Tasks Per Process

We now consider a mapping in which the n tasks of an n-point FFT are mapped onto p processes, where $n > p$. For the sake of simplicity, let us assume that both n and p are powers of two; i.e, $n = 2^r$ and $p = 2^d$. As Figure 11.4 shows, we partition the sequences into blocks of n/p contiguous elements and assign one block to each process.

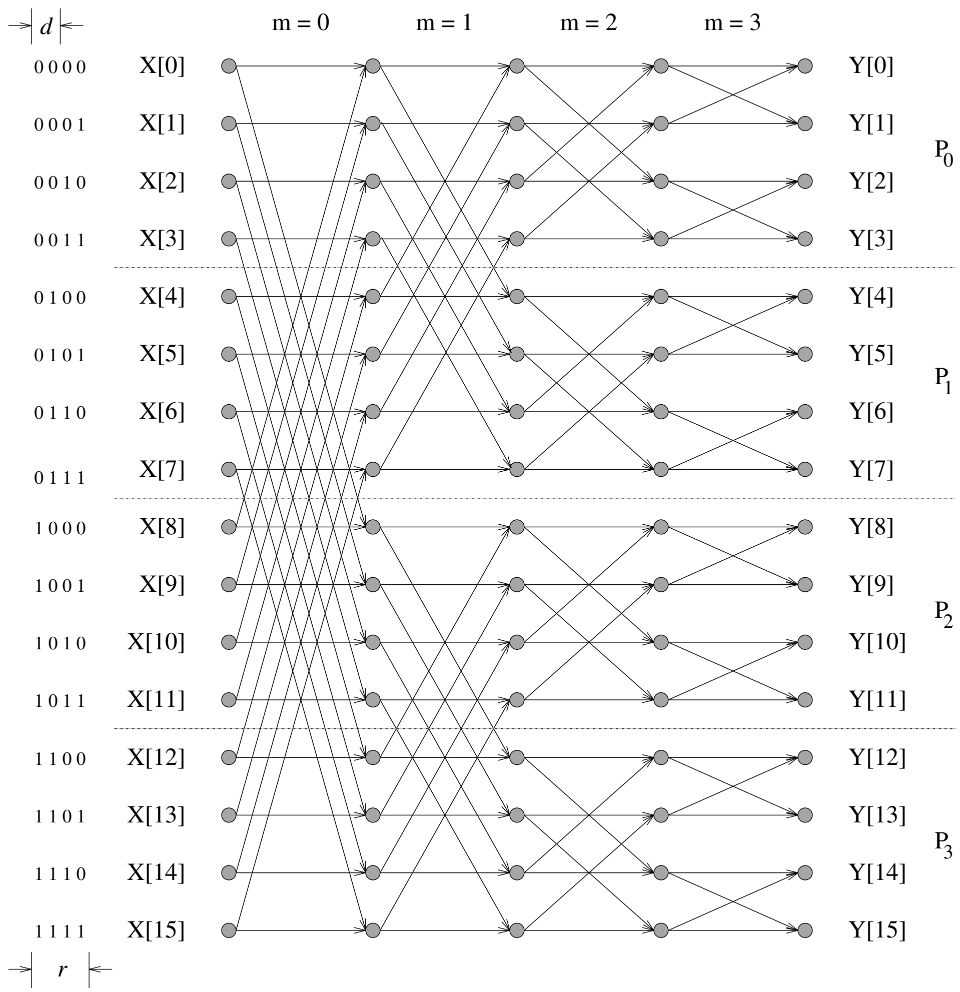


Figure 11.4 A 16-point FFT on four processes. P_i denotes the process labeled i. In general, the number of processes is $p = 2^d$ and the length of the input sequence is $n = 2^r$.

An interesting property of the mapping shown in Figure 11.4 is that, if $(b_0 b_1 \cdots b_{r-1})$ is the binary representation of any i, such that $0 \leq i < n$, then $R[i]$ and $S[i]$ are mapped onto the process labeled $(b_0 \cdots b_{d-1})$. That is, the d most significant bits of the index of any element of the sequence are the binary representation of the label of the process that the element belongs to. This property of the mapping plays an important role in determining the amount of communication performed during the parallel execution of the FFT algorithm.

Figure 11.4 shows that elements with indices differing in their d (= 2) most significant bits are mapped onto different processes. However, all elements with indices having the same d most significant bits are mapped onto the same process. Recall from the previous section that an n-point FFT requires $r = \log n$ iterations of the outer loop. In the m^{th} iteration of the loop, elements with indices differing in the m^{th} most significant bit are combined. As a result, elements combined during the first d iterations belong to different processes, and pairs of elements combined during the last $(r - d)$ iterations belong to the same processes. Hence, this parallel FFT algorithm requires interprocess interaction only during the first $d = \log p$ of the $\log n$ iterations. There is no interaction during the last $r - d$ iterations. Furthermore, in i^{th} of the first d iterations, all the elements that a process requires come from exactly one other process—the one whose label is different in the i^{th} most significant bit.

Each interaction operation exchanges n/p words of data. Therefore, the time spent in communication in the entire algorithm is $t_s \log p + t_w (n/p) \log p$. A process updates n/p elements of R during each of the $\log n$ iterations. If a complex multiplication and addition pair takes time t_c, then the parallel run time of the binary-exchange algorithm for n-point FFT on a p-node hypercube network is

$$T_P = t_c \frac{n}{p} \log n + t_s \log p + t_w \frac{n}{p} \log p. \tag{11.4}$$

The process-time product is $t_c n \log n + t_s p \log p + t_w n \log p$. For the parallel system to be cost-optimal, this product should be $O(n \log n)$—the sequential time complexity of the FFT algorithm. This is true for $p \leq n$.

The expressions for speedup and efficiency are given by the following equations:

$$\begin{aligned} S &= \frac{t_c n \log n}{T_P} \\ &= \frac{pn \log n}{n \log n + (t_s/t_c) p \log p + (t_w/t_c) n \log p} \\ E &= \frac{1}{1 + (t_s p \log p)/(t_c n \log n) + (t_w \log p)/(t_c \log n)} \end{aligned} \tag{11.5}$$

Scalability Analysis

From Section 11.1, we know that the problem size W for an n-point FFT is

$$W = n \log n. \tag{11.6}$$

Since an n-point FFT can utilize a maximum of n processes with the mapping of Figure 11.3, $n \geq p$ or $n \log n \geq p \log p$ to keep p processes busy. Thus, the isoefficiency function of this parallel FFT algorithm is $\Omega(p \log p)$ due to concurrency. We now derive the isoefficiency function for the binary exchange algorithm due to the different communication-related terms. We can rewrite Equation 11.5 as

$$\frac{t_s p \log p}{t_c n \log n} + \frac{t_w \log p}{t_c \log n} = \frac{1-E}{E}.$$

In order to maintain a fixed efficiency E, the expression $(t_s p \log p)/(t_c n \log n) + (t_w \log p)/(t_c \log n)$ should be equal to a constant $1/K$, where $K = E/(1-E)$. We have defined the constant K in this manner to keep the terminology consistent with Chapter **??**. As proposed in Section **??**, we use an approximation to obtain closed expressions for the isoefficiency function. We first determine the rate of growth of the problem size with respect to p that would keep the terms due to t_s constant. To do that, we assume $t_w = 0$. Now the condition for maintaining constant efficiency E is as follows:

$$\begin{aligned} \frac{t_s p \log p}{t_c n \log n} &= \frac{1}{K} \\ n \log n &= K \frac{t_s}{t_c} p \log p \\ W &= K \frac{t_s}{t_c} p \log p \end{aligned} \tag{11.7}$$

Equation 11.7 gives the isoefficiency function due to the overhead resulting from interaction latency or the message startup time.

Similarly, we derive the isoefficiency function due to the overhead resulting from t_w. We assume that $t_s = 0$; hence, a fixed efficiency E requires that the following relation be maintained:

$$\begin{aligned} \frac{t_w \log p}{t_c \log n} &= \frac{1}{K} \\ \log n &= K \frac{t_w}{t_c} \log p \\ n &= p^{K t_w / t_c} \\ n \log n &= K \frac{t_w}{t_c} p^{K t_w / t_c} \log p \\ W &= K \frac{t_w}{t_c} p^{K t_w / t_c} \log p \end{aligned} \tag{11.8}$$

If the term $K t_w / t_c$ is less than one, then the rate of growth of the problem size required by Equation 11.8 is less than $\Theta(p \log p)$. In this case, Equation 11.7 determines the overall isoefficiency function of this parallel system. However, if $K t_w / t_c$ exceeds one, then Equation 11.8 determines the overall isoefficiency function, which is now greater than the isoefficiency function of $\Theta(p \log p)$ given by Equation 11.7.

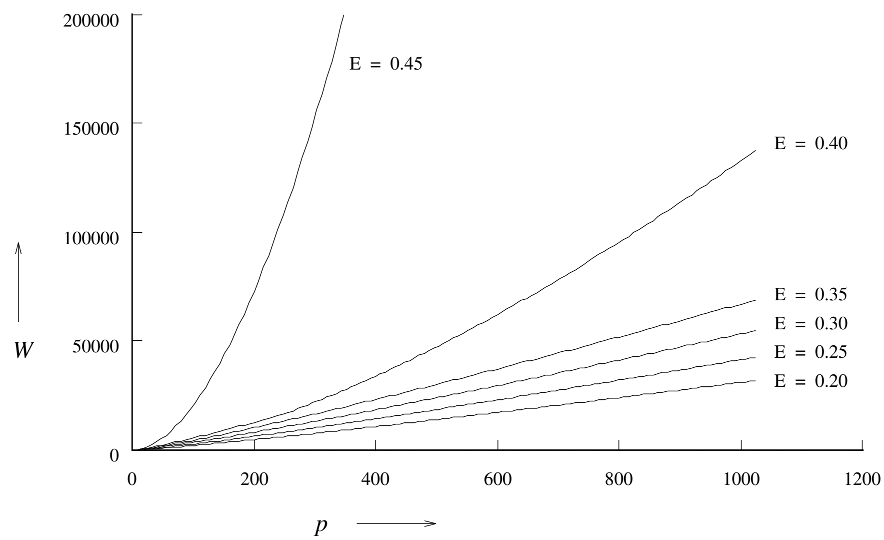


Figure 11.5 Isoefficiency functions of the binary-exchange algorithm on a hypercube with $t_c = 2$, $t_w = 4$, and $t_s = 25$ for various values of E.

For this algorithm, the asymptotic isoefficiency function depends on the relative values of K, t_w, and t_c. Here, K is an increasing function of the efficiency E to be maintained, t_w depends on the communication bandwidth of the parallel computer, and t_c depends on the speed of the processors being used. The FFT algorithm is unique in that the order of the isoefficiency function depends on the desired efficiency and hardware-dependent parameters. In fact, the efficiency corresponding to $Kt_w/t_c = 1$ (that is, $1/(1-E) = t_c/t_w$, or $E = t_c/(t_c + t_w)$) acts as a kind of threshold. For a fixed t_c and t_w, efficiencies up to the threshold can be obtained easily. For $E \leq t_c/(t_c + t_w)$, the asymptotic isoefficiency function is $\Theta(p \log p)$. Efficiencies much higher than the threshold $t_c/(t_c+t_w)$ can be obtained only if the problem size is extremely large. The reason is that for these efficiencies, the asymptotic isoefficiency function is $\Theta(p^{Kt_w/t_c} \log p)$. The following examples illustrate the effect of the value of Kt_w/t_c on the isoefficiency function.

Example 11.1 Threshold Effect in the Binary-Exchange Algorithm

Consider a hypothetical hypercube for which the relative values of the hardware parameters are given by $t_c = 2$, $t_w = 4$, and $t_s = 25$. With these values, the threshold efficiency $t_c/(t_c + t_w)$ is 0.33.

Now we study the isoefficiency functions of the binary-exchange algorithm on a hypercube for maintaining efficiencies below and above the threshold. The isoefficiency function of this algorithm due to concurrency is $p \log p$. From Equations 11.7 and 11.8, the isoefficiency function due to the t_s and t_w terms in the overhead function are $K(t_s/t_c)p \log p$ and $K(t_w/t_c)p^{Kt_w/t_c} \log p$, respectively. To maintain a given ef-

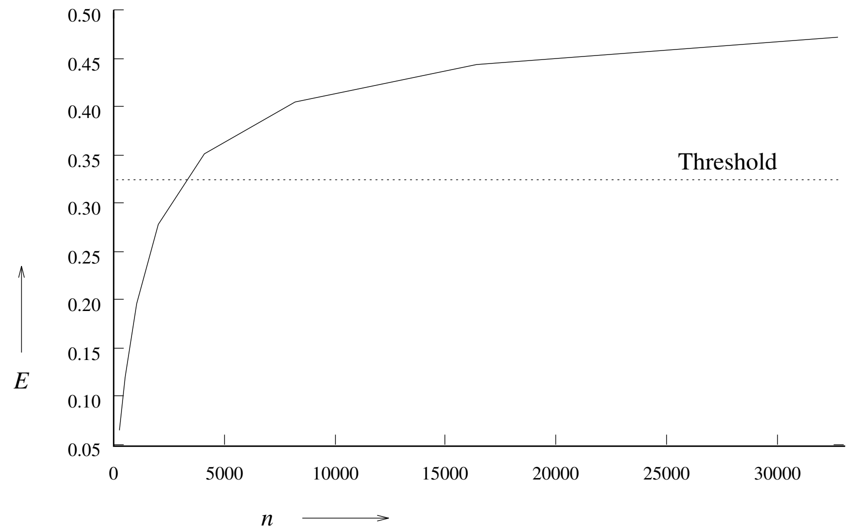


Figure 11.6 The efficiency of the binary-exchange algorithm as a function of n on a 256-process hypercube with $t_c = 2$, $t_w = 4$, and $t_s = 25$.

ficiency E (that is, for a given K), the overall isoefficiency function is given by:

$$W = \max\{p \log p,\, K \frac{t_s}{t_c} p \log p,\, K \frac{t_w}{t_c} p^{K t_w / t_c} \log p\}$$

Figure 11.5 shows the isoefficiency curves given by this function for E = 0.20, 0.25, 0.30, 0.35, 0.40, and 0.45. Notice that the various isoefficiency curves are regularly spaced for efficiencies up to the threshold. However, the problem sizes required to maintain efficiencies above the threshold are much larger. The asymptotic isoefficiency functions for E = 0.20, 0.25, and 0.30 are $\Theta(p \log p)$. The isoefficiency function for $E = 0.40$ is $\Theta(p^{1.33} \log p)$, and that for $E = 0.45$ is $\Theta(p^{1.64} \log p)$.

Figure 11.6 shows the efficiency curve of n-point FFTs on a 256-process hypercube with the same hardware parameters. The efficiency E is computed by using Equation 11.5 for various values of n, when p is equal to 256. The figure shows that the efficiency initially increases rapidly with the problem size, but the efficiency curve flattens out beyond the threshold. ■

Example 11.1 shows that there is a limit on the efficiency that can be obtained for reasonable problem sizes, and that the limit is determined by the ratio between the CPU speed and the communication bandwidth of the parallel computer being used. This limit can be raised by increasing the bandwidth of the communication channels. However, making the CPUs faster without increasing the communication bandwidth lowers the limit. Hence, the binary-exchange algorithm performs poorly on a parallel computer whose communication and computation speeds are not balanced. If the hardware is balanced with respect to its

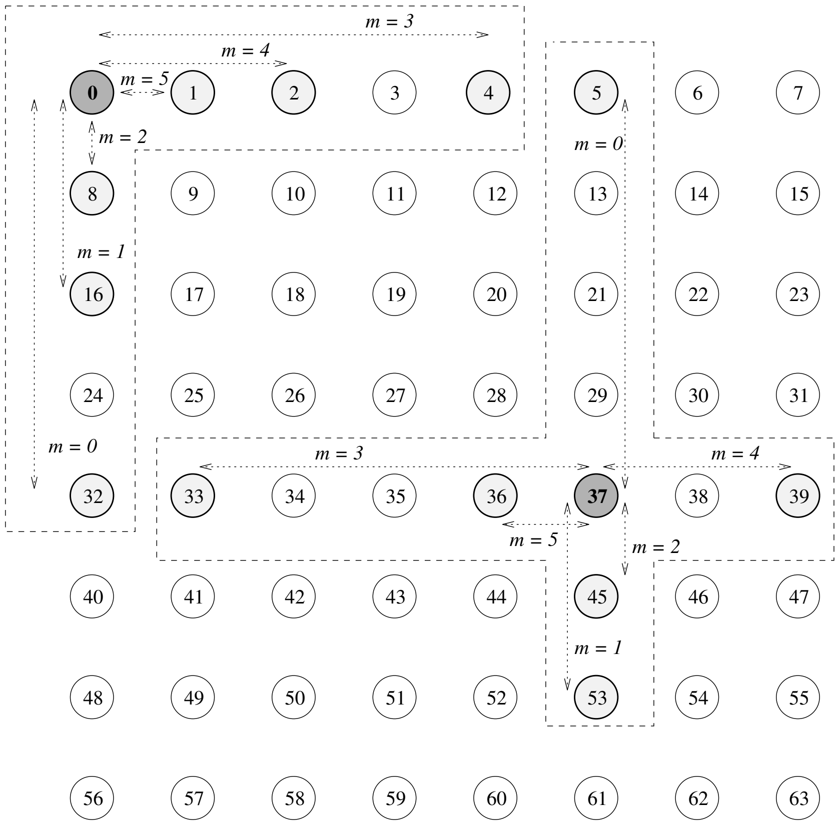


Figure 11.7 Data communication during an FFT computation on a square mesh of 64 processes. The figure shows all the processes with which the processes labeled 0 and 37 exchange data.

communication and computation speeds, then the binary-exchange algorithm is fairly scalable, and reasonable efficiencies can be maintained while increasing the problem size at the rate of $\Theta(p \log p)$.

11.2.2 Limited Bandwidth Network

Now we consider the implementation of the binary-exchange algorithm on a parallel computer whose cross-section bandwidth is less than $\Theta(p)$. We choose a mesh interconnection network to illustrate the algorithm and its performance characteristics. Assume that an n tasks are mapped onto p processes running on a mesh with $\sqrt{p}$ rows and $\sqrt{p}$ columns, and that $\sqrt{p}$ is a power of two. Let $n = 2^r$ and $p = 2^d$. Also assume that the processes are labeled in a row-major fashion and that the data are distributed in the same manner as for the hypercube; that is, an element with index $(b_0 b_1 \cdots b_{r-1})$ is mapped onto the process labeled $(b_0 \cdots b_{d-1})$.

As in case of the hypercube, communication takes place only during the first $\log p$

iterations between processes whose labels differ in one bit. However, unlike the hypercube, the communicating processes are not directly linked in a mesh. Consequently, messages travel over multiple links and there is overlap among messages sharing the same links. Figure 11.7 shows the messages sent and received by processes 0 and 37 during an FFT computation on a 64-process mesh. As the figure shows, process 0 communicates with processes 1, 2, 4, 8, 16, and 32. Note that all these processes lie in the same row or column of the mesh as that of process 0. Processes 1, 2, and 4 lie in the same row as process 0 at distances of 1, 2, and 4 links, respectively. Processes 8, 16, and 32 lie in the same column, again at distances of 1, 2, and 4 links. More precisely, in $\log \sqrt{p}$ of the $\log p$ steps that require communication, the communicating processes are in the same row, and in the remaining $\log \sqrt{p}$ steps, they are in the same column. The number of messages that share at least one link is equal to the number of links that each message traverses (Problem 11.9) because, during a given FFT iteration, all pairs of nodes exchange messages that traverse the same number of links.

The distance between the communicating processes in a row or a column grows from one link to $\sqrt{p}/2$ links, doubling in each of the $\log \sqrt{p}$ iterations. This is true for any process in the mesh, such as process 37 shown in Figure 11.7. Thus, the total time spent in performing rowwise communication is $\Sigma_{m=0}^{d/2-1}(t_s + t_w(n/p)2^m)$. An equal amount of time is spent in columnwise communication. Recall that we assumed that a complex multiplication and addition pair takes time t_c. Since a process performs n/p such calculations in each of the $\log n$ iterations, the overall parallel run time is given by the following equation:

$$
\begin{aligned}
T_P &= t_c \frac{n}{p} \log n + 2 \sum_{m=0}^{d/2-1} (t_s + t_w \frac{n}{p} 2^m) \\
&= t_c \frac{n}{p} \log n + 2(t_s \log \sqrt{p} + t_w \frac{n}{p} (\sqrt{p} - 1)) \\
&\approx t_c \frac{n}{p} \log n + t_s \log p + 2t_w \frac{n}{\sqrt{p}}
\end{aligned}
\tag{11.9}
$$

The speedup and efficiency are given by the following equations:

$$
\begin{aligned}
S &= \frac{t_c n \log n}{T_P} \\
&= \frac{pn \log n}{n \log n + (t_s/t_c) p \log p + 2(t_w/t_c) n \sqrt{p}} \\
E &= \frac{1}{1 + (t_s p \log p)/(t_c n \log n) + 2(t_w \sqrt{p})/(t_c \log n)}
\end{aligned}
\tag{11.10}
$$

The process-time product of this parallel system is $t_c n \log n + t_s p \log p + 2t_w n \sqrt{p}$. The process-time product should be $O(n \log n)$ for cost-optimality, which is obtained when $\sqrt{p} = O(\log n)$, or $p = O(\log^2 n)$. Since the communication term due to t_s in Equation 11.9 is the same as for the hypercube, the corresponding isoefficiency function is again $\Theta(p \log p)$ as given by Equation 11.7. By performing isoefficiency analysis along the same lines as in Section 11.2.1, we can show that the isoefficiency function due to the

t_w term is $2K(t_w/t_c)2^{2K(t_w/t_c)\sqrt{p}}\sqrt{p}$ (Problem 11.5). Given this isoefficiency function, the problem size must grow exponentially with the number of processes to maintain constant efficiency. Hence, the binary-exchange FFT algorithm is not very scalable on a mesh.

The communication overhead of the binary-exchange algorithm on a mesh cannot be reduced by using a different mapping of the sequences onto the processes. In any mapping, there is at least one iteration in which pairs of processes that communicate with each other are at least $\sqrt{p}/2$ links apart (Problem 11.2). Hence, the communication cost in this parallel system cannot be improved asymptotically.

11.2.3 Extra Computations in Parallel FFT

So far, we have described a parallel formulation of the FFT algorithm on a hypercube and a mesh, and have discussed its performance and scalability in the presence of communication overhead on both architectures. In this section, we discuss another source of overhead that can be present in a parallel FFT implementation.

Recall from Algorithm 11.2 that the computation step of line 12 multiplies a power of ω (a twiddle factor) with an element of S. For an n-point FFT, line 12 executes $n \log n$ times in the sequential algorithm. However, only n distinct powers of ω (that is, ω^0, ω^1, ω^2, ..., ω^{n-1}) are used in the entire algorithm. So some of the twiddle factors are used repeatedly. In a serial implementation, it is useful to precompute and store all n twiddle factors before starting the main algorithm. That way, the computation of twiddle factors requires only $\Theta(n)$ complex operations rather than the $\Theta(n \log n)$ operations needed to compute all twiddle factors in each iteration of line 12.

In a parallel implementation, the total work required to compute the twiddle factors cannot be reduced to $\Theta(n)$. The reason is that, even if a certain twiddle factor is used more than once, it might be used on different processes at different times. If FFTs of the same size are computed on the same number of processes, every process needs the same set of twiddle factors for each computation. In this case, the twiddle factors can be precomputed and stored, and the cost of their computation can be amortized over the execution of all instances of FFTs of the same size. However, if we consider only one instance of FFT, then twiddle factor computation gives rise to additional overhead in a parallel implementation, because it performs more overall operations than the sequential implementation.

As an example, consider the various powers of ω used in the three iterations of an 8-point FFT. In the m^{th} iteration of the loop starting on line 5 of the algorithm, ω^l is computed for all i $(0 \le i < n)$, such that l is the integer obtained by reversing the order of the $m+1$ most significant bits of i and then padding them by $\log n - (m+1)$ zeros to the right (refer to Figure 11.1 and Algorithm 11.2 to see how l is derived). Table 11.1 shows the binary representation of the powers of ω required for all values of i and m for an 8-point FFT.

If eight processes are used, then each process computes and uses one column of Table 11.1. Process 0 computes just one twiddle factor for all its iterations, but some processes (in this case, all other processes 2–7) compute a new twiddle factor in each of the three iterations. If $p = n/2 = 4$, then each process computes two consecutive columns of

Table 11.1 The binary representation of the various powers of ω calculated in different iterations of an 8-point FFT (also see Figure 11.1). The value of m refers to the iteration number of the outer loop, and i is the index of the inner loop of Algorithm 11.2.

	i							
	0	1	2	3	4	5	6	7
$m = 0$	000	000	000	000	100	100	100	100
$m = 1$	000	000	100	100	010	010	110	110
$m = 2$	000	100	010	110	001	101	011	111

Table 11.2 The maximum number of new powers of ω used by any process in each iteration of an 8-point FFT computation.

	$p = 1$	$p = 2$	$p = 4$	$p = 8$
$m = 0$	2	1	1	1
$m = 1$	2	2	1	1
$m = 2$	4	4	2	1
Total = $h(8, p)$	8	7	4	3

the table. In this case, the last process computes the twiddle factors in the last two columns of the table. Hence, the last process computes a total of four different powers—one each for $m = 0$ (100) and $m = 1$ (110), and two for $m = 2$ (011 and 111). Although different processes may compute a different number of twiddle factors, the total overhead due to the extra work is proportional to p times the maximum number of twiddle factors that any single process computes. Let $h(n, p)$ be the maximum number of twiddle factors that any of the p processes computes during an n-point FFT. Table 11.2 shows the values of $h(8, p)$ for $p = 1, 2, 4$, and 8. The table also shows the maximum number of new twiddle factors that any single process computes in each iteration.

The function h is defined by the following recurrence relation (Problem 11.6):

$$\begin{aligned} h(n, 1) &= n \\ h(p, p) &= \log p && (p \neq 1) \\ h(n, p) &= h(n, 2p) + n/p - 1 && (p \neq 1, n > p) \end{aligned}$$

The solution to this recurrence relation for $p > 1$ and $n \geq p$ is

$$h(n, p) = 2(\frac{n}{p} - 1) + \log p.$$

Thus, if it takes time t_c' to compute one twiddle factor, then at least one process spends $t_c' 2(n/p - 1) + t_c' \log p$ time computing twiddle factors. The total cost of twiddle factor computation, summed over all processes, is $2t_c'(n - p) + t_c' p \log p$. Since even a serial implementation incurs a cost of $t_c^{'}n$ in computing twiddle factors, the total parallel overhead due to extra work ($T_o^{extra\_work}$) is given by the following equation:

$$\begin{aligned} T_o^{extra\_work} &= (2t_c^{'}(n-p) + t_c^{'} p \log p) - t_c^{'} n \\ &= t_c^{'}(n + p(\log p - 2)) \\ &= \Theta(n) + \Theta(p \log p) \end{aligned}$$

This overhead is independent of the architecture of the parallel computer used for the FFT computation. The isoefficiency function due to $T_o^{extra\_work}$ is $\Theta(p \log p)$. Since this term is of the same order as the isoefficiency terms due to message startup time and concurrency, the extra computations do not affect the overall scalability of parallel FFT.

11.3 The Transpose Algorithm

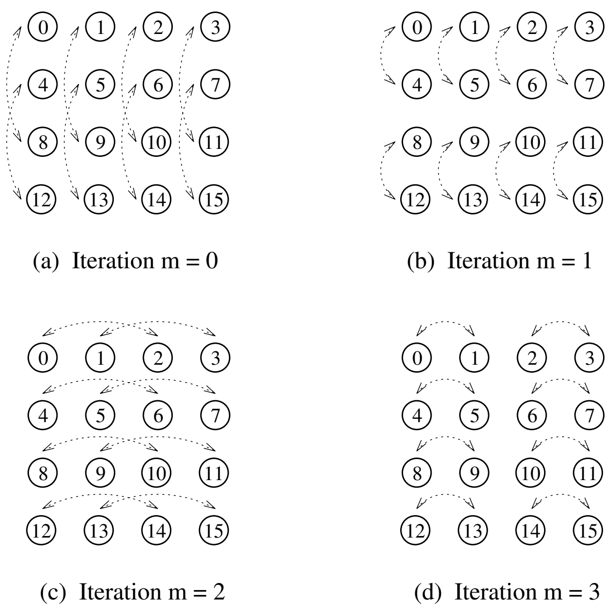
The binary-exchange algorithm yields good performance on on parallel computers with sufficiently high communication bandwidth with respect to the processing speed of the CPUs. Efficiencies below a certain threshold can be maintained while increasing the problem size at a moderate rate with an increasing number of processes. However, this threshold is very low if the communication bandwidth of the parallel computer is low compared to the speed of its processors. In this section, we describe a different parallel formulation of FFT that trades off some efficiency for a more consistent level of parallel performance. This parallel algorithm involves matrix transposition, and hence, is called the ***transpose algorithm***.

The performance of the transpose algorithm is worse than that of the binary-exchange algorithm for efficiencies below the threshold. However, it is much easier to obtain efficiencies above the binary-exchange algorithm's threshold using the transpose algorithm. Thus, the transpose algorithm is particularly useful when the ratio of communication bandwidth to CPU speed is low and high efficiencies are desired. On a hypercube or a p-node network with $\Theta(p)$ bisection width, the transpose algorithm has a fixed asymptotical isoefficiency function of $\Theta(p^2 \log p)$. That is, the order of this isoefficiency function is independent of the ratio of the speed of point-to-point communication and the computation.

11.3.1 Two-Dimensional Transpose Algorithm

The simplest transpose algorithm requires a single transpose operation over a two-dimensional array; hence, we call this algorithm the ***two-dimensional transpose algorithm***.

Assume that $\sqrt{n}$ is a power of 2, and that the sequences of size n used in Algorithm 11.2 are arranged in an $\sqrt{n} \times \sqrt{n}$ two-dimensional square array, as shown in Figure 11.8 for $n = 16$. Recall that computing the FFT of a sequence of n points requires $\log n$ iterations of the outer loop of Algorithm 11.2. If the data are arranged as shown in Figure 11.8, then the FFT computation in each column can proceed independently for $\log \sqrt{n}$ iterations without



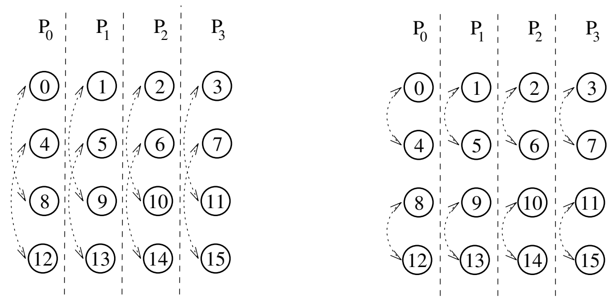
(a) Iteration m = 0 (b) Iteration m = 1

(c) Iteration m = 2 (d) Iteration m = 3

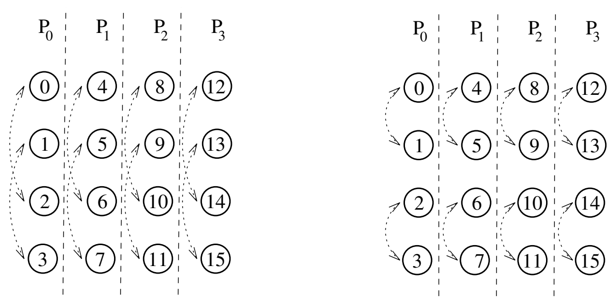
Figure 11.8 The pattern of combination of elements in a 16-point FFT when the data are arranged in a 4×4 two-dimensional square array.

any column requiring data from any other column. Similarly, in the remaining $\log \sqrt{n}$ iterations, computation proceeds independently in each row without any row requiring data from any other row. Figure 11.8 shows the pattern of combination of the elements for a 16-point FFT. The figure illustrates that if data of size n are arranged in an $\sqrt{n} \times \sqrt{n}$ array, then an n-point FFT computation is equivalent to independent $\sqrt{n}$-point FFT computations in the columns of the array, followed by independent $\sqrt{n}$-point FFT computations in the rows.

If the $\sqrt{n} \times \sqrt{n}$ array of data is transposed after computing the $\sqrt{n}$-point column FFTs, then the remaining part of the problem is to compute the $\sqrt{n}$-point columnwise FFTs of the transposed matrix. The transpose algorithm uses this property to compute the FFT in parallel by using a columnwise striped partitioning to distribute the $\sqrt{n} \times \sqrt{n}$ array of data among the processes. For instance, consider the computation of the 16-point FFT shown in Figure 11.9, where the 4×4 array of data is distributed among four processes such that each process stores one column of the array. In general, the two-dimensional transpose algorithm works in three phases. In the first phase, a $\sqrt{n}$-point FFT is computed for each column. In the second phase, the array of data is transposed. The third and final phase is identical to the first phase, and involves the computation of $\sqrt{n}$-point FFTs for each column of the transposed array. Figure 11.9 shows that the first and third phases of the algorithm do not require any interprocess communication. In both these phases, all $\sqrt{n}$ points for each columnwise FFT computation are available on the same process. Only the



(a) Steps in phase 1 of the transpose algorithm (before transpose)



(b) Steps in phase 3 of the transpose algorithm (after transpose)

Figure 11.9 The two-dimensional transpose algorithm for a 16-point FFT on four processes.

second phase requires communication for transposing the $\sqrt{n} \times \sqrt{n}$ matrix.

In the transpose algorithm shown in the Figure 11.9, one column of the data array is assigned to one process. Before analyzing the transpose algorithm further, consider the more general case in which p processes are used and $1 \leq p \leq \sqrt{n}$. The $\sqrt{n} \times \sqrt{n}$ array of data is striped into blocks, and one block of $\sqrt{n}/p$ rows is assigned to each process. In the first and third phases of the algorithm, each process computes $\sqrt{n}/p$ FFTs of size $\sqrt{n}$ each. The second phase transposes the $\sqrt{n} \times \sqrt{n}$ matrix, which is distributed among p processes with a one-dimensional partitioning. Recall from Sections **??** and **??** that such a transpose requires an all-to-all personalized communication.

Now we derive an expression for the parallel run time of the two-dimensional transpose algorithm. The only inter-process interaction in this algorithm occurs when the $\sqrt{n} \times \sqrt{n}$ array of data partioned along columns or rows and mapped onto p processes is transposed. Replacing the message size m by n/p—the amount of data owned by each process—in the expression for all-to-all personalized communication in Table **??** yields $t_s(p-1) + t_w n/p$ as the time spent in the second phase of the algorithm. The first and the third phase each take $t_c \times \sqrt{n}/p \times \sqrt{n} \log \sqrt{n}$ time. Thus, the parallel run time of the transpose algorithm on a hypercube or any $\Theta(p)$ bisection width network is given by the following equation:

$$\begin{aligned} T_P &= 2t_c \frac{\sqrt{n}}{p} \sqrt{n} \log \sqrt{n} + t_s(p-1) + t_w \frac{n}{p} \\ &= t_c \frac{n}{p} \log n + t_s(p-1) + t_w \frac{n}{p} \end{aligned} \tag{11.11}$$

The expressions for speedup and efficiency are as follows:

$$\begin{aligned} S &\approx \frac{pn \log n}{n \log n + (t_s/t_c)p^2 + (t_w/t_c)n} \\ E &\approx \frac{1}{1 + (t_s p^2)/(t_c n \log n) + t_w/(t_c \log n)} \end{aligned} \tag{11.12}$$

The process-time product of this parallel system is $t_c n \log n + t_s p^2 + t_w n$. This parallel system is cost-optimal if $n \log n = \Omega(p^2 \log p)$.

Note that the term associated with t_w in the expression for efficiency in Equation 11.12 is independent of the number of processes. The degree of concurrency of this algorithm requires that $\sqrt{n} = \Omega(p)$ because at most $\sqrt{n}$ processes can be used to partition the $\sqrt{n} \times \sqrt{n}$ array of data in a striped manner. As a result, $n = \Omega(p^2)$, or $n \log n = \Omega(p^2 \log p)$. Thus, the problem size must increase at least as fast as $\Theta(p^2 \log p)$ with respect to the number of processes to use all of them efficiently. The overall isoefficiency function of the two-dimensional transpose algorithm is $\Theta(p^2 \log p)$ on a hypercube or another interconnection network with $\Theta(p)$ bisection width. This isoefficiency function is independent of the ratio of t_w for point-to-point communication and t_c. On a network whose cross-section bandwidth b is less than $\Theta(p)$ for p nodes, the t_w term must be multiplied by an appropriate expression of $\Theta(p/b)$ in order to derive T_P, S, E, and the isoefficiency function (Problem 11.7).

Comparison with the Binary-Exchange Algorithm

A comparison of Equations 11.4 and 11.11 shows that the transpose algorithm has a much higher overhead than the binary-exchange algorithm due to the message startup time t_s, but has a lower overhead due to per-word transfer time t_w. As a result, either of the two algorithms may be faster depending on the relative values of t_s and t_w. If the latency t_s is very low, then the transpose algorithm may be the algorithm of choice. On the other hand, the binary-exchange algorithm may perform better on a parallel computer with a high communication bandwidth but a significant startup time.

Recall from Section 11.2.1 that an overall isoefficiency function of $\Theta(p \log p)$ can be realized by using the binary-exchange algorithm if the efficiency is such that $Kt_w/t_c \le 1$, where $K = E/(1-E)$. If the desired efficiency is such that $Kt_w/t_c = 2$, then the overall isoefficiency function of both the binary-exchange and the two-dimensional transpose algorithms is $\Theta(p^2 \log p)$. When $Kt_w/t_c > 2$, the two-dimensional transpose algorithm is

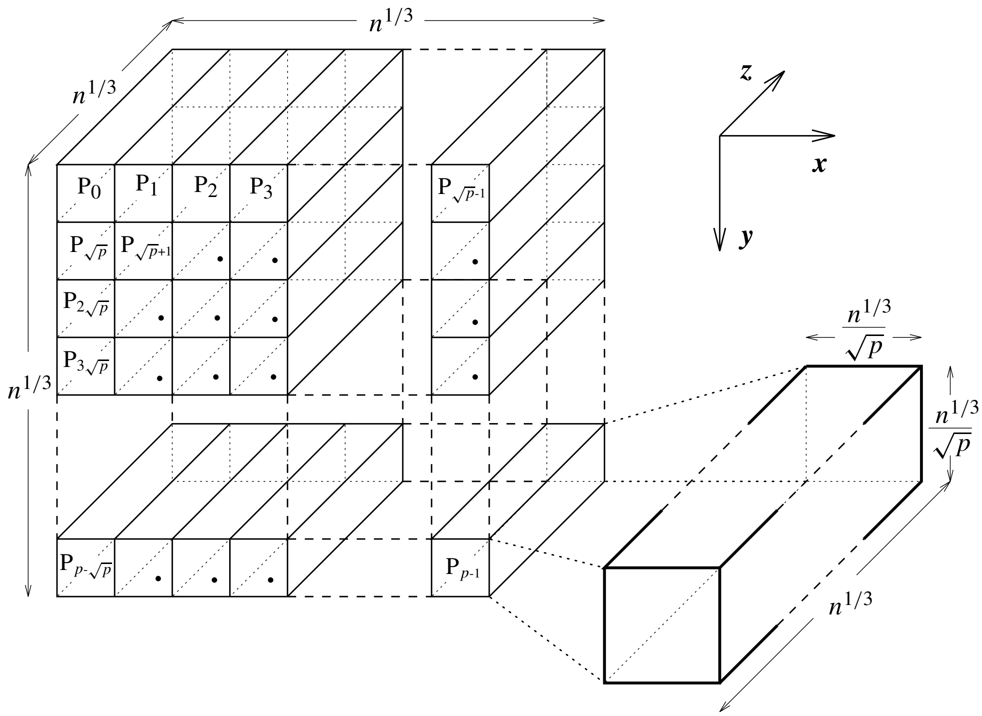


Figure 11.10 Data distribution in the three-dimensional transpose algorithm for an n-point FFT on p processes ($\sqrt{p} \leq n^{1/3}$).

more scalable than the binary-exchange algorithm; hence, the former should be the algorithm of choice, provided that $n \geq p^2$. Note, however, that the transpose algorithm yields a performance benefit over the binary-exchange algorithm only if the target architecture has a cross-section bandwidth of $\Theta(p)$ for p nodes (Problem 11.7).

11.3.2 The Generalized Transpose Algorithm

In the two-dimensional transpose algorithm, the input of size n is arranged in a $\sqrt{n} \times \sqrt{n}$ two-dimensional array that is partitioned along one dimension on p processes. These processes, irrespective of the underlying architecture of the parallel computer, can be regarded as arranged in a logical one-dimensional linear array. As an extension of this scheme, consider the n data points to be arranged in an $n^{1/3} \times n^{1/3} \times n^{1/3}$ three-dimensional array mapped onto a logical $\sqrt{p} \times \sqrt{p}$ two-dimensional mesh of processes. Figure 11.10 illustrates this mapping. To simplify the algorithm description, we label the three axes of the three-dimensional array of data as x, y, and z. In this mapping, the x-y plane of the array is checkerboarded into $\sqrt{p} \times \sqrt{p}$ parts. As the figure shows, each process stores $(n^{1/3}/\sqrt{p}) \times (n^{1/3}/\sqrt{p})$ columns of data, and the length of each column (along the z-axis) is $n^{1/3}$. Thus, each process has $(n^{1/3}/\sqrt{p}) \times (n^{1/3}/\sqrt{p}) \times n^{1/3} = n/p$ elements of data.

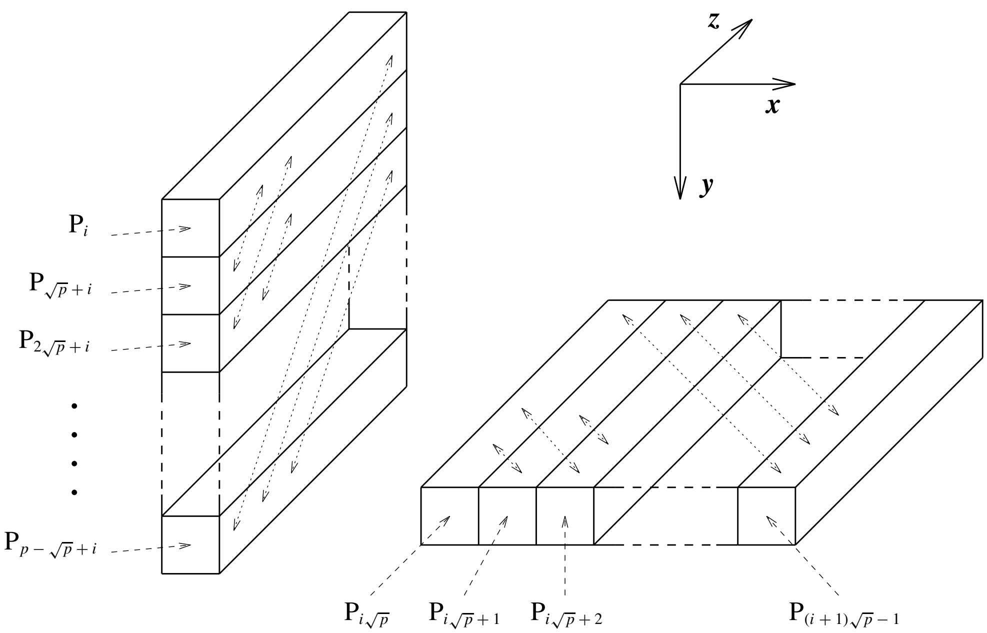
Recall from Section 11.3.1 that the FFT of a two-dimensionally arranged input of size $\sqrt{n} \times \sqrt{n}$ can be computed by first computing the $\sqrt{n}$-point one-dimensional FFTs of all the columns of the data and then computing the $\sqrt{n}$-point one-dimensional FFTs of all the rows. If the data are arranged in an $n^{1/3} \times n^{1/3} \times n^{1/3}$ three-dimensional array, the entire n-point FFT can be computed similarly. In this case, $n^{1/3}$-point FFTs are computed over the elements of the columns of the array in all three dimensions, choosing one dimension at a time. We call this algorithm the ***three-dimensional transpose algorithm***. This algorithm can be divided into the following five phases:

1. In the first phase, $n^{1/3}$-point FFTs are computed on all the rows along the z-axis.
2. In the second phase, all the $n^{1/3}$ cross sections of size $n^{1/3} \times n^{1/3}$ along the y-z plane are transposed.
3. In the third phase, $n^{1/3}$-point FFTs are computed on all the rows of the modified array along the z-axis.
4. In the fourth phase, each of the $n^{1/3} \times n^{1/3}$ cross sections along the x-z plane is transposed.
5. In the fifth and final phase, $n^{1/3}$-point FFTs of all the rows along the z-axis are computed again.

For the data distribution shown in Figure 11.10, in the first, third, and fifth phases of the algorithm, all processes perform $(n^{1/3}/\sqrt{p}) \times (n^{1/3}/\sqrt{p})$ FFT computations, each of size $n^{1/3}$. Since all the data for performing these computations are locally available on each process, no interprocess communication is involved in these three odd-numbered phases. The time spent by a process in each of these phases is $t_c n^{1/3} \log(n^{1/3}) \times (n^{1/3}/\sqrt{p}) \times (n^{1/3}/\sqrt{p})$. Thus, the total time that a process spends in computation is $t_c(n/p)\log n$.

Figure 11.11 illustrates the second and the fourth phases of the three-dimensional transpose algorithm. As Figure 11.11(a) shows, the second phase of the algorithm requires transposing square cross sections of size $n^{1/3} \times n^{1/3}$ along the y-z plane. Each column of $\sqrt{p}$ processes performs the transposition of $(n^{1/3}/\sqrt{p})$ such cross sections. This transposition involves all-to-all personalized communications among groups of $\sqrt{p}$ processes with individual messages of size $n/p^{3/2}$. If a p-node network with $\Theta(p)$ bisection width is used, this phase takes time $t_s(\sqrt{p}-1) + t_w n/p + (t_h\sqrt{p}/4)\log p$. The fourth phase, shown in Figure 11.11(b), is similar. Here each row of $\sqrt{p}$ processes performs the transpose of $(n^{1/3}/\sqrt{p})$ cross sections along the x-z plane. Again, each cross section consists of $n^{1/3} \times n^{1/3}$ data elements. The communication time of this phase is the same as that of the second phase. The total parallel run time of the three-dimensional transpose algorithm for an n-point FFT is

$$T_P = t_c\frac{n}{p}\log n + 2t_s(\sqrt{p}-1) + 2t_w\frac{n}{p} + \frac{1}{2}t_h\sqrt{p}\log p. \tag{11.13}$$



(a) Transpose in the i^{th} column of processors during phase 2

(b) Transpose in the i^{th} row of processors during phase 4

Figure 11.11 The communication (transposition) phases in the three-dimensional transpose algorithm for an n-point FFT on p processes.

Having studied the two- and three-dimensional transpose algorithms, we can derive a more general q-dimensional transpose algorithm similarly. Let the n-point input be arranged in a logical q-dimensional array of size $n^{1/q} \times n^{1/q} \times \cdots \times n^{1/q}$ (a total of q terms). Now the entire n-point FFT computation can be viewed as q subcomputations. Each of the q subcomputations along a different dimension consists of $n^{(q-1)/q}$ FFTs over $n^{1/q}$ data points. We map the array of data onto a logical $(q-1)$-dimensional array of p processes, where $p \leq n^{(q-1)/q}$, and $p = 2^{(q-1)s}$ for some integer s. The FFT of the entire data is now computed in $(2q-1)$ phases (recall that there are three phases in the two-dimensional transpose algorithm and five phases in the three-dimensional transpose algorithm). In the q odd-numbered phases, each process performs $n^{(q-1)/q}/p$ of the required $n^{1/q}$-point FFTs. The total computation time for each process over all q computation phases is the product of: q (the number of computation phases), $n^{(q-1)/q}/p$ (the number of $n^{1/q}$-point FFTs computed by each process in each computation phase), and $t_c n^{1/q} \log(n^{1/q})$ (the time to compute a single $n^{1/q}$-point FFT). Multiplying these terms gives a total computation time of $t_c(n/p)\log n$.

In each of the $(q-1)$ even-numbered phases, sub-arrays of size $n^{1/q} \times n^{1/q}$ are transposed on rows of the q-dimensional logical array of processes. Each such row contains

$p^{1/(q-1)}$ processes. One such transpose is performed along every dimension of the $(q-1)$-dimensional process array in each of the $(q-1)$ communication phases. The time spent in communication in each transposition is $t_s(p^{1/(q-1)} - 1) + t_w n/p$. Thus, the total parallel run time of the q-dimensional transpose algorithm for an n-point FFT on a p-node network with $\Theta(p)$ bisection width is

$$T_P = t_c \frac{n}{p} \log n + (q-1)t_s(p^{1/(q-1)} - 1) + (q-1)t_w \frac{n}{p}. \tag{11.14}$$

Equation 11.14 can be verified by replacing q with 2 and 3, and comparing the result with Equations 11.11 and 11.13, respectively.

A comparison of Equations 11.11, 11.13, 11.14, and 11.4 shows an interesting trend. As the dimension q of the transpose algorithm increases, the communication overhead due to t_w increases, but that due to t_s decreases. The binary-exchange algorithm and the two-dimensional transpose algorithms can be regarded as two extremes. The former minimizes the overhead due to t_s but has the largest overhead due to t_w. The latter minimizes the overhead due to t_w but has the largest overhead due to t_s. The variations of the transpose algorithm for $2 < q < \log p$ lie between these two extremes. For a given parallel computer, the specific values of t_c, t_s, and t_w determine which of these algorithms has the optimal parallel run time (Problem 11.10).

Note that, from a practical point of view, only the binary-exchange algorithm and the two- and three-dimensional transpose algorithms are feasible. Higher-dimensional transpose algorithms are very complicated to code. Moreover, restrictions on n and p limit their applicability. These restrictions for a q-dimensional transpose algorithm are that n must be a power of two that is a multiple of q, and that p must be a power of 2 that is a multiple of $(q-1)$. In other words, $n = 2^{qr}$, and $p = 2^{(q-1)s}$, where q, r, and s are integers.

Example 11.2 A Comparison of Binary-Exchange, 2-D Transpose, and 3-D Transpose Algorithms

This example shows that either the binary-exchange algorithm or any of the transpose algorithms may be the algorithm of choice for a given parallel computer, depending on the size of the FFT. Consider a 64-process version of the hypercube described in Example 11.1 with $t_c = 2$, $t_s = 25$, and $t_w = 4$. Figure 11.12 shows speedups attained by the binary-exchange algorithm, the 2-D transpose algorithm, and the 3-D transpose algorithm for different problem sizes. The speedups are based on the parallel run times given by Equations 11.4, 11.11, and 11.13, respectively. The figure shows that for different ranges of n, a different algorithm provides the highest speedup for an n-point FFT. For the given values of the hardware parameters, the binary-exchange algorithm is best suited for very low granularity FFT computations, the 2-D transpose algorithm is best for very high granularity computations, and the 3-D transpose algorithm's speedup is the maximum for intermediate granularities. ■

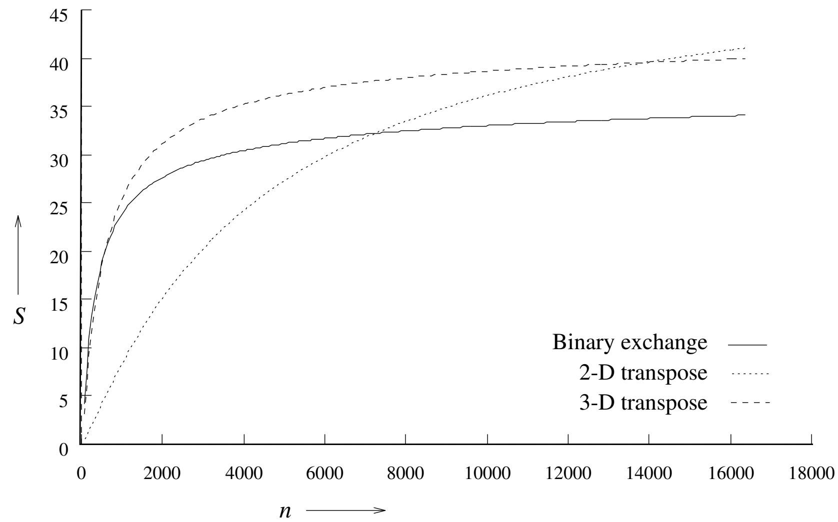


Figure 11.12 A comparison of the speedups obtained by the binary-exchange, 2-D transpose, and 3-D transpose algorithms on a 64-process hypercube with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$. The hypercube is assumed to support cut-through routing.

11.4 Bibliographic Remarks

Due to the important role of Fourier transform in scientific and technical computations, there has been great interest in implementing FFT on parallel computers and on studying its performance. Swarztrauber [Swa87] describes many implementations of the FFT algorithm on vector and parallel computers. Cvetanovic [Cve87] and Norton and Silberger [NS87] give a comprehensive performance analysis of the FFT algorithm on pseudo-shared-memory architectures such as the IBM RP-3. They consider various partitionings of data among memory blocks and, in each case, obtain expressions for communication overhead and speedup in terms of problem size, number of processes, memory latency, CPU speed, and speed of communication. Aggarwal, Chandra, and Snir [ACS89] analyze the performance of FFT and other algorithms on LPRAM—a new model for parallel computation. This model differs from the standard PRAM model in that remote accesses are more expensive than local accesses in an LPRAM. Parallel FFT algorithms and their implementation and experimental evaluation on various architectures have been pursued by many other researchers [AGGM90, Bai90, BCJ90, BKH89, DT89, GK93, JKFM89, KA88, Loa92]. Parts of the discussion of the transpose algorithm in this chapter are due to Rao [Rao92].

The basic FFT algorithm whose parallel formulations are discussed in this chapter is called the unordered FFT because the elements of the output sequence are stored in bit-reversed index order. In other words, the frequency spectrum of the input signal is obtained by reordering the elements of the output sequence Y produced by Algorithm 11.2 in such a way that for all i, $Y[i]$ is replaced by $Y[j]$, where j is obtained by reversing the bits in the binary representation of i. This is a permutation operation (Section **??**) and is known as ***bit reversal***. Norton and Silberger [NS87] show that an ordered transform can be obtained with at most $2d + 1$ communication steps, where $d = \log p$. Since the unordered FFT computation requires only d communication steps, the total communication overhead in the case of ordered FFT is roughly double of that for unordered FFT. Clearly, an unordered transform is preferred where applicable. The output sequence need not be ordered when the transform is used as a part of a larger computation and as such remains invisible to the user [Swa87]. In many practical applications of FFT, such as convolution and solution of the discrete Poisson equation, bit reversal can be avoided [Loa92]. If required, bit reversal can be performed by using an algorithm described by Van Loan [Loa92] for a distributed-memory parallel computer. The asymptotic communication complexity of this algorithm is the same as that of the binary-exchange algorithm on a hypercube.

Several variations of the simple FFT algorithm presented here have been suggested in the literature. Gupta and Kumar [GK93] show that the total communication overhead for mesh and hypercube architectures is the same for the one- and two-dimensional FFTs. Certain schemes for computing the DFT have been suggested that involve fewer arithmetic operations on a serial computer than the simple Cooley-Tukey FFT algorithm requires [Nus82, RB76, Win77]. Notable among these are computing one-dimensional FFTs with radix greater than two and computing multidimensional FFTs by transforming them into a set of one-dimensional FFTs by using the polynomial transform method. A radix-q FFT is computed by splitting the input sequence of size n into q sequences of size n/q each, computing the q smaller FFTs, and then combining the result. For example, in a radix-4 FFT, each step computes four outputs from four inputs, and the total number of iterations is $\log_4 n$ rather than $\log_2 n$. The input length should, of course, be a power of four. Despite the reduction in the number of iterations, the aggregate communication time for a radix-q FFT remains the same as that for radix-2. For example, for a radix-4 algorithm on a hypercube, each communication step now involves four processes distributed in two dimensions rather than two processes in one dimension. In contrast, the number of multiplications in a radix-4 FFT is 25 percent fewer than in a radix-2 FFT [Nus82]. This number can be marginally improved by using higher radices, but the amount of communication remains unchanged.

Problems

11.1 Let the serial run time of an n-point FFT computation be $t_c n \log n$. Consider its implementation on an architecture on which the parallel run time is $(t_c n \log n)/p + (t_w n \log p)/p$. Assume that $t_c = 1$ and $t_w = 0.2$.

1. Write expressions for the speedup and efficiency.
2. What is the isoefficiency function if an efficiency of 0.6 is desired?

3. How will the isoefficiency function change (if at all) if an efficiency of 0.4 is desired?
4. Repeat parts (b) and (c) for the case in which $t_w = 1$ and everything else is the same.

11.2 **[Tho83]** Show that, while performing FFT on a square mesh of p processes by using any mapping of data onto the processes, there is at least one iteration in which the pairs of processes that need to communicate are at least $\sqrt{p}/2$ links apart.

11.3 **[GK93]** Show that the parallel run time of an n-point FFT on a p-process mesh is the same for both store-and-forward and cut-through routing schemes.

11.4 Describe the communication pattern of the binary-exchange algorithm on a linear array of p processes. What are the parallel run time, speedup, efficiency, and isoefficiency function of the binary-exchange algorithm on a linear array?

11.5 Show that, if $t_s = 0$, the isoefficiency function of the binary-exchange algorithm on a mesh is given by $W = 2K(t_w/t_c)2^{2K(t_w/t_c)\sqrt{p}}\sqrt{p}$.
Hint: Use Equation 11.10.

11.6 Prove that the maximum number of twiddle factors computed by any process in the parallel implementation of an n-point FFT using p processes is given by the recurrence relation given in Section 11.2.3.

11.7 Derive expressions for the parallel run time, speedup, and efficiency of the two-dimensional transpose algorithm described in Section 11.3.1 for an n-point FFT on a p-node two-dimensional mesh and a linear array of p nodes.

11.8 Ignoring t_s, by what factor should the communication bandwidth of a p-process mesh with cut-through routing be increased so that it yields the same performance on the two-dimensional transpose algorithm for an n-point FFT on a p-node hypercube with cut-through routing?

11.9 Assume that the cost of building a network is proportional to its bisection bandwidth. What are the relative costs of mesh and hypercube networks employing the same number of processes that yield identical performance on the two-dimensional transpose algorithm for FFT?

11.10 You are given the following sets of communication-related constants for a hypercube network with cut-through routing: (i) $t_s = 250$, $t_w = 1$, (ii) $t_s = 50$, $t_w = 1$, (iii) $t_s = 10$, $t_w = 1$, (iv) $t_s = 2$, $t_w = 1$, and (v) $t_s = 0$, $t_w = 1$.

1. Given a choice among the binary-exchange algorithm and the two-, three-, four-, and five-dimensional transpose algorithms, which one would you use for $n = 2^{15}$ and $p = 2^{12}$ for each of the preceding sets of values of t_s and t_w?
2. Repeat part (a) for (1) $n = 2^{12}$, $p = 2^6$, and (2) $n = 2^{20}$, $p = 2^{12}$.

11.11 **[GK93]** Consider computing an n-point FFT on a $\sqrt{p} \times \sqrt{p}$ mesh. If the channel bandwidth grows at a rate of $\Theta(p^x)$ $(x > 0)$ with the number of nodes p in the mesh, show that the isoefficiency function due to communication overhead is $\Theta(p^{0.5-x} 2^{2(t_w/t_c)p^{0.5-x}})$ and that due to concurrency is $\Theta(p^{1+x} \log p)$. Also show that the best possible isoefficiency for FFT on a mesh is $\Theta(p^{1.5} \log p)$, even if the channel bandwidth increases arbitrarily with the number of nodes in the network.

Bibliography

[ACS89] Alok Aggarwal, Ashok K. Chandra, and Mark Snir. Communication complexity of PRAMs. Technical Report RC 14998 (64644), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1989.

[AGGM90] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan. A parallel FFT on an MIMD machine. *Parallel Computing*, 15:61–74, 1990.

[Bai90] David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4:23–35, 1990.

[BCJ90] Edward C. Bronson, Thomas L. Casavant, and L. H. Jamieson. Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):195–205, 1990.

[BKH89] S. Bershader, T. Kraay, and J. Holland. The giant-Fourier-transform. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications: Volume I*, pages 387–389, 1989.

[Cve87] Z. Cvetanovic. Performance analysis of the FFT algorithm on a shared-memory parallel architecture. *IBM Journal of Research and Development*, 31(4):435–451, 1987.

[DT89] Laurent Desbat and Denis Trystram. Implementing the discrete Fourier transform on a hypercube vector-parallel computer. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications: Volume I*, pages 407–410, 1989.

[GK93] Anshul Gupta and Vipin Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993. A detailed version available as Technical Report TR 90-53, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[JKFM89] S. L. Johnsson, R. Krawitz, R. Frye, and D. McDonald. A radix-2 FFT on the connection machine. Technical report, Thinking Machines Corporation, Cambridge, MA, 1989.

[KA88] Ray A. Kamin and George B. Adams. Fast Fourier transform algorithm design and tradeoffs. Technical Report RIACS TR 88.18, NASA Ames Research Center, Moffet Field, CA, 1988.

[Loa92] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, PA, 1992.

[NS87] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared memory architectures. *IEEE Transactions on Computers*, C-36(5):581–591, 1987.

[Nus82] H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer-Verlag, New York, NY, 1982.

[Rao92] V. N. Rao. Personal Communication. University of Central Florida, Orlando, FL, 1992.

[RB76] C. M. Rader and N. M. Brenner. A new principle for Fast fourier transform. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 24:264–265, 1976.

[Swa87] P. N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[Tho83] Clark D. Thompson. Fourier transforms in VLSI. *IBM Journal of Research and Development*, C-32(11):1047–1057, 1983.

[Win77] S. Winograd. A new method for computing DFT. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 366–368, 1977.