```
Mergesort (list):
  if length(list) = 1 then
    return list
  else
    part1 ← Mergesort (first half of list)
    part2 ← Mergesort (remainder of list)
    return Merge (part1, part2)
  endif
```

If you need a more detailed explanation of mergesort, consult Cormen et al. [18], Baase and Van Gelder [5], or another introductory analysis of algorithms textbook.

a. Design a parallel version of mergesort for a multicomputer. Make these three assumptions. At the beginning of the algorithm's execution all unsorted values are in the memory of one processor. At the end of the algorithm's execution the sorted list is in the memory of one processor. The number of processors p is a power of 2.

b. What is the complexity of this algorithm?

c. What is the isoefficiency of your parallel mergesort algorithm?

d. Write a program implementing your parallel algorithm. Benchmark the program for various combinations of p and n.

14.10 Suppose the values of n keys are uniformly distributed in the interval $[0, k)$. *Bucket sort* divides the interval $[0, k)$ into j equal-sized subintervals called *buckets*. Each key is placed in one of the buckets, based upon its value. After all the keys have been placed in buckets, the keys in each bucket are sorted. Once the keys in each bucket are sorted, the sequence is sorted.

a. Design a parallel version of bucket sort. Initially the n keys are distributed among the p processes. The interval $[0, k)$ should be divided into p buckets. In step 1, every process divides its n/p keys into p groups, one per bucket. In step 2, each process assumes responsibility for one of the buckets. An all-to-all communication routes groups of keys to the correct processes. In step 3, each process sorts the keys in its bucket.

b. What is the complexity of this algorithm?

c. What is the isoefficiency of your parallel bucket sort algorithm?

d. Write a program implementing parallel bucket sort. Benchmark the program for various combinations of p and n.

14.11 Write a parallel program to find the kth largest element in an unsorted list of n elements initially distributed among p processors.

14.12 A file contains n signed integers, each four bytes long. Write a parallel program to determine which integer value occurs most frequently in the file.

CHAPTER

15

The Fast Fourier Transform

The meeting of two personalities is like the contact of two chemical substances; if there is any reaction, both are transformed.
Carl Gustav Jung, *Modern Man in Search of a Soul*

15.1 INTRODUCTION

The discrete Fourier transform has many applications in science and engineering. For example, it is often used in digital signal processing applications such as voice recognition and image processing. A straightforward implementation of the discrete Fourier transform has time complexity $\Theta(n^2)$. The fast Fourier transform is a $\Theta(n \log n)$ algorithm to perform the discrete Fourier transform, and it can be parallelized easily.

In this chapter we illuminate how the discrete Fourier transform works by using an example from speech recognition. We formally present the discrete Fourier transform and the inverse discrete Fourier transform. We move on to present the fast Fourier transform algorithm and describe how to implement it on a multicomputer.

15.2 FOURIER ANALYSIS

Fourier analysis studies the representation of continuous functions by a potentially infinite series of sinusoidal (sine and cosine) functions. We can view the discrete Fourier transform as a function that maps a sequence over time $\{f(k)\}$ to another sequence over frequency $\{F(j)\}$. The sequence $\{f(k)\}$ represents a sampling of a signal's distribution as a function of time. The sequence

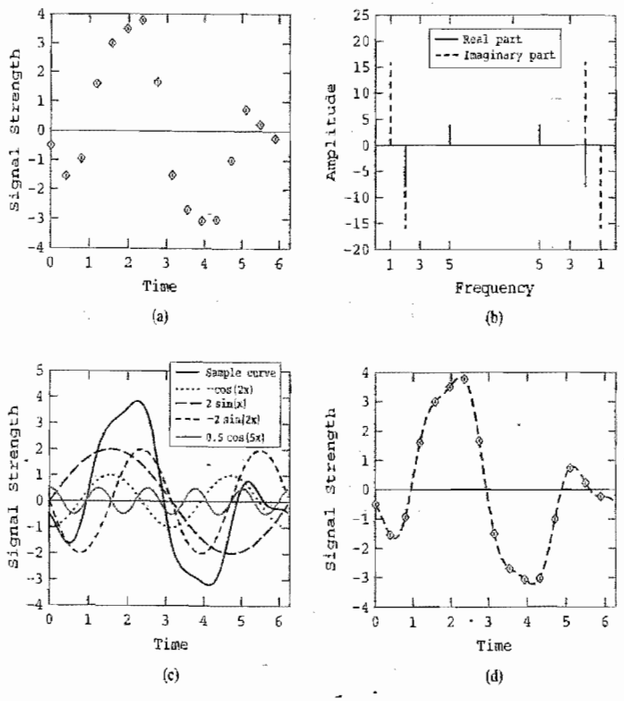

Figure 15.1 Example of the discrete Fourier transform. (a) A set of 16 data points representing samples of signal strength in the time interval 0 to (but not through) 2π. (b) The discrete Fourier transform yields the amplitudes and frequencies of the constituent sine and cosine functions. (c) A plot of the four constituent functions and their sum, a continuous function. (d) A plot of the continuous function and the original 16 samples.

$\{F(j)\}$ represents a distribution of Fourier coefficients as a function of frequency. We can use $\{F(j)\}$ to compute the sinusoidal components of the sampled signal.

Figure 15.1 illustrates this process. We begin, in Figure 15.1a, with a plot of $\{f(k)\}$, 16 samples of signal strength between time 0 and time 2π. Figure 15.1b is the plot of $\{F(j)\}$, a sequence of 16 complex numbers representing the frequency distribution. From the nonzero elements of $\{F(j)\}$ we can determine the frequency of the terms generating the signal, where **frequency** means the number of complete cycles the wave completes between time 0 and time 2π. Nonzero real components correspond to cosine functions; nonzero imaginary components correspond to sine functions. From Figure 15.1b we see that there are nonzero real components with frequency 2 and 5 and nonzero imaginary components with

frequency 1 and 2. Hence the function generating the signal is of the form

$$s_1 \sin x + c_2 \cos(2x) + s_2 \sin(2x) + c_5 \cos(5x)$$

For each frequency, we divide the amplitude shown in the left half of Figure 15.1b by 8 (half of 16, the number of sample points) to determine the coefficients of the various sinusoidal components. The frequency 1 component is $16i$. Dividing 16 by 8 yields a coefficient of 2 for the function $\sin x$. The frequency 2 component is $-8 - 16i$. Dividing -8 by 8 yields a coefficient of -1 for the function $\cos(2x)$, just as the coefficient for the function $\sin(2x)$ is -2. We use the same method to calculate that 0.5 is the coefficient of the function $\cos(5x)$. The four terms generating the signal are

$$2 \sin x - \cos(2x) - 2 \sin(2x) + 0.5 \cos(5x)$$

In Figure 15.1c we plot the four sinusoidal components and their sum, a continuous function, and in Figure 15.1d we plot the continuous function against the sampled data points.

Let's look at how Fourier analysis is used in speech recognition. Most speech analysis has been done by studying the spectral parameters of the speech signal.

> Decomposing complex speech signals into periodically recurrent sinusoidal components is the central activity of signal processing work, and is justified by (1) sinusoids being "natural signals" of linear physical (electronic) systems; (2) resonances being prominent cues to articulation configurations; (3) voice sounds being composed out of harmonics of the voice fundamental frequency; and (4) the ear appearing to do some form of spectral analysis. Also, sinusoids (and some other exponential signals) can be added ("superimposed") in linear systems without interfering with each other; thus the sinusoidal parts that we decompose the signal input into for frequency analysis act as independent, "orthogonal signals" [67].

The discrete Fourier transform can be used to convert digitized samples of human speech into two-dimensional plots (see Figure 15.2). The graph shows detected frequencies as a function of time. Each narrow vertical strip shows the amplitudes of the detected frequencies as shades of gray. As the person talks, the speech signal changes, and so do the frequencies that make up the signal. Plots such as this can be used as inputs to speech recognition systems, which try to identify spoken phonemes through pattern recognition.

15.3 THE DISCRETE FOURIER TRANSFORM

Given an n element vector x, the **discrete Fourier transform (DFT)** is the matrix-vector product $F_n x$, where $f_{i,j} = \omega_n^{ij}$ for $0 \leq i, j < n$ and ω_n is the primitive nth root of unity. (For a review of complex numbers, refer to Appendix D.)

For example, to compute the discrete Fourier transform of the vector (2, 3), we need to know ω_2, the primitive square root of unity. The primitive square root

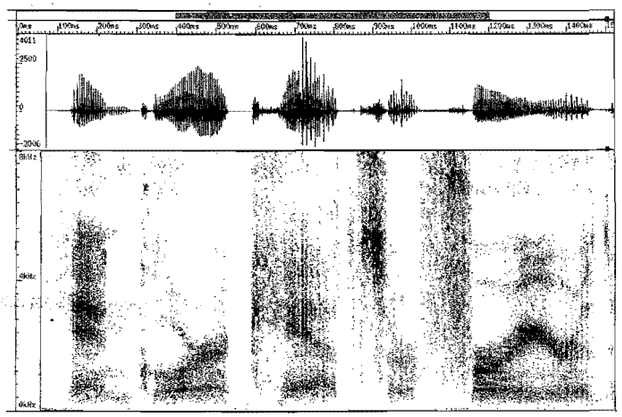

Figure 15.2 Discrete Fourier transform of the waveform corresponding to "Angora cats are furrier...." The upper portion of the chart plots the strength of the input signal as a function of time. The lower portion plots frequency and amplitude as a function of time. Each narrow vertical strip represents the discrete Fourier transform of the waveform using a moving 10 ms window within a 3 ms increment. The darker the plot at some vertical position, the higher the amplitude at that frequency. (Figure courtesy Ron Cole and Yeshwant Muthusamy of the Oregon Graduate Institute.)

of unity is -1. The DFT of (2, 3), then, is

$$\begin{pmatrix} \omega_2^{0\times 0} & \omega_2^{0\times 1} \\ \omega_2^{1\times 0} & \omega_2^{1\times 1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 5 \\ -1 \end{pmatrix}$$

Now let's compute the DFT of the vector (1, 2, 4, 3). We will need to use the primitive fourth root of unity, which is i.

$$\begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ \omega_4^0 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 4 \\ 3 \end{pmatrix} = \begin{pmatrix} 10 \\ -3-i \\ 0 \\ -3+i \end{pmatrix}$$

Let's put the DFT to use by returning to the example presented in the previous section. We have a vector of 16 complex numbers representing signal strength in the time interval 0 to 2π. To simplify the presentation we show each number to only three digits of accuracy:

$$(-0.500, -1.55, -0.939, 1.60, 3.00, 3.51, 3.77, 1.66,$$
$$-1.50, -2.70, -3.06, -3.02, -1.00, 0.736, 0.232, -0.250)$$

The DFT of this vector is

$$(0, 16i, -8-16i, 0, 0, 4, 0, 0, 0, 0, 0, 4, 0, 0, -8+16i, -16i)$$

To determine the coefficients of the sine and cosine functions making up this signal, we examine the nonzero elements in the first half of the transformed sequence. (The terms at positions 9 through 15 are a reflection of the terms in positions 1 through 7, with the signs of the imaginary parts reversed.) If we begin counting at 0, the real portion of term k is 8 times the coefficient of the function $\cos(kx)$, and the imaginary portion of term k is 8 times the coefficient of the function $\sin(kx)$. (Eight is half the number of sample points.) Thus the combination of sine and cosine functions making up the curve is

$$2\sin(x) - \cos(2x) - 2\sin(2x) + 0.5\cos(5x)$$

15.3.1 Inverse Discrete Fourier Transform

Given an n element vector x, the **inverse discrete Fourier transform (inverse DFT)** is $1/n$th the matrix-vector product $F_n^{-1}x$, where $f^{-1}(ij) = \omega_n^{-ij}$ for $0 \le i, j < n$ and ω_n is the primitive n root of unity.

For example, the inverse DFT of the vector $(10, -3-i, 0, -3+i)$ is

$$\frac{1}{4}\begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^{-1} & \omega_4^{-2} & \omega_4^{-3} \\ \omega_4^0 & \omega_4^{-2} & \omega_4^{-4} & \omega_4^{-6} \\ \omega_4^0 & \omega_4^{-3} & \omega_4^{-6} & \omega_4^{-9} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 10 \\ -3-i \\ 0 \\ -3+i \end{pmatrix}$$

$$= \frac{1}{4}\begin{pmatrix} 4 \\ 8 \\ 16 \\ 12 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \\ 3 \end{pmatrix}$$

15.3.2 Sample Application: Polynomial Multiplication

We can use the DFT and inverse DFT to multiply polynomials. First, we need to understand what the DFT and inverse DFT do. The DFT evaluates a polynomial at the n complex nth roots of unity. Let's see why this is true. If $f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$ is a polynomial of degree $n-1$, and ω is the primitive nth root of unity, then

$$\begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ \cdots \\ f(\omega^{n-1}) \end{pmatrix} = F \begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{pmatrix}$$

because $f(\omega^i) = a_0 + a_1\omega^i + a_2\omega^{2i} + \cdots + a_{n-1}\omega^{(n-1)i}$ for $0 \le i < n$.

The inverse DFT takes the values of a polynomial at the n complex nth roots of unity and produces the polynomial's coefficients.

Now, suppose we want to multiply two polynomials

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \quad \text{and} \quad q(x) = \sum_{i=0}^{n-1} b_i x^i$$

The product of these two polynomials of degree $n-1$ is the $(2n-2)$ degree polynomial

$$p(x)q(x) = \sum_{i=0}^{2n-2} \sum_{j=0}^{i} a_j b_{i-j} x^i$$

We can compute the coefficients of the resulting polynomial $p(x)q(x)$ by convoluting the coefficient vectors of the original polynomials.

For example, to multiply the two polynomials

$$p(x) = 2x^3 - 4x^2 + 5x - 1$$
$$q(x) = x^3 + 2x^2 + 3x + 2$$

yielding

$$r(x) = a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

we convolute the coefficient vectors:

$$a_6 = 2 \times 1 = 2$$
$$a_5 = 2 \times 2 + (-4) \times 1 = 0$$
$$a_4 = 2 \times 3 + (-4) \times 2 + 5 \times 1 = 3$$
$$a_3 = 2 \times 2 + (-4) \times 3 + 5 \times 2 + (-1) \times 1 = 1$$
$$a_2 = (-4) \times 2 + 5 \times 3 + (-1) \times 2 = 5$$
$$a_1 = 5 \times 2 + (-1) \times 3 = 7$$
$$a_0 = (-1) \times 2 = -2$$

resulting in

$$r(x) = 2x^6 + 3x^4 + x^3 + 5x^2 + 7x - 2$$

Another way to multiply two polynomials of degree $n-1$ is to evaluate them at the n complex nth roots of unity, perform an element-wise multiplication of the polynomials' values at these points, and then interpolate the results to produce the coefficients of the product polynomial. Let's apply this method to the previous example.

First we perform the DFT on the coefficients of $p(x)$. We list the coefficients in order from low to high. Since the polynomial has degree 3, the last four coefficients

are 0. To simplify the figure, we only show two digits past the decimal point.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ -4 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1.12 + .95i \\ 3 + 3i \\ -3.12 + 8.95i \\ -12 \\ -3.12 - 8.95i \\ 3 - 3i \\ 1.12 - .95i \end{pmatrix}$$

Next we perform the DFT on the coefficients of $q(x)$. Again, we are only showing two digits beyond the decimal point.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & \omega^1 & \omega^4 & 1 & \omega^4 & \omega^1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 2 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 8 \\ 3.41 + 4.83i \\ 2i \\ .59 + .83i \\ 0 \\ .59 - .83i \\ -2i \\ 3.41 - 4.83i \end{pmatrix}$$

Now we perform an element-wise multiplication of the two polynomials at these eight points.

$$\begin{pmatrix} 2 \\ 1.12 + .95i \\ 3 + 3i \\ -3.12 + 8.95i \\ -12 \\ -3.12 - 8.95i \\ 3 - 3i \\ 1.12 - .95i \end{pmatrix} \begin{pmatrix} 8 \\ 3.41 + 4.83i \\ 2i \\ .59 + .83i \\ 0 \\ .59 - .83i \\ -2i \\ 3.41 - 4.83i \end{pmatrix} = \begin{pmatrix} 16 \\ -.76 + 8.66i \\ -6 + 6i \\ -9.25 + 2.66i \\ 0 \\ -9.25 - 2.66i \\ -6 - 6i \\ -.76 - 8.66i \end{pmatrix}$$

In the final step we perform the inverse DFT on the product vector. Note that we have replaced the negative powers of ω with equivalent values expressed as positive powers. For example, when ω is the primitive 8th root of unity, $\omega^{-1} = \omega^7$ and $\omega^{-2} = \omega^6$. Here is the inverse DFT:

$$\frac{1}{8} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \end{pmatrix} \begin{pmatrix} 16 \\ -.76 + 8.66i \\ -6 + 6i \\ -9.25 + 2.66i \\ 0 \\ -9.25 - 2.66i \\ -6 - 6i \\ -.76 - 8.66i \end{pmatrix} = \begin{pmatrix} -2 \\ 7 \\ 5 \\ 1 \\ 3 \\ 0 \\ 2 \\ 0 \end{pmatrix}$$

The vector produced by the inverse DFT contains the coeffients of the product polynomial in order from low to high. In other words,

$$r(x) = 2x^6 + 3x^4 + x^3 + 5x^2 + 7x - 2$$

15.4 THE FAST FOURIER TRANSFORM

At the end of Section 15.3 we demonstrated how we can use the DFT and inverse DFT to multiply two polynomials. Why would we use this complicated algorithm to perform convolutions or multiply polynomials, when these can be done directly in time $\Theta(n^2)$? The reason is that we do not have to perform the DFT and inverse DFT using matrix-vector multiplication. An algorithm with complexity $\Theta(n \log n)$ exists, and (luckily for us) it is amenable to parallelization. The improved algorithm is called the **fast Fourier transform (FFT).**

The FFT uses a divide-and-conquer strategy to evaluate a polynomial of degree n at the n complex nth roots of unity. To evaluate $f(x)$, a polynomial of degree n where n is a power of 2, the algorithm defines two new polynomials of degree $n/2$. Function $f^{[0]}(x)$ contains the elements of $f(x)$ associated with the even powers of x, while function $f^{[1]}(x)$ contains the elements associated with the odd powers of x:

$$f^{[0]} = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$
$$f^{[1]} = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

Note that $f(x) = f^{[0]}(x^2) + x f^{[1]}(x^2)$, so the problem of evaluating $f(x)$ at the points $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to evaluating $f^{[0]}$ and $f^{[1]}$ at $(\omega_n^0)^2$, $(\omega_n^1)^2, \ldots, (\omega_n^{n/2-1})^2$, and then computing $f(x) = f^{[0]}(x^2) + x f^{[1]}(x^2)$.

Halving Lemma If n is an even positive number, then the squares of the n complex nth roots of units are identical to the $n/2$ complex $(n/2)$th roots of unity.

Proof See Appendix D.

By the **halving lemma,** we know that the set of points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$ consists of only $n/2$ unique values. In other words, to evaluate the polynomial $f(x)$ at the n complex nth roots of unity we need only evaluate the polynomials $f^{[0]}(x)$ and $f^{[1]}(x)$ at the $n/2$ complex $(n/2)$th roots of unity. Hence our divide-and-conquer strategy will save us computations.

The most natural way to express the FFT algorithm resulting from the divide-and-conquer strategy is to use recursion. Pseudocode for a recursive implementation of FFT appears in Figure 15.3. The time complexity of this algorithm is easy to determine. Let $T(n)$ denote the time needed to perform the FFT on a polynomial of degree n, where n is a power of 2.

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \log n)$$

Recursive_FFT (a, n)

Parameter	n	Number of elements in a
	$a[0..(n-1)]$	Coefficients
Local	ω_n	Primitive nth root of unity
	ω	Evaluate polynomial at this point
	$a^{[0]}$	Even-numbered coefficients
	$a^{[1]}$	Odd-numbered coefficients
	y	Result of transform
	$y^{[0]}$	Result of FFT of $a^{[0]}$
	$y^{[1]}$	Result of FFT of $a^{[1]}$

```
if n = 1 then return a
else
  ω_n ← e^(2πi/n)
  ω ← 1
  a^[0] ← (a[0], a[2], ..., a[n − 2])
  a^[1] ← (a[1], a[3], ..., a[n − 1])
  y^[0] ← Recursive_FFT (a^[0], n/2)
  y^[1] ← Recursive_FFT (a^[1], n/2)
  for k ← 0 to n/2 − 1 do
    y[k] ← y^[0][k] + ω × y^[1][k]
    y[k + n/2] ← y^[0][k] − ω × y^[1][k]
    ω ← ω × ω_n
  endfor
  return y
endif
```

Figure 15.3 Recursive sequential implementation of the fast Fourier transform algorithm (adapted from Cormen et al. [18]).

While the recursive formulation of the FFT algorithm is (relatively) easy to understand, we have two reasons for developing an iterative FFT algorithm. First, a well-written iterative version of the FFT algorithm can perform fewer index computations and eliminate the second evaluation of $\omega_n^k y^{[1]}[k]$ every iteration of the **for** loop. Second, it is easier to derive a parallel FFT algorithm when the sequential algorithm is in iterative form.

Figure 15.4 illustrates the derivation of an iterative algorithm from the recursive algorithm. In Figure 15.4a we see how the recursive algorithm transforms a vector of four elements. Each rounded rectangle represents a call to function `fft`. The vector to be transformed is inside the parentheses. The function keeps dividing the vector in half and calling itself recursively on each half until the vector size is 1. The DFT of a single value is that value. (Remember the FFT is simply a fast way of performing the DFT.) The heavy curved arrows show the values returned from each invocation of the function. The function combines the values received in two vectors of length i and returns a vector of length $2i$. Performing the FFT on input vector $(1, 2, 4, 3)$ produces the result vector $(10, -3 - i, 0, -3 + i)$.

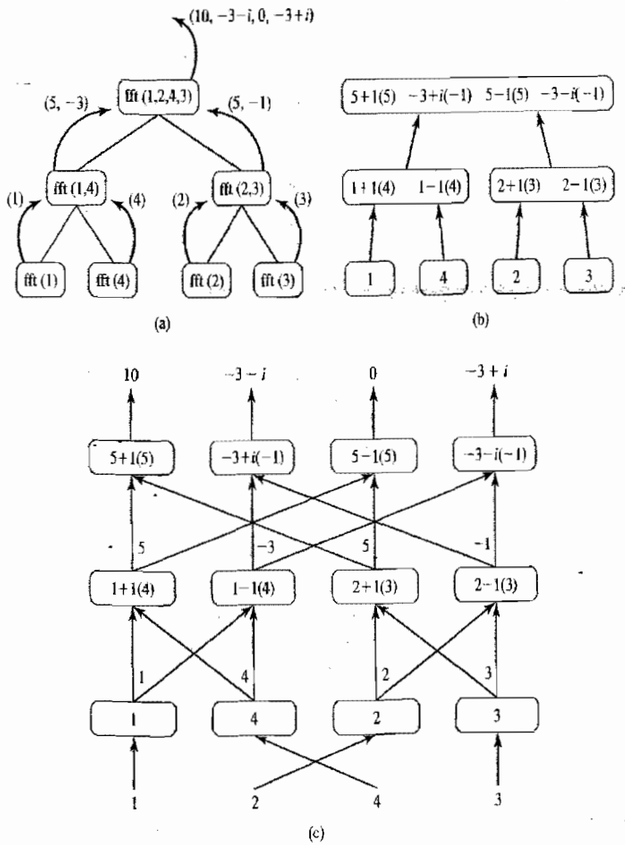

Figure 15.4 Evolution of the iterative algorithm from the recursive algorithm. (a) Recursive implementation of FFT. (b) Determining which computations are performed for each function invocation. (c) Tracking the flow of data values.

In Figure 15.4b we look inside the functions and determine exactly which operations are performed for each invocation. The expressions of form $a + b(c)$ and $a - b(c)$ correspond to the pseudocode statements

$$y[k] \leftarrow y^{[0]}[k] + \omega \times y^{[1]}[k]$$

$$y[k + n/2] \leftarrow y^{[0]}[k] - \omega \times y^{[1]}[k]$$

Figure 15.4c tracks the movement of the data values. At the beginning of the algorithm the vector elements are permuted. The element at index i of the input vector is moved to index $rev(i)$, where $rev(i)$ represents the bits of i in reverse order. Value 2, initially at index 01, is moved to index 10. Value 4, initially at index 10, is moved to index 01. Values 0 (at index 00) and 3 (at index 11) stay put. In the first stage the algorithm is finding the DFT of individual values, and it simply passes the values along. In each of the remaining stages the computation of a new value depends upon two values from the previous stage. The data flow arrows form **butterfly patterns.**

We can derive the iterative algorithm directly from Figure 15.4. After an initial permutation step, the algorithm will iterate $\log n$ times. Each iteration corresponds to a horizontal layer in Figure 15.4c. Within an iteration the algorithm updates values for each of the n indices. The algorithm, illustrated in Figure 15.5, has the same time complexity as the recursive algorithm: $\Theta(n \log n)$. The use of temporary variable t cuts the number of complex number multiplications nearly in half.

Iterative_FFT (a, n):

Parameter	n	Number of elements in a
	$a[0 \ldots (n-1)]$	Coefficients
Local	ω_d	Primitive dth root of unity
	ω	Evaluate polynomial at this point
	y	Result of transform

```
y ← Bit_Reverse_Permutation(a)
for j ← 1 to log n
  d ← 2^j
  ω_d ← e^(2πi/d)
  ω ← 1
  for k ← 0 to d/2 − 1
    for m ← k to n − 1 step d
      t ← ω × y[m + d/2]
      x ← y[k]
      y[k] ← x + t
      y[k + d/2] ← x − t
    endfor
  endfor
  ω ← ω × ω_d
endfor
return y
```

Figure 15.5 Iterative, sequential implementation of the fast Fourier transform algorithm (adapted from Cormen et al. [18]).

15.5 PARALLEL PROGRAM DESIGN

15.5.1 Partitioning and Communication

The efficient iterative algorithm is our starting point for designing a parallel FFT function suitable for implementation on a multicomputer. We'll use a domain

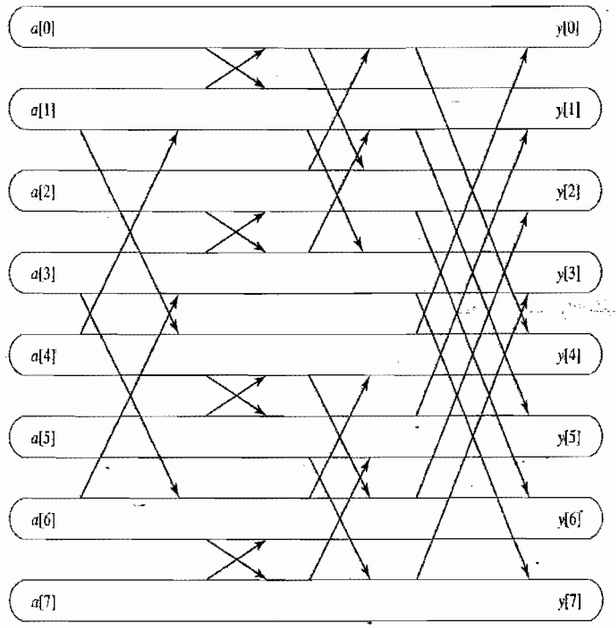

Figure 15.6 Task/channel graph for the FFT algorithm when $n = 8$. Tasks are represented by long, rounded rectangles so that the channels needed at each stage of the algorithm can be distinguished.

decomposition, associating a primitive task with each element of the input vector a and the corresponding element of the output vector y (Figure 15.6).

The first step of the algorithm is to perform the permutation of vector a. Each element $a[i]$ is copied to $y[j]$, where j is the index found by reversing the bits of i. For example, when $n = 8$, we have

$$\begin{aligned} 001 &\rightarrow 100 \\ 010 &\rightarrow 010 \\ 011 &\rightarrow 110 \\ &\cdots \\ 110 &\rightarrow 011 \\ 111 &\rightarrow 111 \end{aligned}$$

We draw channels for this initial communication.

The main loop of the function has $\log n$ iterations. During each iteration, each task computes its new value of $y[k]$ from the previous values of $y[k]$ and either $y[k + m/2]$ or $y[k - m/2]$. The task/channel graph illustrates the butterfly communication pattern.

15.5.2 Agglomeration and Mapping

Agglomerating primitive tasks associated with contiguous elements of the vector eliminates some of the communication steps. For example, if $n = 16$ and $p = 4$, process 0 has coefficients with subscripts 0, 1, 2, and 3; process 1 has coefficients with subscripts 4, 5, 6, and 7; and so on.

We can draw another task/channel diagram that indicates the communication pattern between the processes. In this new diagram (Figure 15.7), each agglomerated task (process) is represented by a gray rectangle. Every process controls two arrays of complex values. The first array, a, contains a contiguous group of input coefficients. The second array, y, holds intermediate values. At the end of the computation, array y contains a contiguous group of transformed values.

The parallel algorithm has three phases. In the first phase the processes permute the a's. This is an example of an all-to-all communication. In the second phase the processes perform the first $\log n - \log p$ iterations of the FFT by performing the required multiplications, additions, and subtractions on complex numbers. No message passing is required. In the third phase the processes perform the final $\log p$ iterations of the FFT by swapping y's and performing the requisite multiplications, additions, and subtractions. Think of the processes as being organized as a logical hypercube. During each of the final $\log p$ iterations, pairs of processes swap values across a different dimension of the hypercube.

15.5.3 Isoefficiency Analysis

Each process performs an equal share of the computations. Since the computational complexity of the sequential algorithm is $\Theta(n \log n)$, the computational complexity of the parallel algorithm is $\Theta(n \log n/p)$.

Each process controls at most $\lceil n/p \rceil$ elements of a. We assume the processes are organized as a logical hypercube. The all-to-all communication step is implemented as a series of swaps across each hypercube dimension; it has time complexity $\Theta[(n/p) \log p]$. There are $\log p$ iterations in which each process swaps about n/p values with a partner process along one of the hypercube dimensions. The total time complexity of these swaps is $\Theta[(n/p) \log p]$. With these assumptions, the overall communication complexity of the algorithm is $\Theta[(n/p) \log p]$.

Let's determine the isoefficiency of the parallel program. The sequential algorithm has time complexity $\Theta(n \log n)$. The parallel overhead is p times the communication complexity. Hence the isoefficiency function is

$$n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

The scalability of the FFT algorithm is similar to the scalability of the hyperquicksort and PSRS algorithms.

many scientific and
m is interesting for
screte Fourier trans-
has time complexity
allelization.
sform algorithm and
ontinuous function,
scribed the inverse
e Fourier transform
ltiply polynomials.
t Fourier transform
nt the fast Fourier

discrete Fourier
form

m in computer sci-
l. [18].
orm algorithm, see

he form $re^{i\theta}$.

principal nth root

15.3 For each of the following vectors, show the result of applying the DFT to it.

a. (7, 11)

b. (13, 17, 19, 23)

c. (2, 1, 3, 7, 5, 4, 0, 6)

15.4 For each of the following vectors, show the result of applying the inverse DFT to it.

a. $(3, -2)$

b. $(10, -2+2i, -2, -2-2i)$

c. $(14, -3-4i, 1-i, -1+3i, 0, -1-3i; 1+i, -3+4i)$

15.5 Implement a parallel FFT program based on the design developed in this chapter. Benchmark your program for various values of n and p.

15.6 Implement a serial program implementing the inverse FFT algorithm.

15.7 Implement a parallel program implementing the inverse FFT algorithm. Benchmark your program for various problem sizes on various numbers of processors.

15.8 Excluding the initial all-to-all communication, the body of the fast Fourier transform algorithm exhibits a butterfly communication pattern. Name a parallel algorithm described in an earlier chapter that also has a butterfly communication pattern.

15.9 The scalability of the FFT algorithm is similar to the scalability of the hyperquicksort algorithm. Explain the similarities between the two algorithms.

CHAPTER

16

Combinatorial Search

Attempt the end, and never stand to doubt;
Nothing's so hard but search will find it out.
Robert Herrick, "Seek and Find," ***Hesperides***

16.1 INTRODUCTION

Combinatorial algorithms perform computations on discrete, finite mathematical structures [97]. **Combinatorial search** is the process of finding "one or more optimal or suboptimal solutions in a defined problem space" [109] and has been used for such diverse problems as:

- laying out circuits in VLSI to minimize the area dedicated to wires
- planning the motion of robot arms to minimize total distance traveled
- assigning crews to airline flights
- proving theorems
- playing games

There are two kinds of combinatorial search problems. An algorithm to solve a **decision problem** attempts to find a solution that satisfies all the constraints. The answer to a decision problem is either yes, meaning a solution exists, or no, meaning a solution does not exist. Here is an example of a decision problem: "Is there a way to route the robot arm so that it visits every drill site and moves no more than 15 meters?" An algorithm that solves an **optimization problem** must find a solution that minimizes (or maximizes) the value of an objective function. Here is an example of an optimization problem: "Find the shortest route for the robot arm that visits every drill site."

This chapter discusses four kinds of combinatorial search algorithms used to solve decision and optimization problems. These algorithms are divide and