the form given by Equation 6.8, but different constants appear for different methods. From a practical perspective, methods have widely varying execution times precisely because of these constants.

Most methods, including the iterative solver, can be generalized to solve block-tridiagonal and narrowly banded systems. The generalization requires replacing scalar arithmetic with matrix arithmetic. Although programs for block-tridiagonal systems are more complicated, they are more efficient, because the increased granularity of these problems leads to an improved ratio of communication over arithmetic time.

Exercises

Exercise 27 *Using Equations 6.6 and 6.7, develop the multicomputer programs that solve $L\vec{y} = \vec{b}$ and $U\vec{x} = \vec{y}$.*

Exercise 28 *Examine the impact on full recursive doubling of the non-associativity of floating-point addition.*

Exercise 29 *Implement the concurrent iterative tridiagonal solver of Section 6.4.*

7
The Fast Fourier Transform

7.1 Fourier Analysis

7.1.1 Fourier Series

We consider the space L_2 of complex-valued functions $f(x)$ on the closed interval $[0, 2\pi]$ that are bounded, continuous, and periodic with period 2π. The inner product of two functions $f(x), g(x) \in L_2$ is defined by

$$(f, g) = \frac{1}{2\pi} \int_0^{2\pi} \overline{f(x)} g(x) dx,$$

where $\overline{f(x)}$ denotes the complex conjugate of $f(x)$. Throughout this chapter, we shall use standard mathematical notation that $i = \sqrt{-1}$.

With respect to this inner product, the set of functions $\{e^{ikx} : k \in \mathbb{Z}\}$ forms an orthonormal basis of L_2. It is, therefore, possible to expand every function $f(x) \in L_2$ into a unique linear combination of the basis functions e^{ikx}, or

$$f(x) = \sum_{k=-\infty}^{+\infty} \hat{f}_k e^{ikx}. \tag{7.1}$$

This linear combination is known as *the Fourier series of* $f(x)$, and the coefficients $\hat{f}_k$ are called the *Fourier-series coefficients*. The orthonormality of the basis functions implies that

$$\hat{f}_k = \frac{(e^{ikx}, f(x))}{(e^{ikx}, e^{ikx})} = \frac{1}{2\pi} \int_0^{2\pi} e^{-ikx} f(x) dx. \tag{7.2}$$

The Fourier-series expansion can be interpreted as a transformation of the function space L_2 into the infinite-dimensional complex vector space ℓ_2, which consists of vectors

$$\hat{\vec{f}} = [\hat{f}_k]_{k=-\infty}^{+\infty}.$$

In ℓ_2, we define the inner product of two vectors $\hat{\vec{f}}$ and $\hat{\vec{g}}$ by:

$$(\hat{\vec{f}}, \hat{\vec{g}}) = \sum_{k=-\infty}^{+\infty} \overline{\hat{f}_k} \hat{g}_k.$$

Using a norm consistent with the inner product defined in each space,

$$\begin{aligned} \| f(x) \|^2 = (f, f) &= \frac{1}{2\pi} \int_0^{2\pi} |f(x)|^2 dx \\ \| \hat{\vec{f}} \|^2 = (\hat{\vec{f}}, \hat{\vec{f}}) &= \sum_{k=-\infty}^{+\infty} |\hat{f}_k|^2, \end{aligned}$$

it is easily verified that

$$\| f(x) \|^2 = \| \hat{\vec{f}} \|^2 . \tag{7.3}$$

In other words, the Fourier-series expansion is a norm-preserving transformation from the space L_2 to the space ℓ_2.

7.1.2 The Discrete Fourier Transform

In numerical computations, functions are approximated by a set of function values on some grid. In certain cases, one can use the *discrete Fourier transform* to compute numerical approximations of Fourier-series coefficients. The precise connection between Fourier series and the discrete Fourier transform will be discussed in Section 7.1.3.

The representation of a 2π-periodic function on an equidistant grid with grid spacing $h = 2\pi/N$ is the set of values $f_j = f(jh)$, where $0 \leq j \leq N$. Because of periodicity, we have that $f_N = f_0$. This leads us to study the complex N-dimensional vector space $\mathbb{C}^N$ of vectors $\vec{f} = [f_j]_{j=0}^{N-1}$.

The inner product of two vectors $\vec{f}, \vec{g} \in \mathbb{C}^N$ is defined by

$$(\vec{f}, \vec{g}) = \sum_{j=0}^{N-1} \overline{f_j} g_j,$$

and the norm implied by this inner product is given by

$$\| \vec{f} \|^2 = (\vec{f}, \vec{f}) = \sum_{j=0}^{N-1} \overline{f_j} f_j = \sum_{j=0}^{N-1} |f_j|^2.$$

Lemma 17 *The set of vectors* $\{\vec{e}_n = [e^{ijnh}]_{j=0}^{N-1} : 0 \leq n < N\}$ *is an orthogonal basis for the space* $\mathbb{C}^N$.

The inner product of any two of the proposed basis vectors is given by:

$$(\vec{e}_n, \vec{e}_m) = \sum_{j=0}^{N-1} \overline{e^{ijnh}} e^{ijmh} = \sum_{j=0}^{N-1} e^{-ijnh} e^{ijmh} = \sum_{j=0}^{N-1} e^{ij(m-n)h}.$$

With $r = e^{i(m-n)h}$ we have that

$$(\vec{e}_n, \vec{e}_m) = \sum_{j=0}^{N-1} r^j = \begin{cases} N & \text{if } r = 1 \\ \frac{r^N - 1}{r - 1} & \text{if } r \neq 1. \end{cases}$$

The case $r = 1$ arises only if $m = n \bmod N$. Otherwise, $r \neq 1$ and

$$r^N = e^{i(m-n)hN} = e^{i2(m-n)\pi} = 1.$$

It follows that the vectors $\vec{e}_n$ are mutually orthogonal and that

$$\forall m, n \in 0..N-1 : \ (\vec{e}_n, \vec{e}_m) = \begin{cases} N & \text{if } n = m \\ 0 & \text{if } n \neq m. \end{cases}$$

□

Corollary 1 *Every vector* $\vec{f} \in \mathbb{C}^N$ *has a unique representation of the form*

$$\vec{f} = [f_j]_{j=0}^{N-1} = \sum_{n=0}^{N-1} \hat{f}_n \vec{e}_n,$$

where the discrete Fourier coefficients are given by

$$\hat{\vec{f}} = [\hat{f}_n]_{n=0}^{N-1} = \left[\frac{(\vec{e}_n, \vec{f})}{(\vec{e}_n, \vec{e}_n)} \right]_{n=0}^{N-1}.$$

(Without proof.) □

The vector $\hat{\vec{f}}$ is called the discrete Fourier transform of $\vec{f}$. The inverse discrete Fourier transform maps $\hat{\vec{f}}$ into $\vec{f}$. The equations of Corollary 1 are the discrete analogs of Equations 7.1 and 7.2. When written out component by component, they become:

$$\forall j \in 0..N-1 : \quad f_j = \sum_{n=0}^{N-1} \hat{f}_n e^{ijnh} \tag{7.4}$$

$$\forall n \in 0..N-1 : \quad \hat{f}_n = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-ijnh}. \tag{7.5}$$

The analog of Equation 7.3, the norm-preservation property, is easily established. We have that:

$$\| \hat{\vec{f}} \| = \frac{1}{\sqrt{N}} \| \vec{f} \| . \tag{7.6}$$

The norms of a vector and its discrete Fourier transform differ by a multiplicative factor, which could be eliminated by using an orthonormal instead of an orthogonal basis for $\mathbb{C}^N$. For all practical purposes, the discrete Fourier transform is a norm-preserving transformation of $\mathbb{C}^N$ onto itself.

7.1.3 *Aliasing*

Lemma 18 establishes the relation between the discrete Fourier transform and Fourier series.

Lemma 18 *Let $\hat{\vec{f}} \in \mathbb{C}^N$ be the discrete Fourier transform of $\vec{f} \in \mathbb{C}^N$. The trigonometric polynomial*

$$p_N(x) = \sum_{n=0}^{N-1} \hat{f}_n e^{inx}$$

interpolates the N data points (jh, f_j), where $0 \leq j < N$.

This is an immediate consequence of Equation 7.4. □

Let $f(x) \in L_2$ and $\vec{f} = [f_j]_{j=0}^{N-1} \in \mathbb{C}^N$ such that $f_j = f(jh)$ with $h = 2\pi/N$. The Fourier series of $f(x)$ and the trigonometric polynomial $p_N(x)$ that interpolates $\vec{f}$ are, respectively, given by

$$\begin{aligned} f(x) &= \sum_{k=-\infty}^{+\infty} \hat{f}_k^c e^{ikx} \\ p_N(x) &= \sum_{n=0}^{N-1} \hat{f}_n^d e^{inx}. \end{aligned}$$

We use superscripts c and d to distinguish between continuous and discrete Fourier coefficients. For sufficiently smooth functions $f(x)$, the sequence $p_N(x)$ converges uniformly to $f(x)$ as the number of interpolation points N increases. However, the precise relationship between the Fourier-series coefficients $\hat{f}_k^c$ and the discrete Fourier coefficients $\hat{f}_n^d$ is more complicated.

The trigonometric polynomial $p_N(x)$ contains only terms e^{inx} with positive n. This asymmetry with respect to the Fourier series of $f(x)$ is easily resolved as follows. For all integer ℓ, the function $e^{i\ell Nx}$ equals one at all interpolation points jh. As a result, a trigonometric polynomial obtained by replacing any basis function e^{inx} by $e^{i(\ell N+n)x}$ still interpolates all points.

This property is known as *aliasing*. With $N = 2M$, we perform the substitution

$$\forall n \in M..N-1 : \; e^{inx} \rightarrow e^{i(-N+n)x}.$$

The interpolating trigonometric polynomial so obtained is given by

$$\begin{aligned} q_N(x) &= \sum_{n=0}^{M-1} \hat{f}_n^d e^{inx} + \sum_{n=M}^{N-1} \hat{f}_n^d e^{i(-N+n)x} \\ &= \sum_{n=0}^{M-1} \hat{f}_n^d e^{inx} + \sum_{n=-M}^{-1} \hat{f}_{N+n}^d e^{inx}. \end{aligned}$$

It can be shown that the discrete Fourier coefficients $\hat{f}_n^d$ with $n \in 0..M-1$ converge to the Fourier-series coefficients $\hat{f}_n^c$ as N increases. Similarly, the coefficients $\hat{f}_n^d$ with $n \in M..N-1$ converge to $\hat{f}_{n-N}^c$.

Because of aliasing, one might be led to the mistaken conclusion that trigonometric interpolation is not unique. Finding the interpolating polynomial through N points is uniquely determined, once the set of basis vectors is defined. Among all trigonometric polynomials that are a linear combination of these basis functions, there is exactly one polynomial that interpolates all function values. Aliasing is the effect of changing from one basis to another, for example, from $\{e^{inx} : 0 \leq n < N\}$ to $\{e^{inx} : -M \leq n < M\}$.

Aliasing is particularly important when one wants to distinguish between high and low frequencies. In Fourier series, the absolute value of n determines whether the term $\hat{f}_n^c e^{inx}$ is of high or low frequency. To make this distinction among the discrete Fourier modes, one must take the corresponding terms in $q_N(x)$, not in $p_N(x)$. Thus, the highest frequency on a regular grid of size N is $M = N/2$. To divide up the discrete spectrum equally between low and high, the following criterion can be used: n is a high frequency if $|\theta_n| \geq \frac{\pi}{2}$, where $\theta_n = nh$ is interpreted as an angle and is reduced to the interval $[-\pi, \pi]$.

7.2 Divide and Conquer

Comparison of Equations 7.4 (the inverse transform) and 7.5 (the forward transform) shows that the inverse transform differs from the forward transform only by a multiplicative factor and a sign in the exponents. Both transforms map vectors of $\mathbb{C}^N$ into vectors of $\mathbb{C}^N$. For the purpose of algorithm development, the discrete Fourier transform is identical to its inverse. Without loss of generality, we shall concentrate on developing an algorithm for the inverse transform. Slightly changing notation, we must compute

$$\forall k \in 0..N-1 : \; \hat{f}_k = \sum_{n=0}^{N-1} f_n e^{iknh}. \tag{7.7}$$

Each of the N discrete Fourier coefficients $\hat{f}_k$ could be computed by summing N terms. This would require $2N^2$ complex floating-point operations. The cost of evaluating the expressions e^{iknh} is not counted, because it can be amortized over many transforms. In practice, the exponential constants are saved in a look-up table.

Given the trigonometric polynomial

$$p(x) = \sum_{n=0}^{N-1} f_n e^{inx} \tag{7.8}$$

and the N equidistant points $x_k = kh$, where $h = \frac{2\pi}{N}$ and $k \in 0..N-1$, it is clear that $\hat{f}_k = p(x_k)$. The problem of computing the N discrete Fourier coefficients $\hat{f}_k$ is thus equivalent to evaluating $p(x)$ at the N points x_k. For the purpose of starting an induction argument, we assume that for all $M < N$ a procedure exists to evaluate trigonometric polynomials of degree $M-1$ at M equidistant points $y_\ell = \ell\frac{2\pi}{M}$, where $\ell \in 0..M-1$. Our goal is to reduce the problem of size N into several smaller problems.

With $N = 2M$, split $p(x)$ into an even and an odd part:

$$\begin{aligned} p(x) &= \sum_{m=0}^{M-1} f_{2m} e^{i2mx} + \sum_{m=0}^{M-1} f_{2m+1} e^{i(2m+1)x} \\ &= \sum_{m=0}^{M-1} f_{2m} e^{i2mx} + e^{ix} \sum_{m=0}^{M-1} f_{2m+1} e^{i2mx} \\ &= q(2x) + e^{ix} r(2x). \end{aligned}$$

By induction, $q(y_k)$ and $r(y_k)$ are known quantities for all $k \in 0..M-1$. Moreover, it follows from $N = 2M$ that

$$\forall k \in 0..M-1 \; : \; \left\{ \begin{array}{lcl} 2x_k & = & y_k \\ 2x_{k+M} & = & y_k + 2\pi. \end{array} \right.$$

This and the 2π-periodicity of trigonometric polynomials imply that

$$\forall k \in 0..M-1 : \left\{ \begin{array}{lcl} p(x_k) & = & q(y_k) + e^{ik\frac{\pi}{M}} r(y_k) \\ p(x_{k+M}) & = & q(y_k) - e^{ik\frac{\pi}{M}} r(y_k). \end{array} \right. \tag{7.9}$$

The evaluation of a trigonometric polynomial of degree $N-1$ at N equidistant points is thus reduced to the evaluation of two trigonometric polynomials of degree $N/2-1$ at $N/2$ equidistant points. If $N = 2^n$, this dividing process can be continued until the trigonometric polynomials are just constants.

Before developing this method further, let us examine the operation count of the divide-and-conquer strategy. Let $W(N)$ be the number of complex floating-point operations to evaluate $p(x)$ at N equidistant points

by this procedure. We must evaluate $q(x)$ and $r(x)$ at $M = N/2$ equidistant points and perform the work specified by Equation 7.9. This implies that

$$W(2M) = 2W(M) + 4M.$$

Again, the cost of evaluating exponential constants like $e^{ik\frac{\pi}{M}}$ is ignored, because it can be amortized over many transforms. Introducing $w_j = W(2^j)$, we have that $w_0 = W(1) = 0$ and that

$$\forall j \in 1..n : \; w_j = 2w_{j-1} + 2 \times 2^j.$$

Multiply the equation for w_j by 2^{n-j}, and sum for $j \in 1..n$:

$$\begin{aligned} \sum_{j=1}^{n} 2^{n-j} w_j &= 2 \sum_{j=1}^{n} 2^{n-j} (w_{j-1} + 2^j) \\ &= 2n2^n + \sum_{j=0}^{n-1} 2^{n-j} w_j. \end{aligned}$$

This implies that $w_n = 2n2^n$ and $W(N) = 2N \log_2 N$. This is a substantial reduction over the primitive procedure, which required $2N^2$ complex floating-point operations.

7.3 The Fast-Fourier-Transform Algorithm

Methods that compute the discrete Fourier transform in $O(N \log N)$ complex floating-point operations are referred to as *fast Fourier transforms*, FFT for short.

Based on the odd-even decomposition of a trigonometric polynomial, a problem of size 2^n is reduced to two problems of size 2^{n-1}. Subsequently, two problems of size 2^{n-1} are reduced to four problems of size 2^{n-2}. Ultimately, 2^n problems of size 1 are obtained, each of which is solved trivially. The implementation hinges on a precise, albeit cumbersome, notation to identify each subproblem. The term subproblem in this context refers to the evaluation of a trigonometric polynomial on a set of equidistant points. The size of a subproblem is the number of points at which the trigonometric polynomial is evaluated.

If the main problem has size 2^n, there are 2^{n-m} subproblems of size 2^m. The trigonometric polynomials corresponding to these subproblems are denoted by:

$$\forall m \in 0..n, \forall r \in 0..2^{n-m}-1 : \; p_r^{(m)}(x). \tag{7.10}$$

There are twice as many subproblems of size 2^m as there are subproblems of size 2^{m+1}. Each problem of size 2^m is either the even or the odd part of a problem of size 2^{m+1}. The indices of the polynomials are chosen such

that $p_r^{(m)}(x)$ is the even part of $p_r^{(m+1)}(x)$ and $p_{r+2^{n-m-1}}^{(m)}(x)$ its odd part. With this choice of indices, we have that

$$\forall m \in 0..n-1, \forall r \in 0..2^{n-m-1}-1 : \\ p_r^{(m+1)}(x) = p_r^{(m)}(2x) + e^{ix} p_{r+2^{n-m-1}}^{(m)}(2x). \qquad (7.11)$$

Consider the first three odd-even decompositions. For $m = n-1$, we obtain the odd-even decomposition of the main problem:

$$p_0^{(n)}(x) = p_0^{(n-1)}(2x) + e^{ix} p_1^{(n-1)}(2x).$$

The decomposition of the problems of size 2^{n-1} is given by:

$$\begin{aligned} p_0^{(n-1)}(x) &= p_0^{(n-2)}(2x) + e^{ix} p_{0+2}^{(n-2)}(2x) \\ p_1^{(n-1)}(x) &= p_1^{(n-2)}(2x) + e^{ix} p_{1+2}^{(n-2)}(2x), \end{aligned}$$

and for the problems of size 2^{n-2} we obtain:

$$\begin{aligned} p_0^{(n-2)}(x) &= p_0^{(n-3)}(2x) + e^{ix} p_{0+4}^{(n-3)}(2x) \\ p_1^{(n-2)}(x) &= p_1^{(n-3)}(2x) + e^{ix} p_{1+4}^{(n-3)}(2x) \\ p_2^{(n-2)}(x) &= p_2^{(n-3)}(2x) + e^{ix} p_{2+4}^{(n-3)}(2x) \\ p_3^{(n-2)}(x) &= p_3^{(n-3)}(2x) + e^{ix} p_{3+4}^{(n-3)}(2x). \end{aligned}$$

These relations define a hierarchy between these polynomials, which is represented by a binary tree:

$$p_0^{(n)}(x) \left\{ \begin{array}{l} p_0^{(n-1)}(x) \left\{ \begin{array}{l} p_0^{(n-2)}(x) \left\{ \begin{array}{l} p_0^{(n-3)}(x) \\ p_4^{(n-3)}(x) \end{array} \right. \\ p_2^{(n-2)}(x) \left\{ \begin{array}{l} p_2^{(n-3)}(x) \\ p_6^{(n-3)}(x) \end{array} \right. \end{array} \right. \\ p_1^{(n-1)}(x) \left\{ \begin{array}{l} p_1^{(n-2)}(x) \left\{ \begin{array}{l} p_1^{(n-3)}(x) \\ p_5^{(n-3)}(x) \end{array} \right. \\ p_3^{(n-2)}(x) \left\{ \begin{array}{l} p_3^{(n-3)}(x) \\ p_7^{(n-3)}(x) \end{array} \right. \end{array} \right. \end{array} \right.$$

The top and bottom descendants in this binary tree correspond to the even and odd part of the parent, respectively.

The trigonometric polynomials $p_r^{(m)}(x)$, expressed in terms of the original data f_j, are given by

$$\forall m \in 0..n, \forall r \in 0..2^{n-m}-1 : \; p_r^{(m)}(x) = \sum_{j=0}^{2^m-1} f_{2^{n-m}j+r} e^{ijx}. \qquad (7.12)$$

Two important special cases of this formula occur for $m = 0$ and $m = n$, for which we obtain, respectively,

$$\forall r \in 0..2^n - 1 : \quad p_r^{(0)}(x) = f_r \qquad (7.13)$$

$$\forall k \in 0..2^n - 1 : \quad p_0^{(n)}(x_k^{(n)}) = \hat{f}_k. \qquad (7.14)$$

The points $x_k^{(n)}$ define the equidistant grid on which $p_0^{(n)}(x)$ is evaluated.

We must evaluate $p_r^{(m+1)}(x)$ at 2^{m+1} equidistant points in the interval $[0, 2\pi]$. These points are given by the familiar formula:

$$\forall k \in 0..2^{m+1}-1 : \; x_k^{(m+1)} = k\frac{2\pi}{2^{m+1}}. \qquad (7.15)$$

It is clear that

$$\forall k \in 0..2^m - 1 : \left\{ \begin{array}{lcl} 2x_k^{(m+1)} &=& x_k^{(m)} \\ 2x_{k+2^m}^{(m+1)} &=& 2\pi + x_k^{(m)}. \end{array} \right. \qquad (7.16)$$

We can now rewrite the fundamental divide-and-conquer formulas of Equation 7.9, using the notation of Equations 7.10, 7.11, 7.15, and 7.16. This results in the following identities:

$$\forall m \in 0..n-1, \forall r \in 0..2^{n-m-1}-1, \forall k \in 0..2^m - 1 : \qquad (7.17)$$
$$\begin{bmatrix} p_r^{(m+1)}(x_k^{(m+1)}) \\ p_r^{(m+1)}(x_{k+2^m}^{(m+1)}) \end{bmatrix} = \begin{bmatrix} 1 & e^{ik\frac{2\pi}{2^{m+1}}} \\ 1 & -e^{ik\frac{2\pi}{2^{m+1}}} \end{bmatrix} \begin{bmatrix} p_r^{(m)}(x_k^{(m)}) \\ p_{r+2^{n-m-1}}^{(m)}(x_k^{(m)}) \end{bmatrix}.$$

A program is derived from these formulas by assigning each value to a program variable. As a first step, store

$$\begin{aligned} p_r^{(m)}(x_k^{(m)}) &\quad \text{in variable} \quad \beta[m,r,k] \\ e^{ik\frac{2\pi}{2^{m+1}}} = e^{i2^{n-m-1}k\frac{2\pi}{2^n}} &\quad \text{in variable} \quad \epsilon[2^{n-m-1}k]. \end{aligned}$$

Equation 7.17 immediately implies the main part of the program: an assignment to a vector of two variables within a concurrent quantification over r and k. The quantification over the index m is sequential, however, because subproblems of size 2^{m+1} depend on subproblems of size 2^m. Equation 7.13 specifies the initialization of the variables $\beta[0,r,0]$. Equation 7.14 tells us which variables hold the values $\hat{f}_k$ upon termination.

$$\begin{array}{l} \langle \parallel \; r \; : \; 0 \le r < 2^n \; :: \; \beta[0,r,0] := f[r] \; \rangle \; ; \\ \langle \; ; \; m \; : \; 0 \le m < n \; :: \\ \quad \langle \parallel \; r,k \; : \; 0 \le r < 2^{n-m-1} \text{ and } 0 \le k < 2^m \; :: \\ \quad\quad \begin{bmatrix} \beta[m+1,r,k] \\ \beta[m+1,r,k+2^m] \end{bmatrix} := \begin{bmatrix} 1 & \epsilon[2^{n-m-1}k] \\ 1 & -\epsilon[2^{n-m-1}k] \end{bmatrix} \begin{bmatrix} \beta[m,r,k] \\ \beta[m,r+2^{n-m-1},k] \end{bmatrix} \\ \quad \rangle \\ \rangle \; ; \\ \langle \parallel \; k \; : \; 0 \le k < 2^n \; :: \; \hat{f}[k] := \beta[n,0,k] \; \rangle \end{array}$$

The remainder of the development of the fast-Fourier-transform algorithm is concerned with efficient use of memory. Our goal is to reorganize the computations such that the transformed data and all intermediate results share identical memory locations. This program transformation must preserve the concurrency of the inner quantification.

The key observation is that the innermost assignment in the above program uses two entries of the array $\beta[m]$ to evaluate two new entries of the array $\beta[m+1]$. Once the assignment has been performed, the values of the right-hand-side variables may be discarded. To replace the array β by a one-dimensional array b, we need a map τ from the β-indices $[m,r,k]$ to a b-index $[t]$ such that each assignment overwrites its own right-hand-side variables. Such a map $t=\tau(m,r,k)$ must satisfy:

$$\forall m \in 0..n-1, \forall r \in 0..2^{n-m-1}-1, \forall k \in 0..2^m-1: \tag{7.18}$$

$$\text{either} \quad \begin{cases} \tau(m+1,r,k) &= \tau(m,r,k) \\ \tau(m+1,r,k+2^m) &= \tau(m,r+2^{n-m-1},k) \end{cases}$$

$$\text{or} \quad \begin{cases} \tau(m+1,r,k) &= \tau(m,r+2^{n-m-1},k) \\ \tau(m+1,r,k+2^m) &= \tau(m,r,k). \end{cases}$$

If such a map τ is available, the program can be transformed as follows.

$$\begin{array}{l} \langle \parallel\ r\ :\ 0 \le r < 2^n\ ::\ b[\tau(0,r,0)] := f_r\ \rangle\ ; \\ \langle\ ;\ m\ :\ 0 \le m < n\ :: \\ \quad \langle \parallel\ r,k\ :\ 0 \le r < 2^{n-m-1} \textbf{ and } 0 \le k < 2^m\ :: \\ \quad\quad \begin{bmatrix} b[\tau(m+1,r,k)] \\ b[\tau(m+1,r,k+2^m)] \end{bmatrix} := \begin{bmatrix} 1 & \epsilon[2^{n-m-1}k] \\ 1 & -\epsilon[2^{n-m-1}k] \end{bmatrix} \begin{bmatrix} b[\tau(m,r,k)] \\ b[\tau(m,r+2^{n-m-1},k)] \end{bmatrix} \\ \quad \rangle \\ \rangle\ ; \\ \langle \parallel\ k\ :\ 0 \le k < 2^n\ ::\ \hat{f}[k] := b[\tau(n,0,k)]\ \rangle \end{array}$$

One particular map $t=\tau(m,r,k)$ that satisfies Equation 7.18 relies on the following definition.

Definition 6 *The* bit-reversal map ρ_n *is a bijection of* $0..2^n-1$ *to itself, such that:*

$$\rho_n\Big(\sum_{j=0}^{n-1}\alpha_j 2^j\Big) = \sum_{j=0}^{n-1}\alpha_{n-1-j}2^j,$$

where $\forall j \in 0..n-1:\ \alpha_j \in \{0,1\}$.

As implied by its name, the bit-reversal map of an integer with binary representation '$\alpha_{n-1}\alpha_{n-2}\ldots\alpha_1\alpha_0$' is the integer with binary representation '$\alpha_0\alpha_1\ldots\alpha_{n-2}\alpha_{n-1}$'.

The following lemma constructs a map $t=\tau(m,r,k)$ that satisfies Equation 7.18.

Lemma 19 *The map*

$$t = \tau(m,r,k) = \rho_n(r) + k \tag{7.19}$$

satisfies $\forall m \in 0..n-1, \forall r \in 0..2^{n-m-1}-1, \forall k \in 0..2^m-1:$

$$\tau(m+1,r,k) = \tau(m,r,k) \tag{7.20}$$

$$\tau(m+1,r,k+2^m) = \tau(m,r+2^{n-m-1},k). \tag{7.21}$$

The proof of Equation 7.20 is trivial. To prove Equation 7.21, let the binary representation of r be given by '$\alpha_{n-m-2}\alpha_{n-m-3}\ldots\alpha_0$'. In this case, we have that

$$\begin{aligned} \tau(m,r+2^{n-m-1},k) &= \rho_n(r+2^{n-m-1})+k \\ &= \rho_n(2^{n-m-1}+\sum_{j=0}^{n-m-2}\alpha_j 2^j)+k \\ &= 2^m+\sum_{\ell=m+1}^{n-1}\alpha_{n-1-\ell}2^\ell+k \\ &= \rho_n(r)+2^m+k. \end{aligned}$$

This and Equation 7.19 imply Equation 7.21. □

Equation 7.19 also implies that

$$\begin{aligned} \tau(n,0,k) &= k \\ \tau(0,r,0) &= \rho_n(r). \end{aligned}$$

Together with Equations 7.20 and 7.21, this leads to the following program.

$$\begin{array}{l} \langle \parallel\ r\ :\ 0 \le r < 2^n\ ::\ b[\rho_n(r)] := f_r\ \rangle\ ; \\ \langle\ ;\ m\ :\ 0 \le m < n\ :: \\ \quad \langle \parallel\ r,k\ :\ 0 \le r < 2^{n-m-1} \textbf{ and } 0 \le k < 2^m\ :: \\ \quad\quad \begin{bmatrix} b[\rho_n(r)+k] \\ b[\rho_n(r)+k+2^m] \end{bmatrix} := \begin{bmatrix} 1 & \epsilon[2^{n-m-1}k] \\ 1 & -\epsilon[2^{n-m-1}k] \end{bmatrix} \begin{bmatrix} b[\rho_n(r)+k] \\ b[\rho_n(r)+k+2^m] \end{bmatrix} \\ \quad \rangle \\ \rangle\ ; \\ \langle \parallel\ k\ :\ 0 \le k < 2^n\ ::\ \hat{f}[k] := b[k]\ \rangle \end{array}$$

We conclude the development of the fast-Fourier-transform algorithm by simplifying the index arithmetic of the program. From the definition of the bit-reversal map, it follows that

$$\{\rho_n(r) : 0 \le r < 2^{n-m-1}\} = \{2^{m+1}j : 0 \le j < 2^{n-m-1}\}.$$

Using the variable j instead of r, all occurrences of $\rho_n(r)$ may be replaced by $2^{m+1}j$, while j ranges over the index set $0..2^{n-m-1}-1$. The expression

$\rho_n(r) + k = 2^{m+1}j + k$ takes on all values in the range $0..2^n - 1$ that have a binary representation in which bit number m is zero. This observation allows us to replace the two indices k and r by one index $t = \rho_n(r) + k$.

The assignments to $b[t]$ and $b[t + 2^m]$ for $t = \rho_n(r) + k$ require the exponential constant $\epsilon[2^{n-m-1}k]$. To replace k and r by the single index t, one must express $2^{n-m-1}k$ in terms of t only. It follows from $t = 2^{m+1}j + k$ that

$$2^{n-m-1}k = 2^{n-m-1}t - 2^n j.$$

However, we already know that $0 \leq k < 2^m$ and $0 \leq 2^{n-m-1}k < 2^{n-1}$. The term $2^n j$ merely shifts $2^{n-m-1}t$ into the right range. This can also be accomplished by modulo arithmetic, and we obtain that

$$2^{n-m-1}k = 2^{n-m-1}t \bmod 2^{n-1}.$$

The binary representation of $2^{n-m-1}t \bmod 2^{n-1}$ is obtained from that of t by zeroing the $n-m$ most significant bits of t and, subsequently, performing $n - m - 1$ shifts to the left.

After incorporating the above changes in index arithmetic, we obtain program Fast-Fourier-1.

program Fast-Fourier-1
declare
  $m,\ t$: integer ;
  ϵ : array$[0..2^{n-1} - 1]$ of complex ;
  b : array$[0..2^n - 1]$ of complex
initially
  $\langle\ ;\ t : 0 \leq t < 2^{n-1} :: \epsilon[t] = e^{it\frac{2\pi}{2^n}}\ \rangle$;
  $\langle\ ;\ t : 0 \leq t < 2^n :: b[\rho_n(t)] = f_t\ \rangle$
assign
  $\langle\ ;\ m : 0 \leq m < n ::$
    $\langle \parallel\ t : t \in 0..2^n - 1$ **and** $t \wedge 2^m = 0 ::$

$$\begin{bmatrix} b[t] \\ b[t + 2^m] \end{bmatrix} := \begin{bmatrix} 1 & \epsilon[2^{n-m-1}t \bmod 2^{n-1}] \\ 1 & -\epsilon[2^{n-m-1}t \bmod 2^{n-1}] \end{bmatrix} \begin{bmatrix} b[t] \\ b[t + 2^m] \end{bmatrix}$$

    $\rangle$
  $\rangle$
end

The information flow of program Fast-Fourier-1 is identical to that of the recursive-doubling procedure: in step m, information is exchanged between array entries $b[t]$ and $b[t\bar{\vee}2^m]$. (Note that $t\bar{\vee}2^m = t + 2^m$ when $t \wedge 2^m = 0$.) However, the fast Fourier transform differs from the standard recursive-doubling procedure, because the assignment to $b[t]$ differs from the assignment to $b[t\bar{\vee}2^m]$. Like the recursive-doubling procedure of the concurrent QR-decomposition (see Section 5.4), the fast-Fourier-transform algorithm is an asymmetric recursive-doubling procedure.

The asymmetry can be made more explicit by rewriting the concurrent quantification over t as follows.

$\langle \parallel\ t : t \in 0..2^n - 1 ::$
  if $t \wedge 2^m = 0$ **then**
    $b[t] := b[t] + \epsilon[2^{n-m-1}t \bmod 2^{n-1}]b[t\bar{\vee}2^m]$
  else
    $b[t] := b[t\bar{\vee}2^m] - \epsilon[2^{n-m-1}t \bmod 2^{n-1}]b[t]$
$\rangle$

The case $t \wedge 2^m \neq 0$ exploits the fact that the expression $2^{n-m-1}t \bmod 2^{n-1}$ does not depend on bit number m of t.

7.4 Multicomputer Implementation

It is our intention to compute the discrete Fourier transform of vectors of dimension $N = 2^n$ using $P = 2^D$ processes, where $P \leq N$ and $D \leq n$. The transformation begins, as usual, with a duplication step. Subsequently, a data distribution is imposed in the selection step. Because P divides N, we shall use perfectly load-balanced data distributions. We shall not impose any other requirements on the data distribution.

The binary representation of global index $t \in 0..2^n - 1$ is a bit string of length n, say

$$t = `\alpha_{n-1}\alpha_{n-2}\ldots\alpha_1\alpha_0'.$$

Any P-fold perfectly load-balanced data distribution is defined by a permutation of the bit positions. Given the permutation

$$k_0, k_1, \ldots, k_{D-1}, k_D, \ldots, k_{n-1},$$

the global index t is mapped to process identifier p and local index j, with

$$\begin{aligned} p &= `\alpha_{k_0}\alpha_{k_1}\ldots\alpha_{k_{D-1}}' \\ j &= `\alpha_{k_D}\alpha_{k_{D+1}}\ldots\alpha_{k_{n-1}}'. \end{aligned}$$

For example, the perfectly load-balanced linear data distribution is based on the permutation

$$n-1, n-2, \ldots, n-D, n-D-1, \ldots, 0,$$

while the perfectly load-balanced scatter data distribution is based on

$$D-1, D-2, \ldots, 0, n-1, \ldots, D.$$

The set $T_D = \{k_0, k_1, \ldots, k_{D-1}\}$ is the set of D leading bit positions, the set of bit positions that forms the process identifier.

A perfectly load-balanced data distribution splits the binary representation of an integer into two parts. To develop the multicomputer version of program Fast-Fourier-1, we shall use the short-hand notation

$$t = t_D, t_l$$

to denote such a splitting. For the global index t, the "leading part" t_D will become the process identifier, and the "trailing part" t_l will become the local index. To split the integer 2^m in this fashion, we write

$$2^m = (2^m)_D, (2^m)_l.$$

The values of $(2^m)_D$ and $(2^m)_l$ are determined by the rank of m in the permutation of the bit positions. Either $(2^m)_D$ or $(2^m)_l$ must vanish; the other must be a power of two.

Before performing a P-fold duplication of program Fast-Fourier-1, replace its quantification over t by the equivalent quantification listed at the end of Section 7.3. This makes explicit the asymmetric nature of the recursive-doubling procedure. We shall focus on the transformation of this quantification. When studying the program segments that follow, bear in mind that those segments are surrounded by a sequential quantification over m and, because of the P-fold duplication, by a concurrent quantification over p. Moreover, all variables have an implicit process identifier p. The inner quantification of the duplicated program is given by:

$\langle \parallel\ t\ :\ t \in 0..2^n - 1\ ::$
  if $t \wedge 2^m = 0$ **then**
    $b[t] := b[t] + \epsilon[t_m] b[t \bar{\vee} 2^m]$
  else
    $b[t] := b[t \bar{\vee} 2^m] - \epsilon[t_m] b[t]$
$\rangle$

This program segment uses the short-hand notation t_m for the expression $2^{n-m-1}t \bmod 2^{n-1}$.

To prepare for the selection step, which will distribute the array b over the processes, the global index t is split into its leading and trailing parts. To do this, we must transform three expressions:

- $t \in 0..2^n - 1$, which is used in the quantification,
- $t \wedge 2^m = 0$, which is used in the **if** test, and
- $t \bar{\vee} 2^m$, which is used as an index for b.

Because the constant array ϵ remains duplicated, we need not consider the transformation of the expression $t_m = 2^{n-m-1}t \bmod 2^{n-1}$ in detail: given t_D and t_l, it is always possible to compute t and, subsequently, t_m.

As t ranges over all values of the index set $0..2^n - 1$, its binary representation ranges over all possible bit strings of length n. It follows that $t \in 0..2^n - 1$ is equivalent to

$$t_D \in 0..2^D - 1 \textbf{ and } t_l \in 0..2^{n-D} - 1.$$

The expression $t \wedge 2^m = 0$ tests whether or not bit number m of t is set. The position of bit number m in the splitting t_D, t_l depends on the rank of m in the permutation of the bit positions. If m is one of the leading positions, say $m \in \mathcal{T}_D$, then bit number m is mapped to the leading part, and the test **if** $t \wedge 2^m = 0$ **then**... is transformed into

$$\textbf{if } t_D \wedge (2^m)_D = 0 \textbf{ then} \ldots$$

If m is one the trailing positions, say $m \notin \mathcal{T}_D$, then bit number m is mapped to the trailing part, and the test is transformed into

$$\textbf{if } t_l \wedge (2^m)_l = 0 \textbf{ then} \ldots$$

The expression $t \bar{\vee} 2^m$ reverses bit number m of t. After locating bit number m in the splitting of t, that bit is reversed. We find that

$$m \in \mathcal{T}_D \Rightarrow \begin{cases} (t \bar{\vee} 2^m)_D = t_D \bar{\vee} (2^m)_D \\ (t \bar{\vee} 2^m)_l = t_l \end{cases}$$

$$m \notin \mathcal{T}_D \Rightarrow \begin{cases} (t \bar{\vee} 2^m)_D = t_D \\ (t \bar{\vee} 2^m)_l = t_l \bar{\vee} (2^m)_l. \end{cases}$$

To incorporate the index splitting into the previous program segment, one must consider separately two important cases: $m \in \mathcal{T}_D$ and $m \notin \mathcal{T}_D$. The remainder of the transformation follows by mere substitution.

if $m \notin \mathcal{T}_D$ **then**
  $\langle \parallel\ t_D, t_l\ :\ t_D \in 0..2^D - 1$ **and** $t_l \in 0..2^{n-D} - 1\ ::$
    if $t_l \wedge (2^m)_l = 0$ **then**
      $b[t_D, t_l] := b[t_D, t_l] + \epsilon[t_m] b[t_D, t_l \bar{\vee} (2^m)_l]$
    else
      $b[t_D, t_l] := b[t_D, t_l \bar{\vee} (2^m)_l] - \epsilon[t_m] b[t_D, t_l]$
  $\rangle$
else
  $\langle \parallel\ t_D, t_l\ :\ t_D \in 0..2^D - 1$ **and** $t_l \in 0..2^{n-D} - 1\ ::$
    if $t_D \wedge (2^m)_D = 0$ **then**
      $b[t_D, t_l] := b[t_D, t_l] + \epsilon[t_m] b[t_D \bar{\vee} (2^m)_D, t_l]$
    else
      $b[t_D, t_l] := b[t_D \bar{\vee} (2^m)_D, t_l] - \epsilon[t_m] b[t_D, t_l]$
  $\rangle$

Remember that the above program segment is surrounded by a sequential quantification over m. It is also implicitly duplicated.

The selection step imposes the data distribution on array b but leaves the entire constant array ϵ duplicated in every process. After data distribution, process p may access only the variables $b[p][t_D, t_l]$ with $t_D = p$. As long as $m \notin \mathcal{T}_D$, it is easy to incorporate this restriction, because assignments involve only entries of b whose indices have the same leading part. When $m \in \mathcal{T}_D$, however, assignments are between entries of b whose indices have different leading parts. Again, process p may assign new values to $b[p][t_D, t_l]$ only if $t_D = p$. However, assignments to $b[p][t_D, t_l]$ require access to $b[p\bar{\vee}(2^m)_D][t_D\bar{\vee}(2^m)_D, t_l]$. The latter must be obtained by communication with process $p\bar{\vee}(2^m)_D$. The following program segment results.

```
if m ∉ T_D then
  ⟨ ‖ t_D, t_l : t_D = p and t_l ∈ 0..2^{n-D} − 1 ::
    if t_l ∧ (2^m)_l = 0 then
      b[t_D, t_l] := b[t_D, t_l] + ε[t_m]b[t_D, t_l ∨̄ (2^m)_l]
    else
      b[t_D, t_l] := b[t_D, t_l ∨̄ (2^m)_l] − ε[t_m]b[t_D, t_l]
  ⟩
else begin
  send {b[t_D, t_l] : t_D = p and t_l ∈ 0..2^{n-D} − 1} to p∨̄(2^m)_D ;
  receive {x[t_D, t_l] : t_D = p and t_l ∈ 0..2^{n-D} − 1} from p∨̄(2^m)_D ;
  ⟨ ‖ t_D, t_l : t_D = p and t_l ∈ 0..2^{n-D} − 1 ::
    if t_D ∧ (2^m)_D = 0 then
      b[t_D, t_l] := b[t_D, t_l] + ε[t_m]x[t_D, t_l]
    else
      b[t_D, t_l] := x[t_D, t_l] − ε[t_m]b[t_D, t_l]
  ⟩
end
```

A few additional transformations, mostly cosmetic, lead to the final version. The leading part of the global index has become superfluous, since it is always equal to p in process p. The trailing part t_l becomes the local index j. Given $j(= t_l)$ and $p(= t_D)$, it is always possible to compute the corresponding global index t. Subsequently, the index t_m used to retrieve exponential constants from the array ϵ can be computed. The details of the calculation of t_m depend, of course, on the permutation of the bit positions that defines the data distribution. In the program, we denote the calculation of t_m as a function of p, j, and m by $t_m(p, j)$. The test

$$\textbf{if } t_D \wedge (2^m)_D = 0 \textbf{ then } \ldots$$

is independent of t_l and can be taken outside of the last quantification.

Because the individual processes of a multicomputer computation are sequential, we convert all concurrent into sequential quantifications. This is easy for the case $m \in \mathcal{T}_D$. In the other case, the quantification is first reformulated as in program Fast-Fourier-1. Subsequently, the concurrent separator is replaced by the sequential separator.


```
0..P − 1 ‖ p program Fast-Fourier-2
declare
  m, j : integer ;
  ε : array [0..2^{n-1} − 1] of complex ;
  b, x : array [0..2^{n-D}] of complex
assign
  ⟨ ; m : 0 ≤ m < n ::
    if m ∉ T_D then
      ⟨ ; j : j ∈ 0..2^{n-D} − 1 and j ∧ (2^m)_l = 0 ::
        [ b[j]            ]   [ 1   ε[t_m(p, j)]  ] [ b[j]            ]
        [ b[j ∨̄ (2^m)_l] ] := [ 1   −ε[t_m(p, j)] ] [ b[j ∨̄ (2^m)_l] ]
      ⟩
    else begin
      send {b[j] : j ∈ 0..2^{n-D} − 1} to p∨̄(2^m)_D ;
      receive {x[j] : j ∈ 0..2^{n-D} − 1} from p∨̄(2^m)_D ;
      if p ∧ (2^m)_D = 0 then
        ⟨ ; j : j ∈ 0..2^{n-D} − 1 :: b[j] := b[j] + ε[t_m(p, j)]x[j] ⟩
      else
        ⟨ ; j : j ∈ 0..2^{n-D} − 1 :: b[j] := x[j] − ε[t_m(p, j)]b[j] ⟩
      end
  ⟩
end
```

The performance analysis can be brief. Because the set $\mathcal{T}_D$ of leading bit positions contains exactly D values in the range of m, there are D steps with and $n - D$ steps without communication, and

$$\begin{aligned} T_P &= (n-D)2^{n-D}2\tau_A + D(\tau_C(2^{n-D}) + 2^{n-D}2\tau_A) \\ &= \frac{2nN}{P}\tau_A + D\tau_C(\frac{N}{P}). \end{aligned} \tag{7.22}$$

The corresponding speed-up is given by:

$$S_P = \frac{T_1^S}{T_P} = P\frac{1}{1 + \frac{D}{2n}(\frac{P}{N}\frac{\tau_S}{\tau_A} + \frac{\beta}{\tau_A})}.$$

Coarse-grained computations with $P \ll N$ can achieve excellent speed-up. For fine-grained computations with $P = N$ and $D = n$, we obtain that

$$S_P = P\frac{1}{1 + \frac{\tau_S + \beta}{2\tau_A}}.$$

The ratio $\frac{\tau_S}{\tau_A}$ can be quite large and severely limits the efficiency of fine-grained computations (under our standard assumption of one process per node).

7.5 Generalizations

7.5.1 Bit-Reversal Maps

Programs Fast-Fourier-1 and Fast-Fourier-2 compute the values $\hat{f}_k$ in natural order, provided the values f_r are stored in bit-reversed order. By using a different map τ (recall Equation 7.18), a version can be obtained that computes the values $\hat{f}_k$ in a bit-reversed order, provided the values f_r are in natural order. Both versions are useful. If the forward transform computes the frequencies in bit-reversed order from the values stored in natural order, it is convenient to use an inverse transform that takes the bit-reversed frequencies as input to transform them into natural-order values. When using this combination of forward and inverse fast-Fourier-transform programs, one merely has to adhere to the convention that vectors in physical space are stored in natural order and those in Fourier space in bit-reversed order. With this convention, the inverse-transform program uses the map τ of Equation 7.19. For the map used in the forward-transform program, see Exercise 30.

If it is acceptable to use an extra array for intermediate and final results, no bit-reversal map is necessary at all. With an auxiliary array, the order in which quantities are stored becomes irrelevant for correctness. However, the storage order remains a performance issue, because it determines memory-access and communication patterns. With the bit-reversal maps, we obtained an efficient recursive-doubling structure. This is probably a far more important reason for using bit-reversal maps than saving memory.

7.5.2 General Dimensions

The divide-and-conquer strategy behind the fast-Fourier-transform algorithm can be generalized to transform vectors of any dimension N, not just dimensions that are a power of 2. For $N = rM$ with r a prime number, a trigonometric polynomial $p(x)$ of degree $N-1$ is decomposed as follows:

$$\begin{aligned} p(x) = \sum_{n=0}^{N-1} f_n e^{inx} &= \sum_{m=0}^{M-1}\sum_{\ell=0}^{r-1} f_{rm+\ell} e^{i(rm+\ell)x} \\ &= \sum_{\ell=0}^{r-1} e^{i\ell x} \sum_{m=0}^{M-1} f_{rm+\ell} e^{irmx} \\ &= \sum_{\ell=0}^{r-1} e^{i\ell x} q_\ell(rx). \end{aligned}$$

As in Section 7.2, this decomposition can be used as the basis for a divide-and-conquer strategy. This generalization is effective as long as N can be factored into a large number of small primes.

7.5.3 The Discrete Real Fourier Transform

Equation 7.7 implies that the discrete Fourier transform $\tilde{f} = [\tilde{f}_k]_{k=0}^{M-1}$ of a real M-dimensional vector $\vec{f} = [f_m]_{m=0}^{M-1}$ satisfies the symmetry:

$$\forall k \in 0..M-1: \quad \tilde{f}_k = \overline{\tilde{f}_{M-k}}. \tag{7.23}$$

Using this symmetry, the transform of a real M-dimensional vector will be reduced to the transform of a complex N-dimensional vector with $N = M/2$. Throughout this section, the tilde notation will be used for transforms of M-dimensional vectors and the hat notation for transforms of N-dimensional vectors.

Writing $\tilde{f}_k = \tilde{a}_k + i\tilde{b}_k$ with $\tilde{a}_k$ and $\tilde{b}_k$ real, Equation 7.23 implies that the discrete Fourier transform of a real vector of dimension $M = 2N$ is defined by the M real coefficients:

$$\tilde{a}_0, \tilde{a}_1, \tilde{b}_1, \ldots, \tilde{a}_{N-1}, \tilde{b}_{N-1}, \tilde{a}_N.$$

Noting that $\tilde{b}_0 = \tilde{b}_N = 0$ or, equivalently, that $\tilde{f}_0$ and $\tilde{f}_N$ are real, it is thus sufficient to compute the N complex coefficients:

$$\tilde{f}_0 + i\tilde{f}_N, \tilde{f}_1, \ldots, \tilde{f}_{N-1}.$$

Consider the discrete Fourier transform of the complex N-dimensional vector $\vec{u} = \vec{v} + i\vec{w} = [f_{2n}] + i[f_{2n+1}]$. Because the discrete Fourier transform is a linear operator, we have that

$$\forall k \in 0..N-1: \quad \hat{u}_k = \hat{v}_k + i\hat{w}_k. \tag{7.24}$$

On the other hand, the symmetry expressed by Equation 7.23 and applied to the real N-dimensional vectors $\vec{v}$ and $\vec{w}$ implies that

$$\forall k \in 0..N-1: \quad \overline{\hat{u}_{N-k}} = \hat{v}_k - i\hat{w}_k. \tag{7.25}$$

From Equations 7.24 and 7.25, it follows that

$$\forall k \in 0..N-1: \quad \begin{cases} \hat{v}_k &= (\hat{u}_k + \overline{\hat{u}_{N-k}})/2 \\ \hat{w}_k &= (\hat{u}_k - \overline{\hat{u}_{N-k}})/2i. \end{cases} \tag{7.26}$$

The relation between $\tilde{f}_k$, $\hat{v}_k$, and $\hat{w}_k$ follows from the definition of discrete Fourier transforms of the M-dimensional vector $\vec{f}$ and the N-dimensional vectors $\vec{v}$ and $\vec{w}$. With

$$H = \frac{2\pi}{M} = \frac{2\pi}{2N} = \frac{h}{2},$$

we find that

$$\begin{cases} & \tilde{f}_0 &= \hat{v}_0 + \hat{w}_0 \\ \forall k \in 1..N-1: & \tilde{f}_k &= \hat{v}_k + e^{ikH}\hat{w}_k \\ & \tilde{f}_N &= \hat{v}_0 - \hat{w}_0. \end{cases}$$

Combined with Equation 7.26, this leads to the relation between the discrete Fourier transforms of the real vector $\vec{f}$ and the complex vector $\vec{u}$:

$$\left\{ \begin{array}{lll} & \tilde{f}_0 + i\tilde{f}_N & = (1+i)\overline{\hat{u}_0} \\ \forall k \in 1..N-1: & \tilde{f}_k & = \frac{1}{2}\left(\hat{u}_k + \overline{\hat{u}_{N-k}} - ie^{ikH}(\hat{u}_k - \overline{\hat{u}_{N-k}})\right). \end{array} \right.$$

These equations imply program Fast-Real-Fourier.

program Fast-Real-Fourier
declare
  $m,\ t$: integer ;
  ϵ : array$[0..2^{n-1}-1]$ of complex ;
  b : array$[0..2^n-1]$ of complex
initially
  $\langle\ ;\ t\ :\ 0 \le t < 2^{n-1}\ ::\ \epsilon[t] = e^{it\frac{2\pi}{2^n}}\ \rangle$;
  $\langle\ ;\ t\ :\ 0 \le t < 2^n\ ::\ b[\rho_n(t)] = f_{2t} + if_{2t+1}\ \rangle$
assign
  $\langle\ ;\ m\ :\ 0 \le m < n\ ::$
    $\langle\ \|\ t\ :\ t \in 0..2^n-1$ **and** $t \wedge 2^m = 0\ ::$
      $\begin{bmatrix} b[t] \\ b[t+2^m] \end{bmatrix} := \begin{bmatrix} 1 & \epsilon[2^{n-m-1}t \bmod 2^{n-1}] \\ 1 & -\epsilon[2^{n-m-1}t \bmod 2^{n-1}] \end{bmatrix} \begin{bmatrix} b[t] \\ b[t+2^m] \end{bmatrix}$
    $\rangle$
  $\rangle$;
  $\langle\ \|\ t\ :\ t \in 1..2^{n-1}-1\ ::$
    $b[t] := (b[t] + \overline{b[N-t]} - ie^{itH}(b[t] - \overline{b[N-t]}))/2\ \|$
    $b[N-t] := (b[N-t] + \overline{b[t]} + ie^{-itH}(b[N-t] - \overline{b[t]}))/2$
  $\rangle$;
  $b[0] := (1+i)\overline{b[0]}$
end

As indicated by the **initially** section, the values $u_t = f_{2t} + if_{2t+1}$, where $0 \le t < N$, are stored in the array b according to the bit-reversal map $\rho_n(t)$. After the complex transform, the array b contains the values of the discrete Fourier coefficients in natural order. Subsequently, the real-transform coefficients $\tilde{f}_k$ are computed in the same array b. To do this in place, the assignments to $b[t]$ and $b[N-t]$ must be done simultaneously.

7.5.4 *The Discrete Fourier-Sine Transform*

The Fourier series of an odd function $f(x)$ reduces to a Fourier-sine series. More precisely, if $f(x)$ is real, periodic with period 2π, and odd, Equation 7.1 becomes

$$f(x) = \sum_{k=1}^{+\infty} \check{f}_k \sin(kx).$$

The Fourier-sine coefficients $\check{f}_k$ in this expansion are given by:

$$\check{f}_k = \frac{2}{\pi}\int_0^\pi f(x)\sin(kx)dx.$$

Discretize the function $f(x)$ on an equidistant grid of the interval $[0,\pi]$. Let $H = \pi/M$ and $f_j = f(jH)$ for all $j \in 0..M$. Because $f(x)$ is periodic and odd, we have that $f(0) = f(\pi) = 0$ and, consequently, that $f_0 = f_M = 0$. The discretized function is, therefore, represented by the real $(M-1)$-dimensional vector $\vec{f} = [f_j]_{j=1}^{M-1}$. The following lemma, which is analogous to Lemma 17, establishes the discrete Fourier-sine transform.

Lemma 20 *The set of vectors* $\{\vec{e}_m = [\sin(jmH)]_{j=1}^{M-1} : 0 < m < M\}$, *with* $H = \pi/M$, *is an orthogonal basis for the space* $\mathbb{R}^{M-1}$.

(Without proof.) □

Every vector of $\mathbb{R}^{M-1}$ has a unique representation as a linear combination of the proposed basis vectors. Using that

$$\forall m \in 1..M-1:\ (\vec{e}_m, \vec{e}_m) = M/2,$$

the following identities are easily obtained:

$$\forall m \in 1..M-1: \qquad f_m = \sum_{k=1}^{M-1} \check{f}_k \sin(mkH) \tag{7.27}$$

$$\forall k \in 1..M-1: \qquad \check{f}_k = \frac{2}{M}\sum_{m=1}^{M-1} f_m \sin(kmH). \tag{7.28}$$

A scalar factor aside, the discrete Fourier-sine transform is its own inverse.

An elementary calculation shows that the discrete Fourier-sine transform of $\vec{f} = [f_m]_{m=1}^{M-1}$ is related to the discrete Fourier transform of the real M-dimensional vector $\vec{y} = [y_m]_{m=0}^{M-1}$ with

$$\forall m \in 0..M-1:\ y_m = \frac{2}{M}\sin(mH)(f_m + f_{M-m}) + \frac{1}{M}(f_m - f_{M-m}).$$

Note that $y_0 = 0$. Because the vector $\vec{y}$ is real, program Fast-Real-Fourier can be used to compute the $N = M/2$ complex coefficients:

$$\hat{y}_0 + i\hat{y}_N, \hat{y}_1, \ldots, \hat{y}_{N-1},$$

where

$$\forall k \in 0..N:\ \hat{y}_k = \sum_{m=0}^{M-1} y_m e^{ikmh},$$

with $h = 2\pi/M = 2H$. Note that we use the tilde notation for the discrete Fourier-sine transform of real $(M-1)$-dimensional vectors and the hat

notation for the discrete Fourier transform of real M-dimensional vectors. The discrete Fourier-sine coefficients $\tilde{f}_k$ are related to the discrete Fourier coefficients $\hat{y}_k$ according to

$$\forall k \in 1..N-1: \ \hat{y}_k = \tilde{f}_{2k+1} - \tilde{f}_{2k-1} + i\tilde{f}_{2k} \tag{7.29}$$

and

$$\hat{y}_0 + i\hat{y}_N = 2(\tilde{f}_1 - i\tilde{f}_{M-1}) \tag{7.30}$$

The imaginary parts of the coefficients $\hat{y}_k$ determine all discrete Fourier-sine coefficients with even indices. Those with odd indices are computed by means of a simple recursion relation, which is derived from the real parts of Equation 7.29 and is initialized by either the real or the imaginary part of Equation 7.30. In a multicomputer program, this recursion could be computed by full recursive doubling.

7.5.5 The Discrete Fourier-Cosine Transform

If the function $f(x)$ is real, periodic with period 2π, and even, Equation 7.1 reduces to the Fourier-cosine series expansion

$$f(x) = \frac{1}{2}\check{f}_0 + \sum_{k=1}^{+\infty} \check{f}_k \cos(kx),$$

with coefficients given by

$$\forall k \geq 0: \ \check{f}_k = \frac{2}{\pi}\int_0^{\pi} f(x)\cos(kx)dx.$$

To derive the discrete Fourier-cosine transform, discretize the function $f(x)$ on an equidistant grid of the interval $[0, 2\pi]$. Let $H = 2\pi/(2M)$ and $f_j = f(jH)$ for all $j \in 0..2M-1$. Because $f(x)$ is even, $f(x) = f(2\pi - x)$ and, consequently, $f_j = f_{2M-j}$ for all $j \in 0..2M-1$. The following lemma, which is analogous to Lemmas 17 and 20, gives us an orthogonal basis for the subspace of even vectors of $\mathbb{R}^{2M}$.

Lemma 21 *The set of vectors* $\{\vec{e}_m = [\cos(jmH)]_{j=0}^{2M-1} : 0 \leq m \leq M\}$, *with* $H = 2\pi/(2M)$, *is an orthogonal basis for the* $(M+1)$*-dimensional subspace of even vectors of* $\mathbb{R}^{2M}$.

(Without proof.) □

Every even vector $\vec{f}$ in $\mathbb{R}^{2M}$ has a unique representation as a linear combination of the proposed basis vectors. As before, the coefficient of $\vec{e}_k$ in this linear combination is given by

$$\frac{(\vec{e}_k, \vec{f})}{(\vec{e}_k, \vec{e}_k)}.$$

An elementary calculation shows that $(\vec{e}_k, \vec{e}_k) = M$ for $k \neq 0$ and that $(\vec{e}_0, \vec{e}_0) = 2M$. We focus, therefore, on the numerators and compute

$$\forall k \in 0..M: \ \tilde{f}_k = (\vec{e}_k, \vec{f}) = \sum_{m=0}^{2M-1} f_m \cos(kmH).$$

Because the vector $\vec{f}$ is even, it is easily derived that

$$\forall k \in 0..M: \ \tilde{f}_k = f_0 + 2\sum_{j=0}^{M-1} f_j \cos(jkH) + (-1)^k f_M. \tag{7.31}$$

The $2M$-dimensional vector $\vec{f}$ has only $M+1$ degrees of freedom, say the components $f_0, f_1, \ldots, f_M$. In this sense, we may refer to the discrete Fourier-cosine transformation of $\vec{f}$ as a transformation of a real $(M+1)$-dimensional vector.

The discrete Fourier-cosine transform of a real $(M+1)$-dimensional vector can be computed via the discrete Fourier transform of an auxiliary real M-dimensional vector $\vec{y} \in \mathbb{R}^M$ with components given by:

$$\forall m \in 0..M-1: \ y_m = \frac{1}{2}(f_m + f_{M-m}) - \sin(mh)(f_m - f_{M-m}).$$

An elementary calculation shows that

$$\forall k \in 0..N: \ 2\hat{y}_k = \tilde{f}_{2k} + i(\tilde{f}_{2k+1} - \tilde{f}_{2k-1}), \tag{7.32}$$

where $N = M/2$. The tilde notation is used for the discrete Fourier-cosine transform of real $(M+1)$-dimensional vectors and the hat notation for the discrete Fourier transform of real M-dimensional vectors.

As seen in Section 7.5.3, Program Fast-Real-Fourier computes the N complex coefficients

$$\hat{y}_0 + i\hat{y}_N, \hat{y}_1, \ldots, \hat{y}_{N-1}.$$

It follows from Equation 7.32 that their real parts trivially determine the Fourier-cosine coefficients with even indices. (This holds for all but $\tilde{f}_M$, which is determined by the imaginary part of $\hat{y}_0 + i\hat{y}_N$.) To compute the discrete Fourier-cosine coefficients with odd indices, a recursion involving the imaginary part of the discrete Fourier coefficients is required. This recursion must be initialized with $\tilde{f}_1$, which is computed directly from Equation 7.31.

7.5.6 Multivariate Discrete Fourier Transforms

Multivariate Fourier transforms of functions $f(x, y)$ over a domain $\Omega \subset \mathbb{R}^2$ are defined by successively applying univariate Fourier transforms. For example, let $\Omega = [0, 2\pi) \times [0, \pi]$, $f(x, y)$ periodic in x with period 2π, and $f(x, 0) = f(x, \pi) = 0$.

For every fixed y, the function $f(x,y)$ is a periodic function of x with period 2π, which can be expanded into a Fourier series. It follows that

$$f(x,y) = \sum_{k=-\infty}^{+\infty} \tilde{f}_k(y) e^{ikx}.$$

Because $f(x,y)$ vanishes for $y = 0$ and $y = \pi$, we obtain that

$$\forall k \in \mathbb{Z}: \ \tilde{f}_k(0) = \tilde{f}_k(\pi) = 0.$$

Every $\tilde{f}_k(y)$ can be extended into an odd function with period 2π and can, therefore, be expanded into a Fourier-sine series. The multivariate Fourier expansion

$$f(x,y) = \sum_{k=-\infty}^{+\infty} \sum_{\ell=1}^{\infty} \hat{f}_{k,\ell} e^{ikx} \sin(\ell y) \tag{7.33}$$

is obtained.

Consider the discrete analog of the above continuous example. The function $f(x,y)$ is represented by function values on a regular rectangular grid defined by the grid spacings $h_x = 2\pi/M$ and $h_y = \pi/N$:

$$\forall (m,n) \in 0..M-1 \times 1..N-1: \ f_{m,n} = f(mh_x, nh_y).$$

To obtain the discrete transform corresponding to Equation 7.33, one must apply transforms in the x- and in the y-direction. In the x-direction, the discrete Fourier transform is applied to $N-1$ real vectors of dimension M. In the y-direction, the discrete Fourier-sine transform is applied to M vectors of dimension $N-1$. The bit-reversal map and the packing of real values into complex arrays, one or both of which may have occurred in the first step, complicate the index arithmetic of the second transformation.

Another complication is that the univariate fast-Fourier-transform programs need some reorganization to perform optimally in a multivariate context. Consider program Multivariate-Fourier, which applies $N-1$ discrete Fourier transforms in the x-direction and $M-1$ discrete Fourier-sine transforms in the y-direction.

```
program Multivariate-Fourier
declare
    m, n, t : integer ;
    f : array[0..M − 1 × 1..N − 1] of real
assign
    ⟨ ‖ n : 0 < n < N :: Real-Fourier([f[m,n]]_{m=0}^{M−1}) ⟩ ;
    ⟨ ‖ m : 0 ≤ m < M :: Fourier-Sine([f[m,n]]_{n=1}^{N−1}) ⟩
end
```


Program Real-Fourier performs discrete real Fourier transforms in the x-direction and program Fourier-Sine performs discrete Fourier-sine transforms in the y-direction. Both programs are left unspecified. The array passed to program Real-Fourier consists of array entries $f[m,n]$ with constant second index, while the array passed to program Fourier-Sine consists of array entries $f[m,n]$ with constant first index. For programs Real-Fourier and Fourier-Sine, successive array entries are, therefore, not necessarily stored contiguously in memory. In fact, the stride between successive array entries should be an argument to these programs, because it is unpredictable in which direction one might want to apply which transform. Avoiding the stride problem by copying the entries to a temporary array is not feasible, particularly if the following optimization is also incorporated.

Inside program Real-Fourier, there are several quantifications that involve the first index of array f. For every value of the second index, identical index arithmetic for the first index occurs. Substantial savings can be accomplished by moving the quantification over n inside the fast-Fourier-transform quantifications used by program Real-Fourier. (From a program-correctness point of view, the position of the concurrent quantification over n is arbitrary.) This program transformation also increases performance for two other reasons besides reducing index arithmetic. First, communication of individual transforms can be grouped into combined messages to amortize latency over many transforms. Second, this program transformation allows for pipelining. Consider two successive instances of the quantification over n, say $n = n_0$ and $n = n_0 + 1$. The first instance uses the variables $\{f[m,n_0] \ : \ 0 \le m < M\}$, while the second uses $\{f[m,n_0+1] \ : \ 0 \le m < M\}$. The variables $f[m,n_0+1]$ and $f[m,n_0]$ are separated in memory by a certain stride that is independent of the value of n_0. Quantifications involving data separated by constant strides can be implemented efficiently on many pipelined processors; see Section 12.1.

For the same reasons, it makes sense to transform program Fourier-Sine such that its innermost quantification is the concurrent quantification over m.

The multicomputer implementation requires a data distribution over a $P \times Q$ process mesh. The process-mesh dimensions, P and Q, and the dimensions of the computational grid, M and N, must be powers of two. It is then possible to introduce perfectly load-balanced data distributions for both indices m and n and to obtain a multicomputer multivariate transform based on the above multicomputer univariate transforms.

Another approach obtains a multivariate transform for multicomputers by using only sequential univariate transforms. If either P or Q is one, all univariate transforms in one direction can be computed sequentially. Before computing the univariate transforms in the other direction, a grid transpose is performed, after which the univariate transforms in the other direction can be computed sequentially.

It is possible, even easy, to write down a naive program for the multicomputer grid-transpose operation and to analyze it under our standard performance assumptions. The resulting execution time is overly optimistic, however, because the naive implementation almost always overloads the capacity of the communication network if applied to grids of a size occurring in typical supercomputing applications. Under conditions of network contention, our simple communication-performance assumptions do not hold, and they underestimate the true cost of the operation. A usable implementation of the grid-transpose operation requires some computer-dependent programming to avoid network contention. A more detailed discussion is found in Section 12.3.3, where a realistic estimate for the communication time required by the grid-transpose operation is obtained.

Both approaches to multicomputer multivariate Fourier transform require the same number of floating-point operations. It is, therefore, sufficient to compare the communication times of both approaches. To this end, assume R nodes are available to compute a problem on an $M \times M$ grid with R processes.

When using the one-dimensional data distribution, all the communication occurs during the grid transpose. The result obtained in Section 12.3.3 allows us to estimate the communication time by

$$\tau_C(M^2/R)\log_2 R.$$

The two-dimensional data distribution uses a $P \times Q$ process mesh with $PQ = R$. From Equation 7.22, it follows that the concurrent fast Fourier transform of a vector of length M over P processes requires a communication time of $\tau_C(M/P)\log_2 P$. However, M such transforms are done simultaneously in Q process columns, and one message can carry communicated values of N/Q transforms, thereby avoiding latency. The fast Fourier transform in the x-direction requires a communication time of

$$\tau_C((M/Q)(M/P))\log_2 P = \tau_C(M^2/R)\log_2 P.$$

Adding to this the communication time for the transform in the y-direction, we obtain that the communication time for the two computational approaches is identical.

The main disadvantage of one-dimensional data distributions is that fine-granularity effects become important as soon as $R \approx M$, instead of $R \approx M^2$ for two-dimensional data distributions. It also must be stressed that the data distribution is often a given and cannot be chosen: the multivariate Fourier transform is just one part of a larger computation, which may impose a two-dimensional data distribution.

7.6 Notes and References

The fast Fourier transform was introduced by Cooley and Tukey [19]. Subsequent developments addressed issues of packing and unpacking of the vector components; the algorithm of Temperton [80], for example, is both self-sorting and in-place. The first concurrent algorithm was proposed by Pease [72]. There exist many other fast-Fourier-transform algorithms, which differ in the details of data organization and order of computation. Van Loan [84] gives a detailed overview of fast-Fourier-transform algorithms.

Writing a fast-Fourier-transform package remains a challenge for any writer of software libraries, because the context in which the package is used has such an important impact on the details of the computation. Consider, for example, how the multivariate transforms of Section 7.5.6 lead to considerable changes of the univariate transforms. These difficulties are amplified by the large number of different, but related, transforms. Any package that claims to be complete must incorporate these related transforms. Moreover, forward and backward transformations use different bit-reversal maps, and all transformations should work for a variety of data distributions. A considerable task indeed!

Exercises

Exercise 30 *Prove that the map*

$$\tau(m, r, k) = r + \rho_n(k) \tag{7.34}$$

also satisfies Equation 7.18. Moreover, this map results in a program that computes the Fourier coefficients in bit-reversed order, starting from a vector stored in natural order.

Exercise 31 *Develop a forward-transform program using the bit-reversal map τ of Equation 7.34.*

Exercise 32 *Develop a multicomputer fast Fourier transform that leaves the vector components of both transformed and nontransformed vectors in natural order. Use extra memory if necessary.*

Exercise 33 *Develop forward and backward transforms for real vectors. Particularly, take care to implement the packing and unpacking of the real data into the complex arrays such that no extra memory is required. Also develop the multicomputer version.*

Exercise 34 *Why is the discrete Fourier-cosine transform different from the real part of the discrete Fourier transform? Why is the discrete Fourier-sine transform different from the imaginary part of a discrete Fourier transform?*

Exercise 35 *Apply the divide-and-conquer technique directly to multivariate Fourier transforms by splitting multivariate trigonometric polynomials into its odd-even parts. Consider, for example, the case of a transformation over the two variables x and y. With $M = 2K$, $N = 2L$, and*

$$p(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f_{m,n} e^{i(mx+ny)},$$

we have that

$$\begin{aligned} p(x,y) &= \sum_{k=0}^{K-1} \sum_{\ell=0}^{L-1} f_{2k,2\ell} e^{i2(kx+\ell y)} \\ &+ e^{ix} \sum_{k=0}^{K-1} \sum_{\ell=0}^{L-1} f_{2k+1,2\ell} e^{i2(kx+\ell y)} \\ &+ e^{iy} \sum_{k=0}^{K-1} \sum_{\ell=0}^{L-1} f_{2k,2\ell+1} e^{i2(kx+\ell y)} \\ &+ e^{i(x+y)} \sum_{k=0}^{K-1} \sum_{\ell=0}^{L-1} f_{2k+1,2\ell+1} e^{i2(kx+\ell y)} \end{aligned}$$

or

$$p(x,y) = q_{0,0}(2x,2y) + e^{ix} q_{1,0}(2x,2y) + e^{iy} q_{0,1}(2x,2y) + e^{i(x+y)} q_{1,1}(2x,2y).$$

Use this identity as the basis for a divide-and-conquer strategy for multivariate Fourier transforms. Develop the corresponding specification and multicomputer programs.

8
Poisson Solvers

In this chapter, some elementary solution methods for the Poisson equation on a rectangular domain will be introduced. The algorithms used in the implementation of these elementary methods are fundamental building blocks in the construction of programs that solve other, more general, partial-differential equations.

8.1 The Poisson Problem

The *Poisson partial-differential equation* on a domain Ω in $\mathbb{R}^2$ is given by

$$\forall (x,y) \in \Omega : \; -\Delta u = f(x,y). \tag{8.1}$$

The partial-differential operator Δ is called the *Laplace operator* and is defined on twice-differentiable functions $u(x,y)$ by

$$\Delta u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

We shall consider only the case of a two-dimensional rectangular domain $\Omega = (a,b) \times (c,d)$.