6-1-1999

Two Algorithms for Performing Multidimensional, Multiprocessor, Out-of-Core FFTs

Lauren M. Baptist
Dartmouth College

Dartmouth College Computer Science Technical Report PCS-TR99-350

Two Algorithms for Performing Multidimensional, Multiprocessor, Out-of-Core FFTs

Lauren M. Baptist
Dartmouth College
Department of Computer Science

Advisor: Thomas H. Cormen

Abstract

We describe two algorithms for computing multidimensional Fast Fourier Transforms (FFTs) on a multiprocessor system with distributed memory when problem sizes are so large that the data do not fit in the memory of the entire system. Instead, data reside on a parallel disk system and are brought into memory in sections. We use the Parallel Disk Model for implementation and analysis.

The first method is a straightforward out-of-core variant of a well-known method for in-core, multidimensional FFTs. It performs 1-dimensional FFT computations on each dimension in turn. This method is easy to generalize to any number of dimensions, and it also readily permits the individual dimensions to be of any sizes that are integer powers of 2. The key step is an out-of-core transpose operation that places the data along each dimension into contiguous positions on the parallel disk system so that the data for the 1-dimensional FFTs are contiguous.

The second method is an adaptation of another well-known method for in-core, multidimensional FFTs. This method computes all dimensions simultaneously. It is more difficult to generalize to arbitrary numbers of dimensions and aspect ratios (even when all dimensions are integer powers of 2) in this method than in the first method. Our present implementation is therefore limited to two dimensions of equal size, that are again integer powers of 2.

We present I/O complexity analyses for both methods as well as empirical results for a DEC 2100 server and an SGI Origin 2000, each of which has a parallel disk system. Our results indicate that the methods are comparable in speed in two dimensions.

Contents

Chapter 1

Introduction

Signal processing, seismic analysis, acoustics, and quantum physics are just a few of the fields in which scientists apply the Fast Fourier Transform (FFT). In fact, over half a million people are running FFTs on their home computers as part of a project called SETI@home, launched in May of 1999 by the Search for Extraterrestrial Intelligence Institute. Hence, many astronomers, engineers, geophysicists, and even extraterrestrial enthusiasts rely upon efficient FFT implementations, which happens to be the subject of this thesis.

Although the data requirements of many such FFT computations are small enough that the data fit in main memory, there are some situations in which the data requirements exceed the memory capacity of even very large systems. The typical way of dealing with such "out-of-core" situations is to have the data reside on a disk system (preferably parallel) and to transfer sections of the data to and from memory. Previous work [CN97, CN98, Cor99, CWN97, VS94] has shown how to perform out-of-core, one-dimensional FFTs on both uniprocessors and multiprocessors when the data reside on a parallel disk system. The Parallel Disk Model, or PDM, originally defined by Vitter and Shriver [VS94], provided both a theoretical and implementation model in the previous work.

Chapter 2 reflects a tangential course this research took. While studying the mathematics behind the FFT computation, we became interested in the issue of accuracy, and we included the results of that inquiry in this thesis. Most of this thesis, however, is a discussion of two multidimensional, out-of-core FFT algorithms for a multiprocessor. We begin by revisiting the *dimensional method* discussed in [BC99]. We extend our previous work by including the full proofs of the algorithm's I/O complexity. We then introduce the *vector-radix algorithm* for computing multidimensional, out-of-core FFTs on a multiprocessor, and we prove this algorithm's I/O complexity, as well. Finally, we compare the two algorithms using empirical data, and we conclude that in two dimensions, they are comparable in speed.

In the remaining sections of this chapter, we will provide the reader with some background on multidimensional FFTs, the PDM, and BMMC permutations.

1.1 Multidimensional FFTs

The FFT is a particular method of computing the the *Discrete Fourier Transform (DFT)* of an array with a total of N elements. We assume in this thesis that the array has k dimensions $N_1, N_2, \ldots, N_k$, where $N = N_1 N_2 \cdots N_k$, and that each dimension is an integer power of 2. We are given a k-dimensional array $A[0 : N_1 - 1, 0 : N_2 - 1, \ldots, 0 : N_k - 1]$, and we wish to compute the k-dimensional array $Y[0 : N_1 - 1, 0 : N_2 - 1, \ldots, 0 : N_k - 1]$ for which

$$Y[\beta_1, \beta_2, \ldots, \beta_k] = \sum_{\alpha_1=0}^{N_1-1} \sum_{\alpha_2=0}^{N_2-1} \cdots \sum_{\alpha_k=0}^{N_k-1} \omega_{N_1}^{\beta_1\alpha_1} \omega_{N_2}^{\beta_2\alpha_2} \cdots \omega_{N_k}^{\beta_k\alpha_k} A[\alpha_1, \alpha_2, \ldots, \alpha_k] \ ,$$

where $\omega_{N_j} = \exp(-2\pi i/N_j)$ and $i = \sqrt{-1}$. In FFT computations, powers of ω_{N_j} are often referred to as *twiddle factors*. If we need to, for any real number u, we can directly compute $\exp(iu) = \cos(u) - i\sin(u)$.

Most multidimensional FFT problems fit in memory, but the few that do not have traditionally been extremely time-consuming to compute, due to high disk-access latencies and the blocked nature of data layout. One specific out-of-core, multidimensional FFT application is authentication of digital audio recordings and photographs. According to H. Farid [Far99], "When a signal is passed through a non-linearity it tends to create 'un-natural' higher-order correlations between the harmonics. The power spectrum (second-order statistics) is blind to such correlations, so we employ the bispectrum to detect the presence of these correlations." Multidimensional FFTs are used in bispectral analysis. Farid also reports, "We hope to eventually look at even higher-order statistics." Crystallography is believed to be another source of very large, multidimensional FFT problems.

1.2 The Parallel Disk Model

This section describes the Parallel Disk Model [VS94], which underlies our out-of-core algorithms.

In the *Parallel Disk Model*, or *PDM*, N records are stored on D *disks* $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$, with N/D *records* stored on each disk. For our purposes, a record is a complex number comprised of two 8-byte double-precision floats. The records on each disk are partitioned into *blocks* of B records each. Any disk access transfers an entire block of records. Disk I/O transfers records between the disks and an M-record *memory*.

We assess an algorithm by the number of parallel I/O operations it requires. Each *parallel I/O operation* transfers up to D blocks between the disks and memory, with at most one block transferred per disk, for a total of up to BD records transferred. The blocks transferred in a given parallel I/O operation may or may not be at the same relative locations on their respective disks.

For a multiprocessor, we assume that there are P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ connected by a network. Network speeds vary greatly, but for the multiprocessors that we consider, interprocessor communication times are far less than I/O latencies. The M-record memory is distributed among the P processors so that each processor holds M/P records. In ViC*, our implementation of the PDM, we assume that $D \geq P$, and each processor $\mathcal{P}_i$ communicates

	$\mathcal{P}_0$				$\mathcal{P}_1$				$\mathcal{P}_2$				$\mathcal{P}_3$			
	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
stripe 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
stripe 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stripe 2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
stripe 3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Figure 1.1: The layout of $N = 64$ records in a parallel disk system with $P = 4$, $B = 2$, and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

only with the D/P disks $\mathcal{D}_{iD/P}, \mathcal{D}_{iD/P+1}, \ldots, \mathcal{D}_{(i+1)D/P-1}$. (If $D < P$ in a given physical configuration, the ViC* implementation provides the illusion that $D = P$ by sharing each physical disk among P/D processors.)

We place some restrictions on the PDM parameters. We assume that P, B, D, M, and N are exact powers of 2. For convenience, we define $p = \lg P$, $b = \lg B$, $d = \lg D$, $m = \lg M$, and $n = \lg N$. We assume that $BD \leq M$ so that the memory can hold the contents of one block from each disk, and we assume that $B \leq M/P$ so that each processor's memory can hold the contents of one block. Finally, we assume that $M < N$ so that the problem is out-of-core.

The PDM lays out data on a parallel disk system as shown in Figure 1.1. A *stripe* consists of the D blocks at the same location on all D disks. A record's index is an n-bit vector. Later on, we will take advantage of interpreting a record index as a sequence of bit fields that give the record's location in the parallel disk system; from most significant bits to least significant bits, the bit fields are

- $\lg(N/BD) = n - (b + d)$ bits containing the number of the stripe (since each stripe has BD records, there are N/BD stripes),
- $\lg D = d$ bits containing the disk number; of these, the most significant $\lg P = p$ contain the processor number,
- $\lg B = b$ bits containing the record's offset within its block.

Since each parallel I/O operation accesses at most BD records, any algorithm that must access all N records requires $\Omega(N/BD)$ parallel I/Os, and so $O(N/BD)$ parallel I/Os is the analogue of linear time in sequential computing. A *pass* consists of reading each record once, doing some computation, and writing it back out, with a cost of $2N/BD$ parallel I/O operations. Vitter and Shriver showed an asymptotically tight bound of $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$ parallel I/Os for the FFT, which appears to be the analogue of the $\Theta(N \lg N)$ bound seen for so many sequential algorithms on the standard RAM model.

Practical considerations

Although there is no theoretical restriction on the parameters N and M beyond the out-of-core requirement of $M < N$, in practice we expect that $N \leq M^2$ or, equivalently, $M \geq \sqrt{N}$. We show why by an example. Consider a 1-terapoint problem, so that $N = 2^{40}$. (We

know of no current application that requires FFTs that large.) At 16 bytes per point, we would need 16 terabytes of disk storage just to hold the data. (In fact with our FFT algorithms, we would need an additional 8 terabytes to hold temporary data, but we will ignore this extra amount.) Now suppose that $M < \sqrt{N}$, in which case we would have $M < 2^{20}$. In other words, the memory cannot hold 1 megapoint, or 16 megabytes. It is safe to say that any computer installation that can afford to buy 16 terabytes of disk capacity would have computers easily capable of holding 16 megabytes of data. (As of this writing, in early 1999, it is virtually impossible to buy even a PC with under 16 megabytes of RAM.) Even if we allow for additional uses of memory beyond holding the FFT data (code, stack, communication buffers, I/O buffers, etc.), the aggregate memory of a system with 16 terabytes of on-line disk storage will always be large enough that $M \geq \sqrt{N}$.

1.3 BMMC permutations

This section defines the class of BMMC permutations and gives their I/O complexity on the PDM. We then describe a technique used in Sections 3.2 and 4.3 to prove the I/O complexity of BMMC permutations in the two multidimensional, multiprocessor FFT algorithms discussed in this thesis. Finally, we describe and sketch the specific types of BMMC permutations we will use in the two algorithms.

Definition and I/O complexity

A *BMMC* (bit-matrix-multiply/complement) permutation on $N = 2^n$ elements is specified by an $n \times n$ *characteristic matrix* $H = (h_{ij})$ whose entries are drawn from $\{0, 1\}$ and that is nonsingular (i.e., invertible) over $GF(2)$.[1] Treating each source index x as an n-bit vector, we perform matrix-vector multiplication over $GF(2)$ to form the corresponding n-bit target index z: $z = H\,x$. As long as the characteristic matrix H is nonsingular, the mapping of source indices to target indices is one-to-one.

A useful property of BMMC permutations is that they are closed under composition. In particular, if we were to apply, in order, BMMC permutations with characteristic matrices $A_1, A_2, \ldots, A_k$, the composite permutation could be achieved by performing a single BMMC permutation whose characteristic matrix is the product $A_k A_{k-1} \cdots A_2 A_1$.

An efficient algorithm for BMMC permutations on the PDM appears in [CSW99]. This algorithm requires at most $\frac{2N}{BD}\left(\left\lceil \frac{\operatorname{rank}\phi}{\lg(M/B)} \right\rceil + 1\right)$ parallel I/Os, where ϕ is the lower left $\lg(N/M) \times \lg M$ submatrix of the characteristic matrix, and where the rank is computed over $GF(2)$. (Note that because of the dimensions of ϕ, its rank is at most $\lg \min(M, N/M)$.) This number of factors is asymptotically optimal and is very close to the best known exact lower bound.

We simplify our I/O complexity calculations in two ways. First, we will simply count passes, where each pass is $2N/BD$ parallel I/Os. Second, we will use the lowercase letters,

[1] Matrix multiplication over $GF(2)$ is like standard matrix multiplication over the reals but with all arithmetic performed modulo 2. Equivalently, multiplication is replaced by logical-and, and addition is replaced by exclusive-or. Technically, the specification of a BMMC permutation also includes a "complement vector" of length n, but we will not need complement vectors in this thesis.

which denote logarithms of uppercase letters. With these conventions, we can restate the I/O complexity of a BMMC permutation as $\left\lceil \frac{\operatorname{rank} \phi}{m-b} \right\rceil + 1$ passes, where ϕ is the lower left $(n-m) \times m$ submatrix.

Proof technique for complexity analyses

In our complexity analyses we use a technique that exploits the properties of matrix multiplication. Our first observation is that to find the rank of the lower left $(n-m) \times m$ submatrix of a characteristic matrix A, it suffices to find the rank of the $(n-m) \times m$ matrix product $\Pi_1 \, X \, A \, Y \, \Pi_2$, where the matrices X and Y have the forms

$$X = \begin{array}{c} \begin{array}{cc} \scriptstyle m & \scriptstyle n-m \end{array} \\ \left[\begin{array}{c|c} 0 & I \end{array}\right] \end{array} \begin{array}{l} \\ \scriptstyle n-m \end{array} \quad \text{and} \quad Y = \begin{array}{c} \scriptstyle m \\ \left[\begin{array}{c} I \\ \hline 0 \end{array}\right] \end{array} \begin{array}{l} \scriptstyle m \\ \scriptstyle n-m \end{array} ,$$

Π_1 is any $(n-m) \times (n-m)$ *permutation matrix* (exactly one 1 in each row and in each column), and Π_2 is any $m \times m$ permutation matrix.

We may view X and Y as row and column selection matrices respectively. $X \, A$ selects the last $(n-m)$ rows of A, and $A \, Y$ selects the leftmost m columns of A. Thus, the matrix product $X \, A \, Y$ is the lower left $\lg(N/M) \times \lg M$ submatrix of A. Clearly then, computing the rank of $X \, A \, Y$ is equivalent to computing the rank of the lower left $\lg(N/M) \times \lg M$ submatrix of A. We also note that we can group the factors Π_1, X, A, Y, and Π_2, as well as the factors that comprise the matrix A, in any convenient fashion.

Finally, it is often useful to view the matrix product $E \, F$, where E and F are permutation matrices, either as a row permutation of F or as a column permutation of E. Hence, $\Pi_1 \, X \, A \, Y \, \Pi_2$ is both a row permutation of $X \, A \, Y \, \Pi_2$ and a column permutation of $\Pi_1 \, X \, A \, Y$. Clearly, permuting the rows of $X \, A \, Y \, \Pi_2$ or the columns of $\Pi_1 \, X \, A \, Y$ does not affect rank, and so we can compose the product $X \, A \, Y$ with the permutation matrices Π_1 and Π_2 without affecting our result.

BMMC permutations in our two multidimensional algorithms

We shall use several types of BMMC permutations to perform multidimensional, multiprocessor FFTs. Each is from the even more restricted class of *bit permutations*, in which the characteristic matrix is a permutation matrix. In other words, each target index is formed by permuting the bits of its corresponding source index.

n_j-partial bit-reversal permutation: In a full bit-reversal permutation, the characteristic matrix has 1s on the antidiagonal and 0s elsewhere. In our multidimensional FFT algorithm, we will reverse only the least significant n_j bits at a time, where n_j is the logarithm of the size of the current dimension, j. Letting I denote an identity submatrix and I^{A} denote a submatrix with 1s on the antidiagonal, and indicating submatrix dimensions along the top and sides, the characteristic matrix for an n_j-partial bit-reversal permutation looks like

$$\begin{array}{c} \begin{array}{cc} n_j & n-n_j \end{array} \\ \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & I \end{array}\right] \begin{array}{l} n_j \\ n-n_j \end{array} \end{array} .$$

two-dimensional bit-reversal permutation: We rotate the characteristic matrix of a full bit-reversal by $n/2$ bits to form the characteristic matrix of a two-dimensional bit-reversal permutation, so that it is of the form

$$\begin{array}{c} \begin{array}{cc} \frac{n}{2} & \frac{n}{2} \end{array} \\ \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & I^{\mathrm{A}} \end{array}\right] \begin{array}{l} \frac{n}{2} \\ \frac{n}{2} \end{array} \end{array} .$$

n_j-bit right-rotation: We rotate the bits of each index by n_j bits to the right, wrapping around at the rightmost position. The characteristic matrix is formed by taking the identity matrix and rotating its columns n_j positions to the right, so that it looks like

$$\begin{array}{c} \begin{array}{cc} n_j & n-n_j \end{array} \\ \left[\begin{array}{c|c} 0 & I \\ \hline I & 0 \end{array}\right] \begin{array}{l} n-n_j \\ n_j \end{array} \end{array} .$$

$(n-m+p)/2$-partial bit-rotation and vice versa: In an $(n-m+p)/2$-partial bit-rotation, we rotate only the most significant $n-(m+p)/2$ bits of each index by $(n-m+p)/2$ bits to the right, while the least significant $(m-p)/2$ bits of each index remain fixed. We thus have the characteristic matrix

$$\begin{array}{c} \begin{array}{ccc} \frac{m-p}{2} & \frac{n-m+p}{2} & \frac{n}{2} \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{m-p}{2} \\ \frac{n}{2} \\ \frac{n-m+p}{2} \end{array} \end{array} .$$

The inverse permutation has the characteristic matrix

$$\begin{array}{c} \begin{array}{ccc} \frac{m-p}{2} & \frac{n}{2} & \frac{n-m+p}{2} \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{m-p}{2} \\ \frac{n-m+p}{2} \\ \frac{n}{2} \end{array} \end{array} .$$

two-dimensional t-bit right-rotation and vice versa: We rotate the least significant $n/2$ bits by t bits to the right, and we do the same to the most significant $n/2$ bits. The characteristic matrix therefore includes two submatrices that are both similar to the n_j-bit right-rotation characteristic matrix:

$$\begin{array}{c} \begin{array}{cccc} t & \frac{n}{2}-t & t & \frac{n}{2}-t \end{array} \\ \left[\begin{array}{c|c|c|c} 0 & I & 0 & 0 \\ \hline I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I \\ \hline 0 & 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{n}{2}-t \\ t \\ \frac{n}{2}-t \\ t \end{array} \end{array} .$$

The characteristic matrix of the inverse permutation is

$$\begin{array}{c} \begin{array}{cccc} \frac{n}{2}-t & t & \frac{n}{2}-t & t \end{array} \\ \left[\begin{array}{c|c|c|c} 0 & I & 0 & 0 \\ \hline I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I \\ \hline 0 & 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} t \\ \frac{n}{2}-t \\ t \\ \frac{n}{2}-t \end{array} .$$

Stripe-major to processor-major and vice-versa: It is much simpler to write FFT code when each processor can work on a contiguous subset of the array. In the usual PDM ordering of Figure 1.1, which we call *stripe-major layout*, each processor has only a small contiguous subset of the data, consisting of only BD/P points. *Processor-major layout* is the ordering in which processor $\mathcal{P}_f$ has the N/P consecutive points with indices fN/P to $(f+1)N/P - 1$. Reordering from stripe-major to processor-major and back is given by the following characteristic matrices, where $s = b + d$:

$$\begin{array}{c} \begin{array}{ccc} s-p & n-s & p \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} s-p \\ p \\ n-s \end{array} ,$$

stripe-major to processor-major

$$\begin{array}{c} \begin{array}{ccc} s-p & p & n-s \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} s-p \\ n-s \\ p \end{array} .$$

processor-major to stripe-major

1.4 Outline

The remaining chapters of the thesis are divided as follows:

Chapter 2: Twiddle Factor Computation. We address the issue of accuracy in out-of-core FFT implementations by looking at several methods for computing the twiddle factors needed in the FFT computation. We discuss both the in-core algorithms and our out-of-core adaptations. We conclude the chapter with empirical results.

Chapter 3: The Dimensional Method. We describe a method for computing multidimensional FFTs one dimension at a time, and we describe our out-of-core, multiprocessor adaptation. We conclude the chapter with an analysis of the I/O complexity of the out-of-core algorithm.

Chapter 4: The Vector-Radix Method. We discuss a second multidimensional FFT algorithm. This method computes all dimensions simultaneously. We describe the in-core method for computing two-dimensional FFTs, and we present our out-of-core, multiprocessor adaptation. As in Chapter 3, we conclude with a complexity analysis.

Chapter 5: Empirical Results. We compare the dimensional and vector-radix methods by presenting empirical results for implementations of both methods. The implementation of the PDM we use is ViC* [CH97], which has been ported to several platforms. We include results for two systems that have parallel disks: a DEC 2100 server and a Silicon Graphics Origin 2000.

Chapter 6: Conclusion. We present some concluding remarks.

Chapter 2

Twiddle Factor Computation

Recall that in an FFT computation, we are given an array $A[0 : N-1]$, from which we must compute $Y[0 : N-1]$, where $Y[k] = \sum_{j=0}^{N-1} A[j]\omega_N^{jk}$ for $k = 0, 1, \ldots, N-1$ and $\omega_N = \exp(-2\pi i/N)$ and $i = \sqrt{-1}$. In FFT computations, powers of ω_N are often referred to as *twiddle factors*.[1] If we need to, for any real number u, we can directly compute $\exp(iu) = \cos(u) - i\sin(u)$. For our purposes, we shall define w_N to be the vector of the $N/2$ twiddle factors needed to compute an N-element FFT, such that $w_N[j] = \omega_N^j$.[2] At this point, we remind the reader that $n = \lg N$.

2.1 In-core algorithms

Van Loan [Van92] details six in-core algorithms for computing twiddle factors. He also includes a brief analysis of each algorithms's roundoff error. Each asymptotic upper bound of error accumulation is in terms of j (the position in w_N) and $\mathbf{u}$, which may be thought of as the "unit roundoff" of the machine. Thus, a system that uses t bits to hold the mantissa would have $\mathbf{u} \approx 2^{-t}$. We adapted several of these methods to our out-of-core environment in an attempt to find one that was both relatively fast and relatively accurate. We dismissed two of the methods, *Forward Recursion* and *Logarithmic Recursion*, because Van Loan's analysis indicated that they were even less accurate than *Repeated Multiplication*, the method in the existing out-of-core implementation [CWN97].[3] Nonetheless, we did implement Logarithmic Recursion, and the experimental error results we obtained from computing the FFT with Repeated Multiplication and Logarithmic Recursion will highlight the relative accuracy of the other methods we implemented: *Subvector Scaling*, *Recursive Bisection*, *Direct Call with Precomputation*, and *Direct Call without Precomputation*. The

[1] Some scientists define ω_N as $\exp(2\pi i/N)$, and others define it as we have defined it. The algorithms we present here hold for either definition of ω_N.

[2] Note that the Greek letter ω denotes an individual twiddle factor, whereas the roman letter w denotes a vector of twiddle factors.

[3] Forward Recursion and Logarithmic Recursion have roundoff errors of $O(\mathbf{u}(|c_1| + \sqrt{|c_1|^2+1})^j)$ and $O(\mathbf{u}(|c_1| + \sqrt{|c_1|^2+1})^{\log j})$ respectively, where $c_1 = \cos(2\pi/N)$.

description of the in-core algorithms, as well as the pseudocode, that follows is borrowed heavily from Van Loan's discussion.

Note that the following in-core algorithms are precomputation techniques, since they compute all of the twiddle factors needed for the FFT and store them in the vector w_N of twiddle factors. We shall see that precomputation is unnecessary in both Direct Call and Repeated Multiplication. Consequently, we modified both methods to exclude precomputation in two of our out-of-core implementations. (We also implemented Direct Call with precomputation.) The other three algorithms, however, depend upon the precomputation of the vector w_N, but in the out-of-core environment, it is impossible to store $N/2$ twiddle factors in memory. Section 2.2 addresses this issue and the subsequent adaptations we made for our out-of-core implementations.

Direct Call

The most straightforward algorithm is to directly compute each twiddle factor, as expressed by

for $j \leftarrow 0$ **to** $N/2 - 1$
    do $w[j] \leftarrow \cos(2\pi j/N) - i \sin(2\pi j/N)$

Although this is the most accurate method, it is also the slowest. Each twiddle factor computation requires two calls to the math library. For an N-element FFT, we compute $N/2 \lg N$ twiddle factors, which entails $N \lg N$ math calls in the Direct Call with Precomputation method. In the out-of-core setting, N is large, and therefore, making $N \lg N$ calls to the math library significantly decreases the overall speed. However, the roundoff error of $O(\mathbf{u})$ signifies that all inaccuracy is in the machine representation of the data.

Repeated Multiplication

As mentioned, the existing out-of-core FFT implementation [CWN97] used Repeated Multiplication to compute the twiddle factors. Clearly, $\omega_N^j = \omega_N^{j-1}\omega_N^1$, for $j = 1, 2, \ldots, N-1$. Therefore, the following algorithm computes the vector of twiddle factors:

$w[0] \leftarrow 1$
$\omega \leftarrow \cos(2\pi/N) - i \sin(2\pi/N)$
for $j \leftarrow 0$ **to** $N/2 - 1$
    do $w[j] \leftarrow \omega \cdot w[j-1]$

We need to compute only ω_N^0 and ω_N^1, and all of the other twiddle factor computations are simply complex multiplications, which require less time than calls to the math library. However, for the very reason that this method is fast, it is also inaccurate. With only two direct twiddle factor computations and repeated multiplications, the accumulation of error substantially reduces the overall accuracy of the FFT. According to Van Loan, the roundofferror is $O(\mathbf{u}j)$, which we shall show is relatively poor.

Subvector Scaling

At the heart of the method called Subvector Scaling is the trigonometric identity

$$w_N[2^{j-1} : 2^j - 1] = \omega_N^{2^{j-1}} w_N[0 : 2^{j-1} - 1]$$

for $j = 1, 2, .., n-1$. As an example, let $N = 16$. We can compute $w_{16}[2:3]$ by multiplying the vector $w_{16}[0:1]$ by ω_{16}^2, i.e.,

$$w_{16}[2:3] = \begin{bmatrix} \omega_{16}^2 \\ \omega_{16}^3 \end{bmatrix} = \omega_{16}^2 \begin{bmatrix} \omega_{16}^0 \\ \omega_{16}^1 \end{bmatrix} = \omega_{16}^2 w_{16}[0:1] \, .$$

We can then compute the next four twiddle factors in a simliar fashion:

$$w_{16}[4:7] = \begin{bmatrix} \omega_{16}^4 \\ \omega_{16}^5 \\ \omega_{16}^6 \\ \omega_{16}^7 \end{bmatrix} = \omega_{16}^4 \begin{bmatrix} \omega_{16}^0 \\ \omega_{16}^1 \\ \omega_{16}^2 \\ \omega_{16}^3 \end{bmatrix} = \omega_{16}^4 w_{16}[0:3] \, .$$

In this manner, we can compute the entire vector of twiddle factors with the pseudocode

$w[0] \leftarrow 1$
for $j \leftarrow 1$ **to** $n-1$
  do $\omega \leftarrow \cos(2^j \pi/N) - i \sin(2^j \pi/N)$
    $w_N[2^{j-1} : 2^j - 1] \leftarrow \omega \cdot w_N[0 : 2^{j-1} - 1]$

The roundoff error of Subvector Scaling is $O(\mathbf{u} \log j)$, which is significantly better than the $O(\mathbf{u}j)$ roundoff error of the Repeated Multiplication algorithm.

Recursive Bisection

The final method that we shall look at in detail is Recursive Bisection. We shall summarize the description of the method given in [Van92]. We begin with the following two trigonometric identities:

$$\begin{aligned} \cos(A+B) + \cos(A-B) &= 2\cos(A)\cos(B) \, , \\ \sin(A+B) + \sin(A-B) &= 2\sin(A)\cos(B) \, . \end{aligned}$$

Note that they can be rearranged in the following way:

$$\begin{aligned} \cos(A) &= \frac{1}{2\cos(B)}(\cos(A-B) + \cos(A+B)) \, , \\ \sin(A) &= \frac{1}{2\cos(B)}(\sin(A-B) + \sin(A+B)) \, . \end{aligned}$$

Therefore, if we have computed the cosine of B, the sine and cosine of $(A-B)$, and the sine and cosine of $(A+B)$, we can compute the sine and cosine of A. Thus, if we begin by computing all $w[j]$, where j is a power of 2, then we can recursively bisect each of those

intervals in $O(\lg N)$ stages. In an example similar to a figure in [Van92], we illustrate the computation of an FFT with an input size $N = 32$. We would fill in the 16-element twiddle factor vector w in $\lg N = 4$ stages as follows:

0	1	2		4				8								16
												12				
						6				10				14		
			3		5		7		9		11		13		15	

The pseudocode for this algorithm is:

$c[0] \leftarrow 1$
$s[0] \leftarrow 0$
for $k \leftarrow 0$ **to** $n - 1$
    do $p \leftarrow 2^k$
        $c[p] \leftarrow \cos(2\pi p/N)$
        $s[p] \leftarrow -\sin(2\pi p/N)$
for $\lambda \leftarrow 1$ **to** $n - 2$
    do $p \leftarrow 2^{n-\lambda-2}$
        $h \leftarrow 1/(2c[p])$
        for $k \leftarrow 0$ **to** $2^\lambda - 2$
            do $j \leftarrow (3 + 2k)p$
                $c[j] \leftarrow h(c[j - p] + c[j + p])$
                $s[j] \leftarrow h(s[j - p] + s[j + p])$
$w[0 : N/2 - 1] \leftarrow c[0 : N/2 - 1] + is[0 : N/2 - 1]$

The result of Van Loan's error analysis of this algorithm is identical to the result of his analysis of Subvector Scaling; both algorithms have a roundoff error of $O(\mathbf{u} \log j)$.

Summary

Before addressing our adaptation of the four competetive twiddle factor algorithms to the out-of-core environment, we summarize our discussion of the in-core algorithms. Figure 2.1 shows a table similar to one in [Van92]. Repeated Multiplication, the twiddle factor algorithm used in the FFT code with which we began, is substantially less accurate than the other three methods. Also note that Subvector Scaling and Recursive Bisection have the same asymptotic upper bound. In Section 2.3, we will show that, although the upper bounds may be the same, Recursive Bisection proved to be both more accurate and faster in our out-of-core implementation.

2.2 Out-of-core adaptations

When we adapted each of Van Loan's in-core algorithms to the out-of-core environment, some interesting questions arose. For in an in-core setting, the order in which the twiddle

Method	Roundoff in ω_N^j
Direct Call	$O(\mathbf{u})$
Repeated Multiplication	$O(\mathbf{u}j)$
Subvector Scaling	$O(\mathbf{u}\log j)$
Recursive Bisection	$O(\mathbf{u}\log j)$

Figure 2.1: Roundoff error in twiddle factor algorithms. This table depicts the asymptotic upper bound of error accumulation for each of the four algorithms in terms of j (the position in the twiddle factor vector) and $\mathbf{u}$, or the unit roundoff of the machine. Therefore, on a system with t-bit mantissas, $\mathbf{u} \approx 2^{-t}$. Clearly, Direct Call is the most accurate of the four algorithms, and Repeated Multiplication is the least accurate, with the two other algorithms sharing an upper bound in the middle.

factors are needed is straightforward. However, in the out-of-core realm, we move data around at various points during the computation so that we can operate on contiguous chunks of memory. Specifically, we move the data by rotating every index by m bits between superlevels and by $n \bmod m$ after the last superlevel. Therefore, it would be rare to need twiddle factors with consecutive exponents.

Also of note is the issue of precomputation. In the Repeated Multiplication method, storing the entire vector of twiddle factors is unnecessary, since computing the jth twiddle factor requires only the $(j-1)$st twiddle factor. On the other hand, Subvector Scaling and Logarithmic Recursion clearly rely on the existence of a vector of twiddle factors from which each new twiddle factor is computed. In an out-of-core setting, we need to maximize the amount of memory we allocate to hold the data. Therefore, we must be cautious about the amount of memory we set aside for the twiddle factor computation. Let us therefore keep this in mind as we explore out-of-core adaptations of the four methods.

In the following discussion we denote a vector of twiddle factors as:

$$\begin{bmatrix} \omega_r \\ \hline e_1 \\ e_2 \\ \cdot \\ \cdot \\ \cdot \\ e_k \end{bmatrix},$$

such that each element e_i in the vector actually represents $\omega_r^{e_i}$. We use this notation because we are primarily interested in the twiddle factor exponents, and because all of the elements in a twiddle-factor vector share the same root r.

Imagine a problem in which $n = 8$ and $m = 4$. We shall begin by isolating the twiddle factors needed in memoryload 0 of superlevel 1.[4] Let us call the vector of twiddle factors

[4] All indexing begins at 0.

that we need for superlevel 1's last level of butterflies w_1', and so

$$w_1' = \begin{bmatrix} \omega_{256} \\ \hline 0 \\ 16 \\ 32 \\ 48 \\ 64 \\ 80 \\ 96 \\ 112 \end{bmatrix}.$$

We can easily adapt all four of our in-core algorithms to compute this sequence of twiddle factors. Furthermore, let us expand the example by sketching out the twiddle factors all four levels of superlevel 1:

$$\begin{bmatrix} \omega_{32} & \omega_{64} & \omega_{128} & \omega_{256} \\ \hline 0 & 0 & 0 & 0 \\ 0 & 16 & 16 & 16 \\ 0 & 0 & 32 & 32 \\ 0 & 16 & 48 & 48 \\ 0 & 0 & 0 & 64 \\ 0 & 16 & 16 & 80 \\ 0 & 0 & 32 & 96 \\ 0 & 16 & 48 & 112 \end{bmatrix}.$$

We stated above that it is not difficult to compute w_1' by modifying one of the in-core algorithms we have discussed. In fact, it is not difficult to compute the twiddle factors in the first three levels. By the cancellation lemma [CLR90], $\omega_{dn}^{dk} = \omega_n^k$, and so every twiddle factor in the first three levels is in w_1'.

Thus, it might appear at first glance that the four algorithms can be modified easily for our purposes. However, if we look at the twiddle factors in all four levels of the computation in memoryload 1 of superlevel 1, we shall see that it is not that simple:

$$\begin{bmatrix} \omega_{32} & \omega_{64} & \omega_{128} & \omega_{256} \\ \hline 1 & 1 & 1 & 1 \\ 1 & 17 & 17 & 17 \\ 1 & 1 & 33 & 33 \\ 1 & 17 & 49 & 49 \\ 1 & 1 & 1 & 65 \\ 1 & 17 & 17 & 81 \\ 1 & 1 & 33 & 97 \\ 1 & 17 & 49 & 113 \end{bmatrix}.$$

In this case, we cannot apply the cancellation lemma to obtain all the twiddle factors in the first three levels from the twiddle factors in the final level. Instead, we have to precompute a vector of twiddle factors for *each* level in the memoryload. Earlier we mentioned the cost

of precomputation and the necessity to minimize the amount of memory we allocate to hold the twiddle factors. Thus, precomputing a vector for each level is certainly less than ideal.

Despite this apparent hindrance, we can increase the accuracy of our computation. We can compute w'_1 by using any of the four algorithms mentioned above, and all the other twiddle factors in memoryload 0 can be found in w'_1. Furthermore, every other twiddle factor in the computation is simply a scaling of a twiddle factor in w'_1. For example, if we multiply each element in w'_1 by ω^1_{256}, we have the last level of twiddle factors in memoryload 1:

$$\omega^1_{256} \cdot \begin{bmatrix} \omega_{256} \\ 0 \\ 16 \\ 32 \\ 48 \\ 64 \\ 80 \\ 96 \\ 112 \end{bmatrix} = \begin{bmatrix} \omega_{256} \\ 1 \\ 17 \\ 33 \\ 49 \\ 65 \\ 81 \\ 97 \\ 113 \end{bmatrix} .$$

Likewise, we can scale some of the elements in w'_1 by ω^1_{128} to obtain the wtiddle factors in memoryload 1, level 2:

$$\omega^1_{128} \cdot \begin{bmatrix} \omega_{256} \\ 0 \\ 32 \\ 64 \\ 96 \\ 0 \\ 32 \\ 64 \\ 96 \end{bmatrix} = \omega^1_{128} \cdot \begin{bmatrix} \omega_{128} \\ 0 \\ 16 \\ 32 \\ 48 \\ 0 \\ 16 \\ 32 \\ 48 \end{bmatrix} = \begin{bmatrix} \omega_{128} \\ 1 \\ 17 \\ 33 \\ 49 \\ 1 \\ 17 \\ 33 \\ 49 \end{bmatrix} .$$

If we were to proceed further along in this example, we would see that we can compute every twiddle factor in the computation by scaling it by a fixed factor dependent upon the superlevel, memoryload, and level within the memoryload. Therefore, we can use Subvector Scaling or Recursive Bisection to precompute w'_s for each superlevel s, and the relative accuracy of these algorithms will be marred only by a single scaling for each twiddle factor. In Section 2.3, we compare the speed and accuracy of these out-of-core adaptations.

2.3 Empirical results

We experimented with six twiddle-factor algorithms to determine which subroutine to include in our uniprocessor out-of-core FFT implementation. We spliced each of the twiddle-factor algorithms into the out-of-core implementation and then measured speed and accuracy on a DEC 2100 server.[5] We incorporated the precomputation technique discussed above to adapt the following three algorithms to the out-of-core environment:

[5] See Chapter 5 for a full description of the platform.

1. Logarithmic Recursion,
2. Subvector Scaling,
3. Recursive Bisection.

We implemented the Direct Call technique both with and without precomputation, and, accordingly, we refer to the two variants as:

4. Direct Call with Precomputation,
5. Direct Call without Precomputation.

As a point of comparison, we did not modify the twiddle factor computations in the existing out-of-core implementation on the uniprocessor. Therefore, this sixth algorithm does not incorporate precomputation:

6. Repeated Multiplication.

Accuracy

We determined accuracy by comparing the actual value of each point after computing the FFT with the target, or correct, value for the point, and we defined a point's *error* to be the absolute value of the difference between these two values. We also calculated the total number of points in each *error group*, or collection of points with errors of the same order of magnitude, and we refer to each error group by the order of magnitude that defines it. In our experiments, the error groups ranged from 2^{-34} to 2^{-44}.

Figures 2.2–2.5 show the accuracy results of the experiments. Each of the figures represents a suite of runs of the out-of-core FFT implementation on a uniprocessor. In a given figure, the memory size and problem size remain fixed, while we vary the method used to compute the twiddle factors. Therefore, with all the other factors held constant, each figure should allow us to compare the accuracy of the different methods. We include only error groups between 2^{-34} and 2^{-38}, because relatively few points fell into the other error groups.

In the runs of Figures 2.2–2.4, we fixed the memory size at $M = 2^{26}$ bytes, while varying the problem size N over the three graphs. Figure 2.2 shows the results for $N = 2^{25}$ points, Figure 2.3 shows the results for $N = 2^{26}$, and Figure 2.4 shows the results for $N = 2^{27}$.

Here we make a couple of observations about our experimental results by interpreting Figures 2.2–2.4. First, we note that large error groups on the left side of the graph are indicative of a relatively inaccurate twiddle-factor algorithm. For we would prefer error to be concentrated at, say, 2^{-36} rather than 2^{-34}. Therefore, Figures 2.2–2.4 indicate that Logarithmic Recursion and Repeated Multiplication are relatively inaccurate, because they are the only two algorithms with significant 2^{-34}-error groups. Furthermore, both Logarithmic Recursion and Repeated Multiplication have larger 2^{-35}-error groups than the other algorithms.

Of the four remaining algorithms, Direct Call without Precomputation is clearly the most accurate, as we would expect, because each twiddle factor is computed on demand via calls to the math library. The other three algorithms–Direct Call with Precomputation, Subvector Scaling, and Recursive Bisection–are roughly similar in accuracy. Direct Call with

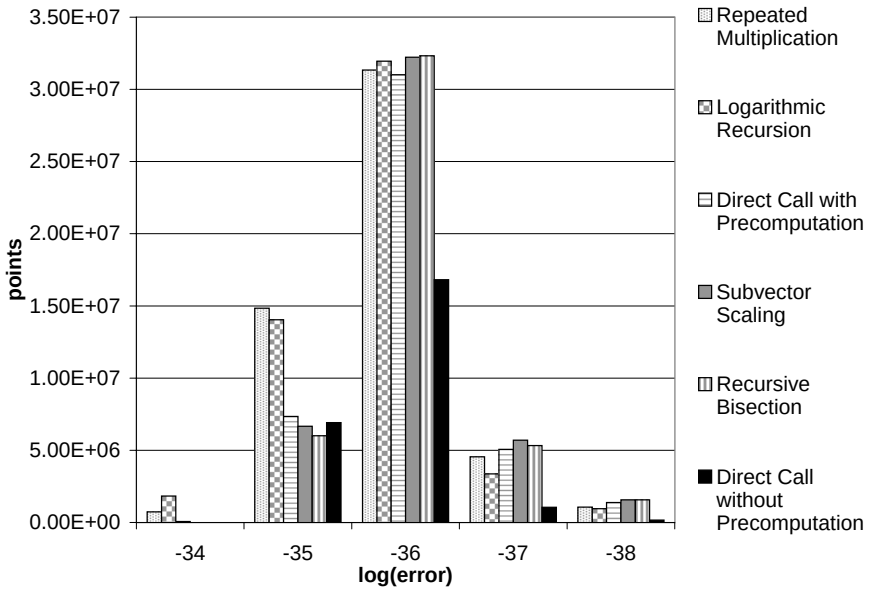

Figure 2.2: Accuracy of six twiddle-factor algorithms for $N = 2^{25}$ points and $M = 2^{26}$ bytes. The x-axis represents the order of magnitude of error in each error group, and the y-axis represents the total number of points in each error group.

Precomputation is slightly more accurate than Subvector Scaling and Recursive Bisection in some cases. However, we might have expected it to be significantly more accurate, because we directly compute the twiddle factors in the precomputed vector and scale them only once before using them in the computation. It is thus difficult to explain the behavior of Direct Call with Precomputation in these figures. As we shall see in Figure 2.5, the method's behavior is even worse with other values of N and M. Our only theory to explain this anomaly is that new error has the chance to inadvertently "undo" accumulated error in Subvector Scaling and Recursive Bisection, whereas Direct Call with Precomputation has no opportunity to reverse any error introduced in the one scaling operation.

As a point of comparison, we also include Figure 2.5, which shows the results for $N = 2^{25}$ points and $M = 2^{25}$ bytes. Once again, Repeated Multiplication and Direct Call without Precomputation represent the low and high poles of accuracy. As we just mentioned, Direct Call with Precomputation is particularly inaccurate in this run, as indicated by its large 2^{-35}-error group. Subvector Scaling and Recursive Bisection again seem comparable in accuracy, but as we shall see in a moment, they distinguish themselves from one another in speed.

Note that we exclude Logarithmic Recursion from Figure 2.5 and from the remaining figures in this chapter. As we mentioned in the previous section, the upper bound on error for the in-core Logarithmic Recursion algorithm is worse than the upper bounds on error for

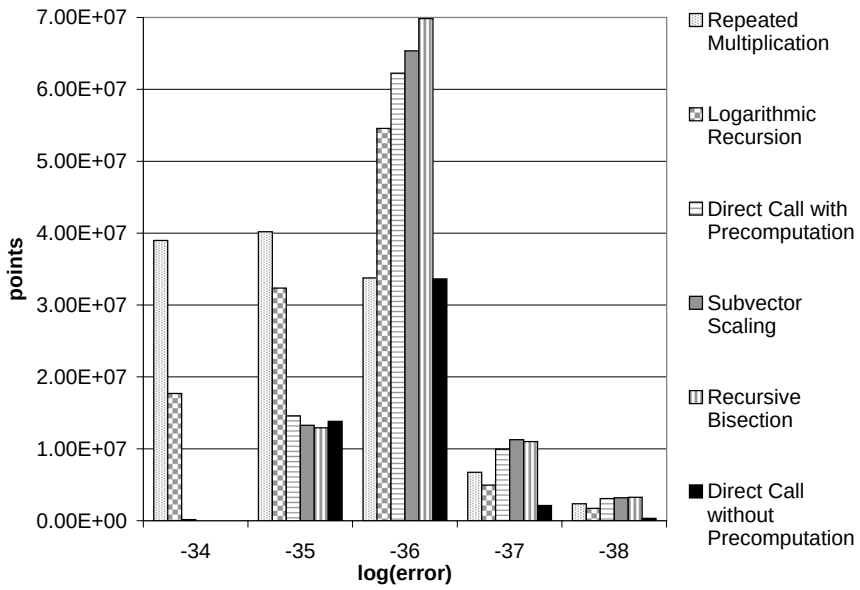

Figure 2.3: Accuracy of six twiddle-factor algorithms for $N = 2^{26}$ points and $M = 2^{26}$ bytes.

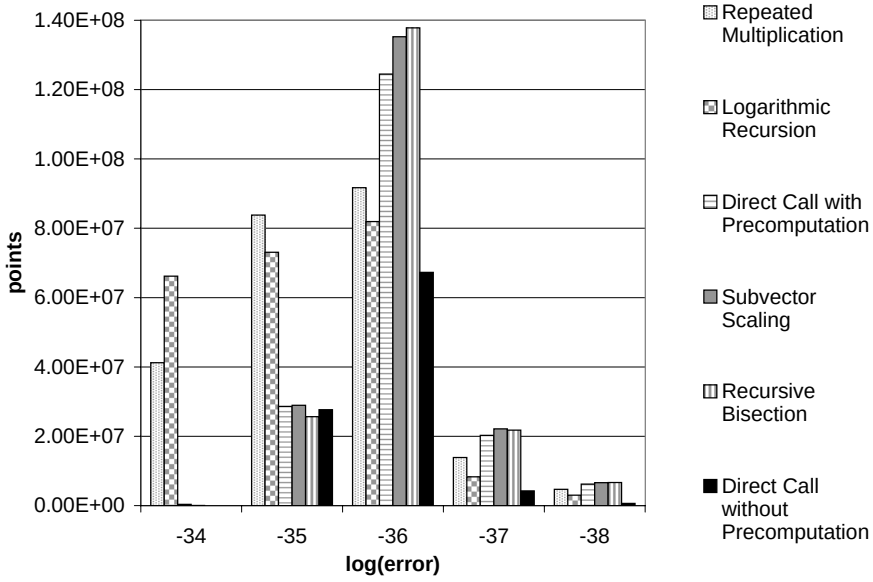

Figure 2.4: Accuracy of six twiddle-factor algorithms for $N = 2^{27}$ points and $M = 2^{26}$ bytes.

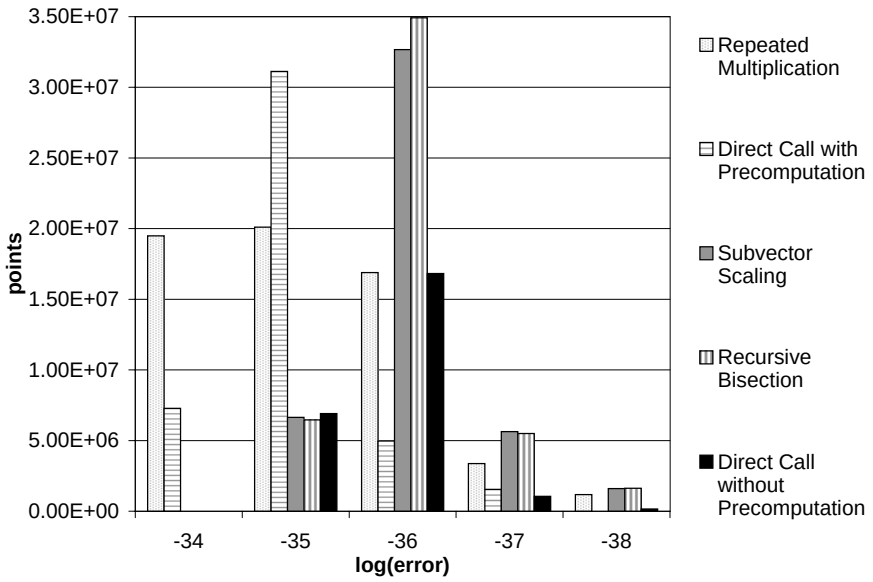

Figure 2.5: Accuracy of five twiddle-factor algorithms for $N = 2^{25}$ points and $M = 2^{25}$ bytes.

the other algorithms. Furthermore, the experimental runs shown in Figures 2.2–2.4 indicate that Logarithmic Recursion is relatively inaccurate in the out-of-core environment, as well.

Speed

We also timed each experimental run to determine the total FFT running times with each of the different twiddle-factor subroutines. Figures 2.6 and 2.7 display our results. Each figure represents a suite of experimental runs, for which we fixed the memory size M and varied the problem size N. First, we observe that our two poles in accuracy are also poles in speed. We saw earlier that Repeated Multiplication is relatively inaccurate, but Figures 2.6 and 2.7 indicate that it is fast. On the other hand, we saw that Direct Call without Precomputation is relatively accurate, but Figures 2.6 and 2.7 indicate that it is by far the slowest algorithm. In both figures, Recursive Bisection is roughly as fast as Repeated Multiplication, whereas Subvector Scaling, which was comparable to Recursive Bisection in accuracy is somewhat slower.

Conclusion

From the results of these experiments, we concluded that Recursive Bisection preserved the speed of Repeated Multiplication, while improving significantly upon its accuracy. Therefore, we modified both the uniprocessor and multiprocessor out-of-core FFT implementations to use this method of computing twiddle factors. As we shall see in Section 3.1, we

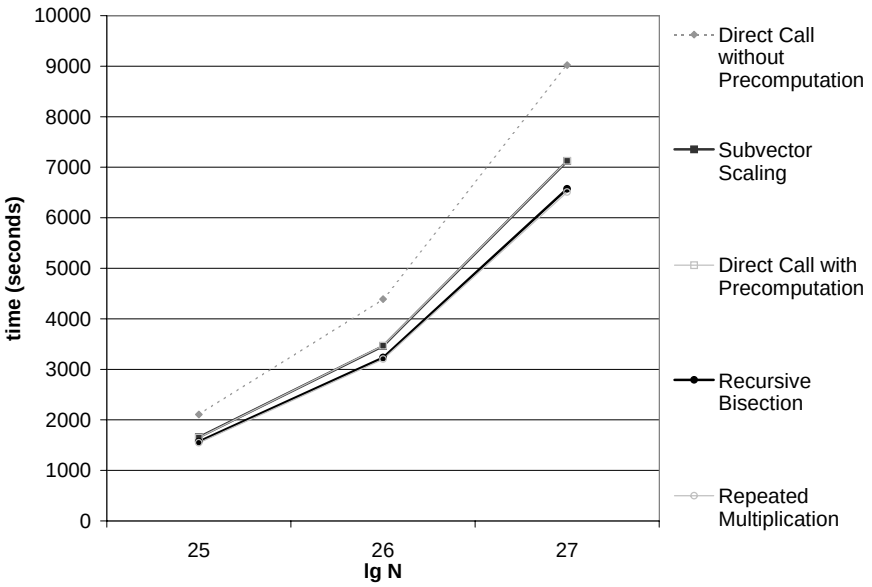

Figure 2.6: Total FFT running time with five twiddle-factor algorithms for $M = 2^{25}$ bytes. The x-axis represents the log of the problem size, and the y-axis represents the total running time in seconds. The running times of Subvector Scaling and Direct Call with Precomputation are almost identical, as are the running times of Repeated Multiplication and Recursive Bisection.

folded Recursive Bisection into our implementation of the dimensional method for computing the multidimensional FFT. In Section 4.2, we briefly discuss how we extended Recursive Bisection to incorporate it into our implementation of the vector-radix method for computing the two-dimensional FFT.

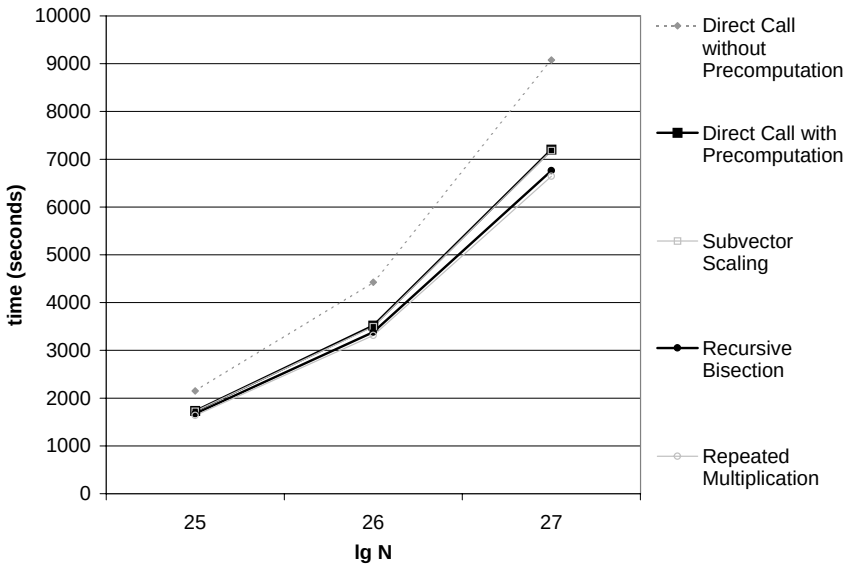

Figure 2.7: Total FFT running time with five twiddle-factor algorithms for $M = 2^{26}$ bytes. Once again, the running times of Subvector Scaling and Direct Call with Precomputation are almost identical, as are the running times of Repeated Multiplication and Recursive Bisection.

Chapter 3

The Dimensional Method

The dimensional method is arguably the simplest approach to computing the multidimensional out-of-core FFT, for it exploits a basic property of multidimensional FFTs: they may be computed by computing one-dimensional FFTs within each dimension in turn. That is, we compute

$$\begin{aligned}
Y^{(1)}[\beta_1, \beta_2, \ldots, \beta_k] &= \sum_{\alpha_1=0}^{N_1-1} \omega_{N_1}^{\beta_1\alpha_1} A[\alpha_1, \alpha_2, \ldots, \alpha_k] \ , \\
Y^{(2)}[\beta_1, \beta_2, \ldots, \beta_k] &= \sum_{\alpha_2=0}^{N_2-1} \omega_{N_2}^{\beta_2\alpha_2} Y^{(1)}[\alpha_1, \alpha_2, \ldots, \alpha_k] \ , \\
&\cdots \\
Y[\beta_1, \beta_2, \ldots, \beta_k] &= \sum_{\alpha_k=0}^{N_k-1} \omega_{N_k}^{\beta_k\alpha_k} Y^{(k-1)}[\alpha_1, \alpha_2, \ldots, \alpha_k] \ .
\end{aligned}$$

3.1 Out-of-core implementation on a multiprocessor

Assuming that the data are stored contiguously in the first dimension (e.g., a 2-dimensional array stored in row-major order with the first 1-dimensional FFTs being within each row), data accesses in the second and subsequent dimensions are not to consecutive memory locations. In an in-core setting, such an access pattern may slow the computation due to cache behavior. The penalty in an out-of-core setting may be more severe, however, as it could easily be the case that reading or writing each point entails a separate disk access. The resulting performance would be unacceptably poor. The obvious solution, which we adopt, is to reorder the data after operating in a given dimension to make the next dimension contiguous in the disk-resident ordering of the data. When all dimensions are powers of 2, this reordering is a BMMC permutation, which we can perform optimally in terms of I/O costs [CH97, CSW99].

We base our 1-dimensional FFT computations on the well-known Cooley-Tukey method, in which we perform a bit-reversal permutation followed by a sequence of "butterfly opera-

tions." We refer the reader to any of [CLR90, CN98, CT65, Van92] for details.

A key subroutine used by our implementation performs a BMMC permutation on the full N-point data set. This subroutine, based on techniques described in [CC99, CSW99], takes an $n \times n$ characteristic matrix (bit-packed into n words) as an input, and performs optimally the BMMC permutation so characterized.

An important issue in performing the 1-dimensional FFTs is whether or not each such FFT fits in the memory of a single processor. In other words, when performing FFTs in dimension j, is $N_j \leq M/P$? If so, then we have the possibility of performing the dimension-j FFTs in-core. Otherwise, we perform the dimension-j FFTs out-of-core. In either case, we start the dimension-j FFTs by performing an n_j-partial bit-reversal permutation, where $n_j = \lg N_j$. We next rearrange the data to put it into processor-major order by performing the stripe-major to processor-major BMMC permutation.

If the dimension-j FFTs fit in the memory of a single processor, we perform them in the obvious way. We repeatedly read in parallel into each processor's memory the data for the $(M/P)/N_j$ FFTs that the memory can hold, perform the necessary butterfly computations, and write the results back out to disk in parallel. This read-compute-write loop entails exactly one pass over the data.

Conversely, if the dimension-j FFTs do not fit in the memory of a single processor (i.e., $N_j > M/P$), then we perform the dimension-j FFTs out-of-core. As shown in [CWN97], we do so in a series of $\lceil n_j/(m-p) \rceil$ "superlevels," each of which entails one pass over the data followed by a BMMC permutation. In the remainder of this chapter, we do not consider the possibility that $N_j > M/P$, except to note that our implementation does handle it correctly.

After processing dimension j, we need to rearrange the data to get dimension $j+1$ into contiguous addresses in the PDM ordering. (After processing the last dimension, k, we must get dimension 1 into contiguous addresses so that our final result is in the correct order.) We do so by performing an n_j-bit right-rotation permutation. However, before we do so, we must rearrange the data into the canonical ordering—stripe-major—that the BMMC permutation code assumes. So we perform the processor-major to stripe-major BMMC permutation and follow it by the n_j-bit right-rotation permutation.

To recap, prior to computing the dimension-j butterfly operations, we first perform an n_j-bit partial bit-reveral permutation, followed by the stripe-major to processor-major BMMC permutation. Within each superlevel, we make one pass over the data to compute mini-butterflies. After computing the dimension-j butterfly operations, we first perform the processor-major to stripe-major BMMC permutation, followed by an n_j-bit right-rotation permutation.

Now we show how to take advantage of closure of BMMC permutations under composition. Let us denote the characteristic matrices of the individual BMMC permutations as follows:

- S characterizes the stripe-major to processor-major permutation.
- S^{-1} characterizes the processor-major to stripe-major permutation.
- V_j characterizes an n_j-bit partial bit-reversal permutation.
- R_j characterizes an n_j-bit right-rotation permutation.

The BMMC closure properties result in our performing the following permutations:

- Prior to computing the dimension-1 butterfly operations, we perform the BMMC permutation characterized by the matrix product $S\,V_1$.
- Between computing the dimension-j and dimension-$j+1$ butterfly operations, where $1 \le j < k$, we compose the permutations that follow dimension j with those that precede dimension $j+1$, performing the BMMC permutation characterized by the matrix product $S\,V_{j+1}\,R_j\,S^{-1}$.
- After computing the dimension-k butterfly operations, we perform the BMMC permutation characterized by the matrix product $R_k\,S^{-1}$.

It is easy to multiply these characteristic matrices together before presenting the product to the BMMC-permutation subroutine.

Implementation notes

Our implementation of the dimensional method for computing the multidimensional FFT is a fairly straightforward extension of previous work on 1-dimensional FFTs [CN98, CWN97]. We continue to call asynchronous (i.e., non-blocking) I/O functions, when the underlying system supports it, by allocating three buffers: for reading into, writing from, and computing in. One difference is that in the implementation of the dimensional method, we use the recursive bisection method for computing twiddle factors discussed in Chapter 2.

3.2 Analytical results

In this section, we analyze the I/O complexity of our out-of-core implementation of the dimensional method for a multiprocessor. Recall that the I/O complexity of a BMMC permutation is $\frac{2N}{BD}\left(\left\lceil \frac{\operatorname{rank}\phi}{\lg(M/B)} \right\rceil + 1\right)$ parallel I/Os, where ϕ is the lower left $\lg(N/M) \times \lg M$ submatrix of the characteristic matrix, and the rank is computed over $GF(2)$. We use the proof technique described in Section 1.3, and therefore we encourage the reader to refer to that exposition as necessary. We also remind the reader of the assumption we made in Section 3.1, that $N_j \le M/P$. We refer to this assumption several times in the analysis that follows. The following lemmas are essential to our analysis of the out-of-core implementation of the dimensional method on a multiprocessor.

Lemma 1 *For the matrix product $S\,V_1$, we have* $\operatorname{rank}\phi = \min(n-m, p)$.

Proof: From the discussion above, we know that to compute $\operatorname{rank}\phi$ of the matrix product $S\,V_1$, we can compute the rank of the $(n-m) \times m$ matrix product $Z = X\,S\,V_1\,Y\,\Pi$, where Π is a column permuation that we will define later. We also know that we can group these factors in any convenient fashion to compute the product, and so we will compute $E = X\,S$, $F = V_1\,Y$, and $G = F\,\Pi$. We can then compute the rank of $Z = E\,G$. We can represent the

grouping as

$$\underbrace{\underbrace{X}_{E}\,\underbrace{S\,\underbrace{V_1}_{F}\,Y}_{G}}_{Z}\,\Pi\,.$$

We begin by finding the subproduct $E = X\,S$, where the matrices X and S have the forms

$$X = \begin{array}{c} \begin{array}{cc} \scriptstyle m & \scriptstyle n-m \end{array} \\ \left[\begin{array}{c|c} 0 & I \end{array}\right] \end{array}_{n-m} \quad \text{and} \quad S = \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \begin{array}{l} s-p \\ p \\ n-s \end{array} .$$

If we view X as a row selection of S, then E is the lower $n-m$ rows of S. Under the PDM assumption that memory can hold the contents of one block from each disk, we have that $b + d = s \leq m$. Therefore, $n - m \leq n - s$. Thus,

$$E = \begin{array}{c} \begin{array}{ccc} \scriptstyle m-p & \scriptstyle n-m & \scriptstyle p \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array}_{n-m} .$$

We now find the product $F = V_1\,Y$, where the matrices V_1 and Y have the forms

$$V_1 = \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & I \end{array}\right] \begin{array}{l} n_1 \\ n-n_1 \end{array} \quad \text{and} \quad Y = \left[\begin{array}{c} I \\ \hline 0 \end{array}\right] \begin{array}{l} m \\ n-m \end{array} .$$

Previously, we made the key assumption that we could perform each dimension-j FFT in-core. In other words, $n_j \leq m - p$. Clearly then, $n_1 \leq m$. Therefore, if we view Y as a column selection of V_1, then we can see that F contains the leftmost n_1 columns of V_1, as well as the next $m - n_1$ columns. The resulting matrix is

$$F = \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & I \\ \hline 0 & 0 \end{array}\right] \begin{array}{l} n_1 \\ m-n_1 \\ n-m \end{array} .$$

We can permute the columns of F by composing it with the column permutation matrix

$$\Pi = \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & I \end{array}\right] \begin{array}{l} n_1 \\ m-n_1 \end{array}$$

to form

$$G = F\,\Pi = \left[\begin{array}{c} I \\ \hline 0 \end{array}\right] \begin{array}{l} m \\ n-m \end{array} .$$

We can now compute the rank of the final product $Z = E\,G$, where E is of the form

$$E = \begin{array}{c} \begin{array}{ccccc} \overset{m-p}{} & & \overset{n-m}{} & & \overset{p}{} \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \\ n-m \end{array} .$$

Let us view E as an $(n-m)$-row selection matrix. Suppose for a moment that $p = 0$. In this case, E selects only the $n-m$ zero rows of G, and the rank of Z is 0. Because $p = 0$, we can restate the rank as $\min(n-m, p)$. If, however, $p > 0$, then the band of $n-m$ rows that E selects from G moves upward to include p nonzero rows from the upper m nonzero rows of G, and so the rank of Z is p. E cannot select more than $n-m$ rows from G, however, so that $\operatorname{rank}\phi = \min(n-m, p)$. ■

Lemma 2 *For the matrix product $S\,V_{j+1}\,R_j\,S^{-1}$, we have*

$$\operatorname{rank}\phi = \min(n-m, n_j)\ .$$

Proof: In the analysis of this matrix product, we will compute $\operatorname{rank}\phi$ by computing the rank of the $(n-m)\times m$ matrix product $Z = X\,S\,V_{j+1}\,R_j\,S^{-1}\,Y\,\Pi_1\,\Pi_2$. (We will choose Π_1 and Π_2 later.) Once again, we group these factors to ease the computation. From the proof of Lemma 1, we already have the subproduct $E = X\,S$. We will now compute $F = E\,V_{j+1}$, $G = S^{-1}\,Y$, $H = G\,\Pi_1$, $J = R_j\,H$, and $K = J\,\Pi_2$. We can then compute the rank of the final composition $Z = F\,K$. A schematic representation of the grouping is

$$\underbrace{\underbrace{X}_{E}\,\underbrace{S}_{F}}\ V_{j+1}\ \underbrace{\underbrace{R_j}_{J}\ \underbrace{\underbrace{S^{-1}}_{G}\,\underbrace{Y}_{H}}_{K}\ \Pi_1}_{Z}\ \Pi_2\,.$$

We begin by finding the subproduct $F = E\,V_{j+1}$, where the matrices E and V_{j+1} have the forms

$$E = \begin{array}{c} \begin{array}{ccccc} \overset{m-p}{} & & \overset{n-m}{} & & \overset{p}{} \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \\ n-m \end{array} ,$$

$$V_{j+1} = \begin{array}{c} \begin{array}{cc} \overset{n_{j+1}}{} & \overset{n-n_{j+1}}{} \end{array} \\ \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} n_{j+1} \\ n-n_{j+1} \end{array} .$$

Let us view E as a row selection of V_{j+1}, in which we select $n-m$ rows that begin $m-p$ rows from the top of V_{j+1}. We again note that we have assumed that each dimension-j FFT fits in-core, and so $n_{j+1} \le m-p$. Therefore, the band of $n-m$ rows falls below the top n_{j+1} rows in V_{j+1}. Thus,

$$F = \begin{array}{c} \begin{array}{ccccc} \overset{m-p}{} & & \overset{n-m}{} & & \overset{p}{} \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \\ n-m \end{array} .$$

To form the matrix subproduct G, we compose S^{-1} and Y, which are of the forms

$$S^{-1} = \begin{array}{c} \begin{array}{ccc} \scriptstyle s-p & \scriptstyle p & \scriptstyle n-s \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \scriptstyle s-p \\ \scriptstyle n-s \\ \scriptstyle p \end{array} \quad \text{and} \quad Y = \begin{array}{c} \scriptstyle m \\ \left[\begin{array}{c} I \\ \hline 0 \end{array}\right] \end{array} \begin{array}{l} \scriptstyle m \\ \scriptstyle n-m \end{array} .$$

We view Y as a column selection of S^{-1}, so that in this composition, we are selecting the leftmost m columns of S^{-1}. Under the PDM assumption that memory can hold the contents of one block from each disk, we have that $s \leq m$. Therefore, we are selecting the leftmost s columns of S^{-1}, as well as the next $m - s$ columns. Hence, G is of the form

$$G = \begin{array}{c} \begin{array}{ccc} \scriptstyle s-p & \scriptstyle p & \scriptstyle m-s \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & 0 & 0 \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \scriptstyle s-p \\ \scriptstyle m-s \\ \scriptstyle n-m \\ \scriptstyle p \end{array} .$$

We now compute $H = G\,\Pi_1$, where Π_1 is the column permutation matrix

$$\Pi_1 = \begin{array}{c} \begin{array}{ccc} \scriptstyle s-p & \scriptstyle m-s & \scriptstyle p \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \scriptstyle s-p \\ \scriptstyle p \\ \scriptstyle m-s \end{array} .$$

By composing G with Π_1, we can combine G's identity submatrices, resulting in

$$H = \begin{array}{c} \begin{array}{cc} \scriptstyle m-p & \scriptstyle p \end{array} \\ \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} \scriptstyle m-p \\ \scriptstyle n-m \\ \scriptstyle p \end{array} .$$

The next subproduct that we will compute is $J = R_j\,H$, where R_j is of the form

$$R_j = \begin{array}{c} \begin{array}{cc} \scriptstyle n_j & \scriptstyle n-n_j \end{array} \\ \left[\begin{array}{c|c} 0 & I \\ \hline I & 0 \end{array}\right] \end{array} \begin{array}{l} \scriptstyle n-n_j \\ \scriptstyle n_j \end{array} .$$

We view H as a column selection of R_j, in which R_j's leftmost $m-p$ columns and rightmost p columns are selected. Because $n_j \leq m - p$, these leftmost $m - p$ columns include R_j's leftmost n_j columns, as well as the next $(m-p) - n_j$ columns. The resulting product is

$$J = \begin{array}{c} \begin{array}{ccc} \scriptstyle n_j & \scriptstyle m-p-n_j & \scriptstyle p \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & I \\ \hline I & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} \scriptstyle m-p-n_j \\ \scriptstyle n-m \\ \scriptstyle p \\ \scriptstyle n_j \end{array} .$$

After composing it with the column permutation matrix

$$\Pi_2 = \begin{array}{c} \begin{array}{ccc} m-p-n_j & p & n_j \end{array} \\ \left[\begin{array}{c|c|c} 0 & 0 & I \\ \hline I & 0 & 0 \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} n_j \\ m-p-n_j \\ p \end{array} ,$$

we have

$$K = J\,\Pi_2 = \begin{array}{c} \begin{array}{cc} m-p-n_j & n_j+p \end{array} \\ \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} m-p-n_j \\ n-m \\ n_j+p \end{array} .$$

We can now compute the rank of the final matrix product $Z = F\,K$. Recall that F is of the form

$$F = \begin{array}{c} \begin{array}{ccc} m-p & n-m & p \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} n-m .$$

Let us view F as an $(n-m)$-row selection matrix. Suppose for a moment that $n_j = 0$. In this case, F selects only the $n-m$ zero rows of K, and the rank of Z is 0. Because $n_j = 0$, we can restate the rank as $\min(n-m, n_j)$. If, however, $n_j > 0$, then the band of $n-m$ rows that F selects from K moves upward to include n_j nonzero rows from the upper $m-p-n_j$ nonzero rows of K, and so the rank of Z is n_j. F cannot select more than $n-m$ rows from K, however, and so in fact $\operatorname{rank}\phi = \min(n-m, n_j)$. ■

Lemma 3 *For the matrix product $R_k\,S^{-1}$, we have* $\operatorname{rank}\phi = \min(n-m, n_k+p)$.

Proof: To analyze the matrix product $R_k\,S^{-1}$, we will compute $\operatorname{rank}\phi$ by computing the rank of the $(n-m)\times m$ matrix product $Z = X\,R_k\,S^{-1}\,Y\,\Pi_1\,\Pi_2$, where Π_1 and Π_2 are column permutation matrices that we will choose later. Once again, we group these factors to ease the computation. From the proof of Lemma 2, we have the subproduct $K = R_j\,S^{-1}\,Y\,\Pi_1\,\Pi_2$. This intermediate result will be of great use to us in proving this lemma, once we make a simple modification. We must substitute n_k for n_j in the matrix subproduct K, because we are now concerned only with dimension k. Once we have modified this subproduct, we can compute the rank of the final matrix composition $Z = X\,K$. We represent this grouping schematically as

$$\underbrace{\underbrace{\underbrace{\underbrace{X\;R_j}_{H}\;\underbrace{S^{-1}\;Y}_{G}}_{J}\;\Pi_1}_{K}\;\Pi_2}_{Z} .$$

X and K are of the forms

$$X = \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \left[\begin{array}{c|c} 0 & I \end{array}\right] \end{array} n-m ,$$

$$K = \begin{array}{c} \begin{matrix} m-p-n_k & n_k+p \end{matrix} \\ \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{matrix} m-p-n_k \\ n-m \\ n_k+p \end{matrix} .$$

Let us view X as an $(n-m)$-row selection matrix. Suppose for a moment that $n_k + p = 0$. In this case, X selects only the $n-m$ zero rows of K, and the rank of Z is 0. Because $n_k + p = 0$, we can restate the rank as $\min(n-m, n_k+p)$. If, however, $n_k + p > 0$, then the band of $n-m$ rows that X selects from K moves upward to include $n_k + p$ nonzero rows from the upper $m-p-n_k$ nonzero rows of K, and so the rank of Z is $n_k + p$. X cannot select more than $n-m$ rows from K, however, so that $\operatorname{rank}\phi = \min(n-m, n_k+p)$. ■

Theorem 4 *Assuming that the k dimensions $N_1, N_2, \ldots, N_k$ are integer powers of 2 and that $N_j \le M/P$ for all $j = 1, 2, \ldots, k$, we can compute a multidimensional, multiprocessor, out-of-core FFT in*

$$\sum_{j=1}^{k-1} \left\lceil \frac{\min(n-m, n_j)}{m-b} \right\rceil + \left\lceil \frac{\min(n-m, n_k+p)}{m-b} \right\rceil + 2k + 2$$

passes, where lowercase letters denote logarithms of corresponding uppercase letters.

Proof: From Lemmas 1–3 and from recognizing that computing the butterfly operations entails one pass for each of the k dimensions, the number of passes to compute a multidimensional, multiprocessor, out-of-core FFT is

$$\left(\left\lceil \frac{\min(n-m, p)}{m-b} \right\rceil + 1 \right) + \sum_{j=1}^{k-1} \left(\left\lceil \frac{\min(n-m, n_j)}{m-b} \right\rceil + 1 \right) + \left(\left\lceil \frac{\min(n-m, n_k+p)}{m-b} \right\rceil + 1 \right) + k$$

$$= \left\lceil \frac{\min(n-m, p)}{m-b} \right\rceil + \sum_{j=1}^{k-1} \left\lceil \frac{\min(n-m, n_j)}{m-b} \right\rceil + \left\lceil \frac{\min(n-m, n_k+p)}{m-b} \right\rceil + 2k + 1 .$$

Observe that $\min(n-m, p) \le p$ and that, under the PDM's assumption that each processor has enough memory to contain one disk block, $\lceil p/(m-b) \rceil \le 1$. Therefore, we can rewrite the number of passes as

$$\sum_{j=1}^{k-1} \left\lceil \frac{\min(n-m, n_j)}{m-b} \right\rceil + \left\lceil \frac{\min(n-m, n_k+p)}{m-b} \right\rceil + 2k + 2 .$$

■

The following corollary restates this theorem in terms of parallel I/O operations and the actual PDM parameters:

Corollary 5 *Assuming that the k dimensions $N_1, N_2, \ldots, N_k$ are integer powers of 2 and that $N_j \leq M/P$ for all $j = 1, 2, \ldots, k$, we can compute a multidimensional, multiprocessor, out-of-core FFT in*

$$\frac{2N}{BD}\left(\sum_{j=1}^{k-1}\left\lceil\frac{\lg\min(N/M, N_j)}{\lg(M/B)}\right\rceil + \left\lceil\frac{\lg\min(N/M, N_kP)}{\lg(M/B)}\right\rceil + 2k + 2\right)$$

parallel I/O operations. ■

Chapter 4

The Vector-Radix Algorithm

In addition to the dimensional method, we implemented a second algorithm to compute multidimensional FFTs. In 1977, Rivard [Riv77] published the in-core vector-radix algorithm for two-dimensional square matrices. Later that year, Harris et al. [HMCS77] generalized it to include arbitrary radices and aspect ratios. These two publications are considered to be the defining works on the vector-radix algorithm. However, the expositions that Dudgeon and Mersereau [DM84] and Lim [Lim90] offer in their textbooks are perhaps more accessible. In this section, we will describe both the serial in-core vector-radix algorithm and the multiprocessor, out-of-core adaptation. We offer some intuition as to how this two-dimensional algorithm is an extension of the one-dimensional Cooley-Tukey method. For a formal proof of the vector-radix algorithm's correctness, we refer the reader to the aforementioned publications.

4.1 In-core algorithm

In the previous discussion, we considered the dimensional method to be an extension of the well-known Cooley-Tukey method of computing one-dimensional FFTs, because the dimensional method repeatedly invokes a subroutine to compute one-dimensional FFTs. The vector-radix method is also an extension of this one-dimensional algorithm, but in a different way. Both the Cooley-Tukey algorithm and the vector-radix method are divide-and-conquer algorithms. The Cooley-Tukey method exploits the principle that we can compute the one-dimensional DFT of N points by computing two $N/2$-point *sub-DFT*s and combining their results. Similarly, the vector-radix method exploits the principle that we can compute the two-dimensional FFT of an $\sqrt{N} \times \sqrt{N}$-point matrix by computing four $\sqrt{N}/2 \times \sqrt{N}/2$-point sub-DFTs and combining their results.

Let us now walk through the computation of a small two-dimensional FFT, in which the problem size is $N = 64$ points, or an 8×8-point matrix. The first step in the vector-radix algorithm is a two-dimensional bit-reversal, which is a straightforward extension of the one-dimensional bit-reversal that begins the Cooley-Tukey algorithm for computing the one-dimensional FFT.

Divide and conquer (or Butterflies: An introduction)

Following the two-dimensional bit-reversal, we divide the problem into successively smaller subproblems until each one is a 2×2-point submatrix. We then perform a series of *butterfly operations* to repeatedly combine the sub-DFTs into increasingly larger sub-DFTs until we have "conquered" the full $\sqrt{N} \times \sqrt{N}$-point problem. Figures 4.1– 4.3 show our 8×8-point example, in which the "combine" step just described unfolds as follows:

Level 0: Perform a butterfly operation on the four points in each of the sixteen 2×2-point submatrices to compute sixteen 2×2-point sub-DFTs.

Level 1: Perform butterfly operations on groups of four points from the sixteen level-0 sub-DFTs to compute four 4×4-point sub-DFTs.

Level 2: Perform butterfly operations on groups of four points from the four level-1 sub-DFTs to compute the full 8×8-point DFT.

For now, we will concentrate solely on understanding which groups of four points interact in butterfly operations in each of the three levels of the 8×8-point example. In the next subsection, we will discuss the way in which the points interact by describing how to perform 2×2-point butterflies.

Figure 4.1 shows level 0 of the 8×8-point example.[1] The bold lines demarcate the sixteen sub-DFTs in level 0, and the shaded region indicates the four points that interact in a butterfly in one of these sub-DFTs.

Figure 4.2 represents level 1 of the 8×8-point example. The heaviest bold lines demarcate the four 4×4-point sub-DFTs in level 1, and the medium bold lines demarcate the four level-0 sub-DFTs in each level-1 sub-DFT. To compute each level-1 sub-DFT, we combine four level-0 sub-DFTs by performing four butterfly operations, where each butterfly involves a different point from each of the four level-0 sub-DFTs. Specifically, the $[i, j]$th points from each of the four level-0 sub-DFTs interact in a butterfly. Similarly shaded points in Figure 4.2 represent each group of four elements that interact in a butterfly in one of the level-1 sub-DFTs.

Once we have computed these four level-1 sub-DFTs, we can combine them in level 2 to compute the DFT of the full 8×8-point problem. Figure 4.3 depicts level 2 of the computation, and the bold lines demarcate the four level-1 sub-DFTs. Once again, each point interacts in a butterfly with three points from different sub-DFTs: point $[i, j]$ of each level-1 sub-DFT interacts with the $[i, j]$th points of the other three level-1 sub-DFTs. The shading in Figure 4.3 indicates all of the four-point groups that interact in butterflies during this final level of the computation.

A *butterfly graph* is an alternate way to represent the butterfly computations just described. Moveover, in Section 4.2, we will describe the out-of-core vector-radix computation in terms of the butterfly graph. Figure 4.4 shows the 3-level butterfly network for the 64-point example. To reduce clutter, we include only one butterfly computation from each level of the graph, and we use shading to represent each sub-DFT. We begin with 64 input lines on the left side of the graph. In level 0 of the computation, we compute sixteen 4-point

[1] In this figure and all related figures in this chapter, the point $A[0, 0]$ is in the lower left corner.

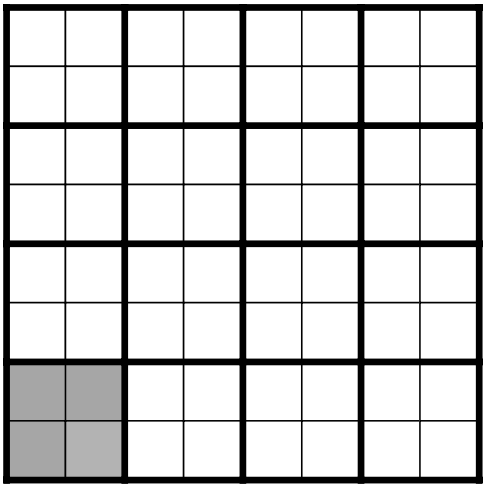

Figure 4.1: Level-0 butterfly groups for $N = 64$. Bold lines demarcate the sixteen level-0 sub-DFTs. The shaded region represents the single butterfly operation in one of the 2×2-point sub-DFTs.

sub-DFTs. In level 1, we combine the results from the level-0 sub-DFTs to compute four 16-point sub-DFTs. In the final level of the graph, we combine the four level-1 sub-DFTs to solve the full 64-point problem. Note the relationship between the butterfly graph and Figures 4.1–4.3. Also, observe that the depth of this graph is $3 = \log_4 64$. In general, the depth of the butterfly graph for a two-dimensional problem with N points is $\log_4 N$.

Butterflies: The details

We showed in the 8×8-point example which points interact at each level of the computation. Let us now examine the way in which these points interact. In other words, we define the 2×2-point butterfly operations. Suppose that we wish to compute a level-k sub-DFT, where $k = \lg K$, and the dimensions of the sub-DFT are $2K \times 2K$. (In level 1 of the 8×8-point problem, for example, $k = 1$ and $K = 2$, and so we compute sub-DFTs of $2K \times 2K$, or 4×4, points.) We know from the previous discussion that we can combine four of the level-$(k-1)$ sub-DFTs to compute one level-k sub-DFT. From Figures 4.1–4.3, we also know that the four points in each butterfly demarcate a square, with corners K positions apart. For convenience, let us therefore refer to each point with the notation $A[x_i, y_j]$, for $i, j = 1, 2$, where x_i and y_j represent the point's position within the level-k sub-DFT and where $x_2 = x_1 + K$ and $y_2 = y_1 + K$. We also observe that $0 \le x_1, y_1 < K$ and $K \le x_2, y_2 < 2K$. The three factors that define each butterfly operation in a level-k

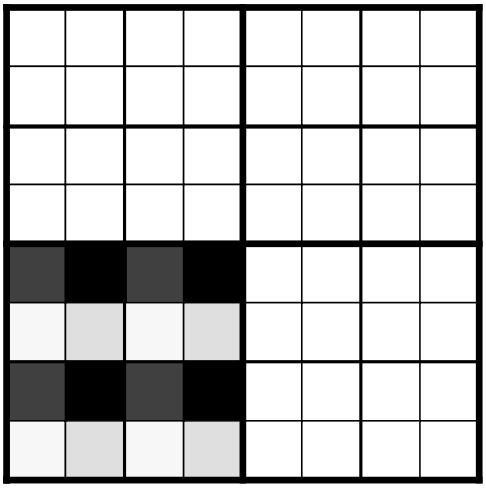

Figure 4.2: Level-1 butterfly groups for $N = 64$. The heaviest lines demarcate the four level-1 sub-DFTs, and the medium bold lines demarcate the sixteen level-0 sub-DFTs. We combine four level-0 sub-DFTs to compute a 4×4-point level-1 sub-DFT. Similarly shaded points form each of the four butterfly groups in one of these level-1 sub-DFTs.

sub-DFT are

1. the value of each of the four participating points after the level-$(k-1)$ sub-DFTs,
2. the size of the level-k sub-DFT, and
3. the location of point $A[x_1, y_1]$ within the level-k sub-DFT.[2]

We now describe the precise arithmetic operations in a single level-k butterfly, as shown in Figure 4.5. When computing a butterfly within a level-k sub-DFT, we begin by scaling each of the four points by a twiddle factor of root $2K$ and an exponent dependent solely upon x_1 and y_1. We define a, b, c, and d to be these scaled values, such that

$$a = A[x_1, y_1] \cdot \omega_{2K}^{0} , \tag{4.1}$$
$$b = A[x_2, y_1] \cdot \omega_{2K}^{x_1} , \tag{4.2}$$
$$c = A[x_1, y_2] \cdot \omega_{2K}^{x_2} , \tag{4.3}$$
$$d = A[x_2, y_2] \cdot \omega_{2K}^{x_1+x_2} . \tag{4.4}$$

Note that $\omega_{2K}^{0} = 1$, and thus, point $A[x_1, y_1]$ is not scaled. We use the following intermediate variables to simplify our work, as well as to reduce the number of arithmetic operations

[2] In Figures 4.1–4.3 and later, in Figures 4.6–4.8, $A[x_1, y_1]$ is the lower left point in the four-point butterfly.

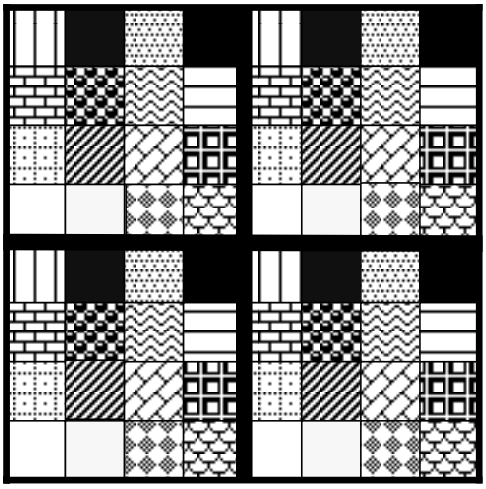

Figure 4.3: Level-3 butterfly groups for $N = 64$. Bold lines demarcate the four level-1 sub-DFTs, which we combine to compute the full 8×8-point DFT. Similarly shaded points interact in a butterfly in level 2 of the computation.

within the butterfly:

$$\begin{aligned} A &= a + b\,, \\ B &= a - b\,, \\ C &= c + d\,, \\ D &= c - d\,. \end{aligned}$$

Finally, we complete the butterfly operation by replacing $A[x_i, y_j]$, for $i, j = 1, 2$, with the result of the level-k DFT:

$$\begin{aligned} A[x_1, y_1] &= A + C\,, \\ A[x_2, y_1] &= B + D\,, \\ A[x_1, y_2] &= A - C\,, \\ A[x_2, y_2] &= B - D\,. \end{aligned}$$

Let us now revisit the previous example, in which $N = 64$. Earlier, we focused on determining which points interact at each level of the computation, and the shading in Figures 4.1–4.3 indicate the groups of four points that interact with one another in a butterfly. In this subsection, we defined the butterfly operation. Figures 4.6–4.8, therefore, provide another representation of the 64-point example, by showing the twiddle factors that

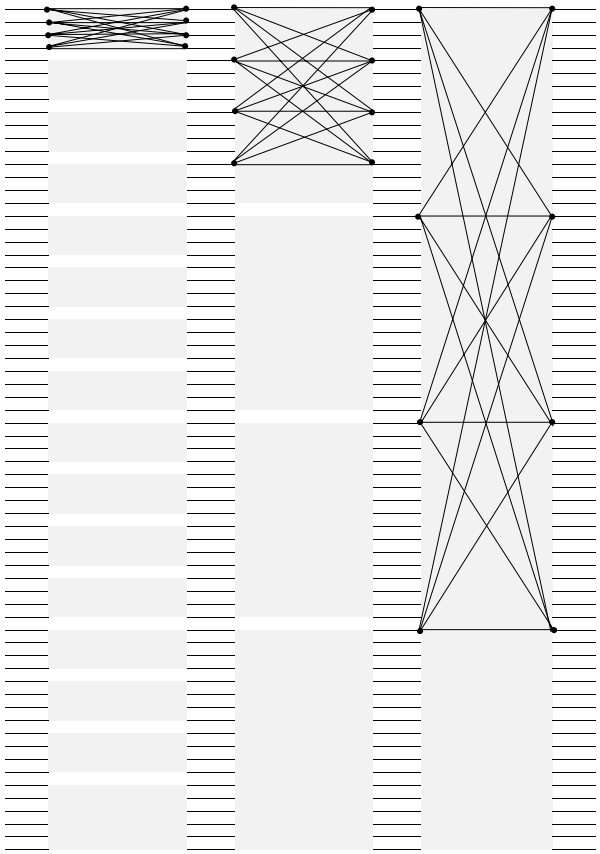

Figure 4.4: Butterfly graph for $N = 64$. (For clarity, only some butterflies are drawn.) The 64 lines on the left represent the 64-point input. In level 0 of the computation, groups of four elements interact in 16 sub-DFTs. In level 1, there are four sub-DFTs, and each one has 16 points. Finally, all 64 elements interact in the level-2 DFT.

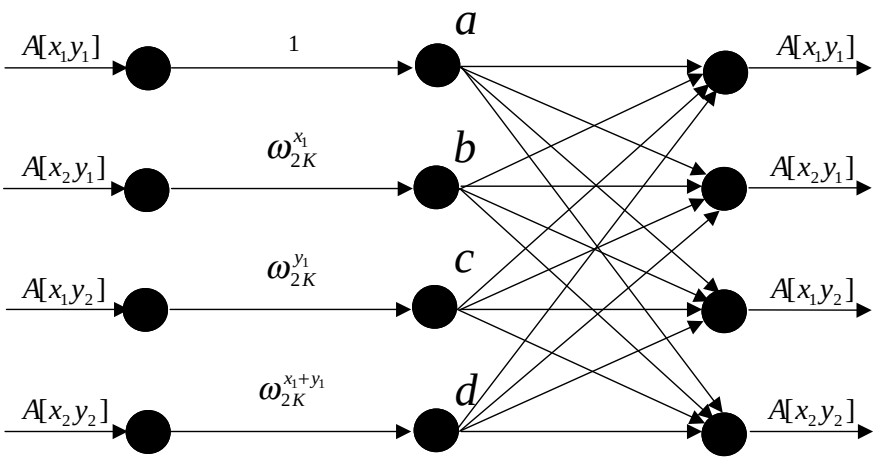

Figure 4.5: A single butterfly operation. Each point in the butterfly is first scaled by a twiddle factor that corresponds to its position in the butterfly, and we call the scaled values a, b, c, and d. The result of the DFT for each point is the sum of the scaled values, with some negations, denoted by a negative sign below the arrow. Thus, for example, $A[x_1, y_2] = a + b - c - d$ and $A[x_2, y_1] = a - b - c + d$.

we use to scale the input at each of the three levels of the computation. In these figures, we include only the exponent of the twiddle factor by which we scale a point, because the root of each level-k twiddle factor is always $2K$. We mentioned earlier that each exponent depends solely on the location of the lower left point, $A[x_1, y_1]$. More specifically, we have, from Equations 4.1–4.4, that Figures 4.6–4.8 represent each shaded four-point grouping in Figures 4.1–4.3 with the exponents

0	for the lower left point,
x_1	for the lower right point,
y_1	for the upper left point,
$x_1 + y_1$	for the upper right point.

In Figure 4.6, for example, we have all zeros, because the lower left point in each of the butterfly operations is in position $[0, 0]$ within its four-point sub-DFT. Let us also examine some of the butterflies in a level-1 sub-DFT as shown in Figure 4.7. In each level-1 sub-DFT, $A[0, 0]$ is the lower left point of a butterfly with all zero exponents, because $x_1, y_1 = 0$. At the other extreme, $A[1, 1]$ is the lower left point of another butterfly in each sub-DFT. In this case, $x_1, y_1 = 1$, and so, from Equations 4.1–4.4, we have exponents of 0, $1 = x_1$, $1 = y_1$, and $2 = x_1 + y_2$, in this butterfly. In Figure 4.8, we find that $A[0, 0]$ is again the lower left point of a butterfly with all zero exponents. $A[3, 3]$, on the other hand, is the lower left point of a butterfly with exponents of 0, $3 = x_1$, $3 = y_1$, and $6 = x_1 + y_1$.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 4.6: Level-0 twiddle factor exponents for $N = 64$. Each number represents the exponent of the twiddle factor by which we scale the point in that position of the matrix. Observe that all 16 of the sub-DFTs in level 0 have the same exponents.

4.2 Out-of-core implementation on a multiprocessor

We have designed and implemented an out-of-core, multiprocessor algorithm based on the in-core serial vector-radix method described above. In many ways, the out-of-core vector-radix method is analogous to the one-dimensional out-of-core algorithm presented in [CN98]. Therefore, as a point of reference, we briefly summarize the one-dimensional out-of-core algorithm here. However, we encourage the reader to refer to [CN98] for further details on out-of-core FFT implementations.

For clarity, we confine ourselves to a uniprocessor for much of the following descriptions of both the one- and two-dimensional out-of-core FFT implementations. Later in this chapter, we show that adapting the two-dimensional out-of-core vector-radix method to a multiprocessor platform is straightforward, given previous work [CWN97].

One-dimensional out-of-core FFT on a uniprocessor

A one-dimensional FFT begins with a one-dimensional bit-reversal, after which we perform a sequence of butterfly operations, as shown in Figure 4.9. The depth of the one-dimensional butterfly graph for a problem size of N is $n = \lg N$. Let us then define a *mini-butterfly* to be an M-point butterfly graph of depth $m = \lg M$, where M is the memory size. We can therefore compute a mini-butterfly by reading in a memoryload, computing the butterfly

1	1	1	2	1	1	1	2
0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	1
1	1	1	2	1	1	1	2
0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	1

Figure 4.7: Level-1 twiddle factor exponents for $N = 64$. Once again, each number represents the exponent of the twiddle factor by which we scale the point in that position of the matrix. Observe that all four level-1 sub-DFTs have the same exponents.

graph on these M values, and writing out the memoryload. If we assume for a moment that m divides n, then we can evenly divide an N-point problem into n/m *superlevels*, where each superlevel contains N/M mini-butterflies.

If m does not divide n, then there are $\lceil n/m \rceil$ superlevels, and we must compensate in the final superlevel. In this final superlevel, there are $r = n \bmod m$ levels remaining to be computed in the full butterfly graph, and therefore we must compute mini-butterflies of depth r. We can still fit M points into memory, however. Thus, we continue to read and write memoryloads of M points, but instead of computing one mini-butterfly per memoryload, we compute $M/2^r$.

After each superlevel, we perform an m-bit right-rotation to reorder the data so that in the next superlevel, the points appearing in each mini-butterfly will be consecutively ordered. When we discuss the two-dimensional algorithm in the next subsection, we will explain in some depth the analogous permutation. However, we refer the reader to [Sni81] for intuition as to why the m-bit right-rotation permutation is used to reorder the data in the one-dimensional algorithm.

A key subroutine in the one-dimensional out-of-core FFT implementation [CN98] performs a BMMC permutation on the full N-point data set. This subroutine, based on techniques described in [CC99, CSW99], takes an $n \times n$ characteristic matrix (bit-packed into n words) as an input, and optimally performs the BMMC permutation so characterized. The bit-reversal permuation is a BMMC permutation, as is the m-bit right-rotation. Therefore,

3	3	3	3	3	4	5	6
2	2	2	2	2	3	4	5
1	1	1	1	1	2	3	4
0	0	0	0	0	1	2	3
0	0	0	0	0	1	2	3
0	0	0	0	0	1	2	3
0	0	0	0	0	1	2	3
0	0	0	0	0	1	2	3

Figure 4.8: Level-2 twiddle factor exponents for $N = 64$. Again, each number represents the exponent of the twiddle factor by which we scale the point in that position of the matrix.

the BMMC-permutation subroutine is called to perform both permutations optimally.

Two-dimensional out-of-core vector-radix FFT on a uniprocessor

The two-dimensional out-of-core vector-radix method is an extension of the one-dimensional out-of-core algorithm just described. Therefore, we will refer to the sketch of the one-dimensional algorithm to illustrate in more detail the two-dimensional implementation. We begin with the two-dimensional bit-reversal that we use to permute the data at the beginning of the vector-radix method. It is clearly analogous to the one-dimensional bit-reversal that begins the one-dimensional computation, and similarly we can use the BMMC subroutine to perform this permutation optimally.

After the two-dimensional bit-reversal, we perform a sequence of butterfly operations, as we did in the one-dimensional algorithm. The two-dimensional butterfly graph has a depth of $\log_4 N$. Therefore, if we divide the butterfly graph into mini-butterflies of depth $\log_4 M$, then we can compute N/M mini-butterflies in each of the $\lceil \log_4 N / \log_4 M \rceil = \lceil n/m \rceil$ superlevels. Thus, despite the change of base, the one- and two-dimensional algorithms both have $\lceil n/m \rceil$ superlevels, each with N/M mini-butterflies. Moreover, we once again compensate for the case in which m does not divide n evenly in the final superlevel by computing $M/2^r$ mini-butterflies per memoryload, where $r = n \bmod m$.

Finally, we shall define the BMMC permutation we use to reorder the data between superlevels so that the mini-butterflies can operate on contiguous M-point chunks in memory.

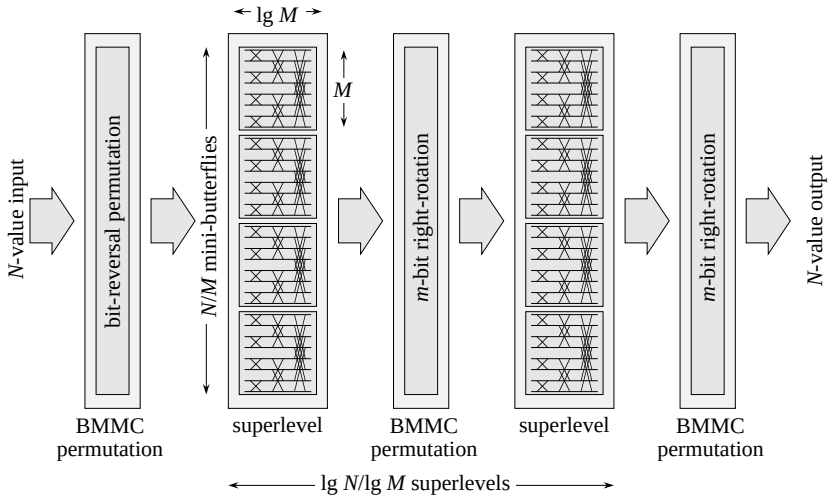

Figure 4.9: Out-of-core butterfly graph for one-dimensional problem. After a bit-reversal permutation, we compute $\lceil n/m \rceil$ superlevels, each of which contains N/M mini-butterflies. After each superlevel, we perform an m-bit right-rotation to reorder the data for the next superlevel. Note that if m does not divide n, then the last rotation is actually an n mod m-bit right-rotation.

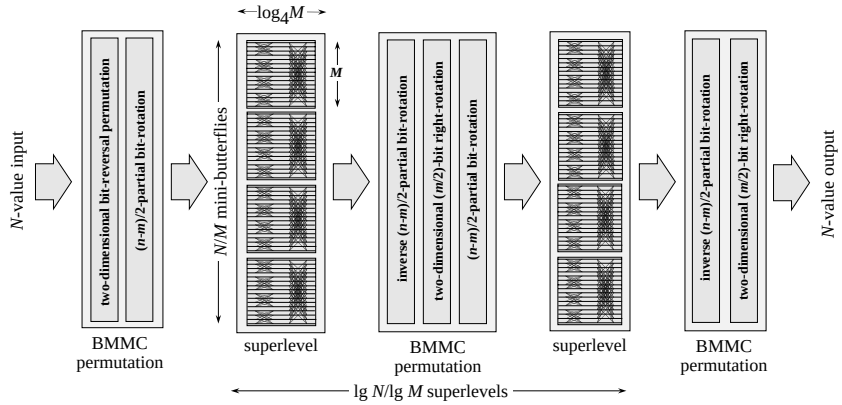

Figure 4.10: Out-of-core butterfly graph for two-dimensional problem. After a two-dimensional bit-reversal permutation, we compute $\lceil n/m \rceil$ superlevels, each of which contains N/M mini-butterflies. After each superlevel, we perform an $m/2$-bit right-rotation to reorder the data for the next superlevel. Note that if m does not divide n, then the last rotation is actually an $(n \bmod m)/2$-bit right-rotation.

This permutation is analogous to the m-bit right-rotation that the one-dimensional algorithm employs between superlevels. However, it is slightly more complex in that it is the composition of more than one characteristic matrix. In Figure 4.10, we sketch out these permutations. Specifically, we perform an $(n-m)/2$-partial bit-rotation before each superlevel. After each superlevel but the last, we perform the inverse of an $(n-m)/2$-partial bit-rotation and a two-dimensional $(m/2)$-bit right-rotation. After the final superlevel, we perform the inverse of an $(n-m)/2$-partial bit-rotation and a two-dimensional $(n \bmod m)/2$-bit right-rotation. We offer an example to illustrate how these permutations reorder the data. In this example, let $N = 256$ and $M = 16$. At times, it will be useful to visualize the input as a 16×16-point matrix and memory as a 4×4-point matrix. Let us assume that we have already computed the two-dimensional bit-reversal permutation, and so the butterfly computations remain. We assign indices from 0 to 255 after the bit-reversal. Assuming that the points are stored in row-major order, we therefore have

240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Because $M = 16$, there are 16 points in each mini-butterfly. In the following representation of the data, the bold lines demarcate the boundaries between mini-butterflies in superlevel 0.

240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Each row represents $M = 16$ contiguous points in memory, and therefore we need to reorder the data so that each mini-butterfly fits in a memoryload, or row. For example, the points in the shaded mini-butterfly are clearly not consecutive in memory, and so to compute this

mini-butterfly, we must reorder the data so that these points fill 16 contiguous memory locations. Thus, we perform an $(n-m)/2$-partial bit-rotation permutation to obtain

204	205	206	207	220	221	222	223	236	237	238	239	252	253	254	255
140	141	142	143	156	157	158	159	172	173	174	175	188	189	190	191
76	77	78	79	92	93	94	95	108	109	110	111	124	125	126	127
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63
200	201	202	203	216	217	218	219	232	233	234	235	248	249	250	251
136	137	138	139	152	153	154	155	168	169	170	171	184	185	186	187
72	73	74	75	88	89	90	91	104	105	106	107	120	121	122	123
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
196	197	198	199	212	213	214	215	228	229	230	231	244	245	246	247
132	133	134	135	148	149	150	151	164	165	166	167	180	181	182	183
68	69	70	71	84	85	86	87	100	101	102	103	116	117	118	119
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
192	193	194	195	208	209	210	211	224	225	226	227	240	241	242	243
128	129	130	131	144	145	146	147	160	161	162	163	176	177	178	179
64	65	66	67	80	81	82	83	96	97	98	99	112	113	114	115
0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51

The data in the shaded region, as well as the data in all of the other mini-butterflies, are now consecutive in memory. We can therefore perform the $N/M = 256/16 = 16$ mini-butterflies (one per row) in superlevel 0.

After superlevel 0, we perform an inverse $(n-m)/2$-partial bit-rotation to return the data to their positions before the superlevel, as depicted by

240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

In superlevel 1, each of the mini-butterflies will once again contain 16 points. This time, however, the mini-butterfly groupings are even more scattered than they were before superlevel 0. The shaded regions in the following drawing represent the points in one of the mini-butterflies:

240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

If we perform a two-dimensional $(m/2)$-bit right-rotation, we obtain

240	244	248	252	241	245	249	253	242	246	250	254	243	247	251	255
176	180	184	188	177	181	185	189	178	182	186	190	179	183	187	191
112	116	120	124	113	117	121	125	114	118	122	126	115	119	123	127
48	52	56	60	49	53	57	61	50	54	58	62	51	55	59	63
224	228	232	236	225	229	233	237	226	230	234	238	227	231	235	239
160	164	168	172	161	165	169	173	162	166	170	174	163	167	171	175
96	100	104	108	97	101	105	109	98	102	106	110	99	103	107	111
32	36	40	44	33	37	41	45	34	38	42	46	35	39	43	47
208	212	216	220	209	213	217	221	210	214	218	222	211	215	219	223
144	148	152	156	145	149	153	157	146	150	154	158	147	151	155	159
80	84	88	92	81	85	89	93	82	86	90	94	83	87	91	95
16	20	24	28	17	21	25	29	18	22	26	30	19	23	27	31
192	196	200	204	193	197	201	205	194	198	202	206	195	199	203	207
128	132	136	140	129	133	137	141	130	134	138	142	131	135	139	143
64	68	72	76	65	69	73	77	66	70	74	78	67	71	75	79
0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15

The matrix is now divided into mini-butterflies in the same way it was divided at the beginning of the example. Because of the previous permutation, however, the indices of the points in a given mini-butterfly are different now than they were before superlevel 0. We performed an $(n-m)/2$-partial bit-rotation to reorder the data before superlevel 0, and so we use the same permutation before superlevel 1 to permute the data so that the points in each mini-butterfly are contiguous in memory. We thus obtain

51	55	59	63	115	119	123	127	179	183	187	191	243	247	251	255
35	39	43	47	99	103	107	111	163	167	171	175	227	231	235	239
19	23	27	31	83	87	91	95	147	151	155	159	211	215	219	223
3	7	11	15	67	71	75	79	131	135	139	143	195	199	203	207
50	54	58	62	114	118	122	126	178	182	186	190	242	246	250	254
34	38	42	46	98	102	106	110	162	166	170	174	226	230	234	238
18	22	26	30	82	86	90	94	146	150	154	158	210	214	218	222
2	6	10	14	66	70	74	78	130	134	138	142	194	198	202	206
49	53	57	61	113	117	121	125	177	181	185	189	241	245	249	253
33	37	41	45	97	101	105	109	161	165	169	173	225	229	233	237
17	21	25	29	81	85	89	93	145	149	153	157	209	213	217	221
1	5	9	13	65	69	73	77	129	133	137	141	193	197	201	205
48	52	56	60	112	116	120	124	176	180	184	188	240	244	248	252
32	36	40	44	96	100	104	108	160	164	168	172	224	228	232	236
16	20	24	28	80	84	88	92	144	148	152	156	208	212	216	220
0	4	8	12	64	68	72	76	128	132	136	140	192	196	200	204

The data in the shaded region, as well as the data in all of the other mini-butterflies, are now consecutive in memory. We can therefore perform the $N/M = 256/16 = 16$ mini-butterflies in superlevel 1.

After superlevel 1, we perform an inverse $(n - m)/2$-partial bit-rotation to return the data to their positions before the superlevel, as depicted by

240	244	248	252	241	245	249	253	242	246	250	254	243	247	251	255
176	180	184	188	177	181	185	189	178	182	186	190	179	183	187	191
112	116	120	124	113	117	121	125	114	118	122	126	115	119	123	127
48	52	56	60	49	53	57	61	50	54	58	62	51	55	59	63
224	228	232	236	225	229	233	237	226	230	234	238	227	231	235	239
160	164	168	172	161	165	169	173	162	166	170	174	163	167	171	175
96	100	104	108	97	101	105	109	98	102	106	110	99	103	107	111
32	36	40	44	33	37	41	45	34	38	42	46	35	39	43	47
208	212	216	220	209	213	217	221	210	214	218	222	211	215	219	223
144	148	152	156	145	149	153	157	146	150	154	158	147	151	155	159
80	84	88	92	81	85	89	93	82	86	90	94	83	87	91	95
16	20	24	28	17	21	25	29	18	22	26	30	19	23	27	31
192	196	200	204	193	197	201	205	194	198	202	206	195	199	203	207
128	132	136	140	129	133	137	141	130	134	138	142	131	135	139	143
64	68	72	76	65	69	73	77	66	70	74	78	67	71	75	79
0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15

Finally, we perform a two-dimensional $(n \bmod m)/2$-bit right-rotation to obtain

240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The data are once again in their original positions, and the computation is completed.

Two-dimensional vector-radix out-of-core FFT on a multiprocessor

Parallelizing the algorithm was a simple extension of the one-dimensional, multiprocessor, out-of-core implementation [CWN97], to which we refer the reader for details. We note that in the multiprocessor implementation of the vector-radix method, we rearrange the data before each superlevel to put it into processor-major order so that all the points of each mini-butterfly reside in a single processor's memory. After each superlevel, we rearrange the data into the canonical ordering—stripe-major—as assumed by the BMMC subroutine that we use to further reorder the data. Furthermore, we must modify the two-dimensional algorithm just described to account for this move to a multiprocessor platform. We are now concerned with the memory on a single processor, and therefore, all previous references to M and m must be replaced by M/P and $m - p$ respectively, where P is the number of processors and $p = \lg P$.

Therefore, we begin the two-dimensional vector-radix, multiprocessor, out-of-core FFT implementation with a two-dimensional bit-reversal permutation. Before each superlevel, we perform an $(n-m+p)/2$-partial bit-rotation, followed by the stripe-major to processor-major BMMC permutation. Within each superlevel, we make one pass over the data to compute mini-butterflies. After every superlevel but the last, we perform the processor-major to stripe-major BMMC permutation, the inverse $(n - m + p)/2$-partial bit-rotation, and the two-dimensional $(m-p)/2$-bit right-rotation. After the final superlevel, we perform the processor-major to stripe-major BMMC permutation, the inverse of an $(n - m + p)/2$-partial bit-rotation, and the two-dimensional $(n \bmod (m - p))/2$-bit right-rotation.

We simplified the complexity analysis of the dimensional method by assuming in Chapter 3 that each dimension-j FFT fit in core. We make the same assumption in this chapter to simplify the complexity analysis of the vector-radix method. Because our implementation is for two-dimensional problems with an aspect ratio of 1, this assumption implies that $\sqrt{N} \leq M/P$. Because, $m < n$, we further have that $\lceil n/(m-p) \rceil = 2$, and so there are exactly two superlevels in the computation. Furthermore, $n \bmod (m - p) = n - m + p$. Therefore, the characteristic matrix of the two-dimensional $(n \bmod (m - p))/2$-bit right-rotation is simply the inverse of the characteristic matrix of the two-dimensional $(m-p)/2$-bit right-rotation. In the remainder of this chapter, we do not consider the possibility that $\sqrt{N} > M/P$, except to note that our implementation does handle it correctly.

Now we show how to take advantage of closure of BMMC permutations under composition. Let us denote the characteristic matrices of the individual BMMC permutations as follows:

- S characterizes the stripe-major to processor-major permutation.
- S^{-1} characterizes the processor-major to stripe-major permutation.
- U characterizes a two-dimensional bit-reversal permutation.
- Q characterizes an $(n - m + p)/2$-partial bit-rotation permutation.
- Q^{-1} characterizes its inverse.
- T characterizes a two-dimensional $(m - p)/2$-bit right-rotation.

- T^{-1} characterizes its inverse.

Hence, the BMMC closure properties result in our performing the following permutations:

- Prior to computing superlevel 0, we perform the BMMC permutation characterized by the matrix product $S\,Q\,U$.
- Between computing superlevel 0 and superlevel 1, we compose the permutations that follow superlevel 0 with those that precede superlevel 1, performing the BMMC permutation characterized by the matrix product $S\,Q\,T\,Q^{-1}\,S^{-1}$.
- After computing superlevel 1, we perform the BMMC permutation characterized by the matrix product $T^{-1}\,Q^{-1}\,S^{-1}$.

It is easy to multiply these characteristic matrices together before presenting the product to the BMMC-permutation subroutine.

Implementation notes

We briefly remark on some of the details of the multiprocessor, out-of-core vector-radix implementation. As in previous out-of-core FFT implementations, [CN98, CWN97], we call asynchronous (i.e., non-blocking) I/O functions, when the underlying system supports it, by allocating three buffers: for reading into, writing from, and computing in. Unlike these implementations, the vector-radix algorithm incorporates the findings from our inquiry into twiddle factor accuracy that we discussed in Section 2.3. We had to modify the out-of-core recursive bisection method before folding it into the out-of-core vector-radix implementation. In the out-of-core implementation, we view each butterfly in terms of the twiddle factors by which we scale the lower right and upper left points in the butterfly. (We can do this because the lower left twiddle factor is always 0, and the upper right twiddle factor is the product of the lower right and upper left twiddle factors.) Fortunately, a given memoryload uses the same values for the lower right twiddle factors as it uses for the upper left twiddle factors. Thus, one precomputed vector of twiddle factors suffices for the superlevel. However, we iterate through this precomputed vector in one way for the lower right twiddle factors and in another way for the upper left twiddle factors.

4.3 Analytical results

In this section, we analyze the I/O complexity of our out-of-core implementation of the vector-radix method for a multiprocessor. Recall that the I/O complexity of a BMMC permutation is $\frac{2N}{BD}\left(\left\lceil \frac{\operatorname{rank}\phi}{\lg(M/B)}\right\rceil + 1\right)$ parallel I/Os, where ϕ is the lower left $\lg(N/M) \times \lg M$ submatrix of the characteristic matrix, and the rank is computed over $GF(2)$. We use the same technique that we used in Section 3.2 to analyze the I/O complexity of the out-of-core implementation of the dimensional method. Once again, we refer the reader to to the exposition in Section 1.3 for details on complexity analysis and the proof technique employed in the following three lemmas, which are central to the analysis. We also remind the reader of the assumption we made in Section 4.2, that $\sqrt{N} \le M/P$. We refer to this assumption several times in the analysis that follows.

Lemma 6 *For the matrix product $S\,Q\,U$, we have*

$$\operatorname{rank}\phi = \min(n-m, (m-p)/2)\ .$$

Proof: We know that to compute $\operatorname{rank}\phi$ of the matrix product $S\,Q\,U$, we can compute the rank of the $(n-m)\times m$ matrix product $Z = X\,S\,Q\,U\,Y\,\Pi$. (We will define Π later.) From the proof of Lemma 1, we already have the subproduct $E = X\,S$. Therefore, we need to compute $F = U\,Y$, $G = Q\,F$, and $H = G\,\Pi$. We can then compute the rank of the final product $Z = E\,H$. We can represent this grouping as

$$\underbrace{\underbrace{X}_{E}\,\underbrace{S}\;Q\;\underbrace{\underbrace{U}_{F}\,\underbrace{Y}}_{G}\;\Pi}_{H}\ .$$
$$Z$$

We begin with the matrix product $F = U\,Y$. Recall that U and Y are of the forms

$$U = \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & I^{\mathrm{A}} \end{array}\right] \begin{array}{l} \frac{n}{2} \\ \frac{n}{2} \end{array} \quad \text{and} \quad Y = \left[\begin{array}{c} I \\ \hline 0 \end{array}\right] \begin{array}{l} m \\ n-m \end{array}\ .$$

If we view Y as a column selection matrix, then F is comprised of the leftmost m columns of U. We have assumed that each dimension-j FFT computation fits in-core. Therefore, $n/2 \le m$, and

$$F = \left[\begin{array}{c|c} I^{\mathrm{A}} & 0 \\ \hline 0 & 0 \\ \hline 0 & I^{\mathrm{A}} \end{array}\right] \begin{array}{l} \frac{n}{2} \\ n-m \\ m-\frac{n}{2} \end{array}\ .$$

We now compute $G = Q\,F$, where Q is of the form

$$Q = \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{m-p}{2} \\ \frac{n}{2} \\ \frac{n-m+p}{2} \end{array}\ .$$

If we view F as a column selection of Q, then the matrix G is the reversal of Q's first $n/2$ columns followed by the reversal of the last $m - n/2$ columns. Hence,

$$G = \left[\begin{array}{c|c|c} 0 & I^{\mathrm{A}} & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & I^{\mathrm{A}} \\ \hline I^{\mathrm{A}} & 0 & 0 \end{array}\right] \begin{array}{l} \frac{m-p}{2} \\ n-m \\ m-\frac{n}{2} \\ \frac{n-m+p}{2} \end{array}\ .$$

To simplify the subproduct G, we permute its columns by composing it with the column permutation matrix

$$\Pi = \begin{array}{c} \begin{array}{ccc} \frac{m-p}{2} & m-\frac{n}{2} & \frac{n-m+p}{2} \end{array} \\ \left[\begin{array}{c|c|c} 0 & 0 & I^{\mathrm{A}} \\ \hline I^{\mathrm{A}} & 0 & 0 \\ \hline 0 & I^{\mathrm{A}} & 0 \end{array}\right] \end{array} \begin{array}{l} \frac{n-m+p}{2} \\ \frac{m-p}{2} \\ m-\frac{n}{2} \end{array}$$

to form

$$H = G\,\Pi = \begin{array}{c} \begin{array}{cc} \frac{m-p}{2} & \frac{m+p}{2} \end{array} \\ \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} \frac{m-p}{2} \\ n-m \\ \frac{m+p}{2} \end{array} .$$

We now compute the rank of the final product $Z = E\,H$, where E is of the form

$$E = \begin{array}{c} \begin{array}{ccc} m-p & n-m & p \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} n-m .$$

Let us view E as an $(n-m)$-row selection matrix. Suppose for a moment that $p = m$. In this case, E would select only the $n-m$ zero rows of H. However, we know that $p < m$, and so we must move the band downward to include $(m+p)/2 - p$ nonzero rows from the lower $(m+p)/2$ nonzero rows of H. Therefore, $\operatorname{rank}\phi = (m+p)/2 - p = (m-p)/2$. The matrix E cannot select more than $n-m$ rows from H, however, and so in fact $\operatorname{rank}\phi = \min(n-m, (m-p)/2)$. ∎

Lemma 7 *For the matrix product $S\,Q\,T\,Q^{-1}\,S^{-1}$, we have* $\operatorname{rank}\phi = n-m$.

Proof: To compute $\operatorname{rank}\phi$ of this matrix product, we will compute the rank of the $(n-m)\times m$ matrix product $Z = X\,S\,Q\,T\,Q^{-1}\,S^{-1}\,Y\,\Pi_1$. We already have the subproduct $E = X\,S$ from the proof of Lemma 1 and the subproduct $H = S^{-1}\,Y\,\Pi_1$ from the proof of Lemma 2, where Π_1 is the column permutation matrix defined in that lemma. Therefore, we can use the two subproducts E and H in this proof. We still need to compute $J = T\,Q^{-1}$, $K = Q\,J$, and $L = E\,K$ and finally the rank of $Z = L\,H$. We can represent this grouping schematically as

$$\underbrace{\underbrace{\underbrace{X\,S}_{E}\;Q\;\underbrace{T\,Q^{-1}}_{J}}_{K}\;\underbrace{S^{-1}\,Y}_{G}}_{L}\;\underbrace{\Pi_1}_{H}\;.$$

We first compute $J = T\,Q^{-1}$, where T and Q^{-1} are of the forms

$$T = \begin{array}{c} \begin{matrix} \frac{m-p}{2} & \frac{n-m+p}{2} & \frac{m-p}{2} & \frac{n-m+p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c|c} 0 & I & 0 & 0 \\ \hline I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I \\ \hline 0 & 0 & I & 0 \end{array}\right] \end{array} \begin{matrix} \frac{n-m+p}{2} \\ \frac{m-p}{2} \\ \frac{n-m+p}{2} \\ \frac{m-p}{2} \end{matrix} ,$$

$$Q^{-1} = \begin{array}{c} \begin{matrix} \frac{m-p}{2} & \frac{n}{2} & \frac{n-m+p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{matrix} \frac{m-p}{2} \\ \frac{n-m+p}{2} \\ \frac{n}{2} \end{matrix} .$$

Let us view Q^{-1} as a column permutation of T. From left to right, J is comprised of T's leftmost $(m-p)/2$ columns, rightmost $n/2$ columns, and the remaining $(n-m+p)/2$ columns. Therefore, J is of the form

$$J = \begin{array}{c} \begin{matrix} \frac{m-p}{2} & \frac{m-p}{2} & \frac{n-m+p}{2} & \frac{n-m+p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c|c} 0 & 0 & 0 & I \\ \hline I & 0 & 0 & 0 \\ \hline 0 & 0 & I & 0 \\ \hline 0 & I & 0 & 0 \end{array}\right] \end{array} \begin{matrix} \frac{n-m+p}{2} \\ \frac{m-p}{2} \\ \frac{n-m+p}{2} \\ \frac{m-p}{2} \end{matrix} .$$

We now compute the subproduct $K = Q\,J$, where Q is of the form

$$Q = \begin{array}{c} \begin{matrix} \frac{m-p}{2} & \frac{n-m+p}{2} & \frac{n}{2} \end{matrix} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{matrix} \frac{m-p}{2} \\ \frac{n}{2} \\ \frac{n-m+p}{2} \end{matrix} .$$

Let us view Q as a row permutation of J. We have assumed that $n/2 \le m-p$, which implies that $(n-m+p)/2 \le (m-p)/2$. Therefore, we can redraw J as

$$J = \begin{array}{c} \begin{matrix} \frac{m-p}{2} - \frac{n-m+p}{2} & \frac{n-m+p}{2} & \frac{m-p}{2} & \frac{n}{2} - \frac{m-p}{2} & \frac{n-m+p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c|c|c} 0 & 0 & 0 & 0 & I \\ \hline I & 0 & 0 & 0 & 0 \\ \hline 0 & I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I & 0 \\ \hline 0 & 0 & I & 0 & 0 \end{array}\right] \end{array} \begin{matrix} \frac{n-m+p}{2} \\ \frac{m-p}{2} - \frac{n-m+p}{2} \\ \frac{n-m+p}{2} \\ \frac{n}{2} - \frac{m-p}{2} \\ \frac{m-p}{2} \end{matrix} .$$

The matrix K is comprised of J's upper $(m-p)/2$ rows, followed by J's lower $n/2$ rows, followed by the remaining $(n-m+p)/2$ rows and is thus of the form

$$K = \begin{array}{c} \begin{array}{ccccc} {\scriptstyle m-p-\frac{n}{2}} & {\scriptstyle \frac{n-m+p}{2}} & {\scriptstyle \frac{m-p}{2}} & {\scriptstyle \frac{n-m+p}{2}} & {\scriptstyle \frac{n-m+p}{2}} \end{array} \\ \left[\begin{array}{c|c|c|c|c} 0 & 0 & 0 & 0 & I \\ \hline I & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I & 0 \\ \hline 0 & 0 & I & 0 & 0 \\ \hline 0 & I & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} {\scriptstyle \frac{n-m+p}{2}} \\ {\scriptstyle m-p-\frac{n}{2}} \\ {\scriptstyle \frac{n-m+p}{2}} \\ {\scriptstyle \frac{m-p}{2}} \\ {\scriptstyle \frac{n-m+p}{2}} \end{array} .$$

We can now compute $L = E\,K$, where E is of the form

$$E = \begin{array}{c} \begin{array}{ccc} {\scriptstyle m-p} & {\scriptstyle n-m} & {\scriptstyle p} \end{array} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} {\scriptstyle n-m} .$$

We must examine two cases: $n-m > p$ and $n-m \leq p$.

<u>Case 1</u>: $n-m > p$

Let us view E as a row selection matrix that selects $n-m$ rows from K's lower $n-m+p$ rows. We have assumed that $n/2 \leq m-p$, and therefore, $n-m+p \leq n/2$. We also know that $n-m+p > (n-m+p)/2$. Hence, the band begins no higher than $n/2$ rows from the bottom of K, but no lower than $(n-m+p)/2$ rows from the bottom of K. In this case, $n-m > p$, and therefore, $(n-m+p)/2 > p$. Hence, the band ends within K's lower $(n-m+p)/2$ rows. We can therefore redraw K as

$$\begin{array}{c} \begin{array}{ccccccc} {\scriptstyle m-p-\frac{n}{2}} & {\scriptstyle \frac{n-m-p}{2}} & {\scriptstyle p} & {\scriptstyle m-p-\frac{n}{2}} & {\scriptstyle \frac{n-m+p}{2}} & {\scriptstyle \frac{n-m+p}{2}} & {\scriptstyle \frac{n-m+p}{2}} \end{array} \\ \left[\begin{array}{c|c|c|c|c|c|c} 0 & 0 & 0 & 0 & 0 & 0 & I \\ \hline I & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & I & 0 \\ \hline 0 & 0 & 0 & I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & I & 0 & 0 \\ \hline 0 & I & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & I & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} {\scriptstyle \frac{n-m+p}{2}} \\ {\scriptstyle m-p-\frac{n}{2}} \\ {\scriptstyle \frac{n-m+p}{2}} \\ {\scriptstyle m-p-\frac{n}{2}} \\ {\scriptstyle \frac{n-m+p}{2}} \\ {\scriptstyle \frac{n-m-p}{2}} \\ {\scriptstyle p} \end{array} .$$

We select the band of $n-m$ rows that ends p rows from the bottom of K to obtain

$$L = E\,K = \begin{array}{c} \begin{array}{ccccc} {\scriptstyle m-p-\frac{n}{2}} & {\scriptstyle \frac{n-m-p}{2}} & {\scriptstyle m-\frac{n}{2}} & {\scriptstyle \frac{n-m+p}{2}} & {\scriptstyle n-m+p} \end{array} \\ \left[\begin{array}{c|c|c|c|c} 0 & 0 & 0 & I & 0 \\ \hline 0 & I & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} {\scriptstyle \frac{n-m+p}{2}} \\ {\scriptstyle \frac{n-m-p}{2}} \end{array} .$$

We now compute the rank of the final product $Z = L\,H$, where H is of the form

$$H = \begin{array}{c} \begin{array}{cc} {\scriptstyle m-p} & {\scriptstyle p} \end{array} \\ \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} {\scriptstyle m-p} \\ {\scriptstyle n-m} \\ {\scriptstyle p} \end{array} .$$

Let us view L as a row selection that selects a total of $n-m$ rows from the upper $m-p$ rows of H. Because H's upper $m-p$ rows are nonzero, L selects $n-m$ nonzero rows, and thus, $\operatorname{rank}\phi = n-m$.

Case 2: $n-m \le p$

We must compute the subproduct $L = E\,K$, where E and K are of the forms

$$E = \begin{array}{c} \begin{matrix} m-p & n-m & p \end{matrix} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} \begin{matrix} \\ n-m \end{matrix},$$

$$K = \begin{array}{c} \begin{matrix} m-p-\frac{n}{2} & \frac{n-m+p}{2} & \frac{m-p}{2} & \frac{n-m+p}{2} & \frac{n-m+p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c|c|c} 0 & 0 & 0 & 0 & I \\ \hline I & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I & 0 \\ \hline 0 & 0 & I & 0 & 0 \\ \hline 0 & I & 0 & 0 & 0 \end{array}\right] \end{array} \begin{matrix} \frac{n-m+p}{2} \\ m-p-\frac{n}{2} \\ \frac{n-m+p}{2} \\ \frac{m-p}{2} \\ \frac{n-m+p}{2} \end{matrix}.$$

As before, we view E as a row selection matrix that selects $n-m$ rows from K's lower $n-m+p$ rows. Again, $n/2 \le m-p$, and once again, $n-m+p \le n/2$. Hence, the band begins no higher than $n/2$ rows from the bottom of K. In this case $n-m \le p$, and so $(n-m+p)/2 \le p$. Thus, the band ends no lower than $(n-m+p)/2$ rows from the bottom of K. Given these constraints, we redraw K as

$$\begin{array}{c} \begin{matrix} m-p-\frac{n}{2} & \frac{n-m+p}{2} & m-p-\frac{n}{2} & n-m & p-\frac{n-m+p}{2} & \frac{n-m+p}{2} & \frac{n-m+p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c|c|c|c|c} 0 & 0 & 0 & 0 & 0 & 0 & I \\ \hline I & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & I & 0 \\ \hline 0 & 0 & I & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & I & 0 & 0 \\ \hline 0 & I & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{matrix} \frac{n-m+p}{2} \\ m-p-\frac{n}{2} \\ \frac{n-m+p}{2} \\ m-p-\frac{n}{2} \\ n-m \\ p-\frac{n-m+p}{2} \\ \frac{n-m+p}{2} \end{matrix}.$$

We select the band of $n-m$ rows that ends p rows from the bottom of K to obtain

$$L = E\,K = \begin{array}{c} \begin{matrix} \frac{3m-n-3p}{2} & n-m & \frac{n-m+3p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} \begin{matrix} \\ n-m \end{matrix}.$$

We now compute the rank of the final product $Z = L\,H$, where H is of the form

$$H = \begin{array}{c} \begin{matrix} m-p & p \end{matrix} \\ \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{matrix} m-p \\ n-m \\ p \end{matrix}.$$

Let us view L as a selection matrix that selects $n-m$ rows from the upper $(3m-n-3p)/2+(n-m) = (n+m-3p)/2$ rows of H. Because $n-m \le p$, we have that $(n+m-3p)/2 \le m-p$. Therefore, L selects $n-m$ rows from H's upper $m-p$ nonzero rows, and thus, $\operatorname{rank}\phi = n-m$. ∎

Lemma 8 *For the matrix product $T^{-1}\,Q^{-1}\,S^{-1}$, we have*

$$\operatorname{rank}\phi = \min(n-m, (n-m+p)/2)\,.$$

Proof: We know that we can compute $\operatorname{rank}\phi$ of $T^{-1}\,Q^{-1}\,S^{-1}$ by computing the rank of the $(n-m)\times m$ matrix product $Z = X\,T^{-1}\,Q^{-1}\,S^{-1}\,Y\,\Pi_1$. From the proof of Lemma 2, we already have the subproduct $H = S^{-1}\,Y\,\Pi_1$, where Π_1 is the column permutation matrix defined in Lemma 2. Therefore, we can use the subproduct H from Lemma 2 in this proof. We must now find the subproducts $E = X\,T^{-1}$ and $F = E\,Q^{-1}$ and then the final matrix product $Z = F\,H$. We can depict this grouping schematically as

$$\underbrace{\underbrace{\underbrace{X\,T^{-1}}_{E}\,Q^{-1}}_{F}\,\underbrace{\underbrace{S^{-1}\,Y}_{G}\,\Pi_1}_{H}}_{Z}\,.$$

The analysis immediately divides into two cases: $n-m > p$ and $n-m \le p$.

<u>Case 1</u>: $n-m > p$

We begin by computing $E = X\,T^{-1}$, where X and T^{-1} are of the forms

$$X = \begin{array}{c} \begin{matrix} m & n-m \end{matrix} \\ \left[\begin{array}{c|c} 0 & I \end{array}\right] \end{array} \begin{matrix} \\ {\scriptstyle n-m} \end{matrix}\,,$$

$$T^{-1} = \begin{array}{c} \begin{matrix} \frac{n-m+p}{2} & \frac{m-p}{2} & \frac{n-m+p}{2} & \frac{m-p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c|c} 0 & I & 0 & 0 \\ \hline I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I \\ \hline 0 & 0 & I & 0 \end{array}\right] \end{array} \begin{matrix} {\scriptstyle \frac{m-p}{2}} \\ {\scriptstyle \frac{n-m+p}{2}} \\ {\scriptstyle \frac{m-p}{2}} \\ {\scriptstyle \frac{n-m+p}{2}} \end{matrix}\,.$$

We can view X as a row selection matrix that selects the lower $n-m$ rows of T^{-1}. In this case, $n-m > p$, and so we have that $n-m > (n-m+p)/2$. Moreover, we have assumed that $n-m \le n/2$. Therefore, X selects at least T^{-1}'s lower $(n-m+p)/2$ rows, but no more than T^{-1}'s lower $n/2$ rows. Hence,

$$E = \begin{array}{c} \begin{matrix} \frac{n}{2} & \frac{n-m+p}{2} & m-\frac{n}{2} & \frac{n-m-p}{2} \end{matrix} \\ \left[\begin{array}{c|c|c|c} 0 & 0 & 0 & I \\ \hline 0 & I & 0 & 0 \end{array}\right] \end{array} \begin{matrix} {\scriptstyle \frac{n-m-p}{2}} \\ {\scriptstyle \frac{n-m+p}{2}} \end{matrix}\,.$$

We now compute the subproduct $F = E\,Q^{-1}$, where Q^{-1} is of the form

$$Q^{-1} = \begin{array}{c} \begin{array}{ccc} \frac{m-p}{2} & \frac{n}{2} & \frac{n-m+p}{2} \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \frac{m-p}{2} \\ \frac{n-m+p}{2} \\ \frac{n}{2} \end{array} .$$

Let us view E as a row selection matrix that selects $n-m$ rows from Q^{-1} in two bands of $(n-m-p)/2$ and $(n-m+p)/2$ rows. It is clear that all $n-m$ rows selected by E are from among Q^{-1}'s lower $n/2$ rows. Let us isolate Q^{-1}'s lower $n/2$ rows and redraw them as

$$Q' = \begin{array}{c} \begin{array}{ccccc} \frac{m-p}{2} & \frac{n-m+p}{2} & m-\frac{n}{2} & \frac{n-m-p}{2} & \frac{n-m+p}{2} \end{array} \\ \left[\begin{array}{c|c|c|c|c} 0 & I & 0 & 0 & 0 \\ \hline 0 & 0 & I & 0 & 0 \\ \hline 0 & 0 & 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \frac{n-m+p}{2} \\ m-\frac{n}{2} \\ \frac{n-m-p}{2} \end{array} .$$

It is clear from this submatrix of Q^{-1} which two bands of rows E selects, and so we have

$$F = \begin{array}{c} \begin{array}{ccccc} \frac{m-p}{2} & \frac{n-m+p}{2} & m-\frac{n}{2} & \frac{n-m-p}{2} & \frac{n-m+p}{2} \end{array} \\ \left[\begin{array}{c|c|c|c|c} 0 & 0 & 0 & I & 0 \\ \hline 0 & I & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} \frac{n-m-p}{2} \\ \frac{n-m+p}{2} \end{array} .$$

We now compute the rank of the final product $Z = F\,H$, where H is of the form

$$H = \begin{array}{c} \begin{array}{cc} m-p & p \end{array} \\ \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} m-p \\ n-m \\ p \end{array} .$$

Let us view F as a row selection matrix that selects $n-m$ rows from H in two bands of $(n-m-p)/2$ and $(n-m+p)/2$ rows. We know that F selects the band of $(n-m+p)/2$ rows from H's upper $n/2$ rows, and we have assumed that $n/2 \le m-p$. Therefore, F selects this entire band from H's upper $m-p$ nonzero rows, and so the band contributes $(n-m+p)/2$ to the rank. We know that the other band of rows begins m rows from the top of H and therefore includes none of H's upper $m-p$ nonzero rows. We also know that this band ends $(n-m+p)/2$ rows from the bottom of H. We have that $n-m > p$ in this case, and thus $(n-m+p)/2 > p$. Therefore, this band does not include any of H's lower p nonzero rows. Hence, the band falls within H's $n-m$ nonzero rows and contributes nothing to the rank. Since $n-m > p$, we have that $\operatorname{rank}\phi = (n-m+p)/2 = \min(n-m, (n-m+p)/2)$.

Case 2: $n-m \le p$

We begin by computing $E = X\,T^{-1}$, where X and T^{-1} are of the forms

$$X = \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \left[\begin{array}{c|c} 0 & I \end{array}\right] \end{array} \begin{array}{l} \\ n-m \end{array} ,$$

$$T^{-1} = \left[\begin{array}{c|c|c|c} 0 & I & 0 & 0 \\ \hline I & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I \\ \hline 0 & 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{m-p}{2} \\ \frac{n-m+p}{2} \\ \frac{m-p}{2} \\ \frac{n-m+p}{2} \end{array} \quad \begin{array}{c} \frac{n-m+p}{2} \quad \frac{m-p}{2} \quad \frac{n-m+p}{2} \quad \frac{m-p}{2} \end{array} .$$

We can view X as a row selection matrix that selects the lower $n-m$ rows of T^{-1}. In this case, $n-m \leq p$, and thus we have that $n-m \leq (n-m+p)/2$. Therefore, X selects $n-m$ rows from T^{-1}'s lower $(n-m+p)/2$ rows, and thus,

$$E = \begin{array}{c} \frac{m+p}{2} \quad n-m \quad \frac{m-p}{2} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \\ n-m \end{array} .$$

We now compute the subproduct $F = E\,Q^{-1}$, where Q^{-1} is of the form

$$Q^{-1} = \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{m-p}{2} \\ \frac{n-m+p}{2} \\ \frac{n}{2} \end{array} \quad \begin{array}{c} \frac{m-p}{2} \quad \frac{n}{2} \quad \frac{n-m+p}{2} \end{array} .$$

Let us view E as a row selection matrix that selects a band of $n-m$ rows from Q^{-1}'s lower $n-(m+p)/2$ rows. In this case, $n-m \leq p$, and thus we have that $n-(m+p)/2 \leq n/2$. Therefore, E selects a band of $n-m$ rows from Q^{-1}'s lower $n/2$ rows. Let us isolate Q^{-1}'s lower $n/2$ rows and redraw them as

$$Q' = \left[\begin{array}{c|c|c|c|c} 0 & I & 0 & 0 & 0 \\ \hline 0 & 0 & I & 0 & 0 \\ \hline 0 & 0 & 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{m+p-n}{2} \\ n-m \\ \frac{m-p}{2} \end{array} \quad \begin{array}{c} \frac{m-p}{2} \quad \frac{m+p-n}{2} \quad n-m \quad \frac{m-p}{2} \quad \frac{n-m+p}{2} \end{array} .$$

It is clear from this submatrix of Q^{-1} which band of $n-m$ rows E selects, and so we have

$$F = \begin{array}{c} m-\frac{n}{2} \quad n-m \quad \frac{n}{2} \\ \left[\begin{array}{c|c|c} 0 & I & 0 \end{array}\right] \end{array} \begin{array}{l} \\ n-m \end{array} .$$

We now compute the rank of the final product $Z = F\,H$, where H is of the form

$$H = \left[\begin{array}{c|c} I & 0 \\ \hline 0 & 0 \\ \hline 0 & I \end{array}\right] \begin{array}{l} m-p \\ n-m \\ p \end{array} \quad \begin{array}{c} m-p \quad p \end{array} .$$

Let us view F as a row selection matrix that selects $n-m$ rows from H. We know that these $n-m$ rows fall within H's upper $n/2$ rows. Moreover, we have assumed that $n/2 \leq m-p$. Therefore, the $n-m$ rows fall within H's upper $m-p$ nonzero rows. Since $n-m \leq (n-m+p)/2$, we have that $\operatorname{rank}\phi = n-m = \min(n-m, (n-m+p)/2)$. ∎

Theorem 9 *Assuming that both dimensions N_1 and N_2 are integer powers of 2, where $N_1 = N_2 \leq M/P$, we can compute a two-dimensional, multiprocessor, out-of-core FFT in*

$$\left\lceil \frac{\min(n-m,(m-p)/2)}{m-b} \right\rceil + \left\lceil \frac{n-m}{m-b} \right\rceil + \left\lceil \frac{\min(n-m,(n-m+p)/2)}{m-b} \right\rceil + 5$$

passes, where lowercase letters denote logarithms of corresponding uppercase letters.

Proof: We have assumed that $n/2 \leq (m-p)$, and in the out-of-core environment, $n > m$. Therefore, our computation consists of exactly two superlevels. From Lemmas 6–8 and from recognizing that computing the butterfly operations entails one pass for each of the two superlevels, the number of passes to compute a two-dimensional, multiprocessor, out-of-core FFT is

$$\begin{aligned}
&\left(\left\lceil \frac{\min(n-m,(m-p)/2)}{m-b} \right\rceil + 1 \right) + \left(\left\lceil \frac{n-m}{m-b} \right\rceil + 1 \right) + \\
&\qquad\qquad \left(\left\lceil \frac{\min(n-m,(n-m+p)/2)}{m-b} \right\rceil + 1 \right) + 2 \\
&= \left\lceil \frac{\min(n-m,(m-p)/2)}{m-b} \right\rceil + \left\lceil \frac{n-m}{m-b} \right\rceil + \left\lceil \frac{\min(n-m,(n-m+p)/2)}{m-b} \right\rceil + 5 \,.
\end{aligned}$$

■

The following corollary restates this theorem in terms of parallel I/O operations and the actual PDM parameters:

Corollary 10 *Assuming that both dimensions N_1 and N_2 are integer powers of 2, where $N_1 = N_2 \leq M/P$, we can compute a two-dimensional, multiprocessor, out-of-core FFT in*

$$\frac{2N}{BD} \left(\left\lceil \frac{\lg \min\left(N/M\sqrt{M/P}\right)}{\lg(M/B)} \right\rceil + \left\lceil \frac{\lg(N/M)}{\lg(M/B)} \right\rceil + \left\lceil \frac{\lg \min\left(N/M, \sqrt{NP/M}\right)}{\lg(M/B)} \right\rceil + 5 \right)$$

parallel I/O operations. ■

Chapter 5

Empirical Results

We implemented both the dimensional method and the vector-radix algorithm for multiprocessors with parallel disks. The interface to the PDM is provided by the ViC* software [CH97], which allows any number of disks and any number of processors, as long as each is some integer power of 2. Here, we present timings for the dimensional method and the vector-radix algorithm on two platforms:

1. A DEC 2100 server with two 175-MHz Alpha processors and eight 2-gigabyte disks. We use this system as a uniprocessor, in as much as the main thread of control is running on at most one processor at any time, and the other processor may be acting as a server for I/O requests to the eight disks. The ViC* implementation on this system performs all disk I/O through direct UNIX File System calls.

2. A Silicon Graphics Origin 2000 SMP with eight 180-MHz R10000 processors and eight 4-gigabyte disks. Although this machine provides a shared-memory abstraction, we use MPI [GLS94, SOHL+96] for interprocessor communication for three reasons:

   - The memory is actually physically distributed.
   - The MPI implementation is produced by Silicon Graphics and is optimized for the Origin 2000.
   - We can use the same source code on distributed-memory machines.

   On this system, the ViC* implementation performs all disk I/O via the ROMIO implementation of MPI-IO (http://www.mcs.anl.gov/romio/).

The underlying software on the Origin 2000, and so parallel-I/O calls within the BMMC-permutation subroutine are asynchronous, but parallel-I/O calls within the rest of the computation are synchronous.

DEC 2100 results

We performed one set of runs on the DEC 2100 server, in which we varied the input size and kept all other parameters fixed for each of the two methods. In particular, the input

	Dimensional Method		Vector-Radix Algorithm	
$\lg N$	Total time (secs)	Normalized time (μsecs)	Total time (secs)	Normalized time (μsecs)
22	139.00	3.01272	145.95	3.16338
24	621.67	3.08787	647.51	3.21622
26	2983.35	3.41964	3012.33	3.45286
28	12346.20	3.28523	12028.60	3.20072

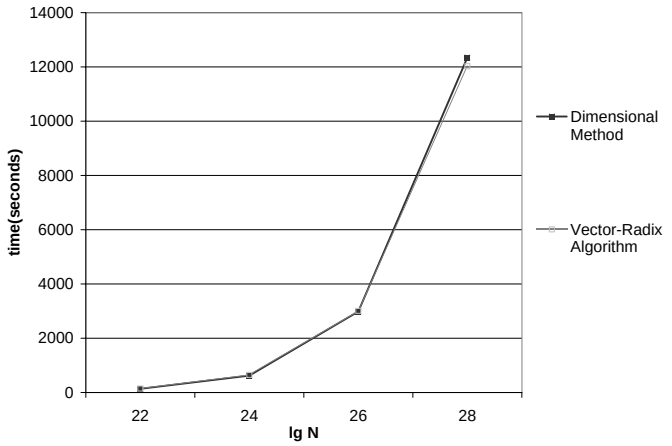

Figure 5.1: Total times and normalized times for the DEC 2100 server.

sizes were $N = 2^{22}$, 2^{24}, 2^{26}, and 2^{28} points, interpreted as 2-dimensional square matrices ($2^{11} \times 2^{11}$, $2^{12} \times 2^{12}$, $2^{13} \times 2^{13}$, and $2^{14} \times 2^{14}$, respectively). Each run used a memory size of 2^{26} bytes (or $M = 2^{20}$ records when we compensate for the data size of 16 bytes per point and for carving memory into four buffers for I/O and in-memory permutations), a block size of $B = 2^{13}$ records, and $D = 8$ disks. Figure 5.1 shows the total times and the normalized times (time per butterfly operation, of which there are $(N/2) \lg N$) for the three problem sizes. On the DEC 2100, it takes just under 3.5 hours to compute a $2^{14} \times 2^{14}$-point FFT using either method. Normalized times vary only by about 13.5% among the four runs of the dimensional method implementation and by about 9% among the four runs of the vector-radix implementation.

Silicon Graphics Origin 2000 results

We performed two sets of runs for the Origin 2000. The first set is similar to the set on the DEC 2100.

In the first set, we varied only the problem size and kept all other parameters fixed for both of the multidimensional algorithms. Here, the problem sizes were $N = 2^{28}$ and

	Dimensional Method		Vector-Radix Algorithm	
$\lg N$	Total time (secs)	Normalized time (μsecs)	Total time (secs)	Normalized time (μsecs)
28	1332.00	0.354435	1308.26	0.348118
30	6137.91	0.381092	6233.21	0.387009

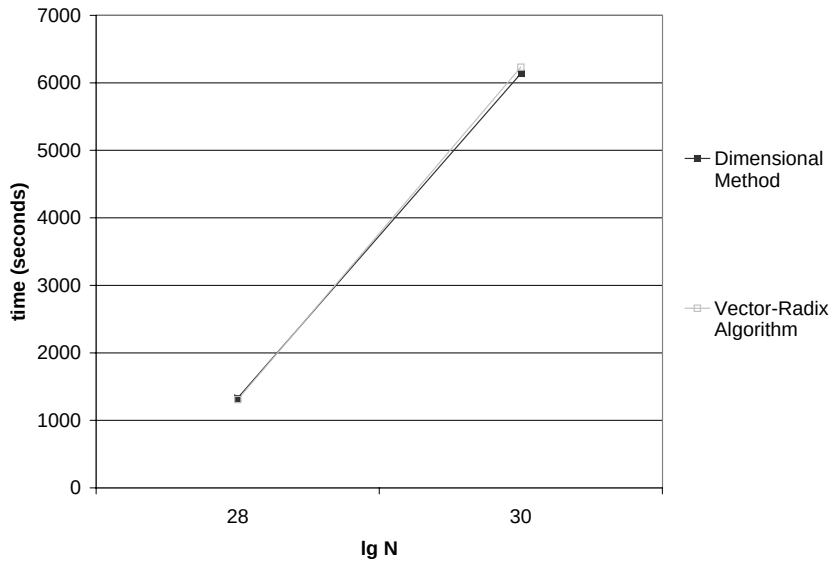

Figure 5.2: Total times and normalized times for the Origin 2000.

2^{30} points, interpreted as $2^{14} \times 2^{14}$ and $2^{15} \times 2^{15}$ matrices, respectively. Each run used a memory size of 2^{28} bytes per processor, or 2^{31} bytes over all eight processors, corresponding to $M = 2^{27}$ records over the entire system. The block size was $B = 2^{13}$ records, and $P = D = 8$. Figure 5.2 shows the total times and the normalized times for the two problem sizes. On the Origin 2000, it takes only about 1.7 hours to compute a $2^{15} \times 2^{15}$-point FFT using either algorithm. Normalized times vary only by about 7.5% between the two runs of the dimensional method implementation and by about 11% between the two runs of the vector-radix implementation.

In the second set of runs on the Origin 2000, we kept the problem size and memory per processor fixed, and we varied the number of processors and disks, maintaining the relationship $P = D$. Here, the problem size was $N = 2^{26}$ points, interpreted as a $2^{13} \times 2^{13}$ matrix. The memory size was 2^{26} bytes per processor. The number of processors varied among 1, 2, 4, and 8. Figure 5.3 shows the results. The speedup appears to be nearly linear in the vector-radix runs, because the work (processors $\times$ total time) is nearly constant across all configurations. However, in the dimensional method runs, the work increases

	Dimensional Method		Vector-Radix Algorithm	
P , D	Total time (secs)	Work (processor-secs)	Total time (secs)	Work (processor-secs)
1	1316.32	1316.32	1443.08	1443.08
2	952.55	1905.09	753.34	1506.68
4	495.16	1980.62	361.54	1446.16
8	212.94	1703.54	183.58	1468.64

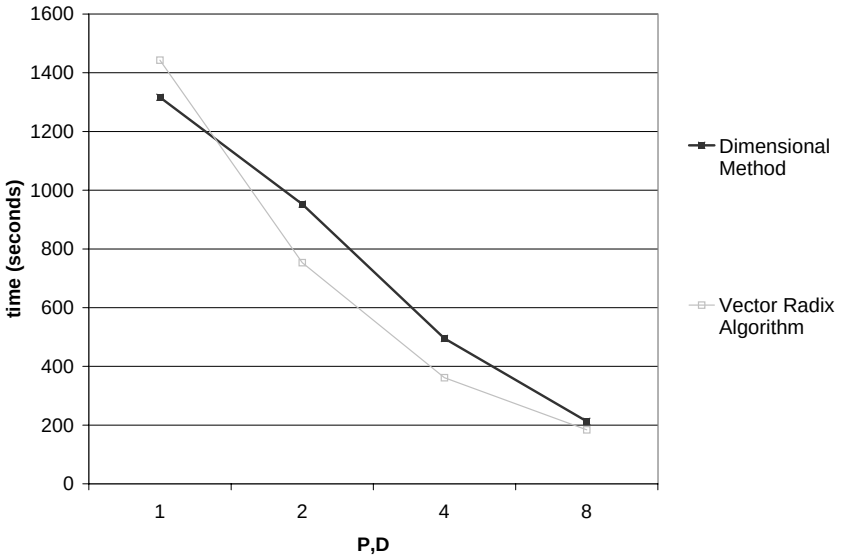

Figure 5.3: Total times and work for the Origin 2000 as the number of processors and disks increases.

sharply between 1 and 2 processors. We suspect that this behavior is due to the additional computation and communication arising in the transition from 1 to 2 processors in the BMMC-permutation subroutine. The breakdown of the timings indicates that the vector-radix method compensates for the increased time spent in communication by significantly decreasing the time spent reading from disk for the FFT computation.

Chapter 6

Conclusion

We have seen two methods for performing multidimensional, multiprocessor out-of-core FFTs with parallel disks. Both are extensions of previous work in the area, and both take advantage of BMMC permutations. We have also presented the exact I/O-complexity analyses and proofs for both FFT methods, along with performance results on two platforms. Among our performance results, we see that we can perform a $2^{15} \times 2^{15}$ out-of-core FFT on an eight-processor Silicon Graphics Origin 2000 in under two hours using either implementation.

From the empirical results, we conclude that in two dimensions, the dimensional method and the vector-radix method are comparable in speed. On some runs, primarily those on a uniprocessor, the dimensional method is faster, whereas on others, including most of our runs on a multiprocessor, vector-radix is faster. On average, when the dimensional method is faster, it is faster by only about 5%. On the other hand, when the vector-radix method is faster, it is faster by about 15%.

We suspect, however, that the vector-radix method may prove to be the more efficient algorithm for higher-dimensional problems. Our ongoing work will determine whether our suspicion is correct. Our reasoning is that the dimensional method computes multiple 1-dimensional FFTs in each dimension, but the vector-radix method processes all dimensions simultaneously. At each stage of the computation, the problem is divided into submatrices, within which we perform butterfly operations. In the Cooley-Tukey algorithm to compute 1-dimensional FFTs, each butterfly has only 2 elements. Correspondingly, when using the vector-radix method to compute a k-dimensional FFT, each butterfly consists of 2^k elements. We wonder whether, by working on more data at once, the vector-radix method enjoys computational efficiencies and performs fewer passes over the data.

Compared to the vector-radix method, the dimensional method has certain advantages. It is relatively simple to implement given the existing unidimensional FFT and BMMC permutation codes. It works for any number of dimensions and for arbitrary dimension sizes, as long as they are integer powers of 2. The vector-radix method, on the other hand, is much more difficult to implement correctly. In particular, handling arbitrary numbers of dimensions and unequal dimension sizes is tricky, and computing the twiddle factors correctly and efficiently is very difficult.

Acknowledgments

I thank Michael Shin and Neal Young for suggesting the proof technique described in Section 1.3. Dan Rockmore, Matteo Frigo, Steven Johnson, and Hany Farid provided valuable background information on multidimensional FFTs. I also appreciate the constructive suggestions made by anonymous reviewers on an early version of [BC99], on which some of this thesis is based.

I thank my parents for their encouragement throughout the course of this project. Despite having no formal training in Computer Science, they pleaded for drafts of my thesis and followed up with questions about matrix compositions and row permutations. I am always touched by my mother's memory for algorithmic epiphanies, and even frustrating bugs.

Finally, I would like to thank Professor Tom Cormen, who has been my advisor almost from the time I began programming. Over the past couple of years, he has seen me every step of the way. He taught me C++, got me involved in his research, encouraged me to apply to graduate school, and most importantly gave me the opportunity to work with him. I am grateful to him for the countless hours he has invested in me and in this work. I know that the relief I shall feel upon finishing this thesis will be tempered by the knowledge that it brings to a close his role as my undergraduate advisor. It has been a fun two years. Thank you.

Bibliography

[BC99] Lauren M. Baptist and Thomas H. Cormen. Multidimensional, multiprocessor, out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 242–250, June 1999.

[CC99] Thomas H. Cormen and James C. Clippinger. Performing BMMC permutations efficiently on distributed-memory multiprocessors with MPI. *Algorithmica*, 24(3/4):349–370, July/August 1999.

[CH97] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, June 1997.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[CN97] Thomas H. Cormen and David M. Nicol. Out-of-core FFTs with parallel disks. *ACM SIGMETRICS Performance Evaluation Review*, 25(3):3–12, December 1997.

[CN98] Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 24(1):5–20, January 1998.

[Cor99] Thomas H. Cormen. Determining an out-of-core FFT decomposition strategy for parallel disks by dynamic programming. In Michael T. Heath, Abhiram Ranade, and Robert S. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105 of *IMA Volumes in Mathematics and its Applications*, pages 307–320. Springer-Verlag, 1999.

[CSW99] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1999.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CWN97] Thomas H. Cormen, Jake Wegmann, and David M. Nicol. Multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS '97)*, pages 68–78, November 1997. Also Dartmouth College Computer Science Technical Report PCS-TR97-303.

[DM84] Dan E. Dudgeon and Russell M. Mersereau. *Multidimensional Digital Signal Processing*. Prentice-Hall, 1984.

[Far99] Hany Farid. Private communication, March 1999.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[HMCS77] D. B. Harris, J. H. McClellan, D. S. K. Chan, and H. W. Schuessler. Vector radix fast Fourier transform. In *1977 IEEE International Conference on Acoustics, Speech, Signal Process Rec.*, pages 548–551, 1977.

[Lim90] Jae S. Lim. *Two-Dimensional Signal and Image Processing*. Prentice-Hall, 1990.

[Riv77] Glenn E. Rivard. Direct fast Fourier transform of bivariate functions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-25(3):250–252, June 1977.

[Sni81] M. Snir. I/O limitations on multi-chip VLSI systems. In *Proceedings of the 19th Allerton Conference on Communication, Control and Computation*, pages 224–233, 1981.

[SOHL+96] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[Van92] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia, 1992.

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.