Dartmouth College Computer Science Technical Report TR2001-388

Applying the Vector Radix Method to Multidimensional, Multiprocessor, Out-of-Core Fast Fourier Transforms

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Michael F. Ringenburg

Dartmouth College

Hanover, New Hampshire

March 19, 2001

Examining Committee:

(chair) Thomas Cormen

Hany Farid

John Mackey

Roger Sloboda
Dean of Graduate Studies

Abstract

We describe an efficient algorithm for calculating Fast Fourier Transforms on matrices of arbitrarily high dimension using the vector-radix method when the problem size is out-of-core (i.e., when the size of the data set is larger than the total available memory of the system). The algorithm takes advantage of multiple processors when they are present, but it is also efficient on single-processor systems. Our work is an extension of work done by Lauren Baptist in [Bapt99], which applied the vector-radix method to 2-dimensional out-of-core matrices.

To determine the effectiveness of the algorithm, we present empirical results as well as an analysis of the I/O, communication, and computational complexity. We perform the empirical tests on a DEC 2100 server and on a cluster of Pentium-based Linux workstations. We compare our results with the traditional dimensional method of calculating multidimensional FFTs, and show that as the number of dimensions increases, the vector-radix-based algorithm becomes increasingly effective relative to the dimensional method.

In order to calculate the complexity of the algorithm, it was necessary to develop a method for analyzing the interprocessor communication costs of the BMMC data-permutation algorithm (presented in [CSW98]) used by our FFT algorithms. We present this analysis method and show how it was derived.

Contents

1 Introduction **4**

1.1 Parallel Disk Model 5

1.2 BMMC permutations 7

1.3 Out-of-core vector-radix FFTs 8

1.3.1 Vector-radix algorithm 8

1.3.2 Multiprocessor out-of-core adaptations 12

1.4 Contributions of this thesis 15

2 The vector-radix method in arbitrarily many dimensions **16**

2.1 The in-core vector-radix method in higher dimensions 17

2.1.1 Butterfly operations 19

2.2 The out-of-core vector-radix method in higher dimensions 23

2.2.1 The k-dimensional bit-reversal permutation 24

2.2.2 Threader and inverse-threader permutations 24

2.2.3 The k-dimensional bit-rotation permutations 26

2.2.4 Implementation details 27

3 Analysis of the higher-dimensional algorithm **29**

3.1 BMMC-permutation communication cost analysis 30

3.2 Complexity of the higher-dimensional algorithm 35

4 Empirical results of the higher-dimensional vector-radix algorithm **38**

4.1 DEC 2100 server results 39

4.2 Linux cluster results 40

4.3 Discussion 42

5 Extending the out-of-core vector-radix method to non-square matrices **44**

5.1 The in-core non-square vector-radix method 45

5.1.1 The problem 45

5.1.2 The solution 46
5.2 The out-of-core non-square vector-radix method 48
5.2.1 BMMC permutations 49
5.2.2 Complete algorithm 50

6 Conclusion **52**

List of Figures

1.1 The Vitter-Shriver Parallel Disk Model 6

1.2 Stripe-major layout 7

1.3 Processor-major layout 12

1.4 Bit-reversal permutation of an 8×8 matrix 14

2.1 Parity submatrices of a $4 \times 4 \times 4$ matrix 18

2.2 3-dimensional butterfly operations 22

2.3 Threader and k-dimensional bit-rotation permutations of a $4 \times 4 \times 4$ matrix . 26

4.1 Results on a DEC2100 server with $N = 2^{24}$ 39

4.2 Results on a DEC2100 server with $N = 2^{28}$ 40

4.3 Results on a Linux cluster with $N = 2^{24}$ 41

4.4 Results on a Linux cluster with $N = 2^{28}$ 41

4.5 Results on a Linux cluster with $N = 2^{30}$ 42

Chapter 1

Introduction

A basic mathematical result states that any wave signal can be represented as a sum of sine and cosine waves of varying frequencies and amplitudes. The Discrete Fourier Transform allows us to determine this representation from a sampling of discrete points on the wave signal. Fast Fourier Transform algorithms are simply algorithms that efficiently compute the Discrete Fourier Transform, typically in $\Theta(N \lg N)$ time, where N is the number of sampled points.

Fast Fourier Transform (FFT) calculations have applications in a wide variety of arenas. Geophysicists use them for seismic analysis. Quantum physicists calculate the FFT of the wave function describing a subatomic particle to determine the probability density for the momentum of the particle [French61]. Molecular physicists use FFTs for nuclear magnetic resonance spectroscopy [HW95]. Engineers use them for acoustics and signal processing. Computer scientists use them for image processing. Even X-Files fans calculate FFTs of radio telescope data to search for alien life on their home computers as part of the SETI@Home project [Bapt99].

Previous work presents efficient algorithms for the computation of FFTs in a variety of settings. [Bing74] gives a nice overview of basic FFT techniques. Standard algorithms texts, such as [CLR90], describe the Cooley-Tukey algorithm for the computation of 1-dimensional FFTs. Van Loan, in [Van92], presents several other algorithms for the efficient calculation of Fourier Transforms. The vector-radix algorithm, first proposed in [Riv77], provides a method for computing 2-dimensional FFTs with 25% fewer complex multiplications [Lim90] than the traditional *dimensional method.*[1]

More recent work presents efficient algorithms for computing out-of-core FFTs. A problem is considered *out-of-core* when the size of the data set exceeds the capacity of the

[1] The dimensional method of calculating a 2-dimensional FFT performs 1-dimensional FFTs on each row and each column of the data matrix.

system's main memory. Programmers often attempt to solve such problems by computing "windowed" FFTs (i.e., FFTs of subsets of the data) and averaging the results. However, these "windowed" FFT's fail to include the low-frequency components of the signal whose periods are larger than the size of the window. In [CN98], the authors solve this problem by showing how to efficiently calculate uniprocessor out-of-core 1-dimensional FFTs. These techniques are extended to multiprocessor systems in [CWN97]. Lauren Baptist, in [Bapt99], shows how to apply the vector-radix method to multiprocessor out-of-core problems when the data matrix is square and 2-dimensional.

The present paper extends Baptist's work on multiprocessor out-of-core vector-radix FFTs to higher-dimensional and non-square matrices. In Chapter 2, we describe the extension to higher-dimensional matrices. Chapter 3 analyzes the I/O, communication, and computational costs of the higher-dimensional algorithm. The analysis in Chapter 3 required the development of a new method to calculate the interprocessor communication cost of the BMMC data-permutation algorithm in [CSW98], and we spend the majority of the chapter developing and explaining our technique. In Chapter 4, we present empirical results of the vector-radix and dimensional algorithms on two systems—a DEC 2100 server and a cluster of Pentium-based Linux workstations. Chapter 5 explains how to extend the vector-radix method to non-square matrices and discusses why the vector-radix method is not as efficient in this situation. Chapter 6 presents some concluding remarks.

The remainder of this chapter provides some necessary background material. We first discuss the computational model assumed by our algorithm. We then briefly describe BMMC permutations and conclude by explaining the basic vector-radix algorithm and the adaptations Baptist made in order to efficiently apply it to out-of-core problems.

1.1 Parallel Disk Model

To describe input and output, we use the *Parallel Disk Model*, or *PDM*, first proposed by Vitter and Shriver [VS94] and illustrated in Figure 1.1. In the Vitter-Shriver model, 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. The records on each disk are organized in *blocks* of B records each. When a disk is read from or written to, an entire block of records is transferred. Disk I/O transfers records between the disks and a *random-access memory* (which we shall refer to simply as "memory") capable of holding M records. 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.

We measure an algorithm's efficiency by the number of parallel I/O operations it requires.

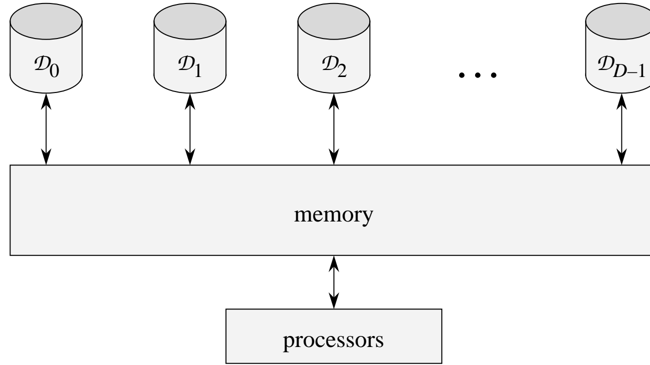


Figure 1.1: The three levels of the Vitter-Shriver Parallel Disk Model. The first level consists of D disks, labeled above as $\mathcal{D}_0$ through $\mathcal{D}_{D-1}$, which store an N-record problem. The next level, the memory, stores up to M records at a time. Records are transferred between the disk level and the memory level through parallel I/O operations. Each parallel I/O operation transfers B records from each disk, for a total of BD records per parallel I/O operation. The third level, the processor level, performs computations on the data stored in the memory level.

Although this cost model does not account for the variation in disk access times caused by head movement and rotational latency, programmers often have no control over these factors. The number of disk accesses, however, can be minimized by carefully designed algorithms.

For convenience, we use the following notation extensively:

$$b = \lg B, \quad d = \lg D, \quad m = \lg M, \quad n = \lg N.$$

We shall assume that b, d, m, and n are nonnegative integers, which implies that B, D, M, and N are exact powers of 2. In order for the memory to accomodate the records transferred in a parallel I/O operation to all D disks, we require that $BD \leq M$. Also, we assume that $M < N$ and the $D \geq P$ (i.e., each processor has at least one dedicated disk). Neither of these assumptions restrict us. If $M \geq N$, we can perform all operations in memory, and thus have no use for the PDM. If $D < P$, we can simulate $D \geq P$ by partitioning each disk into P/D partitions and allocating one partition to each processor. The above requirements imply that $b + d \leq m < n$.

The PDM lays out data on a parallel disk system as shown in Figure 1.2. A *stripe* consists of the D blocks at the same location on all D disks. We indicate the *address*, or *index*, of a record as an n-bit vector x with the least significant bit first: $x = (x_0, x_1, \ldots, x_{n-1})$. Record indices vary most rapidly within a block, then among disks, and finally among stripes. The offset within the block is given by the least significant b bits $x_0, x_1, \ldots, x_{b-1}$, the disk number by the next d bits $x_b, x_{b+1}, \ldots, x_{b+d-1}$, and the stripe number by the $s = n - (b + d)$ most significant bits $x_{b+d}, x_{b+d+1}, \ldots, x_{n-1}$.

When describing the in-core versions of the algorithms in this paper, we often deviate

	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$	
stripe 0	0	1	2	3	4	5	6	7
stripe 1	8	9	10	11	12	13	14	15
stripe 2	16	17	18	19	20	21	22	23
stripe 3	24	25	26	27	28	29	30	31

Figure 1.2: The layout of $N = 32$ records in a parallel disk system with $B = 2$ and $D = 4$. Each box represents one block, each column represents one disk, and each row represents a stripe. There are N/BD stripes(4 in this example). Numbers indicate record indices.

slightly from the above conventions for simplicity. The correctness proofs and the derivations of the butterfly operations are usually neater when we let our data matrix be of size $N \times N \times \cdots \times N$ rather than of size $N^{1/dims} \times N^{1/dims} \times \cdots \times N^{1/dims}$ (i.e., N total elements). However when describing the out-of-core algorithms, we always use the PDM convention that the data matrix has N elements in all. We shall always state explicitly what size data matrix we are assuming.

The preceeding description of the Parallel Disk Model was taken largely from [CSW98].

1.2 BMMC permutations

All the out-of-core algorithms discussed in this paper make use of *BMMC (bit-matrix-multiply/complement) permutations.* A BMMC 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 is nonsingular (i.e., invertible) over $GF(2)$.[2] Treating each source index x as an n-bit vector, we perform matrix-vector multiplication[3] over $GF(2)$ to produce an 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. Because multiplying nonsingular matrices yields a nonsingular matrix, the class of BMMC permutations is closed under composition [CSW98]. Thus performing, in order, the three permutations characterized by the matrices H_1, H_2, and H_3 would have the same result as performing the single permutation characterized by the matrix $H_3 H_2 H_1$. More information on BMMC permutations can be found in [CC99] and [CSW98].

[2] 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.

[3] Technically, the definition of a BMMC permutation requires an n-bit "complement vector" c, and $z = H\,x \oplus c$. All BMMC permutations used in this paper have a complement vector of 0, and so we ignore complement vectors.

1.3 Out-of-core vector-radix FFTs

1.3.1 Vector-radix algorithm

The vector-radix algorithm for 2-dimensional FFTs is a logical extension of the 1-dimensional Cooley-Tukey algorithm. The Cooley-Tukey algorithm divides an N-element vector A into two $N/2$-element subvectors A_{even} and A_{odd}. The vector A_{even} contains the even-indexed elements of A, and A_{odd} contains the odd-indexed elements. The algorithm computes the FFT of the two subvectors recursively and then combines the results. Likewise, the vector-radix algorithm divides an $N \times N$ matrix T into four $N/2 \times N/2$ submatrices T_{ee}, T_{eo}, T_{oe} and T_{oo}. The submatrix T_{ee} contains the elements of T with even row and even column indices, T_{eo} contains the elements with even row and odd column indices, T_{oe} contains the elements with odd row and even column indices, and T_{oo} contains the elements with odd row and odd column indices. We shall refer to these submatrices as the *parity submatrices* of the data matrix.[4] The algorithm recursively computes the FFT of each of the four submatrices and then combines the results. We define a *level* of the FFT computation to be one of these $n = \lg N$ recursive calculations.

Note that we are making the assumption here that the dimension sizes are powers of 2. Many FFT algorithms make the same assumption, and we continue to do so throughout this paper. Implementers use a variety of techniques to work around the power-of-2 size requirements. [SS95] and [OS75] discuss the pros and cons of *zero-padding*—adding 0's to the end of the input array until the size is a power of 2. In many applications, however, a better option for padding to a power-of-2 size is *mirroring*—converting the set a_j for $j = 0, 1, \ldots, n-1$ into the set

$$a'_j = \begin{cases} a_j & \text{if } j < n \, , \\ a_{2n-j-1} & \text{if } j \geq n \, , \end{cases}$$

for $j = 0, 1, \ldots, n' - 1$ where n' is the smallest integer power of 2 greater than n. For algorithms that do not require power-of-2 dimension sizes, see Chapter 2 of [Van92].

To show that a 2-dimensional FFT can be computed by dividing the data matrix into parity submatrices, we simply need to examine the equation for 2-dimensional FFTs. If our data is an $N \times N$ square matrix f, then the FFT matrix F is defined by

$$F(x,y) = \sum_{\alpha_1=0}^{N-1} \sum_{\alpha_2=0}^{N-1} f(\alpha_1, \alpha_2) \omega_N^{x\alpha_1} \omega_N^{y\alpha_2}$$

where $x, y = 0, 1, \ldots, N-1$ and $\omega_N^a = e^{-2a\pi i/N} = \cos(2a\pi/N) - i \sin(2a\pi/N)$. We refer to powers of ω_N as *twiddle factors*. Note that because of the periodicity of sine and cosine,

$$\omega_N^a = \omega_N^{a+N} \, .$$

[4] Some suggest that we instead refer to them as "toe" matrices.

Similarly,

$$\omega_N^a \;=\; \omega_{CN}^{Ca}$$

for any real number C,

$$\omega_N^{N/2} \;=\; -1\;,$$

and

$$\omega_N^N = 1\;.$$

Multiplying twiddle factors, we obtain

$$F(x,y) = \sum_{\alpha_1=0}^{N-1}\sum_{\alpha_2=0}^{N-1} f(\alpha_1,\alpha_2)\omega_N^{x\alpha_1+y\alpha_2}\;.$$

Splitting both summations according to the parity of their indices, we have

$$\begin{aligned} F(x,y) \;=\; & \sum_{\alpha_1\,\text{even}}\;\sum_{\alpha_2\,\text{even}} f(\alpha_1,\alpha_2)\omega_N^{x\alpha_1+y\alpha_2} + \\ & \sum_{\alpha_1\,\text{even}}\;\sum_{\alpha_2\,\text{odd}} f(\alpha_1,\alpha_2)\omega_N^{x\alpha_1+y\alpha_2} + \\ & \sum_{\alpha_1\,\text{odd}}\;\sum_{\alpha_2\,\text{even}} f(\alpha_1,\alpha_2)\omega_N^{x\alpha_1+y\alpha_2} + \\ & \sum_{\alpha_1\,\text{odd}}\;\sum_{\alpha_2\,\text{odd}} f(\alpha_1,\alpha_2)\omega_N^{x\alpha_1+y\alpha_2}\;. \end{aligned} \tag{1.1}$$

We now introduce the concept of a parity vector. Parity vectors are not necessary to explain the 2-dimensional algorithm. However they become important when we explain the higher-dimensional vector-radix algorithm in the next chapter. We present the parity vector treatment of the 2-dimensional problem here in parallel with the standard treatment to show the relationship between the two approaches and to clarify the definition and use of parity vectors.

We define a *k-bit parity vector* to be a vector of the form $p = (p_1, p_2, \ldots, p_k)$ where all p_i are drawn from the set $\{0,1\}$. We use parity vectors to represent the parity of record indices in each dimension. If the parity bit p_i is 0, we are referring to records with even indices in dimension i, and if p_i is 1 we are working with records with odd indices in dimension i. We also define Ψ^k to be the set of all k-bit parity vectors. Note that the cardinality of Ψ^k is simply 2^k. Using parity vectors, we can simplify Equation 1.1 to

$$F(x,y) = \sum_{p\in\Psi^2}\left(\sum_{\alpha_1=0}^{N/2-1}\sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1, 2\alpha_2+p_2)\omega_N^{(2\alpha_1+p_1)x+(2\alpha_2+p_2)y}\right)\;. \tag{1.2}$$

We now calculate the FFT of the parity submatrices. As an intermediate step, we first define the $N \times N$ matrices

$$\begin{aligned}
F_{ee}(x,y) &= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1, 2\beta_2)\omega_{N/2}^{x\beta_1+y\beta_2} \\
&= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1, 2\beta_2)\omega_{N}^{2x\beta_1+2y\beta_2}, \\
F_{eo}(x,y) &= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1, 2\beta_2+1)\omega_{N/2}^{x\beta_1+y\beta_2} \\
&= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1, 2\beta_2+1)\omega_{N}^{2x\beta_1+2y\beta_2}, \\
F_{oe}(x,y) &= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1, 2\beta_2)\omega_{N/2}^{x\beta_1+y\beta_2} \\
&= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1, 2\beta_2)\omega_{N}^{2x\beta_1+2y\beta_2}, \\
F_{oo}(x,y) &= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1, 2\beta_2+1)\omega_{N/2}^{x\beta_1+y\beta_2} \\
&= \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1, 2\beta_2+1)\omega_{N}^{2x\beta_1+2y\beta_2},
\end{aligned}$$

where $x, y = 0, 1, \ldots, N-1$. Alternatively, using parity vectors, we can condense our notation for these matrices as

$$\begin{aligned}
\forall p \in \Psi^2 \qquad F_p(x,y) &= \sum_{\alpha_1=0}^{N/2-1}\sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1, 2\alpha_2+p_2)\omega_{N/2}^{\alpha_1 x+\alpha_2 y} \\
&= \sum_{\alpha_1=0}^{N/2-1}\sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1, 2\alpha_2+p_2)\omega_{N}^{2\alpha_1 x+2\alpha_2 y}
\end{aligned}$$

where once again $x, y = 0, 1, \ldots, N-1$. We shall refer to these matrices as *expanded Fourier parity matrices* (or *expanded matrices* for short), because each matrix contains four copies (one in each quadrant) of the Fourier Transform of the corresponding parity submatrix, e.g., F_{ee} contains four copies of the FFT of T_{ee}, etc. To verify this statement, note that if we restrict the domains of x and y to $0, 1, \ldots, N/2-1$, the expanded matrix equations become the equations for the FFTs of the parity submatrices. This observation implies that the first quadrant of each expanded matrix contains the FFT of the corresponding parity submatrix. To show that the expanded matrices in fact contain four copies of the Fourier Transforms, we simply note that the equations are periodic with period $N/2$. Thus all four quadrants of the expanded matrices are identical.

These definitions allow us to state the FFT formula in terms of the FFTs of the parity

submatrices. First, consider our split-summation FFT formula (1.1), which can be rewritten as

$$\begin{aligned}F(x,y) \;=\; & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1,2\beta_2)\omega_N^{2\beta_1 x+2\beta_2 y} + \\ & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1,2\beta_2+1)\omega_N^{2\beta_1 x+(2\beta_2+1)y} + \\ & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1,2\beta_2)\omega_N^{(2\beta_1+1)x+2\beta_2 y} + \\ & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1,2\beta_2+1)\omega_N^{(2\beta_1+1)x+(2\beta_2+1)y} \;.\end{aligned}$$

We can make the twiddle factors match the twiddle factors of the expanded matrices by writting this equation as

$$\begin{aligned}F(x,y) \;=\; & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1,2\beta_2)\omega_N^{2\beta_1 x+2\beta_2 y} + \\ & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1,2\beta_2+1)\omega_N^{2\beta_1 x+2\beta_2 y}\omega_N^{y} + \\ & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1,2\beta_2)\omega_N^{2\beta_1 x+2\beta_2 y}\omega_N^{x} + \\ & \sum_{\beta_1=0}^{N/2-1}\sum_{\beta_2=0}^{N/2-1} f(2\beta_1+1,2\beta_2+1)\omega_N^{2\beta_1 x+2\beta_2 y}\omega_N^{x+y} \;.\end{aligned}$$

Our parity vector split-summation equation (1.2) can be rewritten as

$$F(x,y)=\sum_{p\in\Psi^2}\left(\sum_{\alpha_1=0}^{N/2-1}\sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1,2\alpha_2+p_2)\omega_N^{2\alpha_1 x+2\alpha_2 y}\omega_N^{p_1 x+p_2 y}\right) \;.$$

Substituting the expanded matrices into either of these equations gives us the formula

$$F(x,y)=F_{ee}(x,y)+F_{eo}(x,y)\omega_N^{y}+F_{oe}(x,y)\omega_N^{x}+F_{oo}(x,y)\omega_N^{x+y} \;.$$

Because of the periodicity of the F_{xy} matrices and because $\omega_N^{N/2} = -1$, we obtain the identities

$$\begin{aligned}F(x,y) &= F_{ee}(x,y)+F_{eo}(x,y)\omega_N^{y}+F_{oe}(x,y)\omega_N^{x}+F_{oo}(x,y)\omega_N^{x+y} \;, \\ F(x+N/2,y) &= F_{ee}(x,y)+F_{eo}(x,y)\omega_N^{y}-F_{oe}(x,y)\omega_N^{x}-F_{oo}(x,y)\omega_N^{x+y} \;, \\ F(x,y+N/2) &= F_{ee}(x,y)-F_{eo}(x,y)\omega_N^{y}+F_{oe}(x,y)\omega_N^{x}-F_{oo}(x,y)\omega_N^{x+y} \;, \\ F(x+N/2,y+N/2) &= F_{ee}(x,y)-F_{eo}(x,y)\omega_N^{y}-F_{oe}(x,y)\omega_N^{x}+F_{oo}(x,y)\omega_N^{x+y} \;.\end{aligned}$$

	P_0				P_1			
	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$	
stripe 0	0	1	2	3	16	17	18	19
stripe 1	4	5	6	7	20	21	22	23
stripe 2	8	9	10	11	24	25	26	27
stripe 3	12	13	14	15	28	29	30	31

Figure 1.3: The ordering of $N = 32$ records in a processor-major layout with $P = 2$, $B = 2$, and $D = 4$. Each box represents one block, each row represents one stripe, and each column represents one disk. Numbers indicate record indices.

Rivard, in [Riv77], shows how these identities can be computed efficiently using *butterfly operations*. We recursively compute the four submatrices, and use the butterfly operations to combine the results. For more information on the vector-radix method, refer to [DM84], [Lim90], [MS81], [Riv77] and [WP89].

1.3.2 Multiprocessor out-of-core adaptations

In order to efficiently apply the vector-radix method to a multiprocessor out-of-core problem, we must introduce the concept of a superlevel [Bapt99]. As described above, the vector-radix method divides the FFT computation of an $\sqrt{N} \times \sqrt{N}$ matrix into $n/2$ levels. We define a *superlevel* to be a series of up to $m/2$ consecutive levels of the FFT computation that are computable in a single *pass* over the data, where a pass consists of reading each stripe from disk exactly once. The out-of-core vector-radix algorithm divides the FFT computation into $\lceil n/m \rceil$ superlevels. Each of the first $\lfloor n/m \rfloor$ superlevels contains $m/2$ levels. If $\lfloor n/m \rfloor \neq \lceil n/m \rceil$ the final superlevel contains $(n/2) \bmod (m/2)$ levels.

Before and after each superlevel, Baptist's algorithm reorders the records in the data matrix [Bapt99]. This rearrangement allows us to compute the next superlevel in a single pass over the data. We perform the data reordering with a series of BMMC permutations. As stated earlier, the closure property of the BMMC-permutation class allows us to combine consecutive permutations into a single permutation. Thus we need only perform one BMMC permutation between each superlevel.

Baptist [Bapt99] begins with an initial *2-dimensional bit-reversal* permutation, with characteristic matrix

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

where I represents an identity submatrix, I^A represents an anti-identity submatrix, and

0 represents a zero submatrix. The 2-dimensional bit-reversal permutation groups each parity submatrix into a single quadrant. The permutation works recursively as well—i.e., the parity submatrices of parity submatrices are grouped into quadrants of quadrants, as in Figure 1.4. Because of this recursive property of the bit-reversal permutation, at each level the input points of the butterfly operations described above will be (x, y), $(x + N'/2, y)$, $(x, y + N'/2)$ and $(x + N'/2, y + N'/2)$, where N' is the length of a side of the submatrix we are currently recursively computing. Note that these points are also the output points of the butterfly operations, and so we can perform the reads and writes in the same memoryload.

Before each superlevel, Baptist performs two rearrangements of the data [Bapt99]. She first permutes the data from the standard PDM layout (see Figure 1.2), also known as *stripe-major layout*, to a layout that is more convenient to work with, known as *processor-major layout* (see Figure 1.3). In processor-major layout, each processor's memory contains N/P consecutive points, whereas in stripe-major layout a processor has contiguous segments of data containing only BD/P points. A *stripe-to-processor-major* permutation, with charactersitic matrix

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

accomplishes this rearrangement. Next, a *partial bit-rotation* permutation, with characteristic matrix

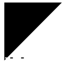
$$\begin{array}{c} \begin{array}{ccc} \frac{m-p}{2} & \frac{n-m+p}{2} & \frac{n}{2} \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 0 & I \\ \hline 0 & I & 0 \end{array}\right] \begin{array}{l} \frac{m-p}{2} \\ \frac{n}{2} \\ \frac{n-m+p}{2} \end{array} \end{array},$$

groups the data points in each parity submatrix into consecutive disk addresses so that they can be loaded into memory by reading consecutive stripes.

After each superlevel, the algorithm rearranges the data in three ways [Bapt99]. An *inverse partial bit-rotation* permutation, with characteristic matrix

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

returns the data points to their positions prior to the partial bit-rotation. After the inverse



Using the above described methods, Baptist sucessfully implemented the vector-radix method for multiprocessor out-of-core problems. She provides further details and the results of analysis and empirical comparisons with the traditional dimensional method in [Bapt99]. Her work, however, restricts the input to 2-dimensional square matrices. The present paper removes those restrictions and analyzes the results.

1.4 Contributions of this thesis

In this chapter, we reviewed previous work involving multi-dimensional and out-of-core Fast Fourier Transforms. In upcoming chapters, we present the following new contributions:

- We extend the vector-radix method to matrices of arbitrarily high dimension.
- We develop an algorithm for efficiently applying the higher-dimensional vector-radix method to out-of-core problems.
- We develop an algorithm for efficiently applying the non-square vector-radix method to out-of-core problems.
- We analyze the communication cost of the out-of-core BMMC-permutation algorithm in [CSW98].
- We derive analytical and empirical results for the higher-dimensional out-of-core vector-radix algorithm.

Chapter 2

The vector-radix method in arbitrarily many dimensions

Higher-dimensional Fast Fourier Transforms have numerous applications. For example, the analysis of seismic wave data requires four dimensions: X, Y, Z, and time. If we consider the complete motion of an earthquake as a wave, taking the Fourier Transform seperates the wave into its sine and cosine components. These components are the short and long period waves we often hear discussed in news coverage of major earthquakes.

Generalizing to higher dimensions has theoretical interest as well. Baptist's empirical test runs showed that the vector-radix method was only marginally more efficient than the dimensional method for 2-dimensional FFTs [Bapt99]. However, intuition suggests that the benefit of the vector-radix method should be more pronounced in higher dimensions. The vector-radix method requires a single pass over the data to compute each of the $\lceil n/m \rceil$ superlevels, regardless of the number of dimensions. The dimensional method, as presently implemented, requires a separate pass for every dimension.[1] Thus, the number of passes performed by the vector-radix method increases as we increase N or decrease M, whereas the number of passes performed by the dimensional method increases as we increase the number of dimensions.

We begin our discussion of the higher-dimensional algorithm by showing how the vector-radix method can be extended to arbitrarily many dimensions. We first prove that the mathematics extend in a natural way. We then describe an efficient method for calculating the newly derived butterfly operations with a minimal number of additions and multiplications.

[1] Jeremy Fineman, however, is presently working on a method to combine multiple dimensions into a single pass. [Cor00].

After showing how the vector-radix method extends to higher dimensions, we decribe our algorithm for efficiently applying the vector-radix method to higher-dimensional out-of-core problems. The algorithm is based on Baptist's 2-dimensional algorithm [Bapt99], described earlier. We begin by showing how the calculations can be broken into superlevels. We next describe the data rearrangements and corresponding BMMC permutations necessary between superlevels. We conclude with a few implementation details.

2.1 The in-core vector-radix method in higher dimensions

We can extend the vector-radix algorithm for 2-dimensional FFTs in a logical manner to handle higher-dimensional FFTs. Recall that the 2-dimensional vector-radix algorithm divides a $\sqrt{N} \times \sqrt{N}$ matrix f into four $\sqrt{N}/2 \times \sqrt{N}/2$ parity submatrices. The algorithm recursively computes the FFT of each of the four submatrices and then combines the results. Likewise, the k-dimensional algorithm splits an $N^{1/k} \times N^{1/k} \times \cdots \times N^{1/k}$ matrix into 2^k $N^{1/k}/2 \times N^{1/k}/2 \times \cdots \times N^{1/k}/2$ parity submatrices, once again based on the parity of the coordinates of each element. For an example in 3 dimensions, see Figure 2.1. The submatrices are then recursively computed and the results combined via butterfly operations. As before, we refer to each of these recursive calculations as a *level* of the FFT computation. Note that each recursive computation divides the size of each dimension by 2. The original size of each dimension is $N^{1/k}$. Thus we have a total of $\lg N^{1/k} = (\lg N)/k = n/k$ levels.

To prove that we can compute a higher-dimensional FFT by dividing the data matrix into 2^k submatrices, we now examine the equation for k-dimensional FFTs. If our data is an $N \times N \times \cdots \times N$ square matrix f, then the equation for the FFT matrix F is

$$F(x_1, x_2, \ldots, x_k) = \sum_{\alpha_1=0}^{N-1} \sum_{\alpha_2=0}^{N-1} \cdots \sum_{\alpha_k=0}^{N-1} f(\alpha_1, \alpha_2, \ldots, \alpha_k) \omega_N^{x_1\alpha_1} \omega_N^{x_2\alpha_2} \cdots \omega_N^{x_k\alpha_k} ,$$

where $x_1, x_2, \ldots, x_k = 0, 1, \ldots, N-1$ and $\omega_N^a = e^{-2a\pi i/N} = \cos(2a\pi/N) - i\sin(2a\pi/N)$. Recall the following identities from the introduction:

$$\begin{aligned} \omega_N^a &= \omega_N^{a+N} , \\ \omega_N^a &= \omega_{CN}^{Ca} , \\ \omega_N^{N/2} &= -1 , \\ \omega_N^N &= 1 . \end{aligned}$$

Combining the ω terms (twiddle factors) in the FFT equation, we obtain

$$F(x_1, x_2, \ldots, x_k) = \sum_{\alpha_1=0}^{N-1} \sum_{\alpha_2=0}^{N-1} \cdots \sum_{\alpha_k=0}^{N-1} f(\alpha_1, \alpha_2, \ldots, \alpha_k) \omega_N^{(x_1\alpha_1 + x_2\alpha_2 + \cdots + x_k\alpha_k)} .$$

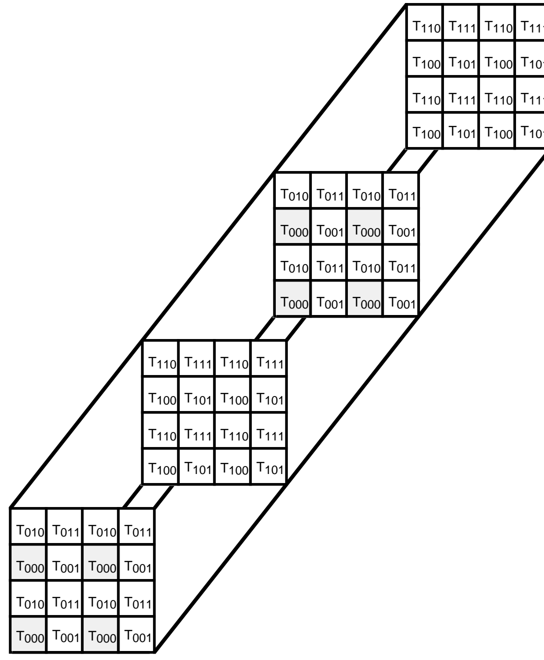


Figure 2.1: Partitioning a $4 \times 4 \times 4$ matrix into parity submatrices. Each box represents an element of the matrix. Each element's label indicates its parity submatrix. The notation T_p indicates the parity submatrix corresponding to parity vector p.

Next we split each of the k summations by parity and, using parity vector notation, derive

$$F(\vec{x}) = \sum_{p \in \Psi^k} \left(\sum_{\alpha_1=0}^{N/2-1} \cdots \sum_{\alpha_k=0}^{N/2-1} f(2\alpha_1 + p_1, \ldots, 2\alpha_k + p_k) \omega_N^{(2\alpha_1+p_1)x_1+\cdots+(2\alpha_k+p_k)x_k} \right) \quad (2.1)$$

where $\vec{x} = (x_1, x_2, \ldots, x_k)$.

We now calculate the Fourier Transform of the parity submatrices described above. First, define the $N \times N \times \cdots \times N$ expanded Fourier parity matrices (or expanded matrices for short):

$$\begin{aligned} \forall p \in \Psi^k \quad F_p(\vec{x}) &= \sum_{\alpha_1=0}^{N/2-1} \cdots \sum_{\alpha_k=0}^{N/2-1} f(2\alpha_1 + p_1, \ldots, 2\alpha_k + p_k) \omega_{N/2}^{\alpha_1 x_1+\cdots+\alpha_k x_k} \quad (2.2) \\ &= \sum_{\alpha_1=0}^{N/2-1} \cdots \sum_{\alpha_k=0}^{N/2-1} f(2\alpha_1 + p_1, \ldots, 2\alpha_k + p_k) \omega_N^{2\alpha_1 x_1+\cdots+2\alpha_k x_k}, \end{aligned}$$

where $x_1, x_2, \ldots, x_k = 0, 1, \ldots, N-1$. Like the 2-dimensional case, each expanded matrix contains 2^k copies (one in each k-quadrant[2]) of the Fourier Transform of the corresponding parity submatrix, e.g., in 3 dimensions, $F_{(0,0,1)}$ contains 8 copies of the even-even-odd parity submatrix, etc. To verify this statement, note than if we restrict the domain of each of the

[2] We define a k-quadrant to be the k-dimensional equivalent of a quadrant in two dimensions. For example, in three dimensions, we have the front-bottom-left 3-quadrant, the front-bottom-right 3-quadrant, the front-top-left 3-quadrant, the front-top-right 3-quadrant, the back-bottom-left 3-quadrant, the back-bottom-right 3-quadrant, the back-top-left 3-quadrant and the back-top-right 3-quadrant.

components of $\vec{x}$ in the equations in (2.2) to $0, 1, \ldots, N/2-1$, the expanded matrix equations become the equations for the FFTs of the parity submatrices. This observation implies that the first k-quadrant of each expanded matrix contains the FFT of the corresponding parity submatrix. To show that the the expanded matrices in fact contain 2^k copies of the Fourier Transforms, we simply note that the equations in (2.2) are periodic with period $N/2$. Thus all 2^k k-quadrants of the expanded matrices are identical.

We are now ready to combine the submatrix transforms and show how to derive the full FFT. First, consider our split-summation FFT formula (2.1), which can be rewritten as

$$F(\vec{x}) = \sum_{p \in \Psi^k} \left(\sum_{\alpha_1=0}^{N/2-1} \cdots \sum_{\alpha_k=0}^{N/2-1} f(2\alpha_1 + p_1, \ldots, 2\alpha_k + p_k) \omega_N^{2\alpha_1 x_1 + \cdots + 2\alpha_k x_k} \omega_N^{p_1 x_1 + \cdots + p_k x_k} \right) .$$

If we substitute the expanded matrices from (2.2) into this equation, we obtain

$$F(\vec{x}) = \sum_{p \in \Psi^k} F_p(\vec{x}) \omega_N^{p_1 x_1 + \cdots + p_k x_k} . \tag{2.3}$$

Since the expanded matrices can be derived from the Fourier Transforms of the parity submatrices of F, we have shown that an FFT of arbitrarily high dimension can be broken into pieces and solved recursively.

2.1.1 Butterfly operations

From equation (2.3), we can derive the identities which allow us to combine the results of the sub-FFTs to compute the whole FFT. If we let $\mathcal{J}^k$ be the set of all k-element bit-vectors with components drawn from $\{0, 1\}$, we obtain[3]

$$\forall j = (j_1, j_2, \ldots, j_k) \in \mathcal{J}^k \quad F(x_1 + j_1 N/2, \ldots, x_k + j_k N/2) =$$
$$\textstyle\sum_{p \in \Psi^k} F_p(x_1 + j_1 N/2, \ldots, x_k + j_k N/2) \omega_N^{p_1 x_1 + j_1 p_1 N/2 + \cdots + p_k x_k + j_k p_k N/2} .$$

Since the expanded matrices are periodic with period $N/2$, this equation simplifies to

$$\forall j = (j_1, j_2, \ldots, j_k) \in \mathcal{J}^k \quad F(x_1 + j_1 N/2, \ldots, x_k + j_k N/2) =$$
$$\textstyle\sum_{p \in \Psi^k} F_p(x_1, \ldots, x_k) \omega_N^{p_1 x_1 + j_1 p_1 N/2 + \cdots + p_k x_k + j_k p_k N/2} .$$

Recall that $\omega_N^{N/2} = -1$. Thus

$$\forall j = (j_1, j_2, \ldots, j_k) \in \mathcal{J}^k \quad F(x_1 + j_1 N/2, \ldots, x_k + j_k N/2) =$$
$$\textstyle\sum_{p \in \Psi^k} F_p(x_1, \ldots, x_k) \omega_N^{p_1 x_1 + \cdots + p_k x_k} (-1)^{j_1 p_1 + \cdots + j_k p_k} .$$

These identities allow us to compute the entire FFT matrix from the first k-quadrants of the expanded matrices. Recall that the first k-quadrant of an expanded matrix will

[3]Note that $\mathcal{J}^k = \Psi^k$. We use $\mathcal{J}^k$ here to make it clear that we are not using the j vectors to indicate parity.

be the FFT of the corresponding parity submatrix. Thus to compute the points $F(x_1 + j_1 N/2, x_2 + j_2 N/2, \ldots, x_k + j_k N/2)$, we simply insert the point $(x_1, x_2, \ldots, x_k)$ from each of the recursively computed sub-FFTs into the above identities.

This process can be made significantly more efficient, however, by noting that the identities we derived contain many of the same additions, subtractions, and twiddle factor multiplications. For instance, in 2 dimensions, we can set

$$\begin{aligned} a &= F_{(0,0)}(x_1, x_2) , \\ b &= F_{(0,1)}(x_1, x_2)\omega_N^{x_2} , \\ c &= F_{(1,0)}(x_1, x_2)\omega_N^{x_1} , \\ d &= F_{(1,1)}(x_1, x_2)\omega_N^{x_1+x_2} . \end{aligned}$$

Next, set

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

Finally, set

$$\begin{aligned} F(x_1, x_2) &= A + C , \\ F(x_1, x_2 + N/2) &= B + D , \\ F(x_1 + N/2, x_2) &= A - C , \\ F(x_1 + N/2, x_2 + N/2) &= B - D . \end{aligned}$$

The above equations are the 2-dimensional butterfly operations described in [Bapt99] and [Riv77]. In 3 dimensions, we can set

$$\begin{aligned} a &= F_{(0,0,0)}(x_1, x_2, x_3) , \\ b &= F_{(0,0,1)}(x_1, x_2, x_3)\omega_N^{x_3} , \\ c &= F_{(0,1,0)}(x_1, x_2, x_3)\omega_N^{x_2} , \\ d &= F_{(0,1,1)}(x_1, x_2, x_3)\omega_N^{x_2+x_3} , \\ e &= F_{(1,0,0)}(x_1, x_2, x_3)\omega_N^{x_1} , \\ f &= F_{(1,0,1)}(x_1, x_2, x_3)\omega_N^{x_1+x_3} , \\ g &= F_{(1,1,0)}(x_1, x_2, x_3)\omega_N^{x_1+x_2} , \\ h &= F_{(1,1,1)}(x_1, x_2, x_3)\omega_N^{x_1+x_2+x_3} . \end{aligned}$$

Next, set

$$\begin{aligned} A &= a+b\,, \\ B &= a-b\,, \\ C &= c+d\,, \\ D &= c-d\,, \\ E &= e+f\,, \\ F &= e-f\,, \\ G &= g+h\,, \\ H &= g-h\,. \end{aligned}$$

Then, set

$$\begin{aligned} A' &= A+C = a+b+c+d\,, \\ B' &= B+D = a-b+c-d\,, \\ C' &= A-C = a+b-c-d\,, \\ D' &= B-D = a-b-c+d\,, \\ E' &= E+G = e+f+g+h\,, \\ F' &= F+H = e-f+g-h\,, \\ G' &= E-G = e+f-g-h\,, \\ H' &= F-H = e-f-g+h\,. \end{aligned}$$

Finally, set

$$\begin{aligned} F(x_1,x_2,x_3) &= A'+E' = a+b+c+d+e+f+g+h\,, && (2.4) \\ F(x_1,x_2,x_3+N/2) &= B'+F' = a-b+c-d+e-f+g-h\,, && (2.5) \\ F(x_1,x_2+N/2,x_3) &= C'+G' = a+b-c-d+e+f-g-h\,, && (2.6) \\ F(x_1,x_2+N/2,x_3+N/2) &= D'+H' = a-b-c+d+e-f-g+h\,, && (2.7) \\ F(x_1+N/2,x_2,x_3) &= A'-E' = a+b+c+d-e-f-g-h\,, && (2.8) \\ F(x_1+N/2,x_2,x_3+N/2) &= B'-F' = a-b+c-d-e+f-g+h\,, && (2.9) \\ F(x_1+N/2,x_2+N/2,x_3) &= C'-G' = a+b-c-d-e-f+g+h\,, && (2.10) \\ F(x_1+N/2,x_2+N/2,x_3+N/2) &= D'-H' = a-b-c+d-e+f+g-h\,. && (2.11) \end{aligned}$$

Figure 2.2 is a graphical representation of these 3-dimensional butterfly operations.

In k dimensions, we have $k+1$ steps, which we shall denote as steps $0, 1, \ldots, k$. In step 0, we multiply each input point by the appropriate twiddle factor. In step i, where

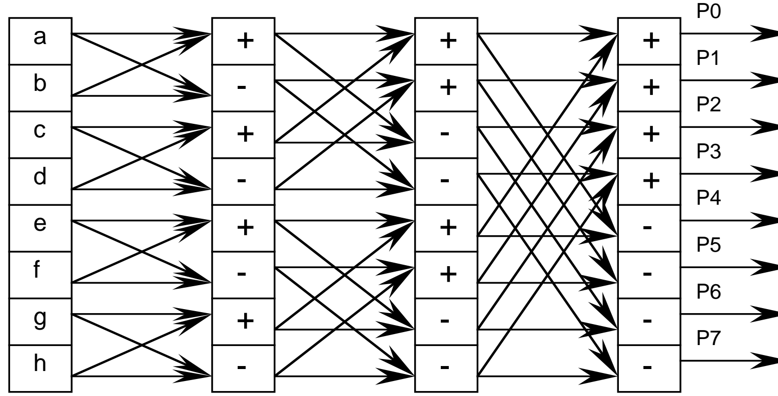


Figure 2.2: Graphical representation of 3-dimensional butterfly operations. The letters on the left represent the input points times the twiddle factors, as described in the text. Each box in the diagram represents an addition or subtraction, as indicated by the label. Data flows from the left to the right. $P_0, P_1, \ldots, P_7$ represent the 8 output points, in the same order as presented in Formulas (2.4) through (2.11)

$i > 0$, we create temporary variables and perform $2^k/2^i$ groups of 2^{i-1} pairs of additions and subtractions of temporary variables from the previous step separated by a stride of 2^{i-1}. The following algorithm formalizes the process:

/* Store initial values of elements of computation into $A[0, i]$ and multiply by
the appropriate twiddle factor (step 0 above) */
for all $p \in \Psi^k$
    let $A[0, p'] = F_p(\vec{x})$, where p' = the bits of p interpreted as a base-2 integer

/* Additions and subtractions (steps 1 to k above) */
for $i = 1$ to k
    let $stride = 2^{i-1}$
    for $group = 0$ to $(2^k/2^i - 1)$
        for $pair = 0$ to $(stride - 1)$
            let $pointA = (2^i)group + pair$
            let $pointB = pointA + stride$
            $A[i, pointA] = A[i-1, pointA] + A[i-1, pointB]$
            $A[i, pointB] = A[i-1, pointA] - A[i-1, pointB]$

/* Store results of computation back into matrix */
for all $j \in \mathcal{J}^k$
    let $F(x_1 + j_1 N/2, \ldots, x_k + j_k N/2) = A[k, j']$, where j' is defined like p' above

Note that we can easily optimize the space requirements of this algorithm by performing the calculations in place rather than using a seperate array row for each step.

2.2 The out-of-core vector-radix method in higher dimensions

Our higher-dimensional out-of-core vector-radix algorithm has the same structure as Baptist's 2-dimensional algorithm. In fact, when we apply our algorithm to a 2-dimensional matrix, it reduces to Baptist's algorithm. Once again, we divide the levels of our $N^{1/k} \times N^{1/k} \times \ldots \times N^{1/k}$ FFT computation into $\lceil n/m \rceil$ superlevels. Between each superlevel, we rearrange the data with a BMMC pemutation in preparation for the next superlevel.

Like the 2-dimensional algorithm, the higher-dimensional algorithm uses six types of BMMC permutations, in the following order:

/* Initial bit-reversal permutation */
permute data with *k-dimensional bit-reversal* BMMC permutation

/* Permutations performed every superlevel */
for *superlevel* $= 1$ to $\lceil n/m \rceil$
    permute with *processor-to-stripe major* BMMC permutation
    permute with *threader* BMMC permutation
    for *memoryload* $= 1$ to N/M
        read in memoryload
        compute butterfly operations
        write out memoryload
    permute with *inverse threader* BMMC permutation
    permute with *stripe-to-processor major* BMMC permutation
    if *superlevel* $= \lceil n/m \rceil$
      permute with *k-dimensional* $(n \# m)/k$ *bit-rotation* BMMC permutation
    else
      permute with *k-dimensional* m/k *bit-rotation* BMMC permutation

The notation $n \# m$ (the *upper modulus*) is defined by

$$x \# y = \begin{cases} x \bmod y & \text{if } x \bmod y > 0 \, , \\ y & \text{if } x \bmod y = 0 \, . \end{cases}$$

The *processor-to-stripe-major* and *stripe-to-processor-major* permuations are exactly the same as in the 2-dimensional case and will not be discussed further. In the following sections, we detail the *k-dimensional bit-reversal*, *threader*, *inverse threader*, and *k-dimensional bit-rotation* permutations.

2.2.1 The k-dimensional bit-reversal permutation

The general k-dimensional bit reversal permutation, with characteristic matrix

$$\begin{array}{c} \begin{array}{ccc} n/k & \cdots & n/k \end{array} \\ \left[\begin{array}{c|c|c} I^{\mathrm{A}} & 0 & 0 \\ \hline 0 & \ddots & 0 \\ \hline 0 & 0 & I^{\mathrm{A}} \end{array}\right] \begin{array}{c} n/k \\ \vdots \\ n/k \end{array} \end{array} ,$$

is an extension of the 2-dimensional bit-reversal permutation described earlier. The 2-dimensional bit-reversal permutation recursively groups elements into quadrants by the parity of their row and column indices. Similarly, our k-dimensional bit-reversal recursively groups elements into k-quadrants by the parities of their indices in each dimension.

To understand why the bit-reversal permutation recursively groups elements by parity, we need to consider the structure of an element's index. The element's index in dimension i is determined by the n/k bits $(i-1)n/k$ through $in/k-1$ of the complete index. The least significant bits of each dimension determine the element's parity submatrix. The next least significant bits determine the parity submatrix within the parity submatrix of the element, and so on, recursively. Similarly, the most significant bits of each dimension determine the element's k-quadrant, the next most significant bits determine the k-quadrant within the k-quadrant, and so on. The k-dimensional bit-reversal permutation reverses the bits of each dimension, causing the least significant bits of each dimensional index to become the most significant bits of that dimension's index. Thus the pre-permutation parities determine the post-permutation k-quadrants, the pre-permutation parities of the parities determine the post-permutation k-quadrants within the k-quadrants, and so on recursively.

2.2.2 Threader and inverse-threader permutations

The *threader* and *inverse threader* permutations are the higher-dimensional equivalents of the partial bit-rotation and and inverse partial bit-rotation permutations from Baptist's 2-dimensional algorithm. Recall that the vector-radix algorithm recursively divides the data matrix into parity submatrices. Let the *memoryload submatrices* be the largest parity submatrices that fit into a single memoryload. The threader permutations, with characteristic

matrix[4]

$$
\begin{array}{c}
\begin{array}{ccccccc}
\frac{m}{k} & \frac{n-m}{k} & \frac{m}{k} & \frac{n-m}{k} & \cdots & \frac{m}{k} & \frac{n-m}{k}
\end{array} \\
\left[\begin{array}{c|c|c|c|c|c|c}
I & 0 & 0 & 0 & \cdots & 0 & 0 \\ \hline
0 & 0 & I & 0 & \cdots & 0 & 0 \\ \hline
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline
0 & 0 & 0 & 0 & \cdots & I & 0 \\ \hline
0 & 0 & 0 & 0 & \cdots & 0 & I \\ \hline
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline
0 & 0 & 0 & I & \cdots & 0 & 0 \\ \hline
0 & I & 0 & 0 & \cdots & 0 & 0
\end{array}\right]
\begin{array}{c}
\frac{m}{k} \\ \frac{m}{k} \\ \vdots \\ \frac{m}{k} \\ \frac{n-m}{k} \\ \vdots \\ \frac{n-m}{k} \\ \frac{n-m}{k}
\end{array}
\end{array} ,
$$

move the elements of each of the memoryload submatrices into consecutive disk addresses so that they can be loaded from disk with a single memoryload. The inverse threader permutations, with characteristic matrix

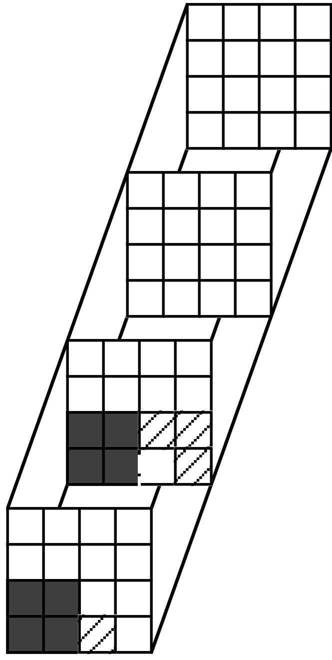
$$
\begin{array}{c}
\begin{array}{cccccccccc}
\frac{m}{k} & \frac{m}{k} & \cdots & \frac{m}{k} & \frac{m}{k} & \frac{n-m}{k} & \frac{n-m}{k} & \cdots & \frac{n-m}{k} & \frac{n-m}{k}
\end{array} \\
\left[\begin{array}{c|c|c|c|c|c|c|c|c|c}
I & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ \hline
0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & I \\ \hline
0 & I & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ \hline
0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & I & 0 \\ \hline
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline
0 & 0 & \cdots & I & 0 & 0 & 0 & \cdots & 0 & 0 \\ \hline
0 & 0 & \cdots & 0 & 0 & 0 & I & \cdots & 0 & 0 \\ \hline
0 & 0 & \cdots & 0 & I & 0 & 0 & \cdots & 0 & 0 \\ \hline
0 & 0 & \cdots & 0 & 0 & I & 0 & \cdots & 0 & 0
\end{array}\right]
\begin{array}{c}
\frac{m}{k} \\ \frac{n-m}{k} \\ \frac{m}{k} \\ \frac{n-m}{k} \\ \vdots \\ \frac{m}{k} \\ \frac{n-m}{k} \\ \frac{m}{k} \\ \frac{n-m}{k}
\end{array}
\end{array}
$$

reverse the threader permutations and return the elements to their original positions.[5]

To understand how the threader permutations work, we once again examine the structure of an element's index. Recall that if we break the n bits of the index into k (n/k)-bit pieces, we obtain the indices for each of the dimensions. The most significant $(n-m)/k$-bits of each dimensional index determine the element's memoryload matrix. The most significant $n-m$ bits of the complete index determine the element's memoryload. The threader permutation simply packs the $(n-m)/k$ most significant bits of each dimension into the $(n-m)$ most significant bits of the complete index. Thus each memoryload matrix is packed into a single

[4] Note that in a k-cubic matrix, where the dimension size is a power of 2, k divides $n = \lg N$. The base-2 logarithm of the size of the parity submatrices will also be divisible by k, and thus the base-2 logarithm of the size of the memoryload matrices will be divisible by k. We set our total memory size to be the size of the largest memoryload matrix that can fit into the system's main memory, since we will never use more than this amount. Thus n, m, and $n-m$ are all divisible by k.

[5] We call these permutations "threader" permutations because we visualize the k-dimensional memoryload matrices being threaded into the 1-dimensional space of disk addresses.



These rotations are responsible for rearranging the elements of the matrix such that elements participating in butterfly operations together in the next superlevel are found in the same memoryload submatrix. The final rotation, with characteristic matrix

$$
\begin{array}{c}
\begin{array}{ccccc} \frac{n\#m}{k} & \frac{n-n\#m}{k} & \cdots & \frac{n\#m}{k} & \frac{n-n\#m}{k} \end{array} \\
\left[\begin{array}{c|c|c|c|c}
0 & I & \cdots & 0 & 0 \\ \hline
I & 0 & \cdots & 0 & 0 \\ \hline
\vdots & \vdots & \ddots & \vdots & \vdots \\ \hline
0 & 0 & \cdots & 0 & I \\ \hline
0 & 0 & \cdots & I & 0
\end{array}\right]
\begin{array}{l} \frac{n-n\#m}{k} \\ \frac{n\#m}{k} \\ \vdots \\ \frac{n-n\#m}{k} \\ \frac{n\#m}{k} \end{array}
\end{array} ,
$$

is an $((n \# m)/k)$-bit-rotation and is responsible for moving the elements back to their original positions.

The (m/k)-bit rotation rotates the m/k least significant bits of each dimensional index into the most significant positions. As we pointed out earlier, the most significant $(n - m)/k$ bits of each dimensional index determine the element's memoryload matrix. Thus the least significant m/k bits of each dimension determine the element's position within the memoryload matrix. By rotating these bits into the most significant positions, we cause elements that are at the same position in different memoryload matrices before the permutation to be grouped together into the same memoryload matrix after the permutation. Since the elements at the same position in different memoryload matrices are the elements we combine in the next superlevel, this permutation does exactly what we wish.

For an example of the effect of the rotation permutation on a 3-dimensional matrix, see Figure 2.3.

2.2.4 Implementation details

Writing code to handle an arbitrary number of dimensions is the primary challenge when implementing our higher-dimensional out-of-core vector-radix FFT algorithm. We need to know the coordinates of the current memoryload and the current butterfly operation. If we know the number of dimensions at compile time, we simply use a series of nested for loops. If we do not know the number of dimensions in advance, we must use an alternate solution.

One such solution involves using only a single for loop. We determine the x_1 coordinate by selecting (with a bit mask) the least significant m/k bits of the loop index. We similarly determine the x_2 coordinate by the next least significant m/k bits. We repeat the process to determine all k coordinates. This solution, however, is inefficient because of the extra calculations required for every butterfly operation and memoryload.

Another solution is to write code that generates FFT code for matrices of a given number

of dimensions. This solution is more efficient but requires the user to rerun the generation code every time she wishes to work with a new number of dimensions. Most real-world applications, however, involve 3, 4, or at most 5 dimensions. Thus, realistically, a user should only need to run the generation code two or three times. Moreover, the time required by the generation and compiling the generated code is insignificant compared to the time required to compute a large multidimensional FFT.

Multiprocessor adaptations

We can easily adapt our higher-dimensional algorithm to take advantage of multiple processors. Instead of loading in a memoryload of size M to a single processor, we load in memoryloads of size M/P to each of the P processors and compute them simultaneously. Since our memoryloads are smaller, we may end up performing more superlevels: $\lceil n/(m-p) \rceil$ rather than $\lceil n/m \rceil$. However in practice we always have two superlevels. In order to have $\lceil n/(m-p) \rceil > 2$, we would need the amount of available disk space to be greater than the *square* of the available main memory per processor. The only other modification necessary to take advantage of a multiprocessor environment is to replace all references to m with $(m-p)$ in the definitions of the BMMC characteristic matrices.

Chapter 3

Analysis of the higher-dimensional algorithm

We now proceed to examine the complexity of the higher-dimensional vector-radix algorithm. Rivard, in [Riv77], shows that the computational complexity is simply $\Theta(N \lg N)$, like many FFT algorithms. The I/O and interprocessor communication complexities, however, are more challenging to analyze. I/O and communication costs are also usually more important because we are focusing on out-of-core, distributed-memory, multiprocessor environments. With modern, fast processors, I/O and communication time will dominate computation time in these settings.

The analysis of the communication complexity is complicated by the fact that the communication cost of Cormen's implementation of the BMMC permutation algorithm in [CSW97] has not been determined until now. We begin this chapter by developing a new method for analyzing this cost. Our method is based on an examination of the way the BMMC algorithm factors the characteristic matrix. Using properties of the generated factors, we are able to bound the communication cost in terms of the rank of the lower left $(n - m) \times m$ submatrix of the characteristic matrix of the permutation.

After discussing our new method, we proceed to determine the total I/O and communication cost of our higher-dimensional vector-radix algorithm. The communication cost is determined solely by the BMMC communication cost, since the rest of our algorithm requires no interprocessor communciation. We obtain the I/O cost by adding the I/O cost of the butterfly operations to the I/O cost of the BMMC permutations, which we determine by applying a formula from [CSW98].

3.1 BMMC-permutation communication cost analysis

Recall that our algorithm makes use of Cormen's implementation of the BMMC-permutation algorithm presented in [CSW98]. The algorithm factors the characteristic matrix into matrices characterizing *one-pass permutations*—permutations that can be performed with a single pass over the data. We refer to the factors by the names given to them in [CSW98]: F, $E_1^{-1}S_1^{-1}P^{-1}$, $E_2^{-1}S_2^{-1}$, $E_3^{-1}S_3^{-1}$, $\ldots$, $E_g^{-1}S_g^{-1}$, where $g = \lceil \operatorname{rank} \phi/(m-b) \rceil$ and ϕ is the lower left $(n-m) \times m$ submatrix of the characteristic matrix. To determine the communication cost of a BMMC permutation, we determine the costs of the individual factors and sum them together.

We now describe some of the matrices defined in [CSW98]. A *permutation matrix* is a square matrix with exactly one 1 in each row and exactly one 1 in each column. Note that an $n \times n$ permutation matrix has rank n (i.e., it has full rank). A *swapper matrix* has the form

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

where π is a specific type of $m \times m$ permutation submatrix described in [CSW98]. We omit the form of π submatrix here, as it is not necessary for our present analysis. Finally, the *eraser matrices* have the form

$$E = \begin{array}{c} \begin{array}{ccc} b & m-b & n-m \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & I & 0 \\ \hline 0 & * & I \end{array}\right] \end{array} \begin{array}{l} \\ b \\ m-b \\ n-m \end{array} ,$$

where $*$ represents a submatrix of 0's and 1's, also described in [CSW98] but unimportant for our present analysis.

We start by investigating the structure of the $E_j^{-1}S_j^{-1}$ factors. Each is the inverse of the product of a swapper matrix (S_j) with an eraser matrix (E_j). As detailed in [CSW98], the swapper matrices used in the BMMC permutation algorithm have the property that when a swapper matrix S is multiplied by a matrix A, the product AS is the matrix A with some columns from the leftmost b columns swapped with some columns from the middle $(m-b)$ columns. The function of the eraser matrices is detailed in [CSW98] but is not important for our present purposes.

Lemma 3.1 *The upper left $m \times m$ submatrix (denoted α) of an $E_j^{-1}S_j^{-1}$ factor of a BMMC characteristic matrix, where $1 \leq j \leq g = \lceil \operatorname{rank} \phi/(m-b) \rceil$, is a permutation matrix. In other words, each row and column of α has exactly one 1, and* $\operatorname{rank} \alpha = m$.

Proof: Because the inverse of a swapper permutation is itself [CSW98], we have

$$S^{-1} = \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \left[\begin{array}{c|c} \pi & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} m \\ n-m \end{array} .$$

We also know from [CSW98] that the eraser matrix is its own inverse, and so

$$E^{-1} = \begin{array}{c} \begin{array}{ccc} b & m-b & n-m \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & I & 0 \\ \hline 0 & * & I \end{array}\right] \end{array} \begin{array}{l} b \\ m-b \\ n-m \end{array} .$$

We write the product $E^{-1}S^{-1}$ as

$$\begin{aligned} E^{-1}S^{-1} &= \begin{array}{c} \begin{array}{ccc} b & m-b & n-m \end{array} \\ \left[\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & I & 0 \\ \hline 0 & * & I \end{array}\right] \end{array} \begin{array}{l} b \\ m-b \\ n-m \end{array} \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \left[\begin{array}{c|c} \pi & 0 \\ \hline 0 & I \end{array}\right] \end{array} \begin{array}{l} m \\ n-m \end{array} \\ &= \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \left[\begin{array}{c|c} \pi & 0 \\ \hline * & I \end{array}\right] \end{array} \begin{array}{l} m \\ n-m \end{array} . \end{aligned}$$

Thus the upper left $m \times m$ submatrix of the product $E^{-1}S^{-1}$ is the permutation matrix π. ■

The communication cost of a factor is entirely determined by the upper left $m \times m$ submatrix referred to in Lemma 3.1 above. To see this, first note that in the Parallel Disk Model, the most significant $n-m$ bits of a record's index indicate the record's memoryload. The next $m-(b+d)$ bits dictate the stripe within the memoryload. The following p bits determine the processor, and the least significant $b+d-p$ bits indicate the offset within the processor. Next, define the *communication rank* of an $n \times n$ characteristic matrix of a one-pass permutation to be the rank of the $p \times (m-p)$ submatrix $[\epsilon \mid \gamma]$ in the following diagram:

$$\begin{array}{c} \begin{array}{cccc} b+d-p & p & m-b-d & n-m \end{array} \\ \left[\begin{array}{c|c|c|c} & & & \\ \hline \epsilon & \beta & \gamma & \delta \\ \hline & & & \\ \hline & & & \end{array}\right] \end{array} \begin{array}{l} b+d-p \\ p \\ m-b-d \\ n-m \end{array} . \qquad (3.1)$$

During every memoryload of a one-pass permutation, each processor creates a separate buffer for every processor to which it sends data. All records going to the same destination processor are placed into the appropriate buffer and then are sent as a single message. The processor to which the record at source index i is sent is determined by the matrix product $[\epsilon \mid \beta \mid \gamma \mid \delta]\, i$. Within a memoryload, the most significant $n-m$ bits of every index are identical. Thus the structure of δ does not affect the number of messages sent per memoryload. Moreover, for each source processor, the p bits in positions $b+d-p$ through $b+d-1$ do not vary; thus the structure of β does not affect the number of messages sent per processor.

We borrow the following lemma from [CSW98].

Lemma 3.2 *([CSW98]) Let A be a $v \times q$ matrix whose entries are drawn from $\{0,1\}$, and let $r = \operatorname{rank} A$. Then $|\Re(A)| = 2^r$, where $\Re(A)$, the range of A, is the set of target indices $\{y : y = Ax$ for some $x \in \{0,1,...,2^q-1\}\}$.* ■

Lemma 3.2 allows us to calculate the communication cost of a one-pass permutation using only the communication rank. Since β and δ are fixed for a given source processor and memoryload, the range of $[\epsilon \mid \gamma]$ determines the number of target processors that can be reached from a given source processor during a given memoryload. We now temporarily make the simplifying assumption that when the target and source processors are the same, the processor still sends the data to itself. Since $|\Re([\epsilon \mid \gamma])| = 2^{\operatorname{rank}[\epsilon|\gamma]}$, the communication rank is equal to the (base-2) logarithm of the number of messages sent per processor per memoryload when performing a one-pass permutation.

Lemma 3.3 *Let B be a BMMC characteristic matrix, and $g = \lceil \operatorname{rank} \phi/(m-b) \rceil$ be the number of pairs of swap and erasure operations in the factoring of B. Then for $j = 2, 3, \ldots, g-1$, each factor $E_j^{-1} S_j^{-1}$ of B has communication rank $p = \lg P$.*

Proof: Lemma 3.2 and the above discussion tell us that we need only examine the upper left $m \times m$ submatrix of the characteristic matrix in order to determine the communication cost of a factor. The ϵ, β, and γ submatrices in diagram (3.1) correspond to the equivalent matrices in the following diagram of the upper left $m \times m$ submatrix (denoted α) of an $E_j^{-1} S_j^{-1}$ factor:

$$\begin{array}{c} \begin{array}{ccc} b+d-p & p & m-b-d \end{array} \\ \left[\begin{array}{c|c|c} & & \\ \hline \epsilon & \beta & \gamma \\ \hline & & \end{array}\right] \begin{array}{l} b+d-p \\ p \\ m-b-d \end{array} \end{array} .$$

By Lemma 3.1, the above matrix is a permutation matrix, and so there is exactly one 1 in each of the p rows of $[\epsilon \mid \beta \mid \gamma]$. Moreover, in every row the 1 appears in a different column. Thus $\operatorname{rank}[\epsilon \mid \beta \mid \gamma] = p$ and $\operatorname{rank}[\epsilon \mid \gamma] = p - \operatorname{rank}\beta$.

Next we show that $\operatorname{rank}\beta = 0$. By Lemma 3.1, the submatrix α is the permutation submatrix from the swapper matrix. Thus for each column i in α that intersects β, there is a 1 in the identity position if and only if column i is not swapped by the swapper operation. If column i *is* swapped with column s, however, the α matrix has a 1 in positions (i, s) and (s, i). Remembering our assumption that $d \geq p$ and noting that this implies $b + d - p \geq b$, if column i of α intersects β, then column i is in the rightmost $m - b$ columns of α. According to [CSW98], column s is then necessarily in the leftmost b columns. Thus row s will be in the topmost b rows, and so (s, i) will not be in the β submatrix. Column i will therefore have its single 1 in the submatrix β if and only if column i is not swapped by the swapper operation. We obtain

$$commrank = p - \beta_{notswapped},$$

where $commrank$ is the communication rank and $\beta_{notswapped}$ is the number of columns that intersect β and are not swapped. But according to [CSW98], if $2 \leq j \leq g - 1$, then S_j swaps all of the rightmost $m - b$ columns. Thus $\operatorname{rank}\beta = \beta_{notswapped} = 0$ and $commrank = p$. ■

Now we can remove the above-stated assumption that processors send messages to themselves. Since the communication rank is p, each processor sends P messages per memoryload. Because there are only P processors, each processor sends a message to every processor, including itself. However, in Cormen's implementation of the BMMC-permutation code, a processor detects when it would be sending data to itself and instead just copies the data from its send buffer to its receive buffer. Thus only $P - 1$ messages are actually sent per processor per memoryload.

We next make some notational definitions. Let

$$\mathcal{K}(a, b, c) = \min(b, \max(a, c)) \; .$$

Note that $\mathcal{K}(a, b, f(x))$ essentially "clips" or "bounds" $f(x)$ at a and b. Also recall, from the previous chapter, that

$$x \# y = \begin{cases} x \bmod y & \text{if } x \bmod y > 0 \,, \\ y & \text{if } x \bmod y = 0 \,. \end{cases}$$

We can now extend our analysis to the $E_g^{-1} S_g^{-1}$ factor.

Lemma 3.4 *Let C be a BMMC characteristic matrix. Using Cormen's implementation of the BMMC algorithm, the communication rank of the $E_g^{-1} S_g^{-1}$ factor is*

$$\mathcal{K}(0, p, (\operatorname{rank}\phi \# (m - b)) - (d - p)) \, .$$

Proof: As before, we must determine the number of β columns that are swapped. Each swap/erase pair erases $m-b$ columns. There are a total of $\operatorname{rank}\phi$ columns to erase [CSW98]. Thus the final swap/erase pair swaps and erases $\operatorname{rank}\phi \# (m-b)$ columns. Cormen's implementation of the BMMC algorithm swaps columns from the leftmost b columns into the leftmost columns of the next $m-b$ columns. Thus $(d-p)$ columns are swapped into before the β columns (see above matrix diagram). The number of β columns swapped is therefore $(\operatorname{rank}\phi \# (m-b)) - (d-p)$, but no fewer than 0 and no more than p. Thus, the communication rank of the $E_g^{-1} S_g^{-1}$ factor is

$$\mathcal{K}\left(0, p, (\operatorname{rank}\phi \# (m-b)) - (d-p)\right) .$$

■

The communication of all but two of the factors has now been characterized, allowing us to determine a bound on the cost of a BMMC permutation. First, though, we must borrow another lemma from [CSW98].

Lemma 3.5 *([CSW98]) Let A be a $p \times q$ matrix whose entries are drawn from $\{0,1\}$, let y be any p-vector in $\Re(A)$, and let $r = \operatorname{rank} A$. Then $|\operatorname{Preimage}(A, y)| = 2^{q-r}$, where $\operatorname{Preimage}(A, y) = \{x : Ax = y\}$.* ■

Combining our previous results, we obtain the following theorem.

Theorem 3.6 *The number of messages sent per processor for a given BMMC permutation characteristic matrix is between*

$$\frac{N}{M}\left(\left(\left\lceil \frac{\operatorname{rank}\phi}{m-b} \right\rceil - 2\right)(P-1) + 2^{\mathcal{K}(0,p,\operatorname{rank}\phi \# (m-b)-(d-p))} - 1\right)$$

and

$$\frac{N}{M}\left(\left(\left\lceil \frac{\operatorname{rank}\phi}{m-b} \right\rceil\right)(P-1) + 2^{\mathcal{K}(0,p,\operatorname{rank}\phi \# (m-b)-(d-p))}\right)$$

The data sent per processor by a BMMC permutation is between

$$\left(\left\lceil \frac{\operatorname{rank}\phi}{m-b} \right\rceil - 2\right)\left(\frac{N(P-1)}{P^2}\right)$$

and

$$\left(\left\lceil \frac{\operatorname{rank}\phi}{m-b} \right\rceil - 2\right)\left(\frac{N(P-1)}{P^2}\right) + \frac{3N}{P}$$

records, where ϕ is the lower left $(n-m) \times m$ submatrix of the characteristic matrix.

Proof: We consider each factor individually. By Lemmas 3.2 and 3.3, from all but the first two factors and last factor, there are

$$\left(\left\lceil \frac{\operatorname{rank}\phi}{m-b} \right\rceil - 2\right)(P-1)$$

messages sent per processor per memoryload. By Lemmas 3.2 and 3.4, the $E_g^{-1}S_g^{-1}$ factor induces either

$$2^{\mathcal{K}(0,p,\operatorname{rank}\phi\#(m-b)-(d-p))} - 1 \text{ or } 2^{\mathcal{K}(0,p,\operatorname{rank}\phi\#(m-b)-(d-p))}$$

messages per processor per memoryload, depending on whether each processor has records destined for itself. The two remaining factors, F and $E_1^{-1}S_1^{-1}P$, each range from 0 to $(P-1)$ messages per processor per memoryload, giving us the rest of the formula for the number of messages sent per processor per memoryload. We multiply by the N/M memoryloads to obtain the number of messages sent per processor.

To calculate the number of records that are actually sent by a processor, we subtract the number of records that remain on their source processor from the total number of records. During the processing of each factor, every processor permutes N/P records over the course of all memoryloads. Lemma 3.5 tells us that an equal number of records go to each target processor that a source processor sends to, including itself. For the $E_j^{-1}S_j^{-1}$ factors, where $j = 2, 3, \ldots, g-1$, we have shown that records are sent from each processor to all P processors. Taking into account the optimization that prevents processors from sending to themselves, we see that while performing each factor, each processor sends

$$\frac{N(P-1)}{P^2}$$

records. Multiplying by the number of these factors and noting that there are three other factors, each of which sends up to N/P records per processor, gives us the rest of the formula. ■

3.2 Complexity of the higher-dimensional algorithm

We can break the complexity of our algorithm into three parts: computation, communication, and disk I/O. Rivard shows, in [Riv77], that the computational complexity is simply $\Theta(N \lg N)$. The communication complexity can be determined by the method described above. The I/O complexity is equal to one pass for each of the $\lceil n/m \rceil$ superlevels, plus the number of passes during the BMMC permutations. Each permutation makes

$$\left\lceil \frac{\operatorname{rank}\phi}{m-b} \right\rceil + 1$$

passes [CSW98], where ϕ is the same lower left $(n-m) \times m$ submatrix of the characteristic matrix used in our communication analysis.

We used the method in [Bapt99] to compute the ranks of the ϕ submatrices of the characteristic matrices of the BMMC permutations used in our algorithm. Because BMMC permutations are closed under composition, we need only consider three types of permutations: the permutation prior to the first superlevel, the permutations that occur between superlevels, and the permutation that occurs after the last superlevel. Their ϕ ranks are given in the following table:

Position	rank ϕ
Before	$\min(n-m, \max(0, \lfloor \frac{md}{n} \rfloor (\frac{n-m+p}{d}) - p) + \min(m \bmod (\frac{n}{d}), \frac{n-m+p}{d}))$
Between	$(n-m)$
After	$(n-m) - \lfloor \frac{d(n-m)}{n} \rfloor (\frac{n-m+p}{d}) -$ $\mathcal{K}(0, \frac{n-m+p}{d}, (n-m) \bmod (\frac{n}{d}) - (\frac{n-m+p}{d})) +$ $\max(0, p - \lfloor \frac{dm}{n} \rfloor (\frac{n-m+p}{d}) - \mathcal{K}(0, \frac{n-m+p}{d}, m \bmod (\frac{n}{d}) - (\frac{2m-n-2p}{d})))$

The method used to calculate these ranks is explained nicely in [Bapt99]. We omit the description of the technique here because it has already appeared in print and is not directly relevant to the topic of this paper.

We are now ready to the calculate the I/O and communication complexity of our high dimensional vector-radix algorithm. The I/O complexity is

$$\left\lceil \frac{n}{m} \right\rceil + \left(\left\lceil \frac{\mathcal{R}_{\text{before}}}{m-b} \right\rceil + 1 \right) + \left(\left\lceil \frac{n}{m} \right\rceil - 1 \right) \left(\left\lceil \frac{\mathcal{R}_{\text{between}}}{m-b} \right\rceil + 1 \right) + \left(\left\lceil \frac{\mathcal{R}_{\text{after}}}{m-b} \right\rceil + 1 \right) ,$$

where $\mathcal{R}_{\text{before}}$ is the before-rank from the above table, $\mathcal{R}_{\text{between}}$ is the between-rank, and $\mathcal{R}_{\text{after}}$ is the after-rank. This expression simplifies to

$$2 \left\lceil \frac{n}{m} \right\rceil + \left\lceil \frac{\mathcal{R}_{\text{before}}}{m-b} \right\rceil + \left(\left\lceil \frac{n}{m} \right\rceil - 1 \right) \left\lceil \frac{\mathcal{R}_{\text{between}}}{m-b} \right\rceil + \left\lceil \frac{\mathcal{R}_{\text{after}}}{m-b} \right\rceil + 1 .$$

The communication complexity of our algorithm is between

$$\frac{N}{M} \left(C_1 + (\lceil n/m \rceil - 1) \, C_2 + C_3 \right)$$

and

$$\frac{N}{M} \left(C_1 + (\lceil n/m \rceil - 1) \, C_2 + C_3 + (2P - 1) \, (\lceil n/m \rceil + 1) \right)$$

messages sent, containing a total of between

$$\frac{N}{P} \left(D_1 + (\lceil n/m \rceil - 1) \, D_2 + D_3 \right)$$

and

$$\frac{N}{P} \left(D_1 + (\lceil n/m \rceil - 1) \, D_2 + D_3 + 3(\lceil n/m \rceil + 1) \right)$$

elements, where

- $C_1 = (P-1)\left(\lceil \mathcal{R}_{\text{before}}/(m-b) \rceil - 2\right) + 2^{\mathcal{K}\left(0,p,\mathcal{R}_{\text{before}}\#(m-b)-(d-p)\right)} - 1$ (the number of messages sent per processor per memoryload before the first superlevel),

- $C_2 = (P-1)\left(\lceil \mathcal{R}_{\text{between}}/(m-b) \rceil - 2\right) + 2^{\mathcal{K}\left(0,p,\mathcal{R}_{\text{between}}\#(m-b)-(d-p)\right)} - 1$ (the number of messages sent per processor per memoryload between superlevels),

- $C_3 = (P-1)\left(\lceil \mathcal{R}_{\text{after}}/(m-b) \rceil - 2\right) + 2^{\mathcal{K}\left(0,p,\mathcal{R}_{\text{after}}\#(m-b)-(d-p)\right)} - 1$ (the number of messages sent per processor per memoryload after the last superlevel),

- $D_1 = \left(\lceil \mathcal{R}_{\text{before}}/(m-b) \rceil - 2\right)((P-1)/P)$ (the data sent per processor before the first superlevel),

- $D_2 = \left(\lceil \mathcal{R}_{\text{between}}/(m-b) \rceil - 2\right)((P-1)/P)$ (the data sent per processor between superlevels),

- $D_3 = \left(\lceil \mathcal{R}_{\text{after}}/(m-b) \rceil - 2\right)((P-1)/P)$ (the data sent per processor after the last superlevel).

Chapter 4

Empirical results of the higher-dimensional vector-radix algorithm

To determine the effectiveness of the out-of-core vector-radix method in higher dimensions, we measure the running times of our algorithm and Baptist's implementation of the dimensional algorithm. We choose to use the more efficient code-generation implementation for the vector-radix results, as we believe that the performance benefits greatly outweigh the added complexity. For the dimensional method results, we use the same code Baptist used in [Bapt99]. We measure the results on two platforms:

- **A DEC2100 server.** The DEC machine has two 175-MHz Alpha processors and eight 2-gigabyte disks. We essentially use this machine as a uniprocessor, as computation is handled by a single thread. For each disk, there is also an additional thread responsible for managing I/O to that disk. The operating system is free to assign ready threads to available processors as it sees fit.

- **A cluster of LINUX workstations.** We use a 4-node cluster connected by a dedicated 5-port 10/100Mbps Ethernet switch with a maximum throughput of 200 Mbps per port in full-duplex mode [Grim01]. Each node has a 450-MHz Pentium II processor, 256 megabytes of main memory, and a 30-gigabyte disk.

The ViC* software [CH97] handles all I/O on both platforms and provides us with an interface implementing the Parallel Disk Model described earlier.

We use two methods to verify the accuracy of our results. We first compute the FFTs of so-called *spike matrices*—matrices with a single 1 in one position, and 0's in all other

	2-dimensional	3-dimensional	4-dimensional
vector-radix	703	649	613
dimensional	634	780	960

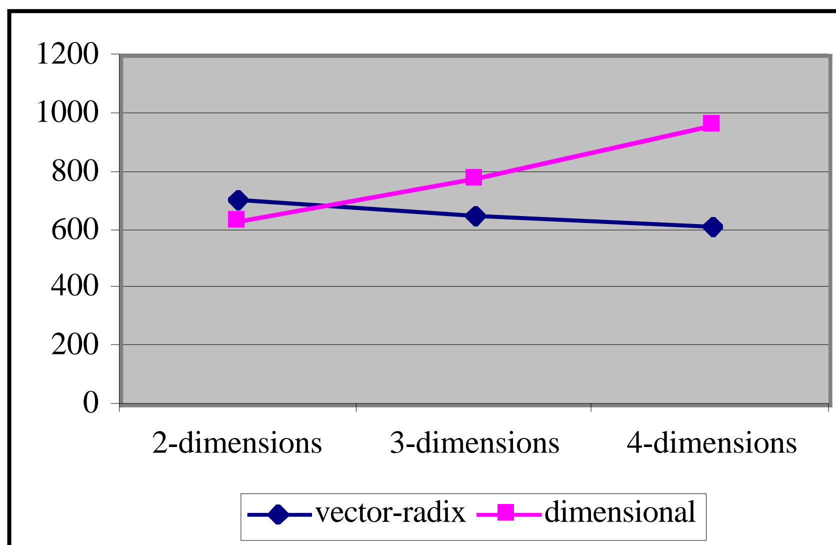


Figure 4.1: Results in seconds on DEC 2100 server with $N = 2^{24}$.

positions. There is a simple formula to compute the Fourier Transform of any spike matrix. We compare the formulaic result with the result generated by our code. Our second verification method involves computing the FFT of a random matrix using our vector-radix code and Baptist's dimensional method code. We compare the results and verify that they are equivalent to within a small tolerance.

The results of our experiments, given below, confirm that in higher dimensions the benefit of the vector-radix method is more pronounced. The algorithms take roughly the same amount of time in 2 dimensions. In 3 dimensions, the benefits of the vector-radix method begin to become apparent. In 4 dimensions, the vector-radix method achieves a speedup of between 1.5 and 2.

4.1 DEC 2100 server results

We examine two sets of results on the DEC2100 server—one set with $N = 2^{24}$-element matrices, shown in Figure 4.1, and the other with $N = 2^{28}$-element matrices, shown in Figure 4.2. Each element consists of two double-precision floating-point numbers — one for the real part of the number and the other for the imaginary part. Thus the $N = 2^{24}$ tests require $N = 2^{28}$ bytes (256 megabytes), and the $N = 2^{28}$ tests require $N = 2^{32}$ bytes (4

	2-dimensional	4-dimensional
vector-radix	13298	11666
dimensional	12176	18331

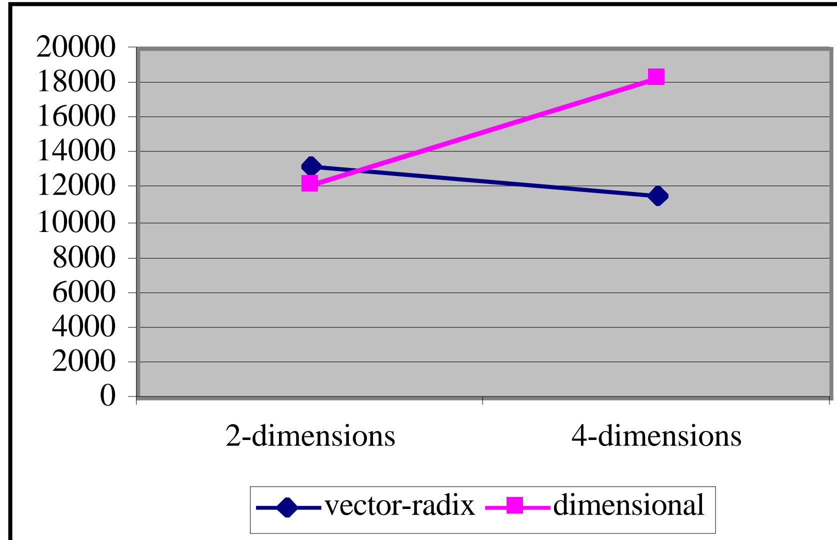


Figure 4.2: Results in seconds on DEC 2100 server with $N = 2^{28}$.

gigabytes). We limit all tests to k-cubic matrices[1] (for reasons which will become clear in the next chapter when we discuss the disadvantages of the non-square vector-radix method). As mentioned earlier, we also require that all dimension sizes be a power of 2. Therefore we can only run a k-dimensional test if $\lg N$ is divisible by k. Thus for the $N = 2^{24}$-element tests we run the FFT algorithms on 2-dimensional, 3-dimensional, and 4-dimensional matrices. For the $N = 2^{28}$-element tests, however, we only run on 2- and 4-dimensional matrices. For all tests, we use a memory size of 64 megabytes, a block size of 2^{13} records, and 8 disks.

4.2 Linux cluster results

We examine three sets of results on the Linux cluster—one set with matrices containing $N = 2^{24}$-elements, or 256 megabytes, shown in Figure 4.3, another with matrices containing $N = 2^{28}$-elements, or 4 gigabytes, shown in Figure 4.4, and a third with matrices containing $N = 2^{30}$-elements, or 16 gigabytes, shown in Figure 4.5. Once again, we use 2-, 3- and 4-dimensional matrices in the $N = 2^{24}$-element tests, and 2- and 4-dimensional matrices in the $N = 2^{28}$-element tests. For the $N = 2^{30}$-element tests, we run on 2- and 3-dimensional matrices. We use 4 nodes of the cluster, and allocate 32 megabytes of memory on each node.

[1] A k-cubic matrix is a k-dimensional matrix whose dimension sizes are all equal.

	2-dimensional	3-dimensional	4-dimensional
vector-radix	250	229	234
dimensional	256	332	431

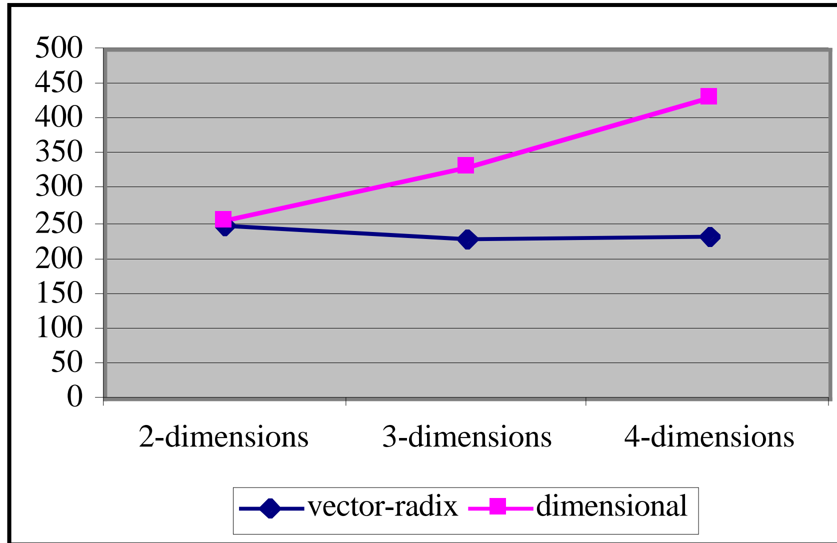


Figure 4.3: Results in seconds on a LINUX cluster with $N = 2^{24}$.

	2-dimensional	4-dimensional
vector-radix	6141	6012
dimensional	6054	10402

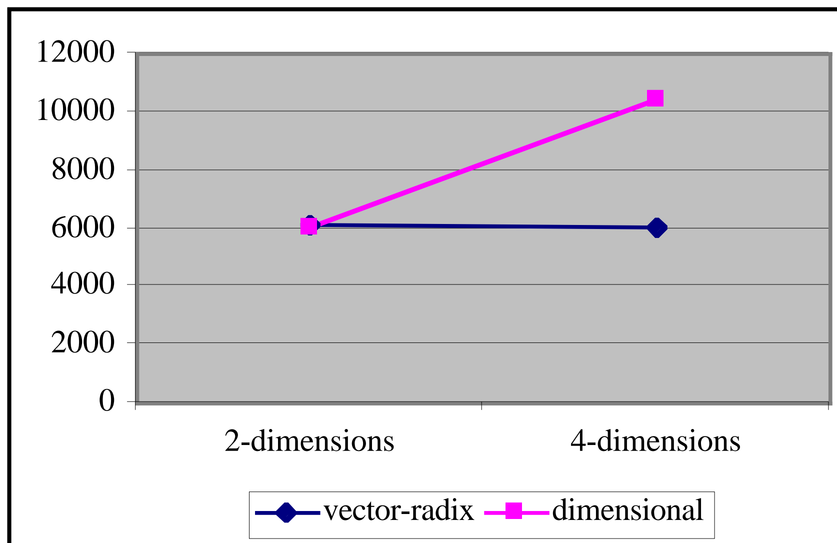


Figure 4.4: Results in seconds on a LINUX cluster with $N = 2^{28}$.

	2-dimensional	3-dimensional
vector-radix	25707	25798
dimensional	25351	34009

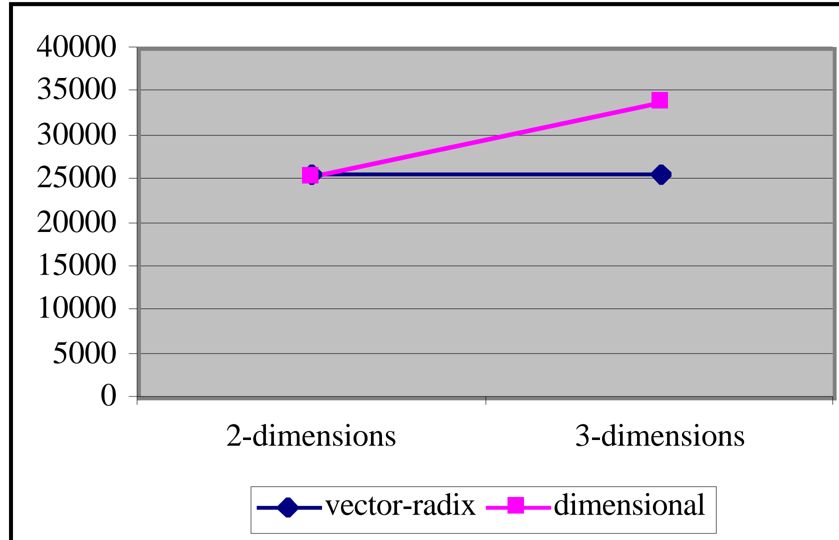


Figure 4.5: Results in seconds on a LINUX cluster with $N = 2^{30}$.

4.3 Discussion

When we look at the results on the two platforms, two characteristics stand out. As we anticipated, increasing the number of dimensions makes the dimensional method significantly slower. We also expected the vector-radix method to be unaffected by increasing the number of dimensions. However, our results show that the vector-radix method actually becomes slightly *faster* as we increase the number of dimensions.

The dimensional-method result is not surprising. Recall that the dimensional method requires a separate I/O pass for every dimension. Thus, increasing the number of dimensions increases the I/O time. When we look at a breakdown of the timings into computation time, communication time, and I/O time, we see that the extra time required in higher dimensions is almost exclusively due to disk I/O, as expected.

We did not, however, expect to discover that the out-of-core vector-radix method becomes faster in higher dimensions. When we study a breakdown of the vector-radix method timings, we see that the improvement appears to be due to the BMMC-permutation time. Recall from Chapter 3 that the time spent in the BMMC-permutation algorithm is determined by the number of factors into which the characteristic matrix is decomposed. The number of factors is in turn determined by the rank of the ϕ submatrix of the characteristic

matrix. If we look at the ranks shown in Chapter 3, we see that the rank of the ϕ submatrix of the characteristic matrix of the first permutation tends to decrease as we increase the number of dimensions. The improved performance of the out-of-core vector-radix method in higher dimensions is due to this first permutation.

The slower performance of the dimensional method, along with the slight improvement of the vector-radix method, causes the vector-radix method to significantly outperform the dimensional method in higher dimensions. We see that in two dimensions, the vector-radix method is on average 4.3% slower than the dimensional method. In three dimensions, the vector-radix method is on average 32.6% faster than the dimensional method. And in four dimensions, the vector-radix method is on average 70.4% faster.

Chapter 5

Extending the out-of-core vector-radix method to non-square matrices

Generalizing the out-of-core vector radix algorithm to handle non-square matrices adds another level of complexity, but is nonetheless important because some real-world problems are naturally non-square. For example, radio telescopes often sweep out a rectangular band across the sky as the earth rotates. Image processing of photographs is another example, because standard print sizes (e.g., 3×5, 4×6, or 5×7) are not square.

Unfortunately, generalizing the vector-radix method to handle non-square matrices makes it significantly less efficient. The non-square vector-radix method requires an extra step consisting of 1-dimensional FFT computations. Rivard analyzes this extra step in [Riv77] and shows that because of it, the vector-radix method is not as good a choice for non-square matrices.

In this chapter, we proceed to detail the non-square algorithm anyway, for completeness. We begin by showing why the most obvious technique for extending the square vector-radix algorithm to non-square matrices does not work. We then describe the correct extension and explain how to implement it efficiently in an out-of-core setting.

5.1 The in-core non-square vector-radix method

5.1.1 The problem

It is tempting to try to extend the vector-radix method to non-square matrices by simply splitting the matrix into contiguous square submatrices, using the square vector-radix method, and combining the results in the same manner as in Chapters 1 and 2. However, examining the equation for 2-dimensional FFTs shows that this method does not produce the desired results. If our input data is an $N \times RN$ element matrix f, then the FFT matrix F is defined by

$$\begin{aligned} F(x,y) &= \sum_{\alpha_1=0}^{RN-1} \sum_{\alpha_2=0}^{N-1} f(\alpha_1,\alpha_2)\omega_{RN}^{x\alpha_1}\omega_N^{y\alpha_2} \\ &= \sum_{\alpha_1=0}^{RN-1} \sum_{\alpha_2=0}^{N-1} f(\alpha_1,\alpha_2)\omega_{RN}^{x\alpha_1}\omega_{RN}^{Ry\alpha_2} \\ &= \sum_{\alpha_1=0}^{RN-1} \sum_{\alpha_2=0}^{N-1} f(\alpha_1,\alpha_2)\omega_{RN}^{x\alpha_1+Ry\alpha_2}, \end{aligned}$$

where $x = 0, 1, \ldots, RN-1$ and $y = 0, 1, \ldots, N-1$. Splitting the summation into summations over square matrices, we obtain

$$\begin{aligned} F(x,y) &= \sum_{\alpha_1=0}^{N-1} \sum_{\alpha_2=0}^{N-1} f(\alpha_1,\alpha_2)\omega_{RN}^{x\alpha_1+Ry\alpha_2} \\ &+ \sum_{\alpha_1=N}^{2N-1} \sum_{\alpha_2=0}^{N-1} f(\alpha_1,\alpha_2)\omega_{RN}^{x\alpha_1+Ry\alpha_2} \\ &\vdots \\ &+ \sum_{\alpha_1=(R-1)N}^{RN-1} \sum_{\alpha_2=0}^{N-1} f(\alpha_1,\alpha_2)\omega_{RN}^{x\alpha_1+Ry\alpha_2}. \end{aligned}$$

We now calculate the DFT of the corresponding square submatrices. Let $j = 0, 1, \ldots, R-1$, $x = 0, 1, \ldots, RN-1$, and $y = 0, 1, \ldots, N-1$ and define the $N \times RN$ expanded matrices

$$\begin{aligned} F_j(x,y) &= \sum_{\beta_1=0}^{N-1} \sum_{\beta_2=0}^{N-1} f(\beta_1 + jN, \beta_2)\omega_N^{x\beta_1+y\beta_2} \\ &= \sum_{\beta_1=0}^{N-1} \sum_{\beta_2=0}^{N-1} f(\beta_1 + jN, \beta_2)\omega_{RN}^{Rx\beta_1+Ry\beta_2}. \end{aligned}$$

Since ω_{RN} is periodic with period RN, we can write

$$F_j(x,y) = \sum_{\beta_1=0}^{N-1} \sum_{\beta_2=0}^{N-1} f(\beta_1 + jN, \beta_2)\omega_{RN}^{Rx\beta_1+RxjN+Ry\beta_2}.$$

Simplifying, we obtain

$$
\begin{aligned}
F_j(x,y) &= \sum_{\beta_1=0}^{N-1}\sum_{\beta_2=0}^{N-1} f(\beta_1+jN,\beta_2)\omega_{RN}^{Rx(\beta_1+jN)+Ry\beta_2} \\
&= \sum_{\beta_1=jN}^{(j+1)N-1}\sum_{\beta_2=0}^{N-1} f(\beta_1,\beta_2)\omega_{RN}^{Rx\beta_1+Ry\beta_2}\,.
\end{aligned}
$$

In order to substitute the F_j matrices into the split-summation DFT formula above, we need to remove an $\omega_{RN}^{(R-1)x\beta_1}$ factor from inside the F_j summations—otherwise the twiddle factors will not match. Unfortunately this is not possible because β_1 is one of the variables being summed over.

Thus splitting a rectangular matrix into contiguous $N \times N$ square matrices, computing the FFTs of the squares, and combining the results in the same manner as before does not work. Intuitively this makes sense, because the Fourier Transforms of the contiguous square matrices will contain no indication of signals whose period in the larger dimension (of size RN) is greater than N. Note that we have only shown that splitting into *contiguous* squares does not work. It may be possible to compute a rectangular FFT by breaking the matrix into square matrices using a different, noncontiguous partitioning, but we did not find any in our search of the literature.

5.1.2 The solution

To overcome this difficulty, we make the key observation that splitting the matrix into submatrices according to parity works just as well in the non-square case. When the aspect ratio is $R \neq 1$ we compute $R \times 1$ one-dimensional FFT base cases rather than single-element FFT base cases. (The FFT of a single element is itself.) We also modify our butterfly operations to combine rectangular submatrices with aspect ratio R rather than square submatrices. Unfortunately, computing 1-dimensional FFT base cases reduces the performance gain of the vector-radix method relative to the dimensional method (see [Riv77]).

To show that we can in fact compute the FFT of a non-square matrix by dividing according to parity and computing sub-FFTs, we once again examine the $N \times RN$ matrix FFT equation (5.1). Splitting both summations according to the parity of their indices, and using the parity vectors described earlier, we have

$$
F(x,y) = \sum_{p\in\Psi^2}\sum_{\alpha_1=0}^{RN/2-1}\sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1,2\alpha_2+p_2)\omega_{RN}^{(2\alpha_1+p_1)x+(2\alpha_2+p_2)Ry} \qquad (5.1)
$$

where Ψ^2 represents the set of all 2-bit parity vectors, as described earlier.

Now consider the Fourier Transform of the parity submatrices in the non-square case.

First, for all $p \in \Psi^2$, define the $N \times RN$ expanded matrices

$$\begin{aligned} F_p(x,y) &= \sum_{\alpha_1=0}^{RN/2-1} \sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1, 2\alpha_2+p_2)\omega_{RN/2}^{\alpha_1 x+\alpha_2 Ry} \\ &= \sum_{\alpha_1=0}^{RN/2-1} \sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1, 2\alpha_2+p_2)\omega_{RN}^{2\alpha_1 x+2\alpha_2 Ry} \end{aligned} \tag{5.2}$$

where $x = 0, 1, \ldots, RN-1$ and $y = 0, 1, \ldots, N-1$. As was the case with the square and higher-dimensional Fourier Transforms, if we restrict the range of x to $0, 1, \ldots, N/2-1$ and the range of y to $0, 1, \ldots, RN/2-1$ in the equations defined by (5.2), the left sides reduce to the formulas for the Fourier Transforms of the corresponding parity submatrices. Once again, we also note that due to the periodicity of the twiddle (ω) factors, each of the expanded matrices is periodic over the rows with period $N/2$ and over the columns with period $RN/2$. Thus each quadrant of each expanded matrix contains the Fourier Transform of the corresponding parity submatrix.

These definitions allow us to state the non-square FFT formula in terms of the FFTs of the parity submatrices. First, consider our split-summation FFT formula (5.1), which can be rewritten as

$$F(x,y) = \sum_{p\in\Psi^2} \sum_{\alpha_1=0}^{RN/2-1} \sum_{\alpha_2=0}^{N/2-1} f(2\alpha_1+p_1, 2\alpha_2+p_2)\omega_{RN}^{2\alpha_1 x+2\alpha_2 Ry}\omega_{RN}^{p_1 x+p_2 Ry} .$$

Notice that removing the summation over Ψ^2 and the final twiddle factor leaves us with the equation for $F_p(x,y)$. Thus the split-summation can be rewritten in terms of the expanded matrices as

$$\begin{aligned} F(x,y) &= \sum_{p\in\Psi^2} F_p(x,y)\omega_{RN}^{p_1 x+p_2 Ry} \\ &= F_{(0,0)}(x,y) + F_{(0,1)}(x,y)\omega_{RN}^{Ry} + F_{(1,0)}(x,y)\omega_{RN}^{x} + F_{(1,1)}(x,y)\omega_{RN}^{x+Ry} . \end{aligned}$$

Since the expanded matrices can be constructed from the FFT's of the parity submatrices, we have shown how to recursively compute the FFT of an $N \times RN$ matrix using the vector-radix method.

We now discuss an efficient method for combining the results of our recursive computations. Because of the periodicity of the F_{xy} matrices and because $\omega_N^{N/2} = -1$, we obtain the identities

$$\begin{aligned} F(x,y) &= F_{(0,0)}(x,y) + F_{(0,1)}(x,y)\omega_{RN}^{Ry} + F_{(1,0)}(x,y)\omega_{RN}^{x} + F_{(1,1)}(x,y)\omega_{RN}^{x+Ry} , \\ F(x+RN/2,y) &= F_{(0,0)}(x,y) + F_{(0,1)}(x,y)\omega_{RN}^{Ry} - F_{(1,0)}(x,y)\omega_{RN}^{x} - F_{(1,1)}(x,y)\omega_{RN}^{x+Ry} , \\ F(x,y+N/2) &= F_{(0,0)}(x,y) - F_{(0,1)}(x,y)\omega_{RN}^{Ry} + F_{(1,0)}(x,y)\omega_{RN}^{x} - F_{(1,1)}(x,y)\omega_{RN}^{x+Ry} , \\ F(x+RN/2,y+N/2) &= F_{(0,0)}(x,y) - F_{(0,1)}(x,y)\omega_{RN}^{Ry} - F_{(1,0)}(x,y)\omega_{RN}^{x} + F_{(1,1)}(x,y)\omega_{RN}^{x+Ry} . \end{aligned}$$

We can compute these identities more efficiently using butterfly operations. First, set

$$\begin{aligned}
a &= F_{(0,0)}(x,y)\ , \\
b &= F_{(0,1)}(x,y)\omega_{RN}^{Ry}\ , \\
c &= F_{(1,0)}(x,y)\omega_{RN}^{x}\ , \\
d &= F_{(1,1)}(x,y)\omega_{RN}^{x+Ry}\ .
\end{aligned}$$

Next, set

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

Finally, set

$$\begin{aligned}
F(x,y) &= A+C\ , \\
F(x+RN/2,y) &= A-C\ , \\
F(x,y+N/2) &= B+D\ , \\
F(x+RN/2,y+N/2) &= B-D\ .
\end{aligned}$$

We can now summarize the in-core non-square vector-radix method. We start with an $N \times RN$ matrix with power of 2 dimension sizes, and recursively split it into submatrices with half the rows and half the columns. This recursive partitioning eventually leaves us with $1 \times R$ matrices. These matrices are the base case of the non-square vector-radix algorithm. We compute their FFTs using any standard 1-dimensional FFT algorithm, and then proceed with computing the levels of butterfly operations. Unfortunately, the extra computation of the 1-dimensional FFTs makes vector-radix a less than ideal choice for non-square matrices, as Rivard makes clear in [Riv77].

5.2 The out-of-core non-square vector-radix method

We are now ready to detail our out-of-core non-square vector-radix algorithm. The algorithm has the same structure as Baptist's square algorithm and our higher-dimensional algorithm:

alternating phases of permuting and computing. In the non-square case, however, we have one extra step—the 1-dimensional FFT computations discussed in the previous section. We begin this section with a discussion of the BMMC permutations we use in the non-square alogrithm and then proceed to give the full algorithm.

5.2.1 BMMC permutations

Our non-square vector-radix algorithm uses the same six BMMC permutations as in Baptist's square matrix algorithm. We must modify the bit-reversal, partial bit-rotation, inverse partial bit-rotation, and right-rotation permutations to work correctly with non-square matrices. The processor-to-stripe-major and stripe-to-processor-major permutations, however, are identical to Baptist's permutations.

We need to modify the bit-reversal and bit-rotation permutations because in the non-square case, the number of the index bits representing each dimension will be different. For instance, in an 8×2 matrix, 3 of the 4 index bits determine the row position of the element, and just 1 bit determines the column position. We let R be the number of columns divided by the number of rows, or the *aspect ratio*, of the data matrix. Note that our matrix will then be of size $\sqrt{N/R} \times \sqrt{NR}$. We also define $r = \lg R$. Since both dimensions are powers of 2, R will also be a power of 2 and thus r will be an integer. Our new non-square bit-reversal permutation now has characteristic matrix

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

The new non-square partial bit-rotation matrix (analagous to the threader permutation in the higher-dimensional case) has characteristic matrix

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

The non-square inverse partial-bit rotation permutation characteristic matrix is

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

Like its higher-dimensional analogue (the d-dimensional rotation permutation), there are two versions of the non-square right-rotation permutation. We perform the first version, an

$((m-r)/2)$-bit rotation with characteristic matrix

$$
\begin{array}{c}
\begin{array}{ccccc} r & \frac{m-r}{2} & \frac{n-m}{2} & \frac{m-r}{2} & \frac{n-m}{2} \end{array} \\
\left[\begin{array}{c|c|c|c|c}
I & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & I & 0 & 0 \\ \hline
0 & I & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & I \\ \hline
0 & 0 & 0 & I & 0
\end{array}\right]
\begin{array}{l} r \\ \frac{n-m}{2} \\ \frac{m-r}{2} \\ \frac{n-m}{2} \\ \frac{m-r}{2} \end{array}
\end{array},
$$

after every superlevel except the last. The $(((n-r) \# (m-r))/2)$-bit rotation has characteristic matrix

$$
\begin{array}{c}
\begin{array}{ccccc} r & \frac{H}{2} & \frac{n-r-H}{2} & \frac{H}{2} & \frac{n-r-H}{2} \end{array} \\
\left[\begin{array}{c|c|c|c|c}
I & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & I & 0 & 0 \\ \hline
0 & I & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & I \\ \hline
0 & 0 & 0 & I & 0
\end{array}\right]
\begin{array}{l} r \\ \frac{n-r-H}{2} \\ \frac{H}{2} \\ \frac{n-r-H}{2} \\ \frac{H}{2} \end{array}
\end{array},
$$

where $H = (n-r) \# (m-r)$; this permutation occurs only after the final superlevel.

5.2.2 Complete algorithm

We are now ready to present the non-square out-of-core vector-radix algorithm:

```
permute data with non-square bit-reversal
compute 1-dimensional R element FFT's
for superlevel = 1 to n/m
    permute with processor-to-stripe-major
    permute with non-square partial bit-rotation
    for memoryload = 1 to N/M
        read in memoryload
        compute butterfly operations
        write out memoryload
    permute with inverse non-square partial bit-rotation
    permute with stripe-to-processor-major
    if superlevel = n/m
       permute with non-square (((n − r) # (m − r))/2)-bit right-rotation
    else
       permute with non-square ((m − r)/2)-bit right-rotation
```

Non-square penalty

In the out-of-core case, the switch from square matrices to non-square matrices incurs a penalty, just as in the in-core case. However, the penalty is not as severe because the primary bottleneck in out-of-core problems is the I/O time, and the extra 1-dimensional

FFT step only adds to the computation time. In our tests, the penalty was around 5.4% when going from a square matrix to a matrix with an aspect ratio of 256.

Multiprocessor adaptations

We can easily adapt our non-square algorithm to take advantage of multiple processors. We use the exact same modifications detailed when we described the higher-dimensional algorithm. Rather than repeating them here, we simply refer the reader to Chapter 2.

Chapter 6

Conclusion

We have detailed how Baptist's out-of-core vector-radix algorithm can be extended to handle both higher-dimensional and non-square data matrices. Both extensions retain the basic structure of the original algorithm: alternating phases of permuting and computing. The higher-dimensional extension requires modifying the permutations and the butterfly computations. The non-square extension requires both of the above, plus an additional 1-dimensional FFT computation. This extra step reduces the efficiency of the non-square algorithm.

We have also presented the results of experimental runs and a complexity analysis of the higher-dimensional algorithm. The experimental results were compared with results generated by Baptist's out-of-core dimensional method. We saw that the two methods were equally efficient in 2 dimensions, but that the vector-radix method was markedly more efficient in higher dimensions. We expected that this result would be the case because the dimensional method requires an extra pass over the data for every dimension, whereas the I/O complexity of the vector-radix method does not change as the number of dimensions is increased.

Some future work remains to be done. For instance, it should be possible to improve on the efficiency of the dimensional method. There are many cases where two or more dimensions fit into memory simultaneously, but the current algorithm does not take advantage of these situations. In these cases, it should be possible to compute the FFTs of multiple dimensions in a single pass, thus reducing the total number of passes necessary. Additionally, Van Loan [Van92] lists a number of other FFT algorithms which could be applied to out-of-core problems. To the best of our knowledge, only the vector-radix and dimensional methods have ever been attempted.

Acknowledgements

I'd first like to thank my advisor, Professor Thomas Cormen. He taught my first Computer Science course when I was an undergraduate and has been my thesis advisor ever since I entered the Master's program.

My thanks also go out to:

- Hany Farid and John Mackey for serving on my thesis committee. Their suggestions and encouragement have been invaluable.
- Lauren Baptist for helping me get started with the project, and for assistance with the I/O and communication analysis.
- Arne Grimstrup. Without his help and patience, I never would have been able to generate results on the Linux cluster.
- My family, for their love and support.

Finally, I'd like to dedicate this thesis to Thomas Hudson, 1942–2000. Tom was my high school biology teacher. But not only was he my teacher, he was my mentor, my inspiration, and my counselor. You will be missed, Tom.

Bibliography

[Bapt99] Lauren M. Baptist. Two Algorithms for Performing Multidimensional, Multiprocessor, Out-of-core FFTs. Dartmouth College Computer Science Technical Report PCS-TR99-350. June 1999.

[Bapt00] Lauren Baptist, Personal communication, March 2000.

[Bing74] E. Oran Bingham. *The Fast Fourier Transform.* Englewood Cliffs, New Jersey: Prentice-Hall, 1974.

[CC99] Thomas H. Cormen and J. C. Clippinger. Performing BMMC Permutations Efficiently on Distributed-Memory Multiprocessors with MPI. In *Algorithmica*, 24 (1999), pp. 349-370.

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

[CH97] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the parallel disk model with the ViC* implementation. In *Parallel Computing*, 23 (1997), pp. 571-600.

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

[Cor00] Thomas H. Cormen, Personal communication, October 2000.

[CSW98] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asympototically Tight Bounds for Performing BMMC Permutations on Parallel Disk Systems. In *SIAM J. Computing*, Vol. 28, No. 1, pp. 105–106, June 1998.

[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.

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

[French61] A. P. Fench. *Principles of Modern Physics*. New York: Wiley, 1961.

[Grim01] Arne Grimstrup, Personal communication, January 2001.

[HW95] Hermann Haken and Hans Christoph Wolf. *Molecular Physics and Elements of Quantum Chemistry*. Springer-Verlag, Berlin, Germany, 1995.

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

[MS81] R. M. Mersereau and T. C. Speake. A unified treatment of Cooley Tukey algorithms for the evaluation of the multidimensional DFT. In *IEEE Transaction on Acoustics, Speech, and Signal Processing*, 29 (1981), pp. 1011-1018.

[OS75] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

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

[SS95] Winthrop W. Smith and Joanne M. Smith. *Handbook of Real-time Fast Fourier Transforms*. IEEE Press, New York, 1995.

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

[VS94] Jeffery 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.

[WP89] Hong Ren Wu and Frank John Paoloni. The Structure of Vector Radix Fast Fourier Transforms. In *IEEE Transaction on Acoustics, Speech, and Signal Processing*, 37 (1989), pp. 1415-1424.