Multidimensional, Multiprocessor, Out-of-Core FFTs with Distributed Memory and Parallel Disks

(Extended Abstract)

Lauren M. Baptist
Thomas H. Cormen*
{lmb, thc}@cs.dartmouth.edu

Dartmouth College
Department of Computer Science

Abstract

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

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

We present an I/O complexity analysis for this method as well as empirical results for a DEC 2100 server, an SGI Origin 2000, and a Beowulf cluster, each of which has a parallel disk system.

1 Introduction

Although the data requirements of many FFT computations are small enough that the data fit in main memory, there are some situations in which the data requirements exceed the memory capacity of even very large systems. The typical way of dealing with such "out-of-core" situations is to have the data reside on a disk system (preferably parallel) and transfer sections of the data to and from memory. Previous work [CN97, CN98, Cor99, CWN97, VS94] has shown how to perform out-of-core, 1-dimensional FFTs on both uniprocessors and multiprocessors when the data reside on a parallel disk system. The Parallel Disk Model, or PDM, originally defined by Vitter and Shriver [VS94], provided both a theoretical and implementation model in the previous work.

This paper extends the previous work to multidimensional FFTs in which all dimensions are integer powers of 2. In a k-dimensional FFT on N points, we view the data as a k-dimensional array with dimension sizes $2^{n_1}, 2^{n_2}, \ldots, 2^{n_k}$, where all n_j are positive integers and $n_1 + n_2 + \cdots n_k = n = \lg N$.

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

Although there are several known methods for computing multidimensional FFTs for "in-core" settings (i.e., the data do fit in memory), we take the simplest approach in this work. In particular, we take advantage of a basic property of multidimensional FFTs: they may be computed by computing 1-dimensional FFTs within each dimension in turn. In other words, we first compute 1-dimensional FFTs within the first dimension. Starting with the results of these FFTs in the first dimension, we next compute 1-dimensional FFTs within the second dimension, and so on.

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

*Contact author. Supported in part by the National Science Foundation under grant CCR-9625894 and in part by funds from Dartmouth College.

To appear in SPAA '99.

This paper analyzes the I/O costs of our out-of-core method for the PDM. The analyses are complicated by additional permutations that we perform between processing the separate dimensions. These additional permutations perform the bit-reversal permutation at the start of computing each set of 1-dimensional FFTs. They also remap the data in such a way as to permit communication-free butterfly computations, in the same manner as in [CWN97]. All of these permutations are BMMC permutations, and the class of BMMC permutations is closed under composition. Hence, the analysis requires us to determine the I/O complexity under the PDM of some complicated permutations.

We also present empirical results for an implementation of our method. The implementation of the PDM we use is ViC* [CH97], which has been ported to several platforms. This paper presents results for three systems that have parallel disks: a DEC 2100 server, a Silicon Graphics Origin 2000, and a Beowulf cluster.

Outline

The remainder of this paper is organized as follows. Section 2 briefly describes the PDM and the implementation of it used here. Section 3 presents BMMC permutations, which we use in our description of the FFT method in Section 4. Sections 5 and 6 present our analytical and empirical results, respectively. Finally, Section 7 presents some concluding remarks.

2 The Parallel Disk Model

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

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

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

For a multiprocessor, we assume that there are P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ connected by a network. Network speeds vary greatly, but for the multiprocessors that we consider, interprocessor communication times are far less than I/O latencies. The M-record memory is distributed among the P processors so that each processor holds M/P records. In ViC*, our implementation of the PDM, we assume that $D \geq P$, and each processor $\mathcal{P}_i$ communicates only with the D/P disks $\mathcal{D}_{iD/P}, \mathcal{D}_{iD/P+1}, \ldots, \mathcal{D}_{(i+1)D/P-1}$. (If $D < P$ in a given physical configuration, the ViC* implementation provides the illusion that $D = P$ by sharing each physical disk among P/D processors.)

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

The PDM lays out data on a parallel disk system as shown in Figure 1. A *stripe* consists of the D blocks at the same location

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

Figure 1: The layout of $N = 64$ records in a parallel disk system with $P = 4$, $B = 2$, and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

on all D disks. A record's index is an n-bit vector. Later on, we will take advantage of interpreting a record index as a sequence of bit fields that give the record's location in the parallel disk system; from most significant bits to least significant bits, the bit fields are

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

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

Practical considerations

Although there is no theoretical restriction on the parameters N and M beyond the out-of-core requirement of $M < N$, in practice we expect that $N \leq M^2$ or, equivalently, $M \geq \sqrt{N}$. We show why by an example. Consider a 1-terapoint problem, so that $N = 2^{40}$. (We know of no current application that requires FFTs that large.) At 16 bytes per point, we would need 16 terabytes of disk storage just to hold the data. (In fact with our FFT algorithms, we would need an additional 8 terabytes to hold temporary data, but we will ignore this extra amount.) Now suppose that $M < \sqrt{N}$, in which case we would have $M < 2^{20}$. In other words, the memory cannot hold 1 megapoint, or 16 megabytes. It is safe to say that any computer installation that can afford to buy 16 terabytes of disk capacity would have computers easily capable of holding 16 megabytes of data. (As of this writing, in early 1999, it is virtually impossible to buy even a PC with under 16 megabytes of RAM.) Even if we allow for additional uses of memory beyond holding the FFT data (code, stack, communication buffers, I/O buffers, etc.), the aggregate memory of a system with 16 terabytes of on-line disk storage will always be large enough that $M \geq \sqrt{N}$.

3 BMMC permutations

This section defines the class of BMMC permutations, gives their I/O complexity on the PDM, and describes the specific types of BMMC permutations we will use in our multidimensional, multiprocessor FFT algorithm.

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

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

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

We shall use several types of BMMC permutations to perform multidimensional, multiprocessor FFTs. Each is from the even more restricted class of *bit permutations*, in which the characteristic matrix is a permutation matrix (exactly one 1 in each row and in each column). In other words, each target index is formed by permuting the bits of its corresponding source index.

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

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

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

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

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

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

stripe-major to processor-major

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

processor-major to stripe-major

In Section 5, we shall determine the I/O complexities of the various products of these matrix forms that characterize the actual BMMC permutations performed in the algorithm.

4 FFTs

This section defines the multidimensional FFT and presents our method for computing it out of core on multiple processors with distributed memory. Van Loan [Van92] is an excellent reference for how to perform FFTs.

Multidimensional FFTs

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

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

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

One way to compute a multidimensional FFT is to compute 1-dimensional FFTs on each dimension in turn. That is, we compute

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

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

There are other ways to compute multidimensional FFTs, such as the "vector-radix method," which we shall discuss briefly in Section 7.

Out-of-core implementation on a multiprocessor

Our present implementation uses the first method above, in which we computed 1-dimensional FFTs on each dimension in turn. We call this the *dimensional method*. We base our 1-dimensional FFT computations on the well-known Cooley-Tukey method, in which we perform a bit-reversal permutation followed by a sequence of "butterfly operations." We refer the reader to any of [CLR90, CN98, CT65, Van92] for details.

A key subroutine used by our implementation performs a BMMC permutation on the full N-point data set. This subroutine, based on techniques described in [CC97, CSW99], takes an $n \times n$ characteristic matrix (bit-packed into n words) as an input, and performs optimally the BMMC permutation so characterized.

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

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

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

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

To recap, prior to computing the dimension-j butterfly operations, we first perform an n_j-bit partial bit-reveral permutation, followed by the stripe-major to processor-major BMMC permutation. After computing the dimension-j butterfly operations, we first perform the processor-major to stripe-major BMMC permutation, followed by an n_j-bit right-rotation permutation.

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

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

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

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

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

Implementation notes

Our implementation of the dimensional method for computing the multidimensional FFT is a fairly straightforward extension of previous work on 1-dimensional FFTs [CN98, CWN97]. We continue to call asynchronous (i.e., non-blocking) I/O functions, when the underlying system supports it, by allocating three buffers: for reading into, writing from, and computing in. We did make some modifications, however. The twiddle factor computation is improved. When each dimension-j FFT fits in-core, all the dimension-j FFT computations can share a common set of twiddle factors. Hence, we allocate a buffer to hold all the twiddle factors, compute these values once, and reuse them in each FFT computation within dimension j. Also, we use a very accurate method to generate the twiddle factors. Our previous work employed repeated multiplication to compute the twiddle factors quickly, but with a high accumulation of error. Instead, we implemented a method that uses recursive bisection [Van92] to compute the twiddle factors more accurately, yet efficiently.

5 Analytical results

In this section, we analyze the I/O complexity of our multidimensional FFT algorithm on the PDM. Our analysis is exact: it counts parallel I/O operations without using asymptotic notation.

We remind the reader of a key assumption that was stated in Section 4: $N_j \leq M/P$ for all $j = 1, 2, \ldots, k$. In other words, we can perform each dimension-j FFT in-core. Although it is possible to analyze the algorithm without this assumption, the resulting I/O-complexity formulas are so unwieldy as to have little value.

Recall that the I/O complexity of a BMMC permutation is $\frac{2N}{BD}\left(\left\lceil \frac{\operatorname{rank}\phi}{\lg(M/B)} \right\rceil + 1\right)$ parallel I/Os, where ϕ is the lower left $\lg(N/M) \times \lg M$ submatrix of the characteristic matrix, and the rank is computed over $GF(2)$. We shall simplify our calculations in two ways. First, we will simply count passes, where each pass is $2N/BD$ parallel I/Os. Second, we will use the lowercase letters, which denote logarithms of uppercase letters. With these conventions, we can restate the I/O complexity of a BMMC permutation as $\left\lceil \frac{\operatorname{rank}\phi}{m-b} \right\rceil + 1$ passes, where ϕ is the lower left $(n-m) \times m$ submatrix.

The following lemmas are central to the analysis.

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

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

$$\operatorname{rank}\phi = \begin{cases} n-m & \text{if } n-p \le n_j + n_{j+1}\ , \\ \min(n-m, n_j) & \text{otherwise}\ . \end{cases} \blacksquare$$

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

The proofs of these lemmas will be in the full paper. We sketch here how we proved them. To find the rank of the lower left $(n-m) \times m$ submatrix of a characteristic matrix A, it suffices to find the rank of the $(n-m) \times m$ matrix product $X\,A\,Y\,\Pi$, where the matrices X and Y have the forms

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

and Π is any $m \times m$ permutation matrix. Because matrix multiplication is associative, one may group these factors, including the factors that comprise the matrix A, in any convenient fashion. Case analyses and reliance on the assumption that $n_j \le m-p$ for all $j = 1, 2, \ldots, k$ helped prove the lemmas.

From Lemmas 1–3, recognizing that computing the butterfly operations entails one pass for each of the k dimensions, and observing that $\lceil p/(m-b) \rceil \le 1$ under the PDM's assumption that each processor has enough memory to contain one disk block, we have the following theorem:

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

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

passes, where

$$f(n, m, p, n_j, n_{j+1}) = \begin{cases} n-m & \text{if } n-p \le n_j + n_{j+1}\ , \\ \min(n-m, n_j) & \text{otherwise} \end{cases}$$

and lowercase letters denote logarithms of corresponding uppercase letters. ∎

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

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

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

parallel I/O operations, where

$$F(N, M, P, N_j, N_{j+1}) = \begin{cases} \lg(N/M) & \text{if } N/P \le N_j N_{j+1}\ , \\ \lg \min(N/M, N_j) & \text{otherwise}\ . \end{cases} \blacksquare$$

There is no known lower bound for the I/O complexity of multidimensional FFTs on the PDM.

Practical considerations

Let us put the results of Theorem 4 and Corollary 5 into realistic contexts.

For example, one of the systems on which we report empirical results in Section 6 is a Silicon Graphics Origin 2000. Machine parameters were $M = 2^{27}$ points, $P = D = 8$, and $B = 2^{13}$ points. Consider problems with $N = 2^{30}$ points, divided as evenly as possible among the k dimensions. Applying Theorem 4 to this situation, the number of passes turns out to be a linear function of the number of dimensions, in particular, $3k+2$ for $k \ge 2$. (We require $k \ge 2$ because of the assumption that $N_j \le M/P$ for each dimension j. When $k = 1$, we have that $N_1 = N > M$, and the assumption does not hold.)

In fact, it is reasonable to expect there to be $3k+2$ passes for $k \ge 2$ in any realistic setting. Why? For systems of reasonable size, we expect $n-m$ to be no larger than $m-b$, in which case every ceiling-expression in the statements of Theorem 4 and Corollary 5 has the value 1 and the number of passes reduces to the expression $3k+2$. We expect $n-m \le m-b$ for the following reasons. Assuming, as we did at the end of Section 2, that $M \ge \sqrt{N}$, we must also have that $n-m \le m$. Although this inequality does not imply that $n-m \le m-b$, it is indeed the case in most practical settings. It would be unusual to have $n-m > 10$, for in such a setting, the amount of disk storage would be over 1000 times the amount of memory. Given the cost differential between memory and disks, per megabyte (about a factor of 40 as of this writing), a system with 1000 times more disk capacity than memory is unbalanced and needs more memory. On the other hand, block sizes are rarely above 2^{13} records and memory sizes are rarely below 2^{23} records for reasonably large machines (certainly for machines with very high disk capacity), and so we expect $m-b \ge 10$. Hence, $n-m \le m-b$ is a reasonable assumption, and we conclude that $3k+2$ passes is common.

6 Empirical results

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

lg N	Total time (secs)	Normalized time (μsecs)
22	139.00	3.01272
24	621.67	3.08787
26	2983.35	3.41964
28	12346.20	3.28523

Figure 2: Total times and normalized times for the DEC 2100 server.

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

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

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

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

3. A Beowulf cluster at Caltech with 114 PCs, each with a 200-MHz Pentium Pro processor and a 3.1-gigabyte EIDE disk. We used either 8 or 16 of the processors for each run. The interconnect is 100 Mb/sec Ethernet. As for the Origin 2000, we use MPI for interprocessor communication and ROMIO for parallel disk I/O.

The underlying software on both the Origin 2000 and the Beowulf cluster does not always perform asynchronous I/O reliably. On the Beowulf cluster, all parallel-I/O calls are implemented synchronously. On the Origin 2000, parallel-I/O calls within the BMMC-permutation subroutine are asynchronous, but parallel-I/O calls within the rest of the computation are synchronous.

DEC 2100 results

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

In the second set of runs on the DEC 2100, we varied the number of dimensions from 1 to 6, keeping all other parameters fixed.

dimensions	Total time (secs)	Normalized time (μsecs)
1	756.38	3.75698
2	609.22	3.02604
3	768.80	3.81867
4	945.84	4.69803
5	1174.51	5.83385
6	1410.93	7.00817

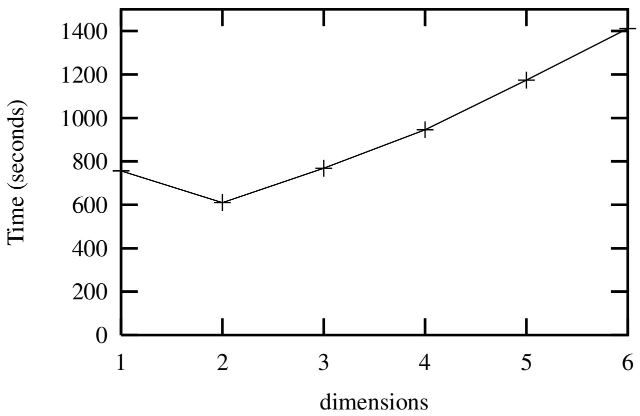

Figure 3: Total times for the DEC 2100 as the number of dimensions increases.

In particular, the input size was $N = 2^{24}$ points, interpreted as a 1-dimensional vector of length 2^{24}, a $2^{12} \times 2^{12}$ matrix, a $2^8 \times 2^8 \times 2^8$ array, a $2^6 \times 2^6 \times 2^6 \times 2^6$ array, a $2^5 \times 2^5 \times 2^5 \times 2^5 \times 2^4$ array, and a $2^4 \times 2^4 \times 2^4 \times 2^4 \times 2^4 \times 2^4$ array. All other parameters are the same as in the first set of runs. Figure 3 shows the total and normalized times in both tabular and graphical form. (The number of butterfly operations remains $(N/2) \lg N$ regardless of the number of dimensions.) With the exception of the 1-dimensional case, the times increase roughly linearly with the number of dimensions, which corresponds well to the discussion of I/O complexity in Section 5.

Silicon Graphics Origin 2000 results

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

In the first set, we varied only the problem size and kept all other parameters fixed. Here, the problem sizes were $N = 2^{28}$ and 2^{30} points, interpreted as $2^{14} \times 2^{14}$ and $2^{15} \times 2^{15}$ matrices, respectively. Each run used a memory size of 2^{28} bytes per processor, or 2^{31} bytes over all eight processors, corresponding to $M = 2^{27}$ records over the entire system. The block size was $B = 2^{13}$ records, and $P = D = 8$. Figure 4 shows the total times and the normalized times for the two problem sizes. On the Origin 2000, it takes only about 1.7 hours to compute a $2^{15} \times 2^{15}$-point FFT. Normalized times vary only by about 7.5% between the two runs.

lg N	Total time (secs)	Normalized time (μsecs)
28	1332.00	0.354435
30	6137.91	0.381092

Figure 4: Total times and normalized times for the Origin 2000.

dimensions	Total time (secs)	Normalized time (μsecs)
1	50.45	0.250608
2	42.81	0.212641
3	54.13	0.268886
4	68.41	0.339798
5	88.22	0.438169
6	88.77	0.440948

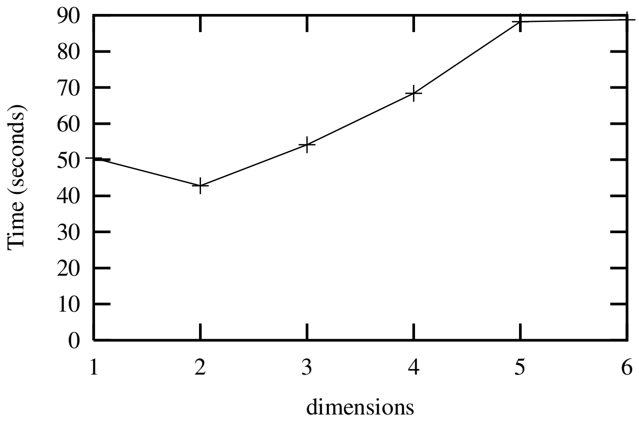

Figure 5: Total and normalized times for the Origin 2000 as the number of dimensions increases.

P, D	Total time (secs)	Work (processor-secs)
1	1316.32	1316.32
2	952.55	1905.09
4	495.16	1980.62
8	212.94	1703.54

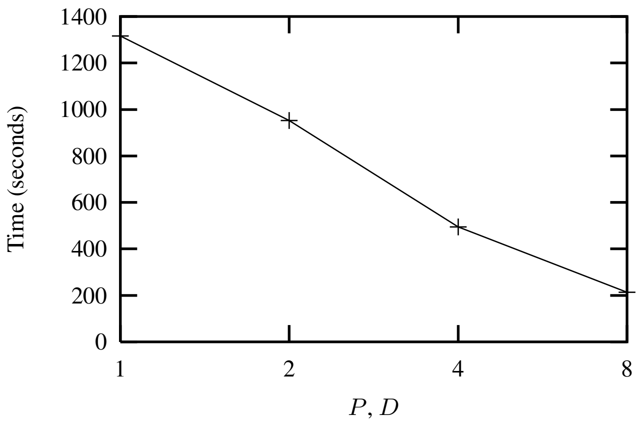

Figure 6: Total times and work for the Origin 2000 as the number of processors and disks increases.

lg N	Total time (secs)	Normalized time (μsecs)
28	2629.88	0.69979
30	11367.90	0.70581

Figure 7: Total times and normalized times for the Beowulf cluster.

dimensions	Total time (secs)	Normalized time (μsecs)
1	207.52	1.03075
2	200.18	0.99432
3	251.18	1.24760
4	334.37	1.66082
5	399.45	1.98407
6	471.52	2.34204

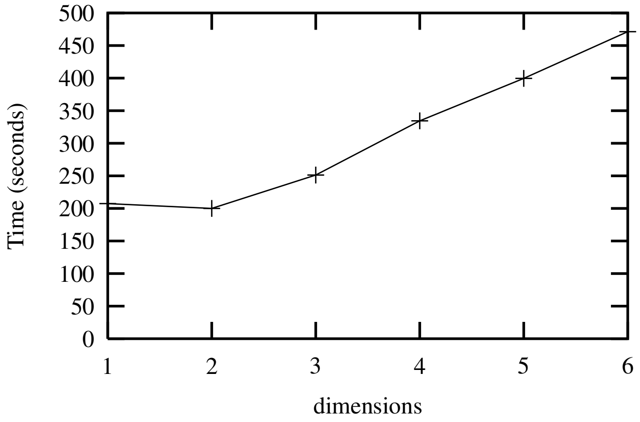

Figure 8: Total and normalized times for the Beowulf cluster as the number of dimensions increases.

In the second set, we varied the number of dimensions from 1 to 6, keeping all other parameters fixed at $N = 2^{24}$ points, $M = 2^{23}$ records, $B = 2^{13}$ records, and $P = D = 8$. Figure 5 shows the total and normalized times in both tabular and graphical form. Although the total time appears to flatten between 5 and 6 dimensions, our detailed measurements (data not given in this paper) show that the I/O times increase linearly with the number of dimensions, as our earlier analysis predicts. The flattening in the total time is due to computation within the BMMC-permutation subroutine taking less time for 6 dimensions than for 5.

In the third set of runs on the Origin 2000, we kept the problem size and memory per processor fixed, and we varied the number of processors and disks, maintaining the relationship $P = D$. Here, the problem size was $N = 2^{26}$ points, interpreted as a $2^{13} \times 2^{13}$ matrix. The memory size was 2^{26} bytes per processor. The number of processors varied among 1, 2, 4, and 8. Figure 6 shows the results. If the speedup were linear, the work (processors $\times$ total time) would be constant across all configurations. Instead, the work increases sharply between 1 and 2 processors because of additional computation and communication arising in the transition from 1 to 2 processors in the BMMC-permutation subroutine.

Beowulf cluster results

We performed three sets of runs on the Beowulf cluster, similar to the runs on the Origin 2000.

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

P, D	Total time (secs)	Work (processor-secs)
1	10685.70	10685.70
2	4509.73	9019.46
4	1982.01	7928.04
8	1317.77	10542.20

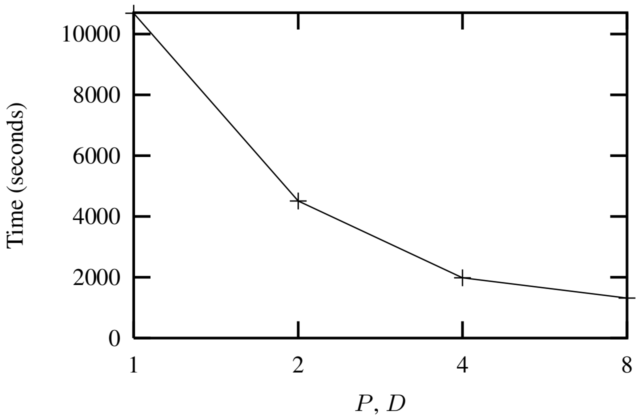

Figure 9: Total times and work for the Beowulf cluster as the number of processors and disks increases.

and 2^{30} points, interpreted as square matrices. Each run used a memory size of 2^{26} bytes per processor over $P = 16$ processors, for a total of 2^{30} bytes altogether, or $M = 2^{26}$ records. The block size was $B = 2^{13}$ records, and $P = D = 16$. Figure 7 shows the total times and the normalized times for the two problem sizes. Normalized times vary by under 1% between the two runs.

In the second set, we varied the number of dimensions from 1 to 6, keeping all other parameters fixed at $N = 2^{24}$ points, $M = 2^{23}$ records, $B = 2^{13}$ records, and $P = D = 8$. Figure 8 shows the total and normalized times, which increase roughly linearly between 2 and 6 dimensions, as predicted by our analysis.

In the third set of runs on the Beowulf cluster, we kept the problem size and memory per processor fixed, and we varied the number of processors and disks, maintaining the relationship $P = D$. Here, the problem size was $N = 2^{26}$ points, interpreted as a $2^{13} \times 2^{13}$ matrix. The memory size was 2^{26} bytes per processor. The number of processors varied among 1, 2, 4, and 8. Figure 9 shows the results. The work amounts show that the speedup is neither linear nor even monotonic, due to nonlinear speedup within the I/O and communication components of the BMMC-permutation subroutine. We do not know the root causes of these nonlinearities.

7 Conclusion

We have seen one method for performing multidimensional, multiprocessor out-of-core FFTs with parallel disks. This method is the most apparent extension of previous work in the area, and it takes advantage of characteristics of multidimensional FFTs and BMMC permutations. We have also presented an exact I/O-complexity analysis of our FFT method, along with performance results on three platforms. Among our performance results, we see that we can perform a $2^{15} \times 2^{15}$ out-of-core FFT on an eight-processor Silicon Graphics Origin 2000 in under two hours.

The dimensional method described in this paper is neither the only, nor necessarily the fastest, way to compute multidimensional FFTs in this environment. The vector-radix method [DM84, HMCS77, Lim90, Riv77] is a promising contender. We recently adapted this algorithm to our environment in an implemention for 2-dimensional square matrices, and preliminary results show that the vector-radix method rivals our implementation of the dimensional method in performance. On some runs, primarily those on a uniprocessor, the dimensional method is faster, whereas on others, including most of our runs on a multiprocessor, vector-radix is faster. On average, when the dimensional method is faster, it is faster by only about 5%. On the other hand, when the vector-radix method is faster, it is faster by about 15%.

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

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

Acknowledgments

Many thanks to James Clippinger for his heroic efforts in porting our implementation to the Origin 2000 and the Beowulf cluster. Additional thanks to Stuart Anderson, Jan Lindheim, and Tom Prince of Caltech with their help in using the Beowulf cluster. Michael Shin and Neal Young suggested the proof technique used in Section 5. Dan Rockmore, Matteo Frigo, Steven Johnson, and Hany Farid provided valuable background information on multidimensional FFTs. We appreciate the constructive suggestions made by anonymous reviewers on an earlier version of this paper.

References

[CC97] Thomas H. Cormen and James C. Clippinger. Performing BMMC permutations efficiently on distributed-memory multiprocessors with MPI. Technical Report PCS-TR97-317, Dartmouth College Department of Computer Science, May 1997. Accepted to *Algorithmica*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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