Dartmouth College Computer Science Technical Report
PCS-TR96-294
(Revised August 1997)

Performing Out-of-Core FFTs on Parallel Disk Systems

Thomas H. Cormen*
David M. Nicol†

Dartmouth College
Department of Computer Science

Abstract

The Fast Fourier Transform (FFT) plays a key role in many areas of computational science and engineering. Although most one-dimensional FFT problems can be entirely solved entirely in main memory, some important classes of applications require out-of-core techniques. For these, use of parallel I/O systems can improve performance considerably. This paper shows how to perform one-dimensional FFTs using a parallel disk system with independent disk accesses. We present both analytical and experimental results for performing out-of-core FFTs in two ways: using traditional virtual memory with demand paging, and using a provably asymptotically optimal algorithm for the Parallel Disk Model (PDM) of Vitter and Shriver. When run on a DEC 2100 server with a large memory and eight parallel disks, the optimal algorithm for the PDM runs up to 144.7 times faster than in-core methods under demand paging. Moreover, even including I/O costs, the normalized times for the optimal PDM algorithm are competitive, or better than, those for in-core methods even when they run entirely in memory.

1 Introduction

Fourier analysis plays a pivotal role in many branches of science and engineering. The Fourier transform's input is an N-vector of complex numbers, representing some discretized function. The Fourier representation of this function is a sum of N weighted sine and cosine functions with specific frequencies. Computing the coefficients of the constituent functions yields a great deal of information about the function. Well-known Fast Fourier Transform (FFT) techniques accomplish the computation in $\Theta(N \lg N)$ operations.

Since the modern discovery of the FFT by Cooley and Tukey in 1965 [CT65], a profusion of FFT methods have been developed, primarily to optimize it for different types of computer architectures such as vector and parallel machines (e.g., see Van Loan [Van92]). The work we present here

*Supported in part by funds from Dartmouth College and in part by the National Science Foundation under grants CCR-9308667 and CCR-9625894.

†This research was supported in part by NSF grants CCR-9201195 and NCR-9527163, and it was also supported in part by NASA Contract NAS1-19480 to the Institute for Computer Applications in Science and Engineering.

continues in that vein, looking at ways of organizing an FFT computation to take advantage of parallel I/O systems. Of course, such an endeavor is useful only if the input vector is too large to fit in the main memory of a computer; in most uses of the FFT, the input vector *will* fit in core.

Some critical applications require extremely large one-dimensional FFTs, particularly when the subject function exhibits critical phenomena at vastly different time scales and high resolution is required. One such application is seismic analysis [Cla85], where an out-of-core one-dimensional FFT is necessary (as part of a higher dimensional FFT) even when the computer memory has 16 gigabytes of available RAM [Rut96]. Another application is in the area of radio astronomy. The High-Speed Data Acquisition and Very Large FFTs Project at Caltech[1] uses FFTs to support searching for fast (millisecond period) pulsars. The project currently requires FFTs with 10 gigapoints, and it desires FFTs with up to 64 gigapoints. Yet another application is for integer multiplication of very large numbers [CF94], which is a key component in the most modern methods of searching for Mersenne prime numbers. FFTs are used in many ways to manipulate data sets, such as convolution/deconvolution, correlation/auto-correlation, filtering, and power spectrum estimation [PFTV88]. Any time the data set is very large and accuracy is essential, very large FFTs are required.

The contribution of the present paper is to present an out-of-core FFT algorithm that exploits parallel I/O and to assess its performance. The algorithm is a variant of one that was sketched by Vitter and Shriver [VS94], and which achieves the lower bound on complexity proven by Aggarwal and Vitter [AV88]. In particular, we show how efficient out-of-core permutation routines can be used throughout the FFT computation. We assess performance by comparison with demand paging; we show analytically and experimentally that well-known in-core FFT algorithms run slowly once the data set size exceeds available in-core memory. Using only a single-disk system, we observe that our out-of-core method runs over 46 times faster than demand paging; with eight disks we observe up to two orders of magnitude improvement using our technique.

The remainder of this paper is organized as follows. Section 2 summarizes some FFT methods for in-core computation, and Section 3 discusses published out-of-core FFT methods for single-disk systems. Section 4 demonstrates why conventional demand-paged in-core FFT algorithms perform badly when the problem size exceeds the physical memory. In Section 5, we define the Parallel Disk Model (PDM). Section 6 describes our out-of-core algorithm. Section 7 presents and analyzes running times for our FFT implementation on two different DEC Alpha-based uniprocessor systems. Finally, we summarize in Section 8.

2 In-core FFTs

This section reviews Fourier transforms and outlines some well-known FFT methods for in-core computation. For further background on the FFT, see any of the texts [CLR90, Nus82, Van92].

Discrete Fourier transforms

Fourier transforms are based on complex roots of unity. The *principal Nth root of unity* is a complex number $\omega_N = e^{2\pi i/N}$, where $i = \sqrt{-1}$. For any real number u, $e^{iu} = \cos(u) + i\sin(u)$.

Given a vector $a = (a_0, a_1, \ldots, a_{N-1})$, where N is a power of 2, the *Discrete Fourier Transform*

[1] See `http://www.cacr.caltech.edu/SIO/APPL/phy02.html`.

(DFT) is a vector $y = (y_0, y_1, \ldots, y_{N-1})$ for which

$$y_k = \sum_{j=0}^{N-1} a_j \omega_N^{jk} \qquad \text{for } k = 0, 1, \ldots, N-1 \; . \tag{1}$$

We also write $y = \mathrm{DFT}_N(a)$.

Fast Fourier Transforms

Viewed merely as a linear system, $\Theta(N^2)$ time is needed to compute vector y. The well-known *Fast Fourier Transform* technique requires only $\Theta(N \lg N)$ time, as follows. Splitting the summation in equation (1) into its odd- and even-indexed terms, we have

$$y_k = \sum_{j=0}^{N/2-1} \omega_{N/2}^{kj} a_{2j} + \omega_N^k \sum_{j=0}^{N/2-1} \omega_{N/2}^{kj} a_{2j+1} \; .$$

Each of these sums is itself a DFT of a vector of length $N/2$. When $0 \le k < N/2$, it is easy to see how to combine the results of these smaller DFTs. When $N/2 \le k < N$, it is easy to show that $\omega_{N/2}^{kj} = \omega_{N/2}^{(k-N/2)j}$ and $\omega_N^k = -\omega_N^{k-N/2}$. Hence, we can compute $y = \mathrm{DFT}_N(a)$ by the following recursive method:

1. Split a into $a^{\mathrm{even}} = (a_0, a_2, \ldots, a_{N-2})$ and $a^{\mathrm{odd}} = (a_1, a_3, \ldots, a_{N-1})$.
2. Recursively compute $y^{\mathrm{even}} = \mathrm{DFT}_{N/2}(a^{\mathrm{even}})$ and $y^{\mathrm{odd}} = \mathrm{DFT}_{N/2}(a^{\mathrm{odd}})$.
3. For $k = 0, 1, \ldots, N/2 - 1$, compute $y_k = y_k^{\mathrm{even}} + \omega_N^k y_k^{\mathrm{odd}}$ and $y_{k+N/2} = y_k^{\mathrm{even}} - \omega_N^k y_k^{\mathrm{odd}}$. The factor ω_N^k is often referred to as a *twiddle factor.*

By fully unrolling the recursion, we can view the FFT computation as Figure 1 shows. First, the input vector undergoes a bit-reversal permutation, and then a butterfly graph of $\lg N$ stages is computed. A *bit-reversal permutation* is a bijection in which the element whose index k in binary is $k_{N-1}, k_{N-2}, \ldots, k_0$ maps to the element whose index in binary is $k_0, k_1, \ldots, k_{N-1}$. In the sth stage of the butterfly graph, elements whose indices are 2^s apart (after the bit-reversal permutation) participate in a butterfly operation, as described in step 3 above. The butterfly operations in the sth stage can be organized into $N/2^s$ *groups* of 2^s operations each.

FFT algorithms

When the FFT is computed according to Figure 1 in a straightforward manner—left to right and top to bottom—the result is the classic Cooley-Tukey FFT method [CT65]. Several other methods have been developed to improve performance on vector machines and in memory hierarchies, by avoiding the bit-reversal permutation to improve locality of reference.

Stockham's method [Van92, pp. 49–58] eliminates bit-reversal by permuting the N values before each of the $\lg N$ stages of the butterfly network. Its memory requirement, however, is twice that of the Cooley-Tukey method.

Another method, attributed by Bailey [Bai90] to P. Swarztrauber as a variation of an algorithm by Gentleman and Sande, and also attributed to E. Granger by Brenner [Bre69], splits the summation of equation (1) into $\sqrt{N}$ summations each with $\sqrt{N}$ terms. (Here we take N to be a power of 4, but the method can be generalized). We split into $\sqrt{N}$ DFTs rather than two; each

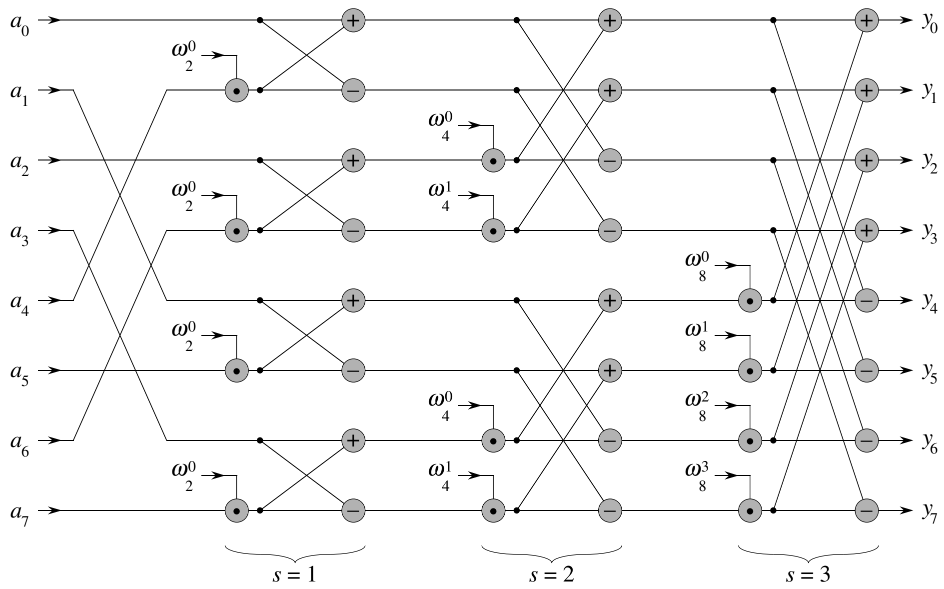

Figure 1: The FFT computation after fully unrolling the recursion, shown here with $N = 8$. Inputs $(a_0, a_1, \ldots, a_{N-1})$ enter from the left and first undergo a bit-reversal permutation. Then $\lg N = 3$ stages of butterfly operations are performed, and the results $(y_0, y_1, \ldots, y_{N-1})$ emerge from the right. This figure is taken from [CLR90, p. 796].

DFT is comprised of all terms whose indices are congruent modulo $\sqrt{N}$. The analog of a butterfly operation adds $\sqrt{N}$ terms (expressible as DFTs) that are computed by recursive calls to problems of size $\sqrt{N}$. This, Swarztrauber's method, is given by the following steps, which operate in place:

1. Treating the vector $a = (a_0, a_1, \ldots, a_{N-1})$ as a $\sqrt{N} \times \sqrt{N}$ matrix stored in row-major order, transpose it so that elements whose original indices are congruent modulo $\sqrt{N}$ appear in the same row.

2. Compute the DFT of each $\sqrt{N}$-element row individually.

3. Scale the resulting matrix by multiplying the entry in row j and column k by ω_N^{jk}.

4. Transpose the matrix.

5. Compute the DFT of each $\sqrt{N}$-element row individually.

6. Transpose the matrix and interpret it once again as an N-element vector to produce the result $y = (y_0, y_1, \ldots, y_{N-1})$.

This method runs in time $\Theta(N \lg N)$. Reliance on smaller DFTs improves locality in memory hierarchies. Experiments reported in Section 4 show this method to be nearly twice as fast as others on in-core computations.

3 Out-of-core FFTs

Here we briefly survey published out-of-core, single-disk, one-dimensional FFT algorithms.

Note that an out-of-core method based on Swarztrauber's method is easy when $M \leq N \leq M^2$, because each $\sqrt{N}$-sized DFT fits in memory. This relation between M and N is entirely reasonable given contemporary memory sizes and prices. The method does require an out-of-core matrix-transpose subroutine to accomplish steps 1, 4, and 6. Bailey recommends an algorithm by Fraser [Fra76] for BPC (bit-permute/complement) permutations on one disk, whereas Brenner details a transposition algorithm.

When the problem size just barely exceeds the memory size, Brenner suggests a method developed by W. Ryder. This method, which is a specialization of Swarztrauber's method, eliminates the first two matrix transpositions. The cost of doing so, however, is that the computation time contains a term proportional to N^2/M, so that if $N \gg M$, the computation time is very high.

Sweet and Wilson [SW95] use an extension of Swarztrauber's method to perform FFTs even when $N > M^2$ on the CM-5 using a Scalable Disk Array (SDA) [TMC92], which appears to the programmer as one large disk. The method used by Sweet and Wilson requires an out-of-core bit-reversal permutation, and they use Fraser's algorithm.

The algorithm we present in Section 6 fleshes out the details of a sketch given by Vitter and Shriver [VS94]. Because they focus on pebbling the butterfly graph, some essential steps are omitted from their description (e.g., the implementation of an efficient out-of-core bit-reverse permutation); nevertheless their paper is properly viewed as the basis for our work.

4 Performance of FFTs with demand paging

In this section, we show that the in-core FFT methods described earlier perform poorly under demand paging once the problem size exceeds the available memory. In particular, we show that the number of page faults for the Cooley-Tukey bit-reversal computation is proportional to N and that even under the best of conditions the butterfly steps for all methods suffer from a poor computation-to-I/O ratio. We substantiate our conclusions with experimental results.

Analysis of bit reversal

The following pseudocode expresses an in-place bit-reversal permutation of N-element array A:

for $j \leftarrow 0$ **to** $N-1$
  do let j' be the $\lg N$-bit reversal of j
    if $j < j'$
      then exchange $A[j] \leftrightarrow A[j']$

Theorem 1 *Suppose that the in-place bit-reversal permutation code above is performed under demand paging with least-recently-used page replacement. Suppose further that there are $N = 2^n$ elements in the array, the physical memory can hold $M = 2^m$ elements, and each page holds $B = 2^b$ elements, where n, m, and b are positive integers, $N \geq 2M$, and $N \geq 2B$. Finally, assume that the array A starts at a page boundary and that no pages of A are initially in memory. Then the bit-reversal permutation induces at least $N/4$ page faults.*

Proof: We will show that each element of the set $F = \{j : 0 < j < N/2 \text{ and } j \text{ is odd}\}$ induces a page fault. Noting that $|F| = N/4$ will then prove the theorem.

Observe that for each element $j \in F$, we have $j < j'$, since j has a most significant bit of 0 and a least significant bit of 1. Thus, the exchange of $A[j]$ and $A[j']$ will occur for each $j \in F$. Let F' be the set of destination pages referenced when processing members of F.

We compute which page an element is on as follows. For a given n-bit index into A, the least significant b bits give the position on the page, and the most significant $n-b$ bits give the page number. Thus, the elements of A that are destined for the same page p have the same value in their *least* significant source indices.

To determine whether a given reference to $A[j']$ causes a page fault, we compute the "stack distance" for the page containing $A[j']$. The *stack distance* [MGST70] of a reference to page p is one plus the number of uniquely different pages referenced since the most recent reference to page p. A reference to a page causes a page fault if and only if the stack distance of that reference exceeds the number of pages that memory can hold, which is exactly M/B. By our assumption that no pages of A are initially in memory, we consider the stack distance to the first reference to a page of A to be infinite.

Next we show that for each page $p \in F'$, as we progress through the values $j = 0, 1, \ldots, N/2-1$, the stack distance between successive references to page p is greater than $N/2B$. Once a reference is made to destination page p, another $N/B-1$ values of j will be considered before the next reference to page p. Of these, $N/2B-1$ are in F and thus cause a reference to a unique destination page in F'. The page containing index j is also referenced, and this page is not in F', and so at least $N/2B$ distinct pages are referenced. As long as no value $j \in F$ resides on the same page as its destination index j', the stack distance between successive references to page p is strictly greater than $N/2B$. But because $N \geq 2B$, there are at least two pages in the array A, and because A also starts at a page boundary, no element in the first $N/2$ positions resides on the same page as an element in the last $N/2$ positions. Since each element $j \in F$ is in the first $N/2$ positions and maps to an element in the last $N/2$ positions, we conclude that the stack distance is indeed greater than $N/2B$.

Because $N \geq 2M$, we have that $N/2B \geq M/B$, and so each reference to a page of F' causes a page fault. Since references to pages in F' are induced by source elements in F, we see that each time a member of F is processed, a page fault ensues, which completes the proof. ∎

The proof of Theorem 1 substantially undercounts page faults. A more extensive analysis using similar ideas shows that the number of page faults is at least $(N/2 - 2\sqrt{N})(1 - 2/(N/M)^2)$.

Analysis of butterfly stages

All of the FFT methods that we have discussed exhibit relatively good locality when executing each butterfly stage. For both Cooley-Tukey and Stockham, each butterfly stage essentially sweeps through all the data pages, exactly once, with no more than 2 data pages actively in use at a time. Swarztrauber's method exhibits more complex behavior because of the matrix transposes, but its constituent butterflies act like the other two methods. The essential point to be noted is that during a butterfly stage, each data point is updated once by a complex addition/subtraction (two floating-point operations), and half the data points also involve a complex multiplication (six floating-point operations). A typical 8 KB data page contains 512 points, and so it entails 2560 floating-point operations. The time required to fault in a data page is on the order of 10^{-2} seconds (most of which is independent of the page size), but the time to process that page is about an order of magnitude less. Even with much better locality than the bit-reversal computation, demand-paged FFT suffers greatly from waiting for I/O to complete. We can mitigate this bottleneck by either increasing the size of block fetched per I/O, and/or by prefetching memory blocks. Our out-of-core technique does both.

Problem size	Method					
	Cooley-Tukey		Stockham		Swarztrauber	
	Seconds	Normalized	Seconds	Normalized	Seconds	Normalized
$N = 2^{16}$	1.54558	*1.47398*	2.15616	*2.05627*	1.13762	*1.08492*
$N = 2^{17}$	3.36112	*1.50843*	4.64709	*2.08556*		
$N = 2^{18}$	7.25278	*1.53707*	9.85693	*2.08896*	5.01269	*1.06233*
$N = 2^{19}$	15.5745	*1.56347*	20.9941	*2.10753*		
$N = 2^{20}$	35.4236	*1.68913*	44.6760	*2.13032*	24.6568	*1.17573*
$N = 2^{21}$	75.1581	*1.70658*	972.035	*22.0715*		
$N = 2^{22}$	11591.7	*125.621*	2022.26	*21.9157*	443.147	*4.80248*
$N = 2^{23}$	42553.5	*220.555*	4097.72	*21.2385*		
$N = 2^{24}$			8746.85	*21.7230*	2226.75	*5.53019*

Table 1: Running times for the three in-core FFT methods on the workstation zayante, with 64 MB of memory. For each method and problem size, we show the time in seconds and also the normalized time (italics, in microseconds) which is the running time divided by $N \lg N$.

Experimental results

Here we present running times of the three demand-paged in-core FFT methods (Cooley-Tukey, Stockham, and Swarztrauber). They were coded in C, compiled using gcc with O2 optimization, and run on a DEC 3000 Alpha-based workstation running Digital UNIX V3.2C. The workstation, named zayante, has a clock cycle of 175 MHz, 64 MB of memory, and a 512 MB virtual-address space.

Table 1 gives running times. The Cooley-Tukey and Swarztrauber methods both use $16N$ bytes; Stockham uses $32N$ and so experiences heavy paging one problem size earlier than the others. Because our implementation of Swarztrauber's method requires N to be a power of 4, timings for odd powers of 2 are omitted.

From Table 1, we see the effects of demand paging. By avoiding bit-reversal, the Stockham and Swarztrauber methods do not experience the degree of thrashing suffered by Cooley-Tukey. (In fact, we did not even run Cooley-Tukey for $N = 2^{24}$, anticipating a run time of about a day.) Swarztrauber's method is notably faster in each case, probably due to its substantially better locality in cache. Nevertheless, we shall see in Section 7 that our explicit out-of-core algorithm runs faster than Swarztrauber's method on the same system for a problem size of $N = 2^{24}$.

5 The Parallel Disk Model

This section describes the Parallel Disk Model [VS94]. We shall use this model in Section 6 to design an out-of-core FFT algorithm.

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.[2] Any disk access transfers an entire block of records. Disk I/O transfers records between the disks and an M-record *random-access memory*. Any set of M records is a *memoryload*. Each *parallel I/O operation* transfers up to D blocks between the disks and memory, with at most one

[2]A block might consist of several sectors of a physical device or, in the case of RAID [CGK+88, Gib92, PGK88], sectors from several physical devices.

	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
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 2: The layout of $N = 64$ records in a parallel disk system with $B = 2$ and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

block transferred per disk, for a total of up to BD records transferred. The most general type of parallel I/O operation is *independent I/O*, in which the blocks accessed in a single parallel I/O may be at any locations on their respective disks. A more restricted operation is *striped I/O*, in which the blocks accessed in a given operation must be at the same location on each disk.

We assess an algorithm by the number of parallel I/O operations it requires. While this does not account for unavoidable variation in disk-access times, the number of disk accesses can be minimized by carefully designed algorithms.

We place some restrictions on the PDM parameters. We assume that B, D, M, and N are exact powers of 2. For convenience, we define $b = \lg B$, $m = \lg M$, and $n = \lg N$. We assume that $BD \leq M$ in order to fully utilize disk bandwidth, and of course we assume that $M < N$.

The PDM lays out data on a parallel disk system as shown in Figure 2. A *stripe* consists of the D blocks at the same location on all D disks. A record's index is 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 most significant $n - m$ bits of an index indicate its memoryload number.

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. The FFT algorithm we implemented has an I/O complexity of $\Theta\left(\frac{N}{BD}\frac{\lg(N/B)}{\lg(M/B)}\right)$, which appears to be the analogue of the $\Theta(N \lg N)$ bound seen for so many sequential algorithms on the standard RAM model.

6 An explicit out-of-core FFT algorithm for the PDM

By taking full advantage of a parallel disk system, we can get considerably better out-of-core FFT performance than we get by using just demand paging. This section presents an explicit out-of-core FFT algorithm designed for the PDM. The key idea is to redraw the butterfly graph by inserting permutations. We then recognize that bit-reversal and the added permutations belong to the larger class of BMMC permutations. We use a prior out-of-core BMMC algorithm to produce an efficient out-of-core FFT.

BMMC permutations on the PDM

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 is nonsingular (i.e., invertible) over $GF(2)$.[3] The specification also includes a *complement vector* $c = (c_0, c_1, \ldots, c_{n-1})$ of

[3]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.

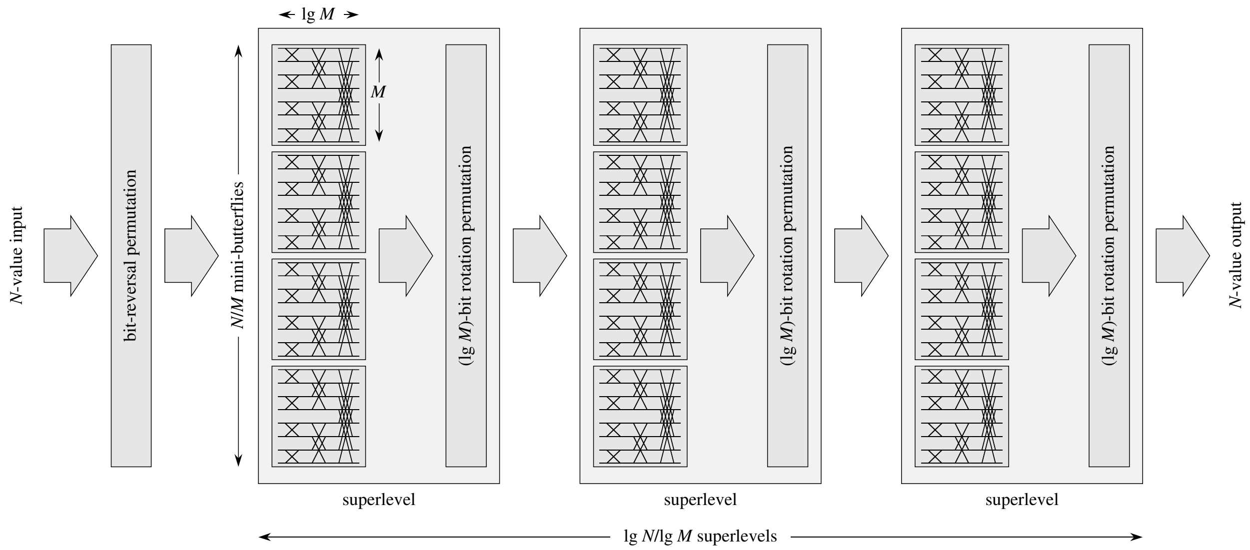

Figure 3: The structure of the out-of-core FFT algorithm for the PDM. After a bit-reversal permutation, we perform $\lg N / \lg M$ superlevels. Each superlevel consists of N/M mini-butterflies on M values, followed by a $(\lg M)$-bit right-rotation permutation on the entire array.

length n. Treating each source index x as an n-bit vector, we perform matrix-vector multiplication over $GF(2)$ and then form the corresponding n-bit target index z by complementing some subset of the resulting bits: $z = H\,x \oplus c$. As long as the characteristic matrix H is nonsingular, the mapping of source indices to target indices is one-to-one.

A very efficient algorithm for BMMC permutations on the PDM appears in [CSW94]. This algorithm requires at most $\frac{2N}{BD}\left(\left\lceil\frac{\operatorname{rank}\gamma}{\lg(M/B)}\right\rceil + 2\right)$ parallel I/Os, where γ is the lower left $\lg(N/B) \times \lg B$ submatrix of the characteristic matrix, and the rank is computed over $GF(2)$. (Note that because of the dimensions of γ, its rank is at most $\lg\min(N/B, B)$.) This number of factors is asymptotically optimal and is very close to the best known exact lower bound.

We shall use two types of BMMC permutations to perform the FFT. Both use a complement vector that is all 0s.

Bit-reversal permutation: The characteristic matrix has 1s on the antidiagonal and 0s elsewhere. The submatrix γ has as much rank as possible, so that $\operatorname{rank}\gamma = \lg\min(B, N/B)$.

***k*-bit right-rotation:** We rotate the bits of each index k bits to the right, wrapping around at the rightmost position. The characteristic matrix is formed by taking the identity matrix and rotating its columns k positions to the right, and $\operatorname{rank}\gamma \leq \min(k, \lg B, \lg(N/B))$.

Redrawing the butterfly

Figure 3 shows the structure of our algorithm. This redrawing of the butterfly was devised by Snir [Sni81] and is implicitly used in the FFT algorithm of Vitter and Shriver [VS94]. Assume for the moment that $\lg M$ divides $\lg N$. As in the Cooley-Tukey method, we start with a bit-reversal permutation. Then there are $\lg N / \lg M$ *superlevels*, where each superlevel consists of N/M separate "mini-butterflies" followed by a $(\lg M)$-bit right-rotation permutation on the entire array.

Each *mini-butterfly* is a butterfly graph on M values, and hence it has depth $\lg M$ and a sequential running time of $\Theta(M \lg M)$. The size M of a mini-butterfly is chosen so that each mini-butterfly is computed by reading in a memoryload, computing the mini-butterfly graph, and writing out the memoryload.

Analysis

This FFT algorithm consists of one bit-reversal permutation followed by $\lg N / \lg M$ superlevels. As noted above, the bit-reversal permutation requires at most $\frac{2N}{BD}\left(\left\lceil \frac{\lg \min(B,N/B)}{\lg(M/B)} \right\rceil + 2\right)$ parallel I/Os. Each superlevel requires $2N/BD$ parallel I/Os to read and write all N/M mini-butterflies plus at most $\frac{2N}{BD}\left(\left\lceil \frac{\lg \min(B,M,N/B)}{\lg(M/B)} \right\rceil + 2\right)$ parallel I/Os to perform the $(\lg M)$-bit rotation permutation. Since the PDM requires that $BD \le M$ and $D \ge 1$, we have $B \le M$, and so the $\lg M$ factor in the numerator drops out. Asymptotically, the number of parallel I/O operations is $\Theta\left(\frac{N}{BD} \frac{\lg N}{\lg M}\left(1 + \frac{\lg \min(B,N/B)}{\lg(M/B)}\right)\right)$, which can be shown via simple manipulations to equal the lower bound of $\Omega\left(\frac{N}{BD} \frac{\lg(N/B)}{\lg(M/B)}\right)$ proven by Aggarwal and Vitter [AV88].

Handling general values of N and M

If $\lg M$ does not divide $\lg N$, then we compensate in the last superlevel. Rather than computing mini-butterflies of depth $\lg M$ in the last superlevel, we compute mini-butterflies of depth $r = (\lg N) \bmod (\lg M)$, which is the number of levels of the full butterfly graph not yet computed. We can still read and write memoryloads of M values, but now each memoryload in the last superlevel consists of $M/2^r$ mini-butterflies.

Out-of-core Swarztrauber's method

If $M < N \le M^2$, we could use an explicit out-of-core version of Swarztrauber's method. The matrix-transpose steps are BMMC permutations, since exchanging row and column numbers within an index is a $((\lg N)/2)$-bit rotation permutation. Thus, there would be three BMMC permutations, which is just as many as the our algorithm performs when $M < N \le M^2$. (Our algorithm has the further advantage of working even when $N > M^2$.)

Moreover, the BMMC permutations that an out-of-core Swarztrauber implementation would perform are no faster than those of our algorithm. When done out-of-core, transposing a square matrix takes just as long as a bit-reversal permutation or a $(\lg M)$-bit right rotation. The in-core portions of an out-of-core Swarztrauber algorithm would also have to perform in-core bit-reversal permutations, and so they would be slower than the in-core portions of our algorithm. Consequently, we did not implement an explicit out-of-core version of Swarztrauber's method.

Implementation notes

This section concludes with some notes on the implementation of our out-of-core FFT algorithm.

We start with the twiddle factors, which were omitted in the above description. The butterfly operations in Figure 3 proceed in $\lg N$ levels from left to right, just as in Figure 1. If we number these levels from 1 to $\lg N$, then all twiddle factors of the lth level are powers of ω_{2^l}. We obtain these powers of ω_{2^l} efficiently by directly computing the exponent of the twiddle factor in superlevel s, mini-butterfly q within the superlayer (starting from 0, and the range of q depends on the superlayer), and the jth butterfly within a group of butterflies as $\left\lfloor \frac{qM^{s+1}}{M^{\lceil \lg N / \lg M \rceil}} \right\rfloor + jM^s$. This computation is easy to move into loops and avoids expensive sine and cosine calls.

Problem size (points)	Memory size (bytes)							
	2^{22}		2^{23}		2^{24}		2^{25}	
	sync	async	sync	async	sync	async	sync	async
2^{22}	440.713 *4.77610*	505.569 *5.47896*	414.395 *4.49088*	372.099 *4.03251*	402.335 *4.36019*	403.913 *4.37729*	437.361 *4.73977*	456.378 *4.94586*
2^{23}	1242.05 *6.43756*	1141.72 *5.91755*	846.030 *4.38498*	779.431 *4.03980*	857.528 *4.44458*	856.236 *4.43788*	931.019 *4.82548*	923.335 *4.78566*
2^{24}	2479.13 *6.15699*	2240.50 *5.56434*	1939.60 *4.81705*	2221.19 *5.51639*	1699.27 *4.22018*	1757.30 *4.36430*	1785.00 *4.43310*	1692.25 *4.20275*
2^{25}	4851.86 *5.78387*	4685.00 *5.58496*	4827.49 *5.75482*	4573.11 *5.45157*	3412.08 *4.06752*	3461.50 *4.12643*	3539.99 *4.22000*	3318.47 *3.95592*
2^{26}	10714.9 *6.14094*	11772.7 *6.74719*	9558.68 *5.47829*	8911.21 *5.10721*	7689.57 *4.40706*	8751.93 *5.01592*	7495.07 *4.29559*	7581.10 *4.34489*

Table 2: Running times for the out-of-core algorithm on zayante with one disk, varying problem and memory sizes, and both synchronous and asynchronous I/O. Times are in seconds, and in italics are the normalized times (the running time divided by $N \lg N$) in microseconds.

The ViC* interface [CH97] provides the appearance of the PDM when performing parallel I/O operations. The interface is portable, and it is implemented as a set of wrappers on top of an existing serial or parallel file system. Here, we used an implementation on top of a traditional UNIX file system (UFS), but with multiple disks.

The BMMC permutation subroutine is taken from the implementation in [CH97]. It calls the ViC* interface to perform striped reads and independent writes. It is carefully optimized for both in-core computation and I/O.

Finally, we implemented the FFT algorithm with both synchronous (i.e., blocking) and asynchronous (non-blocking) I/O calls; the ViC* interface supports both. With asynchronous I/O, as we compute the butterflies of the qth memoryload, we simultaneously prefetch the data of the $(q+1)$st memoryload and write behind the computed data of the $(q-1)$st memoryload. The reduced latency does not come for free, however, as we must allocate prefetch and write-behind buffers of the same size as the compute buffer. Thus, the *effective memory size*, i.e., the value of M used in the algorithm, is smaller with asynchronous I/O than with synchronous I/O. Because we carve memory into three parts and M must be a power of 2, asynchronous I/O reduces the effective memory size by a factor of 4. Context switching is an additional cost, as one kernel-level thread serves each physical disk and is switched in to handle I/O initiation and completion. Nevertheless, we shall see in Section 7 that asynchronous I/O is usually worthwhile.

7 Performance of the out-of-core FFT algorithm

This section presents timing results for the out-of-core FFT algorithm on two different DEC Alpha platforms. In all cases, block sizes were 2^{16} bytes.

We start with a direct comparison of our algorithm and the in-core methods running with demand paging on zayante. With our algorithm, we used $D = 1$ disk and varied the memory size on zayante from 2^{22} to 2^{25}. Using only one disk for the our algorithm makes for a fair comparison to demand paging, since there is only one swap disk. Table 2 shows running times with both

Problem size (points)	Memory size (bytes) 2^{26} sync	2^{26} async	2^{27} sync	2^{27} async
2^{23}	340.659 *1.76564*	293.921 *1.52340*		
2^{24}	799.221 *1.98489*	674.864 *1.67604*	835.317 *2.07453*	714.364 *1.77414*
2^{25}	1718.09 *2.04812*	1541.35 *1.83743*	1712.90 *2.04194*	1458.18 *1.73829*
2^{26}	3500.04 *2.00595*	3092.14 *1.77217*	3496.10 *2.00369*	3054.42 *1.75055*
2^{27}	7232.63 *1.99583*	6226.17 *1.71810*	7105.92 *1.96086*	6252.80 *1.72544*
2^{28}	14671.7 *1.95201*	12695.0 *1.68902*	14243.6 *1.89506*	12597.9 *1.67610*
2^{29}	30319.8 *1.94741*	26431.9 *1.69770*	30281.2 *1.94494*	26173.6 *1.68111*

Table 3: Running times for the out-of-core algorithm on adams with 8 disks, varying problem and memory sizes, and both synchronous and asynchronous I/O. Times are in seconds, and in italics are the normalized times (the running time divided by $N \lg N$) in microseconds. With 2^{27} bytes of memory, a 2^{23}-point FFT fits in memory, so this timing is omitted.

synchronous and asynchronous I/O. In some cases, the asynchronous time exceeds the synchronous time because, we believe, having one processor running both threads (main computation and disk server) causes context switches during butterfly computations and BMMC permutations. Also, in some cases using more memory does not help. Note, however, that at the problem sizes at which the in-core algorithms encounter heavy paging—$N \geq 2^{22}$ for Cooley-Tukey and Swarztrauber—our out-of-core algorithm is faster if it has enough memory to work with. (At $N = 2^{23}$, our algorithm with 32 MB of memory and asynchronous I/O is over 46 times faster than Cooley-Tukey.) Considering the overhead due to the ViC* wrappers and UFS calls, it is impressive that our algorithm can run faster than even Swarztrauber's method.

Table 3 shows running times on a different system, named adams, which is a DEC 2100 server with two 175-MHz Alpha processors, 320 MB of memory, and eight 2-GB disks for data (so that $D = 8$). It has the same software environment as zayante, but with eight disks, its I/O bandwidth is much higher. Compared to the in-core methods in Table 1 even when they run entirely in memory, the normalized times (which do include I/O time) are at worst slightly higher and in some cases even lower! In one case ($N = 2^{23}$), the running time on adams is 144.7 times lower than Cooley-Tukey on zayante. The operating system may choose to run a ready thread on either processor, and so disk-server threads do not interfere with butterfly computations as much as on zayante. Consequently, on adams it is always faster to use asynchronous I/O than to use synchronous I/O.

8 Conclusion

We have examined both analytically and experimentally two classes of methods for computing large Fourier transforms. In-core FFT algorithms run slowly when they execute in a demand-paging environment. Of the three that we examined, Swarztrauber's method is by far the fastest and has the best locality of reference. The explicit out-of-core method that we developed for the PDM is asymptotically optimal in this model, and it has good empirical performance. On a DEC 2100 server with two processors, large memory, and eight data disks, our algorithm's normalized time is competitive with in-core methods, even when they run entirely in memory.

One might ask whether there is an FFT method that is designed to be in-core but is also designed to work well in the presence of demand paging. The authors know of no such algorithm. Swarztrauber's method, however, does work well in the presence of demand paging, even though we do not believe it was designed for out-of-core problems. It reduces the problem size so quickly that subproblems tend to fit in memory. Yet our out-of-core method beat Swarztrauber's.

Although it uses both processors, our current DEC 2100 implementation is essentially a uniprocessor implementation. Our own breakdowns of running times on large problems show that computation time is a bottleneck. We have begun to investigate true parallel out-of-core algorithms, using parallelized versions of the permutation methods described in this paper [CWN97].

Acknowledgments

We thank Barry Fagin, Peter Highnam, David Keyes, and Jeff Rutledge for pointing us to applications of out-of-core FFTs, and also Dennis Healy and Eric Schwabe for their help in describing the mathematical structure of FFTs. Melissa Hirschl wrote the ViC* wrappers. David Kotz and Wayne Cripps advised us in sundry systems issues. The DEC 2100 server named adams was funded in part by an equipment allowance from Digital Equipment Corporation.

References

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[Bai90] David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4:23–35, 1990.

[Bre69] Norman M. Brenner. Fast Fourier transform of externally stored data. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):128–132, June 1969.

[CF94] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation*, 62(205):305–324, January 1994.

[CGK+88] Peter Chen, Garth Gibson, Randy H. Katz, David A. Patterson, and Martin Schulze. Two papers on RAIDs. Technical Report UCB/CSD 88/479, Computer Science Division (EECS), University of California, Berkeley, December 1988.

[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.

[Cla85] Jon F. Claerbout. *Imaging the Earth's Interior*. Blackwell Scientific Publications, 1985.

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

[CSW94] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Department of Computer Science, July 1994. Preliminary version appeared in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures.* Revised version to appear in *SIAM Journal on Computing.*

[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. Technical Report PCS-TR97-303, Dartmouth College Department of Computer Science, January 1997. To appear in IOPADS '97.

[Fra76] Donald Fraser. Array permutation by index-digit permutation. *Journal of the ACM*, 23(2):298–309, April 1976.

[Gib92] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* The MIT Press, Cambridge, Massachusetts, 1992. Also available as Technical Report UCB/CSD 91/613, Computer Science Division (EECS), University of California, Berkeley, May 1991.

[MGST70] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 12(2):78–117, 1970.

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, New York, second edition, 1982.

[PFTV88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vettering. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, 1988.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM International Conference on Management of Data (SIGMOD)*, pages 109–116, June 1988.

[Rut96] Jeff Rutledge. Private communication, 1996.

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

[SW95] Roland Sweet and John Wilson. Development of out-of-core fast Fourier transform software for the Connection Machine. URL `http://www-math.cudenver.edu/~jwilson/final_report/final_report.html`, December 1995.

[TMC92] CM-5 scalable disk array. Thinking Machines Corporation glossy, November 1992.

[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.