Dartmouth College Computer Science Technical Report
PCS-TR97-303

Multiprocessor Out-of-Core FFTs with Distributed Memory and Parallel Disks
(Extended Abstract)

Thomas H. Cormen*
Jake Wegmann
David M. Nicol†

Dartmouth College
Department of Computer Science

Abstract

This paper extends an earlier out-of-core Fast Fourier Transform (FFT) method for a uniprocessor with the Parallel Disk Model (PDM) to use multiple processors. Four out-of-core multiprocessor methods are examined. Operationally, these methods differ in the size of "mini-butterfly" computed in memory and how the data are organized on the disks and in the distributed memory of the multiprocessor. The methods also perform differing amounts of I/O and communication. Two of them have the remarkable property that even though they are computing the FFT on a multiprocessor, all interprocessor communication occurs outside the mini-butterfly computations. Performance results on a small workstation cluster indicate that except for unusual combinations of problem size and memory size, the methods that do not perform interprocessor communication during the mini-butterfly computations require approximately 86% of the time of those that do. Moreover, the faster methods are much easier to implement.

1 Introduction

This paper extends earlier work [CN96] in performing out-of-core Fast Fourier Transforms (FFTs) on parallel disk systems. Whereas the implementation described in [CN96] performs out-of-core FFTs on a uniprocessor with parallel disks, the present paper examines four related ways to perform out-of-core FFTs on a multiprocessor with a distributed memory and parallel disks.

The study in [CN96] showed that an FFT algorithm explicitly designed for out-of-core problems on the Parallel Disk Model (PDM) [VS94] can significantly outperform traditional in-core FFT

*Contact author. Send correspondence to Dartmouth College Department of Computer Science, 6211 Sudikoff Laboratory, Hanover, NH 03755-3510 or to thc@cs.dartmouth.edu. Supported in part by the National Science Foundation under grant 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.

algorithms that run with demand paging on large problem sizes. Moreover, with careful design and implementation, the out-of-core uniprocessor algorithm with parallel disks can be competitive with in-core FFT methods even when they run entirely in memory. That study demonstrated rather convincingly that out-of-core FFT computations should use explicit disk I/O.

In the present paper, we adapt the FFT method used in [CN96] for multiple processors with a distributed-memory architecture. Conceptually, the method adapts easily. There are some design choices to be made, however, and this paper considers two of them. Section 4 examines these choices in more detail, but briefly they are described by the following parameters:

1. The *effective memory size* determines the size of the "mini-butterflies" computed in memory.
2. The *band size* parameterizes how the data are organized on the disks and in the distributed memory of the multiprocessor.

We consider two effective memory sizes and three band sizes. Of the six possible combinations, two make no sense to implement. We have implemented three of the remaining four, and this paper reports on the results. The algorithm used is I/O-optimal in the PDM for a given effective memory size and band size. This paper examines the differences among the four combinations in terms of total time, communication time, number of messages sent, and volume of messages sent.

Our results indicate that the best way overall uses an effective memory size small enough to avoid interprocessor communication during the mini-butterfly computations.

Of course, most one-dimensional FFT problems fit well within memory-size constraints. On the other hand, some problems do not fit in even very large memories. One example is the High-Speed Data Acquisition and Very Large FFTs Project at Caltech[1], which 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.

Although the literature contains some related work, the approach in this paper is unique. There have been a few papers on out-of-core FFTs on uniprocessors [Bai90, Bre69, CN96]. There are also some papers on in-core FFTs on multiprocessors [Cal96, JJK92, Swa87, Zhu90]; each of these papers assumes some interconnection network topology. The only previous out-of-core implementation for a multiprocessor of which we are aware is by Sweet and Wilson [SW95]. They use a CM-5 with a Scalable Disk Array [TMC92], which appears to the programmer as one large disk. The implementation in the present paper uses a PDM interface to access multiple disks independently and MPI [GLS94, SOHL+96] for interprocessor communication; there are no assumptions about the interconnection network topology.

The platform we use is a cluster of IBM RS6000 workstations with a FDDI network. Disk I/O operations are performed by calls to the ViC* API [CH96], which is implemented as a set of wrappers on top of the Galley File System [NK96a, NK96b]. The full paper will also include data for an IBM SP-2, also running Galley.

The remainder of this paper is organized as follows. Section 2 defines the Parallel Disk Model, and Section 3 summarizes the out-of-core uniprocessor algorithm for the PDM from [CN96]. Section 4 describes the modifications to the uniprocessor algorithm for a multiprocessor, detailing the effective memory size and band size parameters. Section 5 discusses the effects of these modifications on I/O and communication complexity. Section 6 compares the performance of the four methods on the network of RS6000 workstations. Finally, Section 7 presents some concluding remarks.

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

	$\mathcal{P}_0$				$\mathcal{P}_1$				$\mathcal{P}_2$				$\mathcal{P}_3$			
	$\mathcal{D}_0$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3$		$\mathcal{D}_4$		$\mathcal{D}_5$		$\mathcal{D}_6$		$\mathcal{D}_7$	
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.

2 The Parallel Disk Model

This section describes the Parallel Disk Model [VS94]. It is the underlying model for both the uniprocessor algorithm in Section 3 and the multiprocessor algorithm in Section 4.

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 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.

In this paper, we assume that there are P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ connected by a network. The M-record memory is distributed among the P processors so that each processor holds M/P records. The implementation of the PDM we use is the ViC* API [CH96], in which $D \geq P$ 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 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 P, B, D, M, and N are exact powers of 2. For convenience, we define $p = \lg P$, $b = \lg B$, $m = \lg M$, 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 1. A *stripe* consists of the D blocks at the same location on all D disks. A record's index is an n-bit vector. In Section 4, 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),

[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.

- $\lg D = d$ 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. The FFT algorithms in this paper have an I/O complexity of $\Theta\left(\frac{N}{BD}\frac{\lg\min(B,N/B)}{\lg(M/B)}\right)$, which appears to be the analogue of the $\Theta(N\lg N)$ bound seen for so many sequential algorithms on the standard RAM model.

3 The uniprocessor out-of-core FFT algorithm

This section summarizes the uniprocessor out-of-core FFT algorithm from [CN96]. We will modify the uniprocessor algorithm in Section 4 to devise multiprocessor versions.

Traditional FFTs

The out-of-core algorithm is based on a redrawing of the butterfly graph, so we start by reviewing the traditional approach of computing FFTs in-core by computing the butterfly graph.

The FFT is a particular method of computing the *Discrete Fourier Transform (DFT)* of an N-element vector. Given a vector $a = (a_0, a_1, \ldots, a_{N-1})$, where N is a power of 2, the *Discrete Fourier Transform (DFT)* is a vector $y = (y_0, y_1, \ldots, y_{N-1})$ for which

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

where $\omega_N = e^{2\pi i/N}$ and $i = \sqrt{-1}$. For any real number u, we can directly compute $e^{iu} = \cos(u) + i\sin(u)$.

Figure 2 shows the butterfly graph as it is used in computing an FFT, drawn for $N = 8$. First, the input vector undergoes a bit-reversal permutation. 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 $(k_0, k_1, \ldots, k_{N-1})$. After the bit-reversal permutation, a butterfly graph of $\lg N$ stages is computed. 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. The butterfly operations in the sth stage can be organized into $N/2^s$ *groups* of 2^s operations each. Each butterfly operation has a third input, known as a *twiddle factor*. The twiddle factor for a butterfly operation in stage s and the jth butterfly within a group ($0 \le j < 2^{s-1}$) is $\omega_{2^s}^{j}$.

Redrawing the butterfly graph for out-of-core FFTs

Figure 3 shows the structure of the out-of-core algorithm. This redrawing of the butterfly was devised by Snir [Sni81] and is implicitly used in the FFT algorithm for the PDM devised by Vitter and Shriver [VS94].

We describe the out-of-core algorithm in terms of an *effective memory size* F, which is a power of 2 in the range $1 \le F \le M$. Assume for the moment that $\lg F$ divides $\lg N$. As before, we start with a bit-reversal permutation. Then there are $\lg N/\lg F$ *superlevels*, where each superlevel consists of N/F separate "mini-butterflies" followed by a permutation on the entire array.

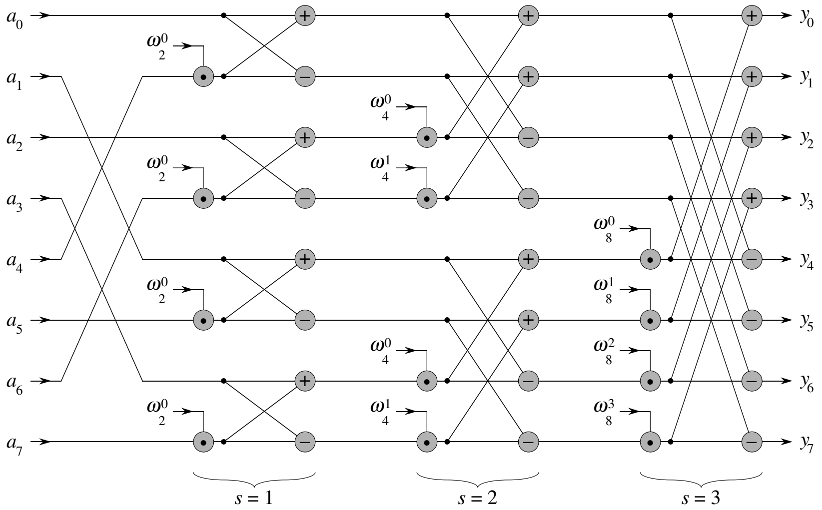


Figure 2: The FFT computation after fully unrolling the recursion, shown here with $N = 8$. Inputs $(a_0, a_1, \ldots, a_{N-1})$ enter from the left and first undergo a bit-reversal permutation. Then $\lg N = 3$ stages of butterfly operations are performed, and the results $(y_0, y_1, \ldots, y_{N-1})$ emerge from the right. This figure is taken from [CLR90, p. 796].

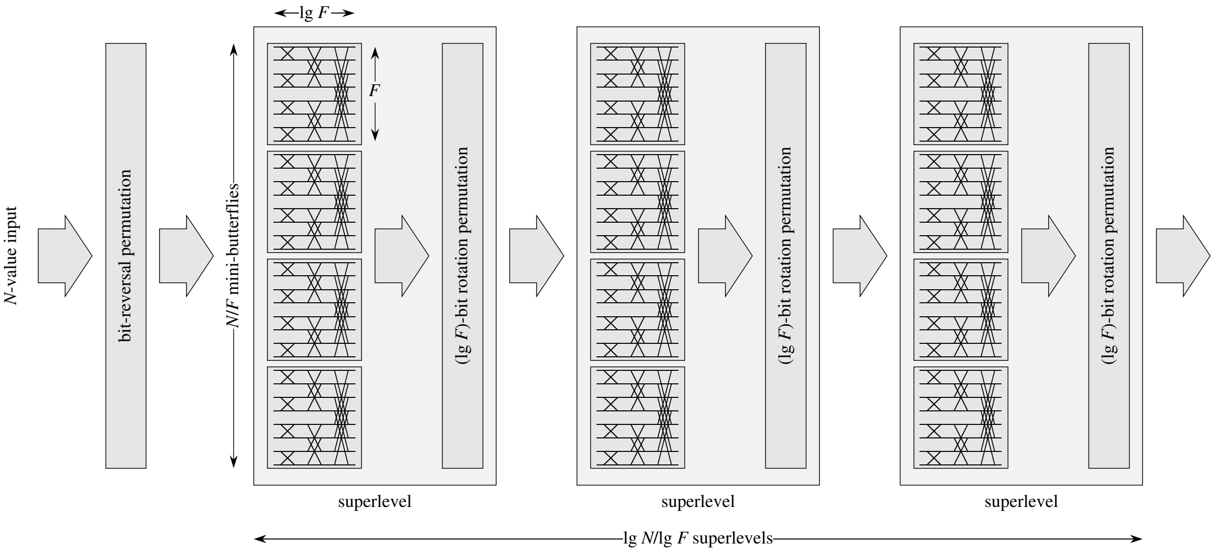


Figure 3: The structure of the out-of-core FFT algorithm for the PDM. After a bit-reversal permutation, we perform $\lg N / \lg F$ superlevels. Each superlevel consists of N/F mini-butterflies on F values, followed by a $(\lg F)$-bit right-rotation permutation on the entire array.

At the end of each superlevel is a $(\lg F)$-bit right-rotation permutation. If we interpret each index x as an n-bit vector $(x_{n-1}, x_{n-2}, \ldots, x_0)$, in a *$k$-bit right-rotation permutation*, the record in position x is moved to position $(x_{k-1}, \ldots, x_0, x_{n-1}, \ldots, x_k)$. That is, the bits of the index are rotated k positions to the right.

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

The total number of butterfly operations under this redrawing is the same as for a traditional butterfly graph with N points: $(N/2)\lg N$. Computationally, we have added the work of performing the bit-reversal and $(\lg F)$-bit right-rotation permutations. Both of these permutations belong to the class of BMMC (bit-matrix-multiply/complement) permutations. We employ the BMMC algorithm for the PDM given in [CSW94] to perform these permutations optimally under the PDM. In particular, we use the BMMC implementation described in [CH96], which is carefully optimized for communication and computational efficiency. The BMMC subroutine is the only part of the out-of-core FFT algorithm that requires independent I/O; it uses independent writes. All other I/O in the FFT algorithm is striped.

For a uniprocessor, we always choose the effective memory size to be M. The I/O complexity of this algorithm is then $\Theta\left(\frac{N}{BD}\frac{\lg \min(B,N/B)}{\lg(M/B)}\right)$ parallel I/Os, which is asymptotically optimal. See [CN96] for details.

Other implementation details

General values of N and F. If $\lg F$ does not divide $\lg N$, then there are $\lceil \lg N / \lg F \rceil$ superlevels and we compensate in the last one. Rather than computing mini-butterflies of depth $\lg F$ in the last superlevel, we compute mini-butterflies of depth $r = (\lg N) \bmod (\lg F)$, which is the number of levels of the full butterfly graph not yet computed. We can still read and write sets of F values, but now each such set in the last superlevel consists of $F/2^r$ mini-butterflies. The bit-rotation permutation in the last superlevel is by r bits rather than $\lg F$.

Twiddle factors. For simplicity, we omitted the twiddle factors in the description of the redrawing. They do have to be correct to compute the FFT, however. If we number the stages of butterfly operations from 1 to $\lg N$, then all twiddle factors of the sth stage are powers of ω_{2^s}. We obtain these powers of ω_{2^s} efficiently by directly computing the exponent of the twiddle factor in superlevel l, mini-butterfly q within the superlevel (starting from 0, and the range of q depends on the superlevel), and the jth butterfly within a group of butterflies as $\left\lfloor \frac{qF^{l+1}}{F^{\lceil \lg N/\lg F \rceil}} \right\rfloor + jF^l$. This computation is easy to move into loops and avoids expensive sine and cosine calls.

Synchronous and asynchronous I/O. We implemented the FFT algorithm with both synchronous (i.e., blocking) and asynchronous (non-blocking) I/O calls; the ViC* API 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 F is smaller with asynchronous I/O than with synchronous I/O. Because we carve memory into three parts and F must be a power of 2, asynchronous I/O reduces the effective memory size by a factor of 4. Nevertheless, we shall see in Section 6 that asynchronous I/O is beneficial.

4 Modifications for multiple processors

In this section, we describe the modifications to the uniprocessor out-of-core FFT algorithm that enable it to work on multiple processors. We start by looking at a straightforward way to extend the uniprocessor algorithm to use multiple processors. Because each mini-butterfly spans all P processors and the data layout is stripe by stripe, this method has nonuniform communication characteristics and is difficult to implement. By laying out the data differently (changing the band size) and keeping each mini-butterfly within a processor (changing the effective memory size), we derive an algorithm that has no interprocessor communication during the mini-butterfly computations.

The straightforward multiprocessor algorithm

Conceptually, the uniprocessor algorithm in Section 3 is already a multiprocessor algorithm if viewed in the right way. Suppose we choose the effective memory size F to be the memory size M over all P processors. (Assume for the moment that we use synchronous I/O so that reducing the effective memory size for additional buffers is not an issue.) Then computing each mini-butterfly is simply computing the butterfly graph on P processors, subject to the twiddle factors being altered as discussed in Section 3.

In reality, however, the multiprocessor algorithm is not quite so simple. Consider the data layout in Figure 1, and suppose that the effective memory size F is 2 stripes, or 32 records. Observe that the F/P records that map to a given processor in a mini-butterfly are not all consecutive. Processor $\mathcal{P}_0$, for example, holds records 0 to 3 and 16 to 19 in the first mini-butterfly. Figure 4 shows what happens when we compute the first mini-butterfly in this situation. Each butterfly operation involves two records whose indices differ in exactly one bit. There are three different communication characteristics, depending on which bit differs.

1. Each processor has BD/P consecutive records from a given stripe. In the first $\lg(BD/P)$ stages, therefore, computation is internal to each processor.

2. Each butterfly operation in the next $\lg P$ stages involves two records from the same stripe but in different processors. Thus, each such operation requires interprocessor communication.

3. In the last $\lg(F/BD)$ stages, each butterfly operation involves two records that are from different stripes but are in the memory of the same processor. These $\lg(F/BD)$ stages, therefore, use only internal computation.

Because there are three different communication characteristics that depend on stage numbers, we found this algorithm quite tricky to implement. Add in the twiddle factors, memory addressing (consider that in $\mathcal{P}_0$, records 0 to 3 and 16 to 19 are in consecutive memory locations), and changes for when $\lg F$ does not divide $\lg N$, and the code becomes rather long and difficult to get right. Even without the ViC* API calls for disk I/O, it is several pages long.

Band size

Part of the problem with the above approach is that the band size is small compared to the effective memory size. We define the *band size* β of a data layout on a parallel disk system as the maximum number of consecutive records per processor times the number of processors. The band size of the layout in Figure 1 is $\beta = BD/P \times P = BD$. In Figure 5(a), the band size is $\beta = F = 2BD$, and the band size in Figure 5(b) is $\beta = N$.

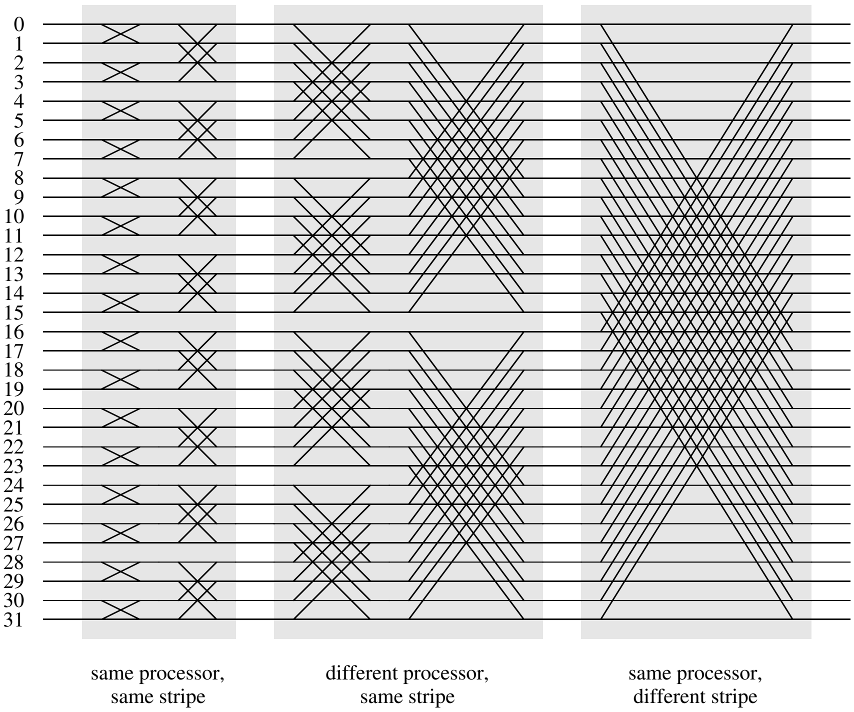


Figure 4: Computing a mini-butterfly when the band size is $\beta = BD$ and the effective memory size is $F > \beta/P$. Here, $B = 2$, $P = 4$, $D = 8$, and $F = 32 = 2BD$. Indices on the left are record numbers in the first mini-butterfly. Twiddle factors are omitted. The first $\lg(BD/P) = 2$ stages use computation internal to each processor, where each butterfly operation uses two values from the same stripe. The next $\lg P = 2$ stages require interprocessor communication to exchange values from the same stripe between processors. The last $\lg(F/BD) = 1$ stages use more computation internal to each processor, where each butterfly operation uses two values from different stripes.

(a) $\beta = F$

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

(b) $\beta = N$

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

Figure 5: Layouts with different band sizes in the same configuration as Figure 1. The effective memory size $F = 32$ is shown by double horizontal lines. **(a)** $\beta = F$. **(b)** $\beta = N$.

When the portion of a band that resides in one processor (β/P records) is smaller than the effective memory size F, computing a mini-butterfly must induce interprocessor communication. Allowing for the band size to be even smaller—$\beta < F$—then we have the situation above in which there are three different communication characteristics.

Varying the effective memory size and band size

In fact, we can vary the effective memory size and band size to simplify the mini-butterfly computations. So far, we have considered an effective memory size of $F = M$. Suppose instead that we use $F = M/P$, so that each mini-butterfly is the size of an individual processor's memory, and suppose further that we can change the data layout to have a band size $\beta \geq M$. Then we could compute each mini-butterfly without any communication at all, since each mini-butterfly would consist of M/P consecutive records within the same processor.

Changing the data layout turns out to be fairly simple. We change the bit-reversal and bit-rotation permutations performed in Figure 3 to other BMMC 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 $GF(2)$.[3] Treating each source index x as an n-bit vector, we perform matrix-vector multiplication over $GF(2)$ to produce an n-bit target index z: $z = H\,x$.[4] 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.

Before we can see how to change the BMMC permutations used in the FFT algorithm, we must first show that conversions between power-of-2 band sizes are BMMC permutations. In the same spirit as in Section 2, we can interpret any record's index as a sequence of three bit fields that give the record's location in the banded layout:

- The most significant $\lg(N/\beta)$ bits give the number of the band containing the record.
- The next $\lg P$ bits contain the number of the processor containing the record.
- The least significant $\lg(N/\beta P)$ bits give the relative location of the record in its processor and within the band.

Converting from one band size to another is actually a matter of "sliding" the $\lg P$ processor bits either left or right. In either direction, it is a bit permutation, and hence a BMMC permutation whose characteristic matrix is a permutation matrix (each row and each column holds exactly one 1).

Now we can see how to alter the BMMC permutations used in the FFT algorithm. The BMMC permutation subroutine assumes that the records are laid out on the disks with a band size of BD, but the reading and writing of mini-butterflies assumes a band size of some value β. Suppose that the $n \times n$ matrix T characterizes a $(\lg F)$-bit right rotation permutation with band size BD. Let the $n \times n$ matrix Π characterize the BMMC permutation that converts a band size of β to a band size of BD. Let Π^{-1} be the inverse of Π, so that Π^{-1} characterizes the BMMC permutation that converts a band size of BD to a band size of β. Then instead of just performing the permutation characterized by T, we first convert from band size β to band size BD, we then perform the

[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.

[4] 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.

permutation characterized by T, and finally we convert from band size BD back to band size β. In other words, we perform just one BMMC permutation, and it is characterized by the matrix product $(\Pi^{-1}T\Pi)$.

The above alteration works for all but the first and last BMMC permutations in the FFT algorithm. The first BMMC permutation differs in two ways: the records start out with band size BD rather than β, and it is a bit-reversal permutation. If the matrix R characterizes the bit-reversal permutation, then we perform the BMMC permutation characterized by the matrix product $(\Pi^{-1}R)$. The last BMMC permutation also differs in two ways: it may be a bit rotation by fewer than $\lg F$ bits, and the records end up with band size BD rather than β. If the matrix T' characterizes the bit-rotation permutation in the last superlevel, then we perform the BMMC permutation characterized by the matrix product $(T'\Pi)$.

Meaningful combinations of effective memory size and band size

In Section 6, we present performance results for two effective memory sizes (M and M/P) and three band sizes (BD, M, and N). Of these six combinations, only four make sense to implement. The two that do not are $F = M/P$, $\beta = BD$ and $F = M$, $\beta = N$. In both cases, each mini-butterfly would not contain F consecutive records.

Of the four meaningful combinations, the two with $F = M/P$ ($\beta = M$ and $\beta = N$) require no interprocessor communication during the mini-butterfly computations. It is remarkably simple to modify the uniprocessor FFT code to implement these configurations. When $F = \beta = M$, the interprocessor communication is much simpler than the $F = M$, $\beta = BD$ case detailed above.

5 Effects on I/O and communication complexity

This section examines how varying the effective memory size and band size affects the I/O and communication complexities of the full out-of-core multiprocessor FFT algorithm. It ends with a look at the pertinent complexity issues among the out-of-core multiprocessor methods.

Effect on I/O complexity

The full paper will show that the I/O complexity is $O\left(\frac{N}{BD}\frac{\lg N}{\lg F}\frac{\lg\min(B,N/B)}{\lg(M/B)}\right)$. Asymptotically, varying the band size has no effect on the I/O complexity.

Reducing the effective memory size F from M to M/P increases the asymptotic I/O complexity. The I/O complexity is off from the asymptotically optimal $\Theta\left(\frac{N}{BD}\frac{\lg\min(B,N/B)}{\lg(M/B)}\right)$ by a factor of $\lg N/\lg F$. When $F = M$, one can show that $O\left(\frac{N}{BD}\frac{\lg N}{\lg M}\frac{\lg\min(B,N/B)}{\lg(M/B)}\right) = O\left(\frac{N}{BD}\frac{\lg\min(B,N/B)}{\lg(M/B)}\right)$. When $F = M/P$, however, there may be additional superlevels, and they introduce additional I/O.

In practice, these additional superlevels occur rarely. Consider a configuration with 16 megabytes of memory per processor, which works out to $M/P = 2^{20}$ records per processor. Additional superlevels occur when there are many processors, so suppose that $P = 256$ and hence $M = 2^{28}$. The number of superlevels is $\lceil\lg N/\lg F\rceil$. When $F = M$, there are two superlevels for all N in the range 2^{29} to 2^{56}. When $F = M/P$, the range of N for which there are two superlevels is smaller—2^{21} to 2^{40}—but it still includes the largest problem size we are likely to see for some time to come.

Effect on communication complexity

Analyzing the change in communication complexity with varying memory size and band size is difficult. Of course, when $F = M/P$, there is no communication when computing the mini-butterflies. When $F = M$, we can determine the number of MPI messages and total volume of data communicated for a particular set of parameters; this calculation is complicated by differences in the last superlevel. Communication analysis becomes difficult when the band size changes. Because the characteristic matrices given to the BMMC subroutine change with the band size, the communication patterns within the BMMC subroutine change as well. We do not know of a purely analytical way to determine the exact nature of this change.

There are two non-analytical ways to determine the effect of band size on communication complexity. One is to instrument the FFT implementations to measure the number of MPI messages and communication volume; the results in Section 6 use this method. The other way does not require the FFT code to actually run. Because the entire FFT algorithm is both deterministic and oblivious (i.e., its control flow does not depend on the values of the N points), if we are given an exact set of parameters N, M, B, D, P, F, and β, then we can calculate the number of MPI messages and total communication volume.

The primary question

The primary question we ask is which effective memory size is better: M or M/P? Under certain conditions, using $F = M/P$ may cause there to be more superlevels. And there may be a tradeoff in communication during mini-butterflies versus communication during BMMC permutations. When $F = M/P$, we can avoid all interprocessor communication during mini-butterfly computations, but the modified characteristic matrices may cause additional interprocessor communication during the BMMC permutations. The performance results in the next section will help answer this question.

6 Performance of the multiprocessor methods

Here we present performance results for the out-of-core multiprocessor FFT methods described in Section 4. We shall see that when the number of superlevels is the same, the methods that avoid interprocessor communication during the mini-butterfly computations are faster. These methods are slightly slower when they have one more superlevel.

The platform is "Fleet," a set of eight IBM RS6000 workstations connected by a FDDI network. Each node runs AIX 4.1. Interprocessor communication is performed via the MPI calls `MPI_Sendrecv()` and `MPI_Sendrecv_replace()`. Parallel I/O calls are through the ViC* API [CH96], which in turn makes calls to the Galley File System [NK96a, NK96b]. Galley uses separate I/O processes (IOPs) to manage parallel I/O calls. The ViC* API treats each IOP like a disk. On Fleet, it is fastest to run the IOPs on separate nodes from the computational processes. Consequently, we report results for $P = 4$ and $D = 4$. All runs were for $N = 2^{25}$ points (or 2^{29} bytes), which is the largest data set possible with this configuration of Galley on Fleet. Because the software interface to Fleet is very similar to an IBM SP-2, the full paper will include performance numbers for the SP-2.

We report on timing runs with the following variations:

1. Effective memory sizes were M and M/P, and band sizes were BD, M, and N. For this extended abstract, we ran three combinations: $F = M$ and $\beta = BD$; $F = M/P$ and $\beta = M$; $F = M/P$ and $\beta = N$. The full paper will also include $F = M$ and $\beta = M$.

2. I/O to read and write mini-butterflies was both synchronous and asynchronous. The BMMC subroutine comes from an established library and uses only asynchronous I/O.

3. In one configuration, each processor used 2^{24} bytes of memory, so that there were two superlevels for both effective memory sizes. In another configuration, each processor used less memory: 2^{16} bytes for synchronous I/O and 2^{18} bytes for asynchronous I/O. These are the largest memory sizes for which an effective memory size of M/P has three superlevels but for $F = M$ there are only two. The best possible block size for Galley was used in each run.

Figure 6 shows the results with synchronous I/O and the larger memory size of 2^{24} bytes per processor. The two methods with effective memory size $F = M/P$ were virtually identical. The total time is dominated by the BMMC subroutine for all methods. When $F = M$, the BMMC subroutine takes slightly less time than for the other two methods. The communication cost in computing the mini-butterflies soaks up these savings and more. Mini-butterfly computation time is slightly longer, too, probably due to context-switching and cache effects. Overall, the methods with $F = M/P$ take approximately 86% of the time of the method with $F = M$.

The tables included in Figures 6–9 show the number of messages and message volume, rounded to the nearest million bytes, per processor for the mini-butterfly and BMMC portions of the computation. In Figure 6, when $F = M$, each processor sends 64 messages for a total of about 268 million bytes during the mini-butterfly computations. The message count and volume during the BMMC portion is the same for all three methods. There is no tradeoff in communication: there is less communication when $F = M/P$.

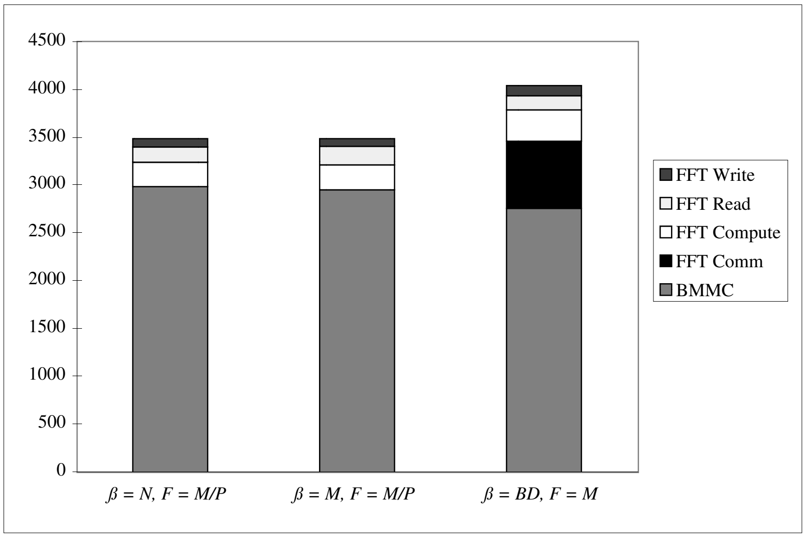
Figure 7 shows results with asynchronous I/O. The I/O times represent time spent waiting for previously issued I/O to complete. Total time is reduced from the synchronous I/O runs by only 1.5–2%. The methods with $F = M/P$ have the same relative advantage over the $F = M$ method as before.

Figures 8 and 9 show results for the smaller memory sizes. These memory sizes are so small that the BMMC subroutine, which is sensitive to the memory size, runs quite slowly. In these runs, the methods with $F = M/P$ take three superlevels rather than the two taken when $F = M$. Consequently, they take longer. The methods with $\beta = N, F = M/P$ and $\beta = M, F = M/P$ take approximately 4.5% and 19.0% longer, respectively, than the $\beta = BD, F = M$ method with synchronous I/O. With asynchronous I/O, these $F = M/P$ methods take about 0.2% and 15.6% longer. When we use asynchronous I/O, therefore, the $\beta = N, F = M/P$ method is usually the fastest, and when it loses, it is not by much. Asynchronous I/O improves all three methods considerably. It is interesting to note that the mini-butterfly communication time when $\beta = BD, F = M$ almost soaks up the benefit of having one fewer superlevel when compared to $\beta = N, F = M/P$. We also see that a band size of M causes more messages during the BMMC subroutine than a band size of N. The BMMC subroutine, and hence the entire program, runs more slowly in this case.

7 Conclusion

We have examined four ways to perform out-of-core multiprocessor FFTs with distributed memory using the Parallel Disk Model. Overall, the best ways avoid interprocessor communication during the in-core mini-butterfly computations. Asynchronous I/O improves performance, sometimes marginally and sometimes significantly.

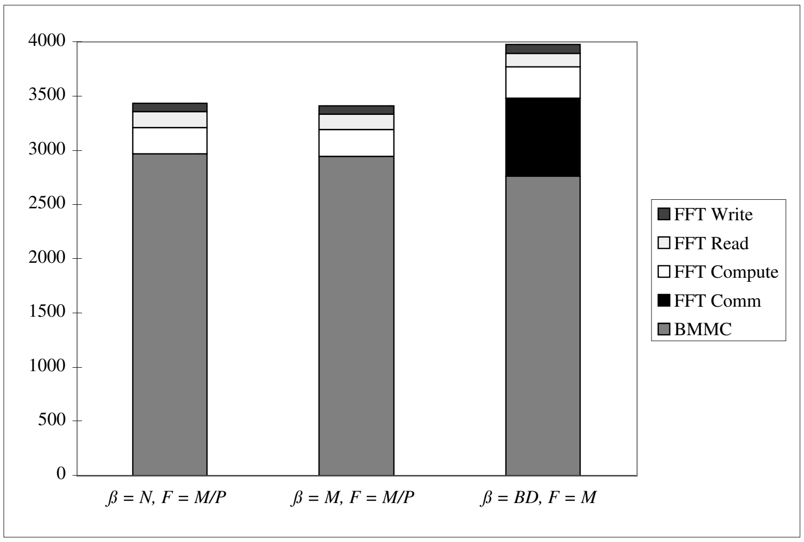
As we noted in Section 6, the advantage of the best ways is far from overwhelming: they save approximately 14% of the total time. In all the methods considered, most of the time is spent in the BMMC subroutine. Mini-butterfly interprocessor communication accounts for a relatively small share of the time. When we run these methods on the IBM SP-2 with its faster communication



Method	FFT messages	FFT volume
$\beta = N$, $F = M/P$	0	0
$\beta = M$, $F = M/P$	0	0
$\beta = BD$, $F = M$	64	268

Method	BMMC messages	BMMC volume
$\beta = N$, $F = M/P$	240	503
$\beta = M$, $F = M/P$	240	503
$\beta = BD$, $F = M$	240	503

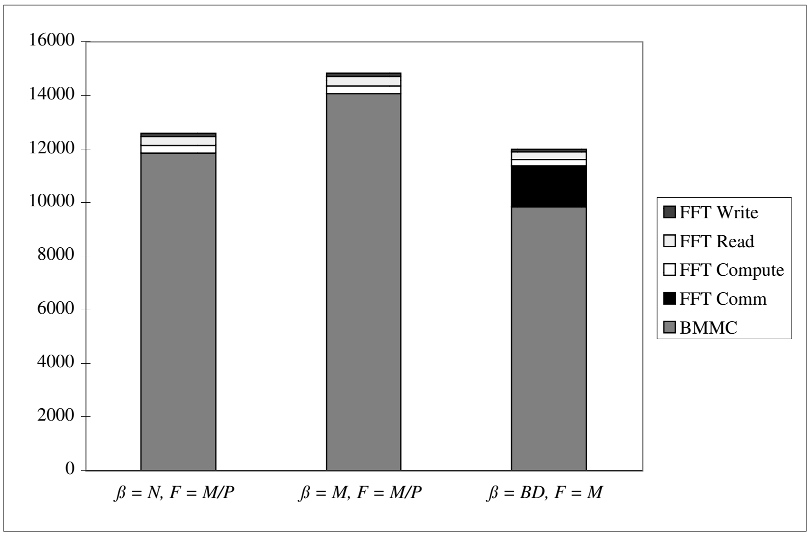
Figure 6: Results for the three methods with synchronous I/O and 2^{24} bytes of memory per processor. The vertical axis is time, in seconds. From bottom to top, each stacked bar shows time spent in the BMMC subroutine, communication time during the mini-butterfly computation, computation time in mini-butterflies, time to read mini-butterflies, and time to write mini-butterflies. Message measurements are per processor, and all message volumes are to the nearest million bytes.



Method	FFT messages	FFT volume
$\beta = N$, $F = M/P$	0	0
$\beta = M$, $F = M/P$	0	0
$\beta = BD$, $F = M$	256	268

Method	BMMC messages	BMMC volume
$\beta = N$, $F = M/P$	216	503
$\beta = M$, $F = M/P$	240	503
$\beta = BD$, $F = M$	240	503

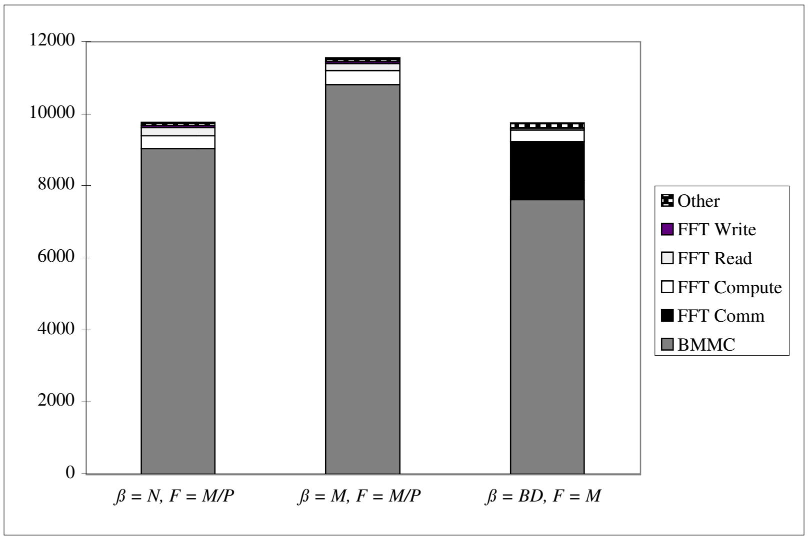
Figure 7: Results for the three methods with asynchronous I/O and 2^{24} bytes of memory per processor. Read and write times are times spend waiting for previously started I/O to complete.



Method	FFT messages	FFT volume
$\beta = N$, $F = M/P$	0	0
$\beta = M$, $F = M/P$	0	0
$\beta = BD$, $F = M$	49152	537

Method	BMMC messages	BMMC volume
$\beta = N$, $F = M/P$	141312	1409
$\beta = M$, $F = M/P$	157696	1376
$\beta = BD$, $F = M$	122880	1208

Figure 8: Results for the three methods with synchronous I/O and 2^{16} bytes of memory per processor.



Method	FFT messages	FFT volume
$\beta = N$, $F = M/P$	0	0
$\beta = M$, $F = M/P$	0	0
$\beta = BD$, $F = M$	49152	537

Method	BMMC messages	BMMC volume
$\beta = N$, $F = M/P$	29952	1107
$\beta = M$, $F = M/P$	31744	1074
$\beta = BD$, $F = M$	23040	906

Figure 9: Results for the three methods with asynchronous I/O and 2^{18} bytes of memory per processor.

network, we may very well find that the running times of the methods are even closer. Our future work will focus on improving the BMMC subroutine, which is the bottleneck in the FFT algorithm.

We alluded in Section 4 to one advantage of the methods with effective memory size M/P: ease of developing code. Starting from a working out-of-core uniprocessor FFT program, it took *under an hour* of programming time to convert it to a multiprocessor program with band size $\beta = N$ and effective memory size $F = M/P$. And it worked the first time. In contrast, starting from the same point, it took *several weeks* to develop and debug the method with $\beta = BD$ and $F = M$. Changing the band size is easy. Adding interprocessor communication when the band size is smaller than the effective memory size is hard.

Acknowledgments

Thanks to David Kotz and Sanjay Khanna for their help using Fleet. Melissa Hirschl wrote the ViC* wrappers.

References

[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.

[Cal96] C. Calvin. Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication. *Parallel Computing*, 22:1255–1279, 1996.

[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.

[CH96] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. Technical Report PCS-TR96-293, Dartmouth College Department of Computer Science, August 1996. To appear in *Parallel Computing*.

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

[CN96] Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. Technical Report PCS-TR96-294, Dartmouth College Department of Computer Science, August 1996. To appear in *Parallel Computing*.

[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*.

[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.

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

[JJK92] S. Lennart Johnsson, Michel Jacquemin, and Robert L. Krawitz. Communication efficient multi-processor FFT. *Journal of Computational Physics*, 102:381–397, 1992.

[NK96a] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[NK96b] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, Philadelphia, May 1996. ACM Press.

[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.

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

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

[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.

[Swa87] Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

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

[Zhu90] J. P. Zhu. An efficient FFT algorithm on multiprocessors with distributed memory. In *Proceedings of the Fifth Distributed Memory Computing Conference*, volume I, pages 358–363, April 1990.