Dartmouth College Computer Science
Technical Report PCS-TR97-322

Determining an Out-of-Core FFT Decomposition Strategy for Parallel Disks by Dynamic Programming

Thomas H. Cormen*
Dartmouth College
Department of Computer Science
thc@cs.dartmouth.edu

July 1997

Abstract

We present an out-of-core FFT algorithm based on the in-core FFT method developed by Swarztrauber. Our algorithm uses a recursive divide-and-conquer strategy, and each stage in the recursion presents several possibilities for how to split the problem into subproblems. We give a recurrence for the algorithm's I/O complexity on the Parallel Disk Model and show how to use dynamic programming to determine optimal splits at each recursive stage. The algorithm to determine the optimal splits takes only $\Theta(\lg^2 N)$ time for an N-point FFT, and it is practical. The out-of-core FFT algorithm itself takes considerably longer.

1 Introduction

Although in most cases, Fast Fourier Transforms (FFTs) can be computed entirely in the main memory of a computer, in a few exceptional cases, the input vector is too large to fit. One must use out-of-core FFT methods in such cases.

In out-of-core methods, data are stored on disk and repeatedly brought into memory a section at a time, operated on there, and written out to disk. Because disk accesses are so much slower than main memory accesses (typically at least 10,000 times slower), efficient out-of-core methods focus on reducing disk I/O costs. We can reduce disk I/O costs in two ways: reduce the cost of each access, and reduce the number of accesses.

*Supported in part by the National Science Foundation under grant CCR-9625894. Portions of this work were performed while the author was visiting the Institute for Mathematics and Its Applications at the University of Minnesota.

To appear in *Proceedings of the Workshop on Algorithms for Parallel Machines*, 1996-97 Special Year on Mathematics of High Performance Computing, Institute for Mathematics and Its Applications, University of Minnesota, Minneapolis, September 1996.

We can reduce the per-access cost by using parallel disk systems. That is, we take advantage of the increase in I/O bandwidth provided by using multiple disks. If we use D disks instead of one disk, the I/O bandwidth may increase by up to a factor of D. Parallel disk systems are available on most parallel computers, and they are relatively simple to construct on networks of workstations.

In this paper, we shall concentrate on reducing the number of parallel disk accesses for performing out-of-core FFTs. We use the Parallel Disk Model (PDM) of Vitter and Shriver [VS94] to compute I/O costs.

Of the many known variants of FFT methods (see Van Loan's excellent book [Van92] for a comprehensive treatment), this paper is based on a lesser-known method, which we shall refer to as Swarztrauber's method.[1] Unlike the traditional Cooley-Tukey formulation [CT65], which uses a 2-way divide-and-conquer strategy, Swarztrauber's method performs a $\sqrt{N}$-way divide-and-conquer for an input vector of length N.

In fact, assuming that N is a power of 2, both the Cooley-Tukey and Swarztrauber methods are specific points in a design strategy of using an N/R-way divide-and-conquer strategy, where R is also a power of 2. Here, R is the size of each subproblem ($N/2$ for Cooley-Tukey and $\sqrt{N}$ for Swarztrauber). Like the recursive form of the Cooley-Tukey method (see [CLR90, Chapter 32]), each problem of size R may be solved recursively.

The question we examine in this paper is what value of R to use in the recursion. That is, which subproblem size yields the fewest number of parallel disk accesses over the course of the FFT computation? We shall see that although this value is not a fixed portion of the problem size N, we can compute it for all stages of the recursion via dynamic programming in only $\Theta(\lg^2 N)$ time. For even out-of-core problems, $\lg N$ is reasonably small ($\lg N = 50$ for a 1-petapoint FFT). Computing the optimal subproblem sizes to use in the recursion is a small in-core problem that runs quickly. On a 175-MHz DEC Alpha workstation, for example, it takes under 25 milliseconds to compute them *and* print them out on the screen. This cost of computing optimal sizes is negligible compared to the hours or days it would take to actually compute huge FFTs.

The remainder of this paper is organized as follows. Section 2 gives fundamental background information on the FFT, focusing on the in-core version of Swarztrauber's method. Section 3 presents the Parallel Disk Model, which provides the cost metric for our out-of-core algorithm, and it also gives I/O costs for relevant algorithms in the PDM. Section 4 describes the modifications we make to the in-core version of Swarztrauber's method to make it work in an out-of-core setting on the PDM, and it also analyzes the I/O cost of the modified algorithm for a given subproblem size. Section 5 shows how to use dynamic programming to compute optimal subproblem sizes. Finally, we conclude in Section 6.

For other work in out-of-core FFTs, see [Bai90, Bre69, CN96, CWN97, SW95].

2 FFT background

This section presents fundamental background information on the FFT in general and the in-core version of Swarztrauber's method in particular. For further background on the FFT, see any of the texts [CLR90, Nus82, Van92].

[1] This method is attributed by Bailey [Bai90] to P. Swarztrauber as a variation of an algorithm by Gentleman and Sande. It is also attributed by Brenner [Bre69] to E. Granger.

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 (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 the 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 $a^{\text{even}} = (a_0, a_2, \ldots, a_{N-2})$ and $a^{\text{odd}} = (a_1, a_3, \ldots, a_{N-1})$.
2. Recursively compute $y^{\text{even}} = \mathrm{DFT}_{N/2}(a^{\text{even}})$ and $y^{\text{odd}} = \mathrm{DFT}_{N/2}(a^{\text{odd}})$.
3. For $k = 0, 1, \ldots, N/2 - 1$, compute $y_k = y_k^{\text{even}} + \omega_N^k y_k^{\text{odd}}$ and $y_{k+N/2} = y_k^{\text{even}} - \omega_N^k y_k^{\text{odd}}$.

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 $k_{\lg N-1}, k_{\lg N-2}, \ldots, k_0$ maps to the element whose index in binary is $k_0, k_1, \ldots, k_{\lg N-1}$. 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.

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, including Swarztrauber's, have been developed to improve performance on vector machines and in memory hierarchies, by avoiding the bit-reversal permutation to improve locality of reference.

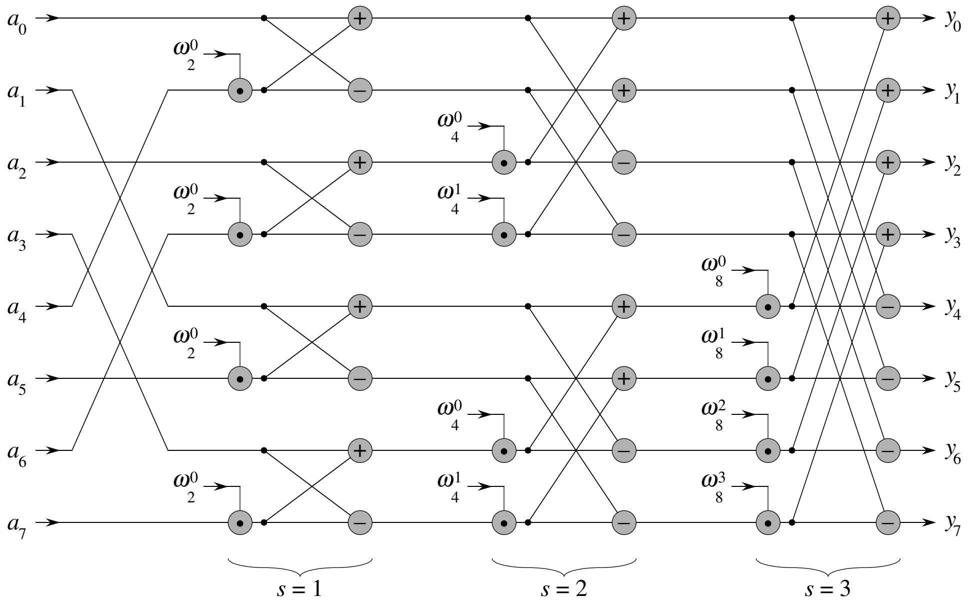

Figure 1: The FFT computation after fully unrolling the recursion, shown here with $N = 8$. Inputs $(a_0, a_1, \ldots, a_{N-1})$ enter from the left and first undergo a bit-reversal permutation. Then $\lg N = 3$ stages of butterfly operations are performed, and the results $(y_0, y_1, \ldots, y_{N-1})$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

Swarztrauber's method

Swarztrauber's method generalizes the above divide-and-conquer method by splitting the summation of equation (1) into $\sqrt{N}$ summations each with $\sqrt{N}$ terms. In this description, however, we shall generalize the method even further to split the summation into N/R summations of R terms each, where R may be any power of 2 such that $2 \leq R \leq N/2$. The DFT in each subproblem is comprised of all terms whose indices are congruent modulo N/R. The analog of a butterfly operation adds N/R terms—also expressible as DFTs—that are computed by recursive calls to subproblems of size N/R. The generalized method is given by the following steps:

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

Step 2. Compute the DFT of each R-element row individually.

Step 3. Scale the resulting matrix by multiplying the entry in row l and column j by ω_N^{lj}.

Step 4. Transpose the matrix back into an $R \times N/R$ shape.

Step 5. Compute the DFT of each N/R-element row individually.

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

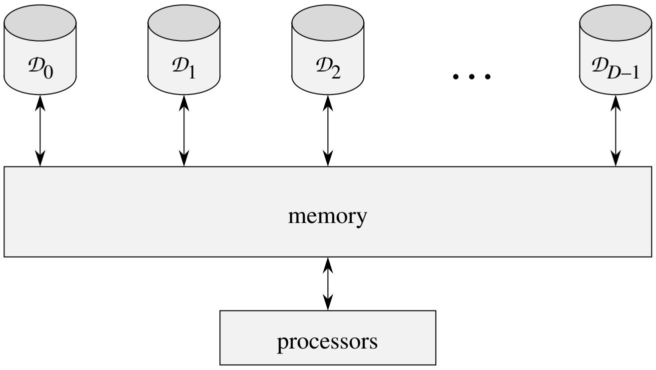

Figure 2: The Parallel Disk Model. Records are stored on disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$, with an equal number of records on each disk. The records on each disk are partitioned into blocks of B records each (not shown here). Disk I/O transfers records between disks and memory that can hold M records. Processor and memory organization are unspecified. An algorithm's cost is the number of parallel I/O operations, each of which transfers one block per disk.

On a sequential machine, the running time of this method is expressed by the recurrence

$$T(N) = \frac{N}{R} T(R) + R\,T(N/R) + \Theta(N) ,$$

which is easily shown to have the solution $T(N) = \Theta(N \lg N)$.

In practice, once the subproblem size becomes small enough, we use a more straightforward method, such as Cooley-Tukey, to solve the subproblems. If we choose $R = \sqrt{N}$ (as was done in the original formulation of the method), then all subproblems in steps 2 and 5 are of size $\sqrt{N}$. On most machines, these subproblems fit in cache, and so Cooley-Tukey is a good way to solve them. Of course, in order to choose $R = \sqrt{N}$, we must have that N is a power of 4.

3 The Parallel Disk Model

This section describes the Parallel Disk Model (PDM) [VS94], upon which our modifications to Swarztrauber's method for out-of-core FFTs are based. It also covers prior PDM algorithms relevant to performing out-of-core FFTs.

PDM structure and cost metric

Figure 2 shows the *Parallel Disk Model*, in which 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 *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.

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. We also require that $BD \leq M$ in order to fully utilize disk bandwidth, and of course we assume that $M < N$ so that the problem is out-of-core.

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.

We note two prior results for the PDM. In the original PDM paper, Vitter and Shriver showed that the I/O complexity of pebbling a butterfly graph is $\Theta\left(\frac{N}{BD}\frac{\lg\min(B,N/B)}{\lg(M/B)}\right)$, which appears to be the analogue of the $\Theta(N \lg N)$ bound seen for so many sequential algorithms on the standard RAM model. The FFT algorithms for the PDM in [CN96, CWN97] achieve this bound.

Low-order transpose

We will rely heavily on the second PDM result, which is based on a class of permutations known as BMMC (bit-matrix-multiply/complement) permutations [CSW94]. The class of BMMC permutations includes many permutations encountered in practice; among them are matrix transpose when both matrix dimensions are powers of 2. Here, we will use a slight variation on matrix transpose.

First, let us examine the usual 2-dimensional matrix-transpose permutation. Suppose we have an N-element matrix, where N is a power of 2, and let the matrix be stored in row-major order. Suppose the matrix is $R \times C$, where $RC = N$, so that both R and C are powers of 2. Each element has a $\lg N$-bit index $x = x_{\lg N-1}, x_{\lg N-2}, \ldots, x_0$, and we can partition this index into the $\lg R$-bit row part $x_{\lg N-1}, x_{\lg N-2}, \ldots, x_{\lg C}$ and the $\lg C$-bit column part $x_{\lg C-1}, x_{\lg C-2}, \ldots, x_0$. Because the matrix transpose permutation simply interchanges the row and column numbers of each element, the original column number becomes the new row number and the original row number becomes the new column number. That is, the mapping from the *source index* to the *target index* for each record interchanges the high-order $\lg R$ and the low-order $\lg C$ bits to produce the target index $x_{\lg C-1}, x_{\lg C-2}, \ldots, x_0, x_{\lg N-1}, x_{\lg N-2}, \ldots, x_{\lg C}$. Using techniques from [CSW94], one can show that transposition of an $R \times C$ matrix can be performed in $\frac{2N}{BD}\left(\left\lceil\frac{\lg\min(R,C,M,N/M)}{\lg(M/B)}\right\rceil + 1\right)$ parallel I/Os.

The variation we use to perform an out-of-core FFT is a *low-order transpose*, and it works as follows. Let K be any integer such that $1 \leq K \leq N$ and K is a power of 2, and let R and C be powers of 2 such that $RC = K$. Intuitively, we consider the data as a sequence of N/K matrices, each of which is $R \times C$. We wish to transpose the individual matrices, leaving them in the same relative order. To do so, we partition a source index x into three parts: the $\lg K$ high-order bits $x_{\lg N-1}, x_{\lg N-2}, \ldots, x_{\lg N/K}$, the next $\lg R$ bits $x_{\lg N/K-1}, x_{\lg N/K-2}, \ldots, x_{\lg C}$, and the $\lg C$ low-order bits $x_{\lg C-1}, x_{\lg C-2}, \ldots, x_0$. In the low-order transpose permutation, the corresponding target index has the same $\lg K$ high-order bits, but the next two parts are interchanged. That is, the target index is $x_{\lg N-1}, x_{\lg N-2}, \ldots, x_{\lg N/K}, x_{\lg C-1}, x_{\lg C-2}, \ldots, x_0, x_{\lg N/K-1}, x_{\lg N/K-2}, \ldots, x_{\lg C}$. Again using techniques from [CSW94], one can show that a low-order transpose can be performed in

$\frac{2N}{BD}\left(\left\lceil\frac{\lg\min(R,C,M,K/M)}{\lg(M/B)}\right\rceil+1\right)$ parallel I/Os. Observe that when $K = N$, low-order transpose is the usual 2-dimensional matrix transpose, and the I/O complexities are equal.

In later sections, we shall use the procedure call Low-Order-Transpose(N, K, R, C) to indicate the low-order transpose operation.[2]

4 Out-of-core Swarztrauber's method

This section describes to how modify the in-core version of Swarztrauber's method for out-of-core operation on the PDM. The algorithm is recursive.

At any point in the recursion, we have a problem size of K, where K is a power of 2 such that $2 \le K \le N$. The value of K will vary during the course of the algorithm, and at any given time, there are N/K distinct problems. Initially, $K = N$ so that there is just one problem of size N. The problems are stored in sequence on the parallel disk system so that each set of K consecutive points comprises a problem.

We express the out-of-core FFT algorithm by the following recursive procedure which takes the problem size K as a parameter; K will always be a power of 2. Because the other parameters (e.g., N and M) do not vary over the course of the algorithm, we do not treat them as parameters. We express the procedure in pseudocode in which indenting indicates control flow and comments start with the character $\triangleright$.

Out-of-Core-FFT(K)

1 **if** $K \le M$

2   **then** $\triangleright$ Base case: problem fits in memory

3   **for** $j \leftarrow 0$ **to** $N/M - 1$

4   **do** read the jth memoryload from the disks into memory

5   **for** $l \leftarrow 0$ **to** $M/K - 1$

6   **do** perform a K-point FFT on points lK to $(l+1)K - 1$ in memory, using any in-core FFT method

7   update the jth memoryload on the disks by writing out the result of the in-core FFT

8   **else** $\triangleright$ Recursive case: problem is larger than memory

9   choose integers R and C such that R and C are powers of 2 and $RC = K$ (we will see in Section 5 how to choose R and C)

10   Low-Order-Transpose(N, K, R, C)

11   Out-of-Core-FFT(R)

12   scale each entry (see below)

13   Low-Order-Transpose(N, K, C, R)

14   Out-of-Core-FFT(C)

15   Low-Order-Transpose(N, K, R, C)

This procedure works as follows. Lines 2–7 handle the base case when the recursion bottoms out; this occurs when $K \le M$ so that each problem fits in memory. In the base case, we perform N/K consecutive FFTs, each of size K. The loop of lines 3–7 reads in each memoryload, performs

[2]Since $RC = K$, only two of the parameters K, R, and C are absolutely necessary. We include all three for clarity.

FFTs on the data in the memoryload, and writes it out. The loop of lines 5–6 performs the FFTs on the data of a given memoryload; there are M/K such FFTs to perform.

The recursive case of lines 8–15 follows the in-core description of Swarztrauber's method from Section 2. Lines 9–10 emulate step 1; each problem of size K is treated as an $R \times C$ matrix and transposed. (Section 5 will address the question of how to pick R and C optimally.) Problems remain in the same order relative to each other, and so the low-order transpose operation is appropriate here. Line 11 emulates step 2, which computes the DFT of each R-element row. We defer discussion of line 12, which emulates step 3, for the moment. Line 13 is the inverse of line 10, just as step 4 is the inverse of step 1. Line 14 emulates step 5, which computes the DFT of each C-element row. Finally, line 15 is the same as line 10, just as step 6 repeats step 1.

The scaling of line 12 works as follows. After the low-order transpose of line 10 and the recursive calls of line 11, each point is in some row l and column j, where $0 \leq l \leq C-1$ and $0 \leq j \leq R-1$. (For any given values of l and j, there are N/K points, one per size-K problem.) We scale each point by multiplying it by ω_K^{lj}; note that the root of unity here is ω_K and not ω_N. At first glance, this operation would seem to require an additional pass through all N records. We can avoid this additional pass, however. Because the scaling immediately follows a recursive call, and that recursive call will eventually bottom out with a single pass through all N records (lines 3–7), we can fold the scaling operation into the base case. We omit the programming details here, although we note that we would add parameters to the procedure Out-of-Core-FFT to indicate whether scaling should occur (since the recursive call of line 14 should not invoke scaling) and values of R, C, and K in the caller.

Analysis

To analyze the I/O complexity of this algorithm in the PDM, we start with the easy base case. Here, we make one pass through the data, and so there are $2N/BD$ parallel I/Os: N/BD to read each record, and N/BD to write each record.

For the recursive case, the I/O costs come from the three low-order transposes and the two recursive calls. For given values of K, R, and C, each low-order transpose operation requires $\frac{2N}{BD}\left(\left\lceil\frac{\lg\min(R,C,M,K/M)}{\lg(M/B)}\right\rceil+1\right)$ parallel I/Os.

If we let $T(K)$ denote the number of parallel I/Os for an out-of-core FFT problem of size K, we have the following recurrence:

$$T(K) = \begin{cases} \dfrac{2N}{BD} & \text{if } K \leq M \ , \\ \displaystyle\min_{\substack{2 \leq R \leq K/2 \\ R \text{ a power of } 2 \\ C = K/R}} \left\{3 \cdot \frac{2N}{BD}\left(\left\lceil\frac{\lg\min(R,C,M,K/M)}{\lg(M/B)}\right\rceil+1\right) + T(R) + T(C)\right\} & \text{if } K > M \ . \end{cases} \tag{2}$$

The next section shows how to determine the optimal values to use for R and C at each point in the recursion.

5 Determining optimal subproblem sizes by dynamic programming

Given the recurrence (2), how do we select the optimal values to use for R and C at a given point in the recursion? This section shows how to use dynamic programming to do so. We refer the reader to Chapter 16 of [CLR90] for background on the technique of dynamic programming.

Recall that there are two properties that an optimization problem must have for dynamic programming to apply:

Optimal substructure: an optimal solution to the problem contains within it optimal solutions to the subproblems. Recurrence (2) exhibits optimal substructure, which we see as follows. Suppose that an optimal solution for a problem of size K used a non-optimal solution for one of the subproblems. Then we could replace the non-optimal subproblem solution by an optimal solution and lower the cost of the original size-K problem, which contradicts our assumption that we had an optimal solution for the size-K problem.

Overlapping subproblems: a recursive algorithm revisits the same problem over and over again. Our FFT algorithm has overlapping subproblems, since each problem of a given size is considered for all larger problems.

A dynamic-programming solution to determining the subproblem sizes takes as input the problem size N, memory size M, block size B, and number of disks D, and it produces two arrays as output. Because problem sizes are powers of 2, which is a sparse index set, we index into the arrays by the base-2 logarithm of the problem size. The arrays are the following:

- $cost[1 \,..\, \lg N]$ is defined such that $cost[k]$ is the minimum I/O cost for a subproblem of size $K = 2^k$.
- $split[1 \,..\, \lg N]$ is defined such that $split[k]$ contains an optimal value of R to use for a subproblem of size $K = 2^k$.

In the following pseudocode for the dynamic-programming algorithm, indices are base-2 logarithms of the variables we have been using so far: $k = \lg K$, $r = \lg R$, and $c = \lg C$.

```
Determine-Optimal-Sizes(N, M, B, D)
for k ← 1 to lg N
    do if 2^k ≤ M
          then cost[k] ← 2N/BD
               split[k] ← 0        ▷ no split: base of recursion
          else cost[k] ← ∞
               for r ← 1 to k − 1
                   do c ← k − r
                      this-cost ← 3 · (2N/BD) (⌈min(r, c, lg M, k − lg M) / lg(M/B)⌉ + 1) + cost[r] + cost[c]
                      if this-cost < cost[k]
                        then cost[k] ← this-cost
                             split[k] ← r
```

Simple inspection of the code shows that the running time of this procedure depends on the inner loop of lines 6–11. This loop runs $\Theta(\lg^2 N)$ times, and so the running time is $\Theta(\lg^2 N)$. We point out that $\Theta(\lg^2 N)$ is merely the time to determine the optimal I/O cost and subproblem sizes. The time to actually perform the out-of-core FFT is far, far greater!

Results

Although one might expect that subproblem sizes should be as close to $\sqrt{K}$ as possible, that does not always turn out to be the case. Figure 3 is a graphic depiction of a run of Determine-Optimal-Sizes with $N = 2^{50}$, $M = 2^{20}$, $B = 2^{12}$, and $D = 2^4$. Each row shows an out-of-core problem size, ranging from 2^{21} to 2^{50}. Column indices indicate possible split positions from 2^1 to 2^{49}. Vertical and diagonal bars delimit the possible ranges of split positions. A `+` indicates a split position that is optimal for the problem size. For example, the `+` in row 23, column 15 indicates that $R = 2^{15}$ is one of the optimal split positions for $K = 2^{23}$. Spaces indicate non-optimal split positions. We show the "halfway points" (values equal to $\sqrt{K}$ if $\lg K$ is even, and values equal to $\sqrt{K/2}$ and $\sqrt{2K}$ if $\lg K$ is odd) specially. A `#` indicates a halfway point that is optimal, and an `O` indicates one that is non-optimal. For $\lg K \leq 40$, all halfway points are optimal split positions. For $\lg K > 40$, however, no halfway point is optimal.

The picture can become even more complicated. Figure 3 shows the optimal split positions determined by a run of Determine-Optimal-Sizes with $N = 2^{60}$, $M = 2^{15}$, $B = 2^9$, and $D = 2^4$.

It is usually the case, however, that $N \leq M^2$. (For example, in a 1-terapoint problem with $N = 2^{40}$, the memory size would have to be under $M = 2^{20}$ points, or under 16 megabytes, in order for N to exceed M^2. Any contemporary workstation, or personal computer for that matter, provides at least that much memory.) Every case we have run with $N \leq M^2$ looks like the top $\lg M$ lines of Figures 3 and 4 (i.e., the top 20 lines of Figure 3 and the top 15 lines of Figure 4). In particular, for a problem of size K, any split position from K/M to M is optimal.

6 Conclusion

We have seen how to use dynamic programming to select subproblem sizes in a recursive out-of-core FFT algorithm. The dynamic-programming algorithm is fast, requiring only $\Theta(\lg^2 N)$ time to determine the decomposition for an N-point FFT.

Unlike previous work in out-of-core FFTs [CN96, CWN97, VS94], we have not presented the asymptotic parallel I/O complexity of the resulting FFT algorithm. Why not? As Figures 3 and 4 show, the solution provided by the dynamic-programming method appears to defy a straightforward analysis. Consequently, we do not know how to determine the asymptotic I/O complexity when $N > M^2$.

Another question not answered here is how well this method works in practice. Our earlier work [CN96, CWN97] presents empirical results for a different (and provably asymptotically optimal) out-of-core FFT method on a uniprocessor and on a multiprocessor. How would the method of the present paper compare? Although it has yet to be implemented, we suspect that it would be slower, due to the expense of three low-order transpose operations per recursive stage.

Practical considerations aside, we view this work as interesting for two reasons. First, it extends Swarztrauber's method to out-of-core FFTs in as straightforward a manner as can be expected, given the existing machinery for performing low-order transpose permutations. Second, it uses

```
             1111111111222222222233333333334444444444
    1234567890123456789012345678901234567890123456789

21 |+++++++++##+++++++++\
22 | +++++++++#+++++++++ \
23 |  ++++++++##++++++++  \
24 |   ++++++++#++++++++   \
25 |    +++++++##+++++++    \
26 |     +++++++#+++++++     \
27 |      ++++++##++++++      \
28 |       ++++++#++++++       \
29 |        +++++##+++++        \
30 |         +++++#+++++         \
31 |          ++++##++++          \
32 |           ++++#++++           \
33 |            +++##+++            \
34 |             +++#+++             \
35 |              ++##++              \
36 |               ++#++               \
37 |                +##+                \
38 |                 +#+                 \
39 |                  ##                  \
40 |                   #                   \
41 |    ++++    ++++   00   ++++    ++++    \
42 |     +++     +++    0    +++     +++     \
43 |      ++      ++    00    ++      ++      \
44 |       +       +     0     +       +       \
45 |    ++++++++++++++++ 00 ++++++++++++++++    \
46 |     +++ +++++++ +++  0  +++ +++++++ +++     \
47 |      ++  ++++++  ++  00  ++  ++++++  ++      \
48 |       +   +++++   +   0   +   +++++   +       \
49 |            ++++       00       ++++            \
50 |             +++        0        +++             \
```

Figure 3: Optimal split positions determined by a run of Determine-Optimal-Sizes with $N = 2^{50}$, $M = 2^{20}$, $B = 2^{12}$, and $D = 2^{4}$. Rows show out-of-core problem sizes, and columns are potential split positions. Vertical and diagonal bars border the range of possible split positions. A `+` indicates an optimal split position for the problem size, and a space indicates a non-optimal split position. "Halfway points" (as close as possible to the square root of the problem size) are indicated specially: `#` and `0` indicate optimal and non-optimal split positions, respectively.

```
             11111111112222222222333333333344444444445555555555
    12345678901234567890123456789012345678901234567890123456789

16 |+++++++#+++++++\
17 | ++++++##++++++ \
18 |  ++++++#++++++  \
19 |   +++++##+++++   \
20 |    +++++#+++++    \
21 |     ++++##++++     \
22 |      ++++#++++      \
23 |       +++##+++       \
24 |        +++#+++        \
25 |         ++##++         \
26 |          ++#++          \
27 |           +##+           \
28 |            +#+            \
29 |             ##             \
30 |              #              \
31 |   +++   +++  00  +++   +++   \
32 |    ++    ++   0   ++    ++    \
33 |     +     +   00   +     +     \
34 |   ++++++++++++ 0 ++++++++++++   \
35 |    ++ +++++ ++ 00 ++ +++++ ++    \
36 |     +  ++++  +  0  +  ++++  +     \
37 |         +++     00     +++         \
38 |          ++      0      ++          \
39 |           +      00      +           \
40 |         ++++++    0    ++++++         \
41 |          ++ ++    00    ++ ++          \
42 |           +  +     0     +  +           \
43 |            +++     00     +++            \
44 |             ++      0      ++             \
45 |              +      00      +              \
46 |   ++++++++++++   +++ 0 +++   ++++++++++++   \
47 |    ++ +++++ ++    ++ 00 ++    ++ +++++ ++    \
48 |     +  ++++  +     +  0  +     +  ++++  +     \
49 |         +++           00           +++         \
50 |          ++            0            ++          \
51 |           +            00            +           \
52 |         ++++++         +#+         ++++++         \
53 |          ++ ++          ##          ++ ++          \
54 |           +  +           #           +  +           \
55 |         ++++++         ++##++         ++++++         \
56 |          ++ ++          ++0++          ++ ++          \
57 |           +  +           +00+           +  +           \
58 |            +++            +#+            +++            \
59 |             ++             ##             ++             \
60 |              +              #              +              \
```

Figure 4: Optimal split positions determined by a run of Determine-Optimal-Sizes with $N = 2^{60}$, $M = 2^{15}$, $B = 2^9$, and $D = 2^4$.

dynamic programming to determine the course of an out-of-core algorithm for the PDM. The only other out-of-core work that we know of using dynamic programming in such a fashion is by Li [Li96].

Acknowledgments

Many thanks to David Nicol for numerous helpful discussions. Thanks also to the Institute for Mathematics and Its Applications at the University of Minnesota for inviting the author to a workshop there in September 1996; conversations at the workshop with Eric Schwabe were most valuable.

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.

[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*.

[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.

[Li96] Zhiyong Li. *Computational Models and Program Synthesis for Parallel Out-of-Core Computation*. PhD thesis, Department of Computer Science, Duke University, 1996.

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

[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.

[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.