LA-SUB--95-178

RECEIVED

OCT 13 1995

OSTI

FINAL REPORT

SUB-CONTRACT #1042N0014-9H

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER ^{kg}

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

DEVELOPMENT OF OUT-OF-CORE FAST FOURIER TRANSFORM SOFTWARE FOR THE CONNECTION MACHINE

by

Roland Sweet and John Wilson
Center for Computational Mathematics
University of Colorado at Denver

1 Abstract

This report describes the algorithm and implementation of an out-of-core Fast Fourier Transform routine for the Thinking Machines Corp. CM-5 parallel computer. The software has the capability of transforming multi-dimensional arrays that are both real and complex.

2 Introduction

The purpose of this project was to produce Fortran software for the Thinking Machines Corp. CM5 computer that would calculate the Fast Fourier Transform (FFT) of a complex multi-dimensional array that was too large to fit within the main memory of a CM5. For each axis of the array that is to be transformed the length of that axis must be a power of two. The software emulates the structure of the CMSSL FFT routines. This report describes the algorithm (section 3), how the vector is divided into blocks for processing (section 4), and gives implementation details in section 5. The extension of this basic out-of-core algorithm is extended to multi-dimensional arrays in section 6. A large portion of this project has been the coding of the out-of-core bit-reversal algorithm which essentially duplicates the data access patterns of the FFT. Therefore, the cost of the bit-reversal will be about equal to the cost of the FFT. The bit-reversal is built upon the topic of index digit permutations which is described in section 7. Section 8 applies this theory to the out-of-core bit-reversal. Brief remarks about the inverse FFT are made in section 9. The treatment of the case when the data array is real is described in section 10. It is likely that the execution time for this software will be measured in hours. (For example, the FFT of a single complex vector of length 2^{30} takes about 45 minutes on the machine "phantom" at TMC.) Hence, it was deemed desirable to incorporate as much of a restart facility as possible into the software. This facility is described in section 11. Other small details are described in the remaining sections 12-14.

3 The DFT and IDFT

The Discrete Fourier Transform (DFT) of a given complex sequence $\{x_n\}_{n=0}^{N-1}$ is the complex sequence $\{X_k\}_{k=0}^{N-1}$ given by

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{nk}, \quad (1)$$

where $\omega_N = e^{-2\pi i/N}$ and $i = \sqrt{-1}$. The inverse DFT (IDFT) is given by

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \omega_N^{-nk}. \quad (2)$$

The Fast Fourier Transform (FFT) is an algorithm that reduces the number of operations required to compute the DFT (or IDFT) from N^2 to just $N \log N$. The algorithm was originally published by Cooley and Tukey [1] and has been the subject of countless papers since then.

We will develop the FFT using ideas and especially the notation of Swarztrauber [3]. His notation is particularly useful for the out-of-core nature of this project. Suppose N has factors $N = N_0 N_1$. We define the two-dimensional arrays

$$x(i, j) = x_n, \quad i = 0, 1, \dots, N_0 - 1, \quad j = 0, 1, \dots, N_1 - 1 \quad (3)$$

and

$$X(l, m) = X_k, \quad l = 0, 1, \dots, N_1 - 1, \quad m = 0, 1, \dots, N_0 - 1 \quad (4)$$

using the index mappings

$$n = i + jN_0, \quad k = l + mN_1. \quad (5)$$

Substituting equations (3-5) into (1) and simplifying gives

$$X(l, m) = \sum_{i=0}^{N_0-1} \left[\omega_N^{il} \sum_{j=0}^{N_1-1} x(i, j) \omega_{N_1}^{jl} \right] \omega_{N_0}^{im}. \quad (6)$$

The terms in the brackets above represent N_0 DFTs of length N_1 multiplied by ω_N^{il} which we denote by

$$X^{(1)}(i, l) = \omega_N^{il} \sum_{j=0}^{N_1-1} x(i, j) \omega_{N_1}^{jl}. \quad (7)$$

The remaining terms represent N_1 DFTs of length N_0 which we denote by

$$X^{(2)}(m, l) = \sum_{i=0}^{N_0-1} X^{(1)}(i, l) \omega_{N_0}^{im}. \quad (8)$$

The final transform in its natural order is obtained by the transpose

$$X(l, m) = X^{(2)}(m, l). \quad (9)$$

This algorithm can be extended to a generalized factoring of N into many factors. To illustrate let m, b, n, r be integers that satisfy the following conditions.

$$N = 2^m, \quad b < m - n, \quad n \geq 1 \quad (10)$$

$$s = \lceil \frac{m-b}{n} \rceil, \quad r = m - b - (s-1)n, \quad (11)$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x . (This definition makes r satisfy the unusual inequalities $0 < r \leq n$). Assuming that N has the special decomposition $N = N_0 N_1 \cdots N_s$, where

$$N_0 = 2^b, \quad N_1 = 2^r, \quad (12)$$

$$N_2 = N_3 = \cdots = N_s = 2^n, \quad (13)$$

we introduce the index mappings

$$n = i_0 + i_1 N_0 + i_2 N_0 N_1 + \cdots + i_s N_0 N_1 \cdots N_{s-1} \quad (14)$$

$$k = k_s + k_{s-1} N_s + k_{s-2} N_s N_{s-1} + \cdots + k_0 N_s N_{s-1} \cdots N_1, \quad (15)$$

and from these define the $(s+1)$ -dimensional arrays

$$x(i_0, i_1, i_2, \dots, i_s) = x_n, \quad X(k_s, k_{s-1}, \dots, k_0) = X_k, \quad (16)$$

where

$$0 \leq i_l \leq N_l - 1, \quad 0 \leq k_l \leq N_l - 1, \quad l = 0, 1, \dots, s. \quad (17)$$

Now, substituting equations (14-16) into (1) and simplifying, we can express the FFT as the $(s+1)$ -step iteration

$$X^{(j+1)}(i_0, \dots, i_{s-j-1}, k_{s-j}, \dots, k_s) = \omega_{N(j)}^{i k_{s-j}} \sum_{i_{s-j}=0}^{N_{s-j}-1} X^{(j)}(i_0, \dots, i_{s-j-1}, i_{s-j}, k_{s-j+1}, \dots, k_s) \omega_{N_{s-j}}^{i_{s-j} k_{s-j}}, \quad (18)$$

for $j = 0, 1, \dots, s$, where $X^{(0)}(i_0, \dots, i_s) = x(i_0, \dots, i_s)$ is the original data vector, and

$$i = i(s) = I(i_0, i_1, \dots, i_{s-j-1}) = i_0 + i_1 N_0 + \cdots + i_{s-j} N_0 \cdots N_{s-j-1}, \quad (19)$$

$$N(s) = N_0 \cdots N_{s-j+1}. \quad (20)$$

The transform is completed in just s passes through the data. The first $s-1$ passes consist of transforming one group of n binary digits at a time starting from the highest and going to the lowest. On the s^{th} pass, all of the remaining $r+b$ digits are transformed. The end result, $X^{(s+1)}$, is the bit-reversed array of Fourier coefficients X_k . To convert $X^{(s+1)}(k_0, k_1, \dots, k_s)$ to a naturally-ordered array, we must perform the index-digit permutation

$$X_k = X(k_s, \dots, k_1, k_0) \leftarrow X^{(s+1)}(k_0, k_1, \dots, k_s). \quad (21)$$

We use the work of Fraser [2] to guide us in the development of an out-of-core transposition routine.

4 Description of the Algorithm for the Out-of-Core FFT

To do the out-of-core transform, we need enough main memory to hold $3 \cdot 2^{b+n}$ elements. We need this much main memory because we read 2^{b+n} elements of the sequence to be transformed into main memory and then we need space for two twiddle factor tables each of size 2^{b+n} . The twiddle factor tables are the ω terms in the factorization of the DFT. One twiddle factor table corresponds to the in-core FFT and the other corresponds to the multiplication by ω on the outside of the left side of equation (18). To avoid confusion, we will call the twiddle factor table outside of the DFT a phase shift table instead of a twiddle factor table.

Assume that the data x_n to be transformed is stored contiguously on a random access disk in blocks of size 2^b elements. So the digit i_0 indexes the contiguous elements within a block and the group of digits $i_1, i_2, \dots, i_s$ represent the location of a block on disk (block addresses). The transform will take s disk passes, one for each index $i_1, i_2, \dots, i_s$ with index i_0 being transformed on the last pass. A disk pass is when every element of the array on disk has been read, processed, and written back to the disk. Now, consider the j^{th} pass corresponding to equation (18). We can use three nested loops to complete one pass and thus compute (18). The digits of the block address can be split into three groups, as shown below, which will define the limits of the loops.

$$m_1(i_1, \dots, i_{s-j-1}) = 0, 1, \dots, M_1 - 1, \quad M_1 = N_1 N_2 \cdots N_{s-j-1} \quad (22)$$

$$m_2(i_{s-j}) = 0, 1, \dots, M_2 - 1, \quad M_2 = N_{s-j} \quad (23)$$

$$m_3(i_{s-j+1}, i_{s-j+2}, \dots, i_s) = 0, 1, \dots, M_3 - 1, \quad M_3 = N_{s-j+1} N_{s-j+2} \cdots N_s \quad (24)$$

Thus, we use the multi-index (m_1, m_2, m_3) to define the block address $m_1 + m_2 M_1 + m_3 M_1 M_2$. The limits M_i can be written in terms of m, b, n , and r as shown in the following table.

j	r	M_1	M_2	M_3
$0, 1, \dots, s-1$	all	$2^{m-b-(j+1)n}$	2^n	2^{jn}
s	n	1	2^n	$2^{(s-1)n}$
s	$< n$	1	2^r	2^{m-b-r}

The first row of this table gives the values of M_1, M_2 , and M_3 for the first $s-1$ "regular" steps. The last two rows give the limits for the last step which depends upon whether $r = n$ or not. Note that for all j , $M_1 M_2 M_3 = 2^{m-b}$, the total number of blocks.

The algorithm for one FFT pass is given in Table 1.

5 Implementation Details

The algorithm is designed for a CM5 which has a distributed memory. We view our program as of type SIMD (Single Instruction Multiple Data), operating in a data-parallel manner. In this case, we want to do our computations on groups of elements which are in the same processor. This is done by specifying the layout of the arrays (lines 3 and 4). We chose the axis of A with extent N_0 to have its elements distributed across the processor (NEWS

ORDER) and the other axis to be local (SERIAL ORDER). Thus, up to N_0 FFTs can be done in parallel using the CMSSL FFT routine. This routine requires calling another routine which calculates a twiddle factor table to be used within the FFT. The resulting twiddle factor table can be used repeatedly for transforms on arrays with the same dimension. Thus, the setup need only be calculated once to take care of all the passes (and subsequent out-of-core FFTs) except in the case when $r < n$. In this case, we will have to calculate a new setup on the last pass. We make no attempt to save the phase shift tables from pass to pass since the memory requirements would defeat the purpose of having an out-of-core FFT. However, we can do the phase shift table computation in parallel (lines 6-7). Also, the way we arrange the order of our loops (notice that m_2 is the inner most loop) allows a phase shift table to be reused M_3 times on a given pass. In fact, on the last pass, we only have to compute one phase shift table. Notice that we bit-reverse axis 2 of the phase shift table (line 8). This is necessary because the in-core FFT produces bit-reversed coefficients and when we multiply by the phase shift table (line 15), the indices must match. The bit-reversal of the phase shift table is performed by the CMSSL FFT routine. This is an efficient way of bit-reversing an index. We can also do all of the computations of the element-wise product of the phase shift table and the matrix A in parallel (line 15). The last pass will require transforming vectors that are distributed across the processors (NEWS ORDER) rather than SERIAL ORDER. We rely on the CMSSL FFT library routine to handle this case appropriately. We need axis 1 to be NEWS ORDERED for the parallel I/O operations. We use the CMSSL parallel I/O routines to do all read and writes.

6 Multi-Dimensional Arrays

This software has the ability to transform some or all of the axes of an array of at most three dimensions. We will show how the previous looping structure is modified to incorporate the extras dimensions.

Suppose x is the three-dimensional array $x(0 : L - 1, 0 : M - 1, 0 : N - 1)$. We assume that L, M , and N are powers of 2 if the corresponding axis of the array will be transformed. If an axis is not transformed, the corresponding extent need not be a power of 2, but some divisibility constraints will be given below.

Now the idea of treating the multi-dimensional array is to recognize that basically the added dimensions only possibly increase the range of the first and third components of the multi-index (m_1, m_2, m_3) used earlier. For instance, if the first axis (L) of x is transformed, then the 2nd and 3rd axes only increase the range of the index m_3 , while if the 3rd axis (N) is transformed, then the 1st and 2nd axes only increase the range of m_1 . If the 2nd axis (M) is transformed, then the ranges of both m_1 and m_3 are increased. However, to fully describe the modifications, we must consider the relationship of the blocksize to the extent L .

Let us now describe the modifications to the algorithm that will allow the software to transform the middle index of a three-dimensional array. (The two other cases will follow readily from this.) We extend the four component multi-index (i_0, m_1, m_2, m_3) that gives not only the block address but also the address within the block to a multi-index with seven components: $(l_0, l_1, m_0, m_1, m_2, m_3, n_0)$, with ranges $0 \leq l_i \leq L_i - 1$, $0 \leq m_i \leq M_i - 1$, $0 \leq n_0 \leq N - 1$. Formerly we denoted the blocksize by 2^b . Now we recognize that it may not be a power of 2 so we just denote it by $BLKSIZE$. There are two cases to distinguish.

1. Case I ($L \geq BLKSIZE$). In this case each block contains only the first axis of the

array which will not be transformed. We set $M_0 = 1, L_0 = BLKSIZE$, and assuming that $BLKSIZE$ divides L , we set $L_1 = L/L_0$. Now the block address consists of the multi-index $(l_1, 0, m_1, m_2, m_3, n_0)$ which is calculated by

$$l_1 + L_1(m_1 + M_1(m_2 + M_2(m_3 + M_3(n_0))))$$

The looping structure now has 5 loops and is shown in Table 2. The number of passes through the data is $\lceil \frac{m}{n} \rceil$.

2. Case II ($L < BLKSIZE$) In this case some of the indices to be transformed lie within the blocks. This must be accounted for on the last pass. We set $L_1 = 1, L_0 M_0 = BLKSIZE, L_0 = L$, and make the assumption that $M_0 = BLKSIZE/L = 2^q$, i.e. that M_0 is a power of 2. The addressing within the block is done through the multi-index $(l_0, 0, m_0)$ which is computed by $l_0 + m_0 L_0$ while the block address is given by (m_1, m_2, m_3, n_0) , which is computed by

$$m_1 + M_1(m_2 + M_2(m_3 + M_3(n_0))).$$

The looping structure now has 4 loops and is shown in Table 3. The number of passes through the data in this case is $\lceil \frac{m-q}{n} \rceil$.

7 Array Permutation by Index Digit Permutation

The non-trivial task of developing an out-of-core bit-reversal algorithm is greatly simplified using Fraser's [2] approach. Array permutation by index-digit permutation is a rich subject, but we will be brief in our treatment of it and only use what is necessary to lead us to an out-of-core bit-reversal algorithm.

We will assume that we are dealing strictly with binary index digits. So an index is made up of digits $d_0, \dots, d_{m-1}$ and the index number is computed by $d_0 + d_1 \cdot 2 + \dots + d_{m-1} \cdot 2^{m-1}$.

We will need two basic index-digit permutations and some notation. Note below that the permutation operator is written on the right-side of its argument and that sequences of permutations are read from left to right.

A *digit-reversal* (bit-reversal in our case) permutation, denoted ${}_g R_h$, is a permutation where the digits d_g through d_{h-1} of an array A are reversed producing the reordered array A' as shown below.

$$\begin{aligned} & A'(d_0, \dots, d_{g-1}, d_{h-1}, d_{h-2}, \dots, d_g, d_h, \dots, d_{m-1}) \\ &= A(d_0, \dots, d_{g-1}, d_g, d_{g+1}, \dots, d_{h-1}, d_h, \dots, d_{m-1}) \cdot [{}_g R_h] \end{aligned}$$

When $g = 0$, the preceding subscript is dropped so that ${}_0 R_h \equiv R_h$. Also note that ${}_g R_h$ is its own inverse transformation.

A *digit-cycled permutation*, denoted ${}_g S_h$, is a permutation where the digits d_g through d_{h-2} of an array A are shifted right one position and the digit d_{h-1} is cycled back to where d_g was. The permutation is shown below.

$$\begin{aligned}
& A'(d_0, \dots, d_{g-1}, d_{h-1}, d_g, \dots, d_{h-2}, d_h, \dots, d_{m-1}) \\
& = A(d_0, \dots, d_{g-1}, d_g, d_{g+1}, \dots, d_{h-1}, d_h, \dots, d_{m-1}) \cdot [{}_g S_h]
\end{aligned}$$

Note that applying the transformation ${}_g S_h$ $h - g$ times reproduces the original sequence, so $({}_g S_h)^{h-g}$ is the identity.

We now present some useful equivalent permutations.

$$({}_g S_h)^n = {}_f R_{h-n} \cdot {}_f R_h \cdot {}_f R_{g+n} \cdot {}_f R_g, \quad f \leq g, \quad n \leq h - g \quad (25)$$

$$({}_g S_h)^n = {}_g R_{h-n} \cdot {}_g R_h \cdot {}_g R_{g+n}, \quad n \leq h - g \quad (26)$$

$${}_g S_m = {}_b S_m \cdot {}_g S_{b+1}, \quad g \leq b \quad (27)$$

$${}_g S_h = R_h \cdot R_{h-g} \cdot R_{h-g-1} \cdot R_h \quad (28)$$

$${}_g S_h = {}_g R_{h-1} \cdot {}_g R_h \quad (29)$$

$$\prod_{i=0}^k ({}_i S_h) = R_{h-k-1} \cdot R_h \quad (30)$$

$$\prod_{i=j}^k ({}_i S_h) = {}_j R_{h+j-k-1} \cdot {}_j R_h \quad (31)$$

8 Algorithm for Out-of-Core Bit-Reversal

Assume that m , b , n , and r are as defined earlier in equations (10-13). Bit-reversal is the permutation $[R_m]$, which from identity (30) can be written as

$$\begin{aligned}
[R_m] &= [R_b] \cdot [R_b \cdot R_m] \\
&= [R_b] \cdot \prod_{i=0}^{m-b-1} [{}_i S_m].
\end{aligned} \quad (32)$$

To implement the out-of-core bit-reversal we must distinguish two cases.

Suppose that $b > (m - 1)/2$. We will break up the right-side product in (32) into sub-products of length n

$$[R_m] = [R_b] \cdot \prod_{i=0}^{n-1} [{}_i S_m] \cdot \prod_{i=n}^{2n-1} [{}_i S_m] \cdots \prod_{i=(s-1)n}^{(s-1)n+r-1} [{}_i S_m]. \quad (33)$$

We simplify each of the products above using the following special case of identity (31):

$$\prod_{i=j}^{j+n-1} [{}_i S_m] = [{}_j R_{m-n} \cdot {}_j R_m]. \quad (34)$$

Using (25) we can show that (34) simplifies to

$$\begin{aligned}
\prod_{i=j}^{j+n-1} [{}_i S_m] &= [{}_j R_{m-n} \cdot {}_j R_m \cdot {}_j R_{b+n} \cdot {}_j R_b] \cdot {}_j R_b \cdot {}_j R_{b+n} \\
&= [{}_b S_m]^n \cdot [{}_j R_b] \cdot [{}_j R_{b+n}]
\end{aligned} \quad (35)$$

and use (35) to re-write equation (33) as

$$[R_m] = [R_b] \cdot [{}_bS_m]^n \cdot [R_b] \cdot [R_{b+n}] \cdot [{}_bS_m]^n \cdot [{}_nR_b] \cdot [{}_nR_{b+n}] \cdot \dots \cdot [{}_bS_m]^n \cdot [{}_{(s-2)n}R_b] \cdot [{}_{(s-2)n}R_{b+n}] \cdot [{}_bS_m]^r \cdot [{}_{(s-1)n}R_b] \cdot [{}_{(s-1)n}R_{b+r}]. \quad (36)$$

Notice that $[R_b]$ and $[{}_bS_m]^n$ operate on different digits so these two permutations commute and thus, the first four terms of the right side of equation (36) reduce to

$$[{}_bS_m]^n \cdot [R_{b+n}], \quad (37)$$

which is the first operation we must perform.

Now refer to the $(s + 1)$ -dimensional arrays defined in (16). Note that $x'(i_0, i_s, i_1, \dots, i_{s-1}) = x(i_0, i_1, \dots, i_s) \cdot [{}_bS_m]^n$. In the out-of-core FFT we read in the n binary digits corresponding to i_s so that in the memory of the computer we have the sub-matrix $x(i_0, i_s)$. The remaining permutation of the first operation that we must perform is R_{b+n} on the permuted array $x'(i_0, i_s, i_1, \dots, i_{s-1})$ involves only the first $b + n$ digits of x' . This is precisely the digits that are in memory. So we reverse those binary digits (using the CMSSL routine) and write the resulting blocks back on disk in the same place. The net effect of the first operation is to reverse the first and last n digits and swap places for these two groups. This procedure is continued, each time placing the next string of highest n digits properly reversed into the appropriate n lower order bit places. This case requires $\lceil \frac{m-b}{n} \rceil = s$ passes through the data.

An example will illustrate what is happening. Let us consider the special case $b = 8, n = 3, r = 2, s = 3, m = 16$. The data array x is indexed with 16 binary digits: $x(n) = x(d_0, \dots, d_7, |d_8, d_9, |d_{10}, d_{11}, d_{12}, |d_{13}, d_{14}, d_{15})$ and is written on a disk in $2^{m-b} = 256$ blocks of $2^b = 256$ elements. The bar $|$ separates the four groups of indices for easy reading. In this example the bit-reversal, equation (36) with equation (37) incorporated, is given by

$$[R_m] = [{}_8S_{16}]^3 \cdot [R_{11}] \cdot [{}_8S_{16}]^3 \cdot [{}_3R_8][{}_3R_{11}] \cdot [{}_8S_{16}]^2 \cdot [{}_6R_8] \cdot [{}_6R_{10}]. \quad (38)$$

In the out-of-core FFT routine we begin by reading all the blocks corresponding to the n highest binary digits (13-15) with a fixed set of $m - b - n$ lower order digits into the memory. This corresponds to having the sub-matrix $x(d_0, \dots, d_7, |d_{13}, d_{14}, d_{15})$ in the memory. We have now performed the equivalent of the first factor $[{}_8S_{16}]^3$ in equation (38). We perform R_{11} to get $x(d_{15}, d_{14}, d_{13}, d_7, \dots, d_3, |d_2, d_1, d_0)$ and, blocking with the usual factor, we write these blocks back to memory. Now on disk x looks like

$$x(d_{15}, d_{14}, d_{13}, d_7, d_6, d_5, d_4, d_3, |d_8, d_9, |d_{10}, d_{11}, d_{12}, |d_2, d_1, d_0).$$

The next block read will read in the indices corresponding to next three lower digits (10-12) so that in memory we have $x(d_{15}, d_{14}, d_{13}, d_7, \dots, d_3, |d_{10}, d_{11}, d_{12})$. Now we perform $[{}_3R_8] \cdot [{}_3R_{11}]$ to get $x(d_{15}, \dots, d_{10}, d_7, d_6, |d_5, d_4, d_3)$. We write this sub-matrix back onto the disk in-place to get

$$x(d_{15}, d_{14}, d_{13}, d_{12}, d_{11}, d_{10}, d_7, d_6, |d_8, d_9, |d_5, d_4, d_3, |d_2, d_1, d_0).$$

Finally we read in the remaining $r = 2$ highest digits (8 and 9) of x to get $x(d_{15}, d_{14}, d_{13}, d_{12}, d_{11}, d_{10}, d_7, d_6, |d_8, d_9)$ and perform $[{}_6R_8] \cdot [{}_6R_{10}]$ to get $x(d_{15}, d_{14}, d_{13}, d_{12}, d_{11}, d_{10}, d_9, d_8, |d_7, d_6)$. When this sub-matrix is written out to disk the new array is

$$x(d_{15}, d_{14}, d_{13}, d_{12}, d_{11}, d_{10}, d_9, d_8, |d_7, d_6, |d_5, d_4, d_3, |d_2, d_1, d_0),$$

the bit-reversed original array x and we are finished. Note that we have made $\lceil \frac{m-b}{n} \rceil = 3$ passes through the data.

Now consider the second case $b \leq (m-1)/2$. We can break up the product in equation (32) to get

$$[R_m] = [R_b] \cdot \prod_{i=0}^{b-1} [{}_iS_m] \cdot \prod_{i=b}^{m-b-1} [{}_iS_m]. \quad (39)$$

Note that the last product above involves only indices within the block address. We show later that these can be reduced to a single permutation. The first product of b terms is broken down into sub-products of n terms each as in the previous case. Hence, we can write

$$[R_m] = [R_b] \cdot \prod_{i=0}^{m-b-1} [{}_iS_m] = [R_b] \cdot \prod_{i=0}^{n-1} [{}_iS_m] \cdots \prod_{i=(j-2)n}^{b-1} [{}_iS_m] \cdot \prod_{i=b}^{m-b-1} [{}_iS_m]. \quad (40)$$

As before, the first permutation $[R_b]$ and the first product can be simplified as in equation (37). Using the same read/writes as in the out-of-core FFT we can effect the first $j-1$ products using the identity (35). As Fraser shows these operations require that the last product in (40) be multiplied by the permutation $[{}_bS_m]^b$. Hence, we find from identities (26) and (31) that

$$\begin{aligned} [{}_bS_m]^b \cdot \prod_{i=b}^{m-b-1} [{}_iS_m] &= [{}_bR_{m-b} \cdot {}_bR_m \cdot {}_bR_{2b}] \cdot [{}_bR_{2b} \cdot {}_bR_m] \\ &= [{}_bR_{m-b}]. \end{aligned} \quad (41)$$

Hence, the final operation $[{}_bR_{m-b}]$ can be performed by making one more pass through the data. On this pass we assume that the data array x is written as $x(i_0, d_b, d_{b+1}, \dots, d_{m-b-1}, i_3)$ where $0 \leq i_0 \leq 2^b - 1$, and $0 \leq i_3 \leq 2^b - 1$. There are 2^{m-2b} block addresses to be bit-reversed. The simple solution would be to create a vector of the integers from 0 to $2^{m-b} - 1$, bit-reverse that vector of indices, and use it to re-write the blocks in their proper order. However, that vector may be too large to fit in the memory so we break the process down into groups of 2^p indices given by $(j-1)2^p, \dots, j \cdot 2^p - 1$, compute the bit-reversal of each group and re-write the blocks in their correct position to complete the bit-reversal. Note that we make $\lceil \frac{b}{n} \rceil + 1$ passes through the data in this case.

To illustrate the algorithm in this case, we consider the case $b = 5, n = 3, r = 2, s = 3, m = 13$. Here, the array x is indexed with 13 binary digits: $x(n) = x(d_0, d_1, d_2, d_3, d_4, |d_5, d_6, |d_7, d_8, d_9, |d_{10}, d_{11}, d_{12})$ and is written on the disk in $2^{m-b} = 256$ blocks of $2^b = 32$ elements. Again the bar $|$ has been added to facilitate reading. For this example the bit-reversal equation (40) with the first $j-1$ products replaced by identity (35) and the last product simplified to identity (41) is given by

$$[R_m] = [{}_5S_{13}]^3 \cdot [R_8] \cdot [{}_5S_{13}]^2 \cdot [{}_3R_5] \cdot [{}_3R_7] \cdot [{}_5R_8].$$

First, we read in the three highest binary digits (10-12) to get into memory $x(d_0, d_1, d_2, d_3, d_4, |d_{10}, d_{11}, d_{12})$ and perform $[R_8]$ to get $x(d_{12}, d_{11}, d_{10}, d_4, d_3, |, d_2, d_1, d_0)$. We write this back to disk in-place leaving the intermediate result

$$x(d_{12}, d_{11}, d_{10}, d_4, d_3, |d_5, d_6, |d_7, d_8, d_9, |d_2, d_1, d_0).$$

We now read in the next two highest digits (8 and 9) to get into memory the sub-matrix $x(d_{12}, d_{11}, d_{10}, d_4, d_3, |d_8, d_9)$ and perform the two permutations $[{}_3R_5] \cdot [{}_3R_7]$ to get $x(d_{12}, d_{11}, d_{10}, d_9, d_8, |d_4, d_3)$. When this sub-matrix is written back to disk in-place, the result is

$$x(d_{12}, d_{11}, d_{10}, d_9, d_8, |d_5, d_6, |d_7, d_4, d_3, |d_2, d_1, d_0).$$

Finally, we make one last pass through the data performing the permutation $[{}_5R_8]$, which when finished leaves the array x on the disk in its natural order. Note that we have made $\lceil \frac{b}{n} \rceil + 1 = 3$ passes through the data to complete the bit-reversal.

9 Inverse Transform

The inverse FFT algorithm that is included in this software was obtained merely by inverting the forward FFT algorithm and software. Hence, all comments made are applicable to both the forward and inverse transform software. There are no special remarks to be made for the inverse transform software.

10 Real-to-Complex and Complex-to-Real FFTs

We have implemented exactly the same pre- and post-processing algorithm that is described in the CMSSL manual. We assume that the user has specified a real array $x(n)$ or length N in natural order, which we pass to our complex-to-complex out-of-core FFT routine as a complex array A of length $\frac{N}{2}$. The output of this routine is the complex Fourier coefficients $\hat{A}(k)$ in bit-reversed (relative to $\frac{N}{2}$ bits) order, written on the disk in blocks of size 2^b .

To motivate our novel treatment of the reversed index $\frac{N}{2} - k$, suppose that there are just two blocks. We read in these two blocks, bit-reverse them (relative to $\frac{N}{2}$). Then they are in their natural order, and we must pay the price of the star-type stencil calculation, i.e. k and $\frac{N}{2} - k$ are combined, which is not efficient on the CM5 as explained in the CMSSL manual. We then bit-reverse the two blocks once again and write them back to the disk. We will then have the conjugate-symmetric Fourier coefficients $\hat{X}(k)$ in their proper bit-reversed order. Let us call this the *star pattern*.

Now suppose there are four blocks. It is easy to see by example that the first two blocks (numbers 0 and 1) should be processed in the same star pattern. Now we read in block 2 in its natural order and block 3 in reverse order. It is not too difficult to prove that this

places the k^{th} and $(\frac{N}{2} - k)^{th}$ components in corresponding locations in the two blocks so that they can be combined in parallel. Then block 2 is written out and block 3 is written out in reverse order. Let's call this operation the *reverse pattern*. The end result is that the Fourier coefficients of x are written on the disk in their proper bit-reversed order.

An example will illustrate this process. Suppose $N = 16$. The output of the complex FFT routine are the eight complex numbers

$$\hat{A}(0), \hat{A}(4), \hat{A}(2), \hat{A}(6), \hat{A}(1), \hat{A}(5), \hat{A}(3), \hat{A}(7),$$

which we have grouped in four blocks of 2 elements. Blocks 0 and 1 are processed using the star pattern. Block 2 is read in natural order— $\hat{A}(1), \hat{A}(5)$ —and block 3 is read in reverse order— $\hat{A}(7), \hat{A}(3)$. Now it is easy to see that the first locations of these two blocks must be combined to get $\hat{X}(1)$ and $\hat{X}(7)$ and the second locations combined to get $\hat{X}(5)$ and $\hat{X}(3)$. Now block 2— $\hat{X}(1), \hat{X}(5)$ —is written out in natural order and block 3— $\hat{X}(7), \hat{X}(3)$ —is written out in reverse order. The result is the transform in its proper order.

If there are 2^{m-b} blocks, then the first two blocks are processed using the star pattern. The remaining blocks are taken in groups of 2, 4, 8, ..., 2^{m-b-1} . We have described how the first group of two blocks is processed. The next group of four (label them 0,1,2,3) are paired first with last (0 with 3) and second with next-to-last (1 with 2). These pairs are then processed as the reverse pattern. The general scheme is given in Table 4.

11 Restart Facility

This software has limited restart capabilities built into it. A restart is possible after the completion of any trip through the m_1 or m_3 loops (refer to Table 1). Hence, at the end of each of these loops the necessary loop parameters are written to *unit3* where they will either be over-written or will be available to user to restart the computation should the machine crash. If a crash occurs during a write, there is no possibility to restart the computation. We see that all writing of data to the disk occurs in the m_2 loop on lines 18-20. Just before entering that loop a logical variable *write_ok* is set to *false* and written to *unit3*. At the successful completion of the write loop, *unit3* is rewound (to erase the previous write), *write_ok* is set to *true* and all intermediate quantities necessary to restart the FFT are written to *unit3*.

12 Parallel I/O and Software Performance

There is a certain amount of latency when using the CM Fortran Parallel I/O facility. We need to transfer large blocks of data to overcome this latency. Our goal is to find an optimal block size given the number of processors of the machine and the number of disks in the SDA. That is, for what value of b will the transfer rate level off? The transfer rates on a 32-processor node machine (each processor node has four processors giving a total of 128 processors) with 47 disks is graphed in Figure 1.

We can see that the rate begins to level off for block transfers of size 2^{21} complex words (16 bytes each). Thus, we would want to use a minimum block exponent size of 21 when computing an out-of-core transform on this particular machine. Suppose we have a sequence of length 2^{26} complex words to transform and only enough memory to do 2^{24} elements at

a time, i.e $b + n \leq 24$. Then, what is the best choice for b and n ? We can examine this question by constructing a table that relates b and n to the number of passes s given by equations (10-11).

$b \backslash n$	1	2	3
21			2
22		2	
23	3		

The diagonal in the table above represents the number of passes s . Here $b = 21$ and $n = 3$ is the best choice.

What if the transform has 2^{28} complex words? Again we construct a table to analyze the best choices for b and n .

$b \backslash n$	1	2	3	4
20				2
21			3	
22		3		
23	5			

In this case the optimal block size would be 2^{22} complex words. The drop in I/O efficiency going from $b = 22$ to $b = 20$ probably would outweigh the time saved by reducing the number of passes by 1. However, experimentation by the user might be appropriate here to choose between these cases.

It would be ideal to have a formula that users could apply to determine the parameters b and n . However, we have made several attempts to analyze the I/O performance and each time gotten completely different execution times for SDA I/O. We suspect that this results from using one of TMCs development machines while they are changing the I/O routines.

Unfortunately, the best we can do is to recommend that a user experiment with his own CM5/SDA configuration to find the optimal blocksize, construct a table as above, and then time various cases of close bs and ns in order to determine the optimal choices. While this seems an undue burden on a user, the amount of computation time that could be saved is well worth the effort.

13 I/O Formats

This software uses FMS I/O utilities for all intermediate read/writes of data. It seemed unreasonable, however, to assume that a user's data would already be in that format so we modified the first and last passes of the FFT and bit-reversal routines to allow the user to provide input and get output as serially-ordered files. This software incorporates the change-of-file formats into the computation phase so there is no inefficiency due to a special internal file format.

14 Known Bugs

There is currently one case that does not compute correctly. It occurs in the multi-dimensional case when $L < BLKSIZE$ and L is not a power of 2. This error surfaced

just as this report was being written. We will continue to examine this case, if it is believed that there is sufficient interest in this special case.

15 References

1. J.W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, 19 (1965), 297-301.
2. D. Fraser, "Array permutation by index-digit permutation." *Journal of the ACM*, 23(1976), 298-309.
3. P. Swarztrauber, "Multiprocessor FFTs," *Parallel Computing*, 5(1987), 197-210.

Table 1: Looping Structure of Out-of-Core Transform

```

1      DOUBLE COMPLEX A(0 : N0 - 1, 0 : Ns-j - 1)
2      DOUBLE COMPLEX W(0 : N0 - 1, 0 : Ns-j - 1)
3 CMF$ LAYOUT A(NEWS:SERIAL)
4 CMF$ LAYOUT W(NEWS:SERIAL)
5      DO (m1 = 0, M1 - 1)
6          FORALL(k1 = m1 * N0 : (m1 + 1) * N0 - 1, k2 = 0 : Ns-j - 1)
7              Compute W(k1, k2) = ωN(j)k1*k2
8              bit-reverse axis 2 of W using CMSSL routine
9              DO (m3 = 0, M3 - 1)
10                 DO (m2 = 0, M2 - 1)
11                     Seek block with address (m1, m2, m3)
12                     Read A(:, m2)
13                 END DO
14                 Transform axis 2 of A using CMSSL routine
15                 Compute A = A*W
16                 IF j = s THEN
17                     Transform axis 1 of A using CMSSL routine
18                 END IF
19                 DO (m2 = 0, M2 - 1)
20                     Seek block with address (m1, m2, m3)
21                     Write A(:, m2)
22                 END DO
23             END DO
24         END DO

```


Table 2: Looping Structure for Multi-dimensional Array: Case I

```

1      DOUBLE COMPLEX A(0 : L0 - 1, 0 : 0, 0 : Ns-j - 1)
2      DOUBLE COMPLEX W(0 : L0 - 1, 0 : 0, 0 : Ns-j - 1)
3 CMF$ LAYOUT A(NEWS:NEWS:SERIAL)
4 CMF$ LAYOUT W(NEWS:NEWS:SERIAL)
5      DO (m1 = 0, M1 - 1)
6          FORALL(k2 = 0 : Ns-j - 1)
7              Compute W(:, 0, k2) =  $\omega_{N(j)}^{m_1 * k_2}$ 
8              bit-reverse axis 3 of W using CMSSL routine
9              DO (l1 = 0, L1 - 1)
10                 DO (n0 = 0, N0 - 1)
11                     DO (m3 = 0, M3 - 1)
12                         DO (m2 = 0, M2 - 1)
13                             Seek block with address (l1, 0, m1, m2, m3, n0)
14                             Read A(:, 0, m2)
15                         END DO
16                         Transform axis 3 of A using CMSSL routine
17                         Compute A = A*W
18                         DO (m2 = 0, M2 - 1)
19                             Seek block with address (l1, 0, m1, m2, m3, n0)
20                             Write A(:, 0, m2)
21                         END DO
22                     END DO
23                 END DO
24             END DO
25         END DO

```

Table 3: Looping Structure for Multi-dimensional Array: Case II

```

1      DOUBLE COMPLEX A(0 : L0 - 1, 0 : M0 - 1, 0 : Ns-j - 1)
2      DOUBLE COMPLEX W(0 : L0 - 1, 0 : M0 - 1, 0 : Ns-j - 1)
3 CMF$ LAYOUT A(NEWS:NEWS:SERIAL)
4 CMF$ LAYOUT W(NEWS:NEWS:SERIAL)
5      DO (m1 = 0, M1 - 1)
6          FORALL(k1 = 0 : M0 - 1, k2 = 0 : Ns-j - 1)
7              Compute W(:, m1 * M0 : (m1 + 1) * M0 - 1, k2) =  $\omega_{N(j)}^{k_1 * k_2}$ 
8              bit-reverse axis 3 of W using CMSSL routine
9              DO (n0 = 0, N - 1)
10                 DO (m3 = 0, M3 - 1)
11                     DO (m2 = 0, M2 - 1)
12                         Seek block with address (m1, m2, m3, n0)
13                         Read A(:, :, m2)
14                     END DO
15                     Transform axis 3 of A using CMSSL routine
16                     Compute A = A * W
17                     IF j=s THEN
18                         Transform axis 2 of A using CMSSL routine
19                     END IF
20                     DO (m2 = 0, M2 - 1)
21                         Seek block with address (m1, m2, m3, n0)
22                         Write A(:, :, m2)
23                     END DO
24                 END DO
25             END DO
26         END DO

```

Table 4: Accessing of Blocks in Real-to-Complex Algorithm

```

DO (l1 = 0, ..., m - b - 2)
    k1 = 2l1+1
    k2 = 2l1+2 - 1
    k3 = 2l1
    DO (l2 = 0, ..., 2l1 - 1)
        Read block k1 + l2 in natural order
        Read block k2 - l2 in reverse order
        Compute appropriate phase-shift factors
        Combine
        Write block k1 + l2
        Write block k2 - l2 in reverse order
    END DO
END DO

```

Figure 1.

TRANSFER RATES