AREA 4 WORKING NOTE 14: REAL AND COMPLEX FAST FOURIER TRANSFORMS ON THE FUJITSU VPP 500[†]

MARKUS HEGLAND^{*}

June 1994

ABSTRACT. Fast Fourier transforms parallelize well but need large amounts of communication. An algorithm which concentrates all the communication in one or two transposition steps is the transpose split algorithm. Different transposition algorithms can be used depending on data size and communication latency. A new transpose split algorithm for real and hermitian data is presented for one, two and three dimensional transforms.

This algorithm is implemented on the Fujitsu VPP 500. The Fujitsu VPP 500 is a parallel processor with a moderate number of very fast vector processors connected by a crossbar switch. Each processor has a peak performance of 1.6 Gflop/s and can simultaneously read and write 400 MByte/s.

Very long vector length stride one implementations of multiple FFTs on one node [Hegland, Numerische Mathematik, to appear, 1994] are combined with optimized transpositions. One third of peak performance was achieved on a configuration with up to 11 processors.

1991 *Mathematics Subject Classification.* 65T,65Y.

Key words and phrases. Fast Fourier transforms, parallel processing, Fujitsu VPP 500, split transpose algorithm.

[†]appeared as CMA report MRR 034-94 and as Joint DCS/CSL report TR-CS-94-07

^{*}Centre for Mathematics and its Applications/Computer Sciences Laboratory, Australian National University, Canberra ACT 0200, Australia, Markus.Hegland@anu.edu.au

1. INTRODUCTION

In 1965 Cooley and Tukey [2] published a fast algorithm for the discrete Fourier transform (FFT) which revolutionized many areas of scientific computing. What made the transform so attractive is the floating point operation count of $O(n \log(n))$ for a transform of an array with n elements. If implemented as an ordinary matrix vector product $O(n^2)$ floating point operations would be needed, even if the computation of the elements of the Fourier transform matrix was free.

It was soon seen that the individual steps of the FFT algorithm involve a large degree of parallelism [11]. This parallelism was very successfully exploited on vector processors [9, 12, 16, 13, 7, 6]. Implementations on distributed memory processors [15, 8, 5] showed that the operation count was so low that even the necessary $O(n)$ communication contributed to a significant part to the overall execution time. Several attempts to reduce communication or at least overlap communication with computation [1] were made. We will discuss the *transpose split* algorithm. It uses very simple data distributions and only requires one communication step which essentially is a matrix transposition. We will also suggest a new variant of the transpose-split algorithm for real data which does essentially show the same features as the complex transform and even mainly uses the same basic operations while requiring half the floating point operations and storage space.

In 1993 Fujitsu Ltd Japan announced its VPP 500 vector parallel computer which seems perfectly suited for FFTs. An earlier prototype of this computer was for a long time number one in the Linpack Benchmark rating [17]. The architecture combines a moderated number (up to 222) of very efficient vector processors with 1.6 Gflop/s peak performance each. This gives a total of 0.36 Tflop/s for the full configuration. As the number of processors is kept relatively low, a crossbar switch is feasible. Communication bandwidth is 400 MByte/s per processor for each sending and receiving data which can be done in parallel. It is this high connectivity and communication bandwidth combined with the computational power that forms the basis for successful FFT implementations presented here.

Real Fourier transforms are usually interpreted as complex transforms with a special symmetry [14]. An extensive review of FFT algorithms can be found in [10].

Our algorithms obtained 1/3 of the peak performance for our FFTs on the VPP 500 where usually only between 1/10 and 1/5 is obtained by the FFT libraries on other platforms. We compare an 11 processor VPP 500 with distributed memory processors in a similar peak performance range in Table 1. As the operation counts can vary for different algorithms we use as performance formula a scaled inverse time:

$$r_{\text{FFT}} := \frac{t}{5n \log_2(n)}$$

where t is the elapsed time and n is the size of the transform. We compared the complex one dimensional transform using 64 Bit floating point arithmetic. The transforms are self-sorting, i.e., the result is in the correct order and $n = 2^{22} \approx 4 \cdot 10^6$, however the CM-200 data is for an unordered transform, and, if required, ordering time would have to be added.

The algorithms which we present here will form part of Fujitsu's SSLII Scientific Subroutine Library of the VPP500. They are implemented in a modular way using basic subroutines for transposition, multiple small FFTs and twiddle factor multi-

Platform	FFT performance r_{FFT}	Peak Performance
11 processor VPP 500	5.8 Gflop/s	17.6 Glop/s
128 processor CM-5	1.6 Gflop/s	16.4 Gflop/s
2048 processor CM-200 [8]	2.7 Gflop/s (unordered)	20 Gflop/s

TABLE 1. Performance of the 1D complex FFT on various distributed memory computers, $n = 2^{22} \approx 4 * 10^6$.

plications [7]. So far they include mixed-radix, real and complex FFTs of one, two and three-dimensional arrays. All transforms are self-sorting.

The remaining parts are as follows: In Section 2 we discuss the implementation of the transpose-split algorithm for complex data. The uniform exposition is new and uses matrices to represent two, three and even one dimensional data. All steps are either matrix matrix products, matrix transpositions or Hadamard (or element wise) matrix matrix products. Two algorithms for matrix transposition are compared using a simple model of communication time. In Section 3 the real FFT is discussed. As the Fourier transform of real arrays is hermitian only half the elements need to be computed and the rest can be obtained from symmetry. We show how this is done for one, two and three dimensional data. The computational steps of the split-transpose algorithms consist of multiple transforms. In order to do the transforms of 2 real arrays the arrays are combined into one complex array and the transforms of the original arrays can be obtained from the transform of the complex array [10]. Finally, we suggest split-transpose algorithms for the transforms of hermitian data.

2. COMPLEX DATA

2.1. The two-dimensional transform. The transform of two dimensional data is the simplest case of the split-transform algorithm. We thus look at it first. The Fourier transform of a two dimensional array

$$X = \begin{bmatrix} \xi_{0,0} & \cdots & \xi_{0,m-1} \\ \vdots & & \vdots \\ \xi_{n-1,0} & \cdots & \xi_{n-1,m-1} \end{bmatrix}$$

is prescribed by a double matrix-matrix product as

$$Y := F_n X F_m.$$

This product suggests *splitting* into two steps using the auxiliary matrix $Z \in \mathbb{C}^{n \times m}$ as:

$$\begin{aligned} Z &:= X F_m \\ Y &:= F_n Z. \end{aligned}$$

We use a distribution of X, Y and Z which partitions the columns into blocks of equal size as in Figure 1. Actually any other column-wise balanced distribution works as well as long as X, Y and Z are distributed in the same way. We choose this distribution because it is widely used and usually does not require too much extra data movement if the data is to be used in other computations as well. Other distributions have been used in [1].

$X^{(0)}$	on processor 0
$X^{(1)}$	on processor 1
$X^{(2)}$	on processor 2
$X^{(3)}$	on processor 3

FIGURE 1. Blocked column distribution of X over 4 processors.

The first step of the splitting does not need any communication and every processor does the same transform, although with different data. If $q \in \{0, \dots, p-1\}$ is the processor number then this is

$$Z^{(q)} := X^{(q)} F_m.$$

This function is part of the local library of every processor.

The second splitting step needs to combine data from different processors. This can be done by transposing the data and using the local transforms again. The second splitting step is then done with the following three steps:

$$\begin{aligned} U &:= Z^T \\ V &:= U F_n \\ Y &:= V^T. \end{aligned}$$

The transform step uses only the local complex Fourier transform. However, two matrix transpositions are needed. We will discuss transposition algorithms in the next subsection.

If the local transforms are in-place then the whole transform can be done using two arrays only. The algorithm for the two dimensional FFT which overwrites the data X with the transform is then

$$\begin{array}{l} X := X F_m \\ Y := X^T \\ Y := Y F_n \\ X := Y^T. \end{array} \quad \text{Split transpose algorithm for 2D FFT}$$

Actually the workspace can be even further reduced. On a 4 processor VPP 500 this algorithm needed 0.224 sec for a 2048 by 2048 transform which corresponds to 2.0 Gflop/s. The local Fourier transforms are done with an algorithm which was presented in [7]. The transposition is done with Algorithm 1 as discussed in the next section.

2.2. Transposition. The matrix transpositions are the communication steps of the split transpose algorithm. Matrix transposition is an all-to-all personalized communication. In this section we recall and discuss two transposition algorithms, see [10, 3].

The local partitions of X are written as block matrices

$$X^{(q)} = (X_0^{(q)}, X_1^{(q)}, \dots, X_{p-1}^{(q)})$$

of p blocks of equal (or almost equal) size. During the transposition processor q has to send the block $X_j^{(q)}$ to processor j and $U_q^{(j)} = X_j^{(q)T}$. If this is not done carefully, network contention can arise even on a crossbar switch. Contention is avoided in the following algorithm:

```

for  $j := 1, \dots, p-1$ 
  send  $X_{k+j \bmod p}^{(q)}$  to processor  $q + j \bmod p$ 
  receive  $Y_{q-j \bmod p}^{(q)}$  from processor  $q - j \bmod p$ 

```

Parallel transpose algorithm 1

We assume that the processor waits with sending any data until previous receives have completed. For the actual transposition all the blocks need to be transposed before or after sending and receiving as well. We are only interested in the communication costs here and so we omit these local steps in the algorithms.

A very simple model for the communication time spent includes communication bandwidth r_c and latency λ . Set $n_{1/2} := \lambda r_c$. With this model the time needed to transpose an n by m matrix with the first algorithm is

$$t = 2r_c^{-1}(n_{1/2} + nm/p^2)(p-1).$$

On the Fujitsu the communication bandwidth is 400 MByte/sec which corresponds to 50 Million 64 Bit words per second and the latency is $\tau \approx 5\mu\text{sec}$. Thus $n_{1/2} \approx 250$. The factor 2 in the timing formula reflects the fact that we transpose a complex matrix.

A second algorithm is based on the binary digit representation of j and k as

$$j = j_0 + j_1 2 + \dots + j_{t-1} 2^{t-1}, \quad j_s \in \{0, 1\}.$$

The algorithm is then $t = \log_2(p)$ after initialization $Y := X$:

```

for  $s := 0, \dots, t$ 
  send all  $Y_{j \oplus 2^s}^{(q)}$  with  $j_s = q_s$  to processor  $q \oplus 2^s$ 
  receive all  $Y_{j \oplus 2^s}^{(q)}$  with  $j_s = q_s$  from processor  $q \oplus 2^s$ 

```

Parallel transpose algorithm 2

Using the same timing model as before we get

$$t = 2r_c^{-1} \log_2(p)(n_{1/2} + 0.5nm/p).$$

For two processors this gives the same as for the first algorithm. For more than two processors the first algorithm is faster for large problem sizes and slower for small ones. The crossover point is at $nm = \kappa n_{1/2}$ and

$$\kappa = \frac{2p^2(p-1-\log_2 p)}{(\log_2 p - 2)p + 2}.$$

Thus in particular for smaller numbers of processors the first algorithm is expected to perform better for the size of transforms which are of interest to most users of a machine of this size. Note, however, that for very large numbers of processors the second algorithm could be favored. The second algorithm has the same order of communication overhead as Cooley-Tukey variants [10]. The ratio of the times needed is for very large problem sizes approximately $0.5 \log_2(p) * p/(p-1)$. Thus the difference should not be too large for small numbers of processors. As we are

mainly interested in a moderate number of processors and very large problem sizes we have only implemented the first algorithm so far.

2.3. Three dimensional FFTs. By concatenating all the columns of the two dimensional array X one obtains one large one dimensional array x . The two dimensional Fourier transform matrix for this array is the Kronecker product $F_n \otimes F_m$. Furthermore splitting becomes

$$\begin{aligned} z &:= (I_n \otimes F_m)x \\ y &:= (F_n \otimes I_m)z. \end{aligned}$$

This is just an algorithmic formulation of the factorization $F_n \otimes F_m = (F_n \otimes I_m)(I_n \otimes F_m)$. This factorization for the three dimensional FFT is

$$F_n \otimes F_m \otimes F_r = (F_n \otimes I_{mr})(I_n \otimes F_m \otimes I_r)(I_{nm} \otimes F_r).$$

We will use a different approach. The key is to represent three dimensional data as a two dimensional array as in Figure 2. More generally if ξ_{ijl} are the components

$$X = \begin{array}{|ccc|ccc|} \hline \xi_{000} & \xi_{010} & \xi_{020} & \xi_{001} & \xi_{011} & \xi_{021} \\ \xi_{100} & \xi_{110} & \xi_{120} & \xi_{101} & \xi_{111} & \xi_{121} \\ \xi_{200} & \xi_{210} & \xi_{220} & \xi_{201} & \xi_{211} & \xi_{221} \\ \hline \end{array}$$

FIGURE 2. A 3D array packed into a 2D array.

of the three dimensional array then we define

$$X := [[\xi_{ij0}], [\xi_{ij1}], \dots, [\xi_{ijr}]].$$

Each bracket $[\xi_{ijl}]$ denotes a n by m submatrix.

The threedimensional transform is in this notation again a double matrix-matrix product:

$$Y := F_n X (F_m \otimes F_r).$$

The *split transpose algorithm* for three dimensional data which overwrites the data X with the transformed data is

$$\boxed{\begin{array}{lcl} X & := & X(I_m \otimes F_r) \\ X & := & X(F_m \otimes I_r) \\ Y & := & X^T \\ Y & := & Y F_n \\ X & := & Y^T. \end{array}} \quad \text{Split transpose algorithm for 3D FFT}$$

The same remarks as we made for the two dimensional case apply here as well. Furthermore, higher dimensional transforms can be treated in a similar fashion. Our implementation on a 4 processor VPP 500 required 0.92 sec for a 256 by 256 by 256 transform which corresponds to 2.2 Gflop/s.

2.4. The one dimensional parallel FFT. We assume that the order of the transform is nm and n and m are of the same order of magnitude. Such factors n and m can certainly be found in the mixed-radix case with radices 2,3 and 5. The one dimensional data is arranged columnwise in a two dimensional array as in

$$X = \begin{bmatrix} \xi_0 & \xi_3 & \xi_6 & \xi_9 \\ \xi_1 & \xi_4 & \xi_7 & \xi_{10} \\ \xi_2 & \xi_5 & \xi_8 & \xi_{11} \end{bmatrix}$$

Now the split-transform algorithm which is often called *4-step algorithm* [15, 10] for one dimensional data is

X	$:=$	XF_m
X	$:=$	$X \circ F_{nm}(0 : n-1, 0 : m-1)$
Y	$:=$	X^T
Y	$:=$	YF_n .

Split transpose algorithm for 1D FFT

We use the same data distribution as in the two and three dimensional case. Note that in contrast to the higher dimensional FFTs the data is overwritten by intermediate results but the final transform is stored in a different array. Note that only half the amount of communication is needed compared with the higher dimensional cases.

The second step of the one dimensional algorithm is not found in the higher dimensional cases. It is often called *twiddle factor multiply* [10] and is the Hadamard or element wise product of X with a sub-matrix of the Fourier transform matrix F_{nm} . Although this step does not need communication it displays some difficulties which are specific for parallel algorithms. On processor q it consists of the Hadamard product

$$X^{(q)} := F_{nm}(kn/p : (k+1)n/p - 1, 0 : m-1) \circ X^{(q)}.$$

Each processor needs to compute nm/p roots of 1. Although this amounts to the same number of roots of 1 as for the case of the one processor algorithm much less symmetries can be used which affects both performance and precision. Note that the main source of numerical error in FFTs is in the computation of these roots of 1. However, the roots of 1 can be computed recursively because of

$$F_{nm}(:, j+1) = F_{nm}(:, 1) \circ F(:, j).$$

Thus only one complex multiplication is needed for each entry after $F(:, 1)$ was computed initially using trigonometric functions. In order to reduce rounding errors it is advisable to recompute $F(:, j)$ for $j = b, 2b, 3b, \dots$ for some $b > 1$, e.g., $b = 10$. This gives a good compromise between precision and performance.

Due to less communication the one dimensional transform needs less time than a two dimensional transform of the same size. In Table 2 we give the performance of our algorithms on the VPP 500 for the transform size $n = 4M = 2^{22}$. The performance degradation as a function of the problem size is unavoidable. However, as the split transpose algorithm is scalable [4] for the crossbar switch network, the performance can be maintained if large enough problem sizes are considered. Note that the memory capacity of one processor on the VPP 500 is up to 256 MByte per processor. Thus a transform of size 4M is relatively small on an 11 processor configuration.

Number of processors	1	2	3	4	5	6	7	8	9	10	11
Elapsed time (sec)	0.58	0.34	0.25	0.19	0.15	0.13	0.12	0.10	0.095	0.085	0.08
Performance (Gflop/s)	0.8	1.3	1.9	2.4	3.0	3.5	4.0	4.5	4.9	5.4	5.8
Percentage of Peak	50	41	40	38	38	36	36	35	34	34	33

TABLE 2. Performance of the 1D complex split-transpose algorithm for various numbers of processors and problem size $n = 2^{22} \approx 4 * 10^6$.

3. REAL DATA

3.1. Preliminaries. In this section we recall some facts which underlay many algorithms for real Fourier transforms including our own. First we show that the first half of the components of a transform of real data determine the transform. Second we show how two real FFTs can be done for the cost (time and memory) of one complex FFT.

In component notation the one dimensional Fourier transform is

$$\eta_k = \sum_{j=0}^{n-1} \omega_n^{kj} \xi_j, \quad k = 0, \dots, n-1.$$

If the data ξ_j are real the conjugate complex is

$$\bar{\eta}_k = \sum_{j=0}^{n-1} \omega_n^{-kj} \xi_j = \sum_{j=0}^{n-1} \omega_n^{(n-k)j} \xi_j = \begin{cases} \eta_0, & \text{if } k = 0 \\ \eta_{n-k}, & \text{if } k \neq 0. \end{cases}$$

Arrays with this symmetry are called *hermitian* [10]. So it suffices to compute the components η_k for $k = 0, \dots, n/2$ and the remaining components are obtained from

$$\eta_k = \bar{\eta}_{n-k}, \quad k = n/2 + 1, \dots, n-1.$$

We will use this type of argument for all our algorithms.

Assume that two real arrays $x^{(1)}$ and $x^{(2)}$ are given and their Fourier transforms are required. Then these arrays are uniquely represented by one complex array $x = x^{(1)} + ix^{(2)}$. The Fourier transform of this array is given by

$$\eta_k = \sum_{j=0}^{n-1} \omega_n^{kj} (\xi_j^{(1)} + i\xi_j^{(2)}), \quad k = 0, \dots, n-1.$$

The conjugate complex of this is

$$\bar{\eta}_k = \sum_{j=0}^{n-1} \omega_n^{(n-k)j} (\xi_j^{(1)} - i\xi_j^{(2)}), \quad k = 0, \dots, n-1$$

or, with $\eta_n := \eta_0$:

$$\bar{\eta}_{n-k} = \sum_{j=0}^{n-1} \omega_n^{kj} (\xi_j^{(1)} - i\xi_j^{(2)}), \quad k = 0, \dots, n-1$$

From this we get

$$(3.1) \quad \frac{1}{2}(\eta_k + \bar{\eta}_{n-k}) = \sum_{j=0}^{n-1} \omega_n^{kj} \xi_j^{(1)} = \eta_k^{(1)}, \quad k = 0, \dots, n-1$$

and

$$(3.2) \quad \frac{1}{2i}(\eta_k - \bar{\eta}_{n-k}) = \sum_{j=0}^{n-1} \omega_n^{kj} \xi_j^{(2)} = \eta_k^{(2)}, \quad k = 0, \dots, n-1$$

So $y^{(1)} = F_n x^{(1)}$ and $y^{(2)} = F_n x^{(2)}$ are obtained by doing one complex Fourier transform $y := F_n x$ and then applying formulas 3.1 and 3.2.

3.2. Two dimensional real FFTs. Let $X = [\xi_{ij}]$ be the two dimensional data array as in the previous sections. The real split transpose algorithm computes

$$Y := F_n X F_m(:, 0 : m/2).$$

For simplicity we will assume here that n and m are divisible by $2p$. The remaining elements of $F_n X F_m$ are then computed using a similar argument as in the previous section. The components of $F_n X F_m$ are

$$\eta_{kh} = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \omega_n^{ki} \omega_m^{hj} \xi_{ij}.$$

For real data ξ_{ij} we get

$$\bar{\eta}_{kh} = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \omega_n^{(n-k)i} \omega_m^{(m-h)j} \xi_{ij} = \eta_{n-k, m-h}.$$

Thus the remaining elements are obtained from

$$\eta_{k,h} = \bar{\eta}_{n-k, m-h}, \quad h = m/2 + 1, \dots, m-1.$$

The basic algorithm for the complex Fourier transform of two dimensional real data is:

$\begin{aligned} Y &:= X F_m(:, 0 : m/2) \\ U &:= Y^T \\ U &:= U F_n \\ Y &:= U^T. \end{aligned}$	<i>Real 2D split transpose algorithm</i>
---	--

We see immediately that only half the communication is required compared with the case of complex data. Furthermore the third step requires half the arithmetic operations. It can also be seen that only half the storage space is needed, which is not obvious because seemingly many auxilliary arrays are used.

The first step also needs only half the operations. We will show this in the following. The first step transforms the rows of X . The key idea is to combine this rows in pairs and then use the idea of the previous section. Thus in a first stage (which does not involve communication nor computation) the array X is *complexified* and this is on each processor:

$$\tilde{X}^{(q)} := [I_{\frac{n}{2p}}, iI_{\frac{n}{2p}}] X^{(q)}$$

Then the complex transform of this array is done:

$$\tilde{X}^{(q)} := \tilde{X}^{(q)} F_m$$

This step requires half the number of floating point operations as the first step of the 2D complex FFT. However, a small extra number of additions are needed to get the elements of $Y^{(q)} = X^{(q)} F_m$ by

$$\begin{aligned}\eta_{k,h}^{(q)} &= \frac{1}{2}(\tilde{\eta}_{k,h}^{(q)} + \tilde{\eta}_{k,m-h}^{(q)}), \quad k = 0, \dots, \frac{n}{2p} - 1 \\ \eta_{k+\frac{n}{2p},h}^{(q)} &= \frac{1}{2i}(\tilde{\eta}_{k,h}^{(q)} - \tilde{\eta}_{k,m-h}^{(q)}), \quad k = 0, \dots, \frac{n}{2p} - 1.\end{aligned}$$

Our implementation on a four processor VPP 500 needs 0.113 seconds for a 2048 by 2048 transform of real data which is almost exactly half the time needed for the corresponding complex transform. Floating point performance for real transforms is

$$r_{\text{real FFT}} := \frac{t}{2.5n \log_2(n)}$$

and is 2.0 Gflop/s in this case.

3.3. Three dimensional real FFTs. As for complex data we write three dimensional data as a two dimensional array X . The real split transpose algorithm computes

$$Y := F_n X (F_m \otimes F_r(:, 0 : r/2)).$$

The remaining elements of $F_n X (F_m \otimes F_k)$ are then computed in a similar way as in the 2D case as

$$\eta_{k,h,g} = \bar{\eta}_{n-k,m-h,r-g}, \quad g = r/2 + 1, \dots, r - 1.$$

The basic algorithm for the complex Fourier transform of three dimensional real data is:

$\begin{aligned} Y &:= X (I_m \otimes F_r(:, 0 : r/2)) \\ Y &:= Y (F_m \otimes I_r) \\ U &:= Y^T \\ U &:= U F_n \\ X &:= U^T. \end{aligned}$	<i>Split transpose algorithm for real 3D FFT</i>
--	--

All the steps have been previously discussed. The time for a 256 by 256 by 256 transform on the VPP 500 is 0.454 seconds, corresponding to 2.2 Gflop/s. Thus for real data the time is halved compared to the case of complex data.

3.4. The one dimensional real FFT. As in the complex case we represent our data as two dimensional arrays. The transform of our real $X \in \mathbb{R}^{n \times m}$ is a complex $Y \in \mathbb{C}^{m \times n}$. The one dimensional split transpose algorithm computes the upper half $U = Y(0 : m/2, :)$. The one dimensional Fourier transform using two dimensional arrays is in components

$$\eta_{h,k} = \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \omega_m^{kj} \omega_{mn}^{hj} \omega_n^{hi} \xi_{ji}.$$

The conjugate complex is

$$\bar{\eta}_{h,k} = \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \omega_m^{(n-k-1)j} \omega_{mn}^{(m-h)j} \omega_n^{(m-h)i} \xi_{ji} = \eta_{m-k-1, n-h}.$$

From this the remaining components are computed with

$$\eta_{hk} = \bar{\eta}_{m-k-1, n-h}, \quad k = m/2 + 1, \dots, m-1.$$

The split transpose algorithm for 1D data is

$$\begin{array}{l} Z := XF_m(:, 0 : m/2) \\ Z := Z \circ F_{nm}(0 : n-1, 0 : m-1) \\ U := Z^T \\ U := UF_n. \end{array}$$

Split transpose algorithm for 1D real FFT

Note that the the same transforms can be done with ω_n replaced by $\bar{\omega}_n$ which gives the inverse transform of real data.

In Table 3 we display the performance of some real 1D transforms for a transform of size $2^{10} = 1M$. As we have looked at a 4 times smaller transform the performance

Number of processors	1	2	3	4
Elapsed time (sec)	0.145	0.085	0.067	0.050
Performance (Gflop/s)	0.7	1.2	1.5	2.1
Percentage of Peak	44	38	31	33

TABLE 3. Performance of the 1D real split transpose algorithm for various numbers of processors and problem size $n = 2^{20} \approx 10^6$.

is a bit lower compared to our other measurements. For $n = 4M$ our algorithm requires 0.179 seconds to complete on a 4 processor VPP 500 which corresponds to 2.2 Gflop/s. Note that in the case of 3 processors the processor number does not divide the problem size and so some extra overhead is introduced.

3.5. Transform of hermitian arrays. In this case the data is hermitian and the resulting array is real. Such transforms are used for inversion of the previously discussed transforms of real data. The algorithms are obtained by reversing the sequence of the steps in the original algorithms and inverting every step. For the two-dimensional transform the data is $Y \in \mathbb{C}^{n \times (\frac{m}{2}+1)}$ and the result is $X \in \mathbb{R}^{n \times m}$.

The split transpose algorithm is in this case:

$$\begin{array}{l} Z := Y^T \\ Z := ZF_n \\ Y := Z^T \\ \text{Find } X \in \mathbb{R}^{n \times m} \text{ such that } X\bar{F}_m(:, 0 : m/2) = mY. \end{array}$$

Hermitian 2D split transpose algorithm

The first three steps are will known. The last step, however, is unlike any other step we considered so far. Here a real array X has to be found such that Y is the first half of its Fourier transform.

This step is different from the previous steps but can also be done with a multiple complex Fourier transform such that the overall operation count is nearly half of what would be needed for the corresponding step for general complex data.

Let $\eta_{ik}^{(q)}$ be the components of $Y^{(q)} \in \mathbb{C}^{\frac{n}{p} \times (\frac{m}{2}+1)}$, the sub-block which resides on processor q . Each row contains the first $\frac{m}{2} + 1$ elements of a hermitian array. If we extend $Y^{(k)}$ to a hermitian array $Y_{\text{ext}}^{(q)}$ by

$$\eta_{ik}^{(q)} = \overline{\eta}_{i, m-k}^{(q)}, \quad k = m/2 + 1, \dots, m - 1$$

we obtain $X^{(q)}$ by

$$X^{(q)} = Y_{\text{ext}}^{(q)} F_m.$$

This, however, would require the same storage and floating point operations as the corresponding step for the complex data transform. To avoid this we store $Y_{\text{ext}}^{(q)}$ in “compressed format” as

$$Z^{(q)} = \begin{bmatrix} I_{\frac{n}{2p}}, iI_{\frac{n}{2p}} \end{bmatrix} Y_{\text{ext}}^{(q)}.$$

Then the last step of the 2D split transpose algorithm becomes

$$\begin{aligned} Z^{(q)} &:= Z^{(q)} F_m \\ X^{(q)} &:= \begin{bmatrix} \text{Re } Z^{(q)} \\ \text{Im } Z^{(q)} \end{bmatrix} \end{aligned}$$

because of the linearity of the Fourier transform. As $Z^{(q)} \in \mathbb{C}^{\frac{n}{2p} \times m}$ these steps need only half the resources of the corresponding steps for complex data.

The performance of the transforms of hermitian data is very similar to the performance of the transforms of real data. A 2048 by 2048 transform took on 4 processors of the VPP 500 0.113 seconds, which corresponds to 2.0 Gflop/s.

The three dimensional and one dimensional transforms of hermitian data are developed from the corresponding complex algorithms in the same way as the 2D transform. For completeness we will display these algorithms here.

$\begin{aligned} U &:= X^T \\ U &:= UF_n \\ Y &:= U^T \\ Y &:= Y(F_m \otimes I_r) \\ \text{Find } X &\in \mathbb{R}^{n \times mr} \text{ such that } X(I_m \otimes \overline{F}_r(:, 0 : r/2)) = rY. \end{aligned}$

Split transpose algorithm for hermitian 3D FFT

A transform of size 256 by 256 by 256 on a 4 processor VPP 500 needs 0.454 seconds, which corresponds to 2.2 Gflop/s.

$\begin{aligned} U &:= UF_n. \\ Z &:= U^T \\ Z &:= Z \circ F_{nm}(0 : n - 1, 0 : m - 1) \\ \text{Find } X &\in \mathbb{R}^{n \times m} \text{ such that } X\overline{F}_m(:, 0 : m/2)) = mY. \end{aligned}$

Split transpose algorithm for 1D hermitian FFT

Our implementation needs for a transform of size $n = 4M$ 0.103 seconds to complete on a 4 processor VPP 500 which corresponds to 2.2 Gflop/s.

4. CONCLUSIONS

The split transpose algorithm is a very efficient algorithm for real and complex one, two and three dimensional data for distributed memory computers with high connectivity and bandwidth like the VPP 500. It is furthermore relatively simple to implement as it only requires a few extra non-local steps.

ACKNOWLEDGEMENTS

The work presented here is part of the Area 4 joint project of ANU and Fujitsu Japan. The resulting software will be included in Fujitsu's SSLII Scientific Subroutine Library.

I would like to thank Mr. Makoto Nakanishi of Fujitsu Ltd Japan for several suggestions and also support for the timings which were run on a VPP 500 in a factory of Fujitsu Ltd Japan. Ross Nobes of ANU did the timings for more than 4 processors. Judy Jenkinson of ANU did extensive testing of the codes on the AP1000 in Canberra and later on the VPP 500 in Japan. The timings on the CM-5 were contributed by David Singleton, ANU. The developments of the codes was done on the Fujitsu AP1000 at ANU using a port of the Fujitsu VPP 500 Fortran compiler to the AP1000.

REFERENCES

1. C. Calvin and F. Desprez, *Minimizing communication overhead using pipelining for multi-dimensional FFT on distributed memory machines*, ParCo 93 Grenoble, 1993.
2. J.W. Cooley and J.W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp. **19** (1965), 297–301.
3. J.O. Eklundh, *Efficient matrix transposition*, Two-Dimensional Digital Signal Processing II (T.S. Huang, ed.), Springer-Verlag, 1981, pp. 9–35.
4. Anshul Gupta and Vipin Kumar, *The scalability of FFT on parallel computers*, IEEE Transactions on Parallel and Distributed Systems **4** (1992), no. 7.
5. Markus Hegland, *Block algorithms for FFTs on vector and parallel computers*, Proceedings of the Parallel Computing Conference ParCo 93, 1993.
6. ———, *An implementation of multiple and multi-variate Fourier transforms on vector processors*, accepted for publication in SIAM J. Sci. Comp., 1994.
7. ———, *A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing*, accepted for publication in Numerische Mathematik, 1994.
8. S. Lennart Johnsson and Robert L. Krawitz, *Cooley-Tukey FFT on the Connection Machine*, Parallel Computing **18** (1992), 1201–1221.
9. D.G. Korn and J.J. Lambiotte, *Computing the fast Fourier transform*, Math. Comp. **33** (1979), 977–992.
10. Charles Van Loan, *Computational frameworks for the fast Fourier transform*, SIAM, 1992.
11. M.C. Pease, *An adaptation of the fast Fourier transform for parallel processing*, J. Assoc. Comput. Mach. **15** (1968), 252–264.
12. Paul N. Swarztrauber, *Vectorizing the FFTs*, Parallel Computations (New York) (G. Rodrigue, ed.), Academic Press, New York, 1982.
13. ———, *FFT algorithms for vector computers*, Parallel Computing **1** (1984), 45–63.
14. ———, *Symmetric FFTs*, Math. Comp. **47** (1986), 323–346.
15. ———, *Multiprocessor FFTs*, Parallel Comput. **5** (1987), 197–210.
16. Clive Temperton, *Fast methods on parallel and vector machines*, Comput. Phys. Comm. **26** (1982), 331–334.
17. Aad J. van der Steen, *An overview of (almost) available parallel systems*, Tech. report, Stichting Nationale Computer Faciliteiten, The Netherlands, 1993.