Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1993

A Distributed Memory Algorithm for 3-D FFT

Calin r. Costian

Dan C. Marinescu

Report Number:
93-057

Costian, Calin r. and Marinescu, Dan C., "A Distributed Memory Algorithm for 3-D FFT" (1993). *Department of Computer Science Technical Reports.* Paper 1071.
https://docs.lib.purdue.edu/cstech/1071

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

A DISTRIBUTED MEMORY ALGORITHM FOR 3-D FFT

Calin R. Costian
Dan C. Marinescu

CSD-TR-93-057

September 1993

A Distributed Memory Algorithm For 3-D FFT*

Calin R. Costian and Dan C. Marinescu
Department of Computer Sciences
Purdue University, IN 47906

September 2, 1993

Abstract. The present paper describes an algorithm for the parallel in-place 3-D FFTs on distributed memory architectures. The calculation is accomplished by partitioning the 3-dimensional input tensor into subtensors (one subtensor per node) using a data partitioning strategy to minimize the communication between the nodes. The dimensions of the tensor need not be powers of 2, although some requirements related to the number of nodes need to be fulfilled. The computation of local FFTs is carried out using Divide-and-Conquer method and results are combined to obtain the final result. An implementation of the algorithm was tested on an iPSC/860 system and on a 2-D mesh, the Touchstone Delta system.

1 Introduction

Since the discovery made by Cooley and Tukey in 1965 about the possibility of drastically reducing the time complexity of the computation of discrete Fourier transforms from quadratic to $\mathcal{O}(n \log n)$ time, a true revolution in the computational techniques has taken place, based on convolution and the direct and inverse Fourier transforms. This has triggered a vast amount of literature on the subject of sequential and parallel Fast Fourier Transform – algorithms for computing Fourier transforms in an efficient way [1], [5].

Multidimensional FFTs have also been extensively considered due to their applications in physics, crystallography, for a variety of differential equations, and in numerous other fields. For instance, the structure analysis of crystals used in modern X-ray crystallography is concerned with solving the phase problem (determining the phases of structure factors starting with their observed amplitudes) the results of which are used in the computation of electron density maps in order to locate the positions of the atoms in the crystal unit cell. In [4], an iterative method for using this phase refinement problem is considered, in which

*Work supported in part by the NSF under grants 9119388 and BIR-9301210.

the Fourier transforms play an essential role, relating the phases from the electron densities: starting with the measured amplitudes and an initial guess of the phases, a set of electron densities is computed (through an inverse FFT performed on the structure factors from the reciprocal space), then follows an averaging process based on the symmetry properties of the crystal, the new electron densities are transformed back into structure factors through a direct FFT, finally the phases are retained but the amplitudes are replaced with the intial (measured) set of values, and the whole process is repeated.

The dimensions of the tensors on which the FFTs are to be performed in this example are of up to 10^9 elements, which with the existing sequential FFT algorithms would take an excessively long time for even one iteration. Therefore, there appears the need to use parallel computing systems to solve both the time and space problem. Distributed memory MIMD systems, which constitute a widely available class of machines nowadays, are considered in this paper for the computation of 3-D FFTs.

A wide variety of parallel FFT algorithms for MIMD, SIMD, shared and distributed memory systems have been developed, most of them originating from the Divide-and-Conquer technique suggested by the Cooley-Tukey algorithm (oftentimes going through the computation of 1-D or 2-D FFTs and then performing their synthesis). The proposed algorithm does not take the approach of combining 1-D and 2-D FFTs, but it uses instead the Divide-and-Conquer principle for all three dimensions simultaneously. It consists of two phases – the local FFT computation within every node, with no inter-node communication and the combining phase, in which the 3-D Divide-and-Conquer technique is used as well.

Another feature which differentiates the proposed algorithm from many other implementations is the data partitioning strategy. Usually, in the multidimensional FFT algorithms the data is partitioned in a contiguous manner (in "slabs", or "slices") whereas in our case the partitioning has more the geometric appearance of a regular three-dimensional lattice, which allows the use of the Divide-and-Conquer method with no preferential dimension. Also, due to this partitioning strategy, the Divide-and-Conquer method has been applied in a slightly different way, involving the need for defining a "generalized" Fourier transform (of which the ordinary transform is a particular case), for which the necessary recurrence relationships have been derived (see next section).

The next section describes the underlying concepts of the algorithm, followed by experimental results and a comparative analysis of the algorithm.

2 Theoretical Framework

2.1 Notations and Conventions

Let the dimensions of the input tensor X be n_0, n_1, n_2 (not necessarily powers of 2). If we denote by $\mathcal{F}_{n_0,n_1,n_2}(X)$ the Fourier Transform of X (which is a tensor of same dimensions) and by $\mathcal{F}_{n_0,n_1,n_2}(X)_{g_0,g_1,g_2}$ the (g_0, g_1, g_2)-th component of this tensor, then by definition:

$$\mathcal{F}_{n_0,n_1,n_2}(X)_{k_0,k_1,k_2} = \sum_{r_0=0}^{n_0-1} \sum_{r_1=0}^{n_1-1} \sum_{r_2=0}^{n_2-1} \omega_{n_0}^{k_0 r_0} \omega_{n_1}^{k_1 r_1} \omega_{n_2}^{k_2 r_2} X_{r_0,r_1,r_2} \tag{1}$$

The indices of X are always counted beginning with 0, and:

$$\omega_n = \cos\frac{2\pi}{n} + i \sin\frac{2\pi}{n}, \ (\forall) n \in N \tag{2}$$

To simplify the notation, we will replace in formulas the triples (and triple sums as well) by letters (and single sums), mentioning explicitly which of the letters designate triples (also, we will assume that all triples consist of nonnegative integers).

For instance, if n and r are triples, then the notation

$$\sum_{r=0}^{n-1}$$

stands for

$$\sum_{r_0=0}^{n_0-1} \sum_{r_1=0}^{n_1-1} \sum_{r_2=0}^{n_2-1}$$

and the notation $r < n$ means

$$r_0 < n_0 \text{ and } r_1 < n_1 \text{ and } r_2 < n_2$$

Also, if x and t are triples, then x^t means $x_0^{t_0} x_1^{t_1} x_2^{t_2}$.

With this notation, formula (1) becomes:

$$\mathcal{F}_n(X)_k = \sum_{r=0}^{n-1} \omega_n^{kr} X_r \tag{3}$$

where n, k, r and ω are triples.

Assumption on the Tensor Dimensions. Let P the number of available nodes. We make the assumption that:

$$P \text{ can be decomposed as a product } p_0 p_1 p_2 \text{ such that } p_i^2 n_i, \ i = 0, 1, 2 \tag{4}$$

where the notation "x|y" means "x divides y". Based on this assumption, we can define the following triples:

$$q = n/p \ \ (\text{i.e. } q_i = n_i/p_i,\ i = 0,1,2) \tag{5}$$

$$s = n/p^2 = q/p \ \ (\text{i.e. } s_i = n_i/p_i^2,\ i = 0,1,2) \tag{6}$$

Observation. If P is a power of 2 (as is the case with a hypercube) and the dimensions of X are also powers of 2, then the above condition is automatically satisfied (provided that $P^2 \leq N$, where $N = n_0 n_1 n_2$, that is, the tensor is large enough compared to the number of nodes).

2.2 Data Partitioning and the Outline of the Algorithm

The following relationship is derived in Appendix A:

$$\mathcal{F}_n(X)_k = \sum_{u=0}^{p-1} \omega_n^{uk} \mathcal{F}_q(X^u)_{\mathrm{mod}(k,q)} \tag{7}$$

where:

k and u are triples;
$\mathrm{mod}(k,q) = k \bmod q = (k_0 \bmod q_0,\ k_1 \bmod q_1,\ k_2 \bmod q_2)$;
X^u $(u < p)$ is a subtensor of dimensions q defined thus:

$$X^{u_0 u_1 u_2}_{a_0 a_1 a_2} = X_{a_0 p_0 + u_0, a_1 p_1 + u_1, a_2 p_2 + u_2},\ \ (\forall) a < q \tag{8}$$

For instance, if $p_0 = p_1 = 2$ and $p_2 = 1$ (so we have $2 \times 2 \times 1 = 4$ nodes) then we will have 4 subtensors: X^{000}, X^{010}, X^{100} and X^{110}. The subtensor X^{010} for example consists of all the elements of the "global" tensor X which have the first index even and the second one odd (the third one could be anything, since $p_2 = 1$).

Another example: if $p_0 = p_1 = 2$ and $p_2 = 4$ (16 nodes) then the subtensor X^{013} consists of all elements of X having the first index even, the second odd, and the third giving the remainder 3 when divided by 4. The "first" element of this subtensor is:

$$X^{013}_{000} = X_{013}$$

We call a_0, a_1, a_2 the **local coordinates** of an arbitrary element with respect to the subtensor X^{u_0,u_1,u_2}, and $a_0 p_0 + u_0, a_1 p_1 + u_1, a_2 p_2 + u_2$ the **global coordinates** of the same element. Thus, 7 gives the relationship between the local and global values of the same element.

The idea is to partition the input tensor X in such a way that each node stores one of the P subtensors. In order to be clear which node holds which subtensor, we renumber the nodes by assigning them triples, in the following way: node i will be called (u_0, u_1, u_2), where

$$i = u_0 p_1 p_2 + u_1 p_2 + u_2 \tag{9}$$

for any triple $u < p$ (note that $u < p$ is equivalent to $i < P$, which is what we want).

This data distribution takes place in Phase 0, in case the tensor wasn't already distributed in this fashion among the nodes. Then, in Phase 1 each node performs a local 3-D FFT and no communication among the nodes is necessary. In this phase node (u_0, u_1, u_2) computes $\mathcal{F}_q(X^u)$. Finally, Phase 2 combines the local FFTs into the global one using (7).

Phase 1: Computation of local FFTs.

Each node computes the FFT of its own subtensor, independently of the other nodes. This is done in parallel by all nodes with no internode communication. The subtensors stored in each node have dimensions $q_0 \times q_1 \times q_2$ (recall that $q = n/p$).

Suppose we have decomposed $q = s_1 t_1$ (s_1, t_1 are **triples**). Then the following relation can be derived (see Appendix B):

$$\mathcal{F}_q(X)_c = \sum_{a=0}^{t_1-1} \omega_q^{ac} \mathcal{F}_{s_1}^{t_1,a}(X)_{\mathrm{mod}(c,s_1)},\ (\forall) c < q \tag{10}$$

where a, c are triples. The "generalized" Fourier transform is defined as:

$$\mathcal{F}_d^{\alpha,\beta}(X)_f = \sum_{m=0}^{d-1} \omega_d^{mf} X_{\alpha m+\beta},\ (\forall) f < d \tag{11}$$

where d, f, α, β and m are triples (d is the dimension of the tensor X to which this "generalized" Fourier transform is applied). We see that the ordinary Fourier transform is obtained from the generalized one when $\alpha_i = 1$, $\beta_i = 0$, $i = 0, 1, 2$.

The recurrence relationship for the generalized Fourier transform is derived in Appendix C:

$$\mathcal{F}_d^{\alpha,\beta}(X)_c = \sum_{m=0}^{t_2-1} \omega_d^{cm} \mathcal{F}_{s_2}^{\alpha t_2, \alpha m+\beta}(X)_{\mathrm{mod}(c,s_2)},\ (\forall) c < d \tag{12}$$

where c, d, α, β, m, s_2, t_2 are triples such that $d = s_2 t_2$.

Note. For each $f < d$, $\mathcal{F}_d^{\alpha,\beta}(X)_f$ represents the transform of the element $X_{\alpha f+\beta}$, and in the present implementation it is stored in the place with indices $\alpha f + \beta$.

The implementation of the generalized FFT, which is an in-place transformation, is carried out with a classical Divide-and-Conquer method: we start with the triple q and decompose it into products $s_2 t_2$, applying the above recurrence relation, till we end up with a triple consisting of prime numbers, when we simply compute the FFT by summation. We have two arrays of the same dimensions $q_0 \times q_1 \times q_2$, and at each step we take the input values (of the subtensor to be transformed) from one of the arrays and calculate the transform in the corresponding subtensor of the other array, which serves as a buffer.

Phase 2: Combining the local FFTs together.

The combination of the local FFTs into the global one is based on the following relationship derived in Appendix A:

$$\mathcal{F}_n(X)_g = \sum_{u=0}^{p-1} \omega_n^{gu} \mathcal{F}_q(X^u)_{\mathrm{mod}(g,q)} \tag{13}$$

Here the triple $g < n$ represents any global indices; recall that n are the dimensions of the global tensor X, while $q = n/p$ are the dimensions of the local subtensors X^u stored in each node $u < p$.

In the above formula, fix an arbitrary global triple g for which we want to compute $\mathcal{F}_n(X)_g$. If we let $g = cp + r$ with $c < q$ and $r < p$ then this element belongs to the node denoted by the triple r and the local indices of this element are given by the triple c.

Consider the triple $h < q$ given by $h = \mathrm{mod}(g, q)$. Then (13) can be rewritten as:

$$\mathcal{F}_n(X)_g = \sum_{u=0}^{p-1} \omega_n^{gu} \mathcal{F}_q(X^u)_h \tag{14}$$

To compute the element $\mathcal{F}_n(X)_g$ we need to import the transforms $\mathcal{F}_q(X^u)_h$ from all nodes $u < p$. But if we have all these P values in this node r, then we can compute not only $\mathcal{F}_n(X)_g$, but also any $\mathcal{F}_n(X)_f$ such that $\mathrm{mod}(f, q) = h$ (equivalently, $f = bq + h$, for all triples $b < p$).

In other words, given the above mentioned P values, we can compute P transforms with these values, i.e. all $\mathcal{F}_n(X)_f$ such that $f = bq + h, (\forall) b < p$. The question arises – to which nodes do these $\mathcal{F}_n(X)_f$ elements belong ? It turns out that all of them belong to the same node r, and thus no exporting of values is needed.

Observation. Indeed, $g = cp + r = bq + h = bps + h$ (since $q = ps$ – see (6)), so $h = (c - bs)p + r$, whence $f = bq + h = bps + (c - bs)p + r = cp + r$, and thus all elements of indices $f = bq + h$, $b < p$ belong to node r.

Considering the overall computations that need to be done in this phase, we see that among the $q_0 q_1 q_2$ elements that need to be computed in each node $r < p$, we need to select

a certain number of "basic" elements for which we import what we need, and with the imported values we are able to compute, for each "basic" element, the transforms of other $P = p_0 p_1 p_2$ elements within the same node (so we need exactly $s = q/p$ basic elements in each node).

We choose these basic elements to be those having the local indices of the form $h = dp+r$, $(\forall) d < s$, where r is the triple denoting the current node.

Observation. This choice is correct, in the sense that each of these basic elements will enable the computation of a group of P elements and these groups are all disjoint, so that all elements of the local tensor are computed and none is computed twice. Indeed, let's assume that

$$f_1 = b_1 q + h_1 = f_2 = b_2 q + h_2, \quad b_1, b_2 < p,$$

$$h_1 = d_1 p + r, h_2 = d_2 p + r, \quad d_1, d_2 < s$$

We have $b_1 q + d_1 p + r = b_2 q + d_2 p + r$ or, dividing by p: $b_1 s + d_1 = b_2 s + d_2$. Since $d_1, d_2 < s$, taking both sides modulo s we obtain $d_1 = d_2$ and from here $b_1 = b_2$ and $f_1 = f_2$.

An outline of the algorithm for this computation is given below.

In each node denoted by the triple r do:
for each $d < s$ do {
$h := dp + r$;
import X_h^u *from every node* $u < p$;
for each $b < p$ *do* {
$f := bq + h$;
$c := bs + d$;
$T1[c] = \sum_{u=0}^{p-1} \omega_n^{fu} X_h^u$
}
}

3 Complexity Analysis

This section analyzes the work complexity and the communication complexity of the two phases of the algorithm as a function of N the total number of tensor elements and P the number of nodes.

Work complexity. The amount of work to be done excluding the communication between the nodes is:

- Phase 1 – local FFTs: $\mathcal{O}(\frac{N}{P}\log_2(\frac{N}{P}))$
- Phase 2 – synthesis: $\mathcal{O}(N)$

Communication complexity. Assuming that the time required for sending a stream of n bytes of data, is $\alpha n + \beta$ (with α and β constants depending only on the system used), the communication complexity of the algorithm is $\mathcal{O}(P^2)$ and it is given by

$$(P-1)\alpha N + P(P-1)\beta \qquad (15)$$

Nevertheless, using a pairing strategy by which the communication is done in parallel, the communication time has been reduced on a 2-D mesh to

$$2\alpha N + 4P\beta \qquad (16)$$

and on a hypercube to

$$2(1-\frac{1}{P})\alpha N + 4(P-1)\beta \qquad (17)$$

which are both linear in P (see next section). Moreover, on the hypercube the pairing strategy has been chosen in an optimal way which makes the communication contention-free ([2], [3]).

4 Implementation and Experimental Results

Since each node needs to get some data from every other node and the needed data is not contiguous (rather it is scattered like an equally spaced lattice), the whole tensor is sent from one node to another, to save time.

Each node requires a buffer of the same size as the local tensor for storing the data received from the other nodes. The selected elements are then placed into the actual tensor in such a way that no overwriting of the needed data occurs, and no additional memory is necessary.

Thus, the implementation on the 2-D mesh as well as on the hypercube requires in each node, besides the memory necessary to hold the local tensor, an equal amount of memory for the auxiliary tensor (workspace).

4.1 On a 2-D mesh

The pairing strategy used in order to obtain an all-to-all node communication in parallel (in P steps, instead of $P(P-1)$ steps which would have resulted if the communication was not done in parallel) is based on the following algorithm (here *me* stands for the current node number):

```
for step := 1 to P do {
   partner := (2 * P - step - me) modulo P;   /* inverted circular shifts */
   if partner = me then
      continue;   /* each node stalls exactly once */
   synchronize with partner by sending and receiving a zero-length message
        of type 'step' (to ensure we are in the same step);
   exchange the local tensor with partner;
   extract the needed values from the received tensor;
}
```

The program has been tested on an Intel Delta 2-D mesh, obtaining the speedup curves shown in the graphs below. The speedup has been computed with the following formula (P is the number of nodes):

$$S(P) = \frac{T(1)}{T(P)} \tag{18}$$

4.2 On an iPSC/860 hypercube

Due to the architectural characteristics of the hypercube, a pairing strategy could be achieved by using the XOR function such that the all-to-all communication is achieved in $P-1$ steps and is contention free (see [3]). The underlying algorithm is the following:

```
for step := 1 to P do {
   partner := me XOR step;
   synchronize with partner by sending and receiving a zero-length message
        of type 'step' (to ensure we are in the same step);
   exchange the local tensor with partner;
   extract the needed values from the received tensor;
}
```

The efficiency of the proposed program has been tested on an iPSC/860 hypercube with 64 nodes and compared to an FFT program developed by David Scott and Ed Kushner ([2]), which uses the same pairing strategy to achieve parallel communication.

Figures 1–4 show the speedup as a function of the number of nodes (for two sets of tensor dimensions – namely $32 \times 32 \times 32$ and $64 \times 64 \times 64$), and the speedup as function of the tensor dimensions (for two configurations with 8 and 16 nodes respectively).

From these graphs we see that the proposed algorithm reaches a best speedup for a configuration of 8 nodes (for tensor dimensions $32 \times 32 \times 32$) respectively 16 nodes (for tensor dimensions $64 \times 64 \times 64$), while the program in [2] doesn't reach yet the peak of its speedup, growing in a linear fashion for the hypercube configurations with at most 16 nodes.

The graphs of the speedup as a function of the number of tensor elements show that the speedup increases very quickly for tensors having less than 2^{16} elements, and then is almost stationary for 8 nodes, while for 16 nodes it continues to grow slowly.

A third analysis was made with a fixed number of nodes and fixed number of tensor elements, but with different shapes, and this has shown that the execution time is practically unaffected by the shape of the tensor. The tests have been run in 2 and 8 nodes respectively, for tensors containing 2^{15} elements.

Tables 1 and 2 show a "variation coefficient" which expresses the percentage of the variation of the computing time for different shapes of a tensor with 2^{15} elements, with respect to a "base shape" which has been chosen to be the most balanced one, i.e. $32 \times 32 \times 32$. This variation coefficient has been defined by:

$$v(x, y, z) = \frac{T(x, y, z)}{T(32, 32, 32)} 100 \tag{19}$$

where $T(x, y, z)$ is the execution time for computing the 3-D FFT of tensor having dimensions x, y, z (in our case, x, y, z will be chosen such that $xyz = 2^{15} = 32 \times 32 \times 32$).

From these tables we can see that the variation coefficient for the chosen shapes stays closer to 100 in our implementation than in the program in [2], showing that the shape of the tensor affects very slightly the efficiency of the program.

Dimensions	Algorithm in [2]	Proposed alogorithm
32,32,32	100	100
64,16,32	96	102
64, 8,64	100	98
128,16,16	98	98
256,8,16	106	99

Table 1: Variation coefficient as defined in (19) for 2 nodes

Dimensions	Algorithm in [2]	Proposed algorithm
32,32,32	100	100
64,16,32	96	99
64, 8,64	93	95
128,16,16	92	98
256,8,16	98	95

Table 2: Variation coefficient as defined in (19) for 8 nodes

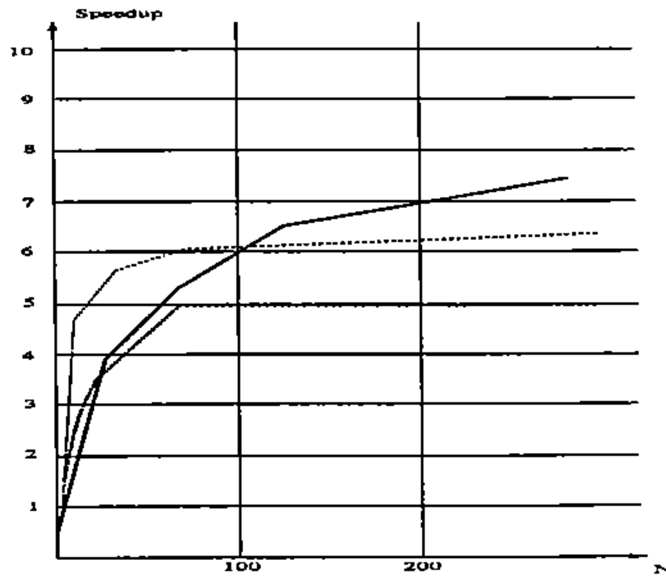

Figure 1. The speedup function of the problem size N (N $\times$ 1000 is the number of tensor elements). 8 PEs are used. The solid curve gives the speedup of the algorithm presented in this paper on a 2-D mesh, the dashed curve the speedup of the same algorithm on a hypercube and the dotted line the speedup exhibited by the algorithm in [2] on a hypercube.

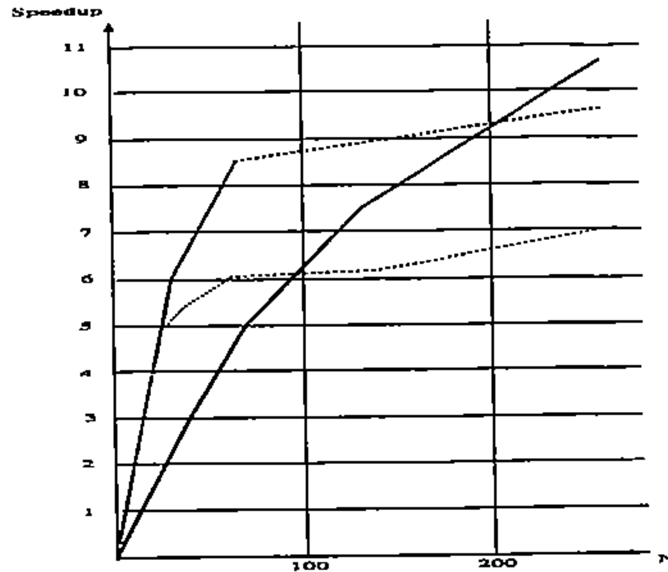

Figure 2. The speedup function of the problem size N (N $\times$ 1000 is the number of tensor elements). 16 PEs are used. The solid curve gives the speedup of the algorithm presented in this paper on a 2-D mesh, the dashed curve the speedup of the same algorithm on a hypercube and the dotted line the speedup exhibited by the algorithm in [2] on a hypercube.

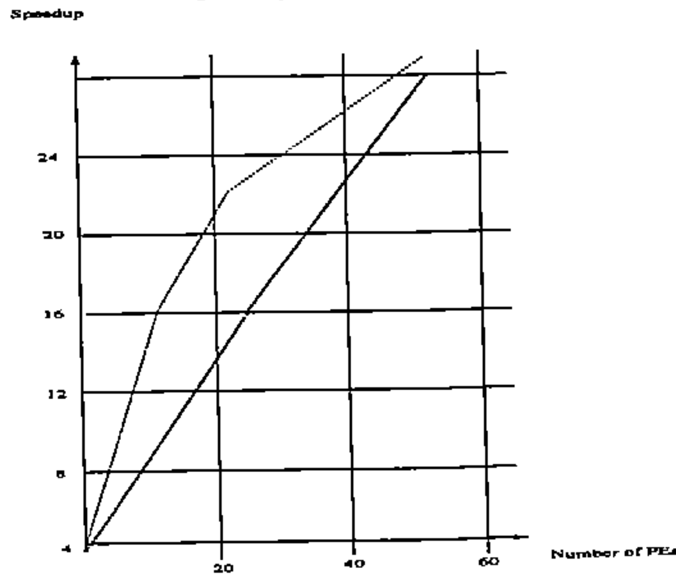

Figure 3. The speedup curves. The dashed curve is for the algorithm presented in this paper and the dotted one is for the algorithm in [2]. The tensor has $32 \times 22 \times 32$ elements and the computation is carried out in a hypercube.

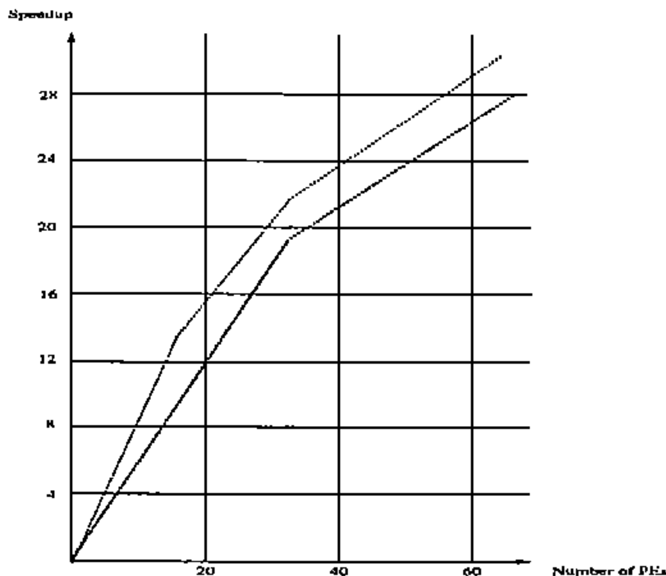

Figure 4. The speedup curves. The dashed curve is for the algorithm presented in this paper, the dotted curve the is for the algorithm in [2]. The tensor has $64 \times 64 \times 64$ elements and computation is carried out on a hypercube.

5 Conclusions

Advantages of the above described algorithm

- Many algorithms for 3-D FFTs are based on the assumption that the three dimensions of the input tensor are powers of 2 [1]. The algorithm proposed in this paper imposes only the less restrictive condition that the number of nodes could be written as a product $p_0 p_1 p_2$ such that $p_i^2 | n_i$, $i = 0, 1, 2$, where n_i are the dimensions of the tensor (this condition is automatically satisfied if everything is a power of two, provided that the tensor dimensions are not too small relatively to the number of nodes).

- All FFT algorithms, including the present one, use at some point a Divide-and-Conquer strategy in order to reduce the time complexity from quadratic to $\mathcal{O}(n \log n)$. But some of them exploit this strategy only when performing the 1-D or 2-D FFTs, while the algorithm described above uses this technique for all three dimensions in a consistent manner. For this reason, in the present algorithm, there is no need for any global transposition.

- Some of the existing (parallel or sequential) 3-D FFT algorithms which use preferentially one (or two) of the three dimensions of the tensor (for instance, partitioning the

tensor into y-slabs and doing 2-D FFTs on the $x \times z$ planes of these slabs). The proposed algorithm does not have any preferred dimension and thus the execution time remains almost constant when varying the shape of the tensor (see Tables 1 and 2 showing the "variation coefficient") – unlike some of the algorithms using preferential dimensions in which the execution time changes considerably with the shape.

Disadvantages of the algorithm

- The amount of memory needed in each node is larger than in some existing algorithms (the workspace has the same size as the tensor, while in some other implementations the computation can be carried out with a workspace smaller than the local tensor). This is due to some extent to the fact that the data is partitioned into "lattices" intead of contiguous "slabs", and thus the whole local tensors must be sent between each pair of nodes.

- Also, due to the data partitioning in "lattices", there might be necessary to spend time on initially distributing the data among the nodes (Phase 0).

Possible improvements to the proposed algorithm

- In the present implementation, the Fourier coefficients are computed within each transform; it could be possible to compute them separately (if there is enough memory available), so that the same coefficients are used for the computation of several FFTs having the same fixed dimensions (since the Fourier coefficients depend only upon the dimensions of the tensor).

- Another possible improvement could be the automatic computation of the factors p_0, p_1, p_2, given P, the number of available nodes, such that $p_0, p_1, p_2 = P$ (in the present implementation, the p factors have to be supplied as input).

References

[1] Karner, H., Ueberhuber, C. W., "Parallel FFT Algorithms: Literature Survey", Technical Report SciPaC/TR 93-2, January 1993.

[2] Kushner, E., "In core 3-D FFT program for iPSC/860", Private communication; also Intel, iPSC/860 Basic Math Library User's Guide (1991).

[3] Marinescu, D. C., Rice, J. R., "Speedup, Communication Complexity and Blocking - A la Recherche du Temps Perdu", *Proc. Int. Parallel Processing Symposium*, IEEE Press, pp 712-721, 1993.

[4] Marinescu, D.C., Rice, J.R., Cornea-Hasegan, M.A., Lynch, R.E., Rossmann, M.G., "Macromolecular Electron Density Averaging on Distributed Memory MIMD Systems", Concurrency: Practice and Experience, 1993 (in press).

[5] Van Loan, Charles, "Computational framework for Fast Fourier Transform," SIAM, 1992.

Appendix A

In this appendix we derive the relationship between the global FFT and the local ones:

$$\mathcal{F}_n(X)_k = \sum_{u=0}^{p-1} \omega_n^{uk} \mathcal{F}_q(X^u)_{\mathrm{mod}(k,q)}$$

where $k < n$, $u < p$ are triples and X^u was defined by:

$$X_a^u = X_{ap+u}, \quad (\forall) a < q$$

For any fixed triple $k < n$, we start with the definition of the Fourier transform:

$$\mathcal{F}_n(X)_k = \sum_{t=0}^{n-1} \omega_n^{kt} X_t$$

where $t < n$ is a triple. We can rewrite the running triple t as $t = ap + u$, where the running triples a and u are such that $a < q$, $u < p$ (since $n = pq$). We obtain:

$$\mathcal{F}_n(X)_k = \sum_{u=0}^{p-1} \sum_{a=0}^{q-1} \omega_n^{k(ap+u)} X_{ap+u} = \sum_{u=0}^{p-1} \omega_n^{ku} \sum_{a=0}^{q-1} \omega_n^{kap} X_a^u$$

Writing $k = cq + r$ with c and r triples such that $r = \mathrm{mod}(k, q)$ and taking into account that $\omega_{xz}^{xy} = \omega_z^y$, we derive:

$$\omega_n^{kap} = \omega_{pq}^{kap} = \omega_q^{ka} = \omega_q^{acq+ar} = \omega_q^{ar} = \omega_q^{a\,\mathrm{mod}(k,q)}$$

and now we can write:

$$\mathcal{F}_n(X)_k = \sum_{u=0}^{p-1} \omega_n^{ku} \sum_{a=0}^{q-1} \omega_q^{a\,\mathrm{mod}(k,q)} X_a^u = \sum_{u=0}^{p-1} \omega_n^{ku} \mathcal{F}_q(X^u)_{\mathrm{mod}(k,q)}$$

since by definition:

$$\mathcal{F}_q(X^u)_b = \sum_{a=0}^{q-1} \omega_q^{ab} X_a^u$$

Appendix B

In this appendix[1] we show how to compute the local FFTs from the generalized Fourier transforms. We prove that if $q = st$ (s, t triples) then

$$\mathcal{F}_q(X)_c = \sum_{a=0}^{t-1} \omega_q^{ac} \mathcal{F}_s^{t,a}(X)_{\mathrm{mod}(c,s)},\ (\forall) c < q$$

where $a < t$ and $c < q$ are triples and the generalized Fourier transform was defined thus:

$$\mathcal{F}_d^{\alpha,\beta}(X)_f = \sum_{m=0}^{d-1} \omega_d^{mf} X_{\alpha m+\beta},\ (\forall) f < d$$

where d, f, α, β and m are triples (d is the dimension of the tensor X to which the generalized Fourier transform is applied).

We start with the definition of the Fourier transform:

$$\mathcal{F}_q(X)_c = \sum_{h=0}^{q-1} \omega_q^{hc} X_h = \sum_{a=0}^{t-1}\sum_{b=0}^{s-1} \omega_q^{btc+ac} X_{bt+a} = \sum_{a=0}^{t-1} \omega_q^{ac} \sum_{b=0}^{s-1} \omega_q^{btc} X_{bt+a}$$

Applying the same technique as in appendix A, we see that

$$\omega_q^{btc} = \omega_p^{btc} s = \omega_s^{bc} = \omega_s^{b\,\mathrm{mod}(c,s)}$$

hence we can write:

$$\mathcal{F}_q(X)_c = \sum_{a=0}^{t-1} \omega_q^{ac} \sum_{b=0}^{s-1} \omega_s^{b\,\mathrm{mod}(c,s)} X_{bt+a} = \sum_{a=0}^{t-1} \omega_q^{ac} \mathcal{F}_s^{t,a}(X)_{\mathrm{mod}(c,s)}$$

since by definition:

$$\mathcal{F}_s^{t,a}(X)_v = \sum_{b=0}^{s-1} \omega_s^{vb} X_{bt+a}$$

Appendix C

[1]In appendix B and C the triple s does <u>not</u> mean q/p, like in all other parts of this paper, but designates an arbitrary triple.

In this appendix we derive the recurrence relationship for generalized Fourier transforms:

$$\mathcal{F}_d^{\alpha,\beta}(X)_c = \sum_{m=0}^{t-1} \omega_d^{cm} \mathcal{F}_s^{\alpha t, \alpha m+\beta}(X)_{\mathrm{mod}(c,s)}$$

where $c < d$, α, β, m, s, t are triples such that $d = st$.

Any triple $b < d$ can be written in the form $b = rt + m$, with $m < t$ and $r < s$. Strting with the definition of the generalized Fourier transform, we can write:

$$\mathcal{F}_d^{\alpha,\beta}(X)_c = \sum_{b=0}^{d-1} \omega_d^{bc} X_{\alpha b+\beta} = \sum_{m=0}^{t-1} \sum_{r=0}^{s-1} \omega_d^{crt+cm} X_{\alpha(rt+m)+\beta}$$

Applying the same rewriting technique as in appendix A, we can write:

$$\omega_d^{crt} = \omega_s^{cr} = \omega_s^{r\,\mathrm{mod}(c,s)}$$

hence

$$\mathcal{F}_d^{\alpha,\beta}(X)_c = \sum_{m=0}^{t-1} \omega_d^{cm} \sum_{r=0}^{s-1} \omega_s^{r\,\mathrm{mod}(c,s)} X_{\alpha tr+\alpha m+\beta} = \sum_{m=0}^{t-1} \omega_d^{cm} \mathcal{F}_s^{\alpha t,\alpha m+\beta}(X)_{\mathrm{mod}(c,s)}$$

since by definition:

$$\mathcal{F}_s^{\alpha t,\alpha m+\beta}(X)_f = \sum_{r=0}^{t-1} \omega_d^{rf} X_{\alpha tr+\alpha m+\beta},\ (\forall) f < s$$