Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1992

The Speedup and Efficiency of 3-D FFT on Distributed Memory MIMD Systems

Dan C. Marinescu

Report Number:
92-013

Marinescu, Dan C., "The Speedup and Efficiency of 3-D FFT on Distributed Memory MIMD Systems" (1992). *Department of Computer Science Technical Reports*. Paper 938.
https://docs.lib.purdue.edu/cstech/938

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.

THE SPEEDUP AND EFFICIENCY OF 3-D FFT ON DISTRIBUTED MEMORY MIMD SYSTEMS

Dan C. Marinescu

CSD-TR-92-013
March 1992

The Speedup and Efficiency of 3-D FFT on Distributed Memory MIMD Systems

Dan C. Marinescu
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

March 6, 1992

Abstract

The paper analyzes a 3-D FFT algorithm for a distributed memory MIMD system. It shows that the communication complexity limits the efficiency even under ideal conditions to at most, $\eta_{opt} = 0.5$. Actual applications which experience load imbalance, duplication of work and blocking are even less efficient, therefore the speedup with P processing elements, $S(P) = \eta \times P$ is disappointingly small. Moreover, the 3-D FFT algorithm is not susceptible to massive parallelization, and the optimal number of PEs is small even for large problem size and fast communication. A strategy to reduce the communication complexity is presented.

Contents

1 Introduction

Multidimensional Fourier transforms are useful for image processing, for a variety of differential equations [Fox 87], and for numerous other applications.

This paper is primarily concerned with applications of 3-D FFT to crystallography, a subject discussed in depth in [Tene 73]. The molecular replacement method which uses crystallographic FFT transforms is analyzed in [Ross 72]. An overview of the basic ideas involved follows.

The 3-D FFT is needed for the phase refinement and extension in the computation of macromolecular structures like proteins and viruses. The phase refinement uses X-ray diffraction data to determine a set of observed structure amplitudes, and an initial guesses for the phases of the structure factors. Then it exploits the non-crystallographic symmetry and performs a sequence of transformations. First it transforms data from the reciprocal space into the real space, then it averages the electron density in the real space and then maps them back to the reciprocal space. To conclude the cycle, the calculated structure amplitudes are replaced by the observed ones. Several iterations are performed until no further improvements of the structure phases can be observed.

In this process the Fourier synthesis is used to compute the electron density at a given grid point (x_p, y_q, z_r) in the real space, given the structure factors in the reciprocal space $F_{h,k,l} = |F_{h,k,l}|e^{i\alpha_{h,k,l}}$ as follows.

$$\rho(x_p, y_q, z_r) = \sum_h \sum_k \sum_l |F_{h,k,l}| \cdot e^{2\pi i(hx_p + ky_q + lz_r - \alpha_{h,k,l})}$$

The Fourier analysis is used to determine the structure factors in the reciprocal space given the electron density in the real space

$$F_{h,k,l} = \sum_p \sum_q \sum_r \rho(x_p, y_q, z_r) \cdot e^{-2\pi i(hx_p + ky_q + lz_r)}$$

The crystallographic asymmetric unit may consist of up to 10^3 lattice points along each dimension, therefore the computing time which is proportional to $N \log N$ with $N = 10^9$ is substantial. Efficient FFT programs written in C run on an INTEL i860 (peak speed 60 Mflops) at speeds of up to 11 Mflops [Onsl 92]. Therefore one can expect execution times of 10^4–10^5 seconds for a problem with $N = 10^9$ grid points on a single processor i860.

To reduce the execution time of such problems, the only alternative is to use parallel systems. Distributed memory MIMD systems like the Intel iPSC/860 are widely available today and are considered in this paper. Yet, parallel algorithms to compute 3-D FFT exhibit a disappointingly small speedup even on massively parallel supercomputers like the

Touchstone Delta. This is due to the extensive communication among the PEs working on a 3-D FFT problem. This paper provides a qualitative explanation, namely that the communication complexity of the 3-D FFT is $E = \mathcal{O}(P^2)$ with E the number of events and P the number of threads of control. Then it performs a quantitative analysis and determines the optimum number of PEs and the optimal speedup for a problem of given size.

The communication pattern of a 3-D FFT algorithm [Kus 91] is analyzed in Section 2 of the paper. In Section 3 upper bounds for the speedup for several classes of algorithms are derived and it is shown that algorithms like the 3-D FFT cannot possibly reach an efficiency η larger than 0.5. The efficiency is informally defined as the ratio of the average time a PE is doing useful work to the total time required by the computation.

The efficiency of the 3-D FFT could be considerably smaller than the 0.5 upper bound due to load imbalance, duplication of work and blocking. This explains why the observed speedup $S(P) = \eta \times P$ is so low. Section 4 discusses data mapping strategies which use an optimal number of PEs. A strategy to reduce the communication complexity is presented in Section 5.

2 A 3-D FFT Algorithm for a Distributed Memory MIMD System

Any parallel algorithm for a MIMD system is based upon a certain data mapping strategy and upon a distribution of the computations to all the PEs of the system/machine partition allocated to the problem.

The basic idea of a 3-D FFT for $N = n_x \times n_y \times n_z$ data is to perform, say 2-D FFTs for the n_y $(x \times z)$ planes and then to perform an additional 1-D FFT along the y axis. Clearly the other two orientations for the planes are possible. In turn, each 2-D FFT in the $(x \times z)$ plane is performed as a sequence of say n_z 1-D FFTs along the x axis.

With this view of a 3-D FFT, the most obvious data partitioning strategy of an in-core transformation is to assign to each PE one or more planes for the 2-D FFT transformation. Such a group of planes is called a *slab*. Figure 2 in Section 4 illustrates this data partitioning and shows that each PE needs to gather the *slices* of a slab assigned to all other PEs for the second step of the algorithm.

The kernel of a program [Kus 91] which implements these ideas, is presented. The node program uses two system calls, `numnodes` and `mynode` to get the size of the partition and the id of the current PE. Then the number and the orientation of planes in a slab, the slab width, d_y and d_z, and size, sb_y and sb_z for y-slabs and z-slabs, and and the slice size, sl_y are

```
    /* Find number of nodes in the machine partition */
nproc = numnodes ( )
    /* Get id of the current node */
me = mynode ( )
    /* Compute slab width, slab size, and slice size */
d_y = slab_width (ny, nproc)
d_z = slab_width (nz, nproc)
sb_y = slab_size (nx,nz,d_y)
sb_z = slab_size (nx,ny,d_z)
sl_y = slice_size (nx,d_y,d_z)
    /* Get the (me)-th y-slab and do a 2-D FFT */
my_slab = get_slab (me, slab_size)
for i=1 to d_y
     call fft2d(my_slab)
end_for
   /* Transposition loop. Node (me) has the (me)-th y-slab and needs
      (nproc-1) slices for the (me)-th z-slab from all other nodes. */
for other = 1 to nproc -1
     my_partner = xor (me, other)
     this_slice = mod (my_partner, nproc)
     msg_type = force_msg + other
        /* Post asynch receieve for a slice expected from my_partner */
     iget = irecv (msg_type, buffer, sl)
        /* Send a message of zero length to my_parter and receieve one
           from him to synchronize */
     call csend (other, dummy, 0, my_partner, procid)
     call crecv (other, dummy, 0)
     /* Send the slice my_partner needs */
     call csend (msg_type, this_slice, sl, my_partner, procid)
     /* Wait to get the slice I need from my_partner */
     call msgwait (iget)
     /* Save the data into node z-slab */
     copy (buf, my_slab(my_partner))
end_for
for i=1 to d
     call fft1d (my_slab)
end_for
```

determined. A slab orthogonal to the Oy axis is called an *y-slab* and one orthogonal to the Oz axis is called a *z-slab*.

The PE performs a 2-D FFT on the d_y planes of the y-slab assigned to it. The second `for` loop distributes slices of the y-slab processed by the PE to all other PEs, and gathers the slices of the z-slab to be processed by the PE.

To make the program more efficient, *forced type messages* are used. Such messages bypass the standard flow control mechanism and are stored directly into the user's buffer. But such messages are lost if a `receive` is not posted by the time the message arrives. For this reason, each node posts an asynchronous `receive` and then chooses a partner with whom it first synchronizes and then sends the corresponding slice to. The two `csend` statements with zero length allows the current node `me` and its partner `my_partner` to synchronize. At each iteration of the second `for` loop, a pair (`me`, `my_partner`) is selected by both nodes. Indeed given a set of p integers from 0 to $p-1$ for any 3 integers i, j, k in this set, the following property holds:

$$\text{if } j = \text{ xor } (i,k) \text{ then } i = \text{ xor } (j,k).$$

The actual speedups observed for several runs using all 16 nodes of an INTEL iPSC/860 system (peak rate 960 Mflops) and the standard library routines [Kus 91] vary from a low of 4.95 for a $32 \times 32 \times 32$ mesh to a high of 10.61 for a $1024 \times 16 \times 16$ mesh. The actual Mflops rates vary from a low of 72.09 Mflops for the $1024 \times 16 \times 16$ mesh to a high of 142.38 Mflops for a $128 \times 128 \times 128$ mesh [Lyn 92].

Two problems of the same size $N = n_x \times n_y \times n_z$ but with different values of n_x, n_y, n_z produce very different results. The following data were obtained for two meshes, a $1028 \times 16 \times 16$ mesh and a $64 \times 64 \times 64$ mesh. In the first case the Mflops rate for a single node execution is 6.79 while in the second case it is 17.02. When all 16 nodes were used the corresponding rates are 72.09 versus 124.19 and the corresponding speedups are 10.61 versus 7.29. It seems rather odd that the higher speedup, 10.61, is obtained for the first mesh $1028 \times 16 \times 16$ while the second one, the $64 \times 64 \times 64$ mesh runs at a much higher Mflops rate.

As pointed out in [Hea 90] the i860 is very sensitive to the cache management. The reason why a speedup of 10.61 was observed is that the cache management was very poor when only one node was used to store the entire mesh of $1028 \times 16 \times 16$ points. For this reason the Mflops rate of the i860 was very small, 6.79 Mflops and the speedup appears artificially high. Interestingly enough, the same i860 runs at 17.02 Mflops for the same problem, the same total amount of data processes but for a different shape of the mesh. A similar effect namely a computing engine which runs very slow when all the data is stored in one node was reported in [Mar 91] where speedups larger than 2 for a two node systems were observed.

It is very difficult to measure experimentally the speedup because often it is impossible to run a large problem in only one node due to space constraints. When possible such an experiment takes a very long time and is prone to errors. The facts discussed above question the suitability of the speedup as a practical measure of performance. The speedup is meaningful only if the PEs run at the same rate throughout the entire experiment. It would make little sense to talk about the speedup of an eight processor CRAY Y-MP versus a one processor i386. But as our experiments show, a sophisticated RISC processor like the i860, runs at very different rates depending upon the amount of data handled by a node, and upon the data reference patterns. Therefore claims of high speedups observed experimentally have to be carefully scrutinized. The aggregate Mflops rate is in our view a more reliable measure of performance.

3 Communication Complexity and the Asymptotic Speedup

In the following, a parallel algorithm with P threads of control is considered. It is assumed that there is a one to one mapping of the P threads of control to the PEs of a parallel system and the terms of threads of control and PE which runs a thread of control are used interchangeably.

The parallel algorithm requires coordination of the P threads. Call any interruption of the flow of control of any thread a *communication/control event* or simply an *event* and denote the total number of events by E.

In [Mar 90], it is shown that massive parallelism is possible only when $E = \mathcal{O}(P)$ and in other cases, for example when $E = \mathcal{O}(P^2)$ as it is the case of 3-D FFT, the speedup reaches a maximum S_{opt} for $P = P_{opt}$ and then decreases asymptotically to zero.

Call $T(1)$ the execution time with one thread only and $T(P)$ the one with P threads. Then the speedup $S(P)$ is

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

If I denotes the instruction execution rate, then the "work" required by the algorithm with one thread is

$$W(1) = I \times T(1).$$

The parallel algorithm with P threads requires additional work for communication and control denoted by $W_{cc}(P)$. Assuming that threads do not duplicate each other's work, then

$W(P)$, the work with P threads of control is

$$W(P) = W(1) + W_{cc}(P).$$

Call $\bar{\theta}$ with $\bar{\theta} > 0$ the expected amount of work associated with one event and express the total work for communication and control as

$$W_{cc} = \bar{\theta} \times E.$$

If the computational load is divided evenly among all threads (the perfect load balanced case) and if the threads do not experience any blocking, then

$$W(P) = P \times (I \times T(P)).$$

It follows that under the three idealistic assumptions, namely no duplication of work among threads of control, perfect load balance and no idle thread during the computation, the speedup is

$$S(P) = \frac{P}{1 + \alpha' E} \qquad \text{with} \quad \alpha' = \frac{\bar{\theta}}{W(1)}.$$

When $E = aP$ the asymptotic speedup is $S_{asy}^{(P)} = \frac{1}{\alpha}$ with $\alpha = \frac{a\bar{\theta}}{W(1)}$.

Two more cases are considered now, namely $E = aP \log P$ and $E = aP^2$. In both cases, the speedup reaches a maximum $S = S_{opt}$ for $P = P_{opt}$ and then goes to zero asymptotically. The maximum speedup is respectively

$$S_{opt}^{(P \log P)} = \frac{1}{\alpha(1 - \log \alpha)} \quad \text{for} \quad P_{opt} = \frac{1}{\alpha}$$

and

$$S_{opt}^{(P^2)} = \frac{1}{2\sqrt{\alpha}} \quad \text{for} \quad P_{opt} = \frac{1}{\sqrt{\alpha}}.$$

The speedup with P processing elements, $S(P)$ and the efficiency, $\eta(P)$ relates as

$$S(P) = \eta(P) \times P.$$

It follows that in the two cases examined above, the efficiency is:

$$\eta_{opt}^{(P \log P)} = \frac{1}{1 - \log \alpha} \quad \text{and} \quad \eta_{opt}^{(P^2)} = \frac{1}{2}.$$

Figure 1 illustrates the dependency of the speedup upon the number of PEs for the three cases discussed above.

The optimal speedups and efficiencies discussed above are upper bounds for the speedup and efficiency attainable by actual applications which are not perfectly load balanced and experience blocking and duplication of work.

A qualitative analysis of the parameter α follows. As shown above

$$\alpha = \frac{a\bar{\theta}}{W(1)}.$$

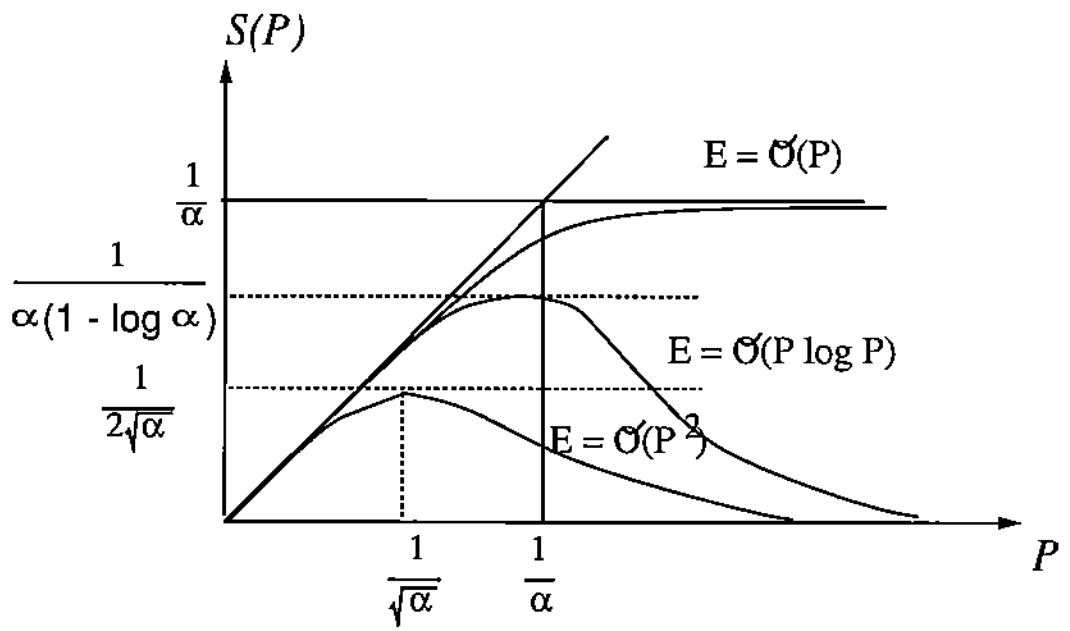

Figure 1. The speedups $S_{(P)}^{(P)}$, $S_P^{(P\log P)}$ and $S_{(P)}^{(P^2)}$ for the three cases $E = \mathcal{O}(P)$, $E = \mathcal{O}(P\log P)$ and $E = \mathcal{O}(P^2)$.

In general $W(1)$ is a function of the problem size, for example, in case of the Fast Fourier Transform $W(1) = bN\log N$ where N is the size of the set to be transformed and b is a constant depends upon the implementation. The larger is the problem size, the higher is the

speedup. Then $\bar{\theta}$ denotes the work associated with one event and depends upon the size of the message, in case of a message passing programming model or upon the size of the critical section, in case of a shared memory model. The parameter a is a constant determined by the actual number of events.

The model presented above is suitable for the shared memory and for the message passing models, and is consistent with the intuition that the larger is the computation to communication ratio of a parallel program, the larger is the speedup. Table 1 illustrates this for several values of the parameter α.

α	$S_{asy}^{(P)}$	$P_{opt}^{(P \log P)}$	$S_{opt}^{(P \log P)}$	$P_{opt}^{(P^2)}$	$S_{opt}^{(P^2)}$
10^{-1}	10	10	5	4	1.58
10^{-2}	100	100	33	10	5
10^{-3}	1,000	1,000	250	32	16.81
10^{-4}	10,000	10,000	2,000	100	50
10^{-5}	100,000	100,000	16,600	317	158.5
10^{-6}	1,000,000	1,000,000	142,857	1000	500

Table 1. The effect of communication to computation ratio upon the speedup.

4 Optimal Data Mapping Strategies for 3-D FFT

The communication complexity of the 3-D FFT algorithm discussed in Section 2 is $\mathcal{O}(P^2)$. Indeed, each thread of control owns a slice of the data that every other thread needs. Therefore each thread must send a slice of data to every other $(P-1)$ threads and must receive $(P-1)$ slices from the other threads.

Assuming that $a = 4$, the optimal speedup is

$$S_{opt} = \frac{1}{2\sqrt{\alpha}} \quad \text{with} \quad \alpha = \frac{4\bar{\theta}}{bN \log N}.$$

The optimal number of PEs and the optimal efficiency are

$$P_{opt} = \frac{1}{\sqrt{\alpha}} \quad \text{and} \quad \eta_{opt} = \frac{1}{2}.$$

As pointed out in Section 3, such algorithms cannot make an effective use of a massively parallel system. The optimum speedup is reached with a relatively modest number of PEs,

even when the communication to computation ratio is very small e.g., $P_{opt} = 1,000$ when $\alpha = 10^{-6}$. For practical implementations which experience blocking, work duplication, and load imbalance, the actual efficiency is lower than its optimal value of 0.5 and the actual speedup is considerably lower than its optimal value.

Two mapping strategies are considered which perform the 3-D FFT using P_{opt} processing elements for a problem of given size, $N = n_x \times n_y \times n_z$ and for a given parallel system with NPROC processing elements each PE with MDATA + MPROG available memory.

We make several assumptions, namely that the system has sufficient resources, namely that $P_{opt} >$ NPROC and that each PE has enough memory to hold a plane of data, $n_x \times n_y \geq$ MDATA, $n_x \times n_z \geq$ MDATA, $n_y \times n_z \geq$ MDATA.

The first data mapping algorithm groups together d planes of data into a "slab", assigns one slab to each one of the P_{opt} processing elements and attempts to minimize the load imbalance among PEs and to reduce the blocking time. The geometry of the data mapping is presented in Figure 2.

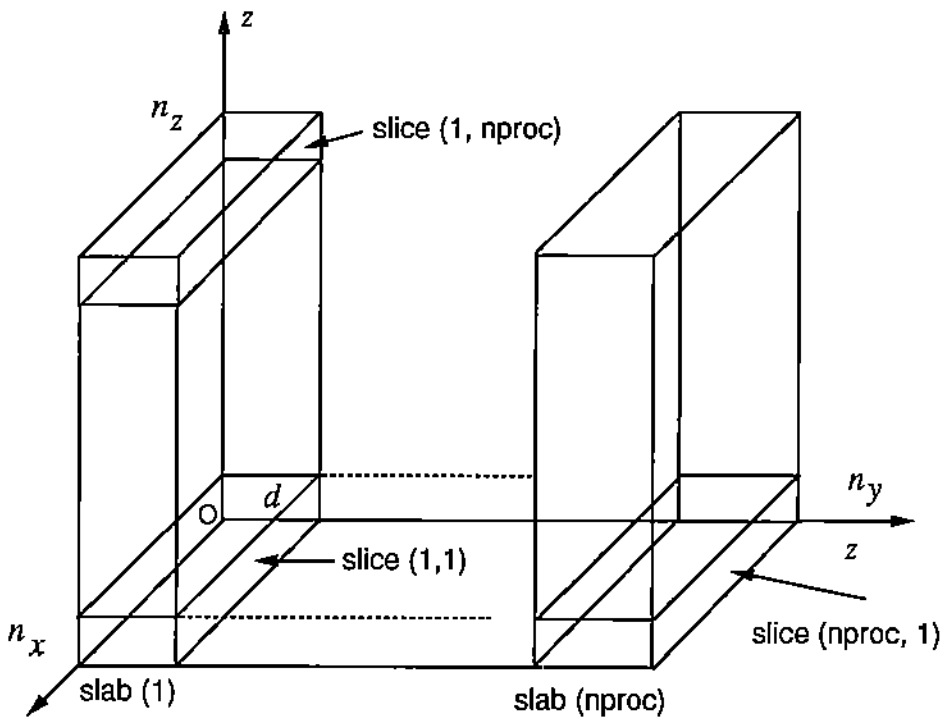

Figure 2. The slabs and slices for a data mapping with slabs orthogonal to the y axis.

The algorithm involves two steps.

Step 1. Compute the optimal slab size.

$$d_x = \frac{n_x}{P_{opt}}, \quad d_y = \frac{n_y}{P_{opt}}, \quad d_z = \frac{n_z}{P_{opt}}$$

and

$$\delta_x = P_{opt} \times d_x - n_x$$
$$\delta_y = P_{opt} \times d_y - n_y$$
$$\delta_z = P_{opt} \cdot d_z - n_z$$
$$\delta = \min(\delta_x, \delta_y, \delta_z).$$

For example when $\delta = \delta_x$ assign to each thread an $(x \times z)$ plane and group d_y such planes into one y-slab.

Step 2. Use P_{opt} processing elements and assign to each PE a group of d such planes. In this case the size of each slice exchanged among PEs is $sl_y = d^2 \frac{n_x \times n_y}{n_z}$. Each PE has assigned to it a slab of size sb_y with $sb_y = n_x \times n_z \times d$.

An alternative data mapping strategy ignores load balance effects and attempts to group planes into slabs to reach the P_{opt} value.

For example, considering the same orientation of planes as in the preceding case, d_y and d_z are determined as follows:

$$P_{opt} = \frac{n_y}{d_y}$$

but

$$P_{opt} = \frac{1}{\sqrt{\alpha}} \quad \text{with} \quad \alpha = \frac{4\bar{\theta}}{bN \log N}.$$

It follows that

$$d_y = n_y \sqrt{\frac{bN \log N}{4\bar{\theta}}} \quad \text{and} \quad d_z = n_z \sqrt{\frac{bN \log N}{4\bar{\theta}}}.$$

Memory constraints may affect the choice of d and may force the use of $P > P_{opt}$ processing elements. This happens when $sb_y >$ MDATA, $sb_x >$ MDATA and $sb_z >$ MDATA.

In this case d will be chosen to minimize $\left(\frac{n_x}{d_x}, \frac{n_y}{d_y}, \frac{n_z}{d_z}\right)$ with d_x, d_y, d_z the largest slab width which fits into memory for a given orientation of the slabs.

In the previous analysis, we have made the assumption that $\bar{\theta} = \bar{\theta}_0 + (sl) \times \bar{\theta}_1$ with $\bar{\theta}_1 << \bar{\theta}_0$ such that the expected work per event, $\bar{\theta}$ is virtually independent upon the size of slice, sl. If this is not true, then the optimal value of d is $d = 1$ and then the number of PEs used is the value n_x, n_y or n_z closest to P_{opt}.

5 Communication, Space, Time Tradeoffs

The communication complexity of the 3-D FFT can be reduced if enough space to store the entire 3-D structure with $N = n_x \times n_y \times n_z$ elements is available. For example, if $N = 200 \times 127 \times 127$ and a hypercube with 128 PEs each with 16 Mbyte of memory is available, then the following communication strategy may be used.

The algorithm presented below allows the transposition of the 3-D structure in the third dimension in $\mathcal{O}(P)$ communication steps with P processing elements. At the beginning of the transposition PE_i, $0 \leq i \leq P-1$ has completed a 2-D FFT for the i-th y-slab (a slab orthogonal to Oy). At the end of the transposition, PE_i will have the i-th z-slab (a slab of orthogonal to Oz) and will be able to perform the transformation along the Oz axis (see Figure 2).

For the sake of simplicity, assume that $d = 1$, each slab consists of exactly one plane and that $n_y = n_z = 2^k - 1$, so that the PEs can be organized as a binary tree. This assumption is not restrictive, for example if $n_y = 2^{k'}$, then a minimum spanning tree like the one for broadcast-collapse communication on a hypercube [Mar 90], can be used. It is only important that each PE knows its level in the tree, its ancestor, and its descendents.

The communication strategy involves two steps. An *upwards propagation* takes place in Step 1. The nodes at level k (the leafs) send their y-slabs to their ancestor at level $k-1$. Then each node at level $k-1$ in the tree adds its own y-slab to the ones received from their descendents and sends the entire package to its ancestor at level $(k-2)$. The process continues and after $k-1$ steps, the root receives all $2(2^{k-1}-1)$ y-slabs. Then the root constructs all n_z z-slabs.

Step 2 (*downwards propagation*) is initiated by the root which retains the z-slab it is processing (slab ϕ) and then packs together two *super slabs*, each consisting of $2^{k-1}-1$ z-slabs, one for its left and one for its right subtree. Each node at level 1 receives the super slabs, retains the slab it is going to process during the second phase of the 3-D FFT and passes down to its descendents two z super slabs consisting of $(2^{k-2}-1)$ z-slabs. The process continues until the nodes at level k have received their z-slabs.

References

[Fox 88] Fox, G., and al, *Solving problems on concurrent processors*, Prentice Hall (1988).

[Kus 91] Kushner, Ed., *In core 3-D FFT program for iPSC/860*, Private communication. Also Intel, iPSC/860 Basic Math Library User's Guide (1991).

[Lyn 92] Lynch E. Robert., Private communication.

[Hea 90] Heath, M.T., Geist, G.A., and Drake, J.B., *Early Experience with the Intel iPSC/860 at the Oak Ridge National Laboratory*, ORNL/TM-11655, (1990).

[Mar 90] Marinescu, D.C. and Rice, J.R., *On high level characterization of parallelism*, to appear in JPDC. Also CSD-TR-1011 (1990).

[Mar 91] Marinescu, D.C., Beaven M., and Stansifer Ryan, *A parallel algorithm to compute invariants of Petri net models*, in Proc. PNPM 91, IEEE Press, pp. 136-143, (1991).

[Onsl 92] Onslott, G., Private communication.

[Ross 72] Rossmann, M.G., *The molecular replacement method*, Gordon and Breach, New York (1972).

[Tene 73] Ten Eyck, L.F., *Crystallographic Fast Fourier Transforms*, Acta Cryst. A29, pp. 183–191 (1973).