CENTER FOR RESEARCH ON PARALLEL COMPUTATION

CRPC-93-10 December 29, 1993

Concurrent Three-Dimensional Fast Fourier Transform Algorithms for Coarse-Grained Distributed Memory Parallel Computers*

Edith Huang[1], Paulett C. Liewer[1], Viktor K. Decyk[2]
and Robert D. Ferraro[1]

Technical Report

CALIFORNIA INSTITUTE OF TECHNOLOGY

Caltech 217-50

Pasadena, California 91125

(818) 395-4562

CRPC-93-10 December 29, 1993

Concurrent Three-Dimensional Fast Fourier Transform Algorithms for Coarse-Grained Distributed Memory Parallel Computers*

Edith Huang[1], Paulett C. Liewer[1], Viktor K. Decyk[2]
and Robert D. Ferraro[1]

[1] *Jet Propulsion Laboratory*
California Institute of Technology
Pasadena, California 91109

[2] *Physics Department*
University of California
Los Angeles, California 90024

* *This work was supported in part by the National Science Foundation under Cooperative Agreement No. CCR-9120008.*

Concurrent Three-Dimensional Fast Fourier Transform Algorithms for Coarse-Grained Distributed Memory Parallel Computers

Edith Huang and Paulett C. Liewer
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

Viktor K. Decyk
Physics Department, University of California, Los Angeles, 90024

Robert D. Ferraro
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

Abstract

Two concurrent three-dimensional real-to-complex Fourier transform algorithms have been developed for coarse-grained distributed memory parallel computers which perform the 3-D transform as three separate sets of 1-D fast Fourier transforms (FFTs) with each processing node performing local sequential 1-D FFTs in parallel. In the first algorithm, the 3-D data is first partitioned into "slabs" (*e.g.*, all x and y for a range of z) and the slabs distributed among the nodes so each performs sequential FFTs on two dimensions with no interprocessor communication. Following a redistribution of the data among the processors, each node then performs sequential FFTs on the third dimension. In the second scheme, the data is partitioned into "rods" (*e.g.*, all x for a range of y and z) and is redistributed between each of the three sets of sequential FFTs. Since each processor performs sequential FFTs locally, optimized assembly language node FFTs can be used. A particular feature of both the slab and rod 3D FFT algorithms is that the Fourier coefficients are packed into the original data array so that the Fourier coefficient array is the same size as the original data array. Timing results for both algorithms are presented from the Intel Delta Touchstone. We find that the Slab Decomposition FFT is faster in all cases. Source codes for the parallel FFTs, as well as a sample program using the FFT to solve Poisson's equation, are given in the Appendix.

1. Introduction

Multi-dimensional fast Fourier transforms (FFTs) are used for the solution of computational problems in virtually all scientific fields. The specific computational problem motivating the present work is the use of real to complex FFTs to solve Maxwell's equations in plasma particle-in-cell simulations codes such as those used in the Numerical Tokamak Project[Dawson *et al*, 1993]. Efficient parallel algorithms are needed for FFTs for MIMD computers such as the Intel Delta which do not provide parallel FFTs as part of the system software.

For a 1-D FFT, the only choice for parallelization is a domain decomposition of the 1-D data such as described in Fox *et al* [1988]. However, for

a multi-dimensional FFT, another choice is possible since an n-dimensional FFT can be done as n consecutive sets of 1-D FFTs(Jackson *et al* [1991]). Here, we describe two methods for implementing a parallel 3D FFT where each processor performs sequential 1-D FFTs in parallel. Interprocessor communication is needed only between sets of 1D FFTs. Since each node performs sequential FFTs independently, machine-specific optimized node FFTs can be used.

Consider a three dimensional array of real data $A(l,m,n)$ on an xyz-grid of dimension $n_x \times n_y \times n_z$. The 3D Fourier transform coefficients of this array are

$$C(l',m',n') = \sum_{l=0}^{n_x-1}\sum_{m=0}^{n_y-1}\sum_{n=0}^{n_z-1} A(l,m,n)\, e^{-2\pi i\left(ll'/n_x + mm'/n_y + nn'/n_z\right)} \qquad (1)$$

for $l' = 0, n_x/2$, $m' = 0, n_y - 1$ and $n' = 0, n_z - 1$. Because the A(l,m,n) are real, the other l' modes, $l' = n_x/2+1, n_x$, can be recovered using the relationship

$$C(n_x - l', m', n') = C^*(l', n_y - m', n_z - n') \qquad (2)$$

which follows from Eq. 1 and the identity

$$e^{-i2\pi n'n/n_z} = e^{-i2\pi(n'-n_z)n/n_z},$$

Equation 1 can be written in the equivalent form

$$C(l',m',n') = \sum_{n=0}^{n_z-1} e^{-i2\pi n'n/n_z}\left(\sum_{m=0}^{n_y-1} e^{-i2\pi m'm/n_y}\left(\sum_{l=0}^{n_x-1} e^{-i2\pi l'l/n_x} A(l,m,n)\right)\right) \qquad (3)$$

from which it is clear that the 3-D transform can be accomplished through three consecutive 1-D transforms. Note that the 1-D transform over the first index l can be done independently for each m and n; this is the basis for the parallel Fourier transform algorithms presented here.

In the first parallel algorithm, presented below, the 3-D transform is accomplished by initially partitioning the real data array $A(l,m,n)$ into "slabs" so that each processor has all l and m (x and y) for a portion of the $n(z)$ values as illustrated in Fig. 1. Each processor then performs sequential transforms in x and y (the sums over l and m) for its range of $n(z)$ values. The partially transformed data is then redistributed so that each processor has all $n(z)$ values; each processors then performs sequential z transforms for its portion of the partially transformed data. We refer to this algorithm as the *Slab Decomposition FFT.*

In the second algorithm, the 3-D transform is accomplished by initially partitioning the $A(l,m,n)$ array into rectangular cross section "rods" so that each processor has all $l(x)$ for a range of m and n (y and z), as illustrated in Fig. 2. Each processors performs the sequential l transforms (*x transforms*) for its portion

of the *A(l,m,n)* array. The partially transformed data is then redistributed among the processors so that each processors has all *m*, (all *y*), but a range of *l* and *n*. (*x* and z). Each processor now performs sequential *m* transforms (*y transforms*) for its portion of *l* and *n*. The partially transformed data is again redistributed so that each processor has all *n* for a portion of *l* and *m;* the processors then perform the sequential *n* transforms(z *transforms*) . Again, inter node communication is needed only to redistribute the data. We refer to this method as the *Rod Decomposition FFT*.

The first algorithm (Slab Decomposition) requires only one redistribution of data among the processors. But the maximum number of processor which can be utilized for this algorithm can be restrictive: Because of the initial partition in z and the intermediate partition in x (at which stage the coefficients are complex), the maximum number of processor which can be used is $nprocs=min(n_Z,n_X/2)$. If more processors are needed for other portions of the user's application as in a PIC code (see, for example Liewer and Decyk, [1989] and Ferraro *et al*, [1993]) some of the processors will, in fact, be idle during the transform. The second algorithm (Rod Decomposition) gives a load balanced decomposition as long as n_Y*n_Z,, $n_X/2*n_Y$ and $n_X/2*n_Z$ are multiples of the number of processors. The maximum number of processor which can be used here, $min(n_Y*n_Z,n_X/2*n_Y,n_X/2*n_Z)$, is less restrictive than for the Slab Decomposition. However, it requires two redistributions of the data and thus more interprocessor communication.

Timing comparisons of the two algorithms on the Intel Delta Touchstone at Caltech show that, for a given problem size on a given number of processors, the Slab Decomposition is faster in all cases, *even when the Rod Decomposition can use more processors*. Times for the Rod Decomposition always begin to *increase* at some point as the number of processors increases because of the communication overhead. Thus we conclude that it is preferable to use the Slab Decomposition algorithm even if this means some processing nodes must be left idle.

Note that following the first (here, the *l* or *x*) transform of the real data *A(l,m,n)* with $n_X*n_Y*n_Z$ words, one obtains $(n_X/2+1)*n_Y*n_Z$ complex Fourier coefficients requiring $2*(n_X/2+1)*n_Y*n_Z$ words and thus it appears that more storage space is need than for the original real array. However, some of these coefficients are related to other coefficients by simple conjugation (see Eq. 2) and it is possible to store all necessary coefficients in the $n_X*n_Y*n_Z$ words allocated for the original data so that the transform can be done "in place," as will be discussed below.

The plan of this paper is as follows. In Sec. 2, our investigation of various schemes for the interprocessor exchange of data is described. In Sec. 3, the scheme used to pack the Fourier coefficients into the original array allocated for the data is described. The two FFT algorithms are described in detail in Sec. 4 and timings are given in Sec. 5. In Sec. 6, a sample problem using the 3D FFT to solve Poisson's equation, is described and the pseudo code presented. The Appendix gives a FORTRAN source code for the sample problem (solution of Poisson's equation) and the two parallel 3D FFT algorithms. The FFT source code package

includes a routine which can be called to give the wave vectors stored in the same manner as the complex Fourier coefficients to enable a use of this parallel FFT package as a "black box."

2. Interprocessor Communication for Data Exchange on Intel Delta

The key feature of the parallel 3D FFT algorithms presented here is that, by performing the 3D FFT as sets of 1-D FFTs, interprocessor communications is needed only between the sets of 1-D FFTs to redistribute the data among the nodes. The 1-D FFTs are performed locally on the nodes. Since the interprocessor communication is the only source of parallel inefficiency when all nodes are participating in the FFT, different methods of data exchange were investigated.

From Fig. 1, it can be seen that the Slab Decomposition 3D FFT requires each processor to exchange a block of data $n_x/nprocs*n_y*n_z/nprocs$ with every other processor where *nprocs* is the number of processors. This is essentially a transpose of the array. The Rod Decomposition (Fig. 2) requires each processor to exchange a block of data with every other processor within a "plane" of processors; the "plane" of processors is different for the two redistributions of the data.

To find the optimum method for the Intel Delta, communication tests were done with each processor exchanging data with every other processor as occurs in the Slab Decomposition parallel FFT. Different message sizes, from 4 bytes up to 1 million bytes on mesh size ranging from 4 nodes to 256 nodes were used in the tests. Initially, we planned to use the fastest, but, of the four schemes investigated, only one worked for large meshes on large message and thus this is the method used in the parallel FFT codes. This method, which uses fully asynchronous communication, is described first below. In the tests and in the FFT itself, the order in which a processor receives the data is unimportant because the first word of the message contains the source node number, and from this, the proper storage location for the data can be determined.

Fully asynchronous exchange (irecv/isend). The method used in the FFT timing result presented below and in the code in the Appendix uses fully asynchronous communication. This is the only method that did not cause communication deadlock on the Delta for large messages on large meshes. It requires the user to allocate a working array equal in size to the processor's portion of the original array. To exchange the data, each processor first posts an asynchronous receive. Next a block of data is sent to a different processor, and then the node waits for receipt of the incoming message. The data received is placed in the temporary working array to avoid overwriting data not yet sent. This process is repeated until each processor has exchanged data with all others. The Intel pseudo code for this asynchronous exchange follows. Note that in this particular example, utilizing the "me.xor.i" construct, the processor which receives the data is the same as the one sending the data, *i. e.*, this is an actual exchange of data.

```
c Asynchronous data exchange among all processors
     do i = 0, nprocs-1
c Post message receipt. me=my processor id
       irmsg = irecv (me, inbuff, nbytes)
c Chose destination node. Each nodes sends to a different
c node to avoid communication contention
       is = me.xor.i
c Put outgoing data in buffer
       .
       .
c Send message
       ismsg = isend (is, outbuff, nbytes, is,0)
c Wait for incoming message
       call msgwait(irmsg)
c Wait for isend to finish
       call msgwait(ismsg)
c Store data in temporary array
       .
       .
     enddo
```

Timing results (bandwidth in units of Mbytes/sec) for this communication scheme on the Intel Delta as a function of message size are shown in Fig. 3 for four different sizes of processor meshes, 16x16, 16x8 and 8x8. The Delta has 512 processors arranged on a 16x32 two-dimensional mesh. Also shown is the peak bandwidth of the Intel Delta (11 Mbytes/sec). Timing results (Mbytes/sec) as a function of processor mesh size for a fixed message size (40 Mbytes) are shown in Fig. 4. Fig. 5 shows the dependence of the bandwidth as a function of message size for two different sized processor arrays, 16x8 and 4x8. In both figures the curves labeled **ir/is** are the results for this asynchronous method.

Synchronous exchange (csend/crecv). A second interprocessor data transpose investigated used synchronous communication. Each processor first sends a block of data to one processor (using the `csend` call), then receives a block of data from another (using the crecv call). The process is repeated until all processors have exchanged data. Again, the data received must be placed in a temporary array to avoid overwriting data which has not yet been sent. Observed bandwidths for this method are shown on the curves in Fig. 4 and 5 labeled **cs/cr**. Communication deadlock occurred on the Delta for exchanging large messages on large numbers of processors (about a megabyte on 512 processors.)

Asynchronous receive/ Synchronous send (irecv/csend). A third interprocessor data transpose investigated used asynchronous receives and synchronous sends. Each processor first posts a message receipt (using the `irecv` call). Next a block of data is sent to a different processor synchronously (using a "blocking" send, `csend`). The processor then waits for receipt of its incoming message. This is repeated until exchanges have been made with all other processors. Bandwidths for this communication scheme are shown in curves in Figs. 4 and 5 labeled **ir/cs**. Here, as in the fully synchronous scheme, communication deadlock occurred when exchanging large messages on a 512 node mesh.

A fourth communication method, which does not require the user to allocate the temporary storage array, was also investigated. Here each processor synchronously sent one block of data to *all* of the other processors (using the call `csend`). It then synchronously receives messages from all other processors (using the call `crecv`). Since all the data has been sent out before any new data is received, the need for the user to allocate a temporary array has been eliminated. Instead, the system must allocate system communication buffer space. Unfortunately, for large messages on a large number of processors, the system communication buffers overflowed on the Delta and communication deadlock occurred. No timings were done for the scheme. The largest case that ran using this approach was on 128 nodes for an array $A(64,64,64)$.

3. Storage of the Fourier Coefficients for "Packed" Algorithm

Both the Slab and Rod Decomposition FFT algorithms presented here utilize packing of the Fourier coefficients into the original data array and here we explain the packing scheme used.

The Fourier transform of the $n_x \times n_y \times n_z$ array of real data produces $(n_x/2+1) \times n_y \times n_z$ complex Fourier coefficients (Eq. 1). If one allocates the extra storage (two additional planes of data $n_y \times n_z$) and uses $2*(n_x/2+1)*n_y*n_z$ words to store the Fourier coefficients, they are stored in a real array $A(l,m,n)$ of dimension $n_x+2 \times n_y \times n_z$ with the real and imaginary parts adjacent:

$$\begin{aligned} ReC(l',m',n') &\quad \text{stored in} \quad A(2l'+1,m'+1,n'+1) \\ ImC(l',m',n') &\quad \text{stored in} \quad A(2l'+2,m'+1,n'+1) \end{aligned} \tag{4}$$

where we have used the FORTRAN convention of starting the array index at 1. This is equivalent to putting the coefficients in a complex array B of dimension $(n_x/2+1)*n_y*n_z$ with $A(l',m',n')$ stored in $B(l'+1,m'+1,n'+1)$. Note the offset of 1 between the array element indices and the Fourier coefficient indices due to the Fortran convention of indexing from 1. Equation 4 gives the basic storage scheme used in the parallel algorithms.

However, there is redundant information in the Fourier coefficients in all stages in the transform process, so that it is possible to "pack" the Fourier coefficients into the original $n_x \times n_y \times n_z$ array. This is often referred to as an "in place" FFT and facilitates use of the FFT as a "black box" since no extra storage need be allocated for the extra planes of Fourier coefficients. Packing is a particular feature of both parallel FFT algorithms presented here. Here, only the redundancy and packing of the fully transformed array is described in detail since a users' application may need access to the Fourier coefficients. Similar considerations apply in the intermediate transforms as discussed in Sec. 4.

Because the original data is real, there is redundancy within the $l'=0$ and $l'=n_x/2$ Fourier coefficients. Only half of the information need be saved for each so that it is possible to pack the needed information for the $l'=n_x/2$ coefficients into the $l'=0$ portion of the array and thus eliminate the need for the extra two

"planes" of storage. No other Fourier coefficients are involved in the packing and all others are stored as in Eq. 4 above. In the parallel algorithms, this implies that only the processor storing the $l'=0$ portion of the array has non-standard mode storage.

For $l'=n_x/2$, Eq. 1 gives

$$C(n_x/2,m',n') = \sum_{n=0}^{n_z-1} e^{-i2\pi n'n/n_z} \sum_{m=0}^{n_y-1} e^{-i2\pi m'm/n_y} \left(\sum_{l=0}^{n_x-1} \cos(\pi l) A(l,m,n) \right) \qquad (5)$$

where the term in parentheses (the sum over l) is real because the data $A(l,m,n)$ are real. An analogous result is obtained for l'=0 with the $\cos(\pi l)$ replaced with 1. Three special cases of redundancy and packing of the $l'=0$ and $l'=n_x/2$ modes need to be treated. The packed Fourier coefficient array is shown in Fig. 6. Figure 6a shows the $l'=0$ plane which would, in the unpacked case, contain all the Re$C(0,m',n')$ and Fig. 6b shows the $l'=1$ plane which would contain the Im$C(0,m',n')$ coefficients. No other parts of the array are affected by the packing.

Case 1. Four coefficients for$(m',n')=\{(0,0), (n_y/2,0),(n_y/2,n_x/2)$ and $(0,n_z/2)\}$. It can be seen from Eq. 5 that for these four sets of (m',n'), the $C(n_x/2,m',n')$, as well as the $C(0,m',n')$, are real. The Re$C(n_x/2,m',n')$ are packed in the locations of the Im$C(0,m',n')$. These modes are indicated by the four dotted squares in Fig. 6b.

Case 2. Two columns of modes $m'=0$ and $n_y/2$ for $n'=1,n_z-1$, excluding $n'=n_z/2$. Half of these modes are redundant. For these modes,

$$C(n_x/2,n_y/2,n') = \sum_{n=0}^{n_z-1} e^{-i2\pi n'n/n_z} \left(\sum_{m=0}^{n_y-1} \cos(\pi m) \sum_{l=0}^{n_x-1} \cos(\pi l) A(l,m,n) \right), \qquad (6)$$

where the term in parenthesis is real and similarly for the coefficients$(n_x/2,0,n')$, $C(0,0,n')$ and $C(0,n_y/2,n')$. Using the identity

$$e^{-i2\pi n'n/n_z} = e^{-i2\pi(n'-n_z)n/n_z}, \qquad (7)$$

it can be seen that

$$C(n_x/2,n_y/2,n') = C^*(n_x/2,n_y/2,n_z-n') \qquad (8)$$

and thus half of the modes can be obtained from the other by complex conjugation using the prescription of Eq. 8. An analogous result holds for the other three columns. The Fourier coefficients for $C(n_x/2,0,n')$, for $n'=n_z/2+1,n_z-1$ are stored in the locations of the $C(0,0,n')$, for $n'=n_z/2+1,n_z-1$ coefficients. The

Fourier coefficients for $C(n_x/2,n_y/2,n')$, for $n'=n_z/2+1,n_z-1$ are stored in the locations of the $C(0,n_y/2,n')$, for $n'=n_z/2+1,n_z-1$ coefficients. When needed, the other modes are generated from those stored by complex conjugation as given in Eq. 8. These mode columns are shown as the striped rectangles in Figs. 6a&b.

Case 3. $l'=nx/2$ Coefficients excluding Cases 1 and 2. From Eq. 5 and Eq. 7, it can be shown that

$$C(n_x/2,n_y-m',n_z-n') = C^*(n_x/2,m',n') \tag{9}$$

and similarly for the $l'=0$ modes. Thus it is necessary to store only half of these modes with the other redundant Fourier coefficients recreated by complex conjugation as needed. The modes $C(n_x/2,m',n')$ for $m'=n_y/2+1,n_y-1$ and $n'=1,n_z/2-1$ are stored in the locations of the modes $C(0,m',n')$ for $m'=n_y/2+1,n_y-1$ and $n'=0,n_z-1$. These modes are shown as the shaded rectangle in Figs. 6a&b.

4. Description of Slab and Rod Decomposition Parallel 3D FFT Algorithms

a. The Slab Decomposition 3-D FFT

In the forward Slab Decomposition FFT algorithm, the global data array $A(n_x,n_y,n_z)$ is assumed to initially partitioned into slabs along the z axis as shown in Fig. 1 so that each node has a fraction 1/nprocs of the total data where nprocs is the total number of processors used. Each node has all x and y data for a portion of the z axis. A is real and n_x, n_y and n_z are all powers of 2. Thus the data is initially distributed locally as $A(n_x,n_y,ns_z)$ where $ns_z=n_z/nprocs$.

First, a node-optimized 1-D FFT is used to perform the transform in x direction. On the Intel machines, Intel's real-to-complex 1-D FFT routine `scfft1d` is called. The x-transform of the data results in $n_x/2+1$ complex coefficients, which exceeds the initial number of data points. However, because some of these coefficients are real, they can be packed into original array. The two planes of coefficients $C(0,m,n)$ and $C(n_x/2,m,n)$ for $m=0,n_y-1$ and $n=0,n_z-1$ are real. We pack the $C(n_x/2,m,n)$ coefficients into the location of the Im$C(0,m,n)$ of an unpacked algorithm (i.e., in the $A(2,m,n)$ plane). Here, and at all stages in the Slab Decomposition FFT, only the first two x-planes ($l=1\&2$ for *all* m,n) of the array are involved in the packing and thus, localized to one processor (presumably logical processor 0).

Next, a node-optimized complex-to-complex FFT is used to perform the transform in y. The two packed planes of modes must be treated as special cases. After the y transform, packing is again necessary. The rows of coefficients $C(0,0,n)$ *and* $C(n_x/2,0,n)$ *for* $n=0,n_z-1$ *are real;* $C(n_x/2,0,n)$ *is packed in the location of the* Im$C(0,0,n)$, e.g. in $A(2,0,n)$. The complex coefficients $C(0,m',n)$ and $C(n_x/2,m',n)$ for $m'=1,n_y-1$ and $n=0,n_z-1$ have redundant modes, *e.g.*, the bottom portion of the $C(0,m',n)$ for $m=n_y/2+1,n_y-1$ can be obtained from the top portion, $m=1,n_y/2$ by complex conjugation using the prescription in Eq. 9. The lower portion of the

coefficients $C(n_x/2,m',n)$ for $m'=n_y/2+1$, $n=0,n_z-1$ are stored in the location of the lower portion of the $C(0,m',n)$; the upper portion can be recreated using Eq. 9. The rows $C(0,ny/2,n)$ and $C(n_x/2,n_y/2,n)$ for $n=0,n_z-1$ are real; the $C(n_x/2,n_y/2,n)$ are stored in the location of the $\mathrm{Im}C(0,n_y/2,n)$.

Following the x- and y- transform, the data is redistributed among the processors using the fully asynchronous method described in Sec. 2. After the exchange, each node will have all of y and z data for a portion of the x axis as shown in Fig. 1. Again, a node-optimized complex-to-complex 1-D FFT is used (here, Intel's `csfft1d` routine) to perform the transforms in the z-direction.

Because of the packing, the z-transform must be done in several steps. First, the row $C(0,0,n)$ for $n=0,n_z-1$ are unpacked and transformed. $C(0,0,0)$ and $C(0,0,n_z/2)$ are real; the others are complex. Only the top portion, $n=1,n_z/2-1$ need be saved because the lower portion, $n=n_z/2-1,n_z-1$, can be recreated using the prescription in Eq. 9. Next, the row $C(n_x/2,0,n)$ for $n=0,n_z-1$ are unpacked and transformed. Again, $C(n_x/2,0,0)$ and $C(n_x/2,0,n_z/2)$ are real and are packed together with $C(n_x/2,0,n_z/2)$ packed into the location of $\mathrm{Im}C(n_x/2,0,0)$. The complex coefficients $C(n_x/2,0,n)$ for $n=1,n_z/2-1$ are saved; the modes for $n_z/2=n_z/2+1,n_z-1$ can be obtain from these. We save $C(n_x/2,0,n)$ for $n=n_z/2+1,n_z-1$ and pack them in the location of the second half of the $C(0,0,n)$ as shown in Fig. 6. The modes $C(0,0,0)$ and $C(n_x/2,0,0)$ are real and packed together, as are $C(0,0,n_z/2)$ and $C(n_x/2,0,n_x/2)$ as shown in Fig. 6

In an analogous manner, the $C(0,n_y/2,n)$ for $n=0,n_z-1$ are unpacked and transformed and only the top portion saved, $n=0,n_z/2+1$. Next, the $C(n_x/2,n_y/2,n)$ for $n=0,n_z-1$ are unpacked and transformed and only the bottom portion saved, $n=n_z/2+1$, n_z-1, in the location below the saved $C(0,n_y/2,n)$ as shown in Fig. 6. $C(0,n_y/2,0)$ and $C(n_x/2,n_y/2,0)$ are both real and stored together as shown in Fig. 6 and likewise for $C(0,n_y/2,n_z/2)$ and $C(n_x/2,n_y/2,n_z/2)$. The 1-D FFT complex-to-complex z-direction transforms are performed on the rest of the data. No special treatment is needed for the packed planes of complex coefficients which contain the saved complex $C(0,m,n)$ and $C(n_x/2,m,n)$.

After the completion of all z-direction transforms the data structure is shown in Fig. 6. The forward in place 3D-FFT real-to-complex has been completed. It is important to note that the packed data are in node 0 only. Only two planes of data - $A(1,l,m)$ and $A(2,l,m)$ - contain any packed data. All other locations are the same as in "unpacked" algorithm, for all of z and y data.

The inverse transform is done by applying the routines and data exchange in reverse order, unpacking and re-creating modes by complex conjugation as needed. At the end of the inverse transform, the data is stored as it was initially, i.e., in a slab decomposition partitioned in z.

Because of the initial partition in z, n_z must be a multiple of the number of processors and the maximum number of processor which can be used is n_z. After the redistribution of data following the x and y transforms, the data is partitioned in the x direction into slabs. Because of this, $n_x/2$ must be a multiple of the number of processors because in this stage, there are $n_x/2$ complex Fourier

coefficients and the real and imaginary parts must be in the same processor. If this is too restrictive, the intermediate stage could also use a "rod" decomposition which would allow $n_x/2*n_y$ processors to be used. The Fortran source code for both the forward (real to complex) slab decomposition FFT routines, sfft3rc and the inverse (complex to real) sfft3cr are included in the Appendix.

B. The Rod Decomposition 3-D FFT

In the Rod Decomposition parallel 3D FFT, the data is partitioned into rectangular cross section "rods" such that each node has all of x data but portion of y and z data as shown in Fig. 2. This is a 2-dimensional decomposition of the 3D data. Let nrows be the number of processors assigned to the y direction and ncols be the number assigned to the z direction. This forms a logical processor mesh of nprocs=nrows*ncols processors. On the Intel Delta, the logical mesh maps to a physical mesh of size nrowsxncols. The real array $A(n_x,n_y,n_z)$ is thus decomposed in such a way that $A(n_x,nsr_y,ns_z)$ is a local matrix on each node, where $nsr_y=n_y/nrows$ and $ns_z=n_z/ncols$. (Whether the processor mesh is real or logical is unimportant in the following discussion. The underlying parallel architecture is irrelevant to the code and all the code here and in the Appendix also works on the Intel Gamma which has a hypercube architecture.)

In the first stage , 1-D FFT real-to-complex x-direction transforms are performed on each node for its portion of the data. After these transforms, the data is packed the same way as it is following the x transform in the Slab Decomposition, described above. All processors have packed data since all have an $l'=0$ mode.

After the x-direction transforms, the data must be redistributed as shown in Fig. 2. Data is exchanged only between nodes on the same mesh column as can be seen in Fig. 2. The size of the block is $ns_x*nsr_y*ns_z+1$ words, where $ns_x=n_x/nrows$, $nsr_y=n_y/nrows$, and $ns_z=n_z/ncols$. Because the exchange is only among processors in the same column, a modified version of the fully asynchronous exchange is used. Again, the order in which the processors receive the data messages is unimportant because the first word of the message contains the source node from which the proper placement for the data can be determined. The code for the asynchronous exchange among processors in the same column follows:

```
c    me = my processor id
          ityper=8000+me
          do ks=0,nrows-1
c   Post message receipt
            imsg = irecv(ityper,ibuf2,len)
c   Select destination node from those within my "column"
            kl = ks*ncols.xor.me
c   Put appropriate data into message buffer
            .
            .
            itype = 8000+kl
```

```
c  Send message
          imsgs=isend(itype,ibuffer,len,kl,0)
c  Wait for receipt of message
          call msgwait(imsg)
c  put input buffer to temporary array
          .
          .
c  Wait until outgoing message finishes
          call msgwait(imsgs)
         enddo
```

After the first block data exchange among nodes, each node has all of y data, but only partial of x and partial of z data. At this stage, only the processors in the first "row" have packed data.

Next, the 1-D FFT complex-to-complex y-direction transforms are performed. All of the nodes in the first row of the processor mesh now have packed data (the two real coefficient planes, $C(0,m,n)$ and $C(n_x/2,m,n)$ for $m=0,n_y-1,n=0,n_z-1)$ which must first be unpacked. No other data requires any special treatment. After the y-transforms, there is redundancy in the coefficients and the data is packed as it was following the y-transform in the Slab algorithm. In this case, since each processor has a portion of the z data, the z index goes from $n=0,ns_z-1$.

After the y transforms, the data is again redistributed as shown in Fig. 2. Here, data is exchanged only among processors in the same row. The size of the block is $nsx*nsc_y*ns_z+1$ words where $nsc_y=n_y/ncols$. The pseudo code for the block data exchange among nodes on the same mesh rows follows:

```
          integer nrows,ncols

          me = mynode()
          ityper=8000+me
          len = 4 + nsx*nscy*nsz*4
             .
          do ks = 0,ncols-1
             imsgr = irecv(ityper,ibuf2,len)
c exchange with processor in the same row only
             kl = ks.xor.me
c            .
c fill up the output buffer

             .
             itype = 8000+kl
             imsgs=isend(itype,ibuffer,len,kl,0)
             call msgwait(imsgr)
c put the input buffer into the proper location in the temporary array
             .
             call msgwait(imsgs)
           enddo
```

Following this exchange, all of z data and portion of x and y data will be on each node (Fig. 2). Before the 1-D FFT complex-to-complex transforms in z-direction can be performed, the packed data must be unpacked and the missing

coefficients reconstructed by complex conjugation as in the Slab Decomposition algorithm. The packed data are located on nodes whose node number is $(n_y/2+1)/nsc_y$, where $nsc_y = n_y/ncols$. The number of special nodes is 1 or 2 depending on the number of processors initially assigned to the y and z directions. Node 0 is always a special node. After the z-transforms, the forward real-to-complex 3D-FFT has been accomplished. The final data structure for the packed coefficient array is the same as in the Slab Decomposition FFT and is shown in Fig. 6.

The inverse 3D FFT complex-to-real transform is performed by doing the operations of the forward FFT in reverse. After the inverse FFT, the data is partitioned into rods as it was initially. The limitation for this data decomposition is that the maximum number of nodes the program can run on must satisfy the following condition: $(n_x/nrows > 1)$, where *nrows* is the number of nodes in the row of the mesh dimension. Source code for both the forward real-to-complex (`rfft3rc`) and reverse complex-to-real (`rfft3cr`) Rod Decomposition algorithms are included in the Appendix.

5. Timings for Slab and Rod Decomposition Parallel 3-D FFTs

Timings were done to study the efficiency of the parallel FFTs and to compare Slab and Rod Decomposition parallel 3-D FFTs to establish which is preferable for various problem and processor mesh sizes. The Slab Decomposition was found to be faster for all problem sizes. In all cases, the times presented are the time for both a forward and inverse 3D FFT.

Fig. 7 plots the time (in sec) for the Slab Decomposition FFT as a function of the number of processors for four different size data array *A(l,m,n)* for *(l,m,n) = (64,64,64), (128,128,128), (1024,32,512)* and *(512,32,1024)*. The last two have the same total number of data elements and, from Fig. 7, the FFT times were essentially the same. It can be seen that the time decreases and the number of processors increases in all cases (but clearly not linearly). The curves end where the maximum number of processor for the problem size is used. Recall that the maximum number of processors that can be used for the Slab Decomposition FFT is $nprocs=min(n_z,n_x/2)$.

Fig. 8 plots the time for the Rod Decomposition FFT as a function of the number of processors for the same four problem sizes as in Fig. 7. For each case, the curves turn over and the time begins to *increase* as the number of processors increases. This is due to a large increase in the ratio of communication to computation time as the number of processors increases. The scales on Fig. 7 and 8 have been made the same to simplify comparison. It can be seen that for all cases, the Slab Decomposition FFT is significantly faster. In Fig. 9, a direct comparison is made for the case with *A(1024,32,512)*.

From these timing results, we conclude that the Slab Decomposition FFT is always faster, even when the Rod Decomposition can utilize more processors. Thus, it may be faster to use the Slab Decomposition and leave some processors idle, assuming that they are needed for other portions of the code, as in a PIC code. However, there will be an additional overhead needed to move the data

from the full set of processors to the set needed for the Slab FFT and the trade-offs must be considered when deciding which FFT to use.

6. Use of the FFT to Solve Poisson's Equation

To present an example of how to use the parallel 3-D FFT algorithms presented above, we will use the Slab Decomposition FFT to solve Poisson's equation which arises in many plasma particle-in-cell codes (Liewer and Decyk, [1989] and Ferraro *et al*, [1993]). In electrostatic PIC codes, a charge density array $\rho(\mathbf{x})$ is calculated on a grid and one needs to solve Poisson's equation to determine the electrostatic potential $\phi(\mathbf{x})$ on the grid:

$$\nabla^2\phi(\mathbf{x}) = \rho(\mathbf{x})$$

This can be solve by using the Fourier transformed equation

$$-k^2\phi(\mathbf{k}) = \rho(\mathbf{k}) \tag{10}$$

so that the $\phi(\mathbf{k})$ can be determined algebraically from the $\rho(\mathbf{k})$. In the simulation, the infinite sum over Fourier modes is approximated as a finite sum where the shortest wavelength that can be resolved on the grid twice the grid spacing.

To solve the equation, let $\rho(l,m,n)$ be the charge array at the grid points $\mathbf{x}(l,m,n) = (x,y,z) = (l\Delta x, m\Delta y, n\Delta z)$. where $l = 1,n_x$, $m = 1,n_y$ and $n = 1,n_z$. The 3D Fourier transform coefficients of this array are

$$\rho(l',m',n') = \sum_{l=0}^{n_x-1}\sum_{m=0}^{n_y-1}\sum_{n=0}^{n_z-1}\rho(l,m,n)e^{-i\mathbf{k}_{l'm'n'}\cdot\mathbf{x}_{lmn}} \tag{11}$$

where the wave vectors are

$$\mathbf{k}(l',m',n') = 2\pi\left(\frac{l'}{n_x\Delta x}, \frac{m'}{n_y\Delta y}, \frac{n'}{n_z\Delta z}\right) \tag{12}$$

for $l' = 0, n_x/2$, $m' = 0, \pm1, \pm2, \ldots \pm n_y/2$ and $n' = \pm0, \pm1, \pm2, \ldots \pm n_z/2$.

To use a fast Fourier transform algorithm to evaluate the Fourier coefficients in Eq. 11, one makes use of the fact that

$$e^{-i2\pi m' n/n_x} = e^{-i2\pi(m'-n_x)n/n_x}$$

and computes the $\rho(l',m',n')$ for $l'=0,n_x/2$, as above , but for $m'=0, n_y-1$ and $n' = 0, n_z-1$. While this is mathematically equivalent to the original range of m' and

n', it must be remembered that to compute the values of the wave number itself in Eq. 10, one must use the original ranges for m' and n' which follow Eq. 12.

Once the Fourier coefficients are computed by calling the `sfft3rc` routine, one computes the $\phi(\mathbf{k}) = \phi(l',m',n')$ algebraically

$$\phi(l',m',n') = -\frac{1}{k^2}\rho(l',m',n')$$

where $k^2 = \mathbf{k}\cdot\mathbf{k}$ and $\mathbf{k}$ is given in Eq. 12. The inverse `sfft3cr` is then called to compute $\phi(\mathbf{x}) = \phi(l,m,n)$. The code included in the FFT package includes a routine which creates an appropriately packed table of the $\mathbf{k}$ indices (l',m',n'). The routine (kprep) returns an array ktable(i,l',m,'n') (where i=1,2,3 for x,y,z components) which is packed in a manner analogous to the packed Fourier coefficients (Fig. 6) so that the Fourier coefficient and mode numbers l',m',n' can be obtained using the same set of indices in the code. This allows the FFT routines to be used without the necessity of the user having any knowledge of the packing scheme utilized. The use for this table is shown in the pseudo code below, which uses the Slab Decomposition FFT to solve Poisson's equation for $\nabla^{-2}A(\mathbf{x})$.

```
c Pseudo code for Slab Decomposition Solution of Poisson's Equation with
c   matrix A(nx,ny,nz) the source term

c   Assumes A is in Slab Decomposition with nzs=nz/nprocs z-grid points
c   per processor

     real A(nx,ny,nzs)

c   ktable is a matrix with k indices needed for wave number evaluation
     integer ktblz(3,nx,ny,nzs)

c Prepare tables for 1D FFTs
     call sfft3prep

c  calculate packed wave number arrays,.e.g. k indices stored as modes
     call kprep(ktblz)

c  call 3dfft real-to-complex to transform A

     call sfft3rc
c Divide each element of the A array by  1/k^2

       do iz = 1,nzs
         do iy=1,ny
           do ix=1,nx+2
              xkx=2.*3.1416*ktblz(1,ix,iy,iz)/nx
              yky=2.*3.1416*ktblz(2,ix,iy,iz)/ny
              zkz=2.*3.1416*ktblz(3,ix,iy,iz)/nz
              pwr2k=zkz**2 + yky**2 + xkx**2
                if (pwr2k.eq.0) then
                  pwr2k=1.
                endif
```

```
            a(ix,iy,iz)=(1./pwr2k)*a(ix,iy,iz)
          enddo
        enddo
      enddo

c call 3dfft complex-to-real with A array as argument

      call sfft3cr

c Solution is in the array A with initial Slab Decomposition
```

The FORTRAN source codes for this sample problem can be found in the Appendix. The codes may also be obtained electronically by contacting edith@cube.jpl.nasa.gov (Edith Huang).

Acknowledgments

Support for this computing research was provided by the JPL Supercomputing Project. Two of the authors (PCL and VKD) were partially supported by the Department of Energy via the Grand Challenge Numerical Tokamak Project. The JPL Supercomputing Project is sponsored by the Jet Propulsion Laboratory and the NASA Office of Space Science and Applications. We wish to thank Warren Wayne for creating test cases on the Cray, Barbara Horner-Miller for her support, and the Caltech Supercomputing Consortium support staff, especially Heidi Lorenz-Wirzba, Sharon Brunett and Jan Lindheim, for their help.

References

John M. Dawson, Viktor Decyk, Richard Sydora and Paulett Liewer, High-Performance Computing and Plasma Physics, *Physics Today*, **46**, 64 (1993).

R. D. Ferraro, P. C. Liewer and V. K. Decyk, Dynamic Load Balancing for a 2D concurrent Plasma PIC Code, *J. Comp. Physics* (to be published, 1993).

E. Jackson, Z-S She, and S. Orzag, A Case Study in Parallel Computing: I. Homogeneous Turbulence on a Hypercube, J. Scientific Comp. **6**, 27 (1991).

G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, *Solving Problems on Concurrent Processors* , Prentice Hall, Englewood Cliffs, New Jersey, 187ff (1988)

P. C. Liewer and V. K. Decyk, A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes, *J. Comp. Physics* **85**, 302 (1989).

Figure Captions

Fig. 1 Partitioning of three dimensional data for the Slab Decomposition parallel FFT. The data is initially partitioned into slabs along the z axis so each processor initially has all x and y data for a range of z. The 1D Fourier transforms in x and y can be done in parallel in each node with no interprocessor communication. Next, the data is redistributed among the processors into slabs partitioned along the x axis. The nodes then performs the z transforms in parallel for their range of x.

Fig. 2 Partitioning of three dimensional data for the Rod Decomposition parallel FFT. The data is initially partitioned into rectangular cross section "rods" so each processor initially has all x data for a range of y and z. The 1D Fourier transforms in x are done in parallel in each node with no interprocessor communication. The data must be redistributed twice in this algorithm: first, before the y transforms and, second, before the z transforms. This algorithm was found to be slower than the Slab Decomposition FFT for all problem sizes.

Fig. 3 Bandwidth timing results (in Mbytes/sec) as a function of message size for three mesh sizes on the Intel Delta: 16x16 (o), 16x8(x) and 8x8(Δ) processors using the asynchronous scheme described in the text. Tests are for each processor exchanging a message with every other processor. The top line is the ideal bandwidth for the Delta (11 Mbytes/sec).

Fig. 4 Results of timing tests of communication schemes. Bandwidth timing results (in Mbytes/sec) as a function of Delta mesh size for three communication schemes: fully asynchronous (o), synchronous (x) and synchronous send/asynchronous receive(Δ) . In this test, each processor exchanged a 40 Mbyte message with every other processor. Times are comparable for the three schemes.

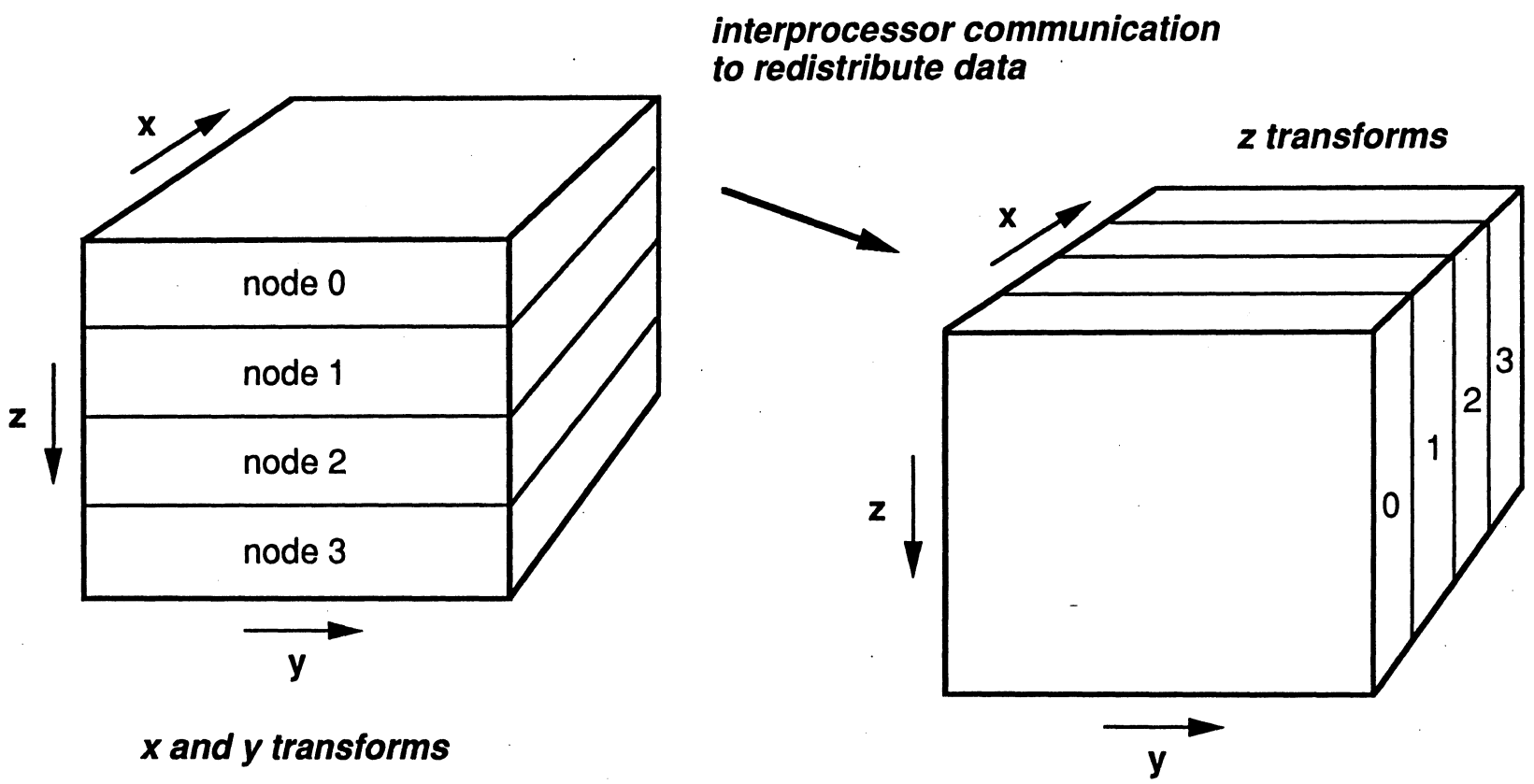
Fig. 5 Results of timing tests of communication schemes. Bandwidth timing results (in Mbytes/sec) as a function of message size for three different communication schemes and two mesh sizes (16x8 open symbols, 4x8 filled symbol): fully asynchronous (o), synchronous (squares) and synchronous send/asynchronous receive(Δ). In this test, each processor exchanged a 40 Mbyte message with every other processor.

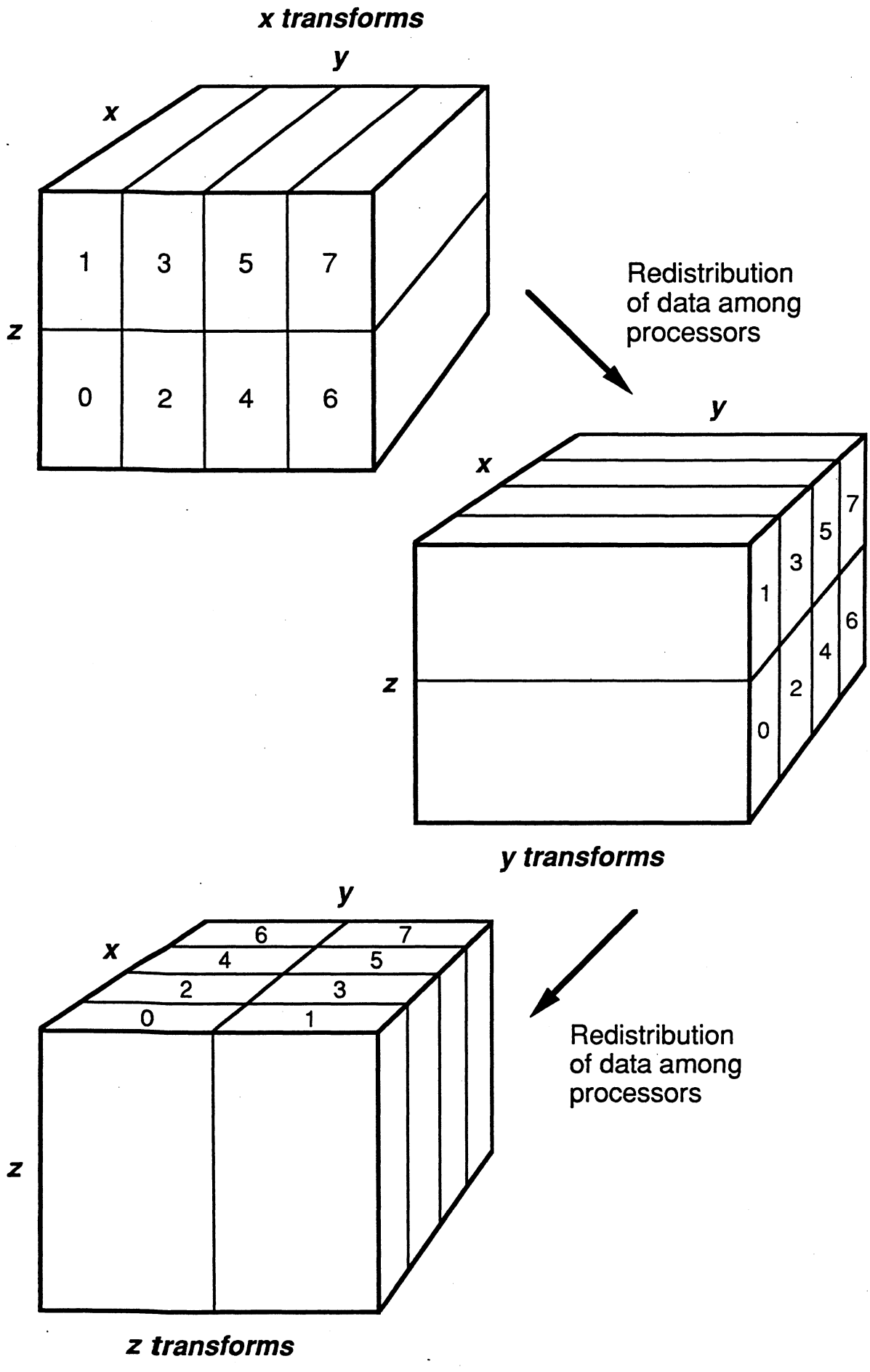
Fig. 6 "Packed" storage scheme for Fourier coefficients used in both parallel FFT algorithms, showing the three "special cases" (dotted, cross-hatched and shaded regions) discussed in the text. Aside from the special cases, the complex Fourier coefficients are stored in the original real data array A(l,m,n) with real and imaginary parts adjacent. The two planes shown here , the A(1,m,n) plane in (a) and the A(2,m,n) plane in (b), are the only planes of data which have packed coefficients. See text for explanation.

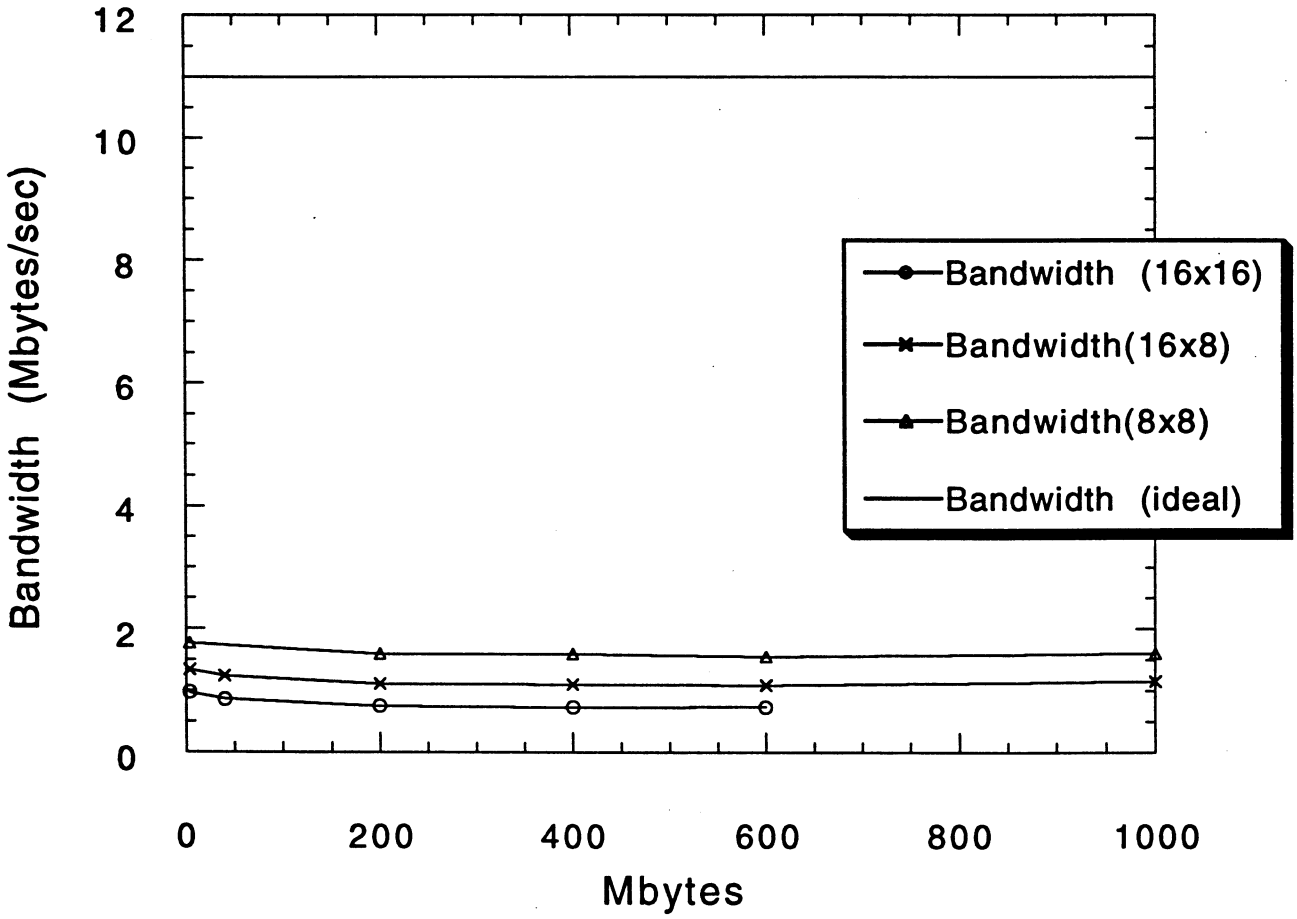
Fig. 7 Times for the Slab Decomposition parallel FFT on the Delta as a function of the number of processors used for four different sizes problems: 64x64x64 array (o), 128x129x128 array (square), 1024x32x512 array (Δ) and 512x32x1024 array(X). For all tests, times are for a forward FFT plus a reverse FFT.

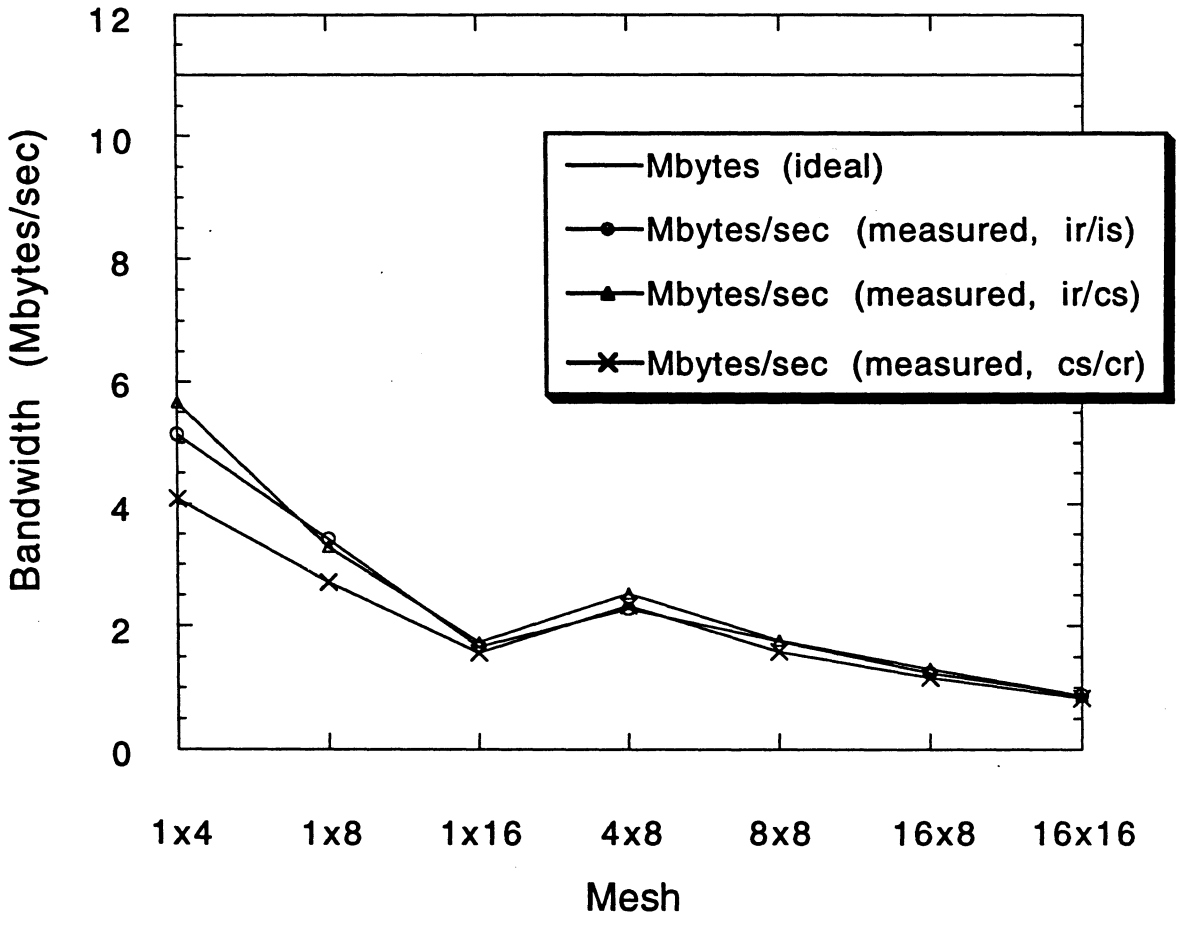
Fig. 8 Times for the Rod Decomposition parallel FFT on the Delta as a function of the number of processors used for the same four problems as in Fig. 7: 64x64x64 array (o), 128x129x128 array (square), 1024x32x512 array (Δ) and 512x32x1024 array(X). At some number of processors, the times begin to *increase* as the number of processors increases because of the addition communication costs. For all problems, the Rod Decomposition was slower than the Slab Decomposition.

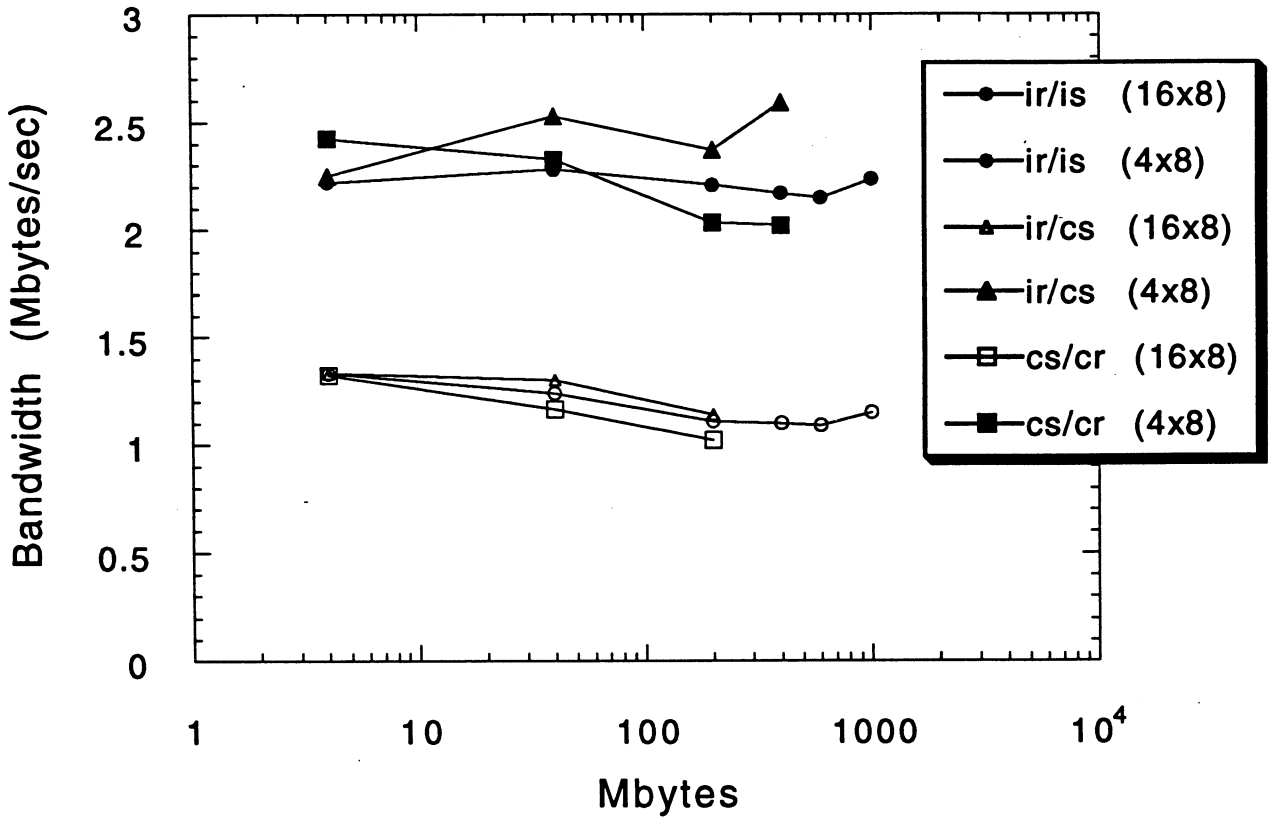
Fig. 9 Direct comparison of Slab and Rod Decomposition FFTs for a (128,128,128) matrix. Note that the Slab Decomposition utilizing its maximum number of nodes is faster even when the Rod Decomposition can utilize more nodes.

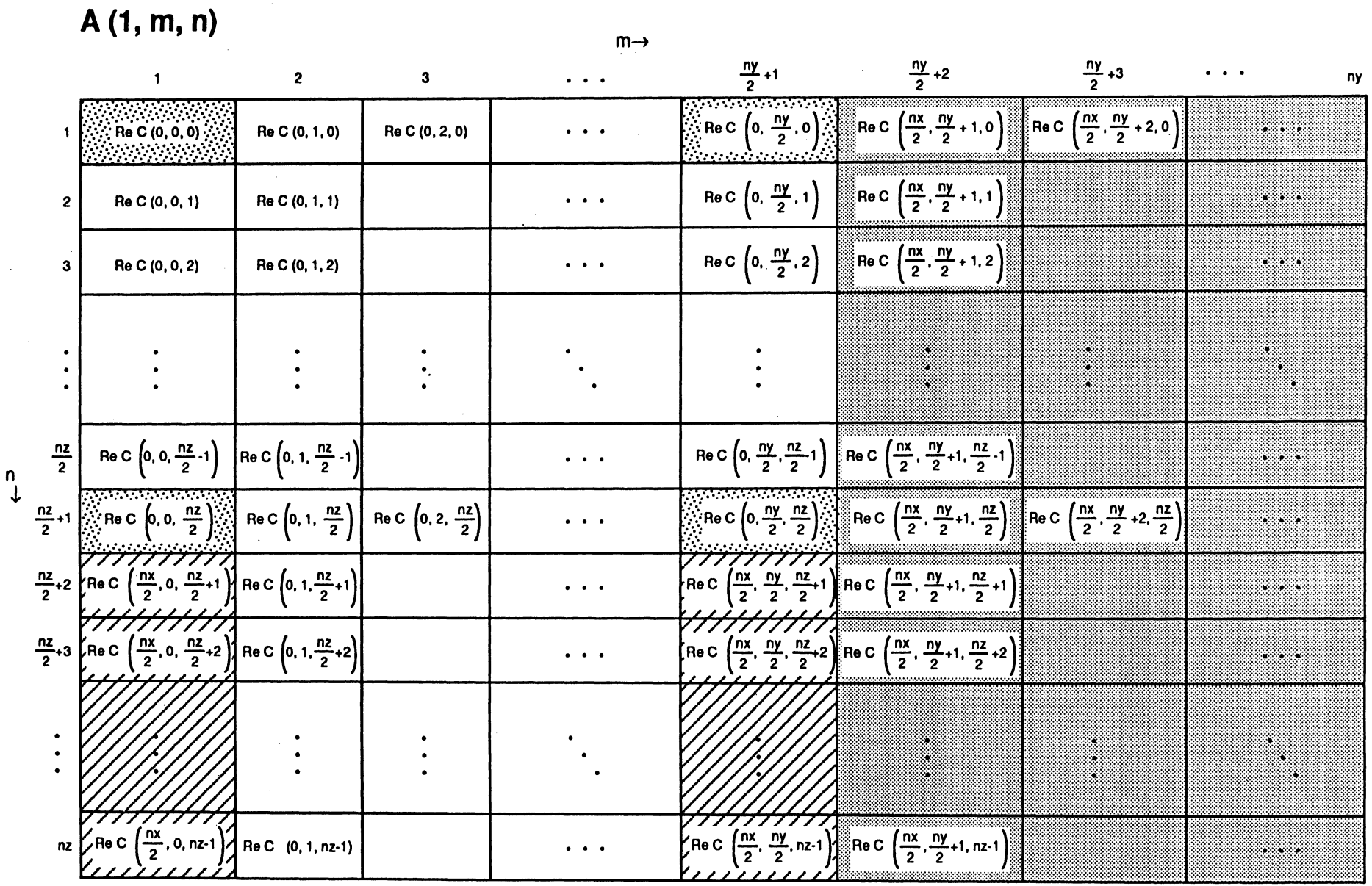

interprocessor communication
to redistribute data
z transforms
x
node 0
node 1
node 2
node 3
z
y
x and y transforms
x
z
y
0
1
2
3


x transforms
y
x
z
1
3
5
7
0
2
4
6
Redistribution of data among processors
y
x
z
1
3
5
7
0
2
4
6
y transforms
y
x
z
6
7
4
5
2
3
0
1
Redistribution of data among processors
z transforms

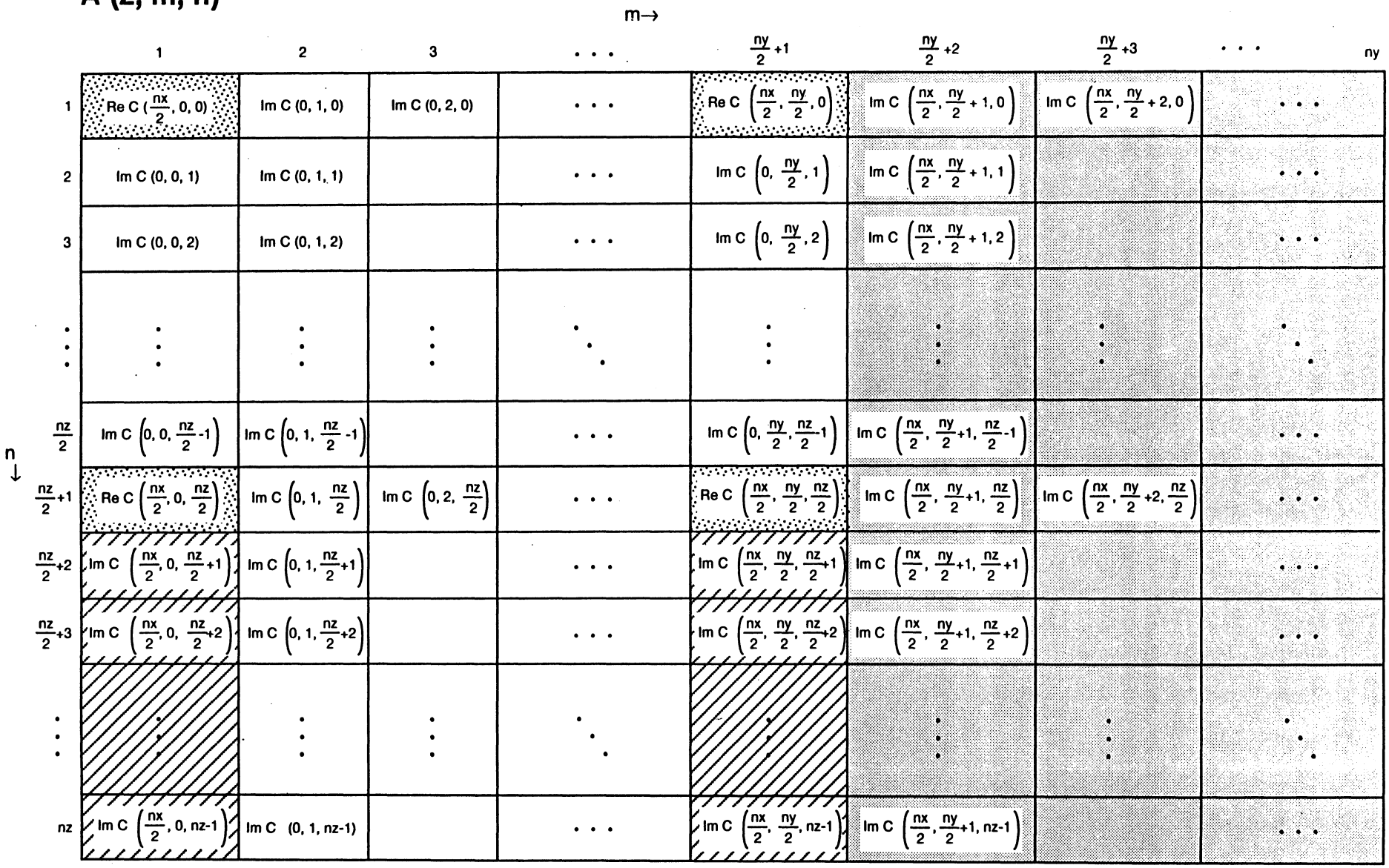

12
10
8
6
4
2
0
Bandwidth (Mbytes/sec)
0
200
400
600
800
1000
Mbytes
Bandwidth (16x16)
Bandwidth(16x8)
Bandwidth(8x8)
Bandwidth (ideal)

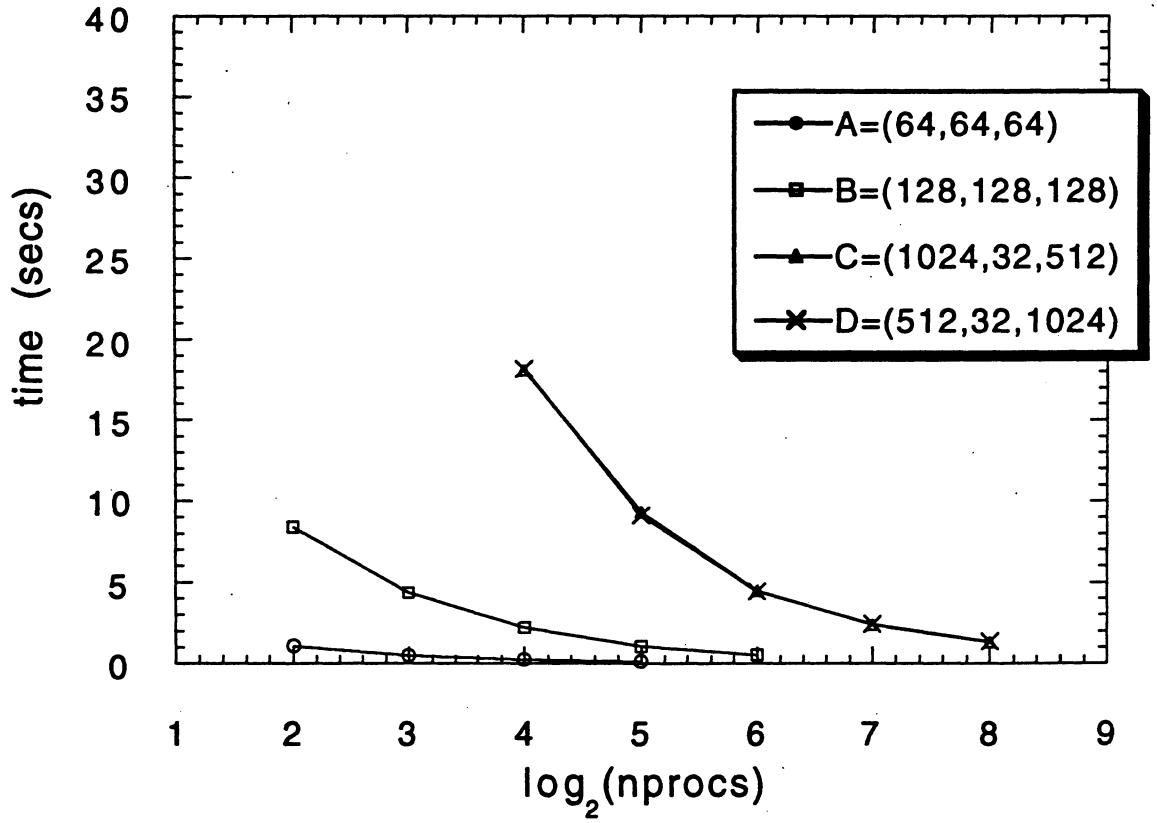

12
10
8
6
4
2
0
Bandwidth (Mbytes/sec)
Mbytes (ideal)
Mbytes/sec (measured, ir/is)
Mbytes/sec (measured, ir/cs)
Mbytes/sec (measured, cs/cr)
1x4
1x8
1x16
4x8
8x8
16x8
16x16
Mesh

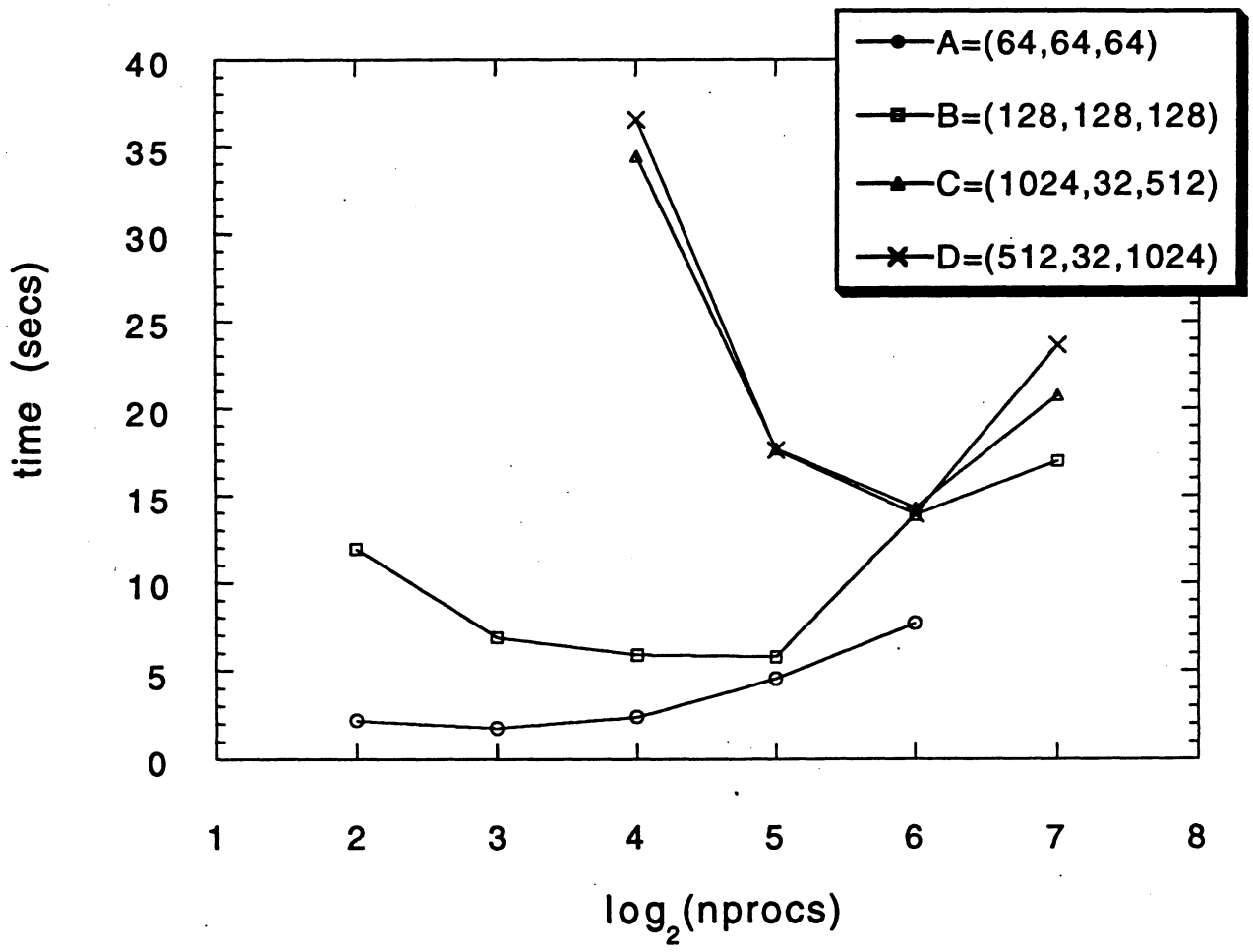

Bandwidth (Mbytes/sec)
3
2.5
2
1.5
1
0.5
0
1
10
100
1000
10^4
Mbytes
ir/is (16x8)
ir/is (4x8)
ir/cs (16x8)
ir/cs (4x8)
cs/cr (16x8)
cs/cr (4x8)

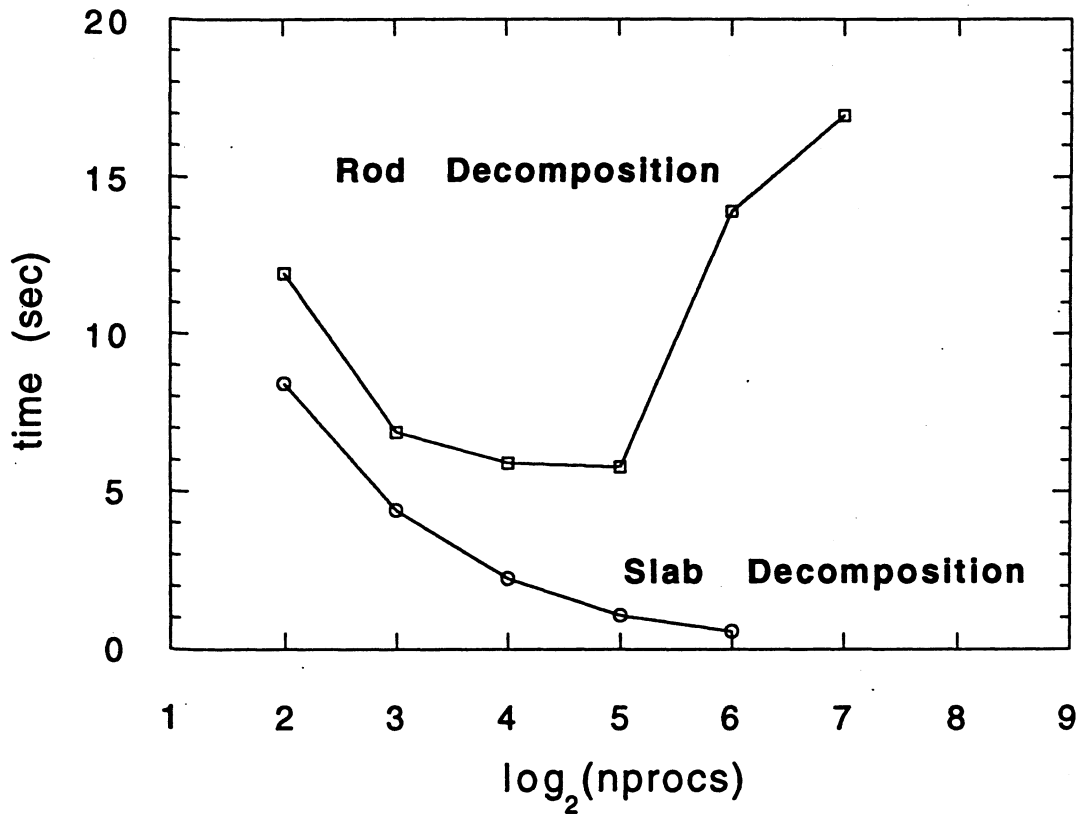

A (1, m, n)
m→
n
1
2
3
. . .
ny/2 +1
ny/2 +2
ny/2 +3
. . .
ny
1
2
3
nz/2
nz/2 +1
nz/2 +2
nz/2 +3
nz
Re C (0, 0, 0)
Re C (0, 1, 0)
Re C (0, 2, 0)
Re C (0, ny/2, 0)
Re C (nx/2, ny/2 + 1, 0)
Re C (nx/2, ny/2 + 2, 0)
Re C (0, 0, 1)
Re C (0, 1, 1)
Re C (0, ny/2, 1)
Re C (nx/2, ny/2 + 1, 1)
Re C (0, 0, 2)
Re C (0, 1, 2)
Re C (0, ny/2, 2)
Re C (nx/2, ny/2 + 1, 2)
Re C (0, 0, nz/2 -1)
Re C (0, 1, nz/2 -1)
Re C (0, ny/2, nz/2 -1)
Re C (nx/2, ny/2 +1, nz/2 -1)
Re C (0, 0, nz/2)
Re C (0, 1, nz/2)
Re C (0, 2, nz/2)
Re C (0, ny/2, nz/2)
Re C (nx/2, ny/2 +1, nz/2)
Re C (nx/2, ny/2 +2, nz/2)
Re C (nx/2, 0, nz/2 +1)
Re C (0, 1, nz/2 +1)
Re C (nx/2, ny/2, nz/2 +1)
Re C (nx/2, ny/2 +1, nz/2 +1)
Re C (nx/2, 0, nz/2 +2)
Re C (0, 1, nz/2 +2)
Re C (nx/2, ny/2, nz/2 +2)
Re C (nx/2, ny/2 +1, nz/2 +2)
Re C (nx/2, 0, nz-1)
Re C (0, 1, nz-1)
Re C (nx/2, ny/2, nz-1)
Re C (nx/2, ny/2 +1, nz-1)

A (2, m, n)


m→
n↓
1
2
3
. . .
ny/2 +1
ny/2 +2
ny/2 +3
. . .
ny
1
2
3
nz/2
nz/2 +1
nz/2 +2
nz/2 +3
nz
Re C (nx/2, 0, 0)
Im C (0, 1, 0)
Im C (0, 2, 0)
Re C (nx/2, ny/2, 0)
Im C (nx/2, ny/2 + 1, 0)
Im C (nx/2, ny/2 + 2, 0)
Im C (0, 0, 1)
Im C (0, 1, 1)
Im C (0, ny/2, 1)
Im C (nx/2, ny/2 + 1, 1)
Im C (0, 0, 2)
Im C (0, 1, 2)
Im C (0, ny/2, 2)
Im C (nx/2, ny/2 + 1, 2)
Im C (0, 0, nz/2 -1)
Im C (0, 1, nz/2 -1)
Im C (0, ny/2, nz/2 -1)
Im C (nx/2, ny/2 +1, nz/2 -1)
Re C (nx/2, 0, nz/2)
Im C (0, 1, nz/2)
Im C (0, 2, nz/2)
Re C (nx/2, ny/2, nz/2)
Im C (nx/2, ny/2 +1, nz/2)
Im C (nx/2, ny/2 +2, nz/2)
Im C (nx/2, 0, nz/2 +1)
Im C (0, 1, nz/2 +1)
Im C (nx/2, ny/2, nz/2 +1)
Im C (nx/2, ny/2 +1, nz/2 +1)
Im C (nx/2, 0, nz/2 +2)
Im C (0, 1, nz/2 +2)
Im C (nx/2, ny/2, nz/2 +2)
Im C (nx/2, ny/2 +1, nz/2 +2)
Im C (nx/2, 0, nz-1)
Im C (0, 1, nz-1)
Im C (nx/2, ny/2, nz-1)
Im C (nx/2, ny/2 +1, nz-1)


40
35
30
25
20
15
10
5
0
time (secs)
1
2
3
4
5
6
7
8
9
log2(nprocs)
A=(64,64,64)
B=(128,128,128)
C=(1024,32,512)
D=(512,32,1024)


A=(64,64,64)
B=(128,128,128)
C=(1024,32,512)
D=(512,32,1024)
time (secs)
0
5
10
15
20
25
30
35
40
1
2
3
4
5
6
7
8
log2(nprocs)


20
15
10
5
0
time (sec)
Rod Decomposition
Slab Decomposition
1
2
3
4
5
6
7
8
9
log2(nprocs)

Appendix

EXPRESS™ Cubix Source Code

1. Makefile for Poisson Solve Program (Makefile)

2. Poisson Solve Main Program utilizing Slab Decomposition 3DFFT (psolve.f).

3. Routines to Initialize Parallel Computer for Slab (meshinit.f)

4. Routines to Initialize Constants for FFTs for Slab (kprep.f)

5. Routines for Forward and Reverse Slab Decomposition FFTs (sfft.f)

6. Include Files for Slab FFT (slabinc.include).

7. Routines to Initialize Parallel Computer for Rod (meshinita.f)

8. Routines to Initialize Constants for FFTs for Rod (krprep.f)

9. Routines for Forward and Reverse Rod Decomposition FFTs (rfft.f)

10. Include Files for Rod FFT (rodinc.include).

```
#!/bin/csh
CUBIXF=kprep.f meshinit.f
RESTF=sfft.f
MAIN1=psolve.f

all: psolve

psolve:$(MAIN1) $(CUBIXF) $(RESTF)
        xf77 -opsolve $(MAIN1) $(CUBIXF) $(RESTF) -Knoieee -kcubix -lkmath
```

```
c*******************************************
      program psolve
c This program solves Poisson equation
c This is a CUBIX program
c This program use logical node map table
c
      character*40 file_n

c include file para1s.inc must be changed for different
c size of the array A and mesh size
c para1s.inc is set to run on 2x2 mesh (4 nodes)
c for this example
      include 'para1s.inc'
      include 'simqb3ds.inc'

c  Z dimension should be the largest
      dimension f(nxx,nyy,nzs1)
      dimension A(nxx,nyy,nzs1),tmp(nyy)
      dimension AA(nxs1,nyy,nzz)
      dimension dx(4*nxx+4),dy(4*nyy+4),dz(4*nzz+4)
c work array
      dimension t(nxx,nyy,nzs1)
      dimension dcopy(nt)
      dimension ktblz(3,nxs1,nyy,nzz)
c After the forward transfoms the data in array A are
c represented as in array AA(nxs1,nyy,nzz)
      equivalence(A(1,1,1),AA(1,1,1))

      nx=nxx
      ny=nyy
      nz=nzz
      nxs=nxs1
      nzs=nzs1
c
c initialize Express, and set up logical ring table
      call meshinit
c
c   From physical node number 'me' to the logical
c   node number in the ring
c
      procnum=ptable(me+1)-1

c  Find out the mesh dimension
c
      call mypart(irows,icols)

c Input file name
c
      file_n='sinput.dat'
c length of the input file name
      lenf=10
c read the input file
c
         call inputf(A,nx,ny,nzs,nprocs,procnum,tmp,file_n,lenf)

c set coefficients
c Get ready to do fft transforms
      call sfft3prep(dx,dy,dz,nx,ny,nz)

c calculate kspace table
c the output table ktblz contains the kspace
      call kprep(ktblz,nxs,nx,ny,nz,procnum)
```

```
c 3D-FFT forward transform, real to complex
      call sfft3rc(A,t,nx,ny,nz,dx,dy,dz,dcopy)

c multiply each k mode by a k square factor
         do k = 1, nz
           do j= 1,ny
             do i = 1,nxs
               xkx=2.*3.1416*ktblz(1,i,j,k)/float(nx)
               yky=2.*3.1416*ktblz(2,i,j,k)/float(ny)
               zkz=2.*3.1416*ktblz(3,i,j,k)/float(nz)
               pwr2k=xkx**2+yky**2+zkz**2
               if (pwr2k.eq.0) then
                  pwr2k=1.
               endif
c must use array AA to get the correct location
               AA(i,j,k)=(1./pwr2k)*AA(i,j,k)
             enddo
            enddo
          enddo

c Invers 3D-FFT, complex to real
c
      call sfft3cr(A,t,nx,ny,nz,dx,dy,dz,dcopy)
      stop
      end

c--------------------------------------------------------------------------
       subroutine inputf(f,nx,ny,nzs,nprocs,procnum,
     1 tmp,filen,len)
       dimension f(nx,ny,nzs),tmp(*)
        character*40 filen
        integer procnum

        open(unit=10,file=filen(1:len),status='old')
         do loop = 1,nprocs
           do i =1, nzs
             do j=1, nx
              read(10,1100)( tmp(k),k=1,ny)
              if (procnum.eq.(loop-1)) then
                do k=1,ny
                  f(j,k,i)=tmp(k)
                enddo
               endif
1100    format(1x 8 f10.7)
             enddo
             read(10,1000)
1000    format(/)
           enddo
         enddo
         return
         end
```

```
c*********************************************************************
c---------------------------------------------------------------------
c
c  This routine initialize Express and setup ring topology
c  for physical node and logical node map table, this is
c  an attempt to make the mesh communication faster on
c  the Delta. It did make too much difference, probably
c  should not use it in the future
c
      subroutine meshinit
      include 'para1s.inc'
      include 'simqb3ds.inc'
      include 'express.inc'
      dimension ienviron(4)

c    initialize Express OS
       call kxinit

c  get environ variables
        call kxpara(ienviron)
        me=ienviron(1)
        nprocs=ienviron(2)

c setup ring topology. Create physical node and logical
c node map table to enable faster communication,for
c the Delta machine
c both ptable and idx are output
        call ring2(ptable,idx,nprocs)
        return
        end


c*******************************************
c_______________________________________________
c
      subroutine ring2(ix,iy,nprocs)
      integer  rows,cols,ia(512),iy(512)
      dimension ix(*)

c  A simple ring mapping program by Edith Huang (JPL)
c  This program will map physical nodes to a ring topology
c  This program will only work with rows equal to power of 2
c  ix contains logical node number
c  iy contains mesh node number
c  This is an attempt to make the communication faster on
c  the Delta.

      call mypart(rows,cols)
c     write(6,*)' rows, cols:',rows,cols

     if (mod(rows,2).ne.0.and.rows.ne.1) then
       write(6,*)' Ring2: number of rows is not a power of 2 '
       stop
     endif

      i=0

      do irow=1,rows
        if (irow.eq.1) then
          do icol=1,cols
            i=i+1
            ix(i)=i
          enddo
```

```
           else
              if (mod(irow,2).eq.0) then
                 i=i+1
                 ix(i) = nprocs-irow+2
                 ip=irow*(cols-1)
                 do icol=1,cols-1
                    i=i+1
                    ix(i)= ip-icol+2
                 enddo
               else
                  i=i+1
                  ix(i)=nprocs-irow+2
                  ip=(irow-1)*(cols-1)+1
                  do icol=1,cols-1
                    i=i+1
                    ix(i)=ip+icol
                  enddo
                endif
              endif
           enddo

c
c generate physical node table iy
c
           do i=1,nprocs
             ia(i)=ix(i)
           enddo
c iy contains the physical node table
           call sorti1(ia,nprocs,iy)
           return
           end

c
c-------------------------------------------------------------------------------
c
          subroutine sorti1(a,n,idx)
          integer a(*),c,t,idx(*)

          if (n-1) 2,6,10
2       write(6,22)
22      format(/' Error return from ''sorti1'' -- N is less than 1')
        stop
6       idx(1)=1
        return
10      do i=1,n
        idx(i)=i
        enddo
        n2=n/2
        n21=n2+2
        ict=1
        i=2
11      n1=n21-i
        nn=n
        ik=n1
15      c=a(ik)
        ic=idx(ik)
100     jk=2*ik
        if(jk.gt.nn) go to 140
        if (jk.eq.nn) go to 120
        if (a(jk+1).le.a(jk)) go to 120
110     jk=jk+1
120     if (a(jk).le.c) go to 140
200     a(ik)=a(jk)
        idx(ik)=idx(jk)
```

```
        ik=jk
        go to 100
140     a(ik)=c
        idx(ik)=ic
        go to (3,45),ict
3       if (i.ge.n2) goto 35
        i=i+1
        go to 11
35      ict=2
        np2 = n + 2
        i=2
37      n1 = np2 - i
        nn=n1
        ik = 1
        go to 15
45      continue
        t = a(1)
        a(1) = a(n1)
        a(n1) = t
        it = idx(1)
        idx(1) = idx(n1)
4       idx(n1) = it
        if (i.ge.n) go to 55
        i=i+1
        go to 37
55      return
        end
```

```
c***********************************************
c-----------------------------------------------------------------------
c This routine creates the K space table for every element
c in the A array which is dimensioned A(nxs,ny,nz).
c nxs is nxx/nprocs

      subroutine kprep(ktbl,nx,nxx,ny,nz,procnum)
      dimension ktbl(3,nx,ny,1)
      integer  procnum
c    nx -- local
c    nxx -- global
c    ny -- global
c    nz -- global
c ktbl -- is the table contains the K modes

      do k=1,nz
        do j=1,ny
          do i=1,nx
            ktbl(1,i,j,k)=0
            ktbl(2,i,j,k)=0
            ktbl(3,i,j,k)=0
          enddo
         enddo
      enddo

      if ( procnum. ne. 0) go to 100
c For Node 0 only
c for jy=1 and jy=ny/2 and jx=1 and jx=2
c
c Z dimension for kz=nz/2+2,nz
c kx=0, kx=nx/2, ky=ny/2
c
      nyl=ny/2+1

      call unpackk(ktbl,nx,nxx,ny,nz,nyl)
c
c Z dimension for kz=nz/2+2,nz
c kx=0, kx=nx/2, ky=0
c
      nyl=1

      call unpackk(ktbl,nx,nxx,ny,nz,nyl)

c Do the rest normally

      nx1=1
      nx2=1
      ly1=2
      ly2=ny/2

      call doworkk(ktbl,nx,nxx,ny,nz,procnum,nx1,nx2,ly1,ly2)

      ly1=ny/2+2
      ly2=ny


      call doworkyk(ktbl,nx,nxx,ny,nz,procnum,nx1,nx2,ly1,ly2)

100   continue
c
c  do the rest of rows, columns normally
c
      if (procnum.eq.0) then
```

```
         nx1=3
      else
         nx1=1
      endif

      nx2=nx
      ly1=1
      ly2=ny

      call doworkk(ktbl,nx,nxx,ny,nz,procnum,nx1,nx2,ly1,ly2)

      return
      end

C-------------------------------------------------------------

      subroutine unpackk(ktbl,nx,nxx,ny,nz,nyl)
      dimension ktbl(3,nx,ny,nz)
c nx -- local dimension
c nxx -- global dimension
c ny --  global dimension
c nz --  global dimension


c for nx/2,ny/2,nz/2
c
      ix=nxx/2+1
      kx=ix-1
      ky=nyl-1
      if (nyl.gt.(ny/2+1)) then
         ky=nyl-1-ny
      endif

      do 9000 k=nz/2+2,nz
        kz=k-1-nz

       ktbl(1,1,nyl,k)=kx
       ktbl(2,1,nyl,k)=ky
       ktbl(3,1,nyl,k)=kz

       ktbl(1,2,nyl,k)=kx
       ktbl(2,2,nyl,k)=ky
       ktbl(3,2,nyl,k)=kz

9000  continue
        ktbl(1,2,nyl,1)=kx
        ktbl(2,2,nyl,1)=ky
        ktbl(3,2,nyl,1)=0

        k=nz/2+1
        kz=k-1

        ktbl(1,2,nyl,nz/2+1)=kx
        ktbl(2,2,nyl,nz/2+1)=ky
        ktbl(3,2,nyl,nz/2+1)=kz

      ix=1
      kx=ix-1
      ky=nyl-1

      do 9010 k=1,nz/2+1
        kz=k-1

       ktbl(1,1,nyl,k)=kx
```

```
          ktbl(2,1,nyl,k)=ky
          ktbl(3,1,nyl,k)=kz

          if (k.ne.1.and.k.ne.(nz/2+1)) then
           ktbl(1,2,nyl,k)=kx
           ktbl(2,2,nyl,k)=ky
           ktbl(3,2,nyl,k)=kz
          endif
9010      continue
         return
         end

C-----------------------------------------------------------------

         subroutine doworkk(ktbl,nx,nxx,ny,nz,procnum,nx1,nx2,ly1,ly2)
         dimension ktbl(3,nx,ny,nz)
         integer procnum
c nx -- local dimension
c nxx -- global
c ny -- global
c nz -- global
c kx1 -- lowest kx in the node

         kxt=procnum*nx
         do 9060 j1=nx1,nx2,2
            kx1=(kxt+j1)/2+1
            kx=kx1-1
         do 9060 jy=ly1,ly2
          ky=jy-1
           if (jy.gt.(ny/2+1)) then
             ky=jy-1-ny
           endif
         do 9040 k=1,nz
            kz=k-1
            if (k.gt.(nz/2+1)) then
               kz=k-1-nz
            endif

         ktbl(1,j1,jy,k)=kx
         ktbl(2,j1,jy,k)=ky
         ktbl(3,j1,jy,k)=kz

         ktbl(1,j1+1,jy,k)=kx
         ktbl(2,j1+1,jy,k)=ky
         ktbl(3,j1+1,jy,k)=kz
9040     continue

9060     continue
         RETURN
         end

C-----------------------------------------------------------------

         subroutine doworkyk(ktbl,nx,nxx,ny,nz,procnum,nx1,nx2,ly1,ly2)
         dimension ktbl(3,nx,ny,nz)
         integer procnum
c nx -- local dimension
c nxx -- global
c ny -- global
c nz -- global
c kx1 -- lowest kx in the node
c
c  This routine for node 0 only
c
```

```
      do 9060 j1=nx1,nx2,2
         if (j1.eq.1) then
            kx1=nxx/2+1
         else
           kx1=j1/2+1
         endif

         kx=kx1-1

      do 9060 jy=ly1,ly2
        ky=jy-1
        if (jy.gt.(ny/2+1)) then
           ky=jy-1-ny
        endif

      do 9040 k=1,nz
        kz=k-1
        if (k.gt.(nz/2+1)) then
           kz=k-1-nz
        endif

       ktbl(1,j1,jy,k)=kx
       ktbl(2,j1,jy,k)=ky
       ktbl(3,j1,jy,k)=kz

       ktbl(1,j1+1,jy,k)=kx
       ktbl(2,j1+1,jy,k)=ky
       ktbl(3,j1+1,jy,k)=kz

9040   continue

9060   continue
       return
       end
```

```
c*********************************************************************
c---------------------------------------------------------------------
c
c   3D fft of real arrays, done by doing 1d ffts along each dimension
c
c       There are 3 routines in this set
c               sfft3prep       - set up internal tables for the 3d fft
c               sfft3rc         - forward fft (real to complex)
c               sfft3cr         - inverse fft (complex to real)
c
c       The storage pattern used for the complex result of an fft on
c       real data is the one used by Decyk.
c               real : f(nx,ny,nz)
c               cmplx: f(nx/nprocs,ny,nz)
c
c  5/25/92          Edith Huang
c  Edith Huang implemented the 3d-fft according to Robert Ferraro's
c  2d parallel fft and Intel's 1d fft
c---------------------------------------------------------------------
c   this program prepares internal tables for the 3d fft. It should
c   be called only once before using sfft3rc or sfft3cr.
c
c     nx      - global number of points in x (1st) dimension
c     ny      - global number of points in y (2nd) dimension
c     nz      - global number of points in z (3rd) dimension
c
c               sfft3rc :
c                       iinput  : f(nx,ny,nzs)
c                       returns : f(nxs,ny,nz)
c               sfft3cr :
c                       input   : f(nxs,ny,nz)
c                       returns : f(nx,ny,nzs)
c
      subroutine sfft3prep(dx,dy,dz,nx,ny,nz)
      dimension dx(*),dy(*),dz(*)

c setup internal tables
      call fft2sprep(dx,dy,nx,ny)
      call cfft1d(f,nz,0,dz)
      return
      end
c---------------------------------------------------------------------
c   this program does a three dimensional fft of real input data

      subroutine sfft3rc(f,t,nx,ny,nz,dumx,dumy,dumz,dcopy)
c
      include 'para1s.inc'
      include 'simqb3ds.inc'
c
c     f       - Real array dimensioned f(nx,ny,nzs)
c               input and output
c     nx      - number of points in x (1st) dimension
c     ny      - number of points in y (2nd) dimension
c     nz      - number of points in z (3rd) dimension
c     dumx,dumy,dumz -- internal tables
c     dcopy   - work vector
c     t       - real array same size as f, work array
      real f(nx,ny,1), dumx(*),dumy(*),dumz(*),dcopy(*)
      real t(*)
      nzs = nz/nprocs
      nxs = nx/nprocs

      if (nzs .lt. 1) then
```

```
         write(*,*) 'nzs too small nzs= ',nzs
         call exit
       elseif (nxs.lt. 2) then
         write(*,*) 'nxs too small = ',nxs
         call exit
       endif
c    real to complex 1d fft - k(nx/2) returned in f(2,k) (Im part of k0)
c    k's greater than nx/2 are redundant since f(k) = CC f(-k)
c    Do the 2dfft
       do 15 k=1,nzs
         call fft2src(f(1,1,k),nx,ny,dumx,dumy,dcopy)
15     continue
c   do blockdata exchange
       if (nprocs.ne.1) then
         call xchpart3dc(f,t,nx,ny,nz,1)
       endif
c   do Z directory transform
       call fftzd(f,nxs,ny,nz,dumz,dcopy)
       return
       end
c-------------------------------------------------------------------------
c    The program does the inverse 3d fft on complex data stored in
c    array f
       subroutine sfft3cr(f,t,nx,ny,nz,dumx,dumy,dumz,dcopy)
c
c      f       - Real array dimensioned f(nx,ny,nzs)
c                input and output
c      nx      - number of points in x (1st) dimension
c      ny      - number of points in y (2nd) dimension
c      nz      - number of points in z (3rd) dimension
c      dumx,dumy,dumz -- internal tables
c      dcopy   - work vector
c      t      - real array same size as f, work array
       include 'para1s.inc'
       include 'simqb3ds.inc'
c
       dimension f(nx,ny,1), dumx(*),dumy(*),dcopy(*)
       dimension t(*),dumz(*)

       nxs=nx/nprocs
       nzs=nz/nprocs
       if (nzs.lt. 1) then
         write(*,*) 'nzs too small nzs = ',nzs
         call exit
       elseif (nxs .lt. 2) then
         write(*,*) 'nxs too small = ',nxs
         call exit
       endif
c Inverse transform the z dimension
       call ifftzd(f,nxs,ny,nz,dumz,dcopy)

c Do blockdata exchange
       if (nprocs.ne.1) then
         call xchpart3dc(f,t,nx,ny,nz,3)
       endif
c Do inverse 2dfft
       do 15  k=1,nzs
         call fft2scr(f(1,1,k),nx,ny,dumx,dumy,dcopy)
15     continue
       return
       end
c-------------------------------------------------------------------------
       subroutine fftzd(f,nx,ny,nz,dumz,dum)
c This routine is called by sfft3rc to do the forward transforms in
```

```
c Z direction.
      include 'para1s.inc'
      include 'simqb3ds.inc'
c     f   -- real array f(nxs,ny,nz)
c     nx (nxs) -- local dimension
c     ny --   global
c     nz --   global
c    dumz -- internal table
c    dum -- work vector

       dimension f(nx,ny,nz),dum(*),dumz(*)

c do complex fft on all columns of f except if I am processor 0
c (f(2*i+1),f(2*i+2)) ith mode ( real, imag)
        if (procnum.eq.0) then
c  Do the ix >=3 first
          nx1=3
          nx2=nx
          ny1=1
          ny2=ny
          call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)

c I am processor 0. Therefore we have to take care of the 1st 2 columns
c For kx=0,ky=0 and kx=nx/2, ky=0, for all z
c pure real
          ny1=1
          call packrl(f,nx,ny,nz,dumz,dum,ny1)

c For kx=0,ky=ny/2 and kx=nx/2,ky=ny/2 for all z
c pure real
          ny1=ny/2+1
          call packrl(f,nx,ny,nz,dumz,dum,ny1)

c Do the rest of the first 2 columns in node 0
          nx1=1
          nx2=1
          ny1=2
          ny2=ny/2
          call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)

          ny1=ny/2+2
          ny2=ny
          call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)
        else
c Not processor 0, do everything normally
          nx1=1
          nx2=nx
          ny1=1
          ny2=ny
          call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)
         endif
         return
         end

C-----------------------------------------------------------------
          subroutine ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)
c This routine is called by fftzd
c  f  -- is input and output array
c dumz -- internal table
c dum  -- work vector
          dimension f(nx,ny,nz),dumz(*),dum(*)

          do 9030 j1=nx1,nx2,2
          do 9030 jy=ny1,ny2
```

```
          do 9010 k=1,nz
            k2=k+k
            dum(k2-1)=f(j1,jy,k)
            dum(k2)=f(j1+1,jy,k)
9010      continue
c   complex to complex fft
            call fft1cc(dum,nz,-1,dumz)
          do 9020 k=1,nz
            k2=k+k
            f(j1,jy,k)=dum(k2-1)
            f(j1+1,jy,k)=dum(k2)
9020      continue
9030      continue
         return
         end

C---------------------------------------------------------------
        subroutine packrl(f,nx,ny,nz,dumz,dum,nyl)
c This routine is called by fftzd
c f  -- real input,output array
c dumz -- internal table
c dum  -- work vector

        dimension f(nx,ny,nz),dumz(*),dum(*)
c transform in z dimension
c nyl -- lower case letter L
c
        do 9040 k=1,nz
          k2=k+k
          dum(k2-1)=f(1,nyl,k)
          dum(k2)=0.
9040    continue
c complex to complex
        call fft1cc(dum,nz,-1,dumz)
c Save the 1st half of the result and retrieving the nx=nx/2 and ny=0
        do 9050 k=1,nz/2+1
          k2=k+k
          f(1,nyl,k)=dum(k2-1)
          dum(k2-1)=f(2,nyl,k)
          f(2,nyl,k)=dum(k2)
          dum(k2)=0.
9050    continue

c Transform in z for n=nx/2
        do 9060 k=nz/2+2,nz
          k2=k+k
          dum(k2-1)=f(2,nyl,k)
          dum(k2)=0.
9060    continue
c   complex to complex fft
        call fft1cc(dum,nz,-1,dumz)

        do 9070 k=nz/2+2,nz
          k2=k+k
          f(1,nyl,k)=dum(k2-1)
          f(2,nyl,k)=dum(k2)
9070    continue
        f(2,nyl,1)=dum(1)
        f(2,nyl,nz/2+1)=dum(nz+1)
        return
        end
c-------------------------------------------------------------------------

        subroutine ifftzd(f,nx,ny,nz,dumz,dcopy)
```

```
c This program is called by sfft3cr to do the inverse
c Z direction fft
c f in real array, input and output array
c dumz -- internal table
c dcopy -- work vector

        dimension f(nx,ny,nz),dumz(*),dcopy(*)
c    nx -- local
c    ny -- global
c    nz -- global
        include 'para1s.inc'
        include 'simqb3ds.inc'
c
        if ( procnum. ne. 0) go to 100
c Processor 0 has to worry about the first 2 column of the packed data
c for jy=1 and jy=ny/2 and jx=1 and jx=2,for all z
c the data are pure real values
C Do inverse tranforms in Z direction for
c kx=0, kx=nx/2, ky=ny/2,for all z
        ny1=ny/2+1
        call unpack(f,nx,ny,nz,dumz,dcopy,ny1)
c
c DO inverse transform in Z dimension for
c kx=0, kx=nx/2, ky=0
c
        ny1=1
        call unpack(f,nx,ny,nz,dumz,dcopy,ny1)
c
c Do the rest of the data normally
        nx1=1
        nx2=1
        ly1=2
        ly2=ny/2
        call dorest(f,nx,ny,nz,dumz,dcopy,nx1,nx2,ly1,ly2)
c
        ly1=ny/2+2
        ly2=ny
        call dorest(f,nx,ny,nz,dumz,dcopy,nx1,nx2,ly1,ly2)
100     continue
c
c  do the rest of rows, columns normally
        if (procnum.eq.0) then
           nx1=3
        else
           nx1=1
        endif
        nx2=nx
        ly1=1
        ly2=ny
        call dorest(f,nx,ny,nz,dumz,dcopy,nx1,nx2,ly1,ly2)
        return
        end

C-------------------------------------------------------------
        subroutine unpack(f,nx,ny,nz,dumz,dum,ny1)
c This routine is called by ifftzd to unpack some of the packed
c data.
c f -- real array, input and output
c dumz -- internal table
c dum -- work vector
        dimension f(nx,ny,nz),dumz(*),dum(*)

c for nx/2,ny/2,nz/2
c
```

```
        do 9000 k=nz/2+2,nz
          k2=k+k
          dum(k2-1)=f(1,nyl,k)
          dum(2*nz+3-k2)=f(1,nyl,k)
          dum(k2)=f(2,nyl,k)
          dum(2*nz+4-k2)=-f(2,nyl,k)
9000    continue
        dum(1)=f(2,nyl,1)
        dum(2)=0.
        dum(nz+1)=f(2,nyl,nz/2+1)
        dum(nz+2)=0
c Complex to complex
        call fft1cc(dum,nz,1,dumz)
c
c Pick up data for nx=0, ny=ny/2 and put the result in the right
c loaction
c
        do 9010 k=1,nz/2+1
          k2=k+k
          dum(k2)=f(2,nyl,k)
          f(2,nyl,k)=dum(k2-1)
          dum(k2-1)=f(1,nyl,k)
9010    continue
        do 9020 k=nz/2+2,nz
           k2=k+k
           dum(k2)=-dum(2*nz+4-k2)
           f(2,nyl,k) = dum(k2-1)
           dum(k2-1) = dum(2*nz+3-k2)
9020     continue
         dum(2)=0.
         dum(nz+2)=0.
c Complex to complex fft
        call fft1cc (dum,nz,1,dumz)
        do 9030 k=1,nz
          k1=k+k-1
          f(1,nyl,k)=dum(k1)
9030    continue
        return
        end

C---------------------------------------------------------------
        subroutine dorest(f,nx,ny,nz,dumz,dum,nx1,nx2,ly1,ly2)
c This routine is called by ifftzd to do normal data
c f -- real array, input and output
c dumz -- internal table
c dum -- work vector
        dimension f(nx,ny,nz),dumz(*),dum(*)

        do 9060 j1=nx1,nx2,2
        do 9060 jy=ly1,ly2
        do 9040 k=1,nz
          k2=k+k
          dum(k2-1)=f(j1,jy,k)
          dum(k2)=f(j1+1,jy,k)
9040    continue
c
c complex to complex fft
        call fft1cc(dum,nz,1,dumz)
        do 9050 k=1,nz
          k2=k+k
          f(j1,jy,k)=dum(k2-1)
          f(j1+1,jy,k)=dum(k2)
9050    continue
9060    continue
```

```
      return
      end


c*************************************************************
c-------------------------------------------------------------------------
c
c   2D fft of real arrays, done by doing 1d ffts along each dimension
c
c       There are 3 routines in this set
c               fft2sprep       - set up internal tables for the 2d fft
c               fft2src         - forward fft (real to complex)
c               fft2scr         - inverse fft (complex to real)
c
c       The storage pattern used for the complex result of an fft on
c       real data is the one used by Decyk.
c               real : f(nx,ny)
c               cmplx: f(nx/nprocs,ny)
c This program was originally written by Robert Ferraro
c Modified   5/22/92 by Edith Huang  To do scalar 2D fft only and
c to use Intel's 1dfft routines
c
c-------------------------------------------------------------------------
c   this program prepares internal tables for the 2d fft. It should
c   be called only once before using fft2src or fft2scr.
c
c     f      - real array
c     nx      - global number of points in x (1st) dimension
c     ny      - global number of points in y (2nd) dimension
c
c               fft2src :
c                       input : f(nx,ny)
c                       returns : f(nx,ny)
c               fft2scr :
c                       input : f(nx,ny)
c                       returns : f(nx,ny)
c
      subroutine fft2sprep(dx,dy,nx,ny)
      dimension dx(*),dy(*)
c  setup internal tables
      call fft1rc(f,nx,0,dx,d)
      call fft1cc(f,ny,0,dy)
      return
      end
c-------------------------------------------------------------------------
c   this program does a two dimensional fft of real input data

      subroutine fft2src(f,nx,ny,dumx,dumy,dcopy)
c
c     f       - Real array dimensioned f(nx,ny)
c     nx      - number of points in x (1st) dimension
c     ny      - number of points in y (2nd) dimension
c     dumx,dumy -- internal tables
c     dcopy  -- work vector
c
      real f(nx,ny), dumx(*),dumy(*),dcopy(*)
      if (ny .lt. 1) then
         write(*,*) 'ny too small = ',ny
         call exit
      elseif (nx .lt. 2) then
         write(*,*) 'nx too small = ',nx
         call exit
      endif
```

```
c first transform in the x direction

      do 9000 k = 1, ny
c   real to complex 1d fft - k(nx/2) returned in f(2,k) (Im part of k0)
c   k's greater than nx/2 are redundant since f(k) = CC f(-k)
      call fft1rc(f(1,k),nx,-1,dumx,dcopy)
 9000 continue

c   now transform in the y direction.
      call fftsyd(f,nx,ny,dumy,dcopy)
      return
      end

c--------------------------------------------------------------------------
c   This program does the inverse 2d fft on complex data stored in
c   f array. It is a real array
      subroutine fft2scr(f,nx,ny,dumx,dumy,dcopy)
c
c     f       - Real array dimensioned f(nx,ny)
c     nx      - number of points in x (1st) dimension
c     ny      - number of points in y (2nd) dimension
c     dumx,dumy -- internal arrays
c     dcopy  - work vector
      dimension f(nx,ny), dumx(*),dumy(*),dcopy(*)
      if (ny .lt. 1) then
         write(*,*) 'ny too small = ',ny
         call exit
      elseif (nx .lt. 2) then
         write(*,*) 'nx too small = ',nx
         call exit
      endif

c  inverse transform the y dimension.
      call ifftsyd(f,nx,ny,dumy,dcopy)

c finally transform in the x direction
      do 9070 k = 1, ny
c   inverse of real to complex 1d fft
      call fft1rc(f(1,k),nx,1,dumx,dcopy)
 9070 continue
      return
      end

c--------------------------------------------------------------------
c  This routine does transforms in y direction
      subroutine fftsyd(f,nx,ny,dumy,dum)
c
c This routine is called by fft2src
c     f       - Real array dimensioned f(nx,ny)
c     nx      - number of points in x (1st) dimension
c     ny      - number of points in y (2nd) dimension
c     dumy -- internal table
c     dum -- work vector
c
      dimension f(nx,ny), dum(*),dumy(*)
c
c transform in y for 0 < n < nx/2
c Handle the first 2 rows differently, because it represent nx=0 and
c nx=nx/2 mode. The data are real for these two mode
c   (f(2*i+1),f(2*i+2)) is the (Real, Imag) ith mode
c   eph
      nx1=3
      do 9030 j1 = nx1, nx, 2
      do 9010 k = 1, ny
```

```
      k2 = k + k
      dum(k2-1) = f(j1,k)
      dum(k2) = f(j1+1,k)
 9010 continue
c   complex to complex fft
      call fft1cc(dum,ny,-1,dumy)

      do 9020 k = 1, ny
      k2 = k + k
      f(j1,k) = dum(k2-1)
      f(j1+1,k) = dum(k2)
 9020 continue

 9030 continue

c transform in y for nx = 0 mode
c   note that nx = 0 mode is purely real
      do 9040 k = 1, ny
      k2 = k + k
      dum(k2-1) = f(1,k)
      dum(k2) = 0.
 9040 continue

c   complex to complex fft
      call fft1cc(dum,ny,-1,dumy)
c
c   store the result as (Real, Imag) while retrieving the nx/2 mode
c   (which is also purely real). Note that the upper half of the result
c   in dum (the -ky modes) are redundant.

      do 9050 k = 1, ny/2 + 1
      k2 = k + k
      f(1,k) = dum(k2-1)
      dum(k2-1) = f(2,k)
      f(2,k) = dum(k2)
      dum(k2) = 0.
 9050 continue
c transform in y for n = nx/2
      do 9060 k = ny/2 + 2, ny
      k2 = k + k
      dum(k2-1) = f(2,k)
      dum(k2) = 0.
 9060 continue
c   complex to complex fft
      call fft1cc(dum,ny,-1,dumy)
      do 9070 k = ny/2 + 2, ny
      k2 = k + k
      f(1,k) = dum(k2-1)
      f(2,k) = dum(k2)
 9070 continue
      f(2,1) = dum(1)
      f(2,ny/2+1) = dum(ny+1)
 9090 return
      end

c--------------------------------------------------------------------
c   this routine does the inverse transform in the y direction
      subroutine ifftsyd(f,nx,ny,dumy,dum)
c
c This routine is called by fft2scr
c     f       - Real array dimensioned f(nx,ny)
c     nx      - number of points in x (1st) dimension
c     ny      - number of points in y (2nd) dimension
c     dumy - internal table
```

```
c     dum  - work vector
c
      dimension f(nx,ny), dum(*),dumy(*)
c first inverse transform in y for n = nx/2
      do 9000 k = ny/2 + 2, ny
      k2 = k + k
      dum(k2-1) = f(1,k)
      dum(2*ny+3-k2) = f(1,k)
      dum(k2) = f(2,k)
      dum(2*ny+4-k2) = -f(2,k)
 9000 continue

      dum(1) = f(2,1)
      dum(2) = 0.
      dum(ny+1) = f(2,ny/2+1)
      dum(ny+2) = 0.
c   complex to complex fft
      call fft1cc(dum,ny,1,dumy)

c transform in y for nx = 0
      do 9010 k = 1, ny/2 + 1
      k2 = k + k
      dum(k2) = f(2,k)
      f(2,k) = dum(k2-1)
      dum(k2-1) = f(1,k)
 9010 continue
      do 9020 k = ny/2 + 2, ny
      k2 = k + k
      dum(k2) = -dum(2*ny+4-k2)
      f(2,k) = dum(k2-1)
      dum(k2-1) = dum(2*ny+3-k2)
 9020 continue
      dum(2) = 0.
      dum(ny+2) = 0.

c   complex to complex fft
      call fft1cc(dum,ny,1,dumy)
      do 9030 k = 1, ny
      k1 = k + k - 1
      f(1,k) = dum(k1)
 9030 continue
9035  continue

c transform in y for 0 < n < nx/2
c   do complex fft on all columns of f except the first two rows
c   (f(2*i+1),f(2*i+2)) is the (Real, Imag) ith mode
         nx1 = 3
      do 9060 j1 = nx1, nx, 2
      do 9040 k = 1, ny
      k2 = k + k
      dum(k2-1) = f(j1,k)
      dum(k2) = f(j1+1,k)
 9040 continue
c   complex to complex fft
      call fft1cc(dum,ny,1,dumy)
      do 9050 k = 1, ny
      k2 = k + k
      f(j1,k) = dum(k2-1)
      f(j1+1,k) = dum(k2)
 9050 continue
 9060 continue
      return
      end
```

```
c-----------------------------------------------------------------------
c
      subroutine fft1rc(a,n,isigx,work,dcopy)
c  fft1rc is called by fft2sprep,fft2src, fft2scr
c       a       - real array dimensioned a(n)
c       n       - dimension of a, must be a power of 2
c       isign   - 0     do setup of internal tables
c               - -1    do fft (real to complex)
c               - 1     do inverse fft (complex to real)
c       work    - real array dimensioned at least work(4*n+3), used
c                 for holding internal tables
c       dcopy  - work vector
c
c       fft1rc should be called once with isign = 0, at which time
c       internal tables are set up in the work array.  After this,
c       fft1rc may be called as many times as needed to do fft's and
c       inverse fft's on arrays with the same dimension as passed
c       to the first call with isign = 0.  Of course, the contents of
c       work must not be disturbed after the first call to fft1rc with
c       isign = 0
c
c  This subroutine is mainly to mask the complications of calling
c  fft1 directly
c
      dimension a(n),work(n*4+3),dcopy(*)

      if (isigx.eq.0) then
          call scfft1d(a,n,isigx,work)
      else
         call fft1pkrc(a,n,isigx,work,dcopy)
      endif
      return
      end

c
c-----------------------------------------------------------------------
c
        subroutine fft1pkrc(a,n,isigx,work,dcopy)
c This routine is called by fft1rc
c      a -- real array
c      work -- internal table
c      dcopy -- work vector
c
        dimension a(n),work(*),dcopy(*)
        if (isigx.eq.-1) then
          do i=1,n
            dcopy(i)=a(i)
          enddo
            call scfft1d(dcopy,n,isigx,work)
          do i=1,n
            a(i)=dcopy(i)
          enddo
            a(2)=dcopy(n+1)
         else
          do i=1,n
            dcopy(i)=a(i)
          enddo
            dcopy(2)=0
            dcopy(n+1)=a(2)
            dcopy(n+2)=0
            call csfft1d(dcopy,n,isigx,work)

           do i=1,n
              a(i)=dcopy(i)
```

```
          enddo
        endif
        return
        end

c-----------------------------------------------------------------------
      subroutine fft1cc(a,n,isigx,work)
c This routine is called by ftcomplex,dorest,packrl,fft2sprep,
c fft2src,ifftsyd.
c Real to complex fft from Intel's 1d fft
c       a       - complex array dimensioned a(n) to be fft'd
c       n       - dimension of complex array a
c       isign   - 0     setup internal tables
c               - -1    do fft on a
c               - 1     do inverse fft on a
c
c       fft1cc should be called once with isign = 0, at which time
c       internal tables are set up in the work array.  After this,
c       fft1cc may be called as many times as needed to do fft's and
c       inverse fft's on complex arrays with the same dimension as passed
c       to the first call with isign = 0.  Of course, the contents of
c       work must not be disturbed after the first call to fft1cc with
c       isign = 0
c
      dimension a(n),work(3+n)
c     complex a
      if (isigx .ne. 0) goto 10
        call cfft1d(a,n,isigx,work)
      return
10      call cfft1d(a,n,isigx,work)
      return
      end


c**************************************************

c-----------------------------------------------------------------------
        subroutine xchpart3dc(f,t,nx,ny,nz,i)
c   This routine does blockdata exchange among processors
c   This routine is called by sfft3rc and sfft3cr
c
c   This routine was originally written by Paulett Liewer for 2dfft
c   Modified by Edith Huang to work for the 3dfft on the Delta
c
c   input : f           - the array being exchanged
c           t           - workspace same dimension as f
c           nx,ny,nz    - global dimensions of f
c           i           - which exchange is being done :
c               = 1     partitioned in z to partitioned in x
c               = 3     partitioned in x to partitioned in z
c
      dimension f(*)
      dimension t(*)
      include 'para1s.inc'
      include 'simqb3ds.inc'
      real abufr(ibufsize-1)
      integer ibufr(ibufsize)
      real abuffer(ibufsize-1)
      equivalence (abuffer,ibuffer(2)),(ip,ibuffer(1))
      equivalence(abufr,ibufr(2)),(ip1,ibufr(1))
      me=mynode()
      nxs = nx/nprocs
      nzs = nz/nprocs
c  exchange partitioning.  redistribute array among processors
```

```
c   len is bytes length of message
       len = 4+nxs*ny*nzs*4

c  buffer up each subblock of f for transmission.  prepend my proc
c  number to the message for reference at the recieving end.
      ityper=8000+me
      ip = procnum

      do 40 kl=0,nprocs-1
        k=kl.xor.procnum
c  post to receive
      msgid=irecv(ityper,ibufr,len)

       if (i .eq. 1) then
c  option 1 : have all x's and y's for some z's of f(x,y,z).
c  Redistribute so
c  that I have all y's and z's for some x's
         ix = nxs*k
         do 10 jz=0,nzs-1
         do 10 jy=0,ny-1
         do 10 jx=1,nxs
10          abuffer(jz*nxs*ny+jy*nxs+jx) =
     1    f(jz*nx*ny+jy*nx+jx+ix)

       else
c  option 3: have all y's and z's for some x's of f(x,y,z).
c  Redistribute so
c  that I have all x's and  y's for some z's
         ix = nxs*ny*nzs*k
         do 15 j=1,nxs*ny*nzs
15       abuffer(j) = f(j+ix)
       endif

c now send messages
c destination is the processor with procnum = k;
      idest =idx(k+1)-1
      itype = 8000+idest
      imsgs= isend(itype,ibuffer, len, idest,0)

c  now collect the incoming messages
c  doesn't matter what order they are read
      call msgwait(msgid)

c Put the received data in the right place
c
      if (i .eq. 1) then
         ix = nxs*ny*nzs*ip1
         do 25 j=1,nxs*ny*nzs
25       t(ix+j) = abufr(j)
      else
         ix = nxs*ip1
         do 30 jz=0,nzs-1
         do 30 jy=0,ny-1
         do 30 jx=1,nxs
30       t(ix+jx+jy*nx+jz*nx*ny)=
     1   abufr(jz*nxs*ny+jy*nxs+jx)
      endif
      call msgwait(imsgs)
40    continue
      if (i.eq.1) then
        do k=1,nxs*ny*nz
          f(k)=t(k)
        enddo
      else
```

```
        do k=1,nx*ny*nzs
          f(k)=t(k)
        enddo
      endif
      return
      end
```

```
c*****************************************
c include files -- 3 include files
c*****************************************

c____  express.inc
            common/xpress/nocare,norder, nonode, ihost,ialnod,ialprc



c***********************************

c____ simqb3ds.inc

c-----------------------------------------------------------------------
c
c
c     parameter (ibufsize=2**18+1)
      parameter (ibufsize=nxs1*nyy*nzs1+1)
      integer nprocs,procnum,iproc,ptable
c procnum is the logical node number in the ring and me is the node number
c in the mesh
      common /simqb/ nprocs,procnum,ibuf2(ibufsize),
     1               ibuffer(ibufsize),me
      common /mapnode/ptable(512),idx(512)


c*****************************************

c_____para1s.inc
cThis include file is setup for 2 by 2 mesh and array size A(8,8,8)

c This file must be changed when the size of array A is changed
c and the mesh size is changed.
c Assuming A(nxx,nyy,nzz), mesh size (irow,icol)
c nprocs=irow*icol
c nxs1=nxx/nprocs, nzs1=nzz/nprocs
c
c 2x2 mesh 2x2 mesh 4 nodes
c a(8,8,8)
      parameter (nxx=8,nyy=8,nzz=8,nxs1=2,nzs1=2)
c
c 16x32 mesh 512 node
c a(1024,32,512)
c     parameter (nxx=1024,nyy=32,nzz=512,nxs1=2,nzs1=1)

c nt= 2*maximun of (nxx,nyy,nzz)
c     parameter(nt=2*nyy)
c     parameter(nt=2*nxx)
c     parameter(nt=2*nzz)
c find the  max of (nxx,nyy)
c dcopy(nt) -- work vector
c dumx(4*nxx+4),dumy(4*nyy+4),dumz(4*nzz+4)--internal tables
      parameter(nt1=((nxx*(nxx/nyy)+nyy*(nyy/nxx))/(nxx/nyy+nyy/nxx)))
c find the max of (nt1,nzz)
      parameter(nt2=((nt1*(nt1/nzz)+nzz*(nzz/nt1))/(nt1/nzz+nzz/nt1)))
      parameter(nt=2*nt2)
```

```
c-----------------------------------------------------------------------
c
c  This routine initialize Express
c
        subroutine meshinita
        include 'para2.inc'
        include 'simqb3da.inc'
        include 'express.inc'
        dimension ienviron(4)

c    initialize Express OS
          call kxinit

c  get environ variables
           call kxpara(ienviron)
           me=ienviron(1)
           nprocs=ienviron(2)
           call mypart(irows,icols)
           return
           end
```

```
C-------------------------------------------------------------

        subroutine krprep(ktbl,nx,nxx,ny,nyy,nz,me,
     1  icols,nprocs)

        dimension ktbl(3,nx,ny,nz)
c
c    nx(nxs) -- local
c    nxx     -- global
c    ny(nysc) -- local
c    nyy -- global
c    nz -- global
c    f(nxs,nys,nz)
c    icols -- mesh dimension
c
c
c special processors have to worry about the first 2 column of the packed data
c
c initialize ktable
c
         do k=1,nz
          do j=1,ny
           do i=1,nx
            ktbl(1,i,j,k)=0
            ktbl(2,i,j,k)=0
            ktbl(3,i,j,k)=0
           enddo
          enddo
         enddo

          iy2=nyy/2+1
          node=iy2/ny
c         call kmulti(6)
c         write(6,*)' node=',node

          if ((me.ne.0).and. (me .ne.node)) go to 100
c     if ((me.eq.0).or.(me.eq.node)) then
c for jy=1 and jy=ny/2 and jx=1 and jx=2
c the data are pure real values
c
c Z dimension for kz=nz/2+2,nz
c kx=0, kx=nx/2, ky=ny/2
c
          if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
                nyl=ny/2+1
               call unpackkf(ktbl,nx,nxx,ny,nyy,nz,me,icols,nyl)
          endif
c
c Z dimension for kz=nz/2+2.nz
c kx=0, kx=nx/2, ky=0
c
          nyl=1

        call unpackkf(ktbl,nx,nxx,ny,nyy,nz,me,icols,nyl)

c Do the rest of the data normally

        nx1=1
        nx2=1
        ly1=2
        if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
          ly2=nyy/2
        else
          ly2=ny
```

```
        endif

        call dorestskf(ktbl,nx,nxx,ny,nyy,nz,me,icols,
     1  nx1,nx2,ly1,ly2)

        if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
           ly1=nyy/2+2
           ly2=nyy
           call dorestykf(ktbl,nx,nxx,ny,nyy,nz,me,icols,
     1     nx1,nx2,ly1,ly2)
        endif
c     endif

100     continue
c
c  do the rest of rows, columns normally
c
         nx2=nx
         ly1=1
         ly2=ny
         if ((me.eq.0).or.(me.eq.node)) then
           nx1=3
         call dorest0kf(ktbl,nx,nxx,ny,nyy,nz,me,node,icols,
     1  nx1,nx2,ly1,ly2)

         else
           nx1=1
         call dorestkf(ktbl,nx,nxx,ny,nyy,nz,me,node,icols,
     1  nx1,nx2,ly1,ly2)
         endif

200     continue
        return
        end

C---------------------------------------------------------------

        subroutine unpackkf(ktbl,nx,nxx,ny,nyy,nz,me,
     1  icols,nyl)
c nx -- local
c nxx -- global
c ny -- local
c nyy -- global
c icols -- column size of the mesh
c kyl -- lowest ky in this node
c
c This routine only called by special nodes
c
        dimension ktbl(3,nx,ny,nz)


c for nx/2,ny/2,nz/2
c
c find the y coordinate for this node in the mesh
c

c       ixc = mod(me,icols)
c       iyc= me/icols
c       nyll=nyl
        kyt=nyl
        if (me.ne.0) then
c          nyll=nyy/2+1
           kyt=nyy/2+1
        endif
```

```
c       ky1 = ixc*ny
c       kyt = ky1 + nyll

        ix=nxx/2+1
c       xkx=2.*3.1416*(ix-1)/nxx
        kx=ix-1

        ky=kyt-1
        if (kyt.gt.(nyy/2+1)) then
           ky=ky-nyy
        endif

c       yky=2.*3.1416*(ky)/nyy

        do 9000 k=nz/2+2,nz
          kz=k-1-nz
c         zkz=2.*3.1416*(kz)/nz
c         pwr2k=xkx**2+yky**2+zkz**2
c         if ( pwr2k.eq.0) then
c            pwr2k=1.
c         endif

c    f(1,nyl,k)=f(1,nyl,k)/pwr2k
          ktbl(1,1,nyl,k)=kx
          ktbl(2,1,nyl,k)=ky
          ktbl(3,1,nyl,k)=kz

c    f(2,nyl,k)=f(2,nyl,k)/pwr2k
          ktbl(1,2,nyl,k)=kx
          ktbl(2,2,nyl,k)=ky
          ktbl(3,2,nyl,k)=kz

9000    continue
c       zkz=0.
        kz=0
c       pwr2k=xkx**2+yky**2+zkz**2
c       f(2,nyl,1)=f(2,nyl,1)/pwr2k
        ktbl(1,2,nyl,1)=kx
        ktbl(2,2,nyl,1)=ky
        ktbl(3,2,nyl,1)=kz

        k=nz/2+1
        kz=k-1
c       zkz=2.*3.1416*(k-1)/nz

c       pwr2k=xkx**2+yky**2+zkz**2
c       f(2,nyl,nz/2+1)=f(2,nyl,nz/2+1)/pwr2k
        ktbl(1,2,nyl,nz/2+1)=kx
        ktbl(2,2,nyl,nz/2+1)=ky
        ktbl(3,2,nyl,nz/2+1)=kz

c       ix=1
c       xkx=0.
         kx=0

cc       yky=2.*3.1416*(nyl-1)/nyy
c
        do 9010 k=1,nz/2+1
c         zkz=2.*3.1416*(k-1)/nz
          kz=k-1
c         pwr2k=xkx**2+yky**2+zkz**2
c         if (pwr2k.eq.0) then
c            pwr2k=1.
```

```
c          endif
c          f(1,nyl,k)=f(1,nyl,k)/pwr2k
           ktbl(1,1,nyl,k)=kx
           ktbl(2,1,nyl,k)=ky
           ktbl(3,1,nyl,k)=kz

           if (k.ne.1.and.k.ne.(nz/2+1)) then
c     f(2,nyl,k)=f(2,nyl,k)/pwr2k
             ktbl(1,2,nyl,k)=kx
             ktbl(2,2,nyl,k)=ky
             ktbl(3,2,nyl,k)=kz
           endif
9010     continue

         return
         end
C-----------------------------------------------------------------

        subroutine dorestskf(ktbl,nx,nxx,ny,nyy,nz,me,icols,
     1  nx1,nx2,ly1,ly2)
c nx --  local
c nxx -- global
c ny --  local
c nyy -- global
c nz --  global
c icols -- mesh column size
c kx1 -- lowest kx in this node
c ky1 -- lowest ky in this node
c ixc -- x coordinate of this node
c iyc -- y coordinate of this node
c
         dimension ktbl(3,nx,ny,nz)

c find the coordinate of this node
c

         if (me.eq.0) then
            kx1=1
            ky1=0
         else
            kx1=nxx/2+1
            ky1=nyy/2
         endif
         do 9060 jy=ly1,ly2
            kyt=ky1+jy
            ky = kyt  - 1
            if (kyt. gt.(nyy/2 + 1)) then
               ky=ky-nyy
            endif

         do 9040 k=1,nz
            kz = k - 1
            if (k.gt.(nz/2 + 1)) then
               kz = k-1-nz
            endif
            kx=kx1-1

c          xkx = 2.*3.1416*(kx1-1)/nxx
c           yky = 2.*3.1416*ky/nyy
c           zkz = 2.*3.1416*kz/nz
c           pwr2k = xkx**2+yky**2+zkz**2
c           if (pwr2k.eq.0) then
c              pwr2k=1.
c           endif
```

```
c          f(1,jy,k)=(1./pwr2k)*f(1,jy,k)
c          f(2,jy,k)=(1./pwr2k)*f(2,jy,k)

           ktbl(1,1,jy,k)=kx
           ktbl(2,1,jy,k)=ky
           ktbl(3,1,jy,k)=kz

           ktbl(1,2,jy,k)=kx
           ktbl(2,2,jy,k)=ky
           ktbl(3,2,jy,k)=kz

9040     continue

9060     continue
         return
         end


C-----------------------------------------------------------------

       subroutine dorest0kf(ktbl,nx,nxx,ny,nyy,nz,me,node,icols,
     1  nx1,nx2,ly1,ly2)
c nx --  local
c nxx -- global
c ny --  local
c nyy -- global
c nz --  global
c icols -- mesh column size
c kx1 -- lowest kx in this node
c ky1 -- lowest ky in this node
c ixc -- x coordinate of this node
c iyc -- y coordinate of this node
c
         dimension ktbl(3,nx,ny,nz)

c find the coordinate of this node
c
         ixc=mod(me,icols)
         iyc = me/icols
         kxt = iyc*nx
         ky1 = ixc*ny

         do 9060 j1=nx1,nx2,2
            kx1 = (kxt+j1)/2 + 1
            kx=kx1-1
         do 9060 jy=ly1,ly2
            kyt=ky1+jy
            ky = kyt  - 1
            if (kyt. gt.(nyy/2 + 1)) then
               ky=ky-nyy
            endif

         do 9040 k=1,nz
            kz = k - 1
            if (k.gt.(nz/2 + 1)) then
               kz = k-1-nz
            endif

c        f(j1,jy,k)=(1./pwr2k)*f(j1,jy,k)
c        f(j1+1,jy,k)=(1./pwr2k)*f(j1+1,jy,k)

          ktbl(1,j1,jy,k)=kx
          ktbl(2,j1,jy,k)=ky
```

```
        ktbl(3,j1,jy,k)=kz

        ktbl(1,j1+1,jy,k)=kx
        ktbl(2,j1+1,jy,k)=ky
        ktbl(3,j1+1,jy,k)=kz
9040    continue

9060    continue
        return
        end

C-----------------------------------------------------------------

       subroutine dorestkf(ktbl,nx,nxx,ny,nyy,nz,me,node,icols,
     1  nx1,nx2,ly1,ly2)
c nx --  local
c nxx -- global
c ny --  local
c nyy -- global
c nz --  global
c icols -- mesh column size
c kx1 -- lowest kx in this node
c ky1 -- lowest ky in this node
c ixc -- x coordinate of this node
c iyc -- y coordinate of this node
c
        dimension ktbl(3,nx,ny,nz)

c find the coordinate of this node
c
        ixc=mod(me,icols)
        iyc = me/icols
        kxt = iyc*nx
        ky1 = ixc*ny

        do 9060 j1=nx1,nx2,2
           kx1 = (kxt+j1)/2 + 1
c Test to see if me is greater than node (special node which
c contains the packed data), and me is on the 1st row of the
c mesh (iyc=0)
         if (j1.eq.1.and.icols.gt.1.and.me.gt.node
     1      .and.iyc.eq.0) then
           kx1=nxx/2+1
         endif
           kx=kx1-1
        do 9060 jy=ly1,ly2
           kyt=ky1+jy
           ky = kyt  - 1
           if (kyt. gt.(nyy/2 + 1)) then
              ky=ky-nyy
           endif

        do 9040 k=1,nz
           kz = k - 1
           if (k.gt.(nz/2 + 1)) then
              kz = k-1-nz
           endif
c          xkx = 2.*3.1416*(kx1-1)/nxx
c          yky = 2.*3.1416*ky/nyy
c          zkz = 2.*3.1416*kz/nz
c          pwr2k = xkx**2+yky**2+zkz**2
c          if (pwr2k.eq.0) then
c             pwr2k=1.
c          endif
```

```
c  f(j1,jy,k)=(1./pwr2k)*f(j1,jy,k)
c  f(j1+1,jy,k)=(1./pwr2k)*f(j1+1,jy,k)

          ktbl(1,j1,jy,k)=kx
          ktbl(2,j1,jy,k)=ky
          ktbl(3,j1,jy,k)=kz

          ktbl(1,j1+1,jy,k)=kx
          ktbl(2,j1+2,jy,k)=ky
          ktbl(3,j1+3,jy,k)=kz
9040    continue

9060    continue
        return
        end

C---------------------------------------------------------------------

        subroutine dorestykf(ktbl,nx,nxx,ny,nyy,nz,me,icols,
     1  nx1,nx2,ly1,ly2)
c nx --  local
c nxx -- global
c ny --  local
c nyy -- global
c nz --  global
c icols -- mesh column size
c kx1 -- lowest kx in this node
c ky1 -- lowest ky in this node
c ixc -- x coordinate of this node
c iyc -- y coordinate of this node
c
c This routine runs on node 0 only
c
        dimension ktbl(3,nx,ny,nz)

c find the coordinate of this node
c
        ky1 =0

        do 9060 j1=nx1,nx2,2
c          kx1 = (kxt+j1)/2 + 1
           if (j1.eq.1) then
              kx1=nxx/2+1
           else
              kx1=j1/2+1
           endif

           kx=kx1-1

        do 9060 jy=ly1,ly2
           kyt=ky1+jy
           ky = kyt - 1
           if (kyt. gt.(nyy/2 + 1)) then
              ky=ky-nyy
           endif

        do 9040 k=1,nz
           kz = k - 1
           if (k.gt.(nz/2 + 1)) then
              kz = k-1-nz
           endif

c  f(j1,jy,k)=(1./pwr2k)*f(j1,jy,k)
```

```
c  f(j1+1,jy,k)=(1./pwr2k)*f(j1+1,jy,k)

            ktbl(1,j1,jy,k)=kx
            ktbl(2,j1,jy,k)=ky
            ktbl(3,j1,jy,k)=kz

            ktbl(1,j1+1,jy,k)=kx
            ktbl(2,j1+1,jy,k)=ky
            ktbl(3,j1+1,jy,k)=kz
9040     continue

9060     continue
         return
         end
```

```
c-----------------------------------------------------------------------
c
c   3D fft of real arrays, done by doing 1d ffts along each dimension
c
c       There are 3 routines in this set
c               rfft3prep       - set up internal tables for the 3d fft
c               rfft3rc         - forward fft (real to complex)
c               rfft3cr         - inverse fft (complex to real)
c
c       The storage pattern used for the complex result of an fft on
c       real data is the one used by Decyk (Packed format).
c
c  9/24/1992          Edith Huang
c  Edith Huang implemented the 3d-fft based on Intal's 1d fft
c  The 2d-fft is based on Robert Ferraro's program, modified
c  by Edith Huangg
c-----------------------------------------------------------------------
c   this program prepares internal tables for the 3d fft. It should
c   be called only once before using rfft3src or rfft3scr.
c
c     nx      - global number of points in x (1st) dimension
c     ny      - global number of points in y (2nd) dimension
c     nz      - global number of points in z (3rd) dimension
c     dumx(nx)    - internal table for x transform
c     dumx(4*nx+4),dumy(4*ny+4),dumz(4*nz+4)
c     dumy(ny)    - internal table for y transform
c     dumz(nz)    - internal table for z transform
c
      subroutine rfft3prep(dumx,dumy,dumz,nx,ny,nz)
      dimension dumx(*),dumy(*),dumz(*)

c setup the internal table
      call fft2sprep(dumx,dumy,nx,ny)
      call cfft1d(f,nz,0,dumz)
      return
      end
c-----------------------------------------------------------------------
c   this program does a three dimensional fft of real input data

      subroutine rfft3rc(f,t,nx,ny,nz,dumx,dumy,dumz,dcopy)
c
      include 'para2.inc'
      include 'simqb3da.inc'
c
c     f       - Real array, it is dimensioned as one of the following;
c     f(nx,nysr,nzs),f(nxs,ny,nzs),f(nxs,nysc,nz).
c     f must be large enough to contain the maximum storage of
c     mstore=max( (nx*nysr*nzs),(nxs*ny*nzs),(nxs*nysc*nz) )
c     In the calling program a dummy array must be dimensioned to
c     dummy(mstore) then equivalence f to this array
c     equivalence (f(1,1,1),dummy(1)
c
c     nxs, nys,nzs -- are local dimension
c     nx      - number of points in x (1st) dimension
c     ny      - number of points in y (2nd) dimension
c     nz      - number of points in z (3rd) dimension
c     f    - input and output real array
c     t    - work array, same size as f
c
      real f(*), dumx(*),dumy(*),dumz(*),dcopy(*)
      real t(*)
      call kmulti(6)
      if (nzs .lt. 1) then
         write(*,*) 'nzs too small nzs= ',nzs
```

```
         call kflush(6)
         call killproc()
      elseif (nysr.lt.1) then
         write(*,*) 'nysr too small nysr = ', nysr
         call kflush(6)
         call killproc()
      elseif (nysc.lt.1) then
         write(*,*) 'nysc too small nysc= ', nysc
         call kflush(6)
         call killproc()
      elseif (nxs.lt. 2) then
         write(*,*) 'nxs too small = ',nxs
         call kflush(6)
         call killproc()
      endif
      call ksingl(6)
c
c   real to complex 1d fft - k(nx/2) returned in f(2,k) (Im part of k0)
c   k's greater than nx/2 are redundant since f(k) = CC f(-k)

c  Do the 1d fft on nx direction on each node. There are partial
c  data on y anf z direction on each node
c  f(nx,nysr,nzs)

      isign=-1
      do k=0,nzs-1
        do j= 0,nysr-1
          ix=k*nx*nysr+j*nx+1
          call fft1rc(f(ix),nx,isign,dumx,dcopy)
        enddo
      enddo
c
c exchange parts so that
c on each node there is complete y, but partial x and partial z

        if (nprocs.ne.1) then
          call xchpart3dfc(f,t,nx,ny,nz,1)
        endif
c
c  do transform in y direction
c  f(nxs,ny,nzs)

       do k=0,nzs-1
         ix=k*nxs*ny+1
         call fftyd(f(ix),nxs,ny,dumy,dcopy,me,icols)
       enddo

       if (nprocs.ne.1) then
          call xchpart3dfc(f,t,nx,ny,nz,11)
       endif

c do xchpart for  z
c on each node there is complete z, but partial x and partial y
c do Z directory transform
c f(nxs,nys,nz)

      call fftzd(f,nxs,nx,nysc,ny,nz,dumz,dcopy,me,icols,nprocs)
      return
      end
c-----------------------------------------------------------------------
c   The program does the inverse 3d fft on complex data stored in
c   f array
      subroutine rfft3cr(f,t,nx,ny,nz,dumx,dumy,dumz,dcopy)
c
```

```
c     f       - Real array
c     dumx    -  dumx(4*nx+4)
c     dumy    -  dumy(4*ny+4)
c     dumz    -  dumz(4*nz+4)
c     nx      - number of points in x (1st) dimension
c     ny      - number of points in y (2nd) dimension
c     nz      - number of points in z (3rd) dimension
c
      include 'para2.inc'
      include 'simqb3da.inc'
c
      dimension f(*), dumx(*),dumy(*),dumz(*),dcopy(*)
      dimension t(*)

      call kmulti(6)
      if (nzs.lt. 1) then
         write(*,*) 'nzs too small nzs = ',nzs
         call kflush(6)
         call killproc()
      elseif (nysr.lt. 1) then
         write(*,*) ' nysr too small nysr = ',nysr
         call kflush(6)
         call killproc()
      elseif (nysc.lt. 1) then
         write(*,*) ' nysc too small nysc = ',nysc
         call kflush(6)
         call killproc()
      elseif (nxs .lt. 2) then
         write(*,*) 'nxs too small = ',nxs
         call kflush(6)
         call killproc()
      endif
      call ksingl(6)

c Inverse transform the z dimension
c f(nxs,nys,nz)
c
      call ifftzd(f,nxs,nx,nysc,ny,nz,dumz,dcopy,me,
     1    icols,nprocs)

c call xchpart so that for partial x and partial z with all y's
       if (nprocs.ne.1) then
          call xchpart3dfc(f,t,nx,ny,nz,3)
       endif

c f(nxs,ny,nzs)
c inverse transform in y
      do k= 0, nzs-1
        ix=k*nxs*ny+1
        call ifftyd(f(ix),nxs,ny,dumy,dcopy,me,icols)
      enddo

c Do xchpart3d
       if (nprocs.ne.1) then
          call xchpart3dfc(f,t,nx,ny,nz,13)
       endif
c inverse x transform
      isign=1
      do  k=0,nzs-1
        do j=0,nysr-1
          call fft1rc(f(k*nx*nysr+j*nx+1),nx,isign,dumx,dcopy)
        enddo
      enddo
      return
```

```
      end
c-----------------------------------------------------------------------
      subroutine fftzd(f,nx,nxx,ny,nyy,nz,dumz,dum,me,icols,nprocs)
c
c  Z transform
c
c     nx (nxs) -- local dimension
c                 f(nxs,nysc,nz)
c     ny (nysc) --   local
c     nxx--   global
c     nyy--   global
c     nz --   global

        dimension f(nx,ny,nz),dum(*),dumz(*)
c do complex fft on all columns of f except if I am a processor
c which contains kx=0 , y=1 and y=ny/2+1
c (f(2*i+1),f(2*i+2)) ith mode ( real, imag)

c check which nodes contain kx=0 and (y=1 or y=ny/2+1)
c calculate the node numbers
c (ix=1,iy=1)  (ix=1,iy=ny/2+1)
c 2 nodes, node 0 and one more node

       iy2=nyy/2+1

c ix1=1,iy1=1, this will be node 0
c      node(1)=(ix1/nxs)*icols+(iy1/nysc)
c ix1=1,iy2=nyy/2+1
c     node(2)=(ix1/nxs)*icols+(iy2/nysc)

      node=iy2/ny

c     call kmulti(6)
c     write(6,*)' special nodes fftzd: me:',me,node
c     call kflush(6)
c     call ksingl(6)

c     if (mod(me,irows).eq.0) then
        if ((me.eq.0).or.(me.eq.node)) then
c  Do the columns >=3 first

           nx1=3
           nx2=nx
           ny1=1
           ny2=ny
           call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)

c I am the processor which cintains y=1 or y=ny/2+1.
c Therefore we have to take care of the 1st 2 columns
c For kx=0,ky=0 and kx=nx/2, ky=0
c pure real

           ny1=1
           call packrl(f,nx,ny,nz,dumz,dum,ny1)

c For kx=0,ky=ny/2 and kx=nx/2,ky=ny/2
c pure real
           if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
              ny1=ny/2+1
c full
              call packrl(f,nx,ny,nz,dumz,dum,ny1)
           endif

c Do the rest of the first 2 columns in node which contains y=1
```

```
c or y=ny/2+1

           nx1=1
           nx2=1
           ny1=2
           if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
              ny2=ny/2
           else
              ny2=ny
           endif

           call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)

           if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
             ny1=ny/2+2
             ny2=ny
             call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)
           endif

         else
c Not special processor

           nx1=1
           nx2=nx
           ny1=1
           ny2=ny
           call ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)
          endif
        return
        end

C--------------------------------------------------------------

         subroutine ftcomplex(f,nx,ny,nz,dumz,dum,nx1,nx2,ny1,ny2)
         dimension f(nx,ny,nz),dumz(*),dum(*)

         do 9030 j1=nx1,nx2,2
         do 9030 jy=ny1,ny2
         do 9010 k=1,nz

           k2=k+k
           dum(k2-1)=f(j1,jy,k)
           dum(k2)=f(j1+1,jy,k)
9010     continue

c   complex to complex fft
           call fft1cc(dum,nz,-1,dumz)

         do 9020 k=1,nz
           k2=k+k
           f(j1,jy,k)=dum(k2-1)
           f(j1+1,jy,k)=dum(k2)
9020     continue
9030     continue
        return
        end

C--------------------------------------------------------------
        subroutine packrl(f,nx,ny,nz,dumz,dum,nyl)
        dimension f(nx,ny,nz),dumz(*),dum(*)

c transform in z dimension
c
c nyl -- lower case letter L
```

```
c
        do 9040 k=1,nz

          k2=k+k
          dum(k2-1)=f(1,nyl,k)
          dum(k2)=0.

9040    continue

c complex to complex
        call fft1cc(dum,nz,-1,dumz)

c Save the 1st half of the result and retrieving the nx=nx/2 and ny=0
        do 9050 k=1,nz/2+1
          k2=k+k
          f(1,nyl,k)=dum(k2-1)
          dum(k2-1)=f(2,nyl,k)
          f(2,nyl,k)=dum(k2)
          dum(k2)=0.
9050    continue

c Transform in z for n=nx/2
        do 9060 k=nz/2+2,nz
          k2=k+k
          dum(k2-1)=f(2,nyl,k)
          dum(k2)=0.
9060    continue

c   complex to complex fft
        call fft1cc(dum,nz,-1,dumz)

        do 9070 k=nz/2+2,nz

          k2=k+k
          f(1,nyl,k)=dum(k2-1)
          f(2,nyl,k)=dum(k2)
9070    continue

        f(2,nyl,1)=dum(1)
        f(2,nyl,nz/2+1)=dum(nz+1)
        return
        end
c-------------------------------------------------------------------------

        subroutine ifftzd(f,nx,nxx,ny,nyy,nz,dumz,dcopy,me,
     1  icols,nprocs)

        dimension f(nx,ny,nz),dumz(*),dcopy(*)
c
c  Inverse Z transform
c    nx(nxs) -- local
c    ny(nysc) -- local
c    nyy -- global
c    nz -- global
c    f(nxs,nys,nz)
c
c special processors have to worry about the first 2 column of the packed data
c
          iy2=nyy/2+1
          node=iy2/ny
          if ((me.ne.0).and. (me .ne.node)) go to 100
c for jy=1 and jy=ny/2 and jx=1 and jx=2
c the data are pure real values
c
```

```
c Do inverse transform in Z dimension
c kx=0, kx=nx/2, ky=ny/2
c
          if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
                nyl=ny/2+1
               call unpack(f,nx,ny,nz,dumz,dcopy,nyl)
          endif
c
c DO inverse transform in Z dimension for
c kx=0, kx=nx/2, ky=0
c
        nyl=1
        call unpack(f,nx,ny,nz,dumz,dcopy,nyl)

c Do the rest of the data normally
        nx1=1
        nx2=1
        ly1=2
        if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
          ly2=ny/2
        else
          ly2=ny
        endif

        call dorest(f,nx,ny,nz,dumz,dcopy,nx1,nx2,ly1,ly2)

        if (nprocs.eq.1.or.(nprocs.gt.1.and.icols.eq.1)) then
           ly1=ny/2+2
           ly2=ny
           call dorest(f,nx,ny,nz,dumz,dcopy,nx1,nx2,ly1,ly2)
        endif

100     continue
c
c  do the rest of rows, columns normally
c
         if ((me.eq.0).or.(me.eq.node)) then
           nx1=3
         else
           nx1=1
         endif

         nx2=nx
         ly1=1
         ly2=ny

         call dorest(f,nx,ny,nz,dumz,dcopy,nx1,nx2,ly1,ly2)
        return
        end

C--------------------------------------------------------------

        subroutine unpack(f,nx,ny,nz,dumz,dum,nyl)
        dimension f(nx,ny,nz),dumz(*),dum(*)

c for nx/2,ny/2,nz/2
c
        do 9000 k=nz/2+2,nz

          k2=k+k
          dum(k2-1)=f(1,nyl,k)
          dum(2*nz+3-k2)=f(1,nyl,k)
          dum(k2)=f(2,nyl,k)
          dum(2*nz+4-k2)=-f(2,nyl,k)
```

```
9000    continue
        dum(1)=f(2,nyl,1)
        dum(2)=0.
        dum(nz+1)=f(2,nyl,nz/2+1)
        dum(nz+2)=0

c Complex to complex
        call fft1cc(dum,nz,1,dumz)

c Pick up data for nx=0, ny=ny/2 and put the result in the right
c loaction
c
        do 9010 k=1,nz/2+1
          k2=k+k
          dum(k2)=f(2,nyl,k)
          f(2,nyl,k)=dum(k2-1)
          dum(k2-1)=f(1,nyl,k)
9010    continue

        do 9020 k=nz/2+2,nz
           k2=k+k
           dum(k2)=-dum(2*nz+4-k2)
           f(2,nyl,k) = dum(k2-1)
           dum(k2-1) = dum(2*nz+3-k2)
9020     continue
         dum(2)=0.
         dum(nz+2)=0.
c
c Complex to complex fft
c
        call fft1cc (dum,nz,1,dumz)
        do 9030 k=1,nz
          k1=k+k-1
          f(1,nyl,k)=dum(k1)
9030    continue
        return
        end

C---------------------------------------------------------------------

        subroutine dorest(f,nx,ny,nz,dumz,dum,nx1,nx2,ly1,ly2)
        dimension f(nx,ny,nz),dumz(*),dum(*)

        do 9060 j1=nx1,nx2,2
        do 9060 jy=ly1,ly2

        do 9040 k=1,nz
          k2=k+k
          dum(k2-1)=f(j1,jy,k)
          dum(k2)=f(j1+1,jy,k)
9040    continue

c complex to complex fft
c
        call fft1cc(dum,nz,1,dumz)

        do 9050 k=1,nz
          k2=k+k
          f(j1,jy,k)=dum(k2-1)
          f(j1+1,jy,k)=dum(k2)
9050    continue
9060    continue
        return
        end
```

```
c-----------------------------------------------------------------------
c   this program prepares internal tables for the 2d fft. It should
c   be called only once before using fft2src or fft2scr.
c
c     dumx(4*nx+4) - internal table for x transform
c     dumy(4*ny+4) - internal table for y transform
c     nx      - global number of points in x (1st) dimension
c     ny      - global number of points in y (2nd) dimension
c
c               fft2src : real to complex transform
c               fft2scr : complex to complex transform
c       each routine returns results partitioned for its inverse
c
      subroutine fft2sprep(dumx,dumy,nx,ny)
      dimension dumx(*),dumy(*)

c setup internal tables
      call fft1rc(f,nx,0,dumx,d)
      call fft1cc(f,ny,0,dumy)
       return
       end
c-----------------------------------------------------------------------
c

c  This routine transforms the y coordinate of the data array f
c
      subroutine fftyd(f,nxs,ny,dumy,dum,me,icols)
c
c     f       - Real array dimensioned f(nx,ny)
c     dum     - work space
c     nxs      - local number of points in x (1st) dimension
c     ny      -  global number of points in y (2nd) dimension
c     dumy   -- internal table
c
      dimension f(nxs,ny), dum(*),dumy(*)

        if ((me/icols).eq.0) then
c special nodes contains Kx=0
          call ftysp(f,nxs,ny,dumy,dum)
        else
           call fty(f,nxs,ny,dumy,dum)
        endif
        return
        end

      subroutine fty(f,nxs,ny,dumy,dum)
      dimension f(nxs,ny),dumy(*),dum(*)

c   nxs - local dimension
c   ny  - global dimension

      nx1=1
      do 9030 j1 = nx1, nxs, 2
      do 9010 k = 1, ny
      k2 = k + k
      dum(k2-1) = f(j1,k)
      dum(k2) = f(j1+1,k)
 9010 continue

c   complex to complex fft
      call fft1cc(dum,ny,-1,dumy)

      do 9020 k = 1, ny
      k2 = k + k
```

```
      f(j1,k) = dum(k2-1)
      f(j1+1,k) = dum(k2)
 9020 continue

 9030 continue
        return
        end

      subroutine ftysp(f,nxs,ny,dumy,dum)
      dimension f(nxs,ny),dumy(*),dum(*)

c transform in y for 0 < n < nx/2
c Handle the first 2 rows differently, because it represent nx=0 and
c nx=nx/2 mode. The data are real for these two mode
c   (f(2*i+1),f(2*i+2)) is the (Real, Imag) ith mode
      nx1=3
      do 9030 j1 = nx1, nxs, 2

      do 9010 k = 1, ny
      k2 = k + k
      dum(k2-1) = f(j1,k)
      dum(k2) = f(j1+1,k)
 9010 continue

c   complex to complex fft
      call fft1cc(dum,ny,-1,dumy)

      do 9020 k = 1, ny
      k2 = k + k
      f(j1,k) = dum(k2-1)
      f(j1+1,k) = dum(k2)
 9020 continue
 9030 continue

c transform in y for nx = 0 mode
c   note that nx = 0 mode is purely real
      do 9040 k = 1, ny
      k2 = k + k
      dum(k2-1) = f(1,k)
      dum(k2) = 0.
 9040 continue

c   complex to complex fft
      call fft1cc(dum,ny,-1,dumy)

c   store the result as (Real, Imag) while retrieving the nx/2 mode
c   (which is also purely real). Note that the upper half of the result
c   in dum (the -ky modes) are redundant.

      do 9050 k = 1, ny/2 + 1
      k2 = k + k
      f(1,k) = dum(k2-1)
      dum(k2-1) = f(2,k)
      f(2,k) = dum(k2)
      dum(k2) = 0.
 9050 continue

c transform in y for n = nx/2
      do 9060 k = ny/2 + 2, ny
      k2 = k + k
      dum(k2-1) = f(2,k)
      dum(k2) = 0.
 9060 continue
```

```
c   complex to complex fft
      call fft1cc(dum,ny,-1,dumy)

      do 9070 k = ny/2 + 2, ny
      k2 = k + k
      f(1,k) = dum(k2-1)
      f(2,k) = dum(k2)
 9070 continue

      f(2,1) = dum(1)
      f(2,ny/2+1) = dum(ny+1)

9090  return
      end

c-----------------------------------------------------------------------
c   this routine does the inverse transform of the y coordinate of the
c   data array f
c
      subroutine ifftyd(f,nxs,ny,dumy,dum,me,icols)
c
c     f       - Real array dimensioned f(nx,ny)
c     dum     - work space
c     nxs     - local number of points in x (1st) dimension
c     ny      - global number of points in y (2nd) dimension
c     dumy    - internal table
c
      dimension f(nxs,ny), dum(*),dumy(*)

      if ((me/icols).eq.0) then
         call iftysp(f,nxs,ny,dumy,dum)
      else
         call ifty(f,nxs,ny,dumy,dum)
      endif
      return
      end

      subroutine iftysp(f,nxs,ny,dumy,dum)
      dimension f(nxs,ny),dumy(*),dum(*)

c first inverse transform in y for n = nx/2
      do 9000 k = ny/2 + 2, ny
      k2 = k + k
      dum(k2-1) = f(1,k)
      dum(2*ny+3-k2) = f(1,k)
      dum(k2) = f(2,k)
      dum(2*ny+4-k2) = -f(2,k)
 9000 continue

      dum(1) = f(2,1)
      dum(2) = 0.
      dum(ny+1) = f(2,ny/2+1)
      dum(ny+2) = 0.

c   complex to complex fft
      call fft1cc(dum,ny,1,dumy)

c transform in y for nx = 0
      do 9010 k = 1, ny/2 + 1
      k2 = k + k
      dum(k2) = f(2,k)
      f(2,k) = dum(k2-1)
      dum(k2-1) = f(1,k)
 9010 continue
```

```
      do 9020 k = ny/2 + 2, ny
      k2 = k + k
      dum(k2) = -dum(2*ny+4-k2)
      f(2,k) = dum(k2-1)
      dum(k2-1) = dum(2*ny+3-k2)
 9020 continue

      dum(2) = 0.
      dum(ny+2) = 0.

c   complex to complex fft
      call fft1cc(dum,ny,1,dumy)

      do 9030 k = 1, ny
      k1 = k + k - 1
      f(1,k) = dum(k1)
 9030 continue
9035  continue

c transform in y for 0 < n < nx/2
c   do complex fft on all columns of f except the first two rows
c   (f(2*i+1),f(2*i+2)) is the (Real, Imag) ith mode

         nx1 = 3
      do 9060 j1 = nx1, nxs, 2
      do 9040 k = 1, ny
      k2 = k + k
      dum(k2-1) = f(j1,k)
      dum(k2) = f(j1+1,k)
 9040 continue

c   complex to complex fft
      call fft1cc(dum,ny,1,dumy)

      do 9050 k = 1, ny
      k2 = k + k
      f(j1,k) = dum(k2-1)
      f(j1+1,k) = dum(k2)
 9050 continue
 9060 continue
      return
      end

      subroutine ifty(f,nxs,ny,dumy,dum)
      dimension f(nxs,ny),dumy(*),dum(*)
c transform in y for 0 < n < nx/2
c   do complex fft on all columns of f except the first two rows
c   (f(2*i+1),f(2*i+2)) is the (Real, Imag) ith mode

      nx1=1

      do 9060 j1 = nx1, nxs, 2

      do 9040 k = 1, ny
      k2 = k + k
      dum(k2-1) = f(j1,k)
      dum(k2) = f(j1+1,k)
 9040 continue

c   complex to complex fft
      call fft1cc(dum,ny,1,dumy)

      do 9050 k = 1, ny
```

```
      k2 = k + k
      f(j1,k) = dum(k2-1)
      f(j1+1,k) = dum(k2)
 9050 continue

 9060 continue
      return
      end

c-----------------------------------------------------------------------

c       a       - real array dimensioned a(n)
c       n       - dimension of a, must be a power of 2
c       isign   - 0     do setup of internal tables
c               - -1    do fft (real to complex)
c               - 1     do fft (complex to real)
c       work    - real array dimensioned at least work(4*n+3), used
c                 for holding internal tables
c
c       fft1rc should be called once with isign = 0, at which time
c       internal tables are set up in the work array.  After this,
c       fft1rc may be called as many times as needed to do fft's
c       and inverse fft's on arrays with the same dimension as passed
c       to the first call with isign = 0.  Of course, the contents of
c       work must not be disturbed after the first call to fft1rc with
c       isign = 0
c
c
      subroutine fft1rc(a,n,isign,work,dcopy)
      dimension a(n),work(n*4+3),dcopy(*)

      if (isign.eq.0) then
          call scfft1d(a,n,isign,work)
      else
         call fft1pkrc(a,n,isign,work,dcopy)
      endif
      return
      end

c
c-----------------------------------------------------------------------
c
c This routine is written by Edith Huang to pack the result of the
c output from the Intel 1dfft to the packed format
        subroutine fft1pkrc(a,n,isign,work,dcopy)
        dimension a(n),work(*),dcopy(*)

        if (isign.eq.-1) then
          do i=1,n
            dcopy(i)=a(i)
          enddo
c real to complex
c
            call scfft1d(dcopy,n,isign,work)
          do i=1,n
            a(i)=dcopy(i)
          enddo
            a(2)=dcopy(n+1)
         else
          do i=1,n
            dcopy(i)=a(i)
          enddo
            dcopy(2)=0
            dcopy(n+1)=a(2)
```

```
            dcopy(n+2)=0
c complex to real
c
            call csfft1d(dcopy,n,isign,work)

           do i=1,n
              a(i)=dcopy(i)
           enddo
          endif
          return
          end

c-----------------------------------------------------------------------
c Real to complex fft from Intel's 1d fft
c       a       - complex array dimensioned a(n) to be fft'd
c       n       - dimension of complex array a
c       isign   - 0     setup internal tables
c               - -1    do fft on a
c               - 1     do inverse fft on a
c
c       fft1cc should be called once with isign = 0, at which time
c       internal tables are set up in the work array.  After this,
c       fft1cc may be called as many times as needed to do fft's and
c       inverse fft's on complex arrays with the same dimension as passed
c       to the first call with isign = 0.  Of course, the contents of
c       work must not be disturbed after the first call to fft1cc with
c       isign = 0
c
c   This subroutine is mainly to mask the complications of calling
c   fft directly
      subroutine fft1cc(a,n,isign,work)
      dimension a(n),work(3+n)
c     complex a
      if (isign .ne. 0) goto 10
        call cfft1d(a,n,isign,work)
      return
10      call cfft1d(a,n,isign,work)
      return
      end

c
c-----------------------------------------------------------------------
c       xchpart3dc(f,t,nx,ny,nz,i) - exchange blockdata among processors
c   This routine is written by Edith Huang at JPL on Sept.23,1992
c   The matrix is decomposed dimenison wise on the mesh
c   Z dimension along the column direction and Y dimension along the
c   row direction
c
c   input : f           - the array being exchanged
c           nx,ny,nz    - global dimensions of f
c           i           - which exchange is being done :
c
c              =1   partitioned in y,z for all x to
c                   partitioned in x,z for all y
c
c              =11  partitioned in x,z for all y to
c                   partitioned in x,y for all z
c
c              =3   partitioned in x,y for all z to
c                   partitioned in x,z for all y
c
c              =13  partitioned in x,Z for all y to
c                   partitioned in y,z for all x to
c
```

```
c
c  mesh dimension (rows,cols)
c
c  i=1
c  from f(nx,nysr,nzs)
c  to   f(nxs,ny,nzs)
c  where
c     nxs = nx/rows
c     nysr = ny/rows
c     nzs = nz/cols
c
c  buf(nxs,nysr,nzs)
c
c  communication along columns
c  ncom = rows
c    number of nodes exchange data
c  ncyle = cols
c
c    ks=0,ncom-1
c      k=ks*ncyle.xor.me
c  k is the destination node for sending messages
c
c
c*****
c  i=11
c  communication along rows
c  from f(nxs,ny,nzs)
c  to   f(nxs,nysc,nz)
c  where
c    nxs = nx/rows
c    nysc = ny/cols
c    nz  - global dimension of f
c
c    ny - global dimension of f
c
c  buf(nxs,nysc,nzs)
c
c  ncom = cols
c  ncyle = rows
c
c   ks=0,ncom-1
c     k=ks.xor.me
c   k is the destination node for sending message
c
c*****
c i=3
c  communication along rows
c  from f(nxs,nysc,nz)
c  to   f(nxs,ny,nzs)
c  where
c     nxs = nx/rows
c     ny - global dimension of f
c     nzs = nz/cols
c
c     nysc = ny/cols
c     nz - global dimension of f
c
c  buf(nxs,nysc,nzs)
c
c  ncom = cols
c  ncyle = rows
c
c  ks=0,ncom-1
c    k=ks.xor.me
```

```
c  k is the destination node for sending message
c
c*****
c i=13
c  communication along columns
c  from f(nxs,ny,nzs)
c  to   f(nx,nysr,nzs)
c  where
c    nx - global dimension of f
c    nysr = ny/rows
c    nzs = nz/cols
c
c    nxs = nx/rows
c    ny - global dimension of f
c buf (nx,nysr,nzs)
c
c  ncom=rows
c  ncyle = cols
c
c  ks = 0,ncom-1
c    k=ks*nclye.xor.me
c  k is the destination node for sending message
c
c  me is the mesh node number, they are the same
c
      subroutine xchpart3dfc(f,t,nx,ny,nz,i)
      dimension f(*),t(*)
      include 'para2.inc'
      include 'simqb3da.inc'
      real buf(ibufsize-1)
      equivalence (buf,ibuffer(2)),(ip,ibuffer(1))
      real bufi(ibufsize-1)
      equivalence (bufi,ibuf2(2)),(ipi,ibuf2(1))

       ityper=8000+me
       ip=me

       if (i.eq.1) then
c  option 1 : have all x's  for some y's and some z's of f(x,y,z).
c  Redistribute so
c  that I have all y's for some x's and some z's
c  i=1 sending data

         ncom=irows
         ncyle=icols
         len = 4+ nxs*nysr*nzs*4

c  buffer up each subblock of f for transmission.  prepend my proc
c  number to the message for reference at the recieving end.
c  sending data

       do ks=0,irows-1
         imsg= irecv(ityper,ibuf2,len)
         kl=ks*icols.xor.me
         k = kl/icols
         kt =k
         ix = nxs*k
         do  jz=0,nzs-1
          do  jy=0,nysr-1
           do  jx=1,nxs
           buf(jz*nxs*nysr+jy*nxs+jx) =
     1     f(jz*nx*nysr+jy*nx+jx+ix)
            enddo
           enddo
```

```
          enddo
           itype = 8000+kl
           imsgs=isend(itype,ibuffer,len,kl,0)
c
          call msgwait(imsg)
           k=ipi/icols
           ix = nxs*nysr*k
           do jz=0,nzs-1
             do jy=0,nysr-1
               do jx=1,nxs
                t(jz*nxs*ny+jy*nxs+jx+ix)=
     1          bufi(jz*nxs*nysr+jy*nxs+jx)
               enddo
             enddo
           enddo
           call msgwait(imsgs)
         enddo
c From a(nx,nysr,nzs) to a(nxs,ny,nzs)
c
c
         m=nxs*ny*nzs
         call scopy(m,t,1,f,1)

       elseif ( i.eq.11) then
c  option 11: have all y's for some x's and some z's of f(x,y,z).
c  Redistribute so
c  that I have all x's and  y's for some z's
c i=11 sending data and receiving
c from f(nxs,ny,nzs) to f(nxs,nysc,nz)
c communicate in row wise

         len=4+nxs*nysc*nzs*4

         do ks = 0,icols-1
c
c post ro receive
c
           imsg=irecv(ityper,ibuf2,len)

           kl= ks.xor.me
           k= mod(kl,icols)
           ix = nxs*nysc*k

           do  jz=0,nzs-1
             do  jy=0,nysc-1
               do  jx=1,nxs
            buf(jz*nxs*nysc+jy*nxs+jx) =
     1     f(jz*nxs*ny+jy*nxs+jx+ix)
               enddo
             enddo
           enddo
           itype = 8000+kl
           imsgs= isend(itype,ibuffer,len,kl,0)
c
           call msgwait(imsg)
           k= mod(ipi,icols)
c data are contigous
           ix=nxs*nysc*nzs*k
           do jz=0,nzs-1
             do jy=0,nysc-1
               do jx=1,nxs
                  t(jz*nxs*nysc+jy*nxs+jx+ix)=
     1            bufi(jz*nxs*nysc+jy*nxs+jx)
               enddo
```

```
             enddo
            enddo
           call msgwait(imsgs)
          enddo
c From a(nxs,ny,nzs) to a(nxs,nysc,nz)
          m=nxs*nysc*nz
          call scopy(m,t,1,f,1)

       elseif ( i.eq.3) then
c i=3 sending data and receiving
c communicate along rows

         len=4+nxs*nysc*nzs*4
         ityper=8000+me

         do ks=0,icols-1

c post to receive
          imsg= irecv(ityper,ibuf2,len)

           kl = ks.xor.me
           k= mod(kl,icols)
           ix = nxs*nysc*nzs*k
           do  j=1,nxs*nysc*nzs
             buf(j) = f(j+ix)
           enddo
            itype = 8000+kl
            imsgs= isend(itype,ibuffer,len,kl,0)
c
            call msgwait(imsg)

           k=mod(ipi,icols)
           ix=nxs*nysc*k
           do jz=0,nzs-1
             do jy=0,nysc-1
               do jx=1,nxs
                t(jz*nxs*ny+jy*nxs+jx+ix)=
     1          bufi(jz*nxs*nysc+jy*nxs+jx)
               enddo
             enddo
           enddo
           call msgwait(imsgs)

         enddo
c From a(nxs,nysc,nz) to a(nxs,ny,nzs)

          m=nxs*ny*nzs
          call scopy(m,t,1,f,1)

       elseif ( i.eq.13) then
c i=13 sending data and receiving data
c communicate in column wise

          len=4+nxs*nysr*nzs*4
          do ks=0,irows-1
c post to receive
            imsg= irecv(ityper,ibuf2,len)

            kl= ks*icols.xor.me
            k= kl/icols
            ix=nxs*nysr*k
            do jz=0,nzs-1
              do jy=0,nysr-1
                do jx=1,nxs
```

```
                buf(jz*nxs*nysr+jy*nxs+jx)=
     1     f(jz*nxs*ny+jy*nxs+jx+ix)
              enddo
            enddo
          enddo
c now send messages
c destination is the processor with procnum = kl;
          itype = 8000+kl
          imsgs=isend(itype,ibuffer,len,kl,0)

          call msgwait(imsg)
          k=ipi/icols
          ix=nxs*k
          do jz=0,nzs-1
            do jy=0,nysr-1
              do jx=1,nxs
                t(jz*nx*nysr+jy*nx+jx+ix)=
     1          bufi(jz*nxs*nysr+jy*nxs+jx)
              enddo
            enddo
          enddo
           call msgwait(imsgs)
        enddo

c From a(nxs,ny,nzs) to a(nx,nysr,nzs)

        m=nx*nysr*nzs
        call scopy (m,t,1,f,1)

       endif
      return
      end
```

```
c*****************************************************************
c express.inc
c-----------------------------------------------------------------
            common/xpress/nocare,norder, nonode, ihost,ialnod,ialprc



c*****************************************************************
c simqb3da.inc
c-----------------------------------------------------------------
c
c
c this is too small for a(128,128,128) and 2x2 mesh, parameter (ibufsize=2**15+1)
c nyss is the max of (nysr,nysc)
c     parameter (ibufsize=nxs1*nyss*nzs1+1)
      integer nprocs,procnum
      common /simqb/ nprocs,ibuf2(ibufsize),
     1               ibuffer(ibufsize),me,
     2               irows,icols,nxs,nysr,nysc,nzs



c*****************************************************************
c  para2.inc
c-----------------------------------------------------------------
c--- para2.inc
c This include file must be changed for different size of A array
c and different size of mesh. This example is setup for
c A(8,8,8) on 2x2 mesh.
c assum mesh size (irow,icol), array A(nxxx,nyyy,nzzz)
c nzs1=nzzz/icol,nysr1=nyyy/irow,nysc1=nyyy/icol,nxs1=nxxx/irow
c
c  for 2X2 mesh
c a(8,8,8)
      parameter(nxxx=8,nyyy=8,nzzz=8,
     1 nzs1=4,nysr1=4,nysc1=4,nxs1=4)
c
c for 16x32 mesh 512 nodes
c a(1024,32,512)
c     parameter(nxxx=1024,nyyy=32,nzzz=512,
c    1 nzs1=16,nysr1=2,nysc1=1,nxs1=64)
c
c find the max of (nysr1,nysc1)
c nt= 2*maximun of (nx,ny,nz)
c     parameter(nt=2*ny)
c     parameter(nt=2*nx)
c     parameter(nt=2*nz)
c find the  max of (nx,ny)
c set nx=nxxx,ny=nyyy,nz=nzzz in the main program
c dcopy(nt) -- work vector
c dumx(4*nxxx+4),dumy(4*nyyy+4),dumz(4*nzzz+4) -- internal tables
c
c nt1=max of (nxxx,nyyy)
c
      parameter(nt1=((nxxx*(nxxx/nyyy)+
     1 nyyy*(nyyy/nxxx))/(nxxx/nyyy+nyyy/nxxx)))
c
c nt2=max of (nt1,nzzz)
      parameter(nt2=((nt1*(nt1/nzzz)+
     1 nzzz*(nzzz/nt1))/(nt1/nzzz+nzzz/nt1)))
      parameter(nt=2*nt2)
c nyss= max of (nysr1,nysc1)
      parameter(nyss=((nysr1*(nysr1/nysc1)+nysc1*(nysc1/nysr1))
```

```
     1 /(nysr1/nysc1+nysc1/nysr1)))
       parameter(ibufsize=nxs1*nyss*nzs1+1)
```