NISTIR 6997

User Manual for Finite Element and Finite Difference Programs: A Parallel Version of NIST IR 6269

Robert B. Bohn
Edward J. Garboczi

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Information Technology Laboratory
Building and Fire Research Laboratory
Gaithersburg, MD 20899

NIST

National Institute of Standards and Technology
Technology Administration
U.S. Department of Commerce

NISTIR 6997

User Manual for Finite Element and Finite Difference Programs: A Parallel Version of NIST IR 6269

Robert B. Bohn
Edward J. Garboczi

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Information Technology Laboratory
Building and Fire Research Laboratory
Gaithersburg, MD 20899

June 19, 2003

U.S. DEPARTMENT OF COMMERCE
Donald L. Evans, Secretary

TECHNOLOGY ADMINISTRATION
Phillip J. Bond, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arden L. Bement, Jr., Director

User Manual for Finite Element and Finite Difference Programs: A Parallel Version of NISTIR-6269

Robert B. Bohn
Information Technology Laboratory

Edward J. Garboczi
Building and Fire Research Laboratory

NIST
Gaithersburg, MD 20899

March 4, 2003

Abstract

This document contains the descriptions, algorithms, user information and listings of the parallel Fortran90/MPI versions of the suite of programs found in NIST-IR 6269 [1], *Finite Element and Finite Difference Programs for Computing the Linear Electric and Elastic Properties of Digital Images of Random Materials*. These programs use 3-D digital image data on random materials as input and then calculate the effective properties of the random material when subjected to applied fields (for example, mechanical/thermal stresses and AC/DC electric fields).

The purpose behind this undertaking is to execute these programs in a parallel computing environment (for example, Linux clusters), so as to decrease real-time execution, increase the potential problem size, and increase digital resolution/problem accuracy.

Keywords: computer modeling, concrete, elastic moduli; electrical conductivity; finite difference; finite element; MPI; parallel computing; random materials; thermal elasticity

Contents

1 Introduction 4

2 Program Theory 5
2.1 Data Model and Memory Management . . . 5
2.1.1 Finite Element . . . 9
2.1.2 Finite Difference . . . 9
2.2 Parallel Algorithms . . . 12
2.2.1 Finite Element . . . 12
2.2.2 Finite Difference . . . 13

3 Program Operation and USER information 14
3.1 Notable changes from the serial version . . . 14
3.2 Inputs . . . 14
3.2.1 ELECFEM3D . . . 15
3.2.2 ELAS3D . . . 15
3.2.3 THERMAL3D . . . 15
3.2.4 DC3D . . . 15
3.2.5 AC3D . . . 15
3.3 Outputs . . . 16
3.3.1 ELECFEM3D . . . 16
3.3.2 ELAS3D . . . 17
3.3.3 THERMAL3D . . . 17
3.3.4 DC3D . . . 17
3.3.5 AC3D . . . 18
3.4 MAIN . . . 18
3.5 Finite Element Subroutines . . . 19
3.5.1 FEMAT_MPI . . . 19
3.5.2 ENERGY_MPI . . . 21
3.5.3 DEMBX_MPI . . . 21
3.5.4 GBAH . . . 22
3.5.5 STRESS_MPI . . . 23
3.5.6 CURRENT_MPI . . . 23
3.6 Finite Difference Subroutines . . . 23
3.6.1 BOND_DC, BOND_AC . . . 23
3.6.2 DEMBX_DC, DEMBX_AC . . . 24
3.6.3 CURRENT_DC, CURRENT_AC . . . 24
3.6.4 PROD_DC, PROD_AC . . . 24
3.7 Other Subroutines . . . 24
3.7.1 z_ghost_int,z_ghost_dp,z_ghost_cmplx . . . 24
3.7.2 xy_ghost_dp,xy_ghost_cmplx . . . 24
3.7.3 t2b,b2t,t2b_dp,b2t_dp, t2b_cmplx,b2t_cmplx . . . 25
3.7.4 m2ijk . . . 25
3.7.5 ipxyz . . . 25
3.7.6 phasemod_init . . . 25
3.7.7 dpixel . . . 26
3.7.8 dassig . . . 26
3.8 Makefiles and Execution scripts . . . 26
3.8.1 PC Linux Cluster using PBS . . . 26

3.8.2 SGI Origin 2000 using NQS 28

4 MPI Primer 29
4.1 Initialization 29
4.2 Sending and Receiving Data 30
4.3 Built-in Mathematical Functions 31
4.4 Summary 31

5 Disclaimer 31

6 Program Listings 32
6.1 Finite Element 32
6.1.1 ELECFEM3D_MPI.f 32
6.1.2 ELAS3D_MPI.f 76
6.1.3 THERMAL3D_MPI.f 129
6.2 Finite Difference 210
6.2.1 DC3D_MPI.f 210
6.2.2 AC3D_MPI.f 239

7 Tools 272
7.1 MEMAPP.f 272

List of Tables

1 Array redimensioning from serial to parallel versions of ELECFEM3D, ELAS3D and THERMAL3D 10

List of Figures

1 Depiction of Data Set split across 8 processing nodes. 6
2 Depiction of d1 and d2 values for Root and Processing Node 1 7
3 Depiction of Top and Bottom Ghost Layers from Root and Node 1. d2+1(Node1) = d1(Node2); d1-1(Node1) = d2(Root); d2+1(Root)= d2(Node1) but, d1-1(Root) = d2 layer of Nth processor. 8
4 Depiction of original finite element data set in a serial based program showing the equivalency of the Z layers. The **Real** layers are real data. 10
5 Depiction of ghost layers in the parallel finite element program showing the equivalency of the Z layers across 4 processors for the $nz = 6$, $nz2 = 8$ case. Note how the $d1 - 1$ layer of the bottom node & the $d2 + 1$ layer of the top node are unused in the calculation. 11

1 Introduction

Calculating the effective properties of random materials is not a trivial procedure. Due to their random composition, random phase shape, and widely varying length scales, these properties cannot be determined analytically, but instead necessitate a numerical computation. But before any computing may begin, detailed microstructural information must be in hand. Some of this information may be obtained experimentally using x-ray tomography or similar techniques or developed in models. In any event, the input is converted to a 2 or 3-D digital image that represents the overall structure of the composite.

One application is to the properties of cement and concrete. They are complex mixtures that can contain 20 to 30 distinct individual chemical phases in the same mixture. The nominal sizes of the data sets are 100^3 to 300^3 voxels, containing several thousand particles. Even though this seems to be a large number of voxels, one would like to increase the potential problem size and increase the digital resolution of the original data image. But the possible overall sizes of problems have been ultimately bound by the available computational resources of a serial based machine. In addition, the wall clock (real) time to perform these calculations routinely reach into the 100 hour regime, making even larger computations impractical.

The original programs from NISTIR-6269 calculate the overall effective properties of the above systems but they suffer from several limitations. Resolution problems (hence accuracy) are affected if the dataset is too large ($\geq 512^3$ voxels) since lack of computer memory becomes a critical issue. At present, datasets this size must be split up into smaller files and processed individually, each producing a subset of results that must be approximately linked together.

Using parallel processing allows one to have the power and storage capacity of several processors. It is then possible to divide a large dataset exactly across multiple processors and, in essence, each processor operates on a dataset of reduced size. In addition, any large arrays calculated in the serial version only have their corresponding sections calculated on each processor as well. The overall functionality of the program is not compromised by operating on smaller sections, but one can gain theoretical speed-ups of execution time on the order of the number of processors used and so be able to handle large problems.

Parallel processing also supports data transfer, i.e. send and receive mechanisms, between the processors. This is important for calculations involving nearest neighbors. Splitting the data across N processors sets up $N-1$ imposed boundaries on the data by direct consequence of the split. Of course, it is necessary to know which data is needed by which processor and when in order to have a successfully operating program.

To accommodate these large calculations and decrease the time to perform them, the suite of original programs from NISTIR-6269 have been converted to run in a parallel environment in FORTRAN90 with MPI (*Message Passing Interface*). This conversion allows datasets of size 400^3 or greater to be routinely used without any compromise of numerical accuracy.

2 Program Theory

This section will examine the basics behind the parallel versions by discussing their data models, memory management, and algorithms. Although one can probably comprehend the principles of the programming model, it is important to have the necessary mathematical background from the original NISTIR 6269 to provide a more complete understanding of the relationships between the serial and parallel versions.

Briefly, these programs operate by performing a series of matrix operations (additions and multiplications) on very large arrays, dimensioned as large as the original data set or greater, in order to minimize the overall energy of the system in question. The minimization technique used is a conjugate gradient solver [2]. This is an iterative procedure which is carried out until a certain minimum threshold (energy gradient) is reached. The serial case defines these arrays as 1-D vectors and then operates or computes each element sequentially and therefore independently from each other. This is good behavior for a program to have if the parallel adaptation is to be made. In fact, it ensures that the operations can proceed in a parallel fashion. Therefore, it is necessary to give each processor a piece of the original data array and most of the calculations can proceed independently until special cases of communication are warranted. It is important to reduce the amount of time one spends communicating to gain the maximum time savings from parallel computations.

The theoretical core behind each program is similar, therefore many of the parallel algorithms were readily modified to adapt to the specific physical case (mechanical/thermal stress, electric fields). Differences between the codes is manifested by a few simple items, namely the dimensionality of the problem (arrays), how many arrays are included in the minimization, the specific applied field and if the program itself is either based on a finite element or finite difference method of solution of the original partial differential equations.

2.1 Data Model and Memory Management

In a parallel environment with a set of N processors, the program is typically set up such that one processor is arbitrarily selected as the root node (rank=0) and the others as workers (rank=1...N-1). For a given processor, P, we call processors with rank equal to $P-1$ and $P+1$, P's south and north neighbor, respectively. Root is in charge of the I/O, assigning data to the workers (using MPI) and does its share of calculating. It should be noted that the user does not actually assign the root or worker identity to any specific processor in the cluster, but the program makes the requirement that the conditions exist and it is the operating system's duty to carry it out. The user can always specify a processor to carry out a certain function in the code. See Section 4, **MPI Primer** for details on how each processor is uniquely identified within a program.

As mentioned previously, each matrix element can be calculated individually and independently from each other. So these parallel programs take advantage of the 3-D nature of the data (stored in array pix) by splitting it (along the z-direction) across multiple processing nodes. Each matrix element is addressed by a unique triplet of (x, y, z) coordinates and only portions (z-specific) of these large arrays exist on all the processors. This data is divided as evenly as possible over N compute nodes in the z direction. The number of data layers, $\mathbf{N}_k$, each of the N processors receives per array is approximately $\frac{nz}{N}$. Therefore, the data model is to take a large array and treat it as nz 2-D arrays with dimensions of (nx, ny). Now each processing node only has to dedicate $\frac{1}{N}$ (Figure 1) the amount of memory to data storage for an equivalent sized problem on a serial machine. Theoretically, this calculation should speed up by a factor of N the amount of time to execute the same problem on a serial based machine. Additionally, problems which are N times as large can be run as well.

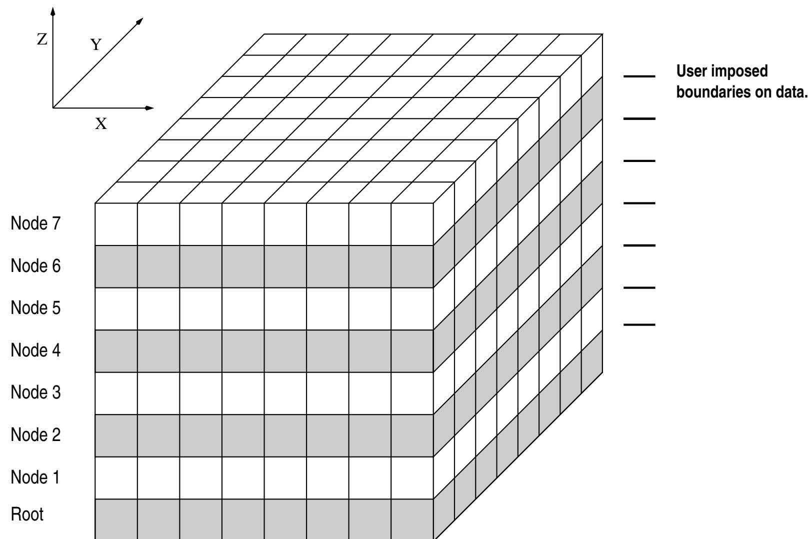

Figure 1: Depiction of Data Set split across 8 processing nodes.

Given the magnitude of nz and the number of processors, root calculates the number of data layers of pix each processing node receives. The lower and upper limits of the z extents for each processor are $d1$ and $d2$. Each node gets its own copy of the layers of pix from root; root stores a master copy of the $d1$ and $d2$ values for all the nodes in arrays $d1s$ and $d2s$. Next, root passes the contiguous layers of the original data in which the value of pix's k indices lie ($d1 \leq k \leq d2$) on the proper processing node. The proper assignment of an array of this type using FORTRAN90 notation is: $pix(nx, ny, d1 : d2)$. The last index in the form $d1 : d2$, is the range of the k values used. For a given processing node, these values are unique.

The inherent question after splitting the original data across a number of processing nodes is: Does a node have all the data it needs to carry out its assigned tasks? We know for these problems, which need nearest neighbor information, that they cannot have the required data after the initial split due to the user imposed boundaries on the dataset. Therefore inter-node communication (data transfer) is necessary. This requires the processors to know which nodes have the data they need and a mechanism for the data transfer.

Since a voxel needs information from its nearest neighbors to perform a correct calculation, problems arise when a processor attempts to calculate using a voxel located in either its top layer ($z = d2$) or its bottom layer ($z = d1$). Since this problem arises for all voxels in their respective $d1$ or $d2$ layers, a given node will need an entire data slice (one 2-d array) from its north and south neighbors, respectively. To be exact, processor P needs the south node ($P - 1$) to send its values of $pix(i, j, d2)$ and the north node ($P + 1$) to send its values of $pix(i, j, d1)$. In another notation, using the rank value as a subscript, processor P needs $pix(i, j, d2)_{P-1}$ and $pix(i, j, d1)_{P+1}$.

The preferred way for handling this situation is to increase the z-size of the array on each node by 2. The new layers occupy $k = d1 - 1$ and $k = d2 + 1$ per processor. They are referred to commonly as ghost layers (Figure 3). These layers are created before any of the calculations proceed since pix does not change during a calculation. This method allows the calculations to proceed uninterrupted unless global sums or other similar actions are called for. We define a new array called vox which is a copy of pix, but also containing the 2 extra data layers. It has no serial counterpart, but will function like pix from the serial code. It is dimensioned in

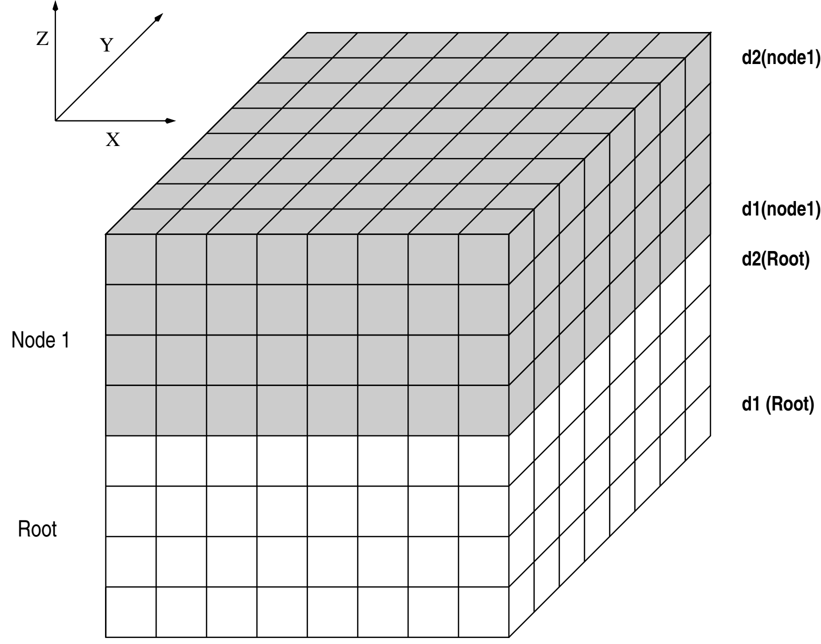

Figure 2: Depiction of d1 and d2 values for Root and Processing Node 1

FORTRAN90 as: $vox(i, j, d1 - 1 : d2 + 1)$. To emphasize its identity,

$$vox(i, j, d1 : d2)_P = pix(i, j, d1 : d2)_P \tag{1}$$

$$vox(i, j, d1 - 1)_P = pix(i, j, d2)_{P-1} \tag{2}$$

$$vox(i, j, d2 + 1)_P = pix(i, j, d1)_{P+1} \tag{3}$$

This makes the total amount of memory usage per node increase slightly. However, it obviates the need for additional inter-node communication during a given calculation that would increase the overall run time of the job. Also remember that one is gaining substantial memory savings compared to the serial version, so this cost is acceptable.

This situation gives rise to two special cases, namely: What is considered south of processor 0 and north of processor $N-1$? The key to this is to know that the original data, pix, has periodic boundary conditions and behaves in a cyclic fashion. Therefore, south of processor 0 is processor $N - 1$ and north of processor $N - 1$ is processor 0. This leads to the following assignments.

Processor 0:

$$vox(i, j, d1 : d2)_0 = pix(i, j, d1 : d2)_0 \tag{4}$$

$$vox(i, j, d1 - 1)_0 = pix(i, j, d2)_{N-1} \tag{5}$$

$$vox(i, j, d2 + 1)_0 = pix(i, j, d1)_1 \tag{6}$$

Processor N-1:

$$vox(i, j, d1 : d2)_{N-1} = pix(i, j, d1 : d2)_{N-1} \tag{7}$$

$$vox(i, j, d1 - 1)_{N-1} = pix(i, j, d2)_{N-2} \tag{8}$$

$$vox(i, j, d2 + 1)_{N-1} = pix(i, j, d1)_0 \tag{9}$$

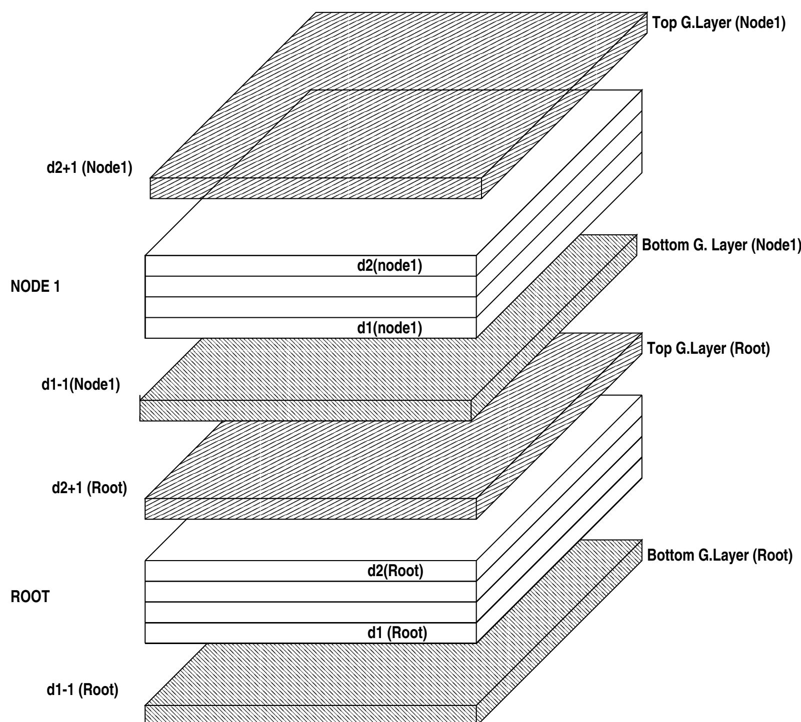

Figure 3: Depiction of Top and Bottom Ghost Layers from Root and Node 1. d2+1(Node1) = d1(Node2); d1-1(Node1) = d2(Root); d2+1(Root)= d2(Node1) but, d1-1(Root) = d2 layer of Nth processor.

In the serial code, memory allocation of the array *pix* and subsequent large arrays is handled in a static way by DIMENSION statements. In each statement, the user substitutes the numerical value of ns, where $ns = nx \times ny \times nz$, into the DIMENSION statement of each individual array. The parallel versions incorporate FORTRAN90's ALLOCATABLE and ALLOCATE statements. Dynamic memory allocation for the large arrays is based on the values of nx, ny, nz, $d1$ and $d2$. The user only needs to correctly enter the values for nx, ny and nz and the program handles the necessary allocation procedures. Assuming nx, ny, nz, $d1$ and $d2$ are already known, then examples of using ALLOCATE per node for arrays *pix* and *vox* are:

```
      integer*2, allocatable :: pix(:,:,:), vox(:,:,:)

c
c myrank on ROOT equals 0,
c so just ROOT does this:

      if (myrank.eq.0) then
         allocate (pix(nx,ny,nz))
      end if

c But all nodes do this:

      allocate (vox(nx,ny,d1-1:d2+1))
```

In this example, only root allocates memory for *pix* and all processors allocate memory space

for its portion of vox. Root is the only processor which needs the entire pix array since it must pass out specific allotments to the workers. In conjunction with the DEALLOCATE statement, the memory used for pix is released after all passing of data is complete. Also vox is defined by its $d1$ and $d2$ limits and not the entire value, nz. This small range is at the heart of defining subsections of arrays per processor for parallel computations. Furthermore, this type of memory allocation used with the array vox is applied to all the large arrays found throughout all the programs. See Table 1 for a description of the array dimensions per program.

2.1.1 Finite Element

In the finite element programs, each voxel, of type INTEGER*2, must know the positions of its 27 nearest neighbors in a cubic array since that is a mathematical requirement of the calculation. In the serial versions, this is accomplished by using an array, ib, which is dimensioned as $ib(ns, 27)$. This array is of type INTEGER*4 and uses 54 (27×2) times the amount of memory as the original dataset. In fact, it serves no purpose as a calculating device, but is only used as a look-up (hash) table since it stores the 1-D positions of the 27 nearest neighbors for a given voxel.

In the parallel program, vox is dimensioned as a rank 3 array, $vox(nx, ny, d1 - 1 : d2 + 1)$. With this arrangement, it is trivial to find the indices of 27 nearest neighbors for a given voxel, $vox(i, j, k)$. The three nearest neighbors (including the voxel itself) in the z-direction have indices of $(i, j, k - 1)$, (i, j, k) and $(i, j, k + 1)$. Therefore the set of 27 nearest neighbors for this element is generated by adding ± 1 or 0 to any or all of the indices of the (i, j, k) triplet. The lowest neighbor has indices of $(i - 1, j - 1, k - 1)$ and the highest has $(i + 1, j + 1, k + 1)$. These values can be calculated on the fly or generated by using an adequately defined set of triply nested do-loops.

Special allowances have to be made when the current voxel is on the outside edges of the data cube (i.e. $i = 1$ or nx or $j = 1$ or ny). At these extremes, the value of i or j is interrogated and the values of $i - 1$, $i + 1$ are compared to 0 and nx. For example, if $i - i$ (or $j - 1$) equals 0, a modification takes place and the $(i - 1)^{th}$ (or $(j - 1)^{th}$) neighbor is replaced by the voxel with $i = nx$ (or $j = ny$). A similar modification takes place when the voxel having $i = nx$ $(j = ny)$. In this instance, the voxel with $i = 1$ (or $j = 1$) is used. This procedure is justified due to the periodic nature of the data.

Therefore, by switching over to a parallel implementation and this new indexing scheme, one has a dramatic improvement in memory savings since the additional storage of particle positions is no longer needed. This memory is now free to be put to better use.

Some small arrays (dk, $cmod$, $sigma$, etc...) that appear throughout the calculations have dimensions that are determined by the number of phases $nphase$ one has in the original dataset; this number is known a-priori, like nx, ny and nz. Arrays which need this value are defined as allocatable as well. This increases the flexibility of the program and contributes to a saving of memory by implementing dynamic allocation of additional arrays. In the serial version of ELAS3D, dk is dimensioned to $dk(100, 8, 3, 8, 3)$; the first index represents the phase number. Therefore, the new code allocates dk as $dk(nphase, 8, 3, 8, 3)$.

2.1.2 Finite Difference

In the finite difference programs, only the nearest neighbors are used to update a nodal voltage, so that the fact that the nearest neighbors are at $\Delta \mathrm{i} = \Delta \mathrm{j} = \Delta \mathrm{k} = \pm 1$ was used from the start. This made an array like ib in the finite element programs, even in the serial version, unnecessary. For an $nx \times ny \times nz$ size microstructure in the serial code, the program actually stores an array that is $(nx + 2) \times (ny + 2) \times (nz + 2)$ in size. The extra two layers of voxels in each direction

are periodic continuations of the microstructure. The real microstructure lies from $2, nx+1$; $2, ny+1$; and $2, nz+1$, and is recorded in the array *list* in the serial version. Therefore, in a manner of speaking, the serial code has a set of ghost layers in the x, y and z directions already built in. The parallel codes can determine (as described in the previous finite element section) if the position is real or belongs to a ghost layer and the array *list* becomes obsolete.

Figure 4: Depiction of original finite element data set in a serial based program showing the equivalency of the Z layers. The **Real** layers are real data.

With periodic updates of the x and y ghost sites, one can operate entirely on the real sites of the array, and freely take $\Delta \mathrm{i} = \Delta \mathrm{j} = \Delta \mathrm{k} = \pm 1$ to update each real site. The periodic boundary conditions are enforced via the regular updates of the boundary sites ($i = 1$ and $nx+2$, $j = 1$ and $ny+2$, and $k = 1$ and $nz+2$).

However, in creating parallel implementations for the finite difference codes, there is a unique twist since the serial versions already have 2 Z-ghost layers built in. (Figure 4) In the serial

Variable	**Serial**	**ELECFEM3D**	**ELAS3D**	**THERMAL3D**
pix/vox	ns	(nx,ny,d1-1:d2+1)	(nx,ny,d1-1:d2+1)	(nx,ny,d1-1:d2+1)
X	ns	(nx,ny,d1-1:d2+1)		
E	(ns,3)		(nx,ny,d1-1:d2+1,3)	
T	(ns+2,3)			(nx,ny,d1-1:d2+1,3), (2,3)
dk	(100,3,3), (100,8,3,8,3)	(nphase,3,3)	(nphase,8,3,8,3)	(nphase,8,3,8,3)
phasemod	(100,2)	(nphase,2)	(nphase,2)	(nphase,2)
sigma	(100,3,3)	(nphase,3,3)		(nphase,3,3)
eigen	(100,3,3)			(nphase,3,3)

Table 1: Array redimensioning from serial to parallel versions of ELECFEM3D, ELAS3D and THERMAL3D

codes, the real data lies in layers $Z' = 2$ through $nz + 1$. The serial ghost layers contain the information from the first ($z = 1$) and last ($z = nz$) real data in the $Z' = 1$ and $Z' = nz2$ layers, respectively. For the rest of the discussion, $nz2 = nz + 2$.

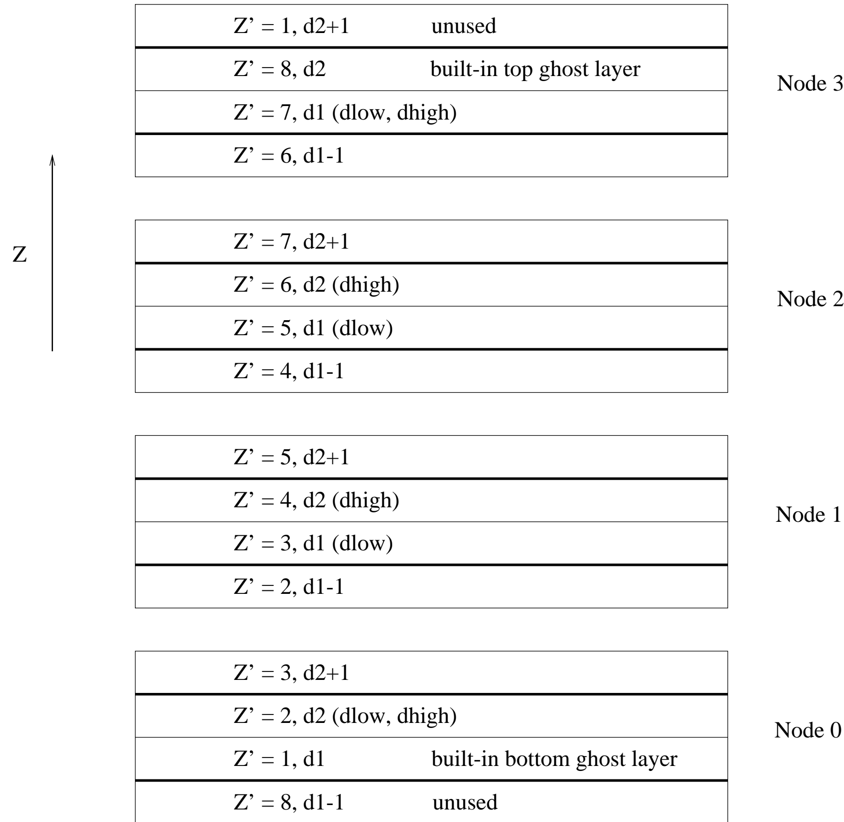

Figure 5: Depiction of ghost layers in the parallel finite element program showing the equivalency of the Z layers across 4 processors for the $nz = 6$, $nz2 = 8$ case. Note how the $d1 - 1$ layer of the bottom node & the $d2 + 1$ layer of the top node are unused in the calculation.

The parallel code is still set-up to split the layers dependent (i.e. calculate $d1$ and $d2$ per node) on the *overall* extent of z (Figure 5). Therefore, each node in the finite difference case will receive $\frac{nz2}{N}$ layers and not $\frac{nz}{N}$ like a finite element problem since $nz2$ is the overall extent of z. Therefore, before the parallelization and data split is made, it is known that the bottom node will already have the information it will need for its $d1-1$ ghost layer and the top node will have the layer which it needs for its $d2 + 1$ ghost layer. And interestingly enough, *they are already stored in the bottom node's d1 and top node's d2 layers, respectively, before the split.* Therefore, when the initial ghost layer production of *vox* (and subsequent arrays) in the parallel versions, the bottom node will only need to receive its $d2+1$ (top ghost) layer and the top node will need it's $d1 - 1$ (bottom ghost) layer through a message passing procedure.

Since it is known that before the splitting takes place, the bottom and top nodes already have 1 needed ghost layer, the code is designed with a new set of variables called *dlow* and *dhigh* which allow for the special cases of the lowest and topmost nodes and makes a one-to-one correspondence for $d1$ with *dlow* and $d2$ with *dhigh* for all other nodes. viz:

```
if (myrank.eq.0) then
   dlow = 2
```

```
else
   dlow = d1
end if

if (myrank.eq.nprocs-1) then
  dhigh = nz1
else
  dhigh = d2
end if
```

This type of code ensures that only the real data is used and the extra ghost layers (made by subroutine z_ghost) for Node 0 and the the northernmost node, remain out of the central calculation.

Figures 4 and 5 depict the case of $nz = 6$ for the serial and parallel cases. This increases the Z extent of this array to $nz + 2$, or 8. The Z=1 and Z=8 layers are the built in ghost layers, while layers with $2 \leq \mathrm{Z} \leq 6$ (nz) are the real data. The shading of the layers Real=1 and Real=6 corresponds to the ghost layer equvalency in the figure.

Figure 5 shows how the splitting of arrays is performed in the parallel case, with the number of processors arbitrarily set to 4. In this example, each processing node receives, $\frac{nz2}{N}$, or 2 layers per node. Notice how the values of *dlow* and *dhigh* are dependent on the rank values; namely *dlow* on root ($myrank = 0$) and *dhigh* on the northernmost ($myrank = nprocs - 1$). Interestingly, in this example, *dlow* and *dhigh* are equal ($dlow, dhigh = 2$) on rank=0 and are also equal ($dlow, dhigh = 7$) on rank=$nprocs - 1$.

When a nodal calculation occurs, the extrema on the Z-type loops occur between dlow and dhigh and in addition, one needs to know the values at Δk = ± 1. Therefore, it can be seen from the figure, the lowest ghost layer on Node 0 will be unused as well as the top most ghost layer on Node 3. And by induction, this will occur for all instances. These type of calculations appear in subroutines **PROD_nC** where matrix calculations and updates are carried out.

2.2 Parallel Algorithms

This section states the problem to be solved, describes the serial version and then uses that information to make relationships that are useful for the development of parallel algorithms for these programs. The suite of programs provided in NISTIR-6269 can be broken down into two distinct kinds of problems. The first set calculates properties based on the finite element method and the other uses finite difference techniques.

2.2.1 Finite Element

As alluded to earlier, the main problem to be solved is the minimization of the system's energy in response to an applied field. Specifically, the total energy stored in the elastic case or the total energy dissipated in the electrical conductivity case is maximized, such that the gradient of the energy with respect to the variables of the problems is zero. To minimize the energy, E_n, a function of many variables, u_m, the various partial derivatives must equal zero,

$$\frac{\partial E_n}{\partial u_m} = 0 \tag{10}$$

In the electrical conductivity case, the energy, E_n is:

$$E_n = \frac{1}{2} u_r D_{rs} u_s \tag{11}$$

where the elements, D_{rs} are the elements of the stiffness matrix.

The energy in the elastic case, E_n is:

$$E_n = \frac{1}{2} u_r D_{rs} u_s + b_r u_r + C \tag{12}$$

The energy in the thermal case, E_n is:

$$E_n = \frac{1}{2} u_r D_{rs} u_s + b_r u_r + C + T_r u_r + Y \tag{13}$$

The energies are described using expressions that are quadratic in the u array; for ELECFEM3D this means quadratic in the nodal voltages and in ELAS3D, it is quadratic in the nodal displacements. The array D, is called the stiffness matrix. It contains all the physical information of the material properties of the system, along with the geometrical properties of each finite element used in this case. All finite elements are just made from voxels (3-D) or pixels (2-D), so are cubes or square.

The basic problem being solved is a composite one: what are the effective properties of a mixture of material phases with different properties. The elastic case just has elastic moduli, the electrical case just has conductivities, while the thermal case has thermal expansion constants as well as elastic moduli. In each case, a system energy is minimized in order to solve the relevant partial differential equations. In the elastic and thermoelastic cases, the overall stored energy is being minimized. In the electrical case, where current flows, the dissipated energy is being minimized. The electrical conductivities could also be interpreted as thermal conductivities if so desired, expanding the range of problem application.

2.2.2 Finite Difference

The energy equations for the finite difference programs are a little different. For a real conduction problem, with conductivity (which could be variable in space, but not time), the dissipated energy is:

$$U = \int d^3 r E_i \sigma_{ij} E_j \tag{14}$$

where E_i is the electric field at a certain point in space, and the integral is over all space. However, to derive the steady state conductivity problem, one starts rather from the charge conservation equation,

$$\nabla \cdot \vec{j} + \frac{\partial \rho}{\partial t} = 0 \tag{15}$$

where $\vec{j}$ is the current density and ρ is the charge density. For steady state problems, one ends up with simply $\nabla \cdot \vec{j} = 0$ or $\nabla \cdot (\sigma \vec{E}) = 0$. Using the definition of E in terms of the gradient of the voltage results in the Laplace equation, $\sigma \nabla^2 V = 0$, in regions of uniform values of the conductivity. The appropriate boundary conditions between regions of different conductivity are applied in the program in subroutine **BOND**, which gives the correct value for the conductance of bonds that pass from one region to another. The program then essentially assumes that the microstructure is made up of a finite number of uniform conductivity regions.

Upon translating into finite difference language, one ends up with the equation $Au = 0$, where u is the array of voltages, and A is a matrix made up of the conductivities and the voxel geometry. To use the conjugate gradient method requires that an energy in quadratic form be minimized, so this equation is squared, resulting in $u^\dagger A^\dagger A u = 0$. The quadratic form being minimized is then really $A^\dagger A$, or the square of a matrix similar to the stiffness matrix in the finite element problems. This is why the finite difference programs do not usually converge as rapidly as do the finite element programs, since the square of the stiffness matrix is used instead of the matrix itself.

3 Program Operation and USER information

Users familiar with the serial version of the code will find it relatively easy to use this new parallel version. This section lists changes with respect to the serial code, describes the necessary user input, generated output, describes the program's operative behavior with specific regard to the MPI subroutines, and finally gives a description of the other supporting subroutines for the finite difference and finite element programs.

3.1 Notable changes from the serial version

Here is a list of the major changes that the user who is accustomed to the serial code may notice.

1. The parallel version defaults to a double precision calculation.
2. All of the principal arrays used in this code have been changed from a one-dimensional vector representation into a 3-D vector representation, i.e. the vector $pix(m)$ from the serial version has its parallel counterpart, $pix(i, j, k)$. This 3-D representation describes the data in a more natural way and also allows easier manipulation in a multiple processing environment for communication between processing nodes.
3. The array $vox(nx, ny, d1 - 1 : d2 + 1)$ is used throughout this program in lieu of pix and is invoked to do the rest of the computations. Pix itself is dynamically allocated and is deallocated as soon as the array vox is created.
4. The principal data arrays in this program have been changed. The arrays u, b, gb, h, and Ah were originally dimensioned as $(ns, 3)$ in the serial problem, but in this new paradigm they are changed to reflect the 3-dimensionality of the problem. They are now dimensioned on a per node basis (akin to the array vox) to $(nx, ny, d1 - 1 : d2 + 1, 3)$; each has their own set of top and bottom ghost layers as well.
5. Arrays that are dependent on nx, ny, nz (like the aforementioned set) and $nphase$ e.g., dk, $cmod$ and $prob$ are now dynamically allocated. This saves the user time editing the code for his problem since the dimensions do not have to be changed from problem to problem. One only has to change the respective values of nx, ny, nz and $nphase$.

3.2 Inputs

This section will briefly describe each of the individual input items (and their data types) a user needs in order to use this code. The following list of variables are the same as in the serial code, except for the flag, $pflag$, which is used for timing information. To find these variables in the code, search for occurrences of **USER** - they are located nearby. The variables are listed in the order in which they are found in the program. The input that is generic to all the codes will be presented first and then in each subsection, a description of the specific input per program is presented.

1. **nx,ny,nz** (integer*4): The dimensionality of the data, i.e. $pix(nx, ny, nz)$.
2. **nphase** (integer*4): The number of phases represented in your microstructure.
3. **gtest** (double precision): The stopping criteria which is compared to gg (=$gb \times gb$).
4. **pflag** (integer*4): A flag used for printing timing information. 0 suppresses printing and 1 prints out all per processor timing information .
5. **Input file name (Unit 9)** (character string): The name of the file which contains the 3-D image under investigation. The input file contains 2-byte integers in which the value of each element represents the phase at that position in the microstructure (1,2,3,...,$nphase$).

6. **Output file name (Unit 7)** (character string): The main set of output is generated and placed into this file as the job proceeds. Description of its contents is in the **Output** section.
7. **npoints** (integer*4): This integer tells the program how many distinct input files will be run. Typically this value equals 1, but can be changed.
8. **kmax** (integer*4): Maximum number of times subroutine **DEMBX_MPI** will be called.
9. **ldemb** (integer*4): Number of conjugate gradient steps performed during each call of **DEMBX_MPI**.

3.2.1 ELECFEM3D

1. **sigma(nphase,3,3)** (integer*4): The electrical conductivity tensor of each phase. The user can make the value of *sigma* to be different for each phase of the microstructure if so desired.
2. **ex,ey,ez** (double precision): Global electric field applied to microstructure.

3.2.2 ELAS3D

1. **phasemod(nphase,2)** (integer*4): An array which contains the Young's modulus, $phasemod(i, 1)$, and Poisson ratio, $phasemod(i, 2)$, of the ith phase. This array is now initialized in its own subroutine called **phasemod_init** instead of within MAIN.
2. **exx,eyy,ezz,exz,eyz,exy** (double precision): Global strains applied to microstructure.

3.2.3 THERMAL3D

1. **phasemod(nphase,2)** (integer*4): An array which contains the Young's modulus, $phasemod(i, 1)$, and Poisson ratio, $phasemod(i, 2)$, of the ith phase. This array is now initialized in its own subroutine called **phasemod_init** instead of within MAIN.
2. **eigen(nphase,6)** (double precision): Thermal strains of each phase: $eigen(nphase, 1) =$ xx, $eigen(nphase, 2) =$ yy, $eigen(nphase, 3) =$ zz, $eigen(nphase, 4) =$ xz, $eigen(nphase, 5)$ = yz, $eigen(nphase, 6) =$ xy.

3.2.4 DC3D

1. **sigma(nphase,3)** (double precision) Values of the diagonal elements of the conductivity tensor for each phase (conductivity tensor is diagonal only). 1,2,3 = x,y,z, respectively.
2. **ex,ey,ez** (double precision): Components of applied field, $E = (ex, ey, ez)$.

3.2.5 AC3D

1. **sigma(nphase,3)** (double complex) Values of the diagonal elements of the conductivity tensor for each phase (conductivity tensor is diagonal only). 1,2,3 = x,y,z, respectively.
2. **ex,ey,ez** (double precision): Components of applied field, $E = (ex, ey, ez)$.
3. **nfreq** (integer*4) : Indicates how many frequencies are to be used at which to compute the complex conductivity. The program was originally set up to simulate the experimental probe of impedance spectroscopy, which scans a sample over a number of frequencies of applied electrical signal. In the numerical code, one often desires to scan over a similar set of frequencies. When doing so, the program converges better if the complex voltages

from the last computation are used for the initial voltage values for the next frequency computation. When the frequencies are close, like they usually are in a sweep, this works well and save an appreciable amount of computer time. If the frequencies are far apart, there is little benefit to this procedure, and one might as well use a uniform field initial condition at all frequencies.

4. **ncheck** (integer*4) : Subroutine **DEMBX_AC** will write out the total current and norm of the gradient squared every *ncheck* gradient steps.

3.3 Outputs

The main set of output data is generated and placed into the output file *Unit*7 which the user names within the program. The output that is generic to all the codes will be presented first and then in each subsection, a description of the specific output per program is presented.

All programs report the following information in the first few lines of the output file.

- Prints out "MICRO" (which microstructure is currently being calculated); the program can be adjusted to use multiple input files.
- Prints out the values for nx, ny, nz, ns, $nprocs$, where $nprocs$ = number of processors for this job.
- The volume fraction of each phase.
- The input values of the applied fields (electrical, mechanical, thermal)
- Values of conductivity tensors or bulk and shear elastic moduli per phase. This is dependent on the specific program, but always appears at this time.
- Final C = C
- Initial Energy = E_0
- gg = g_0

The programs will now print out the following information after each *ldemb* or *ncheck* conjugate gradient steps until convergence.

- Energy = E_1 gg = gg_1
- Number of conjugate steps = up to that point
- Root took S_1 s for *ldemb* conjugate steps.
- *Intermediate results*

The last list entry, *Intermediate results*, refers to the specific observables, which the given program calculates.

If there is more than one microstructure under investigation, each node will print its execution time for that microstructure. Otherwise, they will print their overall execution time.

3.3.1 ELECFEM3D

The intermediate results appear as:

- Current in x direction = **cuxxp**
- Current in y direction = **cuyyp**
- Current in z direction = **cuzzp**

The final results of this program are displayed as:
Average current in x direction= **cuxxp**
Average current in y direction= **cuyyp**
Average current in z direction= **cuzzp**
ic number of conjugate gradient cycles needed

3.3.2 ELAS3D

- The phase number, its Young's modulus $phasemod(i, 1)$, and Poisson ratio $phasemod(i, 2)$ which are defined in **phasemod_init**.

The intermediate results appear as:

- stresses: xx,yy,zz,xz,yz,xy **strxxp,stryyp,strzzp,strxzp,stryzp,strxyp**
- strains : xx,yy,zz,xz,yz,xy **sxxp,syyp,szzp,sxzp,syzp,sxyp**

where *strxxp*, *stryyp*, *strzzp*, *strxyp*, *strxzp*, *stryzp* are the six Voigt volume averaged total stresses and *sxxp*, *syyp*, *szzp*, *sxyp*, *sxzp*, *syzp* are the six Voigt volume averaged total strains.

When the program finishes, the overall bulk modulus, shear modulus, Young's modulus and Poisson's ratio are printed as well.

bulk modulus = **bulk**
shear modulus = **shear**
Youngs modulus = **young**
Poissons ratio = **pois**

3.3.3 THERMAL3D

The intermediate results appear as:

- stresses: xx,yy,zz,xz,yz,xy **strxxp,stryyp,strzzp,strxzp,stryzp,strxyp**
- strains : xx,yy,zz,xz,yz,xy **sxxp,syyp,szzp,sxzp,syzp,sxyp**
- macrostrains in same order **u2(1,1),u2(1,2),u2(1,3),u2(2,1),u2(2,2),u2(2,3)**
- average of the macrostrains **(u2(1,1)+u2(1,2)+u2(1,3))/3)**

When the program finishes, the above values are once again outputted but with their final values.

3.3.4 DC3D

The intermediate results appear as:

- Current in x direction = **cuxxp**
- Current in y direction = **cuyyp**
- Current in z direction = **cuzzp**

The final results of this program are displayed as:
Average current in x direction= **cuxxp**
Average current in y direction= **cuyyp**
Average current in z direction= **cuzzp**
ic number of conjugate gradient cycles needed

3.3.5 AC3D

The intermediate results appear as:

- No. **nf** angular frequency = **w** radians
- **icc** : The number of the current iterations which is a multiple of **ncheck** until it reaches the final value.
- **gg** : A complex number that is the value of the gradient squared ($gb \times gb$).
- cuxxp = **cuxxp**
- cuyyp = **cuyyp**
- cuzzp = **cuzzp**

At the end of the calculation for a given frequency, the program reports the following:
Average current in x direction= **cuxxp**
Average current in y direction= **cuyyp**
Average current in z direction= **cuzzp**
ic number of conjugate gradient cycles needed

3.4 MAIN

The programs are essentially the same as the serial version until the initial data set is read from the input file. Throughout this program, when I/O has to be done, it is only done by the root node (rank=0). Dynamic allocation of several arrays based on the problem size (nx, ny, nz) as well as those based on *nphase* also occur here. There are some small difference in details between the finite element and finite difference programs, which will be obvious when looking more closely at the programs.

- Initialize variables.
- Initialize MPI.
- Root calculates z extents, $d1$ and $d2$.
- Root reads in original data set and passes appropriate sections to other processing nodes.
- Dynamically allocate large arrays per processor.
- Call **FEMAT_MPI** - this computes the local stiffness matrices and any other auxiliary arrays needed. The local stiffness matrices are used in **ENERGY_MPI** and **DEMBX_MPI** to construct the global stiffness matrix as the various arrays are updated in the conjugate gradient process.
- Calculate initial u array per processor.
- Call **ENERGY_MPI** - this computes the initial energy of the system, based on the initial conditions, and also computes the initial value of gb, the gradient of the energy array. Both of these are needed as initial inputs to **DEMBX_MPI**.
- Enter loop which calls **DEMBX_MPI** and **ENERGY_MPI**. **DEMBX_MPI** is the conjugate gradient subroutine that actually performs the conjugate gradient update to the variables of the problem. It is performed *ldemb* number of times, and is then exited and **ENERGY_MPI** is then called again. This is done in order to see how the relaxation is going, and whether the energy has relaxed enough. It is usually not possible, a priori, to set the value of *gtest* so that good results are obtained in a reasonable amount of time. If *gtest* is set too small, many conjugate gradient cycles could be wasted to try to push the accuracy of the answer beyond what is necessary. If *gtest* is too large, then the final answer obtained might not be accurate enough.

- After convergence, calculate final results, i.e. stress, strain, or current per pixel and then produce a global summation.

3.5 Finite Element Subroutines

In this section, a discussion of the functionality of the MPI based subroutines used in the parallel codes, ELECFEM3D_MPI.f, ELAS3D_MPI.f and THERMAL3D_MPI.f are presented. Since each program will have its own specific needs, the operations will be essentially identical except for the dimensionality of the operand in question. See Table 1 for the variations in dimensionalities.

3.5.1 FEMAT_MPI

A generic form of this subroutine is used to calculate

- Elements of the stiffness array, dk.
- Values needed by the minimization procedure: array b and variable C.

The term, C, and array, b, are calculated in essentially the same manner as in the serial code; there are contributions to each from the x,y,z faces, the xy, xz, yz edges and the corners of the original microstructure dataset. But C and b are *per-processor* values in the parallel code. This means that each processor will calculate its own contribution, based on its chunk of data, to the overall value. Once a node determines its contribution, all partial results are sent to the master node who adds them together and passes back the result of this operation and broadcasts it to all nodes.

The nominal loops to calculate the positional contributions to C and b are closely related to the serial case. One usually loops over 2 of the 3 Cartesian coordinates, (x,y,z). In the code, a loop in the x direction uses the variable i as its index, viz: $i = 1, nx$. Similar arrangements are made for the y and z directions using the variables j and k, respectively as in $j = 1, ny$ and $k = 1, nz$. Some k-type loops in the subroutine (z-direction) use the limits $k = 1, nz - 1$, but it is important to be mindful that the data, as well as other large arrays, are split across the processing nodes in the z-direction. Additional accommodations and precautions have to be made for this fact, but at the same time the code must be generic enough so all processors can execute it. It is especially important for the $k = 1, nz - 1$ loops.

This is accomplished by introducing a new variable, dn, in concert with an **if** statement. dn is initially assigned to $d2$, but if $d2$ equals nz, then it takes on the value $nz - 1$. Note that this only occurs at the processor with the highest rank value. So during these calculations, all levels on the lower ranks are included and the $d2$ is excluded only on the highest rank.

Another interesting artifact of this calculation occurs when calculating the contributions for C and b from the Z-face, x=nx z=nz edge, y=ny z=nz edge and x=nx y=ny z=nz corner. In these 4 cases, only the highest rank processor is needed since this is the one where k has values of nz and $nz - 1$.

The accuracy of C is increased in this algorithm. At each instance when the contribution of C is calculated (called *cterm* in the code), it is compared to zero (0). A positive *cterm* is added to *cpos*. Likewise a negative *cterm* is added to *cneg*. At the end of the calculation, *cpos* and *cneg* are summed on a node, which yields the overall C per processor. This prevents adding very large numbers of one sign with very small numbers of the opposite sign.

As mentioned, the elements of array b are calculated with a *per-processor* methodology as well. In the original serial codes, ELECFEM3D.f and ELAS3D.f, the elements of b are calculated as $b(ib(m, is(mm)))$ and $b(ib(m, is(mm)), nn)$, respectively. Notice the dependence on the hash table array, ib. In the parallel versions, the need for the hash table ib is eliminated and b is an array of rank 3 or 4, respectively.

But a calculation for an element of b is more complicated than generating a constant term like C. One must notice that the corresponding loop variables (i, j, k) are not the same as the indices of b, namely (ipx, ipy, ipz) which is currently being calculated. These indices can be thought of as a set of *influenced positional* values. Therefore, a calculation using $vox(i, j, k)$ will influence the values of 27 separate and distinct elements of b, which correspond to its 27 nearest neighbors. In other words, a contribution to an element of b comes from 27 distinct elements of vox. The subroutine **ipxyz** is used to generate the values of an influenced triplet (ipx, ipy, ipz) using a given set of parameters (i, j, k and mm).

So one saves on the overall amount of memory for this calculation by eliminating ib, but the price to be paid is executing **ipxyz** multiple times.

After the distinct faces, edges and corner loops are finished, b is essentially complete. However no one node has the final values of $b(i, j, d1)$ and $b(l, m, d2)$. Part of the results are found on contiguous nodes, i.e. processor $P - 1$ and $P + 1$, respectively. Therefore, it is necessary to pass the bottom ghost layer of processor $P + 1$ to P then add to the $d2$ level of P once it is at the requesting node. This addition completes the calculation of b. A similar set of calculations involving the top layer of $P - 1$ and the $d1$ level of P are also required. In a succinct manner, it can be expressed as:

$$B(i, j, d1)_P = b(i, j, d1)_P + b(i, j, d2 + 1)_{P-1} \tag{16}$$

$$B(i, j, d2)_P = b(i, j, d2)_P + b(i, j, d1 - 1)_{P+1} \tag{17}$$

where capital B represents the final value of the element and lower-case b's represent the partial results.

At this point b is completely determined and updating the top and bottom ghost layers is the final step. This is accomplished by another use of subroutines **b2t_dp** and **t2b_dp**.

THERMAL3D_MPI.f uses a larger and slightly more complex form of **FEMAT_MPI**. Remember that THERMAL3D_MPI.f incorporates 1 additional large array, T. T is a linear term in displacements (like b) but arises from thermal strains and the constant term Y is similar to C, but arising from the thermal strains, not the applied or macrostrains. Since the overall system strains (macrostrains) are dynamic variables in the thermoelastic problem, the value of C will change throughout the operation of the program. However, the value of Y is constant, since it is a function only of the elastic moduli and the thermal strains, which are system variables and thus do not change. In the serial case, the array T was defined differently than b was in ELAS3D. Instead of being defined as $T(ns, 3)$, it was defined as $T(ns + 2, 3)$. This additional expansion of dimensions is handled by making 2 arrays for the parallel counterpart. The first is to dimension T as $T(nx, ny, d1 - 1 : d2 + 1, 3)$ (like b in ELAS3D_MPI.f) but the 6 extra terms of $T(ns + 1 : ns + 2, 3)$ are put into a new array called $T2$ and dimensioned $T2(2, 3)$. Therefore, the generic array in THERMAL3D_MPI.f, $X(ns + 2, 3)$ is converted to 2 arrays, one with the standard parallel dimensions of $X(nx, ny, d1 - 1 : d2 + 1)$ and then also $X2(2, 3)$.

Calculations for T elements are handled in a similar manner as b from ELAS3D_MPI.f. Again, the (i, j, k) indices of the loop do not correspond to the element in question. The program also uses subroutine **ipxyz** and calculates a set of influenced positional parameters as well. To complete the calculation for T, implement equations similar to (16) and (17) above and do the final updating of the ghost layers.

Although the $T2$ array is small (2,3), it has contributions from the faces, edges and corner like the array b in ELAS3D_MPI.f. At the end of determining these per processor arrays, one must also form the element-by-element sum to end up with the final version of $T2$. This is accomplished by using a call to MPI_ALLREDUCE:

```
do ipp=1,2
do jpp=1,3
```

```
      call MPI_ALLREDUCE(T2(ipp,jpp), t2temp, 1, MPI_DOUBLE_PRECISION,
     &                   MPI_SUM,MPI_COMM_WORLD,ierr)
      T2(ipp,jpp) = t2temp
      end do
      end do
```

t2temp is a temporary variable used to store the global sum of the per-processor $T2(ipp, jpp)$ values. Then it reassigns $T2(ipp, jpp)$ with the global sum before it ends. After calculating $T2$, this implementation also has to calculate an array b. This term is linear in displacements as well and is generated identically to b in ELAS3D_MPI.f.

3.5.2 ENERGY_MPI

This subroutine calculates the gradient of the energy, *gb*, but calls the subroutine **GBAH** to do so. In the serial version, the majority of the subroutine **ENERGY** was dedicated to calculating the *gb* array. It was much easier to create a subroutine for this calculation since the next subroutine, **DEMBX_MPI**, has the same functional form for creating its *Ah* array.

The differences here appear when calculating *utot*. One needs to get the per-processor value *dutot* and then make a global sum with a call to MPI_ALLREDUCE. The last two lines in the subroutine are simple assignments:

```
      utot = utot + C
      gb = gb + b
```

The first line does an update on a single variable, *utot*. But the second line does an update of the entire array *gb* with array b using FORTRAN90 syntax. Keep in mind, each processor is just updating its own local copy of *gb* with its local copy of b.

In THERMAL3D_MPI.f, there is additional complexity for this subroutine. After it updates *gb* (like above), it must calculate its value for the "constant" macrostrain energy term, C, using the *zcon* and $u2$ arrays. This value C is added to *utot* (also above), but to get the final value of *utot*, one must add the constant Y from **FEMAT_MPI**.

3.5.3 DEMBX_MPI

This subroutine calculates the gradient relaxation process and the h and Ah arrays, but calls the subroutine **GBAH** to generate them. It behaves similarly to the serial version, except in cases when FORTRAN90 array syntax was implemented for lines like:

```
      u=u-lambda*h
      gb=gb-lambda*Ah
```

In the FORTRAN77 serial version, these lines looked like:

```
      do 540 m3=1,3
      do 540 m=1,ns
      u(m,m3)=u(m,m3)-lambda*h(m,m3)
      gb(m,m3)=gb(m,m3)-lambda*Ah(m,m3)
540   continue
```

3.5.4 GBAH

When calculating the *gb* or *Ah* arrays in the serial program, one can see that they are essentially the same calculation with a 1:1 correspondence between the arrays *u* and *h* as well as *gb* and *Ah*. Therefore, it is logical to create this subroutine and give input parameters which determine if one is calculating *gb* or *Ah*.

Regardless if one wants *gb* or *Ah*, the local array that is calculated in **GBAH** is the array *om*; the place from which **GBAH** is called and its parameters determine if *gb* or *Ah* is going to be calculated.

In this subroutine, elements of the array *om* (output matrix) are being calculated on a per node basis. In addition to that, FORTRAN90 syntax to sum over all values of an array index to calculate the terms removes the extra loop ($n = 1, 3$) that was found in the serial version. An example calculation from ELAS3D_MPI.f and ELAS3D.f should illustrate the point. Note that the last index of *om*, j, would not appear in ELECFEM3D_MPI.f (cf. Table 1).

```
      om(im,jm,km,j) =

c u(ib(m,1),n)
     & SUM ( uh(im,ifya,km,:)*(
     & dk(vox(im,jm,km),1,j,4,:)
     &+dk(vox(ifxb,jm,km),2,j,3,:)
     &+dk(vox(im,jm,km-1),5,j,8,:)
     &+dk(vox(ifxb,jm,km-1),6,j,7,:) ))+

c u(ib(m,2),n)
     & SUM ( uh(ifxa,ifya,km,:)*(dk(vox(im,jm,km),1,j,3,:)
     &+  dk(vox(im,jm,km-1),5,j,7,:) )) + ...
```

The $u(ib(m,1),n)$ and $u(ib(m,2),n)$ terms in the comments refer to the like terms found in the serial version, ie.

```
      gb(m,j)=gb(m,j)+
     &        u(ib(m,1),n)*(
     &        dk(pix(ib(m,27)),1,j,4,n)
     &+       dk(pix(ib(m,7)),2,j,3,n)
     &+       dk(pix(ib(m,25)),5,j,8,n)
     &+       dk(pix(ib(m,15)),6,j,7,n) )+

     &        u(ib(m,2),n)*(
     &        dk(pix(ib(m,27)),1,j,3,n)
     &+       dk(pix(ib(m,25)),5,j,7,n) )+ ...
```

om exhibits the same behavior as *Ah* and *gb*, in other words, it must also have top and bottom ghost layers as well. Therefore, before **GBAH** returns *om* back to the calling routine, it has already created the required ghost layers by judicious use of the subroutines **t2b_dp** and **b2t_dp**.

But the most intriguing part of the calculation involves the determination for the necessary elements of *uh* and *vox* in the above calculation. Note that *uh* is merely a parameter; *uh* equals *u* when called from subroutine **ENERGY_MPI** but equals *h* when called from **DEMBX_MPI**. This parameter is named *uh* to remind the user of its duality.

Before the code was written, it was necessary to determine what the (i, j, k) indices for the uh and vox arrays for each of the 27 terms will be, keeping in mind ib and Table 3. After one has done this, their (i, j, k) indices can be deduced. However, one must also keep in mind the periodic nature of uh and vox in the x and y directions; periodicity in the z direction is assured due to the ghost layers. Therefore when looping over im and jm (ie. x and y directions), special precautions (in the form of if-statements) are taken when $(im + 1) > nx$ or $(im - 1) < 0$, when the value of that particular i-type coordinate is assigned to 1 or nx, respectively, with similar arrangements for jm.

For a given set of (im, jm, km) loop variables, if the serial code says to invoke $ib(m, 1)$, this corresponds to the listing in Table 3 for $ib(m, 1)$, which says the required neighbor has a (Δi,Δj,Δk) setting of (0,1,0). This implies that the proper element to use in this calculation would be $(im, jm + 1, km)$. There is a change in variable for $jm + 1$ which is now assigned to $ifya$. This specialized nomenclature is summarized here:

```
im + 1 => ifxa
im - 1 => ifxb
jm + 1 => ifya
jm - 1 => ifyb
```

Therefore, when one encounters a $ifxa$, $ifxb$, $ifya$ or $ifyb$ term, it can immediately be identified as a term that needs to be fixed or adjusted according to the above specifications. This occurs not only for uh, but also for $vox(im, jm, km)$ as it is a parameter for dk.

3.5.5 STRESS_MPI

This subroutine calculates the individual stresses and strains per pixel per processor. MPI code is needed to sum the individual contributions per processor onto the root (master) node (like the C calculation). The 12 macroscopic stresses and strains are broadcast to each node, but root is the lone processor who uses these values for output purposes.

3.5.6 CURRENT_MPI

This subroutine behaves very like **STRESS_MPI**, except it calculates the current contribution from each pixel. A call to MPI_ALLREDUCE is needed to sum the individual contributions per processor onto the root (master) node. The 3 macroscopic current contributions (x,y,z) are available to each node, but root is the lone processor who uses these values for output purposes.

3.6 Finite Difference Subroutines

In this section, a discussion of the functionality of the MPI based subroutines used in the parallel codes, DC3D_MPI.f and AC3D_MPI.f are presented. The dual labeling of the subroutines in this section reflects the fact that they are used in DC3D_MPI.f or AC3D_MPI.f.

3.6.1 BOND_DC, BOND_AC

The subroutine bond applies the correct boundary conditions between regions of different conductivity by choosing the correct value for the conductivity of a bond that extends across a boundary. Remember that in the finite difference programs, the node is thought of as being at the center of a pixel, with nearest neighbor bonds connecting nodes. A bond that connects nodes that are in regions of different conductivities gets assigned a conductance that is a function of the voxel geometry and is a series combination of the two different values of conductivities. When a

bond connects two nodes of the same conductivity, then that value of the conductivity is simply used to assign the conductance of the bond.

3.6.2 DEMBX_DC, DEMBX_AC

This is structurally like **DEMBX_MPI** from the finite element codes.

3.6.3 CURRENT_DC, CURRENT_AC

This is structurally like **CURRENT_MPI** from the finite element codes.

3.6.4 PROD_DC, PROD_AC

Analogously to the subroutine **GBAH**, in the finite element programs, these routines perform the large matrix multiply necessary for the conjugate gradient updating process. Generically, this subroutine does the matrix multiplication calculation: $yw = A \times xw$, with xw as the input vector. In practice, this subroutine uses the parameter sets, (u, gb) and (h, Ah), where u and h are passed as the input vectors (xw) and used to calculate new gb and Ah (yw) arrays. The large array A in the above equation is substituted with the gx, gy and gz arrays which describe the bond conductance network of the microstructure.

The final calculation on vector yw is to correct for terms at periodic boundaries. This is a straightforward per processor operation for the updates involving the x and y faces. But the Z face updates for the genuine ghost layers occur through a message passing procedure when using multiple processors. In this case, the bottom node gets the information for it's genuine bottom ghost layer ($nz = 1$) from the top most node, (level $nz = nz1$) called *highrank* in the code. The top node gets it genuine top ghost layer ($nz = nz2$) from Node 0, (level $nz = 2$), called *lowrank*. These allowances must take place in conjunction with the actual calling of **z_ghost_dp** since **z_ghost_dp** by itself will not create the aformentioned genuine ghost layers needed for vector yw.

3.7 Other Subroutines

In this section, a discussion of the functionality of the MPI based subroutines used by all the parallel codes are presented.

3.7.1 z_ghost_int,z_ghost_dp,z_ghost_cmplx

Subroutines to create a ghost layer with INTEGER*2, DOUBLE PRECISION or DOUBLE COMPLEX elements in the Z-direction of the data cube. They internally call the **t2b_XX** and **b2t_XX** subroutines below.

```
call z_ghost_int(vox,nx,ny,nz,d1,d2)
call z_ghost_dp( u,nx,ny,nz,d1,d2)
call z_ghost_cmplx( u,nx,ny,nz,d1,d2)
```

3.7.2 xy_ghost_dp,xy_ghost_cmplx

Subroutine to create a ghost layer with DOUBLE PRECISION or DOUBLE COMPLEX elements in the x and y directions of the data cube. Only used in finite difference programs, DC3D_MPI.f and AC3D_MPI.f.

```
call xy_ghost_dp(y,nx,ny,nz,d1,d2)
call xy_ghost_cmp(y,nx,ny,nz,d1,d2)
```

3.7.3 t2b,b2t,t2b_dp,b2t_dp, t2b_cmplx,b2t_cmplx

These subroutines, called by z_ghost_(int,dp,cmplx), are used to create top and bottom ghost layers. **t2b** and **b2t** are used with array *datn* and manipulate data that are 2 bytes long. Calls to **t2b** or **b2t** look like:

```
call t2b(bot,top,nx,ny)
call b2t(bot,top,nx,ny)
```

This sequence will create a new bottom ghost layer on node $N+1$ with the value in *bot* as the new bottom ghost layer and then make a new top ghost layer on node $N-1$ with *top* as the new top ghost layer.

The subroutines **t2b_dp** and **b2t_dp** are used for all other arrays since they are used with double precision values, hence the “dp” suffix.

The calls to **t2b_dp** or **b2t_dp** are similar:

```
call t2b_dp(bot,top,nx,ny,i)
call b2t_dp(bot,top,nx,ny,i)
```

The last parameter, an integer, can in general be anything, but for our specific problem, i equals 3, which is a direct consequence of the dimensionality of u, b, gb, h and Ah in the elastic and thermal programs.

3.7.4 m2ijk

Typically, one is accustomed to using this function to determine the 1-d ordinal count of array elements. The function to perform this is:

$$m = nx \times ny \times (k-1) + nx \times (j-1) + i$$

where (i, j, k) is the set of indices in an array; nx and ny are the dimensions in the x and y directions, respectively; and m is the ordinal count.

However, this subroutine is the inverse function of the equation above, i.e. it takes m, nx, ny and nz as the input and returns the (i, j, k) triplet. It does this through a series of modular mathematical functions. The call to **m2ijk** is:

```
call m2ijk(m,i,j,k,nx,ny,nx)
```

3.7.5 ipxyz

This subroutine is used to calculate which element of the array b will be calculated when using an input triplet of (i, j, k), returning it as the triplet (ipx, ipy, ipz). The call to **ipxyz** is:

```
call ipxyz(mm,i,j,k,ipx,ipy,ipz,nx,ny,nz)
```

3.7.6 phasemod_init

This subroutine initializes the values in the *phasemod* array here instead of in the main program. It occurs here now as it is easier to modify. The call to **phasemod_init** is:

```
call phasemod_init(phasemod,nphase)
```

3.7.7 dpixel

This subroutine reads in the original data set. The call to **dpixel** is:

```
call dpixel(nx,ny,nz,ns,nphase,pix)
```

3.7.8 dassig

This subroutine calculates the volume fraction of phase i. The call to **dassig** is:

```
call dassig(nx,ny,nz,nphase,prob,pix)
```

3.8 Makefiles and Execution scripts

3.8.1 PC Linux Cluster using PBS

This is an example of the Makefile used to create the executable using the Portland Group FORTRAN90 compiler and the LAM libraries.

```
LAM_HOME=$(LAMHOME)

CC  = LAMHCC=cc hcc
FFLAGS = -fast -O2 -r8
F77 = LAMHF77=pgf90 hf77

LIBS=

elas3d_mpi:  elas3d_mpi.o
        $(F77) $(FFLAGS) -o elas3d_mpi $< $(LIBS)

clean:
        rm -f *.o

## Rules
.c.o:
        $(CC) $(CFLAGS) -c $<

.f.o:
        $(F77) $(FFLAGS) -c $<
```

This is a PBS execution script to submit this job in an 8 cpu queue using LAM.

```
#!/bin/bash
#   Example script for submitting a job to PBS on hudson.
#
#  Change the variables "prog" and "prog_args" below as needed.
#  Also, change the -N, -q, and -l nodes=? lines as needed.
#
#PBS -N Elas3d_8p.JOB
#PBS -q medium
```

```
#PBS -l walltime=168:00:00
#PBS -l nodes=8

#  -N ????      : job name
#  -q ????      : queue
#  -l walltime= : wallclock time, 1 hour-> 1:00:00
#  -l nodes=    : number of nodes to use

# Move to the directory from which this file was qsub'd
cd $PBS_O_WORKDIR

jobnum=`echo $PBS_JOBID | awk 'BEGIN {FS = "."} { printf("%05d", $1)}'`

echo jobnum $jobnum

echo
echo Running on nodes:
cat $PBS_NODEFILE
echo
echo


wrkdir=$PBS_O_WORKDIR
prog=elas3d_mpi
prog_args=""

cd $wrkdir

######  Start the program.

echo
echo "Starting user program $prog"
date
#
# note that the PBS nodefile can serve as the LAM boot schema file
LAMBHOST=$PBS_NODEFILE

# boot lam
echo
echo lamboot $LAMBHOST
lamboot $LAMBHOST

# run mpi job using replicated, transposed medium
options="-v -w -D -O -c2c"
echo
cmd="mpirun $options N $wrkdir/$prog -- $prog_args"
echo "Run command:$cmd"
$cmd
```

```
#Kill lam daemons and clean up hostfile
echo
echo $LAMHOME/bin/wipe -v $LAMBHOST
$LAMHOME/bin/wipe -v $LAMBHOST

date
echo
exit
```

3.8.2 SGI Origin 2000 using NQS

To compile on the SGI Origin 2000 with the Fortran90 compiler:

```
f90 -o elas3d_mpi -O2 elas3d_mpi.f -lmpi
```

To submit this job, this example using 8 processors can be modified to suit:

```
#QSUB -r Elas3d_8p.JOB
#QSUB -o Elas3d_8p.out
#QSUB -e Elas3d_8p.err
#QSUB -ro
#QSUB -re
#QSUB -lw 2Gb -lM 2Gb
#QSUB -lt 24:00:00 -lT 24:00:00
#QSUB -l mpp_p=8
date
mpirun -np 8 elas3d_mpi
date
```

4 MPI Primer

MPI is a series of subroutines which are used to facilitate parallel processing. These routines allow data transfer between processing nodes and also support a variety of mathematical functions. They can be invoked using the C or FORTRAN languages; however, since the programs are written in FORTRAN90, only the FORTRAN bindings will be discussed. This section will focus on the bare essentials of MPI needed to understand the programs.

All MPI programs have a few things in common. They need to use a MPI include file, they must be initialized, finalized and the communicating processors must belong to the same communicating group. Here is a list of the generic MPI calls the user will encounter as they read the code. Any variables used in these calls will be defined in turn. Italicized font will be used when there are multiple options available.

- MPI_INIT(ierr)
- MPI_FINALIZE(ierr)
- MPI_COMM_SIZE(MPI_*Comm*, nprocs, ierr)
- MPI_COMM_RANK(MPI_*Comm*, myrank, ierr)
- MPI_SEND(X,count,MPI_*Data_Type*,dest,itag,MPI_*Comm*,ierr)
- MPI_RECV(Y,count,MPI_*Data_Type*,source,itag,MPI_*Comm*,status,ierr)
- MPI_IRECV(Y,count,MPI_*Data_Type*,source,itag,MPI_*Comm*,irequest,ierr)
- MPI_WAIT(irequest,status,ierr)
- MPI_BARRIER(MPI_*Comm*,ierr)
- MPI_ALLREDUCE(dA,A,count,MPI_*Data_Type*,MPI_*Op*,MPI_*Comm*,ierr)

4.1 Initialization

The include file always comes before any MPI calls; its location is system dependent. MPI_INIT is the first genuine MPI call and does initialization. MPI_FINALIZE is called at the end when MPI is no longer needed, usually near the end of the program. The integer *ierr* is used as a return code for the subroutine call. The simplest MPI program in FORTRAN is:

```
include 'mpif.h'
integer ierr
call MPI_INIT(ierr)
call MPI_FINALIZE(ierr)
end
```

To do something useful with MPI, make calls to MPI_COMM_SIZE and MPI_COMM_RANK within the communicating group MPI_COMM_WORLD.

```
include 'mpif.h'
...define variables
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
...body of program
call MPI_FINALIZE(ierr)
```

The call to MPI_COMM_SIZE returns the number of total processors that the job is running on and stores that value as an integer in the variable *nprocs*. The call to MPI_COMM_RANK returns the value of the rank, between 0 and $nprocs - 1$, and stores it into the local copy of the variable *myrank*. Here is also an example of the communicating group in action. All processors belong to MPI_COMM_WORLD by default, so the MPI_COMM_SIZE and MPI_COMM_RANK calls happen globally over all processors.

4.2 Sending and Receiving Data

The essence of doing distributed memory parallel processing involves sending and receiving data from one processing node to another. The core vehicles for data transfer are MPI_SEND and MPI_RECV. There are some particulars of which the user must be aware. The correct MPI_*Data_Type*, MPI_*Comm* and itag variables must be set. The programs in the manual use numerous data types: integer*2, integer*4, double precision, double complex and byte. For each one of these, there is a corresponding MPI data type.

Assume processor P sends a 2-D slice of array $pix(ny, ny, k)$ of data type INTEGER*2 to a 2-D array on processor $P+1$ named $Y(nx, ny)$. The number of elements *count* is simply $nx \times ny$. But since they are being sent "byte-wise", one is really sending $2 \times nx \times ny$ elements. Therefore, the necessary pair of matching SEND and RECV calls are:

```
      source = p
      dest = p+1
      count=nx*ny

      if (myrank.eq.source) then
         itag=0
         call MPI_SEND(pix(:,:,k),2*count,MPI_BYTE,dest,itag,
     &                 MPI_COMM_WORLD,ierr)
      end if

      if (myrank.eq.dest) then
         itag=0
         call MPI_RECV(Y,2*count,MPI_BYTE,source,itag,
     &                 MPI_COMM_WORLD,status,ierr)
      end if
```

This MPI_SEND and MPI_RECV calls are technically called a blocking send and a blocking receive, respectively. In other words, the processor waits (or blocks) for the message to complete before it continues with the executable.

The timing of sending and receiving can be crucial at times. One processor may already know that it has to be sending data, but the receiving node may not be ready. In such cases, it is possible to turn a data receive into a two-stage process with calls to MPI_IRECV and MPI_WAIT. MPI_IRECV performs a non-blocking receive using the handle *irequest*. The MPI_WAIT keeps the MPI_IRECV from completing until it gets *irequest* and a legitimate *status*. This method is used in the subroutines **t2b**, **b2t** and their variants. Of course there is a matching MPI_SEND call posted by another processor, but it is not shown.

```
      call MPI_IRECV(Y,2*count,MPI_BYTE,source,itag,
     &              MPI_COMM_WORLD,irequest,ierr)
      call MPI_WAIT(irequest,status,ierr)
```

There are also cases in which one processor may finish its series of calculations before other processors do. Therefore a mechanism for pausing is built into MPI as MPI_BARRIER. No processor will pass this statement until all are ready to pass.

```
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
```

There have been several integer variables used in this section that have never been explicitly defined in the code examples. It is good programming practice to define them at the beginning of the code, so they are listed here for completeness.

```
include 'mpif.h'
integer itag, ierr, irequest
integer status(MPI_STATUS_SIZE)
integer myrank, nprocs
integer source, dest, count
...rest of program
```

4.3 Built-in Mathematical Functions

There are multiple instances in the code of getting a sum of all the elements in one of the large principal arrays. Since the arrays per processor are only slices, summing up their elements would result in *nprocs* sums; the sum of these sums is the global sum of interest. After the operation, the result is sent to all processors. Fortunately, there is a call in MPI which will perform global mathematical procedures and return the result back to root ($myrank = 0$). To do this, MPI_*Data_Type* and which operation to perform must be defined.

Here is an example of using FORTRAN90 and MPI_SUM to perform the global sum of array A. Each processor has a portion of A, so the partial sum on a processor is made by using the FORTRAN90 operation, **SUM** on array A. Then the operation MPI_SUM performs a global summation of a single ($count = 1$) real variable dX from each node and sums it into variable X and all nodes have this information.

```
dX=SUM(A)
count=1
call MPI_ALLREDUCE(dX,X,count,MPI_REAL,MPI_SUM,MPI_COMM_WORLD,ierr)
```

4.4 Summary

This section was a brief summary to using MPI with FORTRAN, listing specifically which MPI statements were used in the parallel codes described in this manual. Further information on MPI can be found on the web at: http://www.mpi-forum.org/

5 Disclaimer

Certain commercial equipment is identified in this paper to specify the experimental procedure. In no case does such identification imply endorsement by the National Institute of Standards and Technology, nor does it indicate that the products are necessarily the best available for the purpose.

6 Program Listings

6.1 Finite Element

6.1.1 ELECFEM3D_MPI.f

```
c  ***********************  elecfem3d_mpi.f   **************************
c
c  This is the new MPI version of the elecfem3d.f code from
c  Section 9.3.1 of NISTIR 6269.
c
c  The main differences with this code compared to the serial
c  version are:
c
c  1. Removal of ib array.
c  2. Change of dimensionality on pix from pix(m) to pix(i,j,k)
c  Maximum value of m = nx*ny*nz (nx,ny,nz are the array dims).
c  3.  All important arrays (pix,vox,gb,b,u,h,Ah) are dynamically allocated.
c
c  IN THIS VERSION:
c
c  The USER needs the following input:
c  (Search for occurences of "USER" in the code).
c
c 1. A 3-D pixel value data file with input & output names.
c 2. The values of the 3 dimensions: (nx,ny,nz)
c 3. The number of phases in the mixture: nphase
c 4. A convergence value: gtest
c 5. Applied electric field: ex,ey,ez
c 6. Values for DEMBX_MPI and how long it will run: kmax & ldemb
c
c 7. Flag for printing timing info for all data
c    passing MPI routines ( FEMAT_MPI, ENERGY_MPI, DEMBX_MPI)
c    from MAIN is called: pflag
c    pflag Values = 0,1 0=no timing info; 1=print timing info
c
c    pflag is a common value.
c
c    Timing info for the RELAXATION loop is not
c    influenced by the pflag and will always be printed.
c
c    User may edit the code to supress the printing.
c
c 8. Timing info stored in arrays namex X_time(i)
c    Where X=n,f,e ie.
c    n_time is in MAIN
c    f_time is in FEMAT_MPI
c    e_time is in ENERGY_MPI
c
c NB:  One also needs to insure that the values for
```

```
c      phasemod(i,j) are initialized correctly in
c      SUBROUTINE phasemod_init.
c
c
c END of NEW comments.
c
c BEGIN ORIGINAL comments.
c
c  BACKGROUND
c
c  This program solves Laplace's equation in a random conducting
c  material using the finite element method.  Each pixel in the 3-D digital
c  image is a cubic tri-linear finite element,  having its own conductivity.
c  Periodic boundary conditions are maintained.  In the comments below,
c  (USER) means that this is a section of code that the user might
c  have to change for his particular problem.  Therefore the user is
c  encouraged to search for this string.
c
c  PROBLEM AND VARIABLE DEFINITION
c
c  The problem being solved is the minimization of the energy
c  1/2 uAu + b u + C, where A is the Hessian matrix composed of the
c  stiffness matrices (dk) for each pixel/element, b is a constant vector
c  and C is a constant that are determined by the applied field and
c  the periodic boundary conditions, and u is a vector of all the voltages.
c  The method used is the conjugate gradient relaxation algorithm.
c  Other variables are:  gb is the gradient = Au+b, h and Ah are
c  auxiliary variables used in the conjugate gradient algorithm (in dembx),
c  dk(n,i,j) is the stiffness matrix of the n'th phase, sigma(n,i,j) is
c  the conductivity tensor of the n'th phase, pix is a vector that gives
c  the phase label of each pixel, ib is a matrix that gives the labels of
c  the 27 (counting itself) neighbors of a given node, prob is the volume
c  fractions of the various phases, and currx, curry, currz are the
c  volume averaged total currents in the x, y, and z directions.
c
c DIMENSIONS
c
c  The vectors u,gb,b,h, and Ah are dimensioned to be the system size,
c  ns=nx*ny*nz, where the digital image of the microstructure considered
c  is a rectangular parallelipiped ( nx x ny x nz) in size.
c  The arrays pix and ib are also dimensioned to the system size.
c  The array ib has 27 components, for the 27 neighbors of a node.
c  Note that the program is set up at present to have at most 100
c  different phases.  This can easily be changed, simply by changing
c  the dimensions of dk, prob, and sigma. Nphase gives the number of
c  phases being considered.
c  All arrays are passed between subroutines using simple common statements.
c
c  STRONGLY SUGGESTED:  READ THE MANUAL BEFORE USING PROGRAM!!
```

```
      implicit none
      include 'mpif.h'
c
c  (USER) Change the nx,ny,nz dimensions at the beginning.
c  All important arrays are dynamically allocated.
c
      integer*2, allocatable :: dat(:,:,:), datn(:,:,:)
      integer*2, allocatable :: pix(:,:,:), pixn(:,:,:)
      integer*2, allocatable :: vox(:,:,:)

      integer, allocatable :: d1s(:),d2s(:)

      double precision, allocatable :: b(:,:,:,:)
      double precision, allocatable :: gb(:,:,:,:)
      double precision, allocatable :: u(:,:,:,:)
      double precision, allocatable :: h(:,:,:,:)

      double precision, allocatable ::  sigma(:,:,:), prob(:)
      double precision, allocatable ::  dk(:,:,:)

      double precision dgg,gg,utot,gtest,C
      double precision ex,ey,ez
      double precision x,y,z,saves

      double precision cuxxp,cuyyp,cuzzp
      double precision currx,curry,currz

      integer d1,d2,ns,sxip,kkk,mxy
      integer i,j,k,nx,ny,nz,nxy,nphase
      integer count,rem
      integer sz,sized
      integer npoints,micro,m
      integer kmax,ldemb,ltot,lstep
      integer pflag

      integer myrank,ierr,nprocs,irank
      integer status(MPI_STATUS_SIZE)

      double precision starttime,endtime, start_npoint, end_npoint
      double precision kkk_start,kkk_end
      double precision elapsed_time,stress_loop
      double precision n_time(24)

      common/list1/pflag,nphase
      common/list2/ex,ey,ez
      common/list3/currx,curry,currz
      common/list4/cuxxp,cuyyp,cuzzp
```

```
      call MPI_INIT(ierr)

      starttime = MPI_Wtime(ierr)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      if (myrank.eq.0) then
      write(*,*) "There are ",nprocs," processors running this job."
      end if

c
c   USER: Change nx,ny,nz,nphase values to match your data.
c
      nx=100
      ny=100
      nz=100
      nphase=2

      nxy=nx*ny
      ns=nx*ny*nz
      sz=nz/nprocs
      mxy= 3*nx*ny

      gtest=1.d-10*ns

c     pflag=0 for no timing info printed.
c     pflag=1 for timing info printed.
      pflag = 0
c
c End this USER section.
c

      utot =0.0d0

      allocate( sigma(nphase,3,3) )
      allocate( dk(nphase,8,8) )
      allocate( prob(nphase) )

c  (USER) sigma(nphase,3,3) is the electrical conductivity tensor of each phase
c  The user can make the value of sigma to be different for each
c  phase of the microstructure if so desired.
c
c Only diagonal elements need values
c
      sigma=0.0d0

      sigma(1,1,1)=1.0d0
      sigma(1,2,2)=1.0d0
```

```
      sigma(1,3,3)=1.0d0

      sigma(2,1,1)=200.0d0
      sigma(2,2,2)=200.0d0
      sigma(2,3,3)=200.0d0

c  (USER) Set applied electric field.
          ex=1.0d0
          ey=1.0d0
          ez=1.0d0
c
c Calculate d1s(n) & d2s(n); These hold the d1 and d2
c values for processor n.
c
      if (myrank.eq.0) then

      allocate (d1s(0:nprocs-1))
      allocate (d2s(0:nprocs-1))

        do irank=0,nprocs-1
          d1s(irank)=irank*sz+1
          d2s(irank)=(irank+1)*sz
        end do

        rem = nz - nprocs*sz

        if (rem.ne.0) then
           do j=1,rem
             irank=nprocs-rem+j-1
             d1s(irank)=d1s(irank)+ j-1
             d2s(irank)=d2s(irank)+ j
           end do
        end if

c Send all d1s(i) and d2s(i) from ROOT
c to NODE i & store into d1 & d2

        do i=0,nprocs-1
      call MPI_SEND(d1s(i),1,MPI_INTEGER,i,0,MPI_COMM_WORLD,ierr)
      call MPI_SEND(d2s(i),1,MPI_INTEGER,i,1,MPI_COMM_WORLD,ierr)
        end do

      end if

      call MPI_RECV(d1,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,status,ierr)
      call MPI_RECV(d2,1,MPI_INTEGER,0,1,MPI_COMM_WORLD,status,ierr)
      write(*,*) "Rank#",myrank,"d1= ",d1," d2= ",d2

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
```

```
c
c Allocate other arrays which need d1&d2 values.
c
      allocate (gb(nx,ny,d1-1:d2+1))
      gb=0.0d0
      allocate(b(nx,ny,d1-1:d2+1))
      b = 0.0d0

      allocate (u(nx,ny,d1-1:d2+1))
      allocate (h(nx,ny,d1-1:d2+1))

c
c Want the ability to calculate on a series
c of input files based on a value & some if statements.
c
c Compute the average stress and strain in each microstructure.
c (USER) npoints is the number of microstructures to use.

      npoints=1

c (USER) Unit 9 is the microstructure input file,
c  unit 7 is the results output file.

      n_time(1) = MPI_Wtime(ierr)

      do micro=1,npoints
c
c Allocate pix, so root can read it.
c
      if (myrank.eq.0) then
         allocate (pix(nx,ny,nz))
      end if

      start_npoint=MPI_Wtime(ierr)
      n_time(2) = MPI_Wtime(ierr)

      if (myrank.eq.0) then

c Get pix from input file (unit=9).
c
c  (USER) Unit 9 is the microstructure input file, unit 7 is
c  the results output file.

        open(9,file='test20.dat')
        open(7,file='t200_elecmpi.out')

        write(*,*) "MICRO = ", micro
        write(7,*) "MICRO = ", micro
```

```
c
c Finally... read in pix
c
      write(*,*) "call dpixel"
      call dpixel(nx,ny,nz,ns,pix)
      write(*,*) "back from dpixel"

c ns=total number of sites
      write(7,9010) nx,ny,nz,ns,nprocs
9010  format('nx= ',i4,' ny= ',i4,' nz= ',i4,' ns= 'i8,' nprocs= ',i4)

      end if

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c Now that the nodes are set up correctly,
c one can pass the data from the root node (myrank=0)
c to all the rest.

      allocate(dat(nx,ny,d1:d2))
      sized = SIZE(dat)
      dat=0

      n_time(3)=MPI_Wtime(ierr)

      if (nprocs.eq.1) then
         dat=pix
         write(*,*) "dat=pix"
      end if

      if (nprocs.gt.1) then

         if (myrank.eq.0) then
             dat(:,:,d1:d2)=pix(:,:,d1:d2)
             do i=1,nprocs-1
               allocate (pixn(nx,ny,d1s(i):d2s(i)))
               pixn = pix(:,:,d1s(i):d2s(i))
               sxip = SIZE(pixn)
               call MPI_SEND(pixn,2*sxip,MPI_BYTE,
     &                    i,7,MPI_COMM_WORLD,status,ierr)
               deallocate(pixn)
            end do
         else
            allocate(datn(nx,ny,d1:d2))
            call MPI_RECV(datn,2*sized,MPI_BYTE,0,7
     &                   ,MPI_COMM_WORLD,status,ierr)

            dat(:,:,d1:d2) = datn
```

```
           deallocate(datn)
        end if
      end if

      n_time(4)=MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, " time to get original data= ",
     &           n_time(4)-n_time(3)
      endif

      allocate(vox(nx,ny,d1-1:d2+1))

c
c Make the copy
c
      do k=d1,d2
         vox(:,:,k) = dat(:,:,k)
      end do
      deallocate(dat)

c
c Call z_ghost_int to make Z ghost layers of INTEGER*2 values (aka vox).
c
      call z_ghost_int(vox,nx,ny,d1,d2)

77    format(3(a5,i5,2x))
78    format(a,3(i5,2x))

        if (myrank.eq.0) then
           call dassig(nx,ny,nz,prob,pix)

   do i=1,nphase
              write(7,*) 'Volume fraction of phase ',i,' = ',prob(i)
           end do

           call flush(7)
           deallocate(pix)

         end if

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

        if (myrank.eq.0) then

c  write out the phase electrical conductivity tensors
      do 11 i=1,nphase
      write(7,*) 'Phase ',i,' conductivity tensor is:'
      write(7,*) sigma(i,1,1),sigma(i,1,2),sigma(i,1,3)
```

```
      write(7,*) sigma(i,2,1),sigma(i,2,2),sigma(i,2,3)
      write(7,*) sigma(i,3,1),sigma(i,3,2),sigma(i,3,3)
11    continue

      write(7,*) 'Applied field components:'
      write(7,*) 'ex = ',ex,' ey = ',ey,' ez = ',ez

        call flush(7)
        end if

c  Set up the finite element "stiffness" matrices and the Constant and
c  vector required for the energy

      count=0

      n_time(9)=MPI_Wtime(ierr)
      call femat_mpi(nx,ny,nz,d1,d2,vox,sigma,b,dk,C)
      n_time(10)=MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank," femat_mpi time=",n_time(10)-n_time(9)
      endif

      do k=d1,d2
      do j=1,ny
      do i=1,nx
              x=dfloat(i-1)
              y=dfloat(j-1)
              z=dfloat(k-1)
              u(i,j,k)=-x*ex-y*ey-z*ez
      end do; end do; end do

c
c Call z_ghost_dp to make Z ghost layers of DOUBLE PRECISION values (aka u).
c
      call z_ghost_dp(u,nx,ny,d1,d2)

c  RELAXATION LOOP
c  (USER) kmax is the maximum number of times dembx_mpi will be called, with
c  ldemb conjugate gradient steps performed during each call.  The total
c  number of conjugate gradient steps allowed for a given elastic
c  computation is kmax*ldemb.

        kmax=100
        ldemb=100
        ltot=0
c  Call energy to get initial energy and initial gradient

      n_time(15)=MPI_Wtime(ierr)
```

```
      call energy_mpi(u,dk,b,C,nx,ny,nz,d1,d2,gb,utot,vox)

      n_time(16)=MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank,"Initial energy_mpi time=",
     &                 n_time(16)-n_time(15)
      endif

c  gg is the norm squared of the gradient (gg=gb*gb)
      dgg= 0.0d0
      gg = 0.0d0
      dgg = SUM(gb(:,:,d1:d2)*gb(:,:,d1:d2))
      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      n_time(17)=MPI_Wtime(ierr)

      if (myrank.eq.0) then
         write(*,*) " Initial Energy = ",utot, "  gg = ",gg
         write(7,*) " Initial Energy = ",utot, "  gg = ",gg
         call flush(7)
      end if

      elapsed_time=0.0d0

      n_time(18)=MPI_Wtime(ierr)

      kkk=0
      kkk=kkk+1
      do kkk=1,kmax

         kkk_start = MPI_Wtime(ierr)

c  call dembx_mpi to go into the conjugate gradient solver

        call dembx_mpi(nx,ny,nz,d1,d2,Lstep,gb,u,vox,h,
     &                     gg,dk,gtest,ldemb,kkk)

        ltot=ltot+Lstep
        call energy_mpi(u,dk,b,C,nx,ny,nz,d1,d2,gb,utot,vox)

        kkk_end = MPI_Wtime(ierr)
        elapsed_time=elapsed_time+(kkk_end-kkk_start)

      if (myrank.eq.0) then
         write(7,*) "Energy = ",utot," gg = ",gg
         write(7,*) "Number of conjugate steps  = ",ltot
```

```
          write(7,*) "Root took ",kkk_end-kkk_start," s for ",
     &   ltot, "conjugate steps."
          write(7,*) "Elapsed time=",elapsed_time," s for ",
     &   ltot, "conjugate steps."

          write(*,*) "Energy = ",utot," gg = ",gg
          write(*,*) "Number of conjugate steps  = ",ltot
          write(*,*) "Root took ",kkk_end-kkk_start," s for ",
     &   ltot, "conjugate steps."
          write(*,*) "Elapsed time= ",elapsed_time," s for ",
     &   ltot, "conjugate steps."
          call flush(7)
       end if

c Call energy_mpi to compute energy after dembx_mpi call. If gg < gtest,
c this will be the final energy. If gg is still larger than gtest,
c then this will give an intermediate energy with which to check how the
c relaxation process is coming along.

c  If relaxation process is finished, jump out of loop

       if(gg.le.gtest) goto 444

c  If relaxation process will continue, compute and output stresses
c  and strains as an additional aid to judge how the
c  relaxation procedure is progressing.

       call current_mpi(nx,ny,nz,ns,sigma,vox,u,d1,d2)

       if (myrank.eq.0) then
c Output intermediate currents
          write(7,*)
          write(7,*) ' Current in x direction = ',cuxxp
          write(7,*) ' Current in y direction = ',cuyyp
          write(7,*) ' Current in z direction = ',cuzzp
          call flush(7)
       end if

       end do

444     call current_mpi(nx,ny,nz,ns,sigma,vox,u,d1,d2)

       if (myrank.eq.0) then
c Output final currents
          write(7,*)
          write(7,*) ' Current in x direction = ',cuxxp
          write(7,*) ' Current in y direction = ',cuyyp
          write(7,*) ' Current in z direction = ',cuzzp
          write(7,*)
```

```
         call flush(7)

c Output final currents
         write(*,*)
         write(*,*) ' Current in x direction = ',cuxxp
         write(*,*) ' Current in y direction = ',cuyyp
         write(*,*) ' Current in z direction = ',cuzzp

         close(unit=9)
         close(unit=7)

      end if

      deallocate(vox)

      end do

c
c Do another calculation using loop var: npoints
c

      deallocate(u)
      deallocate(b)
      deallocate(gb)
      deallocate(h)

      n_time(24) = MPI_Wtime(ierr)

      CALL MPI_FINALIZE(ierr)

      end
c
c*********************************************************
c
      subroutine femat_mpi(nx,ny,nz,d1,d2,vox,sigma,b,dk,C)

      implicit none

      include 'mpif.h'

      integer i,ierr,nx,j,ny,nz
      integer d1,d2,myrank,nprocs,mxy
      integer ipx,ipy,ipz
      integer nxy,k,nm,ijk,mm,ii,jj,kk,ll
      integer i8,dn,m,m8
      integer pflag,nphase

      integer status(MPI_STATUS_SIZE)
```

```
      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision sum_num,cterm,cpos,cneg
      double precision c,c3,x,y,z
      double precision f_time(24)

      double precision dk(nphase,8,8), sigma(nphase,3,3)
      double precision dndx(8),dndy(8),dndz(8)
      double precision g(3,3,3)
      double precision es(3,8),xn(8)
      double precision b(nx,ny,d1-1:d2+1)
      double precision, allocatable :: ab(:,:), ba(:,:)
      double precision ex,ey,ez

      common/list1/pflag,nphase
      common/list2/ex,ey,ez

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      f_time(1) = MPI_Wtime(ierr)
      nxy=nx*ny
      mxy=3*nxy

      allocate (ab(nx,ny))
      allocate (ba(nx,ny))

c  initialize stiffness matrices

      dk=0.0d0

c  set up Simpson's integration rule weight vector
      do k=1,3
      do j=1,3
      do i=1,3
      nm=0
      if(i.eq.2) nm=nm+1
      if(j.eq.2) nm=nm+1
      if(k.eq.2) nm=nm+1
      g(i,j,k)=4.0d0**nm
      end do
      end do
      end do

c  loop over the nphase kinds of pixels and Simpson's rule quadrature
c  points in order to compute the stiffness matrices.  Stiffness matrices
c  of trilinear finite elements are quadratic in x, y, and z, so that
c  Simpson's rule quadrature gives exact results.
```

```
      do ijk=1,nphase
      do k=1,3
      do j=1,3
      do i=1,3
      x=dfloat(i-1)/2.0d0
      y=dfloat(j-1)/2.0d0
      z=dfloat(k-1)/2.0d0
c  dndx means the negative derivative, with respect to x, of the shape
c  matrix N (see manual, Sec. 2.2), dndy, and dndz are similar.
      dndx(1)=-(1.0d0-y)*(1.0d0-z)
      dndx(2)=(1.0d0-y)*(1.0d0-z)
      dndx(3)=y*(1.0d0-z)
      dndx(4)=-y*(1.0d0-z)
      dndx(5)=-(1.0d0-y)*z
      dndx(6)=(1.0d0-y)*z
      dndx(7)=y*z
      dndx(8)=-y*z
      dndy(1)=-(1.0d0-x)*(1.0d0-z)
      dndy(2)=-x*(1.0d0-z)
      dndy(3)=x*(1.0d0-z)
      dndy(4)=(1.0d0-x)*(1.0d0-z)
      dndy(5)=-(1.0d0-x)*z
      dndy(6)=-x*z
      dndy(7)=x*z
      dndy(8)=(1.0d0-x)*z
      dndz(1)=-(1.0d0-x)*(1.0d0-y)
      dndz(2)=-x*(1.0d0-y)
      dndz(3)=-x*y
      dndz(4)=-(1.0d0-x)*y
      dndz(5)=(1.0d0-x)*(1.0d0-y)
      dndz(6)=x*(1.0d0-y)
      dndz(7)=x*y
      dndz(8)=(1.0d0-x)*y

c  now build electric field matrix

      es=0.0d0

      es(1,:)=dndx
      es(2,:)=dndy
      es(3,:)=dndz

c  Matrix multiply to determine value at (x,y,z), multiply by
c  proper weight, and sum_num into dk, the stiffness matrix

      f_time(2) = MPI_Wtime(ierr)

      do ii=1,8
      do jj=1,8
```

```
c  Define sum over strain matrices and elastic moduli matrix for
c  stiffness matrix
      sum_num=0.0d0
      do kk=1,3
      do ll=1,3
      sum_num=sum_num+es(kk,ii)*sigma(ijk,kk,ll)*es(ll,jj)
      end do; end do
      dk(ijk,ii,jj)=dk(ijk,ii,jj)+g(i,j,k)*sum_num/216.

      end do; end do
      end do; end do; end do; end do

      f_time(3) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "time to calculate dk = ",f_time(3)-f_time(2)
      endif

c  Initialize b and C

      b=0.0d0
      C=0.0d0
      c3=0.0d0

999 format(4(i4,1x,),3(f9.6,1x))

c
c  x=nx face
c
      i=nx
      do i8=1,8
         xn(i8) = 0.0d0
         if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
            xn(i8)=-ex*nx
         end if
      end do

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      dn=d2
      if (dn.eq.nz) then
          dn = nz-1
      end if

      cpos=0.0d0; cneg=0.0d0
      cterm=0.0d0

      do k=d1,dn
      do j=1,ny-1
```

```
      m=nxy*(k-1)+j*nx

      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0

      do m8=1,8

      cterm =0.5d0*xn(m8)*dk(vox(ii,jj,kk),m8,mm)*xn(mm)

      if (cterm.ge.0.0d0) then
      cpos = cpos + cterm
      else
      cneg = cneg + cterm
      end if

      sum_num=sum_num+xn(m8)*dk(vox(ii,jj,kk),m8,mm)

      end do

c Assign b(ipx,ipy,ipz)  = b(ipx,ipy,ipz) + sum_num

      b(ipx,ipy,ipz) =  b(ipx,ipy,ipz) + sum_num

      end do
      end do; end do

c
c  y=ny face
c
      j=ny
      do i8=1,8
         xn(i8)=0.0d0
         if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
           xn(i8)=-ey*ny
         end if
      end do

      do i=1,nx-1
      do k=d1,dn
      m=nxy*(k-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
```

```
      do m8=1,8

      sum_num=sum_num+xn(m8)*dk(vox(ii,jj,kk),m8,mm)

      cterm=0.5d0*xn(m8)*dk(vox(ii,jj,kk),m8,mm)*xn(mm)

      if (cterm.ge.0.0d0) then
         cpos = cpos + cterm
      else
         cneg = cneg + cterm
      end if

      end do

      b(ipx,ipy,ipz) =  b(ipx,ipy,ipz) + sum_num

      end do
      end do; end do


c
c    Zface calcs
c
c    Only the last node does these series of calculations since
c    it contains all the necessary data therefore no data transfer
c    occurs.
c
      if (myrank.eq.nprocs-1) then

      k = nz
      do i8=1,8
      xn(i8)=0.0d0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      xn(i8)=-ez*nz
      end if
      end do

      do i=1,nx-1
      do j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0

      do m8=1,8
      sum_num=sum_num+xn(m8)*dk(vox(ii,jj,kk),m8,mm)
      cterm=0.5d0*xn(m8)*dk(vox(ii,jj,kk),m8,mm)*xn(mm)
```

```
      if (cterm.ge.0.0d0) then
      cpos = cpos + cterm
      else
      cneg = cneg + cterm
      end if

      end do
      b(ipx,ipy,ipz) =  b(ipx,ipy,ipz) + sum_num
      end do
      end do; end do

      end if

c
c  x=nx y=ny edge
c

      i=nx
      y=ny

      do i8=1,8
      xn(i8)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      xn(i8)=-ex*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      xn(i8)=-ey*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      xn(i8)=-ey*ny-ex*nx
      end if
      end do

      dn=d2
      if (dn.eq.nz) then
          dn = nz-1
      end if

      do k=d1,dn
      m=nxy*k
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0d0
      do m8=1,8
      sum_num=sum_num+xn(m8)*dk(vox(ii,jj,kk),m8,mm)
```

```
      cterm=0.5d0*xn(m8)*dk(vox(ii,jj,kk),m8,mm)*xn(mm)

      if (cterm.ge.0.0d0) then
      cpos = cpos + cterm
      else
      cneg = cneg + cterm
      end if

      end do
      b(ipx,ipy,ipz) =  b(ipx,ipy,ipz) + sum_num

      end do
      end do

c
c  x=nx z=nz edge
c

      if (myrank.eq.nprocs-1) then

      i=nx
      k=nz

      do i8=1,8
      xn(i8)=0.0d0

      if(i8.eq.2.or.i8.eq.3) then
      xn(i8)=-ex*nx
      end if

      if(i8.eq.5.or.i8.eq.8) then
      xn(i8)=-ez*nz
      end if

      if(i8.eq.6.or.i8.eq.7) then
      xn(i8)=-ez*nz-ex*nx
      end if

      end do

      do j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      call m2ijk(m,ii,jj,kk,nx,ny,nx)

      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0

      do m8=1,8
```

```
      sum_num=sum_num+xn(m8)*dk(vox(ii,jj,kk),m8,mm)

      cterm=0.5d0*xn(m8)*dk(vox(ii,jj,kk),m8,mm)*xn(mm)

      if (cterm.ge.0.0d0) then
      cpos = cpos + cterm
      else
      cneg = cneg + cterm
      end if

      end do

      b(ipx,ipy,ipz) =  b(ipx,ipy,ipz) + sum_num

      end do
      end do
c
c  y=ny z=nz edge
c
      j=ny
      k=nz

      do i8=1,8
      xn(i8)=0.0d0

      if(i8.eq.5.or.i8.eq.6) then
      xn(i8)=-ez*nz
      end if

      if(i8.eq.3.or.i8.eq.4) then
      xn(i8)=-ey*ny
      end if

      if(i8.eq.7.or.i8.eq.8) then
      xn(i8)=-ey*ny-ez*nz
      end if
      end do

      do i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nx)

      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0

      do m8=1,8
```

```
          sum_num=sum_num+xn(m8)*dk(vox(ii,jj,kk),m8,mm)
          cterm=0.5d0*xn(m8)*dk(vox(ii,jj,kk),m8,mm)*xn(mm)
          if (cterm.ge.0.0d0) then
            cpos = cpos + cterm
          else
            cneg = cneg + cterm
          end if

      end do

      b(ipx,ipy,ipz) =  b(ipx,ipy,ipz) + sum_num
      end do
      end do

c
c  x=nx y=ny z=nz corner
c

      i=nx
      j=ny
      k=nz

      do i8=1,8
      xn(i8)=0.0d0

      if(i8.eq.2) then
        xn(i8)=-ex*nx
      end if

      if(i8.eq.4) then
        xn(i8)=-ey*ny
      end if

      if(i8.eq.5) then
        xn(i8)=-ez*nz
      end if

      if(i8.eq.8) then
        xn(i8)=-ey*ny-ez*nz
      end if

      if(i8.eq.6) then
        xn(i8)=-ex*nx-ez*nz
      end if

      if(i8.eq.3) then
        xn(i8)=-ex*nx-ey*ny
      end if
```

```
      if(i8.eq.7) then
        xn(i8)=-ex*nx-ey*ny-ez*nz
      end if

      end do

      m=nx*ny*nz
      call m2ijk(m,ii,jj,kk,nx,ny,nx)

      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0

      do m8=1,8
      sum_num=sum_num+xn(m8)*dk(vox(ii,jj,kk),m8,mm)
      cterm=0.5d0*xn(m8)*dk(vox(ii,jj,kk),m8,mm)*xn(mm)

      if (cterm.ge.0.0d0) then
      cpos = cpos + cterm
      else
      cneg = cneg + cterm
      end if

      end do
      b(ipx,ipy,ipz) =  b(ipx,ipy,ipz) + sum_num

      end do

c
c End if for (myrank.eq.nprocs-1)
c
      end if

      c3 = cpos + cneg
      CALL MPI_ALLREDUCE(c3,C,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     &        MPI_COMM_WORLD,ierr)

       if (myrank.eq.0) then
         write(*,*) "Final C = ", C
       end if

      f_time(4) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*)myrank,"Etime to calculate C & b= ",f_time(4)-f_time(3)
      end if

      if (nprocs.gt.1) then
c
```

```
c RECV a new slice per node.
c
      ab = 0.0d0
      ba = b(:,:,d2+1)

      f_time(5) = MPI_Wtime(ierr)
      call t2b_dp(ab,ba,nx,ny)
      f_time(6) = MPI_Wtime(ierr)
      b(:,:,d1) = b(:,:,d1) + ab

      if (pflag.eq.1) then
      write(*,*) myrank, " B upddate: t2b time= ",f_time(6)-f_time(5)
      end if

c
c botp = d1-1
c
      ab = 0.0
      ba = b(:,:,d1-1)

      f_time(7) = MPI_Wtime(ierr)
      call b2t_dp(ab,ba,nx,ny)
      f_time(8) = MPI_Wtime(ierr)
      b(:,:,d2) = b(:,:,d2) + ab

      if (pflag.eq.1) then
      write(*,*)  myrank, " B upddate: b2t time= ",f_time(8)-f_time(7)
      end if
c
c Update ghost layers
c
c RECV a new slice per node.
c
      ab = b(:,:,d1)
      ba = b(:,:,d2)

      f_time(9) = MPI_Wtime(ierr)
      call t2b_dp(ab,ba,nx,ny)
      f_time(10) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "B ghost upddate:t2b time= ",
     &                   f_time(10)-f_time(9)
      end if

      b(:,:,d1-1) = ab

      ab = b(:,:,d1)
      ba = b(:,:,d2)
```

```
      f_time(11) = MPI_Wtime(ierr)
      call b2t_dp(ab,ba,nx,ny)
      f_time(12) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "B ghost upddate:b2t time= ",
     &                   f_time(12)-f_time(11)
      end if

      b(:,:,d2+1) = ba

      else
c
c nprocs=1
c
      b(:,:,d1) = b(:,:,d1) + b(:,:,d2+1)
      b(:,:,d2) = b(:,:,d2) + b(:,:,d1-1)
      b(:,:,d2+1) = b(:,:,d1)
      b(:,:,d1-1) = b(:,:,d2)

      end if

      deallocate(ab)
      deallocate(ba)

      f_time(13) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "Femat_mpi elapsed time= ",
     &                   f_time(13)-f_time(1)
      end if

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c**********************************************************
c
      subroutine energy_mpi(u,dk,b,C,nx,ny,nz,d1,d2,gb,utot,vox)
      implicit none

      include 'mpif.h'

      integer nx,ny,nz,d1,d2,myrank,nprocs,ierr
      integer m3,ik,ij,ii
      integer pflag,nphase
```

```
      double precision  u(nx,ny,d1-1:d2+1)
      double precision  b(nx,ny,d1-1:d2+1)
      double precision gb(nx,ny,d1-1:d2+1)
      integer*2 vox(nx,ny,d1-1:d2+1)
      double precision e_time(24)

      double precision c,utot
      double precision dk(nphase,8,8)

      double precision dutot
      double precision ex,ey,ez

      common/list1/pflag,nphase
      common/list2/ex,ey,ez

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      e_time(1) = MPI_Wtime(ierr)

      gb = 0.0d0

c
c After this call, gb is calculated and data slabs
c are updated and passed.

      call gbah(gb,u,dk,vox,nx,ny,nz,d1,d2)

c
c Now do the rest of the gb calculations that appear
c in original "energy" subroutine.
c
c utot will be a per processor value.
c Do an MPI_ALLREDUCE on dutot
c so each node will have the current updated version.
c

      dutot=0.0d0

      do ik=d1,d2
      do ij=1,ny
      do ii=1,nx

      dutot=dutot+0.5d0*u(ii,ij,ik)*gb(ii,ij,ik)+
     &                b(ii,ij,ik)*u(ii,ij,ik)

      end do; end do; end do

      call MPI_ALLREDUCE(dutot,utot,1,MPI_DOUBLE_PRECISION,
```

```
     &                MPI_SUM,MPI_COMM_WORLD,ierr)


      utot = utot + C

c easier to add C here than before the above MPI call.

      gb = gb + b

      return
      end
c
c***********************************************************
c
      subroutine dembx_mpi(nx,ny,nz,d1,d2,Lstep,gb,u,vox,h,
     &                     gg,dk,gtest,ldemb,kkk)

      implicit none

      include 'mpif.h'

      integer nx,ny,nz,d1,d2,ldemb,kkk,ijk
      integer Lstep,myrank,nprocs,ierr
      integer pflag,nphase

      double precision dgg,gg,gglast,lambda,hAh2,hAh,gamma,gtest

      double precision  u(nx,ny,d1-1:d2+1)
      double precision gb(nx,ny,d1-1:d2+1)
      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision dk(nphase,8,8)

      double precision Ah(nx,ny,d1-1:d2+1)
      double precision  h(nx,ny,d1-1:d2+1)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      if(kkk.eq.1) then
        h=gb
      end if

c  Lstep counts the number of conjugate gradient steps taken in
c  each call to dembx

      Lstep=0
```

```
      do ijk=1,ldemb
      Lstep=Lstep+1

      Ah=0.0d0

      call gbah(Ah,h,dk,vox,nx,ny,nz,d1,d2)

      hAh = 0.0d0
      hAh2= 0.0d0

      hAh2 = SUM(h(:,:,d1:d2)*Ah(:,:,d1:d2))

      call MPI_ALLREDUCE(hAh2,hAh,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     &  MPI_COMM_WORLD, ierr)

      lambda=gg/hAh
      u=u-lambda*h
      gb=gb-lambda*Ah
      gglast=gg
      gg=0.0d0

      dgg = SUM(gb(:,:,d1:d2)*gb(:,:,d1:d2))
      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      if (gg.lt.gtest)  goto 1000

      gamma = gg/gglast
      h = gb + gamma*h

      end do

1000  continue

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c***********************************************************
c
      subroutine current_mpi(nx,ny,nz,ns,sigma,vox,u,d1,d2)

      implicit none
      include 'mpif.h'

      integer nx,ny,nz,ns,d1,d2,nxy
      integer i,j,k,m,n,nn
```

```
      integer ifxa,ifya,pflag,nphase

      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision af(3,8)
      double precision u(nx,ny,d1-1:d2+1), uu(8)
      double precision sigma(nphase,3,3)

      double precision cur1,cur2,cur3,ex,ey,ez
      double precision currx,curry,currz
      double precision cuxxp,cuyyp,cuzzp

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      common/list1/pflag,nphase
      common/list2/ex,ey,ez
      common/list3/currx,curry,currz
      common/list4/cuxxp,cuyyp,cuzzp

      nxy=nx*ny

c  af is the average field matrix, average field in a pixel is af*u(pixel).
c  The matrix af relates the nodal voltages to the average field in the pixel.

c Set up single element average field matrix

      af(1,1)=0.25d0
      af(1,2)=-0.25d0
      af(1,3)=-0.25d0
      af(1,4)=0.25d0
      af(1,5)=0.25d0
      af(1,6)=-0.25d0
      af(1,7)=-0.25d0
      af(1,8)=0.25d0
      af(2,1)=0.25d0
      af(2,2)=0.25d0
      af(2,3)=-0.25d0
      af(2,4)=-0.25d0
      af(2,5)=0.25d0
      af(2,6)=0.25d0
      af(2,7)=-0.25d0
      af(2,8)=-0.25d0
      af(3,1)=0.25d0
      af(3,2)=0.25d0
      af(3,3)=0.25d0
      af(3,4)=0.25d0
      af(3,5)=-0.25d0
```

```
      af(3,6)=-0.25d0
      af(3,7)=-0.25d0
      af(3,8)=-0.25d0

c  now compute current in each pixel
      currx=0.0d0
      curry=0.0d0
      currz=0.0d0

c  compute average field in each pixel

      do 470 k=d1,d2
      do 470 j=1,ny
      do 470 i=1,nx
      m=(k-1)*nxy+(j-1)*nx+i

      if ((i+1).GT.nx) then
         ifxa = 1
      else
         ifxa = i+1
      end if

      if ((j+1).GT.ny) then
         ifya = 1
      else
         ifya = j+1
      end if

c  load in elements of 8-vector using pd. bd. conds.

       uu(1)= u(i,j,k)
       uu(2)= u(ifxa,j,k)
       uu(3)= u(ifxa,ifya,k)
       uu(4)= u(i,ifya,k)
       uu(5)= u(i,j,k+1)
       uu(6)= u(ifxa,j,k+1)
       uu(7)= u(ifxa,ifya,k+1)
       uu(8)= u(i,ifya,k+1)

c  Correct for periodic boundary conditions, some voltages are wrong
c  for a pixel on a periodic boundary. Since they come from an opposite
c  face, need to put in applied fields to correct them.
      if(i.eq.nx) then
      uu(2)=uu(2)-ex*nx
      uu(3)=uu(3)-ex*nx
      uu(6)=uu(6)-ex*nx
      uu(7)=uu(7)-ex*nx
      end if
      if(j.eq.ny) then
```

```
      uu(3)=uu(3)-ey*ny
      uu(4)=uu(4)-ey*ny
      uu(7)=uu(7)-ey*ny
      uu(8)=uu(8)-ey*ny
      end if
      if(k.eq.nz) then
      uu(5)=uu(5)-ez*nz
      uu(6)=uu(6)-ez*nz
      uu(7)=uu(7)-ez*nz
      uu(8)=uu(8)-ez*nz
      end if
c  cur1, cur2, cur3 are the local currents averaged over the pixel
      cur1=0.0d0
      cur2=0.0d0
      cur3=0.0d0

      do 465 n=1,8
      do 465 nn=1,3

      cur1=cur1+sigma(vox(i,j,k),1,nn)*af(nn,n)*uu(n)
      cur2=cur2+sigma(vox(i,j,k),2,nn)*af(nn,n)*uu(n)
      cur3=cur3+sigma(vox(i,j,k),3,nn)*af(nn,n)*uu(n)

465   continue

c  sum into the global average currents

      currx=currx+cur1
      curry=curry+cur2
      currz=currz+cur3

470   continue

      call MPI_ALLREDUCE(currx,cuxxp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(curry,cuyyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(currz,cuzzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)


c Volume average currents
      cuxxp=cuxxp/dfloat(ns)
      cuyyp=cuyyp/dfloat(ns)
      cuzzp=cuzzp/dfloat(ns)

      return
```

```
      end
c
c***********************************************************
c
      subroutine gbah(om,uh,dk,vox,nx,ny,nz,d1,d2)

      implicit none
      include 'mpif.h'

      integer nx,ny,nz,d1,d2,mxy,pflag,nphase
      integer im,jm,km,ifxa,ifxb,ifya,ifyb
      integer myrank,nprocs,ierr

      double precision uh(nx,ny,d1-1:d2+1)
      double precision om(nx,ny,d1-1:d2+1)
      double precision gb_time(6)

      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision dk(nphase,8,8)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      gb_time(1) = MPI_Wtime(ierr)

      om = 0.0d0

      do km=d1,d2
      do jm=1,ny
      do im=1,nx

      if ((im+1).GT.nx) then
         ifxa = 1
      else
         ifxa = im+1
      end if

      if ((im-1).LE.0) then
         ifxb = nx
      else
         ifxb = im-1
      end if

      if ((jm+1).GT.ny) then
         ifya = 1
      else
```

```
            ifya = jm+1
         end if

         if ((jm-1).LE.0) then
            ifyb = ny
         else
            ifyb = jm-1
         end if

c SELF TERM

         om(im,jm,km) =

c u(ib(m,1))
     & uh(im,ifya,km)*
     &(dk(vox(im,jm,km),1,4)
     &+dk(vox(ifxb,jm,km),2,3)
     &+dk(vox(im,jm,km-1),5,8)
     &+dk(vox(ifxb,jm,km-1),6,7) )+

c u(ib(m,2))
     & uh(ifxa,ifya,km)*
     & (dk(vox(im,jm,km),1,3)+dk(vox(im,jm,km-1),5,7) )+

c u(ib(m,3))
     & uh(ifxa,jm,km)*(dk(vox(im,jm,km),1,2)
     &+ dk(vox(im,ifyb,km),4,3)
     &+ dk(vox(im,ifyb,km-1),8,7)
     &+ dk(vox(im,jm,km-1),5,6) ) +

c u(ib(m,4))
     &  uh(ifxa,ifyb,km)*(dk(vox(im,ifyb,km),4,2)
     &+  dk(vox(im,ifyb,km-1),8,6) ) +

c u(ib(m,5))
     &   uh(im,ifyb,km)*(dk(vox(ifxb,ifyb,km),3,2)
     & +dk(vox(im,ifyb,km),4,1)
     & +dk(vox(ifxb,ifyb,km-1),7,6)
     & +dk(vox(im,ifyb,km-1),8,5) ) +

c u(ib(m,6))
     &  uh(ifxb,ifyb,km)*(dk(vox(ifxb,ifyb,km),3,1)
     &+  dk(vox(ifxb,ifyb,km-1),7,5) ) +

c u(ib(m,7))
     & uh(ifxb,jm,km)*(
     & dk(vox(ifxb,ifyb,km),3,4)
     &+dk(vox(ifxb,jm,km),2,1)
     &+dk(vox(ifxb,ifyb,km-1),7,8)
```

```
     &+dk(vox(ifxb,jm,km-1),6,5) ) +

c u(ib(m,8))
     & uh(ifxb,ifya,km)*( dk(vox(ifxb,jm,km),2,4)
     &+dk(vox(ifxb,jm,km-1),6,8) ) +

c u(ib(m,9))
     & uh(im,ifya,km-1)*(dk(vox(im,jm,km-1),5,4)
     &+  dk(vox(ifxb,jm,km-1),6,3) ) +

c u(ib(m,10))
     & uh(ifxa,ifya,km-1)*(dk(vox(im,jm,km-1),5,3) )+

c u(ib(m,11))
     & uh(ifxa,jm,km-1)*(dk(vox(im,ifyb,km-1),8,3)
     &+  dk(vox(im,jm,km-1),5,2) ) +

c u(ib(m,12))
     &   uh(ifxa,ifyb,km-1)*( dk(vox(im,ifyb,km-1),8,2) )+

c u(ib(m,13))
     & uh(im,ifyb,km-1)*(dk(vox(im,ifyb,km-1),8,1)
     &+  dk(vox(ifxb,ifyb,km-1),7,2) ) +

c u(ib(m,14))
     &  uh(ifxb,ifyb,km-1)*( dk(vox(ifxb,ifyb,km-1),7,1) )+

c u(ib(m,15))
     & uh(ifxb,jm,km-1)*(dk(vox(ifxb,ifyb,km-1),7,4)
     &+  dk(vox(ifxb,jm,km-1),6,1) )+

c u(ib(m,16))
     &uh(ifxb,ifya,km-1)*( dk(vox(ifxb,jm,km-1),6,4) )+

c u(ib(m,17))
     & uh(im,ifya,km+1)*(dk(vox(im,jm,km),1,8)
     &+  dk(vox(ifxb,jm,km),2,7) )+

c u(ib(m,18))
     & uh(ifxa,ifya,km+1)*( dk(vox(im,jm,km),1,7) )+

c u(ib(m,19))
     & uh(ifxa,jm,km+1)*(dk(vox(im,jm,km),1,6)
     &+ dk(vox(im,ifyb,km),4,7) ) +

c u(ib(m,20))
     & uh(ifxa,ifyb,km+1)*( dk(vox(im,ifyb,km),4,6) )+

c u(ib(m,21))
```

```
     & uh(im,ifyb,km+1)*(dk(vox(im,ifyb,km),4,5)
     &+ dk(vox(ifxb,ifyb,km),3,6) ) +

c u(ib(m,22))
     & uh(ifxb,ifyb,km+1)*( dk(vox(ifxb,ifyb,km),3,5) )+

c u(ib(m,23))
     & uh(ifxb,jm,km+1)*(dk(vox(ifxb,ifyb,km),3,8)
     &+ dk(vox(ifxb,jm,km),2,5) ) +

c u(ib(m,24))
     & uh(ifxb,ifya,km+1)*( dk(vox(ifxb,jm,km),2,8) )+

c u(ib(m,25))
     & uh(im,jm,km-1)*(dk(vox(ifxb,ifyb,km-1),7,3)
     &+ dk(vox(im,ifyb,km-1),8,4)
     &+ dk(vox(ifxb,jm,km-1),6,2)
     &+ dk(vox(im,jm,km-1),5,1) ) +

c u(ib(m,26))
     & uh(im,jm,km+1)*(
     & dk(vox(ifxb,ifyb,km),3,7)
     &+dk(vox(im,ifyb,km),4,8)
     &+dk(vox(im,jm,km),1,5)
     &+dk(vox(ifxb,jm,km),2,6) ) +

c u(ib(m,27))
     &  uh(im,jm,km)* (dk(vox(im,jm,km),1,1)
     &+  dk(vox(ifxb,jm,km),2,2)
     &+  dk(vox(ifxb,ifyb,km),3,3)
     &+  dk(vox(im,ifyb,km),4,4)
     &+  dk(vox(im,jm,km-1),5,5)
     &+  dk(vox(ifxb,jm,km-1),6,6)
     &+  dk(vox(ifxb,ifyb,km-1),7,7)
     &+  dk(vox(im,ifyb,km-1),8,8) )


      end do; end do; end do

      gb_time(2) = MPI_Wtime(ierr)

c
c Do top/bottom layer switch on matrix: om
c

      call z_ghost_dp(om,nx,ny,d1,d2)

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
```

```
      return
      end
c
c***********************************************************
c
      subroutine dpixel(nx,ny,nz,ns,pix)
      implicit none

      integer nx,ny,nz,ns,nphase,nxy
      integer i,j,k,m,pflag
      integer*2 pix(nx,ny,nz)
      integer*2 pix0

      common/list1/pflag,nphase

c  (USER)  If you want to set up a test image inside the program, instead of
c  reading it in from a file, this should be done inside this subroutine.

      nxy=nx*ny
      do 200 k=1,nz
      do 200 j=1,ny
      do 200 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      read(9,*) pix(i,j,k)
200   continue

       do k=1,nz
       do j=1,ny
       do i=1,nx

       pix0 =  pix(i,j,k)

       if(pix0.lt.1) then
        write(7,*) "Phase label in pix < 1--error at ",i,j,k
       end if
       if(pix0.gt.nphase) then
        write(7,*) "Phase label in pix > nphase--error at ",i,j,k
       end if

       end do; end do; end do


      return
      end
c
c***********************************************************
c
      subroutine dassig(nx,ny,nz,prob,pix)
      implicit none
```

```
      integer nx,ny,nz,ns,nphase,ii,jj,kk,i,pflag

      integer*2 pix(nx,ny,nz)
      double precision prob(nphase)

      common/list1/pflag,nphase

        ns=nx*ny*nz
        prob=0.0d0

        do kk=1,nz
        do jj=1,ny
        do ii=1,nx
        do i=1,nphase
           if(pix(ii,jj,kk).eq.i) then
              prob(i)=prob(i)+1.0d0
           end if
        end do; end do
        end do; end do

        prob=prob/dfloat(ns)

        return
        end
c
c***********************************************************
c
      subroutine ipxyz(mm,i,j,k,ipx,ipy,ipz,nx,ny,nz)

      implicit none
      integer mm,i,j,k,ipx,ipy,ipz,nx,ny,nz

      if (mm.le.4) then
         ipz=k
      else
         ipz=k+1
      end if

      if ((mm.eq.1).OR.(mm.eq.5)) then
         ipx=i
         ipy=j
      end if

      if ((mm.eq.2).OR.(mm.eq.6)) then
         ipx = i+1
         ipy=j

         if (i.ge.nx) then
```

```
            ipx=1
         end if

      end if

      if ((mm.eq.3).OR.(mm.eq.7)) then
         ipx = i+1
         if (i.ge.nx) then
            ipx=1
         end if
         ipy = j+1
         if (j.ge.ny) then
            ipy=1
         end if
      end if

      if ((mm.eq.4).OR.(mm.eq.8)) then
         ipx = i
         ipy = j+1

         if (j.ge.ny) then
            ipy=1
         end if

      end if

      return
      end
c
c**********************************************************
c
      subroutine m2ijk(inps,i,j,k,ni,nj,nk)

      implicit none
      integer inps,ns
      integer c
      integer kdiv,jdiv
      integer rj,rk
      integer i,j,k,ni,nj,nk

      ns=ni*nj
      kdiv=inps/ns
      c = ns*kdiv
      rk = inps-c

      if (rk.eq.0) then
         k=kdiv
         j=nj
         i=ni
```

```
      else
         k=kdiv+1
      end if

      if (k.ne.kdiv) then

         jdiv=rk/ni
         c=jdiv*ni
         rj = rk-c

         if (rj.eq.0) then
            j=jdiv
            i=ni
         else
            j=jdiv+1
            i=rj
         end if

      end if

      return
      end
c
c**********************************************************
c
      subroutine z_ghost_int(arr0,mx,my,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      integer*2 arr0(mx,my,d1-1:d2+1)
      integer*2, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
```

```
      call t2b(bot,top,mx,my)

      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call b2t(bot,top,mx,my)

      arr0(:,:,d2+1) = top

      deallocate(bot)
      deallocate(top)

      return
      end
c
c*********************************************************
c
      subroutine z_ghost_dp(arr0,mx,my,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,d1,d2

      double precision arr0(mx,my,d1-1:d2+1)

      double precision, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
      call t2b_dp(bot,top,mx,my)
      arr0(:,:,d1-1) = bot
```

```
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
      call b2t_dp(bot,top,mx,my)
      arr0(:,:,d2+1) = top
      deallocate(bot)
      deallocate(top)

      return
      end
c
c************************************************************
c
      subroutine t2b(b_layer,t_layer,nx,ny)

c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)


      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(b_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)
```

```
      else

      call mpi_recv(b_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c***********************************************************
c
      subroutine b2t(b_layer,t_layer,nx,ny)

c
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(t_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)
```

```
      else

      call mpi_recv(t_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c************************************************************
c
      subroutine t2b_dp(b_layer,t_layer,nx,ny)
c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om top2bottom layers
c
c RECV a new b_layer (BOTTOM layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)
      double precision b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_irecv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else
```

```
      call mpi_recv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*******************************************************
c
      subroutine b2t_dp(b_layer,t_layer,nx,ny)

c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      double precision b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_Irecv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)
```

```
      else

      call mpi_recv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
```

6.1.2 ELAS3D_MPI.f

```
c  **********************  elas3d_mpi.f   *************************
c
c  This is the new MPI version of the elas3d.f code from
c  Section 9.3.2 of NISTIR 6269.
c
c  The main differences with this code compared to the serial
c  version are:
c
c  1. Removal of ib array.
c  2. Change of dimensionality on pix from pix(m) to pix(i,j,k)
c  Maximum value of m = nx*ny*nz (nx,ny,nz are the array dims).
c  3.  All important arrays (pix,vox,gb,b,u,h,Ah) are dynamically allocated.
c
c  IN THIS VERSION:
c
c  The USER needs the following input:
c  (Search for occurences of "USER" in the code).
c
c 1. A 3-D pixel value data file with input & output names.
c 2. The values of the 3 dimensions: (nx,ny,nz)
c 3. The number of phases in the mixture: nphase
c 4. A convergence value: gtest
c 5. Initial values for shears and strains: exx,eyy,ezz,exy,exz,eyz
c 6. Values for DEMBX_MPI and how long it will run: kmax & ldemb
c
c 7. Flag for printing timing info for all data
c    passing MPI routines ( FEMAT_MPI, ENERGY_MPI, DEMBX)
c    from MAIN is called: pflag
c    pflag Values = 0,1 0=no timing info; 1=print timing info
c
c    pflag is a common value.
c
c    Timing info for the RELAXATION loop is not
c    influenced by the pflag and will always be printed.
c
c    User may edit the code to supress the printing.
c
c 8. Timing info stored in arrays namex X_time(i)
c    Where X=n,f,e ie.
c    n_time is in MAIN
c    f_time is in FEMAT_MPI
c    e_time is in ENERGY_MPI
c
c NB:  One also needs to insure that the values for
c      phasemod(i,j) are initialized correctly in
c      SUBROUTINE phasemod_init.
c
c
```

```
c END of NEW comments.
c
c BEGIN ORIGINAL comments.
c
c  BACKGROUND
c
c
c  This program solves the linear elastic equations in a
c  random linear elastic material, subject to an applied macroscopic strain,
c  using the finite element method.  Each pixel in the 3-D digital
c  image is a cubic tri-linear finite element,  having its own
c  elastic moduli tensor. Periodic boundary conditions are maintained.
c  In the comments below, (USER) means that this is a section of code that
c  the user might have to change for his particular problem. Therefore the
c  user is encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The problem being solved is the minimization of the energy
c  1/2 uAu + b u + C, where A is the Hessian matrix composed of the
c  stiffness matrices (dk) for each pixel/element, b is a constant vector
c  and C is a constant that are determined by the applied strain and
c  the periodic boundary conditions, and u is a vector of
c  all the displacements. The solution
c  method used is the conjugate gradient relaxation algorithm.
c  Other variables are:  gb is the gradient = Au+b, h and Ah are
c  auxiliary variables used in the conjugate gradient algorithm (in dembx),
c  dk(n,i,j) is the stiffness matrix of the n'th phase, cmod(n,i,j) is
c  the elastic moduli tensor of the n'th phase, pix is a vector that gives
c  the phase label of each pixel, ib is a matrix that gives the labels of
c  the 27 (counting itself) neighbors of a given node, prob is the volume
c  fractions of the various phases,
c  strxx, stryy, strzz, strxz, stryz, and strxy are the six Voigt
c  volume averaged total stresses, and
c  sxx, syy, szz, sxz, syz, and sxy are the six Voigt
c  volume averaged total strains.

c  DIMENSIONS

c  The vectors u,gb,b,h, and Ah are dimensioned to be the system size,
c  ns=nx*ny*nz, with three components, where the digital image of the
c  microstructure considered is a rectangular paralleliped, nx x ny x nz
c  in size.  The arrays pix and ib are are also dimensioned to the system size.
c  The array ib has 27 components, for the 27 neighbors of a node.
c  Note that the program is set up at present to have at most 100
c  different phases.  This can easily be changed, simply by changing
c  the dimensions of dk, prob, and cmod. The parameter nphase gives the
c  number of phases being considered in the problem.
c  All arrays are passed between subroutines using simple common statements.
```

```
c  STRONGLY SUGGESTED:  READ THE MANUAL BEFORE USING PROGRAM!!

      implicit none
      include 'mpif.h'

c
c  (USER) Change the nx,ny,nz dimensions at the beginning.
c  All important arrays are dynamically allocated.
c

      integer*2, allocatable :: dat(:,:,:), datn(:,:,:)
      integer*2, allocatable :: pix(:,:,:), pixn(:,:,:)
      integer*2, allocatable :: vox(:,:,:)

      integer, allocatable :: d1s(:),d2s(:)

      double precision, allocatable :: b(:,:,:,:)
      double precision, allocatable :: gb(:,:,:,:)
      double precision, allocatable :: u(:,:,:,:)
      double precision, allocatable :: h(:,:,:,:)

      double precision, allocatable ::  phasemod(:,:), prob(:)
      double precision, allocatable ::  dk(:,:,:,:,:),cmod(:,:,:)

      double precision dgg,gg,utot,gtest,C
      double precision exx,eyy,ezz,exz,eyz,exy
      double precision x,y,z,saves
      double precision strxxp,stryyp,strzzp,strxyp,strxzp,stryzp
      double precision sxxp,syyp,szzp,sxyp,sxzp,syzp

      double precision bulk,shear,young,pois

      integer d1,d2,ns,sxip,kkk
      integer i,j,k,nx,ny,nz,nxy,nphase
      integer rem,sz,sized

      integer npoints,micro
      integer kmax,ldemb,ltot,lstep
      integer pflag

      integer irank
      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      double precision starttime,endtime, start_npoint, end_npoint
      double precision kkk_start,kkk_end
      double precision elapsed_time,stress_loop
      double precision n_time(24)
```

```
      common/list1/pflag,nphase
      common/list2/exx,eyy,ezz,exz,eyz,exy
      common/list3/strxxp,stryyp,strzzp,strxyp,strxzp,stryzp
      common/list4/sxxp,syyp,szzp,sxyp,sxzp,syzp

      call MPI_INIT(ierr)

      starttime = MPI_Wtime(ierr)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      if (myrank.eq.0) then
      write(*,*) "There are ",nprocs," processors running this job."
      end if

c
c    USER: Change nx,ny,nz values to match your data.
c
      nx=100
      ny=100
      nz=100
      nphase=88

      nxy=nx*ny
      ns=nx*ny*nz
      sz=nz/nprocs
      mxy= 3*nx*ny

      gtest=1.d-10*ns
c
c     pflag=0 for no timing info printed.
c     pflag=1 for timing info printed.
      pflag = 0
c
c End this USER section.
c

      utot =0.0d0

c
c USER: put phasemod definitions in
c       subroutine "phasemod_init".
c
      allocate(phasemod(nphase,2))
      call phasemod_init(phasemod)

      allocate( dk(nphase,8,3,8,3) )
```

```
      allocate( cmod(nphase,6,6) )
      allocate( prob(nphase) )

      if (myrank.eq.0) then

      allocate (d1s(0:nprocs-1))
      allocate (d2s(0:nprocs-1))

        do irank=0,nprocs-1
          d1s(irank)=irank*sz+1
          d2s(irank)=(irank+1)*sz
        end do

        rem = nz - nprocs*sz

        if (rem.ne.0) then
           do j=1,rem
             irank=nprocs-rem+j-1
             d1s(irank)=d1s(irank)+ j-1
             d2s(irank)=d2s(irank)+ j
           end do
        end if

c Send all d1s(i) and d2s(i) from ROOT
c to NODE i & store into d1 & d2

        do i=0,nprocs-1
      call MPI_SEND(d1s(i),1,MPI_INTEGER,i,0,MPI_COMM_WORLD,ierr)
      call MPI_SEND(d2s(i),1,MPI_INTEGER,i,1,MPI_COMM_WORLD,ierr)
        end do

      end if

      call MPI_RECV(d1,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,status,ierr)
      call MPI_RECV(d2,1,MPI_INTEGER,0,1,MPI_COMM_WORLD,status,ierr)
      write(*,*) "Rank#",myrank,"d1= ",d1," d2= ",d2

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c
c Allocate other arrays which need d1&d2 values.
c
      allocate (gb(nx,ny,d1-1:d2+1,3))
      gb=0.0d0
      allocate(b(nx,ny,d1-1:d2+1,3))
      b = 0.0d0

      allocate (u(nx,ny,d1-1:d2+1,3))
      allocate (h(nx,ny,d1-1:d2+1,3))
```

```
c
c Want the ability to calculate on a series
c of input files based on a value & some if statements.
c
c Compute the average stress and strain in each microstructure.
c (USER) npoints is the number of microstructures to use.

      npoints=1
      n_time(1) = MPI_Wtime(ierr)

      do micro=1,npoints
c
c Allocate pix, so root can read it.
c
      if (myrank.eq.0) then
         allocate (pix(nx,ny,nz))
      end if

      start_npoint=MPI_Wtime(ierr)
      n_time(2) = MPI_Wtime(ierr)

      if (myrank.eq.0) then

c (USER) Unit 9 is the microstructure input file,
c        Unit 7 is the results output file.
c        Get pix from the input file (unit=9).

        if (micro.eq.1) then
        open (unit=9,file='dk.102.100')
        open (unit=7,file='dk.102.100.out')
        end if

        write(*,*) "MICRO = ", micro
        write(7,*) "MICRO = ", micro

c
c Finally... read in pix
c
      write(*,*) "call dpixel"
      call dpixel(nx,ny,nz,ns,pix)
      write(*,*) "back from dpixel"


c ns=total number of sites
      write(7,9010) nx,ny,nz,ns,nprocs
9010  format('nx= ',i4,' ny= ',i4,' nz= ',i4,' ns= 'i8,' nprocs= ',i4)

      end if
```

```
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c Now that the nodes are set up correctly,
c one can pass the data from the root node (myrank=0)
c to all the rest.

      allocate(dat(nx,ny,d1:d2))
      sized = SIZE(dat)
      dat=0

      n_time(3)=MPI_Wtime(ierr)

      if (nprocs.eq.1) then
         dat=pix
         write(*,*) "dat=pix"
      end if

      if (nprocs.gt.1) then

         if (myrank.eq.0) then
             dat(:,:,d1:d2)=pix(:,:,d1:d2)
             do i=1,nprocs-1
               allocate (pixn(nx,ny,d1s(i):d2s(i)))
               pixn = pix(:,:,d1s(i):d2s(i))
               sxip = SIZE(pixn)
               call MPI_SEND(pixn,2*sxip,MPI_BYTE,
     &              i,7,MPI_COMM_WORLD,status,ierr)
               deallocate(pixn)
            end do
         else
            allocate(datn(nx,ny,d1:d2))
            call MPI_RECV(datn,2*sized,MPI_BYTE,0,7
     &             ,MPI_COMM_WORLD,status,ierr)

            dat(:,:,d1:d2) = datn
            deallocate(datn)
         end if
      end if

      n_time(4)=MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, " time to get original data= ",
     &          n_time(4)-n_time(3)
      endif

      allocate(vox(nx,ny,d1-1:d2+1))
      vox = 0
```

```
c
c Make the copy
c
      do k=d1,d2
         vox(:,:,k) = dat(:,:,k)
      end do
      deallocate(dat)

c
c Call z_ghost_int to make Z ghost layers of INTEGER*2 values (aka vox).
c
      call z_ghost_int(vox,nx,ny,nz,d1,d2)

77    format(3(a5,i5,2x))
78    format(a,3(i5,2x))

c Apply chosen strains as a homogeneous macroscopic strain
c as the initial condition.

      if (myrank.eq.0) then
           call dassig(nx,ny,nz,prob,pix)

        do i=1,nphase
             write(7,9020) i,phasemod(i,1),phasemod(i,2)
9020    format("Phase ",i3," bulk = ",f12.6," shear = ",f12.6)
        end do

        do i=1,nphase
     write(7,9065) i,prob(i)
9065 format("Volume fraction of phase ",i3,"  is ",f8.5)
        end do

        call flush(7)
        deallocate(pix)

      end if

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c  (USER) Set applied strains.
c  Actual shear strain applied is exy, exz, and eyz as
c  given in the statements below.  The engineering shear strain, by which
c  the shear modulus is usually defined, is twice these values.

        exx=0.1d0
        eyy=0.1d0
        ezz=0.1d0
        exz=0.1d0/2.d0
```

```
        eyz=0.1d0/2.d0
        exy=0.1d0/2.d0

        if (myrank.eq.0) then
        write(7,*) "Applied engineering strains"
        write(7,*) " exx eyy ezz exz eyz exy"
        write(7,*) exx,eyy,ezz,2.*exz,2.*eyz,2.*exy
        write(*,*) "Applied engineering strains"
        write(*,*) " exx eyy ezz exz eyz exy"
        write(*,*) exx," ",eyy," ",ezz," ",2.*exz,
     &            " ",2.*eyz," ",2.*exy
        call flush(7)
        end if

c Set up the elastic modulus variables, finite element stiffness matrices,
c the constant, C, and vector, b, required for computing the energy.
c  (USER) If anisotropic elastic moduli tensors are used, these need to be
c  input in subroutine femat.

      n_time(9)=MPI_Wtime(ierr)
      call femat_mpi(nx,ny,nz,phasemod,d1,d2,vox,b,dk,C,cmod)
      n_time(10)=MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank," femat_mpi time=",n_time(10)-n_time(9)
      endif

      do k=d1,d2
      do j=1,ny
      do i=1,nx
              x=dfloat(i-1)
              y=dfloat(j-1)
              z=dfloat(k-1)
              u(i,j,k,1)=x*exx+y*exy+z*exz
              u(i,j,k,2)=x*exy+y*eyy+z*eyz
              u(i,j,k,3)=x*exz+y*eyz+z*ezz
      end do; end do; end do

      call z_ghost_dp(u,nx,ny,3,d1,d2)

c  RELAXATION LOOP
c  (USER) kmax is the maximum number of times dembx will be called, with
c  ldemb conjugate gradient steps performed during each call.  The total
c  number of conjugate gradient steps allowed for a given elastic
c  computation is kmax*ldemb.

        kmax=40
        ldemb=100
        ltot=0
```

```
c  Call energy to get initial energy and initial gradient

      n_time(15)=MPI_Wtime(ierr)
      call energy_mpi(u,dk,b,C,nx,ny,nz,d1,d2,gb,utot,vox)
      n_time(16)=MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank,"Initial energy_mpi time=",
     &                 n_time(16)-n_time(15)
      endif

c  gg is the norm squared of the gradient (gg=gb*gb)
      dgg= 0.0d0
      gg = 0.0d0
      dgg = SUM(gb(:,:,d1:d2,:)*gb(:,:,d1:d2,:))
      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      n_time(17)=MPI_Wtime(ierr)

      if (myrank.eq.0) then
         write(*,*) " Initial Energy = ",utot, "  gg = ",gg
         write(7,*) " Initial Energy = ",utot, "  gg = ",gg
         call flush(7)
      end if

      elapsed_time=0.0d0
      stress_loop=0.0d0

      n_time(18)=MPI_Wtime(ierr)
      do kkk=1,kmax
         kkk_start = MPI_Wtime(ierr)

c  call dembx_mpi to go into the conjugate gradient solver

        call dembx_mpi(nx,ny,nz,d1,d2,Lstep,gb,u,vox,h,
     &                     gg,dk,gtest,ldemb,kkk)

        ltot=ltot+Lstep
        call energy_mpi(u,dk,b,C,nx,ny,nz,d1,d2,gb,utot,vox)

         kkk_end = MPI_Wtime(ierr)
         elapsed_time=elapsed_time+(kkk_end-kkk_start)

      if (myrank.eq.0) then
         write(7,*) "Energy = ",utot," gg = ",gg
         write(7,*) "Number of conjugate steps  = ",ltot
         write(7,*) "Root took ",kkk_end-kkk_start," s for ",
     &   ltot, "conjugate steps."
```

```
         write(7,*) "Elapsed time=",elapsed_time," s for ",
     &   ltot, "conjugate steps."

         write(*,*) "Energy = ",utot," gg = ",gg
         write(*,*) "Number of conjugate steps  = ",ltot
         write(*,*) "Root took ",kkk_end-kkk_start," s for ",
     &   ltot, "conjugate steps."
         write(*,*) "Elapsed time= ",elapsed_time," s for ",
     &   ltot, "conjugate steps."
         call flush(7)
      end if

c Call energy_mpi to compute energy after dembx_mpi call. If gg < gtest,
c this will be the final energy. If gg is still larger than gtest,
c then this will give an intermediate energy with which to check how the
c relaxation process is coming along.

c  If relaxation process is finished, jump out of loop
      if(gg.le.gtest) goto 444

c  If relaxation process will continue, compute and output stresses
c  and strains as an additional aid to judge how the
c  relaxation procedure is progressing.

      n_time(19)=MPI_Wtime(ierr)
      call stress_mpi(nx,ny,nz,ns,u,vox,cmod,d1,d2)
      n_time(20)=MPI_Wtime(ierr)

        if (myrank.eq.0) then
          write(7,*) " stresses:  xx,yy,zz,xz,yz,xy"
          write(7,*) strxxp,stryyp,strzzp,strxzp,stryzp,strxyp
          write(7,*) " strains:  xx,yy,zz,xz,yz,xy"
          write(7,*) sxxp,syyp,szzp,sxzp,syzp,sxyp
          call flush(7)
        end if

        end do

        n_time(21)=MPI_Wtime(ierr)
444     call stress_mpi(nx,ny,nz,ns,u,vox,cmod,d1,d2)
        n_time(22)=MPI_Wtime(ierr)

      if (myrank.eq.0) then
         write(7,*) " stresses:  xx,yy,zz,xz,yz,xy"
         write(7,*) strxxp,stryyp,strzzp,strxzp,stryzp,strxyp
         write(7,*) " strains:  xx,yy,zz,xz,yz,xy"
         write(7,*) sxxp,syyp,szzp,sxzp,syzp,sxyp
         write(*,*) "Energy = ",utot," gg = ",gg
         write(*,*) "Number of conjugate steps  = ",ltot
```

```
      end if

        bulk=(strxxp+stryyp+strzzp)/(sxxp+syyp+szzp)/3.0d0
        shear=(strxyp/sxyp+strxzp/sxzp+stryzp/syzp)/3.0d0
        young=9.d0*bulk*shear/(3.d0*bulk+shear)
        pois=(3.d0*bulk-2.d0*shear)/2.d0/(3.d0*bulk+shear)

        if (myrank.eq.0) then
           write(7,*) " bulk modulus = ",bulk
           write(7,*) " shear modulus = ",shear
           write(7,*) " Youngs modulus = ",young
           write(7,*) " Poissons ratio = ",pois

           write(*,*) " bulk modulus = ",bulk
           write(*,*) " shear modulus = ",shear
           write(*,*) " Youngs modulus = ",young
           write(*,*) " Poissons ratio = ",pois
        close(unit=9)
        close(unit=7)

        end if
c
c Do another using loop var: npoints
c
      n_time(23) = MPI_Wtime(ierr)

      write(*,*) myrank," took ",n_time(23)-n_time(2),"s for
     &npoints file ", micro

      deallocate(vox)

      end do

      n_time(24) = MPI_Wtime(ierr)

      write(*,*) myrank," took ",n_time(24)-n_time(1),
     & "for all ", npoints, " micro structures."

      endtime = MPI_Wtime(ierr)
      write(*,*) myrank," took ",endtime-starttime, "s in MAIN."

      CALL MPI_FINALIZE(ierr)

      end
c
c***********************************************************
c
      subroutine femat_mpi(nx,ny,nz,phasemod,d1,d2,vox,b,dk,C,cmod)
```

```
      implicit none

      include 'mpif.h'

      integer i,ierr,nx,j,ny,nz
      integer d1,d2,myrank,nprocs
      integer ipx,ipy,ipz
      integer nxy,k,nm,ijk,mm,nn,ii,jj,kk,ll
      integer i3,i8,dn,m,m3,m8
      integer pflag,nphase

      integer status(MPI_STATUS_SIZE)

      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision sum_num,cterm,cpos,cneg
      double precision c,c3,x,y,z
      double precision f_time(24)

      double precision dk(nphase,8,3,8,3)
      double precision dndx(8),dndy(8),dndz(8)
      double precision g(3,3,3),ck(6,6),cmu(6,6),cmod(nphase,6,6)
      double precision es(6,8,3),delta(8,3)
      double precision b(nx,ny,d1-1:d2+1,3)
      double precision, allocatable :: ab(:,:,:), ba(:,:,:)
      double precision exx,eyy,ezz,exz,eyz,exy

      double precision phasemod(nphase,2)

      common/list1/pflag,nphase

      common/list2/exx,eyy,ezz,exz,eyz,exy

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      f_time(1) = MPI_Wtime(ierr)
      nxy=nx*ny

      allocate (ab(nx,ny,3))
      allocate (ba(nx,ny,3))

c  initialize stiffness matrices

      dk=0.0d0

c  set up elastic moduli matrices for each kind of element
c  ck and cmu are the bulk and shear modulus matrices, which need to be
c  weighted by the actual bulk and shear moduli
```

```
ck(1,1)=1.0d0
ck(1,2)=1.0d0
ck(1,3)=1.0d0
ck(1,4)=0.0d0
ck(1,5)=0.0d0
ck(1,6)=0.0d0
ck(2,1)=1.0d0
ck(2,2)=1.0d0
ck(2,3)=1.0d0
ck(2,4)=0.0d0
ck(2,5)=0.0d0
ck(2,6)=0.0d0
ck(3,1)=1.0d0
ck(3,2)=1.0d0
ck(3,3)=1.0d0
ck(3,4)=0.0d0
ck(3,5)=0.0d0
ck(3,6)=0.0d0
ck(4,1)=0.0d0
ck(4,2)=0.0d0
ck(4,3)=0.0d0
ck(4,4)=0.0d0
ck(4,5)=0.0d0
ck(4,6)=0.0d0
ck(5,1)=0.0d0
ck(5,2)=0.0d0
ck(5,3)=0.0d0
ck(5,4)=0.0d0
ck(5,5)=0.0d0
ck(5,6)=0.0d0
ck(6,1)=0.0d0
ck(6,2)=0.0d0
ck(6,3)=0.0d0
ck(6,4)=0.0d0
ck(6,5)=0.0d0
ck(6,6)=0.0d0

cmu(1,1)=4.0d0/3.0d0
cmu(1,2)=-2.0d0/3.0d0
cmu(1,3)=-2.0d0/3.0d0
cmu(1,4)=0.0d0
cmu(1,5)=0.0d0
cmu(1,6)=0.0d0
cmu(2,1)=-2.0d0/3.0d0
cmu(2,2)=4.0d0/3.0d0
cmu(2,3)=-2.0d0/3.0d0
cmu(2,4)=0.0d0
cmu(2,5)=0.0d0
```

```
      cmu(2,6)=0.0d0
      cmu(3,1)=-2.0d0/3.0d0
      cmu(3,2)=-2.0d0/3.0d0
      cmu(3,3)=4.0d0/3.0d0
      cmu(3,4)=0.0d0
      cmu(3,5)=0.0d0
      cmu(3,6)=0.0d0
      cmu(4,1)=0.0d0
      cmu(4,2)=0.0d0
      cmu(4,3)=0.0d0
      cmu(4,4)=1.0d0
      cmu(4,5)=0.0d0
      cmu(4,6)=0.0d0
      cmu(5,1)=0.0d0
      cmu(5,2)=0.0d0
      cmu(5,3)=0.0d0
      cmu(5,4)=0.0d0
      cmu(5,5)=1.0d0
      cmu(5,6)=0.0d0
      cmu(6,1)=0.0d0
      cmu(6,2)=0.0d0
      cmu(6,3)=0.0d0
      cmu(6,4)=0.0d0
      cmu(6,5)=0.0d0
      cmu(6,6)=1.0d0

      do k=1,nphase
      do j=1,6
      do i=1,6

      cmod(k,i,j)=phasemod(k,1)*ck(i,j)+phasemod(k,2)*cmu(i,j)

      end do; end do; end do

c  set up Simpson's integration rule weight vector
      do k=1,3
      do j=1,3
      do i=1,3
      nm=0
      if(i.eq.2) nm=nm+1
      if(j.eq.2) nm=nm+1
      if(k.eq.2) nm=nm+1
      g(i,j,k)=4.0d0**nm
      end do
      end do
      end do

c  loop over the nphase kinds of pixels and Simpson's rule quadrature
c  points in order to compute the stiffness matrices.  Stiffness matrices
```

```
c  of trilinear finite elements are quadratic in x, y, and z, so that
c  Simpson's rule quadrature gives exact results.

      do ijk=1,nphase
      do k=1,3
      do j=1,3
      do i=1,3
      x=dfloat(i-1)/2.0d0
      y=dfloat(j-1)/2.0d0
      z=dfloat(k-1)/2.0d0
c  dndx means the negative derivative, with respect to x, of the shape
c  matrix N (see manual, Sec. 2.2), dndy, and dndz are similar.
      dndx(1)=-(1.0d0-y)*(1.0d0-z)
      dndx(2)=(1.0d0-y)*(1.0d0-z)
      dndx(3)=y*(1.0d0-z)
      dndx(4)=-y*(1.0d0-z)
      dndx(5)=-(1.0d0-y)*z
      dndx(6)=(1.0d0-y)*z
      dndx(7)=y*z
      dndx(8)=-y*z
      dndy(1)=-(1.0d0-x)*(1.0d0-z)
      dndy(2)=-x*(1.0d0-z)
      dndy(3)=x*(1.0d0-z)
      dndy(4)=(1.0d0-x)*(1.0d0-z)
      dndy(5)=-(1.0d0-x)*z
      dndy(6)=-x*z
      dndy(7)=x*z
      dndy(8)=(1.0d0-x)*z
      dndz(1)=-(1.0d0-x)*(1.0d0-y)
      dndz(2)=-x*(1.0d0-y)
      dndz(3)=-x*y
      dndz(4)=-(1.0d0-x)*y
      dndz(5)=(1.0d0-x)*(1.0d0-y)
      dndz(6)=x*(1.0d0-y)
      dndz(7)=x*y
      dndz(8)=(1.0d0-x)*y

c  now build strain matrix

      es=0.0d0

      es(1,:,1)=dndx
      es(2,:,2)=dndy
      es(3,:,3)=dndz
      es(4,:,1)=dndz
      es(4,:,3)=dndx
      es(5,:,2)=dndz
      es(5,:,3)=dndy
      es(6,:,1)=dndy
```

```
      es(6,:,2)=dndx

c  Matrix multiply to determine value at (x,y,z), multiply by
c  proper weight, and sum_num into dk, the stiffness matrix

      f_time(2) = MPI_Wtime(ierr)

      do mm=1,3
      do nn=1,3
      do ii=1,8
      do jj=1,8
c  Define sum over strain matrices and elastic moduli matrix for
c  stiffness matrix
      sum_num=0.0d0
      do kk=1,6
      do ll=1,6
      sum_num=sum_num+es(kk,ii,mm)*cmod(ijk,kk,ll)*es(ll,jj,nn)
      end do; end do
      dk(ijk,ii,mm,jj,nn)=dk(ijk,ii,mm,jj,nn)+g(i,j,k)*sum_num/216.

      end do; end do; end do; end do
      end do; end do; end do; end do

      f_time(3) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "time to calculate dk = ",f_time(3)-f_time(2)
      endif

c  Initialize b and C

      if (myrank.eq.0) then
         write(*,*) "Initializing b & C."
      end if

      b=0.0d0
      C=0.0d0
      c3=0.0d0

999 format(4(i4,1x,),3(f9.6,1x))

c
c  x=nx face
c
      do i3=1,3
      do i8=1,8
      delta(i8,i3) = 0.0d0

      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
```

```
          delta(i8,1)=exx*nx
          delta(i8,2)=exy*nx
          delta(i8,3)=exz*nx
        end if

        end do; end do

        call MPI_BARRIER(MPI_COMM_WORLD,ierr)

        dn=d2
        if (dn.eq.nz) then
            dn = nz-1
        end if

        cpos=0.0d0; cneg=0.0d0

        cterm=0.0d0

        do k=d1,dn
        do j=1,ny-1
        m=nxy*(k-1)+j*nx
        call m2ijk(m,ii,jj,kk,nx,ny,nz)

        do nn=1,3
        do mm=1,8
        call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
        sum_num=0.0d0
        do m3=1,3
        do m8=1,8

        cterm =0.5d0*delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)*
     &                                              delta(mm,nn)
        if (cterm.ge.0.0d0) then
           cpos = cpos + cterm
        else
           cneg = cneg + cterm
        end if

        sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
        end do; end do
c
c Assign b(ipx,ipy,ipz,nn)  = b(ipx,ipy,ipz,nn) + sum_num

        b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

        end do; end do
        end do; end do

c
```

```
c  y=ny face
c
      do i3=1,3
      do i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      end do; end do

      do i=1,nx-1
      do k=d1,dn
      m=nxy*(k-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do nn=1,3
      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
      do m3=1,3
      do m8=1,8

      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
      cterm=0.5d0*delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)*
     &                                              delta(mm,nn)

      if (cterm.ge.0.0d0) then
         cpos = cpos + cterm
      else
         cneg = cneg + cterm
      end if

      end do; end do

      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

      end do; end do
      end do; end do

c    Zface calcs
c
c    Only the last node does these series of calculations since
c    it contains all the necessary data therefore no data transfer
c    occurs.
c
      if (myrank.eq.nprocs-1) then
```

```
      do i3=1,3
      do i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      end do; end do

      do i=1,nx-1
      do j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do nn=1,3
      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
      do m3=1,3
      do m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
      cterm=0.5d0*delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)*
     &                                              delta(mm,nn)
      if (cterm.ge.0.0d0) then
         cpos = cpos + cterm
      else
         cneg = cneg + cterm
      end if

      end do; end do
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num
      end do; end do
      end do; end do

      end if

c
c  x=nx y=ny edge
c

      do i3=1,3
      do i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
```

```
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
      end do; end do


      dn=d2
      if (dn.eq.nz) then
          dn = nz-1
      end if

      do k=d1,dn
      m=nxy*k
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do nn=1,3
      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0d0
      do m3=1,3
      do m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
      cterm=0.5d0*delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)*
     &                                              delta(mm,nn)

      if (cterm.ge.0.0d0) then
         cpos = cpos + cterm
      else
         cneg = cneg + cterm
      end if

      end do; end do
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

      end do; end do
      end do

c
c  x=nx z=nz edge
c
```

```
 if (myrank.eq.nprocs-1) then

 do i3=1,3
 do i8=1,8
 delta(i8,i3)=0.0d0
 if(i8.eq.2.or.i8.eq.3) then
 delta(i8,1)=exx*nx
 delta(i8,2)=exy*nx
 delta(i8,3)=exz*nx
 end if
 if(i8.eq.5.or.i8.eq.8) then
 delta(i8,1)=exz*nz
 delta(i8,2)=eyz*nz
 delta(i8,3)=ezz*nz
 end if
 if(i8.eq.6.or.i8.eq.7) then
 delta(i8,1)=exz*nz+exx*nx
 delta(i8,2)=eyz*nz+exy*nx
 delta(i8,3)=ezz*nz+exz*nx
 end if
 end do; end do

 do j=1,ny-1
 m=nxy*(nz-1)+nx*(j-1)+nx
 call m2ijk(m,ii,jj,kk,nx,ny,nx)

 do nn=1,3
 do mm=1,8
 call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
 sum_num=0.0d0
 do m3=1,3
 do m8=1,8

 sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
 cterm=0.5d0*delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)*
&                                              delta(mm,nn)

 if (cterm.ge.0.0d0) then
    cpos = cpos + cterm
 else
    cneg = cneg + cterm
 end if

 end do; end do

 b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

 end do; end do
 end do
```

```
c
c  y=ny z=nz edge
c
      do i3=1,3
      do i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      end do; end do

      do i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nx)
      do nn=1,3
      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
      do m3=1,3
      do m8=1,8

      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
      cterm=0.5d0*delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)*
     &                                              delta(mm,nn)

      if (cterm.ge.0.0d0) then
         cpos = cpos + cterm
      else
         cneg = cneg + cterm
      end if

      end do; end do

      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num
      end do; end do
      end do
```

```
c
c  x=nx y=ny z=nz corner
c
      do i3=1,3
      do i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
      end do; end do

      m=nx*ny*nz
      call m2ijk(m,ii,jj,kk,nx,ny,nx)
      do nn=1,3
      do mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
```

```
      sum_num=0.0d0
      do m3=1,3
      do m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
      cterm=0.5d0*delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)*
     &                                              delta(mm,nn)

      if (cterm.ge.0.0d0) then
         cpos = cpos + cterm
      else
         cneg = cneg + cterm
      end if

      end do; end do
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

      end do; end do

c
c End if for (myrank.eq.nprocs-1)
c
      end if

      c3 = cpos + cneg
      CALL MPI_ALLREDUCE(c3,C,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     &        MPI_COMM_WORLD,ierr)

       if (myrank.eq.0) then
         write(*,*) "Final C = ", C
       end if

      f_time(4) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*)myrank,"Etime to calculate C & b= ",f_time(4)-f_time(3)
      end if

      if (nprocs.gt.1) then
c
c RECV a new slice per node.
c

      ab = 0.0d0
      ba = b(:,:,d2+1,:)

      f_time(5) = MPI_Wtime(ierr)
      call t2b_dp(ab,ba,nx,ny,3)
      f_time(6) = MPI_Wtime(ierr)
      b(:,:,d1,:) = b(:,:,d1,:) + ab
```

```
      if (pflag.eq.1) then
      write(*,*) myrank, " B upddate: t2b time= ",f_time(6)-f_time(5)
      end if
c
c botp = d1-1
c
      ab = 0.0
      ba = b(:,:,d1-1,:)

      f_time(7) = MPI_Wtime(ierr)
      call b2t_dp(ab,ba,nx,ny,3)
      f_time(8) = MPI_Wtime(ierr)
      b(:,:,d2,:) = b(:,:,d2,:) + ab

      if (pflag.eq.1) then
      write(*,*)  myrank, " B upddate: b2t time= ",f_time(8)-f_time(7)
      end if
c
c Update ghost layers
c
c RECV a new slice per node.

      ab = b(:,:,d1,:)
      ba = b(:,:,d2,:)

      f_time(9) = MPI_Wtime(ierr)
      call t2b_dp(ab,ba,nx,ny,3)
      f_time(10) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "B ghost upddate:t2b time= ",
     &                   f_time(10)-f_time(9)
      end if

      b(:,:,d1-1,:) = ab

      ab = b(:,:,d1,:)
      ba = b(:,:,d2,:)

      f_time(11) = MPI_Wtime(ierr)
      call b2t_dp(ab,ba,nx,ny,3)
      f_time(12) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "B ghost upddate:b2t time= ",
     &                   f_time(12)-f_time(11)
      end if
```

```
      b(:,:,d2+1,:) = ba

      else
c
c nprocs=1
c
      b(:,:,d1,:) = b(:,:,d1,:) + b(:,:,d2+1,:)
      b(:,:,d2,:) = b(:,:,d2,:) + b(:,:,d1-1,:)
      b(:,:,d2+1,:) = b(:,:,d1,:)
      b(:,:,d1-1,:) = b(:,:,d2,:)

      end if

      deallocate(ab)
      deallocate(ba)

      f_time(13) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, "Femat_mpi elapsed time= ",
     &                   f_time(13)-f_time(1)
      end if

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*********************************************************
c
      subroutine energy_mpi(u,dk,b,C,nx,ny,nz,d1,d2,gb,utot,vox)
      implicit none

      include 'mpif.h'

      integer nx,ny,nz,d1,d2,myrank,nprocs,ierr
      integer m3,ik,ij,ii
      integer pflag,nphase

      double precision  u(nx,ny,d1-1:d2+1,3)
      double precision  b(nx,ny,d1-1:d2+1,3)
      double precision gb(nx,ny,d1-1:d2+1,3)
      integer*2 vox(nx,ny,d1-1:d2+1)
      double precision e_time(24)

      double precision c,c3,utot
      double precision dk(nphase,8,3,8,3)

      double precision dutot
```

```
      double precision exx,eyy,ezz,exz,eyz,exy

      common/list1/pflag,nphase
      common/list2/exx,eyy,ezz,exz,eyz,exy

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      e_time(1) = MPI_Wtime(ierr)

      dutot = 0.0d0

c After this call, gb is calculated and data slabs
c are updated and passed.

      call gbah(gb,u,dk,vox,nx,ny,nz,d1,d2)

c Now do the rest of the gb calculations that appear
c in original "energy" subroutine.
c
c utot will be a per processor value.
c Do an MPI_ALLREDUCE on dutot
c so each node will have the current updated version.

      dutot=0.0d0
      do m3=1,3
      do ik=d1,d2
      do ij=1,ny
      do ii=1,nx

      dutot=dutot+0.5d0*u(ii,ij,ik,m3)*gb(ii,ij,ik,m3)+
     &                b(ii,ij,ik,m3)*u(ii,ij,ik,m3)
      end do; end do; end do; end do

      call MPI_ALLREDUCE(dutot,utot,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      utot = utot + C

c easier to add C here than before the above MPI call.

      gb = gb + b

      return
      end
c
c*********************************************************
c
      subroutine dembx_mpi(nx,ny,nz,d1,d2,Lstep,gb,u,vox,h,
```

```
     &                     gg,dk,gtest,ldemb,kkk)

      implicit none

      include 'mpif.h'

      integer nx,ny,nz,d1,d2,ldemb,kkk,ijk
      integer Lstep,myrank,nprocs,ierr
      integer pflag,nphase

      double precision dgg,gg,gglast,lambda,hAh2,hAh,gamma,gtest

      double precision  u(nx,ny,d1-1:d2+1,3)
      double precision gb(nx,ny,d1-1:d2+1,3)
      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision dk(nphase,8,3,8,3)

      double precision Ah(nx,ny,d1-1:d2+1,3)
      double precision  h(nx,ny,d1-1:d2+1,3)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      if(kkk.eq.1) then
        h=gb
      end if

c  Lstep counts the number of conjugate gradient steps taken in
c  each call to dembx

      Lstep=0

      do ijk=1,ldemb
      Lstep=Lstep+1
      Ah=0.0d0

      call gbah(Ah,h,dk,vox,nx,ny,nz,d1,d2)

      hAh = 0.0d0
      hAh2= 0.0d0

      hAh2 = SUM(h(:,:,d1:d2,:)*Ah(:,:,d1:d2,:))

      call MPI_ALLREDUCE(hAh2,hAh,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     &  MPI_COMM_WORLD, ierr)
```

```
      lambda=gg/hAh
      u=u-lambda*h
      gb=gb-lambda*Ah

      gglast=gg

      gg=0.0d0

      dgg = SUM(gb(:,:,d1:d2,:)*gb(:,:,d1:d2,:))
      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      if (gg.lt.gtest)  goto 1000

      gamma = gg/gglast
      h = gb + gamma*h

      end do

1000  continue

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c***********************************************************
c
      subroutine stress_mpi(nx,ny,nz,ns,u,vox,cmod,d1,d2)

      implicit none
      include 'mpif.h'

      integer nx,ny,ns,d1,d2
      integer ifxa,ifya
      integer pflag,nphase

      double precision u(nx,ny,d1-1:d2+1,3), uu(8,3)
      double precision dndx(8),dndy(8),dndz(8)
      double precision es(6,8,3),cmod(nphase,6,6)
      integer*2 vox(nx,ny,d1-1:d2+1)
      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      integer nz,nxy,i,j,k,m,mm,n3,n8,n

      double precision strxx,stryy,strzz,strxz,stryz,strxy
      double precision str11,str22,str33,str13,str23,str12
      double precision strxxp,stryyp,strzzp
```

```
      double precision strxzp,stryzp,strxyp

      double precision s11,s22,s33,s13,s23,s12
      double precision sxx,syy,szz,sxz,syz,sxy
      double precision sxxp,syyp,szzp,sxzp,syzp,sxyp
      double precision exx,eyy,ezz,exz,eyz,exy

      common/list1/pflag,nphase
      common/list2/exx,eyy,ezz,exz,eyz,exy
      common/list3/strxxp,stryyp,strzzp, strxyp,strxzp,stryzp
      common/list4/sxxp, syyp,szzp,sxyp,sxzp, syzp

        call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
        call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

c  set up single element strain matrix
c  dndx, dndy, and dndz are the components of the average strain
c  matrix in a pixel

      dndx(1)=-0.25d0
      dndx(2)=0.25d0
      dndx(3)=0.25d0
      dndx(4)=-0.25d0
      dndx(5)=-0.25d0
      dndx(6)=0.25d0
      dndx(7)=0.25d0
      dndx(8)=-0.25d0
      dndy(1)=-0.25d0
      dndy(2)=-0.25d0
      dndy(3)=0.25d0
      dndy(4)=0.25d0
      dndy(5)=-0.25d0
      dndy(6)=-0.25d0
      dndy(7)=0.25d0
      dndy(8)=0.25d0
      dndz(1)=-0.25d0
      dndz(2)=-0.25d0
      dndz(3)=-0.25d0
      dndz(4)=-0.25d0
      dndz(5)=0.25d0
      dndz(6)=0.25d0
      dndz(7)=0.25d0
      dndz(8)=0.25d0
c  Build averaged strain matrix, follows code in femat, but for average
c  strain over the pixel, not the strain at a point.

      es = 0.0d0
```

```
      es(1,:,1)=dndx
      es(2,:,2)=dndy
      es(3,:,3)=dndz
      es(4,:,1)=dndz
      es(4,:,3)=dndx
      es(5,:,2)=dndz
      es(5,:,3)=dndy
      es(6,:,1)=dndy
      es(6,:,2)=dndx

c  Compute components of the average stress and strain tensors in each pixel
      strxx=0.0d0
      stryy=0.0d0
      strzz=0.0d0
      strxz=0.0d0
      stryz=0.0d0
      strxy=0.0d0
      sxx=0.0d0
      syy=0.0d0
      szz=0.0d0
      sxz=0.0d0
      syz=0.0d0
      sxy=0.0d0

      strxxp=0.0d0
      stryyp=0.0d0
      strzzp=0.0d0
      strxzp=0.0d0
      stryzp=0.0d0
      strxyp=0.0d0
      sxxp=0.0d0
      syyp=0.0d0
      szzp=0.0d0
      sxzp=0.0d0
      syzp=0.0d0
      sxyp=0.0d0

      do 470 k=d1,d2
      do 470 j=1,ny
      do 470 i=1,nx
      m=(k-1)*nxy+(j-1)*nx+i

      if ((i+1).GT.nx) then
         ifxa = 1
      else
         ifxa = i+1
      end if
```

```
      if ((j+1).GT.ny) then
         ifya = 1
      else
         ifya = j+1
      end if

       do mm=1,3
       uu(1,mm)= u(i,j,k,mm)
       uu(2,mm)= u(ifxa,j,k,mm)
       uu(3,mm)= u(ifxa,ifya,k,mm)
       uu(4,mm)= u(i,ifya,k,mm)
       uu(5,mm)= u(i,j,k+1,mm)
       uu(6,mm)= u(ifxa,j,k+1,mm)
       uu(7,mm)= u(ifxa,ifya,k+1,mm)
       uu(8,mm)= u(i,ifya,k+1,mm)
       end do

c  Correct for periodic boundary conditions, some displacements are wrong
c  for a pixel on a periodic boundary.  Since they come from an opposite
c  face, need to put in applied strain to correct them.
      if(i.eq.nx) then
      uu(2,1)=uu(2,1)+exx*nx
      uu(2,2)=uu(2,2)+exy*nx
      uu(2,3)=uu(2,3)+exz*nx
      uu(3,1)=uu(3,1)+exx*nx
      uu(3,2)=uu(3,2)+exy*nx
      uu(3,3)=uu(3,3)+exz*nx
      uu(6,1)=uu(6,1)+exx*nx
      uu(6,2)=uu(6,2)+exy*nx
      uu(6,3)=uu(6,3)+exz*nx
      uu(7,1)=uu(7,1)+exx*nx
      uu(7,2)=uu(7,2)+exy*nx
      uu(7,3)=uu(7,3)+exz*nx
      end if
      if(j.eq.ny) then
      uu(3,1)=uu(3,1)+exy*ny
      uu(3,2)=uu(3,2)+eyy*ny
      uu(3,3)=uu(3,3)+eyz*ny
      uu(4,1)=uu(4,1)+exy*ny
      uu(4,2)=uu(4,2)+eyy*ny
      uu(4,3)=uu(4,3)+eyz*ny
      uu(7,1)=uu(7,1)+exy*ny
      uu(7,2)=uu(7,2)+eyy*ny
      uu(7,3)=uu(7,3)+eyz*ny
      uu(8,1)=uu(8,1)+exy*ny
      uu(8,2)=uu(8,2)+eyy*ny
      uu(8,3)=uu(8,3)+eyz*ny
      end if
      if(k.eq.nz) then
```

```
      uu(5,1)=uu(5,1)+exz*nz
      uu(5,2)=uu(5,2)+eyz*nz
      uu(5,3)=uu(5,3)+ezz*nz
      uu(6,1)=uu(6,1)+exz*nz
      uu(6,2)=uu(6,2)+eyz*nz
      uu(6,3)=uu(6,3)+ezz*nz
      uu(7,1)=uu(7,1)+exz*nz
      uu(7,2)=uu(7,2)+eyz*nz
      uu(7,3)=uu(7,3)+ezz*nz
      uu(8,1)=uu(8,1)+exz*nz
      uu(8,2)=uu(8,2)+eyz*nz
      uu(8,3)=uu(8,3)+ezz*nz
      end if

c  local stresses and strains in a pixel
      str11=0.0d0
      str22=0.0d0
      str33=0.0d0
      str13=0.0d0
      str23=0.0d0
      str12=0.0d0
      s11=0.0d0
      s22=0.0d0
      s33=0.0d0
      s13=0.0d0
      s23=0.0d0
      s12=0.0d0

      do n3=1,3
      do n8=1,8
      s11=s11+es(1,n8,n3)*uu(n8,n3)
      s22=s22+es(2,n8,n3)*uu(n8,n3)
      s33=s33+es(3,n8,n3)*uu(n8,n3)
      s13=s13+es(4,n8,n3)*uu(n8,n3)
      s23=s23+es(5,n8,n3)*uu(n8,n3)
      s12=s12+es(6,n8,n3)*uu(n8,n3)

      do n=1,6
       str11=str11+cmod(vox(i,j,k),1,n)*es(n,n8,n3)*uu(n8,n3)
       str22=str22+cmod(vox(i,j,k),2,n)*es(n,n8,n3)*uu(n8,n3)
       str33=str33+cmod(vox(i,j,k),3,n)*es(n,n8,n3)*uu(n8,n3)
       str13=str13+cmod(vox(i,j,k),4,n)*es(n,n8,n3)*uu(n8,n3)
       str23=str23+cmod(vox(i,j,k),5,n)*es(n,n8,n3)*uu(n8,n3)
       str12=str12+cmod(vox(i,j,k),6,n)*es(n,n8,n3)*uu(n8,n3)
      end do

      end do; end do

c  sum local strains and stresses into global values
```

```
      strxx=strxx+str11
      stryy=stryy+str22
      strzz=strzz+str33
      strxz=strxz+str13
      stryz=stryz+str23
      strxy=strxy+str12
      sxx=sxx+s11
      syy=syy+s22
      szz=szz+s33
      sxz=sxz+s13
      syz=syz+s23
      sxy=sxy+s12
470   continue

c
c Now do MPI to gather all strNN and sNN terms,
c add them at root, then do this final calculation
c and write them to disk.
c

      call MPI_ALLREDUCE(strxx,strxxp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(stryy,stryyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(strzz,strzzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(strxz,strxzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(strxy,strxyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(stryz,stryzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)


      call MPI_ALLREDUCE(sxx,sxxp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(syy,syyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(szz,szzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(sxz,sxzp,1,MPI_DOUBLE_PRECISION,
```

```
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(sxy,sxyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(syz,syzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

c
c Now root has strxx,stryy, ... sxy
c Let him write out data to disk after this Volume averaging
c
c  Volume average of global stresses and strains
      strxxp=strxxp/dfloat(ns)
      stryyp=stryyp/dfloat(ns)
      strzzp=strzzp/dfloat(ns)
      strxzp=strxzp/dfloat(ns)
      stryzp=stryzp/dfloat(ns)
      strxyp=strxyp/dfloat(ns)
      sxxp=sxxp/dfloat(ns)
      syyp=syyp/dfloat(ns)
      szzp=szzp/dfloat(ns)
      sxzp=sxzp/dfloat(ns)
      syzp=syzp/dfloat(ns)
      sxyp=sxyp/dfloat(ns)

      if (myrank.eq.0) then

      write(*,*) "strxxp = ", strxxp
      write(*,*) "stryyp = ", stryyp
      write(*,*) "strzzp = ", strzzp
      write(*,*) "strxyp = ", strxyp
      write(*,*) "strxzp = ", strxzp
      write(*,*) "stryzp = ", stryzp

      write(*,*) "sxxp = ", sxxp
      write(*,*) "syyp = ", syyp
      write(*,*) "szzp = ", szzp
      write(*,*) "sxyp = ", sxyp
      write(*,*) "sxzp = ", sxzp
      write(*,*) "syzp = ", syzp

      end if

      return
      end
c
c***********************************************************
c
```

```
subroutine gbah(om,uh,dk,vox,nx,ny,nz,d1,d2)
implicit none

include 'mpif.h'

integer nx,ny,nz,d1,d2
integer im,jm,km,j,ifxa,ifxb,ifya,ifyb
integer myrank,nprocs,ierr
integer pflag,nphase

double precision uh(nx,ny,d1-1:d2+1,3)
double precision om(nx,ny,d1-1:d2+1,3)
double precision gb_time(6)

integer*2 vox(nx,ny,d1-1:d2+1)

double precision dk(nphase,8,3,8,3)

common/list1/pflag,nphase

call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

gb_time(1) = MPI_Wtime(ierr)

om = 0.0d0

do km=d1,d2
do jm=1,ny
do im=1,nx

if ((im+1).GT.nx) then
   ifxa = 1
else
   ifxa = im+1
end if

if ((im-1).LE.0) then
   ifxb = nx
else
   ifxb = im-1
end if

if ((jm+1).GT.ny) then
   ifya = 1
else
   ifya = jm+1
end if
```

```
      if ((jm-1).LE.0) then
         ifyb = ny
      else
         ifyb = jm-1
      end if

      do j=1,3

c SELF TERM

      om(im,jm,km,j) =

c u(ib(m,1),n)
     & SUM ( uh(im,ifya,km,:)*(
     & dk(vox(im,jm,km),1,j,4,:)
     &+dk(vox(ifxb,jm,km),2,j,3,:)
     &+dk(vox(im,jm,km-1),5,j,8,:)
     &+dk(vox(ifxb,jm,km-1),6,j,7,:) ))+

c u(ib(m,2),n)
     & SUM ( uh(ifxa,ifya,km,:)*(dk(vox(im,jm,km),1,j,3,:)
     &+  dk(vox(im,jm,km-1),5,j,7,:) )) +

c u(ib(m,3),n)
     & SUM ( uh(ifxa,jm,km,:)*(dk(vox(im,jm,km),1,j,2,:)
     &+ dk(vox(im,ifyb,km),4,j,3,:)
     &+ dk(vox(im,ifyb,km-1),8,j,7,:)
     &+ dk(vox(im,jm,km-1),5,j,6,:) )) +

c u(ib(m,4),n)
     & SUM ( uh(ifxa,ifyb,km,:)*(dk(vox(im,ifyb,km),4,j,2,:)
     &+  dk(vox(im,ifyb,km-1),8,j,6,:) )) +

c u(ib(m,5),n)
     &  SUM ( uh(im,ifyb,km,:)*(dk(vox(ifxb,ifyb,km),3,j,2,:)
     & +dk(vox(im,ifyb,km),4,j,1,:)
     & +dk(vox(ifxb,ifyb,km-1),7,j,6,:)
     & +dk(vox(im,ifyb,km-1),8,j,5,:) )) +

c u(ib(m,6),n)
     & SUM ( uh(ifxb,ifyb,km,:)*(dk(vox(ifxb,ifyb,km),3,j,1,:)
     &+  dk(vox(ifxb,ifyb,km-1),7,j,5,:) )) +

c u(ib(m,7),n)
     & SUM(uh(ifxb,jm,km,:)*(
     & dk(vox(ifxb,ifyb,km),3,j,4,:)
     &+dk(vox(ifxb,jm,km),2,j,1,:)
     &+dk(vox(ifxb,ifyb,km-1),7,j,8,:)
     &+dk(vox(ifxb,jm,km-1),6,j,5,:) )) +
```

```
c u(ib(m,8),n)
     & SUM (uh(ifxb,ifya,km,:)*( dk(vox(ifxb,jm,km),2,j,4,:)
     &+dk(vox(ifxb,jm,km-1),6,j,8,:) )) +

c u(ib(m,9),n)
     & SUM ( uh(im,ifya,km-1,:)*(dk(vox(im,jm,km-1),5,j,4,:)
     &+  dk(vox(ifxb,jm,km-1),6,j,3,:) )) +

c u(ib(m,10),n)
     & SUM ( uh(ifxa,ifya,km-1,:)*(dk(vox(im,jm,km-1),5,j,3,:) ))+

c u(ib(m,11),n)
     & SUM ( uh(ifxa,jm,km-1,:)*(dk(vox(im,ifyb,km-1),8,j,3,:)
     &+  dk(vox(im,jm,km-1),5,j,2,:) )) +

c u(ib(m,12),n)
     &  SUM( uh(ifxa,ifyb,km-1,:)*( dk(vox(im,ifyb,km-1),8,j,2,:) ))+

c u(ib(m,13),n)
     & SUM ( uh(im,ifyb,km-1,:)*(dk(vox(im,ifyb,km-1),8,j,1,:)
     &+  dk(vox(ifxb,ifyb,km-1),7,j,2,:) )) +

c u(ib(m,14),n)
     & SUM( uh(ifxb,ifyb,km-1,:)*( dk(vox(ifxb,ifyb,km-1),7,j,1,:) ))+

c u(ib(m,15),n)
     & SUM ( uh(ifxb,jm,km-1,:)*(dk(vox(ifxb,ifyb,km-1),7,j,4,:)
     &+  dk(vox(ifxb,jm,km-1),6,j,1,:) ))+

c u(ib(m,16),n)
     &SUM(uh(ifxb,ifya,km-1,:)*( dk(vox(ifxb,jm,km-1),6,j,4,:) ))+

c u(ib(m,17),n)
     & SUM ( uh(im,ifya,km+1,:)*(dk(vox(im,jm,km),1,j,8,:)
     &+  dk(vox(ifxb,jm,km),2,j,7,:) ))+

c u(ib(m,18),n)
     & SUM (uh(ifxa,ifya,km+1,:)*( dk(vox(im,jm,km),1,j,7,:) ))+

c u(ib(m,19),n)
     & SUM ( uh(ifxa,jm,km+1,:)*(dk(vox(im,jm,km),1,j,6,:)
     &+ dk(vox(im,ifyb,km),4,j,7,:) )) +

c u(ib(m,20),n)
     & SUM (uh(ifxa,ifyb,km+1,:)*( dk(vox(im,ifyb,km),4,j,6,:) ))+

c u(ib(m,21),n)
     & SUM ( uh(im,ifyb,km+1,:)*(dk(vox(im,ifyb,km),4,j,5,:)
```

```
     &+ dk(vox(ifxb,ifyb,km),3,j,6,:) )) +

c u(ib(m,22),n)
     & SUM(uh(ifxb,ifyb,km+1,:)*( dk(vox(ifxb,ifyb,km),3,j,5,:) ))+

c u(ib(m,23),n)
     & SUM ( uh(ifxb,jm,km+1,:)*(dk(vox(ifxb,ifyb,km),3,j,8,:)
     &+ dk(vox(ifxb,jm,km),2,j,5,:) )) +

c u(ib(m,24),n)
     & SUM(uh(ifxb,ifya,km+1,:)*( dk(vox(ifxb,jm,km),2,j,8,:) ))+

c u(ib(m,25),n)
     & SUM ( uh(im,jm,km-1,:)*(dk(vox(ifxb,ifyb,km-1),7,j,3,:)
     &+ dk(vox(im,ifyb,km-1),8,j,4,:)
     &+ dk(vox(ifxb,jm,km-1),6,j,2,:)
     &+ dk(vox(im,jm,km-1),5,j,1,:) )) +

c u(ib(m,26),n)
     & SUM(uh(im,jm,km+1,:)*(
     & dk(vox(ifxb,ifyb,km),3,j,7,:)
     &+dk(vox(im,ifyb,km),4,j,8,:)
     &+dk(vox(im,jm,km),1,j,5,:)
     &+dk(vox(ifxb,jm,km),2,j,6,:) )) +

c u(ib(m,27),n)
     & SUM( uh(im,jm,km,:)* (dk(vox(im,jm,km),1,j,1,:)
     &+  dk(vox(ifxb,jm,km),2,j,2,:)
     &+  dk(vox(ifxb,ifyb,km),3,j,3,:)
     &+  dk(vox(im,ifyb,km),4,j,4,:)
     &+  dk(vox(im,jm,km-1),5,j,5,:)
     &+  dk(vox(ifxb,jm,km-1),6,j,6,:)
     &+  dk(vox(ifxb,ifyb,km-1),7,j,7,:)
     &+  dk(vox(im,ifyb,km-1),8,j,8,:) ))

      end do

      end do; end do; end do

      gb_time(2) = MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*)myrank, "Etime to calc gb/Ah=",gb_time(2)-gb_time(1)
      endif
c
c Do top/bottom layer switch on matrix: om
c
      call z_ghost_dp(om,nx,ny,3,d1,d2)
```

```
      if (pflag.eq.1) then
      write(*,*)myrank, "Etime for t2b gb/Ah=",gb_time(4)-gb_time(3)
      write(*,*)myrank, "Etime for b2t gb/Ah=",gb_time(6)-gb_time(5)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c***********************************************************
c
      subroutine dpixel(nx,ny,nz,ns,pix)
      implicit none

      integer nx,ny,nz,ns,nphase,nxy
      integer i,j,k,m,pflag
      integer*2 pix(nx,ny,nz)
      integer*2 pix0

      common/list1/pflag,nphase

c  (USER)  If you want to set up a test image inside the program, instead of
c  reading it in from a file, this should be done inside this subroutine.

      nxy=nx*ny
      do 200 k=1,nz
      do 200 j=1,ny
      do 200 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      read(9,*) pix(i,j,k)

      if(pix(i,j,k).eq.0) then
         pix(i,j,k)=46
      end if

200   continue

       do k=1,nz
       do j=1,ny
       do i=1,nx

       pix0 =  pix(i,j,k)

       if(pix0.lt.1) then
        write(7,*) "Phase label in pix < 1--error at ",i,j,k
       end if
       if(pix0.gt.nphase) then
        write(7,*) "Phase label in pix > nphase--error at ",i,j,k
```

```
        end if

        end do; end do; end do

       return
       end
c
c*********************************************************
c
       subroutine dassig(nx,ny,nz,prob,pix)
       implicit none

       integer nx,ny,nz,ns,nphase,ii,jj,kk,i,pflag

       integer*2 pix(nx,ny,nz)
       double precision prob(nphase)

       common/list1/pflag,nphase

         ns=nx*ny*nz
         prob=0.0d0

         do kk=1,nz
         do jj=1,ny
         do ii=1,nx
         do i=1,nphase
            if(pix(ii,jj,kk).eq.i) then
               prob(i)=prob(i)+1.0d0
            end if
         end do; end do
         end do; end do

         prob=prob/dfloat(ns)

         return
         end


c
c*********************************************************
c
       subroutine ipxyz(mm,i,j,k,ipx,ipy,ipz,nx,ny,nz)

       implicit none
       integer mm,i,j,k,ipx,ipy,ipz,nx,ny,nz

       if (mm.le.4) then
          ipz=k
       else
```

```
         ipz=k+1
      end if

      if ((mm.eq.1).OR.(mm.eq.5)) then
         ipx=i
         ipy=j
      end if

      if ((mm.eq.2).OR.(mm.eq.6)) then
         ipx = i+1
         ipy=j

         if (i.ge.nx) then
            ipx=1
         end if

      end if

      if ((mm.eq.3).OR.(mm.eq.7)) then
         ipx = i+1
         if (i.ge.nx) then
            ipx=1
         end if
         ipy = j+1
         if (j.ge.ny) then
            ipy=1
         end if
      end if

      if ((mm.eq.4).OR.(mm.eq.8)) then
         ipx = i
         ipy = j+1

         if (j.ge.ny) then
            ipy=1
         end if

      end if

      return
      end
c
c************************************************************
c
      subroutine m2ijk(inps,i,j,k,ni,nj,nk)

      implicit none
      integer inps,ns
      integer c
```

```
      integer kdiv,jdiv
      integer rj,rk
      integer i,j,k,ni,nj,nk

      ns=ni*nj
      kdiv=inps/ns
      c = ns*kdiv
      rk = inps-c

      if (rk.eq.0) then
         k=kdiv
         j=nj
         i=ni
      else
         k=kdiv+1
      end if

      if (k.ne.kdiv) then

         jdiv=rk/ni
         c=jdiv*ni
         rj = rk-c

         if (rj.eq.0) then
            j=jdiv
            i=ni
         else
            j=jdiv+1
            i=rj
         end if

      end if

      return
      end
c
c***********************************************************
c
      subroutine z_ghost_int(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      integer*2 arr0(mx,my,d1-1:d2+1)
      integer*2, allocatable :: bot(:,:), top(:,:)
```

```
      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c
c Make the Z Ghost
c
      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call t2b(bot,top,mx,my)

      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call b2t(bot,top,mx,my)

      arr0(:,:,d2+1) = top

      deallocate(bot)
      deallocate(top)

      return
      end
c
c***********************************************************
c
      subroutine z_ghost_dp(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      double precision arr0(mx,my,d1-1:d2+1,mz)

      double precision, allocatable :: bot(:,:,:), top(:,:,:)
```

```
      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )
c
c Make the Z Ghost
c
      allocate(bot(mx,my,mz))
      allocate(top(mx,my,mz))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1,:)
      top = arr0(:,:,d2,:)
      call t2b_dp(bot,top,mx,my,3)
      arr0(:,:,d1-1,:) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1,:)
      top = arr0(:,:,d2,:)
      call b2t_dp(bot,top,mx,my,3)
      arr0(:,:,d2+1,:) = top
      deallocate(bot)
      deallocate(top)

      return
      end
c
c*******************************************************
c
      subroutine t2b(b_layer,t_layer,nx,ny)
c
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)
```

```
      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(b_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*******************************************************
c
      subroutine b2t(b_layer,t_layer,nx,ny)
c
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)
```

```
      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(t_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c************************************************************
c
      subroutine t2b_dp(b_layer,t_layer,nx,ny,i)
c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om top2bottom layers
c
c RECV a new b_layer (BOTTOM layer) per node.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,mxy,i
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)
      double precision b_layer(nx,ny,i), t_layer(nx,ny,i)
```

```
      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=i*nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_irecv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c************************************************************
c
      subroutine b2t_dp(b_layer,t_layer,nx,ny,i)
c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,mxy,i
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      double precision b_layer(nx,ny,i), t_layer(nx,ny,i)
```

```
      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=i*nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_Irecv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c************************************************************
c
      subroutine phasemod_init(phasemod)
c
c USER: Put phasemod definitions here
c

      implicit none
      integer pflag,nphase,i
      double precision phasemod(nphase,2),saves

      common/list1/pflag,nphase

      phasemod = 0.0d0

c C3S
        phasemod(1,1)=117.6d0
        phasemod(1,2)=0.314d0
c C2S (same as C3S for now)
        phasemod(2,1)=117.6d0
        phasemod(2,2)=0.314d0
```

```
c C3A (same as C3S for now)
        phasemod(3,1)=117.6d0
        phasemod(3,2)=0.314d0
c C4AF (same as C3S for now)
        phasemod(4,1)=117.6d0
        phasemod(4,2)=0.314d0
c gypsum (use from paper with Sylvain)
        phasemod(5,1)=45.7d0
        phasemod(5,2)=0.33d0
c hemihydrate (same as gypsum for now)
        phasemod(6,1)=0.5*(45.7d0+80.0d0)
        phasemod(6,2)=0.5*(0.33d0+0.275d0)
c anhydrite (same as gypsum for now)
        phasemod(7,1)=80.0d0
        phasemod(7,2)=0.275d0
c pozzolan (no pozzolan)
        phasemod(8,1)=0.0d0
        phasemod(8,2)=0.0d0
c inert
        phasemod(9,1)=0.0d0
        phasemod(9,2)=0.0d0
c slag
        phasemod(10,1)=0.0d0
        phasemod(10,2)=0.0d0
c ASG flyash
        phasemod(11,1)=0.0d0
        phasemod(11,2)=0.0d0
c CAS2 fly ash
        phasemod(12,1)=0.0d0
        phasemod(12,2)=0.0d0
c CH
        phasemod(13,1)=42.3d0
        phasemod(13,2)=0.324d0
c C-S-H
        phasemod(14,1)=22.4d0
        phasemod(14,2)=0.25d0
c C3AH6 (same as C-S-H for now)
        phasemod(15,1)=phasemod(14,1)
        phasemod(15,2)=phasemod(14,2)
c ettringite (from C3A) (1/3 gypsum for now)
        phasemod(16,1)=phasemod(14,1)
        phasemod(16,2)=phasemod(14,2)
c ettringite (from C4AF)
        phasemod(17,1)=phasemod(16,1)
        phasemod(17,2)=phasemod(16,2)
c Afm
        phasemod(18,1)=phasemod(13,1)
        phasemod(18,2)=phasemod(13,2)
c FH3 (same as C-S-H for now)
```

```
        phasemod(19,1)=phasemod(14,1)
        phasemod(19,2)=phasemod(14,2)
c pozzolanic C-S-H
        phasemod(20,1)=phasemod(14,1)
        phasemod(20,2)=phasemod(14,2)
c Slag C-S-H
        phasemod(21,1)=phasemod(14,1)
        phasemod(21,2)=phasemod(14,2)
c CaCl2 (in fly ash)
        phasemod(22,1)=0.0d0
        phasemod(22,2)=0.0d0
c Friedel Salt
        phasemod(23,1)=0.0d0
        phasemod(23,2)=0.0d0
c Stratlingite (from fly ash presence)
        phasemod(24,1)=0.0d0
        phasemod(24,2)=0.0d0
c Secondary gypsum (same modulus as regular gypsum)
        phasemod(25,1)=phasemod(5,1)
        phasemod(25,2)=phasemod(5,2)
c CaCO3
        phasemod(26,1)=79.6d0
        phasemod(26,2)=0.31d0
c Afmc
        phasemod(27,1)=phasemod(13,1)
        phasemod(27,2)=phasemod(13,2)
c Inert aggregate
        phasemod(28,1)=0.0d0
        phasemod(28,2)=0.0d0
c Absorbed gypsum (in C-S-H) treat as regular gypsum
        phasemod(29,1)=phasemod(5,1)
        phasemod(29,2)=phasemod(5,2)
c Fly ash
        phasemod(30,1)=0.0d0
        phasemod(30,2)=0.0d0
c C3A (fly ash)
        phasemod(35,1)=0.0d0
        phasemod(35,2)=0.0d0
c Empty porosity (no water)
        phasemod(45,1)=0.0d0
        phasemod(45,2)=0.0d0
c Water-filled porosity (change from label of zero in hydration program)
c  input as bulk modulus (1) and shear modulus (2), preserve in do 1144 below
        phasemod(46,1)=2.0d0
        phasemod(46,2)=0.0d0

c
c Switched off phase for early age.
c
```

```
        phasemod(88,1)=0.0d0
        phasemod(88,2)=0.0d0

c
c USER: end of phasemod defs
c

c  (USER) Program uses bulk modulus (1) and shear modulus (2), so transform
c  Young's modulis (1) and Poisson's ratio (2).

        do 1144 i=1,nphase

        if(i.eq.46) goto 1144

        saves=phasemod(i,1)
        phasemod(i,1)=phasemod(i,1)/3.d0/(1.d0-2.d0*phasemod(i,2))
        phasemod(i,2)=saves/2.d0/(1.d0+phasemod(i,2))
1144    continue

      return
      end
```

6.1.3 THERMAL3D_MPI.f

```
c  ***********************  thermal3d_mpi.f  ***************************
c
c  This is the new MPI version of the thermal3d.f code from
c  Section 9.3.3 of NISTIR 6269.
c
c  The main differences with this code compared to the serial
c  version are:
c
c  1. Removal of ib array.
c  2. Change of dimensionality on pix from pix(m) to pix(i,j,k)
c  Maximum value of m = nx*ny*nz (nx,ny,nz are the array dims).
c  3.  All important arrays (pix,vox,gb,b,u,h,Ah) are dynamically allocated.
c
c  IN THIS VERSION:
c
c  The USER needs the following input:
c  (Search for occurences of "USER" in the code).
c
c 1. A 3-D pixel value data file with input & output names.
c 2. The values of the 3 dimensions: (nx,ny,nz)
c 3. The number of phases in the mixture: nphase
c 4. A convergence value: gtest
c 5. Initial values for eigenstrains for each phase:
c  eigen(:,1) = xx
c  eigen(:,2) = yy
c  eigen(:,3) = zz
c  eigen(:,4) = xz
c  eigen(:,5) = yz
c  eigen(:,6) = xy
c 6. Values for DEMBX_MPI and how long it will run: kmax & ldemb
c
c 7. Flag for printing timing info for all data
c    passing MPI routines (FEMAT_MPI, ENERGY_MPI, DEMBX)
c    from MAIN is called: pflag
c    pflag Values = 0,1 0=no timing info; 1=print timing info
c
c    pflag is a common value.
c
c    Timing info for the RELAXATION loop is not
c    influenced by the pflag and will always be printed.
c
c    User may edit the code to supress the printing.
c
c 8. Timing info stored in arrays namex X_time(i)
c    Where X=n,f,e ie.
c    n_time is in MAIN
c    f_time is in FEMAT_MPI
c    e_time is in ENERGY_MPI
```

```
c
c NB:  One also needs to insure that the values for
c      phasemod(i,j) are initialized correctly in
c      SUBROUTINE phasemod_init.
c
c
c END of NEW comments.
c
c BEGIN ORIGINAL comments.
c
c  BACKGROUND
c
c  Program adjusts dimensions of unit cell,
c  [(1 + macrostrain) times dimension],
c  in response to phases that have a non-zero eigenstrain and arbitrary
c  elastic moduli tensors.
c  All six macrostrains can adjust their values (3-d program), and are
c  stored in the last two positions in the displacement vector u,
c  as listed below. Periodic boundaries are maintained.
c  In the comments below, (USER) means that this is a section of code
c  that the user might have to change for his particular problem.
c  Therefore the user is encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The problem being solved is the minimization of the elastic energy
c  1/2 uAu + bu + C + Tu + Y, where b and C are also functions of the
c  macrostrains.
c  The small array zcon computes the thermal strain energy associated
c  with macrostrains (C term), T is the thermal energy term linear in the
c  displacements (built from ss), b is the regular energy term linear in the
c  b is the regular energy term linear in the
c  displacements, u is the displacements including the macrostrains,
c  gb is the energy gradient vector, h,Ah are auxiliary vectors,
c  dk is the single pixel stiffness matrix, pix is the phase
c  identification vector, and ib is the
c  integer matrix for mapping labels from the 1-27 nearest neighbor
c  labelling to the 1-d system labelling.
c  The array prob(i) contains the volume fractions of the i'th phase,
c  strxx, etc. are the six independent (Voigt notation) volume
c  averaged stresses, sxx, etc. are the six independent (Voigt notation)
c  volume averaged strains (not counting the thermal strains).
c  The variable cmod(i,6,6) gives the elastic moduli tensor
c  of the i'th phase, eigen(i,6) gives the six independent elements
c  of the eigenstrain tensor for the i'th phase (Voigt notation)
c  and dk(i,8,3,8,3) is the stiffness matrix of the i'th
c  phase. The parameter nphase gives the number of phases being considered
c  in the problem, and is set by the user.
```

```
implicit none
include 'mpif.h'

integer*2, allocatable :: dat(:,:,:), datn(:,:,:)
integer*2, allocatable :: pix(:,:,:), pixn(:,:,:)
integer*2, allocatable :: vox(:,:,:)

integer, allocatable :: d1s(:),d2s(:)

double precision, allocatable :: b(:,:,:,:)
double precision, allocatable :: gb(:,:,:,:)
double precision, allocatable :: u(:,:,:,:)
double precision, allocatable :: h(:,:,:,:)
double precision, allocatable :: T(:,:,:,:)

double precision, allocatable ::  phasemod(:,:), prob(:)
double precision, allocatable ::  dk(:,:,:,:,:),cmod(:,:,:)
double precision, allocatable ::  ss(:,:,:),eigen(:,:)

double precision u2(2,3),gb2(2,3)
double precision h2(2,3),Ah2(2,3),T2(2,3)
double precision zcon(2,3,2,3)

double precision dgg,gg,utot,gtest,C
double precision x,y,z,saves
double precision strxxp,stryyp,strzzp,strxyp,strxzp,stryzp
double precision sxxp,syyp,szzp,sxyp,sxzp,syzp

integer d1,d2,ns,sxip,kkk,iskip
integer i,j,k,nx,ny,nz,nxy,nphase
integer rem,irank
integer sz,sized
integer npoints,micro,m
integer kmax,ldemb,ltot,lstep
integer pflag

integer myrank, ierr, nprocs
integer status(MPI_STATUS_SIZE)

double precision starttime,endtime, start_npoint, end_npoint
double precision kkk_start,kkk_end
double precision elapsed_time,stress_loop
double precision n_time(24)

common/list1/pflag,nphase

call MPI_INIT(ierr)

starttime = MPI_Wtime(ierr)
```

```
      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      if (myrank.eq.0) then
      write(*,*) "There are ",nprocs," processors running this job."
      end if

      pflag=0

c (USER) nx,ny,nz are the size of the lattice

      nx=200
      ny=200
      nz=200

c  ns=total number of sites

      ns=nx*ny*nz
      nxy=nx*ny
      sz=nz/nprocs

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase.
      nphase=3

c  (USER) gtest is the stopping criterion, compared to gg=gb*gb.
c  If gtest=abc*ns, when gg < gtest, the rms value per pixel
c  of gb is less than sqrt(abc)
      gtest=1.d-8*ns
c
c Calculate d1 & d2 limits for each node.
c Then ROOT passes these values to workers.
c
      if (myrank.eq.0) then

      allocate (d1s(0:nprocs-1))
      allocate (d2s(0:nprocs-1))

        do irank=0,nprocs-1
          d1s(irank)=irank*sz+1
          d2s(irank)=(irank+1)*sz
        end do

        rem = nz - nprocs*sz

        if (rem.ne.0) then
           do j=1,rem
             irank=nprocs-rem+j-1
```

```
             d1s(irank)=d1s(irank)+ j-1
             d2s(irank)=d2s(irank)+ j
           end do
        end if

c Send all d1s(i) and d2s(i) from ROOT
c to NODE i & store into d1 & d2 on worker.

        do i=0,nprocs-1
      call MPI_SEND(d1s(i),1,MPI_INTEGER,i,0,MPI_COMM_WORLD,ierr)
      call MPI_SEND(d2s(i),1,MPI_INTEGER,i,1,MPI_COMM_WORLD,ierr)
        end do

      end if

      call MPI_RECV(d1,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,status,ierr)
      call MPI_RECV(d2,1,MPI_INTEGER,0,1,MPI_COMM_WORLD,status,ierr)
      write(*,*) "Rank#",myrank,"d1= ",d1," d2= ",d2

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c  (USER)
c  The parameter phasemod(i,j) is the bulk (i,1) and shear (i,2) moduli of
c  the i'th phase. These can be input in terms of Young's modulus E(i,1)
c  and Poisson's ratio nu (i,2).
c  The program, in the do 1144 loop, changes them to bulk and shear
c  moduli, using relations for isotropic elastic moduli.
c  For anisotropic moduli tensors, one can directly input the whole tensor
c  cmod in subroutine femat, and skip this part.
c  If you wish to input in terms of bulk (i,1) and shear (i,2) moduli,
c  then simply comment out do 1144 loop.

      allocate(phasemod(nphase,2))
      allocate(dk(nphase,8,3,8,3))
      allocate(ss(nphase,8,3))
      allocate(cmod(nphase,6,6))
      allocate(prob(nphase))

      phasemod(1,1)=1.0d0
      phasemod(1,2)=0.3d0
      phasemod(2,1)=1.0d0
      phasemod(2,2)=0.3d0
      phasemod(3,1)=5.0d0
      phasemod(3,2)=0.2d0

      do 1144 i=1,nphase
        saves=phasemod(i,1)
        phasemod(i,1)=phasemod(i,1)/3.0d0/(1.0d0-2.0d0*phasemod(i,2))
        phasemod(i,2)=saves/2.0d0/(1.0d0+phasemod(i,2))
```

```
1144    continue

c  (USER) input eigenstrains for each phase
c  (1=xx, 2=yy, 3=zz, 4=xz, 5=yz, 6=xy).

      allocate(eigen(nphase,6))

        eigen(1,1)=0.1d0
        eigen(1,2)=0.1d0
        eigen(1,3)=0.1d0
        eigen(1,4)=0.d0
        eigen(1,5)=0.d0
        eigen(1,6)=0.d0
        eigen(2,1)=0.2d0
        eigen(2,2)=0.2d0
        eigen(2,3)=0.2d0
        eigen(2,4)=0.d0
        eigen(2,5)=0.d0
        eigen(2,6)=0.d0
        eigen(3,1)=0.d0
        eigen(3,2)=0.d0
        eigen(3,3)=0.d0
        eigen(3,4)=0.d0
        eigen(3,5)=0.d0
        eigen(3,6)=0.d0
c
c Allocate other arrays which need d1&d2 values.
c
      allocate (gb(nx,ny,d1-1:d2+1,3))
      gb=0.0d0
      allocate(b(nx,ny,d1-1:d2+1,3))
      b = 0.0d0
      allocate(T(nx,ny,d1-1:d2+1,3))
      T = 0.0d0

      gb2=0.0d0
      T2=0.0d0

      allocate (u(nx,ny,d1-1:d2+1,3))
      allocate (h(nx,ny,d1-1:d2+1,3))

c  Compute the average stress and strain, as well as the macrostrains (overall
c  system size and shape) in each microstructure.
c  (USER) npoints is the number of microstructures to use.

      npoints=1
      n_time(1) = MPI_Wtime(ierr)

        do 8000 micro=1,npoints
```

```
c
c Allocate pix, so root can read it.
c
      if (myrank.eq.0) then
         allocate (pix(nx,ny,nz))
      end if

           start_npoint=MPI_Wtime(ierr)
           n_time(2) = MPI_Wtime(ierr)

      if (myrank.eq.0) then

c
c Get pix from the input file (unit=9).
c

        open (9,file='micro-200-1.dat')
        open (7,file='micro-200-1.out')

        write(7,9010) nx,ny,nz,ns,nprocs
        write(*,9010) nx,ny,nz,ns,nprocs

        write(7,*) 'relaxation criterion gtest = ',gtest
        write(*,*) 'relaxation criterion gtest = ',gtest

9010  format('nx= ',i4,' ny= ',i4,' nz= ',i4,' ns= 'i18,' nprocs= ',i4)

        write(*,*) "MICRO = ", micro
        write(7,*) "MICRO = ", micro

c
c Finally... read in pix
c
        write(*,*) "call dpixel"
        call dpixel(nx,ny,nz,ns,pix)
        write(*,*) "back from dpixel"

      end if

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c Now that the nodes are set up correctly,
c one can pass the data from the root node (myrank=0)
c to all the rest.

      allocate(dat(nx,ny,d1:d2))
      sized = SIZE(dat)
      dat=0
```

```
      n_time(3)=MPI_Wtime(ierr)

      if (nprocs.eq.1) then
         dat=pix
         write(*,*) "dat=pix"
      end if

      if (nprocs.gt.1) then

         if (myrank.eq.0) then

         write(*,*) "Sending datn."

             dat(:,:,d1:d2)=pix(:,:,d1:d2)
             do i=1,nprocs-1
               allocate (pixn(nx,ny,d1s(i):d2s(i)))
               pixn = pix(:,:,d1s(i):d2s(i))
               sxip = SIZE(pixn)
               call MPI_SEND(pixn,2*sxip,MPI_BYTE,
     &              i,7,MPI_COMM_WORLD,status,ierr)
               deallocate(pixn)
            end do
         else
            allocate(datn(nx,ny,d1:d2))
            call MPI_RECV(datn,2*sized,MPI_BYTE,0,7
     &             ,MPI_COMM_WORLD,status,ierr)

            dat(:,:,d1:d2) = datn
            deallocate(datn)
         end if
      end if

      n_time(4)=MPI_Wtime(ierr)

      if (pflag.eq.1) then
      write(*,*) myrank, " time to get original data= ",
     &           n_time(4)-n_time(3)
      endif

      allocate(vox(nx,ny,d1-1:d2+1))
      vox(:,:,d1:d2) = dat

      call z_ghost_int(vox,nx,ny,nz,d1,d2)

77    format(3(a5,i5,2x))
78    format(a,3(i5,2x))

      if (myrank.eq.0) then
         call dassig(nx,ny,nz,prob,pix)
```

```
        write(7,*) '  Phase Moduli'
        do i=1,nphase
        write(7,9020) i,phasemod(i,1),phasemod(i,2)
9020    format("Phase ",i3," bulk = ",f12.6," shear = ",f12.6)
        end do

do i=1,nphase
        write(7,9065) i,prob(i)
9065 format("Volume fraction of phase ",i3,"  is ",f8.5)
        end do

c  output thermal strains for each phase
        write(7,*) '  Thermal Strains'

        do 119 i=1,nphase
        write(7,9029) i,eigen(i,1),eigen(i,2),eigen(i,3)
        write(7,9029) i,eigen(i,4),eigen(i,5),eigen(i,6)
9029    format('Phase ',i3,'  ',3f6.2)
119     continue

        call flush(7)

        deallocate(pix)

        end if

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c  (USER) Set inital macrostrains of computational cell

        utot = 0.0d0
        u2   = 0.0d0

c Apply homogeneous macroscopic strain as the initial condition
c to displacement variables
        do 1050 k=d1,d2
        do 1050 j=1,ny
        do 1050 i=1,nx
                m=nxy*(k-1)+nx*(j-1)+i
                x=dfloat(i-1)
                y=dfloat(j-1)
                z=dfloat(k-1)
                u(i,j,k,1)=x*u2(1,1)+y*u2(2,3)+z*u2(1,1)
                u(i,j,k,2)=x*u2(2,3)+y*u2(1,2)+z*u2(2,2)
                u(i,j,k,3)=x*u2(2,1)+y*u2(2,2)+z*u2(1,3)
1050 continue

      call z_ghost_dp(u,nx,ny,3,d1,d2)
```

```
c  Set up the finite element stiffness matrices,the constant, C,
c  the vector, b, required for the energy. b and C depend on the macrostrains.
c  When they are updated, the values of b and C are updated too via
c  calling subroutine femat.
c  Only compute the thermal strain terms the first time femat is called,
c  (iskip=0) as they are unaffected by later changes (iskip=1) in
c  displacements and macrostrains.
c  Compute initial value of gradient gb and gg=gb*gb.
        iskip=0
        Y=0.0d0
        gg=0.0d0

      call femat_mpi2(nx,ny,nz,phasemod,d1,d2,vox,b,
     & dk,C,cmod,zcon,u,u2,T,T2,eigen,ss,iskip,Y)

      call energy_mpi2(nx,ny,nz,d1,d2,C,utot,Y,vox,
     &                 dk,u,b,gb,T,zcon,T2,u2,gb2)

        dgg=0.0d0
        gg=0.0d0

        dgg = SUM(gb(:,:,d1:d2,:)*gb(:,:,d1:d2,:))
        call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

        gg = gg + SUM(gb2*gb2)

        if (myrank.eq.0) then
            write(7,9042) utot,gg
            write(*,9042) utot,gg
            call flush(7)
9042 format(' energy = ',d15.8,'  gg=  ',d15.8)
        end if

      kmax=40
      ldemb=100
      ltot=0
      iskip=1

      do 5000 kkk=1,kmax

c  Call dembx to implement conjugate gradient routine

      call dembx_mpi2(nx,ny,nz,ns,Lstep,gg,gtest,ldemb,kkk,
     &           zcon,u,h,gb,u2,h2,gb2,b,d1,d2,vox,dk)

      ltot=ltot+Lstep
```

```
c  Call energy to compute energy after dembx call. If gg < gtest, this
c  will be the final energy.  If gg is still larger than gtest, then this
c  will give an intermediate energy with which to check how the
c  relaxation process is coming along. The call to energy does not
c  change the gradient or the value of gg.
c  Need to first call femat to update the vector b, as the value of the
c  components of b depend on the macrostrains.

      call femat_mpi2(nx,ny,nz,phasemod,d1,d2,vox,b,
     & dk,C,cmod,zcon,u,u2,T,T2,eigen,ss,iskip,Y)

      call energy_mpi2(nx,ny,nz,d1,d2,C,utot,Y,vox,
     &                 dk,u,b,gb,T,zcon,T2,u2,gb2)

      if (myrank.eq.0) then

          write(7,9043) utot,gg,ltot
          write(*,9043) utot,gg,ltot

9043    format(' energy = ',d15.8,' gg=  ',d15.8,' ltot = ',i6)
        call flush(7)

      end if

c  If relaxation process is finished, jump out of loop
        if(gg.lt.gtest) goto 444
c  Output stresses, strains, and macrostrains as an additional aid in judging
c  how well the relaxation process is proceeding.

      call stress_mpi2(nx,ny,nz,ns,u,u2,vox,cmod,d1,d2,
     &           strxxp,stryyp,strzzp,strxyp,strxzp,stryzp,
     &           sxxp,syyp,szzp,sxyp,sxzp,syzp,eigen)

      if (myrank.eq.0) then
      write(7,*) ' stresses:  xx,yy,zz,xz,yz,xy'
      write(7,*) strxxp,stryyp,strzzp,strxzp,stryzp,strxyp
      write(7,*) ' strains:  xx,yy,zz,xz,yz,xy'
      write(7,*) sxxp,syyp,szzp,sxzp,syzp,sxyp
      write(7,*) ' macrostrains in same order'
      write(7,*) u2(1,1),u2(1,2),u2(1,3)
      write(7,*) u2(2,1),u2(2,2),u2(2,3)
      write(7,*) 'avg = ',(u2(1,1)+u2(1,2)+u2(1,3))/3.0d0
      end if

5000  continue

444      call stress_mpi2(nx,ny,nz,ns,u,u2,vox,cmod,d1,d2,
     &           strxxp,stryyp,strzzp,strxyp,strxzp,stryzp,
```

```
     &           sxxp,syyp,szzp,sxyp,sxzp,syzp,eigen)

      if (myrank.eq.0) then
      write(7,*) ' stresses:  xx,yy,zz,xz,yz,xy'
      write(7,*) strxxp,stryyp,strzzp,strxzp,stryzp,strxyp
      write(7,*) ' strains:  xx,yy,zz,xz,yz,xy'
      write(7,*) sxxp,syyp,szzp,sxzp,syzp,sxyp
      write(7,*) ' macrostrains in same order'
      write(7,*) u2(1,1),u2(1,2),u2(1,3)
      write(7,*) u2(2,1),u2(2,2),u2(2,3)
      write(7,*) 'avg = ',(u2(1,1)+u2(1,2)+u2(1,3))/3.0d0
      end if

      deallocate(vox)

8000  continue

c
c Do another calculation using loop var: npoints
c

      deallocate(u)
      deallocate(b)
      deallocate(gb)
      deallocate(h)
      deallocate(T)

      CALL MPI_FINALIZE(ierr)

      end
c
c***********************************************************
c
      subroutine femat_mpi2(nx,ny,nz,phasemod,d1,d2,vox,b,
     & dk,C,cmod,zcon,u,u2,T,T2,eigen,ss,iskip,Y)
c
c  Subroutine sets up the stiffness matrices, the linear term in the
c  regular displacements, b, and the constant term, C, which come from
c  the periodic boundary conditions, the term linear in the displacments,
c  T, that comes from the thermal strains, and the constant term Y.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,nz,iskip,pflag,nphase,nxy
      integer d1,d2,dn,i,j,k,m,mm,nn,mmm,nm,n
      integer ii,jj,kk,ll,ijk,ipp,jpp
      integer i3,i8,m3,m8
```

```
      integer ipx,ipy,ipz

      double precision sum_num,x,y,z,C
      double precision yt,yterm,yneg,ypos,t2temp
      double precision exx,eyy,ezz,exz,eyz,exy

      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision u(nx,ny,d1-1:d2+1,3), u2(2,3)
      double precision b(nx,ny,d1-1:d2+1,3)
      double precision T(nx,ny,d1-1:d2+1,3), T2(2,3)

      double precision dk(nphase,8,3,8,3),phasemod(nphase,2)
      double precision cmod(nphase,6,6), eigen(nphase,6)
      double precision ss(nphase,8,3)

      double precision dndx(8),dndy(8),dndz(8)
      double precision g(3,3,3),ck(6,6),cmu(6,6)
      double precision es(6,8,3),zcon(2,3,2,3),delta(8,3)

      double precision, allocatable :: ab(:,:,:), ba(:,:,:)

      double precision contr

c
c MPI VARIABLES
c

      integer myrank,nprocs,ierr,status(MPI_STATUS_SIZE)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      allocate (ab(nx,ny,3))
      allocate (ba(nx,ny,3))

c  Generate dk, zcon, T, and Y on first pass. After that they are
c  constant, since they are independent of the macrostrains.  Only b gets
c  upgraded as the macrostrains change.

c  Line number 1221 is the routine for b.
      if (iskip.eq.1) goto 1221

c  initialize stiffness matrices
```

```
      dk=0.0d0

c  initialize zcon matrix (gives C term for arbitrary macrostrains)

      zcon=0.0d0

c  (USER) An anisotropic elastic moduli tensor could be input at this point,
c  bypassing this part, which assumes isotropic elasticity, so that there
c  are only two independent numbers making up the elastic moduli tensor,
c  the bulk modulus K and the shear modulus G.

c  Set up elastic moduli matrices for each kind of element
c  ck and cmu are the bulk modulus and shear modulus matrices, which
c  need to multiplied by the actual bulk and shear moduli in each phase.

      ck(1,1)=1.0d0
      ck(1,2)=1.0d0
      ck(1,3)=1.0d0
      ck(1,4)=0.0d0
      ck(1,5)=0.0d0
      ck(1,6)=0.0d0
      ck(2,1)=1.0d0
      ck(2,2)=1.0d0
      ck(2,3)=1.0d0
      ck(2,4)=0.0d0
      ck(2,5)=0.0d0
      ck(2,6)=0.0d0
      ck(3,1)=1.0d0
      ck(3,2)=1.0d0
      ck(3,3)=1.0d0
      ck(3,4)=0.0d0
      ck(3,5)=0.0d0
      ck(3,6)=0.0d0
      ck(4,1)=0.0d0
      ck(4,2)=0.0d0
      ck(4,3)=0.0d0
      ck(4,4)=0.0d0
      ck(4,5)=0.0d0
      ck(4,6)=0.0d0
      ck(5,1)=0.0d0
      ck(5,2)=0.0d0
      ck(5,3)=0.0d0
      ck(5,4)=0.0d0
      ck(5,5)=0.0d0
      ck(5,6)=0.0d0
      ck(6,1)=0.0d0
      ck(6,2)=0.0d0
      ck(6,3)=0.0d0
      ck(6,4)=0.0d0
```

```
      ck(6,5)=0.0d0
      ck(6,6)=0.0d0

      cmu(1,1)=4.0d0/3.0d0
      cmu(1,2)=-2.0d0/3.0d0
      cmu(1,3)=-2.0d0/3.0d0
      cmu(1,4)=0.0d0
      cmu(1,5)=0.0d0
      cmu(1,6)=0.0d0
      cmu(2,1)=-2.0d0/3.0d0
      cmu(2,2)=4.0d0/3.0d0
      cmu(2,3)=-2.0d0/3.0d0
      cmu(2,4)=0.0d0
      cmu(2,5)=0.0d0
      cmu(2,6)=0.0d0
      cmu(3,1)=-2.0d0/3.0d0
      cmu(3,2)=-2.0d0/3.0d0
      cmu(3,3)=4.0d0/3.0d0
      cmu(3,4)=0.0d0
      cmu(3,5)=0.0d0
      cmu(3,6)=0.0d0
      cmu(4,1)=0.0d0
      cmu(4,2)=0.0d0
      cmu(4,3)=0.0d0
      cmu(4,4)=1.0d0
      cmu(4,5)=0.0d0
      cmu(4,6)=0.0d0
      cmu(5,1)=0.0d0
      cmu(5,2)=0.0d0
      cmu(5,3)=0.0d0
      cmu(5,4)=0.0d0
      cmu(5,5)=1.0d0
      cmu(5,6)=0.0d0
      cmu(6,1)=0.0d0
      cmu(6,2)=0.0d0
      cmu(6,3)=0.0d0
      cmu(6,4)=0.0d0
      cmu(6,5)=0.0d0
      cmu(6,6)=1.0d0

      do 31 k=1,nphase
      do 21 j=1,6
      do 11 i=1,6
      cmod(k,i,j)=phasemod(k,1)*ck(i,j)+phasemod(k,2)*cmu(i,j)
11    continue
21    continue
31    continue

c  Set up Simpson's integration rule weight vector
```

```
      do 30 k=1,3
      do 30 j=1,3
      do 30 i=1,3
      nm=0
      if(i.eq.2) nm=nm+1
      if(j.eq.2) nm=nm+1
      if(k.eq.2) nm=nm+1
      g(i,j,k)=4.0d0**nm
30    continue

c  Loop over the nphase kinds of pixels and
c  Simpson's rule quadrature points in order to compute the stiffness
c  matrices.  Stiffness matrices of trilinear finite elements are quadratic
c  in x, y, and z, so that Simpson's rule quadrature gives exact results.
      do 4000 ijk=1,nphase
      do 3000 k=1,3
      do 3000 j=1,3
      do 3000 i=1,3
      x=dfloat(i-1)/2.0d0
      y=dfloat(j-1)/2.0d0
      z=dfloat(k-1)/2.0d0
c  dndx means the negative derivative with respect to x, of the shape
c  matrix N (see manual, Sec. 2.2), dndy and dndz are similar.
      dndx(1)=-(1.0d0-y)*(1.0d0-z)
      dndx(2)=(1.0d0-y)*(1.0d0-z)
      dndx(3)=y*(1.0d0-z)
      dndx(4)=-y*(1.0d0-z)
      dndx(5)=-(1.0d0-y)*z
      dndx(6)=(1.0d0-y)*z
      dndx(7)=y*z
      dndx(8)=-y*z
      dndy(1)=-(1.0d0-x)*(1.0d0-z)
      dndy(2)=-x*(1.0d0-z)
      dndy(3)=x*(1.0d0-z)
      dndy(4)=(1.0d0-x)*(1.0d0-z)
      dndy(5)=-(1.0d0-x)*z
      dndy(6)=-x*z
      dndy(7)=x*z
      dndy(8)=(1.0d0-x)*z
      dndz(1)=-(1.0d0-x)*(1.0d0-y)
      dndz(2)=-x*(1.0d0-y)
      dndz(3)=-x*y
      dndz(4)=-(1.0d0-x)*y
      dndz(5)=(1.0d0-x)*(1.0d0-y)
      dndz(6)=x*(1.0d0-y)
      dndz(7)=x*y
      dndz(8)=(1.0d0-x)*y

c  now build strain matrix
```

```
      es=0.0d0

      es(1,:,1)=dndx
      es(2,:,2)=dndy
      es(3,:,3)=dndz
      es(4,:,1)=dndz
      es(4,:,3)=dndx
      es(5,:,2)=dndz
      es(5,:,3)=dndy
      es(6,:,1)=dndy
      es(6,:,2)=dndx

c  now do matrix multiply to determine value at (x,y,z), multiply by
c  proper weight, and sum into dk, the stiffness matrix
      do 900 mm=1,3
      do 900 nn=1,3
      do 900 ii=1,8
      do 900 jj=1,8
c  define sum over strain matrices and elastic moduli matrix for
c  stiffness matrix
      sum_num=0.0d0
      do 890 kk=1,6
      do 890 ll=1,6
      sum_num=sum_num+es(kk,ii,mm)*cmod(ijk,kk,ll)*es(ll,jj,nn)
890   continue
      dk(ijk,ii,mm,jj,nn)=dk(ijk,ii,mm,jj,nn)+g(i,j,k)*sum_num/216.
900   continue
3000  continue
4000  continue

c  Now compute the ss matrices, which give the thermal strain terms
c  for the i'th phase, single pixel.

      dndx(1)=-0.25d0
      dndx(2)=0.25d0
      dndx(3)=0.25d0
      dndx(4)=-0.25d0
      dndx(5)=-0.25d0
      dndx(6)=0.25d0
      dndx(7)=0.25d0
      dndx(8)=-0.25d0
      dndy(1)=-0.25d0
      dndy(2)=-0.25d0
      dndy(3)=0.25d0
      dndy(4)=0.25d0
      dndy(5)=-0.25d0
      dndy(6)=-0.25d0
      dndy(7)=0.25d0
```

```
      dndy(8)=0.25d0
      dndz(1)=-0.25d0
      dndz(2)=-0.25d0
      dndz(3)=-0.25d0
      dndz(4)=-0.25d0
      dndz(5)=0.25d0
      dndz(6)=0.25d0
      dndz(7)=0.25d0
      dndz(8)=0.25d0
c  now build average strain matrix

      es=0.0d0

      es(1,:,1)=dndx
      es(2,:,2)=dndy
      es(3,:,3)=dndz
      es(4,:,1)=dndz
      es(4,:,3)=dndx
      es(5,:,2)=dndz
      es(5,:,3)=dndy
      es(6,:,1)=dndy
      es(6,:,2)=dndx

      do 3598 mmm=1,nphase
      do 3798 nn=1,3
      do 3798 mm=1,8
      sum_num=0.0d0
      do 3698 nm=1,6
      do 3698 n=1,6
      sum_num=sum_num+cmod(mmm,n,nm)*es(n,mm,nn)*eigen(mmm,nm)
3698  continue
      ss(mmm,mm,nn)=sum_num
3798  continue
3598  continue

c  now call subroutine const to generate zcon
c  zcon is a (2,3) x (2,3) matrix

      call const_mpi(dk,zcon,nx,ny,nz,vox,d1,d2)

c  Now set up linear term, T, for thermal energy. It does not depend
c  on the macrostrains or displacements, so there is no need to update it
c  as the macrostrains change. T is built up out of the ss matrices.

      T=0.0d0
      T2=0.0d0

c  Do all points, but no macrostrain terms
c  note:  factor of 2 on linear thermal term is cancelled
```

```
c  by factor of 1/2 out in front of total energy term
c
c The k loop below is for parallel operation.
c
c Ghost layers on T must still be prepared.
c T2 needs no ghost layers. This is a (2X3) array
c and individual elements are claculated per node.
c T2 needs to be SUMMED at ROOT and then passed pack
c out to all nodes.
c
      do 6601 k=d1,d2
      do 6601 j=1,ny
      do 6601 i=1,nx

      m=nxy*(k-1)+nx*(j-1)+i

      do 6600 mm=1,8
      call ipxyz(mm,i,j,k,ipx,ipy,ipz,nx,ny,nz)

      do 6600 nn=1,3

      T(ipx,ipy,ipz,nn)=T(ipx,ipy,ipz,nn)-ss(vox(i,j,k),mm,nn)

6600  continue
6601  continue

c
c MPI for T update, ghost layers etc...
c

      if (nprocs.gt.1) then

c
c RECV a new slice per node.
c

      ab = 0.0d0
      ba = T(:,:,d2+1,:)

      call t2b_dp(ab,ba,nx,ny,3)
      T(:,:,d1,:) = T(:,:,d1,:) + ab

c
c botp = d1-1
c
      ab = 0.0
      ba = T(:,:,d1-1,:)

      call b2t_dp(ab,ba,nx,ny,3)
```

```
      T(:,:,d2,:) = T(:,:,d2,:) + ab

c Update ghost layers
c
c RECV a new slice per node.

      ab = T(:,:,d1,:)
      ba = T(:,:,d2,:)

      call t2b_dp(ab,ba,nx,ny,3)

      T(:,:,d1-1,:) = ab

      ab = T(:,:,d1,:)
      ba = T(:,:,d2,:)

      call b2t_dp(ab,ba,nx,ny,3)

      T(:,:,d2+1,:) = ba

      else
c
c nprocs=1
c
      T(:,:,d1,:) = T(:,:,d1,:) + T(:,:,d2+1,:)
      T(:,:,d2,:) = T(:,:,d2,:) + T(:,:,d1-1,:)
      T(:,:,d2+1,:) = T(:,:,d1,:)
      T(:,:,d1-1,:) = T(:,:,d2,:)

      end if

c  now need to pick up and sum in all terms multiplying macrostrains
      do ipp=1,2
      do jpp=1,3
      exx=0.0d0
      eyy=0.0d0
      ezz=0.0d0
      exz=0.0d0
      eyz=0.0d0
      exy=0.0d0

      contr=0.0d0

      if(ipp.eq.1.and.jpp.eq.1) exx=1.0d0
      if(ipp.eq.1.and.jpp.eq.2) eyy=1.0d0
      if(ipp.eq.1.and.jpp.eq.3) ezz=1.0d0
      if(ipp.eq.2.and.jpp.eq.1) exz=1.0d0
      if(ipp.eq.2.and.jpp.eq.2) eyz=1.0d0
      if(ipp.eq.2.and.jpp.eq.3) exy=1.0d0
```

```
c
c  x=nx face
c
      do 6001 i3=1,3
      do 6001 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
6001  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 6000 j=1,ny-1
      do 6000 k=d1,dn
      m=nxy*(k-1)+j*nx
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 6900 nn=1,3
      do 6900 mm=1,8
      contr= -ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
      T2(ipp,jpp)=T2(ipp,jpp)-ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
6900  continue
6000  continue

c
c All like index terms of T2 on all nodes must be added.
c

c
c  y=ny face
c
      do 6011 i3=1,3
      do 6011 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
6011  continue

      do 6010 i=1,nx-1
      do 6010 k=d1,dn
      m = nxy*(k-1)+nx*(ny-1) + i
```

```
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 6901 nn=1,3
      do 6901 mm=1,8
      contr= -ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
      T2(ipp,jpp)=T2(ipp,jpp)-ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
6901  continue
6010  continue

c
c  z=nz face
c

      if (myrank.eq.nprocs-1) then

      do 6021 i3=1,3
      do 6021 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
6021  continue

      do 6020 i=1,nx-1
      do 6020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 6902 nn=1,3
      do 6902 mm=1,8
      contr= -ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
      T2(ipp,jpp)=T2(ipp,jpp)-ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
6902  continue
6020  continue

      end if
c
c  x=nx y=ny edge
c
      do 6031 i3=1,3
      do 6031 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
```

```
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
6031  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 6030 k=d1,dn
      m=nxy*k
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 6903 nn=1,3
      do 6903 mm=1,8
      contr= -ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
      T2(ipp,jpp)=T2(ipp,jpp)-ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
6903  continue
6030  continue

      if (myrank.eq.nprocs-1) then

c
c  x=nx z=nz edge
c
      do 6041 i3=1,3
      do 6041 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
```

```
6041  continue

      do 6040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 6904 nn=1,3
      do 6904 mm=1,8
      contr= -ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
      T2(ipp,jpp)=T2(ipp,jpp)-ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
6904  continue
6040  continue

c
c  y=ny z=nz edge
c
      do 6051 i3=1,3
      do 6051 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
6051  continue

      do 6050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 6905 nn=1,3
      do 6905 mm=1,8
      contr= -ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
      T2(ipp,jpp)=T2(ipp,jpp)-ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
6905  continue
6050  continue

c
c  x=nx y=ny z=nz corner
```

```
c
      do 6061 i3=1,3
      do 6061 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
6061  continue

      m=nx*ny*nz
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 6906 mm=1,8
      do 6906 nn=1,3
      contr= -ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
      T2(ipp,jpp)=T2(ipp,jpp)-ss(vox(ii,jj,kk),mm,nn)*delta(mm,nn)
6906  continue
```

```
      end if

      end do
      end do

      if (nprocs.GT.1) then
c
c MPI for T2.
c
      do ipp=1,2
      do jpp=1,3
      t2temp = 0.0d0
      call MPI_ALLREDUCE(T2(ipp,jpp),t2temp,1,
     &               MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,ierr)
      T2(ipp,jpp) = t2temp
      end do
      end do

      end if

c  now compute Y, the 0.5(eigen)Cij(eigen) energy, doesn't ever change
c  with macrostrain or displacements

      yterm=0.0d0
      do 8811 k=d1,d2
      do 8811 j=1,ny
      do 8811 i=1,nx

      do 8811 n=1,6
      do 8811 nn=1,6

      m=nx*ny*(k-1) + nx*(j-1) + i

      yterm=0.5d0*eigen(vox(i,j,k),n)*cmod(vox(i,j,k),n,nn)*
     &  eigen(vox(i,j,k),nn)

      if (yterm.ge.0.0d0) then
      ypos = ypos + yterm
      else
      yneg = yneg + yterm
      end if

8811  continue

      yt = ypos + yneg

      if (nprocs.GT.1) then
```

```
           CALL MPI_ALLREDUCE(yt,Y,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     &          MPI_COMM_WORLD,ierr)

        else

           Y=yt

        end if


c  Following needs to be run after every change in macrostrain
c  when energy is recomputed.

1221  continue

c  Use auxiliary variables (exx, etc.) instead of u() variable, for
c  convenience, and to make the following code easier to read.
      exx=u2(1,1)
      eyy=u2(1,2)
      ezz=u2(1,3)
      exz=u2(2,1)
      eyz=u2(2,2)
      exy=u2(2,3)

c  Now set up vector for linear term that comes from periodic boundary
c  conditions.  Notation and conventions same as for T term.
c  This is done using the stiffness matrices, and the periodic terms
c  in the macrostrains.  It is easier to set up b this way than to
c  analytically write out all the terms involved.

      b=0.0d0
      C=0.0d0

c
c  x=nx face
c
      do 2001 i3=1,3
      do 2001 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
2001  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1
```

```
      do 2000 j=1,ny-1
      do 2000 k=d1,dn

      m=nxy*(k-1)+j*nx
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1900 nn=1,3
      do 1900 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0d0
      do 1899 m3=1,3
      do 1899 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
1899  continue

      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1900  continue
2000  continue

c
c  y=ny face
c
      do 2011 i3=1,3
      do 2011 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
2011  continue
      do 2010 i=1,nx-1
      do 2010 k=d1,dn
      m=nxy*(k-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1901 nn=1,3
      do 1901 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0d0

      do 2099 m3=1,3
      do 2099 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2099  continue
```

```
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1901  continue
2010  continue

c
c  z=nz face
c

      if (myrank.eq.nprocs-1) then

      do 2021 i3=1,3
      do 2021 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1902 nn=1,3
      do 1902 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0

      do 2019 m3=1,3
      do 2019 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2019  continue

      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1902  continue
2020  continue

      end if

c
c  x=nx y=ny edge
c

      do 2031 i3=1,3
      do 2031 i8=1,8
```

```
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
2031  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 2030 k=d1,dn
      m=nxy*k
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1903 nn=1,3
      do 1903 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0
      do 2029 m3=1,3
      do 2029 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2029  continue

      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1903  continue
2030  continue

c
c  x=nx z=nz edge
c

      if (myrank.eq.nprocs-1) then

      do 2041 i3=1,3
      do 2041 i8=1,8
      delta(i8,i3)=0.0
```

```
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1904 nn=1,3
      do 1904 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
      do 2039 m3=1,3
      do 2039 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2039  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1904  continue
2040  continue

c
c  y=ny z=nz edge
c

      do 2051 i3=1,3
      do 2051 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
```

```
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1905 nn=1,3
      do 1905 mm=1,8

      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0

      do 2049 m3=1,3
      do 2049 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2049  continue

      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1905  continue
2050  continue

c
c  x=nx y=ny z=nz corner
c

      do 2061 i3=1,3
      do 2061 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
```

```
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
2061  continue
      m=nx*ny*nz
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1906 nn=1,3
      do 1906 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0d0
      do 2059 m3=1,3
      do 2059 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2059  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num
1906  continue

      end if

      if (nprocs.gt.1) then

c
c RECV a new slice per node.
c

      ab = 0.0d0
      ba = b(:,:,d2+1,:)

      call t2b_dp(ab,ba,nx,ny,3)
      b(:,:,d1,:) = b(:,:,d1,:) + ab
```

```
c
c botp = d1-1
c
      ab = 0.0
      ba = b(:,:,d1-1,:)

      call b2t_dp(ab,ba,nx,ny,3)
      b(:,:,d2,:) = b(:,:,d2,:) + ab

c
c Update ghost layers
c
c RECV a new slice per node.
c
      ab = b(:,:,d1,:)
      ba = b(:,:,d2,:)

      call t2b_dp(ab,ba,nx,ny,3)

      b(:,:,d1-1,:) = ab

      ab = b(:,:,d1,:)
      ba = b(:,:,d2,:)

      call b2t_dp(ab,ba,nx,ny,3)

      b(:,:,d2+1,:) = ba

      else
c
c nprocs=1
c
      b(:,:,d1,:) = b(:,:,d1,:) + b(:,:,d2+1,:)
      b(:,:,d2,:) = b(:,:,d2,:) + b(:,:,d1-1,:)
      b(:,:,d2+1,:) = b(:,:,d1,:)
      b(:,:,d1-1,:) = b(:,:,d2,:)

      end if

      deallocate(ab)
      deallocate(ba)

      return
      end
c
c***********************************************************
c
      subroutine const_mpi(dk,zcon,nx,ny,nz,vox,d1,d2)
```

```
c  Subroutine computes the quadratic term in the macrostrains, that comes
c  from the periodic boundary conditions, and sets it up as a
c  (2,3) x (2,3) matrix that couples to the six macrostrains

      implicit none

      include 'mpif.h'

      integer nx,ny,nz,nphase,d1,d2,dn,pflag
      integer nxy,mmm
      integer ii,jj,i3,i8
      integer i,j,k,m,mi,mj
      integer ia,ja,ka
      integer mm,nn,m3,m8,ipx,ipy,ipz,irank

      double precision dk(nphase,8,3,8,3),zcon(2,3,2,3),delta(8,3)
      double precision pp(6,6),s(6,6)

      double precision econ,eterm,epos,eneg,epart
      double precision exx,eyy,ezz,exz,eyz,exy

      integer*2 vox(nx,ny,d1-1:d2+1)
c
c MPI vars
c
      integer myrank,nprocs,ierr,status(MPI_STATUS_SIZE)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c  routine to set up 6 x 6 matrix for energy term involving macro-strains
c  only, pulled out of femat
c  Idea is to evaluate the quadratic term in the macrostrains repeatedly
c  for choices of strain like
c  exx=1, exy=1, all others = 0, build up 21 choices, then recombine
c  to get matrix elements by themselves

      nxy=nx*ny

      s=0.0d0
      pp=0.0d0

      do 5000 ii=1,6
      do 5000 jj=ii,6
      epart=0.0d0
      epos=0.0d0
```

```
eneg=0.0d0
econ=0.0d0
exx=0.0d0
eyy=0.0d0
ezz=0.0d0
exz=0.0d0
eyz=0.0d0
exy=0.0d0
if(ii.eq.1.and.jj.eq.1) exx=1.0d0
if(ii.eq.2.and.jj.eq.2) eyy=1.0d0
if(ii.eq.3.and.jj.eq.3) ezz=1.0d0
if(ii.eq.4.and.jj.eq.4) exz=1.0d0
if(ii.eq.5.and.jj.eq.5) eyz=1.0d0
if(ii.eq.6.and.jj.eq.6) exy=1.0d0
if(ii.eq.1.and.jj.eq.2) then
exx=1.0d0
eyy=1.0d0
end if
if(ii.eq.1.and.jj.eq.3) then
exx=1.0d0
ezz=1.0d0
end if
if(ii.eq.1.and.jj.eq.4) then
exx=1.0d0
exz=1.0d0
end if
if(ii.eq.1.and.jj.eq.5) then
exx=1.0d0
eyz=1.0d0
end if
if(ii.eq.1.and.jj.eq.6) then
exx=1.0d0
exy=1.0d0
end if
if(ii.eq.2.and.jj.eq.3) then
eyy=1.0d0
ezz=1.0d0
end if
if(ii.eq.2.and.jj.eq.4) then
eyy=1.0d0
exz=1.0d0
end if
if(ii.eq.2.and.jj.eq.5) then
eyy=1.0d0
eyz=1.0d0
end if
if(ii.eq.2.and.jj.eq.6) then
eyy=1.0d0
exy=1.0d0
```

```
      end if
      if(ii.eq.3.and.jj.eq.4) then
      ezz=1.0d0
      exz=1.0d0
      end if
      if(ii.eq.3.and.jj.eq.5) then
      ezz=1.0d0
      eyz=1.0d0
      end if
      if(ii.eq.3.and.jj.eq.6) then
      ezz=1.0d0
      exy=1.0d0
      end if
      if(ii.eq.4.and.jj.eq.5) then
      exz=1.0d0
      eyz=1.0d0
      end if
      if(ii.eq.4.and.jj.eq.6) then
      exz=1.0d0
      exy=1.0d0
      end if
      if(ii.eq.5.and.jj.eq.6) then
      eyz=1.0d0
      exy=1.0d0
      end if

c
c  x=nx face
c
      do 2001 i3=1,3
      do 2001 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
2001  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 2000 j=1,ny-1
      do 2000 k=d1,dn
      m=nxy*(k-1)+j*nx
      call m2ijk(m,ia,ja,ka,nx,ny,nz)

      do 1900 nn=1,3
      do 1900 mm=1,8
```

```
      do 1899 m3=1,3
      do 1899 m8=1,8

      eterm=0.5d0*delta(m8,m3)*dk(vox(ia,ja,ka),m8,m3,mm,nn)
     &           *delta(mm,nn)

      if (eterm.ge.0.0d0) then
          epos = epos + eterm
      else
          eneg = eneg + eterm
      end if

1899  continue
1900  continue
2000  continue

c
c Add epos + eneg to get econ at the end....
c
c
c  y=ny face
c
      do 2011 i3=1,3
      do 2011 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
2011  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 2010 i=1,nx-1
      do 2010 k=d1,dn
      m=nxy*(k-1)+nx*(ny-1)+i
      call m2ijk(m,ia,ja,ka,nx,ny,nz)

      do 1901 nn=1,3
      do 1901 mm=1,8
      do 2099 m3=1,3
      do 2099 m8=1,8
      eterm=0.5d0*delta(m8,m3)*dk(vox(ia,ja,ka),m8,m3,mm,nn)
     &           *delta(mm,nn)
      if (eterm.ge.0.0d0) then
          epos = epos + eterm
      else
```

```
           eneg = eneg + eterm
        end if

2099  continue
1901  continue
2010  continue

c
c  x=nx y=ny edge
c
      do 2031 i3=1,3
      do 2031 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
2031  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 2030 k=d1,dn
      m=nxy*k
      call m2ijk(m,ia,ja,ka,nx,ny,nz)

      do 1903 nn=1,3
      do 1903 mm=1,8
      do 2029 m3=1,3
      do 2029 m8=1,8

      eterm=0.5d0*delta(m8,m3)*dk(vox(ia,ja,ka),m8,m3,mm,nn)
     &           *delta(mm,nn)

      if (eterm.ge.0.0d0) then
         epos = epos + eterm
      else
         eneg = eneg + eterm
```

```
      end if

2029  continue
1903  continue
2030  continue

      if (myrank.eq.nprocs-1) then
c
c  z=nz face
c
      do 2021 i3=1,3
      do 2021 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      call m2ijk(m,ia,ja,ka,nx,ny,nz)


      do 1902 nn=1,3
      do 1902 mm=1,8
      do 2019 m3=1,3
      do 2019 m8=1,8

      eterm=0.5d0*delta(m8,m3)*dk(vox(ia,ja,ka),m8,m3,mm,nn)
     &           *delta(mm,nn)

      if (eterm.ge.0.0d0) then
         epos = epos + eterm
      else
         eneg = eneg + eterm
      end if

2019  continue
1902  continue
2020  continue

c
c  x=nx z=nz edge
c
      do 2041 i3=1,3
      do 2041 i8=1,8
      delta(i8,i3)=0.0d0
```

```
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      call m2ijk(m,ia,ja,ka,nx,ny,nz)

      do 1904 nn=1,3
      do 1904 mm=1,8
      do 2039 m3=1,3
      do 2039 m8=1,8

      eterm=0.5d0*delta(m8,m3)*dk(vox(ia,ja,ka),m8,m3,mm,nn)
     &           *delta(mm,nn)

      if (eterm.ge.0.0d0) then
          epos = epos + eterm
      else
          eneg = eneg + eterm
      end if

2039  continue
1904  continue
2040  continue

c
c  y=ny z=nz edge
c

      do 2051 i3=1,3
      do 2051 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
```

```
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      call m2ijk(m,ia,ja,ka,nx,ny,nz)

      do 1905 nn=1,3
      do 1905 mm=1,8
      do 2049 m3=1,3
      do 2049 m8=1,8

      eterm=0.5d0*delta(m8,m3)*dk(vox(ia,ja,ka),m8,m3,mm,nn)
     &           *delta(mm,nn)

      if (eterm.ge.0.0d0) then
          epos = epos + eterm
      else
          eneg = eneg + eterm
      end if

2049  continue
1905  continue
2050  continue

c
c  x=nx y=ny z=nz corner
c

      do 2061 i3=1,3
      do 2061 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
```

```
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
2061  continue
      m=nx*ny*nz
      call m2ijk(m,ia,ja,ka,nx,ny,nz)

      do 1906 nn=1,3
      do 1906 mm=1,8
      do 2059 m3=1,3
      do 2059 m8=1,8

      eterm=0.5d0*delta(m8,m3)*dk(vox(ia,ja,ka),m8,m3,mm,nn)
     &           *delta(mm,nn)

      if (eterm.ge.0.0d0) then
          epos = epos + eterm
      else
          eneg = eneg + eterm
      end if

2059  continue
1906  continue

      end if
```

```
c Now the nprocs-1 processor is done...

      epart = epos + eneg
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      CALL MPI_ALLREDUCE(epart,econ,1,MPI_DOUBLE_PRECISION,
     &        MPI_SUM,MPI_COMM_WORLD,ierr)

      pp(ii,jj)=econ*2.0d0

5000  continue

      do 6000 i=1,6
      do 6000 j=i,6
      if(i.eq.j) s(i,j)=pp(i,j)
      if(i.ne.j) then
      s(i,j)=pp(i,j)-pp(i,i)-pp(j,j)
      end if
6000  continue

      do 7000 i=1,6
      do 7000 j=1,6
      pp(i,j)=0.5d0*(s(i,j)+s(j,i))
7000  continue

c  now map pp(i,j) into zcon(2,3,2,3)

      do 7200 i=1,2
      do 7200 j=1,2
      do 7200 mi=1,3
      do 7200 mj=1,3
      if(i.eq.1) ii=i+mi-1
      if(i.eq.2) ii=i+mi+1
      if(j.eq.1) jj=j+mj-1
      if(j.eq.2) jj=j+mj+1
      zcon(i,mi,j,mj)=pp(ii,jj)

7200  continue

      return
      end
c
c*******************************************************
c
      subroutine energy_mpi2(nx,ny,nz,d1,d2,C,utot,Y,vox,
     &                 dk,u,b,gb,T,zcon,T2,u2,gb2)

c  Subroutine computes the total energy, utot, and the gradient, gb,
```

```
c  for the regular displacements as well as for the macrostrains

      implicit none

      include 'mpif.h'

        integer nx,ny,nz,ns,nxy,d1,d2,nphase,pflag
        integer ii,i2i,i,j,k,m,ij,ik,m3
        integer mi,mj

        double precision u(nx,ny,d1-1:d2+1,3),gb(nx,ny,d1-1:d2+1,3)
        double precision b(nx,ny,d1-1:d2+1,3),T(nx,ny,d1-1:d2+1,3)

        double precision dk(nphase,8,3,8,3),zcon(2,3,2,3)

        double precision u2(2,3),gb2(2,3),T2(2,3)

        double precision utot,tu,tutot,dutot,C,sum_num,Y
        double precision dusum
        double precision exx,eyy,ezz,exz,eyz,exy

        integer*2 vox(nx,ny,d1-1:d2+1)

c
c MPI VARIABLES
c

      integer myrank,nprocs,ierr,status(MPI_STATUS_SIZE)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      ns = nx*ny*nz
      nxy= nx*ny

      gb=0.0d0

      call gbah(gb,u,dk,vox,nx,ny,nz,d1,d2)

c
c Same gbah subroutine as in ELAS3D_MPI.F
c

      utot = 0.0d0
      dutot= 0.0d0

      do m3=1,3
```

```
      do ik=d1,d2
      do ij=1,ny
      do ii=1,nx

      dutot=dutot+0.5d0*u(ii,ij,ik,m3)*gb(ii,ij,ik,m3)+
     &                b(ii,ij,ik,m3)*u(ii,ij,ik,m3)

      end do; end do; end do; end do

      call MPI_ALLREDUCE(dutot,utot,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      gb = gb + b

c  Compute "constant" macrostrain energy term
c  C is a global value
      C=0.0d0
      do 7200 i=1,2
      do 7200 j=1,2
      do 7200 mi=1,3
      do 7200 mj=1,3
      C=C+0.5d0*u2(i,mi)*zcon(i,mi,j,mj)*u2(j,mj)
7200  continue

      utot=utot+C

c Add in constant term from thermal energy, Y
      utot=utot+Y

c Add in linear term in thermal energy
c
c Add "tutot" from each proc, SUM into "tu"
c then the additional terms of SUM(T2*u2).

      tutot=0.0d0

      do m3=1,3
      do ik=d1,d2
      do ij=1,ny
      do ii=1,nx

      tutot=tutot+T(ii,ij,ik,m3)*u(ii,ij,ik,m3)

      end do; end do; end do; end do

      call MPI_ALLREDUCE(tutot,tu,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      tu = tu + SUM(T2*u2)
```

```
      utot = utot + tu

c  Compute gradient with respect to macrostrains.
c  put in piece from first derivative of zcon quadratic term

      do 7300 i=1,2
      do 7300 mi=1,3
      sum_num=0.0d0
      do 7250 j=1,2
      do 7250 mj=1,3
      sum_num=sum_num+zcon(i,mi,j,mj)*u2(j,mj)
7250  continue
      gb2(i,mi)=sum_num
7300  continue

c  Add in piece of gradient, for displacements as well as macrostrains,
c  that come from linear term in thermal energy

      gb  = gb  + T
      gb2 = gb2 + T2

c  Now generate part that comes from b . u term
c  do by calling b generation with appropriate macrostrain set to 1 to
c  get that partial derivative, just use bgrad (taken from femat),
c  skip dk and zcon part

      do 8100 ii=1,6
      exx=0.0d0
      eyy=0.0d0
      ezz=0.0d0
      exz=0.0d0
      eyz=0.0d0
      exy=0.0d0

      if(ii.eq.1) exx=1.0d0
      if(ii.eq.2) eyy=1.0d0
      if(ii.eq.3) ezz=1.0d0
      if(ii.eq.4) exz=1.0d0
      if(ii.eq.5) eyz=1.0d0
      if(ii.eq.6) exy=1.0d0

      call bgrad_mpi(nx,ny,nz,d1,d2,exx,eyy,ezz,
     &                exz,eyz,exy,dk,b,vox)

      sum_num=0.0d0
      dusum=0.0d0

      do m3=1,3
      do ik=d1,d2
```

```
      do ij=1,ny
      do i2i=1,nx

      dusum=dusum+u(i2i,ij,ik,m3)*b(i2i,ij,ik,m3)

      end do; end do; end do; end do

      call MPI_ALLREDUCE(dusum,sum_num,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      if(ii.eq.1) gb2(1,1)=gb2(1,1)+sum_num
      if(ii.eq.2) gb2(1,2)=gb2(1,2)+sum_num
      if(ii.eq.3) gb2(1,3)=gb2(1,3)+sum_num
      if(ii.eq.4) gb2(2,1)=gb2(2,1)+sum_num
      if(ii.eq.5) gb2(2,2)=gb2(2,2)+sum_num
      if(ii.eq.6) gb2(2,3)=gb2(2,3)+sum_num

8100    continue

        return
        end
c
c************************************************************
c
      subroutine bgrad_mpi(nx,ny,nz,d1,d2,exx,eyy,ezz,
     &                exz,eyz,exy,dk,b,vox)

c  Subroutine that computes derivatives of the b-vector with respect
c  to the macrostrains.  Since b is linear in the macrostrains, the
c  derivative with respect to any one of them can be computed simply
c  by letting that macrostrain, within the subroutine, be equal to one,
c  and all the other macrostrains to be zero.
c  Very similar to 1221 loop in femat for b.

      implicit none
      include 'mpif.h'

      integer nx,ny,nz,d1,d2,dn,nxy,nphase,pflag
      integer i3,i8,i,j,k,m,ii,jj,kk,ipx,ipy,ipz
      integer m3,m8,nn,mm

      double precision  exx,eyy,ezz,exz,eyz,exy,sum_num

      double precision b(nx,ny,d1-1:d2+1,3)
      double precision dk(nphase,8,3,8,3),delta(8,3)

      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision, allocatable :: ab(:,:,:), ba(:,:,:)
```

```
c
c MPI vars....
c
      integer myrank,nprocs,ierr,status(MPI_STATUS_SIZE)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

c  exx, eyy, ezz, exz, eyz, exy are the artificial macrostrains used
c  to get the gradient terms (appropriate combinations of 1's and 0's).

      allocate (ab(nx,ny,3))
      allocate (ba(nx,ny,3))

c  Set up vector for linear term

      b=0.0d0

c
c  x=nx face
c
      do 2001 i3=1,3
      do 2001 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.3.or.i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
2001  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 2000 j=1,ny-1
      do 2000 k=d1,dn
      m=nxy*(k-1)+j*nx
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1900 nn=1,3
      do 1900 mm=1,8
      sum_num=0.0d0

      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
```

```
      do 1899 m3=1,3
      do 1899 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
1899  continue

      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1900  continue
2000  continue

c
c  y=ny face
c
      do 2011 i3=1,3
      do 2011 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.3.or.i8.eq.4.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
2011  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 2010 i=1,nx-1
      do 2010 k=d1,dn
      m=nxy*(k-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nx)

      do 1901 nn=1,3
      do 1901 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
      do 2099 m3=1,3
      do 2099 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2099  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num
1901  continue
2010  continue

c
c  x=nx y=ny edge
c
      do 2031 i3=1,3
      do 2031 i8=1,8
```

```
      delta(i8,i3)=0.0d0
      if(i8.eq.2.or.i8.eq.6) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4.or.i8.eq.8) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.3.or.i8.eq.7) then
      delta(i8,1)=exy*ny+exx*nx
      delta(i8,2)=eyy*ny+exy*nx
      delta(i8,3)=eyz*ny+exz*nx
      end if
2031  continue

      dn=d2
      if (dn.eq.nz) dn=nz-1

      do 2030 k=d1,dn
      m=nxy*k
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1903 nn=1,3
      do 1903 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0d0
      do 2029 m3=1,3
      do 2029 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2029  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1903  continue
2030  continue

      if (myrank.eq.nprocs-1) then

c
c  z=nz face
c
      do 2021 i3=1,3
      do 2021 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6.or.i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exz*nz
```

```
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
2021  continue
      do 2020 i=1,nx-1
      do 2020 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nx)
      do 1902 nn=1,3
      do 1902 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
      do 2019 m3=1,3
      do 2019 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2019  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1902  continue
2020  continue

c
c  x=nx z=nz edge
c
      do 2041 i3=1,3
      do 2041 i8=1,8
      delta(i8,i3)=0.0
      if(i8.eq.2.or.i8.eq.3) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.5.or.i8.eq.8) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.6.or.i8.eq.7) then
      delta(i8,1)=exz*nz+exx*nx
      delta(i8,2)=eyz*nz+exy*nx
      delta(i8,3)=ezz*nz+exz*nx
      end if
2041  continue
      do 2040 j=1,ny-1
      m=nxy*(nz-1)+nx*(j-1)+nx
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1904 nn=1,3
      do 1904 mm=1,8
```

```
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)

      sum_num=0.0d0
      do 2039 m3=1,3
      do 2039 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2039  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num
1904  continue
2040  continue

c
c  y=ny z=nz edge
c
      do 2051 i3=1,3
      do 2051 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.5.or.i8.eq.6) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.3.or.i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.7.or.i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
2051  continue
      do 2050 i=1,nx-1
      m=nxy*(nz-1)+nx*(ny-1)+i
      call m2ijk(m,ii,jj,kk,nx,ny,nz)

      do 1905 nn=1,3
      do 1905 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0

      do 2049 m3=1,3
      do 2049 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2049  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num
1905  continue
2050  continue
```

```
c
c  x=nx y=ny z=nz corner
c
      do 2061 i3=1,3
      do 2061 i8=1,8
      delta(i8,i3)=0.0d0
      if(i8.eq.2) then
      delta(i8,1)=exx*nx
      delta(i8,2)=exy*nx
      delta(i8,3)=exz*nx
      end if
      if(i8.eq.4) then
      delta(i8,1)=exy*ny
      delta(i8,2)=eyy*ny
      delta(i8,3)=eyz*ny
      end if
      if(i8.eq.5) then
      delta(i8,1)=exz*nz
      delta(i8,2)=eyz*nz
      delta(i8,3)=ezz*nz
      end if
      if(i8.eq.8) then
      delta(i8,1)=exy*ny+exz*nz
      delta(i8,2)=eyy*ny+eyz*nz
      delta(i8,3)=eyz*ny+ezz*nz
      end if
      if(i8.eq.6) then
      delta(i8,1)=exx*nx+exz*nz
      delta(i8,2)=exy*nx+eyz*nz
      delta(i8,3)=exz*nx+ezz*nz
      end if
      if(i8.eq.3) then
      delta(i8,1)=exx*nx+exy*ny
      delta(i8,2)=exy*nx+eyy*ny
      delta(i8,3)=exz*nx+eyz*ny
      end if
      if(i8.eq.7) then
      delta(i8,1)=exx*nx+exy*ny+exz*nz
      delta(i8,2)=exy*nx+eyy*ny+eyz*nz
      delta(i8,3)=exz*nx+eyz*ny+ezz*nz
      end if
2061  continue
      m=nx*ny*nz
      call m2ijk(m,ii,jj,kk,nx,ny,nz)
      do 1906 nn=1,3
      do 1906 mm=1,8
      call ipxyz(mm,ii,jj,kk,ipx,ipy,ipz,nx,ny,nz)
      sum_num=0.0d0
```

```
      do 2059 m3=1,3
      do 2059 m8=1,8
      sum_num=sum_num+delta(m8,m3)*dk(vox(ii,jj,kk),m8,m3,mm,nn)
2059  continue
      b(ipx,ipy,ipz,nn) =  b(ipx,ipy,ipz,nn) + sum_num

1906  continue

      end if

      if (nprocs.gt.1) then
c
c RECV a new slice per node.
c
      ab = 0.0d0
      ba = b(:,:,d2+1,:)

      call t2b_dp(ab,ba,nx,ny,3)
      b(:,:,d1,:) = b(:,:,d1,:) + ab
c
c botp = d1-1
c
      ab = 0.0d0
      ba = b(:,:,d1-1,:)

      call b2t_dp(ab,ba,nx,ny,3)
      b(:,:,d2,:) = b(:,:,d2,:) + ab
c
c Update ghost layers
c
c RECV a new slice per node.
c
      ab = b(:,:,d1,:)
      ba = b(:,:,d2,:)

      call t2b_dp(ab,ba,nx,ny,3)

      b(:,:,d1-1,:) = ab

      ab = b(:,:,d1,:)
      ba = b(:,:,d2,:)

      call b2t_dp(ab,ba,nx,ny,3)

      b(:,:,d2+1,:) = ba

      else
c
```

```
c nprocs=1
c
      b(:,:,d1,:) = b(:,:,d1,:) + b(:,:,d2+1,:)
      b(:,:,d2,:) = b(:,:,d2,:) + b(:,:,d1-1,:)
      b(:,:,d2+1,:) = b(:,:,d1,:)
      b(:,:,d1-1,:) = b(:,:,d2,:)

      end if

      deallocate(ab)
      deallocate(ba)

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      end
c
c***********************************************************
c
      subroutine dembx_mpi2(nx,ny,nz,ns,Lstep,gg,gtest,ldemb,kkk,
     &           zcon,u,h,gb,u2,h2,gb2,b,d1,d2,vox,dk)

      implicit none

      include 'mpif.h'

      integer nx,ny,nz,nphase,d1,d2,pflag
      integer nxy,ns,Lstep,kkk,ldemb
      integer i,j,k,m,m1,ijk,ii

      double precision u(nx,ny,d1-1:d2+1,3),gb(nx,ny,d1-1:d2+1,3)
      double precision b(nx,ny,d1-1:d2+1,3)
      double precision h(nx,ny,d1-1:d2+1,3)
      double precision, allocatable ::  Ah(:,:,:,:)

      double precision u2(2,3),gb2(2,3)
      double precision h2(2,3),Ah2(2,3)

      double precision dk(nphase,8,3,8,3),zcon(2,3,2,3)
      double precision e11,e22,e33,e13,e23,e12
      double precision x1,x2,x3,x4,x5,x6

      double precision dgg,gg,gglast,lambda,hAh2,hAh,gamma,gtest
      double precision dAh2,dgb2

      integer*2 vox(nx,ny,d1-1:d2+1)
      double precision, allocatable :: ab(:,:,:), ba(:,:,:)

c
c MPI vars
```

```
c
      integer myrank,nprocs,ierr,status(MPI_STATUS_SIZE)
      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      allocate (ab(nx,ny,3))
      allocate (ba(nx,ny,3))
      allocate (Ah(nx,ny,d1-1:d2+1,3))

      ns = nx*ny*nz

c  Initialize the conjugate direction vector on first call to dembx only.
c  For calls to dembx after the first, we want to continue using the value
c  of h determined in the previous call.  Of course, if npoints
c  is greater than 1, then this initialization step will be run each time
c  a new microstructure is used, as kkk will be reset to 1 every time
c  the counter micro is increased.

      if(kkk.eq.1) then
        h=gb
        h2=gb2
      end if

c  Lstep counts the number of conjugate gradient steps taken
c  in each call to dembx
      Lstep=0

      do 800 ijk=1,ldemb
      Lstep=Lstep+1

      Ah  =0.0d0
      Ah2 =0.0d0

c  Do global matrix multiply via small stiffness matrices, Ah = A * h
c  The long statement below correctly brings in all the terms from
c  the global matrix A using only the small stiffness matrices.

      call gbah(Ah,h,dk,vox,nx,ny,nz,d1,d2)

c  The above accurately gives the second derivative matrix with respect
c  to nodal displacements, but fails to give the 2nd derivative terms that
c  include the macrostrains [ du d(strain) and d(strain)d(strain) ].
c  Use repeated calls to bgrad to generate mixed 2nd derivatives terms,
c  plus use zcon in order to correct the matrix multiply and correctly bring
c  in macrostrain terms (see manual, Sec. 2.4).

      do 8100 ii=1,6
```

```
      e11=0.0d0
      e22=0.0d0
      e33=0.0d0
      e13=0.0d0
      e23=0.0d0
      e12=0.0d0
      if(ii.eq.1) e11=1.0d0
      if(ii.eq.2) e22=1.0d0
      if(ii.eq.3) e33=1.0d0
      if(ii.eq.4) e13=1.0d0
      if(ii.eq.5) e23=1.0d0
      if(ii.eq.6) e12=1.0d0

      call bgrad_mpi(nx,ny,nz,d1,d2,e11,e22,e33,
     &               e13,e23,e12,dk,b,vox)

      do k=d1,d2
      do j=1,ny
      do i=1,nx
      do m1=1,3

      m = nx*ny*(k-1) + ny*(j-1) + i

      if(ii.eq.1) Ah(i,j,k,m1)=Ah(i,j,k,m1)+b(i,j,k,m1)*h2(1,1)
      if(ii.eq.2) Ah(i,j,k,m1)=Ah(i,j,k,m1)+b(i,j,k,m1)*h2(1,2)
      if(ii.eq.3) Ah(i,j,k,m1)=Ah(i,j,k,m1)+b(i,j,k,m1)*h2(1,3)
      if(ii.eq.4) Ah(i,j,k,m1)=Ah(i,j,k,m1)+b(i,j,k,m1)*h2(2,1)
      if(ii.eq.5) Ah(i,j,k,m1)=Ah(i,j,k,m1)+b(i,j,k,m1)*h2(2,2)
      if(ii.eq.6) Ah(i,j,k,m1)=Ah(i,j,k,m1)+b(i,j,k,m1)*h2(2,3)
      end do; end do; end do; end do

c
c UPDATE AH GHOST LAYERS HERE!!!
c

      if (nprocs.gt.1) then

c
c Update ghost layers
c
c RECV a new slice per node.
c
      ab = Ah(:,:,d1,:)
      ba = Ah(:,:,d2,:)

      call t2b_dp(ab,ba,nx,ny,3)

      Ah(:,:,d1-1,:) = ab
```

```
      ab = Ah(:,:,d1,:)
      ba = Ah(:,:,d2,:)

      call b2t_dp(ab,ba,nx,ny,3)

      Ah(:,:,d2+1,:) = ba

      else
c
c nprocs=1
c
      Ah(:,:,d2+1,:) = Ah(:,:,d1,:)
      Ah(:,:,d1-1,:) = Ah(:,:,d2,:)

      end if

      x1=Ah2(1,1)
      x2=Ah2(1,2)
      x3=Ah2(1,3)

      x4=Ah2(2,1)
      x5=Ah2(2,2)
      x6=Ah2(2,3)

      do k=d1,d2
      do j=1,ny
      do i=1,nx

      m=nx*ny*(k-1) + nx*(j-1) + i

      if (ii.eq.1) x1=x1 + b(i,j,k,1)*h(i,j,k,1)+
     &            b(i,j,k,2)*h(i,j,k,2)+b(i,j,k,3)*h(i,j,k,3)
      if (ii.eq.2) x2=x2 + b(i,j,k,1)*h(i,j,k,1)+
     &            b(i,j,k,2)*h(i,j,k,2)+b(i,j,k,3)*h(i,j,k,3)
      if (ii.eq.3) x3=x3 + b(i,j,k,1)*h(i,j,k,1)+
     &            b(i,j,k,2)*h(i,j,k,2)+b(i,j,k,3)*h(i,j,k,3)
      if (ii.eq.4) x4=x4 + b(i,j,k,1)*h(i,j,k,1)+
     &            b(i,j,k,2)*h(i,j,k,2)+b(i,j,k,3)*h(i,j,k,3)
      if (ii.eq.5) x5=x5 + b(i,j,k,1)*h(i,j,k,1)+
     &            b(i,j,k,2)*h(i,j,k,2)+b(i,j,k,3)*h(i,j,k,3)
      if (ii.eq.6) x6=x6 + b(i,j,k,1)*h(i,j,k,1)+
     &            b(i,j,k,2)*h(i,j,k,2)+b(i,j,k,3)*h(i,j,k,3)

      end do; end do; end do


      if(ii.eq.1) call MPI_ALLREDUCE(x1,Ah2(1,1),1,MPI_DOUBLE_PRECISION,
     &                   MPI_SUM,MPI_COMM_WORLD,ierr)
```

```
      if(ii.eq.2) call MPI_ALLREDUCE(x2,Ah2(1,2),1,MPI_DOUBLE_PRECISION,
     &                   MPI_SUM,MPI_COMM_WORLD,ierr)

      if(ii.eq.3) call MPI_ALLREDUCE(x3,Ah2(1,3),1,MPI_DOUBLE_PRECISION,
     &                   MPI_SUM,MPI_COMM_WORLD,ierr)

      if(ii.eq.4) call MPI_ALLREDUCE(x4,Ah2(2,1),1,MPI_DOUBLE_PRECISION,
     &                   MPI_SUM,MPI_COMM_WORLD,ierr)

      if(ii.eq.5) call MPI_ALLREDUCE(x5,Ah2(2,2),1,MPI_DOUBLE_PRECISION,
     &                   MPI_SUM,MPI_COMM_WORLD,ierr)

      if(ii.eq.6) call MPI_ALLREDUCE(x6,Ah2(2,3),1,MPI_DOUBLE_PRECISION,
     &                   MPI_SUM,MPI_COMM_WORLD,ierr)

c  now do righthand corner terms, ns+1 to ns+2
      do 3335 m=1,2
      do 3335 m1=1,3
      if(ii.eq.1) Ah2(1,1)=Ah2(1,1)+zcon(1,1,m,m1)*h2(m,m1)
      if(ii.eq.2) Ah2(1,2)=Ah2(1,2)+zcon(1,2,m,m1)*h2(m,m1)
      if(ii.eq.3) Ah2(1,3)=Ah2(1,3)+zcon(1,3,m,m1)*h2(m,m1)
      if(ii.eq.4) Ah2(2,1)=Ah2(2,1)+zcon(2,1,m,m1)*h2(m,m1)
      if(ii.eq.5) Ah2(2,2)=Ah2(2,2)+zcon(2,2,m,m1)*h2(m,m1)
      if(ii.eq.6) Ah2(2,3)=Ah2(2,3)+zcon(2,3,m,m1)*h2(m,m1)
3335  continue

8100  continue

      hAh = 0.0d0
      hAh2= 0.0d0

      hAh2 = SUM(h(:,:,d1:d2,:)*Ah(:,:,d1:d2,:))

      call MPI_ALLREDUCE(hAh2,hAh,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD, ierr)

      dAh2 = SUM(h2*Ah2)

      hAh = hAh + dAh2

      lambda=gg/hAh

      u = u -lambda*h
      u2= u2-lambda*h2

      gb = gb -lambda*Ah
      gb2= gb2-lambda*Ah2

      gglast=gg
```

```
      gg=0.0d0

      dgg = SUM(gb(:,:,d1:d2,:)*gb(:,:,d1:d2,:))
      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

c
c Now add contributions from gb2 to gg.
c
      dgb2 = SUM(gb2*gb2)
      gg = gg + dgb2

      if(gg.lt.gtest) goto 1000

      gamma=gg/gglast

      h  = gb  + gamma*h
      h2 = gb2 + gamma*h2

800   continue

1000  continue

c
c u2,h2,gb2,lambda
c

      deallocate(ab)
      deallocate(ba)
      deallocate(Ah)

      return
      end
c
c***********************************************************
c

      subroutine stress_mpi2(nx,ny,nz,ns,u,u2,vox,cmod,d1,d2,
     &           strxxp,stryyp,strzzp,strxyp,strxzp,stryzp,
     &           sxxp,syyp,szzp,sxyp,sxzp,syzp,eigen)

      implicit none
      include 'mpif.h'

      integer nx,ny,ns,d1,d2,nxy
      integer ifxa,ifya
      integer pflag,nphase

      double precision u(nx,ny,d1-1:d2+1,3), uu(8,3), u2(2,3)
```

```
      double precision dndx(8),dndy(8),dndz(8)
      double precision es(6,8,3),cmod(nphase,6,6),eigen(nphase,6)
      integer*2 vox(nx,ny,d1-1:d2+1)

      integer myrank, ierr, nprocs, status(MPI_STATUS_SIZE)

      integer nz,i,j,k,m,mm,n3,n8,n

      double precision strxx,stryy,strzz,strxz,stryz,strxy
      double precision str11,str22,str33,str13,str23,str12
      double precision strxxp,stryyp,strzzp
      double precision strxzp,stryzp,strxyp

      double precision s11,s22,s33,s13,s23,s12
      double precision sxx,syy,szz,sxz,syz,sxy
      double precision sxxp,syyp,szzp,sxzp,syzp,sxyp
      double precision exx,eyy,ezz,exz,eyz,exy

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny
      ns = nx*ny*nz

      exx=u2(1,1)
      eyy=u2(1,2)
      ezz=u2(1,3)
      exz=u2(2,1)
      eyz=u2(2,2)
      exy=u2(2,3)

c  set up single element strain matrix
c  dndx, dndy, and dndz are the components of the average strain
c  matrix in a pixel

      dndx(1)=-0.25d0
      dndx(2)=0.25d0
      dndx(3)=0.25d0
      dndx(4)=-0.25d0
      dndx(5)=-0.25d0
      dndx(6)=0.25d0
      dndx(7)=0.25d0
      dndx(8)=-0.25d0
      dndy(1)=-0.25d0
      dndy(2)=-0.25d0
      dndy(3)=0.25d0
      dndy(4)=0.25d0
```

```
      dndy(5)=-0.25d0
      dndy(6)=-0.25d0
      dndy(7)=0.25d0
      dndy(8)=0.25d0
      dndz(1)=-0.25d0
      dndz(2)=-0.25d0
      dndz(3)=-0.25d0
      dndz(4)=-0.25d0
      dndz(5)=0.25d0
      dndz(6)=0.25d0
      dndz(7)=0.25d0
      dndz(8)=0.25d0
c  Build averaged strain matrix, follows code in femat, but for average
c  strain over the pixel, not the strain at a point.

      es = 0.0d0

      es(1,:,1)=dndx
      es(2,:,2)=dndy
      es(3,:,3)=dndz
      es(4,:,1)=dndz
      es(4,:,3)=dndx
      es(5,:,2)=dndz
      es(5,:,3)=dndy
      es(6,:,1)=dndy
      es(6,:,2)=dndx

c  Compute components of the average stress and strain tensors in each pixel
      sxx=0.0d0
      syy=0.0d0
      szz=0.0d0
      sxz=0.0d0
      syz=0.0d0
      sxy=0.0d0
      strxx=0.0d0
      stryy=0.0d0
      strzz=0.0d0
      strxz=0.0d0
      stryz=0.0d0
      strxy=0.0d0

      sxxp=0.0d0
      syyp=0.0d0
      szzp=0.0d0
      sxzp=0.0d0
      syzp=0.0d0
      sxyp=0.0d0
      strxxp=0.0d0
      stryyp=0.0d0
```

```
      strzzp=0.0d0
      strxzp=0.0d0
      stryzp=0.0d0
      strxyp=0.0d0

      do 470 k=d1,d2
      do 470 j=1,ny
      do 470 i=1,nx
      m=(k-1)*nxy+(j-1)*nx+i

      if ((i+1).GT.nx) then
         ifxa = 1
      else
         ifxa = i+1
      end if

      if ((j+1).GT.ny) then
         ifya = 1
      else
         ifya = j+1
      end if

       do mm=1,3
       uu(1,mm)= u(i,j,k,mm)
       uu(2,mm)= u(ifxa,j,k,mm)
       uu(3,mm)= u(ifxa,ifya,k,mm)
       uu(4,mm)= u(i,ifya,k,mm)
       uu(5,mm)= u(i,j,k+1,mm)
       uu(6,mm)= u(ifxa,j,k+1,mm)
       uu(7,mm)= u(ifxa,ifya,k+1,mm)
       uu(8,mm)= u(i,ifya,k+1,mm)
       end do

c  Correct for periodic boundary conditions, some displacements are wrong
c  for a pixel on a periodic boundary.  Since they come from an opposite
c  face, need to put in applied strain to correct them.
      if(i.eq.nx) then
      uu(2,1)=uu(2,1)+exx*nx
      uu(2,2)=uu(2,2)+exy*nx
      uu(2,3)=uu(2,3)+exz*nx
      uu(3,1)=uu(3,1)+exx*nx
      uu(3,2)=uu(3,2)+exy*nx
      uu(3,3)=uu(3,3)+exz*nx
      uu(6,1)=uu(6,1)+exx*nx
      uu(6,2)=uu(6,2)+exy*nx
      uu(6,3)=uu(6,3)+exz*nx
      uu(7,1)=uu(7,1)+exx*nx
      uu(7,2)=uu(7,2)+exy*nx
      uu(7,3)=uu(7,3)+exz*nx
```

```
      end if
      if(j.eq.ny) then
      uu(3,1)=uu(3,1)+exy*ny
      uu(3,2)=uu(3,2)+eyy*ny
      uu(3,3)=uu(3,3)+eyz*ny
      uu(4,1)=uu(4,1)+exy*ny
      uu(4,2)=uu(4,2)+eyy*ny
      uu(4,3)=uu(4,3)+eyz*ny
      uu(7,1)=uu(7,1)+exy*ny
      uu(7,2)=uu(7,2)+eyy*ny
      uu(7,3)=uu(7,3)+eyz*ny
      uu(8,1)=uu(8,1)+exy*ny
      uu(8,2)=uu(8,2)+eyy*ny
      uu(8,3)=uu(8,3)+eyz*ny
      end if
      if(k.eq.nz) then
      uu(5,1)=uu(5,1)+exz*nz
      uu(5,2)=uu(5,2)+eyz*nz
      uu(5,3)=uu(5,3)+ezz*nz
      uu(6,1)=uu(6,1)+exz*nz
      uu(6,2)=uu(6,2)+eyz*nz
      uu(6,3)=uu(6,3)+ezz*nz
      uu(7,1)=uu(7,1)+exz*nz
      uu(7,2)=uu(7,2)+eyz*nz
      uu(7,3)=uu(7,3)+ezz*nz
      uu(8,1)=uu(8,1)+exz*nz
      uu(8,2)=uu(8,2)+eyz*nz
      uu(8,3)=uu(8,3)+ezz*nz
      end if

c  local stresses and strains in a pixel
      str11=0.0d0
      str22=0.0d0
      str33=0.0d0
      str13=0.0d0
      str23=0.0d0
      str12=0.0d0
      s11=0.0d0
      s22=0.0d0
      s33=0.0d0
      s13=0.0d0
      s23=0.0d0
      s12=0.0d0
c********compute average stress and strain tensor in each pixel*************
c  First put thermal strain-induced stresses into stress tensor
      do 465 n=1,6
      str11=str11-cmod(vox(i,j,k),1,n)*eigen(vox(i,j,k),n)
      str22=str22-cmod(vox(i,j,k),2,n)*eigen(vox(i,j,k),n)
      str33=str33-cmod(vox(i,j,k),3,n)*eigen(vox(i,j,k),n)
```

```
      str13=str13-cmod(vox(i,j,k),4,n)*eigen(vox(i,j,k),n)
      str23=str23-cmod(vox(i,j,k),5,n)*eigen(vox(i,j,k),n)
      str12=str12-cmod(vox(i,j,k),6,n)*eigen(vox(i,j,k),n)
465   continue
      do 466 n3=1,3
      do 466 n8=1,8
c  compute non-thermal strains in each pixel
      s11=s11+es(1,n8,n3)*uu(n8,n3)
      s22=s22+es(2,n8,n3)*uu(n8,n3)
      s33=s33+es(3,n8,n3)*uu(n8,n3)
      s13=s13+es(4,n8,n3)*uu(n8,n3)
      s23=s23+es(5,n8,n3)*uu(n8,n3)
      s12=s12+es(6,n8,n3)*uu(n8,n3)
      do 466 n=1,6
c  compute stresses in each pixel that include both non-thermal
c  and thermal strains
      str11=str11+cmod(vox(i,j,k),1,n)*es(n,n8,n3)*uu(n8,n3)
      str22=str22+cmod(vox(i,j,k),2,n)*es(n,n8,n3)*uu(n8,n3)
      str33=str33+cmod(vox(i,j,k),3,n)*es(n,n8,n3)*uu(n8,n3)
      str13=str13+cmod(vox(i,j,k),4,n)*es(n,n8,n3)*uu(n8,n3)
      str23=str23+cmod(vox(i,j,k),5,n)*es(n,n8,n3)*uu(n8,n3)
      str12=str12+cmod(vox(i,j,k),6,n)*es(n,n8,n3)*uu(n8,n3)
466   continue


c  sum local strains and stresses into global values
      strxx=strxx+str11
      stryy=stryy+str22
      strzz=strzz+str33
      strxz=strxz+str13
      stryz=stryz+str23
      strxy=strxy+str12
      sxx=sxx+s11
      syy=syy+s22
      szz=szz+s33
      sxz=sxz+s13
      syz=syz+s23
      sxy=sxy+s12
470   continue

c
c Now do MPI to gather all strNN and sNN terms,
c add them at root, then do this final calculation
c and write them to disk.
c

      call MPI_ALLREDUCE(strxx,strxxp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)
```

```
      call MPI_ALLREDUCE(stryy,stryyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(strzz,strzzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(strxz,strxzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(strxy,strxyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(stryz,stryzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)


      call MPI_ALLREDUCE(sxx,sxxp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(syy,syyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(szz,szzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(sxz,sxzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(sxy,sxyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(syz,syzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

c
c Now root has strxx,stryy, ... sxy
c Let him write out data to disk after this Volume averaging
c
c  Volume average of global stresses and strains
      strxxp=strxxp/dfloat(ns)
      stryyp=stryyp/dfloat(ns)
      strzzp=strzzp/dfloat(ns)
      strxzp=strxzp/dfloat(ns)
      stryzp=stryzp/dfloat(ns)
      strxyp=strxyp/dfloat(ns)
      sxxp=sxxp/dfloat(ns)
      syyp=syyp/dfloat(ns)
      szzp=szzp/dfloat(ns)
      sxzp=sxzp/dfloat(ns)
```

```
      syzp=syzp/dfloat(ns)
      sxyp=sxyp/dfloat(ns)


      if (myrank.eq.0) then

      write(*,*) "strxxp = ", strxxp
      write(*,*) "stryyp = ", stryyp
      write(*,*) "strzzp = ", strzzp
      write(*,*) "strxyp = ", strxyp
      write(*,*) "strxzp = ", strxzp
      write(*,*) "stryzp = ", stryzp

      write(*,*) "sxxp = ", sxxp
      write(*,*) "syyp = ", syyp
      write(*,*) "szzp = ", szzp
      write(*,*) "sxyp = ", sxyp
      write(*,*) "sxzp = ", sxzp
      write(*,*) "syzp = ", syzp

      end if

      return
      end
c
c***********************************************************
c
      subroutine gbah(om,uh,dk,vox,nx,ny,nz,d1,d2)

      implicit none
      include 'mpif.h'

      integer nx,ny,nz,d1,d2,pflag,nphase
      integer im,jm,km,j,ifxa,ifxb,ifya,ifyb
      integer myrank,nprocs,ierr

      double precision uh(nx,ny,d1-1:d2+1,3)
      double precision om(nx,ny,d1-1:d2+1,3)
      double precision gb_time(6)

      integer*2 vox(nx,ny,d1-1:d2+1)

      double precision dk(nphase,8,3,8,3)

      common/list1/pflag,nphase

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )
```

```
      gb_time(1) = MPI_Wtime(ierr)

      om = 0.0d0

      do km=d1,d2
      do jm=1,ny
      do im=1,nx

      if ((im+1).GT.nx) then
         ifxa = 1
      else
         ifxa = im+1
      end if

      if ((im-1).LE.0) then
         ifxb = nx
      else
         ifxb = im-1
      end if

      if ((jm+1).GT.ny) then
         ifya = 1
      else
         ifya = jm+1
      end if

      if ((jm-1).LE.0) then
         ifyb = ny
      else
         ifyb = jm-1
      end if

      do j=1,3

c SELF TERM

      om(im,jm,km,j) =

c u(ib(m,1),n)
     & SUM ( uh(im,ifya,km,:)*(
     & dk(vox(im,jm,km),1,j,4,:)
     &+dk(vox(ifxb,jm,km),2,j,3,:)
     &+dk(vox(im,jm,km-1),5,j,8,:)
     &+dk(vox(ifxb,jm,km-1),6,j,7,:) ))+

c u(ib(m,2),n)
     & SUM ( uh(ifxa,ifya,km,:)*(dk(vox(im,jm,km),1,j,3,:)
     &+  dk(vox(im,jm,km-1),5,j,7,:) )) +
```

```
c u(ib(m,3),n)
     & SUM ( uh(ifxa,jm,km,:)*(dk(vox(im,jm,km),1,j,2,:)
     &+ dk(vox(im,ifyb,km),4,j,3,:)
     &+ dk(vox(im,ifyb,km-1),8,j,7,:)
     &+ dk(vox(im,jm,km-1),5,j,6,:) )) +

c u(ib(m,4),n)
     & SUM ( uh(ifxa,ifyb,km,:)*(dk(vox(im,ifyb,km),4,j,2,:)
     &+  dk(vox(im,ifyb,km-1),8,j,6,:) )) +

c u(ib(m,5),n)
     &  SUM ( uh(im,ifyb,km,:)*(dk(vox(ifxb,ifyb,km),3,j,2,:)
     & +dk(vox(im,ifyb,km),4,j,1,:)
     & +dk(vox(ifxb,ifyb,km-1),7,j,6,:)
     & +dk(vox(im,ifyb,km-1),8,j,5,:) )) +

c u(ib(m,6),n)
     & SUM ( uh(ifxb,ifyb,km,:)*(dk(vox(ifxb,ifyb,km),3,j,1,:)
     &+  dk(vox(ifxb,ifyb,km-1),7,j,5,:) )) +

c u(ib(m,7),n)
     & SUM(uh(ifxb,jm,km,:)*(
     & dk(vox(ifxb,ifyb,km),3,j,4,:)
     &+dk(vox(ifxb,jm,km),2,j,1,:)
     &+dk(vox(ifxb,ifyb,km-1),7,j,8,:)
     &+dk(vox(ifxb,jm,km-1),6,j,5,:) )) +

c u(ib(m,8),n)
     & SUM (uh(ifxb,ifya,km,:)*( dk(vox(ifxb,jm,km),2,j,4,:)
     &+dk(vox(ifxb,jm,km-1),6,j,8,:) )) +

c u(ib(m,9),n)
     & SUM ( uh(im,ifya,km-1,:)*(dk(vox(im,jm,km-1),5,j,4,:)
     &+  dk(vox(ifxb,jm,km-1),6,j,3,:) )) +

c u(ib(m,10),n)
     & SUM ( uh(ifxa,ifya,km-1,:)*(dk(vox(im,jm,km-1),5,j,3,:) ))+

c u(ib(m,11),n)
     & SUM ( uh(ifxa,jm,km-1,:)*(dk(vox(im,ifyb,km-1),8,j,3,:)
     &+  dk(vox(im,jm,km-1),5,j,2,:) )) +

c u(ib(m,12),n)
     &  SUM( uh(ifxa,ifyb,km-1,:)*( dk(vox(im,ifyb,km-1),8,j,2,:) ))+

c u(ib(m,13),n)
     & SUM ( uh(im,ifyb,km-1,:)*(dk(vox(im,ifyb,km-1),8,j,1,:)
     &+  dk(vox(ifxb,ifyb,km-1),7,j,2,:) )) +
```

```
c u(ib(m,14),n)
     & SUM( uh(ifxb,ifyb,km-1,:)*( dk(vox(ifxb,ifyb,km-1),7,j,1,:) ))+

c u(ib(m,15),n)
     & SUM ( uh(ifxb,jm,km-1,:)*(dk(vox(ifxb,ifyb,km-1),7,j,4,:)
     &+  dk(vox(ifxb,jm,km-1),6,j,1,:) ))+

c u(ib(m,16),n)
     &SUM(uh(ifxb,ifya,km-1,:)*( dk(vox(ifxb,jm,km-1),6,j,4,:) ))+

c u(ib(m,17),n)
     & SUM ( uh(im,ifya,km+1,:)*(dk(vox(im,jm,km),1,j,8,:)
     &+  dk(vox(ifxb,jm,km),2,j,7,:) ))+

c u(ib(m,18),n)
     & SUM (uh(ifxa,ifya,km+1,:)*( dk(vox(im,jm,km),1,j,7,:) ))+

c u(ib(m,19),n)
     & SUM ( uh(ifxa,jm,km+1,:)*(dk(vox(im,jm,km),1,j,6,:)
     &+ dk(vox(im,ifyb,km),4,j,7,:) )) +

c u(ib(m,20),n)
     & SUM (uh(ifxa,ifyb,km+1,:)*( dk(vox(im,ifyb,km),4,j,6,:) ))+

c u(ib(m,21),n)
     & SUM ( uh(im,ifyb,km+1,:)*(dk(vox(im,ifyb,km),4,j,5,:)
     &+ dk(vox(ifxb,ifyb,km),3,j,6,:) )) +

c u(ib(m,22),n)
     & SUM(uh(ifxb,ifyb,km+1,:)*( dk(vox(ifxb,ifyb,km),3,j,5,:) ))+

c u(ib(m,23),n)
     & SUM ( uh(ifxb,jm,km+1,:)*(dk(vox(ifxb,ifyb,km),3,j,8,:)
     &+ dk(vox(ifxb,jm,km),2,j,5,:) )) +

c u(ib(m,24),n)
     & SUM(uh(ifxb,ifya,km+1,:)*( dk(vox(ifxb,jm,km),2,j,8,:) ))+

c u(ib(m,25),n)
     & SUM ( uh(im,jm,km-1,:)*(dk(vox(ifxb,ifyb,km-1),7,j,3,:)
     &+ dk(vox(im,ifyb,km-1),8,j,4,:)
     &+ dk(vox(ifxb,jm,km-1),6,j,2,:)
     &+ dk(vox(im,jm,km-1),5,j,1,:) )) +

c u(ib(m,26),n)
     & SUM(uh(im,jm,km+1,:)*(
     & dk(vox(ifxb,ifyb,km),3,j,7,:)
     &+dk(vox(im,ifyb,km),4,j,8,:)
     &+dk(vox(im,jm,km),1,j,5,:)
```

```
     &+dk(vox(ifxb,jm,km),2,j,6,:) )) +

c u(ib(m,27),n)
     & SUM( uh(im,jm,km,:)* (dk(vox(im,jm,km),1,j,1,:)
     &+  dk(vox(ifxb,jm,km),2,j,2,:)
     &+  dk(vox(ifxb,ifyb,km),3,j,3,:)
     &+  dk(vox(im,ifyb,km),4,j,4,:)
     &+  dk(vox(im,jm,km-1),5,j,5,:)
     &+  dk(vox(ifxb,jm,km-1),6,j,6,:)
     &+  dk(vox(ifxb,ifyb,km-1),7,j,7,:)
     &+  dk(vox(im,ifyb,km-1),8,j,8,:) ))

      end do

      end do; end do; end do

      gb_time(2) = MPI_Wtime(ierr)

c
c Do top/bottom layer switch on matrix: om
c

      call z_ghost_dp(om,nx,ny,3,d1,d2)

      if (pflag.eq.1) then
      write(*,*)myrank, "Etime for t2b gb/Ah=",gb_time(4)-gb_time(3)
      write(*,*)myrank, "Etime for b2t gb/Ah=",gb_time(6)-gb_time(5)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*********************************************************
c
      subroutine dpixel(nx,ny,nz,ns,pix)
      implicit none

      integer nx,ny,nz,ns,nphase,nxy
      integer i,j,k,m,pflag
      integer*2 pix(nx,ny,nz)
      integer*2 pix0

      common/list1/pflag,nphase

c  (USER)  If you want to set up a test image inside the program, instead of
c  reading it in from a file, this should be done inside this subroutine.
```

```
      nxy=nx*ny
      do 200 k=1,nz
      do 200 j=1,ny
      do 200 i=1,nx
      m=nxy*(k-1)+nx*(j-1)+i
      read(9,*) pix(i,j,k)

200   continue

       do k=1,nz
       do j=1,ny
       do i=1,nx

       pix0 =  pix(i,j,k)

       if(pix0.lt.1) then
        write(7,*) "Phase label in pix < 1--error at ",i,j,k
       end if
       if(pix0.gt.nphase) then
        write(7,*) "Phase label in pix > nphase--error at ",i,j,k
       end if

       end do; end do; end do

      return
      end
c
c***********************************************************
c
      subroutine dassig(nx,ny,nz,prob,pix)
      implicit none

      integer nx,ny,nz,ns,nphase,ii,jj,kk,i,pflag

      integer*2 pix(nx,ny,nz)
      double precision prob(nphase)

      common/list1/pflag,nphase

        ns=nx*ny*nz
        prob=0.0d0

        do kk=1,nz
        do jj=1,ny
        do ii=1,nx
        do i=1,nphase
           if(pix(ii,jj,kk).eq.i) then
              prob(i)=prob(i)+1.0d0
           end if
```

```
        end do; end do
        end do; end do

        prob=prob/dfloat(ns)

        return
        end
c
c*****************************************************************
c
      subroutine ipxyz(mm,i,j,k,ipx,ipy,ipz,nx,ny,nz)

      implicit none

      integer mm,i,j,k,ipx,ipy,ipz,nx,ny,nz

      if (mm.le.4) then
         ipz=k
      else
         ipz=k+1
      end if

      if ((mm.eq.1).OR.(mm.eq.5)) then
         ipx=i
         ipy=j
      end if

      if ((mm.eq.2).OR.(mm.eq.6)) then
         ipx = i+1
         ipy=j

         if (i.ge.nx) then
            ipx=1
         end if

      end if

      if ((mm.eq.3).OR.(mm.eq.7)) then
         ipx = i+1
         if (i.ge.nx) then
            ipx=1
         end if
         ipy = j+1
         if (j.ge.ny) then
            ipy=1
         end if
      end if

      if ((mm.eq.4).OR.(mm.eq.8)) then
```

```
         ipx = i
         ipy = j+1

         if (j.ge.ny) then
            ipy=1
         end if

      end if

      return
      end
c
c************************************************************
c
      subroutine m2ijk(inps,i,j,k,ni,nj,nk)

      implicit none

      integer inps,ns
      integer c
      integer kdiv,jdiv
      integer rj,rk
      integer i,j,k,ni,nj,nk

      ns=ni*nj
      kdiv=inps/ns
      c = ns*kdiv
      rk = inps-c

      if (rk.eq.0) then
         k=kdiv
         j=nj
         i=ni
      else
         k=kdiv+1
      end if

      if (k.ne.kdiv) then

         jdiv=rk/ni
         c=jdiv*ni
         rj = rk-c

         if (rj.eq.0) then
            j=jdiv
            i=ni
         else
            j=jdiv+1
            i=rj
```

```
         end if

      end if

      return
      end
c
c*****************************************************
c
      subroutine z_ghost_int(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      integer*2 arr0(mx,my,d1-1:d2+1)
      integer*2, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call t2b(bot,top,mx,my)

      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call b2t(bot,top,mx,my)

      arr0(:,:,d2+1) = top

      deallocate(bot)
      deallocate(top)
```

```
      return
      end
c
c*********************************************************
c
      subroutine z_ghost_dp(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      double precision arr0(mx,my,d1-1:d2+1,mz)

      double precision, allocatable :: bot(:,:,:), top(:,:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      allocate(bot(mx,my,mz))
      allocate(top(mx,my,mz))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1,:)
      top = arr0(:,:,d2,:)
      call t2b_dp(bot,top,mx,my,3)
      arr0(:,:,d1-1,:) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1,:)
      top = arr0(:,:,d2,:)
      call b2t_dp(bot,top,mx,my,3)
      arr0(:,:,d2+1,:) = top
      deallocate(bot)
      deallocate(top)

      return
      end
c
c*********************************************************
c
      subroutine t2b(b_layer,t_layer,nx,ny)
```

```
c
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
c
      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(b_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*****************************************************************
c
      subroutine b2t(b_layer,t_layer,nx,ny)
c
```

```
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
c
      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(t_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c***********************************************************
c
      subroutine t2b_dp(b_layer,t_layer,nx,ny,i)
```

```
c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om top2bottom layers
c
c RECV a new b_layer (BOTTOM layer) per node.
c
c
      implicit none

      include 'mpif.h'

      integer nx,ny,mxy,i
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)
      double precision b_layer(nx,ny,i), t_layer(nx,ny,i)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=i*nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_irecv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*****************************************************
c
      subroutine b2t_dp(b_layer,t_layer,nx,ny,i)
```

```
c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
c
      implicit none

      include 'mpif.h'

      integer nx,ny,mxy,i
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      double precision b_layer(nx,ny,i), t_layer(nx,ny,i)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=i*nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_Irecv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
```

6.2 Finite Difference

6.2.1 DC3D_MPI.f

```
c  ************************  DC3D_MPI.f  *******************************
c
c  This is the new MPI version of the DC3D.f code from
c  Section 9.3.4 of NISTIR 6269.
c
c  The main differences with this code compared to the serial
c  version are:
c
c  1. Removal of list array.
c  2. Change of dimensionality on pix from pix(m) to pix(i,j,k)
c  3. All important arrays (pix,vox,gx,gy,gz,gb,b,u,h,ah) are dynamically allocated.
c
c  IN THIS VERSION:
c
c  The USER needs the following input:
c  (Search for occurences of "USER" in the code).
c
c 1. A 3-D pixel value data file with input & output names.
c 2. The values of the 3 dimensions: (nx,ny,nz)
c 3. The number of phases in the mixture: nphase
c 4. A convergence value: gtest
c 5. Components of applied field: ex,ey,ez
c 6. Values for:
c    ncgsteps =>  total number of conjugate gradient steps.
c    ncheck => forces dembx to write out the total current and the norm of
c              the gradient squared, every ncheck conjugate gradient steps
c
c 7. Flag for printing timing info for MPI routines
c    from MAIN is called: pflag
c    pflag Values = 0,1 0=no timing info; 1=print timing info
c
c    pflag is a common value.
c
c    User may edit the code to supress the printing.
c
c END of NEW comments.
c
c BEGIN ORIGINAL comments.
c
c  BACKGROUND
c  This program accepts as input a 3-d digital image, converting it
c  into a real conductor network. The conjugate gradient method
c  is used to solve this finite difference representation of Laplace's
c  equation for real conductivity problems.
c  Periodic boundary conditions are maintained.
c  In the comments below, (USER) means that this is a section of code
```

```
c  that the user might have to change for his particular problem.
c  Therefore the user is encouraged to search for this string.
c
c  PROBLEM AND VARIABLE DEFINITION
c
c  The mathematical problem that the conjugate gradient algorithm solves
c  is the minimization of the quadratic form 1/2 uAu, where
c  u is the vector of voltages, and A is generated from the bond
c  conductances between pixels. Nodes are thought of as being in the
c  center of pixels. The minimization is constrained by maintaining an
c  general direction applied electric field across the sample.
c  The vectors gx,gy,gz are bond conductances, u is the voltage array,
c  and gb,h, and Ah are auxiliary variables, used in subroutine dembx.
c  The vector pix contains the phase labels for each pixel.
c  The small vector a(i) is the volume fraction
c  of the i'th phase, and currx, curry, currz are the total volume-averaged
c  currents in the x,y, and z directions.
c
c  DIMENSIONS
c
c  The vectors gx,gy,gz,u,gb,h,Ah,list,pix are all dimensioned
c  ns2 = (nx+2)*(ny+2)*(nz+2).  This number is used, rather than the
c  system size nx x ny x nz, because an extra layer of pixels is
c  put around the system to be able to maintain periodic boundary
c  conditions (see manual, Sec. 3.3). The arrays pix and list are also
c  dimensioned this way.
c  At present the program is set up for up to 100
c  phases, but that can easily be changed by the user, by changing the
c  dimension of sigma, a, and be. Note that be has both dimensions
c  equal to each other. The parameter nphase gives the number of
c  phases being considered. The parameter ntot is the total number
c  of phases possible in the program, and should be equal to the
c  dimension of sigma, a, and be.
c  All arrays are passed to subroutines in the call statements.
c
c  STRONGLY RECOMMENDED:  READ MANUAL BEFORE USING THE PROGRAM!!
c
      implicit none
      include 'mpif.h'

      integer*2, allocatable :: dat(:,:,:), datn(:,:,:)
      integer*2, allocatable :: pix(:,:,:), pixn(:,:,:)
      integer*2, allocatable :: vox(:,:,:)

      integer, allocatable :: d1s(:),d2s(:)

      double precision, allocatable :: gx(:,:,:)
      double precision, allocatable :: gy(:,:,:)
      double precision, allocatable :: gz(:,:,:)
```

```
      double precision, allocatable :: gb(:,:,:)
      double precision, allocatable :: h(:,:,:)
      double precision, allocatable :: ah(:,:,:)
      double precision, allocatable :: u(:,:,:)
      double precision, allocatable :: sigma(:,:)
      double precision, allocatable :: a(:), be(:,:,:)

      double precision ex,ey,ez

      integer nx,nx1,nx2,ny,ny1,ny2,nz,nz1,nz2,L22,ns,ns2
      integer i,j,k,d1,d2,sized,sxip
      integer nphase,ntot,ic

      integer myrank, ierr, nprocs, irank, sz, rem
      integer status(MPI_STATUS_SIZE)

      double precision cuxxp,cuyyp,cuzzp,gtest

      call MPI_INIT(ierr)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      if (myrank.eq.0) then
         write(*,*) "There are ",nprocs," processors running this job."

c  (USER) Unit 9 is the microstructure input file, unit 7 is the
c  results output file.

         open(unit=9,file='test8k.dat')
         open(unit=7,file='pix_dat.out')

      end if

c  (USER) double precision image size is nx x ny x nz
      nx=20
      ny=20
      nz=20

      nx2=nx+2
      ny2=ny+2
      nz2=nz+2
      L22=nx2*ny2

c Variables changed here to reflect the fact our
c calculations are now based on "NZ2" instead of "NZ"
c as in the previous MPI programs.
c
      ns=nx2*ny2*nz2
```

```
      sz=nz2/nprocs

      if (myrank.eq.0) then
         allocate (pix(nx2,ny2,nz2))
         write(7,1111) nx,ny,nz,nx*ny*nz
      end if

1111  format(' Image is ',3i6,' No. of double precision sites = ',i8)

c  auxiliary variables involving the system size
      nx1=nx+1
      ny1=ny+1
      nz1=nz+1

c  computational image size ns2 is nx2 x ny2 x nz2
      ns2=nx2*ny2*nz2

c  (USER) set cutoff for norm squared of gradient, gtest.  gtest is
c  the stopping criterion, compared to gb*gb. When gb*gb is less
c  than gtest=abc*ns2, then the rms value of the gradient at a pixel
c  is less than sqrt(abc).
      gtest=1.0d-16*ns2

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase. ntot is the total
c  number of phases possible in the program, and is the dimension of
c  sigma, a, and be.

      nphase=2
      ntot=100

      allocate(a(ntot))
      allocate(sigma(nphase,3))

c Program calculates the d1 & d2 limits for the principle arrays.
c
c NB: In this program, d1 & d2 are not based on "nz".
c     They are calculated on the basis of "nz2".
c
      if (myrank.eq.0) then

      allocate (d1s(0:nprocs-1))
      allocate (d2s(0:nprocs-1))

        do irank=0,nprocs-1
          d1s(irank)=irank*sz+1
          d2s(irank)=(irank+1)*sz
        end do
```

```
        rem = nz2 - nprocs*sz

        if (rem.ne.0) then
           do j=1,rem
             irank=nprocs-rem+j-1
             d1s(irank)=d1s(irank)+ j-1
             d2s(irank)=d2s(irank)+ j
           end do
        end if

c Send all d1s(i) and d2s(i) from ROOT
c to NODE i & store into d1 & d2 on worker.

        do i=0,nprocs-1
      call MPI_SEND(d1s(i),1,MPI_INTEGER,i,0,MPI_COMM_WORLD,ierr)
      call MPI_SEND(d2s(i),1,MPI_INTEGER,i,1,MPI_COMM_WORLD,ierr)
        end do

      end if

      call MPI_RECV(d1,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,status,ierr)
      call MPI_RECV(d2,1,MPI_INTEGER,0,1,MPI_COMM_WORLD,status,ierr)
      write(*,*) "Rank#",myrank,"d1= ",d1," d2= ",d2

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c Read in initial PIX data.
c Pass layers of PIX data to each node.
c Make Z-Ghost layers.
c
      if (myrank.eq.0) then
         call dpixel_dc(pix,nx2,ny2,nz2,a,nphase,ntot)

         do i=1,nphase
            write(7,299) i,a(i)
         end do

299   format(' Phase fraction of ',i3,' = ',f12.6)

      end if

c Layer passing based on d1 & d2.

c
c Now that the nodes are set up correctly,
c one can pass the data from the root node
c (myrank=0) to all the rest.
c
```

```
      allocate(dat(nx2,ny2,d1:d2))
      sized = SIZE(dat)

      if (nprocs.eq.1) then
         dat=pix
         write(*,*) "dat=pix"
      end if

      if (nprocs.gt.1) then

         if (myrank.eq.0) then

             dat(:,:,d1:d2)=pix(:,:,d1:d2)

             do i=1,nprocs-1
               allocate (pixn(nx2,ny2,d1s(i):d2s(i)))
               pixn = pix(:,:,d1s(i):d2s(i))
               sxip = SIZE(pixn)
               call MPI_SEND(pixn,2*sxip,MPI_BYTE,
     &              i,7,MPI_COMM_WORLD,status,ierr)
               deallocate(pixn)
            end do

         else

            allocate(datn(nx2,ny2,d1:d2))
            call MPI_RECV(datn,2*sized,MPI_BYTE,0,7
     &             ,MPI_COMM_WORLD,status,ierr)

            dat(:,:,d1:d2) = datn
            deallocate(datn)
         end if

      end if

c
c At this point, all the nodes have the initial correct data.
c
c The data on the nodes is dimensioned:  dat(nx,ny,d1:d2)
c
c Need an array vox, st: vox(nx2,ny2,d1-1:d2+1)
c
c Allocate vox & initialize.
c
      allocate(vox(nx2,ny2,d1-1:d2+1))
      vox = 0
c
c Make the copy
c
```

```
      do k=d1,d2
         vox(:,:,k) = dat(:,:,k)
      end do
      deallocate(dat)

c
c Call z_ghost_int to make Z ghost layers of INTEGER*2 values (aka vox).
c
      call z_ghost_int(vox,nx2,ny2,nz2,d1,d2)

c  (USER) Set components of applied field, E = (ex,ey,ez)

      ex=1.0d0
      ey=1.0d0
      ez=1.0d0

      if (myrank.eq.0) then

         write(7,*) 'Applied field components:'
         write(7,*) 'ex = ',ex,' ey = ',ey,' ez = ',ez

      end if

c  Initialize the voltage distribution by putting on uniform field.

      allocate(u(nx2,ny2,d1-1:d2+1))
      u = 0.0d0

      do 30 k=d1,d2
      do 30 j=1,ny2
      do 30 i=1,nx2
      u(i,j,k)=-ex*i-ey*j-ez*k
30    continue

c
c Call z_ghost_dp to make Z ghost layers
c of DOUBLE PRECISION values (aka u).
c
      call z_ghost_dp(u,nx2,ny2,nz2,d1,d2)

c  (USER) input value of double precision conductivity tensor for
c  each phase (diagonal only). 1,2,3 = x,y,z, respectively.

      sigma(1,1)=1.0d0
      sigma(1,2)=1.0d0
      sigma(1,3)=1.0d0
      sigma(2,1)=0.5d0
      sigma(2,2)=0.5d0
      sigma(2,3)=0.5d0
```

```
      allocate (be(nphase,nphase,3))

      allocate (gx(nx2,ny2,d1-1:d2+1))
      allocate (gy(nx2,ny2,d1-1:d2+1))
      allocate (gz(nx2,ny2,d1-1:d2+1))
      allocate (gb(nx2,ny2,d1-1:d2+1))

      allocate (h(nx2,ny2,d1-1:d2+1))
      allocate (Ah(nx2,ny2,d1-1:d2+1))

      gx=0.0d0; gy=0.0d0; gz=0.0d0; be=0.0d0
      gb=0.0d0;  h=0.0d0; Ah=0.0d0

c  Subroutine bond sets up conductor network in gx,gy,gz 1-d arrays

      call bond_dc(vox,gx,gy,gz,nx2,ny2,nz2,ns2,
     &              sigma,be,nphase,ntot,d1,d2)

      call dembx_dc(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,Ah,
     &  gtest,d1,d2)

      call current_dc(nx2,ny2,nz2,ns2,cuxxp,cuyyp,cuzzp,
     &                      u,gx,gy,gz,d1,d2)

      if (myrank.eq.0) then
         write(7,*) 'Average current in x direction= ',cuxxp
         write(7,*) 'Average current in y direction= ',cuyyp
         write(7,*) 'Average current in z direction= ',cuzzp
         write(7,*) ic,' number of conjugate gradient cycles needed'
         call flush(7)
      end if

      CALL MPI_FINALIZE(ierr)

      end
c
c************************************************************
c
      subroutine dembx_dc(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,Ah,
     &  gtest,d1,d2)

      implicit none

      include 'mpif.h'

      integer nx2,ny2,nz2,ns2,d1,d2,L22
      integer nx1,ny1,nz1
```

```
      double precision gx(nx2,ny2,d1-1:d2+1)
      double precision gy(nx2,ny2,d1-1:d2+1)
      double precision gz(nx2,ny2,d1-1:d2+1)

      double precision u(nx2,ny2,d1-1:d2+1)
      double precision gb(nx2,ny2,d1-1:d2+1)

      double precision Ah(nx2,ny2,d1-1:d2+1)
      double precision h(nx2,ny2,d1-1:d2+1)

      double precision gg,hAh,lambda,gglast,gamma
      double precision gtest,cuxxp,cuyyp,cuzzp

      double precision dgg,hAh2

      integer ncheck,icc,ncgsteps,ic

      integer nx,ny,nz,dlow,dhigh

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c  Note:  voltage gradients are maintained because in the conjugate gradient
c  relaxation algorithm, the voltage vector is only modified by adding a
c  periodic vector to it.

      L22=nx2*ny2

      nx1= nx2-1; ny1= ny2-1; nz1= nz2-1
      nx = nx2-2; ny = ny2-2;  nz= nz2-2

      if (nprocs.gt.1) then

          if (myrank.eq.0) then
             dlow = 2
          else
             dlow = d1
          end if

          if (myrank.eq.nprocs-1) then
            dhigh = nz1
          else
            dhigh = d2
          end if

      end if
```

```
      if (nprocs.eq.1) then
         dlow=2
         dhigh=nz1
      end if

c  First stage, compute initial value of gradient (gb), initialize h, the
c  conjugate gradient direction, and compute norm squared of gradient vector.

      call prod_dc(nx2,ny2,nz2,ns2,gx,gy,gz,u,gb,d1,d2)

      h=gb

c  Variable gg is the norm squared of the gradient vector
      gg=0.0d0

      dgg= 0.0d0
      gg = 0.0d0

      dgg = SUM(gb(2:nx1,2:ny1,dlow:dhigh)*gb(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

c  Second stage, initialize Ah variable, compute parameter lamdba,
c  make first change in voltage array, update gradient (gb) vector

      if(gg.lt.gtest) goto 44

      call prod_dc(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah,d1,d2)

      hAh = 0.0d0
      hAh2= 0.0d0

      hAh2 = SUM(h(2:nx1,2:ny1,dlow:dhigh)*Ah(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(hAh2,hAh,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     &  MPI_COMM_WORLD, ierr)

      lambda=gg/hAh

      u=u-lambda*h
      gb=gb-lambda*Ah

c  third stage:  iterate conjugate gradient solution process until
c  gg < gtest criterion is satisfied.
c  (USER) The parameter ncgsteps is the total number of conjugate gradient steps
c  to go through.  Only in very unusual problems, like when the conductivity
c  of one phase is much higher than all the rest, will this many steps be
```

```
c  used.
      ncgsteps=30000

      do 33 icc=1,ncgsteps

      gglast=gg
      gg = 0.0d0
      dgg= 0.0d0

      dgg = SUM(gb(2:nx1,2:ny1,dlow:dhigh)*gb(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      if (myrank.eq.0) then
          call flush(7)
      end if

      if(gg.lt.gtest) goto 44
      gamma=gg/gglast

c  update conjugate gradient direction

      h = gb + gamma*h

      call prod_dc(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah,d1,d2)
      hAh=0.0d0
      hAh2= 0.0d0
      hAh2 = SUM(h(2:nx1,2:ny1,dlow:dhigh)*Ah(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(hAh2,hAh,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     &  MPI_COMM_WORLD, ierr)

      lambda=gg/hAh

c  update voltage, gradient vectors
      u = u-lambda*h
      gb=gb-lambda*Ah

c  (USER) This piece of code forces dembx to write out the total current and
c  the norm of the gradient squared, every ncheck conjugate gradient steps,
c  in order to see how the relaxation is proceeding. If the currents become
c  unchanging before the relaxation is done, then gtest was picked to be
c  smaller than was necessary.
      ncheck=30

      if (ncheck*(icc/ncheck).eq.icc) then

         if (myrank.eq.0) then
```

```
            write(7,*) icc
            write(7,*) ' gg = ',gg
         end if

c  call current subroutine
      call current_dc(nx2,ny2,nz2,ns2,cuxxp,cuyyp,cuzzp,
     &                u,gx,gy,gz,d1,d2)

         if (myrank.eq.0) then
            write(7,*) ' cuxxp = ',cuxxp
            write(7,*) ' cuyyp = ',cuyyp
            write(7,*) ' cuzzp = ',cuzzp
            call flush(7)
         end if

      end if

33    continue

      if (myrank.eq.0) then
      write(7,*) ' Iteration failed to converge after',ncgsteps,' steps'
      end if

44    continue
      ic=icc

      return
      end
c
c***********************************************************
c
      subroutine prod_dc(nx2,ny2,nz2,ns2,gx,gy,gz,xw,yw,d1,d2)
c
c The matrix product subroutine
c
      implicit none

      include 'mpif.h'

      integer nx2,ny2,nz2,ns2,d1,d2
      integer nx,ny,nz
      integer nx1,ny1,nz1
      integer L22
      integer dlow,dhigh

      integer im,jm,km,ijk
      integer ip1,jp1,kp1,im1,jm1,km1
      integer ipl,jpl,kpl,iml,jml,kml
      integer ipn,jpn,kpn,imn,jmn,kmn
```

```
      double precision gx(nx2,ny2,d1-1:d2+1)
      double precision gy(nx2,ny2,d1-1:d2+1)
      double precision gz(nx2,ny2,d1-1:d2+1)
      double precision xw(nx2,ny2,d1-1:d2+1)
      double precision yw(nx2,ny2,d1-1:d2+1)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      integer lowrank,highrank

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      lowrank=0; highrank=0

c  xw is the input vector, yw = (A)(xw) is the output vector

c  auxiliary variables involving the system size
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2

      if (nprocs.gt.1) then

          if (myrank.eq.0) then
             dlow = 2
          else
             dlow = d1
          end if

          if (myrank.eq.nprocs-1) then
            dhigh = nz1
          else
            dhigh = d2
          end if

      end if

      if (nprocs.eq.1) then
         dlow=2
         dhigh=nz1
      end if
```

```
c  Perform basic matrix multiplication, results in incorrect information at
c  periodic boundaries.

      yw=0.0d0

      do km=dlow,dhigh
      do jm=1,ny2
      do im=1,nx2

      ijk = L22*(km-1) + nx2*(jm-1) + im

      if ( (ijk.ge.L22+1).AND.(ijk.le.ns2-L22) ) then

c
c Calculation 1
c
      call m2ijk(ijk-1,im1,jm1,km1,nx2,ny2,nz2)
      call m2ijk(ijk-L22,iml,jml,kml,nx2,ny2,nz2)
      call m2ijk(ijk-nx2,imn,jmn,kmn,nx2,ny2,nz2)

      yw(im,jm,km) = -xw(im,jm,km) *
     &            (gx(im1,jm1,km1) + gx(im,jm,km) +
     &             gz(im,jm,km-1)  + gz(im,jm,km) +
     &             gy(im, jm, km)  + gy(imn,jmn,kmn) )

c
c Calculation 2
c

      call m2ijk(ijk+1,ip1,jp1,kp1,nx2,ny2,nz2)
      call m2ijk(ijk+L22,ipl,jpl,kpl,nx2,ny2,nz2)
      call m2ijk(ijk+nx2,ipn,jpn,kpn,nx2,ny2,nz2)

      yw(im,jm,km) = yw(im,jm,km) +
     &              gx(im1,jm1,km1) * xw(im1,jm1,km1) +
     &              gy(imn,jmn,kmn) * xw(imn,jmn,kmn) +
     &              gz(iml,jml,kml) * xw(iml,jml,kml) +
     &              gx(im,jm,km)    * xw(ip1,jp1,kp1) +
     &              gy(im,jm,km)    * xw(ipn,jpn,kpn) +
     &              gz(im,jm,km)    * xw(ipl,jpl,kpl)

      end if

      end do; end do; end do

c
c Calculation 3
c
c  Correct terms at periodic boundaries
```

```
c
c  x faces
      yw(nx2,:,:) = yw(2,:,:)
      yw(1,:,:)   = yw(nx1,:,:)

c  y faces
      yw(:,1,:) = yw(:,ny1,:)
      yw(:,ny2,:)   = yw(:,2,:)

c  z faces

      if (nprocs.eq.1) then

      yw(:,:,1)   = yw(:,:,nz1)
      yw(:,:,nz2) = yw(:,:,2)

      end if

      if (nprocs.gt.1) then

         if ( (d1.le.2).AND.(2.le.d2)) then
              lowrank = myrank

      call MPI_SEND(yw(:,:,2),L22,MPI_DOUBLE_PRECISION,nprocs-1,99,
     &              MPI_COMM_WORLD,ierr)

         end if

         if ( (d1.le.nz1).AND.(nz1.le.d2)) then
              highrank = myrank

      call MPI_SEND(yw(:,:,nz1),L22,MPI_DOUBLE_PRECISION,0,77,
     &              MPI_COMM_WORLD,ierr)

         end if

         if (myrank.eq.0) then
      call MPI_RECV(yw(:,:,1),L22,MPI_DOUBLE_PRECISION,
     &              MPI_ANY_SOURCE,77,MPI_COMM_WORLD,status,ierr)
         end if

         if (myrank.eq.nprocs-1) then
      call MPI_RECV(yw(:,:,nz2),L22,MPI_DOUBLE_PRECISION,
     &              MPI_ANY_SOURCE,99,MPI_COMM_WORLD,status,ierr)
         end if

      end if

      call xy_ghost_dp(yw,nx2,ny2,nz2,d1,d2)
```

```
      call  z_ghost_dp(yw,nx2,ny2,nz2,d1,d2)

      return
      end
c
c************************************************************
c
      subroutine bond_dc(arr0,gx,gy,gz,nx2,ny2,nz2,ns2,
     &                   sigma,be,nphase,ntot,d1,d2)

      implicit none

      include 'mpif.h'

      integer nx2,ny2,nz2,ns2,nphase,ntot,d1,d2
      integer nx,ny,nz,nx1,ny1,nz1,L22
      integer i,j,k,m,dlow,dhigh,k1,j1,i1

      double precision gx(nx2,ny2,d1-1:d2+1)
      double precision gy(nx2,ny2,d1-1:d2+1)
      double precision gz(nx2,ny2,d1-1:d2+1)

      double precision sigma(nphase,3),be(nphase,nphase,3)

      integer*2 arr0(nx2,ny2,d1-1:d2+1)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1
      L22=nx2*ny2

c  Set values of conductor for phase(i,m)--phase(j,m) interface,
c  store in array be(i,j,m), m=1,2,3. If either phase i or j
c  has zero conductivity in the m'th direction, then be(i,j,m)= 0.0.

      be=0.0d0

      do 10 m=1,3
      do 10 i=1,nphase
```

```
      do 10 j=1,nphase

      if(sigma(i,m).eq.0.0) then
      be(i,j,m)=0.
      goto 10
      end if
      if(sigma(j,m).eq.0.0) then
      be(i,j,m)=0.0
      goto 10
      end if
      be(i,j,m)=1./(0.5/sigma(i,m)+0.5/sigma(j,m))

10    continue

c  Trim off x and y faces so that no current can flow past periodic
c  boundaries.  This step is not double precisionly necessary, as the
c  voltages on the periodic boundaries will be matched to the
c  corresponding double precision voltages in each conjugate gradient step.

      gx(nx2,:,:) = 0.0d0
      gy(:,ny2,:) = 0.0d0

      if (myrank.eq.nprocs-1) then
         dhigh = nz1
      else
         dhigh = d2
      end if

      do 30 i=1,nx2
      do 30 j=1,ny2
      do 30 k=d1,dhigh
      k1=k+1

      gz(i,j,k)=be(arr0(i,j,k),arr0(i,j,k1),3)

30    continue

c
c  bulk---gy
c
      if (myrank.eq.0) then
         dlow = 2
      else
         dlow = d1
      end if

      do 40 i=1,nx2
      do 40 j=1,ny1
      do 40 k=dlow,dhigh
```

```
      j1=j+1
      gy(i,j,k)=be(arr0(i,j,k),arr0(i,j1,k),2)

40    continue

c  bulk--gx
      do 50 i=1,nx1
      do 50 j=1,ny2
      do 50 k=dlow,dhigh
      m=(k-1)*L22+(j-1)*nx2+i

      i1=i+1
      j1=j
      k1=k

      gx(i,j,k)=be(arr0(i,j,k),arr0(i1,j,k),1)

50    continue

      gx(nx2,:,:) = 0.0d0
      gy(:,ny2,:) = 0.0d0

      call xy_ghost_dp(gx,nx2,ny2,nz2,d1,d2)
      call  z_ghost_dp(gx,nx2,ny2,nz2,d1,d2)

      call xy_ghost_dp(gy,nx2,ny2,nz2,d1,d2)
      call  z_ghost_dp(gy,nx2,ny2,nz2,d1,d2)

      call xy_ghost_dp(gz,nx2,ny2,nz2,d1,d2)
      call  z_ghost_dp(gz,nx2,ny2,nz2,d1,d2)

      gx(nx2,:,:) = 0.0d0
      gy(:,ny2,:) = 0.0d0

      return
      end
c
c*********************************************************
c
      subroutine dpixel_dc(pix,nx2,ny2,nz2,a,nphase,ntot)

      implicit none

      integer nx2,ny2,nz2,nphase,ntot
      integer nx,ny,nz

      integer i,j,k,m

      double precision a(ntot)
```

```
      integer*2 pix(nx2,ny2,nz2)

c  (USER) If you want to set up a test image inside the program, instead
c  of reading it in from a file, this should be done inside this subroutine.

      nx=nx2-2
      ny=ny2-2
      nz=nz2-2

c  Initialize phase fraction array.

      a=0.0d0

c Use 3-d labelling scheme as shown in manual.
      do 100 k=2,nz2-1
      do 100 j=2,ny2-1
      do 100 i=2,nx2-1
      read(9,*) pix(i,j,k)
      a(pix(i,j,k))=a(pix(i,j,k))+1.0d0
100   continue

      a=a/dfloat(nx*ny*nz)

c  now map periodic boundaries of pix
c  F90 syntax

      pix(nx2,:,:) = pix(2,:,:)
      pix(1,:,:)   = pix(nx2-1,:,:)

      pix(:,ny2,:) = pix(:,2,:)
      pix(:,1,:)   = pix(:,ny2-1,:)

      pix(:,:,1)   = pix(:,:,nz2-1)
      pix(:,:,nz2) = pix(:,:,2)

c  Check for wrong phase labels--less than 1 or greater than nphase
      m = 0
      do 500 k=1,nz
      do 500 j=1,ny
      do 500 i=1,nx
      m = m + 1

       if(pix(i,j,k).lt.1) then
        write(7,*) 'Phase label in PIX < 1--error at ',m
       end if
       if(pix(i,j,k).gt.nphase) then
        write(7,*) 'Phase label in PIX > nphase--error at ',m
       end if
500    continue
```

```
      return
      end
c
c*****************************************************
c
      subroutine current_dc(nx2,ny2,nz2,ns2,cuxxp,cuyyp,cuzzp,
     &                      u,gx,gy,gz,d1,d2)

      implicit none
      include 'mpif.h'

      integer nx2,ny2,nz2,ns2,nx,ny,nz,d1,d2,i,j,k,klow,khigh,m

      integer i0,j0,k0,L22

      integer im1,jm1,km1
      integer ip1,jp1,kp1

      integer iml,jml,kml
      integer ipl,jpl,kpl

      integer imn,jmn,kmn
      integer ipn,jpn,kpn

      double precision gx(nx2,ny2,d1-1:d2+1)
      double precision gy(nx2,ny2,d1-1:d2+1)
      double precision gz(nx2,ny2,d1-1:d2+1)
      double precision u(nx2,ny2,d1-1:d2+1)

      double precision cur1,cur2,cur3
      double precision currx,curry,currz
      double precision cuxxp,cuyyp,cuzzp

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2

c  initialize the volume averaged currents

      cur1=0.0d0
```

```
      cur2=0.0d0
      cur3=0.0d0

      currx=0.0d0
      curry=0.0d0
      currz=0.0d0

      cuxxp=0.0d0
      cuyyp=0.0d0
      cuzzp=0.0d0

c  Only loop over real sites and bonds in order to get true total current

      if (myrank.eq.0) then
         klow = 2
      else
         klow=d1
      end if

      if (myrank.eq.nprocs-1) then
         khigh = nz2-1
      else
         khigh = d2
      end if

      do 10 k=klow,khigh
      do 10 j=2,ny2-1
      do 10 i=2,nx2-1
      m=L22*(k-1)+nx2*(j-1)+i

      call m2ijk(m,i0,j0,k0,nx2,ny2,nz2)
      call m2ijk(m-1,im1,jm1,km1,nx2,ny2,nz2)
      call m2ijk(m+1,ip1,jp1,kp1,nx2,ny2,nz2)
      call m2ijk(m-L22,iml,jml,kml,nx2,ny2,nz2)
      call m2ijk(m-nx2,imn,jmn,kmn,nx2,ny2,nz2)
      call m2ijk(m+L22,ipl,jpl,kpl,nx2,ny2,nz2)
      call m2ijk(m+nx2,ipn,jpn,kpn,nx2,ny2,nz2)

c  cur1, cur2, cur3 are the currents in one pixel
      cur1=0.5d0*((u(im1,jm1,km1)-u(i,j,k)) * gx(im1,jm1,km1)+
     &            (u(i,j,k)-u(ip1,jp1,kp1)) * gx(i,j,k) )

      cur2=0.5d0*((u(imn,jmn,kmn) - u(i,j,k)) * gy(imn,jmn,kmn)+
     &            (u(i,j,k)-u(ipn,jpn,kpn) )  * gy(i,j,k) )

      cur3=0.5d0*((u(iml,jml,kml) - u(i,j,k)) * gz(iml,jml,kml)+
     &            (u(i,j,k)-u(ipl,jpl,kpl))   * gz(i,j,k) )

c  sum pixel currents into volume averages
```

```
      currx=currx+cur1
      curry=curry+cur2
      currz=currz+cur3
10    continue

      call MPI_ALLREDUCE(currx,cuxxp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(curry,cuyyp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(currz,cuzzp,1,MPI_DOUBLE_PRECISION,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

c Volume average currents
      cuxxp=cuxxp/dfloat(nx*ny*nz)
      cuyyp=cuyyp/dfloat(nx*ny*nz)
      cuzzp=cuzzp/dfloat(nx*ny*nz)

      return
      end
c
c***********************************************************
c
      subroutine m2ijk(inps,i,j,k,ni,nj,nk)

      integer inps,ns
      integer c
      integer kdiv,jdiv
      integer rj,rk
      integer i,j,k,ni,nj,nk

      ns=ni*nj
      kdiv=inps/ns
      c = ns*kdiv
      rk = inps-c

      if (rk.eq.0) then
         k=kdiv
         j=nj
         i=ni
      else
         k=kdiv+1
      end if

      if (k.ne.kdiv) then

         jdiv=rk/ni
         c=jdiv*ni
```

```
         rj = rk-c

         if (rj.eq.0) then
            j=jdiv
            i=ni
         else
            j=jdiv+1
            i=rj
         end if

      end if

      return
      end
c
c************************************************************
c
      subroutine xy_ghost_dp(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2
      integer mx1,my1,mz1

      double precision arr0(mx,my,d1-1:d2+1)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mx1 = mx -1
      my1 = my -1
      mz1 = mz -1
c
c Make the X Ghost
c
      arr0(1,:,:)  = arr0(mx1,:,:)
      arr0(mx,:,:) = arr0(2,:,:)
c
c Make the Y Ghost
c
      arr0(:,1,:)  = arr0(:,my1,:)
      arr0(:,my,:) = arr0(:,2,:)

      return
```

```
      end
c
c*****************************************************
c
      subroutine z_ghost_dp(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      double precision arr0(mx,my,d1-1:d2+1)

      double precision, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )
c
c Make the Z Ghost
c
      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
      call t2b_dp(bot,top,mx,my)
      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
      call b2t_dp(bot,top,mx,my)
      arr0(:,:,d2+1) = top
      deallocate(bot)
      deallocate(top)

      return
      end
c
c*****************************************************
c
      subroutine z_ghost_int(arr0,mx,my,mz,d1,d2)
```

```
      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      integer*2 arr0(mx,my,d1-1:d2+1)
      integer*2, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c
c Make the Z Ghost
c
      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call t2b(bot,top,mx,my)

      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call b2t(bot,top,mx,my)

      arr0(:,:,d2+1) = top

      deallocate(bot)
      deallocate(top)

      return
      end
c
c*****************************************************
c
      subroutine t2b(b_layer,t_layer,nx,ny)
```

```
c
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(b_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c***********************************************************
c
      subroutine b2t(b_layer,t_layer,nx,ny)
c
c This is an INTEGER*2 subroutine.
```

```
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(t_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c***********************************************************
c
      subroutine t2b_dp(b_layer,t_layer,nx,ny)
c
c This is a double precision subroutine.
```

```
c
c Used for transferring: u,b,and om top2bottom layers
c
c RECV a new b_layer (BOTTOM layer) per node.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)
      double precision b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_irecv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*****************************************************
c
      subroutine b2t_dp(b_layer,t_layer,nx,ny)
c
c This is a double precision subroutine.
c
```

```
c Used for transferring: u,b,and om bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.
c
      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      double precision b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_Irecv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
```

6.2.2 AC3D_MPI.f

```
c  *************************  AC3D_MPI.f  *********************************
c
c  This is the new MPI version of the AC3D.f code from
c  Section 9.3.5 of NISTIR 6269.
c
c  The main differences with this code compared to the serial
c  version are:
c
c  1. Removal of list array.
c  2. Change of dimensionality on pix from pix(m) to pix(i,j,k)
c  3. All important arrays (pix,vox,gx,gy,gz,gb,b,u,h,ah) are dynamically allocated.
c
c  IN THIS VERSION:
c
c  The USER needs the following input:
c  (Search for occurences of "USER" in the code).
c
c 1. A 3-D pixel value data file with input & output names.
c 2. The values of the 3 dimensions: (nx,ny,nz)
c 3. The number of phases in the mixture: nphase
c 4. A convergence value: gtest
c 5. Components of applied field: ex,ey,ez
c 6. The number of frequencies need to be computed: nfreq
c 7. Define frequency to use each time: w
c 8. Input value of complex conductivity tensor for each phase
c 9. Values for:
c    ncgsteps =>  total number of conjugate gradient steps.
c    ncheck => forces dembx to write out the total current and the norm of
c              the gradient squared, every ncheck conjugate gradient steps
c
c 10. Flag for printing timing info for MPI routines
c    from MAIN is called: pflag
c    pflag Values = 0,1 0=no timing info; 1=print timing info
c
c    pflag is a common value.
c
c    User may edit the code to supress the printing.
c
c END of NEW comments.
c
c BEGIN ORIGINAL comments.
c
c  BACKGROUND

c  This program accepts as input a 3-d digital image, converting it
c  into a complex conductor network. The conjugate gradient method
c  is used to solve this finite difference representation of Laplace's
c  equation for complex conductivity problems.
```

```
c  Periodic boundary conditions are maintained.
c  In the comments below, (USER) means that this is a section of code
c  that the user might have to change for his particular problem.
c  Therefore the user is encouraged to search for this string.

c  PROBLEM AND VARIABLE DEFINITION

c  The mathematical problem that the conjugate gradient algorithm solves
c  is the minimization of the quadratic form 1/2 uAu, where
c  u is the vector of voltages, and A is generated from the bond
c  conductances between pixels. Nodes are thought of as being in the
c  center of pixels. The minimization is constrained by maintaining an
c  general direction applied electric field across the sample.
c  The vectors gx,gy,gz are bond conductances, u is the voltage array,
c  and gb,h, and Ah are auxiliary variables, used in subroutine dembx.
c  The vector pix contains the phase labels for each pixel.
c  The small vector a(i) is the volume fraction
c  of the ith phase, and currx, curry, currz are the total volume-averaged
c  complex currents in the x,y, and z directions.

c  DIMENSIONS

c  The vectors gx,gy,gz,u,gb,h,Ah,list,pix are all dimensioned
c  ns2 = (nx+2)*(ny+2)*(nz+2).  This number is used, rather than the
c  system size nx x ny x nz, because an extra layer of pixels is
c  put around the system to be able to maintain periodic boundary
c  conditions (see manual, Sec. 3.3).  The arrays pix and list are also
c  dimensioned this way.  At present the program is set up for 100 phases,
c  but that can easily be changed by the user, by changing the dimensions
c  of sigma, a, and be.  Note that be has both dimensions equal to
c  each other.  The parameter nphase gives the number of phases
c  being considered.  The parameter ntot is the total number of phases
c  possible in the program, and should be equal to the dimension
c  of sigma, a, and be.  All arrays are passed to subroutines in
c  the call statements.

c  STRONGLY RECOMMENDED:  READ MANUAL BEFORE USING THE PROGRAM!!

c  (USER)  Change these dimensions for different system sizes.  All
c  dimensions in the subroutines are passed, so do not need to be
c  changed.  The dimensions of sigma, a, and be should be equal to
c  the value of ntot.

      implicit none
      include 'mpif.h'

      integer*2, allocatable :: dat(:,:,:), datn(:,:,:)
      integer*2, allocatable :: pix(:,:,:), pixn(:,:,:)
      integer*2, allocatable :: vox(:,:,:)
```

```
      integer, allocatable :: d1s(:),d2s(:)

      double complex, allocatable :: gx(:,:,:)
      double complex, allocatable :: gy(:,:,:)
      double complex, allocatable :: gz(:,:,:)
      double complex, allocatable :: gb(:,:,:)
      double complex, allocatable :: h(:,:,:)
      double complex, allocatable :: ah(:,:,:)
      double complex, allocatable :: u(:,:,:)
      double complex, allocatable :: sigma(:,:)
      double complex, allocatable :: a(:), be(:,:,:)

      double precision ex,ey,ez,gtest,pi,w

      integer nx,nx1,nx2,ny,ny1,ny2,nz,nz1,nz2,L22,ns,ns2
      integer i,j,k,d1,d2,sized,sxip
      integer nphase,ntot,ic,nfreq,nf

      integer myrank, ierr, nprocs, irank, sz, rem
      integer status(MPI_STATUS_SIZE)

      double complex cuxxp,cuyyp,cuzzp

      call MPI_INIT(ierr)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      pi = 4.0*atan(1.0)

      if (myrank.eq.0) then
         write(*,*) "There are ",nprocs," processors running this job."

c  (USER) Unit 9 is the microstructure input file, unit 7 is the
c  results output file.

         open(unit=9,file='microstucture.dat')
         open(unit=7,file='micro_ac_mpi.out')

      end if

c  (USER) double precision image size is nx x ny x nz
      nx=20
      ny=20
      nz=20

      nx2=nx+2
      ny2=ny+2
```

```
      nz2=nz+2
      L22=nx2*ny2

c Variables changed here to reflect the fact our
c calculations are now based on "NZ2" instead of "NZ"
c as in the previous MPI programs.
c
      ns=nx2*ny2*nz2
      sz=nz2/nprocs

      if (myrank.eq.0) then
         allocate (pix(nx2,ny2,nz2))
         write(7,1111) nx,ny,nz,nx*ny*nz
      end if

1111  format(' Image is ',3i6,' No. of double precision sites = ',i8)

c  auxiliary variables involving the system size
      nx1=nx+1
      ny1=ny+1
      nz1=nz+1

c  computational image size ns2 is nx2 x ny2 x nz2
      ns2=nx2*ny2*nz2

c  (USER) set cutoff for norm squared of gradient, gtest.  gtest is
c  the stopping criterion, compared to gb*gb. When gb*gb is less
c  than gtest=abc*ns2, then the rms value of the gradient at a pixel
c  is less than sqrt(abc).
      gtest=1.0d-16*ns2

c  (USER) nphase is the number of phases being considered in the problem.
c  The values of pix(m) will run from 1 to nphase. ntot is the total
c  number of phases possible in the program, and is the dimension of
c  sigma, a, and be.

      nphase=2
      ntot=100

      allocate(a(ntot))
      allocate(sigma(nphase,3))

c
c Program calculates the d1 & d2 limits for the principle arrays.
c
c In this program, d1 & d2 are not based on "nz".
c They are calculated on the basis of "nz2".
c
      if (myrank.eq.0) then
```

```
      allocate (d1s(0:nprocs-1))
      allocate (d2s(0:nprocs-1))

        do irank=0,nprocs-1
          d1s(irank)=irank*sz+1
          d2s(irank)=(irank+1)*sz
        end do

        rem = nz2 - nprocs*sz

        if (rem.ne.0) then
           do j=1,rem
             irank=nprocs-rem+j-1
             d1s(irank)=d1s(irank)+ j-1
             d2s(irank)=d2s(irank)+ j
           end do
        end if

c Send all d1s(i) and d2s(i) from ROOT
c to NODE i & store into d1 & d2 on worker.

        do i=0,nprocs-1
      call MPI_SEND(d1s(i),1,MPI_INTEGER,i,0,MPI_COMM_WORLD,ierr)
      call MPI_SEND(d2s(i),1,MPI_INTEGER,i,1,MPI_COMM_WORLD,ierr)
        end do

      end if

      call MPI_RECV(d1,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,status,ierr)
      call MPI_RECV(d2,1,MPI_INTEGER,0,1,MPI_COMM_WORLD,status,ierr)
      write(*,*) "Rank#",myrank,"d1= ",d1," d2= ",d2

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      if (myrank.eq.0) then
         call dpixel_ac(pix,nx2,ny2,nz2,a,nphase,ntot)

         do i=1,nphase
            write(7,299) i,a(i)
         end do

299   format(' Phase fraction of ',i3,' = ',f12.6)

      end if

c
c Now that the nodes are set up correctly,
c one can pass the data from the root node
```

```
c (myrank=0) to all the rest.
c

      allocate(dat(nx2,ny2,d1:d2))
      sized = SIZE(dat)

      if (nprocs.eq.1) then
         dat=pix
         write(*,*) "dat=pix"
      end if

      if (nprocs.gt.1) then

         if (myrank.eq.0) then

             dat(:,:,d1:d2)=pix(:,:,d1:d2)

             do i=1,nprocs-1
               allocate (pixn(nx2,ny2,d1s(i):d2s(i)))
               pixn = pix(:,:,d1s(i):d2s(i))
               sxip = SIZE(pixn)
               call MPI_SEND(pixn,2*sxip,MPI_BYTE,
     &              i,7,MPI_COMM_WORLD,status,ierr)
               deallocate(pixn)
            end do

         else

            allocate(datn(nx2,ny2,d1:d2))
            call MPI_RECV(datn,2*sized,MPI_BYTE,0,7
     &             ,MPI_COMM_WORLD,status,ierr)

            dat(:,:,d1:d2) = datn
            deallocate(datn)
         end if

      end if

c
c At this point, all the nodes have the initial correct data.
c
c The data on the nodes is dimensioned:  dat(nx,ny,d1:d2)
c
c Need an array vox, st: vox(nx2,ny2,d1-1:d2+1)
c
c Make a copy of dat into vox.
c Make the Z Ghosts
c
c
```

```
c Allocate vox & initialize.
c
      allocate(vox(nx2,ny2,d1-1:d2+1))
      vox = 0
c
c Make the copy
c
      do k=d1,d2
         vox(:,:,k) = dat(:,:,k)
      end do
      deallocate(dat)

c
c Call z_ghost_int to make Z ghost layers
c    of INTEGER*2 values (aka vox).
c
      call z_ghost_int(vox,nx2,ny2,nz2,d1,d2)

c  (USER) Set components of applied field, E = (ex,ey,ez)

      ex=1.0d0
      ey=1.0d0
      ez=1.0d0

      if (myrank.eq.0) then

         write(7,*) 'Applied field components:'
         write(7,*) 'ex = ',ex,' ey = ',ey,' ez = ',ez

      end if

c  Initialize the voltage distribution by putting on uniform field.

      allocate(u(nx2,ny2,d1-1:d2+1))
      u = dcmplx(0.0d0,0.0d0)

      do 30 k=d1,d2
      do 30 j=1,ny2
      do 30 i=1,nx2
      u(i,j,k)=-ex*i-ey*j-ez*k
30    continue

c
c Call z_ghost_dp to make Z ghost layers of DOUBLE COMPLEX values (aka u).
c
      call z_ghost_cmplx(u,nx2,ny2,nz2,d1,d2)

      allocate (be(nphase,nphase,3))
      allocate (gx(nx2,ny2,d1-1:d2+1))
```

```
      allocate (gy(nx2,ny2,d1-1:d2+1))
      allocate (gz(nx2,ny2,d1-1:d2+1))
      allocate (gb(nx2,ny2,d1-1:d2+1))
      allocate (h(nx2,ny2,d1-1:d2+1))
      allocate (Ah(nx2,ny2,d1-1:d2+1))

c  (USER) Set how many frequencies need to be computed.
      nfreq=50
c  Loop over desired frequencies.
      do 300 nf=1,nfreq

c  (USER) Define frequency to use each time.  Alter this statement to get
c  different frequencies. The frequencies are in Hz, according to
c  the units used for sigma.
      w=10.**((nf-1)*11.4/49.-3.)
c  convert to angular frequency
      w=w*2.*pi
      write(7,*) 'No.',nf, ' angular frequency = ',w,' radians'
c  (USER) input value of complex conductivity tensor for each phase
c  (diagonal only). 1,2,3 = x,y,z,  respectively.
      sigma(1,1)=dcmplx(1.0d0,10.d0*w)
      sigma(1,2)=dcmplx(1.0d0,10.d0*w)
      sigma(1,3)=dcmplx(1.0d0,10.d0*w)
      sigma(2,1)=dcmplx(0.5d0,1.d0*w)
      sigma(2,2)=dcmplx(0.5d0,1.d0*w)
      sigma(2,3)=dcmplx(0.5d0,1.d0*w)

      gx=0.0d0; gy=0.0d0; gz=0.0d0; be=0.0d0
      gb=0.0d0;  h=0.0d0; Ah=0.0d0

c  Subroutine bond sets up conductor network in gx,gy,gz 1-d arrays

      call bond_ac(vox,gx,gy,gz,nx2,ny2,nz2,ns2,
     &              sigma,be,nphase,ntot,d1,d2)

      call dembx_ac(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,Ah,
     &  gtest,d1,d2)

      call current_ac(nx2,ny2,nz2,ns2,cuxxp,cuyyp,cuzzp,
     &                      u,gx,gy,gz,d1,d2)

      if (myrank.eq.0) then
         write(7,*) 'Average current in x direction= ',cuxxp
         write(7,*) 'Average current in y direction= ',cuyyp
         write(7,*) 'Average current in z direction= ',cuzzp
         write(7,*) ic,' number of conjugate gradient cycles needed'
         call flush(7)
      end if
```

```
300   continue

      CALL MPI_FINALIZE(ierr)

      end
c
c***********************************************************
c
      subroutine dembx_ac(nx2,ny2,nz2,ns2,gx,gy,gz,u,ic,gb,h,Ah,
     &  gtest,d1,d2)

      implicit none

      include 'mpif.h'

      integer nx2,ny2,nz2,ns2,d1,d2,L22
      integer nx1,ny1,nz1

      double complex gx(nx2,ny2,d1-1:d2+1)
      double complex gy(nx2,ny2,d1-1:d2+1)
      double complex gz(nx2,ny2,d1-1:d2+1)

      double complex u(nx2,ny2,d1-1:d2+1)
      double complex gb(nx2,ny2,d1-1:d2+1)

      double complex Ah(nx2,ny2,d1-1:d2+1)
      double complex h(nx2,ny2,d1-1:d2+1)

      double complex gg,hAh,hAh2,lambda,gglast,gamma
      double complex cuxxp,cuyyp,cuzzp

      double complex dgg
      double precision gtest

      integer ncheck,icc,ncgsteps,ic

      integer nx,ny,nz,dlow,dhigh

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c  Note:  voltage gradients are maintained because in the conjugate gradient
c  relaxation algorithm, the voltage vector is only modified by adding a
c  periodic vector to it.

      L22=nx2*ny2
```

```
      nx1= nx2-1; ny1= ny2-1; nz1= nz2-1
      nx = nx2-2; ny = ny2-2;  nz= nz2-2

      if (nprocs.gt.1) then

          if (myrank.eq.0) then
             dlow = 2
          else
             dlow = d1
          end if

          if (myrank.eq.nprocs-1) then
            dhigh = nz1
          else
            dhigh = d2
          end if

      end if

      if (nprocs.eq.1) then
         dlow=2
         dhigh=nz1
      end if

c  First stage, compute initial value of gradient (gb), initialize h, the
c  conjugate gradient direction, and compute norm squared of gradient vector.

      call prod_ac(nx2,ny2,nz2,ns2,gx,gy,gz,u,gb,d1,d2)

      h=gb

c  Variable gg is the norm squared of the gradient vector

      dgg= dcmplx(0.0d0,0.0d0)
      gg = dcmplx(0.0d0,0.0d0)

      dgg = SUM(gb(2:nx1,2:ny1,dlow:dhigh)*gb(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_COMPLEX,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

c  Second stage, initialize Ah variable, compute parameter lamdba,
c  make first change in voltage array, update gradient (gb) vector

      if(abs(real(gg)).lt.gtest) goto 44

      call prod_ac(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah,d1,d2)
```

```
      hAh = dcmplx(0.0d0,0.0d0)
      hAh2= dcmplx(0.0d0,0.0d0)

      hAh2 = SUM(h(2:nx1,2:ny1,dlow:dhigh)*Ah(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(hAh2,hAh,1,MPI_DOUBLE_COMPLEX,MPI_SUM,
     &  MPI_COMM_WORLD, ierr)

      lambda=gg/hAh

      u=u-lambda*h
      gb=gb-lambda*Ah

c  third stage:  iterate conjugate gradient solution process until
c  gg < gtest criterion is satisfied.
c
c  (USER) The parameter ncgsteps is the total number of conjugate gradient steps
c  to go through.  Only in very unusual problems, like when the conductivity
c  of one phase is much higher than all the rest, will this many steps be
c  used.

      ncgsteps=30000

      do 33 icc=1,ncgsteps

      gglast=gg
      gg = dcmplx(0.0d0,0.0d0)
      dgg= dcmplx(0.0d0,0.0d0)

      dgg = SUM(gb(2:nx1,2:ny1,dlow:dhigh)*gb(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(dgg,gg,1,MPI_DOUBLE_COMPLEX,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      if (myrank.eq.0) then
          call flush(7)
      end if

      if(abs(real(gg)).lt.gtest) goto 44

      gamma=gg/gglast

c  update conjugate gradient direction

      h   = gb + gamma*h

      call prod_ac(nx2,ny2,nz2,ns2,gx,gy,gz,h,Ah,d1,d2)
      hAh=   dcmplx(0.0d0,0.0d0)
      hAh2=  dcmplx(0.0d0,0.0d0)
```

```
      hAh2 = SUM(h(2:nx1,2:ny1,dlow:dhigh)*Ah(2:nx1,2:ny1,dlow:dhigh))

      call MPI_ALLREDUCE(hAh2,hAh,1,MPI_DOUBLE_COMPLEX,MPI_SUM,
     &  MPI_COMM_WORLD, ierr)

      lambda=gg/hAh

c  update voltage, gradient vectors
      u = u-lambda*h
      gb=gb-lambda*Ah

c  (USER) This piece of code forces dembx to write out the total current and
c  the norm of the gradient squared, every ncheck conjugate gradient steps,
c  in order to see how the relaxation is proceeding. If the currents become
c  unchanging before the relaxation is done, then gtest was picked to be
c  smaller than was necessary.

      ncheck=30

      if (ncheck*(icc/ncheck).eq.icc) then

         if (myrank.eq.0) then
            write(7,*) icc
            write(7,*) ' gg = ',gg
         end if

c  call current subroutine
      call current_ac(nx2,ny2,nz2,ns2,cuxxp,cuyyp,cuzzp,
     &                u,gx,gy,gz,d1,d2)

         if (myrank.eq.0) then
            write(7,*) ' cuxxp = ',cuxxp
            write(7,*) ' cuyyp = ',cuyyp
            write(7,*) ' cuzzp = ',cuzzp
            call flush(7)
         end if

      end if

33    continue

      if (myrank.eq.0) then
      write(7,*) ' Iteration failed to converge after',ncgsteps,' steps'
      end if

44    continue
      ic=icc

      return
```

```
      end
c
c*************************************************************
c
      subroutine prod_ac(nx2,ny2,nz2,ns2,gx,gy,gz,xw,yw,d1,d2)
c
c The matrix product subroutine
c
      implicit none

      include 'mpif.h'

      integer nx2,ny2,nz2,ns2,d1,d2
      integer nx,ny,nz
      integer nx1,ny1,nz1
      integer L22
      integer dlow,dhigh

      integer im,jm,km,ijk
      integer ip1,jp1,kp1,im1,jm1,km1
      integer ipl,jpl,kpl,iml,jml,kml
      integer ipn,jpn,kpn,imn,jmn,kmn

      double complex gx(nx2,ny2,d1-1:d2+1)
      double complex gy(nx2,ny2,d1-1:d2+1)
      double complex gz(nx2,ny2,d1-1:d2+1)
      double complex xw(nx2,ny2,d1-1:d2+1)
      double complex yw(nx2,ny2,d1-1:d2+1)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      integer lowrank,highrank

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      lowrank=0; highrank=0

c  xw is the input vector, yw = (A)(xw) is the output vector

c  auxiliary variables involving the system size
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2
```

```
      if (nprocs.gt.1) then

          if (myrank.eq.0) then
             dlow = 2
          else
             dlow = d1
          end if

          if (myrank.eq.nprocs-1) then
            dhigh = nz1
          else
            dhigh = d2
          end if

      end if

      if (nprocs.eq.1) then
         dlow=2
         dhigh=nz1
      end if

c  Perform basic matrix multiplication, results in incorrect information at
c  periodic boundaries.

      yw = dcmplx(0.0d0,0.0d0)

      do km=dlow,dhigh
      do jm=1,ny2
      do im=1,nx2

      ijk = L22*(km-1) + nx2*(jm-1) + im

      if ( (ijk.ge.L22+1).AND.(ijk.le.ns2-L22) ) then

c
c Calculation 1
c
      call m2ijk(ijk-1,im1,jm1,km1,nx2,ny2,nz2)
      call m2ijk(ijk-L22,iml,jml,kml,nx2,ny2,nz2)
      call m2ijk(ijk-nx2,imn,jmn,kmn,nx2,ny2,nz2)

      yw(im,jm,km) = -xw(im,jm,km) *
     &            (gx(im1,jm1,km1) + gx(im,jm,km) +
     &             gz(im,jm,km-1)  + gz(im,jm,km) +
     &             gy(im, jm, km)  + gy(imn,jmn,kmn) )

c
c Calculation 2
```

```
c

      call m2ijk(ijk+1,ip1,jp1,kp1,nx2,ny2,nz2)
      call m2ijk(ijk+L22,ipl,jpl,kpl,nx2,ny2,nz2)
      call m2ijk(ijk+nx2,ipn,jpn,kpn,nx2,ny2,nz2)

      yw(im,jm,km) = yw(im,jm,km) +
     &               gx(im1,jm1,km1) * xw(im1,jm1,km1) +
     &               gy(imn,jmn,kmn) * xw(imn,jmn,kmn) +
     &               gz(iml,jml,kml) * xw(iml,jml,kml) +
     &               gx(im,jm,km)    * xw(ip1,jp1,kp1) +
     &               gy(im,jm,km)    * xw(ipn,jpn,kpn) +
     &               gz(im,jm,km)    * xw(ipl,jpl,kpl)

      end if

      end do; end do; end do

c
c Calculation 3
c
c  Correct terms at periodic boundaries
c
c  x faces
      yw(nx2,:,:) = yw(2,:,:)
      yw(1,:,:)   = yw(nx1,:,:)

c   y faces
      yw(:,1,:) = yw(:,ny1,:)
      yw(:,ny2,:)   = yw(:,2,:)

c   z faces

      if (nprocs.eq.1) then

      yw(:,:,1)   = yw(:,:,nz1)
      yw(:,:,nz2) = yw(:,:,2)

      end if

      if (nprocs.gt.1) then

         if ( (d1.le.2).AND.(2.le.d2)) then
              lowrank = myrank

      call MPI_SEND(yw(:,:,2),L22,MPI_DOUBLE_COMPLEX,nprocs-1,99,
     &             MPI_COMM_WORLD,ierr)

         end if
```

```
            if ( (d1.le.nz1).AND.(nz1.le.d2)) then
                 highrank = myrank

         call MPI_SEND(yw(:,:,nz1),L22,MPI_DOUBLE_COMPLEX,0,77,
     &              MPI_COMM_WORLD,ierr)

            end if

            if (myrank.eq.0) then
         call MPI_RECV(yw(:,:,1),L22,MPI_DOUBLE_COMPLEX,
     &              MPI_ANY_SOURCE,77,MPI_COMM_WORLD,status,ierr)
            end if

            if (myrank.eq.nprocs-1) then
         call MPI_RECV(yw(:,:,nz2),L22,MPI_DOUBLE_COMPLEX,
     &              MPI_ANY_SOURCE,99,MPI_COMM_WORLD,status,ierr)
            end if

         end if

         call xy_ghost_cmplx(yw,nx2,ny2,nz2,d1,d2)
         call  z_ghost_cmplx(yw,nx2,ny2,nz2,d1,d2)

         return
         end
c
c***********************************************************
c
         subroutine bond_ac(arr0,gx,gy,gz,nx2,ny2,nz2,ns2,
     &                   sigma,be,nphase,ntot,d1,d2)

         implicit none

         include 'mpif.h'

         integer nx2,ny2,nz2,ns2,nphase,ntot,d1,d2
         integer nx,ny,nz,nx1,ny1,nz1,L22
         integer i,j,k,m,dlow,dhigh,k1,j1,i1

         double complex gx(nx2,ny2,d1-1:d2+1)
         double complex gy(nx2,ny2,d1-1:d2+1)
         double complex gz(nx2,ny2,d1-1:d2+1)

         double complex sigma(nphase,3),be(nphase,nphase,3)

         integer*2 arr0(nx2,ny2,d1-1:d2+1)

         integer myrank, ierr, nprocs
```

```
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      nx1=nx2-1
      ny1=ny2-1
      nz1=nz2-1
      L22=nx2*ny2

c  Set values of conductor for phase(i,m)--phase(j,m) interface,
c  store in array be(i,j,m), m=1,2,3. If either phase i or j
c  has zero conductivity in the m'th direction, then be(i,j,m)= 0.0.

      be=dcmplx(0.0d0,0.0d0)

      do 10 m=1,3
      do 10 i=1,nphase
      do 10 j=1,nphase

      if(real(sigma(i,m)).eq.0.0.and.aimag(sigma(i,m)).eq.0.0) then
      be(i,j,m)=dcmplx(0.0,0.0)
      goto 10
      end if
      if(real(sigma(j,m)).eq.0.0.and.aimag(sigma(j,m)).eq.0.0) then
      be(i,j,m)=dcmplx(0.0,0.0)
      goto 10
      end if
      be(i,j,m)=1.d0/(0.5d0/sigma(i,m)+0.5d0/sigma(j,m))

10    continue

c  Trim off x and y faces so that no current can flow past periodic
c  boundaries.  This step is not double precisionly necessary, as the
c  voltages on the periodic boundaries will be matched to the
c  corresponding double precision voltages in each conjugate gradient step.

      gx(nx2,:,:) = dcmplx(0.0d0,0.0d0)
      gy(:,ny2,:) = dcmplx(0.0d0,0.0d0)

      if (myrank.eq.nprocs-1) then
         dhigh = nz1
      else
         dhigh = d2
      end if
```

```
      do 30 i=1,nx2
      do 30 j=1,ny2
      do 30 k=d1,dhigh
      k1=k+1

      gz(i,j,k)=be(arr0(i,j,k),arr0(i,j,k1),3)

30    continue

c
c     bulk---gy
c
      if (myrank.eq.0) then
         dlow = 2
      else
         dlow = d1
      end if

      do 40 i=1,nx2
      do 40 j=1,ny1
      do 40 k=dlow,dhigh
      j1=j+1
      gy(i,j,k)=be(arr0(i,j,k),arr0(i,j1,k),2)

40    continue

c
c  bulk--gx
c
      do 50 i=1,nx1
      do 50 j=1,ny2
      do 50 k=dlow,dhigh
      m=(k-1)*L22+(j-1)*nx2+i

      i1=i+1
      j1=j
      k1=k

      gx(i,j,k)=be(arr0(i,j,k),arr0(i1,j,k),1)

50    continue

      gx(nx2,:,:) = dcmplx(0.0d0,0.0d0)
      gy(:,ny2,:) = dcmplx(0.0d0,0.0d0)

      call xy_ghost_cmplx(gx,nx2,ny2,nz2,d1,d2)
      call  z_ghost_cmplx(gx,nx2,ny2,nz2,d1,d2)
```

```
      call xy_ghost_cmplx(gy,nx2,ny2,nz2,d1,d2)
      call  z_ghost_cmplx(gy,nx2,ny2,nz2,d1,d2)

      call xy_ghost_cmplx(gz,nx2,ny2,nz2,d1,d2)
      call  z_ghost_cmplx(gz,nx2,ny2,nz2,d1,d2)

      gx(nx2,:,:) = dcmplx(0.0d0,0.0d0)
      gy(:,ny2,:) = dcmplx(0.0d0,0.0d0)

      return
      end
c
c************************************************************
c
      subroutine dpixel_ac(pix,nx2,ny2,nz2,a,nphase,ntot)

      implicit none

      integer nx2,ny2,nz2,nphase,ntot
      integer nx,ny,nz

      integer i,j,k,m

      double precision a(ntot)
      integer*2 pix(nx2,ny2,nz2)

c  (USER) If you want to set up a test image inside the program, instead
c  of reading it in from a file, this should be done inside this subroutine.

      nx=nx2-2
      ny=ny2-2
      nz=nz2-2

c Initialize phase fraction array.

      a=0.0d0

c Use 3-d labelling scheme as shown in manual
      do 100 k=2,nz2-1
      do 100 j=2,ny2-1
      do 100 i=2,nx2-1
      read(9,*) pix(i,j,k)
      a(pix(i,j,k))=a(pix(i,j,k))+1.0d0
100   continue

      a=a/dfloat(nx*ny*nz)

c now map periodic boundaries of pix
c F90 syntax
```

```
      pix(nx2,:,:) = pix(2,:,:)
      pix(1,:,:)   = pix(nx2-1,:,:)

      pix(:,ny2,:) = pix(:,2,:)
      pix(:,1,:)   = pix(:,ny2-1,:)

      pix(:,:,1)   = pix(:,:,nz2-1)
      pix(:,:,nz2) = pix(:,:,2)

c Check for wrong phase labels--less than 1 or greater than nphase
      m = 0
      do 500 k=1,nz
      do 500 j=1,ny
      do 500 i=1,nx
      m = m + 1

       if(pix(i,j,k).lt.1) then
        write(7,*) 'Phase label in PIX < 1--error at ',m
       end if
       if(pix(i,j,k).gt.nphase) then
        write(7,*) 'Phase label in PIX > nphase--error at ',m
       end if
500    continue

      return
      end
c
c*********************************************************
c
      subroutine current_ac(nx2,ny2,nz2,ns2,cuxxp,cuyyp,cuzzp,
     &                      u,gx,gy,gz,d1,d2)

      implicit none
      include 'mpif.h'

      integer nx2,ny2,nz2,ns2,nx,ny,nz,d1,d2,i,j,k,klow,khigh,m

      integer i0,j0,k0,L22

      integer im1,jm1,km1
      integer ip1,jp1,kp1

      integer iml,jml,kml
      integer ipl,jpl,kpl

      integer imn,jmn,kmn
      integer ipn,jpn,kpn
```

```
      double complex gx(nx2,ny2,d1-1:d2+1)
      double complex gy(nx2,ny2,d1-1:d2+1)
      double complex gz(nx2,ny2,d1-1:d2+1)
      double complex u(nx2,ny2,d1-1:d2+1)

      double complex cur1,cur2,cur3
      double complex currx,curry,currz
      double complex cuxxp,cuyyp,cuzzp

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

c  auxiliary variables involving the system size
      nx=nx2-2
      ny=ny2-2
      nz=nz2-2
      L22=nx2*ny2

c  initialize the volume averaged currents

      cur1= dcmplx(0.0d0,0.0d0)
      cur2= dcmplx(0.0d0,0.0d0)
      cur3= dcmplx(0.0d0,0.0d0)

      currx= dcmplx(0.0d0,0.0d0)
      curry= dcmplx(0.0d0,0.0d0)
      currz= dcmplx(0.0d0,0.0d0)

      cuxxp= dcmplx(0.0d0,0.0d0)
      cuyyp= dcmplx(0.0d0,0.0d0)
      cuzzp= dcmplx(0.0d0,0.0d0)

c  Only loop over real sites and bonds in order to get true total current

      if (myrank.eq.0) then
         klow = 2
      else
         klow=d1
      end if

      if (myrank.eq.nprocs-1) then
         khigh = nz2-1
      else
         khigh = d2
      end if
```

```
      do 10 k=klow,khigh
      do 10 j=2,ny2-1
      do 10 i=2,nx2-1
      m=L22*(k-1)+nx2*(j-1)+i

      call m2ijk(m,i0,j0,k0,nx2,ny2,nz2)
      call m2ijk(m-1,im1,jm1,km1,nx2,ny2,nz2)
      call m2ijk(m+1,ip1,jp1,kp1,nx2,ny2,nz2)
      call m2ijk(m-L22,iml,jml,kml,nx2,ny2,nz2)
      call m2ijk(m-nx2,imn,jmn,kmn,nx2,ny2,nz2)
      call m2ijk(m+L22,ipl,jpl,kpl,nx2,ny2,nz2)
      call m2ijk(m+nx2,ipn,jpn,kpn,nx2,ny2,nz2)

c  cur1, cur2, cur3 are the currents in one pixel
      cur1=0.5d0*((u(im1,jm1,km1)-u(i,j,k)) * gx(im1,jm1,km1)+
     &            (u(i,j,k)-u(ip1,jp1,kp1)) * gx(i,j,k) )

      cur2=0.5d0*((u(imn,jmn,kmn) - u(i,j,k)) * gy(imn,jmn,kmn)+
     &            (u(i,j,k)-u(ipn,jpn,kpn) )  * gy(i,j,k) )

      cur3=0.5d0*((u(iml,jml,kml) - u(i,j,k)) * gz(iml,jml,kml)+
     &            (u(i,j,k)-u(ipl,jpl,kpl))   * gz(i,j,k) )


c  sum pixel currents into volume averages
      currx=currx+cur1
      curry=curry+cur2
      currz=currz+cur3
10    continue

      call MPI_ALLREDUCE(currx,cuxxp,1,MPI_DOUBLE_COMPLEX,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(curry,cuyyp,1,MPI_DOUBLE_COMPLEX,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

      call MPI_ALLREDUCE(currz,cuzzp,1,MPI_DOUBLE_COMPLEX,
     &                MPI_SUM,MPI_COMM_WORLD,ierr)

c Volume average currents
      cuxxp=cuxxp/dfloat(nx*ny*nz)
      cuyyp=cuyyp/dfloat(nx*ny*nz)
      cuzzp=cuzzp/dfloat(nx*ny*nz)

      return
      end
c
c***********************************************************
c
```

```
      subroutine m2ijk(inps,i,j,k,ni,nj,nk)

      integer inps,ns
      integer c
      integer kdiv,jdiv
      integer rj,rk
      integer i,j,k,ni,nj,nk

      ns=ni*nj
      kdiv=inps/ns
      c = ns*kdiv
      rk = inps-c

      if (rk.eq.0) then
         k=kdiv
         j=nj
         i=ni
      else
         k=kdiv+1
      end if

      if (k.ne.kdiv) then

         jdiv=rk/ni
         c=jdiv*ni
         rj = rk-c

         if (rj.eq.0) then
            j=jdiv
            i=ni
         else
            j=jdiv+1
            i=rj
         end if

      end if

      return
      end
c
c***********************************************************
c
      subroutine xy_ghost_dp(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2
```

```
      integer mx1,my1,mz1

      double precision arr0(mx,my,d1-1:d2+1)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mx1 = mx -1
      my1 = my -1
      mz1 = mz -1
c
c Make the X Ghost
c
      arr0(1,:,:)  = arr0(mx1,:,:)
      arr0(mx,:,:) = arr0(2,:,:)
c
c Make the Y Ghost
c
      arr0(:,1,:)  = arr0(:,my1,:)
      arr0(:,my,:) = arr0(:,2,:)

      return
      end
c
c***********************************************************
c
      subroutine xy_ghost_cmplx(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2
      integer mx1,my1,mz1

      double complex arr0(mx,my,d1-1:d2+1)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mx1 = mx -1
      my1 = my -1
      mz1 = mz -1
```

```
c
c Make the X Ghost
c
      arr0(1,:,:)  = arr0(mx1,:,:)
      arr0(mx,:,:) = arr0(2,:,:)
c
c Make the Y Ghost
c
      arr0(:,1,:)  = arr0(:,my1,:)
      arr0(:,my,:) = arr0(:,2,:)

      return
      end
c
c*******************************************************
c
      subroutine z_ghost_dp(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      double precision arr0(mx,my,d1-1:d2+1)

      double precision, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )
c
c Make the Z Ghost
c
      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
      call t2b_dp(bot,top,mx,my)
      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
```

```
      top = arr0(:,:,d2)
      call b2t_dp(bot,top,mx,my)
      arr0(:,:,d2+1) = top
      deallocate(bot)
      deallocate(top)

      return
      end
c
c************************************************************
c
      subroutine z_ghost_cmplx(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      double complex arr0(mx,my,d1-1:d2+1)

      double complex, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )
c
c Make the Z Ghost
c
      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
      call t2b_cmplx(bot,top,mx,my)

      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)
      call b2t_cmplx(bot,top,mx,my)
      arr0(:,:,d2+1) = top
      deallocate(bot)
```

```
      deallocate(top)

      return
      end
c
c*******************************************************
c
      subroutine z_ghost_int(arr0,mx,my,mz,d1,d2)

      implicit none

      include 'mpif.h'

      integer mx,my,mz,d1,d2

      integer*2 arr0(mx,my,d1-1:d2+1)
      integer*2, allocatable :: bot(:,:), top(:,:)

      integer myrank, ierr, nprocs
      integer status(MPI_STATUS_SIZE)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      allocate(bot(mx,my))
      allocate(top(mx,my))
c
c  Get new bottom ghost plane.
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call t2b(bot,top,mx,my)

      arr0(:,:,d1-1) = bot
c
c Get new top ghost plane
c
      bot = arr0(:,:,d1)
      top = arr0(:,:,d2)

      call b2t(bot,top,mx,my)

      arr0(:,:,d2+1) = top

      deallocate(bot)
      deallocate(top)

      return
```

```
      end
c
c*******************************************************
c
      subroutine t2b(b_layer,t_layer,nx,ny)
c
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(b_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(t_layer,2*nxy,MPI_BYTE,ides,9,MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
```

```
c*******************************************************
c
      subroutine b2t(b_layer,t_layer,nx,ny)
c
c This is an INTEGER*2 subroutine.
c
c Used for transferring: pix bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,nxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      integer*2 b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      nxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call MPI_Irecv(t_layer,2*nxy, MPI_BYTE, isrc,
     &               9,MPI_COMM_WORLD, irequest, ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,2*nxy,MPI_BYTE,isrc,9,MPI_COMM_WORLD,
     &              status,ierr)
      call mpi_send(b_layer,2*nxy,MPI_BYTE,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
```

```
c*******************************************************
c
      subroutine t2b_dp(b_layer,t_layer,nx,ny)
c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om top2bottom layers
c
c RECV a new b_layer (BOTTOM layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)
      double precision b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_irecv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c*******************************************************
```

```
c
      subroutine b2t_dp(b_layer,t_layer,nx,ny)

c
c This is a double precision subroutine.
c
c Used for transferring: u,b,and om bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      double precision b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_Irecv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,mxy,MPI_DOUBLE_PRECISION,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_PRECISION,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
```

```
c******************************************************
c
      subroutine t2b_cmplx(b_layer,t_layer,nx,ny)
c
c This is a double complex subroutine.
c
c Used for transferring: u,b,and om top2bottom layers
c
c RECV a new b_layer (BOTTOM layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)
      double complex  b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+1,nprocs)
      isrc = mod(myrank+nprocs-1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_irecv(b_layer,mxy,MPI_DOUBLE_COMPLEX,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_COMPLEX,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(b_layer,mxy,MPI_DOUBLE_COMPLEX,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(t_layer,mxy,MPI_DOUBLE_COMPLEX,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
c
c******************************************************
```

```fortran
c
      subroutine b2t_cmplx(b_layer,t_layer,nx,ny)
c
c This is a double complex subroutine.
c
c Used for transferring: u,b,and om bottom2top layers
c
c RECV a new t_layer (TOP layer) per node.

      implicit none

      include 'mpif.h'

      integer nx,ny,mxy
      integer ides,isrc,irequest
      integer myrank,nprocs,ierr
      integer status(MPI_STATUS_SIZE)

      double complex b_layer(nx,ny), t_layer(nx,ny)

      call MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

      mxy=nx*ny

      ides = mod(myrank+nprocs-1,nprocs)
      isrc = mod(myrank+1,nprocs)

      if (myrank.eq.nprocs-1) then
      call mpi_Irecv(t_layer,mxy,MPI_DOUBLE_COMPLEX,isrc,9,
     &              MPI_COMM_WORLD,irequest,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_COMPLEX,ides,9,
     &              MPI_COMM_WORLD,ierr)
      call MPI_WAIT(irequest,status,ierr)

      else

      call mpi_recv(t_layer,mxy,MPI_DOUBLE_COMPLEX,isrc,9,
     &              MPI_COMM_WORLD,status,ierr)
      call mpi_send(b_layer,mxy,MPI_DOUBLE_COMPLEX,ides,9,
     &              MPI_COMM_WORLD,ierr)
      endif

      call MPI_BARRIER(MPI_COMM_WORLD,ierr)

      return
      end
```

7 Tools

7.1 MEMAPP.f

This FORTRAN90 program makes approximate calculations for the memory requirements need for a given job based on (nx,ny,nz) triplet and number of phases. The user is queried for the (nx,ny,nz) triplet, the number of phases and also the specific program. The resulting output is a list of the approximate memory requirements per processing node.

This is useful information since the user will be able to use these results to decide how much memory will be required per node of their cluster. User should have knowledge of the amount of RAM on each of their cluster nodes.

```
      program memapp

      integer i0
      integer nx,ny,nz,nphase
      integer nx2,ny2,nz2
      integer ns,ns2
      integer nprocs,sz,sz2
      integer npa
      double precision dk,sigma,cmod,pmod,ss,eig,be
      double precision a0, amem0, amem

      write(*,*) "This program allows the user to calculate"
      write(*,*) "the approximate amount of memory they need"
      write(*,*) "in order to perform a parallel NISTIR-6269 job."

      i0= -99

c     nprocs is the number of available processors.
c     It can be changed to suit the user.

      nprocs = 32

      write(*,*) "Enter the problem size, nx,ny,nz and # of phases."
      write(*,*) "Enter nx"; read (5,*) nx
      write(*,*) "Enter ny"; read (5,*) ny
      write(*,*) "Enter nz"; read (5,*) nz
      write(*,*) "Enter number of phases "; read (5,*) nphase

      write(*,*) "nx = ", nx
      write(*,*) "ny = ", ny
      write(*,*) "nz = ", nz
      write(*,*) "nphase = ", nphase

      nx2 = nx+2
      ny2 = ny+2
      nz2 = nz+2
      ns  = nx*ny*nz
      ns2 = nx2*ny2*nz2
```

```
nxy = nx*ny
nxy2 = nx2*ny2

do while ((i0.lt.1).OR.(i0.gt.5))

write(*,*)
write(*,*) "Enter the program which you are going to use:"
write(*,*) "1= ELECFEM3D_MPI"
write(*,*) "2= ELAS3D_MPI"
write(*,*) "3= THERMAL3D_MPI"
write(*,*) "4= DC3D_MPI"
write(*,*) "5= AC3D_MPI"
read(5,*)  i0

select case (i0)

       case (1)
       write(*,*) "You have selected ELECFEM3D_MPI."
       npa = 5
       dk     = float(nphase)*8.0*8.0
       sigma  = float(nphase)*3.0*3.0
       a0 = (dk + sigma) * 8.0

       case (2)
       write(*,*) "You have selected ELAS3D_MPI."
       npa = 5
       dk   = float(nphase) * 8.0*3.0*8.0*3.0
       cmod = float(nphase) * 6.0*6.0
       pmod = float(nphase) * 2.0
       a0 = (dk + cmod + pmod) * 8.0

       case (3)
       write(*,*) "You have selected THERMAL3D_MPI."
       npa = 6
       dk   = float(nphase) * 8.0*3.0*8.0*3.0
       cmod = float(nphase) * 6.0*6.0
       pmod = float(nphase) * 2.0
       ss   = float(nphase) * 8.0*3.0
       eig  =float(nphase) * 6.0
       a0 = (dk + cmod + pmod + ss + eig) * 8.0

       case (4)
       write(*,*) "You have selected DC3D_MPI."
       npa = 7
       sigma = float(nphase) * 3.0
       be    = float(nphase*nphase)*3.0
       a0    = (sigma + be) * 8.0

       case (5)
```

```
            write(*,*) "You have selected AC3D_MPI."
            npa = 7
            sigma = float(nphase) * 3.0
            be    = float(nphase*nphase)*3.0
            a0    = (sigma + be) * 16.0


      end select

      end do

c Output also written to file: memory.dat

      open(unit=7,file='memory.dat')

      do n=1,nprocs

      sz=nz/n
      rem = nz - n*sz
      if (rem.ne.0) then
         sz=sz+1
      end if

      sz2 = nz2/n
      rem2 = nz2 - n*sz2
      if (rem2.ne.0) then
         sz2=sz2+1
      end if

      select case (i0)
c
c Memory for ELECFEM3D
c
         case (1)
         amem0 = 8.0*float(npa)*float(nxy)*float(sz+2) +
     &           2.0*float(nxy)*(sz+2) + a0
c
c Memory for ELAS3D
c
         case (2)
         amem0 = 8.0 *3.0* float(npa)*float(nxy)*float(sz+2) +
     &           2.0*float(nxy)*float(sz+2) + a0
c
c Memory for THERMAL3D
c
         case (3)
         amem0 = 8.0 *3.0*float(npa)*float(nxy)*(sz+2) +
     &           2.0*float(nxy)*float(sz+2) + a0
c
```

```
c Memory for DC3D
c
         case (4)
         amem0 = 8.0 *float(npa)*float(nxy2)*float(sz2+2) +
     &           2.0*float(nxy2)*float(sz2+2) + a0
c
c Memory for AC3D
c
         case (5)
         amem0 = 16.0 *float(npa)*float(nxy2)*float(sz2+2) +
     &           2.0*float(nxy2)*float(sz2+2) + a0

      end select
c
c amem0 is in bytes, convert to MB.
c
      amem = amem0/1e6

      if (rem.eq.0) then
         write(*,99) "*Mem req = ",amem,"MB on ",n," processors."
      else
         write(*,99) "Mem req = ",amem,"MB on ",n," processors."
      end if

      write(7,16) n,amem
99    format(a,f12.2,a,i2,a)

      end do

      close(7)
16    format(i2,2x,f12.2)

      end
```

References

[1] Edward J. Garboczi. Finite element and finite difference programs for computing the linear electric and elastic properties of digital images of random materials. NIST Interagency Report NISTIR 6269, Building and Fire Research Laboratory, National Institute of Standards and Technology, 1998.

[2] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.