# MAE 267 – Project 5
# Parallel, Multi-Block, Finite-Volume Methods
# For Solving 2D Heat Conduction

**Logan Halstrom**
PhD Graduate Student Researcher
Center for Human/Robot/Vehicle Integration and Performance
Department of Mechanical and Aerospace Engineering
University of California, Davis
Davis, California 95616
Email: ldhalstrom@ucdavis.edu

## 1   Statement of Problem

This analysis demonstrates the fundamentals of parallel computing through the numerical solution of the steady-state, two-dimensional temperature distribution of a 1m x 1m steel block with properties listed in Table 1.

Table 1: Steel Block Properties

| Dimensions | 1m x 1m |
|---|---|
| Thermal Conductivity | $k = 18.8 \frac{W}{m \cdot K}$ |
| Density | $\rho = 8000 \frac{kg}{m^3}$ |
| Specific Heat Ratio | $c_p = 500$ |

The demonstration of parallel computing techniques was accomplished in stages, starting with a serial (single-processor) solution of a single grid of dimensions 101x101 and 501x501, which serves as a solver basis and performance benchmark for later parallel codes.

The next stage was to divide the grid into NxM subdomains (blocks), on each of which the solution for a given iteration was calculated independently. 5x4 and 10x10 block decompositions of both previous grid dimensions were solved to demonstrate compartmentalization of solver processes, which is a necessary step for distributing processes in parallel computing.

Finally, the code was adapted to solve multi-block decompositions on multiple processors for the 501x501 grid decomposed into 10x10 blocks running on 1 to 8 processors. For this solution, the domain is decomposed and blocks are distributed on to processors. Decompositions are saved to restart files for each processor to be loaded by each processor in the parallel solver.

## 2   Methods and Equations

The core of this demonstration code is the heat transfer solver developed in the first project, but a number of domain decomposition functions have since been included, as will be detailed in this section.

### 2.1   Grid Initialization

The numerical solution is initialized with the Dirichlet boundary conditions (Eqn 1) using a single processor.

$$T_{BCs} = \begin{cases} 5.0 \left[ \sin\left(\pi x_p\right) + 1.0 \right] & \text{for } j = j_{max} \\ \left| \cos\left(\pi x_p\right) \right| + 1.0 & \text{for } j = 0 \\ 3.0 y_p + 2.0 & \text{for } i = 0, i_{max} \end{cases} \quad (1)$$

$$
\begin{aligned}
rot &= 30.0 \frac{\pi}{180.0} \\
x_p(i) &= \cos\left[ 0.5\pi \frac{i_{max} - i}{i_{max} - 1} \right] \\
y_p(j) &= \cos\left[ 0.5\pi \frac{j_{max} - j}{j_{max} - 1} \right] \\
x &= x_p \cos(rot) + (1.0 - y_p) \sin(rot) \\
y &= y_p \cos(rot) + x_p \sin(rot)
\end{aligned}
\quad (2)
$$

Square grids are generated according to Eqn 2 to create non-uniform spacing in both the x and y directions (with finer spacing at the larger indices). The "prime" system is then rotated by angle *rot* to create the final grid.

### 2.2   Numerical Solver

The solver developed for this analysis utilizes a finite-volume numerical solution method to solve the transient heat conduction equation (Eqn 3).

$$\rho c_p \frac{\partial T}{\partial t} = k \left[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right] \quad (3)$$

To solve Eqn 3 numerically, the equation is discretized according to a node-centered, finite-volume scheme, where first-derivatives at the nodes are found using Green's theorem integrating around the secondary control volumes. Trapezoidal, counter-clockwise integration for the first-derivative in the x-direction is achieved with Eqn 4.

$$\frac{\partial T}{\partial x} = \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}} [(T_{i+1,j} + T_{i+1,j+1}) Ayi_{i+1,j}$$
$$- (T_{i,j} + T_{i,j+1}) Ayi_{i,j} \quad (4)$$
$$- (T_{i,j+1} + T_{i+1,j+1}) Ayi_{i,j+1}$$
$$- (T_{i,j} + T_{i+1,j}) Ayi_{i,j}]$$

A similar scheme is used to find the first-derivative in the y-direction.

## 2.3 Subdomain Decomposition

After grid initialization, the grid is divided into N blocks in the x/I direction and M blocks in the y/J direction, creating a total number of blocks $NBLK = N \cdot M$. All blocks are constrained to have the same number of nodes, so the dimensions of every block *IBLKMAX* and *JBLKMAX* are calculated in Eqn 6 as a fraction of the total number of nodes in each direction, including one point overlap at each inter-block boundary (Ghost nodes are excluded for the moment). In the I-direction, the total number of nodes including overlap (Eqn 5) is:

$$IMAX_{tot} = IMAX + (N-1) \quad (5)$$

and the total number of nodes per block in the I-direction (Eqn 6) is:

$$IMAXBLK = \frac{IMAX_{tot}}{N} = \frac{IMAX + (N-1)}{N} = 1 + \frac{IMAX - 1}{N} \quad (6)$$

Note: For points in J-direction, replace I with J and N with M

Blocks are distributed from 1 to NBLK starting in the lower-left corner of the grid and zipping left to right (the x/I/N direction), then up one (the y/J/M direction) starting again at the left. This is accomplished by two DO loops, the outer loop stepping through J from 1 to M and the inner loop stepping through I from 1 to N. Block locations are stored by assigning global starting indices to each block according to Eqn 7.

$$IMIN_{block} = IMIN_{global} + (IMAXBLK - 1)(I - 1) \quad (7)$$

where I counts blocks in the direction of N and $IMIN_{global} = 1$. The first block in the N-direction has a global starting index of 0, and IMAXBLK must be reduced by one to account for the single-point overlap at block boundaries.

Information for each block is stored as an element in an array of BLKTYPE derived data types. BLKTYPE stores local mesh, temperature, and solver information as well as the block ID, global indices, iteration bounds to prevent over-writing boundary conditions (discussed in Section 2.5), and neighbor identification information.

## 2.4 Processor Distribution

For the parallel code, blocks are distributed among *NPROCS* processors (determined in 'miprun call), with the goal of equal load balancing for all processors (Eqn 8). Load balance is the ratio of a processor's workload to the "Perfect Load Balance" (*PLB*), the total load of all blocks divided by *NPROCS*. In this code, a block's load is refered to as its *SIZE*, so a processor's work load is equal to the sum of the *SIZEs* of its blocks.

$$P_{LoadBalance} = \frac{SUM(SIZEs)}{PLB} \quad (8)$$

The workload of each block (*SIZE*) is calculated as a weighted sum (Eqn 11) of its geometric cost *GEOM* due to grid size (Eqn 9) and communication cost *COMM* due to boundary size ( 10). Geometric cost is essentially the node area of the block iteration bounds:

$$GEOM = (IMAXLOC - IMINLOC) \cdot (JMAXLOC - JMINLOC) \quad (9)$$

Geometric cost will be greater for cells that are not on physical boundaries as they require more ghosts nodes for their inter-block boundaries. Communication cost is calculated as the total length of all faces and corners at interblock boundaries:

$$COMM(i) = \begin{cases} 0, & \text{if BC} \\ IMAXBLK - IMINBLK, & \text{if N or S Face Neighbor} \\ JMAXBLK - JMINBLK, & \text{if E or W Face Neighbor} \\ 1, & \text{if Corner Neighbor} \end{cases}$$
$$COMM = SUM(COMM(i))$$
$$(10)$$

where Eqn 10 must be evaluated for all faces and corners of a given block and the results must be summed.

Weights of each type of cost are currently set to make the maximum possible geometric cost equal to the maximum possible communication cost, as accomplished by Eqn 11.

$$WGEOM = 1$$
$$WCOMM = FACTOR \cdot \frac{(IMAXBLK+2)(JMAXBLK+2)}{(2 \cdot IMAXBLK) + (2 \cdot IMAXBLK) + 4}$$
$$SIZE = (WGEOM \cdot GEOM) + (WCOMM \cdot COMM)$$

$$(11)$$

where $FACTOR$ is a number that can be varied to tune cost weighting, but is currently set to 1.

Once block loads are calculated, they are sorted by size in order of greatest to least. They are then distributed to the processors in this order, where each block is assigned to the current processor with the least load. This produces the theoretical load balancing presented in Section 3. Actual load balancing performance will be determined in Project 5 and tuning will be performed to optimized load balancing.

### 2.5 Ghost Nodes and Neighbor Indentification

In order for each block to function independently for a given iteration of the solver, it must know information about the nodes immediately outside of its boundaries, or, in other words, the interior nodes of its neighbors. To preserve block independence, each block stores the information it needs from its neighbor at the beginning of each iteration in extra, off-block nodes called ghost nodes. These nodes change the local size of each block and necessitate the local iteration parameters $ILOCMIN$, $ILOCMAX$, etc. discussed earlier.

To update each ghost boundary, the identity of the neighbor block for each face is stored in a variable $NB$, which is a neighbor derived data type $NBRTYPE$, which contains IDs for the north, south, east, and west faces and the north east, south east, south west, and north west corners. If the block boundary is a physical boundary instead of an inter-block boundary, the corresponding neighbor identifier is instead set to 0 to indicate a BC boundary. For parallel computing, if a neighbor block is on a different processor (indicating a processor boundary), the neighbor block ID is negated to indicate as such while still preserving the neighbor block ID.

Neighbor information is used to populate a linked list for each boundary type with block IDs so that all similar types of boundaries may be looped through in sequence, rather than using logical sorting at the beginning of each iteration. (Linked lists were shown to produces a 25% speed-up compared to logical sorting for the serial, multi-block code).

When moving to parallel computing, the ID of the neighbor blocks processor must also be bookeeped, as it is required information for accessing the neighbor block for ghost updating. In addition to the neighbor blocks processor, the local index of the neighbor block on its processor must also be stored for this same reason. Thus, this data is stored in corresponding $NBRTYPEs$. Neighbor processor IDs are stored in the variable $NP$. If a block boundary is a BC, the processor ID is negated to indicate as such. Local

indices of neighbor blocks on neighbor processors are stored in $NBLOC$ and are set to 0 if a boundary is a BC.

### 2.6 Configuration Restart Files

After all of the above mentioned initialization processes have been completed, this information is stored in individual restart files for each processor so that the solver may start up independently from these files without needing to determine boundary procedures. Neighbor information, grid, and temperature files are written **for each processor**.

### 2.7 Parallel Performance

Performance of the parallel solver is measured in the results section using Amdahl's Law and the concepts of parallel speedup $S_P$ (Eqn 12) and parallel computational efficiency $E_P$ (Eqn 13).

$$S_P = \frac{t_s}{t_p} \tag{12}$$

where $t_s$ is the serial solution time and $t_p$ is the parallel solution time.

Ideal speedup produces a line with a slope of 1, with the serial solution time decreasing by exactly the amount of processors used. Ideal efficiency is 1, and corresponds to the ideal speedup.

$$E_P = \frac{S_p}{N_P} \tag{13}$$

where $N_P$ is the number of processors in the parallel solution.

Speedup can be predicted as a function of the fraction of the code that is run in parallel according to Eqn 14. Because only the parallel portion of this code is timed, optimal speedup is expected to by near-ideal.

$$S_{P,Opt} = \frac{N_P}{f + (1-f)N_P} \tag{14}$$

### 3 Results and Discussion

Solutions of a 501x501 grid decomposed into 10x10 blocks were solved on 2, 4, 6, and 8 processors on the hpc1 front end (Batch jobs on hpc1 took significantly longer, which seemed to be due to running more processes than allocated cores).

Another important lesson learned from this project was the importance of shared variables in MPI. Many days of debugging ultimatley resulted in discovering that the global grid size was only read in by Processor 0, and was not available to the other processors for calculation of mesh parameters. This was solved with a simple MPI_BCAST.

Another pitfall was in creating unique tag IDs for MPI sends and recieves. Each send is identified by the sending and recieving processor, as well as a tag, so the first attempt

used 8 independent tags for the 8 directions of communication. This proved to be non-unique as one processor could send to another processor more than once in the same direction for a given iteration. In the end, a unique tag was created for each send as the concatination of the direction number with the global sending block ID.
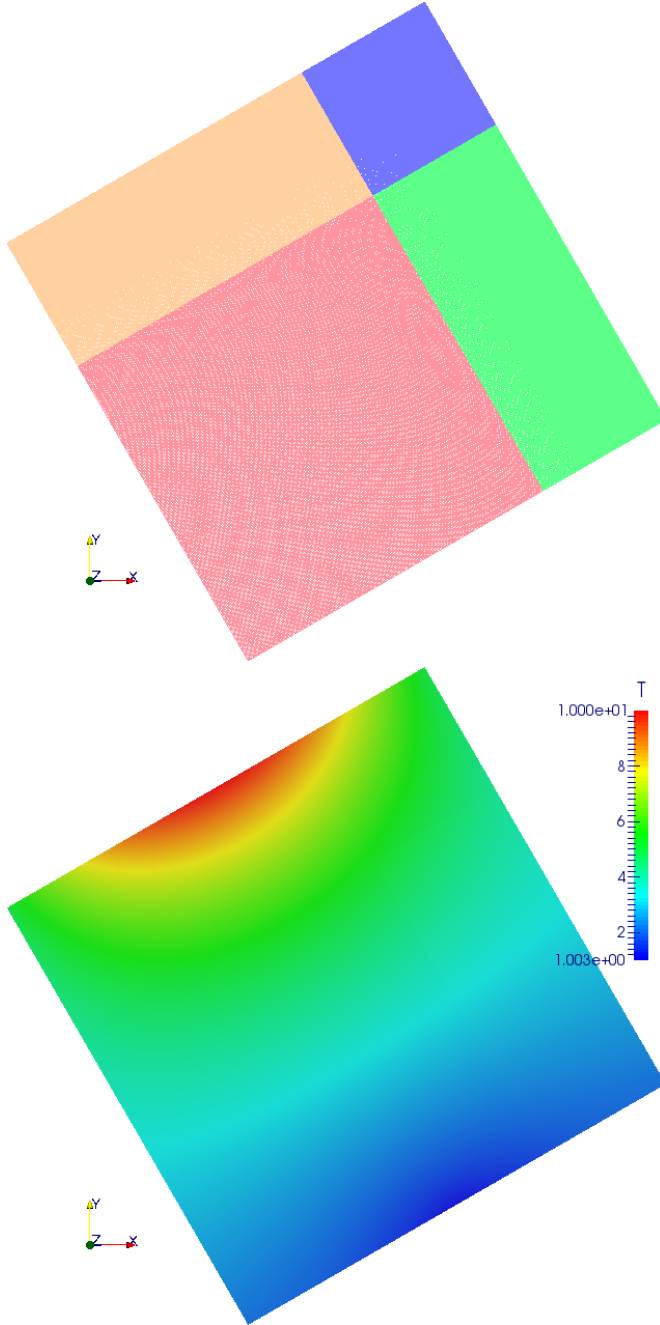


Fig. 1: Grid colored by processor distribution (Top) and steady-state heat transfer solution (Bottom) for a 501x501 grid decomposed into 10x10 blocks solved on 4 processors

Convergence history for the serial, single-block and multi-block cases are compared to the parallel case in Fig 2.
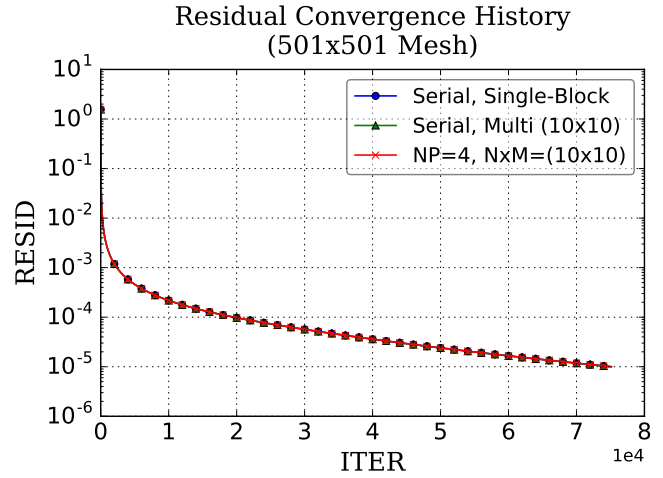


Fig. 2: Serial and parallel convergence history demonstrating similar performance between all solution methods

It can be seen from the convergence comparison that the parallel solver performs almost identically to its serial analogs, and Fig 1 demonstrates that the parallel solution is accurate.

After much deliberation with hpc1, results demonstrating actual parallel speedup were produced and total solution wall times are presented for serial and parallel runs of the Project 5 code in Table 2.

Table 2: **Serial (Left) and Parallel (Right) Solutions Times**

| $N_P$ | (1x1) | (10x10) | 2 | 4 | 6 | 8 |
|-------|-------|---------|-------|-------|-------|-------|
| **t (s)** | 360.7 | 377.8 | 179.4 | 91.95 | 70.56 | 76.09 |

It can be seen that the parallel code does indeed reach a solution faster than its serial analogs. The speedup and efficiency of the parallel calculations are further demonstrated in Figures 3 and 4.

Table 3: **Processor Theoretical Load Balances**

| $NPROCS$ | 4 | 8 |
|----------|--------|---------|
| **Proc0** | 1.0000 | 1.0306 |
| **Proc1** | 1.0000 | 1.0306 |
| **Proc2** | 1.0000 | 1.0306 |
| **Proc3** | 1.0000 | 1.0306 |
| **Proc4** | N/A | 0.96943 |
| **Proc5** | N/A | 0.96943 |
| **Proc6** | N/A | 0.96943 |
| **Proc7** | N/A | 0.96943 |

Because near-perfect load balancing was accomplished (shown in Table 3) and because the timed portion of the code was entirely parallel (see Eqn 14), it was expected that optimal speedups would be near the ideal case, as Fig 3 demonstrates.
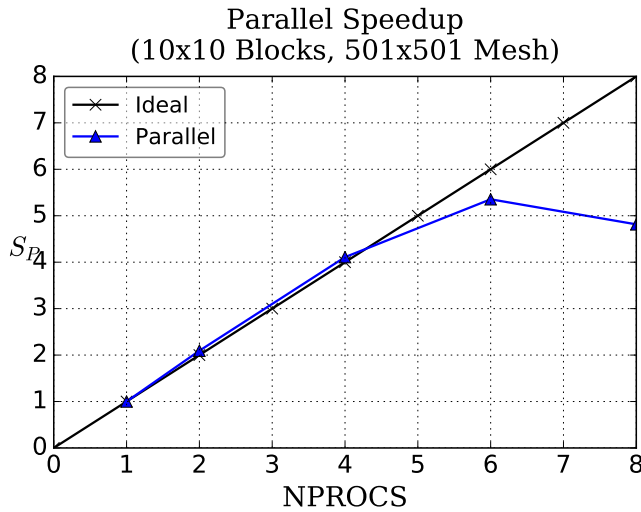
Fig. 3: Parallel speedup for a 501x501 grid decomposed into 10x10 blocks demonstrating an ideal speedup and sometimes super-linear that tapers off as the number of processors increases



Fig. 4: Parallel efficiency for a 501x501 grid decomposed into 10x10 blocks demonstrating a greater than ideal efficiency for lower amounts of processors that may be due to variations in core loading

Fig 3 also demonstrates that the maximum possible speedup for this case is approximately 5.25 running on 6 processors, which is a significant improvement on the performance of the serial version.

The parallel efficiency of the solver is greatest for lower processor numbers. It actualy exceedes maximum efficiency for the 2 and 4 processor cases, but this phenomenon may be due to variation in loading of the hpc1 front-end cores between runs.

A steep drop-off in efficiency is noted as the number of processors is increased. To extend the curve further out, the solver could be applied to a larger grid system decomposed into more sub-domains.

## 4   Conclusion

The product of this project is a parallel, multi-block heat conduction solver demonstrating the fundamental principles of computations in parallel. It increased knowledge of parallel information passing, parallel speedup performance, load balancing optimization.

The working code is just a beginning of what can be accomplished with parallel computing. This code itself could be improved with better sub-domain decomposition optimization and optimization of calculations like send/recieve tags. Beyond that, the principles of this code could be applied to numerical solvers of different principles such as fluid dynamics or structures. Parallel computing could be adapted to GPUs instead of CPU cores. The list is endless.

These projects served as an effective introduction to parallel computing by gradually building on principles; starting with a standard numeric heat transfer solver and then incrementally incorperating parallel computing concepts like domain decomposition and solution restart files so that each concept could be solidified before the next was attempted.
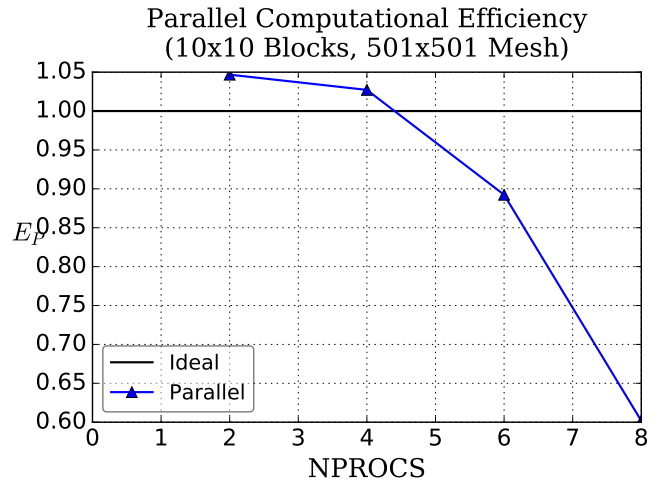
Aside from developing a strong parallel computing basis from which to build off of, the course also developed skills for aquiring further knowledge, leaving the student prepared to delve further into the field of parallel computing.

This course is highly recommended for students interested in numerical solutions of engineering problems, as the principles of these projects can be adapted to any application that may be of interest.

**Appendix A: Parallel, Multi-Block Grid Wrapper Code**

```fortran
! MAE 267
! PROJECT 5
! LOGAN HALSTROM
! 29 NOVEMBER 2015


! DESCRIPTION:  Solve heat conduction equation for single block of steel.

! INPUTS: Set grid size, block decomposition, debug in 'config.in'
!         Set number of processors in 'run.sh'

! TO COMPILE:
    ! mpif90 -o main -O3 modules.f90 inout.f90 subroutines.f90 main.f90
        ! makes executable file 'main'
        ! 'rm *.mod' afterward to clean up unneeded compiled files
! TO RUN:
    ! on hpc1 nodes: sbatch run.sh
    ! on hpc1 front end: ./main or ./run.sh



PROGRAM heatTrans
!     USE CONSTANTS
    USE subroutines

    IMPLICIT NONE

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!! INITIALIZE VARIABLES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    ! ALL BLOCKS IN ONE LIST
    TYPE(BLKTYPE), ALLOCATABLE :: allblocks(:)

    ! PROCESSORS
    TYPE(PROCTYPE), ALLOCATABLE :: procs(:)
    CHARACTER(2) :: procname
    CHARACTER(20) :: xfile, qfile
    ! ITERATION PARAMETERS
    ! Residual history linked list
    TYPE(RESLIST), POINTER :: res_hist
    ! Maximum number of iterations
    INTEGER :: iter = 1, IBLK, IP
    REAL(KIND=8) :: start_total, end_total
    REAL(KIND=8) :: start_solve, end_solve
    ! CLOCK TOTAL TIME OF RUN
    start_total = MPI_Wtime()


    write(*,*) 'starting mpi'

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!! START MPI !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    ! INITIALIZE MPI
    CALL MPI_Init(IERROR)
    ! DETERMINE MY PROCESSOR ID
    ! ARGUMENTS: COMM, MYID, IERROR
    CALL MPI_Comm_rank(MPI_COMM_WORLD, MYID, IERROR)
!    write(*,*) mpi_comm_world
    ! FIND OUT HOW MANY PROCESSORS ARE USED
    ! ARGUMENTS: COMM, NPROCS, IERROR
    CALL MPI_Comm_size(MPI_COMM_WORLD, NPROCS, IERROR)

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!! INITIALIZE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```fortran
      ! READ INPUTS FROM FILE
      CALL read_input()

      ! have the first processor only set up problem
      IF(MYID == 0) THEN

          write(*,*) 'initializing'

!           ! READ INPUTS FROM FILE
!           CALL read_input()
          ALLOCATE( allblocks(NBLK) )
          ALLOCATE( procs(NPROCS) )
          ! INIITIALIZE GRID SYSTEM
          WRITE(*,*) 'Making mesh...'
          CALL init_gridsystem(allblocks, procs)

          ! CLEAN UP INITIALIZATION
          DEALLOCATE(allblocks, procs)
      END IF

!      ! ONLY PROC 0 READS IN CONFIG DATA, SO BRODCAST TO ALL PROCS
!      ! (syntax: variable to brodcast, size, type, which proc to bcast from, otherstuff)
!      CALL MPI_Bcast(IMAX, 1, MPI_INT, 0, mpi_comm_world, ierror)
!      CALL MPI_Bcast(JMAX, 1, MPI_INT, 0, mpi_comm_world, ierror)

      ! HOLD ALL PROCESSORS UNTIL INITIALIZATION IS COMPLETE
      CALL MPI_Barrier(MPI_COMM_WORLD, IERROR)

      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      !!! SOLVER !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

      ! INITIALIZE SOLUTION
      write(*,*) "Initialize for proc ", MYID
      CALL init_solution(blocks, nbrlists, mpilists)

!      if (nprocs == 4) then
!          if (myid == 3) then
!              write(*,*) "block ",   blocks(4)%ID
! !              write(*,*) "iminloc ", blocks(3)%IMINLOC
! !              write(*,*) "Imaxloc ", blocks(3)%IMaxLOC
! !              write(*,*) "jminloc ", blocks(3)%jMINLOC
! !              write(*,*) "jmaxloc ", blocks(3)%jmaxLOC

!              write(*,*) blocks(4)%mesh%term( imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(4)%mesh%V2nd( imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(4)%mesh%V   ( imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(4)%mesh%dt(  imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(4)%mesh%xp(  imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(4)%mesh%x(   imaxblk+1, jmaxblk+1)


!          end if


!      else if (nprocs == 1) then
!          if (myid == 0) then
!              write(*,*) "block ",   blocks(9)%ID
! !              write(*,*) "iminloc ", blocks(10)%IMINLOC
! !              write(*,*) "Imaxloc ", blocks(10)%IMaxLOC
! !              write(*,*) "jminloc ", blocks(10)%jMINLOC
! !              write(*,*) "jmaxloc ", blocks(10)%jmaxLOC
!              write(*,*) blocks(14)%mesh%term( imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(14)%mesh%V2nd( imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(14)%mesh%V   ( imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(14)%mesh%dt(  imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(14)%mesh%xp(  imaxblk+1, jmaxblk+1)
!              write(*,*) blocks(14)%mesh%x(   imaxblk+1, jmaxblk+1)
```

```fortran
137 !                 write(*,*) Imax
138 !            end if
139
140 !      end if
141
142
143     CALL MPI_Barrier(MPI_COMM_WORLD, IERROR)
144     ! SOLVE
145     WRITE(*,*) 'Solving heat conduction with Processor ', MYID
146     CALL solve(blocks, nbrlists, mpilists, iter, res_hist)
147
148     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
149     !!! SAVE RESULTS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
150     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
151
152     WRITE(*,*) 'Writing results...'
153
154     !TURN THIS ON FOR PJ5!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
155
156     ! SAVE SOLUTION AS PLOT3D FILES
157         ! MAKE FILE NAME
158         IF (MYID<10) THEN
159             ! IF SINGLE DIGIT, PAD WITH 0 IN FRONT
160             WRITE(procname, '(A,I1)') '0', MYID
161         ELSE
162             WRITE(procname, '(I2)') MYID
163         END IF
164         xfile = "p" // procname // ".grid"
165         qfile = "p" // procname // ".T"
166         CALL plot3D(blocks, MYNBLK, xfile, qfile)
167     ! CALC TOTAL WALL TIME
168 !     end_total = MPI_Wtime()
169 !     wall_time_total = end_total - start_total
170
171     !TURN THIS ON FOR PJ5!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
172
173     IF (MYID == 0) THEN
174         ! SAVE RESIDUAL HISTORY
175         CALL write_res(res_hist)
176     END IF
177     ! SAVE SOLVER PERFORMANCE PARAMETERS
178     CALL output(blocks, iter)
179
180
181 !     if (myid == 0) then
182 !         call compositePlot3D()
183 !     end if
184
185     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
186     !!! CLEAN UP !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
187     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
188
189     DEALLOCATE(blocks)
190
191     IF (MYID == 0) THEN
192         WRITE(*,*) 'Done!'
193     END IF
194
195     CALL MPI_Finalize(ierror)
196
197 END PROGRAM heatTrans
```

Listing 1: Wrapper program that performs domain decomposition on a single processor and then independently reads in solution for solver

## Appendix B: Parallel, Multi-Block Grid Solver Code

```fortran
1  ! MAE 267
2  ! PROJECT 5
3  ! LOGAN HALSTROM
4  ! 29 NOVEMBER 2015
5
6  ! DESCRIPTION:  Subroutines used for solving heat conduction of steel plate.
7  ! Subroutines utilizing linked lists are here so that linked lists do not need
8  ! to be function inputs.
9  ! Utilizes modules from 'modules.f90'
10
11 ! CONTENTS:
12     ! init_gridsystem
13         ! Initialize the solution with dirichlet B.C.s.  Save to restart files.
14
15     ! init_solution
16         ! Read initial conditions from restart files.  Then calculate parameters
17         ! used in solution
18
19     ! solve
20         ! Solve heat conduction equation with finite volume scheme
21         ! (within iteration loop)
22
23     ! output
24         ! Save solution performance parameters to file
25
26 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
27
28 MODULE subroutines
29 !     USE CONSTANTS
30 !     USE BLOCKMOD
31     USE IO
32
33     IMPLICIT NONE
34
35     ! SOLUTION BLOCKS
36     ! (initialized individually for each parallel processor,
37     !   holds specific blocks distributed to each specific processor)
38     TYPE(BLKTYPE), POINTER :: blocks(:)
39     ! LINKED LISTS STORING NEIGHBOR INFO
40     TYPE(NBRLIST) :: nbrlists
41     ! neighbors on other processors
42     TYPE(NBRLIST) :: mpilists
43
44 CONTAINS
45     SUBROUTINE init_gridsystem(blocks, procs)
46         ! Initialize the solution with dirichlet B.C.s.  Save to restart files.
47
48         TYPE(BLKTYPE)  :: blocks(:)
49         TYPE(PROCTYPE) :: procs(:)
50
51         ! INITIALIZE BLOCKS
52         CALL init_blocks(blocks)
53
54         ! CALC LOCAL BOUNDARIES OF CELLS
55         CALL set_block_bounds(blocks)
56
57         ! INITIALIZE MESH
58         CALL init_mesh(blocks)
59         ! INITIALIZE TEMPERATURE WITH DIRICHLET B.C.
60         CALL init_temp(blocks)
61
62         ! DISTRIBUTE BLOCKS TO PROCESSORS
63         CALL dist_blocks(blocks, procs)
64         ! DETERMIN NEIGHBOR PROCESSOR INFORMATION
65         CALL init_neighbor_procs(blocks, procs)
66
67
```

```fortran
68            ! WRITE BLOCK CONNECTIVITY FILE
69            CALL write_config(procs)
70
71        END SUBROUTINE init_gridsystem
72
73        SUBROUTINE init_solution(blocks, nbrlists, mpilists)
74            ! Read initial conditions from restart files.  Then calculate parameters
75            ! used in solution
76
77            TYPE(BLKTYPE), POINTER  :: blocks(:)
78            ! LINKED LISTS STORING NEIGHBOR INFO
79            TYPE(NBRLIST) :: nbrlists, mpilists
80
81 !          write(*,*) "read config", MYID
82            ! READ BLOCK CONFIGURATION INFORMATION FROM CONFIG FILE
83            CALL read_config(blocks)
84
85            ! INITIALIZE LINKED LISTS CONTAINING BOUNDARY INFORMATION
86 !          write(*,*) 'make linked lists', MYID
87            CALL init_linklists(blocks, nbrlists, mpilists)
88            ! POPULATE BLOCK GHOST NODES
89 !          write(*,*) 'update ghosts', MYID
90            CALL update_ghosts_sameproc(blocks, nbrlists)
91            CALL update_ghosts_diffproc_send(blocks, mpilists)
92            CALL update_ghosts_diffproc_recv(blocks, mpilists)
93
94            ! CALC AREAS FOR SECONDARY FLUXES
95 !          write(*,*) 'calc solution  stuff', MYID
96            CALL calc_cell_params(blocks)
97            ! CALC CONSTANTS OF INTEGRATION
98            CALL calc_constants(blocks)
99
100       END SUBROUTINE init_solution
101
102
103       SUBROUTINE solve(blocks, nbrlists, mpilists, iter, res_hist)
104            ! Solve heat conduction equation with finite volume scheme
105            ! (within iteration loop)
106
107            TYPE(BLKTYPE) :: blocks(:)
108            ! LINKED LISTS STORING NEIGHBOR INFO
109            TYPE(NBRLIST) :: nbrlists, mpilists
110            ! Residual history linked list
111            TYPE(RESLIST), POINTER :: res_hist
112            ! pointer to iterate linked list
113            TYPE(RESLIST), POINTER :: hist
114            ! Minimum residual criteria for iteration, actual residual
115            REAL(KIND=8) :: res = 1000.D0, resloc=0.D0, resmax=0.D0
116            ! iter in function inputs so it can be returned to main
117            INTEGER :: iter, IBLK, IBLKRES
118
119
120            REAL(KIND=8) :: start_solve, end_solve
121            IF (MYID == 0) THEN
122                ! START SOLVER CLOCK
123                start_solve = MPI_Wtime()
124            END IF
125
126            ! residual history
127            ALLOCATE(res_hist)
128            hist => res_hist
129
130            iter_loop: DO WHILE (res >= min_res .AND. iter <= max_iter)
131                ! Iterate FV solver until residual becomes less than cutoff or
132                ! iteration count reaches given maximum
133
134
135
136 !                if (myid == 2 .or. myid == 3) then
```

```fortran
137  !                    write(*,*) "Proc, iter: ", myid, iter
138  !                 end if
139
140  !             if (nprocs == 4) then
141
142  !                 ! 4 proc 5x4
143  ! !                 if (myid == 0) then
144  ! !                     write(*,*) "blk3 east interior values ", blocks(3)%mesh%T(IMAXBLK-1, 2)
145  ! !                     write(*,*) "blk3 east face values ",     blocks(3)%mesh%T(IMAXBLK,   2)
146  ! !                     write(*,*) "blk3 east ghost values ",    blocks(3)%mesh%T(IMAXBLK+1, 2)
147  ! !                 end if
148
149  ! !                 if (myid == 3) then
150  ! !                     write(*,*) "blk4 west ghost values ",    blocks(3)%mesh%T(0, 2)
151  ! !                     write(*,*) "blk4 west face values ",     blocks(3)%mesh%T(1, 2)
152  ! !                     write(*,*) "blk4 west interior values ", blocks(3)%mesh%T(2, 2)
153  ! !                 end if
154
155  ! !                 ! compare node value, should be the same
156  ! !                 if (myid == 2) then
157  ! !                     write(*,*) "        block", blocks(3)%ID, &
158  ! !                                 "SW node", blocks(3)%mesh%T(1, 1)
159  ! !                     write(*,*) "        block", blocks(1)%ID, &
160  ! !                                 "NW node", blocks(1)%mesh%T(1, jmaxblk)
161  ! !                 end if
162  ! !                 if (myid == 3) then
163  ! !                     write(*,*) "        block", blocks(4 )%ID, &
164  ! !                                 "NE node", blocks(4 )%mesh%T(IMAXBLK, JMAXBLK)
165  ! !                     write(*,*) "        block", blocks(5)%ID, &
166  ! !                                 "SE node", blocks(5)%mesh%T(IMAXBLK, 1)
167  ! !                 end if
168
169  !                 ! compare ghost info transfer
170  !                 if (myid == 2) then
171  !                     write(*,*) "        block", blocks(3)%ID, &
172  !                                 "send SW node", blocks(3)%mesh%T(2, 2)
173  !                 end if
174  !                 if (myid == 3) then
175  !                     write(*,*) "        block", blocks(4)%ID, &
176  !                                 "recv NE node", blocks(4)%mesh%T(imaxblk+1, jmaxblk+1)
177  !                 end if
178
179
180
181
182  !             else if (nprocs == 1) then
183
184  !                 ! 1 proc 5x4
185  !                 if (myid == 0) then
186  !                     write(*,*) "blk3 east interior values ", blocks(11)%mesh%T(IMAXBLK-1, 2)
187  !                     write(*,*) "blk3 east face values ",     blocks(11)%mesh%T(IMAXBLK,   2)
188  !                     write(*,*) "blk3 east ghost values ",    blocks(11)%mesh%T(IMAXBLK+1, 2)
189  !                     write(*,*)
190  !                     write(*,*) "blk4 west ghost values ",    blocks(10)%mesh%T(0, 2)
191  !                     write(*,*) "blk4 west face values ",     blocks(10)%mesh%T(1, 2)
192  !                     write(*,*) "blk4 west interior values ", blocks(10)%mesh%T(2, 2)
193  !                 end if
194  !             end if
195
196
197  !             write(*,*) "calc temp ", myid
198
199              ! CALC NEW TEMPERATURE AT ALL POINTS
200              CALL calc_temp(blocks)
201
202  !             write(*,*) "update ghosts ", myid
203
204              ! UPDATE GHOST NODES WITH NEW TEMPERATURE SOLUTION
205              CALL update_ghosts_sameproc(blocks, nbrlists)
```

```fortran
                CALL update_ghosts_diffproc_send(blocks, mpilists)
                CALL update_ghosts_diffproc_recv(blocks, mpilists)

!                 write(*,*) "residual ", myid

                ! CALC RESIDUAL FOR LOCAL BLOCKS
                resmax = 0.D0
                DO IBLK = 1, MYNBLK
                    ! Find max of each block
                    resloc = MAXVAL( ABS( blocks(IBLK)%mesh%Ttmp(2:IMAXBLK-1, 2:JMAXBLK-1) ) )
                    ! keep biggest residual
                    IF (resloc > resmax) THEN
                        resmax = resloc
                    END IF
                END DO
                ! FINAL MAX RESIDUAL (FOR ALL PROCESSORS)
                CALL MPI_ALLREDUCE(resmax, res, 1, MPI_REAL8, MPI_MAX, &
                                    MPI_COMM_WORLD, IERROR)

                ! SWITCH TO NEXT LINK
                    ! (skip first entry)
                ALLOCATE(hist%next)
                hist => hist%next
                NULLIFY(hist%next)
                ! STORE RESIDUAL HISTORY
                hist%iter = iter
                hist%res = res


                ! INCREMENT ITERATION COUNT
                iter = iter + 1

            END DO iter_loop

        ! HOLD UNTIL ALL PROCESSORS HAVE FINISHED ITERATION LOOP
        CALL MPI_Barrier(MPI_COMM_WORLD, IERROR)

        ! there was an extra increment after final iteration we need to subtract
        iter = iter - 1

        IF (MYID == 0) THEN

            ! CALC SOLVER WALL CLOCK TIME
            end_solve = MPI_Wtime()
            wall_time_solve = end_solve - start_solve

            IF (iter > max_iter) THEN
              WRITE(*,*) 'DID NOT CONVERGE (NUMBER OF ITERATIONS:', iter, ')'
            ELSE
              WRITE(*,*) 'CONVERGED (NUMBER OF ITERATIONS:', iter, ')'
              WRITE(*,*) '          (MAXIMUM RESIDUAL   :', res,  ')'
            END IF

        END IF
    END SUBROUTINE solve

    SUBROUTINE output(blocks, iter)
        ! Save solution performance parameters to file

        TYPE(BLKTYPE), TARGET :: blocks(:)
        TYPE(BLKTYPE), POINTER :: b
        REAL(KIND=8), POINTER :: tmpT(:,:), tempTemperature(:,:)
        REAL(KIND=8) :: resloc, resmax
        INTEGER :: iter, I, J, IBLK, IRES, iresmax, jresmax

!         Temperature => mesh%T(2:IMAX-1, 2:JMAX-1)
!         tempTemperature => mesh%Ttmp(2:IMAX-1, 2:JMAX-1)

        ! CALC RESIDUAL
```

```fortran
            resmax = 0.D0
            DO IBLK = 1, MYNBLK
                b => blocks(IBLK)
                DO J = b%JMINLOC, B%JMAXLOC
                    DO I = b%IMINLOC, b%IMAXLOC
                        resloc = ABS( b%mesh%Ttmp(I, J) )
                        IF (resloc > resmax) THEN
                            ! MAX LOCAL RESIDUAL ON PROC
                            resmax = resloc
                            ! Local index of block with max residual
                            IRES = IBLK
                            ! local indices of max residual
                            iresmax = I
                            jresmax = J
                            ! Global indices of max residual
                            iresmax = b%imin + iresmax - 2
                            jresmax = b%jmin + jresmax - 2
                        END IF
                    END DO
                END DO
            END DO


!               ! Find max of each block
!               resloc = MAXVAL( ABS( blocks(IBLK)%mesh%Ttmp(2:IMAXBLK-1, 2:JMAXBLK-1) ) )
!               ! keep biggest residual
!               IF (resloc > resmax) THEN
!                   resmax = resloc
!                   IRES = IBLK
!               END IF
!           END DO

            CALL MPI_Barrier(MPI_COMM_WORLD, IERROR)
            CALL MPI_Bcast(wall_time_solve, 1, MPI_REAL8, 0, mpi_comm_world, ierror)


            ! Write final maximum residual and location of max residual
!           OPEN(UNIT = 1, FILE = casedir // "SteadySoln.dat")
!           DO i = 1, IMAX
!               DO j = 1, JMAX
!                   WRITE(1,'(F10.7, 5X, F10.7, 5X, F10.7, I5, F10.7)'), mesh%x(i,j), mesh%y(i,j), mesh%T(i,j)
!               END DO
!           END DO
!           CLOSE (1)

            ! Screen output
!           tmpT => blocks(IRES)%mesh%Ttmp
!           WRITE (*,*), "IMAX/JMAX", IMAX, JMAX
!           WRITE (*,*), "N/M", N, M
!           WRITE (*,*), "iters", iter
!           WRITE (*,*), "max residual", MAXVAL(tmpT(2:IMAXBLK-1, 2:JMAXBLK-1))
!           WRITE (*,*), "on block id", IRES
!           WRITE (*,*), "residual ij", MAXLOC(tmpT(2:IMAXBLK-1, 2:JMAXBLK-1))




            ! Write to file
            IF (MYID == 0) THEN
                OPEN (UNIT = 2, FILE = "SolnInfo.dat")
                WRITE (2,*), "Running a", IMAX, "by", JMAX, "grid,"
                WRITE (2,*), "On", NPROCS, "processors, With NxM:", N, "x", M, "blocks,"
                WRITE (2,*), iter, "iterations"
!                   WRITE (2,*), wall_time_total, "seconds (Total CPU walltime)"
                WRITE (2,*), wall_time_solve, "seconds (Solver CPU walltime)"
!                   WRITE (2,*), wall_time_iter, "seconds (Iteration CPU walltime)"
                CLOSE (2)
```

```
344         END IF
345
346         ! WRITE RESIDUAL FOR EACH PROC SEQUESENTIALLY
347         DO I = 0, NPROCS-1
348
349             IF (MYID == I) THEN
350                 ! WRITE MAX RESIDUAL/LOCATION FOR EACH PROC
351                 tmpT => blocks(IRES)%mesh%Ttmp
352                 WRITE (*,*)
353 !                   WRITE (*,*), "MAX RESIDUAL FOR PROCESSOR ", MYID
354 !                   WRITE (*,*), "Found max residual of ", MAXVAL(tmpT(2:IMAXBLK-1, 2:JMAXBLK-1))
355 !                   WRITE (*,*), "on block id", blocks(IRES)%ID
356 !                   WRITE (*,*), "At ij of ", MAXLOC(tmpT(2:IMAXBLK-1, 2:JMAXBLK-1))
357                 WRITE (*,*), "MAX RESIDUAL FOR PROCESSOR ", MYID
358                 WRITE (*,*), "Found max residual of ", resmax
359                 WRITE (*,*), "on block id", blocks(IRES)%ID
360                 WRITE (*,*), "At ij of ", iresmax, jresmax
361
362
363             END IF
364             ! WAIT FOR CURRENT PROC TO WRITE
365             CALL MPI_Barrier(MPI_COMM_WORLD, IERROR)
366         END DO
367
368
369
370     END SUBROUTINE output
371
372
373
374 END MODULE subroutines
```

Listing 2:  Grid decomposition and solution processess are organized in this code

## Appendix C: Parallel, Multi-Block Grid Decomposition Code

```
1  ! MAE 267
2  ! PROJECT 5
3  ! LOGAN HALSTROM
4  ! 29 NOVEMBER 2015
5
6  ! DESCRIPTION:  Modules used for solving heat conduction of steel plate.
7  ! Initialize and store constants used in all subroutines.
8
9  ! CONTENTS:
10
11 ! CONSTANTS --> Module that reads, initializes, and stores constants.
12     ! Math and material contants, solver parameters, block sizing
13     ! CONTAINS:
14
15     ! read_input:
16         ! Reads grid/block size and other simulation parameters from
17         ! "config.in" file.  Avoids recompiling for simple input changes
18
19 ! BLOCKMOD --> Module that contains data types and functions pertaining to
20     ! block mesh generation and solution.  Derived data types include;
21     ! MESHTYPE containing node information like temperature, and area,
22     ! NBRTYPE containing information about cell neighbors
23     ! LNKLIST linked list for storing similar neighbor information
24     ! CONTAINS:
25
26         ! init_blocks
27         ! Assign individual block global indicies, neighbor, BCs, and
28         ! orientation information
29
30         ! write_blocks
31         ! Write block connectivity file with neighbor and BC info
```

```fortran
        ! read_blocks
        ! Read block connectivity file

        ! init_mesh
        ! Create xprime/yprime non-uniform grid, then rotate by angle 'rot'.
        ! Allocate arrays for node parameters (i.e. temperature, cell area, etc)

        ! init_temp
        ! Initialize temperature across mesh with dirichlet BCs
        ! or constant temperature BCs for DEBUG=1

        ! set_block_bounds
        ! Calculate iteration bounds for each block to avoid overwriting BCs.
        ! Call after reading in mesh data from restart file

        ! init_linklists
        ! Calculate iteration bounds for each block to avoid overwriting BCs.
        ! Call after reading in mesh data from restart file

        ! update_ghosts
        ! Update ghost nodes of each block based on neightbor linked lists.
        ! Ghost nodes contain solution from respective block face/corner
        ! neighbor for use in current block solution.

        ! update_ghosts_debug
        ! Update ghost nodes of each block using logical statements.
        ! used to debug linked lists

        ! calc_cell_params
        ! calculate areas for secondary fluxes and constant terms in heat
        ! treansfer eqn. Call after reading mesh data from restart file

        ! calc_constants
        ! Calculate terms that are constant regardless of iteration
        !(time step, secondary volumes, constant term.)  This way,
        ! they don't need to be calculated within the loop at each iteration

        ! calc_temp
        ! Calculate temperature at all points in mesh, excluding BC cells.
        ! Calculate first and second derivatives for finite-volume scheme

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! CONSTANTS MODULE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE CONSTANTS
    ! Initialize constants for simulation.  Set grid size.

    IMPLICIT NONE

    ! INCLUDE MPI FOR ALL SUBROUTINES THAT USE CONSTANTS
    INCLUDE "mpif.h"
    ! MPI PROCESSOR ID, NUMBER OF BLOCKS PER PROCESSOR
    ! (initialized by each processor in parallel)
    INTEGER :: MYID, MYNBLK
    ! MPI ERROR STATUS, NUMBER OF MPI PROCESSORS
    INTEGER :: IERROR, NPROCS, request
    INTEGER :: STATUS(MPI_STATUS_SIZE)

    ! CFL number, for convergence (D0 is double-precision, scientific notation)
    REAL(KIND=8), PARAMETER :: CFL = 0.95D0
    ! Material constants (steel): thermal conductivity [W/(m*K)],
                                ! density [kg/m^3],
                                ! specific heat ratio [J/(kg*K)]
                                ! initial temperature
    REAL(KIND=8), PARAMETER :: k = 18.8D0, rho = 8000.D0, cp = 500.D0, T0 = 3.5D0
    ! Thermal diffusivity [m^2/s]
    REAL(KIND=8), PARAMETER :: alpha = k / (cp * rho)
```

```fortran
      ! Pi, grid rotation angle (30 deg)
      REAL(KIND=8), PARAMETER :: pi = 3.141592654D0, rot = 30.D0*pi/180.D0
      ! ITERATION PARAMETERS
      ! Minimum Residual
      REAL(KIND=8) :: min_res = 0.00001D0
      ! Maximum number of iterations
      INTEGER :: max_iter = 250000
!     INTEGER :: max_iter = 1+1
      ! CPU Wall Times
      REAL(KIND=8) :: wall_time_total, wall_time_solve, wall_time_iter(1:5)
      ! read square grid size, Total grid size, size of grid on each block (local)
      INTEGER :: nx, IMAX, JMAX, IMAXBLK, JMAXBLK
      ! Dimensions of block layout, Number of Blocks
      INTEGER :: M, N, NBLK
      ! Block boundary condition identifiers
          ! If boundary is on a different proc, multiply bnd type by proc boundary
      INTEGER :: BND=0, PROCBND = -1
      ! boundary indicators for each direction, use for mpi sends
      INTEGER :: NBND = 1, NEBND=2, EBND = 3, SEBND=4, SBND = 5, SWBND = 6, WBND=7, NWBND=8
      ! Output directory
      CHARACTER(LEN=18) :: casedir
      ! RUN MODE: debug = 0, normal =1, optimizd for 10x10 blocks = 2
      INTEGER :: OPT
      ! Value for constant temperature BCs for debugging
      REAL(KIND=8), PARAMETER :: TDEBUG = T0 - T0 * 0.5

CONTAINS

    SUBROUTINE read_input()
        ! Reads grid/block size and other simulation parameters from
        ! "config.in" file.  Avoids recompiling for simple input changes

        INTEGER :: I
        CHARACTER(LEN=3) :: strNX
        CHARACTER(LEN=1) :: strN, strM

        ! READ INPUTS FROM FILE
            !(So I don't have to recompile each time I change an input setting)
!         WRITE(*,*) ''
!         WRITE(*,*) 'Reading input...'
        OPEN (UNIT = 1, FILE = 'config.in')
        DO I = 1, 3
            ! Skip header lines
            READ(1,*)
        END DO
        ! READ GRIDSIZE (4th line)
        READ(1,*) nx
        ! READ BLOCKS (6th and 8th line)
        READ(1,*)
        READ(1,*) N
        READ(1,*)
        READ(1,*) M
        ! DEBUG MODE (10th line)
        READ(1,*)
        READ(1,*) OPT

        ! SET GRID SIZE
        IMAX = nx
        JMAX = nx
        ! CALC NUMBER OF BLOCKS
        NBLK = M * N
        ! SET SIZE OF EACH BLOCK (LOCAL MAXIMUM I, J)
        IMAXBLK = 1 + (IMAX - 1) / N
        JMAXBLK = 1 + (JMAX - 1) / M

!         ! OUTPUT DIRECTORIES
!         ! write integers to strings
!         WRITE( strNX, '(I3)') nx
!         IF ( N - 10 < 0 ) THEN
```

```fortran
170  !                 ! N is a single digit (I1)
171  !                     WRITE( strN,  '(I1)') N
172  !               ELSE
173  !                 ! N is a tens digit
174  !                     WRITE( strN,  '(I2)') N
175  !               END IF
176  !               IF ( M - 10 < 0 ) THEN
177  !                     WRITE( strM,  '(I1)') M
178  !               ELSE
179  !                     WRITE( strM,  '(I2)') M
180  !               END IF
181  !               ! case output directory: nx_NxM (i.e. 'Results/101_5x4')
182  !               casedir = 'Results/' // strNX // '_' // strN // 'x' // strM // '/'
183  !               ! MAKE DIRECTORIES (IF THEY DONT ALREADY EXIST)
184  !               CALL EXECUTE_COMMAND_LINE ("mkdir -p " // TRIM(casedir) )
185
186          IF (MYID == 0) THEN
187                  ! OUTPUT TO SCREEN
188                  WRITE(*,*) ''
189                  WRITE(*,*) 'Solving Mesh of size ixj:', IMAX, 'x', JMAX
190                  WRITE(*,*) 'Number of processors:', NPROCS
191                  WRITE(*,*) 'With MxN blocks:', M, 'x', N
192                  WRITE(*,*) 'Number of blocks:', NBLK
193                  WRITE(*,*) 'Block size ixj:', IMAXBLK, 'x', JMAXBLK
194                  WRITE(*,*) 'OPT=', OPT
195                  IF (OPT == 0) THEN
196                      WRITE(*,*) 'RUNNING IN DEBUG MODE'
197                  END IF
198                  WRITE(*,*) ''
199          END IF
200      END SUBROUTINE read_input
201  END MODULE CONSTANTS
202
203  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
204  !!!! BLOCK GRID MODULE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
205  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
206
207  MODULE BLOCKMOD
208      ! Initialize grid with correct number of points and rotation,
209      ! set boundary conditions, etc.
210
211      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
212      !
213      ! BLOCK SUBDOMAIN DIAGRAM WITH BOUNDARY CONDITIONS (FOR MXN = 4X5)
214      !
215      !                     NBND = -1
216      !
217      !     JMAX -|-----|-----|-----|-----|-----|
218      !           |     |     |     |     |     |
219      !           | 16  | 17  | 18  | 19  | 20  |
220      !           |-----|-----|-----|-----|-----|
221      !           |     |     |     |     |     |
222      !     J^    | 11  | 12  | 13  | 14  | 15  |
223      !     M=4   |-----|-----|-----|-----|-----|  EBND = -2
224      !   WBND=-4 |     |     |     |     |     |
225      !           |  6  |  7  |  8  |  9  | 10  |
226      !           |-----|-----|-----|-----|-----|
227      !           |     |     |     |     |     |
228      !           |  1  |  2  |  3  |  4  |  5  |
229      !        1 -|-----|-----|-----|-----|-----|
230      !           |                             |
231      !           1              I  ->        IMAX
232      !                          N=5
233      !                     SBND = -3
234      !
235      !  Where IMAX, N, NBND, etc are all global variable stored in CONSTANTS
236      !
237      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
238      !
```

```fortran
! LOCAL/GLOBAL BLOCK INDICIES
!
!                         GLOBAL
!    block(IBLK)%IMIN                block(IBLK)%IMAX
!          |                               |
!  JMAXBLK -|------------------------------|- block(IBLK)%JMAX
!           |                              |
!           |                              |
!           |                              |
!     L     |                              |    G
!     O     |                              |    L
!     C  J^ |        LOCAL BLOCK INDICES   |    O
!     A     |                              |    B
!     L     |                              |    A
!           |                              |    L
!           |                              |
!           |                              |
!         1 -|------------------------------|- block(IBLK)%JMIN
!           |                              |
!           1                I ->        IMAXBLK
!                          LOCAL
!
!  Where block is block data type, IBLK is index of current block
!
!  Convert from local to global (where I is local index):
!  Iglobal = block(IBLK)%IMIN + (I-1)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! LOCAL BLOCK INDICIES WITH GHOST NODES
!
! JMAXBLK+1 |---|------------------------------|---|
!           |NWG|      NORTH GHOST NODES       |NEG|
!   JMAXBLK |---|------------------------------|---|
!           |   |                              |   |
!           | W |                              | E |
!           | E |                              | A |
!           | S |                              | S |
!           | T |                              | T |
!       J^  |   |        LOCAL BLOCK INDICES   |   |
!           | G |                              | G |
!           | H |                              | H |
!           | O |                              | O |
!           | S |                              | S |
!           | T |                              | T |
!           |   |                              |   |
!         1 |---|------------------------------|---|
!           |SWG|      SOUTH GHOST NODES       |SEG|
!         0 |---|------------------------------|---|
!           |   1                           IMAXBLK |
!           0                I ->              IMAXBLK+1
!
!  Where NWG, NEG, etc are corner ghosts
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! BLOCK NEIGHBORS
!
!                    |                |
!                    |      North     |
!               NW|  (IBLK + N)    |NE
!  (IBLK + N - 1)|                    |(IBLK + N + 1)
!  -----------------------------------------------
!                    |                |
!      West          |    Current     |      East
!    (IBLK - 1)  |      (IBLK)     |   (IBLK + 1)
!                    |                |
!  -----------------------------------------------
!               SW|                    |SE
```

```fortran
      ! (IBLK - N - 1)|     South      |(IBLK - N + 1)
      !               |  (IBLK - N)    |
      !               |                |
      !
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      !
      ! LOCAL ITERATION BOUNDS (TO INCLUDE GHOSTS/EXCLUDE BC'S)
      !  -------------
      ! | ~ ~ = BC    |
      ! | . . = Ghost |
      !  -------------
      ! JMAXBLK+1 -|---|--------------|---|
      !            | ~ |. . . . . . . | . |
      !   JMAXBLK -|---|--------------|---|   JMAXBLK -|---|--------------|---|
      !            | ~ |              | . |           | ~ |~ ~ ~ ~ ~ ~ ~ | ~ |
      !            | ~ |              | . | JMAXBLK-1-|---|--------------|---|
      !       J^   | ~ |   M=1, N=1   | . |           | . |              | ~ |
      !            | ~ |              | . |           | . |              | ~ |
      !            | ~ |              | . |     J^    | . |    M=M, N=N   | ~ |
      !         2 -|---|--------------|---|           | . |              | ~ |
      !            | ~ |~ ~ ~ ~ ~ ~ ~ | ~ |           | . |              | ~ |
      !         1 -|---|--------------|---|        1 -|---|--------------|---|
      !            |   2          IMAXBLK |           | . |. . . . . . . | ~ |
      !            1        I ->     IMAXBLK+1     0 -|---|--------------|---|
      !                                              |   1          IMAXBLK-1 |
      !                                              0        I ->     IMAXBLK
      !
      !    Solver  : I = 1 --> IMAXBLK        | Solver  : I = 0 --> IMAXBLK-1
      !        to get: dT: 1 --> IMAXBLK+1    |     to get: dT: 0 --> IMAXBLK
      !    Update T: I = 2 --> IMAXBLK        | Update T: I = 1 --> IMAXBLK-1
      !        (avoid updating BC's at I=1)   |     (avoid updating BC's at I=IMAXBLK)
      !        (IMAXBLK+1 ghost updated later)|     (I=0 ghost updated later)
      !
      !  RESULT:  Set local iteration bounds IMINLOC, IMAXLOC, etc according to solver limits
      !           Update temperature starting at IMINLOC+1 to avoid lower BC's
      !              (upper BC's automatically avoided by explicit scheme solving for i+1)
      !
      !
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

      ! INITIALIZE VARIABLES/DEPENDANCIES
      USE CONSTANTS

      IMPLICIT NONE
      PUBLIC

      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      !!!! DERIVED DATA TYPES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

      ! DERIVED DATA TYPE FOR GRID INFORMATION

      TYPE MESHTYPE
          ! Grid points, see cooridinate rotaion equations in problem statement
          REAL(KIND=8), ALLOCATABLE, DIMENSION(:, :) :: xp, yp, x, y
          ! Temperature at each point, temporary variable to hold temperature sum
          REAL(KIND=8), ALLOCATABLE, DIMENSION(:, :) :: T, Ttmp
          ! Iteration Parameters: timestep, cell volume, secondary cell volume,
          !                       ! equation constant term
          REAL(KIND=8), ALLOCATABLE, DIMENSION(:, :) :: dt, V, V2nd, term
          ! Areas used in alternative scheme to get fluxes for second-derivative
          REAL(KIND=8), ALLOCATABLE, DIMENSION(:, :) :: Ayi, Axi, Ayj, Axj
          ! Second-derivative weighting factors for alternative distribution scheme
          REAL(KIND=8), ALLOCATABLE, DIMENSION(:, :) :: yPP, yNP, yNN, yPN
          REAL(KIND=8), ALLOCATABLE, DIMENSION(:, :) :: xNN, xPN, xPP, xNP
      END TYPE MESHTYPE

      ! DATA TYPE FOR INFORMATION ABOUT NEIGHBORS
```

```fortran
TYPE NBRTYPE
    ! Information about face neighbors (north, east, south, west)
        ! And corner neighbors (Northeast, southeast, southwest, northwest)
    INTEGER :: N, E, S, W, NE, SE, SW, NW
END TYPE NBRTYPE

! DERIVED DATA TYPE WITH INFORMATION PERTAINING TO SPECIFIC BLOCK

TYPE BLKTYPE
    ! DER. DATA TYPE STORES LOCAL MESH INFO
    TYPE(MESHTYPE) :: mesh
    ! IDENTIFY FACE AND CORNER NEIGHBOR BLOCKS AND PROCESSORS
    ! AND LOCAL PROCESSOR BLOCK INDICIES
    TYPE(NBRTYPE) :: NB, NP, NBLOC
    ! BLOCK NUMBER, PROCESSOR NUMBER
    INTEGER :: ID, procID
    ! GLOBAL INDICIES OF MINIMUM AND MAXIMUM INDICIES OF BLOCK
    INTEGER :: IMIN, IMAX, JMIN, JMAX
    ! LOCAL ITERATION BOUNDS TO AVOID UPDATING BC'S + UTILIZE GHOST NODES
    INTEGER :: IMINLOC, JMINLOC, IMAXLOC, JMAXLOC, IMINUPD, JMINUPD
    ! BLOCK LOAD PARAMETERS FOR PROCESSOR LOAD BALANCING
    INTEGER :: SIZE
    ! BLOCK ORIENTATION
    INTEGER :: ORIENT
END TYPE BLKTYPE

! DATA TYPE FOR PROCESSOR INFORMATION

TYPE PROCTYPE
    ! Information pertaining to each processor: procID, number of blocks
    ! on proc
    INTEGER :: ID, NBLK=0
    ! processor load, load balance
    INTEGER :: load=0
    REAL(KIND=8) :: balance=0
    ! Blocks contained on processor
    TYPE(BLKTYPE), ALLOCATABLE :: blocks(:)
END TYPE PROCTYPE

! LINKED LIST: RECURSIVE POINTER THAT POINTS THE NEXT ELEMENT IN THE LIST

TYPE LNKLIST
    ! Next element in linked list
    TYPE(LNKLIST), POINTER :: next
    ! Identify what linked list belongs to
    INTEGER :: ID
END TYPE LNKLIST

! Collection of linked lists for faces and corners

TYPE NBRLIST
    TYPE(LNKLIST), POINTER :: N, E, S, W, NE, SE, SW, NW
END TYPE NBRLIST

CONTAINS

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!! INITIALIZE GRID AND WRITE TO FILE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    SUBROUTINE init_blocks(b)
        ! Assign individual block global indicies, neighbor, BCs, and
        ! orientation information

        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), TARGET :: b(:)
        ! Neighbor information pointer
        TYPE(NBRTYPE), POINTER :: NB
        ! COUNTER VARIABLES
```

```fortran
            ! IM, IN COUNT BLOCK INDICIES
            ! (IBLK COUNTS BLOCK NUMBERS, INBR IS BLOCK NEIGHBOR INDEX)
        INTEGER :: I, J, IBLK, INBR

    ! STEP THROUGH BLOCKS, ASSIGN IDENTIFYING INFO

    ! START AT BLOCK 1 (INCREMENT IN LOOP)
    IBLK = 0

    DO J = 1, M
        DO I = 1, N
            ! INCREMENT BLOCK NUMBER
            IBLK = IBLK + 1

            ! Neighbor information pointer
            NB => b(IBLK)%NB

            ! ASSIGN BLOCK NUMBER
            b(IBLK)%ID = IBLK
            ! ASSIGN GLOBAL MIN/MAX INDICIES OF LOCAL GRID
            b(IBLK)%IMIN = 1 + (IMAXBLK - 1) * (I - 1)
            b(IBLK)%JMIN = 1 + (JMAXBLK - 1) * (J - 1)
            b(IBLK)%IMAX = b(IBLK)%IMIN + (IMAXBLK - 1)
            b(IBLK)%JMAX = b(IBLK)%JMIN + (JMAXBLK - 1)

            ! ASSIGN NEIGHBORS
            ! (Numbers of face and corner neighbor blocks)
            ! (if boundary face, assign bc later)
            NB%N  = IBLK + N
            NB%S  = IBLK - N
            NB%E  = IBLK + 1
            NB%W  = IBLK - 1
            NB%NE = IBLK + N + 1
            NB%NW = IBLK + N - 1
            NB%SW = IBLK - N - 1
            NB%SE = IBLK - N + 1

            ! ASSIGN BOUNDARY CONDITIONS

            ! Assign faces and corners on boundary of the actual
            ! computational grid with number corresponding to which
            ! boundary they are on.
                ! Corners on actual corners of the computational grid are
                ! ambiguously assigned.
            IF ( b(IBLK)%JMAX == JMAX ) THEN
                ! NORTH BLOCK FACE AND CORNERS ARE ON MESH NORTH BOUNDARY
                    ! AT ACTUAL CORNERS OF MESH, CORNERS ARE AMBIGUOUS
                NB%N  = BND
                NB%NE = BND
                NB%NW = BND
            END IF
            IF ( b(IBLK)%IMAX == IMAX ) THEN
                ! EAST BLOCK FACE IS ON MESH EAST BOUNDARY
                NB%E  = BND
                NB%NE = BND
                NB%SE = BND

            END IF
            IF ( b(IBLK)%JMIN == 1 ) THEN
                ! SOUTH BLOCK FACE IS ON MESH SOUTH BOUNDARY
                NB%S  = BND
                NB%SE = BND
                NB%SW = BND
            END IF
            IF ( b(IBLK)%IMIN == 1 ) THEN
                ! WEST BLOCK FACE IS ON MESH WEST BOUNDARY
                NB%W  = BND
                NB%SW = BND
                NB%NW = BND
```

```fortran
                END IF

                ! BLOCK ORIENTATION
                    ! same for all in this project
                b(IBLK)%ORIENT = 1

            END DO
        END DO
    END SUBROUTINE init_blocks

    SUBROUTINE dist_blocks(blocks, procs)
        ! Distribute blocks to processors.  Calculate processor load of each
        ! block based on geometry and communication costs and weighting factors
        ! for each.
        ! Initialize processor list with proc ID's and allocate proc block lists
        ! Distribute blocks to processors by sorting blocks in decreasing order
        ! of load, then distributing sequentially to the processor with the
        ! least load.
        ! Calculate load balance of all processors.

        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), TARGET :: blocks(:)
        TYPE(BLKTYPE), POINTER :: b
        TYPE(NBRTYPE), POINTER :: NB
        ! PROCESSOR DATA TYPE
        TYPE(PROCTYPE), TARGET :: procs(:)
        TYPE(PROCTYPE), POINTER :: p
        ! COUNTER VARIABLE
        INTEGER :: IBLK, I, IPROC, II
        ! CURRENT BLOCK DIMENSIONS
        INTEGER :: NXLOC, NYLOC
        ! COMPUTATIONAL COST PARAMETERS
        ! (geometric (grid size) and communication weights)
        INTEGER :: GEOM=0, COMM=0, MAXCOMM, MAXGEOM
        ! WEIGHTS FOR LOAD BALANCING
        ! (geometry, communication, fudge factor)
        REAL(KIND=8) :: WGEOM = 1.0D0, WCOMM, FACTOR=1.D0
        ! Perfect load balance
        INTEGER :: PLB = 0
        ! VARIABLES FOR SORTING BLOCKS BY LOAD
        ! maximum block load
        INTEGER :: MAXSIZE=0, MINLOAD
        ! 'sorted' is list of IDs of blocks in order of size greatest to least
        ! 'claimed' idicates if a block has already been sorted (0/1 --> unsorted/sorted)
            ! initial list is all zeros
        ! 'IMAXSIZE' is index of remaining block with greatest size
        INTEGER :: sorted(NBLK), claimed(NBLK), IMAXSIZE, IMINLOAD
        INTEGER :: locIDs(NBLK), proclist(NBLK), idsSort(Nblk), procSort(NBLK)
        ! OPTIMIZED DISTRIBUTION
        INTEGER :: METHOD

        ! INITIALIZE LISTS
        DO I = 1, NBLK
            claimed(I) = 0
            sorted(I) = 0
        END DO

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!! SET WEIGHTING FACTORS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        ! SET COMMUNICATION WEIGHT TO BE PROPORTIONAL TO GEOMETRY
        ! Maximum geometry cost is all cells with ghost nodes at all faces
        MAXGEOM = ( IMAXBLK + 2 ) * ( JMAXBLK + 2 )
        ! Maximum communication cost is all face boundaries plus four corners
        MAXCOMM = ( 2 * IMAXBLK ) + ( 2 * JMAXBLK ) + 4
        ! Put comm cost on same scale as geom
        WCOMM = FACTOR * ( DFLOAT(MAXGEOM) / DFLOAT(MAXCOMM) )
        ! COME UP WITH A BETTER WEIGHTING FACTOR IN PROJECT 5 WHEN YOU CAN BENCHMARK TIMES!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```fortran
 10         FORMAT(10A12)
            WRITE(*,*)
            WRITE(*,*) 'Processor Load Weighting Factors:'
            WRITE(*,*) 'WGEOM=', WGEOM, 'WCOMM=', WCOMM
            WRITE(*,*)
            WRITE(*,*) 'SIZE = WGEOM*GEOM + WCOMM*COMM'
            WRITE(*,*)
            WRITE(*,*) 'Block Load Factors:'
            WRITE(*,10) 'BLKID', 'GEOM', 'COMM', 'SIZE'

            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            !!! CALC BLOCK WEIGHTS FOR PROCESSOR LOAD BALANCING !!!!!!!!!!!!
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

            ! need local block sizes
            CALL set_block_bounds(blocks)
            DO IBLK = 1, NBLK
                b => blocks(IBLK)
                NB => b%NB

                ! RESET COST SUMS
                GEOM = 0
                COMM = 0

                ! LOCAL BLOCK DIMENSIONS
                NXLOC = b%IMAXLOC - b%IMINLOC
                NYLOC = b%JMAXLOC - b%JMINLOC

                ! GEOMETRIC BLOCK WEIGHT ("VOLUME")
                GEOM = NXLOC * NYLOC

                ! COMMUNICATION BLOCK WEIGHT
                ! NORTH
                IF (NB%N > 0) THEN
                    ! Interior faces have communication cost for populating ghosts
                    COMM = COMM + IMAXBLK
                END IF
                ! EAST
                IF (NB%E > 0) THEN
                    COMM = COMM + JMAXBLK
                END IF
                ! SOUTH
                IF (NB%S > 0) THEN
                    COMM = COMM + IMAXBLK
                END IF
                ! WEST
                IF (NB%W > 0) THEN
                    COMM = COMM + JMAXBLK
                END IF
                ! NORTHEAST
                IF (NB%N > 0) THEN
                    ! Interior corners have communication cost for populating ghosts
                    COMM = COMM + 1
                END IF
                ! SOUTHEAST
                IF (NB%E > 0) THEN
                    COMM = COMM + 1
                END IF
                ! SOUTHWEST
                IF (NB%S > 0) THEN
                    COMM = COMM + 1
                END IF
                ! NORTHWEST
                IF (NB%W > 0) THEN
                    COMM = COMM + 1
                END IF

                ! CALCULATE TOTAL LOAD OF BLOCK WITH WEIGHTING FACTORS
```

```fortran
            b%SIZE = INT( WGEOM * DFLOAT(GEOM) + WCOMM * DFLOAT(COMM) )

            ! WRITE BLOCK LOADS
            WRITE(*,*) IBLK, GEOM, COMM, b%SIZE

            ! SUM BLOCK LOADS
            PLB = PLB + b%SIZE
        END DO

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!! CALC OPTIMAL LOAD DISTRIBUTION (PERFECT LOAD BALANCE) !!!!!!
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        ! (total load of all blocks divided by number of processors)
        PLB = INT( DFLOAT(PLB) / DFLOAT(NPROCS) )

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!! SORT BLOCKS BY LOAD IN DECREASING ORDER !!!!!!!!!!!!!!!!!!!!
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        DO IBLK = 1, NBLK

            ! Reset current max size
            MAXSIZE = 0

            ! FIND MAX SIZE OF REMAINING BLOCKS
            DO I = 1, NBLK
                b => blocks(I)

                ! (all sorted blocks will be excluded by 'claimed')
                IF (claimed(I)==0 .AND. b%SIZE>=MAXSIZE) THEN
                    ! CURRENT BLOCK HAS GREATEST LOAD SIZE OF REMAINING BLOCKS
                    MAXSIZE = b%SIZE
                    ! INDEX OF MAX REMAINING SIZE BLOCK
                    IMAXSIZE = I
                END IF
            END DO
            ! MARK LATEST MAX AS SORTED (so it doesn't come up again)
            claimed(IMAXSIZE) = 1
            ! ADD INDEX OF LATEST MAX TO SORTED INDEX LIST
            sorted(IBLK) = IMAXSIZE
        END DO

        ! write block size order
        write(*,*) " "
        write(*,*) "Blocks ordered by size, greatest to least, with sizes:"
        DO I = 1, NBLK
            b => blocks( sorted(I) )
            write(*,*) b%ID, b%SIZE
        END DO
        write(*,*) " "

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!! INITIALIZE PROCS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        DO IPROC = 1, NPROCS
            ! SET EACH PROCESSOR'S ID
            ! (Processor indexing starts at zero)
            procs(IPROC)%ID = IPROC-1
            ! ALLOCATE BLOCK LISTS FOR EACH PROC
            ! (Make them NBLK long even though they will contain less than that
            ! so we dont have to reallocate)
            ALLOCATE( procs(IPROC)%blocks(NBLK) )
        END DO


        IF (OPT == 2) THEN
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```fortran
            !!! HARDCODED OPTIMIZED DIST FOR 10X10 BLOCKS !!!!!!!!!!!!!!!!!!!
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

            ! DISTRIBUTION OPTIONS
            IF (NPROCS == 10) THEN

                ! 10X10 BLOCKS, 10 PROCS

                ! ASSIGN EACH ROW TO ONE INDIVIDUAL PROCESSOR
                DO IPROC = 0, NPROCS-1
                    DO I = 1, 10
                        CALL assign_block( blocks( I + IPROC*10 ), procs(IPROC) )
                    END DO
                END DO

            ELSE IF (NPROCS == 8) THEN
                ! 10X10 BLOCKS, 8 PROCS

                ! TRY TO ASSIGN AS MANY PROCESSORS IN ONE ROW TO ONE PROC
                    ! (assign sequesentially)
                DO I = 1, 13
                    CALL assign_block( blocks(I), procs(1) )
                END DO
                DO I = 14, 25
                    CALL assign_block( blocks(I), procs(2) )
                END DO
                DO I = 26, 37
                    CALL assign_block( blocks(I), procs(3) )
                END DO
                DO I = 38, 50
                    CALL assign_block( blocks(I), procs(4) )
                END DO
                DO I = 51, 63
                    CALL assign_block( blocks(I), procs(5) )
                END DO
                DO I = 64, 75
                    CALL assign_block( blocks(I), procs(6) )
                END DO
                DO I = 76, 87
                    CALL assign_block( blocks(I), procs(7) )
                END DO
                DO I = 88, 100
                    CALL assign_block( blocks(I), procs(8) )
                END DO

            ELSE IF (NPROCS == 6) THEN
                ! 10X10 BLOCKS, 6 PROCS

                ! ASSIGN FIRST 3 ROWS TO PROCS 1 & 2,
                ! DIVIDED IN HALF (PROC1 LEFT, PROC2 RIGHT)
                DO I = 0, 2
                    DO II = 1, 5
                        CALL assign_block( blocks( II + I*10 ), procs(1) )
                    END DO
                    DO II = 6, 10
                        CALL assign_block( blocks( II + I*10 ), procs(2) )
                    END DO
                END DO
                ! GIVE CENTER BLOCKS OF 4TH ROW TO PROCS 1 & 2,
                ! 2 TO EACH PROC ON EACH PROC'S SIDE
                ! left 2 center blocks to proc 1
                CALL assign_blocks( blocks, procs(1), [34, 35] )
                ! right 2 center blocks to proc 2
                CALL assign_blocks( blocks, procs(2), [36, 37] )
                ! GIVE EDGE BLOCKS OF 4TH ROW TO PROCS 3 & 4
                ! leftmost 3 blocks to proc 3
                CALL assign_blocks( blocks, procs(3), [31, 32, 33] )
                ! rightmost 3 blocks to proc 4
                CALL assign_blocks( blocks, procs(4), [38, 39, 40] )
```

```fortran
                    ! ASSIGN 5TH AND 6TH ROWS TO PROCS 3 & 4,
                    ! DIVIDED IN HALF
                    DO I = 4, 5
                        DO II = 1, 5
                            CALL assign_block( blocks( II + I*10 ), procs(3) )
                        END DO
                        DO II = 6, 10
                            CALL assign_block( blocks( II + I*10 ), procs(4) )
                        END DO
                    END DO
                    ! GIVE EDGE BLOCKS OF 7TH ROW TO PROCS 3 & 4
                    ! leftmost 3 blocks to proc 3
                    CALL assign_blocks( blocks, procs(3), [61, 62, 63] )
                    ! rightmost 3 blocks to proc 4
                    CALL assign_blocks( blocks, procs(4), [68, 69, 70] )
                    ! GIVE CENTER BLOCKS OF 7TH ROW TO PROCS 5 & 6,
                    ! left 2 center blocks to proc 5
                    CALL assign_blocks( blocks, procs(5), [64, 65] )
                    ! right 2 center blocks to proc 6
                    CALL assign_blocks( blocks, procs(6), [66, 67] )
                    ! ASSIGN LAST 3 ROWS TO PROCS 5 & 6, DIVIDED IN HALF
                    DO I = 7, 9
                        DO II = 1, 5
                            CALL assign_block( blocks( II + I*10 ), procs(5) )
                        END DO
                        DO II = 6, 10
                            CALL assign_block( blocks( II + I*10 ), procs(6) )
                        END DO
                    END DO

                ELSE IF (NPROCS == 4) THEN
                    ! 10X10 BLOCKS, 4 PROCS
                    ! DIVIDE GRID INTO QUADRANTS

                    ! ASSIGN BOTTOM CORNERS TO PROCS 1 AND 2
                    DO I = 0, 4
                        DO II = 1, 5
                            CALL assign_block( blocks( II + I*10 ), procs(1) )
                        END DO
                        DO II = 6, 10
                            CALL assign_block( blocks( II + I*10 ), procs(2) )
                        END DO
                    END DO
                    ! ASSIGN TOP CORNERS TO PROCS 3 AND 4
                    DO I = 5, 9
                        DO II = 1, 5
                            CALL assign_block( blocks( II + I*10 ), procs(3) )
                        END DO
                        DO II = 6, 10
                            CALL assign_block( blocks( II + I*10 ), procs(4) )
                        END DO
                    END DO

                ELSE IF (NPROCS == 2) THEN
                    ! 10X10 BLOCKS, 2 PROCS
                    ! DIVIDE GRID IN HALF (TOP AND BOTTOM)

                    ! ASSIGN BOTTOM BLOCKS TO PROC 1
                    DO I = 0, 4
                        DO II = 1, 10
                            CALL assign_block( blocks( II + I*10 ), procs(1) )
                        END DO
                    END DO
                    ! ASSIGN TOP BLOCKS TO PROC 2
                    DO I = 5, 9
                        DO II = 1, 10
                            CALL assign_block( blocks( II + I*10 ), procs(2) )
                        END DO
                    END DO
```

```fortran
860
861              END IF
862
863          ELSE
864              !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
865              !!! DISTRIBUTE TO PROCESSOR WITH LEAST LOAD !!!!!!!!!!!!!!!!!!!!!
866              !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
867
868              write(*,*) " "
869              write(*,*) "Block ID assigned to Proc ID:"
870
871              ! LOOP THROUGH BLOCKS IN DECREASING ORDER OF SIZE
872              DO I = 1, NBLK
873                  ! sorted gives the indicies of blocks sorted by size
874                  b => blocks( sorted(I) )
875
876                  ! Reset minimum load
877                  MINLOAD = 1000000
878                  ! FIND CURRENT PROCESSOR WITH LEAST LOAD
879                  DO IPROC = 1, NPROCS
880                      p => procs(IPROC)
881
882                      IF (p%load<MINLOAD) THEN
883                          MINLOAD = p%load
884                          IMINLOAD = IPROC
885                      END IF
886                  END DO
887                  ! write block processor assignment
888                  write(*,*) b%ID, procs(IMINLOAD)%ID
889                  proclist(I) = procs(IMINLOAD)%ID
890                  locIDs(I) = procs(IMINLOAD)%NBLK+1
891
892                  ! ASSIGN BLOCK TO MIN. LOAD PROC
893                  CALL assign_block( b, procs(IMINLOAD) )
894              END DO
895
896              ! CALC LOAD BALANCE
897  20          FORMAT(10A13)
898              WRITE(*,*)
899              WRITE(*,*) 'Processor Load Balancing:'
900              WRITE(*,20) 'ID', 'LOAD BALANCE'
901
902              DO IPROC = 1, NPROCS
903                  procs(IPROC)%balance = DFLOAT( procs(IPROC)%load ) / DFLOAT( PLB )
904
905                  WRITE(*,*) procs(IPROC)%ID, procs(IPROC)%balance
906              END DO
907              WRITE(*,*)
908
909
910
911              do Iblk = 1, Nblk
912                  do i = 1, nblk
913                      if (sorted(I) == IBLK) then
914                          idsSort(Iblk) = locIds(I)
915                          procSort(Iblk) = proclist(I)
916                      end if
917                  end do
918              end do
919
920
921              ! write block amalgamation file
922              OPEN(UNIT=55,FILE = 'blockrebuild.dat',FORM='formatted')
923              write(55,*) "block, processor, local id"
924              do I = 1, NBLK
925                  write(55,*) I, procsort(I), IDsSort(I)
926              end do
927              CLOSE(55)
928
```

```fortran
                    OPEN(UNIT=65,FILE = 'procrebuild.dat',FORM='formatted')
                    do I = 1, NPRocs
                        write(65,*) procs(I)%NBLK
                    end do
                    CLOSE(65)

            END IF

        END SUBROUTINE dist_blocks

        SUBROUTINE assign_block(b, p)
            ! Assign block to given processor

            ! Block to assign (not list)
            TYPE(BLKTYPE), TARGET :: b
            ! Processor to assign to
            TYPE(PROCTYPE), TARGET :: p

            ! INCREMENT NUMBER OF BLOCKS ON PROC
            p%NBLK = p%NBLK + 1
            ! ADD BLOCK LOAD TO TOTAL PROCESSOR LOAD
            p%load = p%load + b%SIZE
            ! ADD BLOCK TO PROC
            p%blocks(p%NBLK) = b
            ! ADD BLOCK TO PROC
            p%blocks(p%NBLK) = b

        END SUBROUTINE assign_block

        SUBROUTINE assign_blocks(blocks, proc, IDs)
            ! Like assign_blocks, but assign multiple blocks

            ! all blocks
            TYPE(BLKTYPE), TARGET :: blocks(:)
            ! Processor to assign to
            TYPE(PROCTYPE), TARGET :: proc
            ! IDs of blocks to assign
            INTEGER :: IDs(:), I

            ! assign each block in IDs to proc
            DO I = 1, SIZE(IDs)
                CALL assign_block( blocks( IDS(I) ), proc )
            END DO
        END SUBROUTINE assign_blocks


        SUBROUTINE init_neighbor_procs(blocks, procs)
            ! Initialize neighbor processor information for each block

            ! BLOCK DATA TYPE
            TYPE(BLKTYPE), TARGET :: blocks(:)
            TYPE(BLKTYPE), POINTER :: bcur, bnbr
            ! PROCESSOR DATA TYPE
            TYPE(PROCTYPE), TARGET :: procs(:)
            TYPE(PROCTYPE), POINTER :: pcur, pnbr
            ! Neighbor information pointer
            TYPE(NBRTYPE), POINTER :: NBCUR, NBNBR, NPCUR
            ! COUNTER VARIABLES
                ! index of current processor, index of current proc's neighbor proc
            INTEGER :: IPCUR, IPNBR, IBCUR, IBNBR

            ! ASSIGN PROC INFORMATION TO PROC DATA TYPE LIST
            DO IPCUR = 1, NPROCS
                pcur => procs(IPCUR)

                ! ALL BLOCKS ASSIGNED TO CURRENT PROC ARE ON CURRENT PROC
                pcur%blocks%procID = pcur%ID
                ! DEFAULT ALL NEIGHBORS TO -1
                    ! (indicates boundary with no neighbor if not reassigned later)
```

```fortran
            DO IBCUR = 1, pcur%NBLK
                NPCUR => pcur%blocks(IBCUR)%NP

                NPCUR%N  = -1
                NPCUR%S  = -1
                NPCUR%E  = -1
                NPCUR%W  = -1
                NPCUR%NE = -1
                NPCUR%SE = -1
                NPCUR%SW = -1
                NPCUR%NW = -1
            END DO
        END DO

        ! INITIALIZE LOCAL PROCESSOR INDICIES OF BLOCKS
        DO IPCUR = 1, NPROCS
            pcur => procs(IPCUR)
            DO IBCUR = 1, pcur%NBLK
                bcur => pcur%blocks(IBCUR)
                bcur%NBLOC%N  = 0
                bcur%NBLOC%S  = 0
                bcur%NBLOC%E  = 0
                bcur%NBLOC%W  = 0
                bcur%NBLOC%NE = 0
                bcur%NBLOC%SE = 0
                bcur%NBLOC%SW = 0
                bcur%NBLOC%NW = 0
            END DO
        END DO

        ! FIND PROC WITH NEIGHBOR FOR EACH BLOCK
        DO IPCUR = 1, NPROCS
            pcur => procs(IPCUR)

            ! FOR EACH PROC, STEP THROUGH EACH CONTAINED BLOCK AND FIND NEIGHBORS
            DO IBCUR = 1, pcur%NBLK
                bcur => pcur%blocks(IBCUR)

                ! STEP THROUGH EACH NEIGHBOR PROCESSOR TO FIND NEIGHBOR BLOCK
                DO IPNBR = 1, NPROCS
                    pnbr => procs(IPNBR)

                    ! STEP THROUGH BLOCKS ON NEIGHBOR PROCESSORS
                    DO IBNBR = 1, pnbr%NBLK
                        bnbr => pnbr%blocks(IBNBR)

                        ! CHECK EACH FACE/CORNER FOR MATCH AND ASSIGN
                        ! (neighbor procID and local index of
                        !  neighbor block on neighbor proc)

                        ! NORTH
                        IF (bcur%NB%N == bnbr%ID) THEN
                            ! PROCESSOR CONTAINING NEIGHBOR BLOCK
                            bcur%NP%N = pnbr%ID
                            ! NEIGHBOR BLOCK LOCAL INDEX ON NEIGHBOR PROCESSOR
                                !(used to access neighbor on neighbor processor)
                            bcur%NBLOC%N = IBNBR

                            ! IF NEIGHBOR PROC IS DIFFERENT FROM CURRENT PROC,
                            ! COMMUNICATION WILL BE REQUIRED
                            ! (indicate processor boundary by making the block
                            !  neighbor number negative)
                            IF (pcur%ID /= pnbr%ID) THEN
                                bcur%NB%N = -bcur%NB%N
                            END IF
                        END IF
                        ! SOUTH
                        IF (bcur%NB%S == bnbr%ID) THEN
                            bcur%NP%S = pnbr%ID
```

```fortran
                                    bcur%NBLOC%S = IBNBR
                                    IF (pcur%ID /= pnbr%ID) THEN
                                        bcur%NB%S = -bcur%NB%S
                                    END IF
                                END IF
                                ! EAST
                                IF (bcur%NB%E == bnbr%ID) THEN
                                    bcur%NP%E = pnbr%ID
                                    bcur%NBLOC%E = IBNBR
                                    IF (pcur%ID /= pnbr%ID) THEN
                                        bcur%NB%E = -bcur%NB%E
                                    END IF
                                END IF
                                ! WEST
                                IF (bcur%NB%W == bnbr%ID) THEN
                                    bcur%NP%W = pnbr%ID
                                    bcur%NBLOC%W = IBNBR
                                    IF (pcur%ID /= pnbr%ID) THEN
                                        bcur%NB%W = -bcur%NB%W
                                    END IF
                                END IF
                                ! NORTH EAST
                                IF (bcur%NB%NE == bnbr%ID) THEN
                                    bcur%NP%NE = pnbr%ID
                                    bcur%NBLOC%NE = IBNBR
                                    IF (pcur%ID /= pnbr%ID) THEN
                                        bcur%NB%NE = -bcur%NB%NE
                                    END IF
                                END IF
                                ! SOUTH EAST
                                IF (bcur%NB%SE == bnbr%ID) THEN
                                    bcur%NP%SE = pnbr%ID
                                    bcur%NBLOC%SE = IBNBR
                                    IF (pcur%ID /= pnbr%ID) THEN
                                        bcur%NB%SE = -bcur%NB%SE
                                    END IF
                                END IF
                                ! SOUTH WEST
                                IF (bcur%NB%SW == bnbr%ID) THEN
                                    bcur%NP%SW = pnbr%ID
                                    bcur%NBLOC%SW = IBNBR
                                    IF (pcur%ID /= pnbr%ID) THEN
                                        bcur%NB%SW = -bcur%NB%SW
                                    END IF
                                END IF
                                ! NORTH WEST
                                IF (bcur%NB%NW == bnbr%ID) THEN
                                    bcur%NP%NW = pnbr%ID
                                    bcur%NBLOC%NW = IBNBR
                                    IF (pcur%ID /= pnbr%ID) THEN
                                        bcur%NB%NW = -bcur%NB%NW
                                    END IF
                                END IF
                            END DO
                        END DO
                    END DO
                END DO

!         10      FORMAT(10A12)
!             WRITE(*,*)
!             WRITE(*,*) 'Check proc neighbors'
!             WRITE(*,10) 'BLKID', 'NB%N', 'NP%N', 'NBLOC%N'
!             DO IPCUR = 1, NPROCS
!                 pcur => procs(IPCUR)
!                 WRITE(*,*) 'Proc:', pcur%ID
!                 DO IBCUR = 1, pcur%NBLK
!                     bcur => pcur%blocks(IBCUR)
!                     WRITE(*,*) bcur%ID, bcur%NB%N, bcur%NP%N, bcur%NBLOC%N
!                 END DO
```

```fortran
!            END DO

     END SUBROUTINE init_neighbor_procs

     SUBROUTINE init_mesh(b)
         ! Create xprime/yprime non-uniform grid, then rotate by angle 'rot'.
         ! Allocate arrays for node parameters (i.e. temperature, cell area, etc)

         ! BLOCK DATA TYPE
         TYPE(BLKTYPE), TARGET :: b(:)
         TYPE(MESHTYPE), POINTER :: m
         INTEGER :: IBLK, I, J

         DO IBLK = 1, NBLK

             m => b(IBLK)%mesh

             ! ALLOCATE MESH INFORMATION
                 ! ADD EXTRA INDEX AT BEGINNING AND END FOR GHOST NODES
             ALLOCATE( m%xp(  0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%yp(  0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%x(   0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%y(   0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%T(   0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%Ttmp(0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%dt(  0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%V2nd(0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%term(0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%Ayi( 0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%Axi( 0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%Ayj( 0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%Axj( 0:IMAXBLK+1,   0:JMAXBLK+1) )
             ALLOCATE( m%V(   0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%yPP( 0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%yNP( 0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%yNN( 0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%yPN( 0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%xNN( 0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%xPN( 0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%xPP( 0:IMAXBLK,     0:JMAXBLK  ) )
             ALLOCATE( m%xNP( 0:IMAXBLK,     0:JMAXBLK  ) )

             ! STEP THROUGH LOCAL INDICIES OF EACH BLOCK
             DO J = 0, JMAXBLK+1
                 DO I = 0, IMAXBLK+1
                     ! MAKE SQUARE GRID
                         ! CONVERT FROM LOCAL TO GLOBAL INDEX:
                             ! Iglobal = Block%IMIN + (Ilocal - 1)
                     m%xp(I, J) = COS( 0.5D0 * PI * DFLOAT(IMAX - ( b(IBLK)%IMIN + I - 1) ) / DFLOAT(IMAX - 1) )
                     m%yp(I, J) = COS( 0.5D0 * PI * DFLOAT(JMAX - ( b(IBLK)%JMIN + J - 1) ) / DFLOAT(JMAX - 1) )

                     ! ROTATE GRID
                     m%x(I, J) = m%xp(I, J) * COS(rot) + (1.D0 - m%yp(I, J) ) * SIN(rot)
                     m%y(I, J) = m%yp(I, J) * COS(rot) + (        m%xp(I, J) ) * SIN(rot)
                 END DO
             END DO
         END DO
     END SUBROUTINE init_mesh

     SUBROUTINE init_temp(blocks)
         ! Initialize temperature across mesh with dirichlet BCs
         ! or constant temperature BCs for OPT=0

         ! BLOCK DATA TYPE
         TYPE(BLKTYPE), TARGET  :: blocks(:)
         TYPE(BLKTYPE), POINTER :: b
         TYPE(MESHTYPE), POINTER :: m
         TYPE(NBRTYPE), POINTER :: NB
         INTEGER :: IBLK, I, J
```

```fortran
      DO IBLK = 1, NBLK
          b => blocks(IBLK)
          m => blocks(IBLK)%mesh
          NB => blocks(IBLK)%NB
          ! FIRST, INITIALIZE ALL POINT TO INITIAL TEMPERATURE (T0)
          m%T(0:IMAXBLK+1, 0:JMAXBLK+1) = T0
          ! THEN, INITIALIZE BOUNDARIES DIRICHLET B.C.
          IF (OPT /= 0) THEN

              ! DIRICHLET B.C.
              ! face on north boundary
              IF (NB%N == BND) THEN
                  DO I = 1, IMAXBLK
                      m%T(I, JMAXBLK) = 5.D0 * (SIN(PI * m%xp(I, JMAXBLK)) + 1.D0)
                  END DO
              END IF
              IF (NB%S == BND) THEN
                  DO I = 1, IMAXBLK
                      m%T(I, 1) = ABS(COS(PI * m%xp(I, 1))) + 1.D0
                  END DO
              END IF
              IF (NB%E == BND) THEN
                  DO J = 1, JMAXBLK
                      m%T(IMAXBLK, J) = 3.D0 * m%yp(IMAXBLK, J) + 2.D0
                  END DO
              END IF
              IF (NB%W == BND) THEN
                  DO J = 1, JMAXBLK
                      m%T(1, J) = 3.D0 * m%yp(1, J) + 2.D0
                  END DO
              END IF

          ELSE

              ! DEBUG BCS
              IF (NB%N == BND) THEN
                  DO I = 1, IMAXBLK
                      m%T(I, JMAXBLK) = TDEBUG
                  END DO
              END IF
              IF (NB%S == BND) THEN
                  DO I = 1, IMAXBLK
                      m%T(I, 1) = TDEBUG
                  END DO
              END IF
              IF (NB%E == BND) THEN
                  DO J = 1, JMAXBLK
                      m%T(IMAXBLK, J) = TDEBUG
                  END DO
              END IF
              IF (NB%W == BND) THEN
                  DO J = 1, JMAXBLK
                      m%T(1, J) = TDEBUG
                  END DO
              END IF
          END IF
      END DO
  END SUBROUTINE init_temp

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !!! INITIALIZE SOLUTION AFTER RESTART FILE READ IN !!!!!!!!!!!!!!!!!!!!!!!!!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  SUBROUTINE set_block_bounds(blocks)
      ! Calculate iteration bounds for each block to avoid overwriting BCs.
      ! Call after reading in mesh data from restart file

      TYPE(BLKTYPE), TARGET :: blocks(:)
```

```fortran
        TYPE(BLKTYPE), POINTER :: b
        TYPE(NBRTYPE), POINTER :: NB
        INTEGER :: IBLK, I, J

        DO IBLK = 1, NBLK
            b => blocks(IBLK)
            NB => b%NB

            ! Set iteration bounds of each block to preserve BCs
                ! south and west boundaries:
                    ! interior: iminloc, jminloc = 0 (use ghost)
                    ! boundary: iminloc, jminloc = 2 (1st index is BC)
                ! north and east boundaries:
                    ! interior: imaxloc, jmaxloc = maxblk (use ghost)
                    ! boundary: imaxloc, jmaxloc = maxblk-1 (max index is BC)

            ! NORTH
            IF (NB%N > 0) THEN
                ! Interior faces have positive ID neighbors
                b%JMAXLOC = JMAXBLK
            ELSE
                ! At North Boundary
                b%JMAXLOC = JMAXBLK - 1
            END IF

            ! EAST
            IF (NB%E > 0) THEN
                ! Interior
                b%IMAXLOC = IMAXBLK
            ELSE
                ! At east Boundary
                b%IMAXLOC = IMAXBLK - 1
            END IF

            ! SOUTH
            IF (NB%S > 0) THEN
                ! Interior
                b%JMINLOC = 0
            ELSE
                ! At south Boundary
                b%JMINLOC = 1
                ! boundary for updating temperature (dont update BC)
                b%JMINUPD = 2
            END IF

            ! WEST
            IF (NB%W > 0) THEN
                ! Interior
                b%IMINLOC = 0
            ELSE
                ! At west Boundary
                b%IMINLOC = 1
                b%IMINUPD = 2
            END IF
        END DO
    END SUBROUTINE set_block_bounds

    SUBROUTINE make_link(NB, list, nbrl, ID)
        ! make a single link in a linked list
        ! NB --> neighbor information (i.e. NB%N)
        ! list --> the neighbor linked list (i.e. nbrlists%N)
        ! nbrl --> pointer for neighbor linked list (i.e. nbrl%N)
        !          needs to be stored throughout the loop
        ! ID --> the block id to assign

        ! Neighbor information pointer
        INTEGER :: NB, ID
        ! Linked lists of neighbor communication instructions
        TYPE(LNKLIST), POINTER :: list
```

```fortran
        TYPE(LNKLIST), POINTER :: nbrl

        IF ( .NOT. ASSOCIATED(list) ) THEN
            ! Allocate linked list if it hasnt been accessed yet
            ALLOCATE(list)
            ! Pointer linked list that will help iterate through the
            ! primary list in this loop
            nbrl => list
        ELSE
            ! linked list already allocated (started).  Allocate next
            ! link as assign current block to it
            ALLOCATE(nbrl%next)
            nbrl => nbrl%next
        END IF

        ! associate this linked list entry with the current block
        nbrl%ID = ID
        ! break link to pre-existing pointer target.  We will
        ! allocated this target later as the next item in the linked list
        NULLIFY(nbrl%next)

    END SUBROUTINE make_link

    SUBROUTINE link_type(NB, list, nbrl, mpi, mpil, ID)
        ! make a single link in a linked list for a neighbor either on same
        ! processor or different processor
        ! NB --> neighbor information (i.e. NB%N)
        ! list --> the neighbor linked list (i.e. nbrlists%N)
        ! nbrl --> pointer for neighbor linked list (i.e. nbrl%N)
        !          needs to be stored throughout the loop
        ! mpi, mpil --> same as list/nbrl but for faces on other procs
        ! ID --> the block id to assign

        ! Neighbor information pointer
        INTEGER :: NB, ID
        ! Linked lists of neighbor communication instructions
        TYPE(LNKLIST), POINTER :: list, nbrl, mpi, mpil

        ! If block face is internal, add it to appropriate linked list
        ! for internal faces of a certian face (i.e. north).
        IF (NB > 0) THEN
            ! NEIGHBOR IS ON SAME PROCESSOR
            CALL make_link(NB, list, nbrl, ID)
        ELSE IF (NB < 0) THEN
            ! NEIGHBOR IS ON DIFFERENT PROCESSOR
            CALL make_link(NB, mpi, mpil, ID)
        END IF
    END SUBROUTINE link_type


    SUBROUTINE init_linklists(blocks, nbrlists, mpilists)
        ! Create linked lists governing block boundary communication.
        ! Separate list for each neighbor type so we can avoid logic when
        ! updating ghost nodes.

        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), TARGET :: blocks(:)
        ! Neighbor information pointer
        TYPE(NBRTYPE), POINTER :: NB
        ! Linked lists of neighbor communication instructions
        TYPE(NBRLIST) :: nbrlists
        TYPE(NBRLIST) :: nbrl
        TYPE(NBRLIST) :: mpilists
        TYPE(NBRLIST) :: mpil
        INTEGER :: IBLK

        ! INITIALIZE LINKED LISTS (HPC1 REQUIRES THIS)
        NULLIFY(nbrlists%N)
        NULLIFY(nbrlists%S)
```

```fortran
        NULLIFY(nbrlists%E)
        NULLIFY(nbrlists%W)
        NULLIFY(nbrlists%NW)
        NULLIFY(nbrlists%NE)
        NULLIFY(nbrlists%SE)
        NULLIFY(nbrlists%SW)

        NULLIFY(mpilists%N)
        NULLIFY(mpilists%S)
        NULLIFY(mpilists%E)
        NULLIFY(mpilists%W)
        NULLIFY(mpilists%NW)
        NULLIFY(mpilists%NE)
        NULLIFY(mpilists%SE)
        NULLIFY(mpilists%SW)

        DO IBLK = 1, MYNBLK
            NB => blocks(IBLK)%NB

            ! NORTH
!             IF (NB%N > 0) THEN
!                 ! NEIGHBOR IS ON SAME PROCESSOR
!                 CALL make_link(NB%N, nbrlists%N, nbrl%N, ID%N)
!             ELSE IF (NB%N < 0) THEN
!                 ! NEIGHBOR IS ON DIFFERENT PROCESSOR
!                 CALL make_link(NB%N, mpi%N, mpil%N, ID%N)
!             END IF
            CALL link_type(NB%N, nbrlists%N, nbrl%N, mpilists%N, mpil%N, IBLK)
            ! SOUTH
            CALL link_type(NB%S, nbrlists%S, nbrl%S, mpilists%S, mpil%S, IBLK)
            ! EAST
            CALL link_type(NB%E, nbrlists%E, nbrl%E, mpilists%E, mpil%E, IBLK)
            ! WEST
            CALL link_type(NB%W, nbrlists%W, nbrl%W, mpilists%W, mpil%W, IBLK)
            ! NORTH EAST
            CALL link_type(NB%NE, nbrlists%NE, nbrl%NE, mpilists%NE, mpil%NE, IBLK)
            ! SOUTH EAST
            CALL link_type(NB%SE, nbrlists%SE, nbrl%SE, mpilists%SE, mpil%SE, IBLK)
            ! SOUTH WEST
            CALL link_type(NB%SW, nbrlists%SW, nbrl%SW, mpilists%SW, mpil%SW, IBLK)
            ! NORTH WEST
            CALL link_type(NB%NW, nbrlists%NW, nbrl%NW, mpilists%NW, mpil%NW, IBLK)
        END DO

!         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!         if (myid == 2) then
!             write(*,*) "proc ", myid, "sw mpi list"
!             nbrl%N => nbrlists%N
!             mpil%sw => mpilists%sw
!             do
!                 IF ( .NOT. ASSOCIATED(mpil%sw) ) EXIT
!                 write(*,*) blocks(mpil%sw%ID)%ID
!                 mpil%sw => mpil%sw%next
!             end do
!         end if


    END SUBROUTINE init_linklists

    SUBROUTINE update_ghosts_sameproc(b, nbrlists)
        ! Update ghost nodes of each block based on neightbor linked list for
        ! neighbors on same processor as current block.
        ! Ghost nodes contain solution from respective block face/corner
        ! neighbor for use in current block solution.

        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), TARGET :: b(:)
        ! temperature information pointers for ghost and neighbor nodes
        REAL(KIND=8), POINTER, DIMENSION(:, :) :: Tgh, Tnb
```

```fortran
        ! Linked lists of neighbor communication instructions
        TYPE(NBRLIST) :: nbrlists
        TYPE(NBRLIST) :: nbrl
        ! iteration parameters, index of neighbor
        INTEGER :: I, J, INBR

        !!! FACES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        ! NORTH FACE GHOST NODES
        nbrl%N => nbrlists%N
        ! Step through linked list of north faces with ghosts until end of list
        DO
            ! If next link in list doesnt exist (end of list), stop loop
            IF ( .NOT. ASSOCIATED(nbrl%N) ) EXIT

            ! Otherwise, assign neighbor values to all ghost nodes:

            ! TEMPERATURE OF CURRENT BLOCK (CONTAINS GHOST NODES)
                ! (identified by linked list id)
            Tgh => b( nbrl%N%ID )%mesh%T

            ! index of north neighbor
            INBR = b( nbrl%N%ID )%NBLOC%N
            ! TEMPERATURE OF NEIGHBOR BLOCK (UPDATE GHOSTS WITH THIS)
            Tnb => b( INBR )%mesh%T

            DO I = 1, IMAXBLK
                ! NORTH FACE GHOST NODE TEMPERATURE IS EQUAL TO TEMPERATURE OF
                ! SECOND-FROM-SOUTH FACE OF NORTH NEIGHBOR
                ! (Remember face nodes are shared between blocks)
                Tgh(I, JMAXBLK+1) = Tnb(I, 2)
            END DO
            ! switch pointer to next link in list
            nbrl%N => nbrl%N%next
        END DO

        ! SOUTH FACE GHOST NODES
        nbrl%S => nbrlists%S
        DO
            IF ( .NOT. ASSOCIATED(nbrl%S) ) EXIT
            Tgh => b( nbrl%S%ID )%mesh%T
            INBR = b( nbrl%S%ID )%NBLOC%S
            Tnb => b( INBR )%mesh%T

            DO I = 1, IMAXBLK
                ! ADD NORTH FACE OF SOUTH NEIGHBOR TO CURRENT SOUTH FACE GHOSTS
                Tgh(I, 0) = Tnb(I, JMAXBLK-1)
            END DO
            nbrl%S => nbrl%S%next
        END DO

        ! EAST FACE GHOST NODES
        nbrl%E => nbrlists%E
        DO
            IF ( .NOT. ASSOCIATED(nbrl%E) ) EXIT
            Tgh => b( nbrl%E%ID )%mesh%T
            INBR = b( nbrl%E%ID )%NBLOC%E
            Tnb => b( INBR )%mesh%T

            DO J = 1, JMAXBLK
                ! ADD WEST FACE OF EAST NEIGHBOR TO CURRENT WEST FACE GHOSTS
                Tgh(IMAXBLK+1, J) = Tnb(2, J)
            END DO
            nbrl%E => nbrl%E%next
        END DO

        ! WEST FACE GHOST NODES
        nbrl%W => nbrlists%W
        DO
```

```fortran
            IF ( .NOT. ASSOCIATED(nbrl%W) ) EXIT
            Tgh => b( nbrl%W%ID )%mesh%T
            INBR = b( nbrl%W%ID )%NBLOC%W
            Tnb => b( INBR )%mesh%T

            DO J = 1, JMAXBLK
                ! ADD EAST FACE OF WEST NEIGHBOR TO CURRENT EAST FACE GHOSTS
                Tgh(0, J) = Tnb(IMAXBLK-1, J)
            END DO
            nbrl%W => nbrl%W%next
        END DO

        !!! CORNERS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        ! NORTH EAST CORNER GHOST NODES
        nbrl%NE => nbrlists%NE
        DO
            IF ( .NOT. ASSOCIATED(nbrl%NE) ) EXIT
            Tgh => b( nbrl%NE%ID )%mesh%T
            INBR = b( nbrl%NE%ID )%NBLOC%NE
            Tnb => b( INBR )%mesh%T
            ! ADD SW CORNER OF NE NEIGHBOR TO CURRENT NE CORNER GHOSTS
            Tgh(IMAXBLK+1, JMAXBLK+1) = Tnb(2, 2)
            nbrl%NE => nbrl%NE%next
        END DO

        ! SOUTH EAST CORNER GHOST NODES
        nbrl%SE => nbrlists%SE
        DO
            IF ( .NOT. ASSOCIATED(nbrl%SE) ) EXIT
            Tgh => b( nbrl%SE%ID )%mesh%T
            INBR = b( nbrl%SE%ID )%NBLOC%SE
            Tnb => b( INBR )%mesh%T
            ! ADD NW CORNER OF SE NEIGHBOR TO CURRENT SE CORNER GHOSTS
            Tgh(IMAXBLK+1, 0) = Tnb(2, JMAXBLK-1)
            nbrl%SE => nbrl%SE%next
        END DO

        ! SOUTH WEST CORNER GHOST NODES
        nbrl%SW => nbrlists%SW
        DO
            IF ( .NOT. ASSOCIATED(nbrl%SW) ) EXIT
            Tgh => b( nbrl%SW%ID )%mesh%T
            INBR = b( nbrl%SW%ID )%NBLOC%SW
            Tnb => b( INBR )%mesh%T
            ! ADD NE CORNER OF SW NEIGHBOR TO CURRENT SW CORNER GHOSTS
            Tgh(0, 0) = Tnb(IMAXBLK-1, JMAXBLK-1)
            nbrl%SW => nbrl%SW%next
        END DO

        ! NORTH WEST CORNER GHOST NODES
        nbrl%NW => nbrlists%NW
        DO
            IF ( .NOT. ASSOCIATED(nbrl%NW) ) EXIT
            Tgh => b( nbrl%NW%ID )%mesh%T
            INBR = b( nbrl%NW%ID )%NBLOC%NW
            Tnb => b( INBR )%mesh%T
            ! ADD SE CORNER OF NW NEIGHBOR TO CURRENT NW CORNER GHOSTS
            Tgh(0, JMAXBLK+1) = Tnb(IMAXBLK-1, 2)
            nbrl%NW => nbrl%NW%next
        END DO
    END SUBROUTINE update_ghosts_sameproc

    SUBROUTINE update_ghosts_diffproc_send(blocks, mpilists)
        ! Gather information on different processors to update current ghosts.
        ! send ghost info to neighbor processor to buffer its ghost nodes
        ! store in buffers and send via MPI.  Buffers will be MPI recieved
        ! and distributed to corresponding blocks
```

```fortran
      ! BLOCK DATA TYPE
      TYPE(BLKTYPE), TARGET :: blocks(:)
      TYPE(BLKTYPE), POINTER :: b
      ! temperature information pointers for ghost and neighbor nodes
      REAL(KIND=8), POINTER, DIMENSION(:, :) :: Tgh, Tnb
      ! buffers to store temperature information for I/J faces, and corners
      REAL(KIND=8) :: bufferI(IMAXBLK), bufferJ(JMAXBLK), buffer
      ! Linked lists of neighbor communication instructions
      TYPE(NBRLIST) :: mpilists
      TYPE(NBRLIST) :: mpil
      ! iteration parameters, index of neighbor, communication tag, destination
      INTEGER :: I, J, INBR, tag, dest
      ! counts number of sends in certian direction by proc
      INTEGER :: sends

      !!! FACES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      ! NORTH FACE GHOST INFO
      ! Send north face data to north neighbor to put in neighbor's ghost nodes
      mpil%N => mpilists%N
      DO
          IF ( .NOT. ASSOCIATED(mpil%N) ) EXIT

          ! CURRENT BLOCK
          b => blocks( mpil%N%ID )
          ! SEND FACE INFORMATION FOR THIS BLOCK TO NEIGHBORS ON OTHER PROCS
          DO I = 1, IMAXBLK
              ! FILL BUFFER WITH TEMPERATURE FROM THIS BLOCK FOR GHOSTS OF OTHER BLOCK
              ! (indexing is opposite from project 3 as we are now sending
              ! ghost info to neighbor rather than recieving it from them here)
              bufferI(I) = b%mesh%T(I, JMAXBLK-1)
          END DO

          ! FIND DESITNATION OF GHOST INFO (proc id of neighbor)
          dest = b%NP%N
          ! MAKE UNIQUE TAG
          CALL make_mpi_tag(NBND, b%ID, tag)
          ! SEND INFO TO NEIGHBOR PROC AND CONTINUE OPERATIONS WITHOUT CONFIRMATION
          CALL MPI_Isend(bufferI, IMAXBLK, MPI_REAL8, dest, tag, &
                          MPI_COMM_WORLD, REQUEST, IERROR)

          mpil%N => mpil%N%next
      END DO

      ! SOUTH FACE GHOST NODES
      mpil%S => mpilists%S
      DO
          IF ( .NOT. ASSOCIATED(mpil%S) ) EXIT

          b => blocks( mpil%S%ID )
          DO I = 1, IMAXBLK
              bufferI(I) = b%mesh%T(I, 2)
          END DO

          dest = b%NP%S
          CALL make_mpi_tag(SBND, b%ID, tag)
          CALL MPI_Isend(bufferI, IMAXBLK, MPI_REAL8, dest, tag, &
                          MPI_COMM_WORLD, REQUEST, IERROR)

          mpil%S => mpil%S%next
      END DO

      ! EAST FACE GHOST NODES
      mpil%E => mpilists%E
      DO
          IF ( .NOT. ASSOCIATED(mpil%E) ) EXIT

          b => blocks( mpil%E%ID )
          DO J = 1, JMAXBLK
              bufferJ(J) = b%mesh%T(IMAXBLK-1, J)
```

```fortran
              END DO

          dest = b%NP%E
          CALL make_mpi_tag(EBND, b%ID, tag)
          CALL MPI_Isend(bufferJ, JMAXBLK, MPI_REAL8, dest, tag, &
                         MPI_COMM_WORLD, REQUEST, IERROR)

          mpil%E => mpil%E%next
      END DO

      ! WEST FACE GHOST NODES
      mpil%W => mpilists%W
      DO
          IF ( .NOT. ASSOCIATED(mpil%W) ) EXIT

          b => blocks( mpil%W%ID )
          DO J = 1, JMAXBLK
              bufferJ(J) = b%mesh%T(2, J)
          END DO

          dest = b%NP%W
          CALL make_mpi_tag(WBND, b%ID, tag)
          CALL MPI_Isend(bufferJ, JMAXBLK, MPI_REAL8, dest, tag, &
                         MPI_COMM_WORLD, REQUEST, IERROR)

!             !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!             if (myid == 3 .and. b%ID == 4) then
!                 write(*,*)
!                 write(*,*) "send east ghosts from: ", b%ID
!                 write(*,*) "buffer values: ", bufferJ(2)
!             end if

          mpil%W => mpil%W%next
      END DO

      ! NORTH EAST CORNER GHOST NODES
      mpil%NE => mpilists%NE
      DO
          IF ( .NOT. ASSOCIATED(mpil%NE) ) EXIT

          b => blocks( mpil%NE%ID )
          buffer = b%mesh%T(IMAXBLK-1, JMAXBLK-1)

          dest = b%NP%NE
          CALL make_mpi_tag(NEBND, b%ID, tag)
          CALL MPI_Isend(buffer, 1, MPI_REAL8, dest, tag, &
                         MPI_COMM_WORLD, REQUEST, IERROR)

          mpil%NE => mpil%NE%next
      END DO

      ! SOUTH EAST CORNER GHOST NODES
      mpil%SE => mpilists%SE
      DO
          IF ( .NOT. ASSOCIATED(mpil%SE) ) EXIT

          b => blocks( mpil%SE%ID )
          buffer = b%mesh%T(IMAXBLK-1, 2)

          dest = b%NP%SE
          CALL make_mpi_tag(SEBND, b%ID, tag)
          CALL MPI_Isend(buffer, 1, MPI_REAL8, dest, tag, &
                         MPI_COMM_WORLD, REQUEST, IERROR)

          mpil%SE => mpil%SE%next
      END DO

      ! SOUTH WEST CORNER GHOST NODES
      mpil%SW => mpilists%SW
```

```fortran
        DO
            IF ( .NOT. ASSOCIATED(mpil%SW) ) EXIT

            b => blocks( mpil%SW%ID )
            buffer = b%mesh%T(2, 2)

            dest = b%NP%SW
            CALL make_mpi_tag(SWBND, b%ID, tag)
            CALL MPI_Isend(buffer, 1, MPI_REAL8, dest, tag, &
                            MPI_COMM_WORLD, REQUEST, IERROR)
            mpil%SW => mpil%SW%next
        END DO

        ! NORTH WEST CORNER GHOST NODES
        mpil%NW => mpilists%NW
        DO
            IF ( .NOT. ASSOCIATED(mpil%NW) ) EXIT

            b => blocks( mpil%NW%ID )
            buffer = b%mesh%T(2, JMAXBLK-1)

            dest = b%NP%NW
            CALL make_mpi_tag(NWBND, b%ID, tag)
            CALL MPI_Isend(buffer, 1, MPI_REAL8, dest, tag, &
                            MPI_COMM_WORLD, REQUEST, IERROR)
            mpil%NW => mpil%NW%next
        END DO

    END SUBROUTINE update_ghosts_diffproc_send

    SUBROUTINE update_ghosts_diffproc_recv(blocks, mpilists)
        ! Recieve information on different processors to update current ghosts.
        ! store in buffers and send via MPI.  Buffers will be MPI recieved
        ! and distributed to corresponding blocks

        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), TARGET :: blocks(:)
        TYPE(BLKTYPE), POINTER :: b
        ! temperature information pointers for ghost and neighbor nodes
        REAL(KIND=8), POINTER, DIMENSION(:, :) :: Tgh, Tnb
        ! buffers to store temperature information for I/J faces, and corners
        REAL(KIND=8) :: bufferI(IMAXBLK), bufferJ(JMAXBLK), buffer
        ! Linked lists of neighbor communication instructions
        TYPE(NBRLIST) :: mpilists
        TYPE(NBRLIST) :: mpil
        ! iteration parameters, index of neighbor, communication tag, source proc id
        INTEGER :: I, J, INBR, tag, src
        ! counts number of sends in certian direction by proc
        INTEGER :: sends

        !!! FACES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        ! NORTH FACE GHOST NODES
        ! get info sent by north neighbor and add to ghost node
        mpil%N => mpilists%N
        DO
            IF ( .NOT. ASSOCIATED(mpil%N) ) EXIT

            ! CURRENT BLOCK
            b => blocks( mpil%N%ID )
            ! SOURCE PROCESSOR ID
            src = b%NP%N
            ! TAG OF NEIGHBOR IS OPPOSITE OF THIS FACE
            CALL make_mpi_tag(SBND, ABS(b%NB%N), tag)
            ! GET INFO FROM SOURCE PROCESSOR
            CALL MPI_RECV(bufferI, IMAXBLK, MPI_REAL8, src, tag, &
                MPI_COMM_WORLD, STATUS, IERROR)

            ! ASSIGN GHOST INFORMATION
            DO I = 1, IMAXBLK
```

```fortran
                b%mesh%T(I, JMAXBLK+1) =  bufferI(I)
            END DO

            mpil%N => mpil%N%next
        END DO

        ! SOUTH FACE GHOST NODES
        mpil%S => mpilists%S
        DO
            IF ( .NOT. ASSOCIATED(mpil%S) ) EXIT

            b => blocks( mpil%S%ID )
            src = b%NP%S
            CALL make_mpi_tag(NBND, ABS(b%NB%S), tag)
            CALL MPI_RECV(bufferI, IMAXBLK, MPI_REAL8, src, tag, &
                MPI_COMM_WORLD, STATUS, IERROR)
            DO I = 1, IMAXBLK
                b%mesh%T(I, 0) =  bufferI(I)
            END DO

            mpil%S => mpil%S%next
        END DO

        ! EAST FACE GHOST NODES
        mpil%E => mpilists%E
        DO
            IF ( .NOT. ASSOCIATED(mpil%E) ) EXIT

            b => blocks( mpil%E%ID )
            src = b%NP%E
            CALL make_mpi_tag(WBND, ABS(b%NB%E), tag)
            CALL MPI_RECV(bufferJ, JMAXBLK, MPI_REAL8, src, tag, &
                MPI_COMM_WORLD, STATUS, IERROR)

!                !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                if (myid == 0 .and. b%ID == 3) then
!                    write(*,*)
!                    write(*,*) "recieve east ghosts for: ", b%ID
!                    write(*,*) "buffer values: ", bufferJ(2)
!                end if

            DO J = 1, JMAXBLK
                b%mesh%T(IMAXBLK+1, J) = bufferJ(J)
            END DO

            mpil%E => mpil%E%next
        END DO

        ! WEST FACE GHOST NODES
        mpil%W => mpilists%W
        DO
            IF ( .NOT. ASSOCIATED(mpil%W) ) EXIT

            b => blocks( mpil%W%ID )
            src = b%NP%W
            CALL make_mpi_tag(EBND, ABS(b%NB%W), tag)
            CALL MPI_RECV(bufferJ, JMAXBLK, MPI_REAL8, src, tag, &
                MPI_COMM_WORLD, STATUS, IERROR)
            DO J = 1, JMAXBLK
                b%mesh%T(0, J) = bufferJ(J)
            END DO

            mpil%W => mpil%W%next
        END DO

        ! NORTH EAST CORNER GHOST NODES
        mpil%NE => mpilists%NE
        DO
            IF ( .NOT. ASSOCIATED(mpil%NE) ) EXIT
```

```fortran
                b => blocks( mpil%NE%ID )
                src = b%NP%NE
                CALL make_mpi_tag(SWBND, ABS(b%NB%NE), tag)
                CALL MPI_RECV(buffer, 1, MPI_REAL8, src, tag, &
                    MPI_COMM_WORLD, STATUS, IERROR)
                b%mesh%T(IMAXBLK+1, JMAXBLK+1) = buffer

                mpil%NE => mpil%NE%next
            END DO

            ! SOUTH EAST CORNER GHOST NODES
            mpil%SE => mpilists%SE
            DO
                IF ( .NOT. ASSOCIATED(mpil%SE) ) EXIT

                b => blocks( mpil%SE%ID )
                src = b%NP%SE
                CALL make_mpi_tag(NWBND, ABS(b%NB%SE), tag)
                CALL MPI_RECV(buffer, 1, MPI_REAL8, src, tag, &
                    MPI_COMM_WORLD, STATUS, IERROR)
                b%mesh%T(IMAXBLK+1, 0) = buffer

                mpil%SE => mpil%SE%next
            END DO

            ! SOUTH WEST CORNER GHOST NODES
            mpil%SW => mpilists%SW
            DO
                IF ( .NOT. ASSOCIATED(mpil%SW) ) EXIT

                b => blocks( mpil%SW%ID )
                src = b%NP%SW
                CALL make_mpi_tag(NEBND, ABS(b%NB%SW), tag)
                CALL MPI_RECV(buffer, 1, MPI_REAL8, src, tag, &
                    MPI_COMM_WORLD, STATUS, IERROR)
                b%mesh%T(0, 0) = buffer

                mpil%SW => mpil%SW%next
            END DO

            ! NORTH WEST CORNER GHOST NODES
            mpil%NW => mpilists%NW
            DO
                IF ( .NOT. ASSOCIATED(mpil%NW) ) EXIT

                b => blocks( mpil%NW%ID )
                src = b%NP%NW
                CALL make_mpi_tag(SEBND, ABS(b%NB%NW), tag)

                CALL MPI_RECV(buffer, 1, MPI_REAL8, src, tag, &
                    MPI_COMM_WORLD, STATUS, IERROR)
                b%mesh%T(0, JMAXBLK+1) = buffer

                mpil%NW => mpil%NW%next
            END DO

    END SUBROUTINE update_ghosts_diffproc_recv

    SUBROUTINE make_mpi_tag(dir, srcblk, tag)
        ! Make unique tag for mpi send/revieve.  Sends are always from one
        ! proc to another, so we just need unique tags for each block in each
        ! direction.
        ! Accomplish this by making a unique number for each direction and
        ! concatenating the global block number of the sending block to it

        ! dir --> direction of send
        ! srcblk --> global id of block sending information
```

```fortran
        INTEGER :: dir, srcblk, tag
        CHARACTER(len=1) :: dirstr
        CHARACTER(len=25) :: srcstr
        CHARACTER(LEN=50) :: tagstr

        ! CONVERT INTEGERS TO STRINGS
        WRITE(dirstr, '(I1)') dir
        WRITE(srcstr, *) srcblk

        ! CONCATENATE STRINGS INTO INTEGER VALUE
        ! (direction, then block number)
        ! adjust right and left so numbers line up
        tagstr = ADJUSTR(TRIM(dirstr)) // ADJUSTL(TRIM(srcstr))
        ! CONVERT TO INTEGER
        READ(tagstr, *) tag

    END SUBROUTINE make_mpi_tag


    SUBROUTINE calc_cell_params(blocks)
        ! calculate areas for secondary fluxes and constant terms in heat
        ! treansfer eqn. Call after reading mesh data from restart file

        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), TARGET :: blocks(:)
        TYPE(MESHTYPE), POINTER :: m
        INTEGER :: IBLK, I, J
        ! Areas used in counter-clockwise trapezoidal integration to get
        ! x and y first-derivatives for center of each cell (Green's thm)
        REAL(KIND=8) :: Ayi_half, Axi_half, Ayj_half, Axj_half

        DO IBLK = 1, MYNBLK
            m => blocks(IBLK)%mesh

            DO J = 0, JMAXBLK+1
                DO I = 0, IMAXBLK+1
                    ! CALC CELL VOLUME
                        ! cross product of cell diagonals p, q
                        ! where p has x,y components px, py and q likewise.
                        ! Thus, p cross q = px*qy - qx*py
                        ! where, px = x(i+1,j+1) - x(i,j), py = y(i+1,j+1) - y(i,j)
                        ! and    qx = x(i,j+1) - x(i+1,j), qy = y(i,j+1) - y(i+1,j)
                    m%V(I,J) = ABS( ( m%x(I+1,J+1) - m%x(I,  J) ) &
                                  * ( m%y(I,  J+1) - m%y(I+1,J) ) &
                                  - ( m%x(I,  J+1) - m%x(I+1,J) ) &
                                  * ( m%y(I+1,J+1) - m%y(I,  J) ) )
                END DO
            END DO

            ! CALC CELL AREAS (FLUXES) IN J-DIRECTION
            DO J = 0, JMAXBLK+1
                DO I = 0, IMAXBLK
                    m%Axj(I,J) = m%x(I+1,J) - m%x(I,J)
                    m%Ayj(I,J) = m%y(I+1,J) - m%y(I,J)
                END DO
            END DO
            ! CALC CELL AREAS (FLUXES) IN I-DIRECTION
            DO J = 0, JMAXBLK
                DO I = 0, IMAXBLK+1
                    ! CALC CELL AREAS (FLUXES)
                    m%Axi(I,J) = m%x(I,J+1) - m%x(I,J)
                    m%Ayi(I,J) = m%y(I,J+1) - m%y(I,J)
                END DO
            END DO

            ! Actual finite-volume scheme equation parameters
            DO J = 0, JMAXBLK
                DO I = 0, IMAXBLK
```

```fortran
                              Axi_half = ( m%Axi(I+1,J) + m%Axi(I,J) ) * 0.25D0
                              Axj_half = ( m%Axj(I,J+1) + m%Axj(I,J) ) * 0.25D0
                              Ayi_half = ( m%Ayi(I+1,J) + m%Ayi(I,J) ) * 0.25D0
                              Ayj_half = ( m%Ayj(I,J+1) + m%Ayj(I,J) ) * 0.25D0

                              ! (NN = 'negative-negative', PN = 'positive-negative',
                                 ! see how fluxes are summed)
                              m%xNN(I, J) = ( -Axi_half - Axj_half )
                              m%xPN(I, J) = (  Axi_half - Axj_half )
                              m%xPP(I, J) = (  Axi_half + Axj_half )
                              m%xNP(I, J) = ( -Axi_half + Axj_half )
                              m%yPP(I, J) = (  Ayi_half + Ayj_half )
                              m%yNP(I, J) = ( -Ayi_half + Ayj_half )
                              m%yNN(I, J) = ( -Ayi_half - Ayj_half )
                              m%yPN(I, J) = (  Ayi_half - Ayj_half )
                      END DO
                  END DO
              END DO
      END SUBROUTINE calc_cell_params

      SUBROUTINE calc_constants(blocks)
          ! Calculate terms that are constant regardless of iteration
          !(time step, secondary volumes, constant term.)  This way,
          ! they don't need to be calculated within the loop at each iteration

          TYPE(BLKTYPE), TARGET :: blocks(:)
          TYPE(MESHTYPE), POINTER :: m
          INTEGER :: IBLK, I, J

          ! CALC PRIME GRID FOR TIME STEP
          DO IBLK = 1, MYNBLK
              m => blocks(IBLK)%mesh
              DO J = 0, JMAXBLK + 1+1
                  DO I = 0, IMAXBLK + 1+1
                      m%xp(I, J) = COS( 0.5D0 * PI * DFLOAT(IMAX - ( blocks(IBLK)%IMIN + I - 1) ) / DFLOAT(IMAX - 1) )
                      m%yp(I, J) = COS( 0.5D0 * PI * DFLOAT(JMAX - ( blocks(IBLK)%JMIN + J - 1) ) / DFLOAT(JMAX - 1) )
                  END DO
              END DO
          END DO

          DO IBLK = 1, MYNBLK
              m => blocks(IBLK)%mesh
              DO J = 0, JMAXBLK + 1
                  DO I = 0, IMAXBLK + 1
                      ! CALC TIMESTEP FROM CFL
                      m%dt(I,J) = ((CFL * 0.5D0) / alpha) * m%V(I,J) ** 2 &
                                      / ( (m%xp(I+1,J) - m%xp(I,J))**2 &
                                        + (m%yp(I,J+1) - m%yp(I,J))**2 )
!                     write(*,*) "dt ",m%dt(I,J)
                      ! CALC SECONDARY VOLUMES
                      ! (for rectangular mesh, just average volumes of the 4 cells
                      !  surrounding the point)
                      m%V2nd(I,J) = ( m%V(I,  J) + m%V(I-1,  J) &
                                      + m%V(I,J-1) + m%V(I-1,J-1) ) * 0.25D0

                      ! CALC CONSTANT TERM
                      ! (this term remains constant in the equation regardless of
                      !  iteration number, so only calculate once here,
                      !  instead of in loop)
                      m%term(I,J) = m%dt(I,J) * alpha / m%V2nd(I,J)
                  END DO
              END DO
          END DO
      END SUBROUTINE calc_constants

      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      !!!! SOLVER !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```fortran
2102    SUBROUTINE calc_temp(b)
2103        ! Calculate temperature at all points in mesh, excluding BC cells.
2104        ! Calculate first and second derivatives for finite-volume scheme
2105
2106        TYPE(BLKTYPE), TARGET :: b(:)
2107        TYPE(MESHTYPE), POINTER :: m
2108        ! First partial derivatives of temperature in x and y directions
2109        REAL(KIND=8) :: dTdx, dTdy
2110        INTEGER :: IBLK, I, J
2111
2112        DO IBLK = 1, MYNBLK
2113            m => b(IBLK)%mesh
2114
2115            ! RESET SUMMATION
2116            m%Ttmp = 0.D0
2117
2118            ! PREVIOUSLY SET ITERATION LIMITS TO UTILIZE GHOST NODES ONLY
2119               !ON INTERIOR FACES
2120            DO J = b(IBLK)%JMINLOC, b(IBLK)%JMAXLOC
2121                DO I = b(IBLK)%IMINLOC, b(IBLK)%IMAXLOC
2122                    ! CALC FIRST DERIVATIVES
2123                    dTdx = + 0.5d0 &
2124                                * (( m%T(I+1,J) + m%T(I+1,J+1) ) * m%Ayi(I+1,J) &
2125                                -  ( m%T(I,  J) + m%T(I,  J+1) ) * m%Ayi(I,  J) &
2126                                -  ( m%T(I,J+1) + m%T(I+1,J+1) ) * m%Ayj(I,J+1) &
2127                                +  ( m%T(I,  J) + m%T(I+1,  J) ) * m%Ayj(I,  J) &
2128                                   ) / m%V(I,J)
2129                    dTdy = - 0.5d0 &
2130                                * (( m%T(I+1,J) + m%T(I+1,J+1) ) * m%Axi(I+1,J) &
2131                                -  ( m%T(I,  J) + m%T(I,  J+1) ) * m%Axi(I,  J) &
2132                                -  ( m%T(I,J+1) + m%T(I+1,J+1) ) * m%Axj(I,J+1) &
2133                                +  ( m%T(I,  J) + m%T(I+1,  J) ) * m%Axj(I,  J) &
2134                                   ) / m%V(I,J)
2135
2136                    ! Alternate distributive scheme second-derivative operator.
2137                    m%Ttmp(I+1,  J) = m%Ttmp(I+1,  J) + m%term(I+1,  J) * ( m%yNN(I,J) * dTdx + m%xPP(I,J) * dTdy )
2138                    m%Ttmp(I,    J) = m%Ttmp(I,    J) + m%term(I,    J) * ( m%yPN(I,J) * dTdx + m%xNP(I,J) * dTdy )
2139                    m%Ttmp(I,  J+1) = m%Ttmp(I,  J+1) + m%term(I,  J+1) * ( m%yPP(I,J) * dTdx + m%xNN(I,J) * dTdy )
2140                    m%Ttmp(I+1,J+1) = m%Ttmp(I+1,J+1) + m%term(I+1,J+1) * ( m%yNP(I,J) * dTdx + m%xPN(I,J) * dTdy )
2141                END DO
2142            END DO
2143            ! SAVE NEW TEMPERATURE DISTRIBUTION
2144                ! (preserve Ttmp for residual calculation in solver loop)
2145
2146            ! Previously set bounds, add one to lower limit so as not to
2147            ! update BC. (dont need to for upper limit because explicit scheme)
2148            DO J = b(IBLK)%JMINLOC + 1, b(IBLK)%JMAXLOC
2149                DO I = b(IBLK)%IMINLOC + 1, b(IBLK)%IMAXLOC
2150                    m%T(I,J) = m%T(I,J) + m%Ttmp(I,J)
2151                END DO
2152            END DO
2153        END DO
2154    END SUBROUTINE calc_temp
2155
2156 END MODULE BLOCKMOD
```

Listing 3: Grids are decomposed into blocks and mapped onto NPROCS processors and information pertaining to neighbors is stored using the GRIDMOD module

## Appendix D: Multi-Block Plot3D Reader-Writer

```fortran
! MAE 267
! PROJECT 5
! LOGAN HALSTROM
! 29 NOVEMBER 2015

! DESCRIPTION:  This module contains functions for information input and output.
! Write grid and temperature files in PLOT3D format.
! Write and read block grid configuration file

! NOTE: How to Visualize Blocks in Paraview:
    ! open unformatted PLOT3D file.
    ! Change 'Coloring' from 'Solid' to 'vtkCompositeIndex'

MODULE IO
!      USE CONSTANTS
    USE BLOCKMOD
    IMPLICIT NONE

    ! VARIABLES
    INTEGER :: gridUnit  = 30   ! Unit for grid file
    INTEGER :: tempUnit = 21     ! Unit for temp file
    INTEGER :: resUnit = 23
    REAL(KIND=8) :: tRef = 1.D0          ! tRef number
    REAL(KIND=8) :: dum = 0.D0           ! dummy values

    ! LINKED LIST OF RESIDUAL HISTORY

    TYPE RESLIST
        ! Next element in linked list
        TYPE(RESLIST), POINTER :: next
        ! items in link:
        REAL(KIND=8) :: res
        INTEGER :: iter
    END TYPE RESLIST

    CONTAINS

        SUBROUTINE write_config(procs)
        ! Write block connectivity file with neighbor and BC info
        ! for each processor.
        ! Also write PLOT3D restart files for each processor.

        TYPE(PROCTYPE), TARGET :: procs(:)
        TYPE(PROCTYPE), POINTER :: p
        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), POINTER :: b
        INTEGER :: IP, IB, BLKFILE = 99
        CHARACTER(2) :: procname
        CHARACTER(20) :: xfile, qfile

        33 FORMAT(A)
        11 FORMAT( 3I7)
        22 FORMAT(33I7)
        44 FORMAT(33A7)

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!! WRITE CONFIG FILE FOR EACH PROCESSOR !!!!!!!!!!!!!!!!!!!!!!!
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        DO IP = 1, NPROCS
            p => procs(IP)

            ! MAKE FILE NAME (i.e. 'p01.config')
            IF (p%ID<10) THEN
                ! IF SINGLE DIGIT, PAD WITH 0 IN FRONT
                WRITE(procname, '(A,I1)') '0', p%ID
            ELSE
```

```fortran
                     WRITE(procname, '(I2)') p%ID
              END IF

              OPEN (UNIT = BLKFILE , FILE = TRIM("p" // procname // ".config"), form='formatted')

              ! WRITE AMOUNT OF BLOCKS AND DIMENSIONS
              WRITE(BLKFILE, 33) 'NBLK' // ' IMAXBLK' // ' JMAXBLK'
              WRITE(BLKFILE, 11) p%NBLK, IMAXBLK, JMAXBLK

              ! HEADER
              WRITE(BLKFILE, 44) 'ID', 'IMIN', 'JMIN', 'SIZE', &
                                 'IMNL', 'IMXL', 'JMNL', 'JMXL', &
                                 'NNB', 'NNP', 'NLOC', &
                                 'SNB', 'SNP', 'SLOC', &
                                 'ENB', 'ENP', 'ELOC', &
                                 'WNB', 'WNP', 'WLOC', &
                                 'NENB', 'NENP', 'NEL', &
                                 'SENB', 'SENP', 'SEL', &
                                 'SWNB', 'SWNP', 'SWL', &
                                 'NWNB', 'NWNP', 'NWL', &
                                 'ORI'
              DO IB = 1, p%NBLK
                   b => p%blocks(IB)
                   ! FOR EACH BLOCK, WRITE BLOCK NUMBER, STARTING/ENDING GLOBAL INDICES.
                   ! THEN BOUNDARY CONDITION AND NEIGHBOR NUMBER FOR EACH FACE:
                   ! NORTH EAST SOUTH WEST
                   WRITE(BLKFILE, 22) b%ID, b%IMIN, b%JMIN, INT(b%SIZE), &
                                      b%IMINLOC, b%IMAXLOC, b%JMINLOC, b%JMAXLOC, &
                                      b%NB%N,  b%NP%N,  b%NBLOC%N, &
                                      b%NB%S,  b%NP%S,  b%NBLOC%S, &
                                      b%NB%E,  b%NP%E,  b%NBLOC%E, &
                                      b%NB%W,  b%NP%W,  b%NBLOC%W, &
                                      b%NB%NE, b%NP%NE, b%NBLOC%NE, &
                                      b%NB%SE, b%NP%SE, b%NBLOC%SE, &
                                      b%NB%SW, b%NP%SW, b%NBLOC%SW, &
                                      b%NB%NW, b%NP%NW, b%NBLOC%NW, &
                                      b%ORIENT
              END DO
              CLOSE(BLKFILE)
         END DO

         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         !!! WRITE SOLUTION RESTART FILES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

         DO IP = 1, NPROCS
             p => procs(IP)
             ! MAKE FILE NAME
             IF (p%ID<10) THEN
                  ! IF SINGLE DIGIT, PAD WITH 0 IN FRONT
                  WRITE(procname, '(A,I1)') '0', p%ID
             ELSE
                  WRITE(procname, '(I2)') p%ID
             END IF
             xfile = "p" // procname // ".grid"
             qfile = "p" // procname // ".T"
             CALL plot3D(p%blocks, p%NBLK, xfile, qfile)
         END DO
    END SUBROUTINE write_config

    SUBROUTINE read_config(blocks)
         ! Called by each processor individually for its own blocks
         ! For given processor, read corresponding configuration file.
         ! Get neighbor connectivity info
         ! Also read PLOT3D restart files for grids for given processor

         ! BLOCK DATA TYPE
         TYPE(BLKTYPE), POINTER :: blocks(:)
         TYPE(BLKTYPE), POINTER :: b
```

```fortran
        TYPE(MESHTYPE), POINTER :: m
        INTEGER :: IP, IB, BLKFILE = 99
        CHARACTER(2) :: procname
        CHARACTER(20) :: xfile, qfile

        33 FORMAT(A)
        11 FORMAT( 3I7)
        22 FORMAT(33I7)
        44 FORMAT(33A7)


        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        !!! READ CONFIG FILE FOR GIVEN PRO!SSOR !!!!!!!!!!!!!!!!!!!!!!!
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


        ! FILE NAME (i.e. 'p01.config')
        IF (MYID<10) THEN
            ! IF SINGLE DIGIT, PAD WITH 0 IN FRONT
            WRITE(procname, '(A,I1)') '0', MYID
        ELSE
            WRITE(procname, '(I2)') MYID
        END IF

        OPEN (UNIT = BLKFILE , FILE = TRIM("p" // procname // ".config"), form='formatted')

        ! WRITE AMOUNT OF BLOCKS AND DIMENSIONS
        READ(BLKFILE, *)
        READ(BLKFILE, 11) MYNBLK, IMAXBLK, JMAXBLK
        ! (MYNBLK is global variable on this processor that is the number of
        !   blocks allocated to this processor)

        ! ALLOCATE BLOCKS FOR THIS PROCESSOR
        ! ('blocks' stores just the blocks for this processor.  It is
        !   in parallel for each processor)
        ALLOCATE( blocks(1:MYNBLK) )

        ! ALLOCATE MESH STUFF TO BE READ IN
        DO IB = 1, MYNBLK
            m => blocks(IB)%mesh

            ! ALLOCATE MESH INFORMATION
                ! ADD EXTRA INDEX AT BEGINNING AND END FOR GHOST NODES
            ALLOCATE( m%xp(  0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%yp(  0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%x(   0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%y(   0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%T(   0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%Ttmp(0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%dt(  0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%V2nd(0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%term(0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%Ayi( 0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%Axi( 0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%Ayj( 0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%Axj( 0:IMAXBLK+1,   0:JMAXBLK+1) )
            ALLOCATE( m%V(   0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%yPP( 0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%yNP( 0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%yNN( 0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%yPN( 0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%xNN( 0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%xPN( 0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%xPP( 0:IMAXBLK,     0:JMAXBLK  ) )
            ALLOCATE( m%xNP( 0:IMAXBLK,     0:JMAXBLK  ) )
        END DO

        ! HEADER
        READ(BLKFILE, *)
        DO IB = 1, MYNBLK
```

```fortran
                    b => blocks(IB)
                    ! FOR EACH BLOCK, READ BLOCK NUMBER, STARTING/ENDING GLOBAL INDICES.
                    ! THEN BOUNDARY CONDITION AND NEIGHBOR NUMBER FOR EACH FACE:
                    ! NORTH EAST SOUTH WEST
                    READ(BLKFILE, 22) b%ID, b%IMIN, b%JMIN, b%SIZE, &
                                      b%IMINLOC, b%IMAXLOC, b%JMINLOC, b%JMAXLOC, &
                                      b%NB%N,  b%NP%N,  b%NBLOC%N, &
                                      b%NB%S,  b%NP%S,  b%NBLOC%S, &
                                      b%NB%E,  b%NP%E,  b%NBLOC%E, &
                                      b%NB%W,  b%NP%W,  b%NBLOC%W, &
                                      b%NB%NE, b%NP%NE, b%NBLOC%NE, &
                                      b%NB%SE, b%NP%SE, b%NBLOC%SE, &
                                      b%NB%SW, b%NP%SW, b%NBLOC%SW, &
                                      b%NB%NW, b%NP%NW, b%NBLOC%NW, &
                                      b%ORIENT
            END DO
            CLOSE(BLKFILE)

            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            !!! READ SOLUTION RESTART FILES !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

            ! MAKE FILE NAME
            IF (MYID<10) THEN
                ! IF SINGLE DIGIT, PAD WITH 0 IN FRONT
                WRITE(procname, '(A,I1)') '0', MYID
            ELSE
                WRITE(procname, '(I2)') MYID
            END IF
            xfile = "p" // procname // ".grid"
            qfile = "p" // procname // ".T"
            CALL readPlot3D(blocks, xfile, qfile)
    END SUBROUTINE read_config


    SUBROUTINE plot3D(blocks, NBLKS, xfile, qfile)
        ! write plt 2d file given blocks, number of blocks,
        ! x and q file names (no file extension), and the bounds for writing
        ! (0 for real grid, 1 to include ghosts)
        IMPLICIT NONE

        TYPE(BLKTYPE) :: blocks(:)
        INTEGER :: IBLK, I, J, NBLKS, bound = 1
        ! OUTPUT FILES (without file exension)
        CHARACTER(20) :: xfile, qfile

        ! FORMAT STATEMENTS
            ! I --> Integer, number following is number of sig figs
            ! E --> scientific notation,
                        ! before decimal is sig figs of exponent?
                        ! after decimal is sig figs of value
            ! number before letter is how many entries on single line
                ! before newline (number of columns)
    10      FORMAT(I10)
    20      FORMAT(10I10)
    30      FORMAT(10E20.8)

        !!! FORMATTED !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        ! OPEN FILES
        OPEN(UNIT=gridUnit,FILE = TRIM(xfile) // '.form.xyz',FORM='formatted')
        OPEN(UNIT=tempUnit,FILE = TRIM(qfile) // '.form.dat',FORM='formatted')

        ! WRITE TO GRID FILE
        WRITE(gridUnit, 10) NBLKS
        WRITE(gridUnit, 20) ( IMAXBLK, JMAXBLK, IBLK=1, NBLKS)
!          WRITE(gridUnit, 20) ( blocks(IBLK)%IMAX, blocks(IBLK)%JMAX, IBLK=1, NBLK)
        DO IBLK = 1, NBLKS
            WRITE(gridUnit, 30) ( (blocks(IBLK)%mesh%x(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
```

```fortran
                                                ( (blocks(IBLK)%mesh%y(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound)
          END DO


          ! WRITE TO TEMPERATURE FILE
              ! When read in paraview, 'density' will be equivalent to temperature
          WRITE(tempUnit, 10) NBLKS
          WRITE(tempUnit, 20) ( IMAXBLK, JMAXBLK, IBLK=1, NBLKS)
          DO IBLK = 1, NBLKS

              WRITE(tempUnit, 30) tRef,dum,dum,dum
              WRITE(tempUnit, 30) ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
                                  ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
                                  ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
                                  ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound)
          END DO

          ! CLOSE FILES
          CLOSE(gridUnit)
          CLOSE(tempUnit)

          !!! UNFORMATTED !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

          ! OPEN FILES
          OPEN(UNIT=gridUnit,FILE = TRIM(xfile) // '.xyz',FORM='unformatted')
          OPEN(UNIT=tempUnit,FILE = TRIM(qfile) // '.dat',FORM='unformatted')

          ! WRITE TO GRID FILE (UNFORMATTED)
              ! (Paraview likes unformatted better)
          WRITE(gridUnit) NBLKS
          WRITE(gridUnit) ( IMAXBLK, JMAXBLK, IBLK=1, NBLKS)
!          WRITE(gridUnit) ( blocks(IBLK)%IMAX, blocks(IBLK)%JMAX, IBLK=1, NBLK)
          DO IBLK = 1, NBLKS
              WRITE(gridUnit) ( (blocks(IBLK)%mesh%x(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
                              ( (blocks(IBLK)%mesh%y(I,J), I=1,IMAXBLK), J=1,JMAXBLK)
          END DO


          ! WRITE TO TEMPERATURE FILE
              ! When read in paraview, 'density' will be equivalent to temperature
          WRITE(tempUnit) NBLKS
          WRITE(tempUnit) ( IMAXBLK, JMAXBLK, IBLK=1, NBLKS)
          DO IBLK = 1, NBLKS

              WRITE(tempUnit) tRef,dum,dum,dum
              WRITE(tempUnit) ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
                              ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
                              ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
                              ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK)
          END DO

          ! CLOSE FILES
          CLOSE(gridUnit)
          CLOSE(tempUnit)
      END SUBROUTINE plot3D

      SUBROUTINE readPlot3D(blocks, xfile, qfile)
          IMPLICIT NONE

          TYPE(BLKTYPE) :: blocks(:)
          INTEGER :: IBLK, I, J, NBLKS
          INTEGER :: NBLKREAD, IMAXBLKREAD, JMAXBLKREAD, bound = 1
          REAL(KIND=8) :: dum1, dum2, dum3, dum4
          ! OUTPUT FILES (without file exension)
          CHARACTER(20) :: xfile, qfile

          ! FORMAT STATEMENTS
              ! I --> Integer, number following is number of sig figs
              ! E --> scientific notation,
```

```fortran
                              ! before decimal is sig figs of exponent?
                              ! after decimal is sig figs of value
                     ! number before letter is how many entries on single line
                         ! before newline (number of columns)
          10      FORMAT(I10)
          20      FORMAT(10I10)
          30      FORMAT(10E20.8)


          !!! FORMATTED !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

          ! OPEN FILES
          OPEN(UNIT=gridUnit,FILE = TRIM(xfile) // '.form.xyz',FORM='formatted')
          OPEN(UNIT=tempUnit,FILE = TRIM(qfile) // '.form.dat',FORM='formatted')

          ! READ GRID FILE
          READ(gridUnit, 10) NBLKREAD
          READ(gridUnit, 20) ( IMAXBLKREAD, JMAXBLKREAD, IBLK=1, NBLKREAD)
          DO IBLK = 1, NBLKREAD
              READ(gridUnit, 30) ( (blocks(IBLK)%mesh%x(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
                                  ( (blocks(IBLK)%mesh%y(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound)
          END DO

          ! READ TEMPERATURE FILE
          READ(tempUnit, 10) NBLKREAD
          READ(tempUnit, 20) ( IMAXBLKREAD, JMAXBLKREAD, IBLK=1, NBLKREAD)
          DO IBLK = 1, NBLKREAD

!                 READ(tempUnit, 30) tRef,dum,dum,dum
              READ(tempUnit, 30) dum1, dum2, dum3, dum4
              READ(tempUnit, 30) ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
                                  ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
                                  ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound), &
                                  ( (blocks(IBLK)%mesh%T(I,J), I=1-bound,IMAXBLK+bound), J=1-bound,JMAXBLK+bound)
          END DO

          ! CLOSE FILES
          CLOSE(gridUnit)
          CLOSE(tempUnit)


     END SUBROUTINE readPlot3D

!      SUBROUTINE readPlot3D(blocks)
!          IMPLICIT NONE

!          TYPE(BLKTYPE) :: blocks(:)
!          INTEGER :: IBLK, I, J
!          ! READ INFO FOR BLOCK DIMENSIONS
!          INTEGER :: NBLKREAD, IMAXBLKREAD, JMAXBLKREAD
!          ! OUTPUT FILES
!          CHARACTER(20) :: xfile, qfile

!          ! FORMAT STATEMENTS
!              ! I --> Integer, number following is number of sig figs
!              ! E --> scientific notation,
!                          ! before decimal is sig figs of exponent?
!                          ! after decimal is sig figs of value
!              ! number before letter is how many entries on single line
!                  ! before newline (number of columns)
!          10      FORMAT(I10)
!          20      FORMAT(10I10)
!          30      FORMAT(10E20.8)

!          !!! FORMATTED !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!          ! OPEN FILES
! !          OPEN(UNIT=gridUnit,FILE= TRIM(casedir) // 'grid_form.xyz',FORM='formatted')
! !          OPEN(UNIT=tempUnit,FILE= TRIM(casedir) // 'T_form.dat',FORM='formatted')
!          OPEN(UNIT=gridUnit,FILE= 'grid_form.xyz',FORM='formatted')
```

```fortran
413 !            OPEN(UNIT=tempUnit,FILE= 'T_form.dat',FORM='formatted')
414
415 !            ! READ GRID FILE
416 !            READ(gridUnit, 10) NBLKREAD
417 !            READ(gridUnit, 20) ( IMAXBLKREAD, JMAXBLKREAD, IBLK=1, NBLKREAD)
418 ! !              WRITE(gridUnit, 20) ( blocks(IBLK)%IMAX, blocks(IBLK)%JMAX, IBLK=1, NBLK)
419 !            DO IBLK = 1, NBLKREAD
420 !                READ(gridUnit, 30) ( (blocks(IBLK)%mesh%x(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
421 !                                   ( (blocks(IBLK)%mesh%y(I,J), I=1,IMAXBLK), J=1,JMAXBLK)
422 !            END DO
423
424
425 !            ! READ TEMPERATURE FILE
426 !                ! When read in paraview, 'density' will be equivalent to temperature
427 !            READ(tempUnit, 10) NBLKREAD
428 !            READ(tempUnit, 20) ( IMAXBLKREAD, JMAXBLKREAD, IBLK=1, NBLKREAD)
429 !            DO IBLK = 1, NBLKREAD
430
431 !                READ(tempUnit, 30) tRef,dum,dum,dum
432 !                READ(tempUnit, 30) ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
433 !                                   ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
434 !                                   ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK), &
435 !                                   ( (blocks(IBLK)%mesh%T(I,J), I=1,IMAXBLK), J=1,JMAXBLK)
436 !            END DO
437
438 !            ! CLOSE FILES
439 !            CLOSE(gridUnit)
440 !            CLOSE(tempUnit)
441 !        END SUBROUTINE readPlot3D
442
443 !        SUBROUTINE compositePlot3D()
444 !            type(blktype), ALLOCATABLE :: blocks(:)
445 !            type(proctype), target :: procs(nprocs)
446 !            type(proctype), pointer :: p
447 !            CHARACTER(2) :: procname
448 !            CHARACTER(20) :: xfile, qfile
449
450 !            integer :: procsort(NBLK), IDsSort(NBLK), I, ii
451 !            allocate(blocks(NBLK))
452 !            ! read block amalgamation file
453 !            OPEN(UNIT=55,FILE = 'blockrebuild.dat',FORM='formatted')
454 !            read(55,*)
455 !            do I = 1, NBLK
456 !                read(55,*) Ii, procsort(I), IDsSort(I)
457 !            end do
458 !            CLOSE(55)
459
460 !            OPEN(UNIT=65,FILE = 'procrebuild.dat',FORM='formatted')
461
462 !            do i = 1, NPROCs
463 !                p => procs(I)
464 !                READ(65,*) p%NBLK
465 !                allocate(p%blocks(p%NBLK))
466
467 !                IF (p%ID<10) THEN
468 !                    ! IF SINGLE DIGIT, PAD WITH 0 IN FRONT
469 !                    WRITE(procname, '(A,I1)') '0', p%ID
470 !                ELSE
471 !                    WRITE(procname, '(I2)') p%ID
472 !                END IF
473 !                xfile = "p" // procname // ".grid"
474 !                qfile = "p" // procname // ".T"
475 !                call readplot3d(p%blocks, xfile, qfile)
476
477 !            end do
478 !            CLOSE(65)
479
480 !            do i = 1, nblk
481 !                blocks(I) = procs( procsort(i) )%blocks( idsSort(i) )
```

```fortran
482 !          end do
483
484
485 !          call plot3d(blocks, nblk, 'grid', 'T')
486
487
488
489 !      END SUBROUTINE compositePlot3D
490
491
492      SUBROUTINE write_res(res_hist)
493          TYPE(RESLIST), POINTER :: res_hist
494          ! pointer to iterate linked list
495          TYPE(RESLIST), POINTER :: hist
496
497          ! open residual file
498 !          OPEN(UNIT=resUnit,FILE= TRIM(casedir) // 'res_hist.dat')
499          OPEN(UNIT=resUnit,FILE = 'res_hist.dat')
500          ! column headers
501          WRITE(resUnit,*) 'ITER       RESID'
502
503          ! point to residual linked list
504          hist => res_hist
505          ! skip first link, empty from iteration loop design
506          hist => hist%next
507          ! write residual history to file until list ends
508          DO
509              IF ( .NOT. ASSOCIATED(hist) ) EXIT
510              ! write iteration and residual in two columns
511              WRITE(resUnit,*) hist%iter, hist%res
512              hist => hist%next
513          END DO
514
515          CLOSE(resUnit)
516      END SUBROUTINE write_res
517
518
519 END MODULE IO
```

Listing 4:  Code for saving formatted multiblock PLOT3D solution files and reading restart files