MAE 267 – Project 2
Serial, 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 details the solution of the steady-state temperature distribution on a 1m x 1m block of steel with Dirichlet boundary conditions (Eqn 2). Single-processor solutions were previously performed on a square, non-uniform grids rotatated in the positive z-direction by $rot = 30^o$. Two grids of 101x101 points and 501x501 points were used to solve the equation of heat transfer. Temperature was uniformly initialized to a value of 3.5 and the solution was iterated until the maximum residual found was less than $1.0x10^{-5}$. The equation for heat conduction (Eqn 1) was solved using an explicit, node-centered, finite-volume scheme, with an alternative distributive scheme for the second-derivative operator. Steady-state temperature distribution was saved in a PLOT3D unformatted file, and CPU wall time of the solver was recorded.

Now, the code has been modified to decompose the domain into sub-domains refered to as blocks. Boundary and neighbor information for each block is stored so that connectivity can be accurately assessed when communication between blocks is required.

2 Equations and Algorithms

The solver developed for this analysis utilizes a finite-volume numerical solution method to solve the transient heat conduction equation (Eqn 1).

$$\rho c_p \frac{\partial T}{\partial t} = k \left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right] \tag{1}$$

The solution is initialized with the Dirichlet boundary conditions (Eqn 2).

$$T = \begin{cases} 5.0\left[\sin\left(\pi x_p\right) + 1.0\right] & \text{for} \quad j = j_{max} \\ \left|\cos\left(\pi x_p\right)\right| + 1.0 & \text{for} \quad j = 0 \\ 3.0 y_p + 2.0 & \text{for} \quad i = 0,\, i_{max} \end{cases} \tag{2}$$

Grids were generated according to the following (Eqn 3)

$$\begin{aligned} rot &= 30.0\frac{\pi}{180.0} \\ x_p &= \cos\left[0.5\pi \frac{i_{max} - i}{i_{max} - 1}\right] \\ y_p &= \cos\left[0.5\pi \frac{j_{max} - j}{j_{max} - 1}\right] \\ x(i,j) &= x_p \cos(rot) + (1.0 - y_p)\sin(rot) \\ y(i,j) &= y_p \cos(rot) + x_p \sin(rot) \end{aligned} \tag{3}$$

To solve Eqn 1 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.

$$\begin{aligned} \frac{\partial T}{\partial x} = \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}} \big[&(T_{i+1,j} + T_{i+1,j+1}) Ayi_{i+1,j} \\ &- (T_{i,j} + T_{i,j+1}) Ayi_{i,j} \\ &- (T_{i,j+1} + T_{i+1,j+1}) Ayi_{i,j+1} \\ &- (T_{i,j} + T_{i+1,j}) Ayi_{i,j} \big] \end{aligned} \tag{4}$$

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

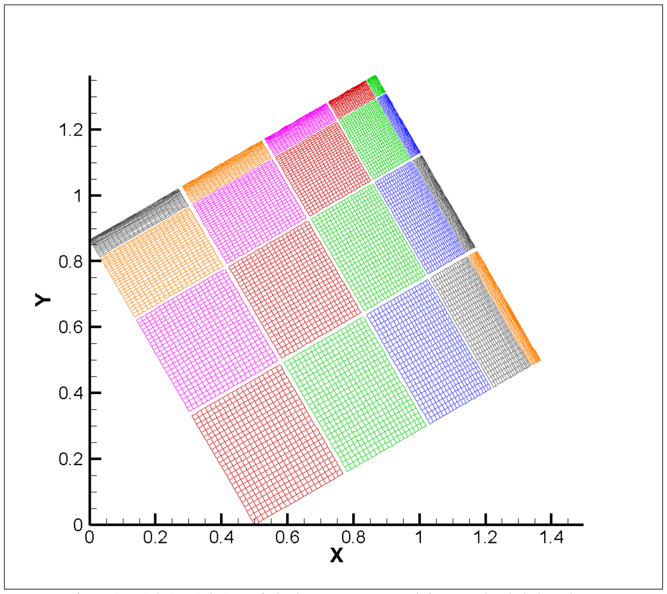

Fig. 1: 101x101 grid decomposed into 5x4 blocks

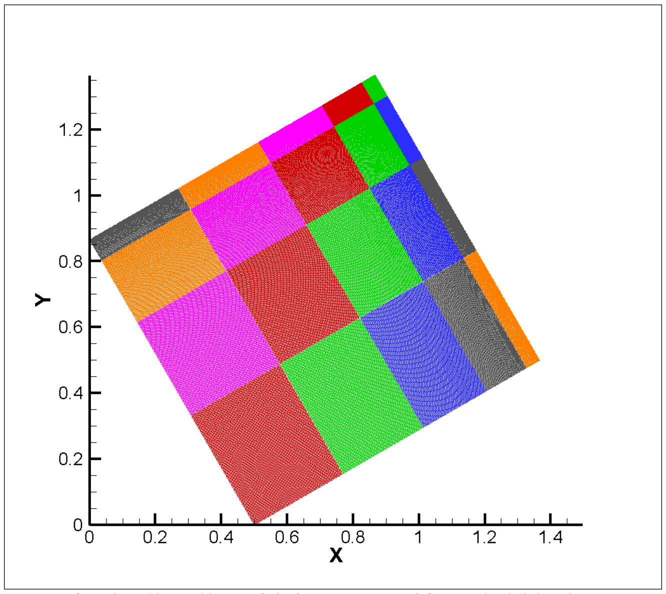

Fig. 3: 501x501 grid decomposed into 5x4 blocks

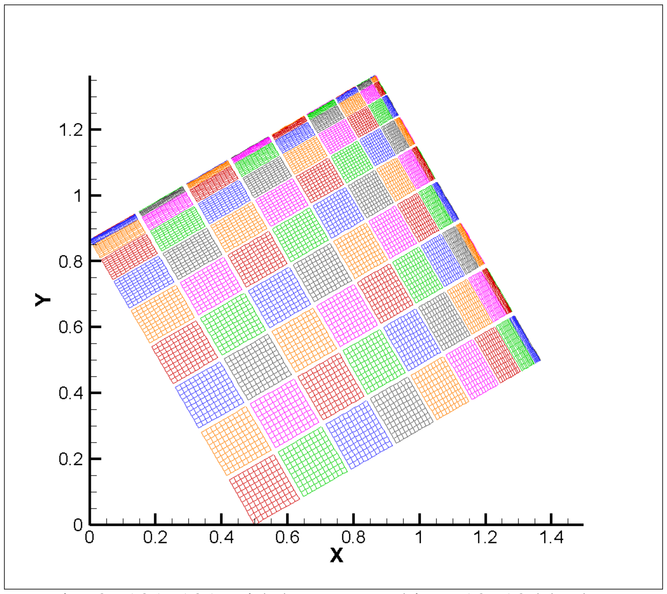

Fig. 2: 101x101 grid decomposed into 10x10 blocks

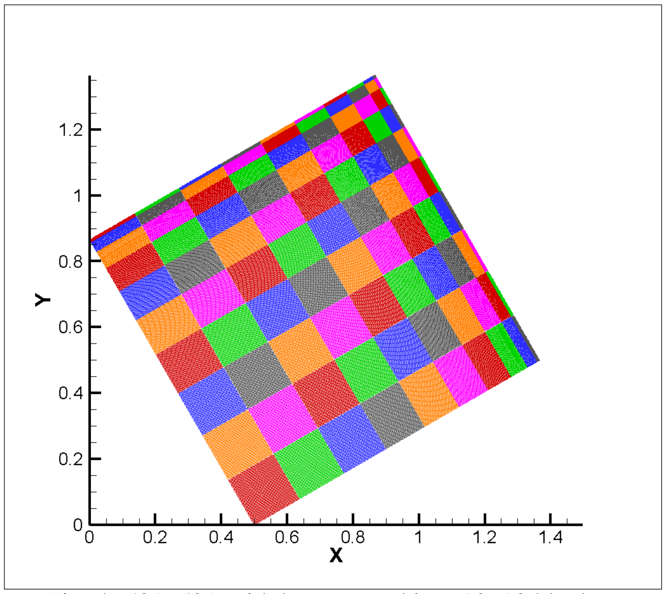

Fig. 4: 501x501 grid decomposed into 10x10 blocks

3 Results and Discussion

Both grids used in this analysis were non-uniformly distributed according to the same function and decomposed into 10x10 and 5x4 block systems.

Comparing the two grids, it can be seen that the block sizes for a given block decomposition are the same between the coarse and fine grid. The fine grid, however, has many more points contained in each block than does the coarse grid. It is also interesting to note that the edges of the grid that are most refined have the smallest blocks, and block size tends to decrease as a function of grid spacing. This decrease in area is required, since blocks are constrained to contain the same number of points.

4 Conclusion

Decomposing the domain is the first step towards solving a computational grid with parallel computing. Much more is still required, however, to create the complete infrastructure of such a solver. In the next project, the block decomposition will be improved with multi-block solver running on a single processor.

Appendix A: Grid Decomposition Code

```fortran
! MAE 267
! PROJECT 1
! LOGAN HALSTROM
! 12 OCTOBER 2015

! DESCRIPTION:  Modules used for solving heat conduction of steel plate.
! Initialize and store constants used in all subroutines.

! CONTENTS:
! CONSTANTS --> Initializes constants for simulation.  Sets grid size.
! CLOCK --> Calculates clock wall-time of a process.
! MAKEGRID --> Initialize grid with correct number of points and rotation,
!                 set boundary conditions, etc.
! CELLS -->  Initialize finite volume cells and do associated calculations
! TEMPERATURE --> Calculate and store new temperature distribution
!                     for given iteration

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! CONSTANTS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE CONSTANTS
    ! Initialize constants for simulation.  Set grid size.
    IMPLICIT NONE
    ! 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)
    ! Pi, grid rotation angle (30 deg)
    REAL(KIND=8), PARAMETER :: pi = 3.141592654D0, rot = 30.D0*pi/180.D0
    ! 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 block face is on North,east,south,west of main grid, identify
    INTEGER :: NBND = 1, SBND = 2, EBND = 3, WBND = 4

CONTAINS

    SUBROUTINE read_input()
        INTEGER :: I

        ! READ INPUTS FROM FILE
            !(So I don't have to recompile each time I change an input setting)
        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,*) M
        READ(1,*)
        READ(1,*) N

        ! 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 TO SCREEN
        WRITE(*,*) ''
        WRITE(*,*) 'Solving Mesh of size ixj:', IMAX, 'x', JMAX
        WRITE(*,*) 'With MxN blocks:', M, 'x', N
        WRITE(*,*) 'Number of blocks:', NBLK
        WRITE(*,*) ''
    END SUBROUTINE read_input
END MODULE CONSTANTS

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! INITIALIZE GRID !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE BLOCKMOD
    ! Initialize grid with correct number of points and rotation,
    ! set boundary conditions, etc.
    USE CONSTANTS

    IMPLICIT NONE
    PUBLIC

    ! 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

    TYPE NBRTYPE
        ! STORE NUMBER OF NEIGHBOR BLOCK, OR NUMBER OF BOUNDARY CONDITION
        INTEGER :: BC, NB
    END TYPE NBRTYPE

    ! DERIVED DATA TYPE WITH INFORMATION PERTAINING TO SPECIFIC BLOCK

    TYPE BLKTYPE
        ! DER. DATA TYPE STORES LOCAL MESH INFO
        TYPE(MESHTYPE) :: mesh
        ! Information about face neighbors (north, east, south, west)
        TYPE(NBRTYPE) :: FaceN, FaceE, FaceS, FaceW
        ! BLOCK NUMBER
        INTEGER :: ID
        ! GLOBAL INDICIES OF MINIMUM AND MAXIMUM OF LOCAL INDICIES FOR BLOCK
        INTEGER :: IMIN, IMAX, JMIN, JMAX
        ! BLOCK ORIENTATION
        INTEGER :: orientation
    END TYPE BLKTYPE

CONTAINS
```

```fortran
    SUBROUTINE init_blocks(b)
        ! BLOCK DATA TYPE
        TYPE(BLKTYPE) :: b(:)
        ! COUNTER VARIABLES
            ! 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
                ! ASSIGN BLOCK NUMBER
                b(IBLK)%ID = IBLK
                ! ASSIGN GLOBAL MIN/MAX INDICIES OF LOCAL GRID
                b(IBLK)%IMAX = 1 + I         * (IMAXBLK - 1)
                b(IBLK)%JMAX = 1 + J         * (JMAXBLK - 1)
                b(IBLK)%IMIN = b(IBLK)%IMAX - (IMAXBLK - 1)
                b(IBLK)%JMIN = b(IBLK)%JMAX - (JMAXBLK - 1)

                ! ASSIGN NUMBERS OF FACE NEIGHBOR BLOCKS
                    !if boundary face, assign bc later
                b(IBLK)%FaceN%NB = IBLK + N
                b(IBLK)%FaceS%NB = IBLK - N
                b(IBLK)%FaceE%NB = IBLK + 1
                b(IBLK)%FaceW%NB = IBLK - 1

                ! ASSIGN FACE BOUNDARY CONDITIONS
                    ! initialize as all internal
                b(IBLK)%FaceN%BC = -1
                b(IBLK)%FaceS%BC = -1
                b(IBLK)%FaceE%BC = -1
                b(IBLK)%FaceW%BC = -1
                ! Assign faces on boundary of the actual computational grid
                ! with number corresponding to which boundary they are on
                IF ( b(IBLK)%JMAX == JMAX ) THEN
                    ! NORTH BLOCK FACE IS ON MESH NORTH BOUNDARY
                    b(IBLK)%FaceN%BC = NBND
                    ! un-assign neighbor that wasnt real
                    b(IBLK)%FaceN%NB = -1
                END IF
                IF ( b(IBLK)%IMAX == IMAX ) THEN
                    ! EAST BLOCK FACE IS ON MESH EAST BOUNDARY
                    b(IBLK)%FaceE%BC = EBND
                    b(IBLK)%FaceE%NB = -1
                END IF
                IF ( b(IBLK)%JMIN == 1 ) THEN
                    ! SOUTH BLOCK FACE IS ON MESH SOUTH BOUNDARY
                    b(IBLK)%FaceS%BC = SBND
                    b(IBLK)%FaceS%NB = -1
                END IF
                IF ( b(IBLK)%IMIN == 1 ) THEN
                    ! WEST BLOCK FACE IS ON MESH WEST BOUNDARY
                    b(IBLK)%FaceW%BC = WBND
                    b(IBLK)%FaceW%NB = -1
                END IF

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

            END DO
        END DO
    END SUBROUTINE init_blocks

    SUBROUTINE write_blocks(b)
        ! WRITE BLOCK CONNECTIVITY FILE

        ! BLOCK DATA TYPE
        TYPE(BLKTYPE) :: b(:)
        INTEGER :: I, BLKFILE = 99

        11 format(3I5)
        22 format(33I5)

        OPEN (UNIT = BLKFILE , FILE = "blockconfig.dat", form='formatted')
        ! WRITE AMOUNT OF BLOCKS AND DIMENSIONS
        WRITE(BLKFILE, 11) NBLK, IMAXBLK, JMAXBLK
        DO I = 1, NBLK
            ! FOR EACH BLOCK, WRITE BLOCK NUMBER, AND STARTING GLOBAL INDICES.
            ! THEN BOUNDARY CONDITION AND NEIGHBOR NUMBER FOR EACH FACE:
            ! NORTH EAST SOUTH WEST
            WRITE(BLKFILE, 22) b(I)%ID, b(I)%IMIN, b(I)%JMIN, &
                b(I)%FaceN%BC, b(I)%FaceN%NB, &
                b(I)%FaceE%BC, b(I)%FaceE%NB, &
                b(I)%FaceS%BC, b(I)%FaceS%NB, &
                b(I)%FaceW%BC, b(I)%FaceW%NB, &
                b(I)%orientation
        END DO
        CLOSE(BLKFILE)
    END SUBROUTINE write_blocks

    SUBROUTINE init_mesh(b)
        ! BLOCK DATA TYPE
        TYPE(BLKTYPE) :: b(:)
        INTEGER :: IBLK, I, J

        DO IBLK = 1, NBLK

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

            DO J = 0, JMAXBLK+1
                DO I = 0, IMAXBLK+1
                    ! MAKE SQUARE GRID
                    b(IBLK)%mesh%xp(I, J) = COS( 0.5D0 * pi * DFLOAT(IMAX - (I + b(IBLK)%IMIN - 1) ) / DFLOAT(IMAX -
                    b(IBLK)%mesh%yp(I, J) = COS( 0.5D0 * pi * DFLOAT(JMAX - (J + b(IBLK)%JMIN - 1) ) / DFLOAT(JMAX -
                    ! ROTATE GRID
                    b(IBLK)%mesh%x(I, J) = b(IBLK)%mesh%xp(I, J) * COS(rot) + (1.D0 - b(IBLK)%mesh%yp(I, J) ) * SIN(
                    b(IBLK)%mesh%y(I, J) = b(IBLK)%mesh%yp(I, J) * COS(rot) + (b(IBLK)%mesh%xp(I, J)) * SIN(rot)
                END DO
            END DO
            DO J = 0, JMAXBLK+1-1
                DO I = 0, IMAXBLK+1-1
                    ! 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)
                    b(IBLK)%mesh%V(I,J) = ( b(IBLK)%mesh%x(I+1,J+1) &
                                          - b(IBLK)%mesh%x(I,   J) ) &
                            * ( b(IBLK)%mesh%y(I,   J+1) - b(IBLK)%mesh%y(I+1,J) ) &
                            - ( b(IBLK)%mesh%x(I,   J+1) - b(IBLK)%mesh%x(I+1,J) ) &
                            * ( b(IBLK)%mesh%y(I+1,J+1) - b(IBLK)%mesh%y(I,   J) )
                END DO
            END DO
        END DO
    END SUBROUTINE init_mesh

    SUBROUTINE init_temp(blocks)
        ! Initialize temperature across mesh
        ! BLOCK DATA TYPE
        TYPE(BLKTYPE), TARGET  :: blocks(:)
        TYPE(BLKTYPE), POINTER :: b
        TYPE(MESHTYPE), POINTER :: m
        INTEGER :: IBLK, I, J


        !PUT DEBUG BC HERE!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        DO IBLK = 1, NBLK
            b => blocks(IBLK)
            m => blocks(IBLK)%mesh
            ! FIRST, INITIALIZE ALL POINT TO INITIAL TEMPERATURE (T0)
            m%T(1:IMAXBLK, 1:JMAXBLK) = T0
            ! THEN, INITIALIZE BOUNDARIES DIRICHLET B.C.
            ! face on north boundary
            IF (b%FaceN%BC == NBND) THEN
                DO I = 1, IMAXBLK
                    m%T(I,JMAX) = 5.D0 * (SIN(pi * m%xp(I,JMAX)) + 1.D0)
                END DO
            END IF
            IF (b%FaceS%BC == SBND) THEN
                DO I = 1, IMAXBLK
                    m%T(I,1) = ABS(COS(pi * m%xp(I,1))) + 1.D0
                END DO
            END IF
            IF (b%FaceE%BC == EBND) THEN
                DO J = 1, JMAXBLK
                    m%T(IMAX,J) = 3.D0 * m%yp(IMAX,J) + 2.D0
                END DO
            END IF
            IF (b%FaceW%BC == WBND) THEN
                DO J = 1, JMAXBLK
                    m%T(I,1) = ABS(COS(pi * m%xp(I,1))) + 1.D0
                END DO
            END IF

!             DO J = 1, JMAXBLK
!                 b(IBLK)%mesh%T(1,J) = 3.D0 * b(IBLK)%mesh%yp(1,J) + 2.D0
!                 b(IBLK)%mesh%T(IMAX,J) = 3.D0 * b(IBLK)%mesh%yp(IMAX,J) + 2.D0
!             END DO
!             DO I = 1, IMAXBLK
!                 b(IBLK)%mesh%T(I,1) = ABS(COS(pi * b(IBLK)%mesh%xp(I,1))) + 1.D0
!                 b(IBLK)%mesh%T(I,JMAX) = 5.D0 * (SIN(pi * b(IBLK)%mesh%xp(I,JMAX)) + 1.D0)
!             END DO
        END DO
    END SUBROUTINE init_temp

    SUBROUTINE calc_2nd_areas(blocks)
        ! calculate areas for secondary fluxes.
        ! 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, NBLK
            m => blocks(IBLK)%mesh
            DO J = 1, JMAX
                DO I = 1, IMAX-1
                    ! CALC CELL AREAS
                    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
            DO J = 1, JMAX-1
                DO I = 1, IMAX
                    ! CALC CELL AREAS
                    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 = 1, JMAX-1
                DO I = 1, IMAX-1

                    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_2nd_areas

    SUBROUTINE calc_constants(blocks)
        ! Calculate constants for a given iteration loop.  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
        DO IBLK = 1, NBLK
            m => blocks(IBLK)%mesh
            DO J = 2, JMAX - 1
                DO I = 2, IMAX - 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 )
                    ! 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

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!! CALCULATE TEMPERATURE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    SUBROUTINE calc_temp(blocks)
        ! Calculate first and second derivatives for finite-volume scheme
        TYPE(BLKTYPE), TARGET :: blocks(:)
        TYPE(MESHTYPE), POINTER :: m
        ! First partial derivatives of temperature in x and y directions
        REAL(KIND=8) :: dTdx, dTdy
        INTEGER :: IBLK, I, J

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

            ! RESET SUMMATION
            m%Ttmp = 0.D0

            DO J = 1, JMAX - 1
                DO I = 1, IMAX - 1
                    ! CALC FIRST DERIVATIVES
                    dTdx = + 0.5d0 &
                            * (( m%T(I+1,J) + m%T(I+1,J+1) ) * m%Ayi(I+1,J) &
                            -  ( m%T(I,   J) + m%T(I,   J+1) ) * m%Ayi(I,   J) &
                            -  ( m%T(I,J+1) + m%T(I+1,J+1) ) * m%Ayj(I,J+1) &
                            +  ( m%T(I,   J) + m%T(I+1,   J) ) * m%Ayj(I,   J) &
                                ) / m%V(I,J)
                    dTdy = - 0.5d0 &
                            * (( m%T(I+1,J) + m%T(I+1,J+1) ) * m%Axi(I+1,J) &
                            -  ( m%T(I,   J) + m%T(I,   J+1) ) * m%Axi(I,   J) &
                            -  ( m%T(I,J+1) + m%T(I+1,J+1) ) * m%Axj(I,J+1) &
                            +  ( m%T(I,   J) + m%T(I+1,   J) ) * m%Axj(I,   J) &
                                ) / m%V(I,J)

                    ! Alternate distributive scheme second-derivative operator.
                    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 )
                    m%Ttmp(I,    J) = m%Ttmp(I,    J) + m%term(I,    J) * ( m%yPN(I,J) * dTdx + m%xNP(I,J) * dTdy )
                    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 )
                    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 )
                END DO
            END DO
        END DO
    END SUBROUTINE calc_temp
END MODULE BLOCKMOD
```

Listing 1: Grids are decomposed into blocks and information pertaining to neighbors is stored using the GRIDMOD module

Appendix B: Multi-Block Plot3D Writer

```fortran
! MAE 267
! LOGAN HALSTROM
! 12 OCTOBER 2015

! DESCRIPTION:  This module creates a grid and temperature file in
!               the plot3D format for steady state solution

MODULE plot3D_module
    USE CONSTANTS
    USE BLOCKMOD
    IMPLICIT NONE

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

    CONTAINS
    SUBROUTINE plot3D(blocks)
        IMPLICIT NONE

        TYPE(BLKTYPE) :: blocks(:)
        INTEGER :: IBLK, I, J

        ! FORMAT STATEMENTS
        10     FORMAT(I10)
        20     FORMAT(10I10)
        30     FORMAT(10E20.8)

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

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

        ! WRITE TO GRID FILE (UNFORMATTED)
            ! (Paraview likes unformatted better)
        WRITE(gridUnit, 10) NBLK
        WRITE(gridUnit, 20) ( IMAXBLK, JMAXBLK, IBLK=1, NBLK)
!         WRITE(gridUnit, 20) ( blocks(IBLK)%IMAX, blocks(IBLK)%JMAX, IBLK=1, NBLK)
        DO IBLK = 1, NBLK
            WRITE(gridUnit, 30) ( (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, 10) NBLK
        WRITE(tempUnit, 20) ( IMAXBLK, JMAXBLK, IBLK=1, NBLK)
        DO IBLK = 1, NBLK

            WRITE(tempUnit, 30) tRef,dum,dum,dum
            WRITE(tempUnit, 30) ( (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)

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

        ! OPEN FILES
        OPEN(UNIT=gridUnit,FILE='grid.xyz',FORM='unformatted')
        OPEN(UNIT=tempUnit,FILE='T.dat',FORM='unformatted')

        ! WRITE TO GRID FILE (UNFORMATTED)
            ! (Paraview likes unformatted better)
        WRITE(gridUnit) NBLK
        WRITE(gridUnit) ( IMAXBLK, JMAXBLK, IBLK=1, NBLK)
!         WRITE(gridUnit) ( blocks(IBLK)%IMAX, blocks(IBLK)%JMAX, IBLK=1, NBLK)
        DO IBLK = 1, NBLK
            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) NBLK
        WRITE(tempUnit) ( IMAXBLK, JMAXBLK, IBLK=1, NBLK)
        DO IBLK = 1, NBLK

            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
END MODULE plot3D_module
```

Listing 2: Code for saving formatted multiblock PLOT3D solution files

Appendix C: Sample Domain Connectivity File

```
   20   26   21
    1    1    1   -1    5   -1    2    2   -1    4   -1    1
    2   26    1   -1    6   -1    3    2   -1   -1    1    1
    3   51    1   -1    7   -1    4    2   -1   -1    2    1
    4   76    1   -1    8    3   -1    2   -1   -1    3    1
    5    1   21   -1    9   -1    6   -1    1    4   -1    1
    6   26   21   -1   10   -1    7   -1    2   -1    5    1
    7   51   21   -1   11   -1    8   -1    3   -1    6    1
    8   76   21   -1   12    3   -1   -1    4   -1    7    1
    9    1   41   -1   13   -1   10   -1    5    4   -1    1
   10   26   41   -1   14   -1   11   -1    6   -1    9    1
   11   51   41   -1   15   -1   12   -1    7   -1   10    1
   12   76   41   -1   16    3   -1   -1    8   -1   11    1
   13    1   61   -1   17   -1   14   -1    9    4   -1    1
   14   26   61   -1   18   -1   15   -1   10   -1   13    1
   15   51   61   -1   19   -1   16   -1   11   -1   14    1
   16   76   61   -1   20    3   -1   -1   12   -1   15    1
   17    1   81    1   -1   -1   18   -1   13    4   -1    1
   18   26   81    1   -1   -1   19   -1   14   -1   17    1
   19   51   81    1   -1   -1   20   -1   15   -1   18    1
   20   76   81    1   -1    3   -1   -1   16   -1   19    1
```

Listing 3: Sample file containing information pertaining to connectivity of grid sub-domains and boundary conditions

Appendix D: Other Relevant Codes

```fortran
! MAE 267
! PROJECT 2
! LOGAN HALSTROM
! 23 OCTOBER 2015


! DESCRIPTION:  Solve heat conduction equation for single block of steel.
! To compile: mpif90 -o main -O3 modules.f90 plot3D_module.f90 subroutines.f90 main.f90
    ! makes executable file 'main'
    ! 'rm *.mod' afterward to clean up unneeded compiled files
! To run: ./main or ./run.sh or sbatch run.sh on hpc1


PROGRAM heatTrans
!     USE CLOCK
    USE CONSTANTS
    USE subroutines
    USE plot3D_module

    IMPLICIT NONE

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

    ! BLOCKS
    TYPE(BLKTYPE), ALLOCATABLE :: blocks(:)
    ! GRID
    TYPE(MESHTYPE) :: mesh
    ! ITERATION PARAMETERS
    ! Minimum Residual
    REAL(KIND=8) :: min_res = 0.00001D0
    ! Maximum number of iterations
    INTEGER :: max_iter = 1000000, iter = 0, IBLK

    INCLUDE "mpif.h"
    REAL(KIND=8) :: start_total, end_total
    REAL(KIND=8) :: start_solve, end_solve
    ! CLOCK TOTAL TIME OF RUN
    start_total = MPI_Wtime()

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!! INITIALIZE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    ! READ INPUTS FROM FILE
    CALL read_input()
    ALLOCATE( blocks(NBLK) )
    ! INIITIALIZE SOLUTION
    WRITE(*,*) 'Making mesh...'
    CALL init(blocks, mesh)

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

    WRITE(*,*) 'Solving heat conduction...'
!     CALL solve(mesh, min_res, max_iter, iter)

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!! SAVE RESULTS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    WRITE(*,*) 'Writing results...'
    ! SAVE SOLUTION AS PLOT3D FILES
    CALL plot3D(blocks)
    ! CALC TOTAL WALL TIME
    end_total = MPI_Wtime()
    wall_time_total = end_total - start_total
    ! SAVE SOLVER PERFORMANCE PARAMETERS
!      CALL output(mesh, iter)

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!! CLEAN UP !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    DO IBLK = 1, NBLK
        DEALLOCATE( blocks(IBLK)%mesh%xp   )
        DEALLOCATE( blocks(IBLK)%mesh%yp   )
        DEALLOCATE( blocks(IBLK)%mesh%x    )
        DEALLOCATE( blocks(IBLK)%mesh%y    )
        DEALLOCATE( blocks(IBLK)%mesh%T    )
        DEALLOCATE( blocks(IBLK)%mesh%Ttmp )
        DEALLOCATE( blocks(IBLK)%mesh%dt   )
        DEALLOCATE( blocks(IBLK)%mesh%V  )
        DEALLOCATE( blocks(IBLK)%mesh%V2nd )
        DEALLOCATE( blocks(IBLK)%mesh%term )
        DEALLOCATE( blocks(IBLK)%mesh%yPP)
        DEALLOCATE( blocks(IBLK)%mesh%yNP)
        DEALLOCATE( blocks(IBLK)%mesh%yNN)
        DEALLOCATE( blocks(IBLK)%mesh%yPN)
        DEALLOCATE( blocks(IBLK)%mesh%xNN)
        DEALLOCATE( blocks(IBLK)%mesh%xPN)
        DEALLOCATE( blocks(IBLK)%mesh%xPP)
        DEALLOCATE( blocks(IBLK)%mesh%xNP)
    END DO

    WRITE(*,*) 'Done!'


END PROGRAM heatTrans
```

Listing 4: Wrapper program

```fortran
! MAE 267
! PROJECT 1
! LOGAN HALSTROM
! 12 OCTOBER 2015

! DESCRIPTION:  Subroutines used for solving heat conduction of steel plate.
! Utilizes modules from 'modules.f90'
! CONTENTS:
! init --> Initialize the solution with dirichlet B.C.s
! solve --> Solve heat conduction equation with finite volume scheme
! output --> Save solution parameters to file

MODULE subroutines
    USE CONSTANTS
    USE BLOCKMOD

    IMPLICIT NONE

CONTAINS
    SUBROUTINE init(blocks, mesh)
        ! Initialize the solution with dirichlet B.C.s
        TYPE(BLKTYPE)  :: blocks(:)
        TYPE(MESHTYPE) :: mesh

        ! INITIALIZE BLOCKS
        CALL init_blocks(blocks)
        ! WRITE BLOCK CONNECTIVITY FILE
        CALL write_blocks(blocks)
        ! INITIALIZE MESH
        CALL init_mesh(blocks)
        ! INITIALIZE TEMPERATURE WITH DIRICHLET B.C.
        CALL init_temp(blocks)
        ! CALC SECONDARY AREAS OF INTEGRATION
        CALL calc_2nd_areas(blocks)
        ! CALC CONSTANTS OF INTEGRATION
        CALL calc_constants(blocks)
    END SUBROUTINE init

!     SUBROUTINE solve(blocks, min_res, max_iter, iter)
!         ! Solve heat conduction equation with finite volume scheme
!         TYPE(BLKTYPE) :: blocks
!         ! Minimum residual criteria for iteration, actual residual
!         REAL(KIND=8) :: min_res, res = 1000.D0
!         ! iteration number, maximum number of iterations
!         ! iter in function inputs so it can be returned to main
!         INTEGER :: iter, max_iter
!         INTEGER :: i, j

!         INCLUDE "mpif.h"
!         REAL(KIND=8) :: start_solve, end_solve
!         WRITE(*,*) 'Starting clock for solver...'
!         start_solve = MPI_Wtime()

!         iter_loop: DO WHILE (res >= min_res .AND. iter <= max_iter)
!             ! Iterate FV solver until residual becomes less than cutoff or
!             ! iteration count reaches given maximum

!             ! INCREMENT ITERATION COUNT
!             iter = iter + 1
!             ! CALC NEW TEMPERATURE AT ALL POINTS
!             CALL calc_temp(blocks)
!             ! SAVE NEW TEMPERATURE DISTRIBUTION
!             DO j = 2, JMAX - 1
!                 DO i = 2, IMAX - 1
!                     mesh%T(i,j) = mesh%T(i,j) + mesh%Ttmp(i,j)
!                 END DO
!             END DO

!             ! CALC RESIDUAL
!             res = MAXVAL( ABS( mesh%Ttmp(2:IMAX-1, 2:JMAX-1) ) )
!         END DO iter_loop

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

!         ! SUMMARIZE OUTPUT
!         IF (iter > max_iter) THEN
!           WRITE(*,*) 'DID NOT CONVERGE (NUMBER OF ITERATIONS:', iter, ')'
!         ELSE
!           WRITE(*,*) 'CONVERGED (NUMBER OF ITERATIONS:', iter, ')'
!         END IF
!     END SUBROUTINE solve

    SUBROUTINE output(mesh, iter)
        ! Save solution parameters to file
        TYPE(MESHTYPE), TARGET :: mesh
        REAL(KIND=8), POINTER :: Temperature(:,:), tempTemperature(:,:)
        INTEGER :: iter, i, j

        Temperature => mesh%T(2:IMAX-1, 2:JMAX-1)
        tempTemperature => mesh%Ttmp(2:IMAX-1, 2:JMAX-1)
        ! Write final maximum residual and location of max residual
        OPEN(UNIT = 1, FILE = "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
        WRITE (*,*), "IMAX/JMAX", IMAX, JMAX
        WRITE (*,*), "iters", iter
        WRITE (*,*), "residual", MAXVAL(tempTemperature)
        WRITE (*,*), "ij", MAXLOC(tempTemperature)

        ! Write to file
        OPEN (UNIT = 2, FILE = "SolnInfo.dat")
        WRITE (2,*), "Running a", IMAX, "by", JMAX, "grid took:"
        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)"
        WRITE (2,*)
        WRITE (2,*), "Found max residual of ", MAXVAL(tempTemperature)
        WRITE (2,*), "At ij of ", MAXLOC(tempTemperature)
        CLOSE (2)
    END SUBROUTINE output
END MODULE subroutines
```

Listing 5: Main subroutines for solver (initialization/solution/output)