Lecture 7 – Spatial Domain Decomposition and Data Structures for Multi-Block (Domain) Computations

- ***Domain Decomposition*** **has several meanings**
- **There are different types of domain decomposition:**
  - ***Spatial/grid domain decomposition*** is where a single domain is broken up into multiple "blocks" or sub-domains in order to
    - Make grid generation easier
    - Use parallel computing to solve larger problems and/or solve a problem faster
  - ***Discipline domain decomposition*** is where a domain is broken up into different physics (with different equation sets and/or models)
    - For different disciplines
    - For different physics
  - ***Algorithm domain decomposition*** is where different algorithms are used depending on purpose
    - Convergence acceleration (Eg multi-grid) vs fine-grid
  - ***Processor/Parallel block domain*** *decomposition* is where multiple blocks are distributed across multiple processors in such a way
    - Evenly distribute the work-load (load balancing) across the processors

Data Structures for Spatial/Grid and Processor/Parallel Domain Decomposition

- **When developing a parallel code to solve an engineering simulation, we have to consider**
  - Where data must be exchanged at block and processor boundaries during the course of the algorithm
  - What data must be exchanged
  - If barriers are necessary to synchronize the processors
  - What blocks and processors must exchange data

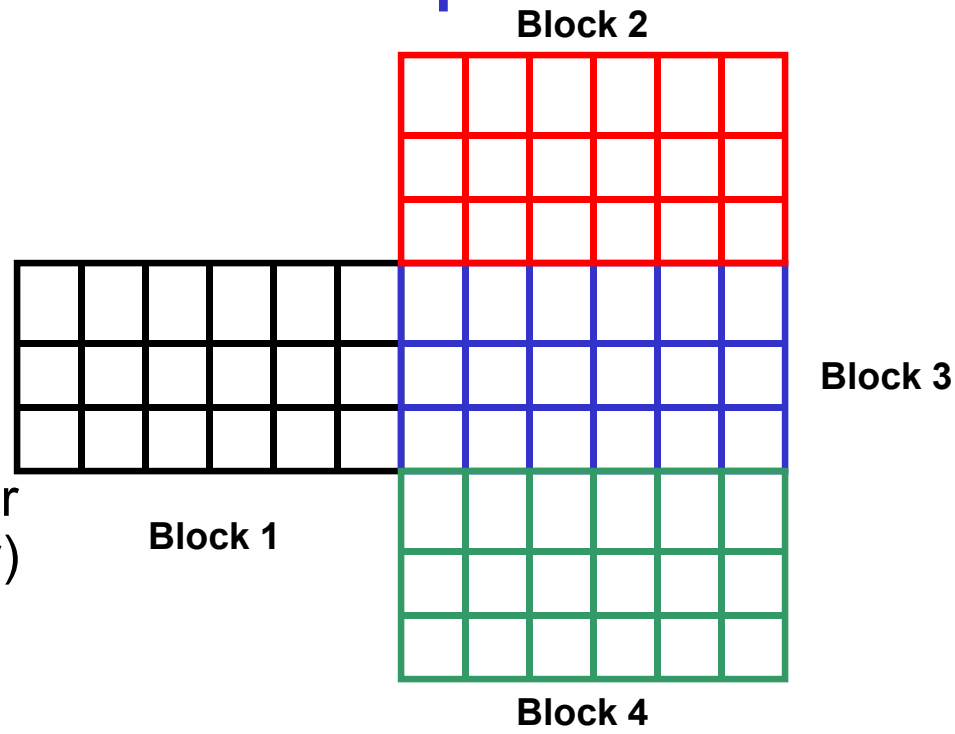
- **Different engineering simulation codes may use different types of computational grid and differencing (or integration) stencil**
  - Multi-block structured grids
  - Multi-block unstructured grids
  - Multi-blocks of hybrid, structured/unstructured grids

Multi-Block Data Structures

- **Inter-block and inter-processor communication is dependent on block neighbor information**
- **Each block (of computational points or cells) must keep track of**
  - Its own block and processor number
  - Block and processor numbers of its neighbors (E, W, N, S, H, Y)
  - The boundary conditions to be applied along its face (edge)
  - Additional information for special applications (such as parent/child relationships for embedding-adaptation)
- **Upon execution, each block can then determine**
  - The processor which has been assigned to it
  - The processors associated with its neighbors
  - If communication is necessary for each block face (edge)

Example: Spatial/Grid Decomposition

- **Flow through a sudden expansion, or heat transfer in a "T" strut, etc.**
- **Each block must store**
  - Its N, S, W, E, H, Y neighbor block numbers (connectivity)
    - Example: Block-1 has neighbors 0,0,0,3,0,0
  - The boundary condition types (if any) at each point along is N, S, W, E, H, Y faces (edges)
    - Example: Block-1 has a BC type of 1 (solid boundary) along its southern boundary or –3 along its eastern boundary (which says that this is a block/block boundary



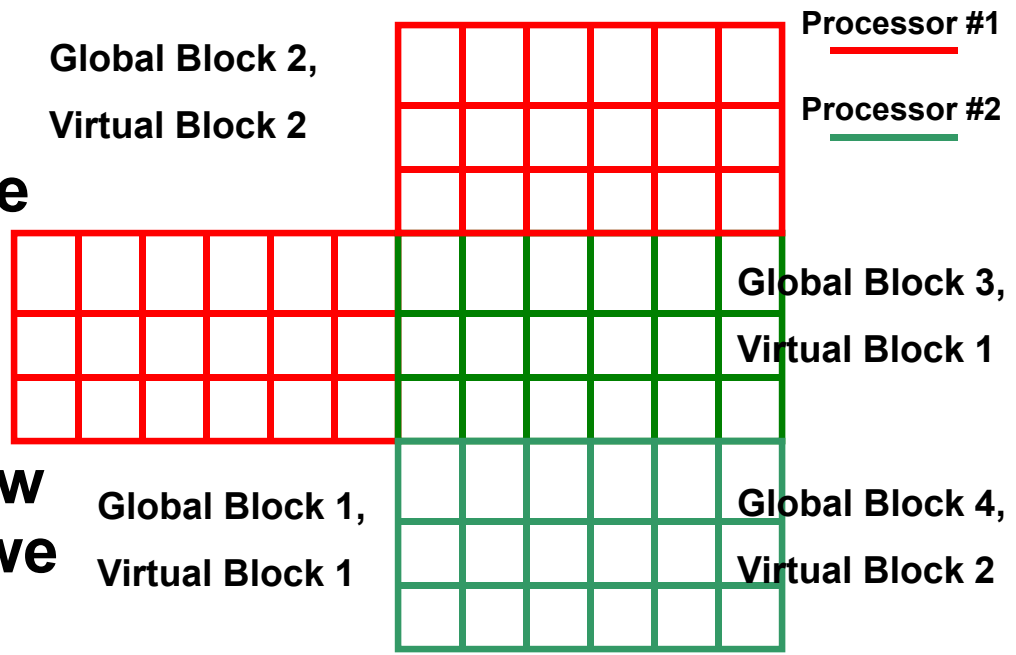
- Another way to store BC information would be to store regions of a face (sub-faces) with descriptions of the region dimensions and BC type
- Yet another way to store BC information would be point-by-point. This may make the user input description tedious, however.

Creation of Multi-Block Grids

- **The creation of a multi-block structured, unstructured, or hybrid grid is generally performed with a separate grid-generation code.**
  - A *multi-block restart file* can be written that contains the block dimensions, connectivity, and coordinates
    - PLOT3D is an “old” file format for restart files
    - CGNS is a new (modern) data-base format for restart files (you can find information regarding CGNS at: www.cgns.org
  - The initial values of the simulation code may be written in these same formats either as a separate file (PLOT3D) or as part of the data-base (CGNS)
- **In the case of your project simulation code, the next step will be to write a separate code to break up the original single-block grid into multiple blocks with connectivity and BC information (spatial/grid domain decomposition) for serial execution (projects 2 and 3).**
  - Note, however, that you will ultimately need to create the data structure for parallel execution (projects 4 and 5).

Example: Processor/Parallel Domain Decomposition

- **Processor/parallel domain decomposition may be used to determine the processor assignments to each block**
- **For instance, say we knew ahead of execution that we could use 2 processors**
  - Processor #1 is assigned to block 1 and 2
  - Processor #2 is assigned to block 3 and 4
- **So at this point, each processor can find out its neighboring processors via the block neighbor information**



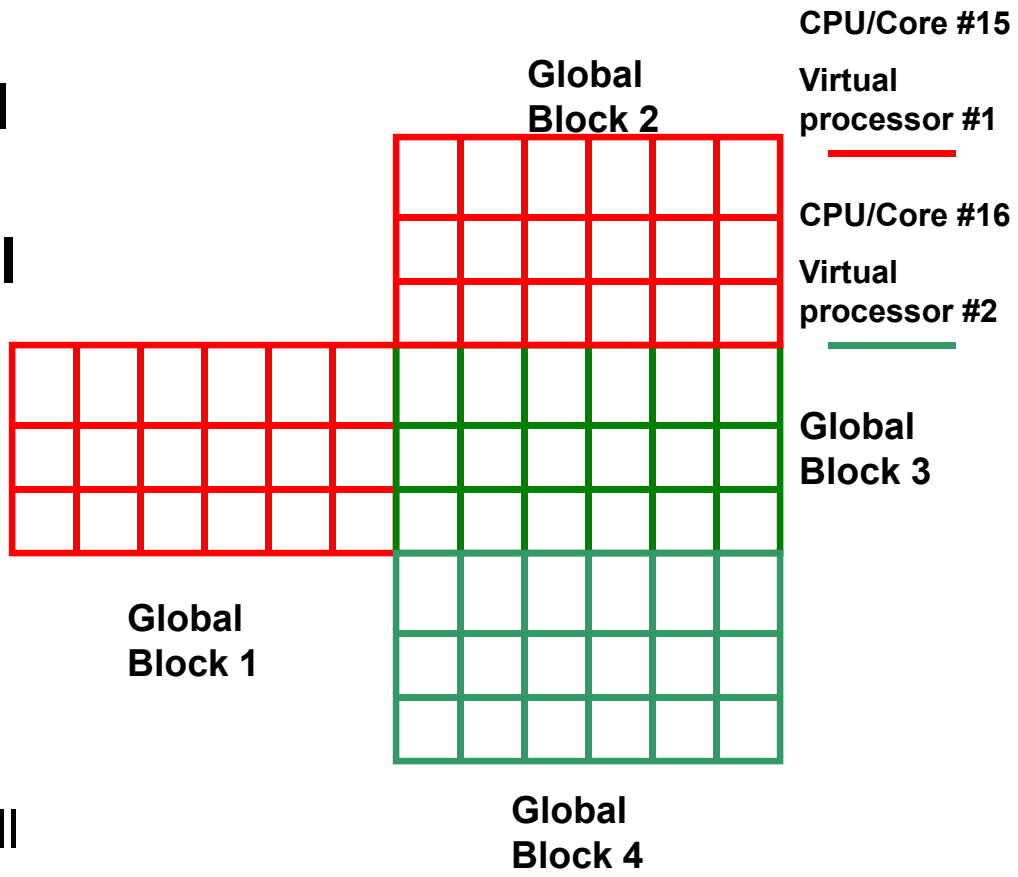
- **For instance, block-1:**
  - Knows that its east boundary is a block/block interface from its BC information for that face (edge)
  - Knows its east neighbor is block-3
  - Can determine that block-3's processor is #2 so the east face (edge) is also an inter-processor boundary

Example: Block and Processor Decomposition

- **So before execution even begins, we know the following for each processor:**
  - Block numbers of all blocks
  - Grid dimensions of all blocks
  - Coordinates and physical variables at nodes or cell-centers of all blocks
  - Boundary condition types for all points on all faces (edges) of all blocks
  - Neighbors of all blocks (N, S, W, E, H, Y)
  - Inter-processor boundaries (if any) for all blocks
  - The processor numbers assigned to the neighbors across inter-processor boundaries (This tells us who to communicate with)
- **Most of this information could be generated by a pre-processor spatial decomposition or grid-generation code, then written out to a file for each processor that could be read at the beginning of simulation execution.**
  - Determination of inter-processor boundaries and block assignments to virtual processors (last 2 steps above) would come later during processor/parallel domain decomposition

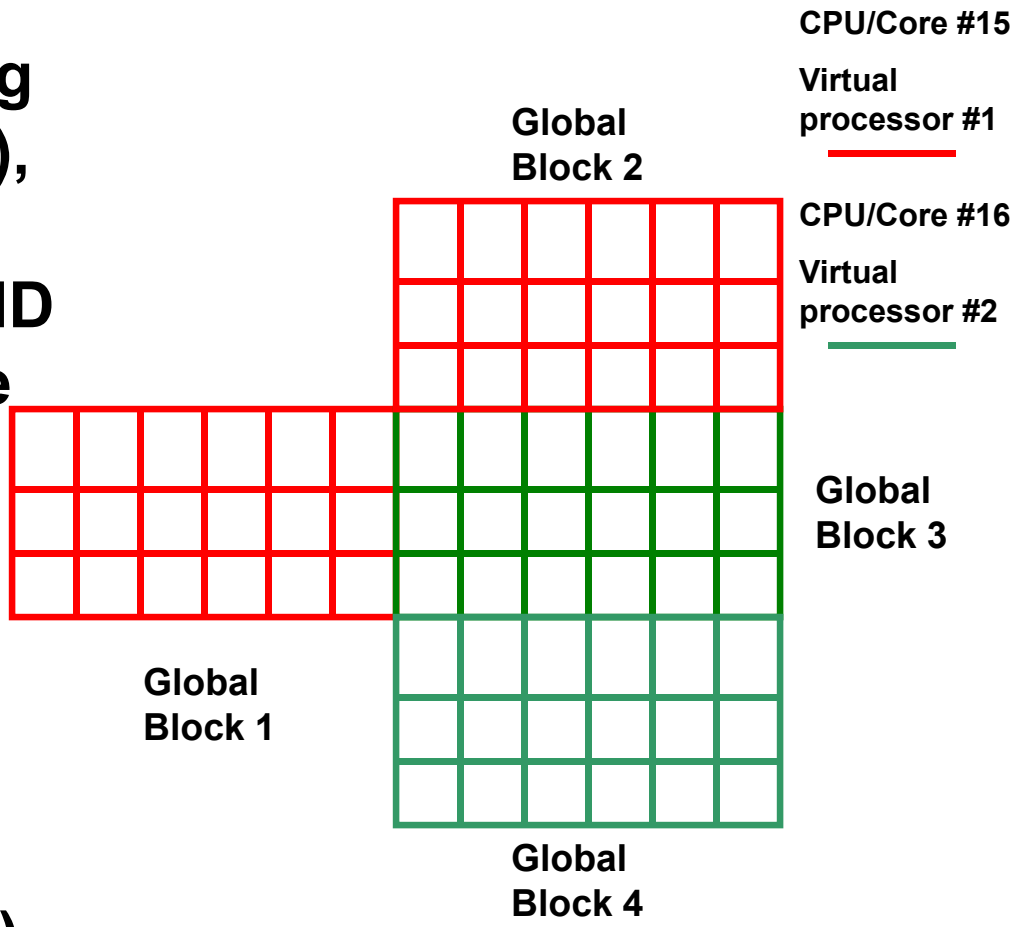
Example: Virtual to Physical Processor Mapping

- **Upon execution, physical processor numbers are mapped to computational processor numbers by MPI**
  - Example: CPU/Core #15 is assigned all the blocks of computational processor #1 (red blocks) and reads the pertinent block information
  - CPU/Core #16 is assigned all the blocks of computational processor #2 and reads all the pertinent block information (green blocks)
  - Solution iteration begins



Example: SPMD vs MPMD Parallel Computing

- If all blocks are executing the same analysis (code), then this type of parallel computing is called SPMD (Single Process, Multiple Data)
- If the red blocks were executing one type of analysis and the green blocks were executing a different analysis (Multi-Disciplinary Simulations), then this type of parallel computing is called MPMD (Multiple Process,Multiple Data)



Data Structures to Aid in Multi-Block Discretization and Parallel Communication

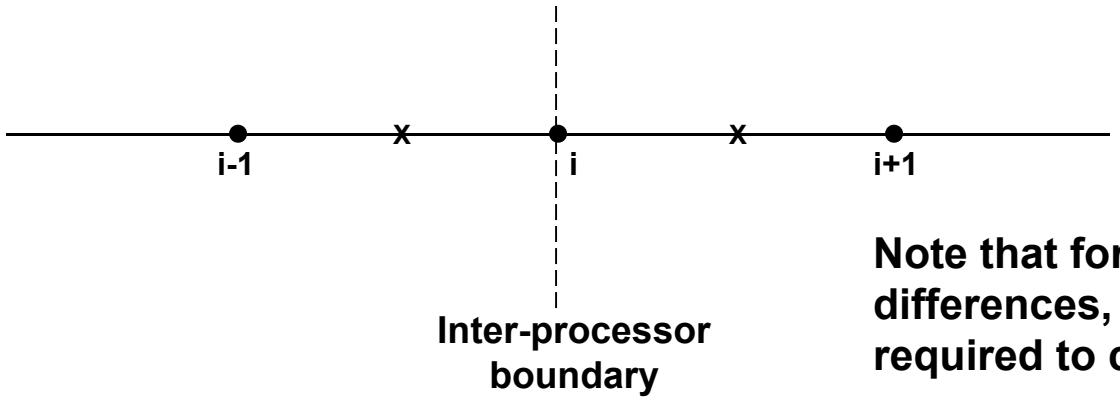
- **So far, we have only addressed issues related to breaking up the grid, assigning sub-groups of blocks to processors, and making the boundary condition specification more general**
- **We now must address the communication that takes place between blocks and processors during iteration of the solution algorithm**
- **This involves the discretization stencil which is dependent on the type of algorithm being used**
  - Finite difference, finite volume, finite element, other

Data Structures to Aid in Multi-Block Discretization and Parallel Communication

- **The differencing (or integration) stencil in engineering simulations depends on the governing equations to be solved and the numerical algorithm used to solve them**
  - For instance, most 2nd order-accurate difference operators used for the physical science simulations use a 3-point stencil

$$\left.\frac{\partial \phi}{\partial x}\right|_i = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} \quad or \quad \left.\frac{\partial^2 \phi}{\partial x^2}\right|_i = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2}$$

  - One must keep the largest stencil used during the course of execution in mind when developing parallel code



Note that for typical 1st and 2nd differences, one adjacent point is required to create the stencil

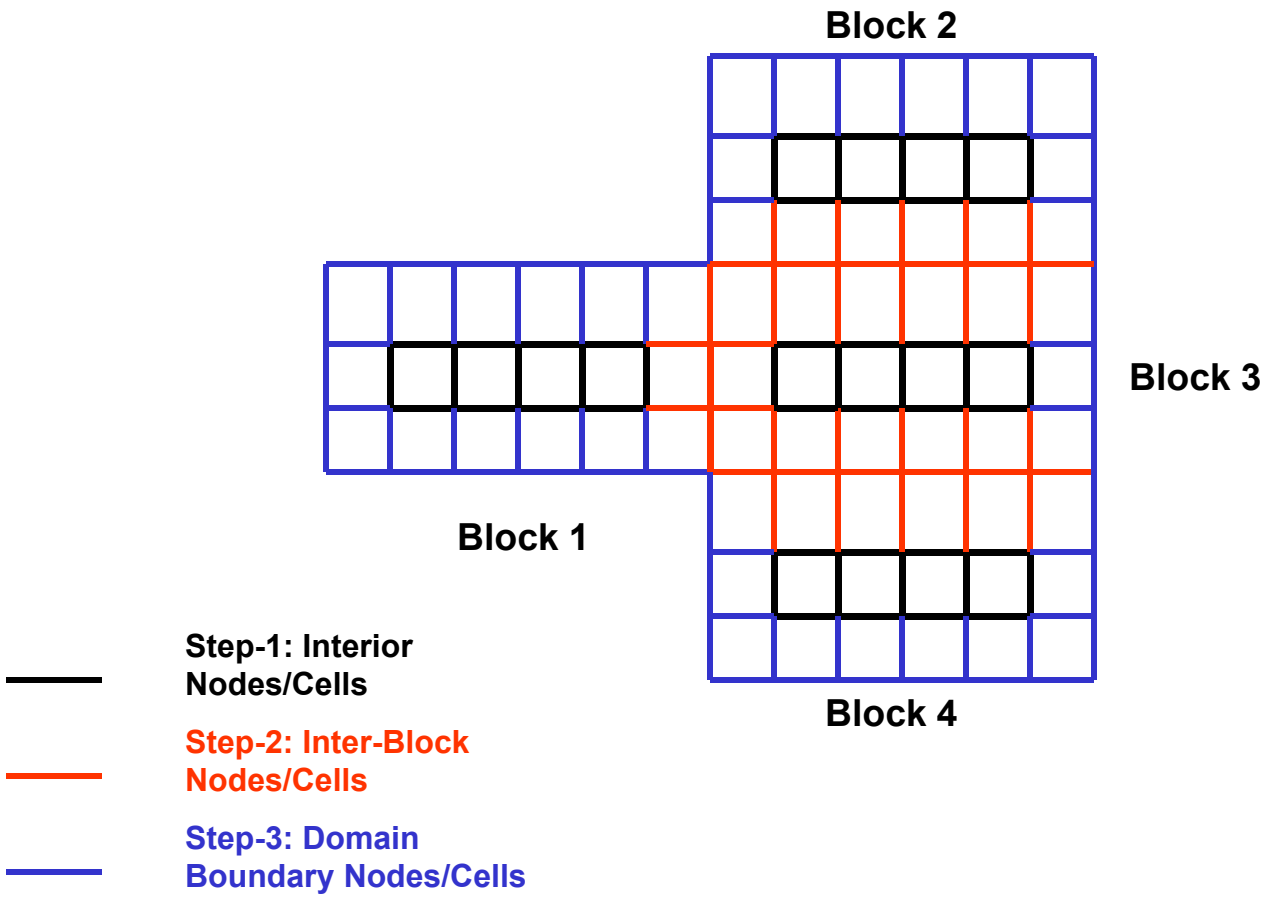
Data Structures to Aid in Multi-Block Discretization and Parallel Communication

- **Different strategies may be used to create the stencil at the inter-block boundary**
  - Since the block neighbor and boundary condition information is known for each side of every block, we can quickly determine the adjacent block processor numbers
- **Method-1: Information may be passed on the fly during the construction of the stencil at the edge**
  - Tends to be inefficient due to a large number of small-size packets of information exchanged
- **Method-2: Stencil may be constructed via an "accumulation" of forward and backward stencils at the edges (lecture 5)**
  - Sum the contributions from different processors at the edges at the end of stencil construction (cell-vertex schemes)
  - Requires additional logic to handle various stencil operators toward edges
- **Method-3: Information may be passed at the end of each iteration into buffers (halo or ghost cells/nodes) that "extend" each block**
  - Requires additional storage but is efficient
  - Ghost cells/nodes may also be used to perform boundary conditions

Method-1: "On the Fly" Stencil Construction

- **Pseudo-Code for multi-block, structured-grid simulations using method-1:**
  - Loop over blocks
    - Loop to construct discretization of stencil at internal nodes/cells
    - Loop to construct discretization of stencil at internal north boundary nodes/cells (reach across to north neighbor)
    - Loop to construct discretization of stencil at internal south boundary nodes/cells (reach across to south neighbor)
    - Loop to construct discretization of stencil at internal west boundary nodes/cells (reach across to west neighbor)
    - Loop to construct discretization of stencil at internal east boundary nodes/cells (reach across to east neighbor)
    - Loop to construct discretization of stencil at internal hither boundary nodes/cells (reach across to hither neighbor)
    - Loop to construct discretization of stencil at internal yonder boundary nodes/cells (reach across to yonder neighbor)
    - Construct discretization of stencil at block corner nodes/cells (reach across to two appropriate neighbors for each)
    - Perform boundary conditions on all non-inter-block faces
  - Update variables and make sure inter-block boundary nodes/cells are consistent

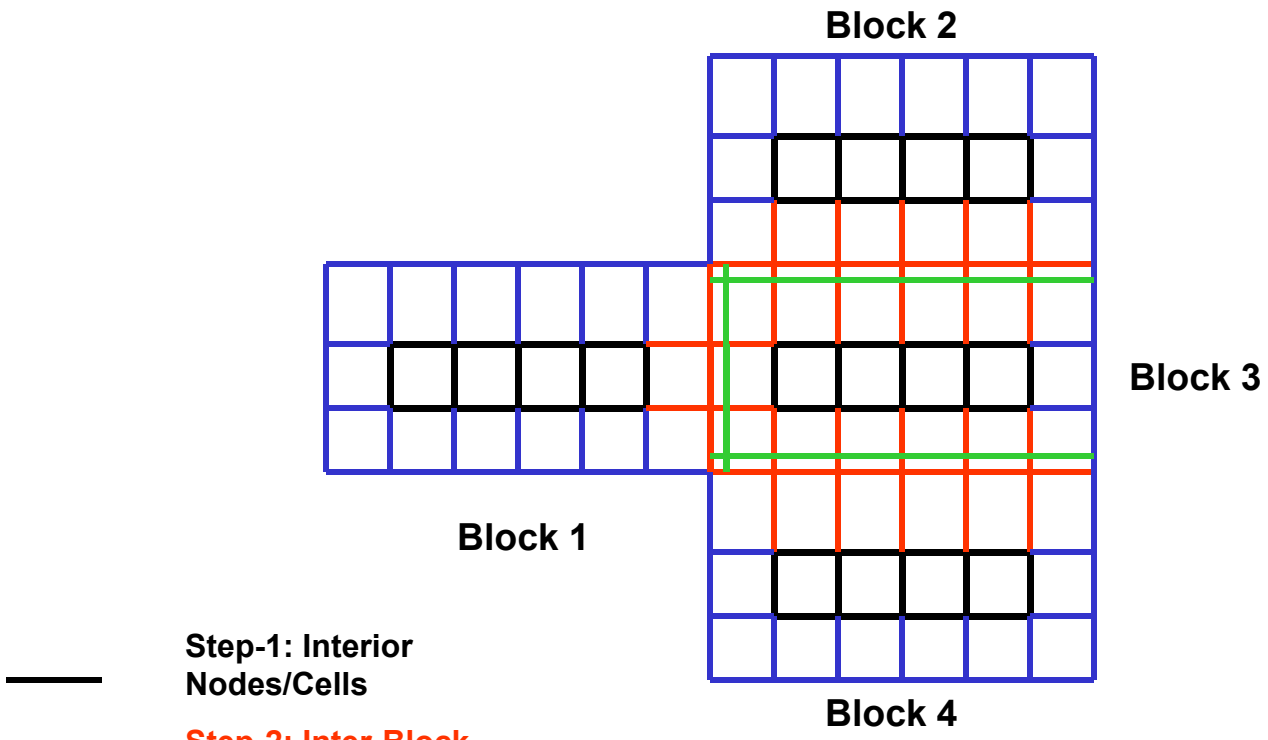
Example: “On the Fly” Stencil Construction



Method-2: "Accumulation" Stencil Construction

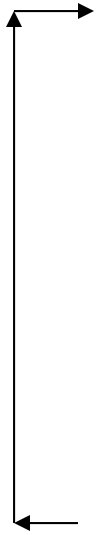
- **Pseudo-Code for multi-block, structured-grid simulations using method-2:**
  - Loop over blocks
    - Loop to construct discretization of stencil at internal nodes/cells
    - Loop to construct discretization of stencil at north boundary using combination of central and one-sided treatments
    - Loop to construct discretization of stencil at south boundary using combination of central and one-sided treatments
    - Loop to construct discretization of stencil at west boundary using combination of central and one-sided treatments
    - Loop to construct discretization of stencil at east boundary using combination of central and one-sided treatments
    - Loop to construct discretization of stencil at hither boundary using combination of central and one-sided treatments
    - Loop to construct discretization of stencil at yonder boundary using combination of central and one-sided treatments
    - Perform boundary conditions on all non-inter-block faces
  - Accumulate contributions at inter-block faces
    - Message pass (send/receive) boundary contributions
  - Update variables

Example: “Accumulation” Stencil Construction



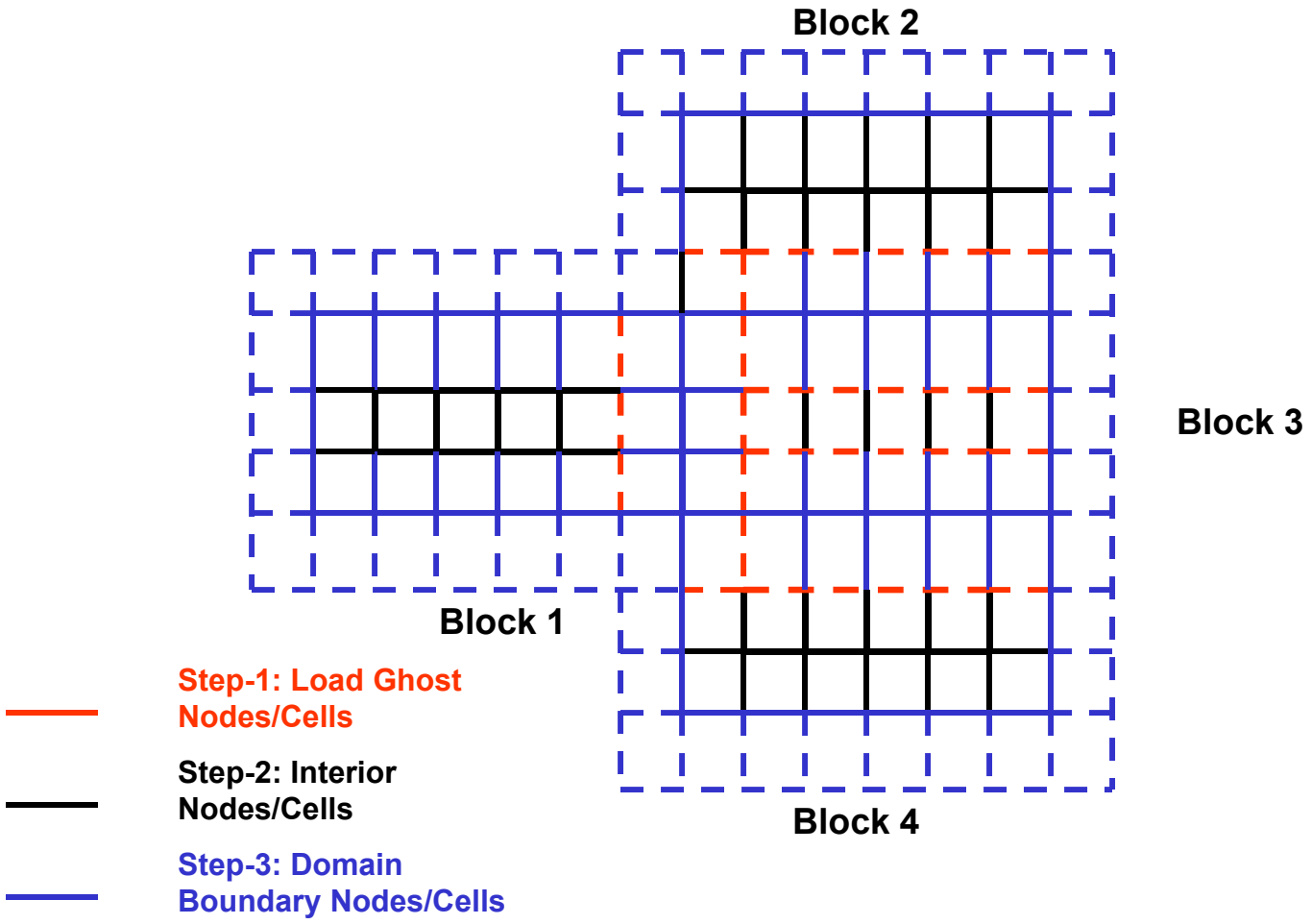
Method-3: "Ghost Cell/Node" Stencil Construction

- **Pseudo-Code for multi-block, structured-grid simulations using method-3:**
  - Create enlarged-block data-structure that has additional planes beyond each face (halo or ghost nodes/cells)
    - Number of additional planes beyond each face is (stencil_size-1)/2
  - Read original blocks into enlarged-block data structure



  - Loop over enlarged-blocks
    - At inter-block boundaries, pack halo/ghost nodes/cell information with appropriate neighbor information
    - At domain boundaries, create appropriate halo/ghost node/cell information for boundary condition treatment
  - Loop over enlarged-blocks
    - Loop to construct discretization of stencil at all original block nodes/cells
    - Perform boundary conditions on all non-inter-block faces
  - Update variables

Example: “Ghost Cell/Node” Stencil Construction



Multi-Block Unstructured-Grid Treatments

- **The multi-block data structure and inter-block stencil construction for unstructured-grids is similar**
  - Methods 1-3 can be constructed for unstructured-grids
  - Except instead of looping over faces/corners, the boundary is treated as an unstructured column vector of cells/nodes
  - The indirect-addressing associated with the randomness of the grid cells/nodes makes the treatments more tedious, time-consuming, and memory-consuming
  - For unstructured-grids, spatial/grid decomposition is often combined with processor/parallel decomposition
    - Unstructured grid-generation tools often create the grid over the entire domain as a single block

Hybrid Data Structures for Multi-Block Solution Procedures

- **Unstructured-grid procedures have random node numbering for each cell**
  - This allows for cells to be created arbitrarily to fill in a domain and straightforward cell division for grid-adaptation schemes
  - However, it requires a data structure that has:

    nodes → edges
    edges → faces
    faces → cells
    neighbor cells → cells

    Note that these are essentially objects (classes)

    Thus the data structure between the nodes and the cells can be multi-level which leads to expensive indirect addressing of memory
- **We can greatly reduce this overhead by treating blocks in an unstructured grid format while keeping the cells inside of the block in a structured-grid format**
  - This is the concept of "multi-block" structured grids

Multi-Block Structured Grids

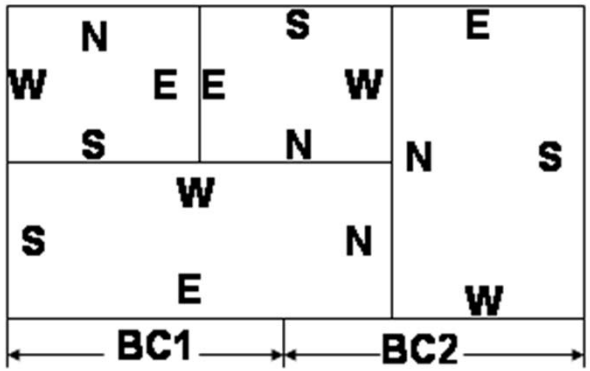
- **So the advantage of multi-block structured grids are:**
  - Practically any geometry can be gridded using unstructured-grid like block topologies that are connected using a block-neighbor data structure
    - The resulting data structure does not need all of the overhead of a totally unstructured-grid procedure
    - Only neighbor information is necessary since the nodes on adjacent faces of different blocks can be easily mapped
    - Lends very well to both distributed- and shared-memory parallel computer architectures
  - The operators inside of the block are all structured using direct addressing of memory and rapid indexing
    - Lends very well to vector and/or graphical processors
  - Grid adaptation can be performed uniformly and directionally on a per-block or per-sub-block basis

Multi-Block Structured Grids – Additional Considerations

- **Additional information must now be added to single-block solvers to allow for multi-block topologies:**
  - For each given block, store:
    - Global block number
    - Block type (in multi-disciplinary problems, “solid” or “fluid”)
      - This could also include “overlaid”, “structured”, “unstructured”
    - Processor number (parallel) the block belongs to
    - Possibly the orientation
  - For each given block face, store:
    - The start and end indices (note that start index can be larger than end index to reverse order of processing)
    - The adjacent global block number (if one exists)
    - The face of the adjacent block (if one exists)
    - The orientation of the adjacent block OR the start and end indices in the directions of the adjacent block

Possibility of Sub-Faces

- **In many cases, you will find that it is inconvenient topology-wise to create blocks in such a fashion that the block faces have a single boundary condition over the entire face**
- **Multiple sub-faces can be used to allow for:**
  - Blocks to be staggered
  - Multiple physical boundary conditions to be applied along a face
- **This de-couples the block topology from the physical boundary conditions**



- **Sub-faces are not necessary for your projects. However, they are handy for general purpose procedures.**

Sub-Face Considerations

- **Incorporating sub-faces into a solver requires**
  - Number of sub-faces for each face (S,N,W,E,H,Y) as part of the stored block information
  - Face information:
    - Start and stop indices (in each direction) of the sub-face
      - Must have the capability to traverse in positive or negative i-, j-, k- directions (ie start index can be larger than stop index)
      - Note that traversing of face information MUST be consistent at inter-block boundaries since message passing has an order to it
    - Physical boundary condition type or
      - Along with any physical information regarding that boundary condition (eg total pressure, total temperature, etc)
    - Neighboring block number
      - Along with the start and stop indices (in each direction) that is point-matched to the given sub-face. Note that traverse direction of data in neighbor must be consistent with your own face traverse direction.

Blocks and Faces as Objects (Abstraction)

- **When coding up your projects, you could think of block faces abstractly (generically).**
  - You could create a (linked) list of faces that have
    - The list could consist of pointers to the temperature of the face nodes
    - Physical boundary conditions
    - Inter-block boundary treatments
  - For each face in the face list, you could store
    - Start and end point indices to loop over
    - For physical boundaries, the boundary condition type
    - For inter-block boundaries,
      - Neighbor block, face, and processor numbers
      - Indices of start and end points to loop over in each direction
- **You could also think of blocks abstractly (generically).**
  - You could create a (linked) list of blocks that have
    - Block type (for your projects, you only have solid blocks)
    - Processor number
    - Dimensions in each direction
    - Number of sub-faces in each direction

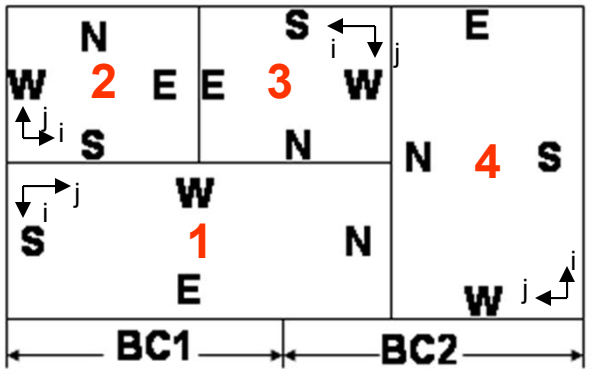
This is the unstructured-grid manner of processing blocks and faces

Blocks and Faces

- **<u>OR</u> you could simply loop over the south, north, west, and east faces**
  - Performing kernels for
    - Physical boundary conditions
    - Inter-block boundary conditions
  - This tends to produce repeatable sections of code, however, since the faces are doing similar tasks
- **<u>AND</u> loop over the blocks in a Cartesian grid system (since in your problem, the grid is broken up into N x M blocks)**
  - This is NOT typical, however, so the creation of a list of blocks is more general

Options for Orientation

- **Keeping track of block orientation is only important in terms of neighbor information to find out what neighbor face is adjacent to your face**
- **Block orientation can be performed either by**
  - Keeping track of how a given face maps to a standard-oriented block face
    - Let standard i- be 1 and standard j- be 2.
      - block 2 would have orientation of 1,2
      - block 1 would then have an orientation of -2,1.
      - block 3 would then have an orientation of -1,-2
      - block 4 would then have an orientation of 2,-1
    - BE CAREFUL! All blocks should be right-handed to make integrations come out correct.
  - OR Keeping track of start and end indices of each face (or sub-face) and matching them to neighbor correctly
    - Note that block 2's east face is ordered in the opposite direction as block 3's east face



TAGS for Send/Receiving Messages at Inter-Block Boundaries

- **Tags of Send and Receive Messages must be unique AND must be the same on the Send and Receive sides**
- **Combinatorial mathematics can be used to determine a unique TAG number:**
  - A combination of 2 positive, non-zero numbers A and B will be unique if A+BN is unique. For this to happen, N=max(A).
  - A combination of 3 positive, non-zero numbers A, B, and C will be unique if A+BN+CM is unique or when N=max(A) and M=max(A)+max(B)max(A).
  - A combination of 4 positive, non-zero numbers A, B, C, and D will be unique if A+BN+CM+DQ is unque or when N=max(A), M=max(A)+max(B)max(A), and

    Q=max(A)+max(B)max(A)+(max(A)+max(B)max(A))max(C) or Q=(max(A)+max(B)max(A))(1+max(C))
  - and so on...

TAGS

- **The number of unique tags available will depend on the number of variables that you use, A, B, C, D, etc.**
- **For your projects,**
  - A could be your block number
  - B could be your face number (S=1, N=2, W=3, E=4)
  - C could be your neighbor's block number
  - D could be your neighbor's face number that is adjacent to you
  - Etc.
- **There are other combination of variables that can be used to make up TAGS**
- **Try to keep the TAG formulas as simple as possible that give you the required number of unique values**

Project-2: Single-Block Decomposition

- **Now that we have shown that we can develop a single-block engineering simulation code, let's break up the problem into multiple blocks.**

- **For your simulation code developed under project-1, you need to write a code that will break up the domain into multiple blocks that will still run on a single processor**

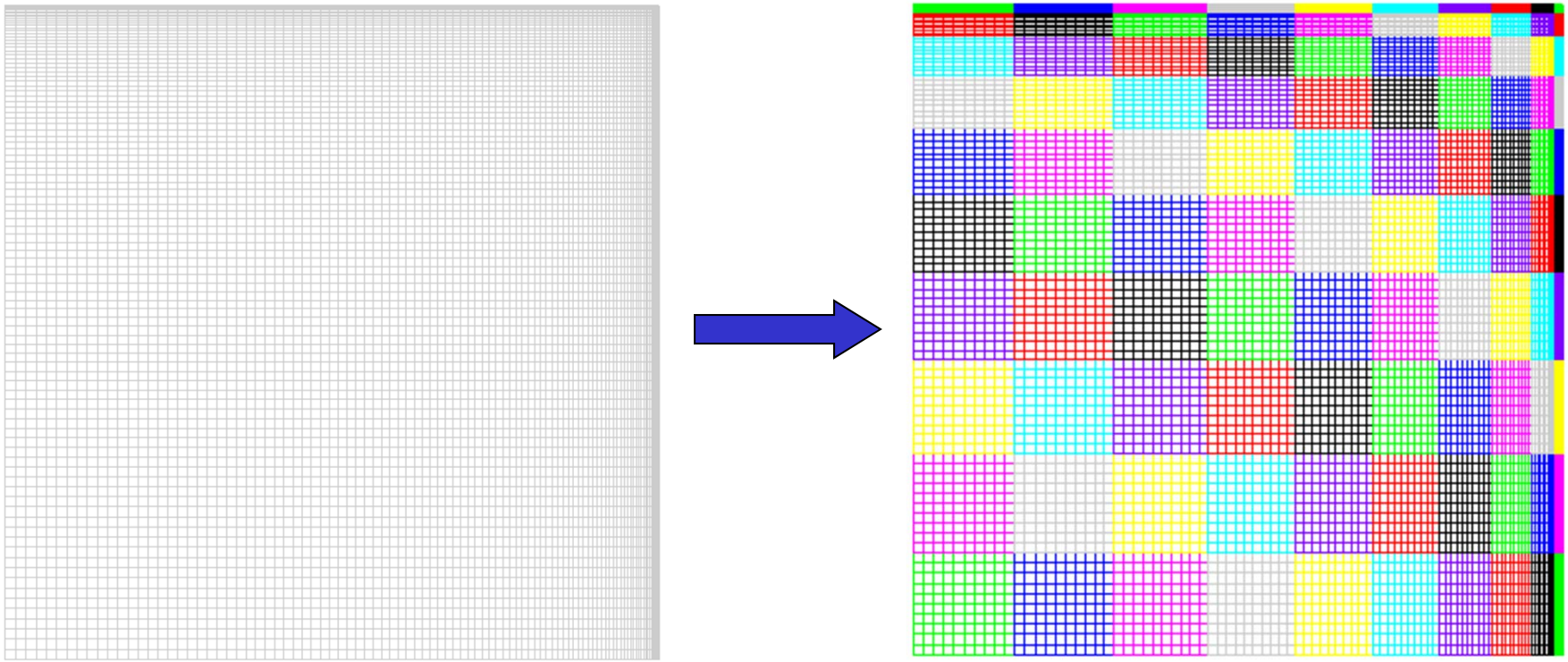
- **This is an intermediate step prior to developing a code for multiple processors**

Project-2: Single-Block Decomposition

- **For the Heat Conduction (Default) problem, develop a code that will divide the domain into N x M blocks, each with dimensions of 1+(IMAX-1)/N x 1+(JMAX-1)/M**
  - Where IMAX = JMAX = 101 and 501 for default heat-conduction problem
  - The code should be completely general so that N and M can be arbitrary
- **Develop a multi-block data structure that writes out a connectivity file with the following information for each block:**
  - Global block number of each block
  - Neighbor block numbers of each block
  - Boundary conditions for each side of block
    - Could be unique to each side
    - Could have regions for each side
      - Number of sub-regions for each side (if you decide to have sub-regions)
      - Dimensions of each sub-region (start and stop indices)
      - Boundary condition for each sub-region
    - Could be defined point-by-point
  - Block orientation (all blocks will probably have the same orientation for our problems).
- **And multi-block grid and temperature files (Plot3D or other format) that has the coordinates and initial temperature along with**
  - Number of global blocks
  - Block dimensions of each global block

Project-2: Single-Block Decomposition

- **You should end up with something similar to:**



in the x', y' frame with associated connectivity and boundary condition file(s)

- **Project-3 will be to write a multi-block serial code that will solve your simulation problem**

Project-2: Single-Block Decomposition

- **Due Tuesday 10/27:**
  - An overview of your simulation problem
    - Describe the problem, algorithm, and boundary conditions
  - Listing of your spatial/grid decomposition code
  - A plot of your decomposed computational grid (or data decomposition) for two-different decompositions:
    - 10 x 10 (N=10, M=10) grid for heat-conduction problem
    - 5 x 4 (N=5, M=4) grid for heat-conduction problem
    - Using the 101 x 101 and 501 x 501 grids
  - A sample listing of your connectivity/boundary condition file(s)