Controlling computational efficiency

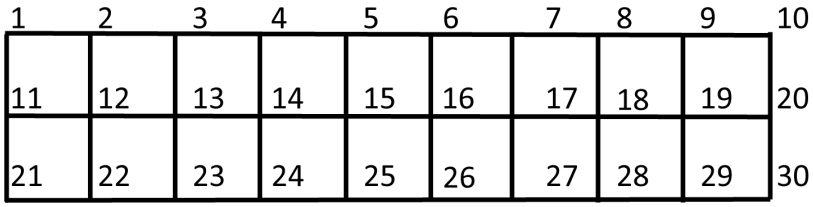
Two, maybe not so obvious, items that have a strong influence on how efficiently a large scale simulation is carried out are:

Data Ordering: Influences two key areas: Cost of equation solving and cost of data access. The ordering of the unknowns has a strong influence on the time required to solve the global systems. The order data is stored is very important since on today's machines the "cost" of memory access is higher than computation. Thus you want the physical location of the data needed for a calculation to be in cache memory as often as possible. (This issue has to be addressed at multiple levels, we just consider the highest level in what we look at here.)

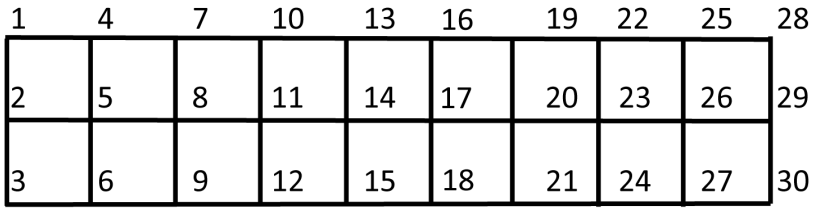
Load Balance: Without maintaining load balance between processes parallel scalability is lost. Must determine methods to distribute the computing load as equally as possibly while keeping the communication cost as low as possible.

Cost of ordering on the global system solution is a function of the type of solver used, but is important in all cases. Lets look at the simple case of a banded direct solver: The solution time is proportional to nb^2 where n in the number of equations and b is the bandwidth defined as b=max[(high node # - low node #) +1] over all elements. Typically the rows and columns of the global system are based on the nodal numbering (or based on the dof

holders which can be associated with mesh topological entities). Thus changing the node numbering will change the rows and columns in the stiffness matrix. Consider the simple mesh with 1 dof/node and two different node point numberings:



$b=(12-1)+1=12$
$nb^2=30(12^2)=4320$



$b=(5-1)+1=5$
$nb^2=30(5^2)=750$

There is a factor of 5.76=4320/750 in this simple case.

A reference on some basics related ordering, and other aspects of direct solution to matrix equations:
Alan George and Joseph Liu, *Solution of Sparse Positive Definite Systems*, Prentice Hall, 1981.

There are a number of approaches that have been taken to defining ordering with methods based on “space filling curves” or heuristics based on operations on “graphs of interactions”. Lets start with the graph-based methods.

The graph concepts used are pretty straightforward. However, we will have some terminology complications since terms like node, vertex and edge are used in each to mean different things. WRT graphs vertices, or nodes, are the things we order and edges indicated connections of vertices or nodes.

An unordered graph, $G=(X,E)$, consist of a set of entities, nodes, X, that we want to order, and a set of edges, E, that are pairs of "connected" nodes.

$$G=(X,E)$$

An ordered graph will be one were we have ordered the nodes with a labeling that we want to have. We will add a superscript to indicate particular labeling.

$$G^{\alpha}=(X^{\alpha},E^{\alpha})$$

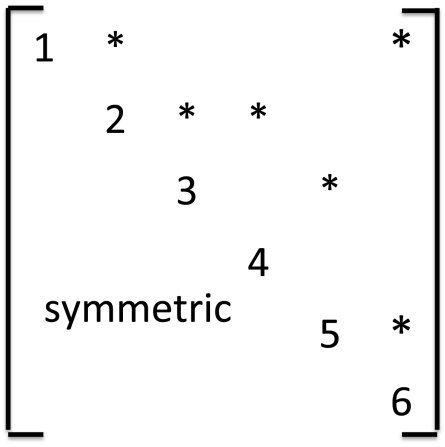
We are concerned with the labeling of the rows and columns of a stiffness matrix. Lets assume a symmetric matrix K called matrix of n equations.

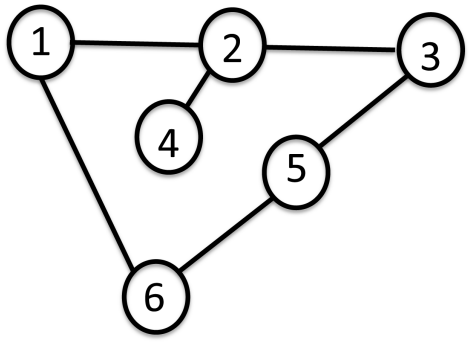
$$G^{K}=(X^{K},E^{K})$$

where each $x_i \in X^K$ corresponds to a row and we will have a set of edges E^K where the edges are defined by

$$\{x_i,x_j\}\in E^K \text{ iff } K_{ij}\neq 0 \text{ and } i\neq j$$

consider simple example:

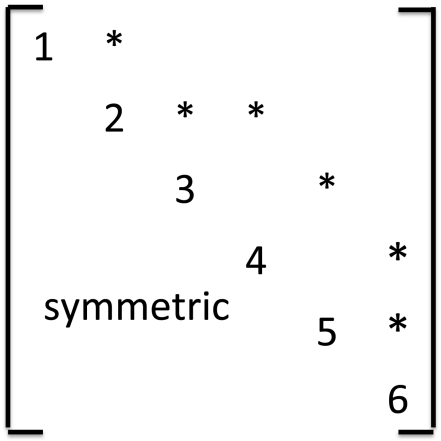


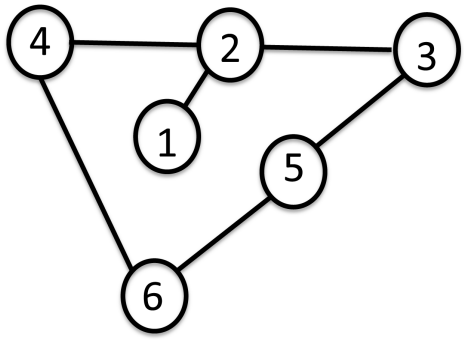


* Is a non-zero entry

Because of the non-zero 1,6 location the bandwidth is 6.

Note by changing the graph labeling we change the ordering of rows and columns which changes the sparseness pattern.





* Is a non-zero entry

With this ordering the bandwidth is only 3.

Note the overall graph structure is based on the connections, the unlabeled graph is fixed. However, the labeling does influence memory access time and cost of computation.

A few more terms:

Two nodes x_i and y_i are adjacent if $\{x_i, y_j\} \in E^k$

The adjacent set of a subset $Y \subset X$ will be denoted $Adj(\{Y\})$ and is defined as

$$Adj(\{Y\}) = \{x \in (X - Y) \,|\, \{x, y\} \in E \text{ for } y \in Y\}$$

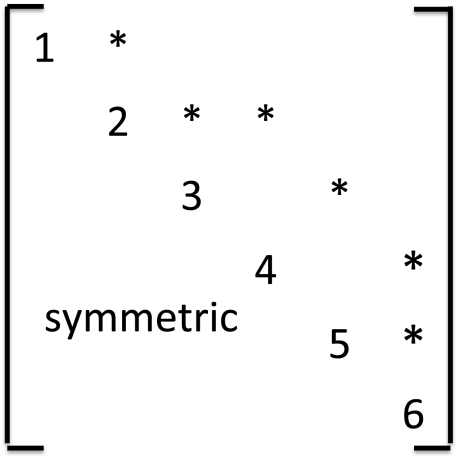
Consider graph above labeled graph and the set $Y = \{x_2, x_4\}$ then $Adj(\{Y\}) = \{x_1, x_3, x_6\}$. If Y contains a single node we write $Adj(Y)$. For example $Adj(x_4) = \{x_2, x_6\}$. The degree of Y is denoted $\deg(Y)$ is $|\,Adj(Y)\,|$ which is the number of terms in $Adj(Y)$. The $\deg(x_2, x_4) = 3$ and $\deg(x_4) = 2$.

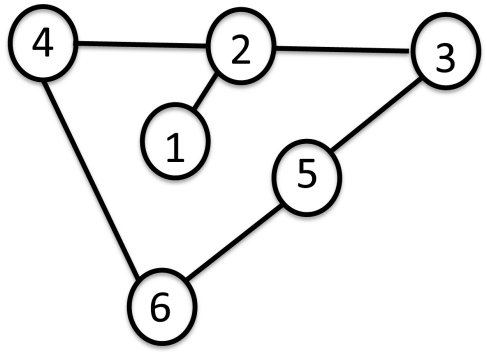
A subgraph, $G' = (X', E')$, of the graph $G = (X, E)$ is a graph for which $X' \subset X$ and $E' \subset E$

For $Y \subset X$ the section graph $G(Y)$ is the subgraph $(Y, E(Y))$ where

$$E(Y) = \{\{x, y\} \in E \mid x \in Y, y \in Y\}$$

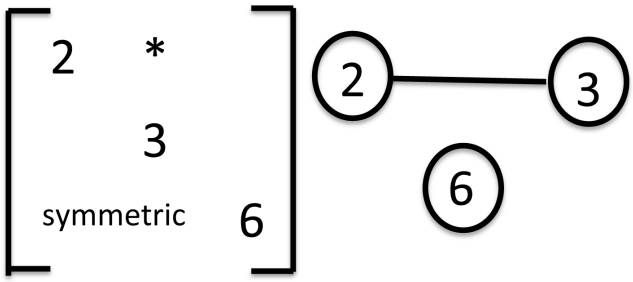
The subgraph of a matrix is equivalent to deleting rows and columns of the matrix. Consider again





* Is a non-zero entry

and select $Y = \{x_2, x_3, x_6\}$ then $E(Y) = \{x_2, x_3\}$ (just 1 edge)



Connectedness of a graph:

Consider two nodes x, y in a graph G. A path from x to y of length $l \geq 1$ is the ordered set of $l+1$ nodes $(v_1, v_2, v_3, ..., v_{l+1})$ such that

$$v_{i+1} \in Adj(v_i) \quad i = 1(1)l \text{ with } v_1 = x \text{ and } v_{l+1} = y$$

A graph is connected if each distinct pair (set of two) nodes is connected by at least one path between them. If the

whole graph is not connected, there are two or more connected component.

$Y \subset X$ is a separator if the graph $G(X-Y)$ is disconnected.

As already mentioned, a key item we use reordering for is to label the dof (i.e., row and columns of the global stiffness matrix) since this can have a strong influence of equation solution time, especially for direct solvers. Some terms related to this case for symmetric matrices are:
For row i the highest non-zero term in the i^{th} column is

$$m_i(K) = \min\{j \mid K_{ij} \neq 0, j < i\}$$

The column height is then

$$\beta_i(K) = i - m_i(K)$$

The maximum column height is defined as (one less than bandwidth defined before)

$$\beta(K) = \max\{\beta_i(K) \mid 1 \le i \le n\}$$

The envelop size (number of terms above the main diagonal we will be concerned with for a skyline solver) is

$$|env(K)| = \sum_{i=1}^{n} \beta_i(K)$$

$$\begin{bmatrix} 1 & * & & * & & \\ & 2 & & & & \\ & & 3 & & * & * \\ & & & 4 & * & \\ & & & & 5 & * \\ & & & & & 6 \end{bmatrix}$$

i	M_i	β_i
1	1	0
2	1	1
3	3	0
4	1	3
5	3	2
6	3	3

* Is a non-zero entry

For this example $|env(K)| = 0+1+0+3+2+3 = 9$

Because solving the problem of optimal ordering in NP (nondeterministic in polynomial time) various heuristic ordering algorithms have been developed.

Reverse Cuthill-McKee Algorithm: The idea is straight forward. If x and y are adjacent nodes, where x is already labeled and y is not yet labeled, you want to label y to have a label index as close that of x as possible. The reversing of the labeling was added since it was found to slightly reduce the envelop (also referred to as the profile).

RCM

Classes

RCM

getStart( ) : node // get a starting node
addList(adj) // enqueues nodes into nodes in adj in order of degree
renumber( ) // renumbers the nodes

Queue q

Queue

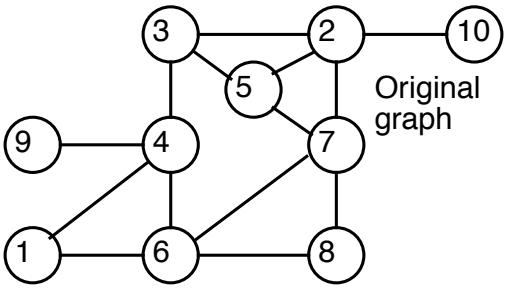
enqueue(item) // adds an item to the queue
dequeue( ) : item // removes an item from the queue
size ( ) : int // returns the current size of the queue

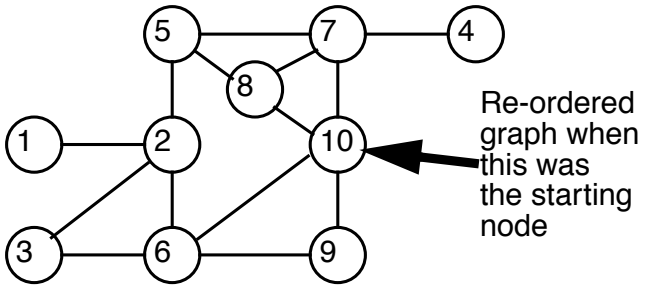
Node

SetLabel(label) // sets node's label to label
adjacentNodes(node) : adj // returns list of unlabeled adjacent nodes that are also not in the
// queue

Pseudo Code

```
renumber
{
int label = n+1 // set the value of the label to the total number of nodes plus 1, this allows the
                // automatic accounting for the reversing process
Node node = getStart( ) // get the starting node that will be labeled first
q • enqueue(node)
while (q size( ) > 0) //process nodes until the queue is empty
    {
    label = label -1  // want the reverse order
    node = q • dequeue( )
    node • setLabel(label)  // sets the label of the object node
    adj = node • adjacentNodes(node) // get unnumbered adjacent nodes not already in queue
    addList(adj) // add the nodes in adj to the queue in order of degree
    }
```





In the example above n=10. Lets start the process by having getStart return node 7 as the starting node. Each of the rows in the table below indicates the status of things for that pass through the while loop. Note that the new label assigned to a node reflects the reversing of the order as we go. As we enter the while loop node 7 is the only one in the queue. It is removed from the queue and the nodes adjacent to it are put in the queue. Note that the adjacencies are ordered by increasing degree. For example deg(8)=2, deg(5)=3, deg(2)=4, deg(6)=4. Note that on subsequent steps the first node in the queue is removed and only those in the adjacency list that are neither already relabeled, or already in the list are added.

original node label	**new node Label**	**nodes adjacent to node**	**nodes in queue**
7	10	8,5,2,6	8,5,2,6
8	9	7,6	5,2,6
5	8	3,2,7	2,6,3
2	7	10,3,5,7	6,3,10
6	6	8,1,4,7	3,10,1,4
3	5	5,2,4	10,1,4
10	4	2	1,4
1	3	4,6	4
4	2	9,1,3,6	9
9	1	4	-

An alternative implementation is to put all the adjacent nodes in the queue. However, instead of popping just one node from the queue each time through the while loop, you pop already relabeled nodes until you get to one that has not been relabeled, you will relabel that one.

The original matrix

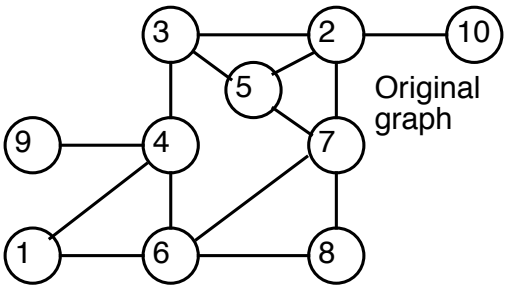
										node - i	**M(i)**	**β (i)**
1			x		x					1	1	0
	2	x		x		x			x	2	2	0
		3	x	x						3	2	1
			4		x			x		4	1	1
				5		x				5	2	3
					6	x	x			6	1	5
						7	x			7	2	5
							8			8	6	2
								9		9	4	5
									10	10	2	8

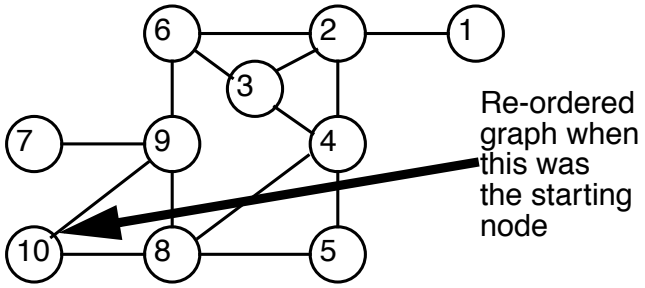
$\beta = 8$

$|Env(k)| = 32$

The reordered matrix										**node - i**	**M(i)**	**β (i)**	
1	x									1	1	0	
	2	x		x	x					2	1	1	
		3			x					3	2	1	β =4
			4			x				4	4	0	|Env(k)|=22
				5		x	x			5	2	3	
					6			x	x	6	2	4	
						7	x		x	7	4	3	
							8		x	8	5	3	
								9	x	9	6	3	
									10	10	6	4	

Note that the selection of the starting node makes a difference. If we started with node 1 as the starting node we will get a different result as given below.





The reordered matrix now is										**node - i**	**M(i)**	**β (i)**	
1	x									1	1	0	
	2	x	x		x					2	1	1	
		3	x		x					3	2	1	β =4
			4	x			x			4	2	2	|Env(k)|=18
				5			x			5	4	1	
					6			x		6	2	4	
						7		x		7	7	0	
							8	x	x	8	4	4	
								9	x	9	6	3	
									10	10	8	2	

How do we get a good starting node? It has been found that given a connected graph, you want to find the two nodes that are “furthest apart”, selecting one of them is a good starting node to label. What do we mean by furthest apart?

A path between of length k between two nodes x_0 and x_k in a connected graph is $(x_0, x_1, x_2, ...x_{k-1})$ where $x_i \in adj(x_{i+1}), 0 \le i \le k-1.$

The distance, $d(x,y)$, between x and y is the minimum length path between x and y.

The eccentricity of node $x \in X$ is defined as

$$l(x) = \max\{d(x,y) \mid y \in X\}$$

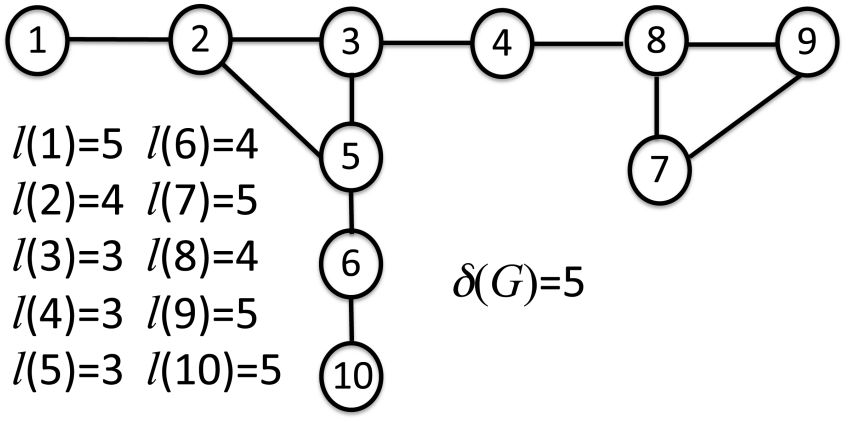
The diameter of a graph, G, is

$$\delta(G) = \max\{l(x) \mid x \in X\}$$

or equivalently

$$\delta(G) = \max\{d(x,y) \mid x \in X, y \in X\}$$

node $x \in X$ is said to be a peripheral node if $l(x) = \delta(G)$.



Again it will be too expensive to do the work needed to find the nodes associated with the eccentricity ($O(n^2)$), or even its value. Will settle for one that gets close.

One such algorithm builds on a rooted level structure. The rooted level structure for node $x \in X$ is defined as

$$\xi(x_i) = \{L_0(x_i), L_1(x_i), L_2(x_i), \ldots L_{l(x_i)}(x_i)\}$$

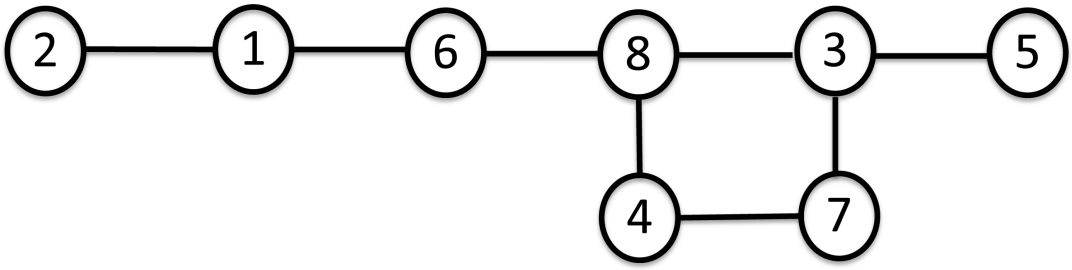
where

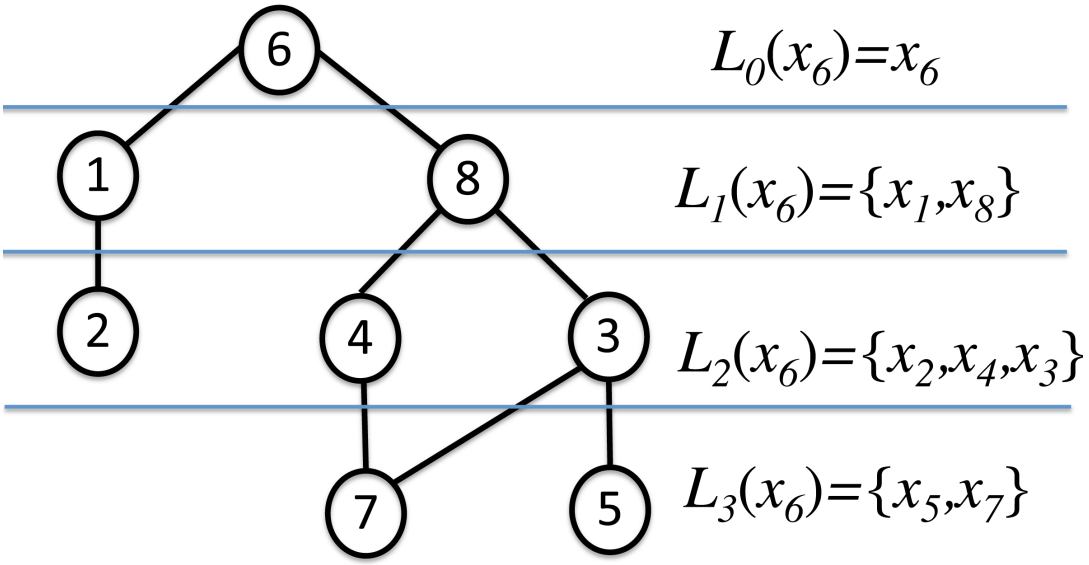
$$L_0(x_i) = \{x_i\} \text{ the node itself}$$

$$L_1(x_i) = Adj(L_0) = Adj(x_i) \text{ the set of nodes adjacent to } x_i$$

$$L_j(x_i) = Adj(L_{j-1}) - Adj(L_{j-2}), i = 2,3 \ldots l(x_i)$$

$L_j(x_i)$ picks up the ones adjacent to the ones in the previous level that were not already accounted for in a previous one.





Construction of $\xi(x_6)$ for the given graph.

begin {FindStart}

$r_0 \in X$ {select any starting node you like}

$\zeta(r_0) = \{L_0(r_0), L_1(r_0), L_2(r_0), \ldots L_{l(r_0)}(r_0)\}$ {construct level structure and determine $l(r_0)$}

$r_1 \in L_{l(r_0)}$ {select node in last level with min degree}

$\zeta(r_1) = \{L_0(r_1), L_1(r_1), L_2(r_1), \ldots L_{l(r_1)}(r_1)\}$ {construct level structure and determine $l(r_1)$}

$i = 1$

While $l(r_i) > l(r_{i-1})$ {do as long as eccentricity is increasing}

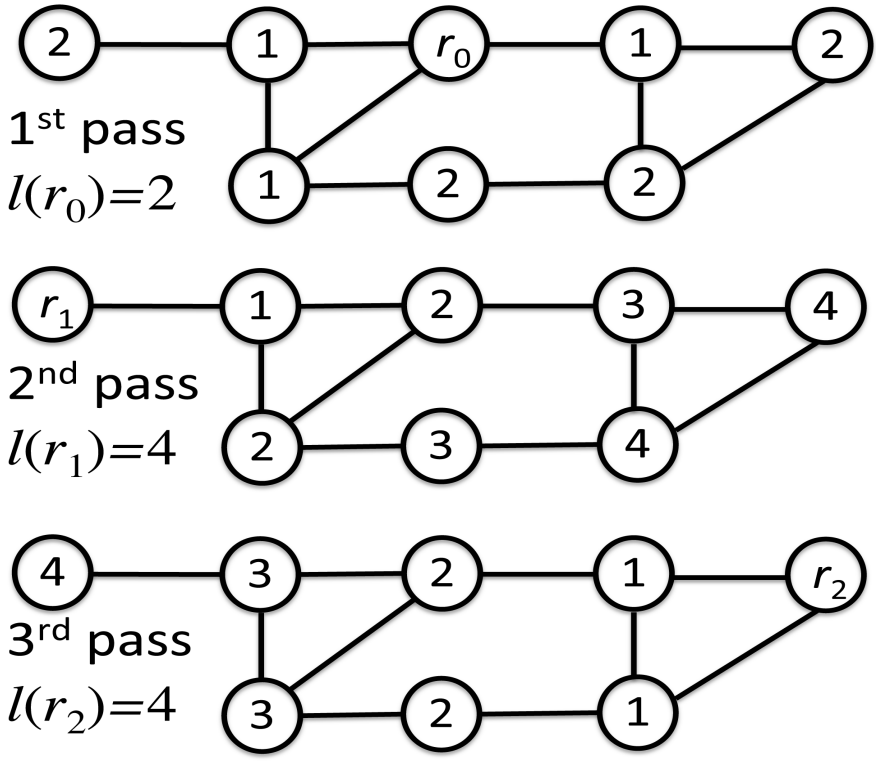
$i = i + 1$

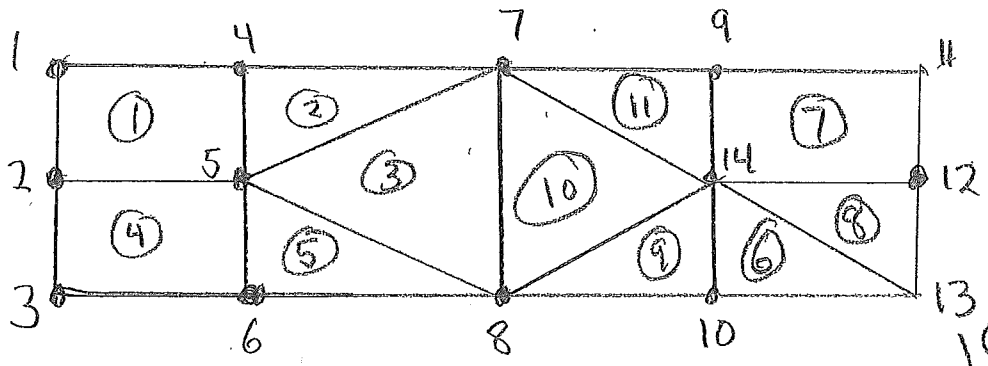
$\zeta(r_i) = \{L_0(r_i), L_1(r_i), L_2(r_i), \ldots L_{l(r_i)}(r_i)\}$

end {while}

$x_1 = r_i$ {starting node is found}

end {FindStart)





Construct Graph from Standard F.E. input

Connectivity Input

①	1, 2, 5, 4
②	4, 5, 7
③	5, 8, 7
④	2, 3, 6, 5
⑤	6, 8, 5
⑥	10, 13, 14
⑦	14, 12, 11, 9
⑧	14, 13, 12
⑨	8, 10, 14
⑩	7, 8, 14
⑪	7, 14, 9

Loop over elements adding "new" nodes to the Node - Node table

Node	Connected nodes from elements
1	① 2, 5, 4 ← Defines 3 edges {1,2}, {1,5}, {1,4}
2	① 1, 5, 4, ④ 3, 6 ← Defines 5 edges
3	④ 2, 6, 5 etc.
4	① 1, 2, 5, ② 7
5	① 1, 2, 4, ② 7, ③ 8, ④ 3, ⑤ -
6	④ 2, 3, 5, ⑤ 8
7	② 4, 5, ③ 8, ⑩ 14, ⑪ 9
8	③ 6, 7, ⑤ 6, ⑨ 10, 14, ⑩ 7
etc.	

For Frontal Solver want Element - Element

Fastest way - 2 Steps

(A) Construct List of elements for each node (Node - Element)

(B) Traverse Node - Element like above to build Element - Element.

Since we are working with methods where we have a mesh topology that contains lots of adjacency information, is there a way to directly execute reordering based on that information? Of course, the answer will be yes or I would not ask the question.

Continuing to look into minimizing bandwidth or profile, lets consider the fact that the degrees of freedom (dof) are actually associated with mesh topological entities. That is all mesh entities can be “dof holders”.

Consider C^0 finite elements based on Lagrange shape functions. Then the dof are associated with the finite element nodes. For linear elements nodes are associated with only mesh vertices. For quadratic serendipity elements we add edge dof. For tensor product elements greater than linear we pick-up dof on edges, faces and regions. The same for C^0 p-version elements. In this case of C^{-1} (DG or finite volume methods) all the dof are associated with the “element” (regions in 3D and faces in 2D) – even if they are “located” at coordinates on the element boundary. This is because the values of the dof are different for each element. However, all DG and finite volume methods have operations to relate the dof across boundaries. Thus, the specifics of these operations define the graph edges. With isogeometric you need to couple across multiple elements.

Lets look at the basics of RCM working from mesh adjacency information. Lets considering labeling the finite element nodes in C^0 Lagrangian finite elements, The finite element nodes are the graph nodes. Given a next node in the queue, we want to label it and add to the queue all

nodes adjacent to it that are not already labeled, or in the queue, preferably with the ones with lowest degree going in first. Consider a mesh where I start with the node at a well selected vertex, the nodes that need to be added to the queue are nodes adjacent to it based on the mesh adjacencies. We have some flexibility is determining which adjacencies we account for and how. Remember the RCM algorithm is based on a simple heuristic where we have flexibility, within the limit of being sure to have edges connecting what need to be connected, in how we want to define the details of what we define as adjacent at a particular point in the process and how we want order adding a set of adjacent nodes to the queue.

Initializing the queue: Doing the level structures is a pain and expensive. Want something simple. One option that is been found to work is to look the mesh vertices classified on model vertices, $\{M_i^0\}[\{G_j^0\}$, select one that has the minimum number of edges coming into it, or one that is closest to a corner of the bounding box of the domain. Note that if one has a reason for trying to get a pattern in the ordering, one could actually queue a set of node. For example flow over an airfoil, you may want to have nodes associated with the fine mesh ordered first. Assuming the mesh is fine near the airfoil, you could initialize the queue with all the nodes on the airfoil surface. This will give me a numbering that spirals going out from the airfoil – form the fine mesh the coarse mesh.

The following is the logic of the adjacency reordering algorithm we found to works well.

AdjReorder

Classes (not all items indicated)

AdjReorder

getStart( ) : entity // get starting mesh vertex
renumber(mesh) // renumbers the nodes and elements

Queue q Mesh mesh

List

add(entity) // adds an entity to a list
emptyList( ) // empties a list

Queue

enqueue(item) // enqueues an item into the queue
enqueueList(List) // enqueues list into the queue
dequeue( ) : item // removes an item from the queue
size( ) : int // returns the number of entities in the queue

MeshEntity

dimension( ) : int // indicates the dimension of a mesh entity 0-vertex, 1-edge, 2-face, 3-region
numEdges( ) : int // indicates the number of edges bounding or coming into an entity
numFaces( ) : int // indicates the number of faces bounding or coming into an entity
edge(i) : MeshEdge // returns the i th edge bounding or coming into an entity
face(j) : MeshFace // returns the j th face bounding or coming into an entity
getNode( ) : Node // gets the node on the mesh entity

MeshEdge
otherVertex(vertex) // gets the other vertex for a given edge

Node
setLabel(label) // sets node label to label

Pseudo Code

renumber(mesh)

{ // Reorders nodes and elements in a 2-D mesh assuming that only the mesh faces are elements

// It is also assumed that all dof associated with an entity are associated with a single node on that entity

int labelnode = nnode + 1 // value set to the total number of nodes plus 1, this allows auto. reversing

int labelface = nface + 1 // same issues in labeling elements

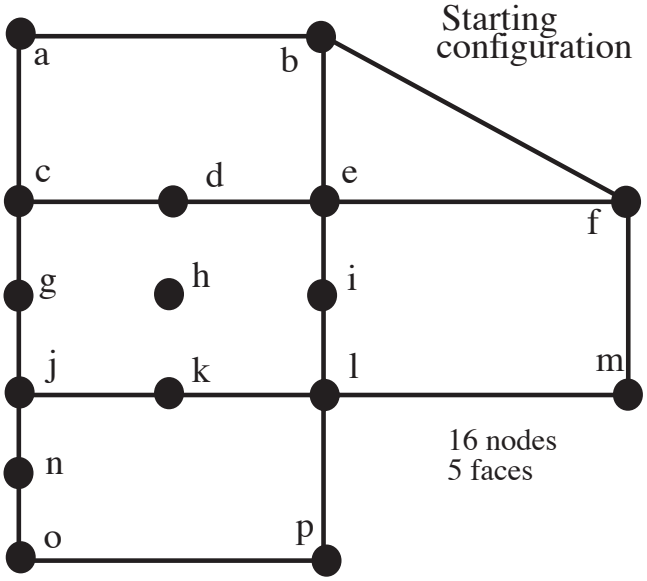
Node node

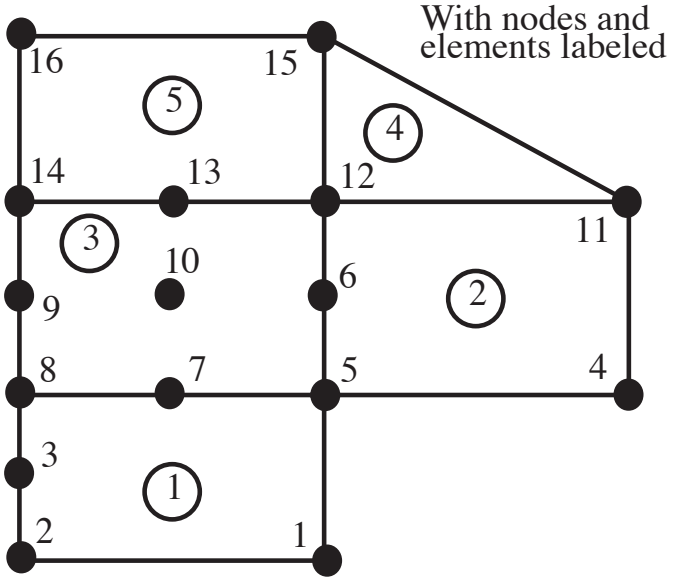
MeshEntity entity = getStart( ) // get starting entity. Use a $M_i^0 \sqsubset G_j^0$ with min. number of G_k^1 using it.

```
q → enqueue(entity)
while (q → size( ) > 0) { // process entities until the queue is empty
    entity = q → dequeue( )
    node = entity → getNode( )
    if (/*node is unlabeled*/) then {
        labelnode = labelnode -1
        node → setLabel(labelnode)
    }
    // Want to find any unnumbered adjacent mesh entities and label faces
    // All the additions to the queue will be done by looking at adjacencies keying from vertices
    if (entity → dimension( )=0) then { // need to load adjacent entities by adjaceny order
        // Also label neighboring mesh faces and specific edge nodes
        MeshVertex vertex = entity
        for (i = 1 to vertex → numEdges( ) ) { // loop over number of edges using the vertex
            MeshEdge edge = vertex → edge(i)
            for (j = 1 to edge → numFaces( )) do {
                MeshFace face = edge → face(j)
                if (/* face not already labeled*/) then {
                    labelface = labelface -1
                    face → setLabel(labelface)
                }
                if (/* face has node that needs queueing */) then { // queue the face
                q → enqueue(face)
                }
            }
            othervertex = edge → otherVertex(vertex)
            if (node = edge → getNode( )) then { // if the edge has a node on it
                if (/* othervertex labeled or in queue and edge node not labeled*/ ) then {
                    labelnode = labelnode -1
                    edge → getNode( ) → setLabel(labelnode)
                } else {
                    q → enqueue(edge)
                    list → add(othervertex)
                }
            } else {
                if (/* node at other vertex is not labeled) then {
                    list → add(othervertex)
                }
            }
        } // finished the loop over the edges coming into the current vertex
        q → enqueueList(list) // now queue the other vertices loaded into the list
        emptyList( )
    }
}
```

The current pseudo code sets the labeling of the nodes and elements. This is what is needed for assignment 2. In assignment 4 you will not really care if you order the labels on the mesh entities, what you really care about is ordering the dof in the global system in the best way. This can be done directly using essentially the same procedure. That is you work your way through the mesh entities in exactly the same manner. However, in the code segments above where the nodes are labeled is replaced by code labeling the dof associated with those mesh entities (again starting with the last equation number and working down to 1).

Example of the procedures on setting mesh entity labels





The status of the queue and the nodes labeled on each pass through

Queue	**At**	**Node label assigned**
a	-	start of process - none assigned yet
b,c	a	16
c,e,f	b	15
e,f,h,g,j	c	14,13 (d done since e already in queue)
f,h,g,j,i,l	e	12
h,g,j,i,l,m	f	11
g,j,i,l,m	h	10
j,i,l,m	g	9
i,l,m,n,o	j	8,7 (k done since l already in queue)
l,m,n,o	i	6
m,n,o,p	l	5
n,o,p	m	4
o,p	n	3
p	o	2
-	p	1

Mesh Generation - Reordering

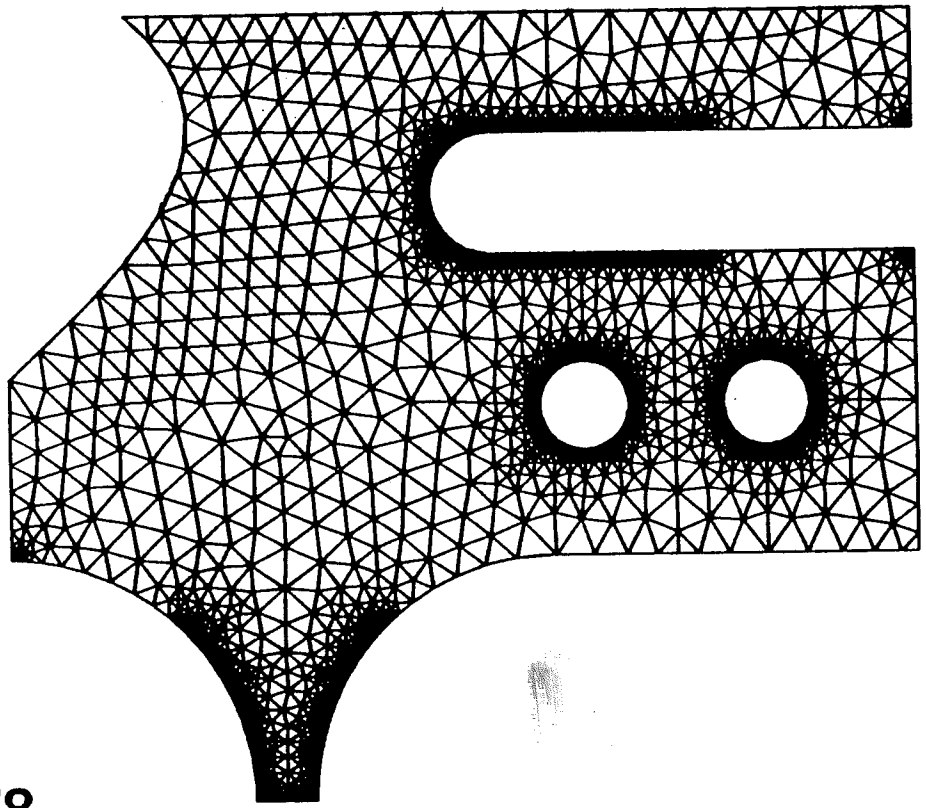


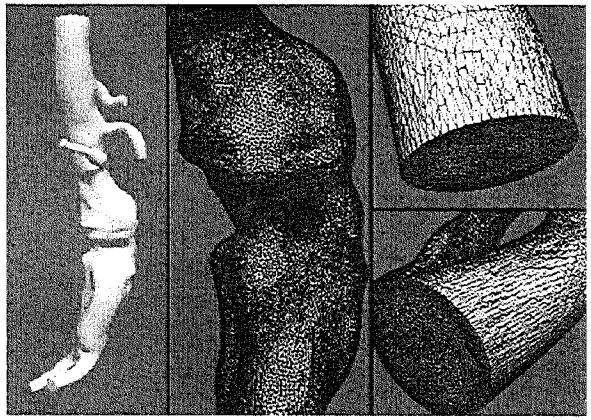
Figure 78

Mesh		BANDWIDTH				PROFILE			
nodes	elements	initial	NDQ	RCM	GPS	initial	NDQ	RCM	GPS
4330	7619	3860	188	262	213	4,096,362	393,989	495,865	398,162

IMPROVING UNSTRUCTURED MESH CACHE PERFORMANCE

Algorithm for reordering using a mesh adjacency based traversal developed

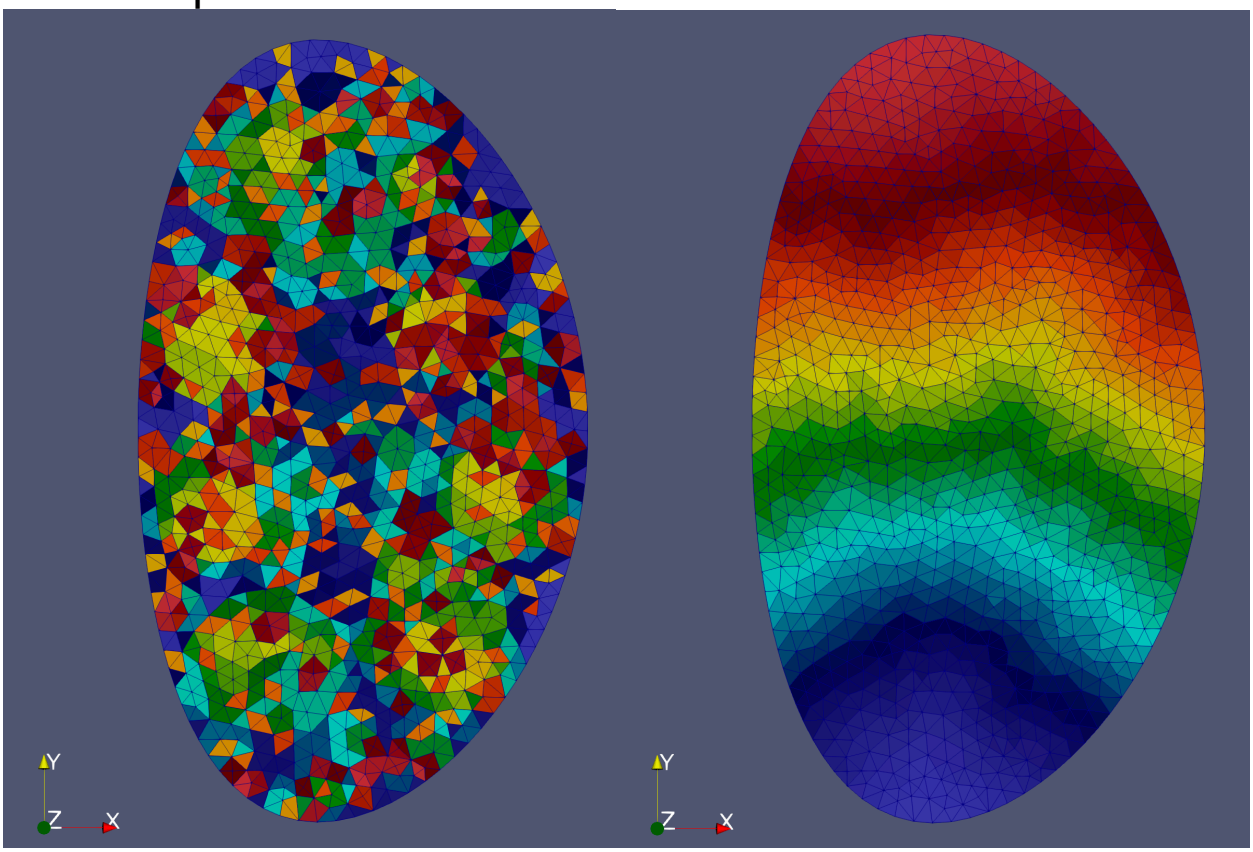
- A complete mesh adjacency structures effectively supports the reordering of data associated with any mesh entity type
- Single traversal executed on the part level quickly reorders both nodes (dof holders) and elements
- Substantial on core performance improvements
  - Up to 40% on Cray XT5
  - Up to 24% in Blue Gene/L
- Total simulation time improvements as high as 20%



Data reordering effect on different phases of implicit FEA

Time in seconds (in %)	Without data reordering	Reordering of vertices only	Reordering of vertices and elements
Localization and globalization	61.5 (1.1%)	67.2 (1.2%)	27.9 (0.5%)
Sparse-matrix filling	163.8 (3.0%)	162.8 (3.1%)	132.2 (2.4%)
Element level integral	997.7 (18.4%)	998.6 (18.4%)	994.3 (18.3%)
Total of system formation	1520.4 (28.0%)	1524.1 (28.1%)	1429.9 (26.3%)
System solution	3894.9 (71.8%)	3005.7 (55.4%)	2992.8 (55.2%)
Total of analysis	5422.8 (100%)	4537.3 (83.6%)	4430.3 (81.7%)

An example result:



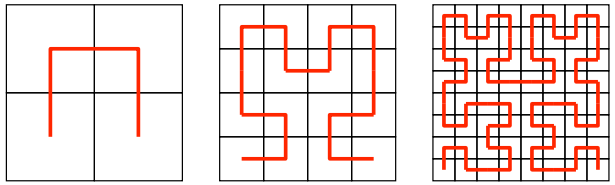
Note – Another popular ordering method, particularly for memory control, is space filling curves. As indicated in the discussion on dynamic load balancing, they are also of interest for that need.

In work we have done, we have used graph-based algorithm at the inter-process level (MPI level) and adjacency based reordering for better memory access on process.

Space-Filling Curve Partitioning (SFC)

Sandia National Laboratories

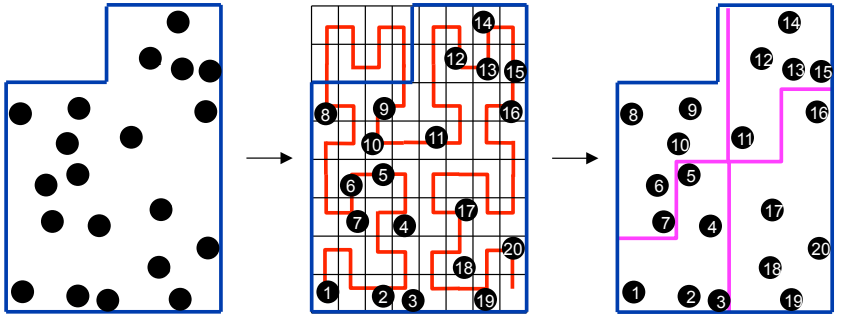
- Developed by Peano, 1890.
- Space-Filling Curve:
  - Mapping between R^3 to R^1 that completely fills a domain.
  - Applied recursively to obtain desired granularity.
- Used for partitioning by …
  - Warren and Salmon, 1993, gravitational simulations.
  - Pilkington and Baden, 1994, smoothed particle hydrodynamics.
  - Patra and Oden, 1995, adaptive mesh refinement.



SFC Algorithm

Sandia National Laboratories

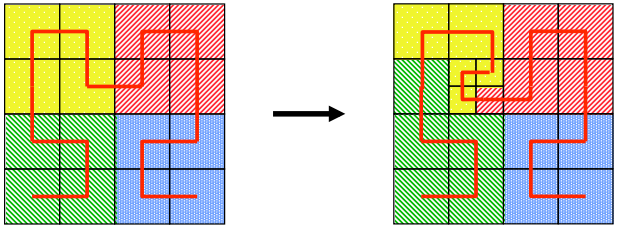
- Run space-filling curve through domain.
- Order objects according to position on curve.
- Perform 1-D partition of curve.



SFC Repartitioning

Sandia National Laboratories

- Implicitly incremental.
- Small changes in data results in small movement of cuts in linear ordering.



SFC Advantages and Disadvantages

Sandia National Laboratories

- Advantages:
  - Simple, fast, inexpensive.
  - Maintains geometric locality of objects in processors.
  - Linear ordering of objects may improve cache performance.
- Disadvantages:
  - No explicit control of communication costs.
  - Can generate disconnected subdomains.
  - Often lower quality partitions than RCB.
  - Geometric coordinates needed.