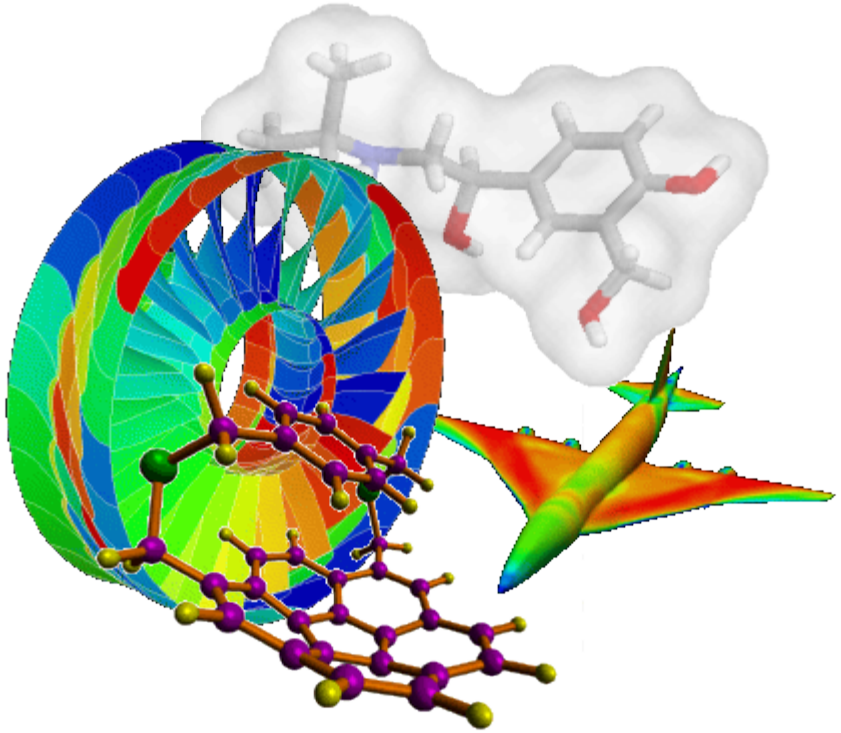
CME342 - Parallel Methods in Numerical Analysis

Graph Partitioning Algorithms



April 21-23, 2014

Lectures 7-8

Announcements

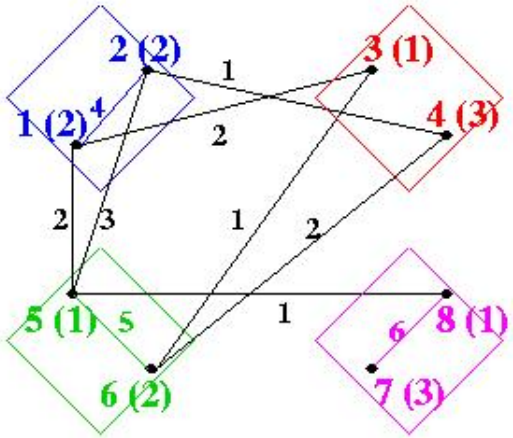
- Homework 2 on web page later today
- With all remaining homeworks, you will be asked to provide a short (< 1 page) writeup regarding your approach to the solution of the problems
- Comments on source code will account for 5% of the grade.
- Graph partitioning (2 lectures)

Outline

- Definition of graph partitioning problem
- Sample applications
- N-P complete problem
- Available heuristic algorithms (with and without nodal coordinates)
  - Inertial partitioning
  - Breadth first search
  - Kernighan-Lin
  - Spectral bisection
- Multilevel acceleration (multigrid for graph partitioning problems)
- Metis, ParMetis, and others

Definition of Graph Partitioning

- Given a graph G = (N, E, W_N, W_E)
  - N = nodes (or vertices), E = edges
  - W_N = node weights, W_E = edge weights
- N can be thought of as tasks, W_N are the task costs, edge (j,k) in E means task j sends W_E(j,k) words to task k
- Choose a partition N = N_1 U N_2 U … U N_P such that
  - The sum of the node weights in each N_j is distributed evenly (load balance)
  - The sum of all edge weights of edges connecting all different partitions is minimized (decrease parallel overhead)
- In other words, divide work evenly and minimize communication
- Partition into two parts is called graph bisection, which recursively applied can be turned into algorithms for complete graph partitioning

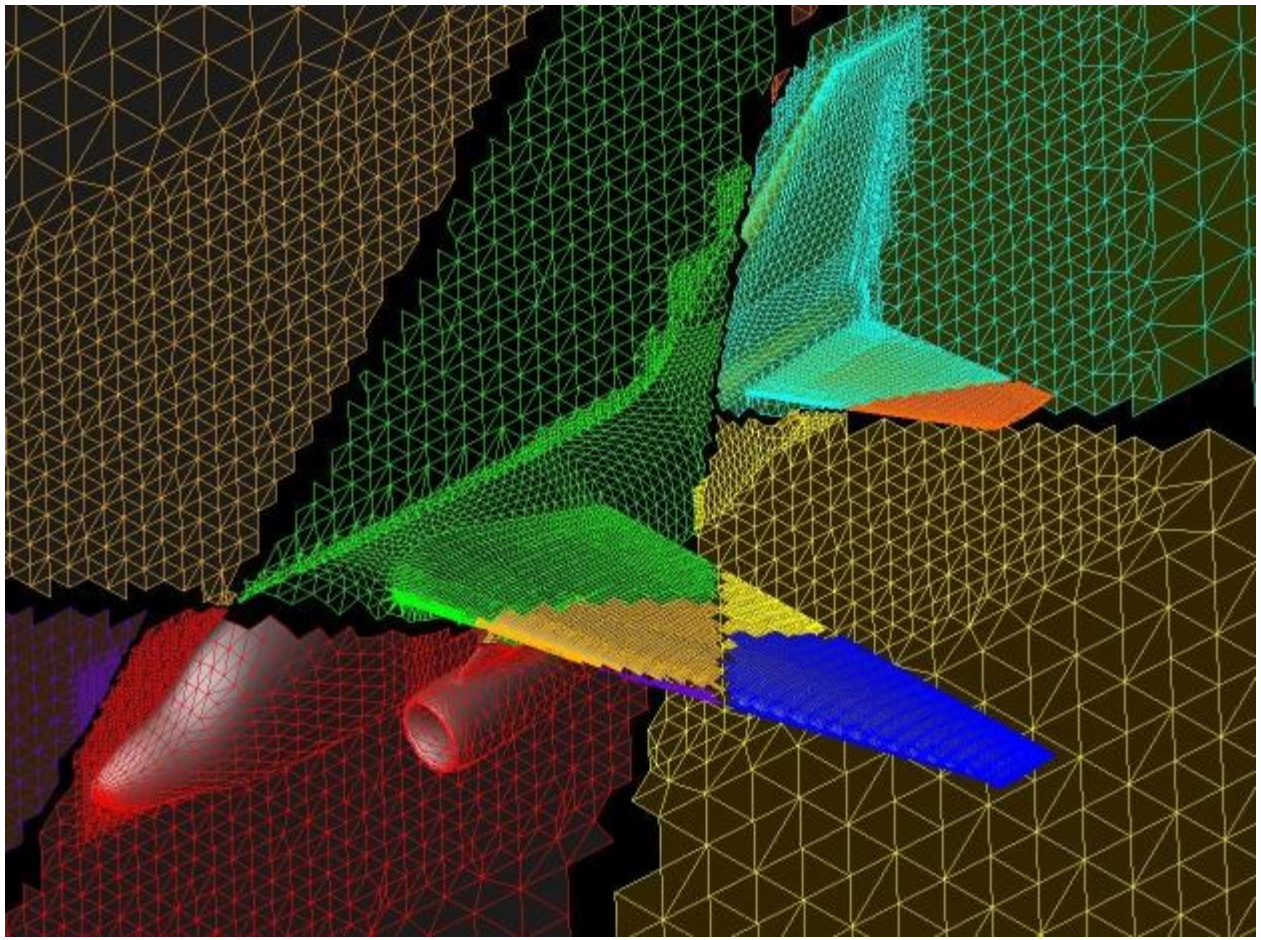


Applications

- Load balancing while minimizing communication
- Structured and unstructured mesh distribution for distributed memory parallel computing (FEM, CFD, RCS, etc.)
- Sparse matrix times vector multiplication
  - Solving PDEs (above)
  - N = {1,…,n}, (j,k) in E if A(j,k) nonzero,
  - $W_N(j)$ = #nonzeros in row j, $W_E(j,k) = 1$
- VLSI Layout
  - N = {units on chip}, E = {wires}, $W_E(j,k)$ = wire length
- Telephone network design
  - Original application, algorithm due to Kernighan
- Sparse Gaussian Elimination
  - Used to reorder rows and columns to increase parallelism, decrease "fill-in"

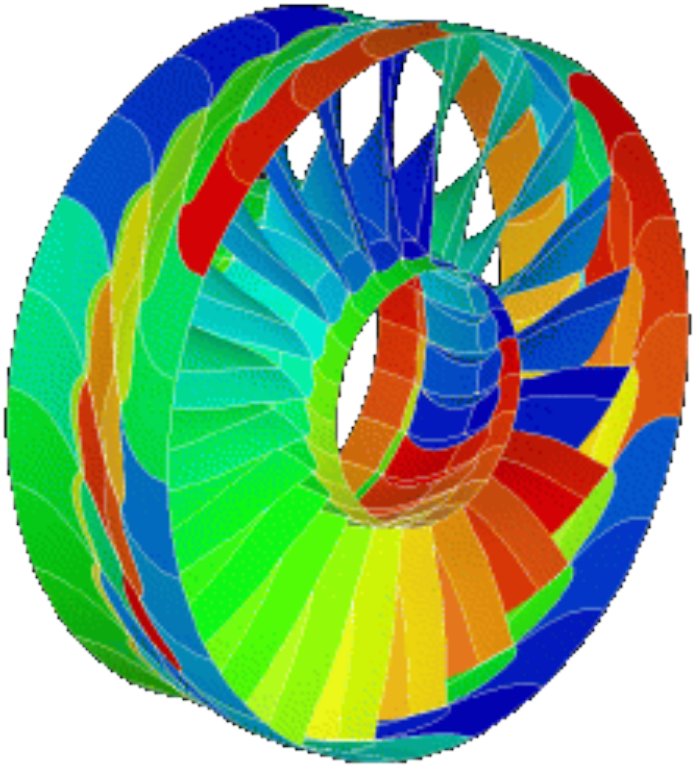
Applications-Unstructured CFD

Partitioning of an undirected nodal graph for parallel computation of the flow over an S3A aircraft using 16 processors of an IBM SP2 system (1995). Colors denote the partition number. Edge separators not shown. Solution via AIRPLANE code.



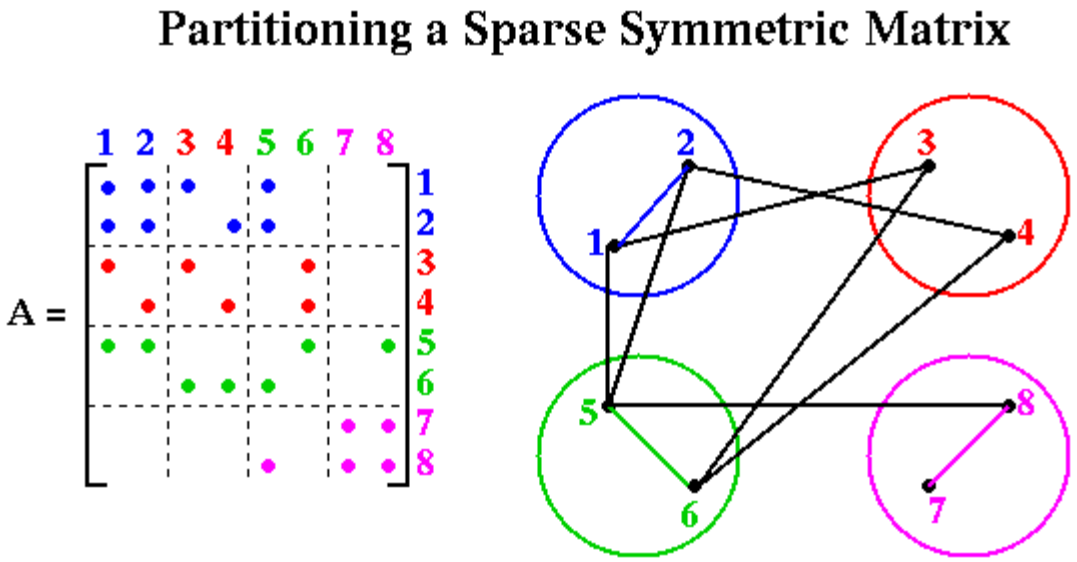
Applications- SUmb Load Balancing

The static load balancing procedure for the multiblock-structured flow solver, SUmb, developed for the ASC project at SU, uses a graph partitioning algorithm where the original graph has nodes corresponding to mesh blocks with weights equal to the total number of cells in the block, and where the edges represent the communication patterns in the mesh; the edge weights are proportional to the surface area of the face that is being communicated



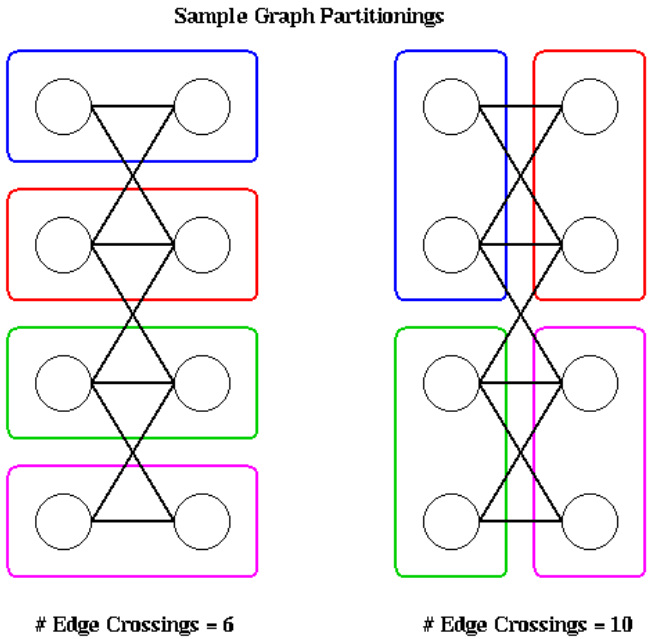
Now, that is where that silly picture comes from!!!!!

Sparse Matrix Vector Multiplication



Cost of Graph Partitioning

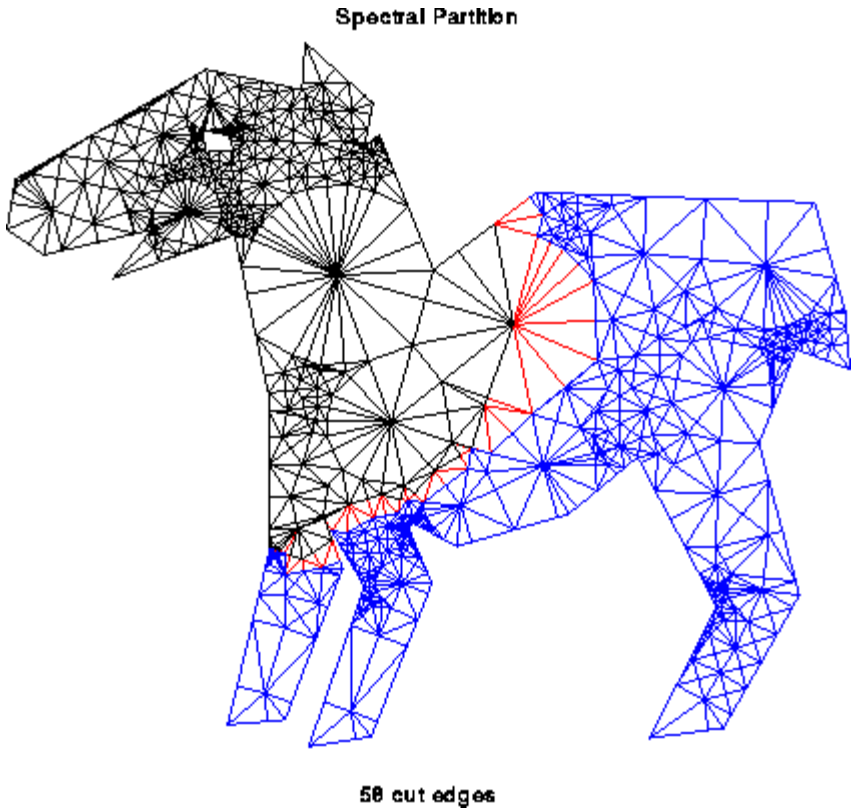
- Many possible partitionings to search:



- n choose n/2 ~ sqrt(2n/pi)*2^n bisection possibilities
- Choosing optimal partitioning is NP-complete
  - Only known exact algorithms have cost that is exponential in the number of nodes in the graph, n
- We need good heuristics-based algorithms!!

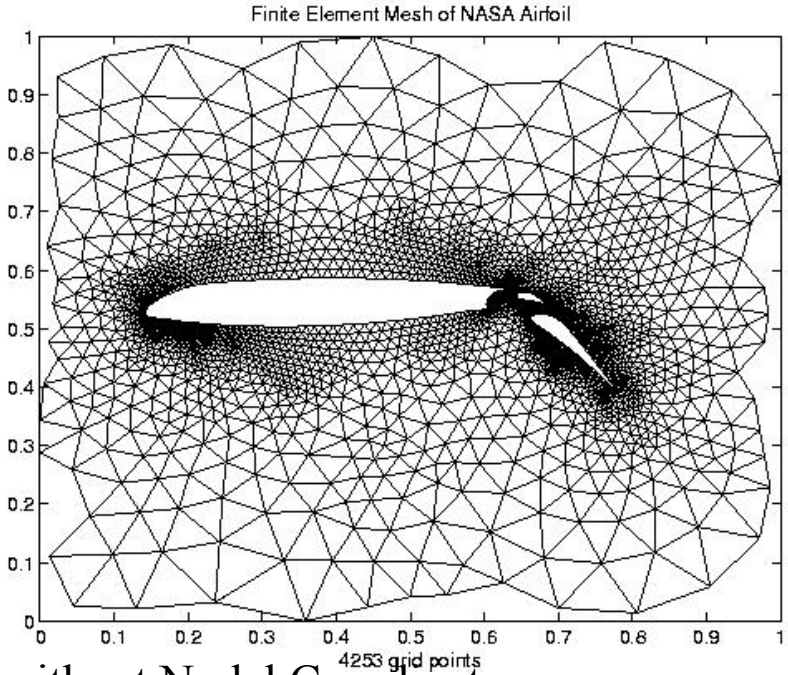
First Heuristic: Repeated Graph Bisection

- To partition N into 2k parts, bisect graph recursively k times
  - Henceforth discuss mostly graph bisection



Overview of Partitioning Heuristics for Bisection

- Partitioning with Nodal Coordinates
  - Each node has x,y,z coordinates
  - Partition nodes by partitioning space



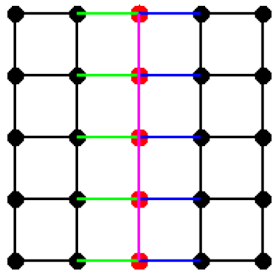
- Partitioning without Nodal Coordinates
  - Sparse matrix of Web: A(j,k) = # times keyword j appears in URL k
- Multilevel acceleration
  - Approximate problem by "coarse graph", do so recursively

Edge Separators vs. Vertex Separators of G(N,E)

- Edge Separator: E_s (subset of E) separates G if removing E_s from E leaves two ~equal-sized, disconnected components of N: N_1 and N_2
- Vertex Separator: N_s (subset of N) separates G if removing N_s and all incident edges leaves two ~equal-sized, disconnected components of N: N_1 and N_2
- Edge cut: Sum of the weights of all edges that form an edge separator

Edge Separators and Vertex Separators

E_S = green edges or blue edges
N_S = red vertices



- Making an N_s from an E_s: pick one endpoint of each edge in E_s
  - How big can $|N_s|$ be, compared to $|E_s|$?
- Making an E_s from an N_s: pick all edges incident on N_s
  - How big can $|E_s|$ be, compared to $|N_s|$?
- We will find Edge or Vertex Separators, as convenient

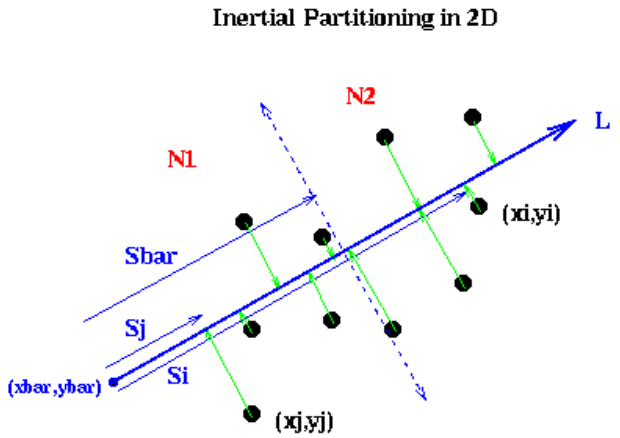
Graphs with Nodal Coordinates - Planar graphs

- Planar graph can be drawn in plane without edge crossings
- Ex: m x m grid of m^2 nodes: ∃ vertex separator N_s with $|N_s| = m = \text{sqrt}(|N|)$ (see last slide for m=5)
- *Theorem* (Tarjan, Lipton, 1979): If G is planar, ∃ N_s such that
  - $N = N_1 \cup N_s \cup N_2$ is a partition,
  - $|N_1| <= 2/3\ |N|$ and $|N_2| <= 2/3\ |N|$
  - $|N_s| <= \text{sqrt}(8 * |N|)$
- Theorem motivates intuition of following algorithms

Graphs with Nodal Coordinates: Inertial Partitioning

- For a graph in 2D, choose line with half the nodes on one side and half on the other
  - In 3D, choose a plane, but consider 2D for simplicity
- Choose a line L, and then choose an $L^{\perp}$ perpendicular to it, with half the nodes on either side

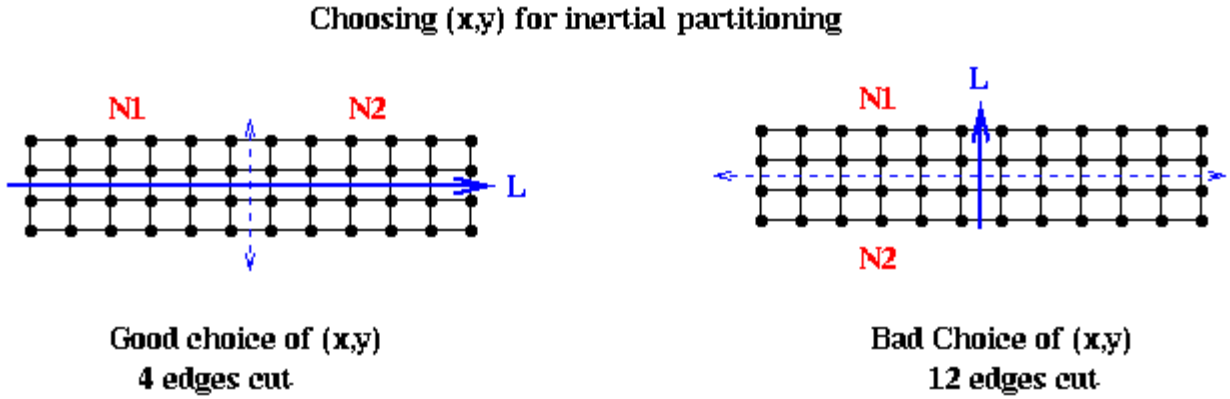
1) **L given by a*(x-xbar)+b*(y-ybar)=0, with $a^2+b^2=1$; (a,b) is unit vector $\perp$ to L**
2) **For each nj = (xj,yj), compute coordinate $S_j = -b*(x_j-xbar) + a*(y_j-ybar)$ along L**
3) **Let Sbar = median($S_1,\ldots,S_n$)**
4) **Let nodes with S_j < Sbar be in N_1, rest in N_2**



- Remains to choose L

Inertial Partitioning: Choosing L

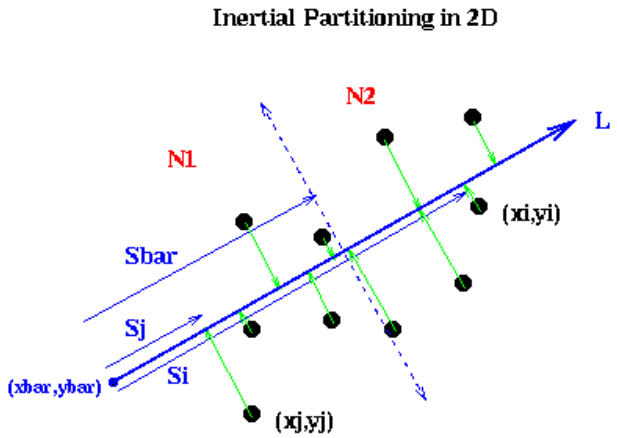
- Clearly prefer L on left below



- Mathematically, choose L to be a total least squares fit of the nodes
  - Minimize sum of squares of distances to L (green lines on last slide)
  - Equivalent to choosing L as axis of rotation that minimizes the moment of inertia of nodes (unit weights) - source of name

Inertial Partitioning: choosing L

(a,b) is unit vector perpendicular to L



Σ_j (length of j-th green line)^2

$= \Sigma_j \; [(x_j - xbar)^2 + (y_j - ybar)^2 - (-b*(x_j - xhar) + a*(y_j - ybar))^2]$

… Pythagorean Theorem

$= a^2 * \Sigma_j (x_j - xbar)^2 \; + \; 2*a*b* \Sigma_j (x_j - xbar)*(y_j - ybar) \; + \; b^2 \Sigma_j (y_j - ybar)^2$

$= a^2 * X1 \; + \; 2*a*b* \; X2 \; + \; b^2 * X3$

$$= [a\ b] * \begin{bmatrix} X1 & X2 \\ X2 & X3 \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix}$$

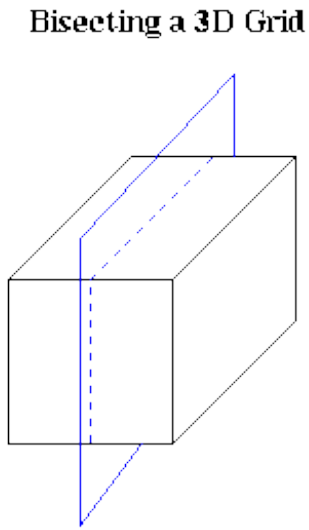
Minimized by choosing

(xbar , ybar) = $(\Sigma_j x_j , \Sigma_j y_j) / N$ = center of mass

(a,b) = eigenvector of smallest eigenvalue of $\begin{bmatrix} X1 & X2 \\ X2 & X3 \end{bmatrix}$

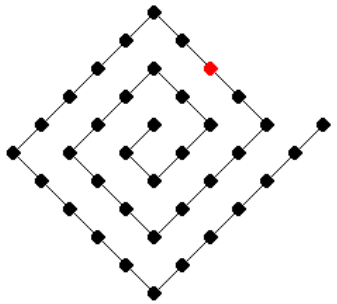
Inertial Partitioning: Three Dimensions

- In 3D, the situation is almost identical only that the line separating the partitions is now a plane, and the vectors and points have three components.
- The matrix problem is simply 3x3, but conclusions are the same:
  - Choose plane that contains the center of mass of the graph, and
  - Has normal vector given by the eigenvector of the 3x3 eigenvalue problem
- Repeat recursively



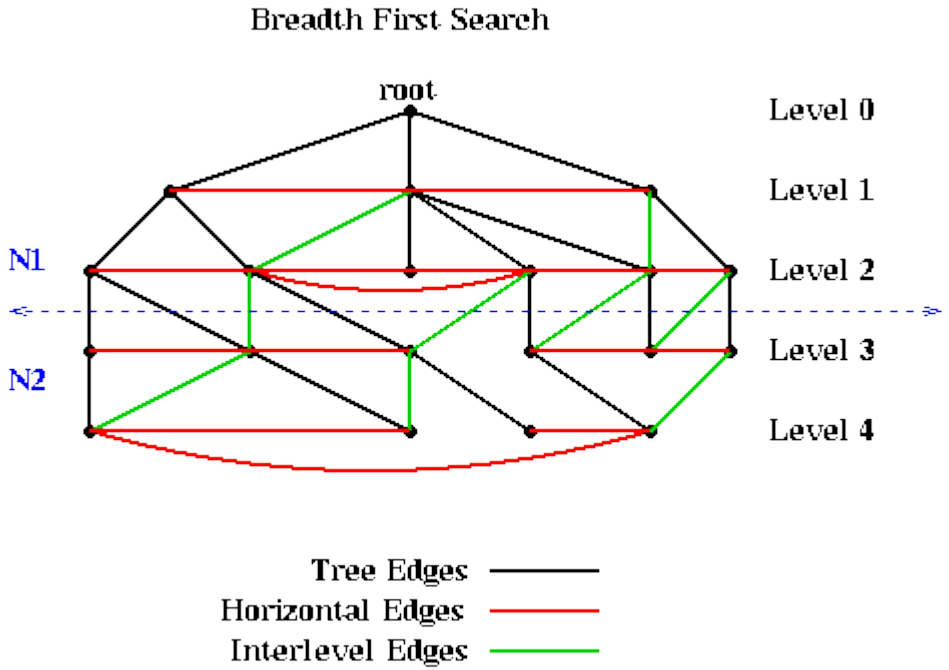
Partitioning with Nodal Coordinates - Summary

- Other algorithms and variations are available (random spheres, etc.)
- Algorithms are efficient
- Rely on graphs having nodes connected (mostly) to "nearest neighbors" in space
  - algorithm does not depend on where actual edges are!
- Common when graph arises from physical model
- Can be used as good starting guess for subsequent partitioners, which do examine edges
- Can do poorly if graph less connected:



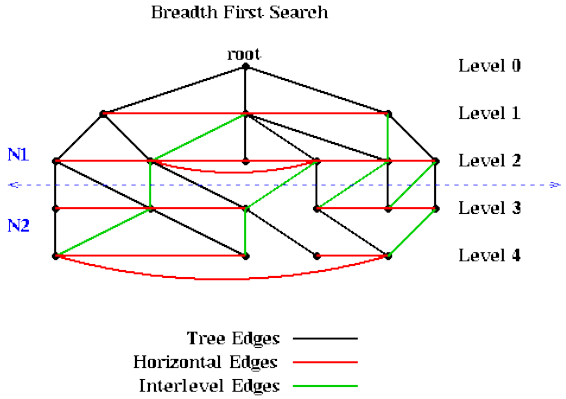
Partitioning without Nodal Coordinates- Breadth First Search (BFS)

- Given G(N,E) and a root node r in N, BFS produces
  - A subgraph T of G (same nodes, subset of edges)
  - T is a tree rooted at r
  - Each node assigned a level = distance from r



Breadth First Search

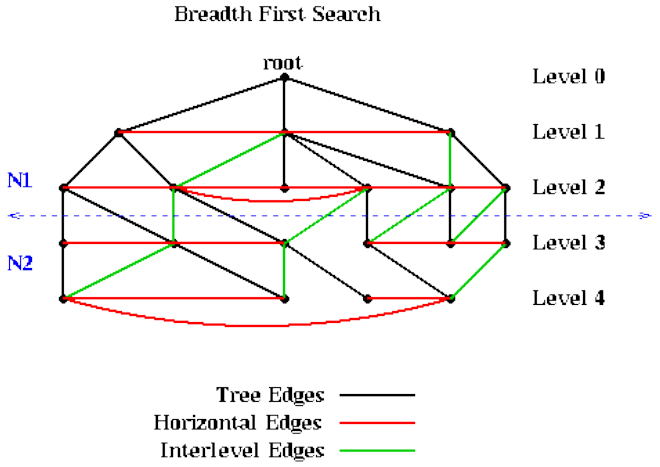
- Queue (First In First Out, or FIFO)
  - Enqueue(x,Q) adds x to back of Q
  - x = Dequeue(Q) removes x from front of Q
- Compute Tree $T(N_T,E_T)$



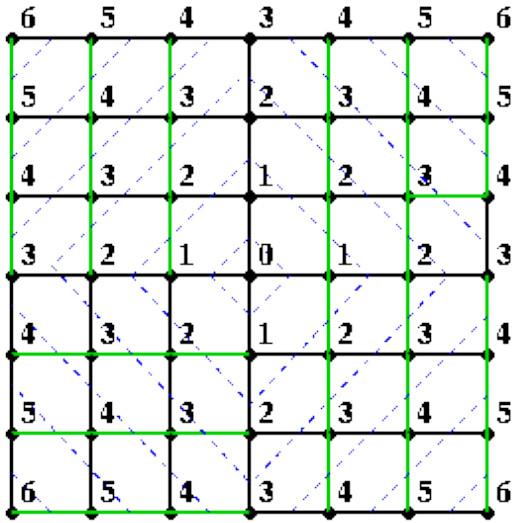
N_T = {(r,0)}, E_T = empty set … Initially T = root r, which is at level 0
Enqueue((r,0),Q) … Put root on initially empty Queue Q
Mark r … Mark root as having been processed
While Q not empty … While nodes remain to be processed
    (n,level) = Dequeue(Q) … Get a node to process
    For all unmarked children c of n
        N_T = N_T U (c,level+1) … Add child c to N_T
        E_T = E_T U (n,c) … Add edge (n,c) to E_T
        Enqueue((c,level+1),Q)) … Add child c to Q for processing
        Mark c … Mark c as processed
    Endfor
Endwhile

Partitioning via Breadth First Search

- BFS identifies 3 kinds of edges
  - Tree Edges - part of T
  - Horizontal Edges - connect nodes at same level
  - Interlevel Edges - connect nodes at adjacent levels
- No edges connect nodes in levels differing by more than 1 (why?)
- BFS partitioning heuristic
  - N = N_1 U N_2, where
    - N_1 = {nodes at level <= L},
    - N_2 = {nodes at level > L}
  - Choose L so $|N_1|$ close to $|N_2|$



Breadth First Seach on a 7 by 7 Grid
Starting at the center node
Nodes labeled by level



Partitioning without nodal coordinates - Kernighan/Lin

- Take a initial partition and iteratively improve it
  - Kernighan/Lin (1970), cost = $O(|N|^3)$ but easy to understand, better version has cost = $O(|E| \log |E|)$
  - Fiduccia/Mattheyses (1982), cost = $O(|E|)$, much better, but more complicated (it uses the appropriate data structures)
- Given $G = (N,E,W_E)$ and a partitioning $N = A \cup B$, where $|A| = |B|$
  - T = cost(A,B) = edge cut of A and B partitions
  - Find subsets X of A and Y of B with $|X| = |Y|$
  - Swapping X and Y should decrease cost:
    - $newA = (A - X) \cup Y$ and $newB = (B - Y) \cup X$
    - $newT = cost(newA, newB) < cost(A,B)$, lower edge cut
- Need to compute newT efficiently for many possible X and Y, choose smallest

Kernighan/Lin - Preliminary Definitions

- T = cost(A, B), newT = cost(newA, newB)
- Need an efficient formula for newT; will use
  - E(a) = external cost of a in A = Σ {W(a,b) for b in B}
  - I(a) = internal cost of a in A = Σ {W(a,a') for other a' in A}
  - D(a) = cost of a in A = E(a) - I(a)
  - E(b), I(b) and D(b) defined analogously for b in B
- Consider swapping X = {a} and Y = {b}
  - newA = (A - {a}) U {b}, newB = (B - {b}) U {a}
- newT = T - (D(a) + D(b) - 2*w(a,b)) = T - gain(a,b)
  - gain(a,b) measures improvement gotten by swapping a and b
- Update formulas
  - newD(a') = D(a') + 2*w(a',a) - 2*w(a',b) for a' in A, a' != a
  - newD(b') = D(b') + 2*w(b',b) - 2*w(b',a) for b' in B, b' != b

Kernighan/Lin Algorithm

```
Compute  T = cost(A,B) for initial A, B                    … cost = O(|N|^2)
Repeat
     Compute costs D(n) for all n in N                     … cost = O(|N|^2)
     Unmark all nodes in N                                 … cost = O(|N|)
     While there are unmarked nodes                        … |N|/2 iterations
          Find an unmarked pair (a,b) maximizing gain(a,b) … cost = O(|N|^2)
          Mark a and b (but do not swap them)              … cost = O(1)
          Update D(n) for all unmarked n,
               as though a and b had been swapped          … cost = O(|N|)
      Endwhile
          … At this point we have computed a sequence of pairs
          …  (a1,b1), … , (ak,bk)   and gains gain(1),…., gain(k)
          …  for k = |N|/2,  ordered by the order in which we marked them
     Pick j maximizing Gain = Σ_{k=1 to j} gain(k)          … cost = O(|N|)
          … Gain is reduction in cost from swapping (a1,b1) through (aj,bj)
     If Gain > 0 then   … it is worth swapping
          Update newA = (A - { a1,…,ak }) U { b1,…,bk }     … cost = O(|N|)
          Update newB = (B - { b1,…,bk }) U { a1,…,ak }     … cost = O(|N|)
          Update T = T - Gain                               … cost = O(1)
     endif
 Until Gain <= 0
```

- One pass greedily computes |N|/2 possible X and Y to swap, picks best

Comments on Kernighan/Lin Algorithm

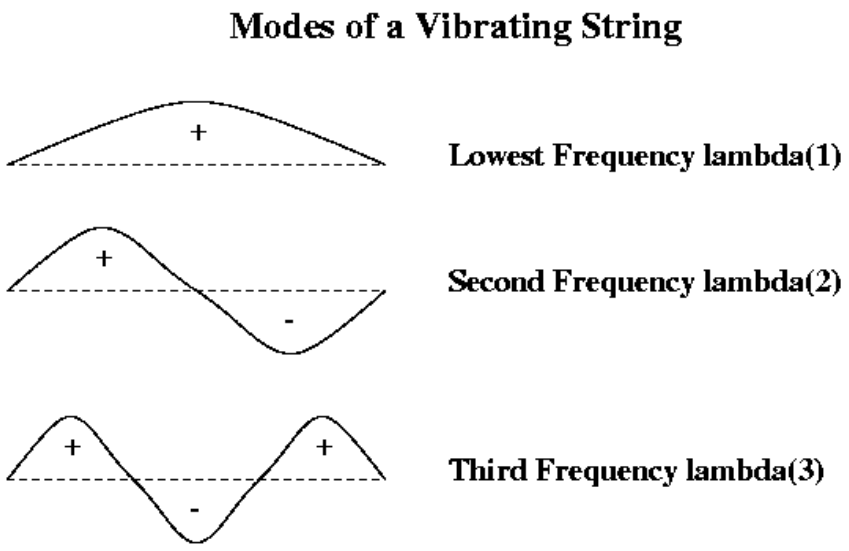
- Most expensive line show in red
- Some gain(k) may be negative, but if later gains are large, then final Gain may be positive
  - can escape "local minima" where switching no pair helps
- How many times do we Repeat?
  - K/L tested on very small graphs (|N|<=360) and got convergence after 2-4 sweeps
  - For random graphs (of theoretical interest) the probability of convergence in one step appears to drop like $2^{-|N|/30}$

Partitioning without nodal coordinates - Spectral Bisection

- Based on theory of Fiedler (1970s), popularized by Pothen, Simon, Liou (1990)
- Motivation, by analogy to a vibrating string
- Basic definitions
- Implementation via the Lanczos Algorithm
  - To optimize sparse-matrix-vector multiply, we graph partition
  - To graph partition, we find an eigenvector of a matrix associated with the graph
  - To find an eigenvector, we do sparse-matrix vector multiply
  - No free lunch ...

Motivation for Spectral Bisection: Vibrating String

- Think of G = 1D mesh as masses (nodes) connected by springs (edges), i.e. a string that can vibrate
- Vibrating string has modes of vibration, or harmonics
- Label nodes by whether mode - or + to partition into N- and N+
- Same idea for other graphs (eg planar graph ~ trampoline)



Basic Definitions

- *Definition*: The incidence matrix In(G) of a graph G(N,E) is an |N| by |E| matrix, with one row for each node and one column for each edge. If edge e=(i,j) then column e of In(G) is zero except for the i-th and j-th entries, which are +1 and -1, respectively.

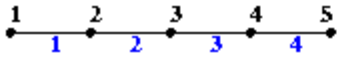
- Slightly ambiguous definition because multiplying column e of In(G) by -1 still satisfies the definition, but this won't matter...

- *Definition*: The Laplacian matrix L(G) of a graph G(N,E) is an |N| by |N| symmetric matrix, with one row and column for each node. It is defined by
  - L(G) (i,i) = degree of node I (number of incident edges)
  - L(G) (i,j) = -1 if i != j and there is an edge (i,j)
  - L(G) (i,j) = 0 otherwise

Example of In(G) and L(G) for 1D and 2D meshes

Incidence and Laplacian Matrices

Graph G

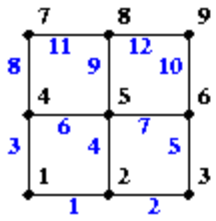


Incidence Matrix In(G)

	1	2	3	4
1	-1			
2	1	-1		
3		1	-1	
4			1	-1
5				1

Laplacian Matrix L(G)

	1	2	3	4	5
1	1	-1			
2	-1	2	-1		
3		-1	2	-1	
4			-1	2	-1
5				-1	1



	1	2	3	4	5	6	7	8	9	10	11	12
1	-1		1									
2	1	-1		1								
3		1			1							
4			-1			-1		1				
5				-1		1	-1		1			
6					-1		1			1		
7								-1			-1	
8									-1		1	-1
9										-1		1

	1	2	3	4	5	6	7	8	9
1	2	-1		-1					
2	-1	3	-1		-1				
3		-1	2			-1			
4	-1			3	-1		-1		
5		-1		-1	4	-1		-1	
6			-1		-1	3			-1
7				-1			2	-1	
8					-1		-1	3	-1
9						-1		-1	2

Nodes numbered in black

Edges numbered in blue

Properties of Incidence and Laplacian matrices

- *Theorem 1:* Given G, In(G) and L(G) have the following properties
- L(G) is symmetric. (This means the eigenvalues of L(G) are real and its eigenvectors are real and orthogonal.)
  - Let e = $[1,\ldots,1]^T$, i.e. the column vector of all ones. Then L(G)*e=0.
  - In(G) * $(In(G))^T$ = L(G). This is independent of the signs chosen for each column of In(G).
  - Suppose L(G)*v = λ*v, v != 0, so that v is an eigenvector and λ an eigenvalue of L(G). Then

    $\lambda = || In(G)^T * v ||^2 / || v ||^2$
    $= \Sigma \{ (v(i)-v(j))^2 \text{ for all edges } e=(i,j) \} / \Sigma_i v(i)^2$

    … $||x||^2 = \Sigma_k x_k^2$
  - The eigenvalues of L(G) are nonnegative:
    - $0 = \lambda_1 <= \lambda_2 <= \ldots <= \lambda_n$
  - The number of connected components of G is equal to the number of λ_i equal to 0. In particular, λ_2 != 0 if and only if G is connected.
- *Definition*: $\lambda_2(L(G))$ is the algebraic connectivity of G

Spectral Bisection Algorithm

- Spectral Bisection Algorithm:
  - Compute eigenvector v_2 corresponding to $\lambda_2(L(G))$
  - For each node n of G
    - if $v_2(n) < 0$ put node n in partition N-
    - else put node n in partition N+
- Why does this make sense? First reasons.
- *Theorem 2 (Fiedler, 1975):* Let G be connected, and N- and N+ defined as above. Then N- is connected. If no $v_2(n) = 0$, then N+ is also connected. Proof available.
- Recall $\lambda_2(L(G))$ is the algebraic connectivity of G
- *Theorem 3 (Fiedler):* Let $G_1(N,E_1)$ be a subgraph of G(N,E), so that G_1 is "less connected" than G. Then $\lambda_2(L(G)) <= \lambda_2(L(G))$, i.e. the algebraic connectivity of G_1 is less than or equal to the algebraic connectivity of G.

References

- A. Pothen, H. Simon, K.-P. Liou, “Partitioning sparse matrices with eigenvectors of graphs”, SIAM J. Mat. Anal. Appl. 11:430-452 (1990)
- M. Fiedler, “Algebraic Connectivity of Graphs”, Czech. Math. J., 23:298-305 (1973)
- M. Fiedler, Czech. Math. J., 25:619-637 (1975)
- B. Parlett, “The Symmetric Eigenproblem”, Prentice-Hall, 1980

Review

- Partitioning with nodal coordinates
  - Rely on graphs having nodes connected (mostly) to "nearest neighbors" in space
  - Common when graph arises from physical model
  - Finds a circle or line that splits nodes into two equal-sized groups
  - Algorithm very efficient, does not depend on edges
- Partitioning without nodal coordinates
  - Depends on edges
  - Breadth First Search (BFS)
  - Kernighan/Lin - iteratively improve an existing partition
  - Spectral Bisection - partition using signs of components of second eigenvector of L(G), the Laplacian of G

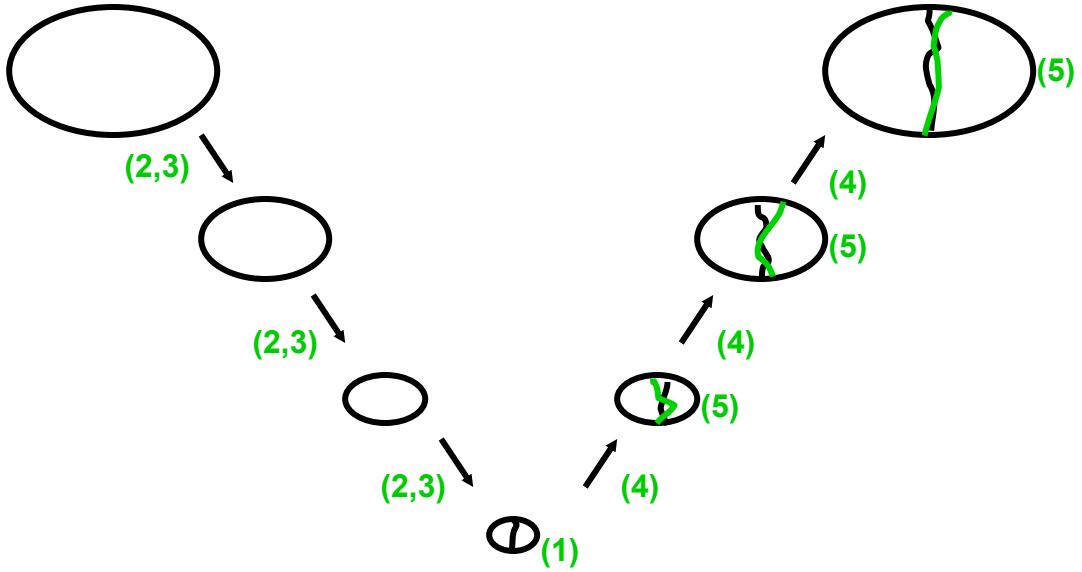
Introduction to Multilevel Partitioning

- If we want to partition G(N,E), but it is too big to do efficiently, what can we do?
  - 1) Replace G(N,E) by a coarse approximation $G_c(N_c,E_c)$, and partition G_c instead
  - 2) Use partition of G_c to get a rough partitioning of G, and then iteratively improve it
- What if G_c still too big?
  - Apply same idea recursively
- This is identical to the multigrid procedure that is used in the solution of elliptic and hyperbolic PDEs

Multilevel Partitioning - High Level Algorithm

```
(N+,N- ) = Multilevel_Partition( N, E )
      … recursive partitioning routine returns N+ and N- where N = N+ U N-
      if |N| is small
(1)         Partition G = (N,E)  directly to get N = N+ U N-
            Return (N+, N- )
      else
(2)         Coarsen G to get an approximation G_C = (N_C, E_C)
(3)         (N_C+ , N_C- ) = Multilevel_Partition( N_C, E_C )
(4)         Expand (N_C+ , N_C- ) to a partition  (N+ , N- ) of N
(5)         Improve the partition ( N+ , N- )
            Return ( N+ , N- )
      endif
```

"V - cycle:"



How do we
- Coarsen?
- Expand?
- Improve?

Multilevel Kernighan-Lin

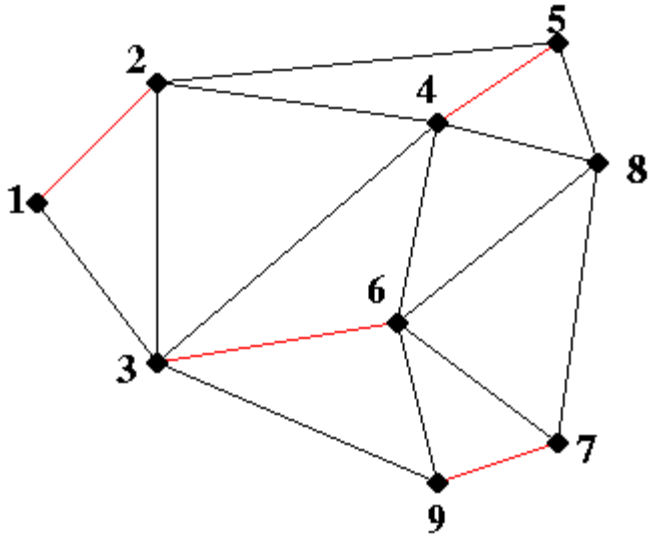
- Coarsen graph and expand partition using maximal matchings
- Improve partition using Kernighan-Lin
- This is the algorithm that is implemented in Metis (see references in web page)

Maximal Matching

- *Definition*: A matching of a graph G(N,E) is a subset E_m of E such that no two edges in E_m share an endpoint
- *Definition:* A maximal matching of a graph G(N,E) is a matching E_m to which no more edges can be added and remain a matching
- A simple greedy algorithm computes a maximal matching:

```
let Em be empty
mark all nodes in N as unmatched
for i = 1 to |N|      … visit the nodes in any order
    if i has not been matched
          if there is an edge e=(i,j)  where j is also unmatched,
               add e to Em
               mark i and j as matched
           endif
    endif
endfor
```

Maximal Matching - Example



Maximal matching given by red edges:

Any additional edge will connect to one of the nodes already present

Coarsening using a maximal matching

Construct a maximal matching E_m of G(N,E)

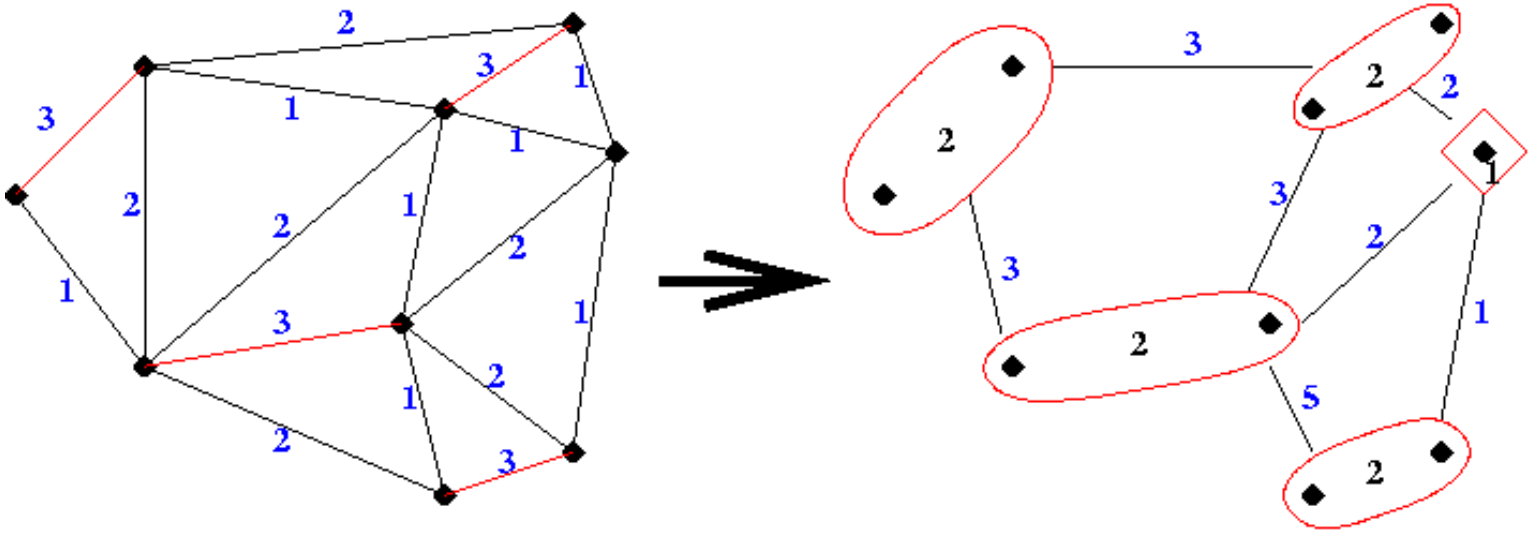
for all edges e=(j,k) in E_m
Put node n(e) in N_c
W(n(e)) = W(j) + W(k) … gray statements update node/edge weights
for all nodes n in N not incident on an edge in E_m
Put n in N_c … do not change W(n)
… Now each node r in N is "inside" a unique node n(r) in N_c

… Connect two nodes in Nc if nodes inside them are connected in E
for all edges e=(j,k) in E_m
for each other edge e' =(j,r) in E incident on j
Put edge ee = (n(e),n(r)) in E_c
W(ee) = W(e')
for each other edge e' =(r,k) in E incident on k
Put edge ee = (n(r),n(e)) in E_c
W(ee) = W(e')

If there are multiple edges connecting two nodes in N_c, collapse them,
adding edge weights

Example of Coarsening

How to coarsen a graph using a maximal matching



$G = (N, E)$

E_m is shown in red

Edge weights shown in blue

Node weights are all one

$G_c = (N_c, E_c)$

N_c is shown in red

Edge weights shown in blue

Node weights shown in black

Example of Coarsening

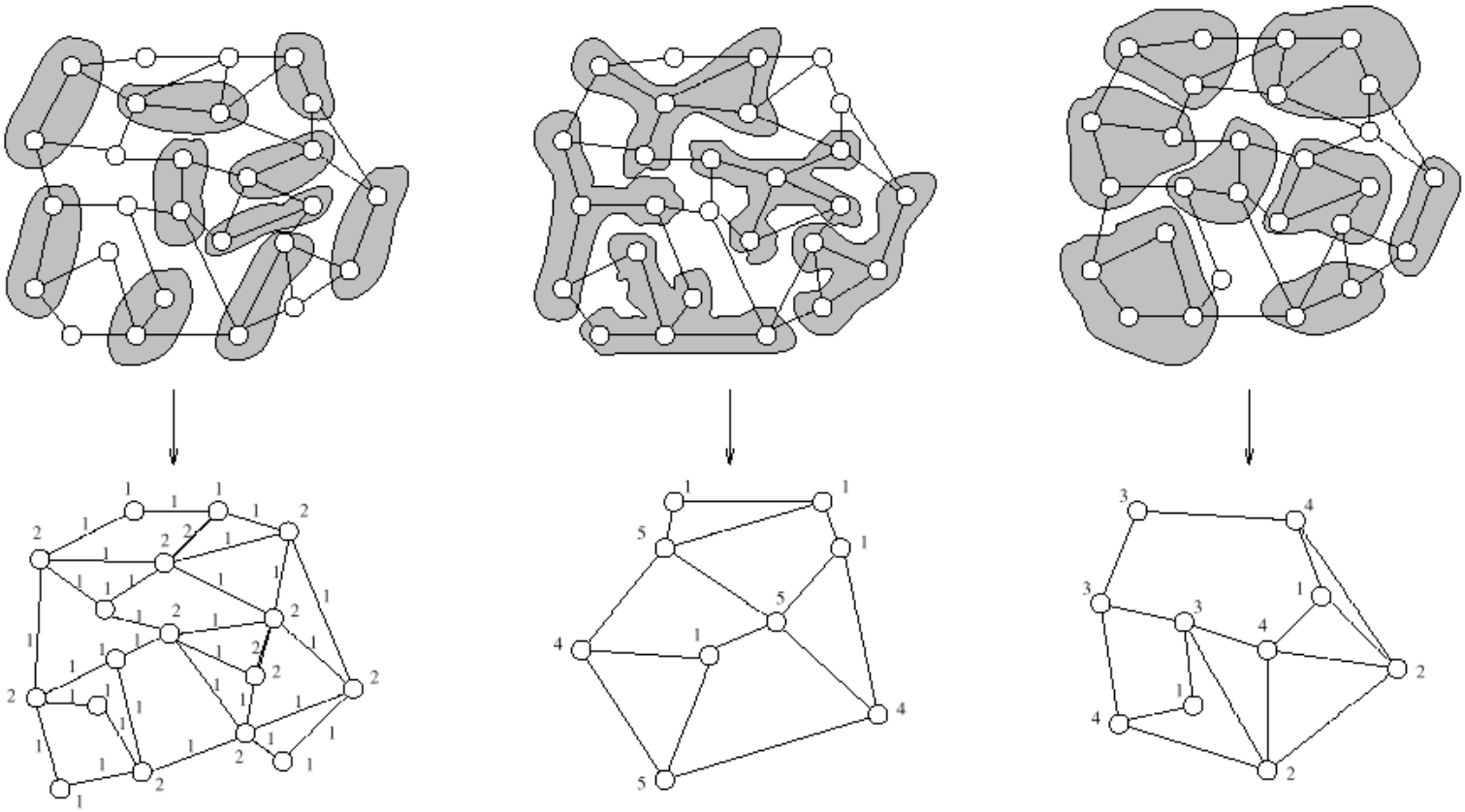
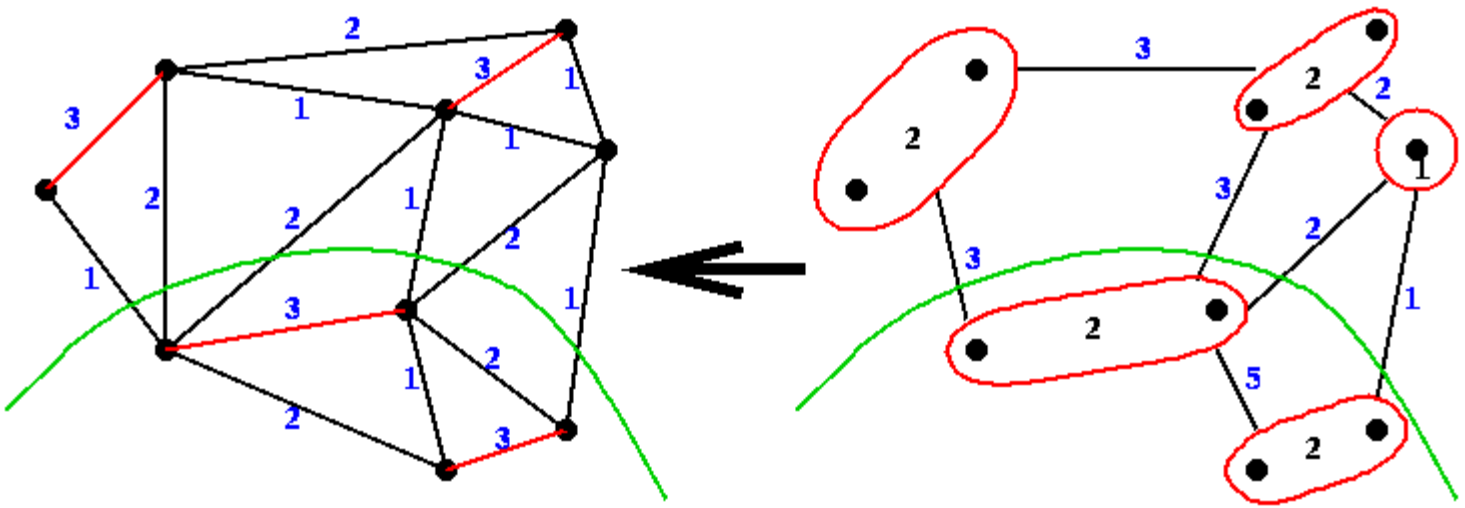


Figure 2: Different ways to coarsen a graph.

Expanding a partition of G_c to a partition of G

Converting a coarse partition to a fine partition



Partition shown in green

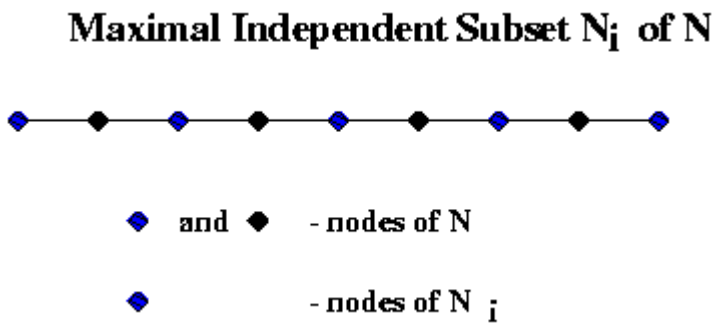
Multilevel Spectral Bisection

- Coarsen graph and expand partition using maximal independent sets
- Improve partition using Rayleigh Quotient Iteration

Maximal Independent Sets

- *Definition*: An independent set of a graph G(N,E) is a subset N_i of N such that no two nodes in N_i are connected by an edge
- *Definition:* A maximal independent set of a graph G(N,E) is an independent set N_i to which no more nodes can be added and remain an independent set
- A simple greedy algorithm computes a maximal independent set:

```
let Ni be empty
for i = 1 to |N|      … visit the nodes in any order
    if  node i is not adjacent to any node already in Ni
        add i to Ni
    endif
endfor
```



Coarsening using Maximal Independent Sets

… Build "domains" D(i) around each node i in N_i to get nodes in N_C
… Add an edge to E_C whenever it would connect two such domains

```
E_C = empty set
for all nodes i in N_i
    D(i) = ( {i}, empty set )
    … first set contains nodes in D(i), second set contains edges in D(i)
unmark all edges in E
repeat
    choose an unmarked edge e = (i,j) from E
    if exactly one of i and j (say i) is in some D(k)
        mark e
        add j and e to D(k)
    else if i and j are in two different D(k)'s (say D(ki) and D(kj))
        mark e
        add edge (ki, kj) to E_C
    else if both i and j are in the same D(k)
        mark e
        add e to D(k)
    else
        leave e unmarked
    endif
until no unmarked edges
```

Available Implementations

- Multilevel Kernighan/Lin
  - METIS (www.cs.umn.edu/~metis)
  - ParMETIS - parallel version
- Multilevel Spectral Bisection
  - S. Barnard and H. Simon, "A fast multilevel implementation of recursive spectral bisection …", Proc. 6th SIAM Conf. On Parallel Processing, 1993
  - Chaco (www.cs.sandia.gov/CRF/papers_chaco.html)
- Hybrids possible
  - Ex: Using Kernighan/Lin to improve a partition from spectral bisection

Available Implementations

- Multilevel Kernighan/Lin
  - Demonstrated in experience to be the most efficient algorithm available.
- Multilevel Spectral Bisection
  - Gives good partitions but cost is higher than multilevel K/L
- Hybrids possible
  - For example: Using Kernighan/Lin to improve a partition from spectral bisection