# Partitioning Problem and Usage

## Lecture 8

CSCI 4974/6971

26 Sep 2016

# Today's Biz

1. Reminders
2. Review
3. Graph Partitioning overview
4. Graph Partitioning Small-world Graphs
5. Partitioning Usage example

# Today's Biz

1. **Reminders**
2. Review
3. Graph Partitioning overview
4. Graph Partitioning Small-world Graphs
5. Partitioning Usage example

# Reminders

- Assignment 2: Thursday 29 Sept 16:00
- Project Presentation 1: in class 6 October
  - Email me your slides (pdf only please) before class
  - 5-10 minute presentation
  - Introduce topic, give background, current progress, expected results
- Assignment 3: Thursday 13 Oct 16:00 (social analysis, posted soon)
- Office hours: Tuesday & Wednesday 14:00-16:00 Lally 317
  - Or email me for other availability

# Today's Biz

1. Reminders
2. **Review**
3. Graph Partitioning overview
4. Graph Partitioning Small-world Graphs
5. Partitioning Usage example

# Quick Review

- Communities in social networks
  - Explicitly formed by users
  - Implicit through interactions
- Community detection
  - Identifying communities - very subjective definition
  - Node, Group, Network, Hierarchical (top-down and/or bottom-up) methods
- Evaluation of detection methods
  - Based on method - e.g. found a k-clique?
  - Comparison to ground truth - explicitly formed communities
  - Implicit based on measurement - modularity, cut ratio, etc.

# Today's Biz

1. Reminders
2. Review
3. **Graph Partitioning Overview**
4. Graph Partitioning Small-world Graphs
5. Partitioning Usage example

## Graph Partitioning Overview

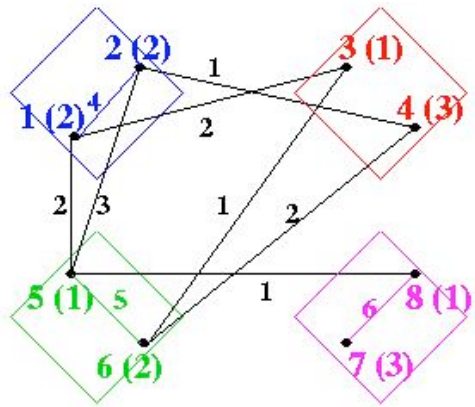
*Slides from Spring 2005 CME342/AA220/CS238 - Parallel Methods in Numerical Analysis, Stanford University*

# 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

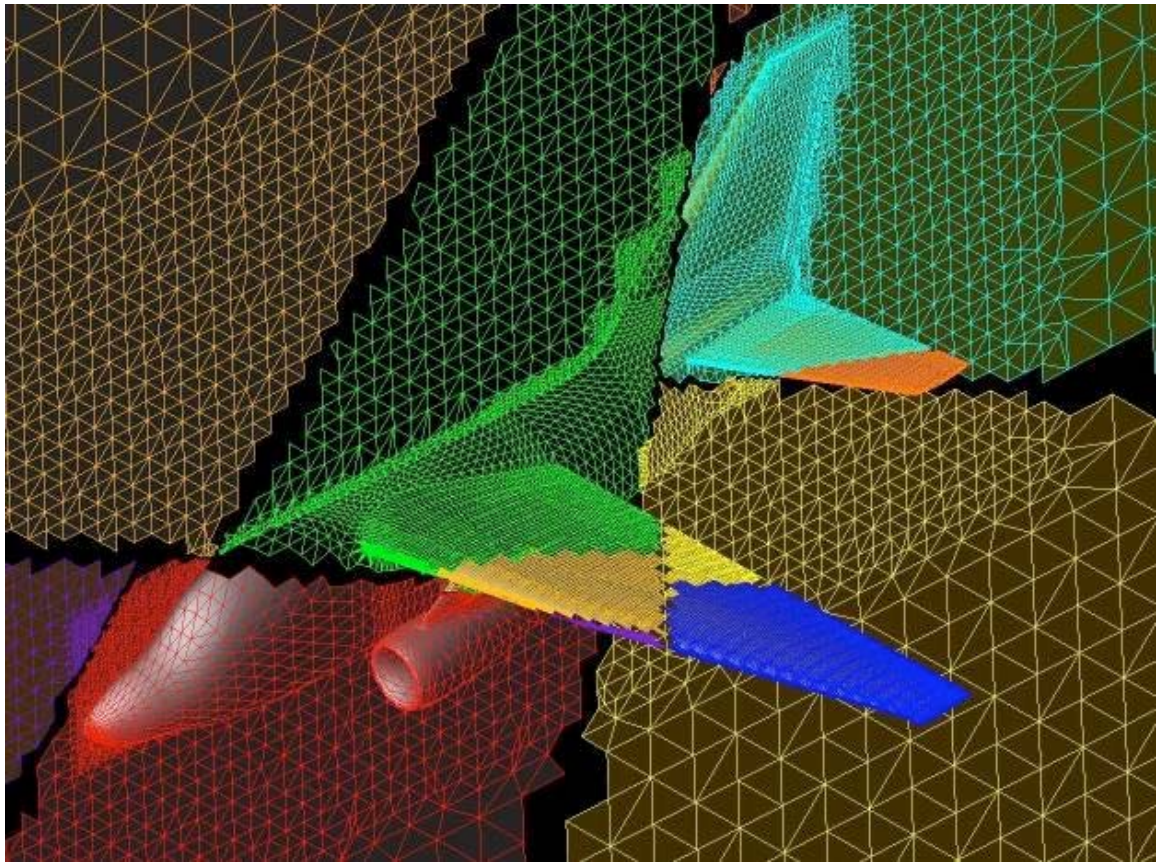


# Applications

- 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”

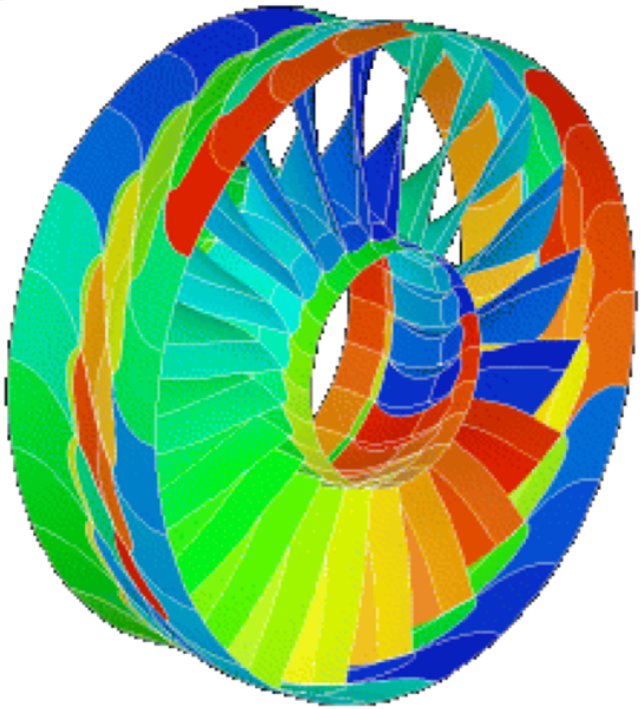
# Applications-Unstructured CFD

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.



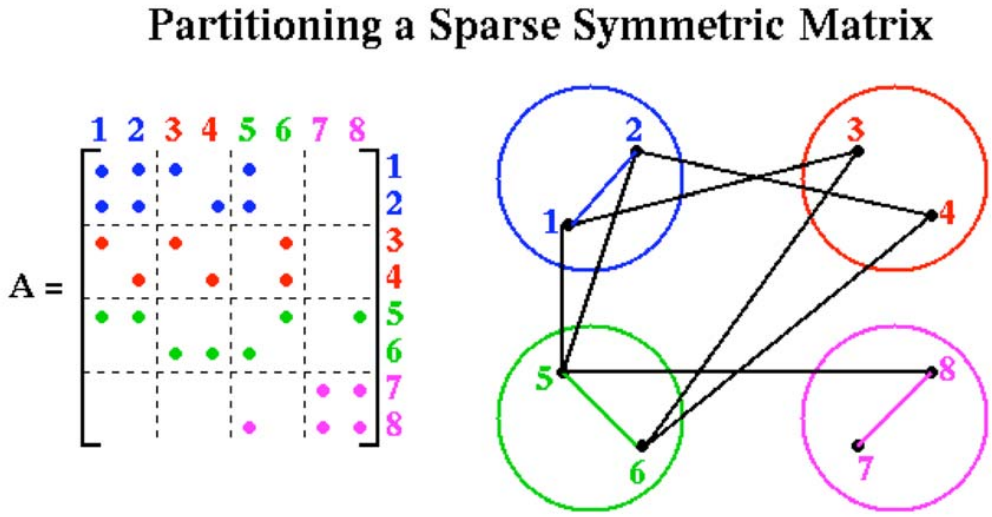
# Applications- TFLO Load Balancing

The static load balancing procedure for the multiblock-structured flow solver, TFLO, developed for the ASCI 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



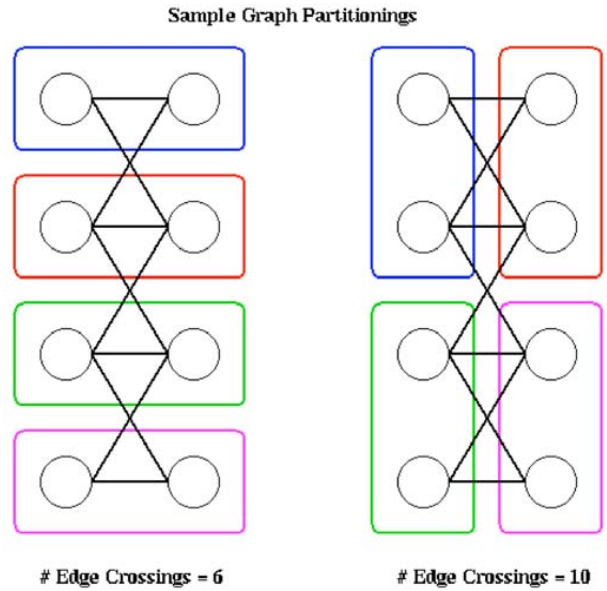
Now, that is where that silly picture comes from!!!!!

# Sparse Matrix Vector Multiplication



# Cost of Graph Partitioning

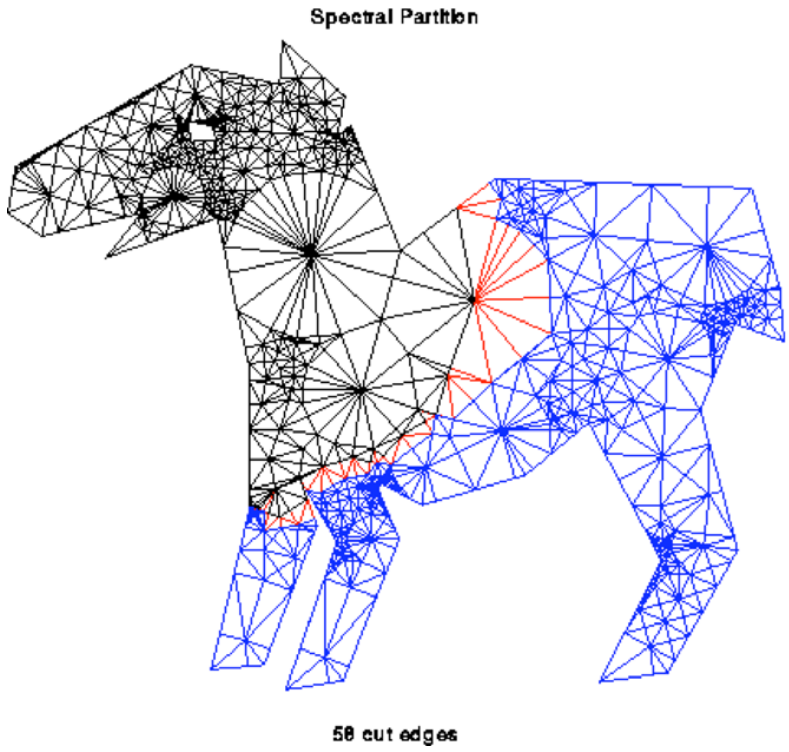
- Many possible partitionings to search:




- 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!!

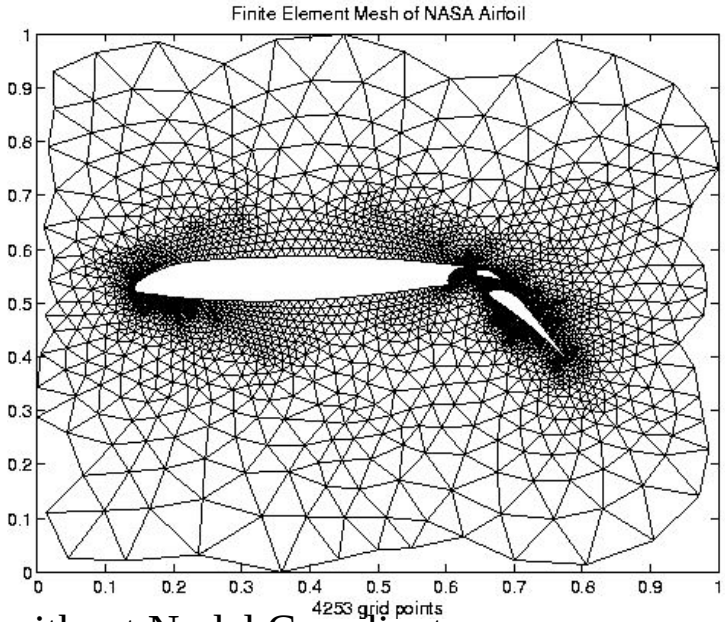
# First Heuristic: Repeated Graph Bisection

- 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



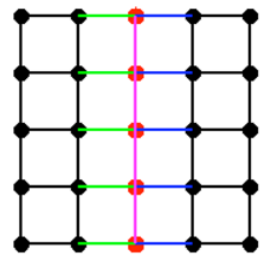

- Partitioning without Nodal Coordinates
  - Sparse matrix of Web: A(j,k) = # times keyword j appears in URL k
- Multilevel acceleration
  - Approximate problem by “coarse graph”, do so recursively

# 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

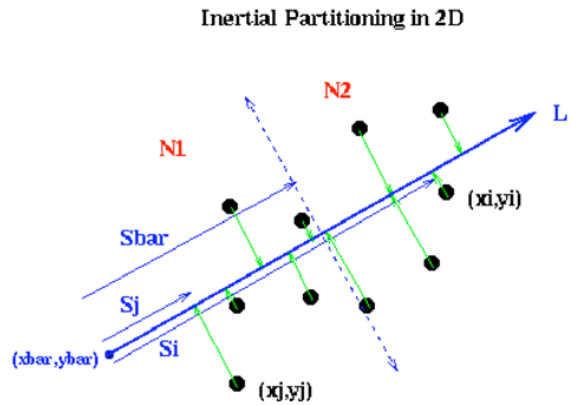
# Graphs with Nodal Coordinates - Planar graphs

- Planar graph can be drawn in plane without edge crossings
- Ex: m x m grid of $m^2$ nodes: ∃ vertex separator $N_s$ with $|N_s| = m = sqrt(|N|)$ (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 $|N_2| <= 2/3 |N|$
  - $|N_s| <= sqrt(8 * |N|)$
- 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

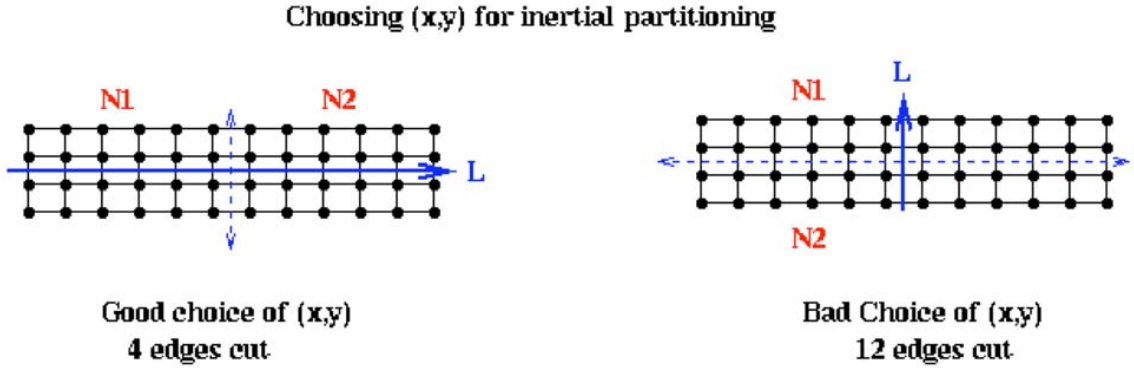
1) **L given by a*(x-xbar)+b*(y-ybar)=0, with $a^2+b^2=1$; (a,b) is unit vector $\perp$ to L**
2) **For each nj = (xj,yj), compute coordinate $S_j = -b*(x_j-xbar) + a*(y_j-ybar)$ along L**
3) **Let Sbar = median($S_1,\ldots,S_n$)**
4) **Let nodes with $S_j$ < Sbar be in $N_1$, rest in $N_2$**




- Remains to choose L

# Inertial Partitioning: Choosing L

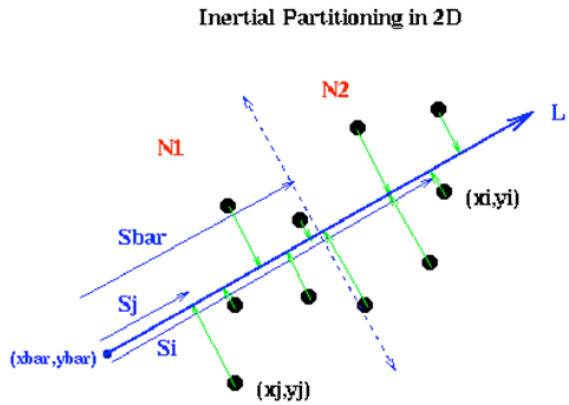
- Clearly prefer L on left below




- 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




$\Sigma_j$ (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

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

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

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

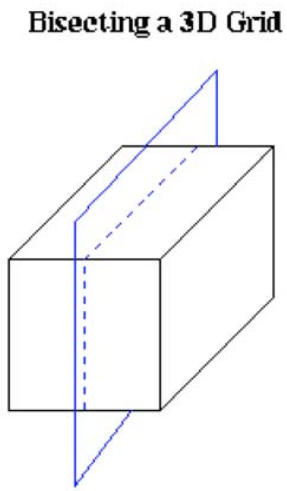
Minimized by choosing

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

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

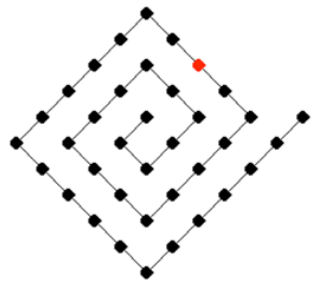
# Inertial Partitioning: Three Dimensions

- 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



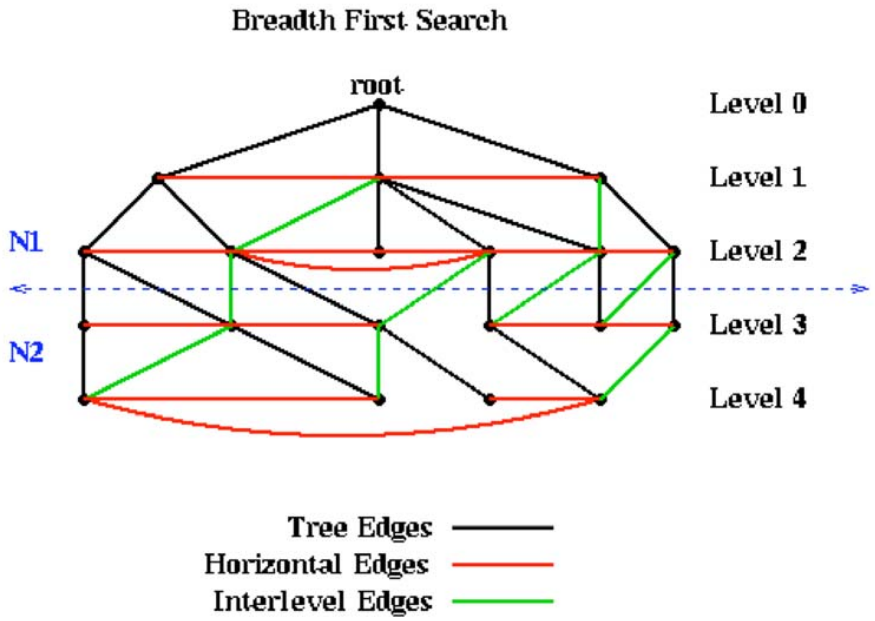
# Partitioning with Nodal Coordinates - Summary

- 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:



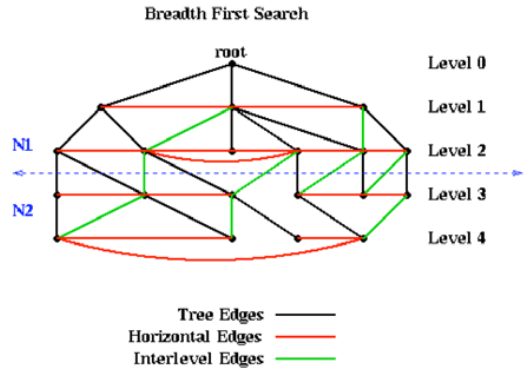
# Partitioning without Nodal Coordinates- Breadth First Search (BFS)

- 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

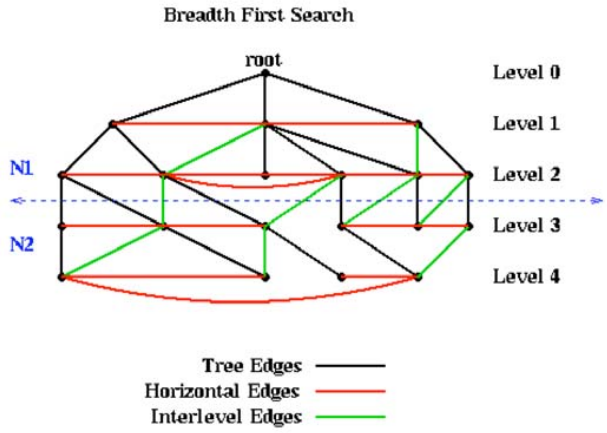
- Queue (First In First Out, or FIFO)
  - Enqueue(x,Q) adds x to back of Q
  - x = Dequeue(Q) removes x from front of Q
- Compute Tree $T(N_T,E_T)$



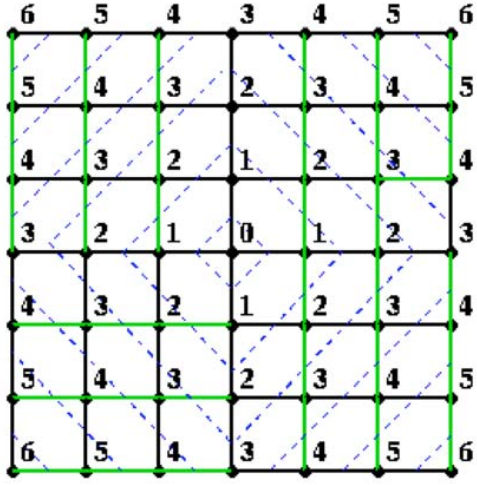

**$N_T$ = {(r,0)}, $E_T$ = empty set** … Initially T = root r, which is at level 0
**Enqueue((r,0),Q)** … Put root on initially empty Queue Q
**Mark r** … Mark root as having been processed
**While Q not empty** … While nodes remain to be processed
    **(n,level) = Dequeue(Q)** … Get a node to process
    **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
  - $N = N_1 \cup N_2$, 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) U Y and newB = (B - Y) U 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

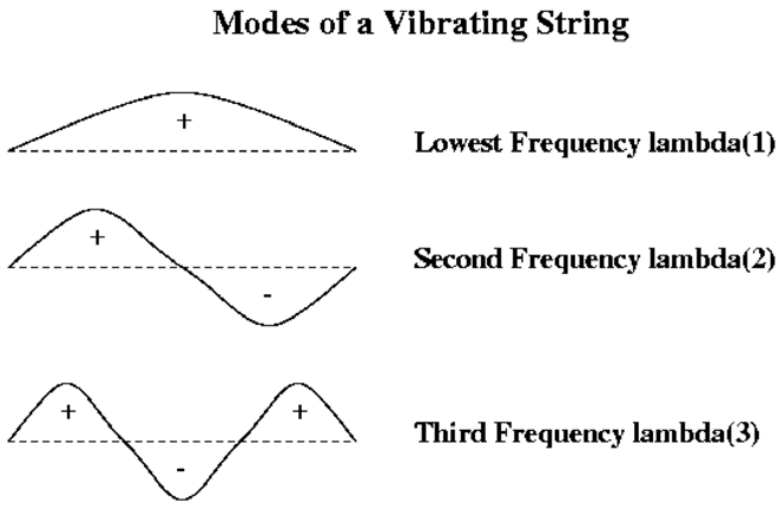
- Most expensive line show in red
- Some gain(k) may be negative, but if later gains are large, then final Gain may be positive
  - can escape “local minima” where switching no pair helps
- How many times do we Repeat?
  - K/L tested on very small graphs (|N|<=360) and got convergence after 2-4 sweeps
  - 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.

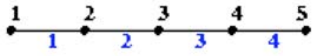
- Slightly ambiguous definition because multiplying column e of In(G) by -1 still satisfies the definition, but this won't matter...

- *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**

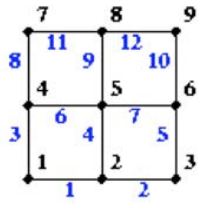



**Incidence Matrix In(G)**

| | 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

    $\lambda = || In(G)^T * v ||^2 / || v ||^2$ … $||x||^2 = \Sigma_k x_k^2$

    $= \Sigma \{ (v(i)-v(j))^2$ for all edges e=(i,j) $\} / \Sigma_i v(i)^2$
  - 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 $\lambda_i$ equal to 0. In particular, $\lambda_2$ != 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

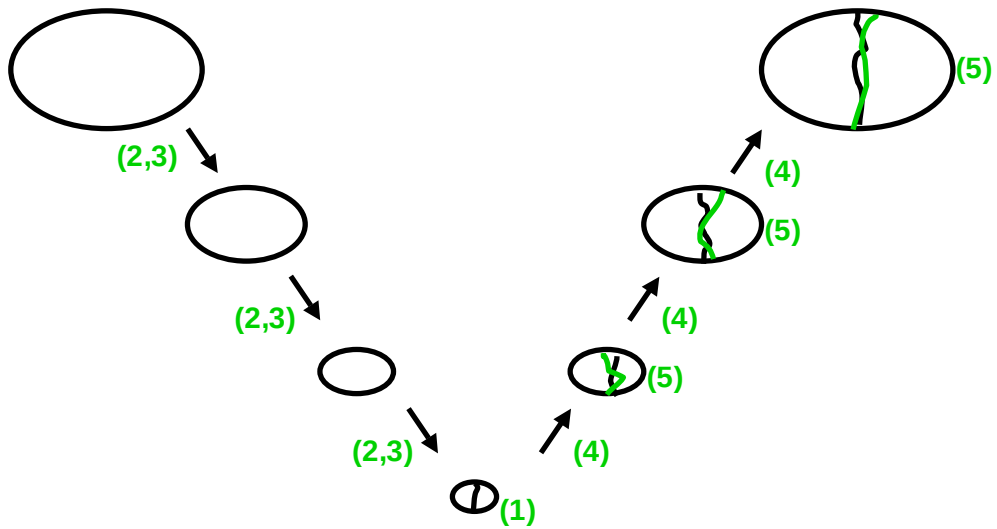
# Introduction to Multilevel Partitioning

- 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

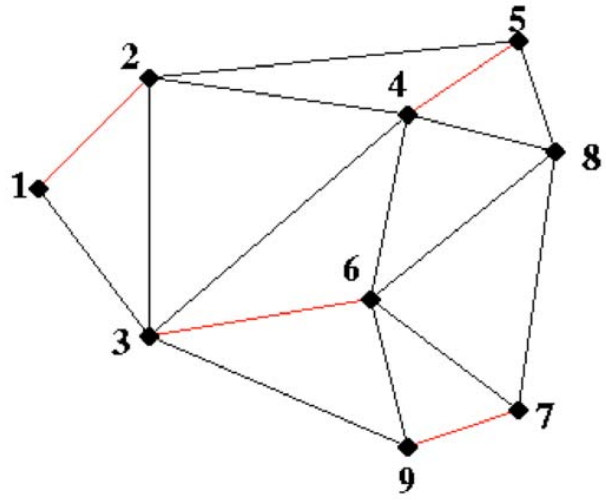
- Coarsen graph and expand partition using maximal matchings
- Improve partition using Kernighan-Lin
- This is the algorithm that is implemented in Metis (see references in web page)

# 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)**

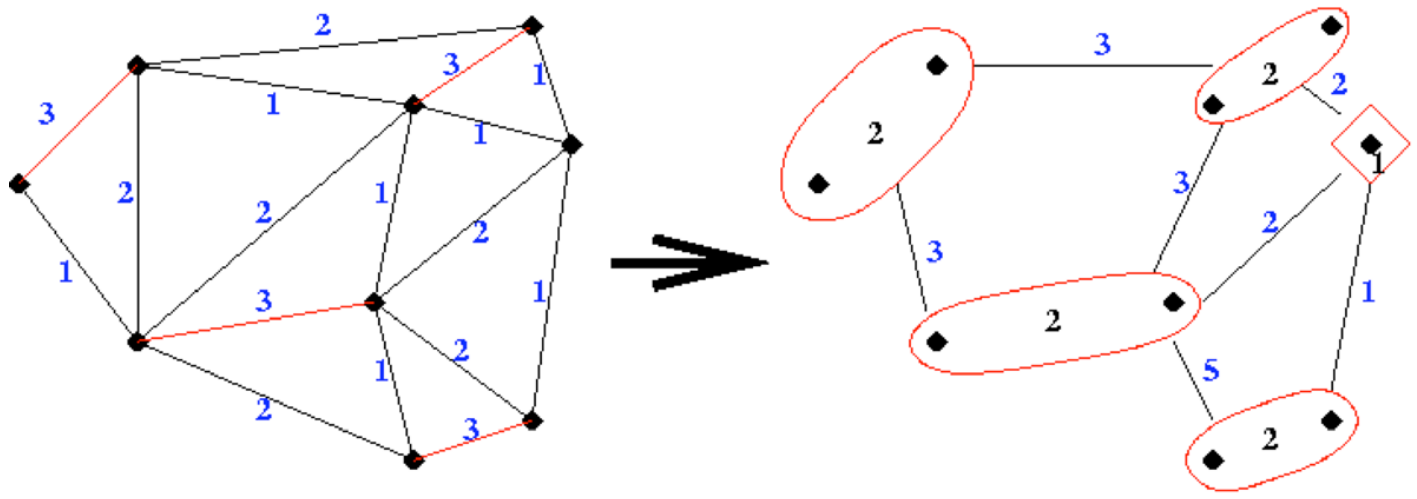
**for all edges e=(j,k) in $E_m$**
**Put node n(e) in $N_c$**
**W(n(e)) = W(j) + W(k)** … gray statements update node/edge weights
**for all nodes n in N not incident on an edge in $E_m$**
**Put n in $N_c$** … do not change W(n)
**… Now each node r in N is "inside" a unique node n(r) in $N_c$**

**… 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

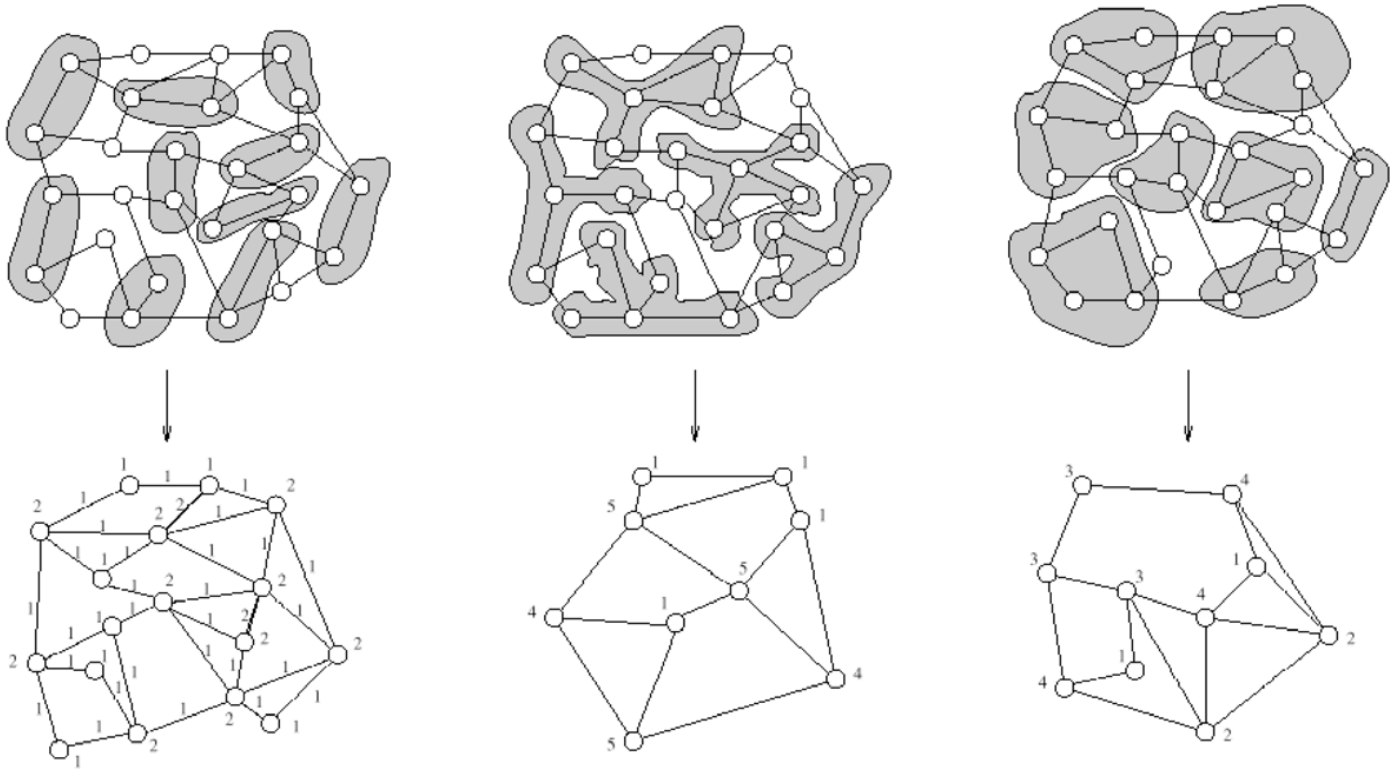
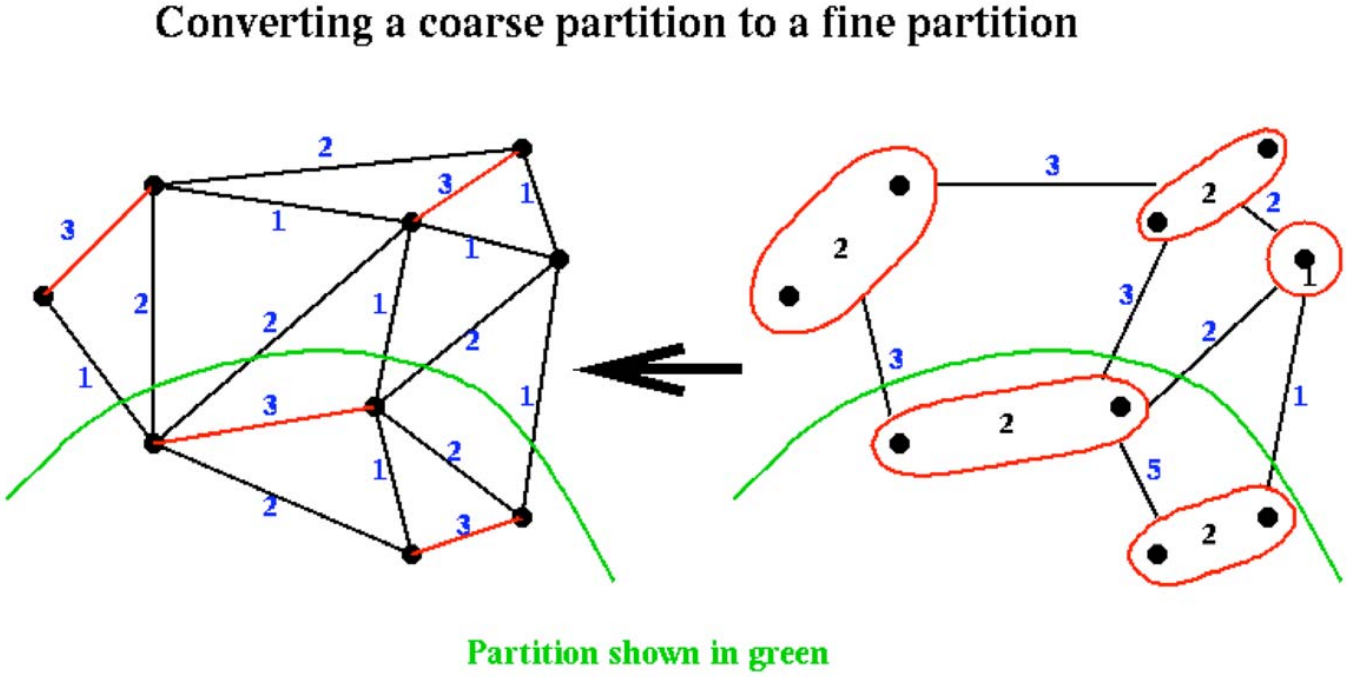


**Figure 2**: Different ways to coarsen a graph.

# Expanding a partition of $G_c$ to a partition of G



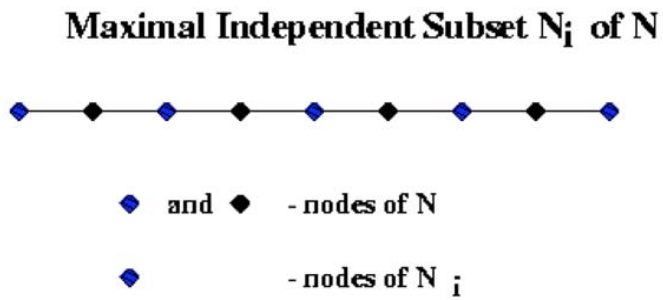
# Multilevel Spectral Bisection

- 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

# Today's Biz

1. Reminders
2. Review
3. Graph Partitioning overview
4. **Graph Partitioning of Small-world Graphs**
5. Partitioning Usage example

# Graph Partitioning of Small-world Graphs

- Large and irregular graphs require a different approach
  - Direct methods (spectral/KM): $O(n^2)$ - not feasible
  - Multilevel methods:
    - Matching difficult with high degree vertices
    - Coarsening comes with high memory costs
- Techniques for large small-world graphs:
  - Simple clustering heuristics - balanced label propagation
  - Streaming methods - make greedy decisions as you scan a graph
  - Both linear time complexity, avoid coarsening overheads

# Label Propagation Partitioning (PuLP)

# Overview
## Partitioning

- **Graph Partitioning**: Given a graph $G(V, E)$ and $p$ processes or tasks, assign each task a $p$-way disjoint subset of vertices and their incident edges from $G$
  - Balance constraints – (weighted) vertices per part, (weighted) edges per part
  - Quality metrics – edge cut, communication volume, maximal per-part edge cut
- We consider:
  - Balancing edges **and** vertices per part
  - Minimizing edge cut ($EC$) **and** maximal per-part edge cut ($EC_{max}$)

# Overview
## Partitioning - Objectives and Constraints

- Lots of graph algorithms follow a certain iterative model
  - BFS, SSSP, FASCIA subgraph counting (Slota and Madduri 2014)
  - computation, synchronization, communication, synchronization, computation, etc.
- Computational load: proportional to vertices and edges per-part
- Communication load: proportional to total edge cut and max per-part cut
- We want to minimize the maximal time among tasks for each comp/comm stage

- Balance vertices and edges:

$$(1-\epsilon_l)\frac{|V|}{p} \leq |V(\pi_i)| \leq (1+\epsilon_u)\frac{|V|}{p} \quad (1)$$

$$|E(\pi_i)| \leq (1+\eta_u)\frac{|E|}{p} \quad (2)$$

- $\epsilon_l$ and $\epsilon_u$: lower and upper vertex imbalance ratios
- $\eta_u$: upper edge imbalance ratio
- $V(\pi_i)$: set of vertices in part $\pi_i$
- $E(\pi_i)$: set of edges with both endpoints in part $\pi_i$

# Overview
## Partitioning - Objectives

- Given a partition $\Pi$, the set of *cut edges* ($C(G, \Pi)$) and cut edge per partition ($C(G, \pi_k)$) are

$$C(G, \Pi) = \{\{(u, v) \in E\} \mid \Pi(u) \neq \Pi(v)\} \tag{3}$$
$$C(G, \pi_k) = \{\{(u, v) \in C(G, \Pi)\} \mid (u \in \pi_k \vee v \in \pi_k)\} \tag{4}$$

- Our partitioning problem is then to minimize total edge cut $EC$ and max per-part edge cut $EC_{max}$:

$$EC(G, \Pi) = |C(G, \Pi)| \tag{5}$$
$$EC_{max}(G, \Pi) = \max_k |C(G, \pi_k)| \tag{6}$$

# Overview
## Partitioning - HPC Approaches

- (Par)METIS (Karypis et al.), PT-SCOTCH (Pellegrini et al.), Chaco (Hendrickson et al.), etc.
- Multilevel methods:
  - *Coarsen* the input graph in several iterative steps
  - At coarsest level, partition graph via local methods following balance constraints and quality objectives
  - Iteratively *uncoarsen* graph, refine partitioning
- **Problem 1**: Designed for traditional HPC scientific problems (e.g. meshes) – limited balance constraints and quality objectives
- **Problem 2**: Multilevel approach – high memory requirements, can run slowly and lack scalability

# Overview
## Label Propagation

- **Label propagation**: randomly initialize a graph with some $p$ labels, iteratively assign to each vertex the maximal per-label count over all neighbors to generate clusters (Raghavan et al. 2007)
  - Clustering algorithm - dense clusters hold same label
  - Fast - each iteration in $O(n+m)$, usually fixed iteration count (doesn't necessarily converge)
  - Naïvely parallel - only per-vertex label updates
  - *Observation*: Possible applications for large-scale small-world graph partitioning

# Overview
## Partitioning - "Big Data" Approaches

- Methods designed for small-world graphs (e.g. social networks and web graphs)
- Exploit label propagation/clustering for partitioning:
  - Multilevel methods - use label propagation to coarsen graph (Wang et al. 2014, Meyerhenke et al. 2014)
  - Single level methods - use label propagation to directly create partitioning (Ugander and Backstrom 2013, Vaquero et al. 2013)
- **Problem 1**: Multilevel methods still can lack scalability, might also require running traditional partitioner at coarsest level
- **Problem 2**: Single level methods can produce sub-optimal partition quality

**PuLP** : **P**artitioning **U**sing **L**abel **P**ropagation

- Utilize label propagation for:
  - Vertex balanced partitions, minimize edge cut (PuLP)
  - Vertex and edge balanced partitions, minimize edge cut (PuLP-M)
  - Vertex and edge balanced partitions, minimize edge cut and maximal per-part edge cut (PuLP-MM)
  - Any combination of the above - multi objective, multi constraint

# Algorithms
## Primary Algorithm Overview

- PuLP-MM Algorithm
  - Constraint 1: balance vertices, Constraint 2: balance edges
  - Objective 1: minimize edge cut, Objective 2: minimize per-partition edge cut
  - Pseudocode gives default iteration counts

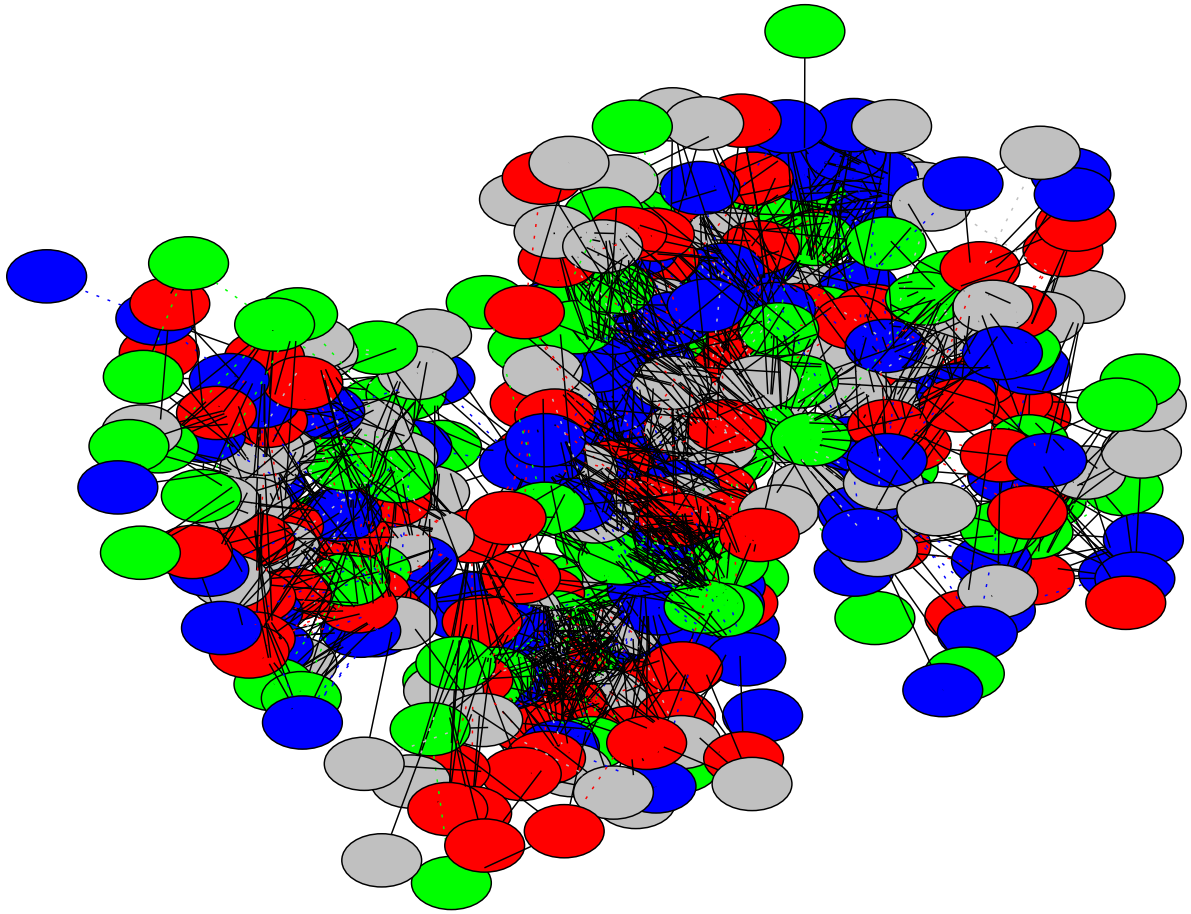
Initialize $p$ random partitions
Execute 3 iterations degree-weighted label propagation (LP)
**for** $k_1 = 1$ iterations **do**
  **for** $k_2 = 3$ iterations **do**
    Balance partitions with 5 LP iterations to satisfy constraint 1
    Refine partitions with 10 FM iterations to minimize objective 1
  **for** $k_3 = 3$ iterations **do**
    Balance partitions with 2 LP iterations to satisfy constraint 2 and minimize objective 2 with 5 FM iterations
    Refine partitions with 10 FM iterations to minimize objective 1

# Algorithms
## Primary Algorithm Overview

Initialize $p$ random partitions
Execute degree-weighted label propagation (LP)
**for** $k_1$ iterations **do**
    **for** $k_2$ iterations **do**
        Balance partitions with LP to satisfy vertex constraint
        Refine partitions with FM to minimize edge cut
    **for** $k_3$ iterations **do**
        Balance partitions with LP to satisfy edge constraint and minimize max per-part cut
        Refine partitions with FM to minimize edge cut

# Algorithms
## Primary Algorithm Overview

Randomly initialize $p$ partitions ($p = 4$)



Network shown is the Infectious network dataset from KONECT (http://konect.uni-koblenz.de/)

# Algorithms
## Primary Algorithm Overview

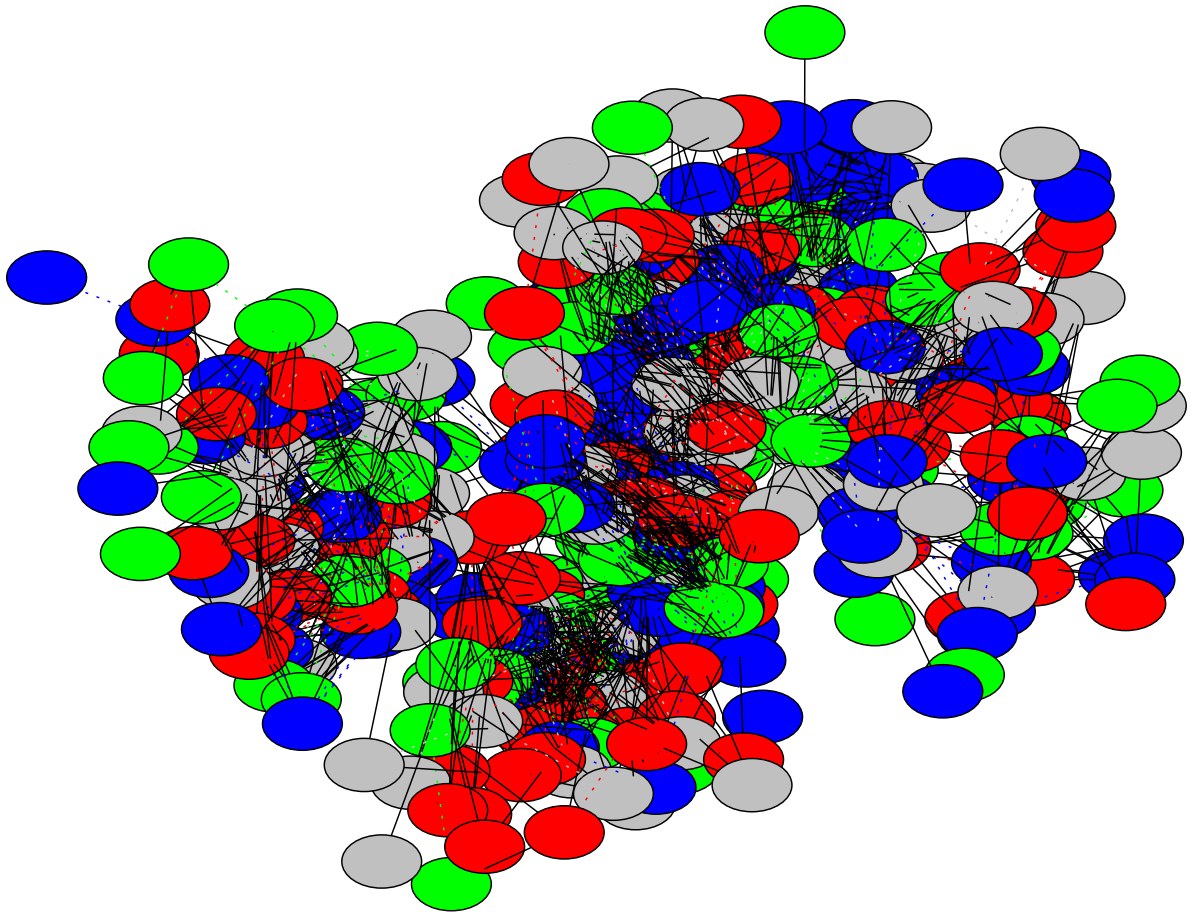
- After random initialization, we then perform label propagation to create partitions
- **Initial Observations**:
  - Partitions are unbalanced, for high $p$, some partitions end up empty
  - Edge cut is good, but can be better
- **PuLP Solutions**:
  - Impose loose balance constraints, explicitly refine later
  - Degree weightings - cluster around high degree vertices, let low degree vertices form boundary between partitions

# Algorithms
## Primary Algorithm Overview

Initialize $p$ random partitions
Execute degree-weighted label propagation (LP)
**for** $k_1$ iterations **do**
    **for** $k_2$ iterations **do**
        Balance partitions with LP to satisfy vertex constraint
        Refine partitions with FM to minimize edge cut
    **for** $k_3$ iterations **do**
        Balance partitions with LP to satisfy edge constraint and minimize max per-part cut
        Refine partitions with FM to minimize edge cut

# Algorithms
## Primary Algorithm Overview

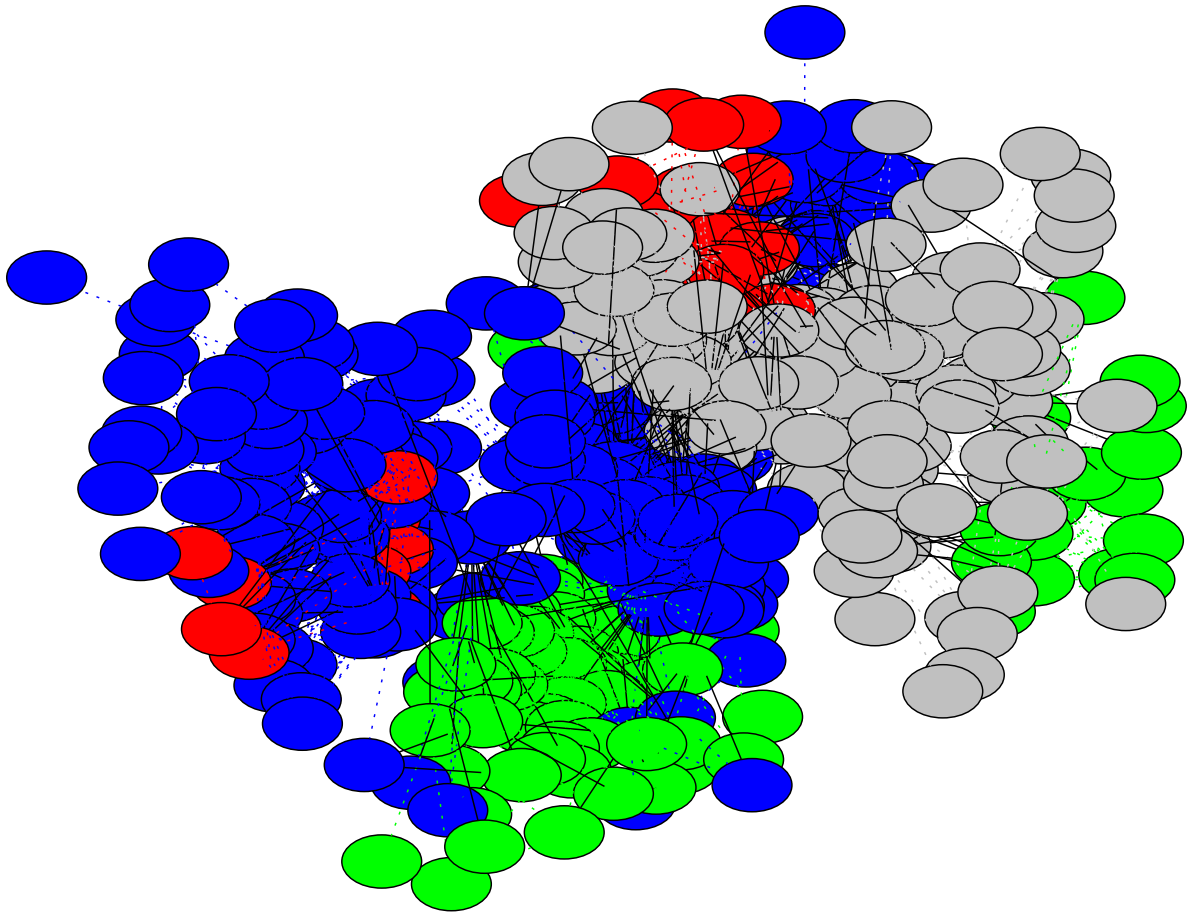
Part assignment after random initialization.



Network shown is the Infectious network dataset from KONECT (http://konect.uni-koblenz.de/)

# Algorithms
## Primary Algorithm Overview

Part assignment after degree-weighted label propagation.



Network shown is the Infectious network dataset from KONECT (http://konect.uni-koblenz.de/)

# Algorithms
## Primary Algorithm Overview

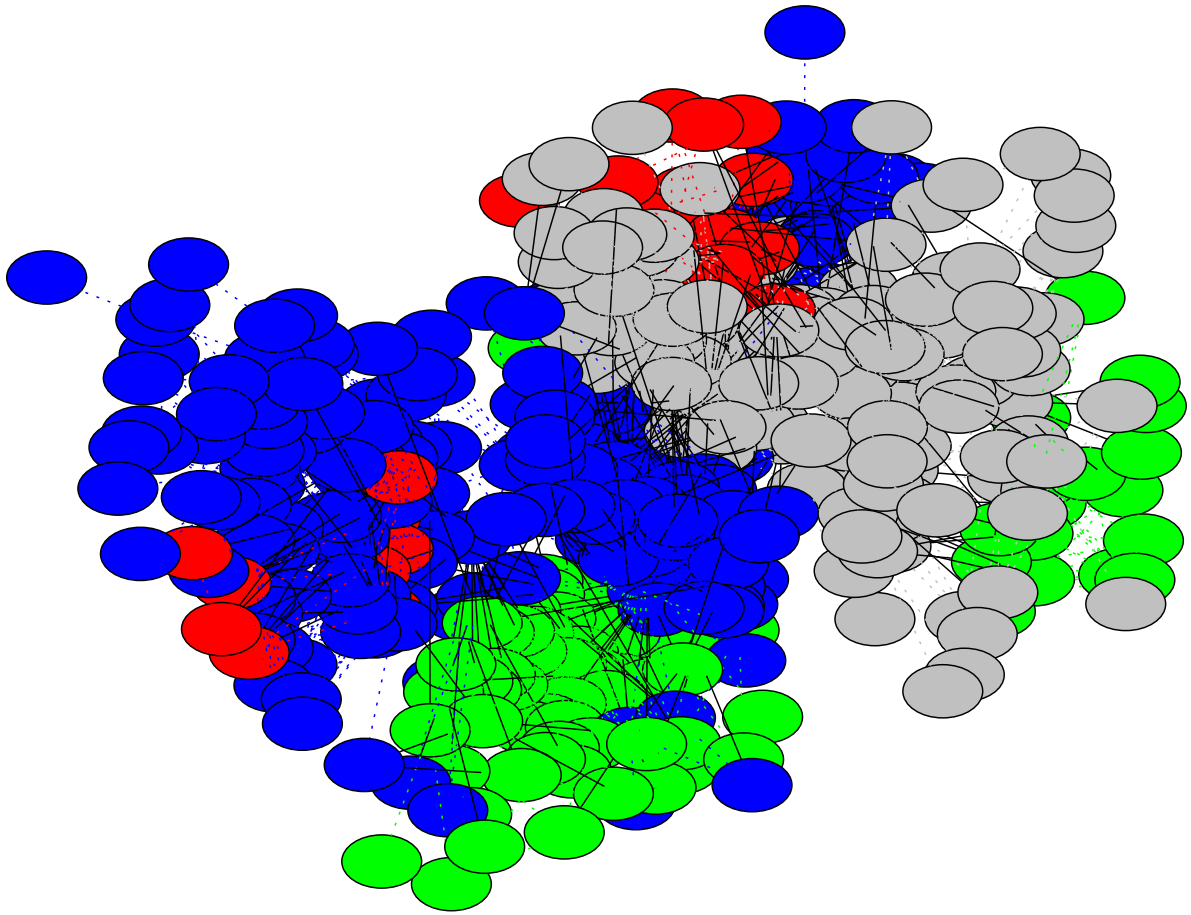
- After label propagation, we balance vertices among partitions and minimize edge cut (baseline PuLP ends here)
- **Observations**:
  - Partitions are still unbalanced in terms of edges
  - Edge cut is good, max per-part cut isn't necessarily
- **PuLP-M and PuLP-MM Solutions**:
  - Maintain vertex balance while explicitly balancing edges
  - Alternate between minimizing total edge cut and max per-part cut (for PuLP-MM, PuLP-M only minimizes total edge cut)

# Algorithms
## Primary Algorithm Overview

Initialize $p$ random partitions
Execute degree-weighted label propagation (LP)
**for** $k_1$ iterations **do**
    **for** $k_2$ iterations **do**
        Balance partitions with LP to satisfy vertex constraint
        Refine partitions with FM to minimize edge cut
    **for** $k_3$ iterations **do**
        Balance partitions with LP to satisfy edge constraint and minimize max per-part cut
        Refine partitions with FM to minimize edge cut

# Algorithms
## Primary Algorithm Overview

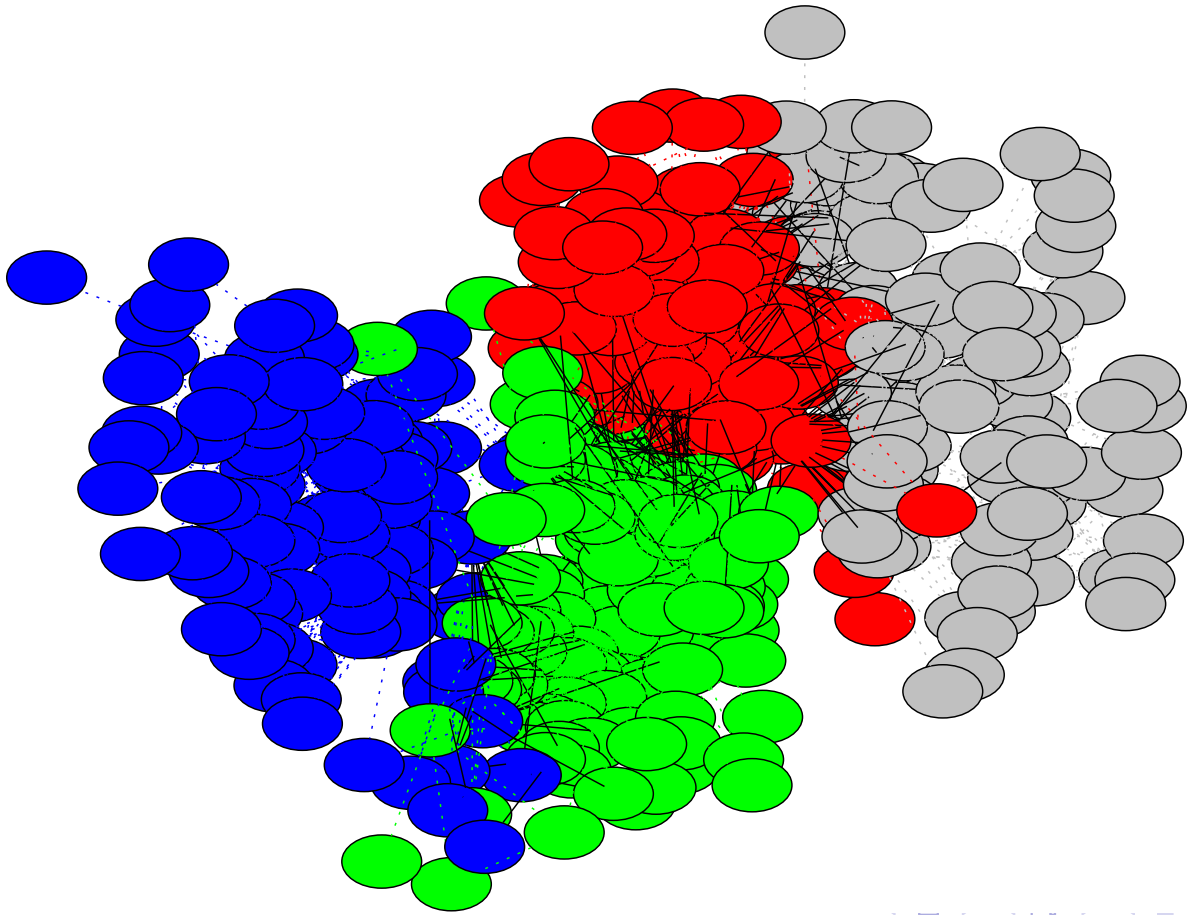
Part assignment after degree-weighted label propagation.



Network shown is the Infectious network dataset from KONECT (http://konect.uni-koblenz.de/)

# Algorithms
## Primary Algorithm Overview

Part assignment after balancing for vertices and minimizing edge cut.



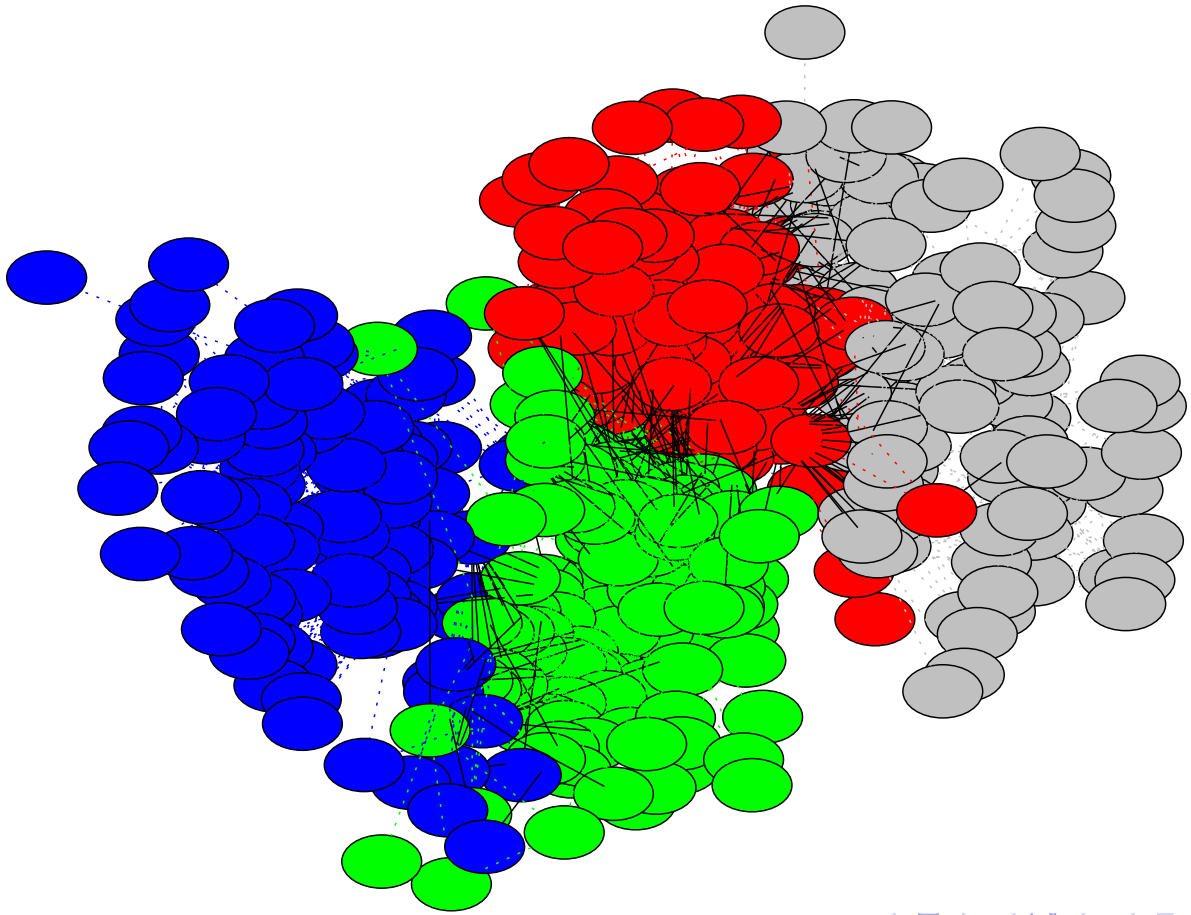
Network shown is the Infectious network dataset from KONECT (http://konect.uni-koblenz.de/)

# Algorithms
## Primary Algorithm Overview

Initialize $p$ random partitions
Execute degree-weighted label propagation (LP)
**for** $k_1$ iterations **do**
    **for** $k_2$ iterations **do**
        Balance partitions with LP to satisfy vertex constraint
        Refine partitions with FM to minimize edge cut
    **for** $k_3$ iterations **do**
        Balance partitions with LP to satisfy edge constraint and minimize max per-part cut
        Refine partitions with FM to minimize edge cut

# Algorithms
## Primary Algorithm Overview

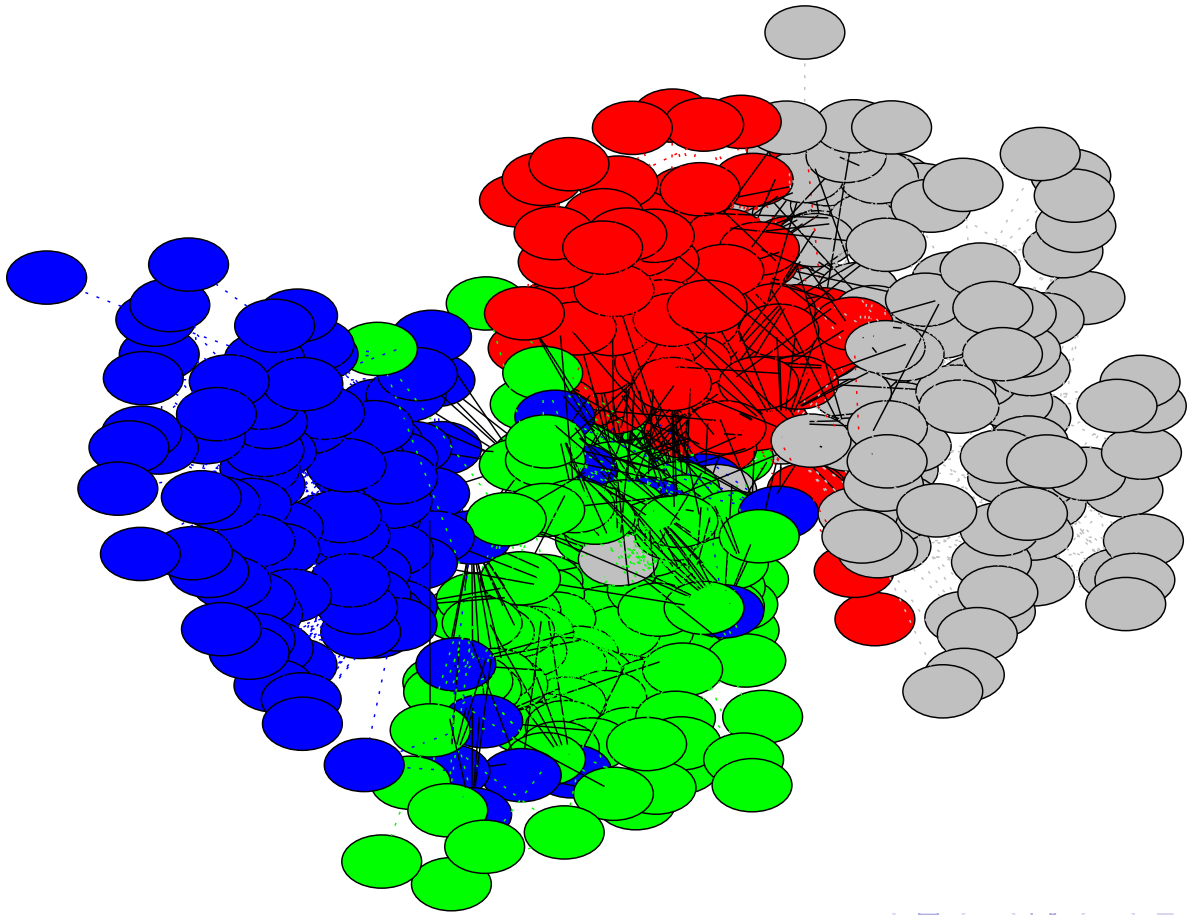
Part assignment after balancing for vertices and minimizing edge cut.



Network shown is the Infectious network dataset from KONECT (http://konect.uni-koblenz.de/)

# Algorithms
## Primary Algorithm Overview

Part assignment after balancing for edges and minimizing total edge cut and max per-part edge cut



Network shown is the Infectious network dataset from KONECT (http://konect.uni-koblenz.de/)

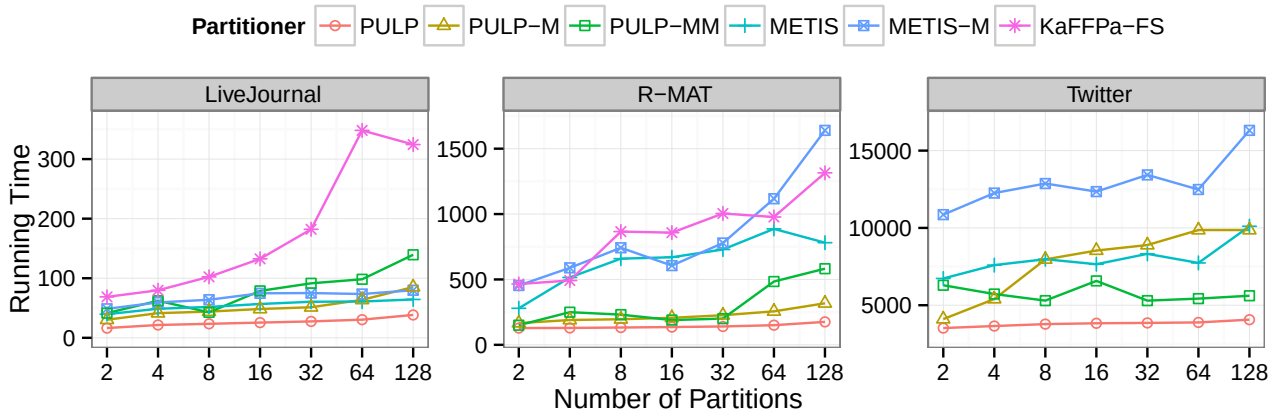
# Results
## Test Environment and Graphs

- Test system: *Compton*
  - Intel Xeon E5-2670 (Sandy Bridge), dual-socket, 16 cores, 64 GB memory.
- Test graphs:
  - LAW graphs from UF Sparse Matrix, SNAP, MPI, Koblenz
  - Real (one R-MAT), small-world, 60 K–70 M vertices, 275 K–2 B edges
- Test Algorithms:
  - **METIS** - single constraint single objective
  - **METIS-M** - multi constraint single objective
  - **ParMETIS** - METIS-M running in parallel
  - **KaFFPa** - single constraint single objective
  - **PuLP** - single constraint single objective
  - **PuLP-M** - multi constraint single objective
  - **PuLP-MM** - multi constraint multi objective
- Metrics: 2–128 partitions, serial and parallel running times, memory utilization, edge cut, max per-partition edge cut

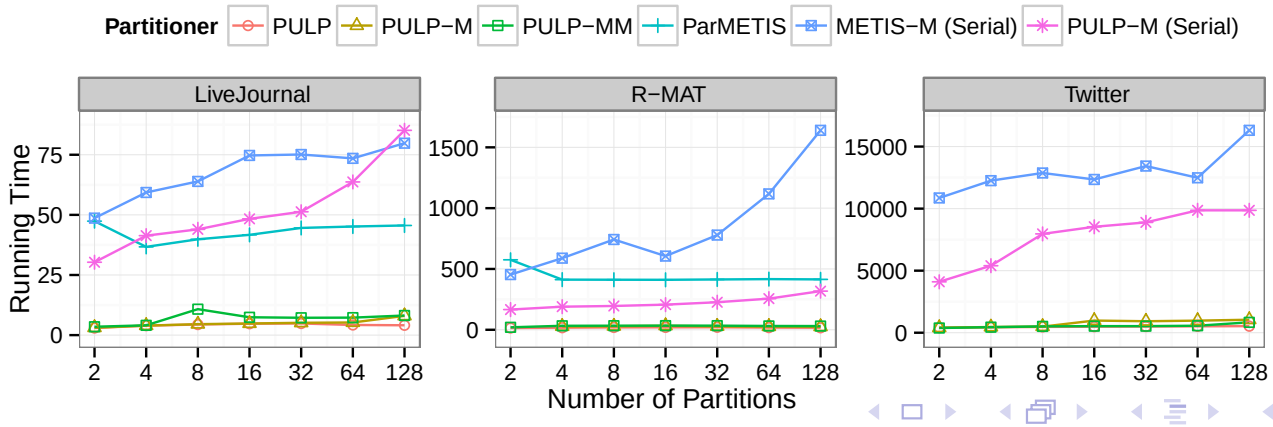
# Results

## Running Times - Serial (top), Parallel (bottom)

- In serial, PuLP-MM runs 1.7× faster (geometric mean) than next fastest




- In parallel, PuLP-MM runs 14.5× faster (geometric mean) than next fastest (ParMETIS times are fastest of 1 to 256 cores)



# Results

## Memory utilization for 128 partitions

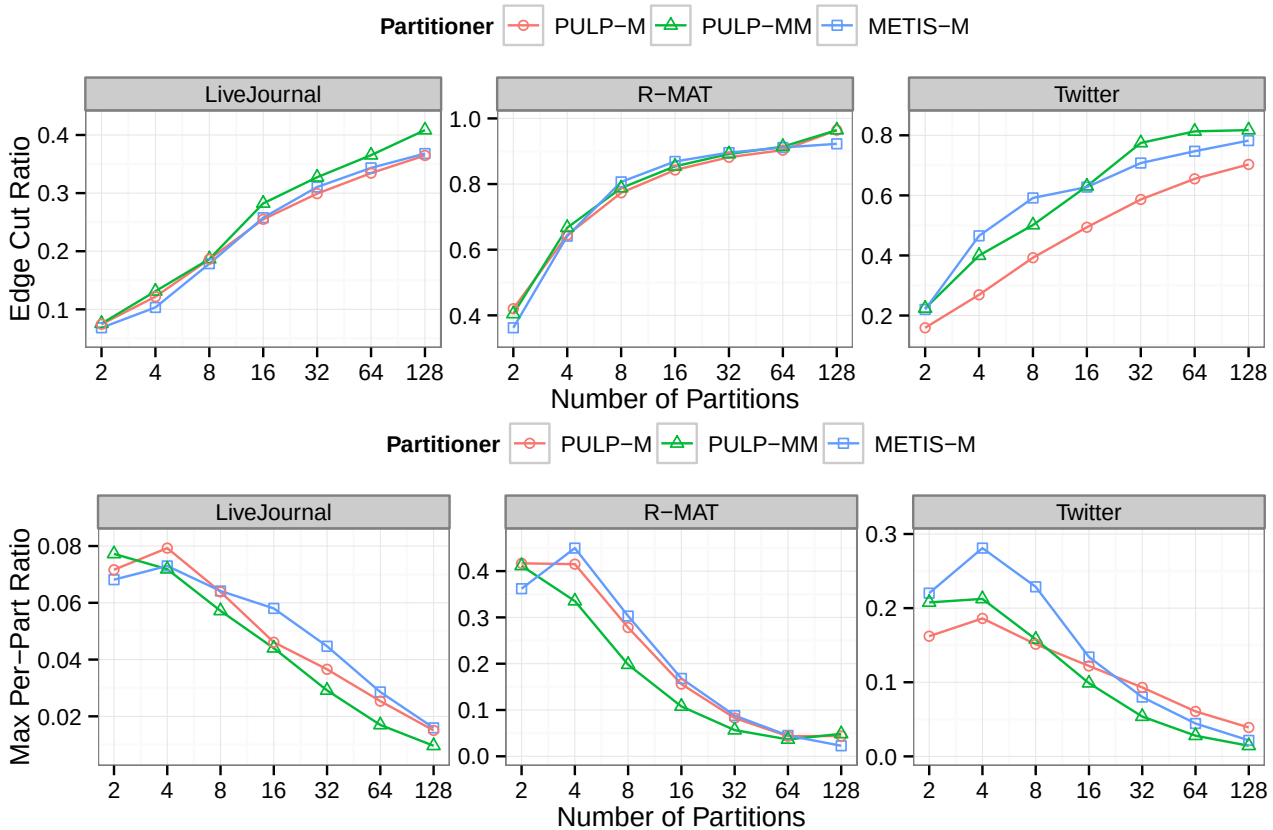
- PuLP utilizes minimal memory, $O(n)$, 8-39× less than other partitioners
- Savings are mostly from avoiding a multilevel approach

| Network | Memory Utilization | | | | **Improv.** |
|---|---|---|---|---|---|
| | METIS-M | KaFFPa | PuLP-MM | Graph Size | |
| LiveJournal | 7.2 GB | 5.0 GB | 0.44 GB | 0.33 GB | 21× |
| Orkut | 21 GB | 13 GB | 0.99 GB | 0.88 GB | 23× |
| R-MAT | 42 GB | - | 1.2 GB | 1.02 GB | 35× |
| DBpedia | 46 GB | - | 2.8 GB | 1.6 GB | 28× |
| WikiLinks | 103 GB | 42 GB | 5.3 GB | 4.1 GB | 25× |
| sk-2005 | 121 GB | - | 16 GB | 13.7 GB | 8× |
| Twitter | 487 GB | - | 14 GB | 12.2 GB | 39× |

# Results

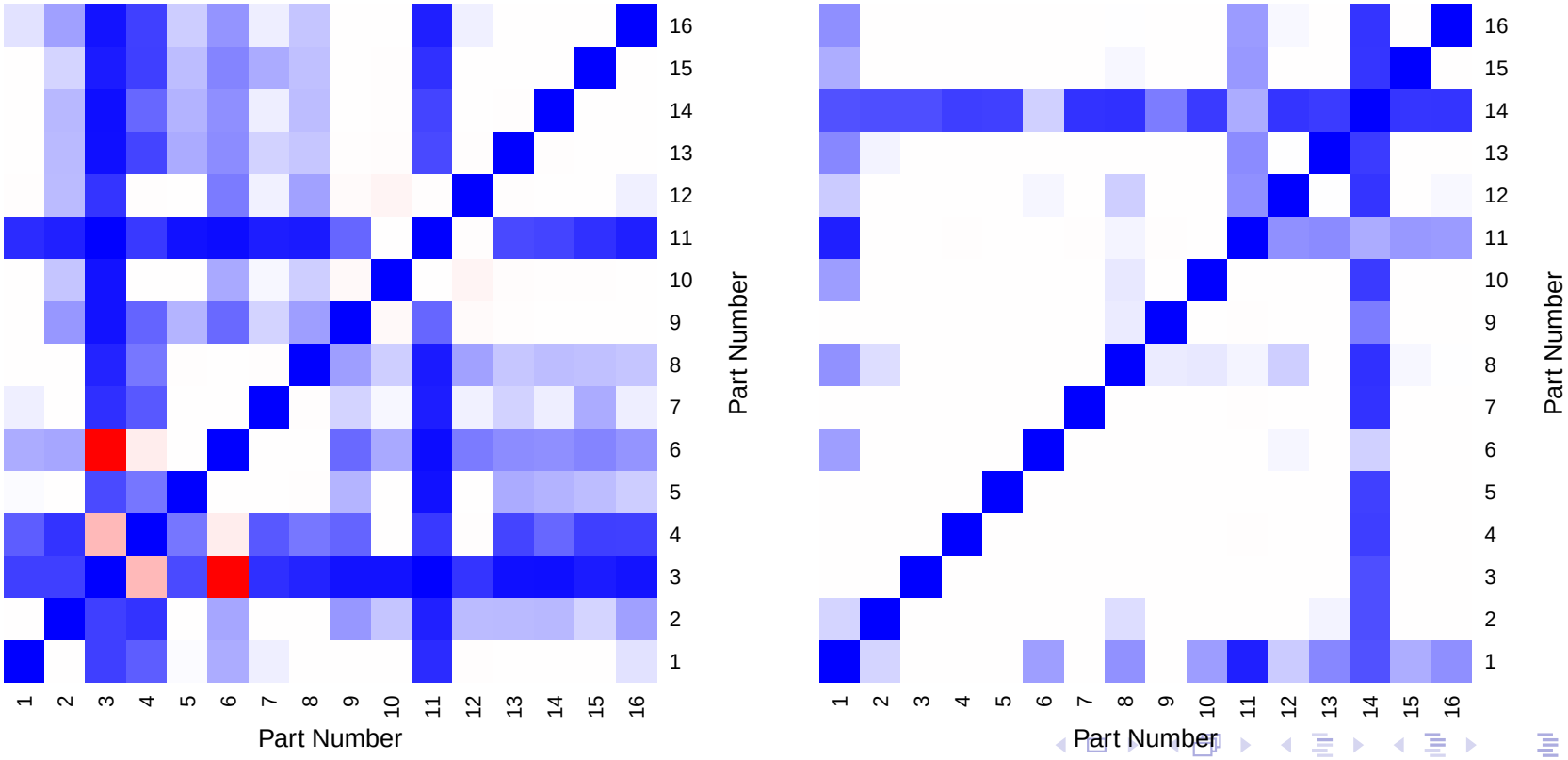
## Performance - Edge Cut and Edge Cut Max

- PuLP-M produces better edge cut than METIS-M over most graphs
- PuLP-MM produces better max edge cut than METIS-M over most graphs



# Results
## Balanced communication

- uk-2005 graph from LAW, METIS-M (left) vs. PuLP-MM (right)
- Blue: low comm; White: avg comm; Red: High comm
- PuLP reduces max inter-part communication requirements and balances total communication load through all tasks

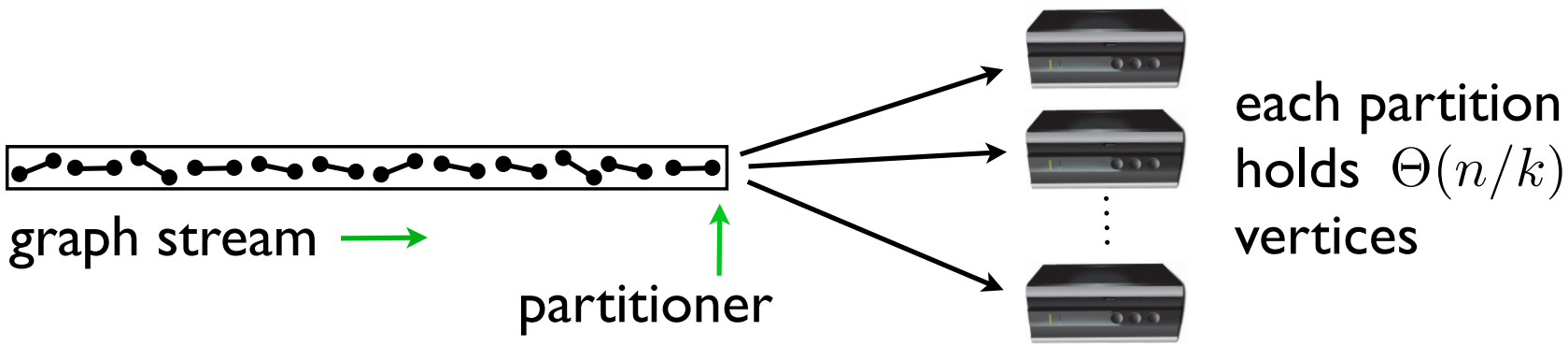


## Streaming Partitioning (FENNEL)

*Slides from Tsourakakis et al., Aalto University and MSR-UK*

# streaming $k$-way graph partitioning

- input is a data stream
- graph is ordered
  - arbitrarily
  - breadth-first search
  - depth-first search
- generate an approximately balanced graph partitioning



# Graph representations

- incidence stream
  - at time $t$, a vertex arrives with its neighbors
- adjacency stream
  - at time $t$, an edge arrives

# Partitioning strategies

- hashing: place a new vertex to a cluster/machine chosen uniformly at random
- neighbors heuristic: place a new vertex to the cluster/machine with the maximum number of neighbors
- non-neighbors heuristic: place a new vertex to the cluster/machine with the minimum number of non-neighbors

# Partitioning strategies

[Stanton and Kliot, 2012]

- $d_c(v)$: neighbors of $v$ in cluster $c$
- $t_c(v)$: number of triangles that $v$ participates in cluster $c$

- balanced: vertex v goes to cluster with least number of vertices
- hashing: random assignment
- weighted degree: $v$ goes to cluster $c$ that maximizes $d_c(v) \cdot w(c)$
- weighted triangles: $v$ goes to cluster $j$ that maximizes $t_c(v)/\binom{d_c(v)}{2} \cdot w(c)$

# Weight functions

- $s_c$: number of vertices in cluster $c$
- unweighted: $w(c) = 1$
- linearly weighted: $w(c) = 1 - s_c(k/n)$
- exponentially weighted: $w(c) = 1 - e^{(s_c - n/k)}$

# FENNEL algorithm

The standard formulation hits the ARV barrier

$$\begin{aligned} \text{minimize}_{\ \mathcal{P}=(S_1,\ldots,S_k)} \quad & |\partial\, e(\mathcal{P})| \\ \text{subject to} \quad & |S_i| \leq \nu \frac{n}{k}, \ \text{for all } 1 \leq i \leq k \end{aligned}$$

- We relax the hard cardinality constraints

$$\text{minimize}_{\ \mathcal{P}=(S_1,\ldots,S_k)} \quad |\partial\, E(\mathcal{P})| + c_{\text{IN}}(\mathcal{P})$$

where $c_{\text{IN}}(\mathcal{P}) = \sum_i s(|S_i|)$, so that objective self-balances

# FENNEL algorithm

- for $S \subseteq V$, $f(S) = e[S] - \alpha|S|^{\gamma}$, with $\gamma \geq 1$
- given partition $\mathcal{P} = (S_1, \ldots, S_k)$ of $V$ in $k$ parts define

$$g(\mathcal{P}) = f(S_1) + \ldots + f(S_k)$$

- the goal: maximize $g(\mathcal{P})$ over all possible $k$-partitions
- notice:

$$g(\mathcal{P}) = \underbrace{\sum_i e[S_i]}_{\substack{m-\text{number of} \\ \text{edges cut}}} - \underbrace{\alpha \sum_i |S_i|^{\gamma}}_{\substack{\text{minimized for} \\ \text{balanced partition!}}}$$

# Connection

notice

$$f(S) = e[S] - \alpha \binom{|S|}{2}$$

- related to modularity
- related to optimal quasicliques [Tsourakakis et al., 2013]

# FENNEL algorithm

## Theorem

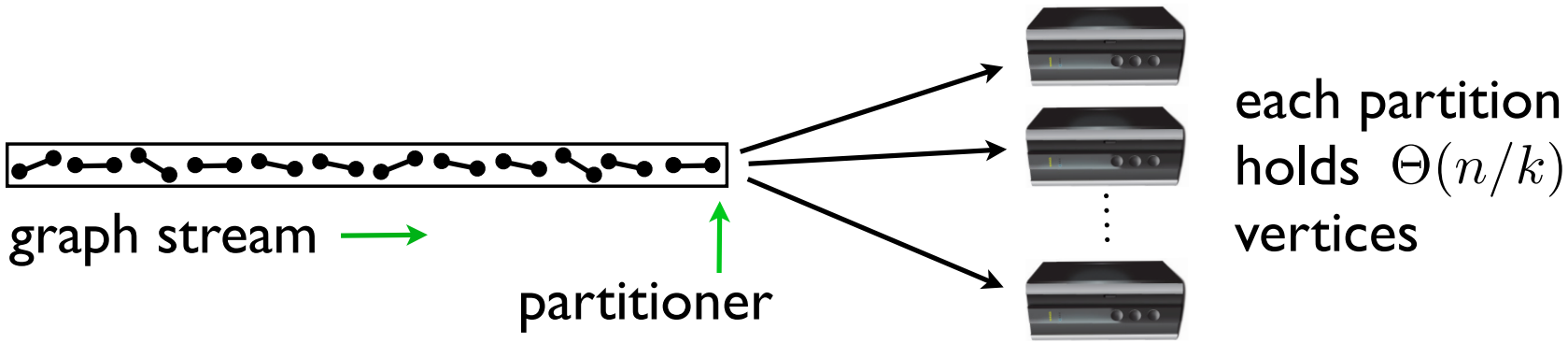
- For $\gamma = 2$ there exists an algorithm that achieves an approximation factor $\log(k)/k$
  for a shifted objective where $k$ is the number of clusters
  - semidefinite programming algorithm
  - in the shifted objective the main term takes care of the load balancing and the second order term minimizes the number of edges cut
  - Multiplicative guarantees not the most appropriate
- random partitioning gives approximation factor $1/k$
- no dependence on $n$

  mainly because of relaxing the hard cardinality constraints

# FENNEL algorithm — greedy scheme

- $\gamma = 2$ gives non-neighbors heuristic
- $\gamma = 1$ gives neighbors heuristic
- interpolate between the two heuristics, e.g., $\gamma = 1.5$

# FENNEL algorithm — greedy scheme



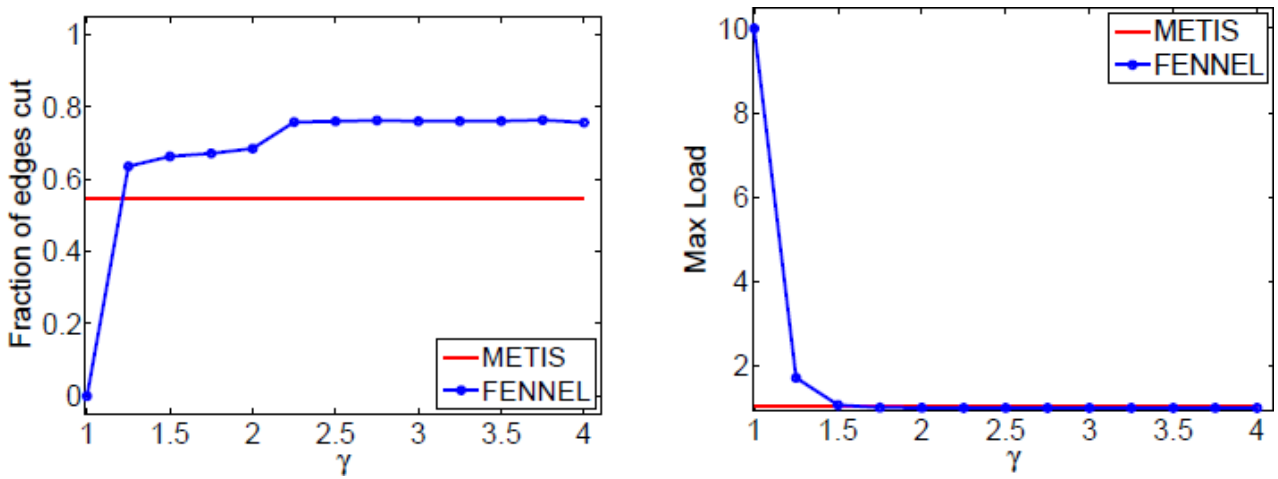

- send $v$ to the partition / machine that maximizes

$$
\begin{aligned}
& f(S_i \cup \{v\}) - f(S_i) \\
& \qquad = e[S_i \cup \{v\}] - \alpha(|S_i| + 1)^\gamma - (e[S_i] - \alpha|S_i|^\gamma) \\
& \qquad = d_{S_i}(v) - \alpha\mathcal{O}(|S_i|^{\gamma-1})
\end{aligned}
$$

- fast, amenable to streaming and distributed setting

# FENNEL algorithm — $\gamma$

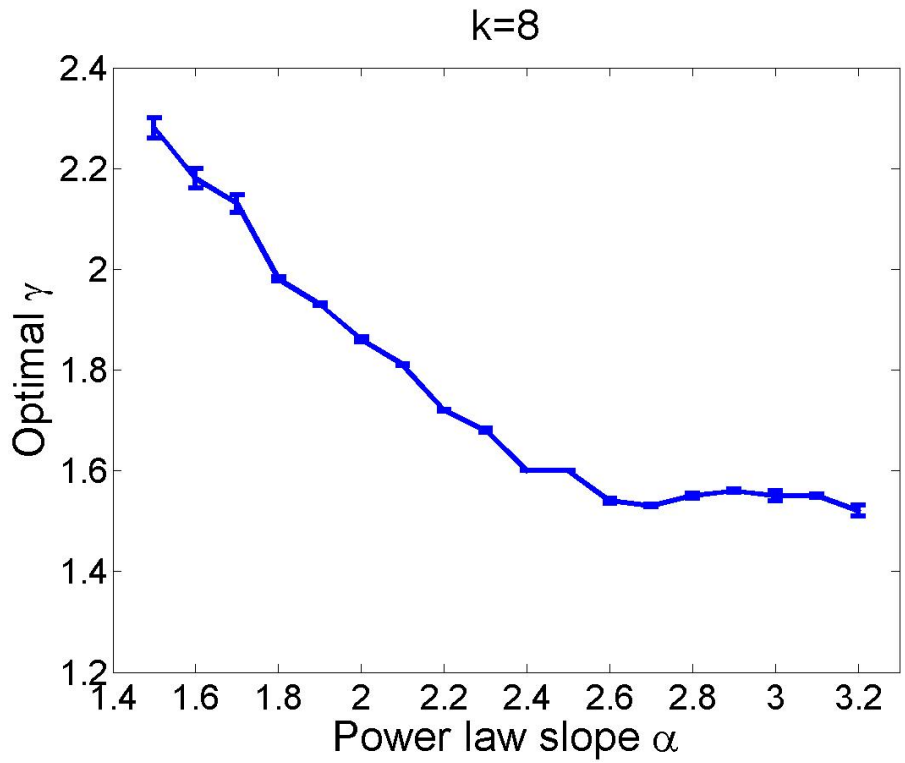
Explore the tradeoff between the number of edges cut and load balancing.




Fraction of edges cut $\lambda$ and maximum load normalized $\rho$ as a function of $\gamma$, ranging from 1 to 4 with a step of 0.25, over five randomly generated power law graphs with slope 2.5. The straight lines show the performance of METIS.

- Not the end of the story ... choose $\gamma^*$ based on some "easy-to-compute" graph characteristic.

# FENNEL algorithm — $\gamma^*$




y-axis Average optimal value $\gamma^*$ for each power law slope in the range $[1.5, 3.2]$ using a step of 0.1 over twenty randomly generated power law graphs that results in the smallest possible fraction of edges cut $\lambda$ conditioning on a maximum normalized load $\rho = 1.2$, $k = 8$. x-axis Power-law exponent of the degree sequence. Error bars indicate the variance around the average optimal value $\gamma^*$.

# FENNEL algorithm — results

Twitter graph with approximately 1.5 billion edges, $\gamma = 1.5$

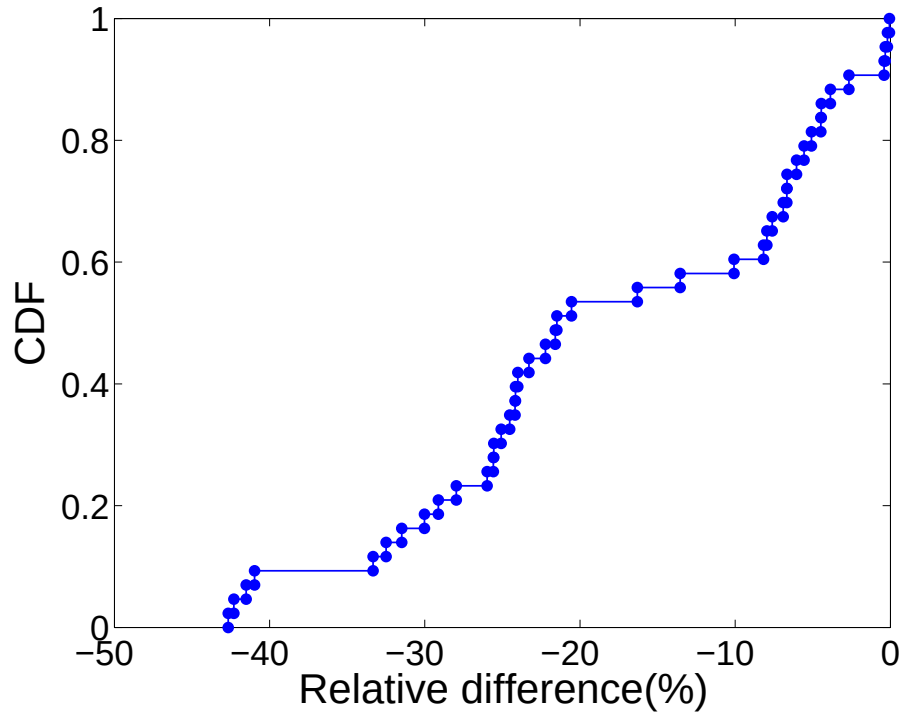
$$\lambda = \frac{\#\{\text{edges cut}\}}{m} \qquad \rho = \max_{1 \le i \le k} \frac{|S_i|}{n/k}$$

| | Fennel | | Best competitor | | Hash Partition | | METIS | |
|---|---|---|---|---|---|---|---|---|
| $k$ | $\lambda$ | $\rho$ | $\lambda$ | $\rho$ | $\lambda$ | $\rho$ | $\lambda$ | $\rho$ |
| 2 | 6.8% | 1.1 | 34.3% | 1.04 | 50% | 1 | 11.98% | 1.02 |
| 4 | 29% | 1.1 | 55.0% | 1.07 | 75% | 1 | 24.39% | 1.03 |
| 8 | 48% | 1.1 | 66.4% | 1.10 | 87.5% | 1 | 35.96% | 1.03 |

Table: Fraction of edges cut $\lambda$ and the normalized maximum load $\rho$ for Fennel, the best competitor and hash partitioning of vertices for the Twitter graph. Fennel and best competitor require around 40 minutes, METIS more than $8\frac{1}{2}$ hours.

# FENNEL algorithm — results

Extensive experimental evaluation over $>$ 40 large real graphs [Tsourakakis et al., 2012]




CDF of the relative difference $\frac{\lambda_{fennel}-\lambda_c}{\lambda_c} \times 100\%$ of percentages of edges cut of FENNEL and the best competitor (pointwise) for all graphs in our dataset.

# FENNEL algorithm — "zooming in"

Performance of various existing methods on amazon0312 for $k = 32$

| | BFS | | Random | |
|---|---|---|---|---|
| Method | $\lambda$ | $\rho$ | $\lambda$ | $\rho$ |
| H | 96.9% | 1.01 | 96.9% | 1.01 |
| B [Stanton and Kliot, 2012] | 97.3% | 1.00 | 96.8% | 1.00 |
| DG [Stanton and Kliot, 2012] | 0% | 32 | 43% | 1.48 |
| LDG [Stanton and Kliot, 2012] | 34% | 1.01 | 40% | 1.00 |
| EDG [Stanton and Kliot, 2012] | 39% | 1.04 | 48% | 1.01 |
| T [Stanton and Kliot, 2012] | 61% | 2.11 | 78% | 1.01 |
| LT [Stanton and Kliot, 2012] | 63% | 1.23 | 78% | 1.10 |
| ET [Stanton and Kliot, 2012] | 64% | 1.05 | 79% | 1.01 |
| NN [Prabhakaran and et al., 2012] | 69% | 1.00 | 55% | 1.03 |
| Fennel | 14% | 1.10 | 14% | 1.02 |
| METIS | 8% | 1.00 | 8% | 1.02 |

# Today's Biz

1. Reminders
2. Review
3. Graph Partitioning overview
4. Graph Partitioning of Small-world Graphs
5. **Partitioning Usage example**

**Graph Partitioning**
**Blank code and data available on website**
**(Lecture 8)**
www.cs.rpi.edu/~slotag/classes/FA16/index.html