hMetis

Contributors: Nikolaos Sketopoulos, Christos Georgakidis, Christos Sotiriou

1

Outline

- hMetis Overview
  - Quick Overview
  - Hypergraph Input File Format
  - Output File Format
- hMetis as Clustering Tool
  - Using hMetis
  - Using k-way hMetis
- Comparing ASP with hMetis

2

hMetis Overview
Quick Overview (1)

- hMetis is a tool for partitioning large hypergraphs, especially those arising in circuit design.
- The algorithms in hMetis are based on multilevel partitioning algorithms which are extensions of traditional graph partitioning algorithms. (Fig. 1)

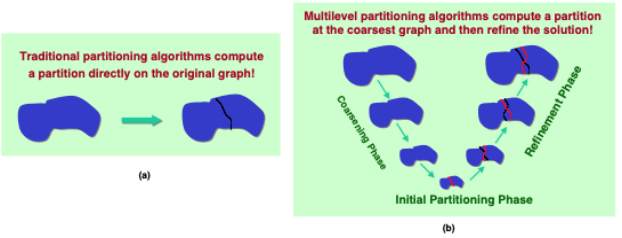


Figure 1: (a) Traditional partitioning algorithms. (b) Multilevel partitioning algorithms.

3

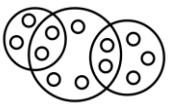
hMetis Overview
Quick Overview (2)

- *Coarsening Phase*:
  - A sequence of successively smaller hypergraphs is constructed
  - Its purpose is to create a small hypergraph, such that there is no significant difference in the quality of a bisection performed on the small hypergraph compared to the original one.
  - Successively reduces the size of the hyperedges -> helpful for refinement KL/FM-based heuristics, which are ineffective in refining hyperedges with a large number of vertices belonging to different partitions.
  - Coarsening schemes:
    - Hybrid First-Choice scheme (HFC)
    - First-Choice scheme (FC)
    - Greedy First-Choice scheme (GFC)
    - HyperEdge Coarsening scheme (HEC)
    - Edge Coarsening scheme (EC)

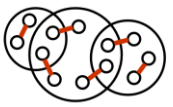
4

hMetis Overview
Quick Overview (3)

- Edge Coarsening (*EC*):
  - Steps:
    - The vertices are visited in random order
    - For each vertex v, all unmatched vertices that belong to hyperedges incident to v are considered, and the one that is connected via the edge with the largest weight is matched with v.
    - The weight of an edge connecting two vertices v and u is computed as the sum of the *edge weights* of all the hyperedges that contain v and u.
    - Each hyperedge e of size $|e|$ is assigned an edge-weight of
      $$w_e = 1/(|e| - 1)$$
      - as hyperedges collapse on each other during the coarsening, their edge weights are added up accordingly.



Original Graph



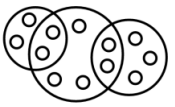
Edge Coarsening

hMetis Overview
Quick Overview (4)

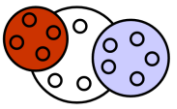
- Edge Coarsening (*EC*):
  - In this scheme, each group consists of at most two vertices (some vertices are not combined at all), and each vertex belongs to exactly one group.
  - *EC* scheme can produce successively coarser hypergraphs
    - However, it decreases the hyperedge weight of the coarser graph only for those pairs of matched vertices that are connected via a hyperedge of size two.
  - The total hyperedge weight of successively coarser graphs does not decrease very fast.
  - *To ensure that for every group of vertices that are contracted together, there is a decrease in the hyperedge weight in the coarser graph, each such group of vertices must be connected by a hyperedge.*
    - → *Hyperedge Coarsening Scheme (HEC)*

hMetis Overview
Quick Overview (5)

- Hyperedge Coarsening (*HEC*):
  - Steps:
    - The hyperedges are initially sorted in a non-increasing hyperedge-weight order and the hyperedges of the same weight are sorted in a non-decreasing hyperedge size order.
    - Then, the hyperedges are visited in that order, and for each hyperedge that connect vertices that have not yet been matched, the vertices are matched together.
    - Thus, this scheme gives preference to the hyperedges that have large weight and those that are of small size.
    - After all of the hyperedges have been visited, the groups of vertices that have been matched are contracted together to form the next level coarser graph.
    - The vertices that are not part of any contracted hyperedges are comply copied to the next level coarser graph.



Original Graph



Hyperedge Coarsening

7

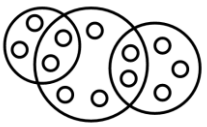
hMetis Overview
Quick Overview (6)

- Hyperedge Coarsening (*HEC*):
  - In this scheme, each group can have an arbitrary number of vertices (even though preference is given to smaller groups), and each vertex also belongs to exactly one group.
  - *HEC* can significantly reduce the amount of hyperedge weight that is left exposed in successively coarser graphs
  - However, during each coarsening phase, a majority of the hyperedges do not get contracted because vertices that belong to them have been contracted via other hyperedges.
    - The size of many hyperedges does not decrease sufficiently, making FM-based refinement difficult.
    - The weight of the vertices (i.e. the number of vertices that have been collapsed together) is successively coarser graphs becomes significantly different, which distorts that shape of the contracted hypergraph.
  - *Modified HyperEdge Coarsening scheme (MHEC)*

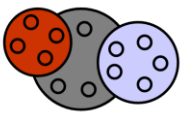
8

hMetis Overview
Quick Overview (7)

- Modified Hyperedge Coarsening (*MHEC*):
  - Steps:
    - *HEC* coarsening scheme
    - The list of hyperedges is traversed again, and for each hyperedge that has not yet been contracted, the vertices that do not belong to another contracted hyperedge are contracted together



Original Graph

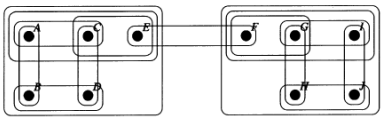


Modified Hyperedge Coarsening

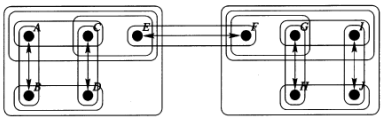
9

hMetis Overview
Quick Overview (7)

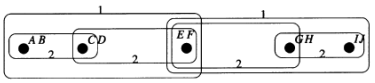
- Both *EC* and *HEC* schemes can potentially lead to less than ideal coarse representations of the original hypergraph
  - The *EC* and *HEC* will find as many groups of vertices as they can, that are pair- or hyperedge-wise independent.
    - The independence (and to a certain degree, the maximality) requirement may destroy some clusters of vertices that naturally exist in the hypergraph.



(a) Initial Hypergraph



(b) Groups Determined by Edge-Coarsening

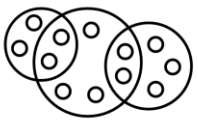


(c) Coarse Hypergraph

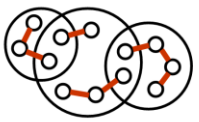
10

hMetis Overview
Quick Overview (8)

- First-Choice (*FC*):
  - Derived from the *EC* scheme by relaxing the requirement that a vertex is matched only with another unmatched vertex.
  - Steps:
    - The vertices are visited in a random order
    - However, for each vertex v, all vertices (both matched and unmatched) that belong to hyperedges incident to v are considered
    - The one that is connected via the edge with the largest weight is matched with v
      - Break ties in favor of unmatched vertices
    - Each group of vertices to be merged can contain an arbitrarily large number of vertices



Original Graph



First Choice

hMetis Overview
Quick Overview (9)

- First-Choice (*FC*):
  - Potential Problem:
    - The number of vertices in successive coarse graphs may decrease by a large factor
      - Potentially limiting the effect of refinement
  - At each coarsening level:
    - We stop the *FC* scheme as soon as the size of the resulting coarse graph has been reduced by a factor of **1.7**

hMetis Overview
Quick Overview (10)

- Greedy First-Choice (*GFC*):
  - Vertices are grouped based on the *FC* scheme, **BUT** the grouping is biased in favor of faster reduction in the number of the hyperedges that remain in the coarse hypergraphs.
- Hybrid First-Choice (*HFC*):
  - Combination of the *FC* and *GFC*

13

hMetis Overview
Quick Overview (11)

- *Initial Partitioning Phase*:
  - A bisection of the coarsened hypergraph is computed
  - Since this hypergraph has a very small number of vertices (< 100) many algorithms can be used.
  - hMetis uses multiple random bisections followed by the FM refinement algorithm

14

hMetis Overview
Quick Overview (12)

- *Uncoarsening & Refinement Phase*:
  - The partitioning of the coarsest hypergraph is used to obtain a partitioning for the finer hypergraph.
  - This is done by successively projecting the partitioning to the next level finer hypergraph and using a partitioning refinement algorithm to reduce the cut and thus improve the quality of the partitioning.
  - FM-based refinement schemes:
    - Fiduccia-Mattheyses (FM) scheme
    - One-way FM : during each iteration of FM, vertices are allowed to move only in a single direction
    - Early-exit FM : the FM iteration is aborted if the quality of the solution doesn't improve after a relatively small number of vertex moves.

15

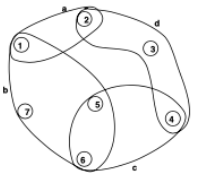
hMetis Overview
Hypergraph Input File Format (1)

- The primary input of hMetis is the hypergraph to be partitioned.
- 3 types of hypergraphs are supported:
  1. Unweighted Hypergraphs
  2. Hypergraphs with Weighted Hyperedges
  3. Hypergraphs with Weighted Vertices
  4. Hypergraphs with Weighted Hyperedges and Vertices
- Any line that starts with '%' is a comment line and is skipped.

17

hMetis Overview
Hypergraph Input File Format (2)

- Unweighted Hypergraph:
  - A plain text with $|E^h| + 1$ lines
  - The first line specifies the number of hyperedges and the number of nodes
  - After that, each line i specifies the nodes that consist the (i-1) hyperedge
    - E.g. hyperedge 0 consists of nodes 1 and 2
  - The number of the lines after the first one is equal to the number of hyperedges ($|E^h|$)



GraphFile

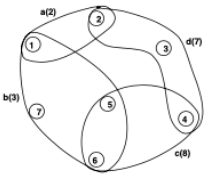
```
4 7
1 2
1 7 5 6
5 6 4
2 3 4
```

(a)

18

hMetis Overview
Hypergraph Input File Format (3)

- Hypergraph with Weighted Hyperedges:
  - A plain text with $|E^h| + 1$ lines
  - The first line specifies the number of hyperedges and the number of nodes and the type of weights (1 – weighted hyperedges)
  - After that, each line i specifies the nodes that consist the (i-1) hyperedge. The first integer of each line specifies the weight of the corresponding hyperedge
    - E.g. hyperedge 0 consists of nodes 1 and 2 and has a weight = 2
  - The number of the lines after the first one is equal to the number of hyperedges ($|E^h|$)



GraphFile

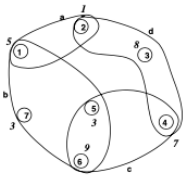
```
4 7 1
2 1 2
3 1 7 5 6
8 5 6 4
7 2 3 4
```

(b)

19

hMetis Overview
Hypergraph Input File Format (4)

- Hypergraph with Weighted Vertices:
  - A plain text with $|E^h| + |V| + 1$ lines
  - The first line specifies the number of hyperedges and the number of nodes and the type of weights (10 – weighted vertices)
  - After that, each line I (of the next $|E^h|$) specifies the nodes that consist the (i-1) hyperedge.
    - E.g. hyperedge 0 consists of nodes 1 and 2
  - The next |V| lines specify the weights for each node (in increasing order of their node index)
    - E.g. node 1 has weight = 5, node 2 has weight = 1, etc..

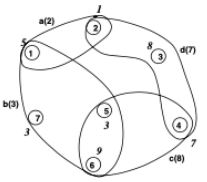


GraphFile
```
4 7 10
1 2
1 7 5 6
5 6 4
2 3 4
5
1
8
7
3
9
3
```
(c)

20

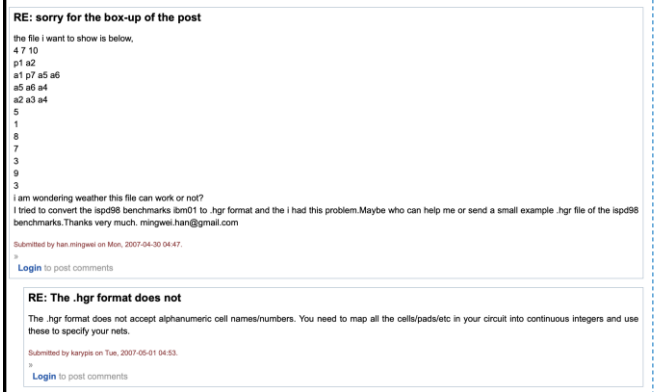
hMetis Overview
Hypergraph Input File Format (5)

- Hypergraph with Weighted Hyperedges and Vertices:
  - A plain text with $|E^h| + |V| + 1$ lines
  - The first line specifies the number of hyperedges and the number of nodes and the type of weights (11 – weighted hyperedges and vertices)
  - After that, each line I (of the next $|E^h|$) specifies the nodes that consist the (i-1) hyperedge. The first integer of each line specifies the weight of the corresponding hyperedge
    - E.g. hyperedge 0 consists of nodes 1 and 2 and has a weight = 2
  - The next |V| lines specify the weights for each node (in increasing order of their node index)
    - E.g. node 1 has weight = 5, node 2 has weight = 1, etc..



GraphFile
```
4 7 11
2 1 2
3 1 7 5 6
8 5 6 4
7 2 3 4
5
1
8
7
3
9
3
```
(d)

21


RE: sorry for the box-up of the post

the file i want to show is below,
4 7 10
p1 a2
a1 p7 a5 a6
a5 a6 a4
a2 a3 a4
5
1
8
7
3
9
3
i am wondering weather this file can work or not?
I tried to convert the ispd98 benchmarks ibm01 to .hgr format and the i had this problem.Maybe who can help me or send a small example .hgr file of the ispd98 benchmarks.Thanks very much. mingwei.han@gmail.com

Submitted by han.mingwei on Mon, 2007-04-30 04:47.

Login to post comments

RE: The .hgr format does not

The .hgr format does not accept alphanumeric cell names/numbers. You need to map all the cells/pads/etc in your circuit into continuous integers and use these to specify your nets.

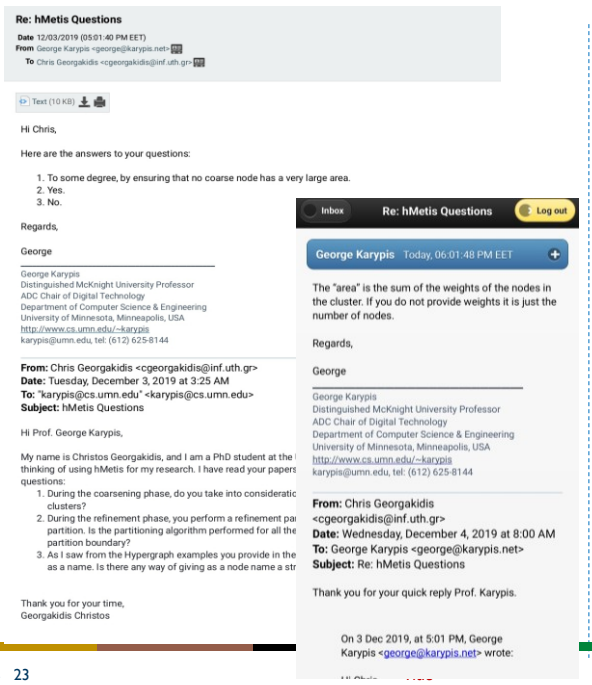
Submitted by karypis on Tue, 2007-05-01 04:53.

Login to post comments

hMetis Overview Hypergraph Input File Format (7)

The .hgr does not accept alphanumeric cell names/numbers. It is needed to map cells in the circuit to continuous integers use these to specify the nets

22


Re: hMetis Questions

Date 12/03/2019 (05:01:40 PM EET)
From George Karypis <george@karypis.net>
To Chris Georgakidis <cgeorgakidis@inf.uth.gr>

Text (10 KB)

Hi Chris,

Here are the answers to your questions:

1. To some degree, by ensuring that no coarse node has a very large area.
2. Yes.
3. No.

Regards,

George

George Karypis
Distinguished McKnight University Professor
ADC Chair of Digital Technology
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, USA
http://www.cs.umn.edu/~karypis
karypis@umn.edu, tel: (612) 625-8144

From: Chris Georgakidis <cgeorgakidis@inf.uth.gr>
Date: Tuesday, December 3, 2019 at 3:25 AM
To: "karypis@cs.umn.edu" <karypis@cs.umn.edu>
Subject: hMetis Questions

Hi Prof. George Karypis,

My name is Christos Georgakidis, and I am a PhD student at the
thinking of using hMetis for my research. I have read your papers
questions:

1. During the coarsening phase, do you take into consideratic clusters?
2. During the refinement phase, you perform a refinement pa partition. Is the partitioning algorithm performed for all the partition boundary?
3. As I saw from the Hypergraph examples you provide in the as a name. Is there any way of giving as a node name a str

Thank you for your time,
Georgakidis Christos

Inbox **Re: hMetis Questions** Log out

George Karypis Today, 06:01:48 PM EET

The "area" is the sum of the weights of the nodes in the cluster. If you do not provide weights it is just the number of nodes.

Regards,

George

George Karypis
Distinguished McKnight University Professor
ADC Chair of Digital Technology
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, USA
http://www.cs.umn.edu/~karypis
karypis@umn.edu, tel: (612) 625-8144

From: Chris Georgakidis <cgeorgakidis@inf.uth.gr>
Date: Wednesday, December 4, 2019 at 8:00 AM
To: George Karypis <george@karypis.net>
Subject: Re: hMetis Questions

Thank you for your quick reply Prof. Karypis.

On 3 Dec 2019, at 5:01 PM, George Karypis <george@karypis.net> wrote:

hMetis Overview Hypergraph Input File Format (8)

hMetis ensures an area balance between the clusters ensuring the no coarse node has a very large area

The area is the sum of the weights of the nodes in the cluster, if node weights are specified, otherwise it is the number of nodes.

23

hMetis Overview
Output File Format

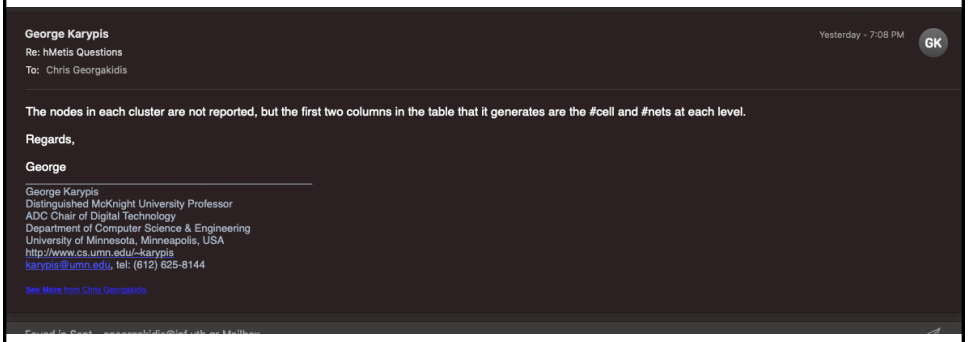
- The output of hMetis is a partition file.
- Format:
  - The partition file of a hypergraph with $|V|$ vertices, consists of $|V|$ lines with a single number per line.
  - The i^{th} line of the file contains the partition number that the i^{th} vertex belongs to. Partition numbers start from 0.
  - If foo.graph is the name of the file storing the hypergraph, the partition for a 2-way partition is stored in a file named foo.graph.part.2

Outline

- hMetis Overview
  - Quick Overview
  - Hypergraph Input File Format
  - Output File Format
- hMetis as Clustering Tool
  - Using hMetis
  - Using k-way hMetis
- Comparing ASP with hMetis

hMetis as Clustering Tool (1)

- No clustering information can be exported during the coarsening phase


George Karypis
Re: hMetis Questions
To: Chris Georgakidis

Yesterday - 7:08 PM GK

The nodes in each cluster are not reported, but the first two columns in the table that it generates are the #cell and #nets at each level.

Regards,

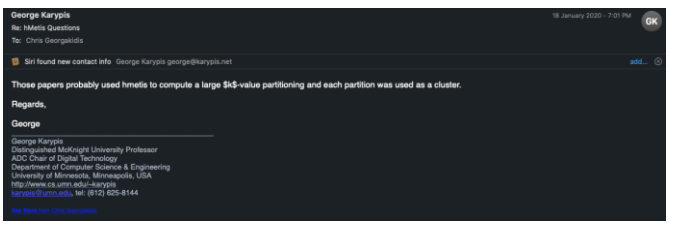
George

George Karypis
Distinguished McKnight University Professor
ADC Chair of Digital Technology
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, USA
http://www.cs.umn.edu/~karypis
karypis@umn.edu, tel: (612) 625-8144

See More from Chris Georgakidis

hMetis as Clustering Tool (2)

- hMetis can be used as a clustering tool taking into account the following assumption:
  - hMetis is a partitioning algorithms and it outputs a file assigning each node to a partition.
  - Each partition can be encountered as a cluster.
  - Thus in order to create N clusters we can use hMetis setting N as the number of final partitions.


George Karypis
Re: hMetis Questions
To: Chris Georgakidis

18 January 2020 - 7:01 PM GK

Siri found new contact info George Karypis george@karypis.net add...

Those papers probably used hmetis to compute a large k-value partitioning and each partition was used as a cluster.

Regards,

George

George Karypis
Distinguished McKnight University Professor
ADC Chair of Digital Technology
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, USA
http://www.cs.umn.edu/~karypis
karypis@umn.edu, tel: (612) 625-8144

See More from Chris Georgakidis

hMetis as Clustering Tool (3)

- We can use hMetis as clustering tool in 2 ways:
  - Using hMetis (Recursive Bisections)
  - Using khMetis (k-way Partitioning)
- KhMetis should never be used to compute a bisection (*i.e.*, 2-way partitioning) as it produces worse results than hMetis.
- Furthermore, the quality of the partitionings produced by khmetis for small values of *k* will be worse, in general, than the corresponding partitionings computed by hMetis.
- However, khmetis is particularly useful for computing *k*-way partitionings for relatively large values of *k*, as it often produces better partitionings and it can also enforce tighter balancing constraints.
- In general, when *k* is large (*e.g.*, $k > 16$) khmetis should be preferred over hMetis, as it is faster and enforces load imbalance constraints that are more natural than the bisection imbalance constraints enforced by hMetis.

28

Outline

- hMetis Overview
  - Quick Overview
  - Hypergraph Input File Format
  - Output File Format
- hMetis as Clustering Tool
  - Using hMetis
  - Using k-way hMetis
- Comparing ASP with hMetis

29

Comparing ASP with hMetis

- Comparing metrics:
  - Number of created clusters
  - Area balance between the clusters
  - Number of reduction levels
  - …

30

METIS Partitioning Tool

Karypis, George, and Vipin Kumar. "Multilevel algorithms for multi-constraint graph partitioning." *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 1998.

31

Definition and Notation

Partition Boundary | Boundary Vertex / Neighbourhood

- graph $G = (V, E)$
- Let P be the partitioning vector of size $|V|$
  - for each vertex v, P[v] stores the partition number that v belongs to.
- A vertex v that is adjacent to a vertex that belongs to a different partition is said to be in the ***partition boundary*** and it is called a ***boundary vertex***.
- For each vertex $v \in V$, in the partition boundary we define the ***neighborhood*** N(v) of v to be the union of the partitions other than P[v], that vertices adjacent to v (*i.e.*, Adj(v)) belong to.
  - $N(v) = \cup u \in Adj(v) \wedge P[u] \neq P[v] P[u]$
- In the case of a two-way partitioning, the neighborhood of each boundary vertex v contains a single partition.

32

Definition and Notation

Gain | External Degree | Internal Degree

- For each vertex v we compute the gains of moving v to one of its neighbor partitions.
- In particular, for every $b \in N$ (v) we compute $E\,D$[v]b as the sum of the weights of the edges (v, u) such that $P[u] = b$.
- Also we compute $I\,D$[v] as the sum of the weights of the edges (v, u) such that $P[u] = P$[v].
- The quantity $E\,D$[v]b is called the ***external degree*** of v to partition b, while the quantity $I\,D$[v] is called the ***internal degree*** of v.
- Given these definitions, the gain of moving vertex v to partition $b \in N$(v) is g[v]$b = E\,D$[v]$b - I\,D$[v].

33

Multi-Constraint k-way Refinement

- The key step in the multilevel k-way partitioning algorithm is the scheme used to perform the k-way refinement.
- relatively simple greedy k-way refinement heuristics are sufficient for producing high-quality partitionings
  - the initial partitioning is of very high quality, and
  - the multilevel paradigm, by applying these simple heuristics at different coarsening levels, is able to sufficiently explore the space of feasible solutions and find a high-quality solution.

34

Greedy k-way refinement algorithm

- The vertices that are along the boundary of the partitioning are visited in a random order and they are moved to one of their adjacent partitions if this movement:
  - improves the quality of the partitioning subject to the balance constraints,
  - improves the balance without worsening the quality.
- Specifically, consider a graph $G = (V, E)$, and its partitioning vector P. For each vertex $v \in V$ on the partitioning boundary, let $N(v)$ be its neighborhood.
- Now during the greedy k-way refinement, the partition $b \in N(v)$ to which vertex v is moved to is determined as follows.
  - First we compute a subset $N' \subseteq N(v)$ of the neighboring domains of v such that the movement of v to any of these domains does not increase the edge-cut; that is, for all $b \in N'$, $g[v]b \geq 0$.
  - Next, we remove from N' all the domains for which the movement of v into them will violate the balancing constraints. Let $N'' \subseteq N'$ be this subset of N'.
  - Now, we move v to one of the subdomains b in N'' for which we achieve the largest reduction in the edge-cut; that is, $g[v]_b = \max_{i \in N''}(g[v]_i)$, and breaking ties in favor of the domains for which we achieve the best balance.
  - If for all domains in N'', the gain achieved in moving to them is zero and the balance is no better than it was originally, then v does not move.
  - Whenever a vertex is moved, the neighboring and gain information of the existing and currently emerging boundary vertices is updated to reflect the current state of the partitioning.

35

Greedy *k*-way refinement algorithm

- ***IsBalanceOK***(v, *a*, *b*):
  - This function determines if the movement of a vertex v from domain *a* into domain *b* satisfies the balancing constraints. This function is essential in constructing the set N'' from the set N' of moves that do not increase the edge-cut.
- ***IsBalanceBetterFT***(v, *a*, *b*)
  - This function determines if the movement of a vertex v from domain *a* to domain *b* will lead to a better balance compared to not moving v. This function is used to determine if v will be moved to *b* when the gain associated with that move is zero.
- ***IsBalanceBetterTT***(v, *a*, *b*1, *b*2)
  - This function determines if the movement of a vertex v from domain *a* to domain *b*2 will lead to a better balance compared to a move to domain *b*1. This function is used to select among the domains in N'' with the highest gain, the one that leads the best balance.

```
SelectMoveTarget(v, N(v))
{
  N' = {b | b ∈ N(v) and ED[v]_b − ID[v] ≥ 0}
  N'' = {b | b ∈ N' and IsBalanceOK(v, P[v], b)}

  if (|N''| = 0) then return -1

  b = N''[1];
  for (i = 2; i ≤ |N''|; i = i + 1) {
    j = N''[i]
    if (ED[v]_j > ED_b[v]) then b = j
    elif (ED[v]_j = ED_b[v] and IsBalanceBetterTT(v, P[v], b, j)) then b = j
  }

  if (ED[v]_b-ID[v] > 0 or (ED[v]_b-ID[v] = 0 and IsBalanceBetterFT(v, P[v], b))) then return b
  else return -1;
}
```

Algorithm 6.1: The above routine returns the domain in which v should be moved, or -1 if no such domain was found.

Multi-constraint Iterative *k*-way Balancing

- The primary focus of the *k*-way refinement algorithm described in the previous section is to improve the quality of a *k*-way partitioning subject to multiple constraints.
- However, if the partitioning does not initially satisfy the balancing constraints, then the *k*-way refinement algorithm may not be able to balance the partitioning.
  - Since the *k*-way refinement algorithm moves only vertices that do not decrease the edge-cut, it may not be able to balance an unbalanced partitioning if such a balancing requires the movement of vertices with negative gains.
- For this reason, after projecting a partitioning to the next level finer graph and prior to calling the *k*-way refinement algorithm, we use a *k*-way balancing algorithm to improve the load imbalance whenever such a load imbalance exists.

Multilevel k-way Partitioning Scheme for Irregular Graphs

38

Uncoarsening Phase

Our Implementation

However, refining a k-way partitioning is significantly more complicated because vertices can move from a partition to many other partitions; thus, increasing the optimization space combinatorially. An extension of the KL refinement algorithm in the case of k-way refinement is described in [10]. This algorithm uses $k(k-1)$ priority queues, one for each type of move. In each step of the algorithm, the moves with the highest gain are found from each of these $k(k-1)$ queues, and the move with the highest gain that preserves or improves the balance is performed. After the move, all of the $k(k-1)$ priority queues are updated. The complexity of k-way refinement is significantly higher than that of 2-way refinement, and for a graph with m edges, this complexity is $O(k * m)$. This approach is only practical for small values of k. Due to this high complexity, the multilevel recursive octasection algorithm described in [10], requires the same amount of time as multilevel recursive bisection, even though recursive octasection spends much less time for coarsening.

39

k-way Refinement Algorithms
Greedy Refinement (GR)

Greedy Refinement (GR). The lookahead in KL algorithm serves a very important purpose. It allows movement of an entire cluster of vertices across a partition boundary. Note that it is quite possible that as the cluster is moved across the partition boundary, the edge-cut increases, but after the entire cluster of vertices moves across the partition, then the overall edge-cut comes down. In the context of multilevel schemes, this lookahead becomes less important. The reason is that these clusters of vertices are coarsened into a single vertex at successive coarsening phases. Hence, movement of a vertex at a coarse level actually corresponds to the movement of a group of vertices in the original graph.

If the lookahead part of KL is eliminated (i.e., if vertices are moved only if they lead to positive gain), then it becomes less useful to maintain a priority queue. In particular, vertices whose move results in a large positive gain will be moved anyway even if they are not moved earlier in the priority order. Hence, a variation of KL that simply visits the boundary vertices in a random order and moves them if they result in a positive gain is likely to work well in the multilevel context. Our *greedy refinement* algorithm is based on this observation. It consists of a number of iterations. In each iteration, all the vertices are checked to see if they can be moved so that either the edge-cut of the partitioning can be decreased (while preserving balance), or the balance is improved.

40

k-way Refinement Algorithms
Greedy Refinement (GR) Algorithm

In particular, GR works as follows. Consider a graph $G_i = (V_i, E_i)$, and its partitioning vector P_i. The vertices are checked in a random order. Let v be such a vertex, let $P_i[v] = a$ be the partition that v belongs to. If v is a node internal to partition a then $N(v) = \emptyset$, and v is not moved. If v is at the boundary of the partition, then $N(v)$ is nonempty. Let $N'(v)$ be the subset of $N(v)$ that contains all partitions b such that movement of vertex v to partition b does not violate the Balancing Condition. Now vertex v is moved to one of the adjacent partitions b, if either one of the following conditions is satisfied:

1. $ED[v]_b > ID[v]$ and $ED[v]_b$ is maximum among all $b \in N'(v)$.
2. $ED[v]_b = ID(v)$ and $W_i[a] - W_i[b] > w(v)$.

That is, the GR algorithm moves v to a partition that leads to the largest reduction in the edge-cut without violating the balance condition. If no reduction in the edge-cut is possible, by moving v, then v is moved to the partition (if any) that leads to no increase in the edge-cut but improves the balance. After moving vertex v, the algorithm updates the internal and external degrees of the vertices adjacent to v to reflect the change in the partition.

41

k-way Refinement Algorithms

Global Kernighan-Lin Refinement (GKLR)

Global Kernighan–Lin Refinement (GKLR). As discussed in the previous section, the GR algorithm lacks any capabilities of climbing out of local minima. Our second refinement heuristic called *global Kernighan–Lin*, is somewhat more powerful and is closer to the original KL algorithm in spirit. It adds some limited hill-climbing capabilities to the GR algorithm and also uses a priority queue to determine the sequence of vertex moves.

The GKLR algorithm uses a global priority queue that stores the vertices according to their gains. Initially, all the vertices are scanned, and those whose sum of external degrees [5] is greater or equal to their internal degrees are inserted into the priority queue. In particular, let v be such a vertex, let $N(v)$ be the neighborhood of v, and $b \in N(v)$ such that $ED[v]_b$ is maximum over the external degrees of partitions in $N(v)$. We insert v into the priority queue with a gain equal to $ED[v]_b - ID[v]$.

The algorithm then proceeds and selects the vertex from the priority queue with the highest gain. Having selected such a vertex v, the algorithm selects a part $b \in N(v)$ to move v such that $ED[v]_b$ is maximized while satisfying the balance condition (Eqs. (2) and (3)). Note that these swaps may lead to an increase in the edge-cut, since vertices are moved even if they have a negative gain value. The GKLR algorithm continues moving vertices until it has performed x vertex moves that have not decreased the overall edge-cut. In that case, the last x moves are undone. Once a vertex is moved, it is not considered for movement in the same iteration. This is repeated for a small number of iterations or until convergence.

Note that in each step, the vertices selected for movement by the GKLR algorithm and by the generalized KL of [11] may be quite different. GKLR selects a vertex v that has a move (among all possible moves to neighboring partitions $N(v)$) with the highest gain $g[v]_{max}$. However, depending on the weight of the partitions, this move may never take place, and instead v can be moved to a partition $a \in N(v)$ that leads to a smaller gain $g[a]_v$. However, there may be another vertex u on the priority queue that has a move with the highest gain $g[u]_{max}$ that may be permissible. Now if $g[v]_v < g[u]_{max} < g[v]_{max}$, the generalization of the KL algorithm will select to move vertex u before considering vertex v. Thus, in each step, GKLR does not necessarily select the vertex with the largest realizable gain. Furthermore, since the single priority queue contains only vertices whose sum of the external degrees in greater or equal to the internal degree, GKLR has less powerful hill-climbing capabilities than the generalized KL [11] that uses multiple priority queues and considers all the vertices.

42