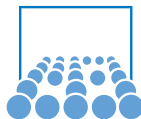


# HPC – Algorithms and Applications

## Dwarf #6 – Unstructured Grids

Michael Bader

Winter 2013/2014



# Dwarf #6 – Unstructured Grids

1. dense linear algebra
2. sparse linear algebra
3. spectral methods
4. N-body methods
5. structured grids
6. **unstructured grids**
7. Monte Carlo



# Unstructured Grids – Characterisation

- (almost) no restrictions on grid generation, maximum flexibility
- explicit storage of basic geometric and topological information → usually complicated data structures



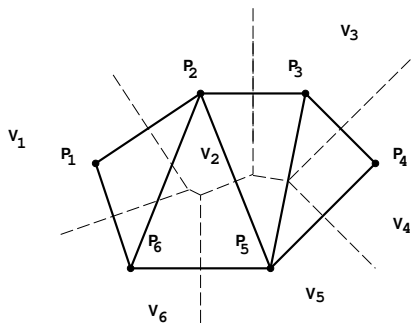
## Example: Delaunay Triangulation

- assume: grid points are already given
- to do: generate triangular grid cells
- satisfy Delaunay property: circumcircle of any grid triangle does not contain other grid vertices
- leads to triangles with favourable properties: avoid acute/obtuse angles
- related to **Voronoi** diagrams (next slide)
- widespread (computer graphics, meshes for Finite Element methods, etc.)

# Delaunay Triangulation and Voronoi Diagrams

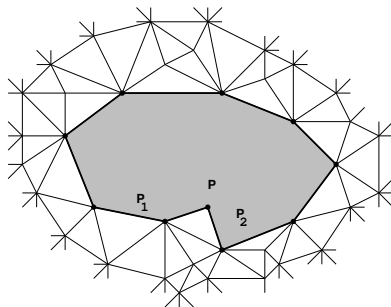
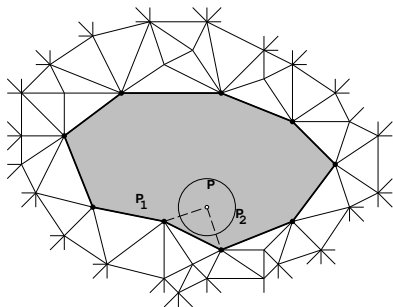
## Algorithm

1. Voronoi region around each given grid point:  
$$V_i = \{P : \|P - P_i\| < \|P - P_j\| \forall j \neq i\}$$
2. connect points from adjacent Voronoi regions
3. leads to set of disjoint triangles (tetrahedra in 3D)



## Example: Advancing Front Methods

- approach to generate both grid points and grid cells
- advance a *front* step-by-step towards interior
- starting from the boundary (*starting front*)



## Advancing Front Methods (2)

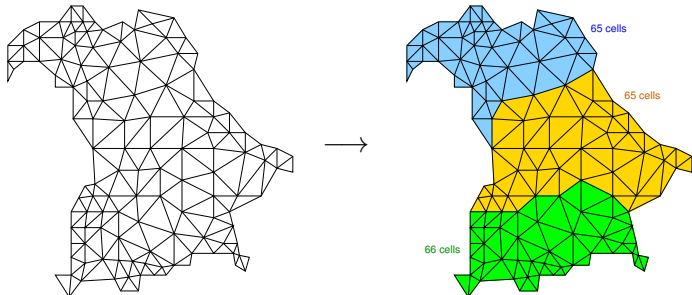
Algorithm:

1. choose an edge on the current front, say PQ
2. create a new point R at equal distance  $d$  from P and Q
3. determine all grid points lying within a circle around R, radius  $r$
4. order these points w.r.t. distance from R
5. for all points, form triangles with P and Q; select one of these triangles
6. add triangle to grid (unless: intersections, ...)
7. update triangulation and front line: add new cell, update edges

# Partitioning Unstructured Grids

## Partitioning problem:

- divide grid into  $K$  partitions
- with uniform computational load
  - usually: partitions of equal size
- with minimal communication effort
  - minimise number of grid cells at partition boundaries

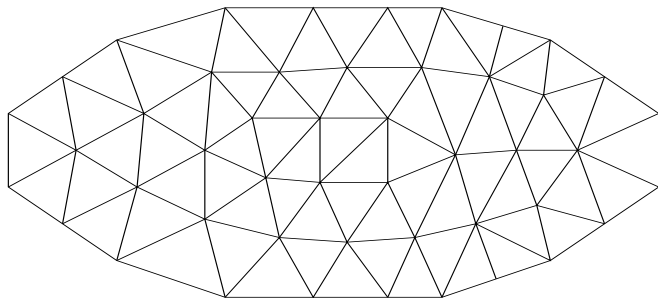




# Graph-Based Partitioning

Graph-Representation of Grids:

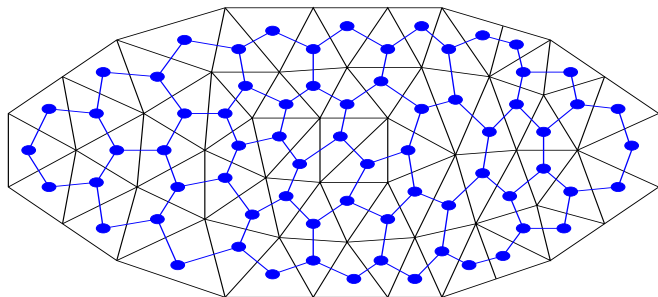
- “standard” graph  $(V, E)$  for a grid:  
 $V$  = grid vertices,  $E$  = set of all grid cell edges
- vs. “dual” graph  $(V', E')$ :  
 $V'$  = grid cells,  $E'$  = tuples of adjacent grid cells



# Graph-Based Partitioning

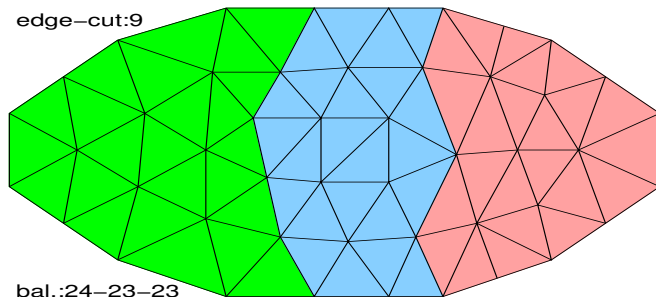
Graph-Representation of Grids:

- “standard” graph  $(V, E)$  for a grid:  
 $V$  = grid vertices,  $E$  = set of all grid cell edges
- vs. “dual” graph  $(V', E')$ :  
 $V'$  = grid cells,  $E'$  = tuples of adjacent grid cells



# K-way Graph Partitioning

- divide  $V$  (or  $V'$ ) into  $K$  equal-sized partitions  $V_k$ :  
 $\bigcup_k V_k = V, |V_k| = |V|/K, V_k \cap V_j = \emptyset$  (if  $k \neq j$ )
- minimise **edge cut**:  $\{(e, f) \in E : e \in V_k, f \notin V_k\}$
- *NP*-complete problem  $\Rightarrow$  use heuristics-based algorithms



# Multilevel k-Way Partitioning

Algorithm by Karypis and Kumar (1998):

## 1. coarsening phase:

- successively collapse sets of vertices to reduce problem size
- conserve vertex/edge weights

## 2. partitioning phase:

- perform  $K$ -way partitioning on a coarse graph

## 3. uncoarsening phase:

- successively expand collapsed vertices to obtain respective partitioning of the original graph
- postprocessing after each uncoarsening step to improve load balance

# Coarsening Phase

## Coarsening by **Matching**:

- “matching”: set of edges, where no two edges share a common vertex
- “maximal” matching: a matching, where no further edges can be added  
(but some vertices might still be without a match)
- in contrast: “perfect” matching (matching covers all vertices)

## Matching-based Coarsening:

- two vertices connected by an edge of the matching will be collapsed
- stop coarsening, if graph is small enough or matching does no longer lead to sufficient coarsening

# Algorithms for Matching

## Random Matching:

- vertices are visited in random order
  - an unmatched vertex  $u$  randomly selects an unmatched connected vertex  $v$   
→  $(u, v)$  is added to the matching
  - vertices stay unmatched, if they no longer have an unmatched neighbour
- ⇒ simple, greedy approach; however, does not consider minimisation of edge-cut

## Algorithms for Matching (2)

### Heavy Edge Matching:

- use weighted edges:  $W(e)$ ;  
and  $W(A) := \sum_{e \in A} W(e)$  for a set  $A$  of edges
  - $E_{i+1}$  and  $E_i$  the edges of coarse/fine graph due to a matching  $M_i$ , then:  $W(E_{i+1}) = W(E_i) - W(M_i)$
  - heuristics: use heavy edges for matching
  - again: visit vertices in random order;  
pick edge (to unmatched vertex) with the largest edge weight
- ⇒ greedy approach, heuristics to keep edge-cut low, but does not guarantee minimisation of edge-cut

## Algorithms for Matching (3)

### Modified Heavy Edge Matching:

- experience: coarse graphs with low average degree (number of outgoing edges) of edges lead to partitions with lower edge-cut
- chose random vertex  $v$   
 $\Rightarrow H(v)$  the set of adjacent edges with maximum weight
- for each  $u \in H(v)$ , define  $W(v, u) := \sum_e W(e)$  for all edges  $e$  that
  - are adjacent to  $v$ , i.e.  $e = (v, u')$
  - $u'$  is connected to  $u$
- determine maximum  $W(v, u)$  and pick respective  $(v, u)$  for matching



# Collapse Graph after Matching

## Determine Coarse Vertices:

- matching  $M_i$  computed for  $(V_i, E_i)$
- each  $m \in M_i$  becomes a vertex of  $V_{i+1}$
- each non-matched  $v \in V_i$  becomes a vertex of  $V_{i+1}$
- weight vertices to preserve load balance info:  $\mathcal{W}(v)$  by adding the weights of the two matched edges

## Determine Coarse Edges:

- an edge between two vertices of  $V_{i+1}$  is generated, if an edge in  $E_i$  connects any of the former members
- the edge weights are added over all such connections  
⇒ preserve information to determine edge-cut

# Partitioning of the Coarse Graph

Options:

- coarsen until only  $k$  graph vertices are left?
  - bad partitions (vertices no longer equally weighted);
  - matching does not reduce graph size well for small partitions
- switch to multilevel recursive bisection
  - turns out as successful choice
- Fiedler vector for partitioning (spectral methods)
  - solve eigenvalue problem on the adjacency matrix
- geometric methods (coordinates required)
- combinatorial methods

# Uncoarsening of the Graph Partitions

## Backprojection:

- partitioning  $P_{i+1}$  given on coarse graph
- put vertex  $v$  of  $P_i$  to partition  $p \in P_i$ , if match-vertex of  $v$  belongs to  $p$  in  $P_{i+1}$

## Local Refinement:

- even, if  $P_{i+1}$  might be (locally) optimal,  $P_i$  can be improved, as more degrees of freedom are available
- approach: swap vertices between partitions to reduce edge cut (until a local minimum is reached)

# Local Refinement Algorithm

- define *neighbourhood*  $N(v)$  for each vertex  $v$ :  
set of adjacent partitions
- for each vertex  $v$  and partition  $B \in N(v)$ :  
compute gain  $g(v, B)$  for moving  $v$  into partition  $B$
- move vertex from its original partition  $A$  to  $B$ , if
  1. gain  $g(v, B)$  is largest among  $N(v)$  and  $> 0$  and
  2. load balance is maintained:

$$\mathcal{W}(B) + \mathcal{W}(v) \leq \mathcal{W}_{\max} \quad \text{and} \quad \mathcal{W}(A) - \mathcal{W}(v) \geq \mathcal{W}_{\min}$$

- *greedy refinement*: visit vertices at partition boundaries in random order; move to the partition with largest gain
- in addition: move vertex, if edge cut stays equal, i.e.  $g(v, B) = 0$ , but balance is improved

## Local Refinement Algorithm (2)

Determine gain of vertex:

- sum up weights of edges to neighbour partition

$$\rightarrow \text{external degree: } ED(v, B) := \sum_{u \in P_B} W(v, u)$$

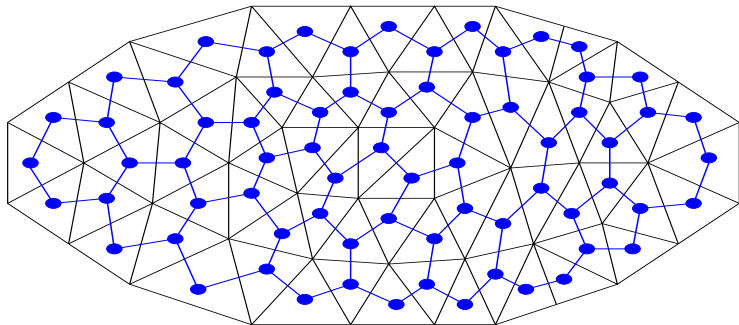
- sum up weights of edges in the same partition  $A$

$$\rightarrow \text{internal degree: } ID(v) := \sum_{u \in P_A} W(v, u)$$

- gain of moving  $v$  to  $b$ :  $g(v, B) = ED(v, B) - ID(v)$

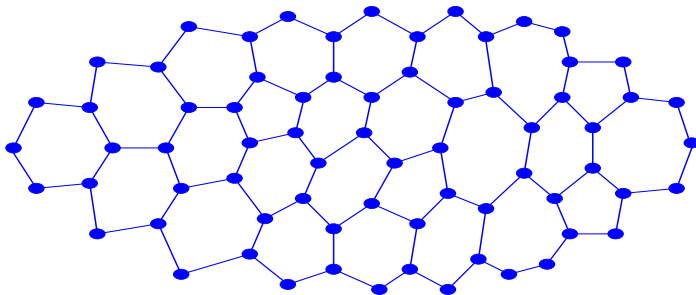
# MLkP-Example – Coarsening Phase

Start with dual graph:



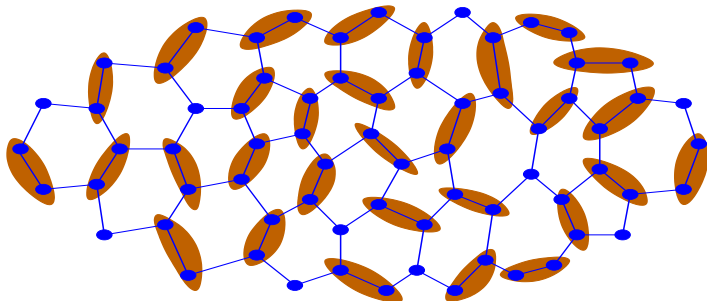
# MLkP-Example – Coarsening Phase

Start with dual graph:



# MLkP-Example – Coarsening Phase

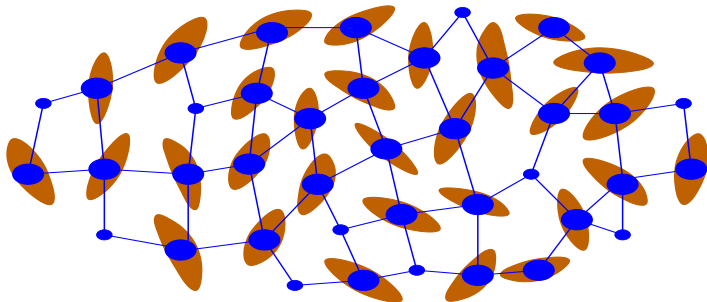
Random matching:





# MLkP-Example – Coarsening Phase

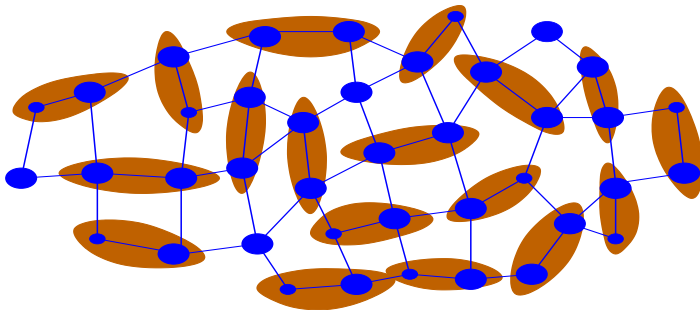
Collapse vertices and re-build adjacency graph:



(bigger discs indicated heavier vertices, i.e. multiple grid cells)

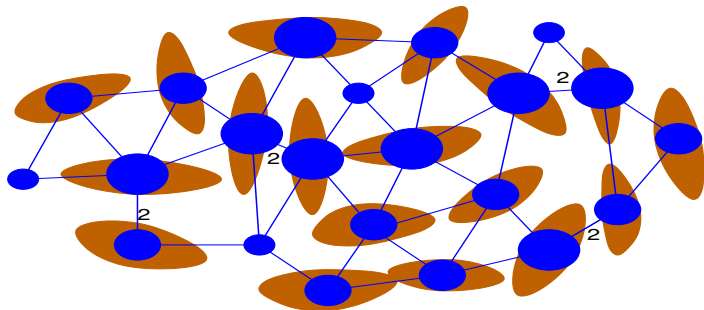
# MLkP-Example – Coarsening Phase

Random matching:



# MLkP-Example – Coarsening Phase

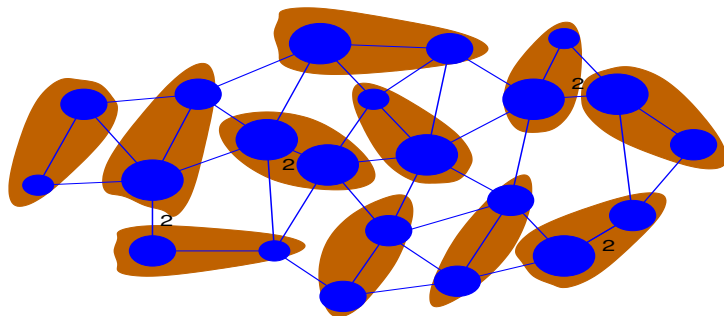
Collapse vertices and re-build adjacency graph:



(multiple edges between matchings lead to edge weights  $> 1$ )

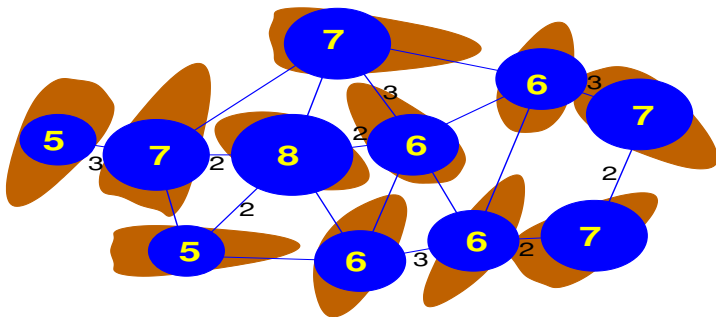
# MLkP-Example – Coarsening Phase

Random matching:



# MLkP-Example – Coarsening Phase

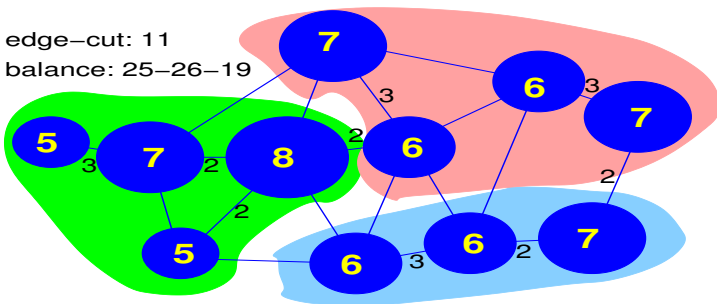
Collapse vertices and re-build adjacency graph:



(yellow numbers indicated vertex weights)

# MLkP-Example – Partitioning

Determine initial partitioning on coarsened graph:



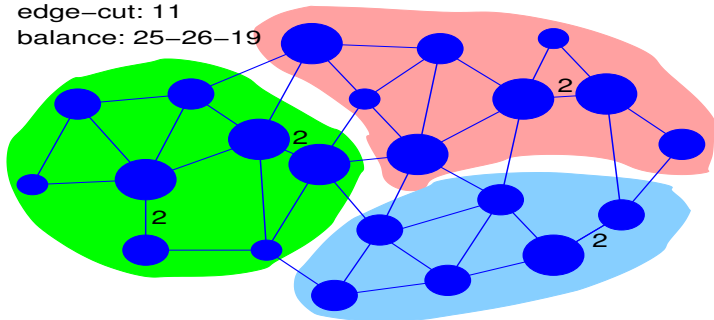
(minimize edge-cut: do not cut 2-/3-weighted edges)

# MLkP-Example – Uncoarsening Phase

Inflate collapsed vertices:

edge-cut: 11

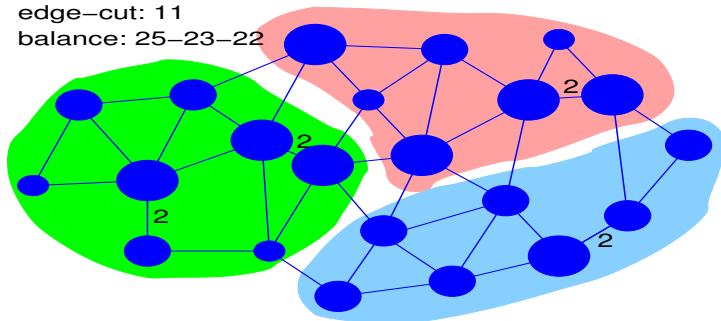
balance: 25–26–19



# MLkP-Example – Uncoarsening Phase

Local improvement:

edge-cut: 11  
balance: 25–23–22

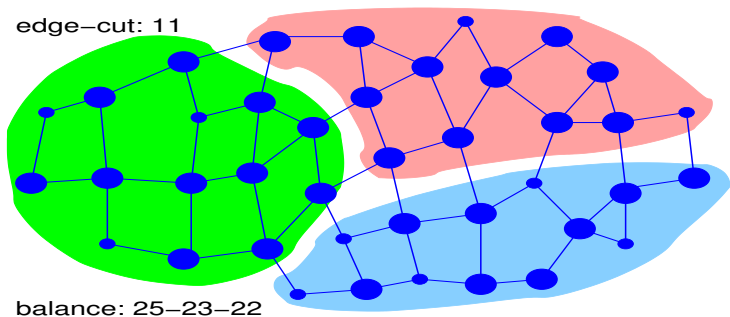


(right-most vertex moves from pink to blue partition)



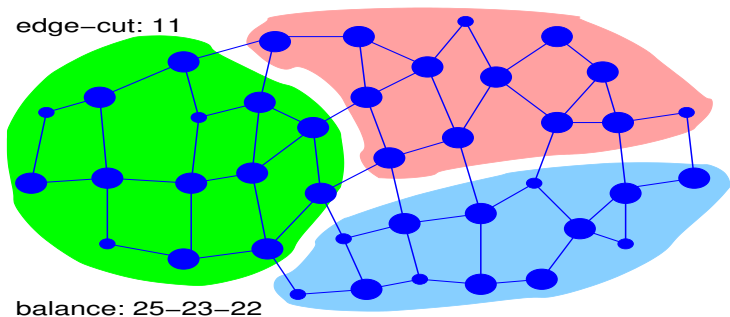
# MLkP-Example – Uncoarsening Phase

Inflate collapsed vertices:



# MLkP-Example – Uncoarsening Phase

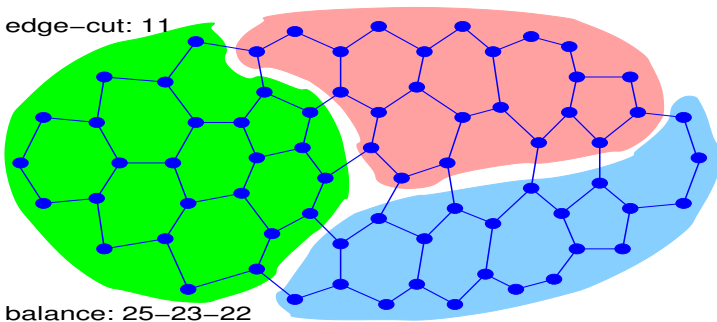
Local improvement:



(here: no vertex moves that improve edge-cut or balance)

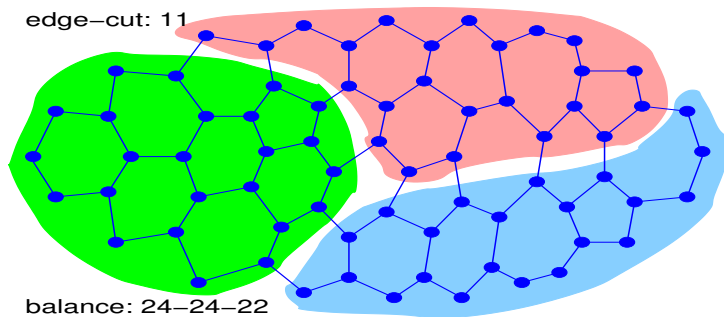
# MLkP-Example – Uncoarsening Phase

Inflate collapsed vertices:



# MLkP-Example – Uncoarsening Phase

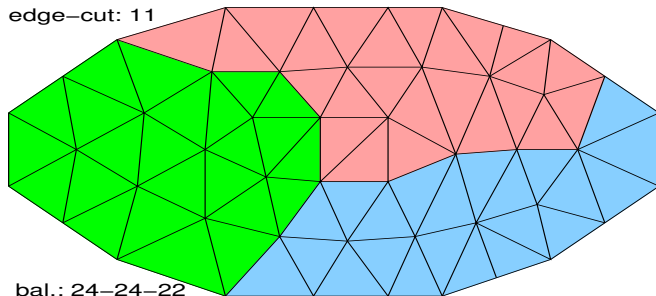
Local improvement:



(top-left vertex moves from green to pink partition)

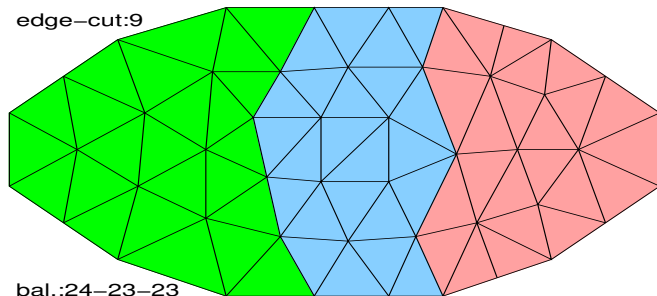
# MLkP-Example – Computed Partition

Partitioning obtained via (our) MLkP algorithm:



# MLkP-Example – Computed Partition

Compare with optimal(?) partitioning:



Analyse: what choices lead to different partitioning?