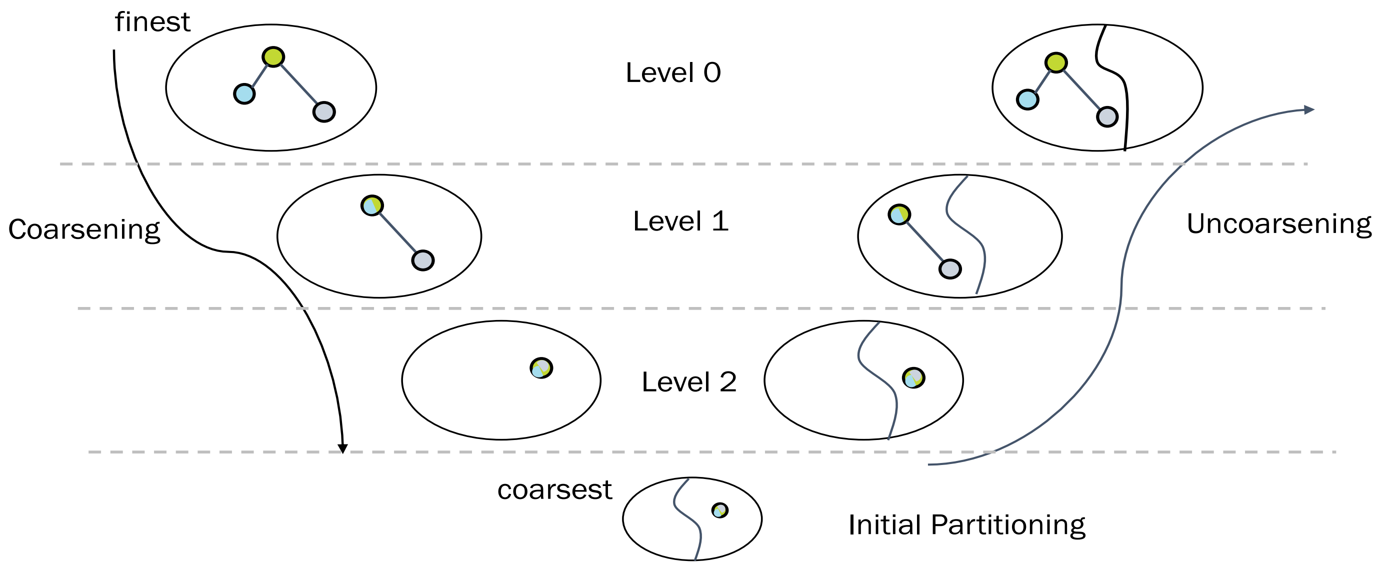
HyperG: A Multilevel GPU-Accelerated k-way Hypergraph Partitioner

Wan Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu,
Ulf Schlichtmann, and Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, WI

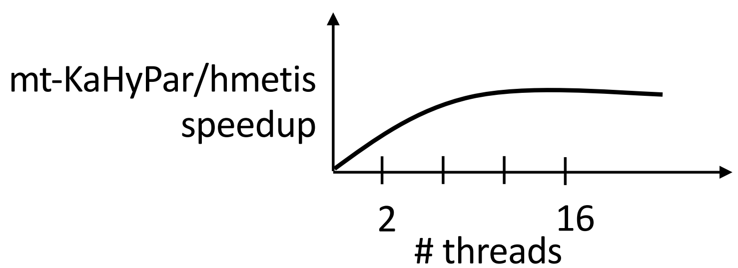
Hypergraph Partitioning is Important in CAD

- Breaks down a large circuit into manageable pieces
  - Ex: divide & conquer
- Mainstream graph partitioning algorithms are multi-level



However, Hypergraph Partitioning is Time-consuming

- Modern circuit complexity and size continue to increase
  - Ex: Four minutes for hmetis to partition a circuit with five million-gate
  - Partitioning can be performed multiple times during a CAD algorithm
- CPU parallel hypergraph partitioners mitigate the runtime challenges
  - Ex: Mt-KaHyPar
  - Speedup plateaus at 8–16 CPU threads
- GPU non-hypergraph partitioners
  - G-kway
  - GKSG
- There is a need for a GPU-accelerated hypergraph partitioning algorithm



GPU-accelerated Hypergraph Partitioner is NOT EASY

- Uses GPU non-hypergraph partitioning algorithm on hypergraph results in poor quality
  - Extra cost of transforming hypergraph into non-hypergraph
  - Transformed graph fails to accurately represent the original hypergraph
- Distinct performance characteristics of CPU and GPU require different data layout designs to maximize computing efficiency
  - Ex: Mt-KaHyPar's coarsening algorithm requires frequent synchronization, which is costly on GPUs

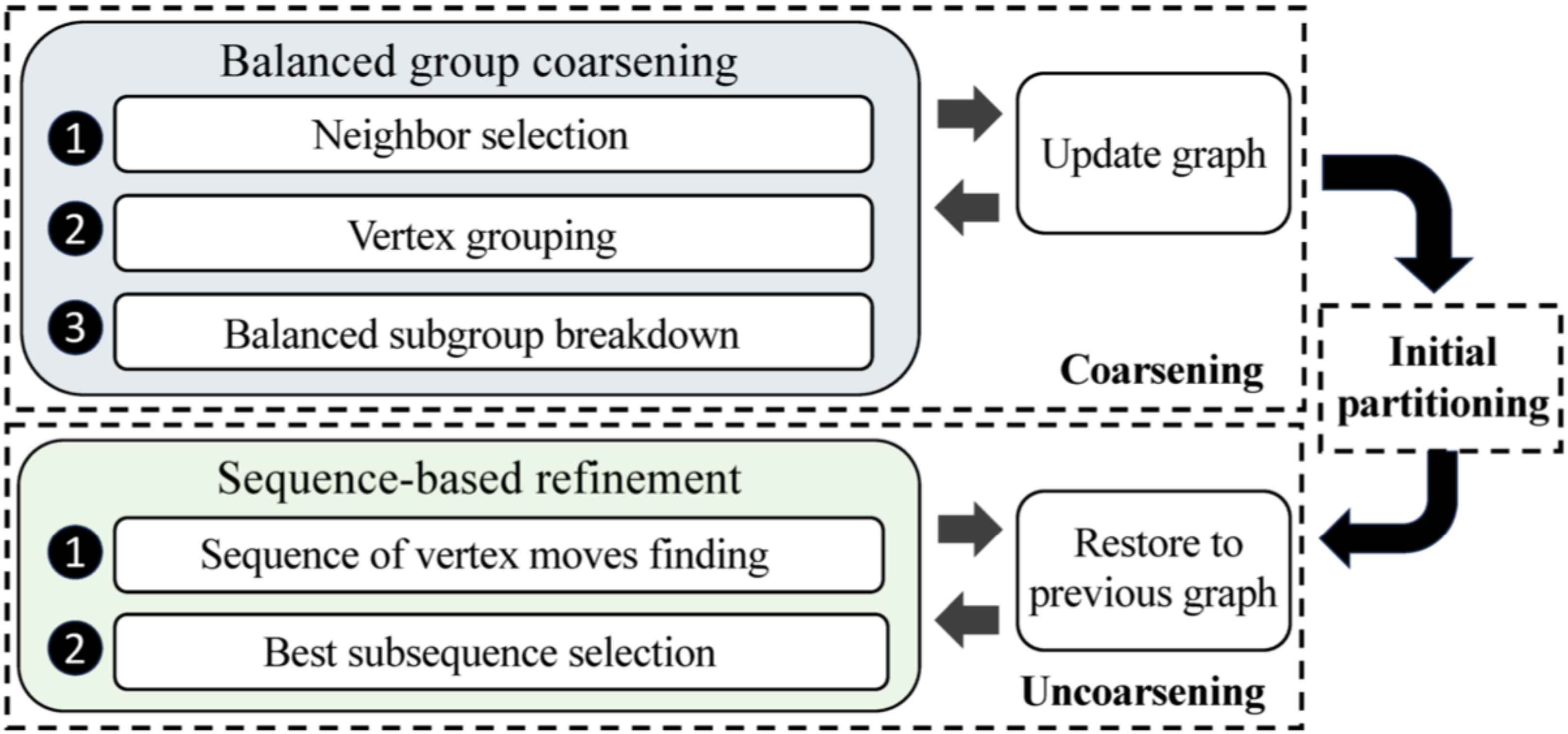
HyperG: A GPU-accelerated Hypergraph Partitioner

- Among the earliest attempts to parallelize both coarsening and uncoarsening stages on a GPU
- Balanced group coarsening algorithm
  - Groups many vertices into balanced subgroups
- Sequence-based refinement
  - Simultaneously moves the best vertices to improve partitioning quality
- Modern CUDA warp-level primitives
  - Achieves fine-grained synchronization and efficient intra-warp communication

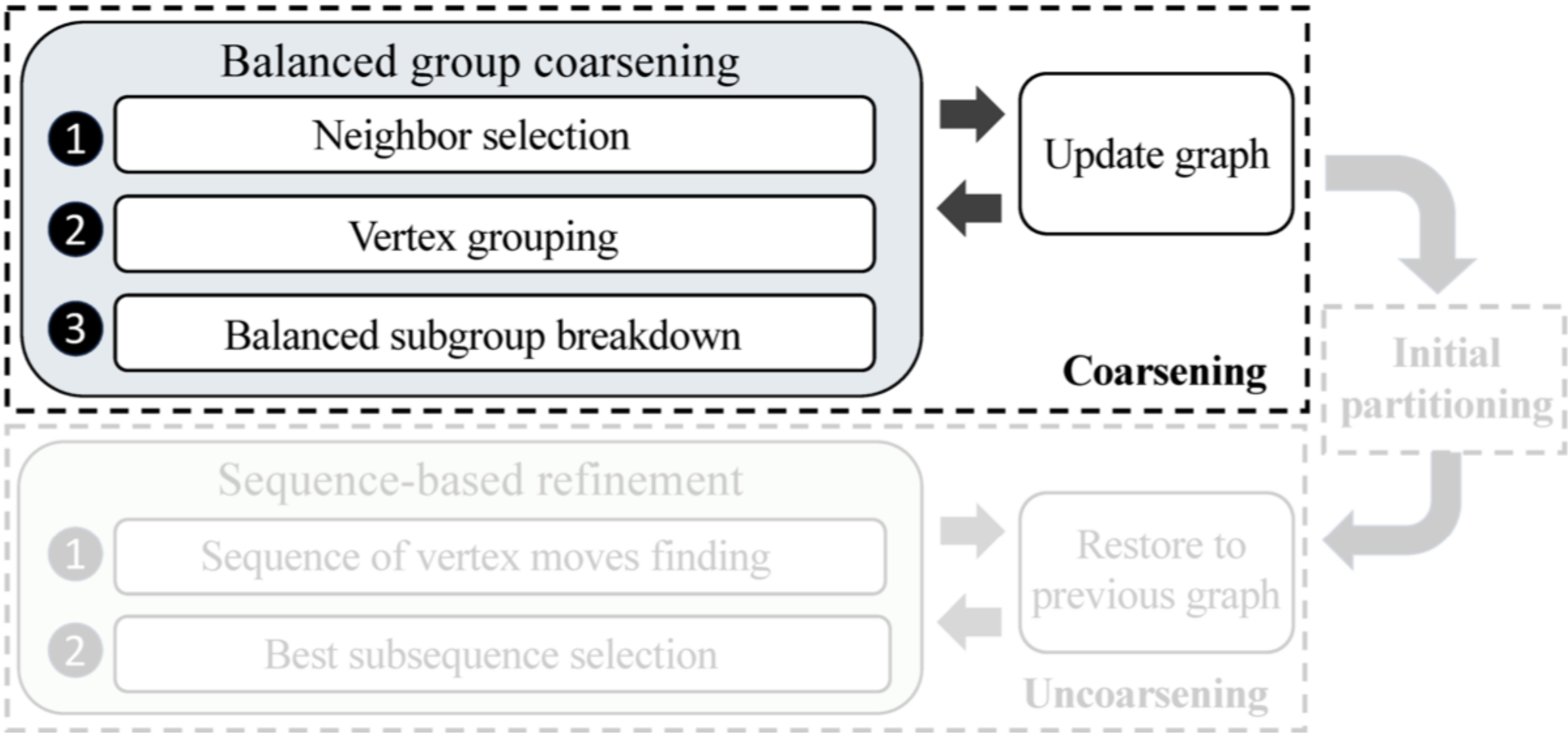
Hypergraph Partitioning Problem

- A hypergraph is a graph where edges (hyperedges) can connect multiple vertices
- Goal: Divide the vertices of the hypergraph into k disjoint sets (partitions) of roughly equal size while minimizing the cut size
- Cut size: Sum of the weights of hyperedges connecting vertices in different partitions
- $Cut = \Sigma_{e \in E} \delta(e)$
- $\delta(e) = \begin{cases} 1, \exists v_1, v_2 \in e \;\; s.t. \; p(v_1) \neq p(v_2) \\ 0, \qquad\qquad\qquad\qquad\qquad otherwise \end{cases}$

HyperG Overview



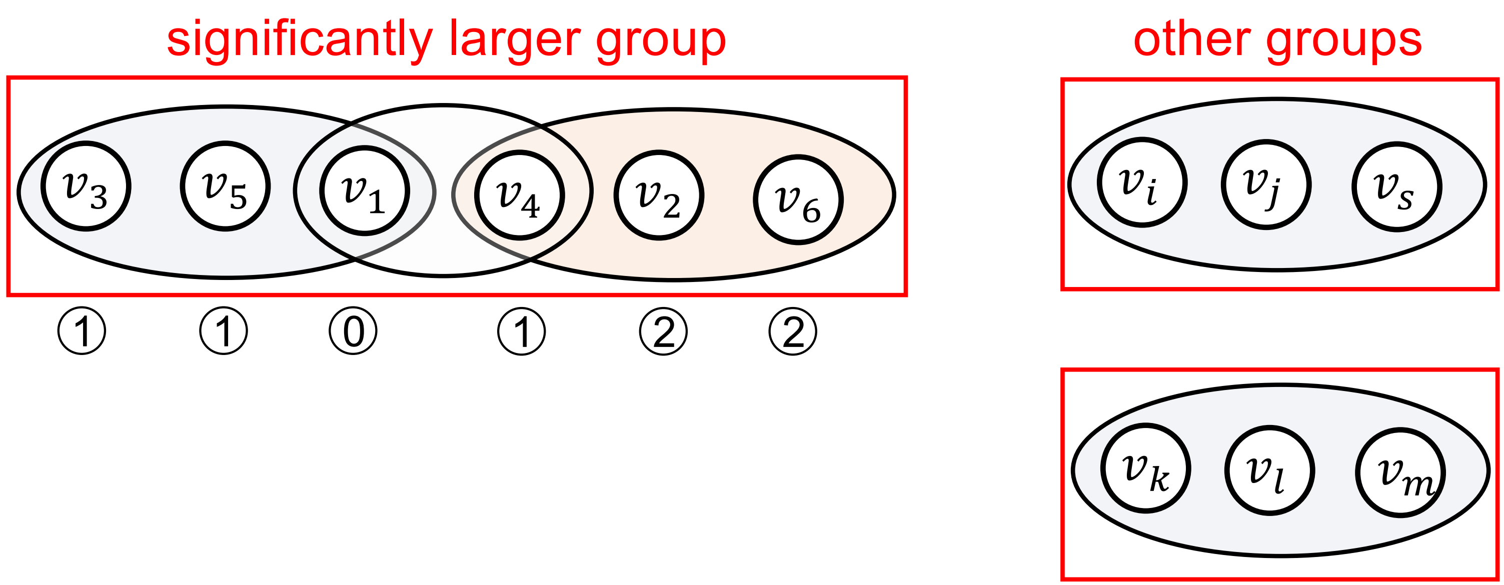
Coarsening Stage



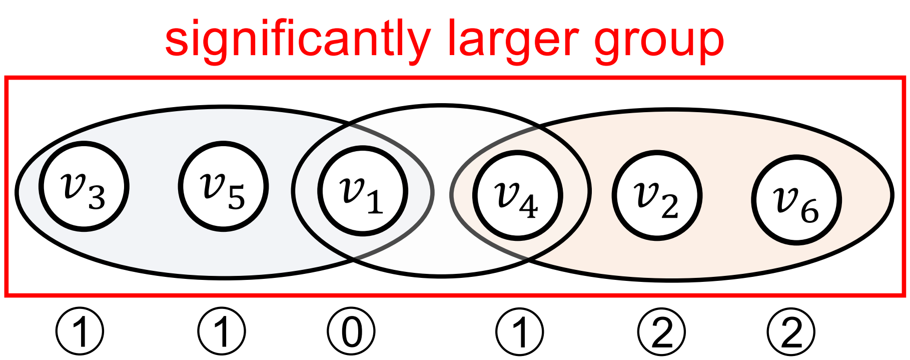
HyperG Balanced Group Coarsening

- Groups vertices and coarsens all vertices within the same group together
  - Largely reduces coarsening time by coarsening many vertices together
- However, it can cause imbalanced sizes in coarsened vertices, making it challenging to achieve a balanced partition during the initial partitioning stage
- Solution: Sorts vertices within groups by distance and divides each group into fixed-size subgroups

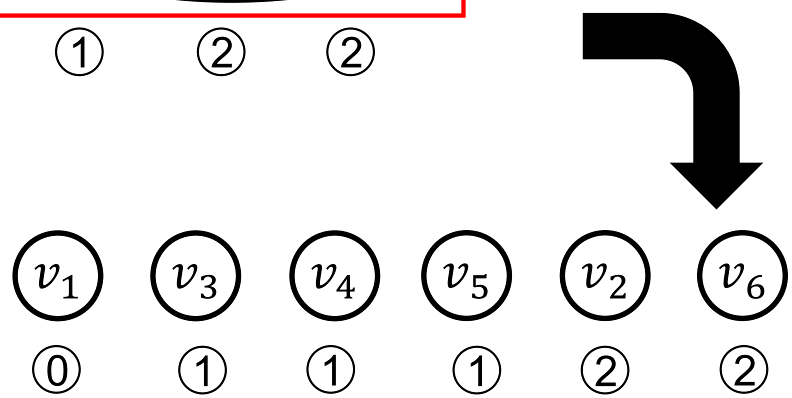
Balanced Group Coarsening Example



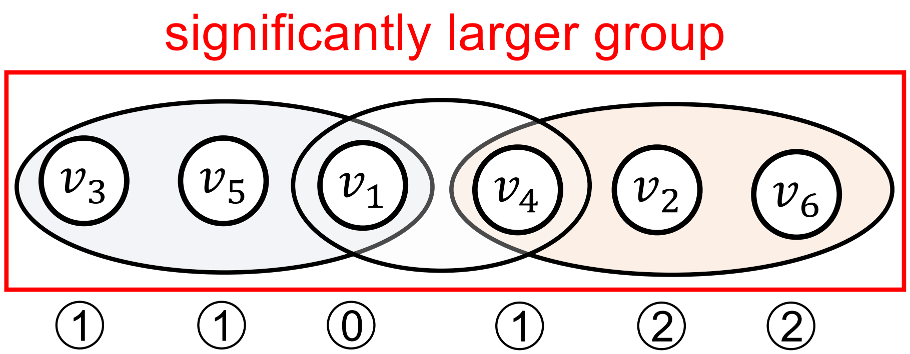
Balanced Group Coarsening Example

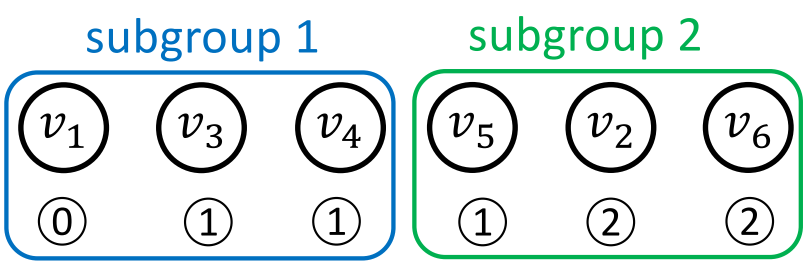


Sort by distance to the group leader (vertex with the smallest ID)

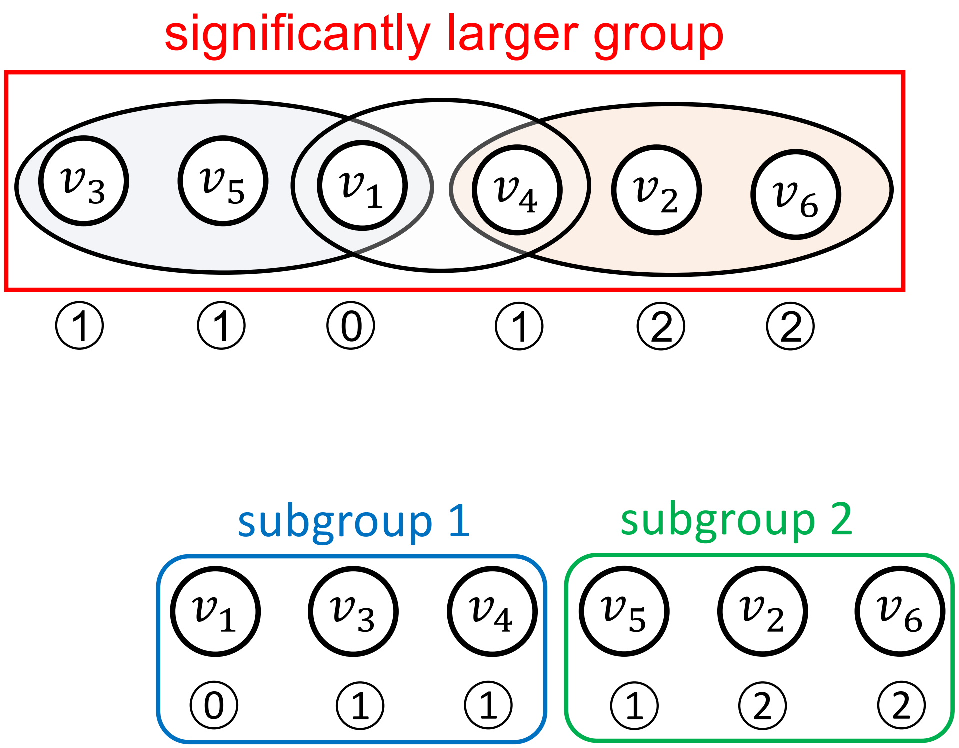


Balanced Group Coarsening Example

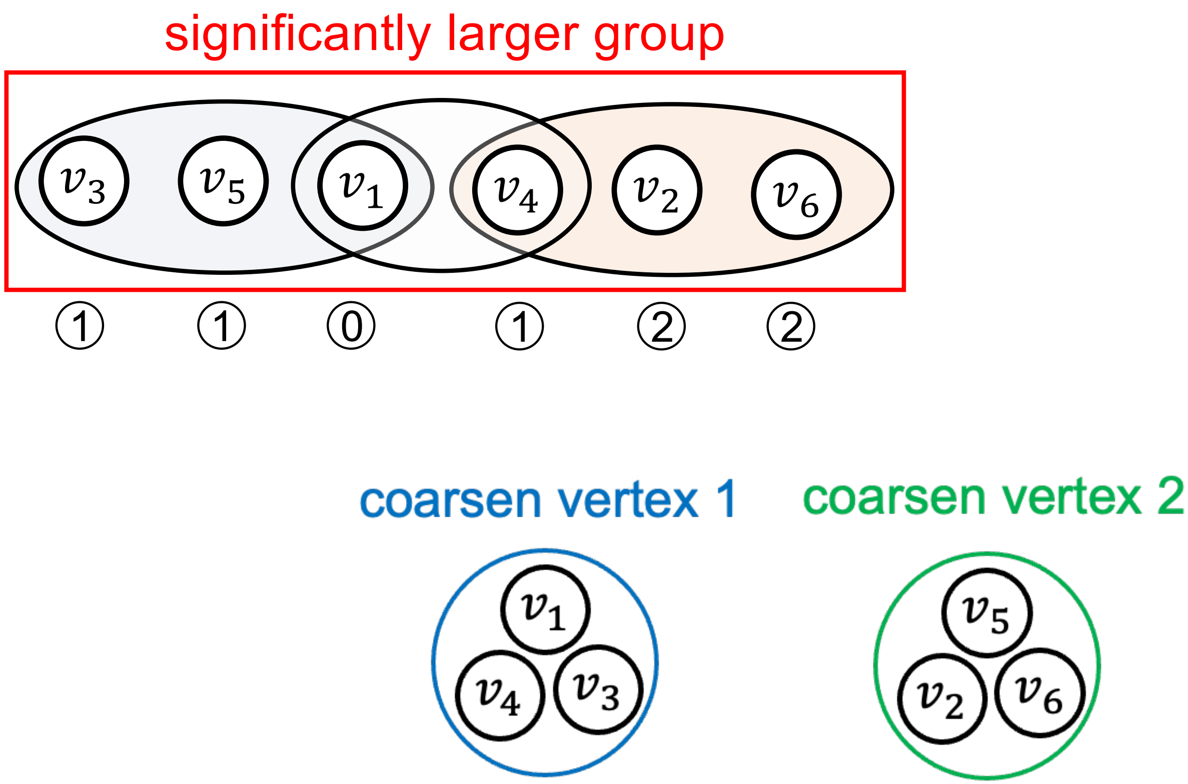




Balanced Group Coarsening Example



Balanced Group Coarsening Example



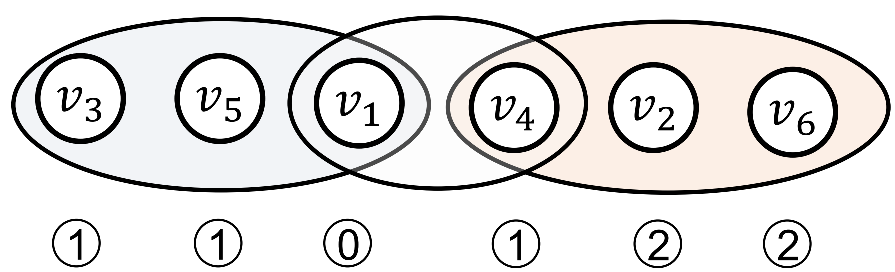
Balanced Group Coarsening Parallelization

group ptr	0	6	9	12

distance	0	2	1	1	1	2	0	…
vertex ID	1	2	3	4	5	6	7	…

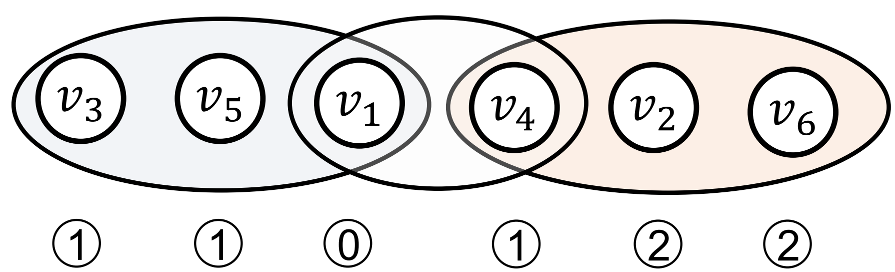
parallel seg. sort

group 1



Balanced Group Coarsening Parallelization

group ptr	0	6	9	12				
distance	0	1	1	1	2	2	0	…
vertex ID	1	3	4	5	2	6	7	…
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_n
in-group ID	0-0=0	1-0=1	2-0=2	3-0=3	4-0=4	5-0=5	6-6=0	

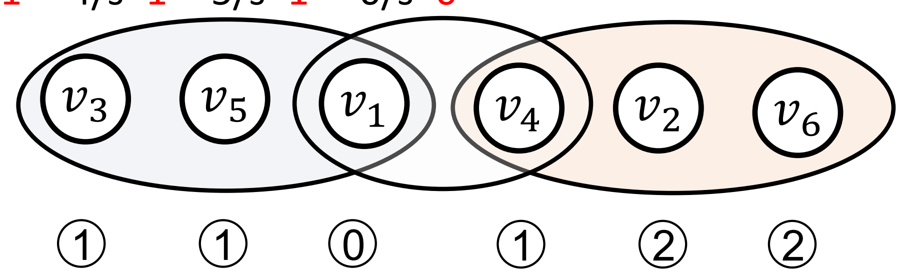


Balanced Group Coarsening Parallelization

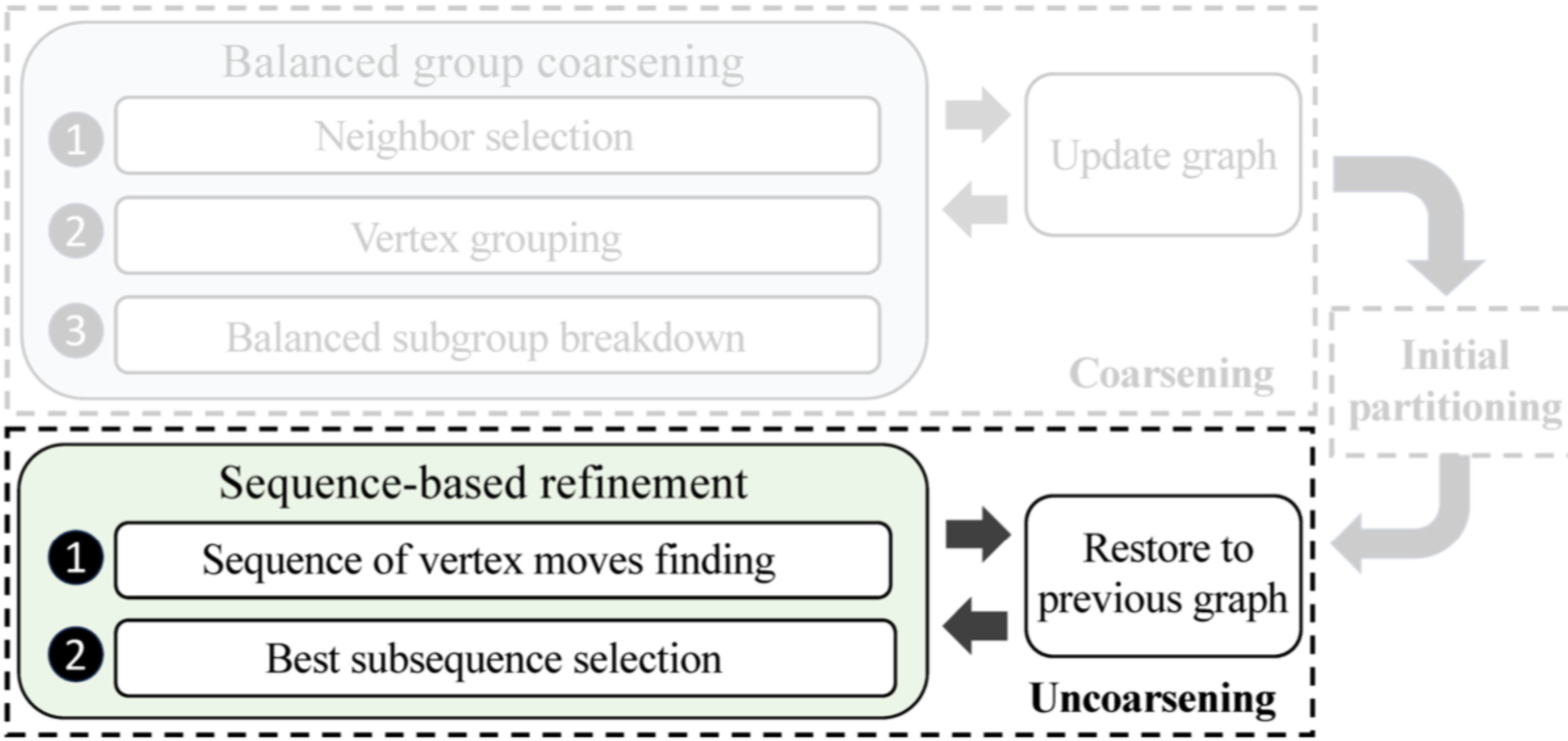
group ptr	0	6	9	12

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_n
distance	0	1	1	1	2	2	0	...
vertex ID	1	3	4	5	2	6	7	...
in-group ID	0-0=0	1-0=1	2-0=2	3-0=3	4-0=4	5-0=5	6-6=0	
subgroup ID	0/s=0	1/s=0	2/s=0	3/s=1	4/s=1	5/s=1	0/s=0	

subgroup size s=3

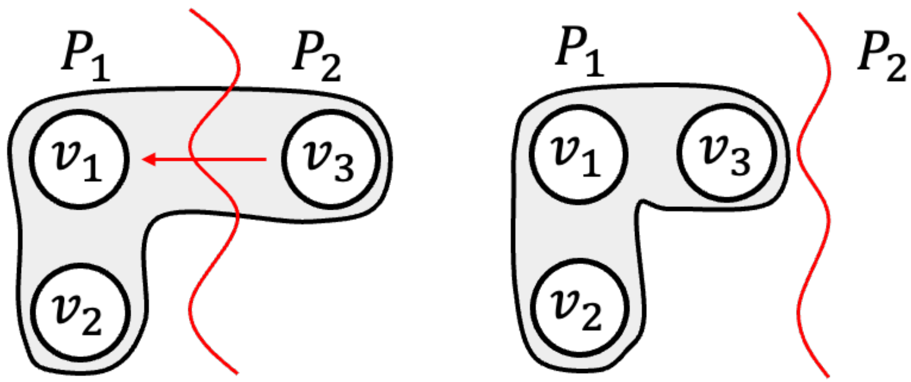


Uncoarsening Stage



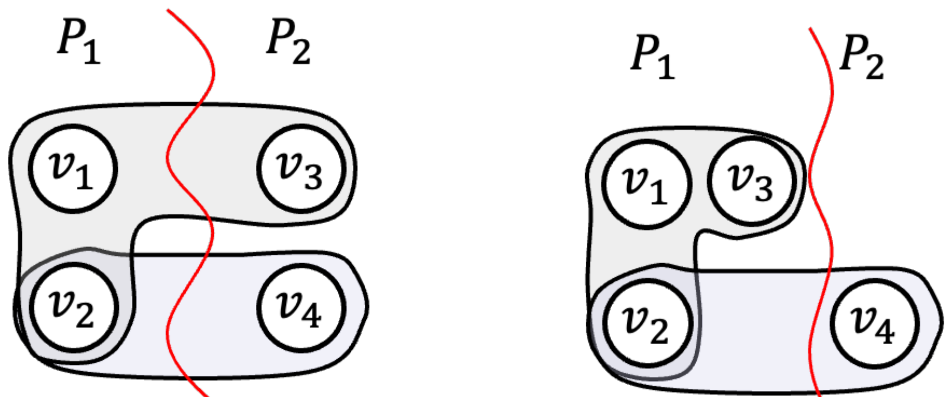
Refinement Overview

- Goal: Minimizes cut hyperedges by moving vertices with positive gains while maintaining a balanced partition
- $gain(u, P_{dst})$: The reduction in cut size if u is moved from its current partition to P_{dst}
- HyperG moves large number of vertices in parallel to speed up the refinement step



Yet, Parallel Refinement is Challenging

- Moving vertices in parallel can …
  - make each vertex’s gain inconsistent due to concurrent movements of adjacent vertices
  - lead to an imbalanced partition



We thought

$gain(v_3, P_1) = 1$

$gain(v_2, P_2) = 1$

But after moving v_3,

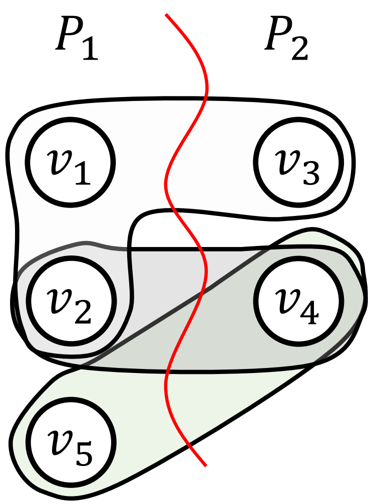
$gain(v_2, P_2) = \cancel{1}\ 0$

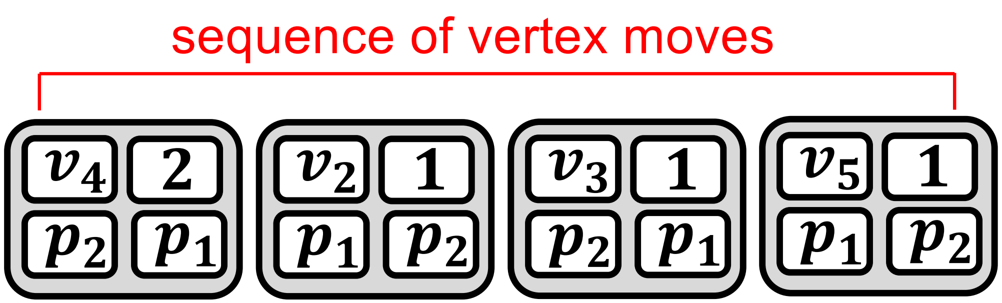
Sequence-based Refinement

- Finds a sequence of vertex moves with positive gains in descending order
  - Prioritizes vertices with larger gains
- Updates each vertex move's gain in the sequence, assuming neighbors with smaller indexes already applied
  - Ensures gains are consistent
- Accumulates gains to identify the best subsequence of vertex moves yielding the largest gain while maintaining balanced partitions
  - Guarantees a balanced partition after moving

Sequence-based Refinement Example

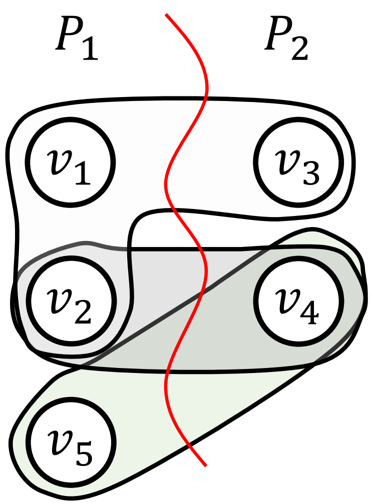
- Finds a sequence of vertex moves with positive gains in descending order

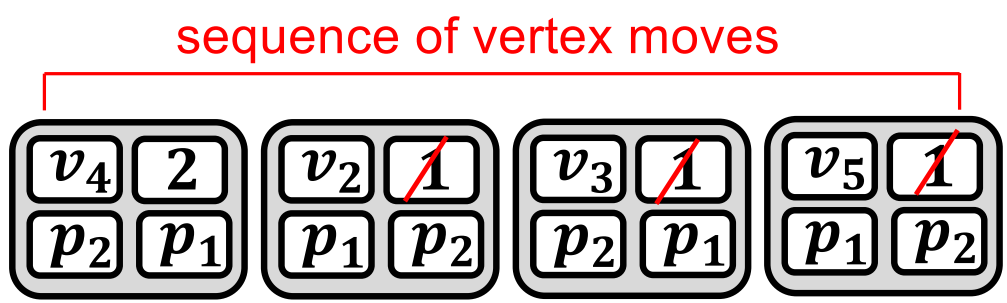




Sequence-based Refinement Example

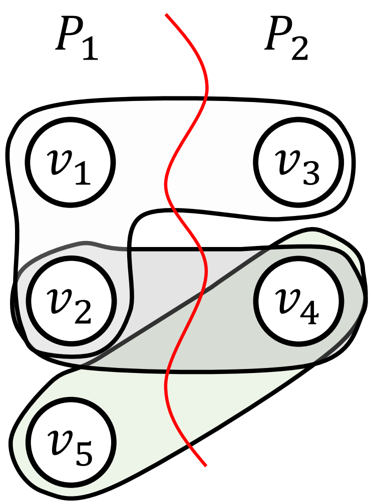
- Updates the gain of each vertex move in the sequence, assuming neighbors with smaller indexes already applied

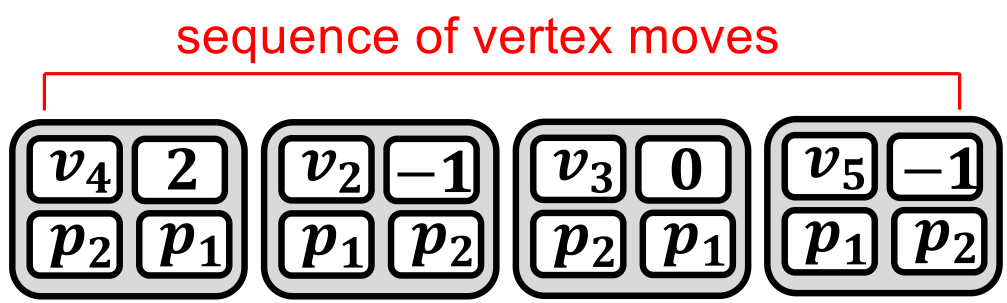




Sequence-based Refinement Example

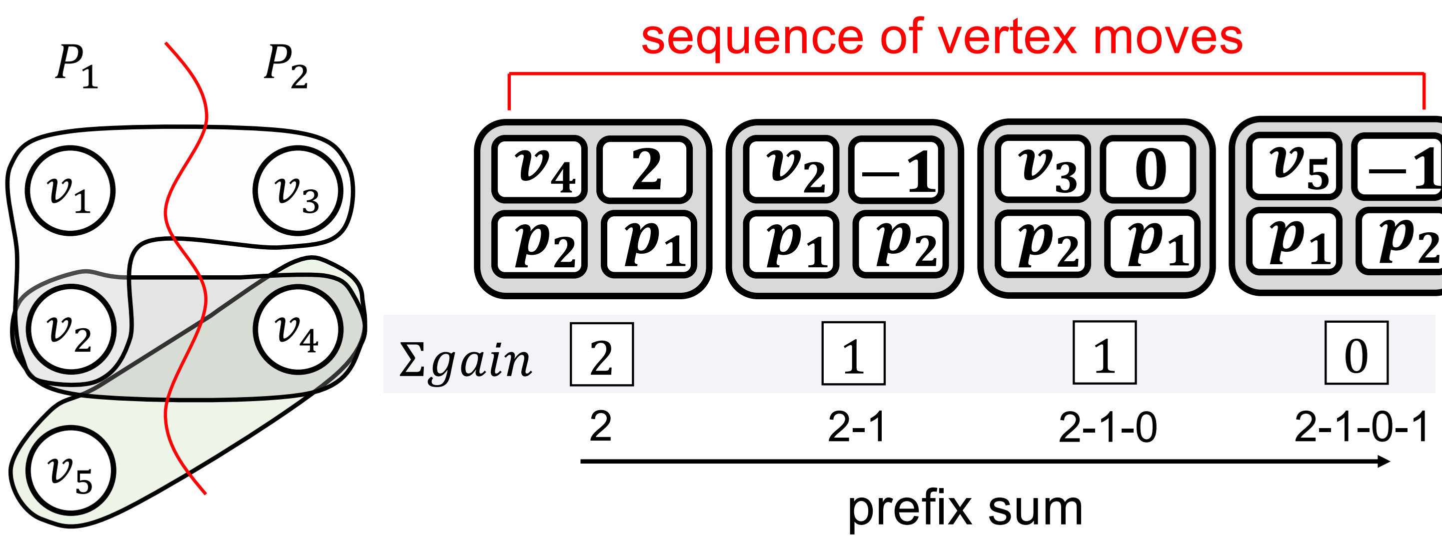
- Updates the gain of each vertex move in the sequence, assuming neighbors with smaller indexes already applied





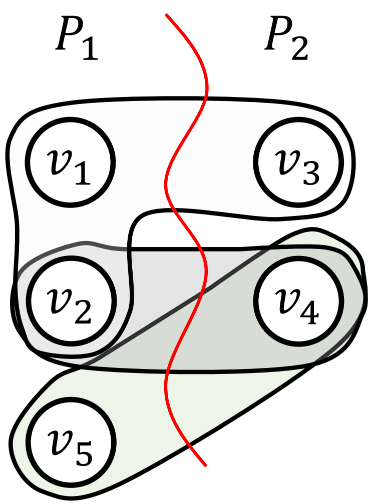
Sequence-based Refinement Example

- Accumulates gains to identify the best subsequence of vertex moves yielding the largest gain while maintaining balanced partitions



Sequence-based Refinement Example

- Accumulates gains to identify the best subsequence of vertex moves yielding the largest gain while maintaining balanced partitions



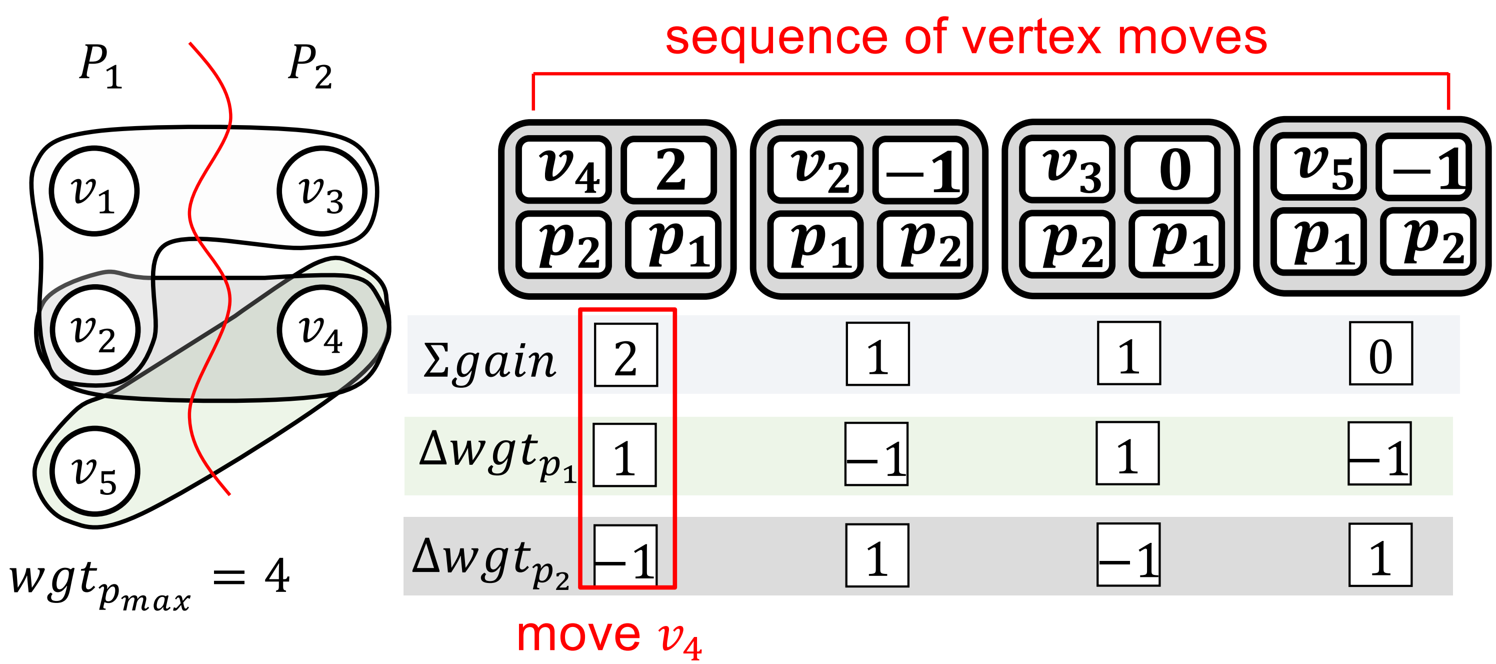
sequence of vertex moves

	v_4 2 p_2 p_1	v_2 -1 p_1 p_2	v_3 0 p_2 p_1	v_5 -1 p_1 p_2
$\Sigma gain$	2	1	1	0
Δwgt_{p_1}	1	−1	1	−1
Δwgt_{p_2}	−1	1	−1	1

prefix sum

Sequence-based Refinement Example

- Accumulates gains to identify the best subsequence of vertex moves yielding the largest gain while maintaining balanced partitions



GPU Optimization

- Uses an array of size $|V|$ to store each vertex's index in the sequence of vertex moves
  - Efficiently locates each vertex's order in the sequence without searching
- Uses warp-level primitives (e.g., $\_\_ballot\_sync$ and $\_\_popc$) to update the gains of vertex moves
  - Assigns each vertex move to a GPU warp
  - Each thread in the warp fetches a neighbor of the vertex moves
  - Simultaneously finds neighbors with smaller indices in the sequence

Experimental Results

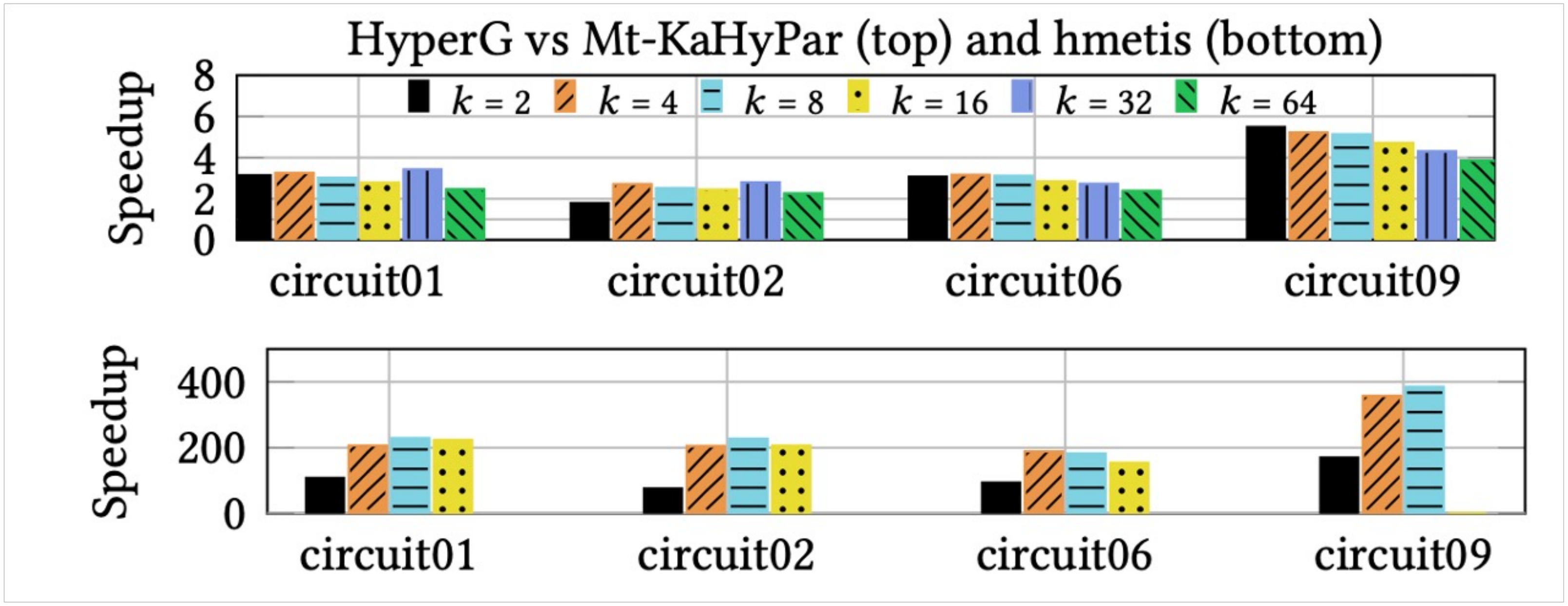
- Baselines
  - Sequential hypergraph partitioner hmetis
  - CPU parallel hypergraph partitioner mt-KaHyPar (16 threads)
- Benchmarks
  - 18 industrial circuit graphs from the ISPD98 VLSI Circuit Benchmark Suite
  - Expands 100–1000 times with random vertex and edge insertions

Overall Performance

Hypergraph benchmark			hmetis (Sequential)		Mt-KaHyPar (16 threads)		HyperG		Speedup vs	
Name	# Vertices	# Edges	Time (s)	Cut size	Time (s)	Cut size	Time (s)	Cut size	hmetis	Mt-KaHyPar
circuit01	2,639,664	2,920,977	83.359	**1,480**	2.415	1,513	0.770	1,498	108.3×	3.1×
circuit02	5,076,659	5,072,256	246.654	**1,570**	5.784	1,597	1.692	1,572	145.8×	3.4×
circuit03	3,215,904	3,808,739	116.118	1,666	3.368	1,666	0.908	**1,665**	127.9×	3.7×
circuit04	3,273,333	3,804,430	114.703	**1,686**	3.440	1,693	1.567	1,699	73.2×	2.2×
circuit05	5,898,747	5,717,646	243.087	815	6.608	828	2.134	**814**	113.9×	3.1×
circuit06	5,817,142	6,233,854	235.252	1,708	7.179	1,692	1.351	**1,691**	174.1×	5.3×
circuit07	5,648,898	5,918,391	208.098	**1,744**	6.717	**1,744**	1.318	1,745	157.9×	5.1×
circuit08	2,001,051	1,970,007	67.163	**853**	1.734	854	1.896	**853**	35.4×	0.9×
circuit09	4,965,735	5,663,886	175.453	**1,784**	5.652	1,794	1.031	1,794	170.2×	5.5×
circuit10	6,179,181	6,692,444	251.446	**1,804**	7.855	1,807	1.526	1,808	164.8×	5.1×
circuit11	5,856,314	6,760,682	210.619	**1,802**	7.155	**1,802**	**1.197**	**1,802**	176.0×	6.0×
circuit12	3,767,028	4,093,720	141.953	**1,801**	4.237	1,806	1.956	1,811	72.6×	2.2×
circuit13	3,620,557	4,285,638	122.969	1,836	4.322	**1,835**	0.998	**1,835**	123.2×	4.3×
circuit14	4,163,763	12,487,976	176.301	1,848	6.194	1,848	1.197	**1,802**	147.3×	5.2×
circuit15	5,166,175	5,347,020	264.711	**1,859**	8.505	**1,862**	2.136	**1,859**	123.9×	4.0×
circuit16	7,889,812	8,172,064	338.495	1,868	10.840	**1,866**	2.005	**1,866**	168.8×	5.4×
circuit17	11,686,185	11,943,603	N/A	N/A	18.168	**1,857**	3.225	1,861	N/A	5.6×
circuit18	7,371,455	7,067,200	122.969	**1,852**	9.899	1,853	2.191	**1,852**	177.1×	4.3×
Average									133.0×	4.1×

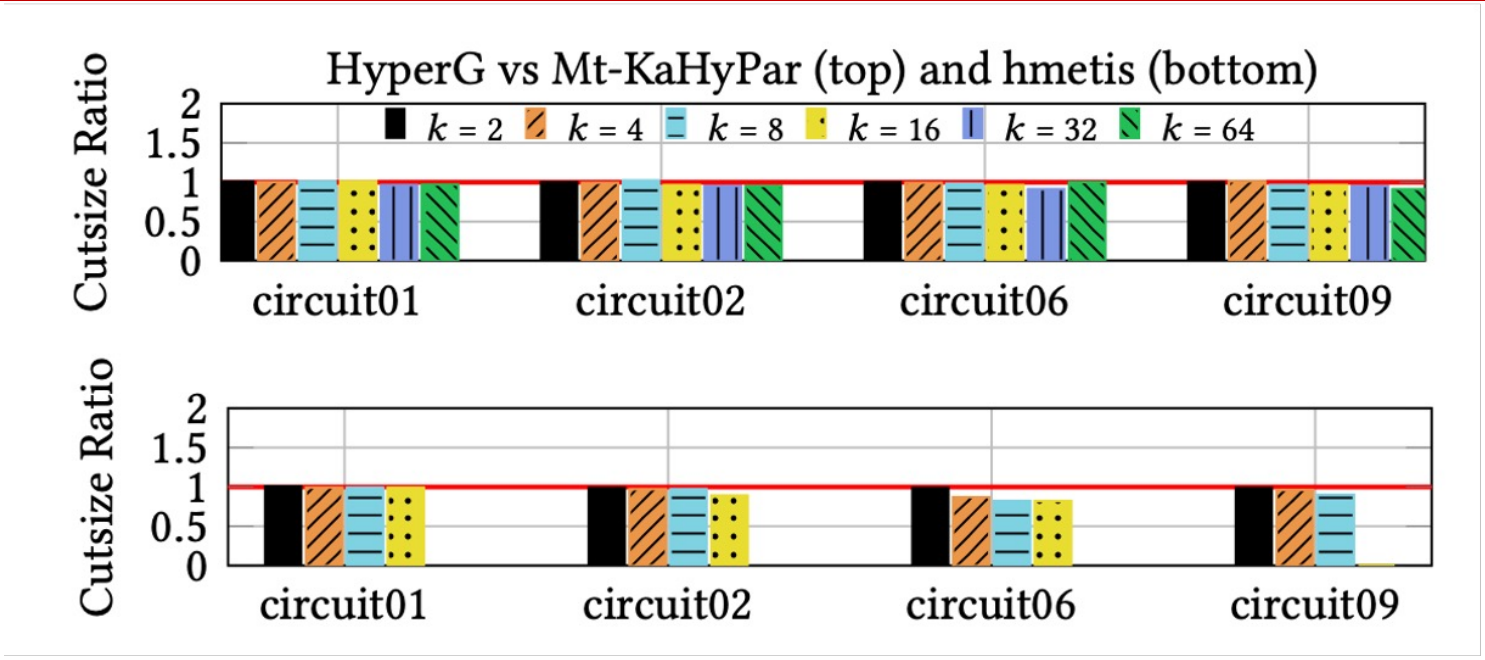
Overall runtime comparison at k = 2

Runtime Analysis



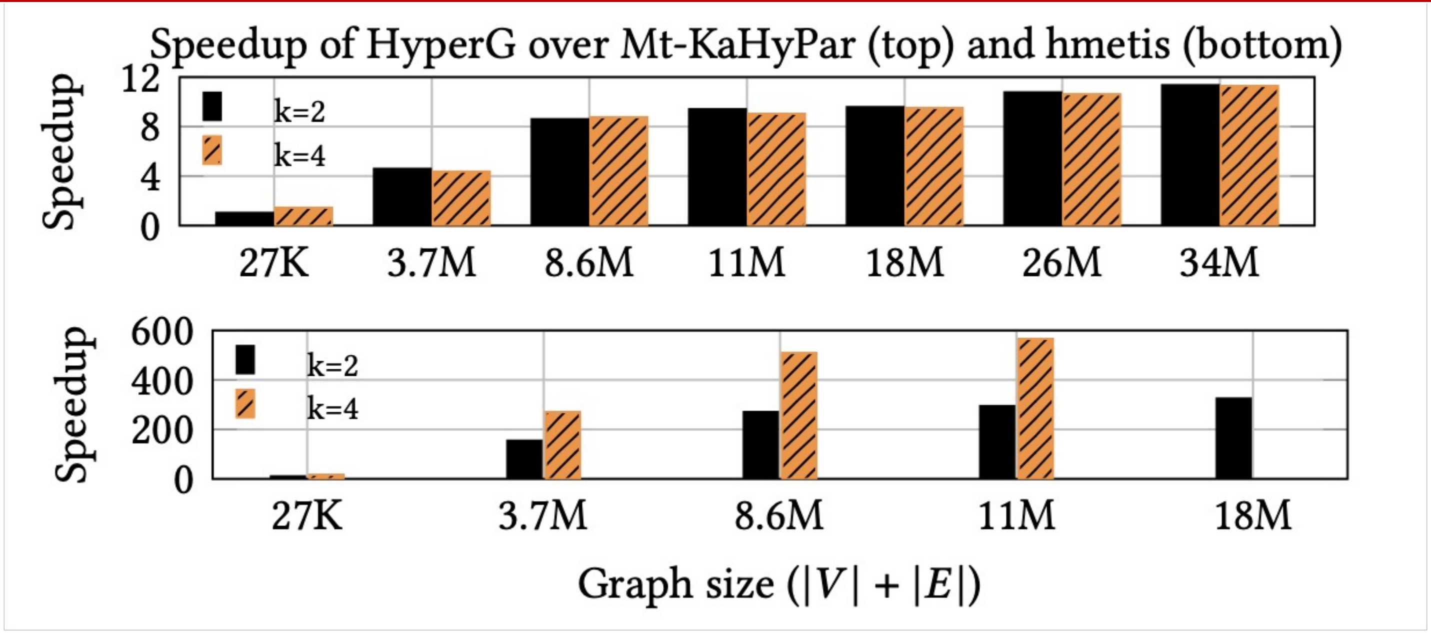
The speedup of HyperG over Mt-KaHyPar (top) and hmetis (bottom) at different k

Cut Size Analysis



Cut size ratio of HyperG to Mt-KaHyPar (top) and hMETIS (bottom) at different k. Results are left blank where hMETIS fails to partition the circuit graph

Scalability Analysis



Speedup of HyperG over Mt-KaHyPar (top) and hMETIS (bottom) for varying circuit graph sizes modified from ibm01 at $k = 2$ and $k = 4$. hMETIS fails to partition circuit graphs larger than 18M