Parallel k-Core Decomposition: Theory and Practice

Youzhe Liu, Xiaojun Dong, Yan Gu, Yihan Sun

https://www.youtube.com/watch?v=eU0YPJNneNI

Authors

Youzhe Liu

Xiaojun Dong

Yan Gu

Yihan Sun

https://open.spotify.com/playlist/4ntQld1Fr9O7myiNgKPa81?si=554114910c4e4a43&nd=1&dlsi=149351c67e684c7b
https://open.spotify.com/playlist/7APYv0SnF5VycR0VMpOjzy?si=b81226f11f614f5f&nd=1&dlsi=2a2eedd091b24d94

Motivation

- k-core decomposition has a wide range of important applications
- Modern graphs are becoming increasingly large
- Modern hardware is becoming increasingly parallel
- SOTA parallel implementations of k-core decomposition exhibit worse than sequential performance on variety of graphs.

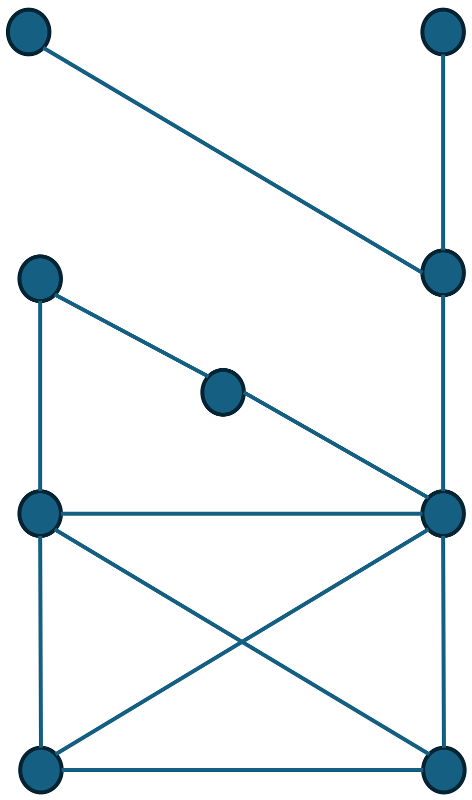
Definitions

- Given an undirected graph G = (V,E), k-core of G is the maximal subgraph in which every vertex has degree at least k
- k-core decomposition identifies the sequence of non-empty subgraphs for all k values
- Coreness of a vertex k[v] is the maximum k-core containing v
- Output: k[v] for $v \in V$

- Sequential Solution:
  - Pop vertex with degree < k and decrement each neighbor’s current degree
  - O(V+E)

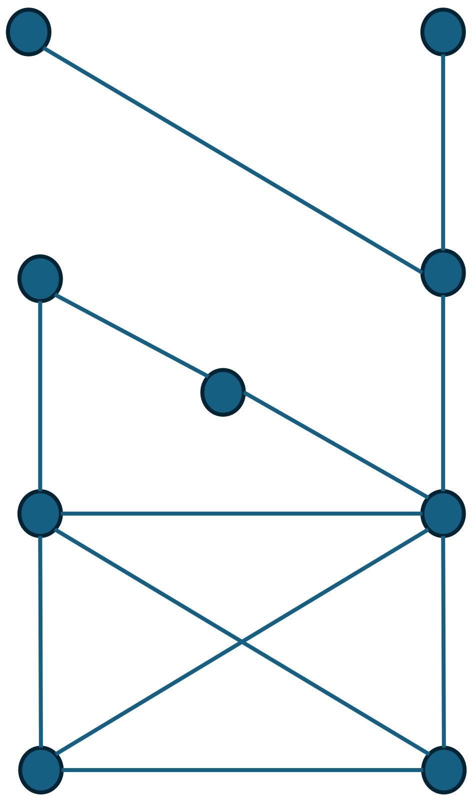
k-Core Decomposition

Computing 1-Core



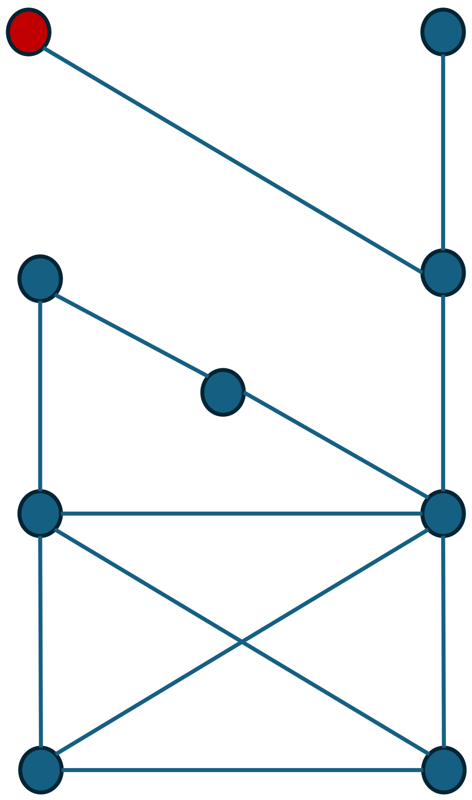
k-Core Decomposition

Computing 2-Core



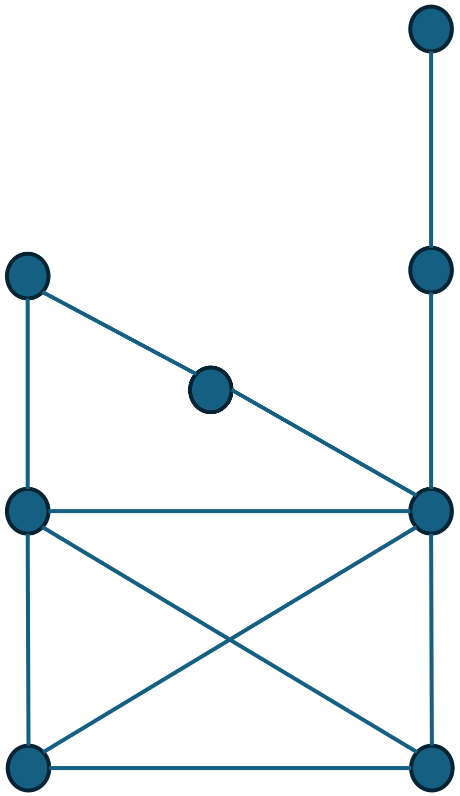
k-Core Decomposition

Computing 2-Core



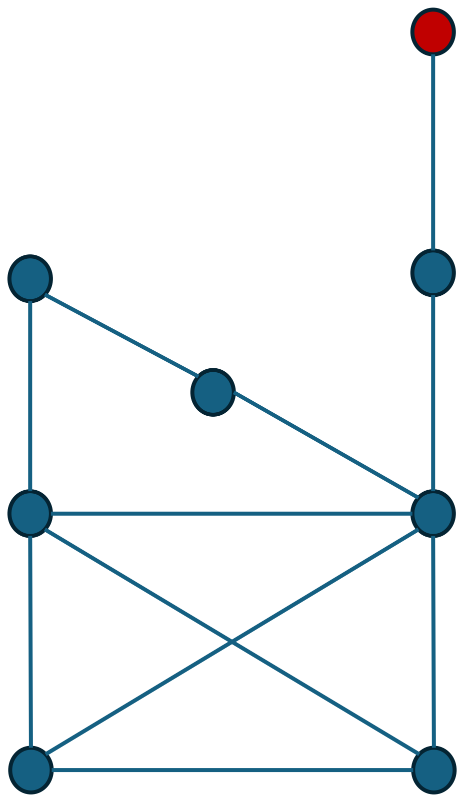
k-Core Decomposition

Computing 2-Core



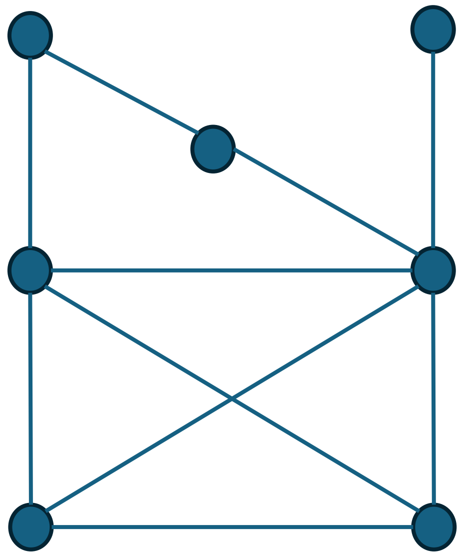
k-Core Decomposition

Computing 2-Core



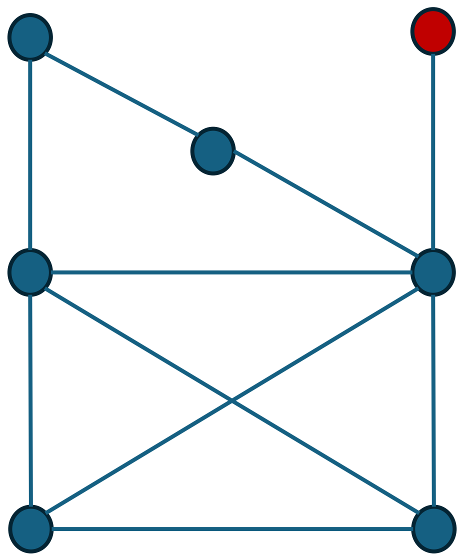
k-Core Decomposition

Computing 2-Core



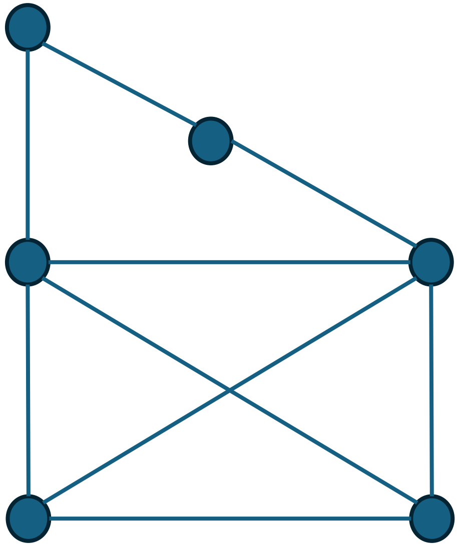
k-Core Decomposition

Computing 2-Core



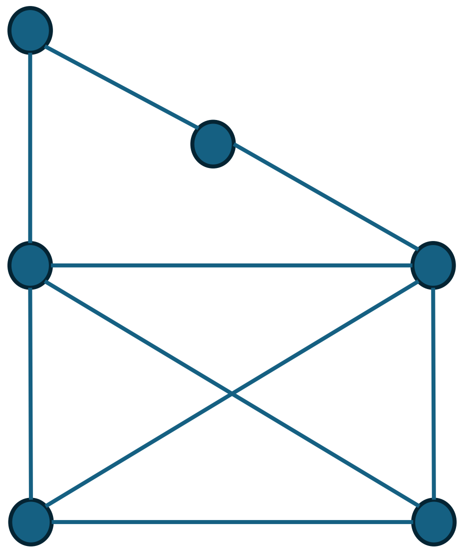
k-Core Decomposition

Computing 2-Core



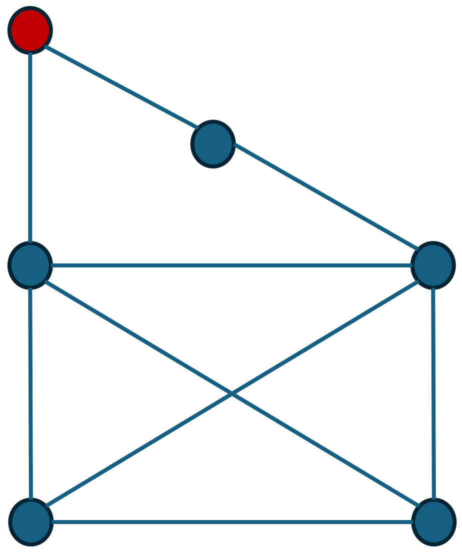
k-Core Decomposition

Computing 3-Core



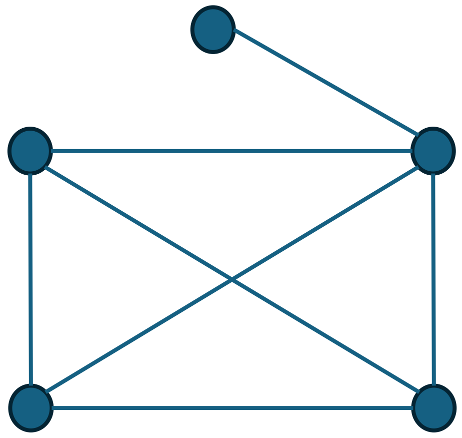
k-Core Decomposition

Computing 3-Core



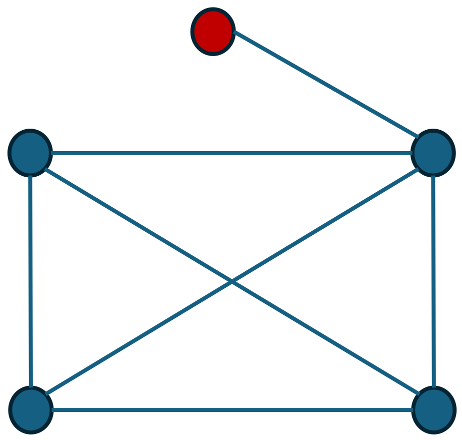
k-Core Decomposition

Computing 3-Core



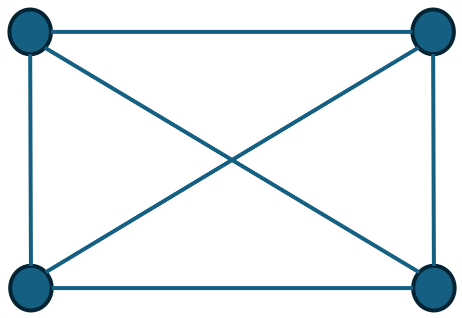
k-Core Decomposition

Computing 3-Core



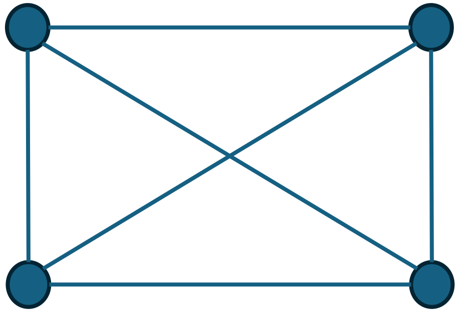
k-Core Decomposition

Computing 3-Core



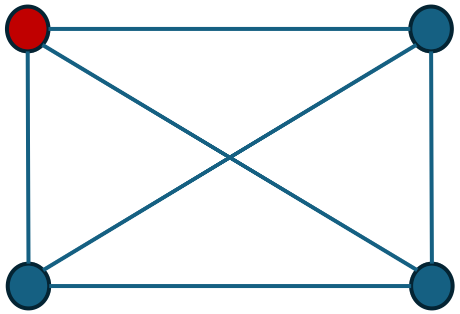
k-Core Decomposition

Computing 4-Core



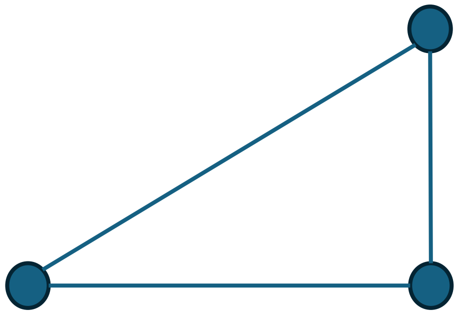
k-Core Decomposition

Computing 4-Core



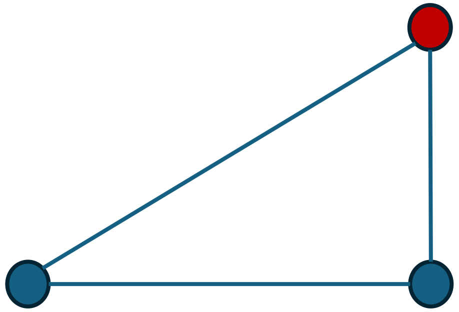
k-Core Decomposition

Computing 4-Core



k-Core Decomposition

Computing 4-Core



k-Core Decomposition

Computing 4-Core

k-Core Decomposition

Computing 4-Core

k-Core Decomposition

Computing 4-Core

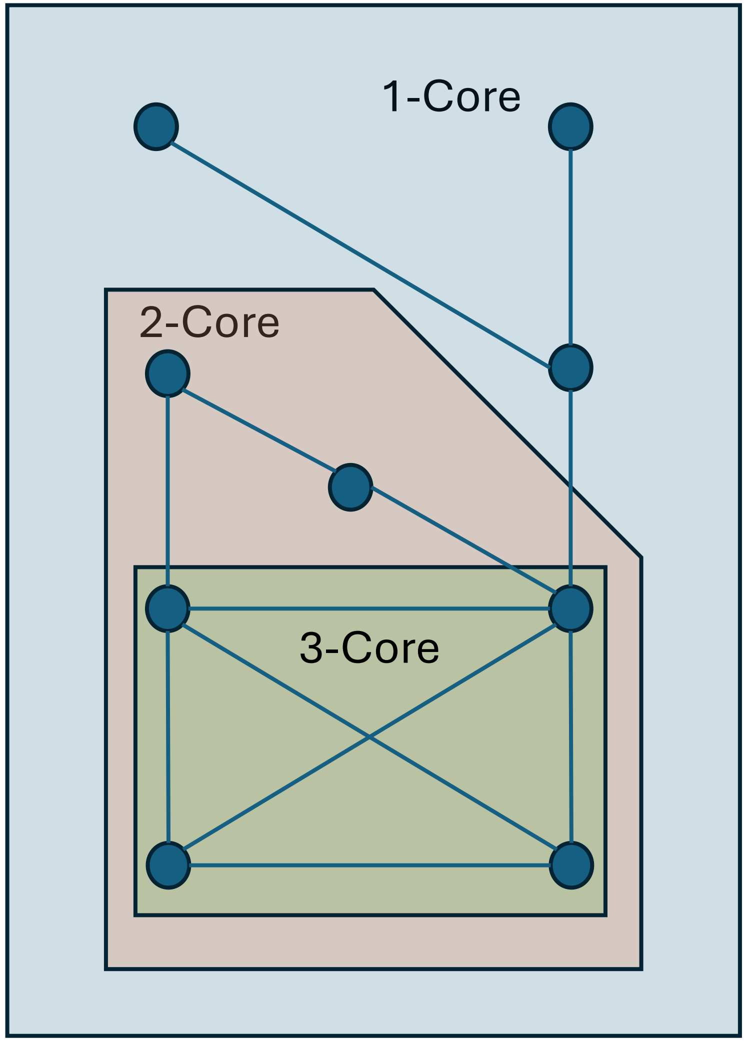
k-Core Decomposition

Computing 4-Core

k-Core Decomposition

Computing 4-Core

k-Core Decomposition



k-Core Decomposition

Algorithm 1: Work-efficient parallel k-core framework

Input: Graph $G = (V, E)$
Output: The coreness for each vertex

$\tilde{d}[\cdot] \leftarrow d(\cdot)$ *// Initialize the induced degree set of all vertices*
$\mathcal{A} \leftarrow V$ *// Active set: vertices that have not been peeled*
$k \leftarrow 0$
while $\mathcal{A} \neq \emptyset$ **do**
    $\mathcal{F} \leftarrow \{v \mid v \in \mathcal{A}, \tilde{d}[v] = k\}$ *// The initial frontier in round k*
    while $\mathcal{F} \neq \emptyset$ **do**
        foreach $v \in \mathcal{F}$ **do** $\kappa[v] \leftarrow k$ *// Sets coreness to k*
        $\mathcal{F} \leftarrow \textsc{Peel}(\mathcal{F}, k)$
    $\mathcal{A} \leftarrow \{v \mid v \in \mathcal{A}, \tilde{d}[v] > k\}$ *// Refines the active set*
    $k \leftarrow k + 1$
return $\kappa[\cdot]$

/ A sequential version. Parallel versions are discussed in Sec. 3.2. */*
Function $\textsc{Peel}(\mathcal{F}, k)$
    Initialize $\mathcal{F}_{\text{next}} \leftarrow \emptyset$ *// Buffers the next frontier*
    foreach $v \in \mathcal{F}$ **do**
        foreach $u \in N(v)$ **do**
            $\tilde{d}[u] \leftarrow \tilde{d}[u] - 1$
            if $\tilde{d}[u] = k$ **then** Add u to $\mathcal{F}_{\text{next}}$
    return $\mathcal{F}_{\text{next}}$ *// Returns the next frontier*

Existing Peeling Implementations

- Offline (Julienne)
  - Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2017
  - Collect all vertices requiring a degree decrement, compute histogram, and subtract in parallel
  - Requires explicit synchronization between sub-rounds = Bad on high-diameter graphs
- Online (PKC, ParK)
  - Humayun Kabir and Kamesh Madduri. 2017 (PKC)
  - Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014 (ParK)
  - When peeling vertex v, atomically decrement induced degree
  - Work per vertex varies with degree distribution
  - High degree vertices can experience contention = Bad for power-law degree graphs

Existing Peeling Implementations

Algorithm 2: Offline peeling process

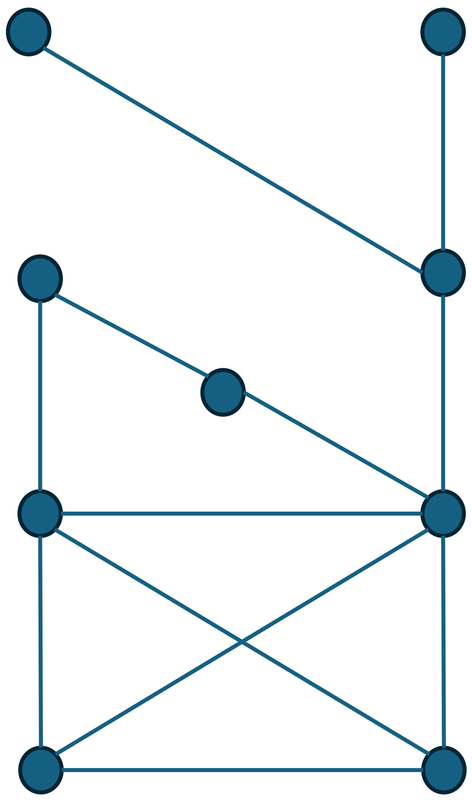
1 **Function** PEEL($\mathcal{F}, k$)
2 $L \leftarrow$ the list of vertices u, s.t. $(u, v) \in E, v \in \mathcal{F}$; duplicates are kept.
3 $H \leftarrow$ HISTOGRAM(L) *// Count the frequency for each u in L*
4 **parallel_for** $(u, f_u) \in H$ **do** *// for each u with frequency f_u*
5 **if** $\tilde{d}[u] > k$ **then** $\tilde{d}[u] \leftarrow \tilde{d}[u] - f_u$
6 $\mathcal{F}_{\text{next}} \leftarrow \{u \in L, \tilde{d}[u]$ dropped to k or lower by line 5 $\}$
7 **return** $\mathcal{F}_{next}$

Algorithm 3: Online peeling process

Function PEEL($\mathcal{F}, k$)
Initialize $\mathcal{F}_{\text{next}} \leftarrow \emptyset$ *// Buffers the next frontier*
parallel_foreach $v \in \mathcal{F}$ **do**
parallel_foreach $u \in N(v)$ **do**
$\delta \leftarrow$ atomic_dec($\tilde{d}[u]$) *// decrement atomically*
// The last decrement adds u to the next frontier
if $\delta = k + 1$ **then** Add u to $\mathcal{F}_{\text{next}}$
return $\mathcal{F}_{\text{next}}$ *// Returns the next frontier*

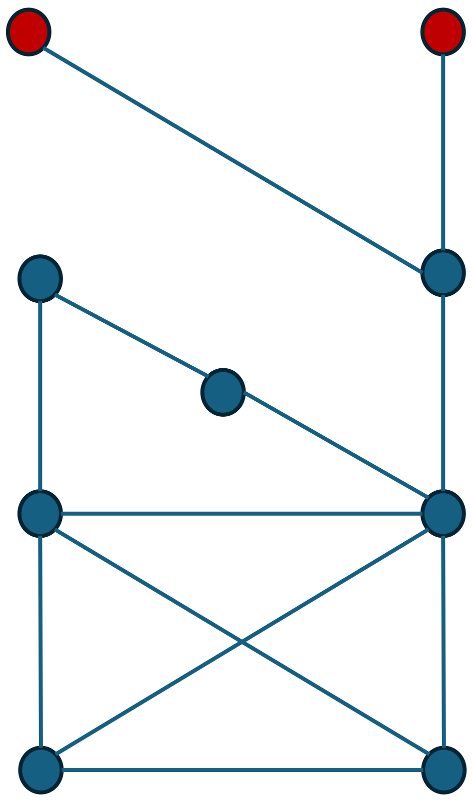
k-Core Decomposition

Computing 2-core



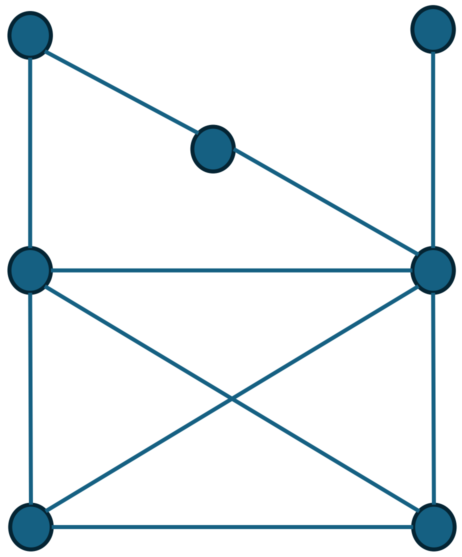
k-Core Decomposition

Computing 2-core



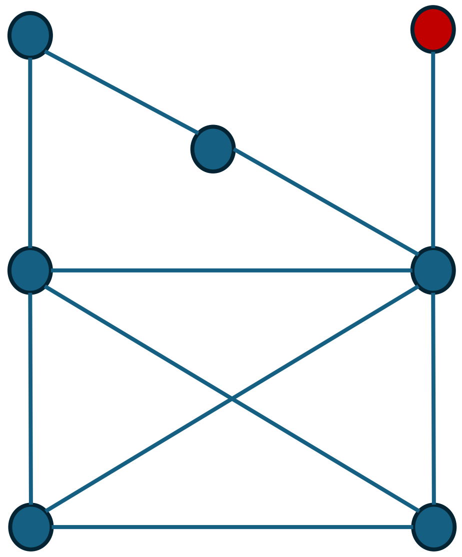
k-Core Decomposition

Computing 2-core



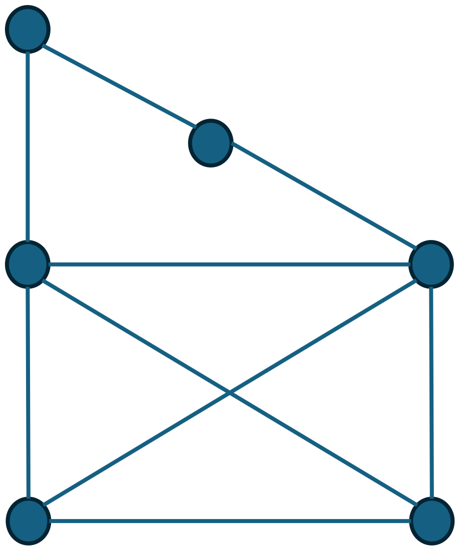
k-Core Decomposition

Computing 2-core



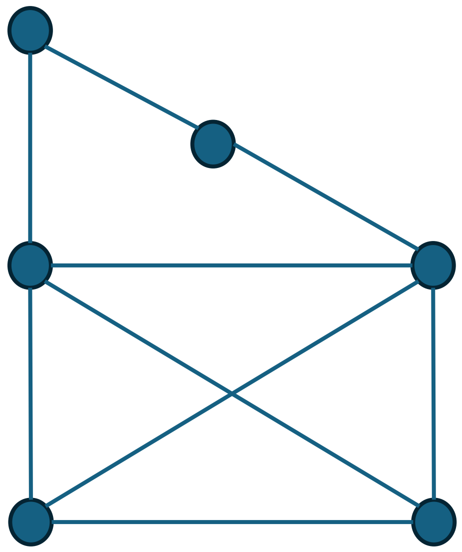
k-Core Decomposition

Computing 2-core



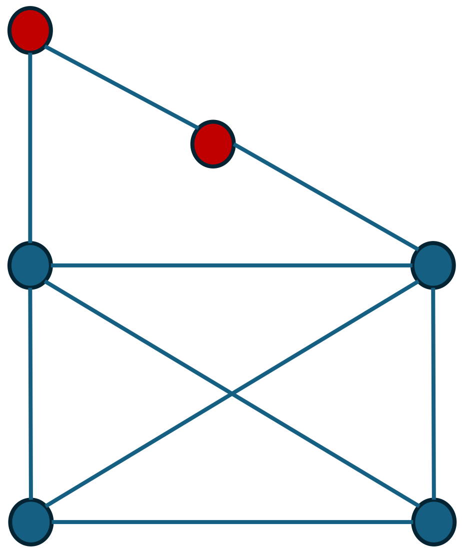
k-Core Decomposition

Computing 3-core



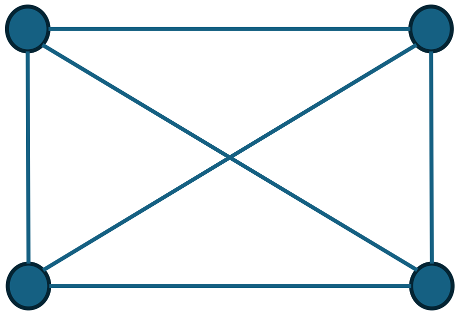
k-Core Decomposition

Computing 3-core



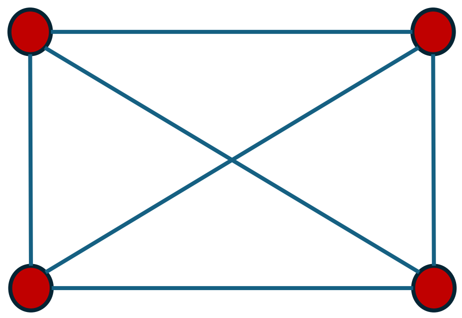
k-Core Decomposition

Computing 4-core



k-Core Decomposition

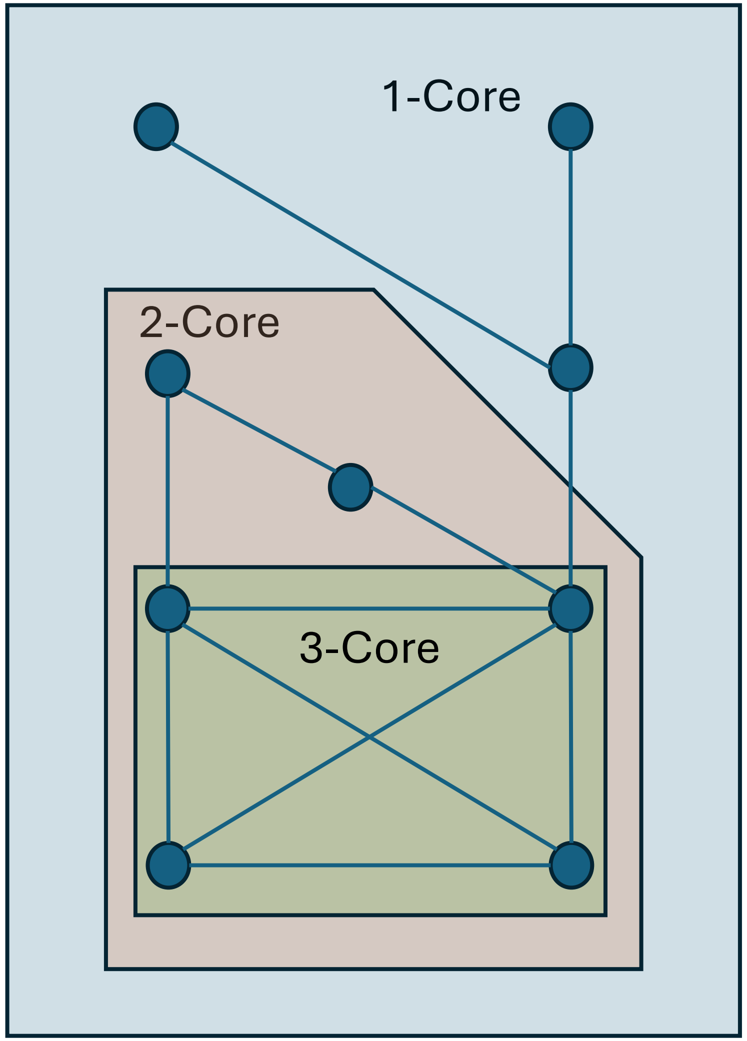
Computing 4-core



k-Core Decomposition

Computing 4-core

k-Core Decomposition



Idea 1: Sampling to Reduce Contention

- When the degree of a vertex is above some threshold, apply a random sample of decrements with some fixed probability
  - Approximate induced degree with high probability
  - Reduces contention
- Sampler structure
  - Sample mode (true / false)
  - Sample rate (probability of incrementing count)
  - Sample count (number of samples taken)

Idea 1: Sampling to Reduce Contention

Algorithm 4: Our algorithm framework with sampling

Input: Graph $G = (V, E)$. **Output:** $\kappa[\cdot]$: Coreness of each vertex Initialize $\mathcal{A} \leftarrow V$; $k \leftarrow 0$; $\tilde{d}[v] \leftarrow d(v)$ for all $v \in V$

parallel_foreach $v \in V$ **do** $\textsc{SetSampler}(v, 0)$ *// Initialize $\sigma[v]$*

while $\mathcal{A} \neq \emptyset$ **do**

  $\mathcal{F} \leftarrow \{v \mid v \in \mathcal{A}, \tilde{d}[v] = k\}$

  parallel_foreach $v \in V$: *v is in sample mode* **do**

    if $\textsc{Validate}(v, k) = \text{false}$ **then** $\textsc{Resample}(v, k, \mathcal{F})$

  while $\mathcal{F} \neq \emptyset$ **do**

    parallel_foreach $v \in \mathcal{F}$ **do** $\kappa[v] \leftarrow k$

    $\langle \mathcal{F}, C \rangle \leftarrow \textsc{Peel}(\mathcal{F}, k)$ *// C: vertices that require to reset samplers*

    parallel_foreach $v \in C$ **do** $\textsc{Resample}(v, k, \mathcal{F})$

  $\mathcal{A} \leftarrow \{v \mid v \in \mathcal{A}, \tilde{d}[v] > k\}$

  $k \leftarrow k + 1$

return $\kappa[\cdot]$

Idea 1: Sampling to Reduce Contention

Function SetSampler(v, k)

  // If the expected induced degree of v is still large and far from k even after decrementing to a factor of r, then v can be sampled safely.

  if $(\tilde{d}[v] \cdot r > k) \wedge (\tilde{d}[v] > \mathit{threshold})$ **then**

    $\sigma[v].\mathit{mode} \leftarrow \mathit{true}$

    // Set the sample rate. This formula is explained in Sec. 4.1.

    $\sigma[v].\mathit{rate} \leftarrow \mu/((1 - r) \cdot \tilde{d}[v])$

    $\sigma[v].\mathit{cnt} \leftarrow 0$

  else $\sigma[v].\mathit{mode} \leftarrow \mathit{false}$

Function Resample$(v, k, \mathcal{F})$

  $\tilde{d}[v] \leftarrow$ the number of active vertices in $N(v)$

  if $\tilde{d}[v] \leq k$ **then** Add v to $\mathcal{F}$

  SetSampler(v, k)

Function Validate(v, k) *// Explained in Sec. 4.1.3*

  return $(\tilde{d}[v] \cdot r > k) \wedge (\sigma[v].\mathit{cnt} < \sigma[v].\mathit{rate} \cdot (\tilde{d}[v] - k)/4)$

Idea 1: Sampling to Reduce Contention

Algorithm 5: Functions used in our algorithm with sampling

Parameters: r: when $\tilde{d}[v]$ decrement to a factor of r, we resample v
$\mu = 4c \ln n$: expected number of hits each sampler, $c > 2$

struct *sampler* // *For each* $v \in V$, *maintain a sampler structure* $\sigma[v]$
  mode: boolean flag indicating whether v is in sample mode
  rate: the sample rate for v
  cnt: the number of hits in the sampling process

Function Peel($\mathcal{F}, k$) // *peeling process with sampling*
  $\mathcal{F}_{\text{next}} \leftarrow \emptyset$; $C \leftarrow \emptyset$ // *C: vertices to recount their induced degrees*
  parallel_foreach $v \in \mathcal{F}$ **do**
    parallel_foreach $u \in N(v)$ **do**
      if $\sigma[u]$.*mode* **then**
        $\delta \leftarrow$ atomic_inc($\sigma[u]$.*cnt*) with probability $\sigma[u]$.*rate*
        // *If sufficient samples are collected, add u to C*
        if $\delta = \mu - 1$ **then** Add u to C
      else
        $\delta \leftarrow$ atomic_dec($\tilde{d}[u]$)
        if $\delta = k + 1$ **then** Add u to $\mathcal{F}_{\text{next}}$
  return $\langle \mathcal{F}_{\text{next}}, C \rangle$

Idea 2: Vertical Granularity Control

Problem

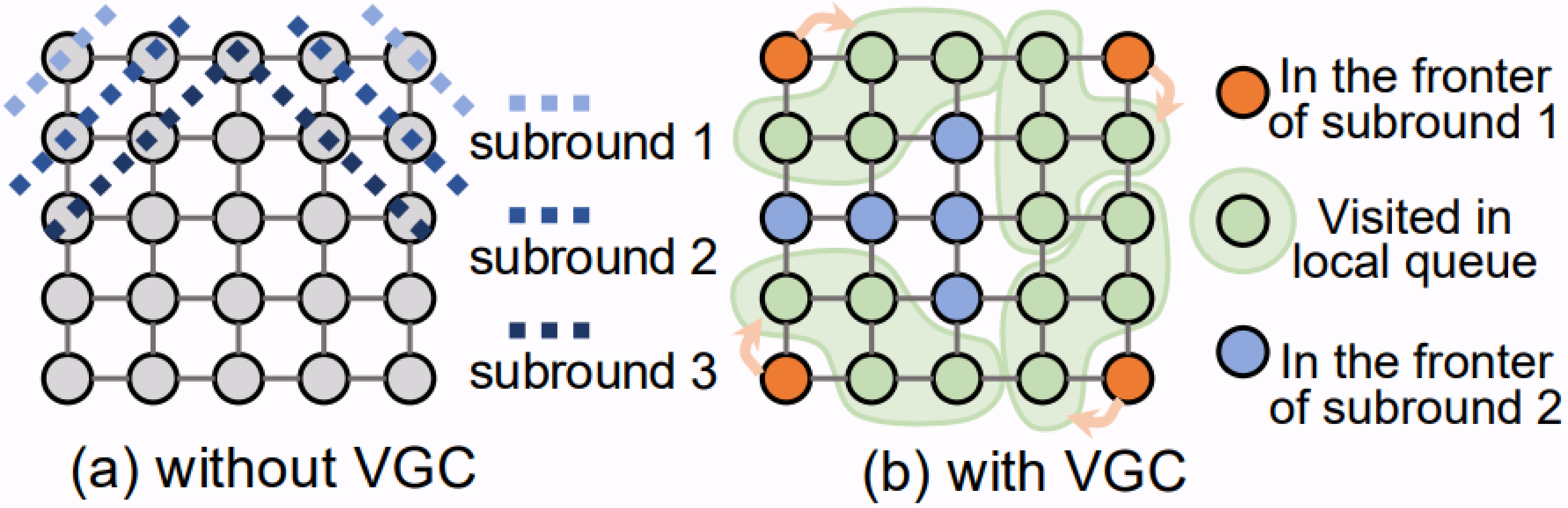
- When we peel a low degree vertex, the computation to process all its neighbors may be less than the overhead of creating and synchronizing the thread
  - In practice, each fork/join operation incurs a (large) constant cost
  - Burdened span (Cilkview): Cost of ω for fork/join operation

Idea 2: Vertical Granularity Control

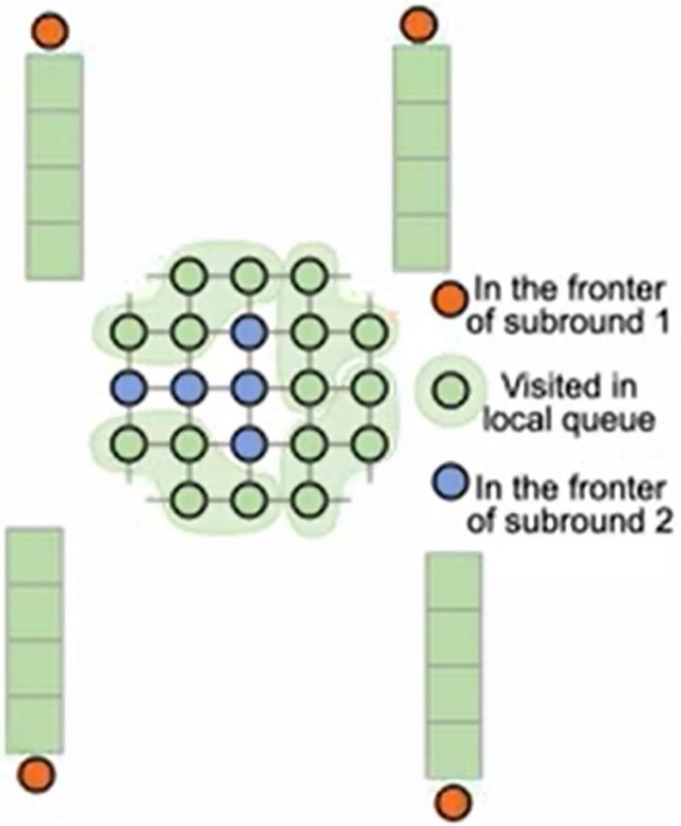
Solution

- When peeling low-deg vertex v, we place neighbors falling below the degree threshold into a FIFO queue
  - Do not add to frontier
  - Processed sequentially in the same subround

Idea 2: Vertical Granularity Control



Idea 2: Vertical Granularity Control



Idea 2: Vertical Granularity Control

- Reduce number of subrounds
- Better load balancing across threads
- Does not affect work efficiency
- Improves performance on sparse graphs

Idea 3: Hierarchical Bucketing Structure

- Recomputing active vertices each round is slow in practice
- Instead, maintain a bucket from each i to all active vertices with d[v] = i
- Move vertices between buckets on decrement

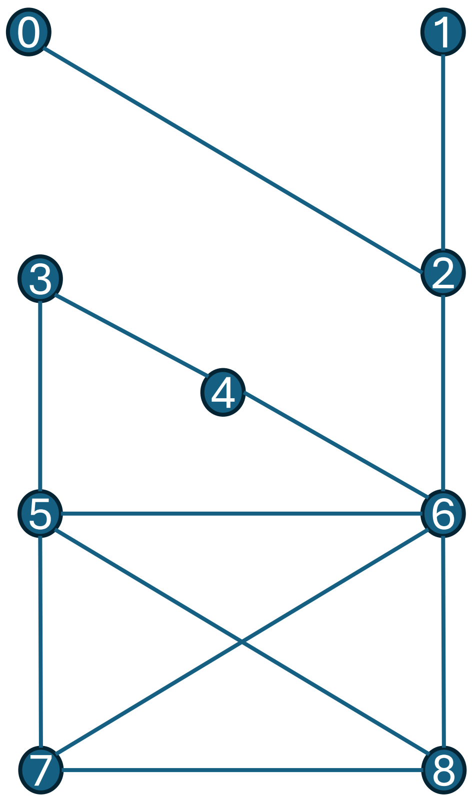
Algorithm 1: Work-efficient parallel k-core framework

Input: Graph $G = (V, E)$
Output: The coreness for each vertex

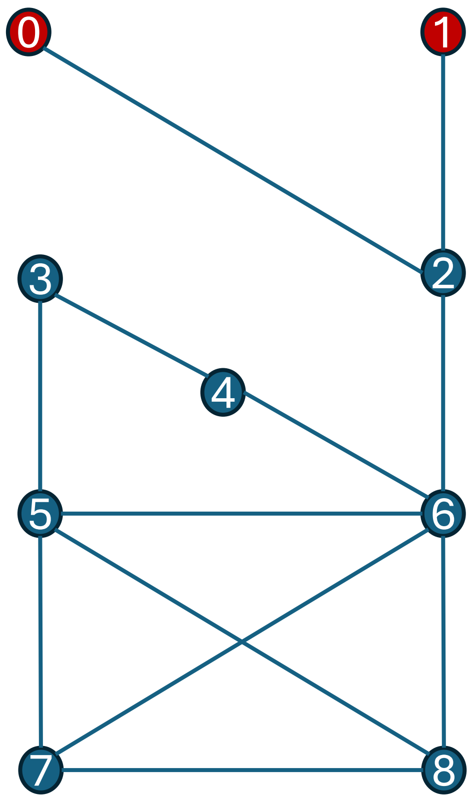
$\tilde{d}[\cdot] \leftarrow d(\cdot)$ *// Initialize the induced degree set of all vertices*
$\mathcal{A} \leftarrow V$ *// Active set: vertices that have not been peeled*
$k \leftarrow 0$
while $\mathcal{A} \neq \emptyset$ **do**
  $\mathcal{F} \leftarrow \{v \mid v \in \mathcal{A}, \tilde{d}[v] = k\}$ *// The initial frontier in round k*
  while $\mathcal{F} \neq \emptyset$ **do**
    foreach $v \in \mathcal{F}$ **do** $\kappa[v] \leftarrow k$ *// Sets coreness to k*
    $\mathcal{F} \leftarrow \textsc{Peel}(\mathcal{F}, k)$
  $\mathcal{A} \leftarrow \{v \mid v \in \mathcal{A}, \tilde{d}[v] > k\}$ *// Refines the active set*
  $k \leftarrow k + 1$
return $\kappa[\cdot]$

/ A sequential version. Parallel versions are discussed in Sec. 3.2. */*
Function $\textsc{Peel}(\mathcal{F}, k)$
  Initialize $\mathcal{F}_{\text{next}} \leftarrow \emptyset$ *// Buffers the next frontier*
  foreach $v \in \mathcal{F}$ **do**
    foreach $u \in N(v)$ **do**
      $\tilde{d}[u] \leftarrow \tilde{d}[u] - 1$
      if $\tilde{d}[u] = k$ **then** Add u to $\mathcal{F}_{\text{next}}$
  return $\mathcal{F}_{\text{next}}$ *// Returns the next frontier*

Idea 3: Hierarchical Bucketing Structure

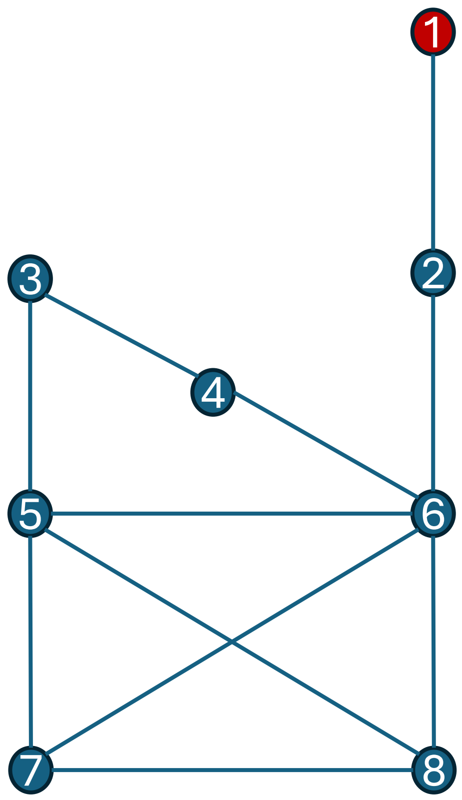


Idea 3: Hierarchical Bucketing Structure



GetNextBucket()

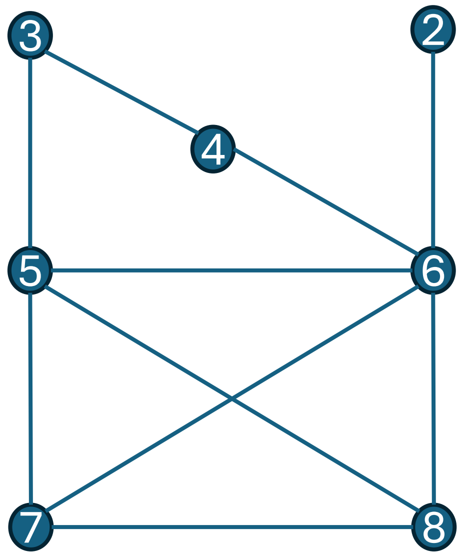
Idea 3: Hierarchical Bucketing Structure



DecreaseKey(2)

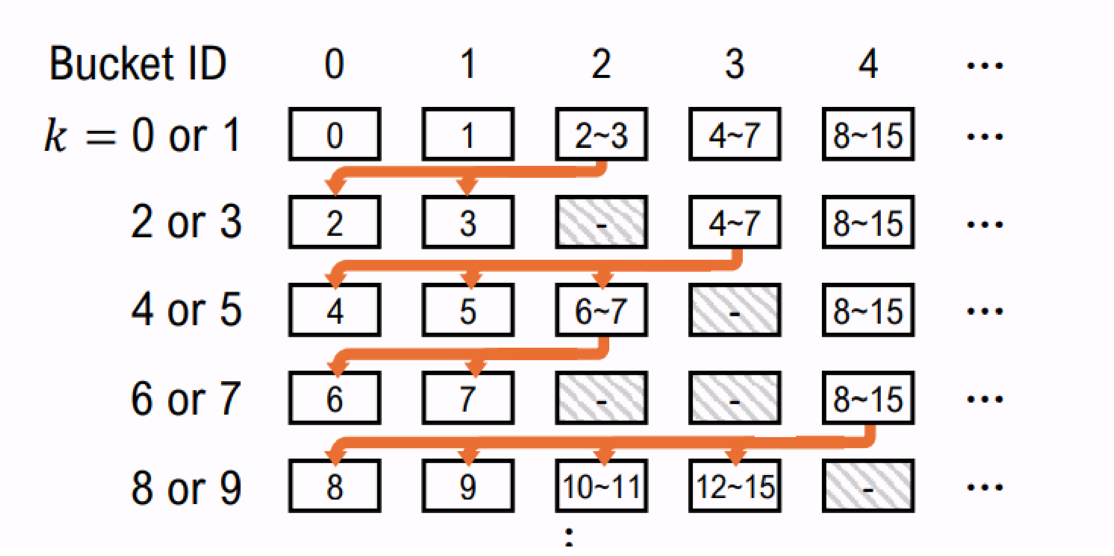
Idea 3: Hierarchical Bucketing Structure

DecreaseKey(2)



Idea 3: Hierarchical Bucketing Structure

- Problem with linear buckets:
  - Vertex moved at most b-1 times
  - Accessed d[v] / b times by BuildBuckets
  - O(d[v] / b + b) work per vertex



- Solution: Logarithmic Buckets
  - O(log(d[v])) work per vertex
  - Buckets are hash bags

Experiment Setup

- 25 real-world and synthetic graphs
- 3 state-of-the-art parallel baselines
  - Julienne, ParK, PKC

Results

	Graph Statistics					Ours			Baselines				Notes
	Name	n	m	k_{max}	ρ	seq.*	par.	spd.	BZ*	Julienne	ParK	PKC	
Social	LJ	4.85M	85.7M	372	3,480	2.37	**.203**	11.7	1.49	.631	.637	.518	soc-LiveJournal1 [7]
	OK	3.07M	234M	253	5,667	3.94	**.526**	7.49	3.65	1.23	1.38	.810	com-orkut [79]
	WB	58.7M	523M	193	2,910	29.5	**.935**	31.6	14.3	1.16	2.64	2.18	soc-sinaweibo [64]
	TW	41.7M	2.41B	2,488	14,964	62.2	**2.72**	22.9	61.2	4.79	857	75.6	Twitter [44]
	FS	65.6M	3.61B	304	10,034	126	**3.68**	34.3	174	6.18	416	33.1	Friendster [79]
	Geomean for Social Networks					*18.5*	***.999***		*15.3*	*1.93*	*15.3*	*4.70*	
Web	EH	11.3M	522M	9,877	7,393	8.21	**.795**	10.3	5.49	1.39	5.67	8.22	eu-host [58]
	SD	89.3M	3.88B	10,507	19,063	140	**4.39**	32.0	143	6.56	410	57.5	sd-arc [58]
	CW	978M	74.7B	4,244	106,819	2453	**28.6**	85.8	2328	53.2	*T/O*	*T/O*	ClueWeb [58]
	HL14	1.72B	124B	4,160	58,737	3587	**54.7**	65.5	*OOM*	72.0	*OOM*	*OOM*	Hyperlink14 [58]
	HL12	3.56B	226B	10,565	130,737	9177	**108**	84.6	*OOM*	152	*OOM*	*OOM*	Hyperlink12 [58]
	Geomean for Web Networks					*622*	***14.3***		*N/A*	*22.1*	*N/A*	*N/A*	
Road	AF	33.5M	88.9M	3	189	9.83	**.155**	63.2	5.54	.281	.363	.253	OSM Africa [63]
	NA	87.0M	220M	4	286	32.4	.432	74.8	12.4	.682	.724	**.417**	OSM North America [63]
	AS	95.7M	244M	4	343	34.8	**.480**	72.5	16.0	.709	.878	.656	OSM Asia [63]
	EU	131M	333M	4	513	47.4	.679	69.8	33.2	.925	.869	**.609**	OSM Europe [63]
	Geomean for Road Networks					*26.9*	***.385***		*13.8*	*.595*	*.669*	*.453*	
k-NN	CH5	4.21M	29.7M	5	7	.826	**.021**	39.1	.431	.042	.037	**.021**	Chem [28, 76], $k = 5$
	GL2	24.9M	65.3M	2	12	6.96	**.109**	64.1	7.69	.167	.155	.113	GeoLife [76, 86], $k = 2$
	GL5	24.9M	157M	5	42	6.81	**.125**	54.7	3.54	.196	.179	.249	GeoLife [76, 86], $k = 5$
	GL10	24.9M	310M	10	16	8.46	**.162**	52.4	5.57	.277	.175	.168	GeoLife [76, 86], $k = 10$
	COS5	321M	1.96B	2	23	117	**2.06**	56.6	61.9	3.66	2.74	2.08	Cosmo50 [45, 76], $k = 5$
	Geomean for k-NN Graphs					*8.27*	***.157***		*5.26*	*.268*	*.218*	*.183*	
Others	TRCE	16.0M	48.0M	2	1,839	2.03	**.066**	31.0	1.49	1.96	.424	.067	Huge traces [64]
	BBL	21.2M	63.6M	2	1,915	3.18	**.077**	41.1	3.36	1.80	.203	.081	Huge bubbles [64]
	GRID	100M	400M	2	50,499	6.21	**.282**	22.1	14.1	14.8	8.03	3.21	Synthetic grid graph
	CUBE	1.00B	6.0B	3	2,895	183	**4.01**	45.7	162	9.46	110	10.8	Synthetic cubic graph
	HCNS	0.1M	5.0B	50,000	50,000	27.8	**2.01**	13.8	23.5	16.0	49.7	*OOM*	Synthetic high-coreness graph
	HPL	100M	1.20B	3,980	6,297	47.3	**1.77**	26.8	38.9	3.59	30.4	59.1	Synthetic power-law graph
	Geomean for Other Graphs					*14.6*	***.523***		*14.8*	*5.52*	*6.97*	*N/A*	

Results

- Comparative baselines exhibit worse than sequential performance on some graphs
- Proposed algorithm beats sequential by 6.9-85x
- Outperformed all baselines on 23 of 25 graphs
  - Within 12% of best baseline on EU and NA

- Speedups due to Novel Ideas:
  - Sampling: Up to 4.3x
  - VGC: 2.3-31.2x
  - HBS: Up to 47.8x over 1 bucket and 2.01x over 16-bucket structure

Results

	Graph Statistics					Ours			Baselines				Notes
	Name	n	m	$k_{\max}$	ρ	seq.*	par.	spd.	BZ*	Julienne	ParK	PKC	
Social	LJ	4.85M	85.7M	372	3,480	2.37	**.203**	11.7	1.49	.631	.637	.518	soc-LiveJournal1 [7]
	OK	3.07M	234M	253	5,667	3.94	**.526**	7.49	3.65	1.23	1.38	.810	com-orkut [79]
	WB	58.7M	523M	193	2,910	29.5	**.935**	31.6	14.3	1.16	2.64	2.18	soc-sinaweibo [64]
	TW	41.7M	2.41B	2,488	14,964	62.2	**2.72**	22.9	61.2	4.79	857	75.6	Twitter [44]
	FS	65.6M	3.61B	304	10,034	126	**3.68**	34.3	174	6.18	416	33.1	Friendster [79]
	Geomean for Social Networks					*18.5*	***.999***		*15.3*	*1.93*	*15.3*	*4.70*	
Web	EH	11.3M	522M	9,877	7,393	8.21	**.795**	10.3	5.49	1.39	5.67	8.22	eu-host [58]
	SD	89.3M	3.88B	10,507	19,063	140	**4.39**	32.0	143	6.56	410	57.5	sd-arc [58]
	CW	978M	74.7B	4,244	106,819	2453	**28.6**	85.8	2328	53.2	*T/O*	*T/O*	ClueWeb [58]
	HL14	1.72B	124B	4,160	58,737	3587	**54.7**	65.5	*OOM*	72.0	*OOM*	*OOM*	Hyperlink14 [58]
	HL12	3.56B	226B	10,565	130,737	9177	**108**	84.6	*OOM*	152	*OOM*	*OOM*	Hyperlink12 [58]
	Geomean for Web Networks					*622*	***14.3***		*N/A*	*22.1*	*N/A*	*N/A*	
Road	AF	33.5M	88.9M	3	189	9.83	**.155**	63.2	5.54	.281	.363	.253	OSM Africa [63]
	NA	87.0M	220M	4	286	32.4	.432	74.8	12.4	.682	.724	**.417**	OSM North America [63]
	AS	95.7M	244M	4	343	34.8	**.480**	72.5	16.0	.709	.878	.656	OSM Asia [63]
	EU	131M	333M	4	513	47.4	.679	69.8	33.2	.925	.869	**.609**	OSM Europe [63]
	Geomean for Road Networks					*26.9*	***.385***		*13.8*	*.595*	*.669*	*.453*	
k-NN	CH5	4.21M	29.7M	5	7	.826	**.021**	39.1	.431	.042	.037	**.021**	Chem [28, 76], $k = 5$
	GL2	24.9M	65.3M	2	12	6.96	**.109**	64.1	7.69	.167	.155	.113	GeoLife [76, 86], $k = 2$
	GL5	24.9M	157M	5	42	6.81	**.125**	54.7	3.54	.196	.179	.249	GeoLife [76, 86], $k = 5$
	GL10	24.9M	310M	10	16	8.46	**.162**	52.4	5.57	.277	.175	.168	GeoLife [76, 86], $k = 10$
	COS5	321M	1.96B	2	23	117	**2.06**	56.6	61.9	3.66	2.74	2.08	Cosmo50 [45, 76], $k = 5$
	Geomean for k-NN Graphs					*8.27*	***.157***		*5.26*	*.268*	*.218*	*.183*	
Others	TRCE	16.0M	48.0M	2	1,839	2.03	**.066**	31.0	1.49	1.96	.424	.067	Huge traces [64]
	BBL	21.2M	63.6M	2	1,915	3.18	**.077**	41.1	3.36	1.80	.203	.081	Huge bubbles [64]
	GRID	100M	400M	2	50,499	6.21	**.282**	22.1	14.1	14.8	8.03	3.21	Synthetic grid graph
	CUBE	1.00B	6.0B	3	2,895	183	**4.01**	45.7	162	9.46	110	10.8	Synthetic cubic graph
	HCNS	0.1M	5.0B	50,000	50,000	27.8	**2.01**	13.8	23.5	16.0	49.7	*OOM*	Synthetic high-coreness graph
	HPL	100M	1.20B	3,980	6,297	47.3	**1.77**	26.8	38.9	3.59	30.4	59.1	Synthetic power-law graph
	Geomean for Other Graphs					*14.6*	***.523***		*14.8*	*5.52*	*6.97*	*N/A*	

Results

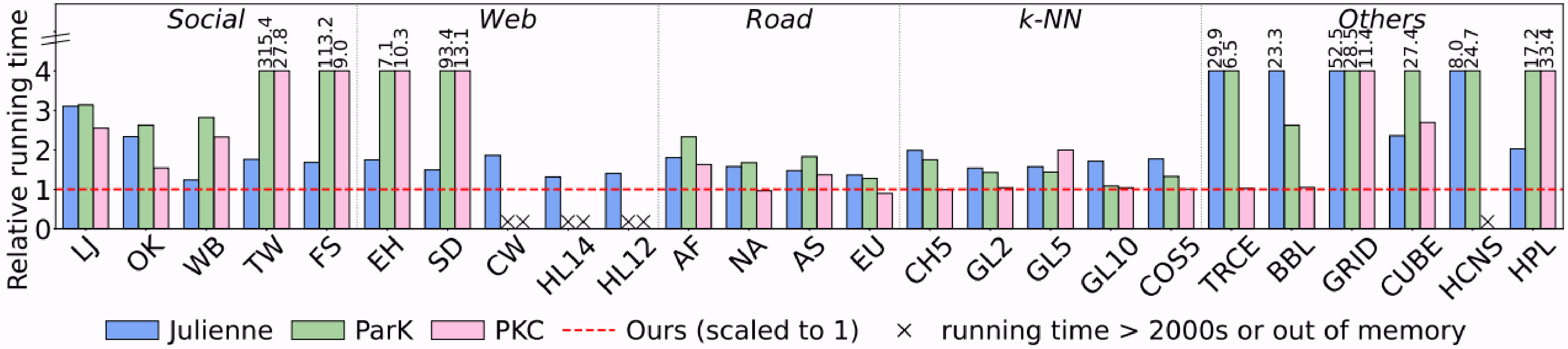


Fig. 5. Relative running time of ParK [17], PKC [38] and Julienne [18, 19] normalized to our running time (red dotted line) on all graphs. Lower is better. The bars are truncated at 4 for better visualization. The text on the bars are actual relative running time.

Results

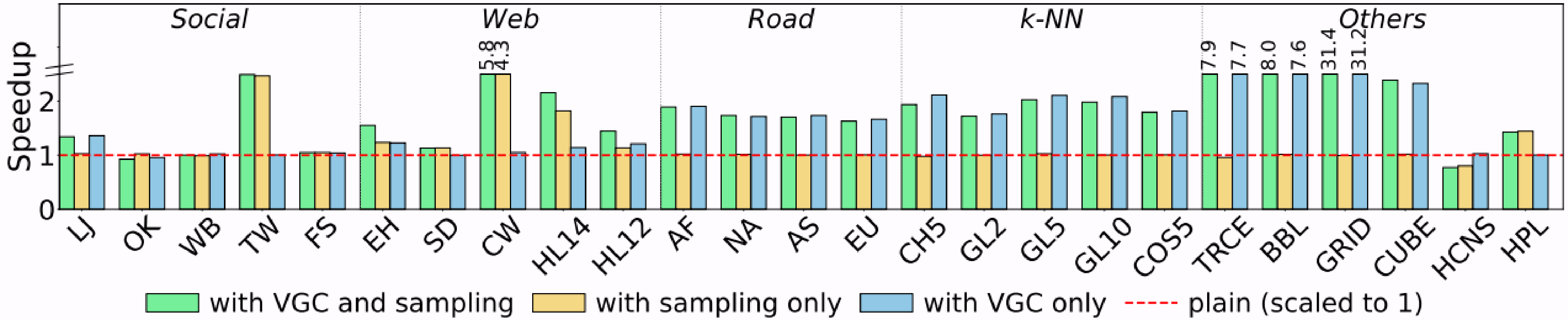


Fig. 6. Speedup of VGC and sampling over a plain implementation. Higher is better. The plain version does not use VGC or sampling. The bars are truncated at 2.5 for better visualization. The text on the bars are the actual speedup values.

Results

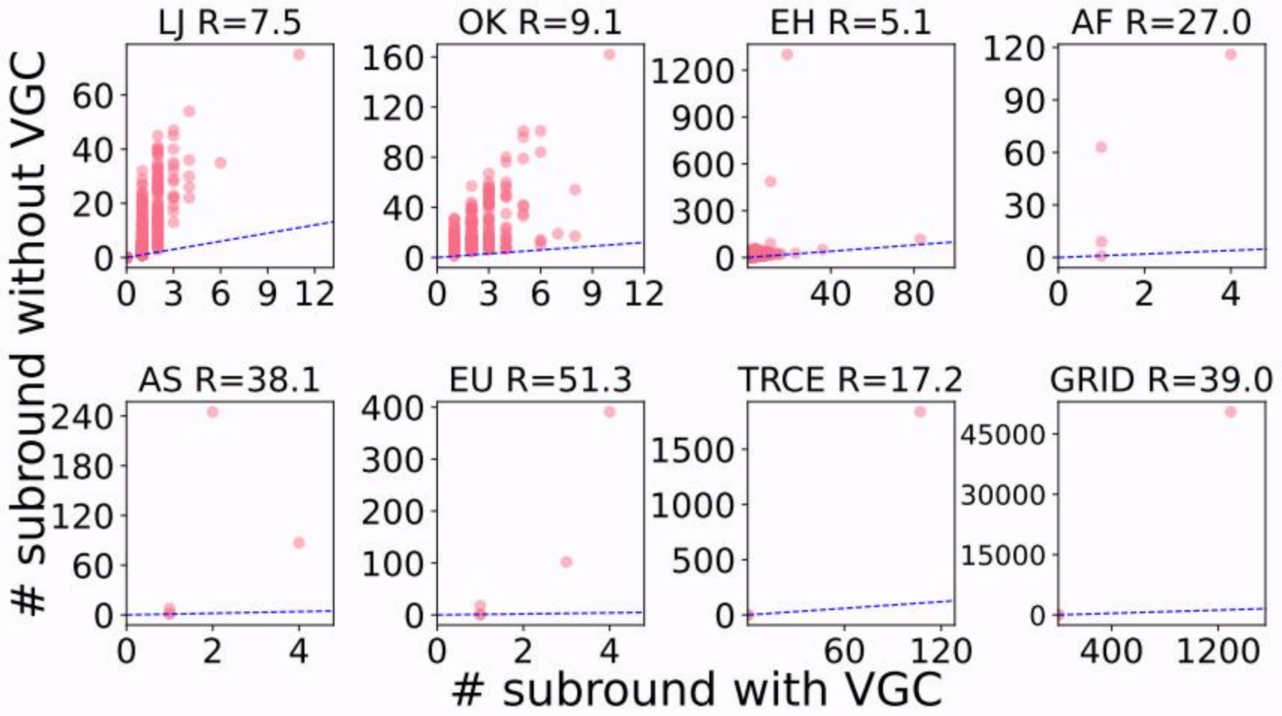


Fig. 7. Number of subrounds with and without VGC. Each point (x, y) means VGC reduces a round with y subrounds to x subrounds. The blue dotted line is the baseline where $y = x$. The number R in each subtitle is the reduction ratio of the number of subrounds.

Results

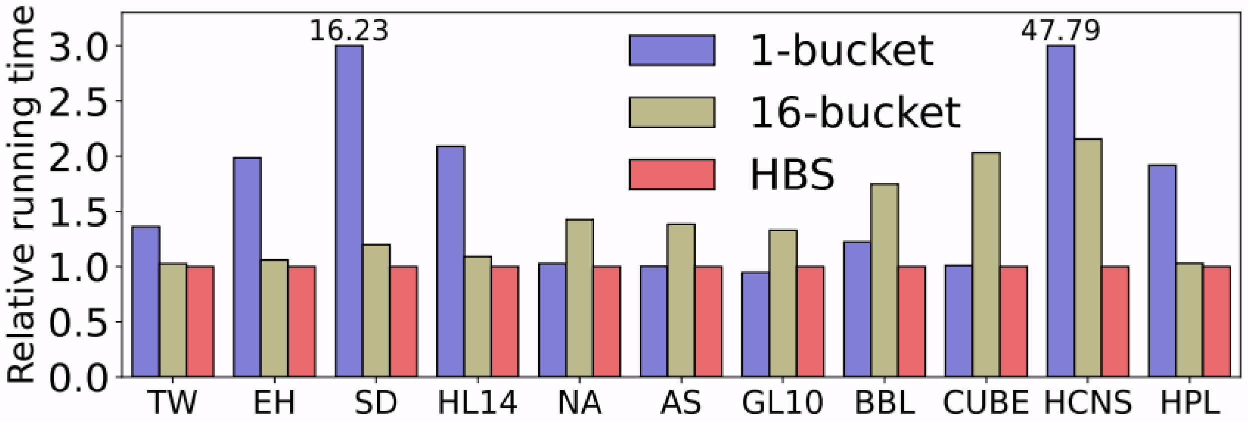


Fig. 8. Relative running time of different bucketing strategies, normalized to HBS. Lower is better.

Strengths

- Proofs of work efficiency for online peeling
- Strong parallel performance on general graphs
- Solid mathematical foundations
- Novel sampling strategy

Weaknesses

- Parameter sensitive
  - Sampling depends on r
  - VGC depends on queue size
- No discussion of memory footprint

Future Directions

- Applying novel strategies to GPUs, external memory, low-memory settings
- Dynamic and streaming graphs
- Approximate k-core
- Directed and weighted graphs
- Related graph problems

Discussion Questions

- How can we apply adaptive sampling thresholds?
- How will performance scale beyond 92 cores? On distributed architectures?
- What is the cost of restarting the algorithm? Is it acceptable in practice?