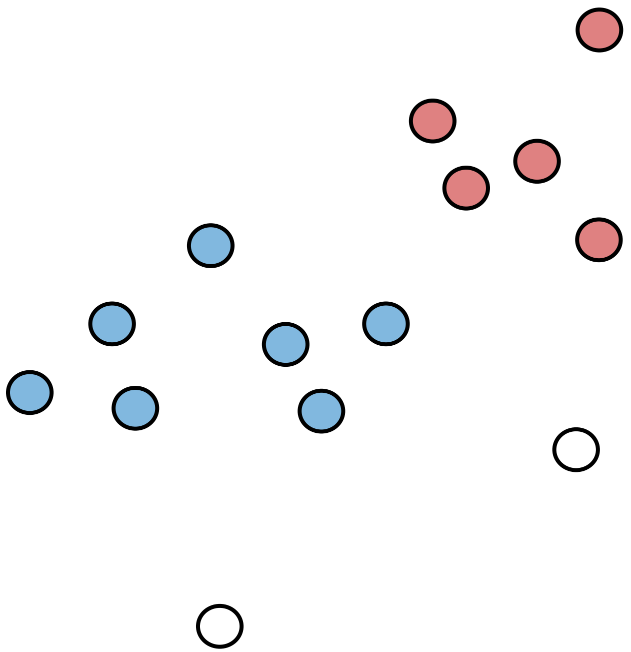
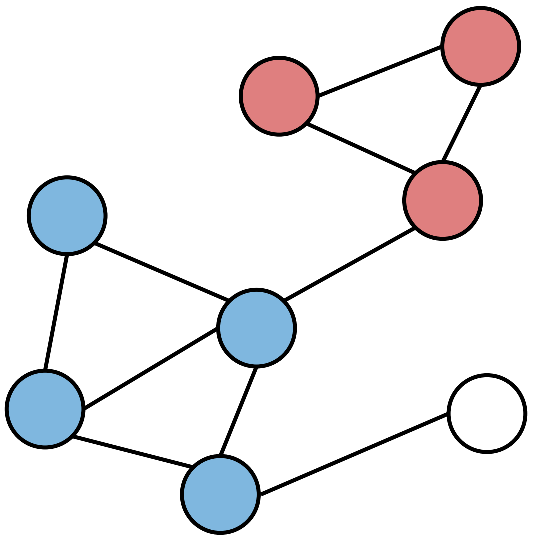


Parallel Algorithms for Density-Based and Structural Clustering

Julian Shun (MIT CSAIL)

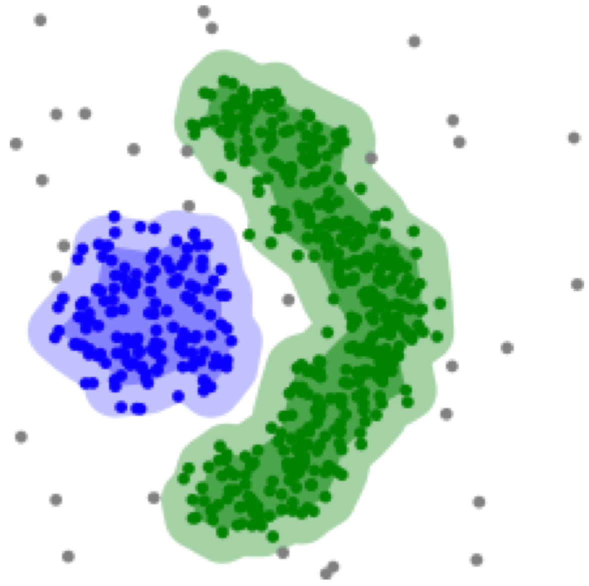


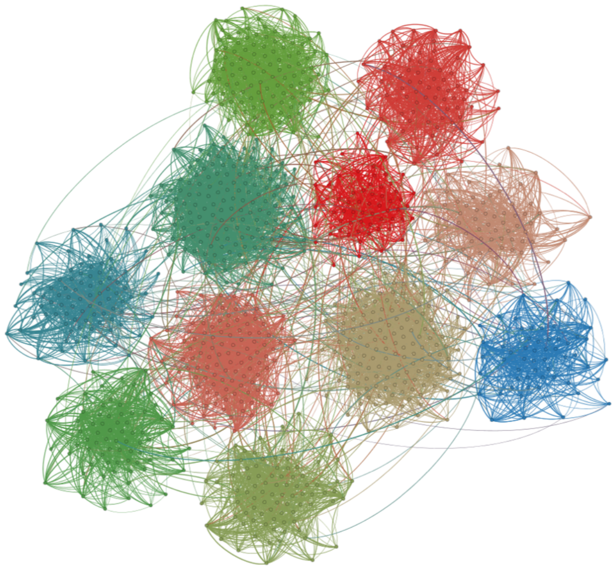
Yiqiu Wang, Yan Gu, and Julian Shun, *Theoretically-Efficient and Practical Parallel DBSCAN,* SIGMOD 2020.

Tom Tseng, Laxman Dhulipala, and Julian Shun, *Parallel Index-Based Structural Clustering and Its Approximation*, SIGMOD 2021.

Clustering

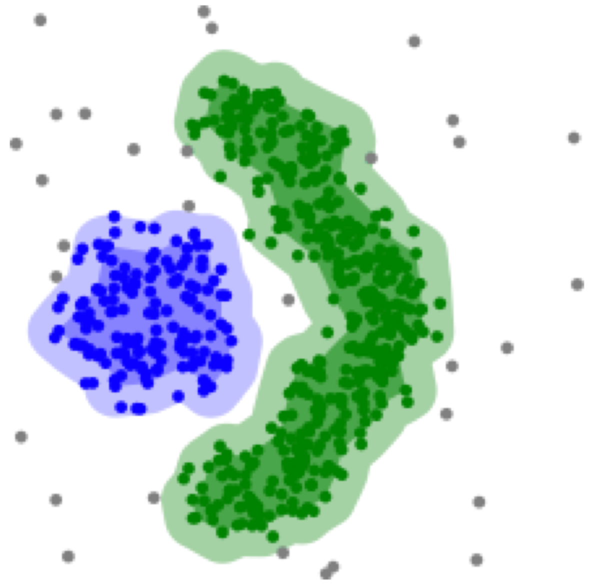
- Group “similar” objects together, and separate “dissimilar” objects
- Can be applied to spatial data and graph data
- Applications
  - Community detection, bioinformatics, parallel/distributed processing, visualization, image segmentation, anomaly detection, document analysis, machine learning, etc.

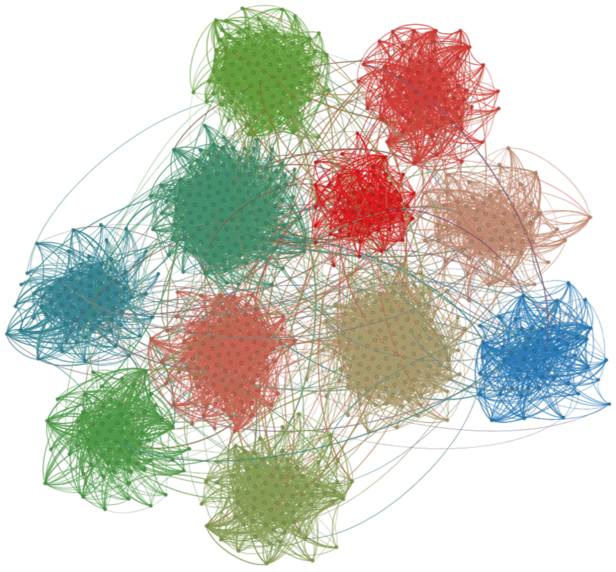




Clustering

- Very well-studied topic
  - Hundreds of textbooks on this topic
- No universally accepted definition for cluster quality, many metrics have been proposed
- At least thousands of different clustering algorithms

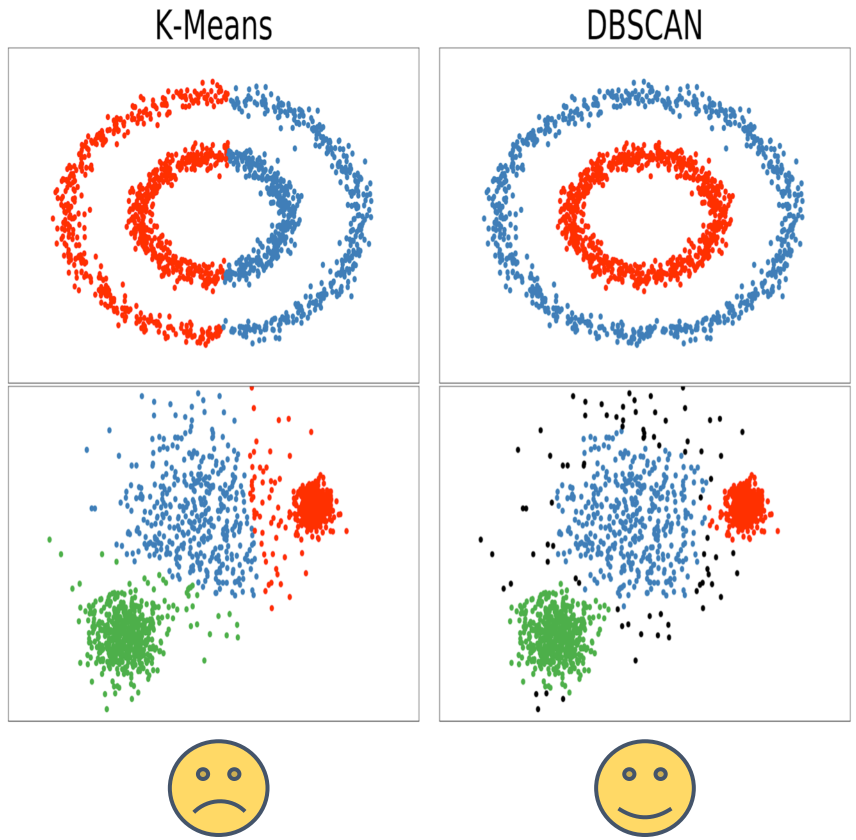




DBSCAN for Spatial Clustering

- DBSCAN (Density-Based Spatial Clustering of Applications with Noise)
  - Ester et al. [KDD’96]

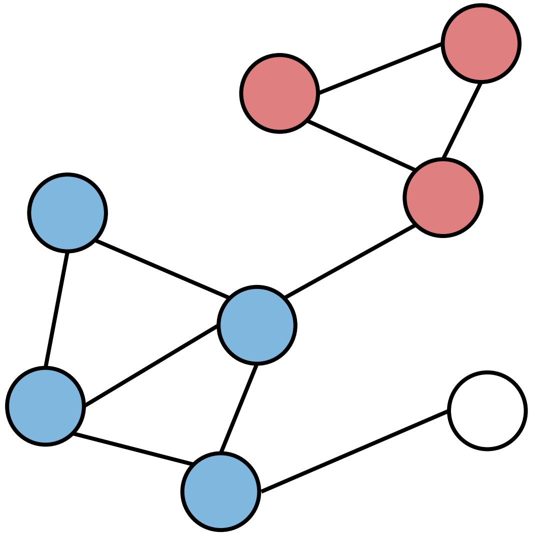
- Areas of high density form clusters
- Does not require number of clusters beforehand
- Detects arbitrarily shaped clusters
- Robust to noise



SCAN for Graph Clustering

- SCAN (Structural Clustering Algorithm for Networks)
  - Xu et al. [KDD'07]

- DBSCAN, but on graphs
- Similarity of vertices based on their number of shared neighbors
- "Dense" areas contain many vertices who have many similar neighbors
- Can identify clusters and outliers

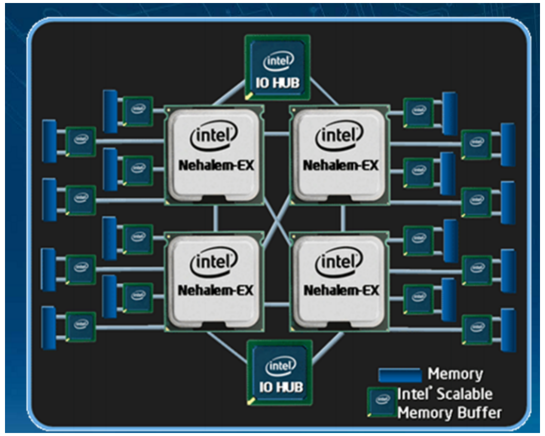


Processing Large Datasets

- Publicly-available graphs have up to hundreds of billions of edges
- Spatial datasets can be even larger

Need high-performance solutions to process large datasets in a timely fashion

- We design state-of-the-art parallel multicore algorithms for DBSCAN and SCAN
  - Strong theoretical guarantees
  - Can process the largest datasets used in the literature for these problems on a multicore, more quickly than existing solutions

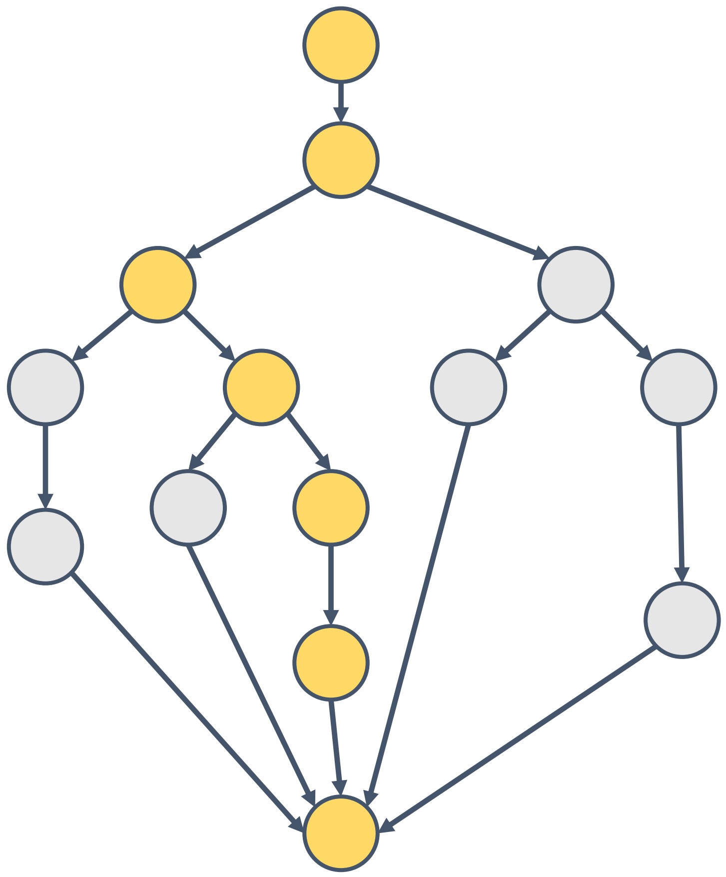


Work-Span Model of Parallel Computation

- Work: number of operations used
- Span: length of the longest sequential dependence
- Parallelism = Work / Span
- Running time ≤ Work/P + Span (when run on P processors)
- A work-efficient parallel algorithm has work that asymptotically matches that of the best sequential algorithm for the problem

Goal: work-efficient and low-span parallel algorithms

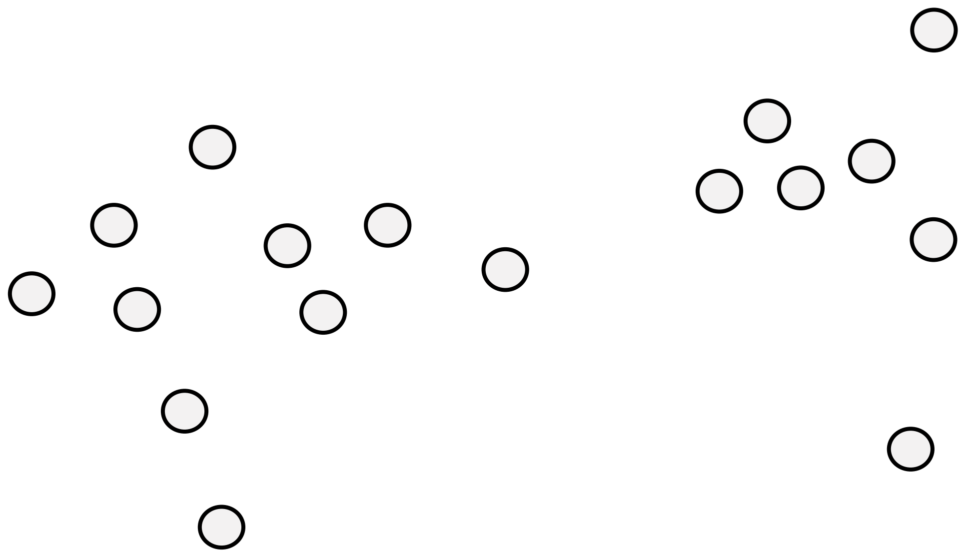
Computation graph



DBSCAN for Spatial Clustering

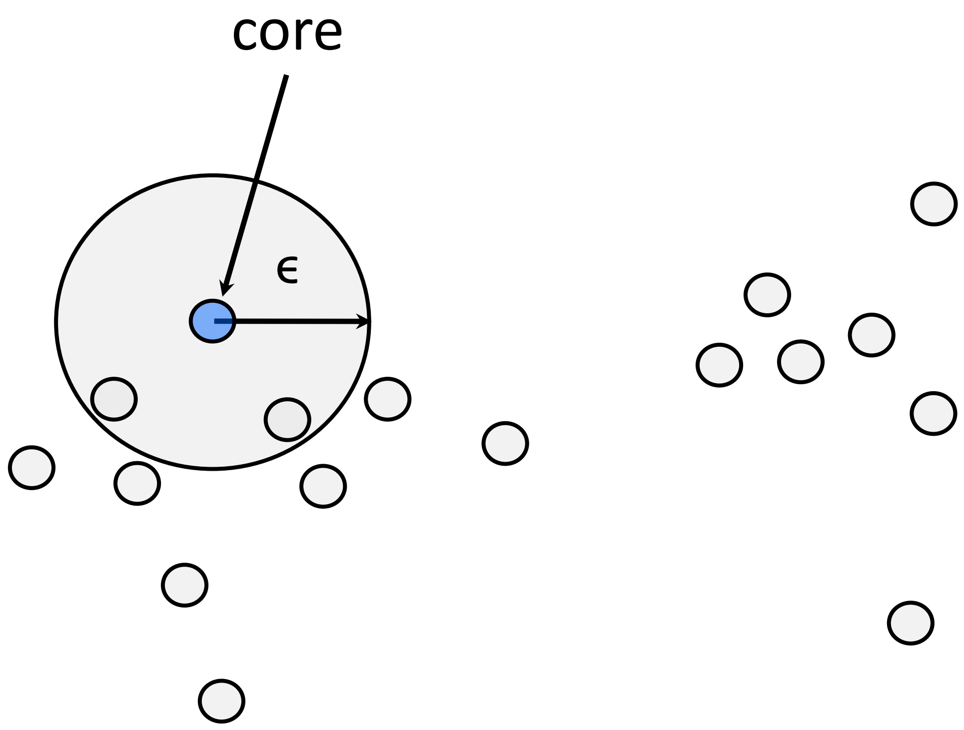
Problem Definition - DBSCAN

- Parameters
  - ϵ
  - minPts



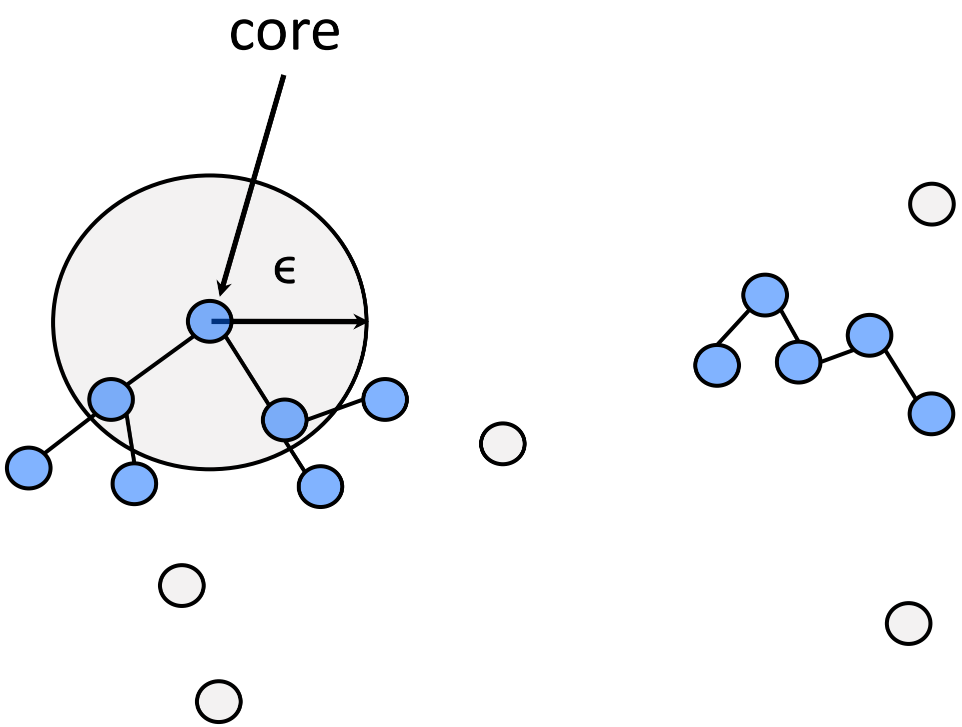
Problem Definition - DBSCAN

- Parameters
  - ϵ
  - minPts=3
- Core point
  - At least minPts points in ϵ-circle



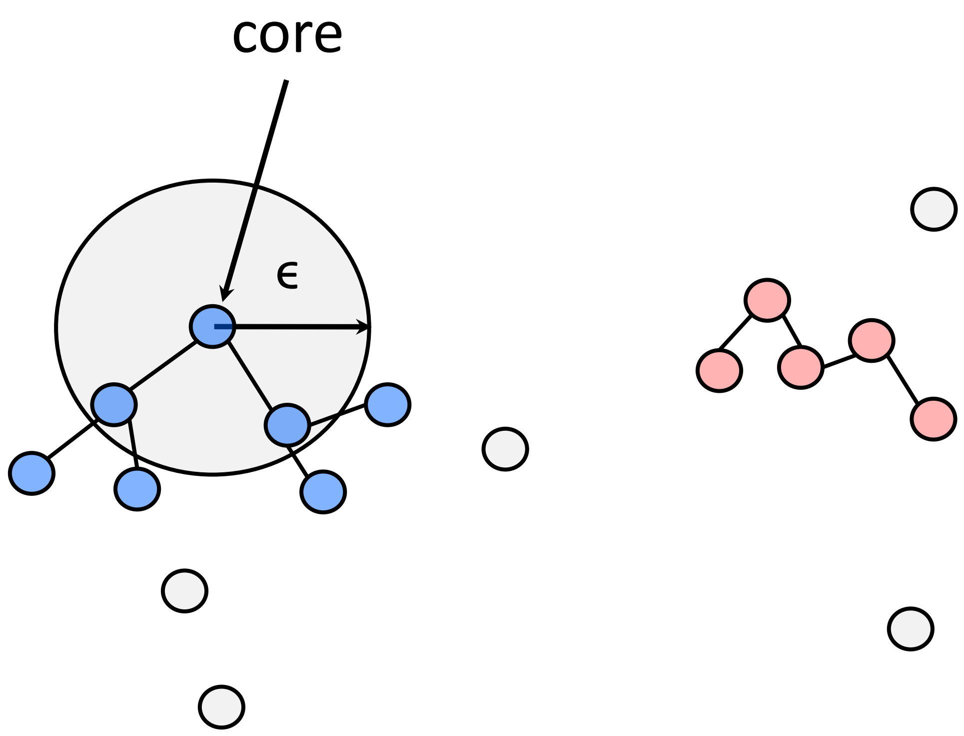
Problem Definition - DBSCAN

- Parameters
  - ϵ
  - minPts=3
- Core point
  - At least minPts points in ϵ-circle
  - Connected if in ϵ-circle



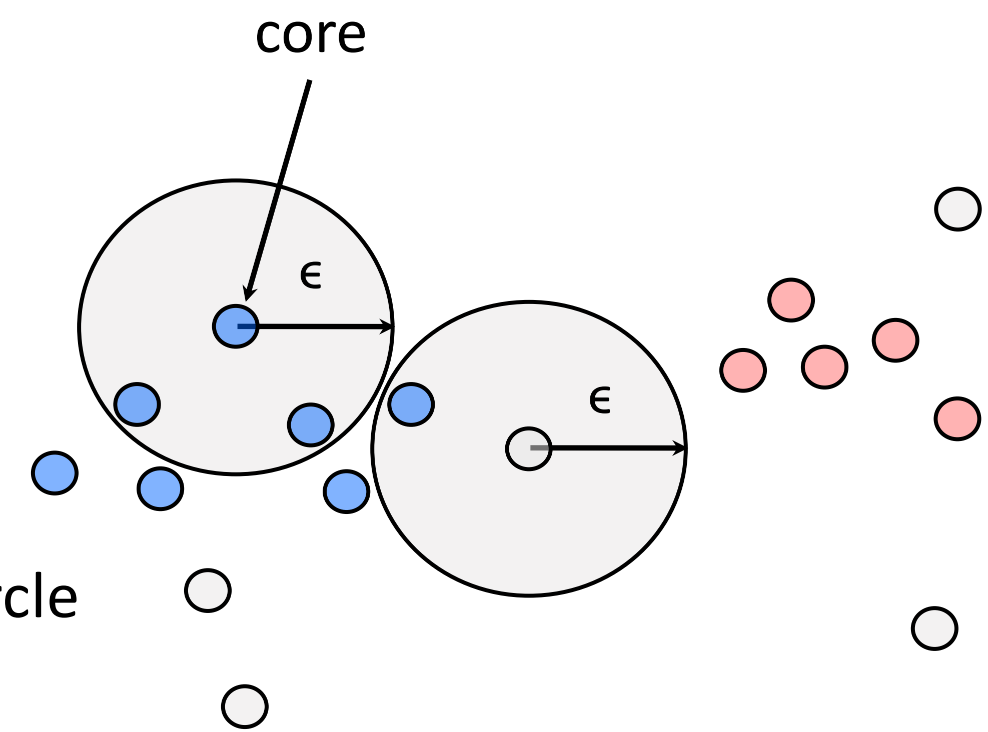
Problem Definition - DBSCAN

- Parameters
  - ϵ
  - minPts=3
- Core point
  - At least minPts points in ϵ-circle
  - Connected if in ϵ-circle



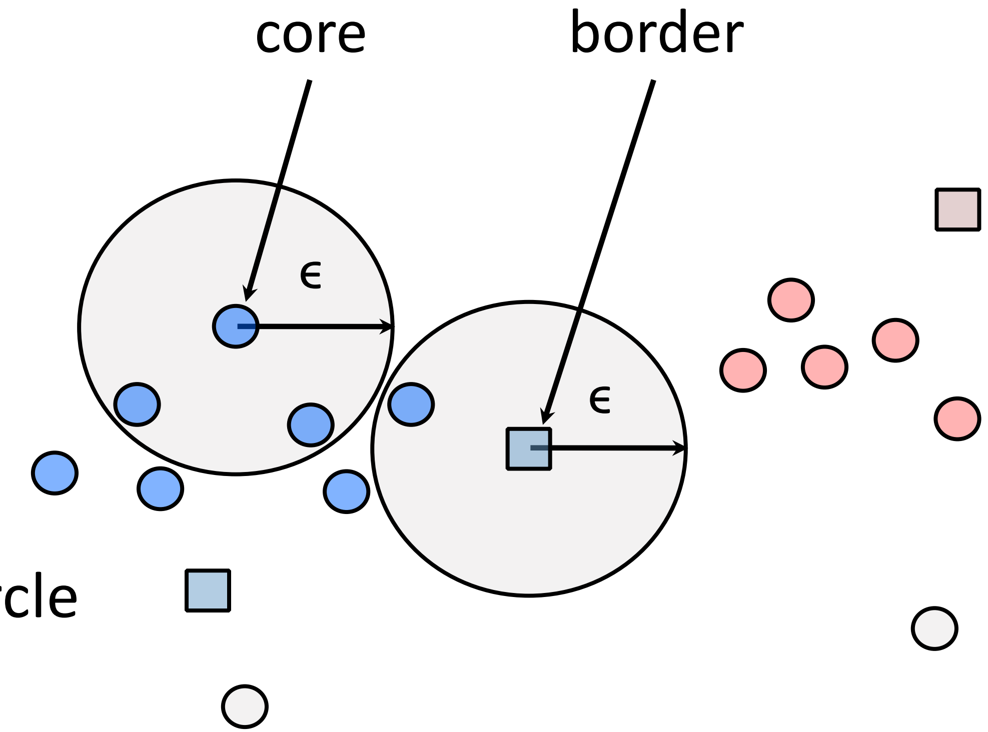
Problem Definition - DBSCAN

- Parameters
  - ϵ
  - minPts=3
- Core point
  - At least minPts points in ϵ-circle
  - Connected if in ϵ-circle
- Border point
  - Fewer than minpts points in ϵ-circle
  - Contains a core point in ϵ-circle



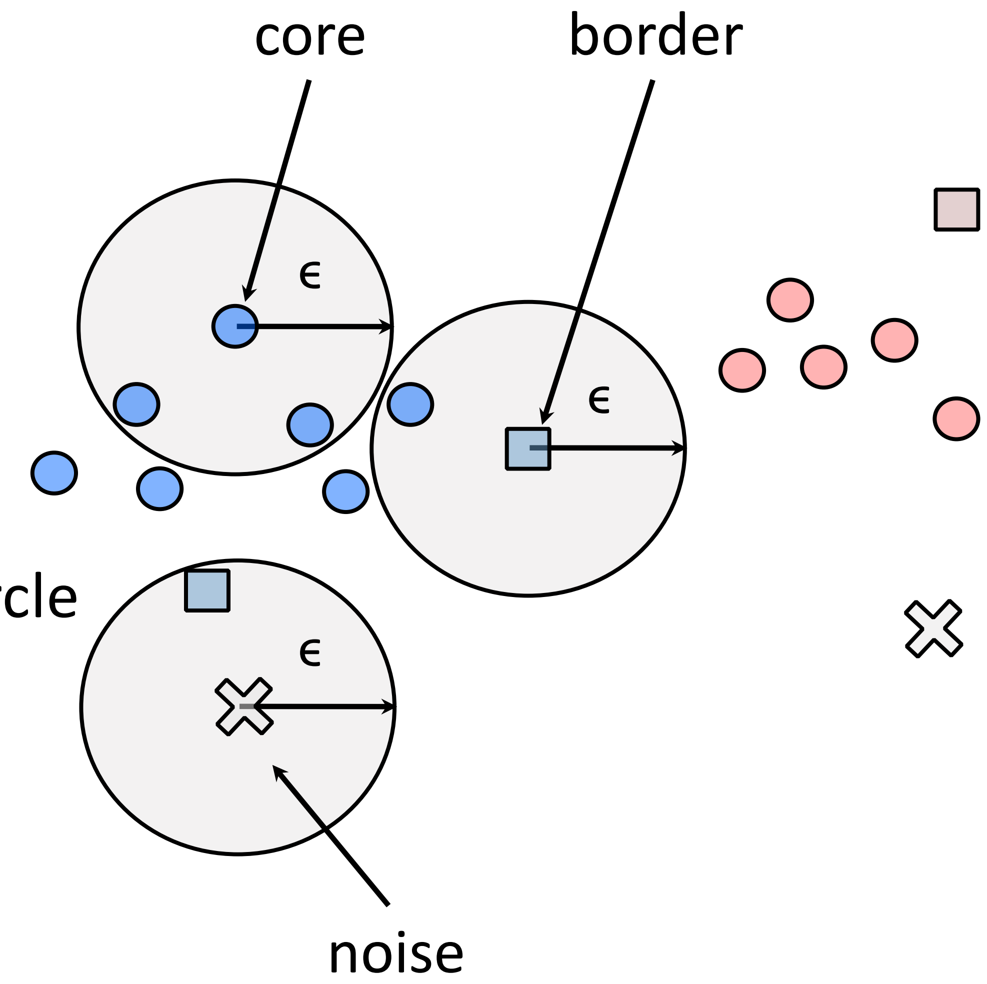
Problem Definition - DBSCAN

- Parameters
  - ϵ
  - minPts=3
- Core point
  - At least minPts points in ϵ-circle
  - Connected if in ϵ-circle
- Border point
  - Fewer than minpts points in ϵ-circle
  - Contains a core point in ϵ-circle



Problem Definition - DBSCAN

- Parameters
  - ϵ
  - minPts=3
- Core point
  - At least minPts points in ϵ-circle
  - Connected if in ϵ-circle
- Border point
  - Fewer than minPts points in ϵ-circle
  - Contains a core point in ϵ-circle
- Noise point



Related Work

- Sequential
  - de Berg et al., ISAAC’17 (Exact algorithms)
  - Gan and Tao, SIGMOD’15 Best Paper Award (Approximate algorithm, hardness result)
- Parallel
  - Xu et al., HPDM’99 (PDBSCAN, distributed R-Tree)
  - Patwary et al., SC’12 (PDSDBSCAN, parallel lock-based union-find)
  - Gotz et al., MLHPC’15 (HPDBSCAN, data splitting and merging)
  - Song et al., SIGMOD’18 (RP-DBSCAN, random partitioning, Map-Reduce)
  - Many more
- Challenges
  - Lack of theoretical guarantees in parallel implementations
  - High scalability but low work-efficiency

Our Contributions

- Parallel algorithms with work matching best sequential bounds (work-efficient)
- Highly-optimized multicore implementations
- Comprehensive experimental study showing that our algorithms outperform state-of-the-art

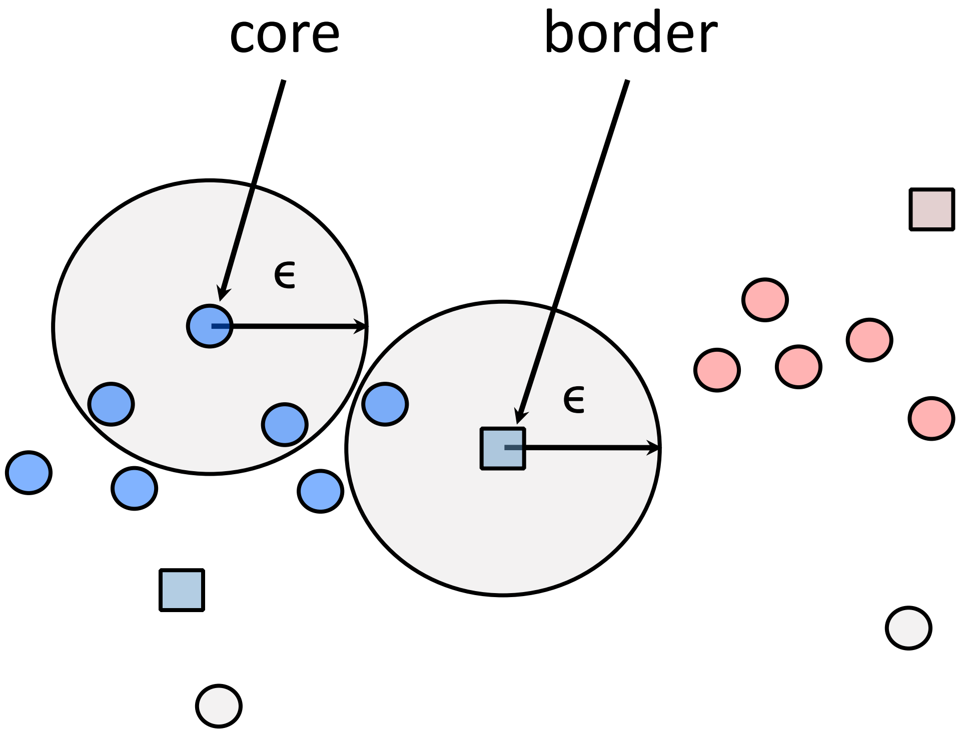
All of Our Algorithms are Theoretically Efficient

2D Algorithms	Delaunay Triangulation	Unit-spherical Emptiness Checking
	O(n log n) expected work; O(log n) span with high probability	O(n log n) expected work; O($\log^2$ n) span with high probability
3D Algorithm	O((n log n)$^{4/3}$) expected work; Polylogarithmic span with high probability	
Any Constant Dimension Algorithm	O($n^{2-(2/(\lceil d/2\rceil+1))+\delta}$) expected work; Polylogarithmic span with high probability	
Approximate Algorithm	O(n) expected work; O(log n) span with high probability	

- Our work bounds match the best sequential bounds by de Berg et al. and Gan and Tao (work-efficient)

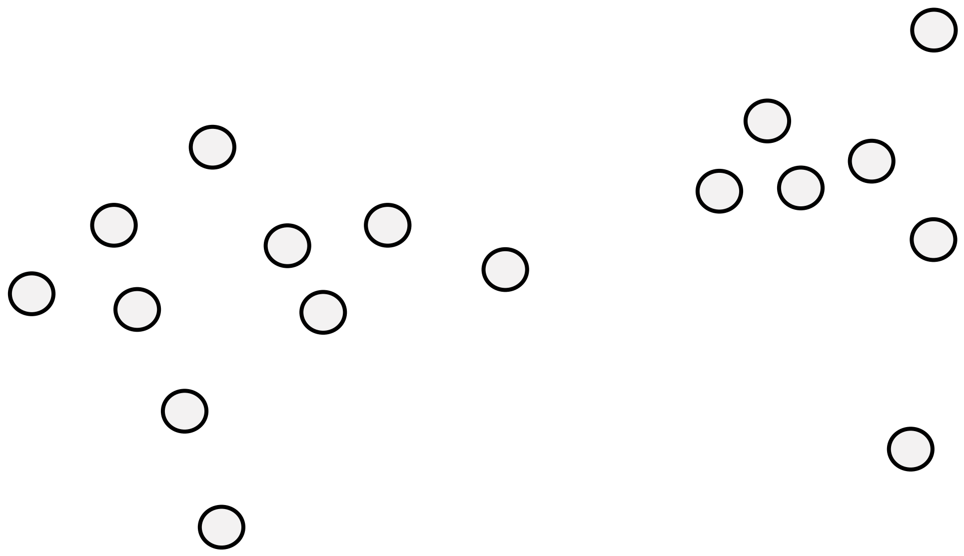
Naive Parallel Algorithm

- Points issue range queries in parallel
- Parallel connected components
- Quadratic work in the worst case
  - Worst-case linear work per point for range query



Our Parallel DBSCAN Algorithm

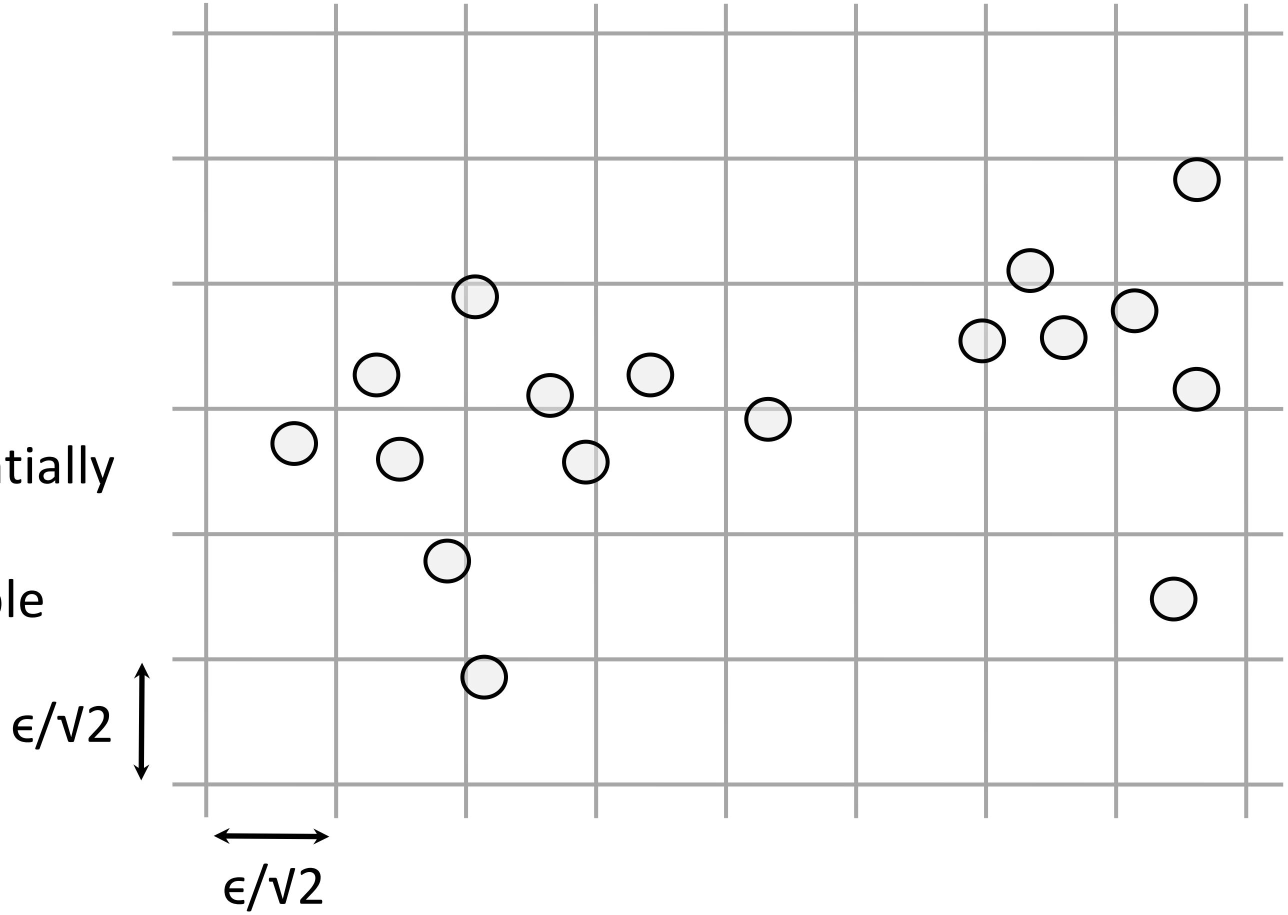
1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points



Our Parallel DBSCAN Algorithm

1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points

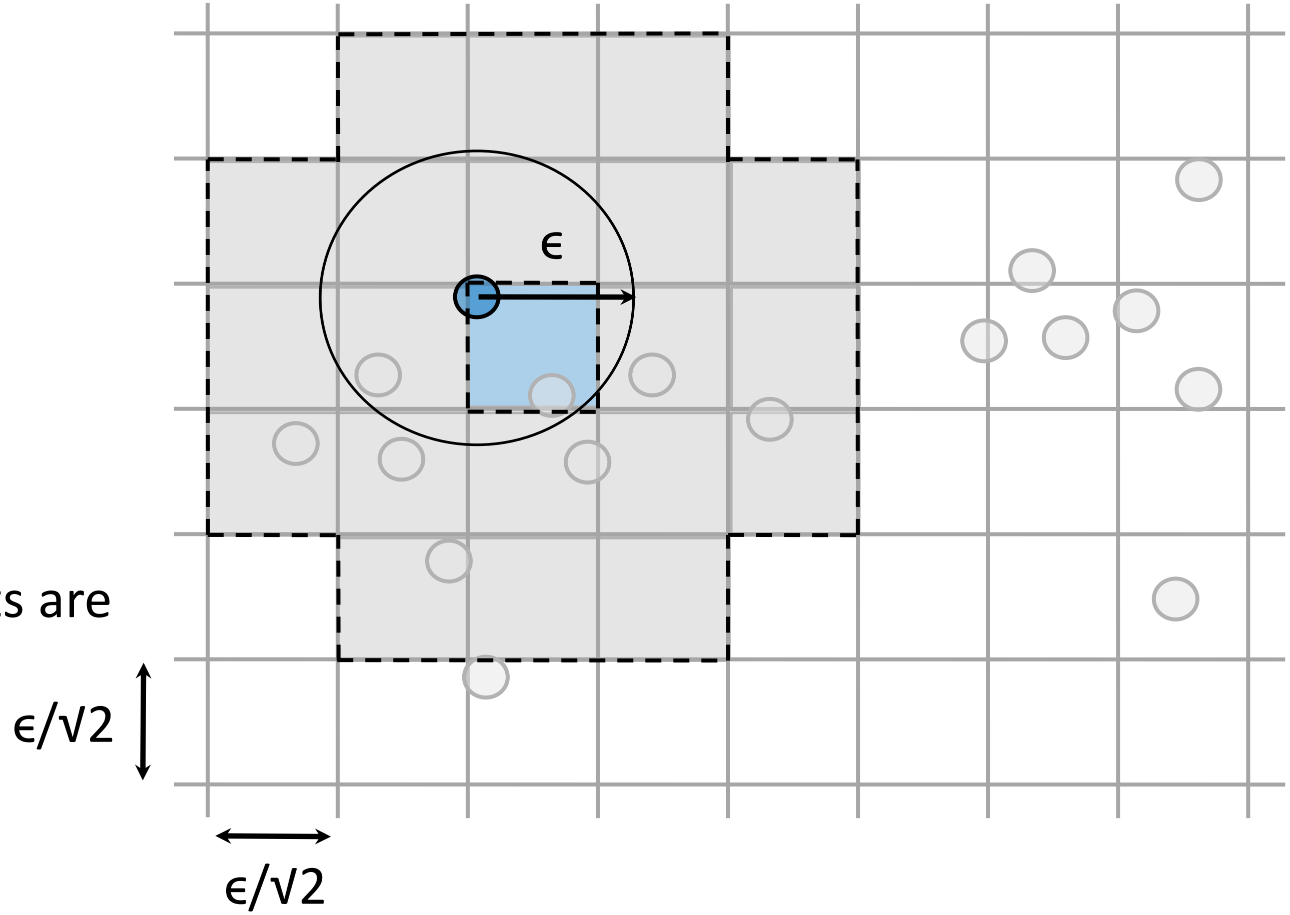
- First used by de Berg et al. sequentially
- Sort based on cell ID
- Insert points into parallel hash table



Our Parallel DBSCAN Algorithm

1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points

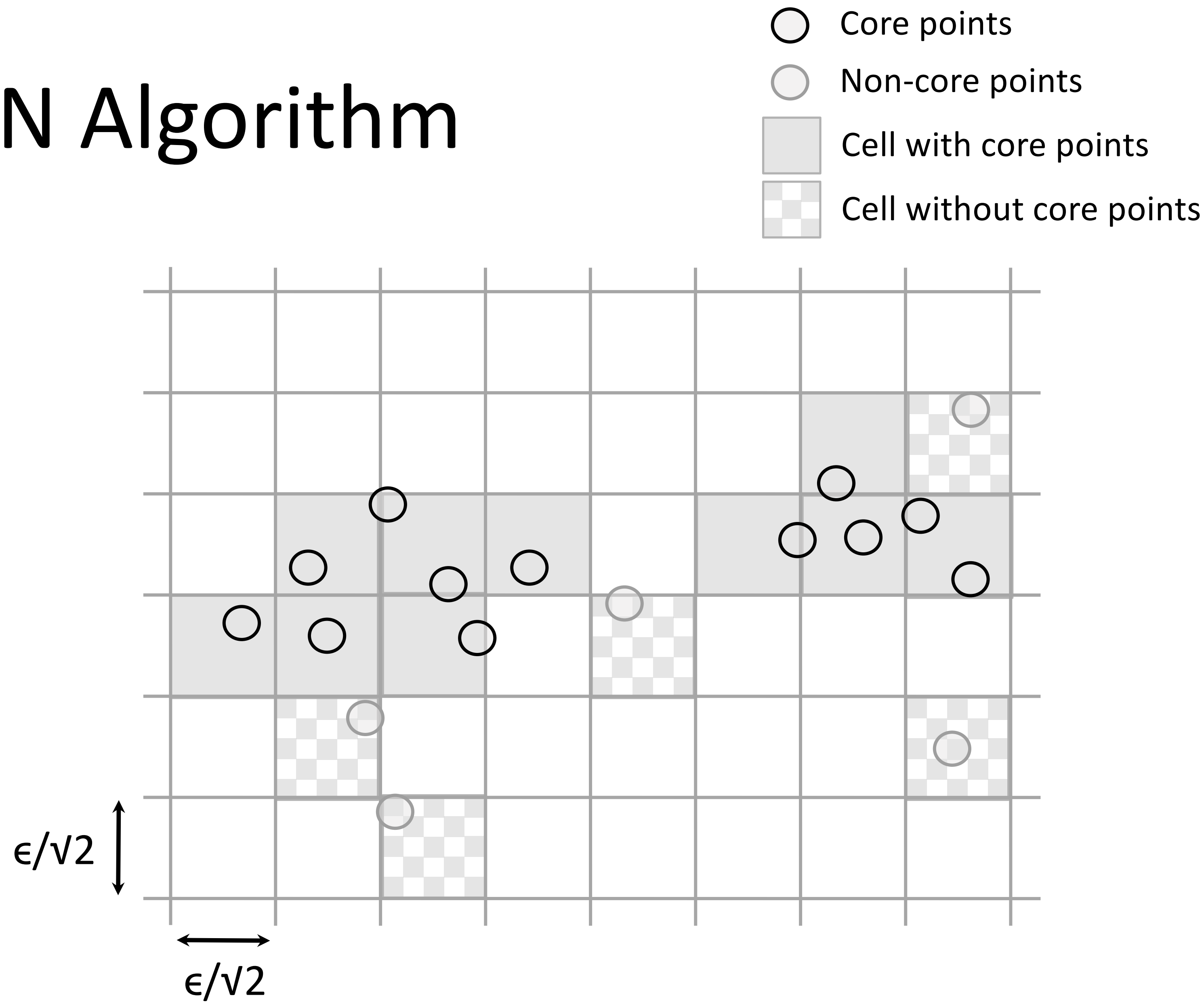
- Loop through points in parallel
- Check 21-cell neighborhood
- Cell with ≥ minPts points, all points are core



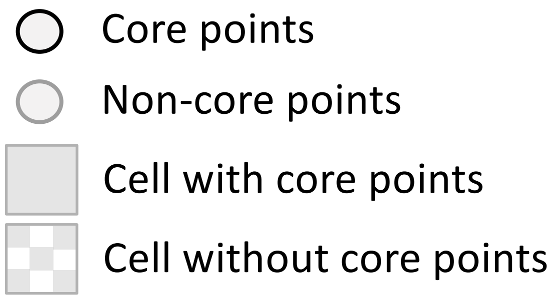
Our Parallel DBSCAN Algorithm

1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points

- "Core cells" and "non-core cells"

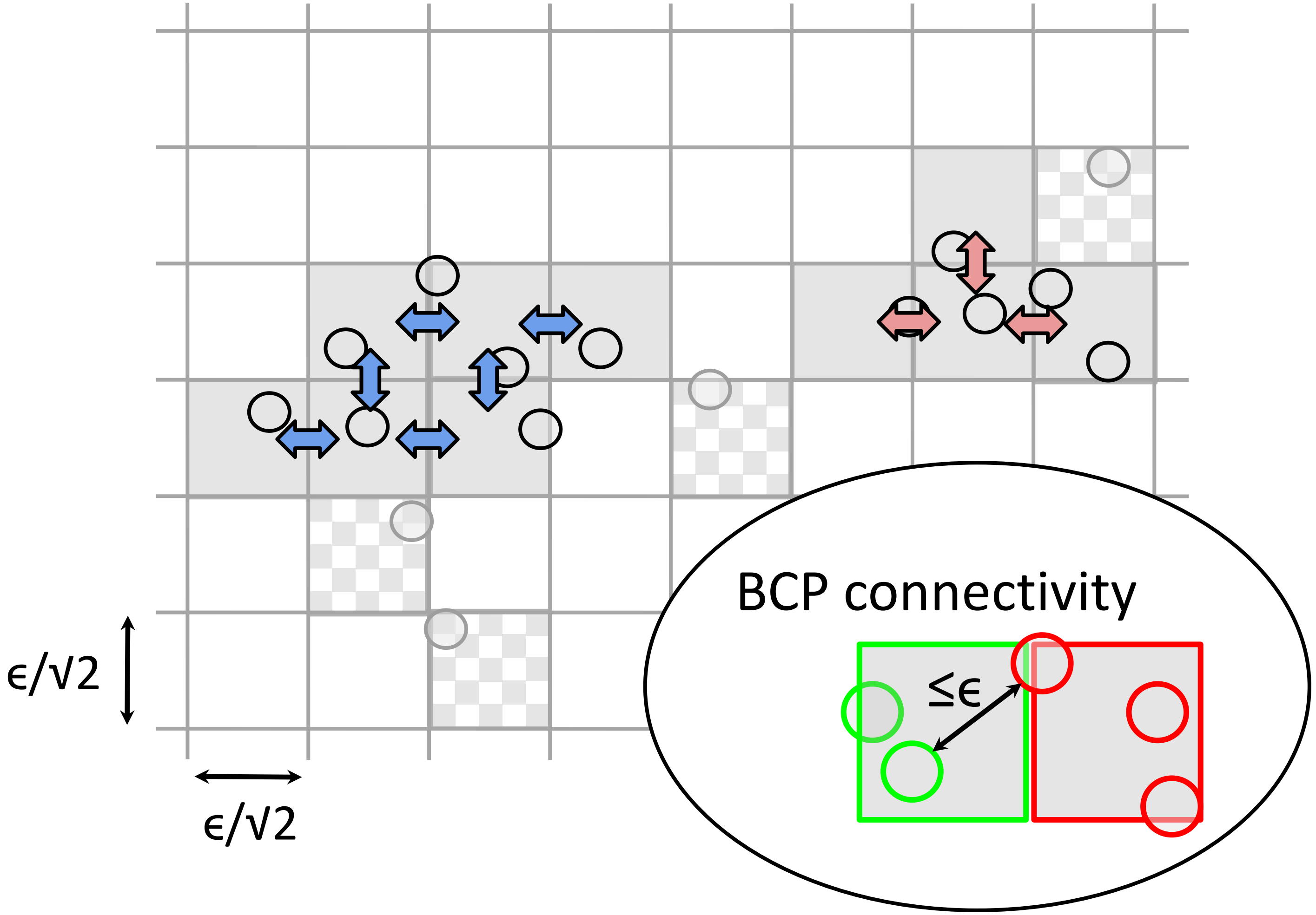


Our Parallel DBSCAN Algorithm

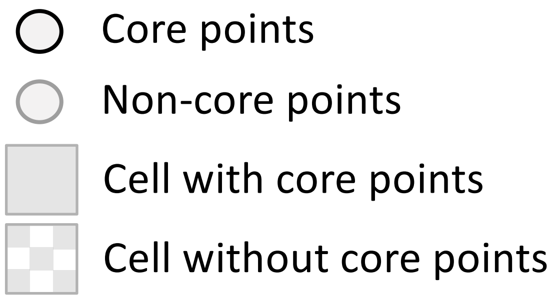


1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points

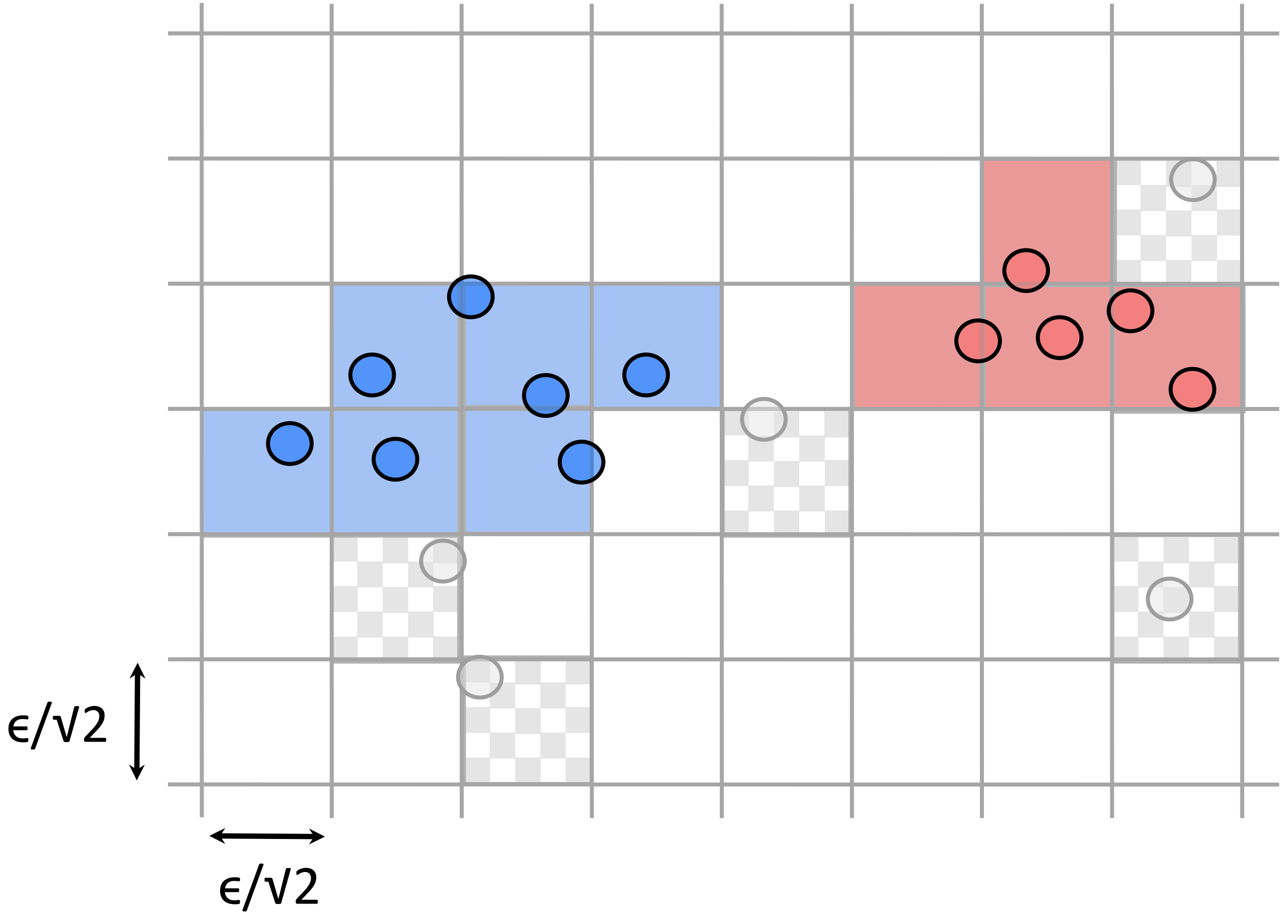
- Bichromatic closest pair (BCP) connectivity
  - Finds closest pair of points between two cells
  - Connect cells if distance ≤ ϵ
  - Used by Gan-Tao sequentially
- Run connected components on core cells to form clusters for core points



Our Parallel DBSCAN Algorithm

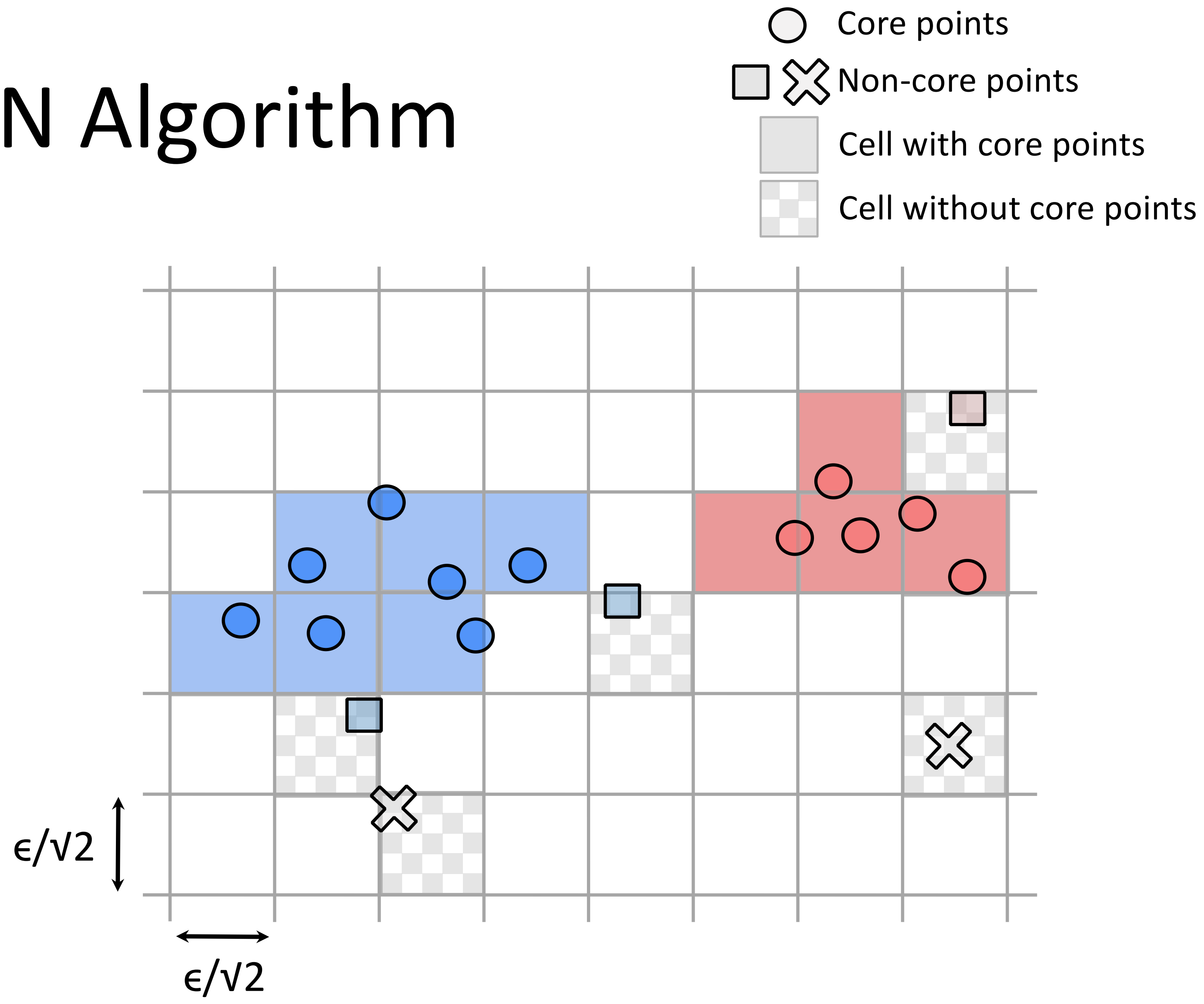


1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points



Our Parallel DBSCAN Algorithm

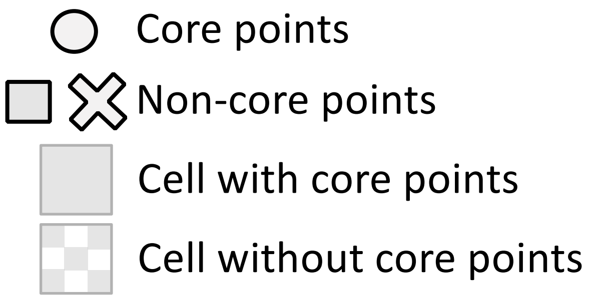
1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points



Our Parallel DBSCAN Algorithm – Analysis (w.h.p.)

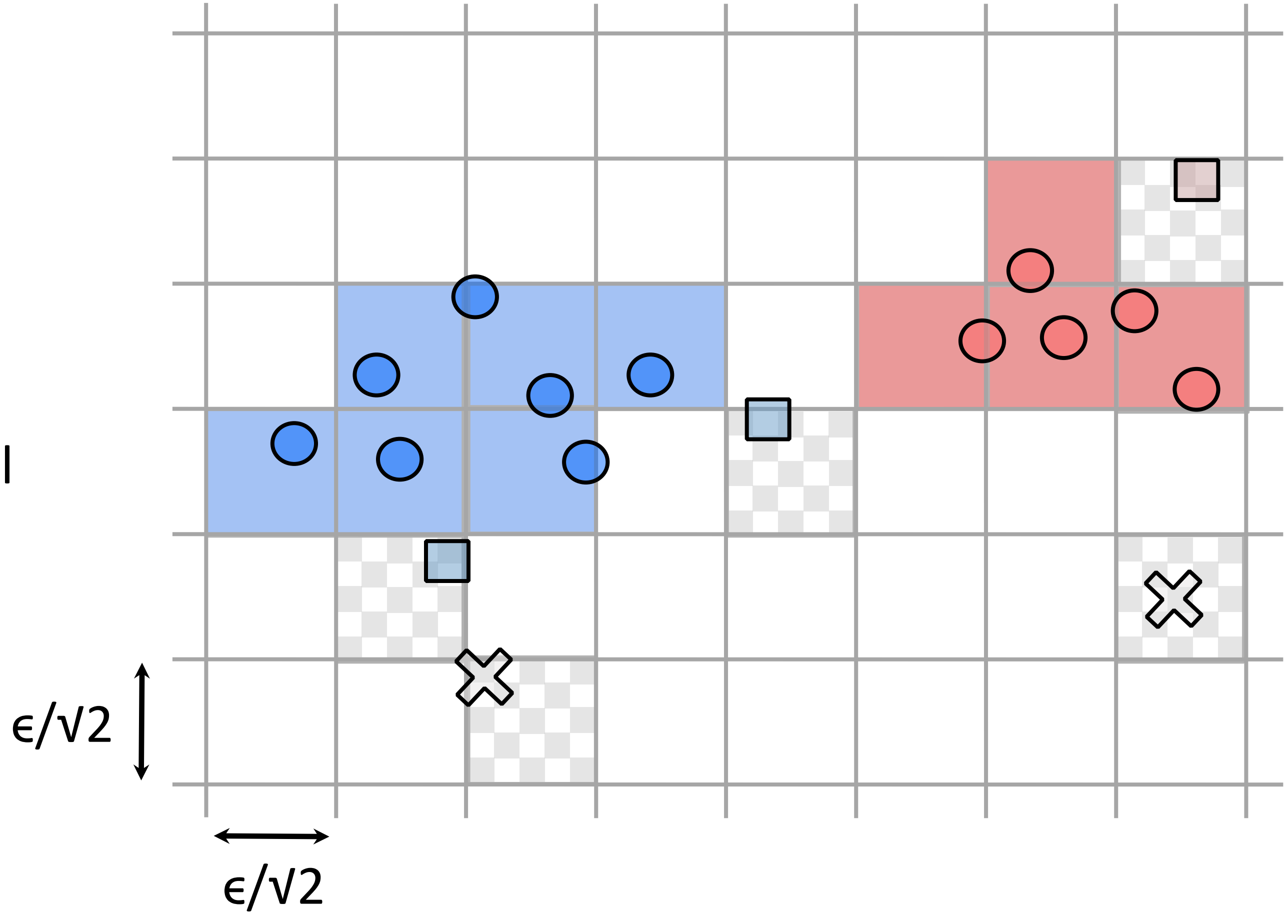
1. Construct grid cells
    - O(n) work and O(log n) span for semisorting and hash table construction
2. Mark core points
    - Each cell will be checked by 21 * O(minPts) many points
    - O(log n) span for summing counts
    - Thus, range queries take O(n * minPts) work and O(log n) span
3. Cell graph
    - Can build Delaunay triangulation and keep the triangulation edges of distance $\leq \epsilon$ (other approaches described in paper)
        - O(n log n) work and O(log n) span
    - Use connectivity algorithm to find clusters in O(n) work and O(log n) span
4. Cluster border points
    - Similar analysis as marking core points
5. Total: O(n log n) work and O(log n) span (for constant minPts)

Our Parallel DBSCAN Algorithm

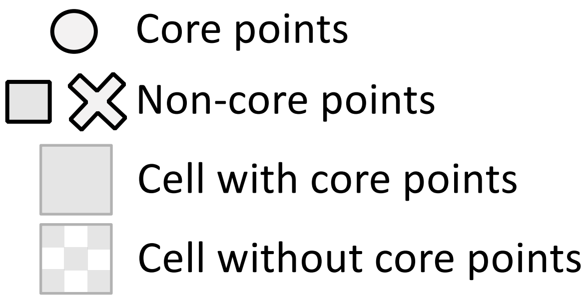


1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points

- Differences for higher-dimensional exact and approximate algorithms
  - Grid size is $\epsilon/\sqrt{d}$ instead of $\epsilon/\sqrt{2}$
  - How BCP queries are computed

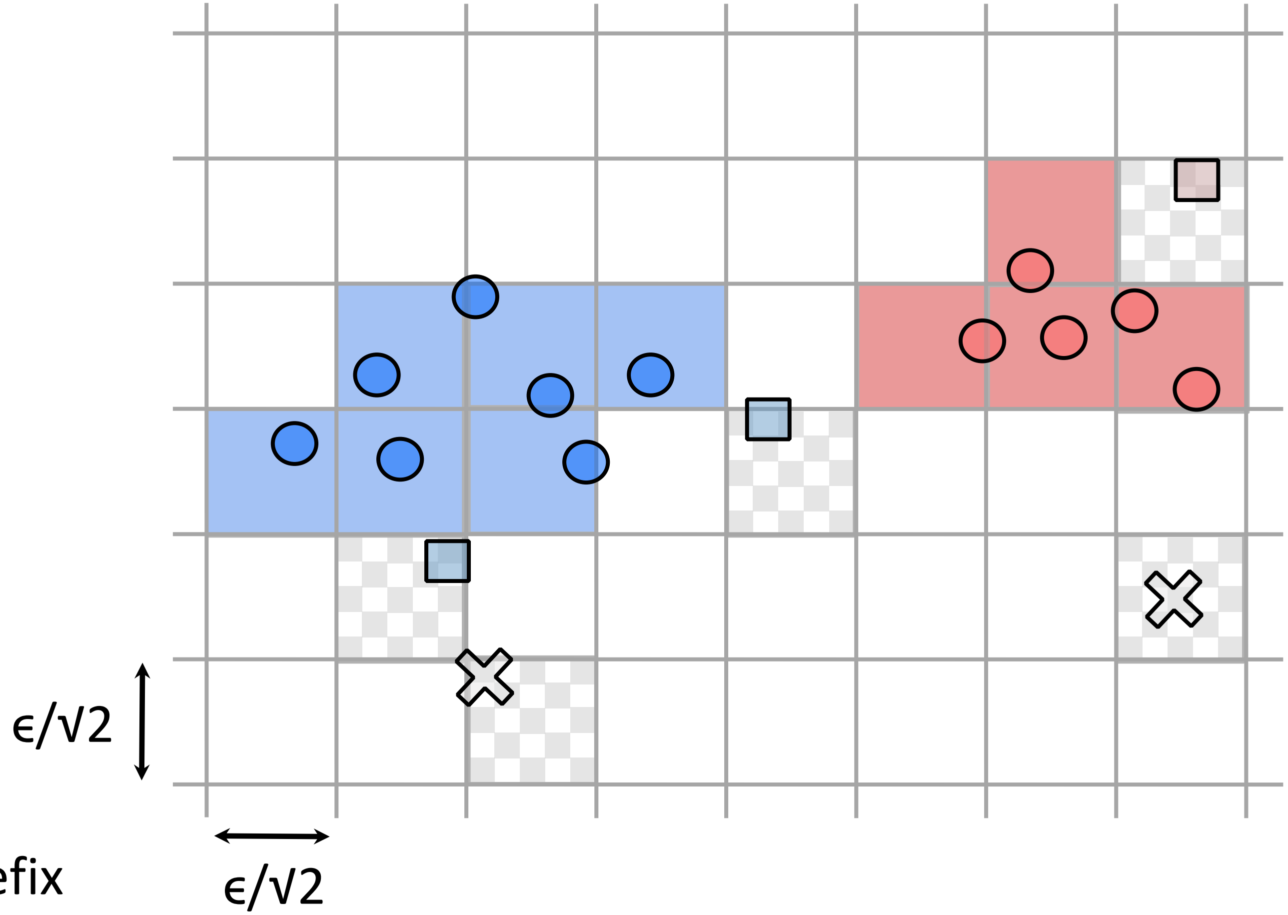


Our Parallel DBSCAN Algorithm

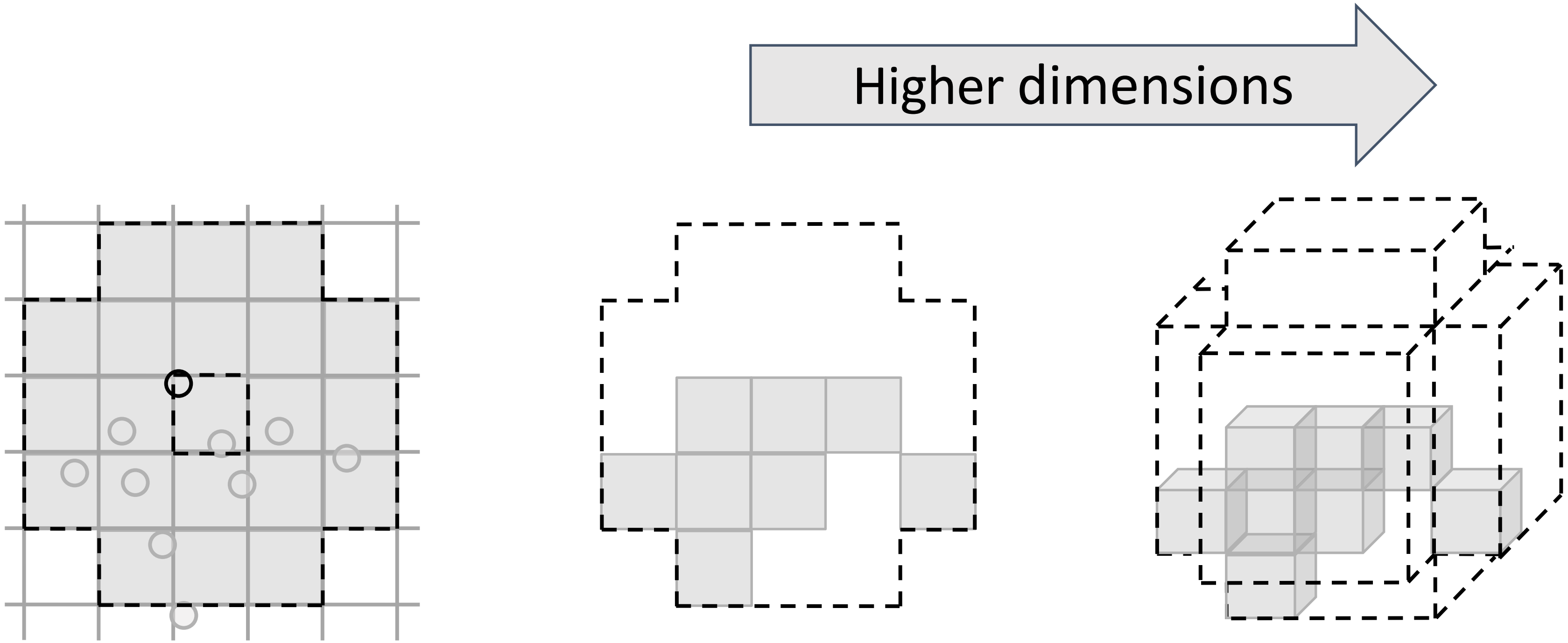


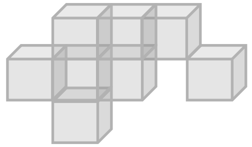
1. Construct grid cells
2. Mark core points
3. Cell graph
4. Cluster border points

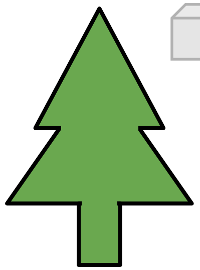
- Our work bound matches the sequential bounds of de Berg et al. and Gan and Tao
  - O(n log n) for 2D, subquadratic for d > 2, O(n) for approximate
  - BCP queries dominate work
- Can implement all operations in polylogarithmic span
  - Parallel primitives: hashing, prefix sums, semisorting, merging, pointer jumping, Delaunay



Optimization - Spatial Tree



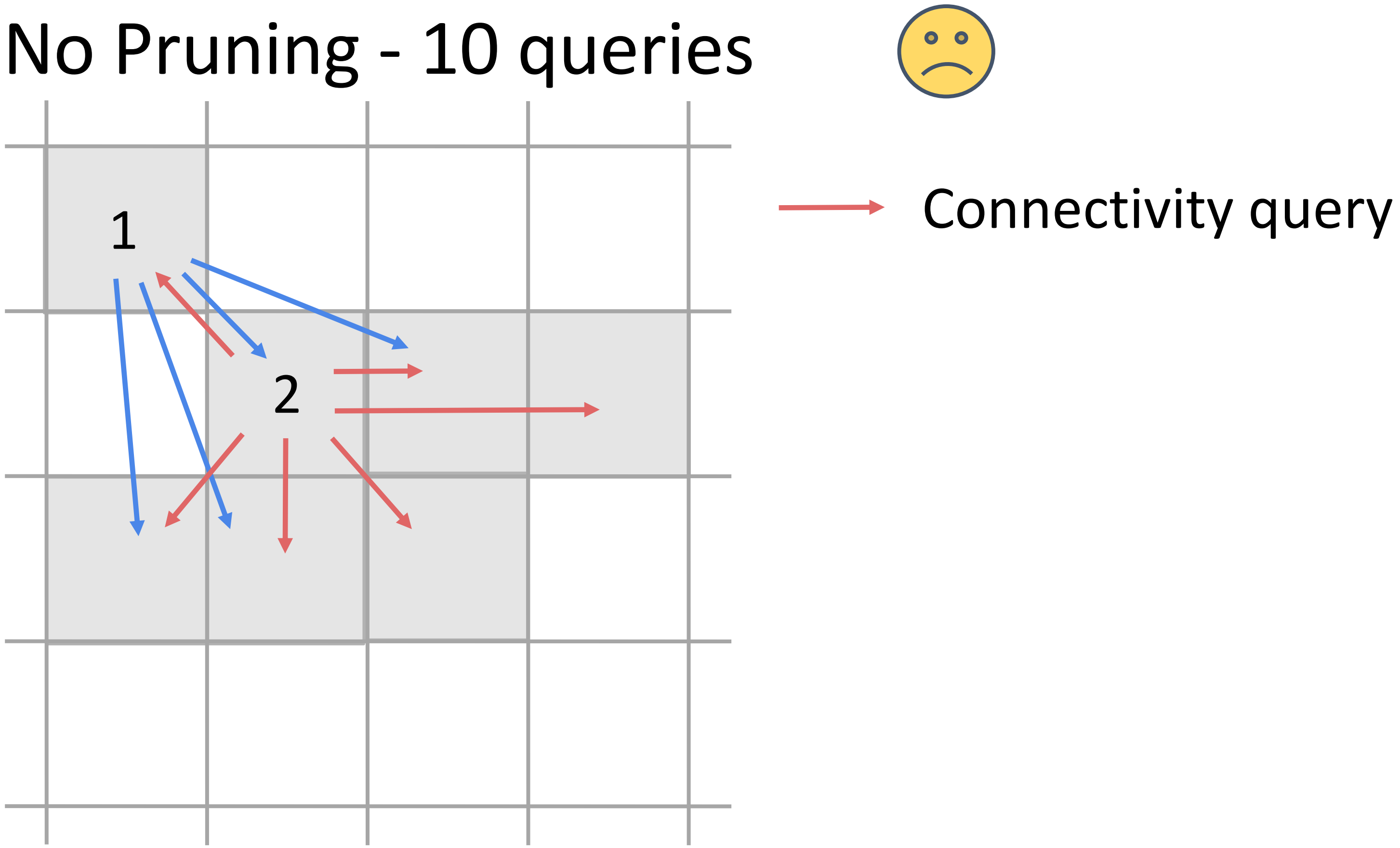




Maintain cells in a kd-tree

Optimization - Parallel Pruning of BCP Queries

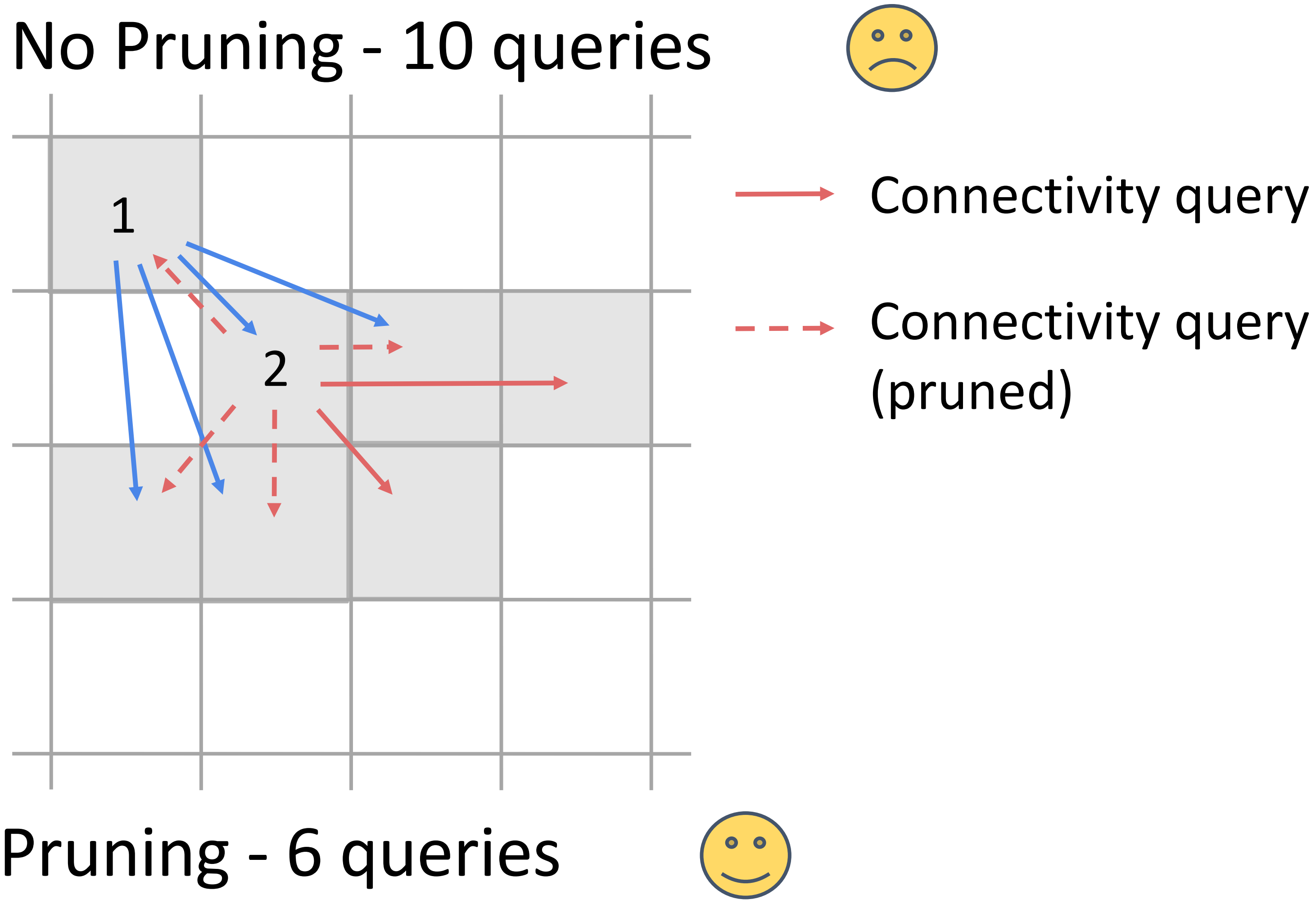
No Pruning - 10 queries



Optimization - Parallel Pruning of BCP Queries

- Parallel union-find keeps connectivity on-the-fly
  - First used by Gan and Tao sequentially
- Prunes query if already connected
- Prunes query if repeated

- Order in which cells are processed affects pruning quality
  - Bucket cells based on #points and process each bucket in parallel

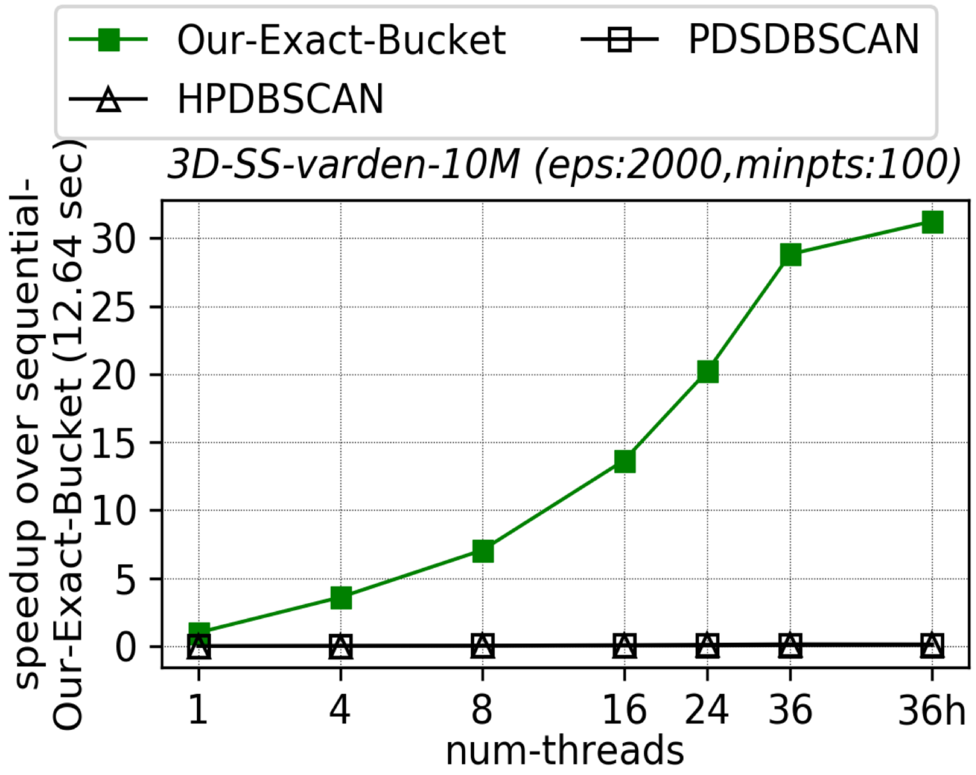


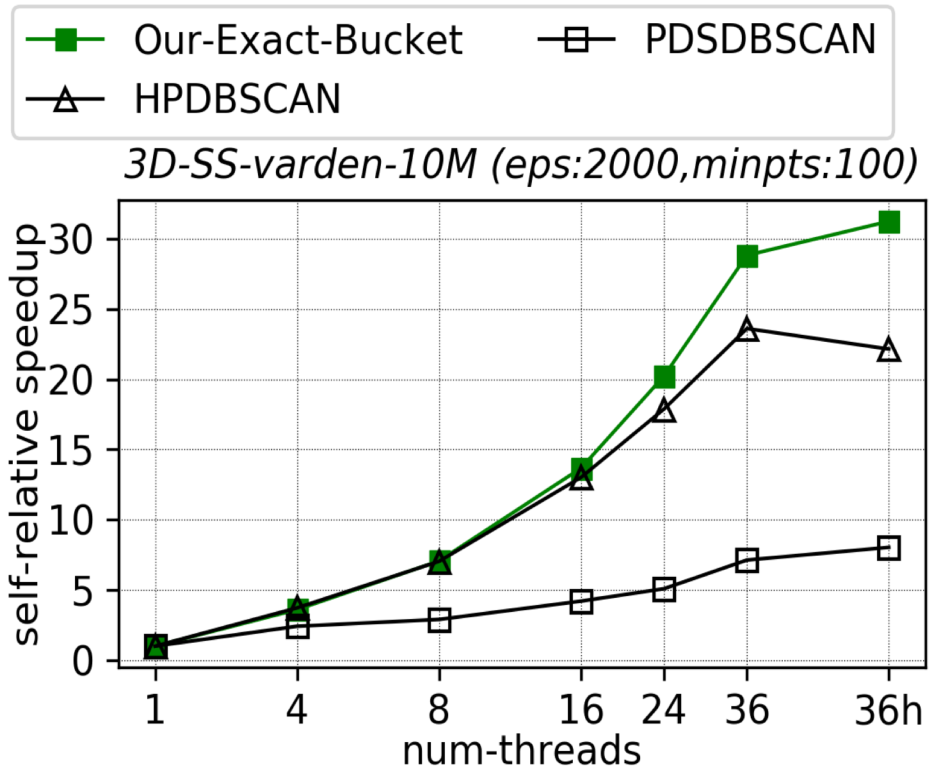
Experimental Setup

- AWS c5.18x Large
  - 2 × Intel Xeon Platinum 8124M (3.00GHz) CPUs
  - 36 cores, 2-way hyperthreading
  - 144 GiB RAM

- AWS r5.24x Large (only used for larger datasets)
  - 2 × Intel Xeon Platinum 8175M (2.50 GHz) CPUs
  - 48 cores, 2-way hyperthreading
  - 768 GiB RAM

Good Work-Efficiency and Scalability

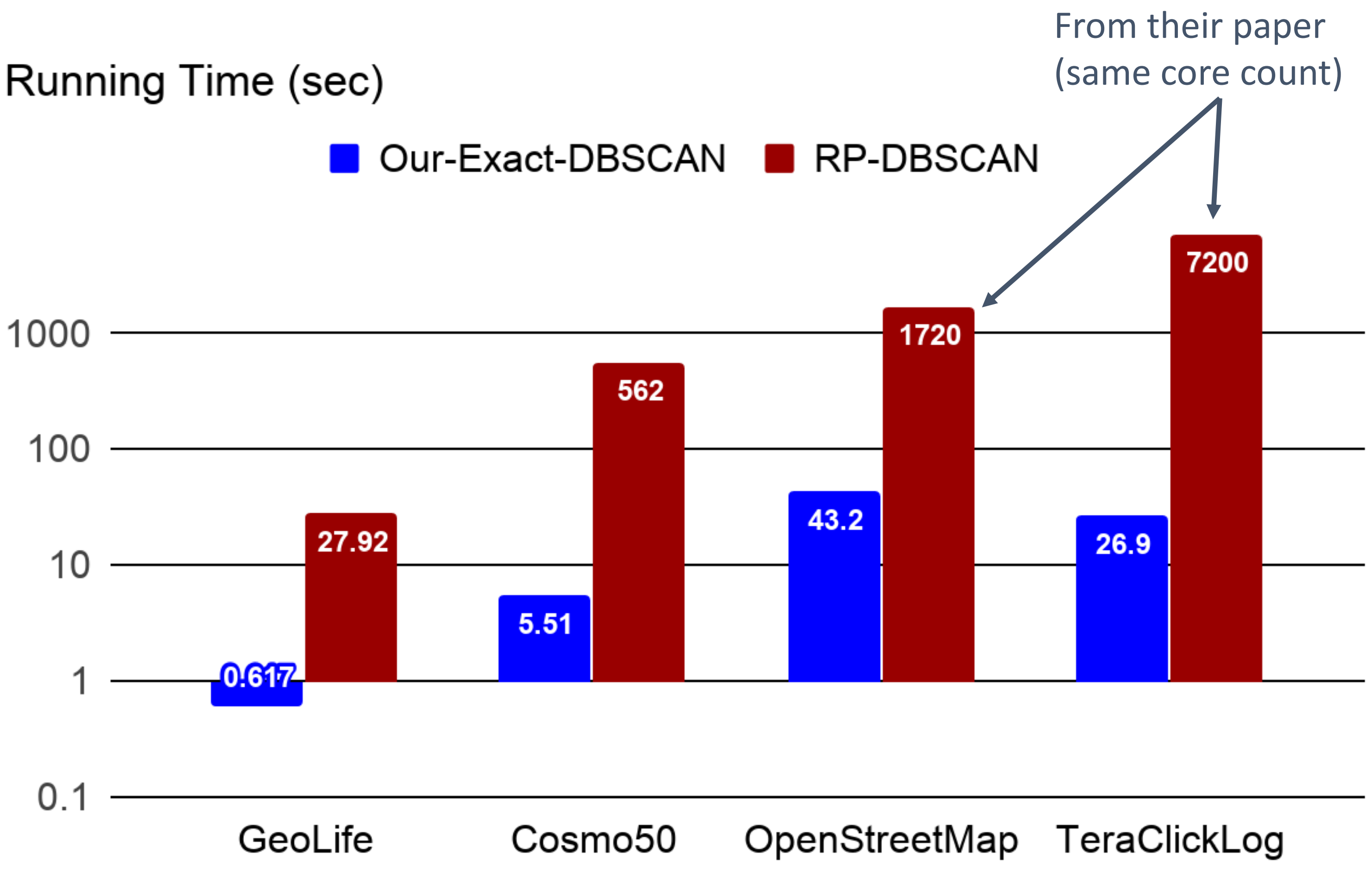




- 16-6102x faster than HPDBSCAN and PDSDBSCAN across all datasets and parameter settings

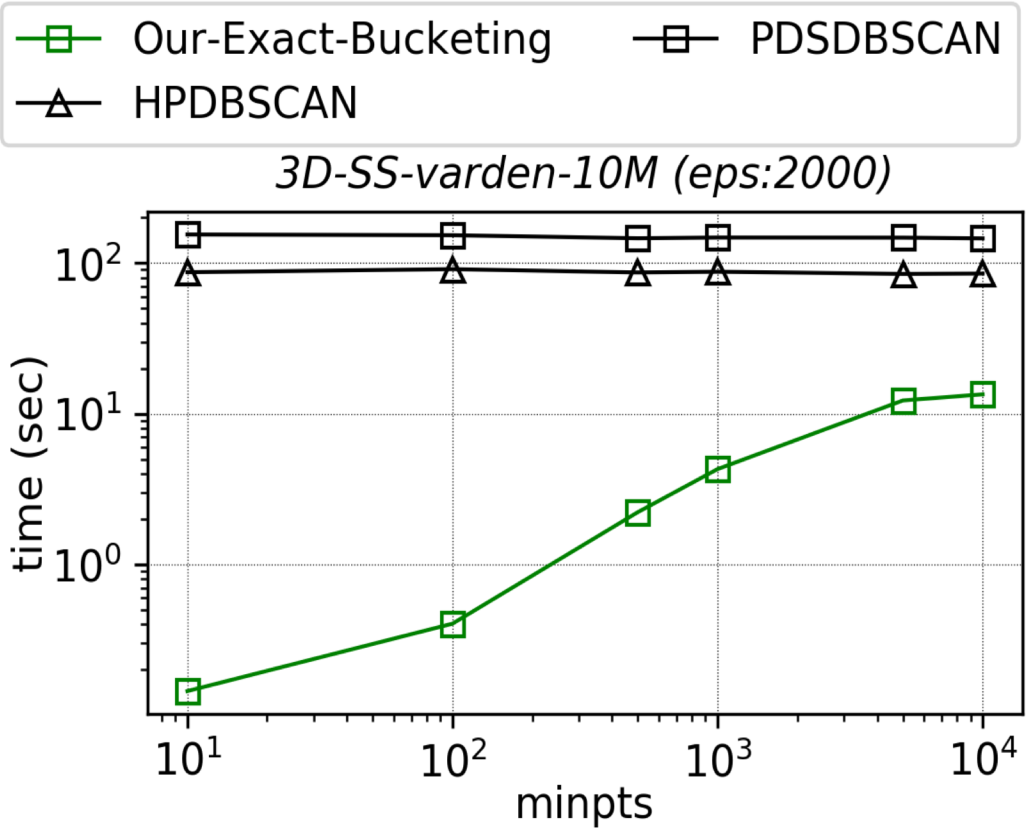
Good Speedup over State-of-art Parallel Implementation

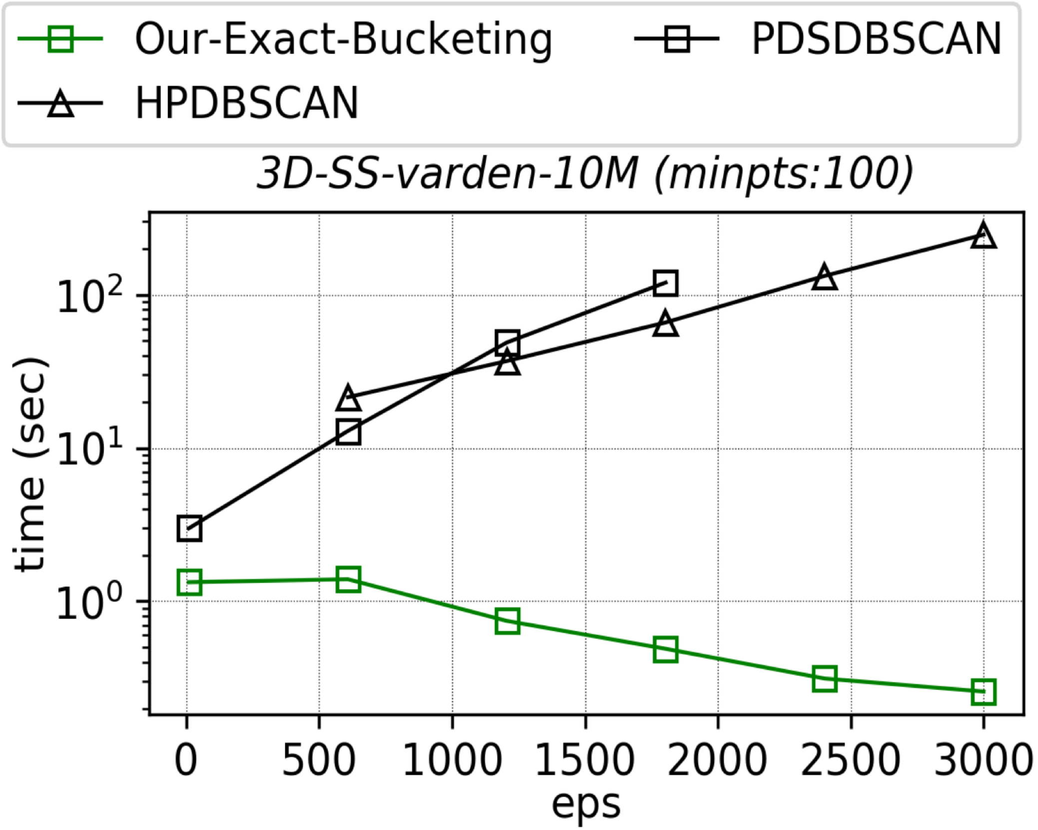
	#Data Points	Dimension
GeoLife	24.9 M	3
Cosmo50	321 M	3
OpenStreetMap	2770 M	2
TeraClickLog	4373 M	13



- 18-577x faster than RP-DBSCAN

Varying Parameters

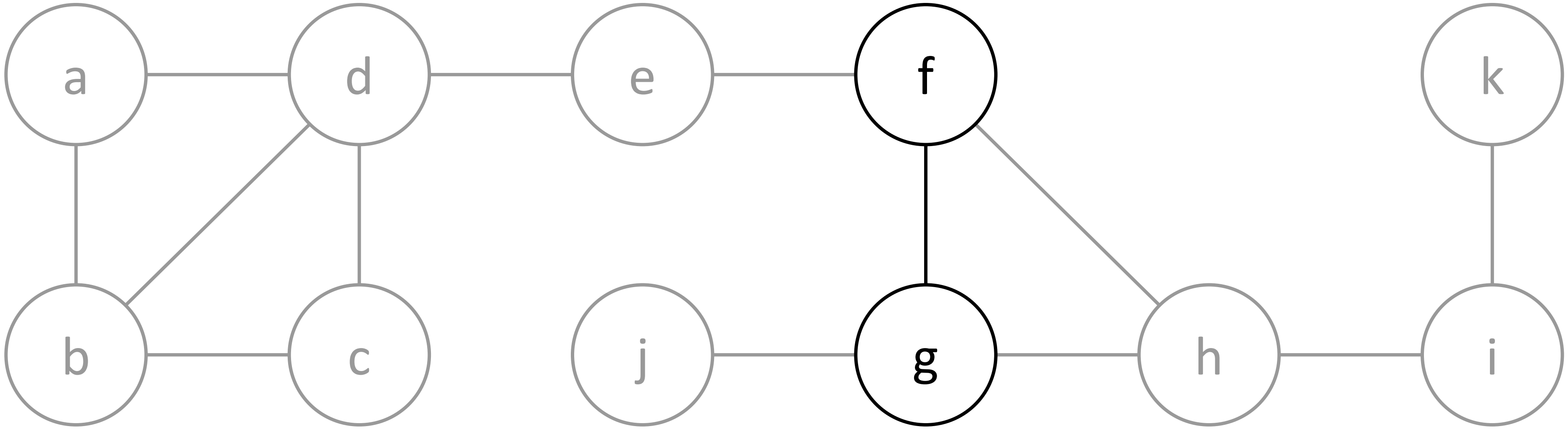




SCAN for Graph Clustering

SCAN Definition

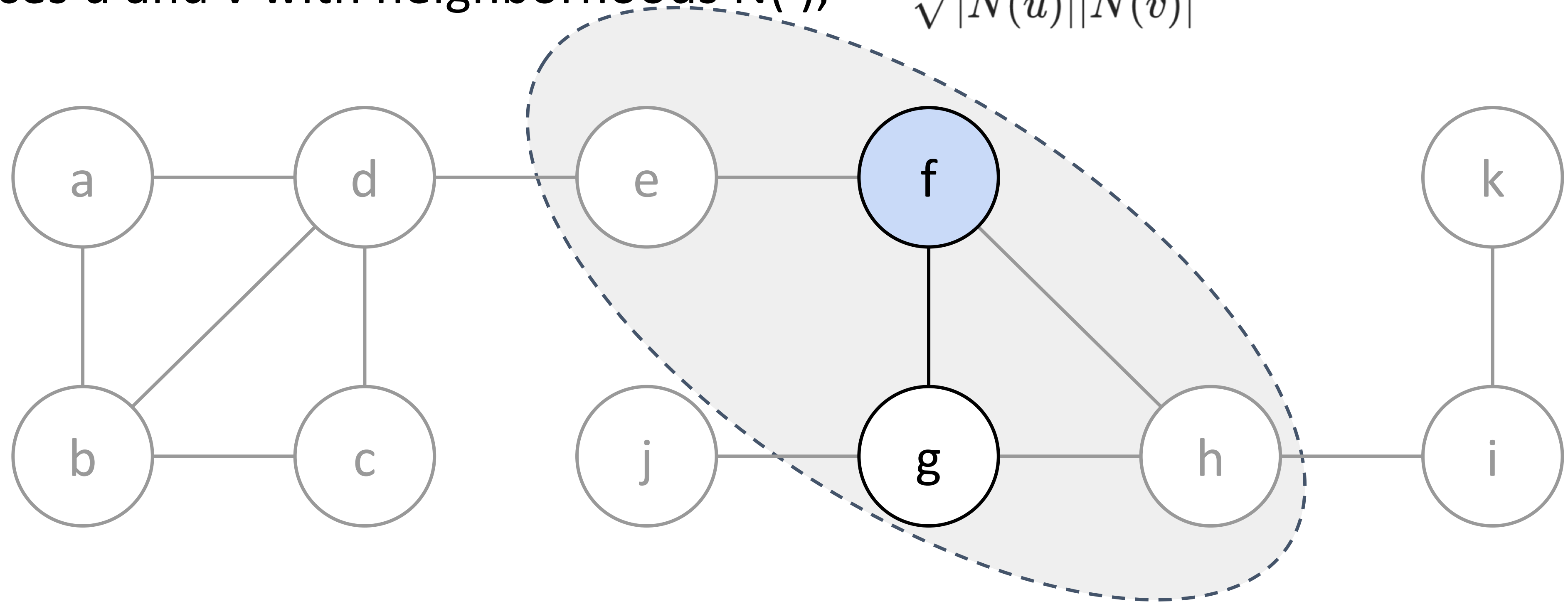
- A pair of adjacent vertices is **similar** if they share many neighbors
- Original SCAN algorithm uses cosine similarity
  - for vertices u and v with neighborhoods N(·), $\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$



SCAN Definition

- A pair of adjacent vertices is similar if they share many neighbors
- Original SCAN algorithm uses cosine similarity
  - for vertices u and v with neighborhoods N(·), $\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$

$|N(f)| = 4$

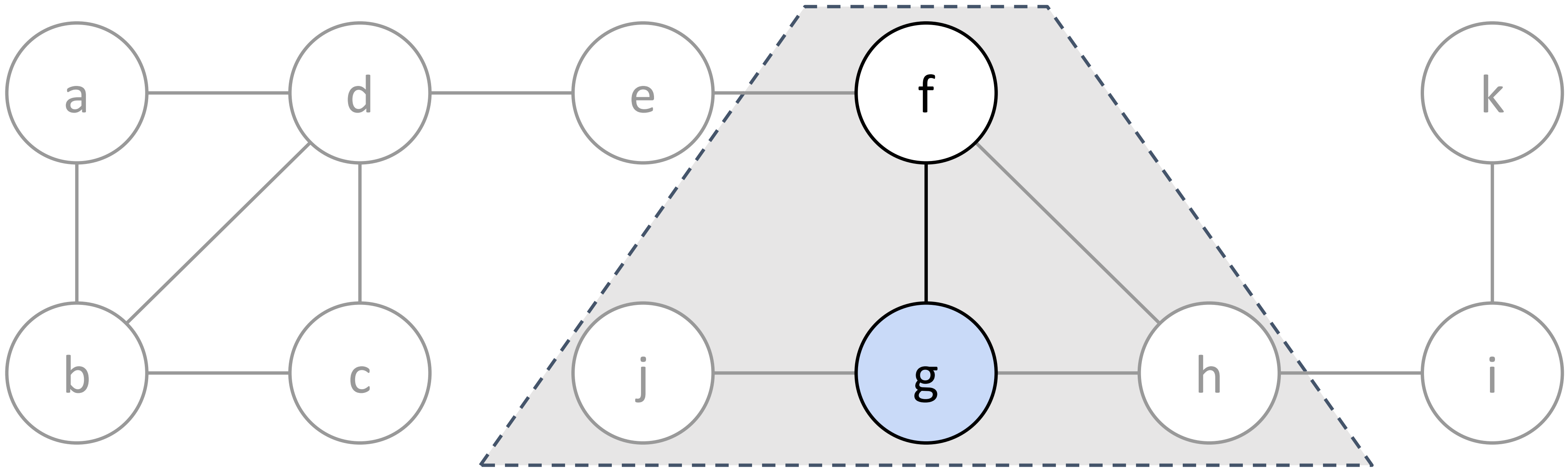


SCAN Definition

- A pair of adjacent vertices is similar if they share many neighbors
- Original SCAN algorithm uses cosine similarity
  - for vertices u and v with neighborhoods N(·), $\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$

$|N(f)| = 4$

$|N(g)| = 4$



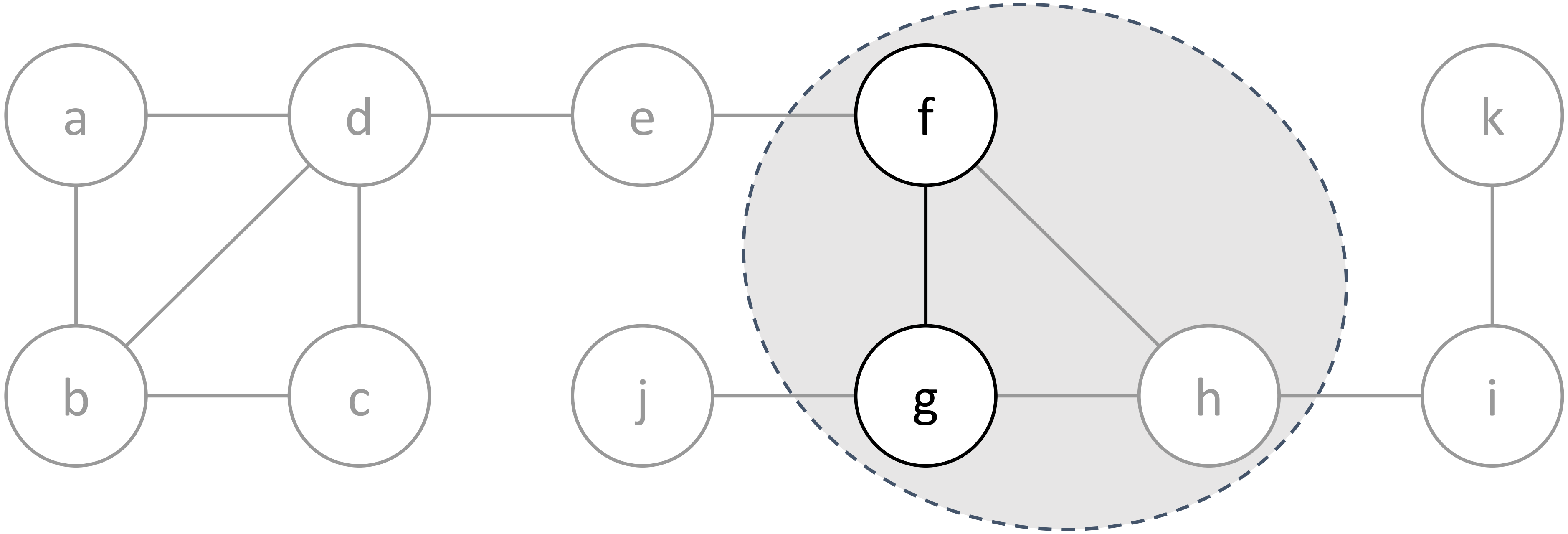
SCAN Definition

- A pair of adjacent vertices is similar if they share many neighbors
- Original SCAN algorithm uses cosine similarity
  - for vertices u and v with neighborhoods N(·), $\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$

$|N(f)| = 4$

$|N(g)| = 4$

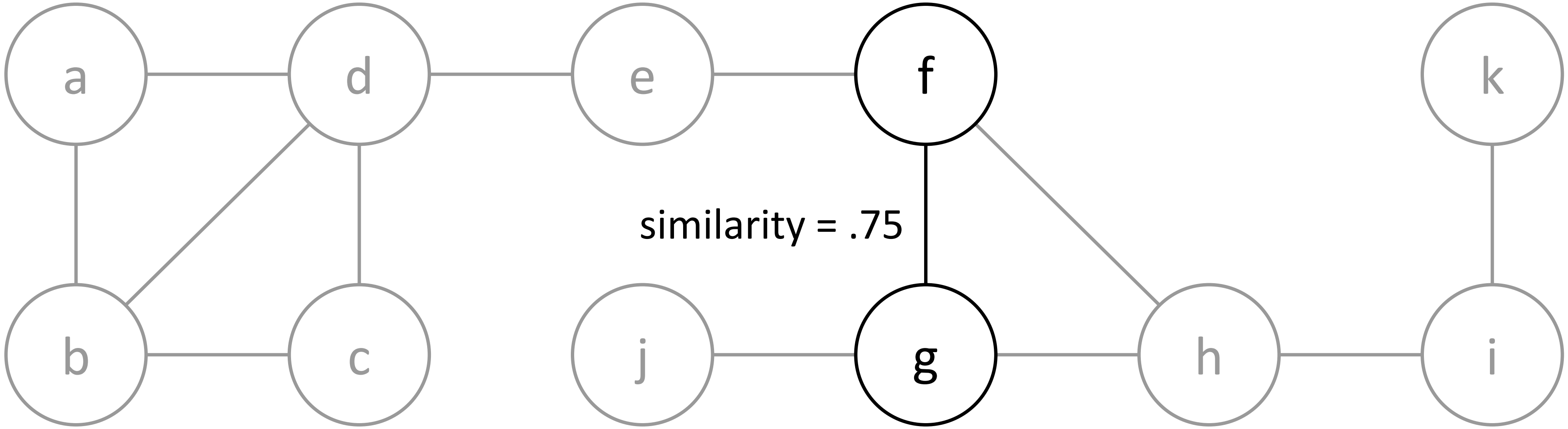
$|N(f) \cap N(g)| = 3$



SCAN Definition

- A pair of adjacent vertices is similar if they share many neighbors
- Original SCAN algorithm uses cosine similarity
    - for vertices u and v with neighborhoods N(·), $\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$

$$|N(f)| = 4$$
$$|N(g)| = 4$$
$$|N(f) \cap N(g)| = 3$$



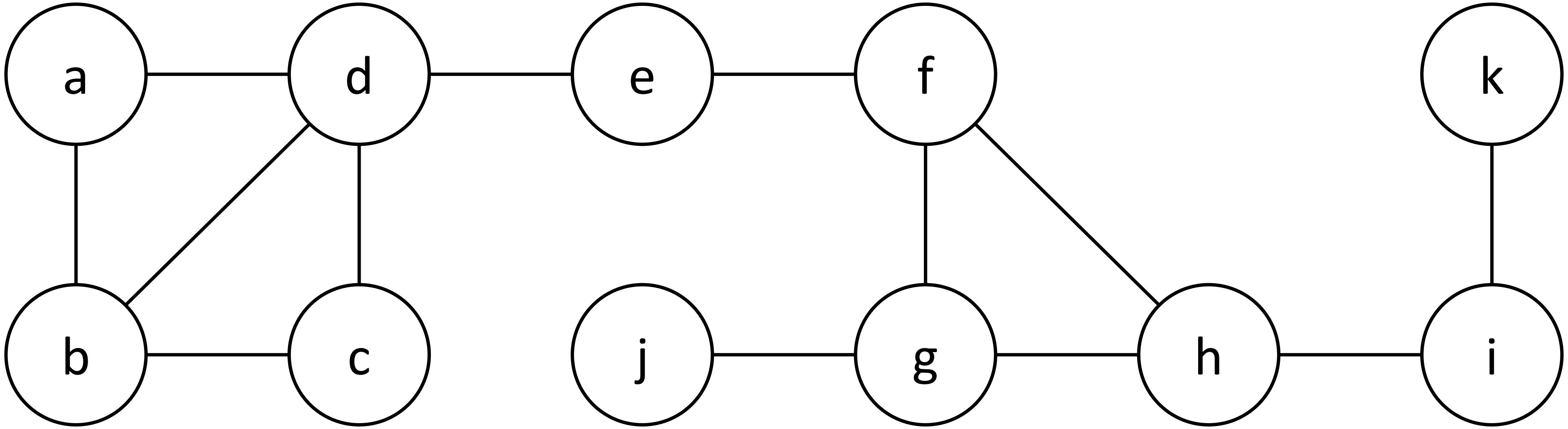
SCAN Definition

- A pair of adjacent vertices is similar if they share many neighbors
- Original SCAN algorithm uses cosine similarity
  - for vertices u and v with neighborhoods N(·), $\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$
- Other similarity functions we consider:
  - Jaccard similarity
  - Weighted cosine similarity

SCAN Definition

- User-selected parameters: μ, ε
- Vertex is a **core** vertex if it has at least μ neighbors that are ε-similar

μ = 3
ε = .6

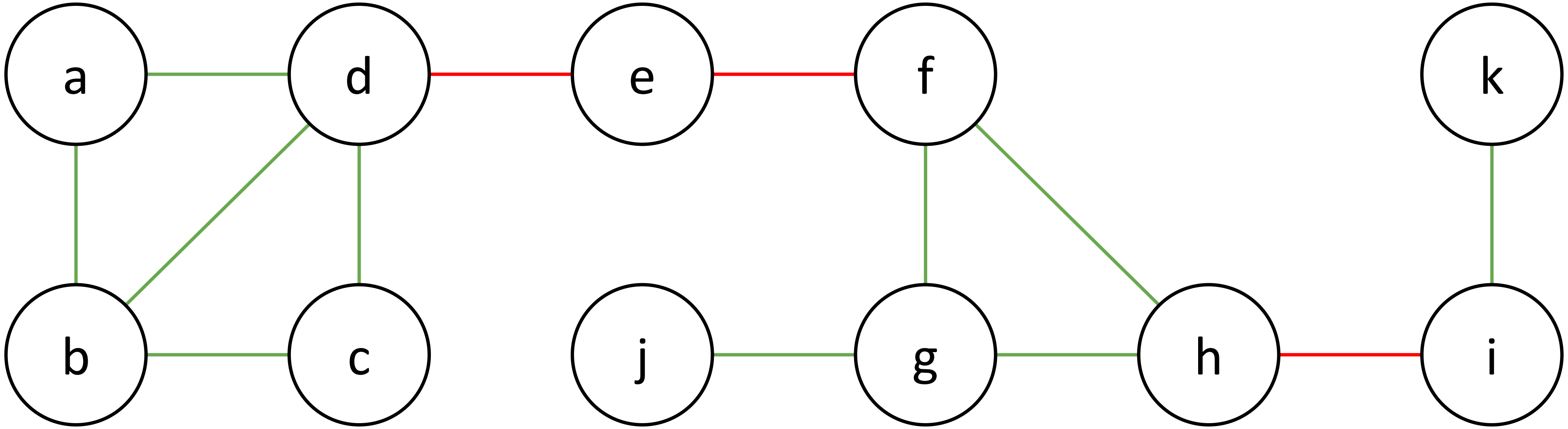


SCAN Definition

- User-selected parameters: μ, ε
- Vertex is a **core** vertex if it has at least μ neighbors that are ε-similar

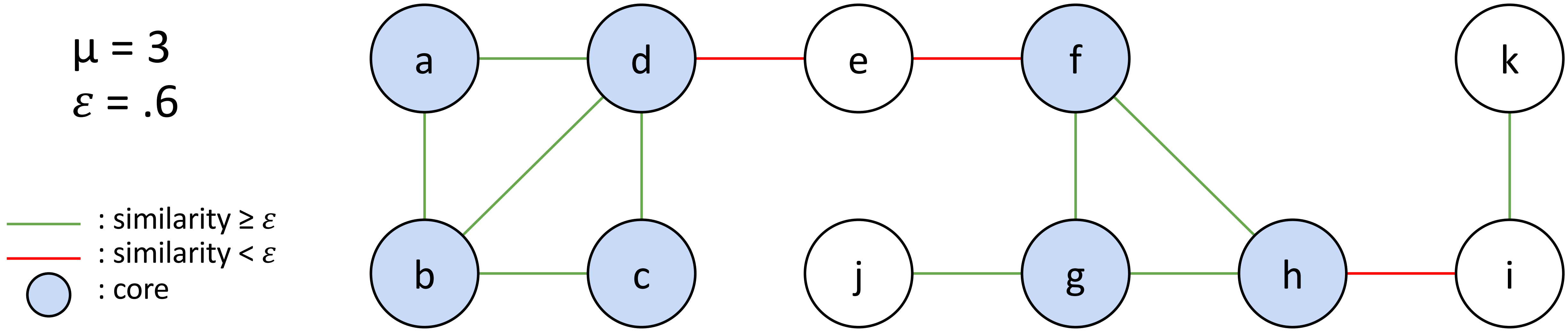
$\mu = 3$
$\varepsilon = .6$

: similarity ≥ ε
: similarity < ε



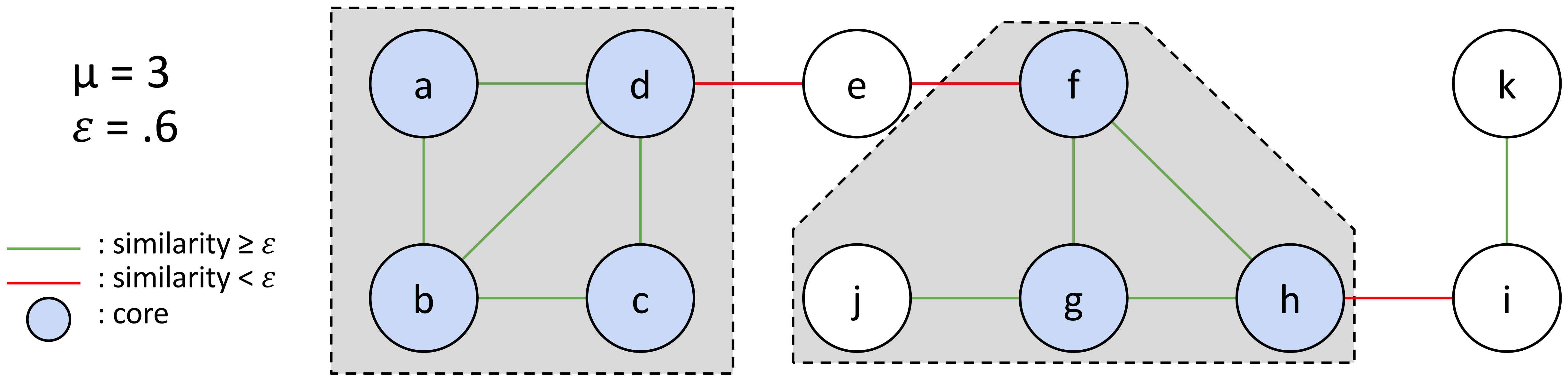
SCAN Definition

- User-selected parameters: μ, ε
- Vertex is a **core** vertex if it has at least μ neighbors that are ε-similar



SCAN Definition

- Clusters: connected component of core vertices along with any other ε-similar neighbors (**border** vertices)
- **Outliers** are vertices not belonging to any cluster



SCAN Complexity

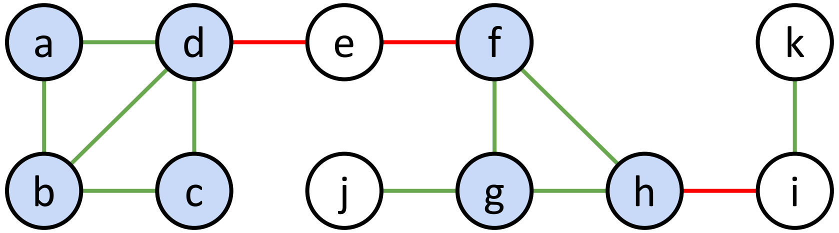
- Work of SCAN: $O(m\alpha) \leq O(m^{1.5})$
  - Arboricity (α): a measure of graph sparsity
  - Computing similarities is the expensive part: $O(m\alpha)$
  - Finding clusters from similarities: $O(m)$
- SCAN is especially costly for dense graphs
- Furthermore, users often have to try many different parameters to obtain good clusters

GS-Index: precompute index to test parameters quickly

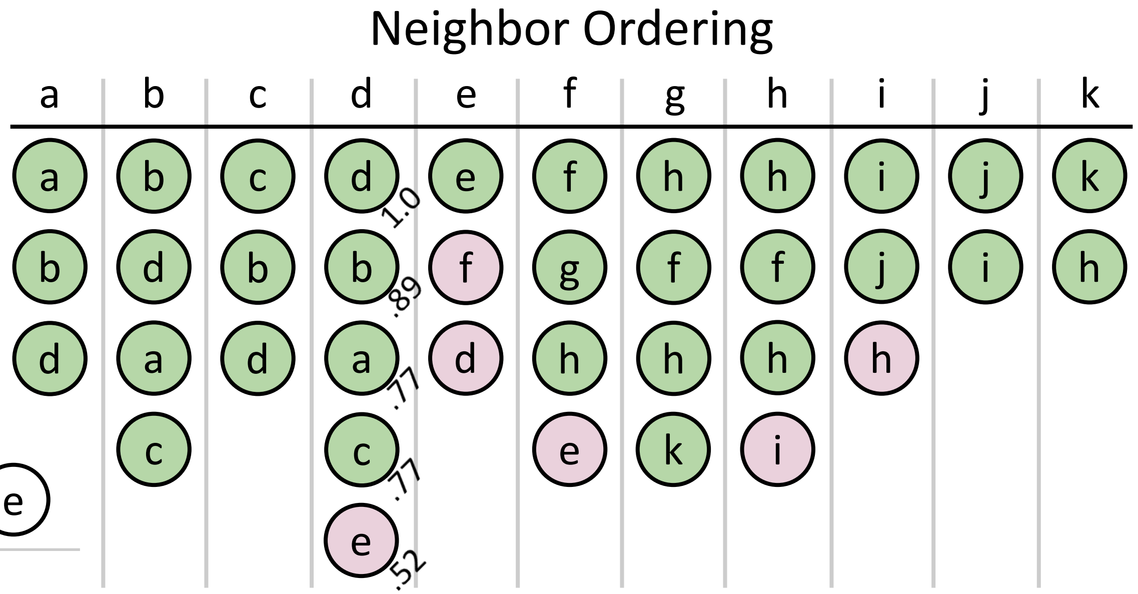
- SCAN variant GS-Index constructs an index from which querying for clusters under arbitrary μ and ε is fast (Wen et al., VLDB 2017)
- Maintain **neighbor ordering** to quickly find ε-similar neighbors
  - Vertices' neighbor lists are sorted in decreasing order by similarity
- Maintain **core ordering** to quickly find core vertices
  - For each μ, store list of vertices sorted in decreasing order by the maximum value of ε such that the vertex is a core vertex

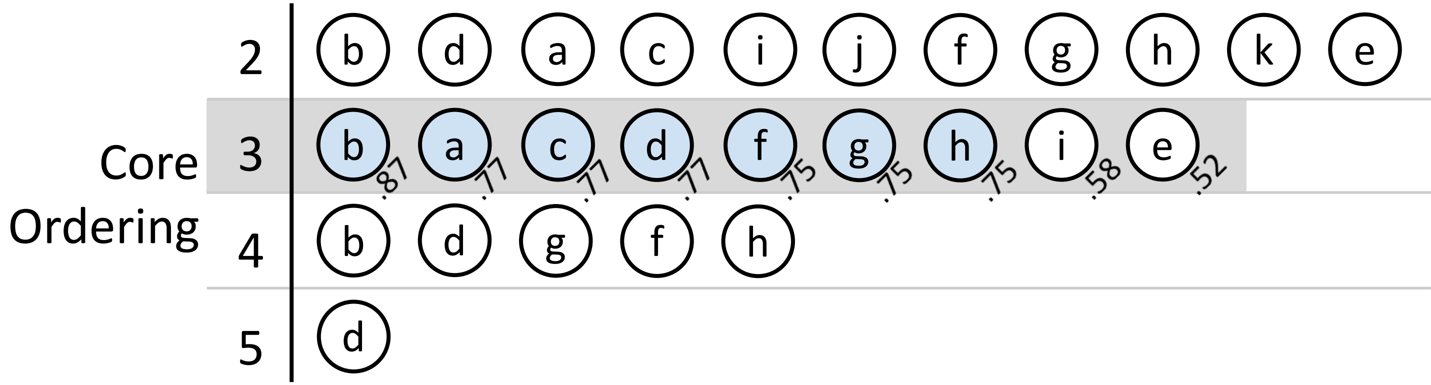
GS-Index: precompute index to test parameters quickly

- **Neighbor ordering**: vertices' neighbor lists sorted by similarity
- **Core ordering**: For each μ, vertices sorted by max ε at which vertex is a core



$\mu = 3$, $\varepsilon = .6$





Get clusters by BFS on core vertices and ε-similar edges extracted from index

GS-Index gives fast queries but is still sequential

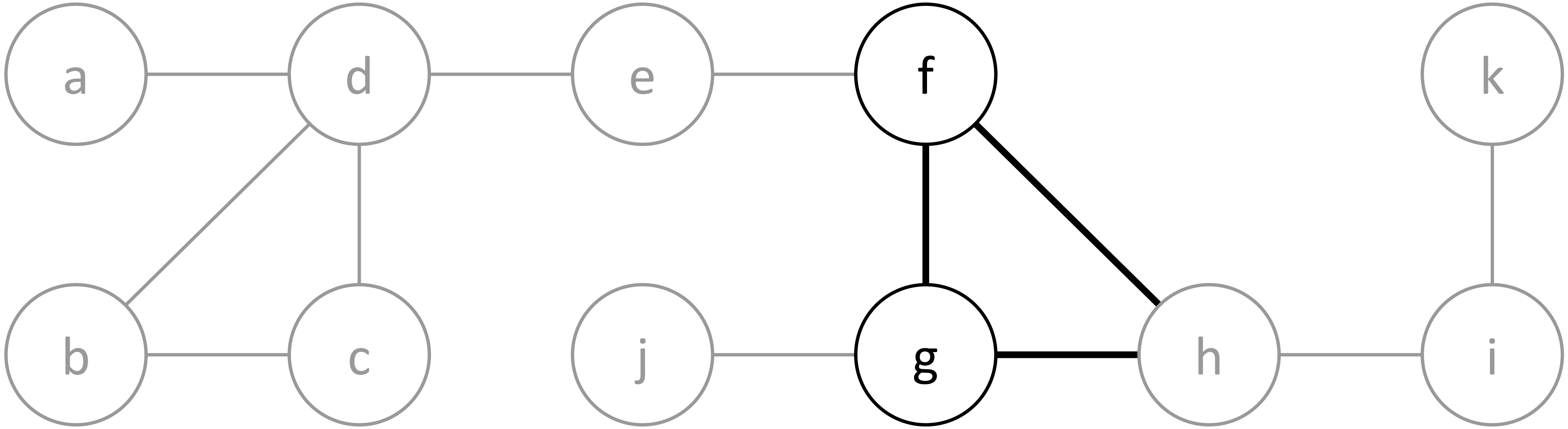
- Work to compute index: O((α + log n)m)
  - Cost for computing similarities and sorting
- Work to query for clusters: linear in the total sizes of clusters
  - No work done for non-ε-similar edges and unclustered vertices
- Queries are fast, but computing the index sequentially is slow

Our contributions

- Parallel index-based SCAN algorithm
    - Provably work-efficient with logarithmic span
- Approximate similarity computation via locality-sensitive hashing for even greater speedups
- Practical, optimized multicore implementations that empirically outperform state-of-the-art SCAN algorithms

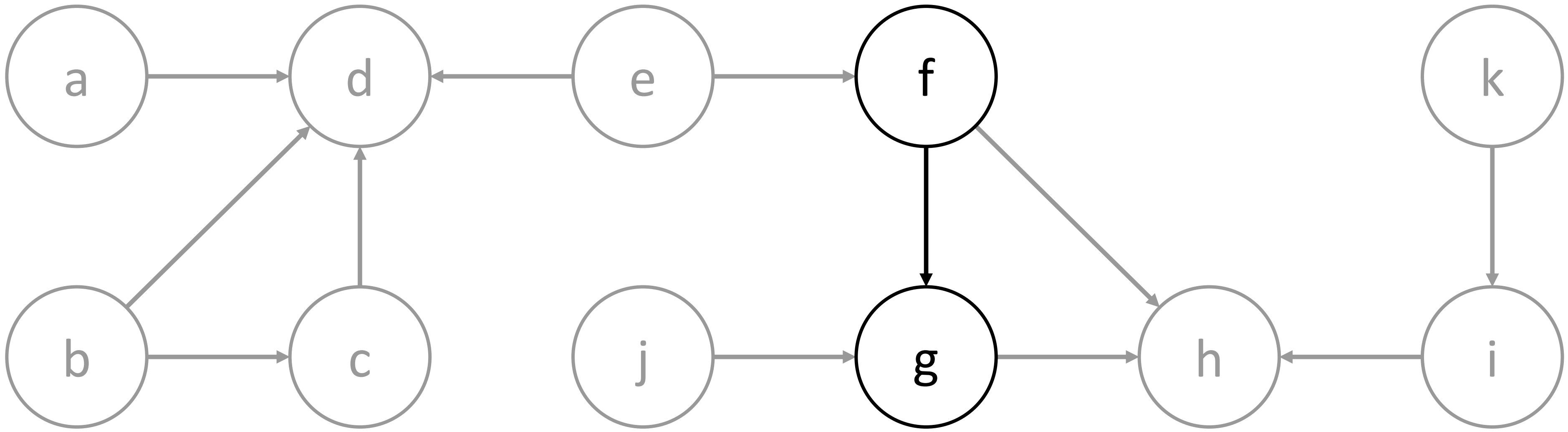
Computing similarities

- Finding shared neighbors is counting triangles
  - This can be done in O(αm) work and O(log n) span with high probability using parallel hash tables
- Important to optimize similarity computation since it's so costly



Computing similarities

- Count each triangle once instead of three times by directing the graph and counting directed triangles (Latapy 2008)
    - Direct each edge from lower-degree to higher-degree endpoint
- For better cache locality, instead of using parallel hash tables, intersect sorted neighbor lists with parallel merge (Shun and Tangwongsan 2015)

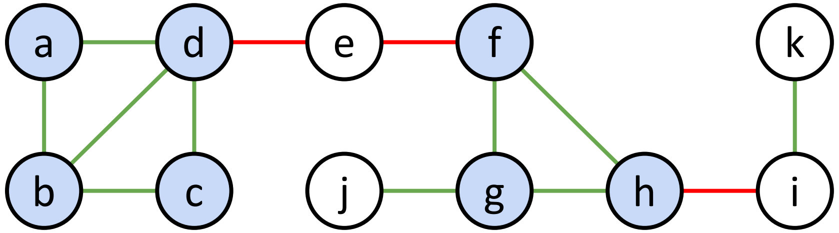


Computing neighbor and core orderings

- Use parallel comparison sort
- Additional observation: can integer sort on unweighted graphs to get better work bounds
  - Transform similarities monotonically into integers
    - $\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}} \rightarrow \left\lfloor \left(\frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}} \right)^2 n^4 \right\rfloor$
  - Reduces the log n term in the O((α + log n)m) work bound
    - O(αm) work with O(n^β) span, or
    - O((α + log log n)m) work and O(log n) span

Querying: doubling search on index

- Doubling search to find core vertices and ε-similar edges from index



$\mu = 3$, $\varepsilon = .6$

Neighbor Ordering

a	b	c	d	e	f	g	h	i	j	k
a	b	c	d	e	f	h	h	i	j	k
b	d	b	b	f	h	f	f	j	i	h
d	a	d	a	d	h	h	h	h		
			c		e	k	i			
			e							

Core Ordering

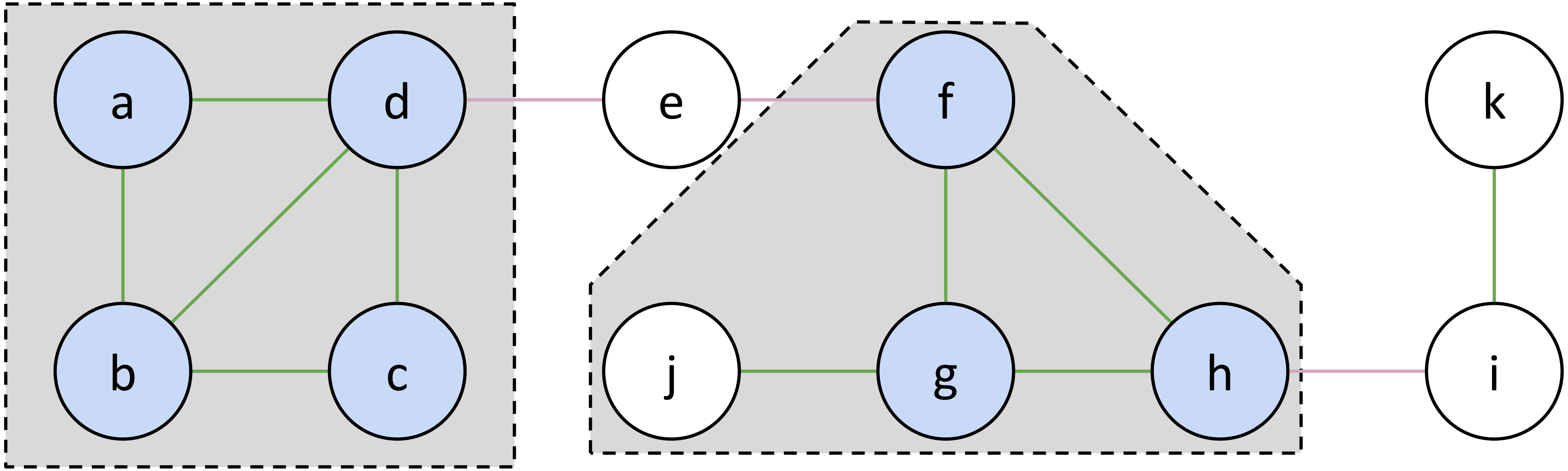
2	b	d	a	c	i	j	f	g	h	k
3	b	a	c	d	f	g	h	i	e	
4	b	d	g	f	h					
5	d									

Querying: finding clusters

- Parallel connectivity on core vertices and ε-similar edges
- In theory, we use a linear work and O(log n) span connected components algorithm
- In practice, we use a parallel union-find data structure

$\mu = 3$
$\varepsilon = .6$

: similarity $\geq \varepsilon$
: similarity $< \varepsilon$
: core



Our Work: Approximating similarities

- Similarity computation in index construction is still the computational bottleneck, especially on dense graphs
- Locality-sensitive hashing (LSH) approximates similarity between vertices
    - SimHash for cosine similarity
    - MinHash for Jaccard similarity
- LSH sample size k trades accuracy vs. running time

LSH increases speed on dense graphs

- For sample size k, further reduce the $O((\alpha + \log n)m)$ work bound to
  - $O(km)$ work with $O(n^{\beta})$ span, or
  - $O((k + \log \log n)m)$ work and $O(\log n)$ span

LSH still maintains guarantees on resulting clusters

- We prove that if the number of samples k is sufficiently large, we correctly "classify" all edges as above or below ε in similarity, except inside a small interval around ε

LSH heuristic: only LSH on high-degree vertices

- If neighborhoods are small, better to just compute exact similarities
- Solution: use LSH on pairs of high-degree vertices, and use triangle counting elsewhere

Experimental Setup

- AWS machine
  - 48 cores, two-way hyperthreading (max 96 hyper-threads)
  - 192 GiB of RAM

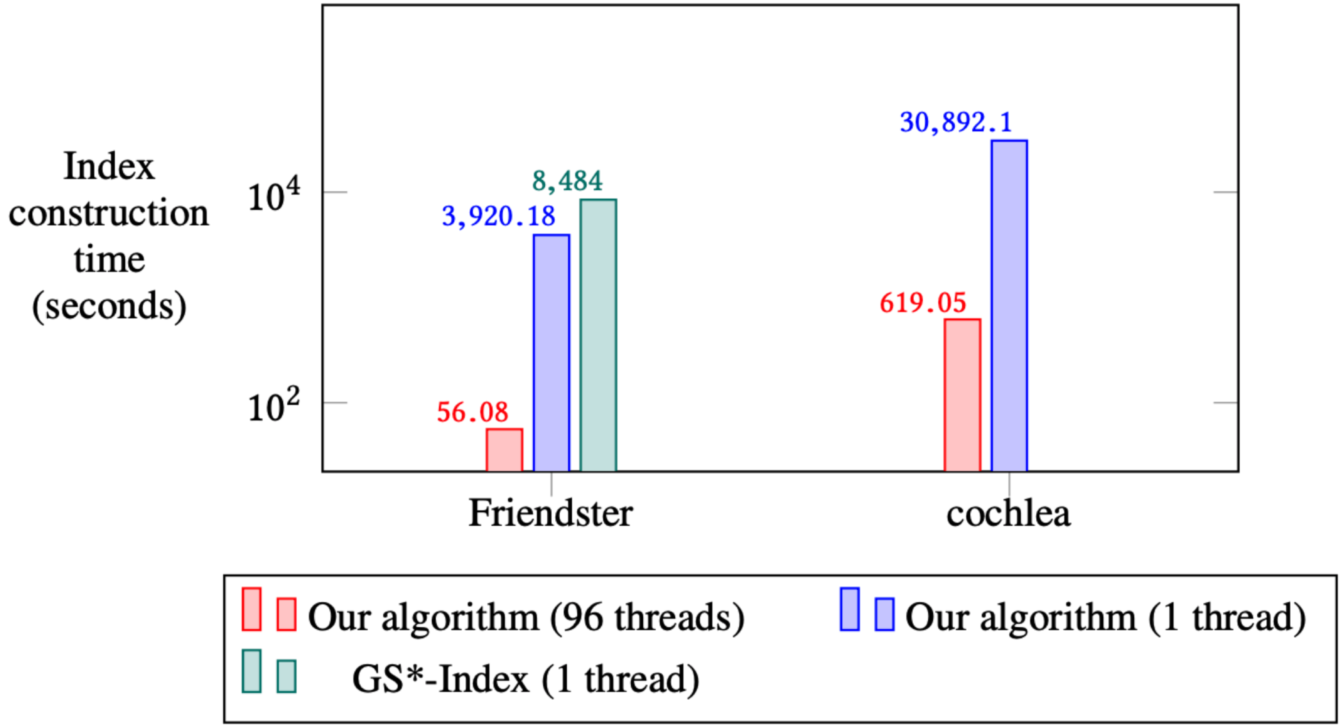
Comparison against state-of-the-art

- ppSCAN: fastest parallel SCAN algorithm (Che et al., ICPP 2018)
- GS-Index: original (sequential) index-based SCAN algorithm (Wen et al., VLDB 2017)

Exact index construction: 50–151× speedup vs. GS-Index

Friendster graph: large social network (65M vertices, 1.8B edges)

Cochlea graph: dense, weighted biological graph (26K vertices, 282M edges)

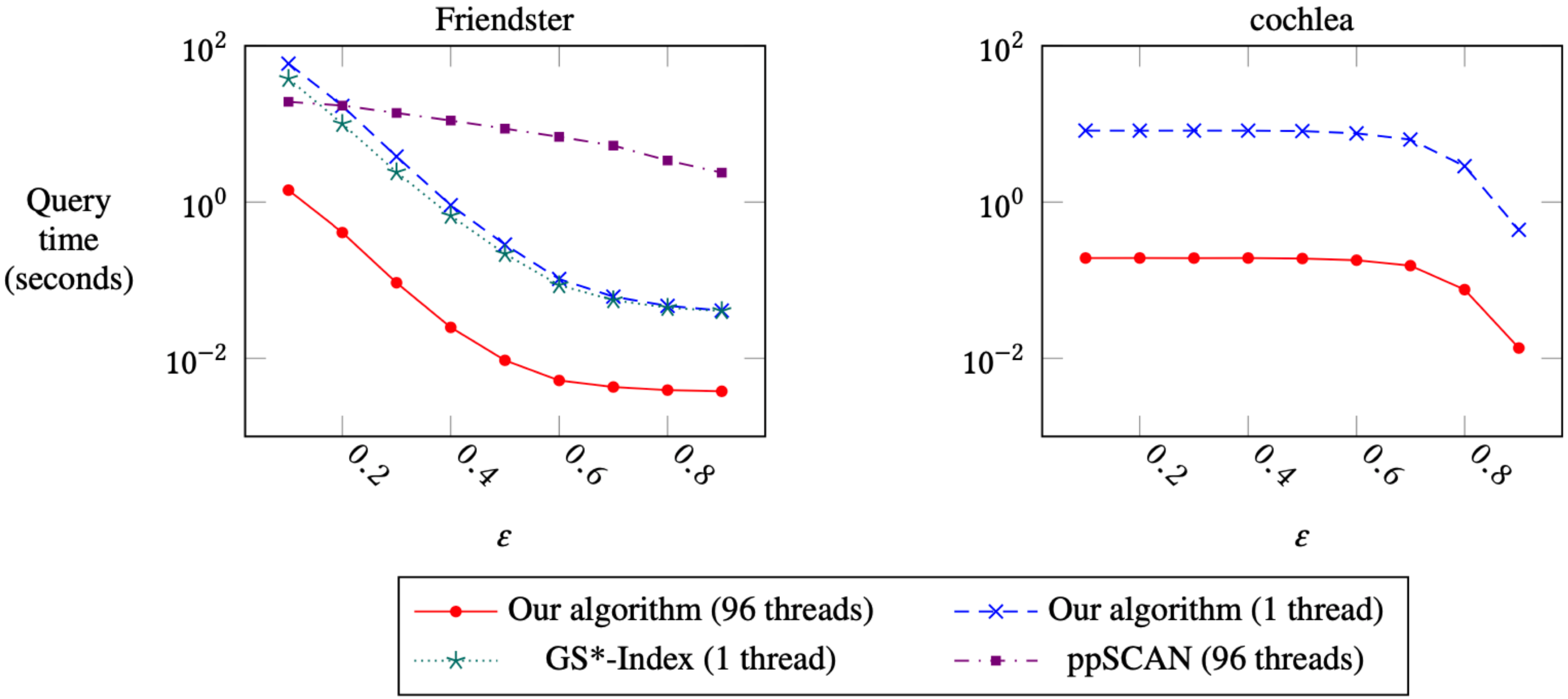


- Even sequentially, 1.4–2.2× speedup over GS-Index
- 23–70× self-relative parallel speedup

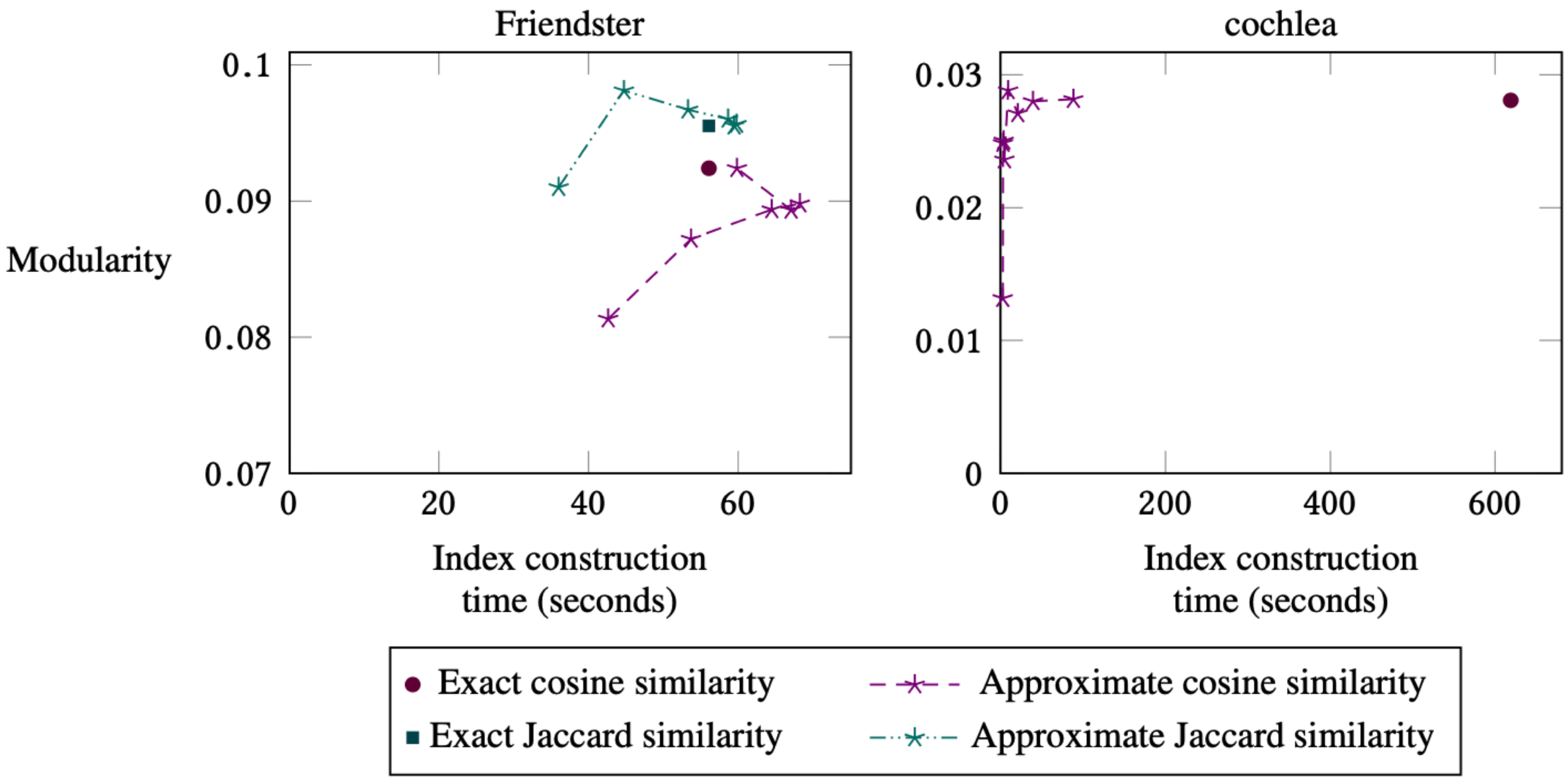
Query time: always faster than ppSCAN

- 1.26–12,070× speedup vs. ppSCAN
- 5–32× speedup vs. GS-Index

Fix μ=5 and vary ε



LSH gives faster index construction with similar cluster quality



- Modularity: popular and standard clustering metric based on how many edges are within clusters

Conclusion

- Theoretically-efficient and practical parallel algorithms for density-based spatial clustering (DBSCAN) and structural graph clustering (SCAN)
- Code publicly available
  - DBSCAN: https://sites.google.com/view/yiqiuwang/dbscan
  - SCAN: https://github.com/ParAlg/gbbs/tree/master/benchmarks/SCAN/IndexBased

Questions?

Future work

- Index-based DBSCAN
- Hierarchical versions of DBSCAN and SCAN
- Dynamic updates
- Framework for evaluating speed vs. accuracy of parallel clustering algorithms