Parallelism Background and Fast Parallel Spatial Clustering Algorithms

Laxman Dhulipala
University of Maryland, College Park
cs.umd.edu/~laxman

My Research

My Research

High-performance systems and algorithms that provide theoretical guarantees and are easy to use

My Research

High-performance systems and algorithms that provide theoretical guarantees and are easy to use

My Research

High-performance systems and algorithms that provide theoretical guarantees and are easy to use


Practice

Theory

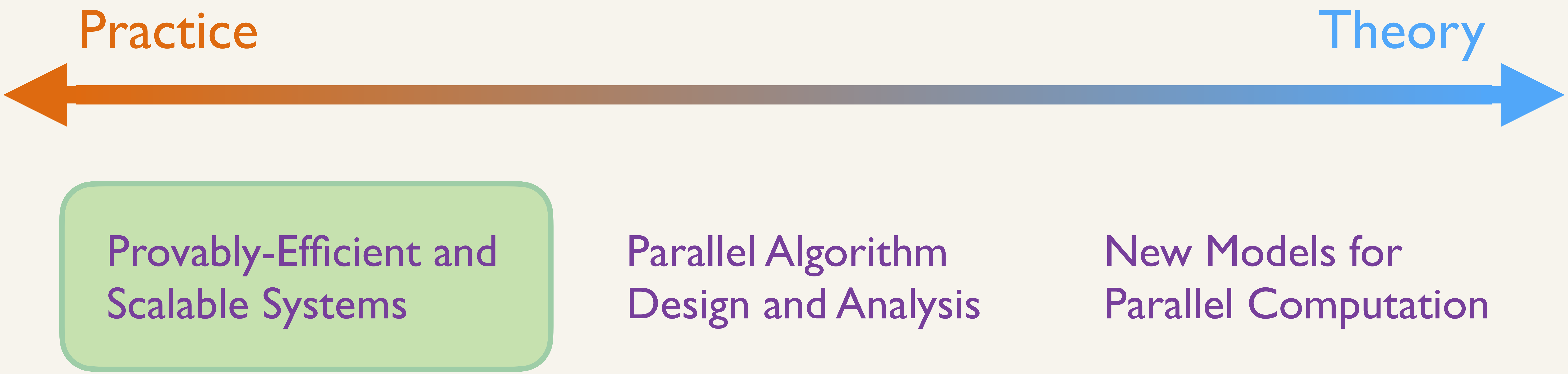
Provably-Efficient and Scalable Systems

Parallel Algorithm Design and Analysis

New Models for Parallel Computation

My Research

High-performance systems and algorithms that provide theoretical guarantees and are easy to use



Practice

Theory

Provably-Efficient and Scalable Systems

Parallel Algorithm Design and Analysis

New Models for Parallel Computation

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store
- Largest publicly available graph

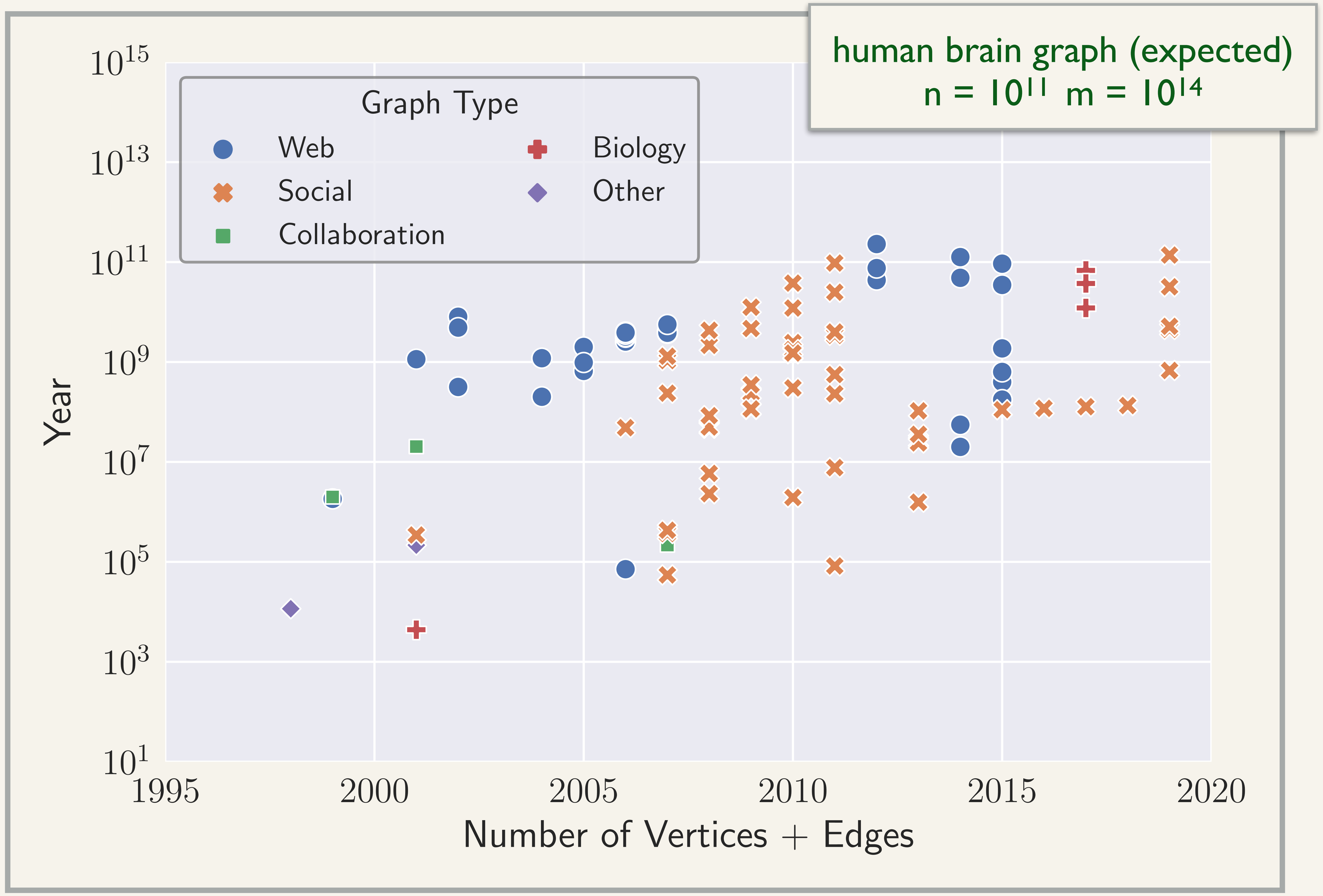
> *"…[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft."*

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store
- Largest publicly available graph

"…[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft."



human brain graph (expected)
$n = 10^{11}$ $m = 10^{14}$

Year of sourcing vs total number of vertices and edges for real-world graphs from the SNAP and LAW datasets

Sources: https://snap.stanford.edu/data/, http://law.di.unimi.it/datasets.php, http://webdatacommons.org/hyperlinkgraph/

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store
- Largest publicly available graph

"…[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft."

Parallelism is the key to processing very large graphs in a timely manner

Year of sourcing vs total number of vertices and edges for real-world graphs from the SNAP and LAW datasets

Sources: https://snap.stanford.edu/data/, http://law.di.unimi.it/datasets.php, http://webdatacommons.org/hyperlinkgraph/

Parallelism

Parallel machines are everywhere!

Parallelism

Parallel machines are everywhere!

Parallelism

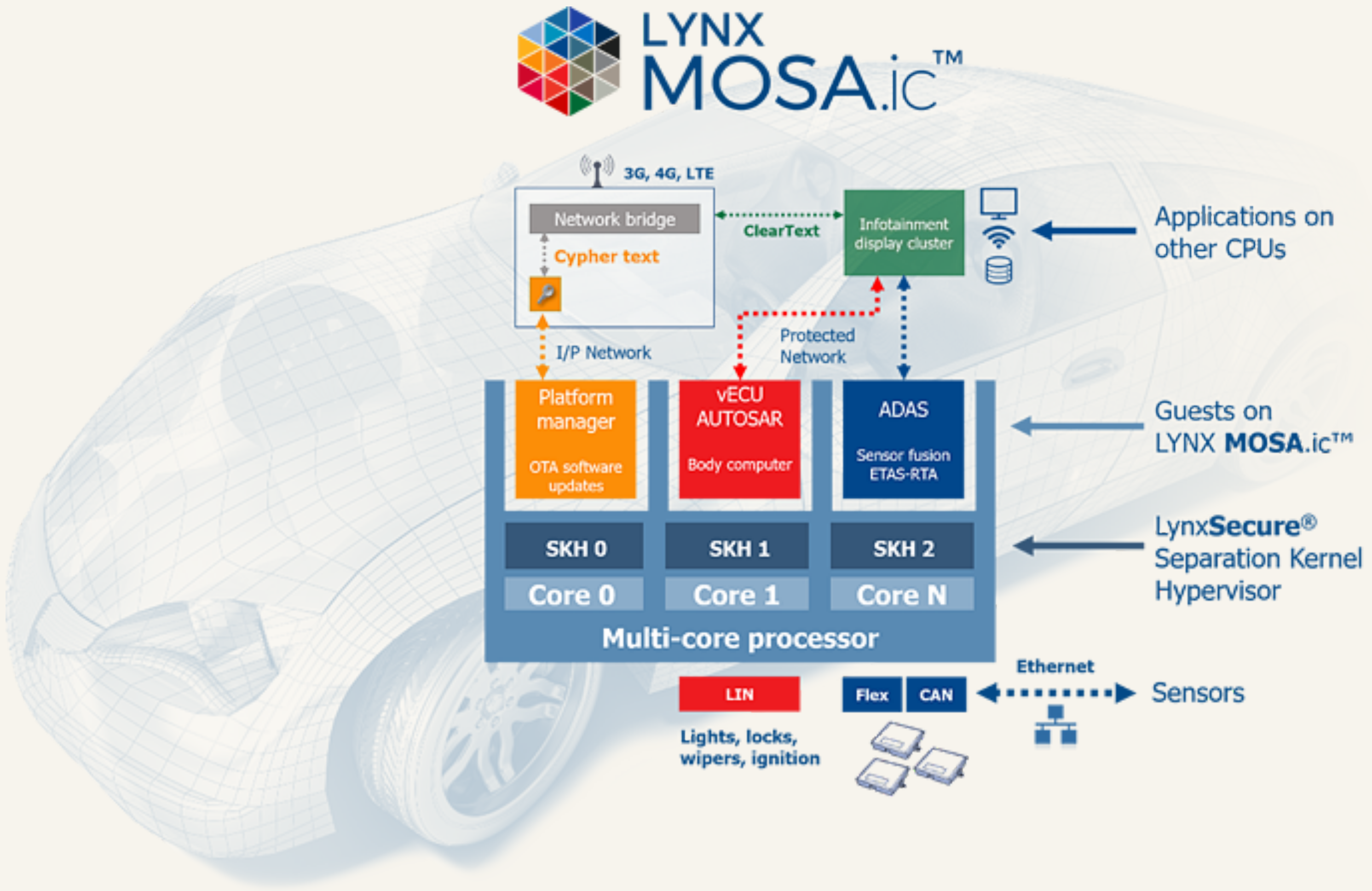
Parallel machines are everywhere!

Parallelism

Parallel machines are everywhere!

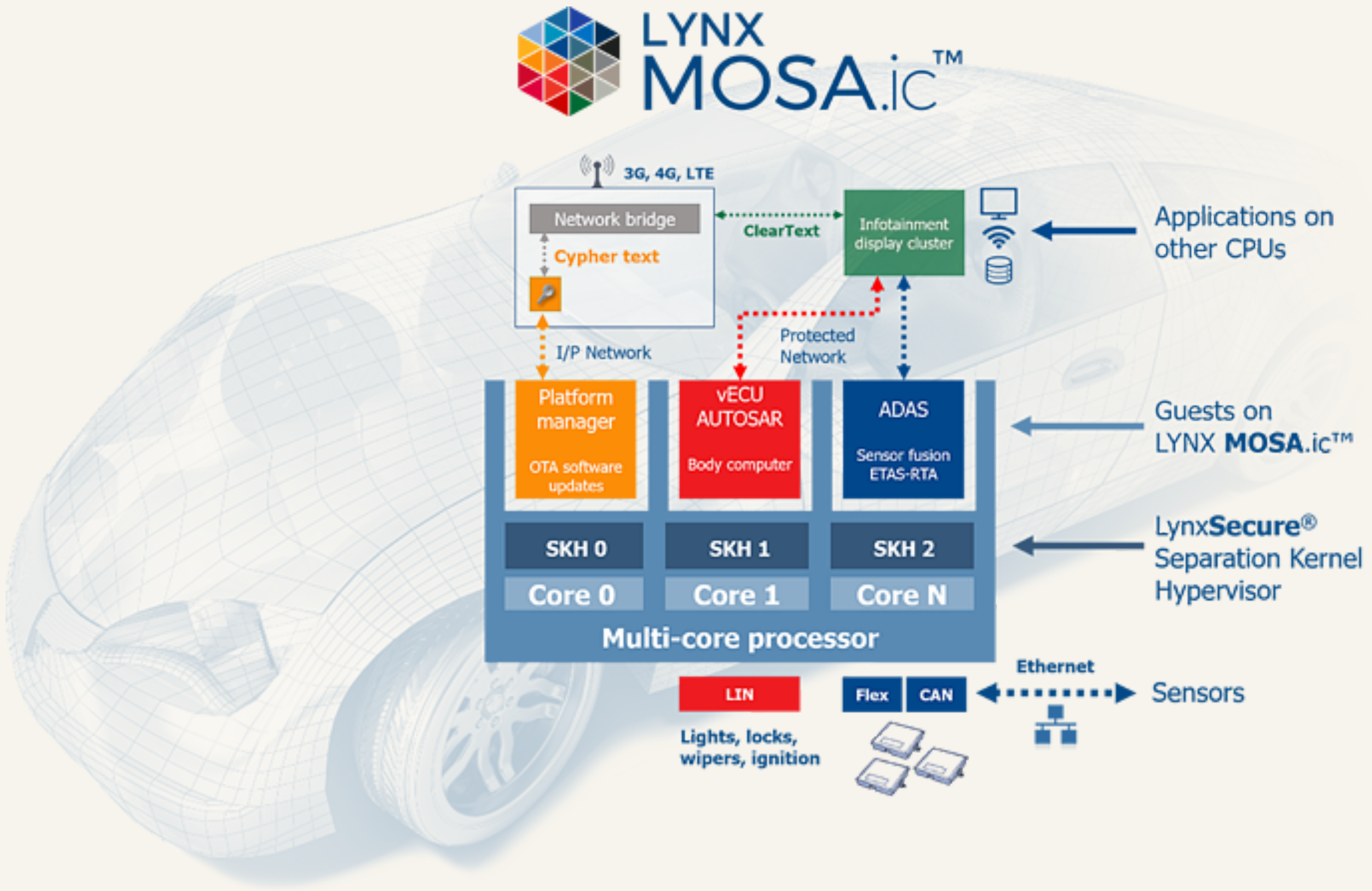
Parallelism

Parallel machines are everywhere!



Parallelism

Parallel machines are everywhere!



Main focus of my work is *shared-memory parallelism*

Shared-Memory Parallelism

Shared-Memory Parallelism

Shared-Memory Machines

- Cost for a 1TB memory machine with 72 processors is about $20,000.

Shared-Memory Parallelism

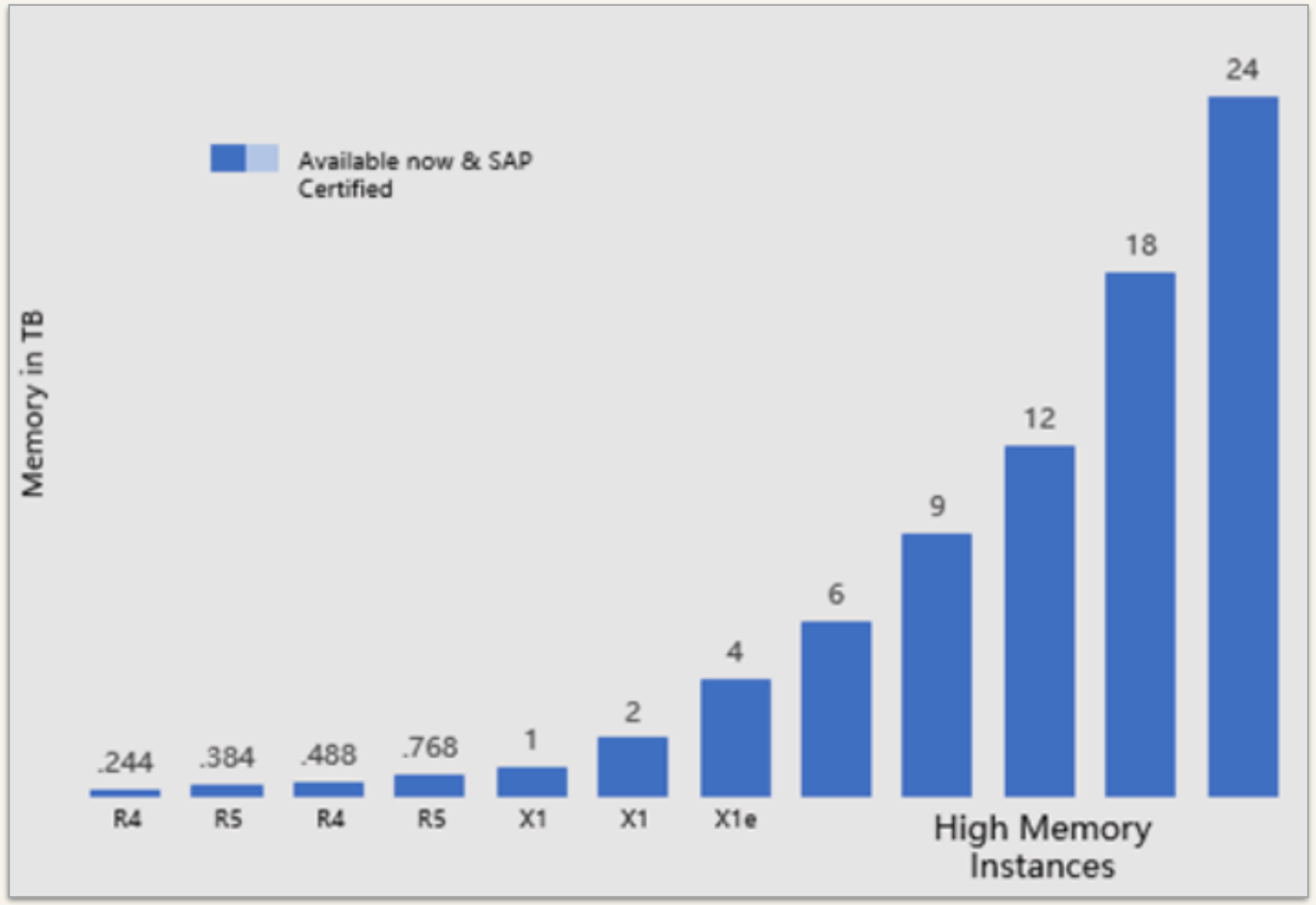
Shared-Memory Machines

- Cost for a 1TB memory machine with 72 processors is about $20,000.

Shared-Memory Parallelism

Shared-Memory Machines

- Cost for a 1TB memory machine with 72 processors is about $20,000.
- Can rent a similar machine (96 processors and 1.5TB memory) for $11/hour on Google Cloud



Shared-Memory Parallelism

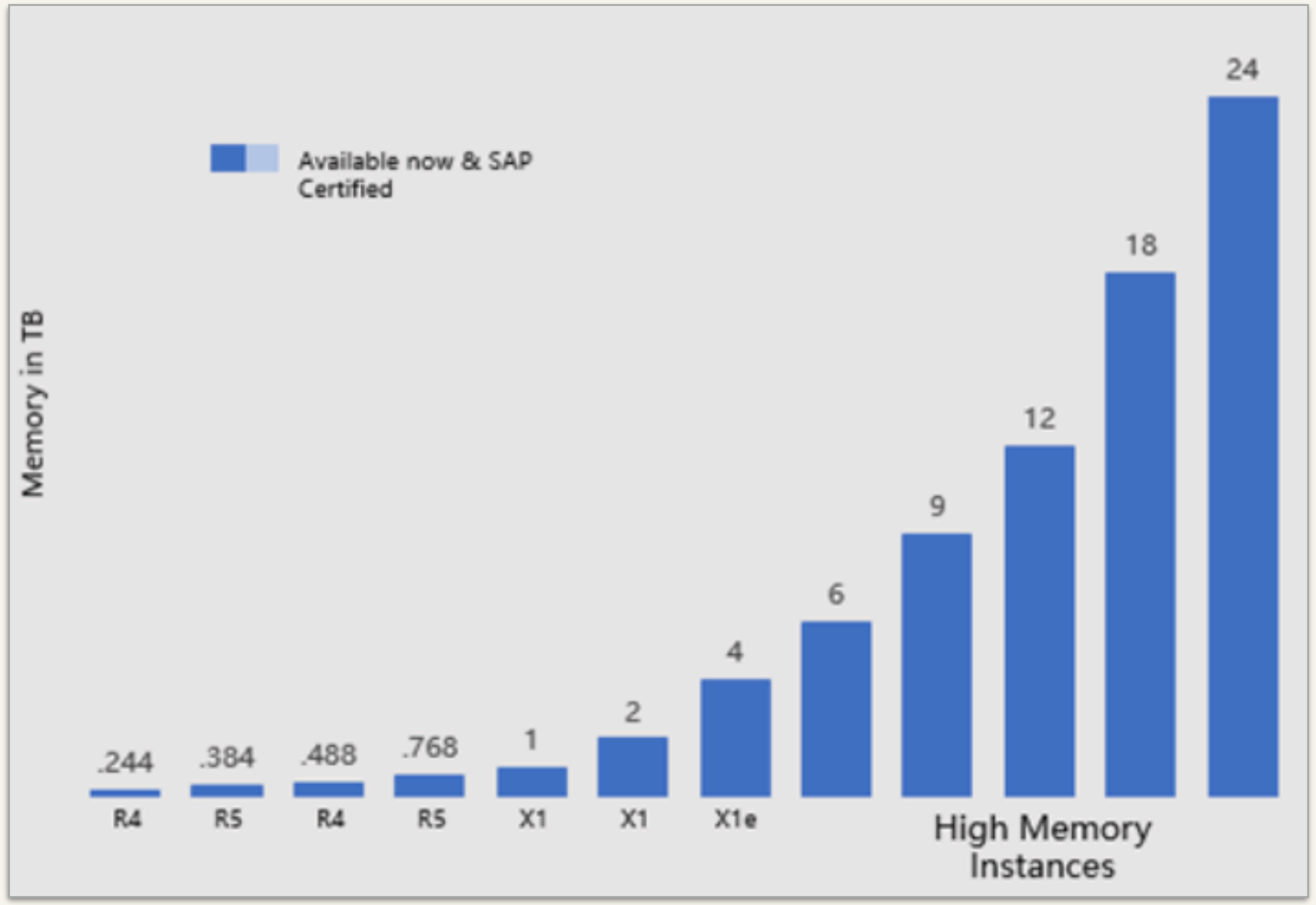
Shared-Memory Machines

- Cost for a 1TB memory machine with 72 processors is about $20,000.
- Can rent a similar machine (96 processors and 1.5TB memory) for $11/hour on Google Cloud

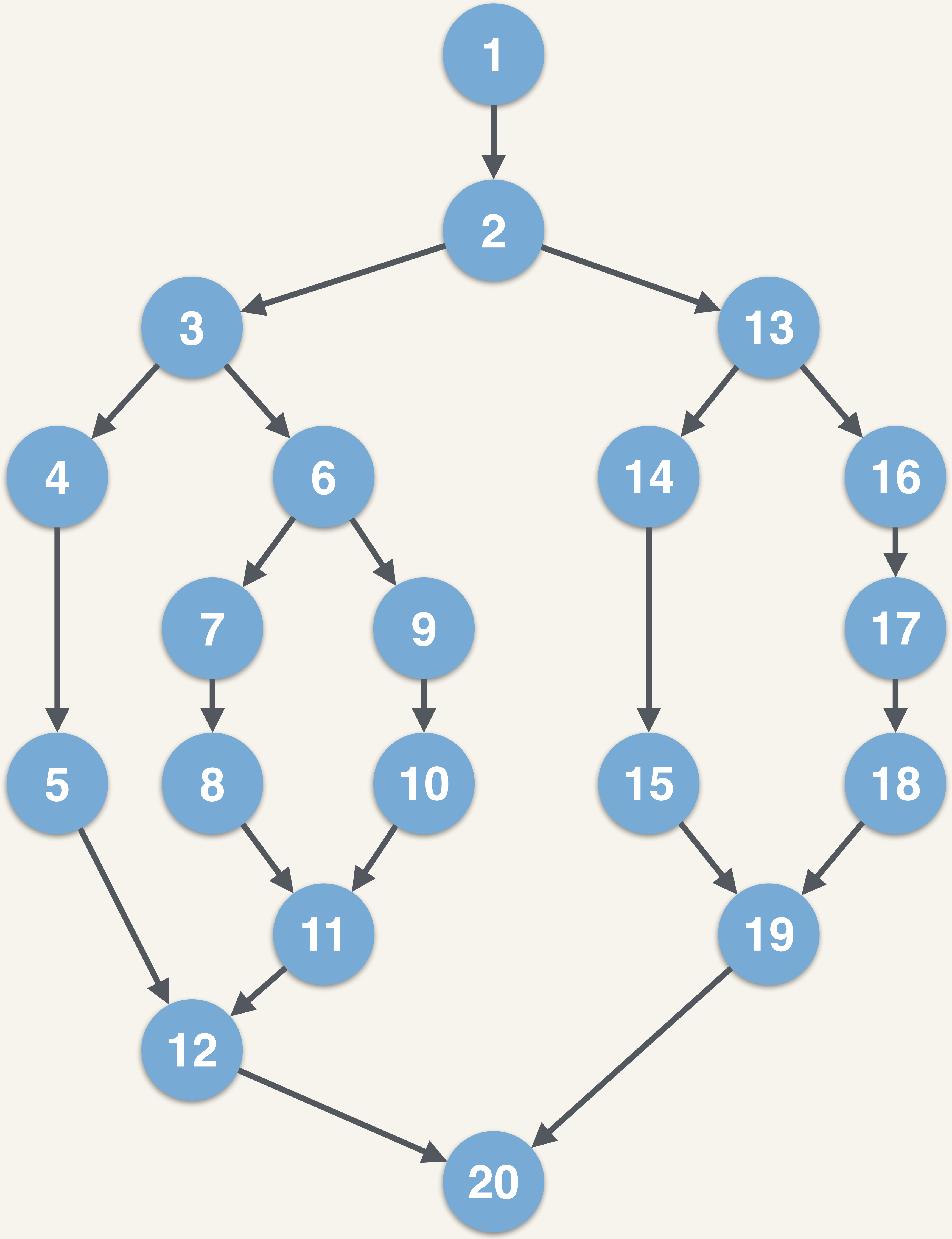
WebDataCommons Graph

- 3.5 billion vertices and 128 billion edges

A single shared-memory machine can already store the largest publicly available graph datasets, with plenty of room to spare

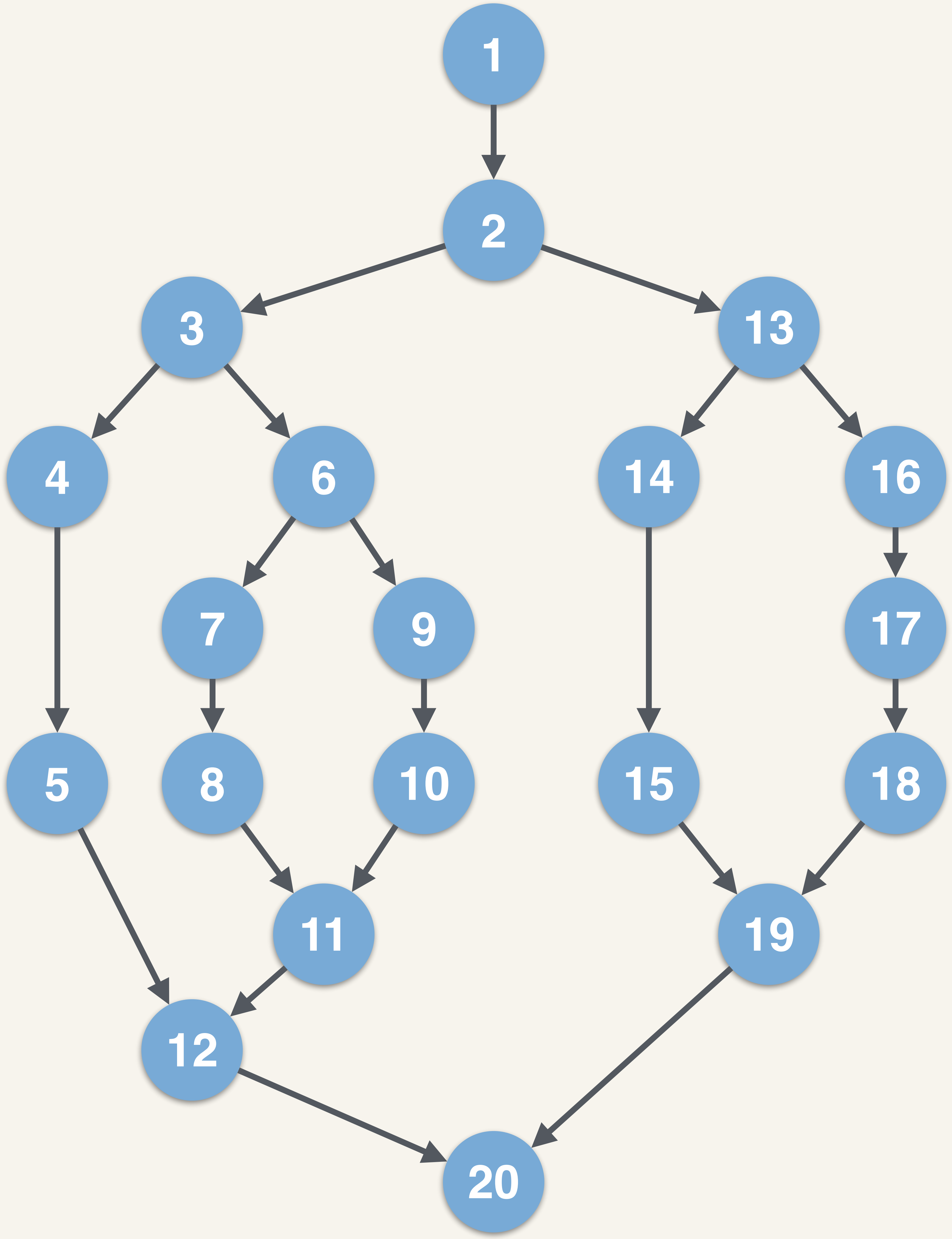


Work-Depth Model



Computation Graph

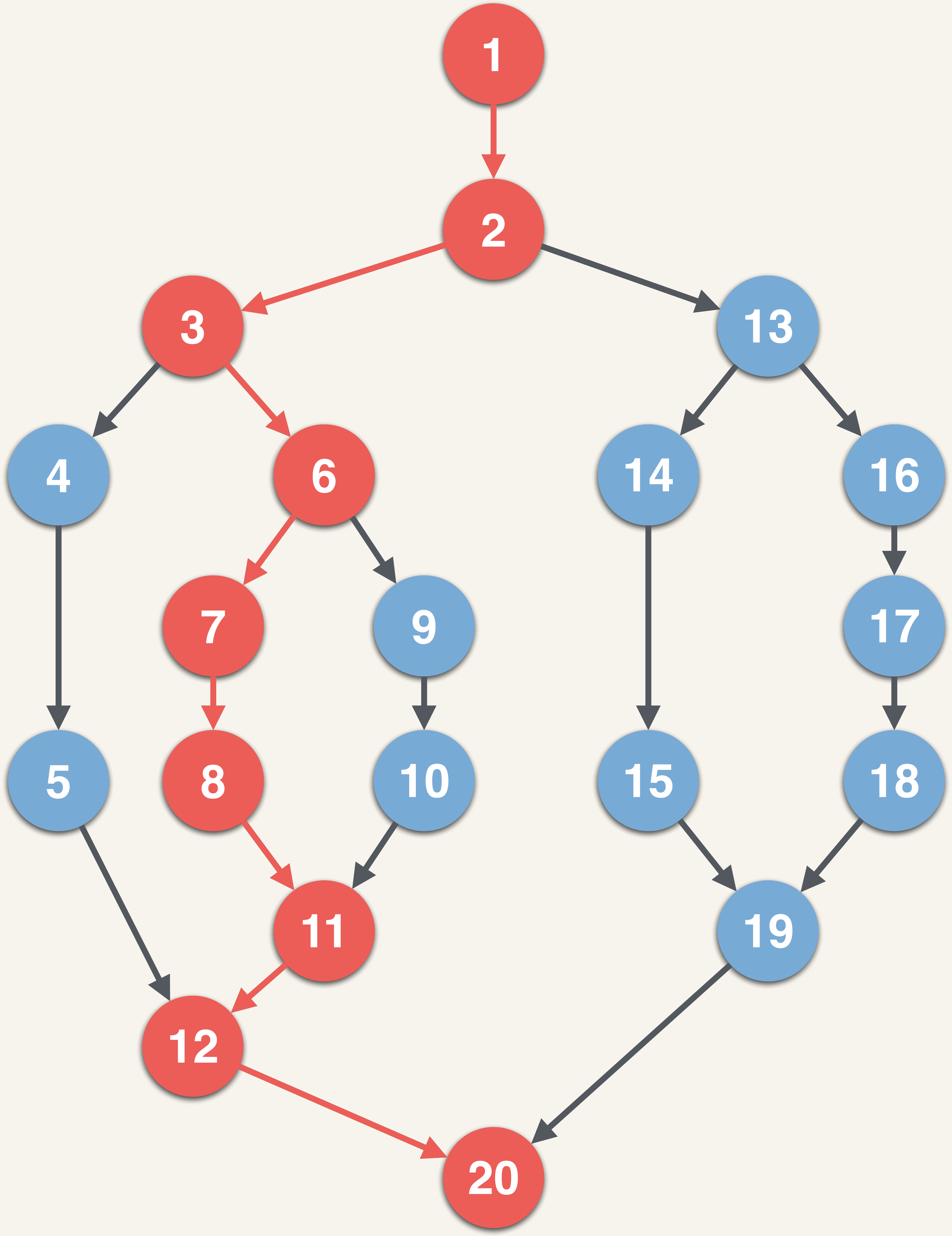
Work-Depth Model



Computation Graph

Work = total number of vertices in the computation graph

Work-Depth Model

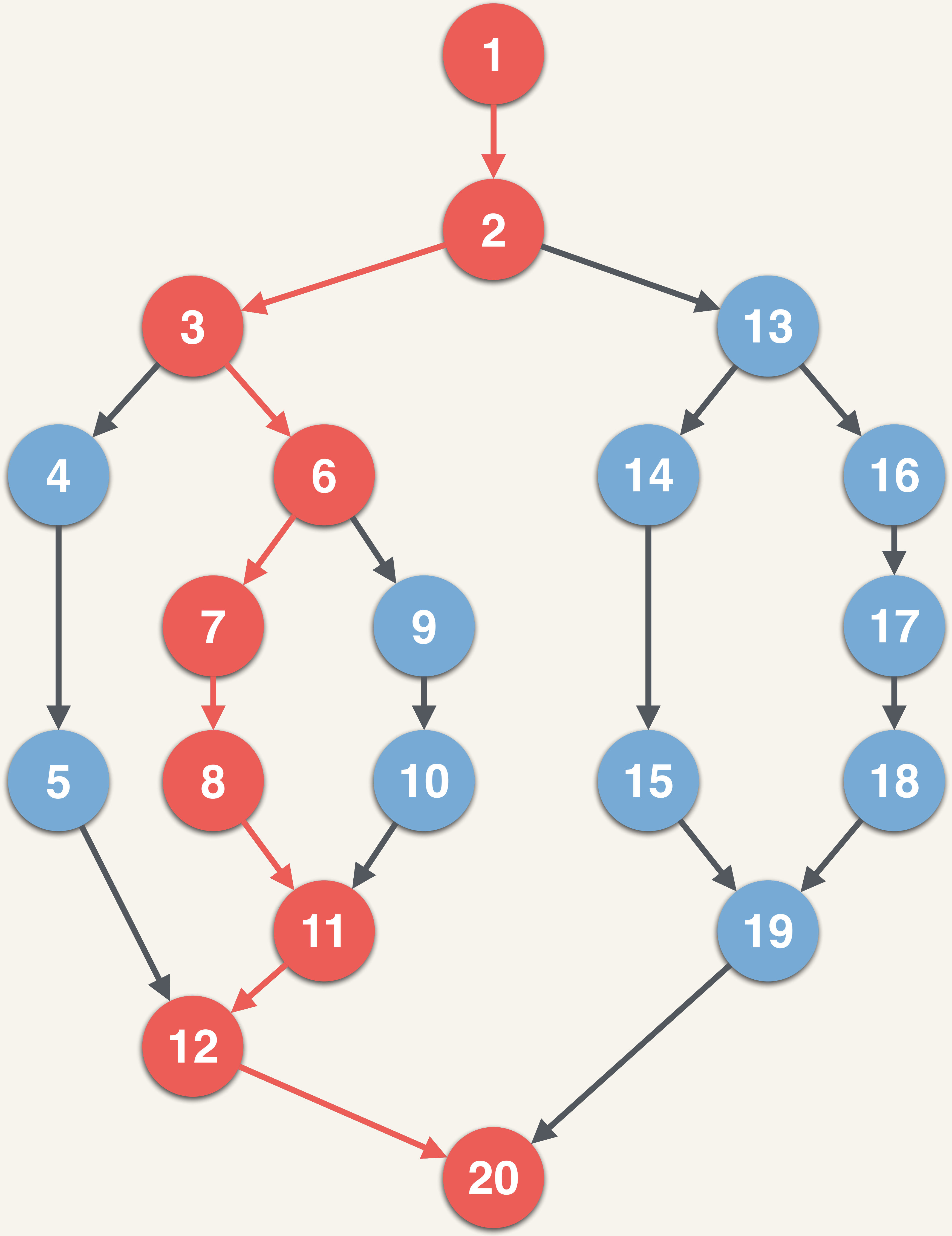


Computation Graph

Work = total number of vertices in the computation graph

Depth = longest directed path in the graph (dependence length)

Work-Depth Model



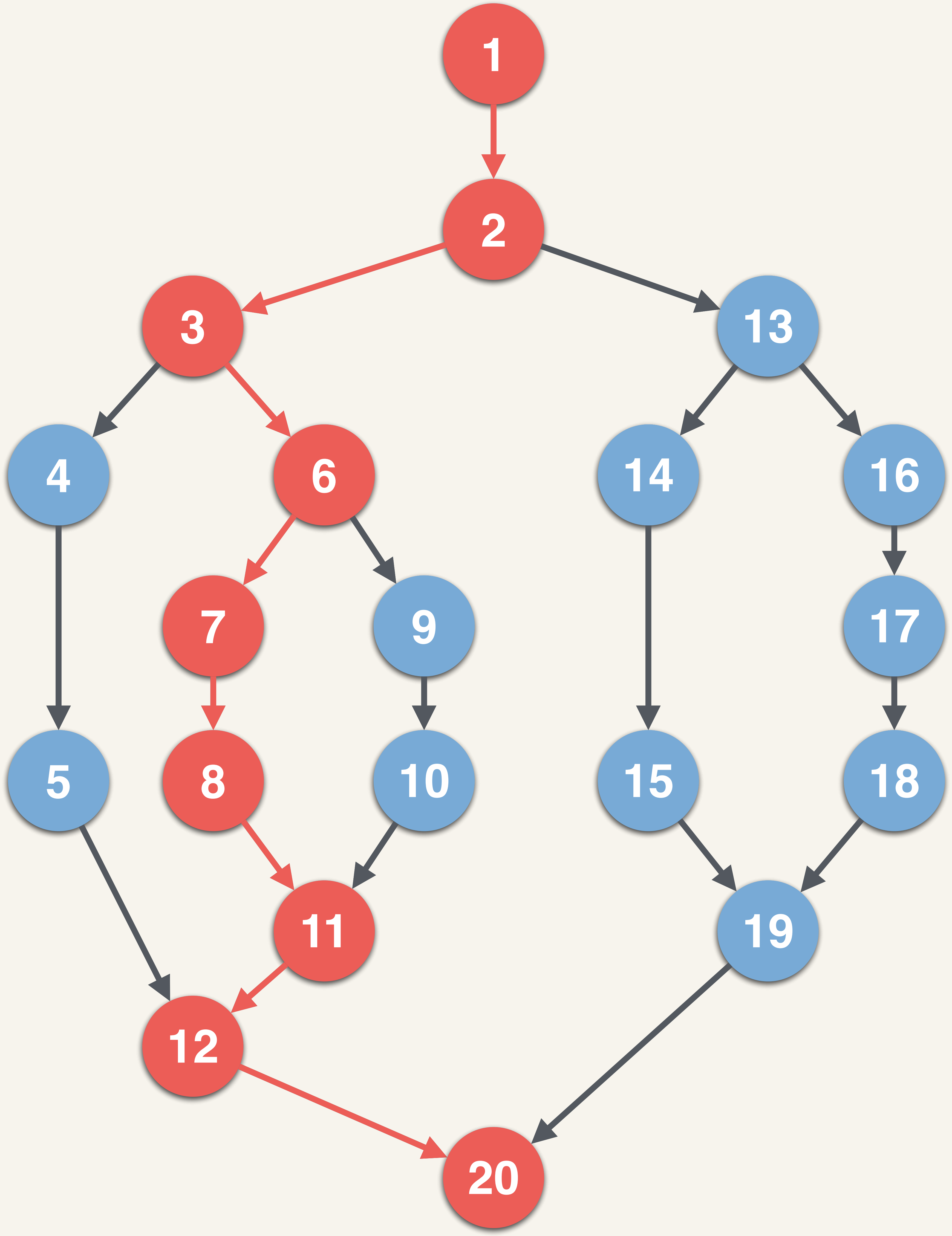
Computation Graph

Work = total number of vertices in the computation graph

Depth = longest directed path in the graph (dependence length)

Running Time = Work/#Processors + O(Depth)

Work-Depth Model



Computation Graph

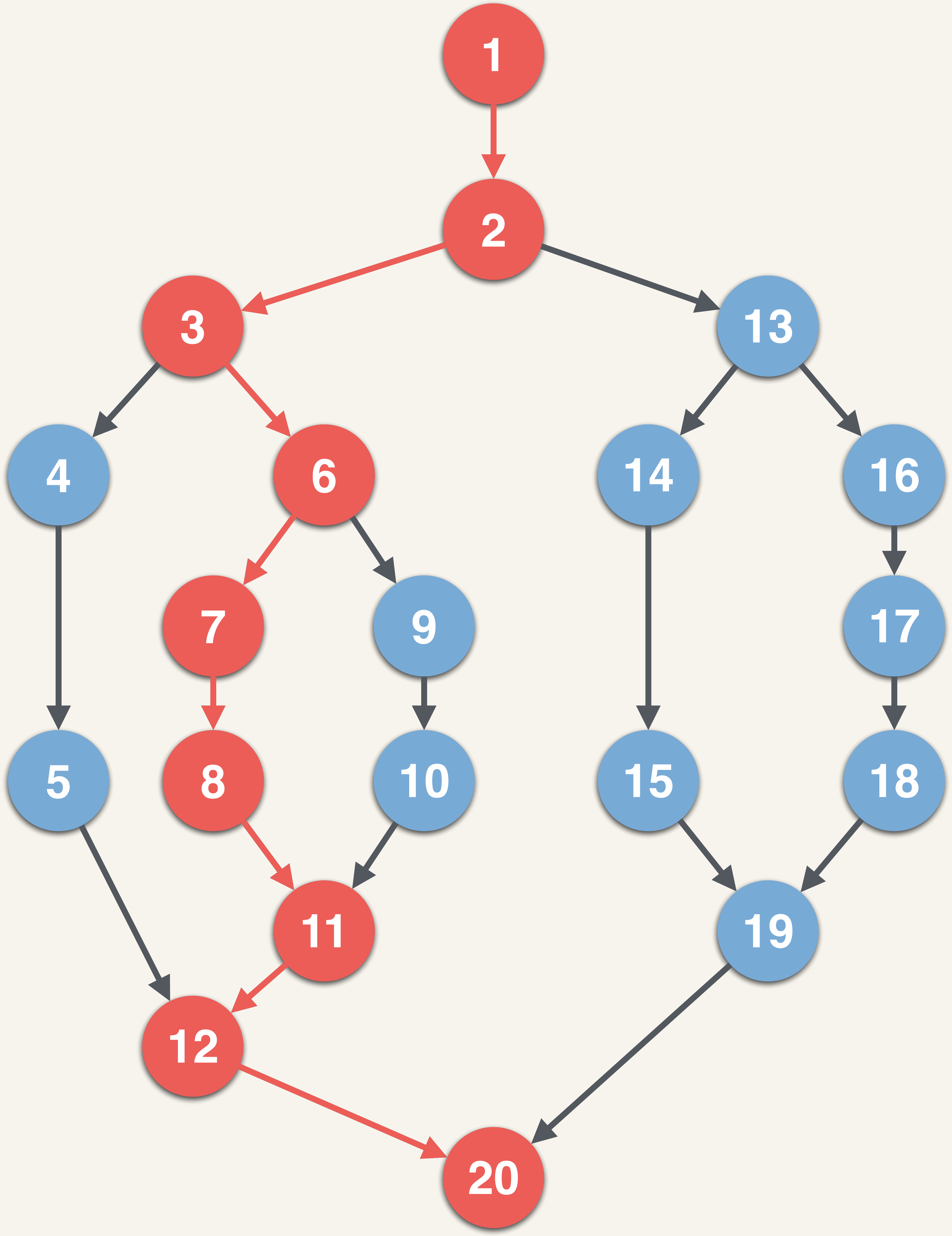
Work = total number of vertices in the computation graph

Depth = longest directed path in the graph (dependence length)

Running Time = Work/#Processors + O(Depth)

A *work-efficient* parallel algorithm has work that asymptotically matches that of the best sequential algorithm for the problem

Work-Depth Model



Computation Graph

Work = total number of vertices in the computation graph

Depth = longest directed path in the graph (dependence length)

Running Time = Work/#Processors + O(Depth)

A *work-efficient* parallel algorithm has work that asymptotically matches that of the best sequential algorithm for the problem

Goal: work-efficient and low (polylogarithmic) depth algorithms

Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth

Why do we care about theoretical bounds?

Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth

Why do we care about theoretical bounds?

Input-agnostic design

- Design codes without worrying too much about your datasets

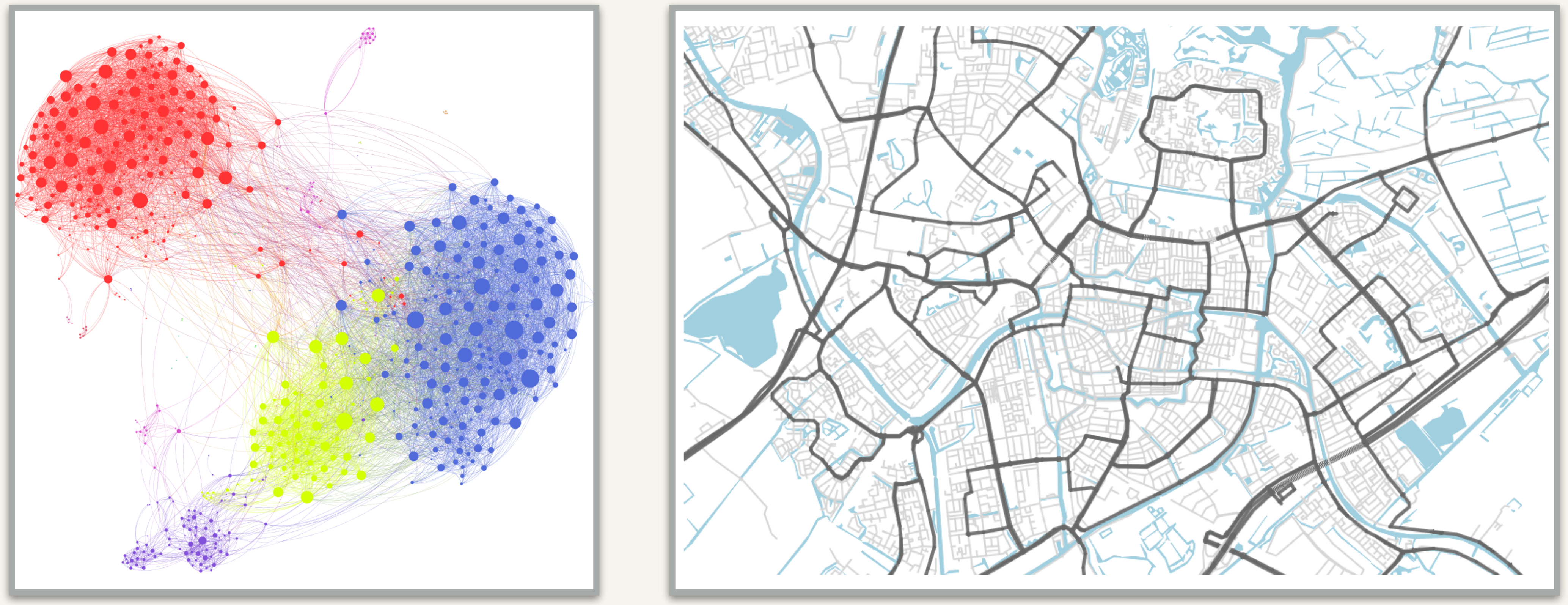
Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth

Why do we care about theoretical bounds?

Input-agnostic design

- Design codes without worrying too much about your datasets



Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth

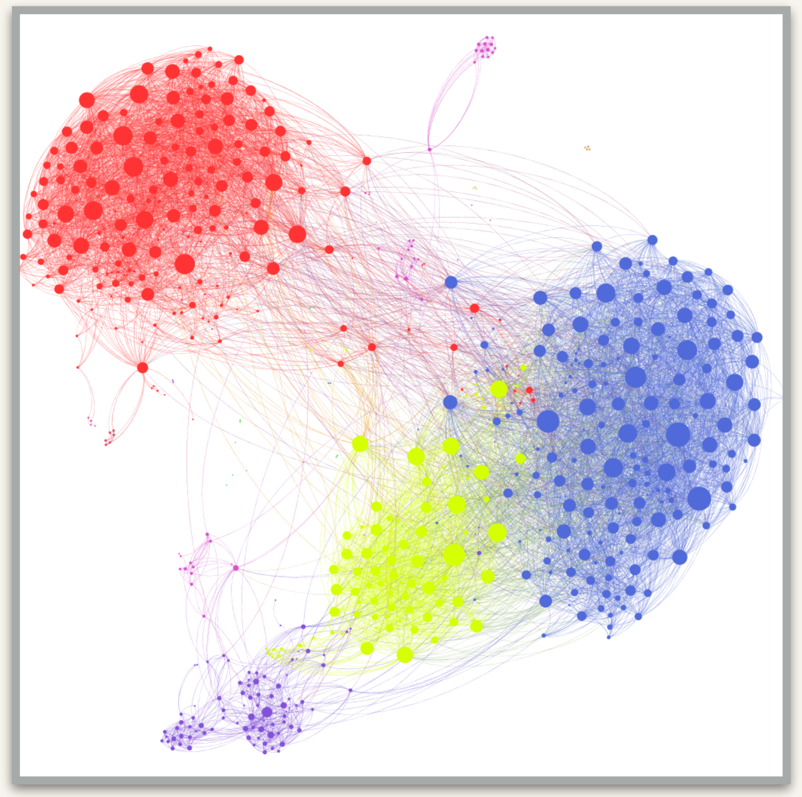
Why do we care about theoretical bounds?

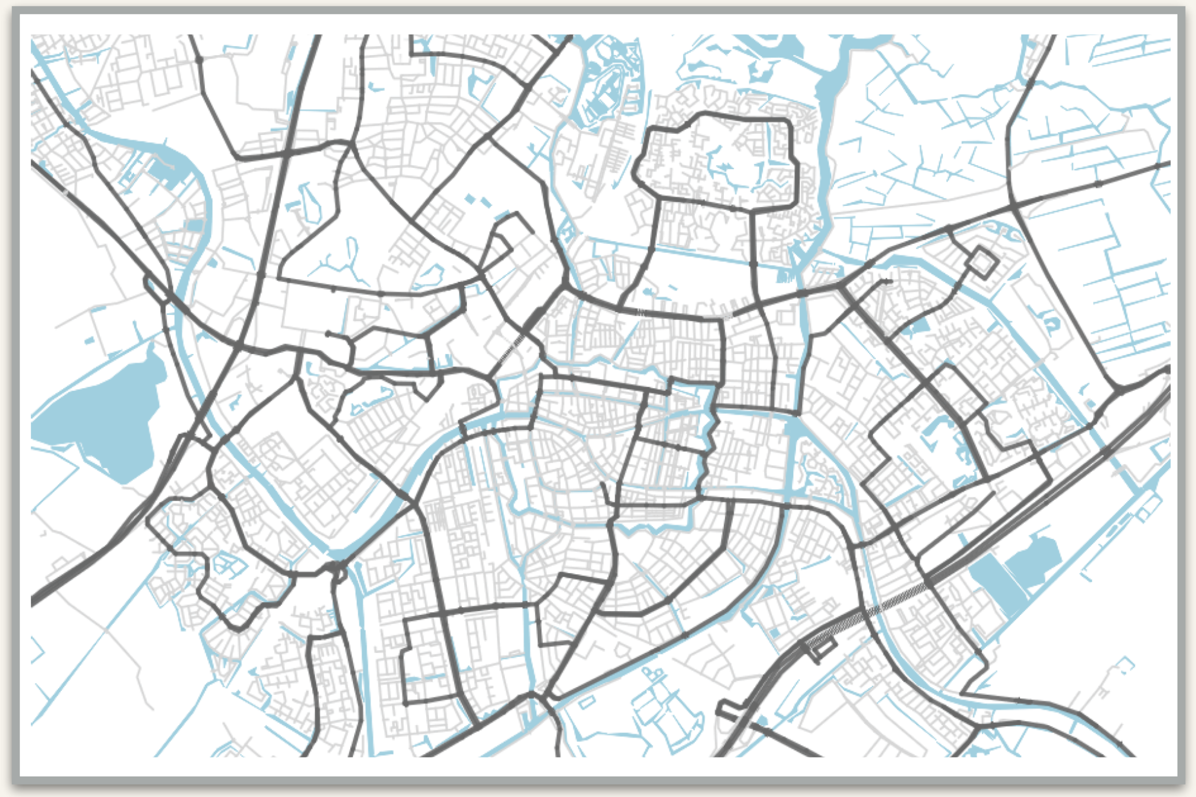
Input-agnostic design

- Design codes without worrying too much about your datasets

Robustness to bad inputs

- Perform well even on new classes of graphs
- Understand how they will scale on larger graphs

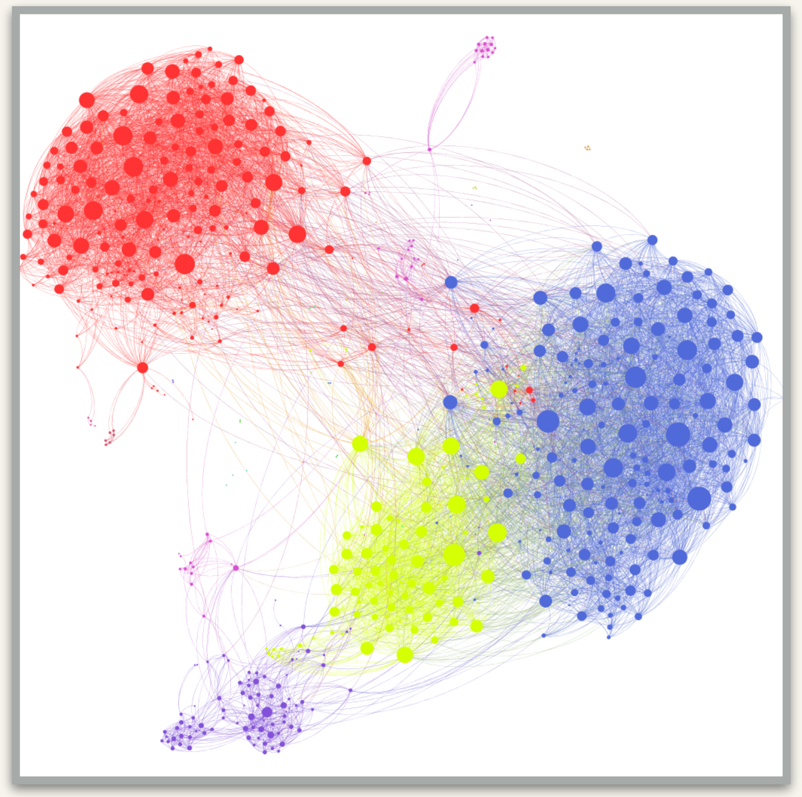


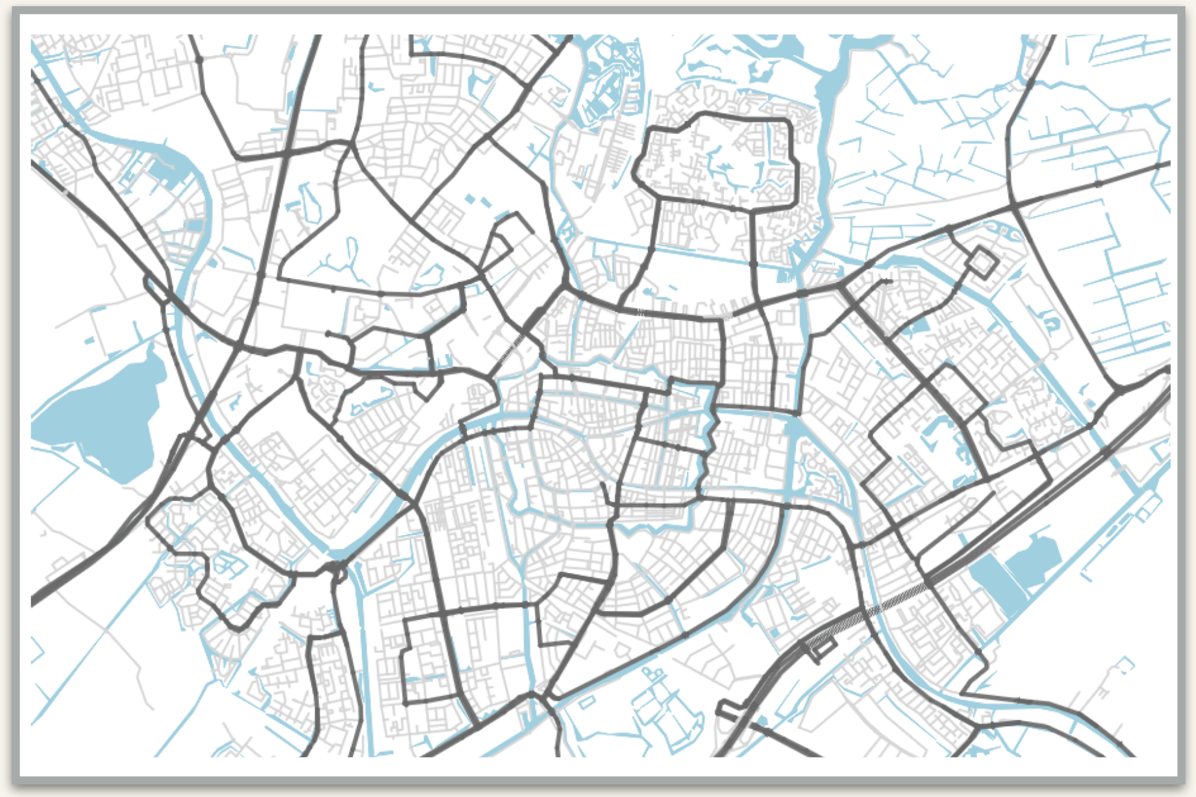


Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth

Why do we care about theoretical bounds?





Input-agnostic design

- Design codes without worrying too much about your datasets

Robustness to bad inputs

- Perform well even on new classes of graphs
- Understand how they will scale on larger graphs

Work-efficiency matters in practice

- Work-efficient algorithms can be much faster than work-inefficient algorithms

Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth

Why do we care about theoretical bounds?

Input-agnostic design

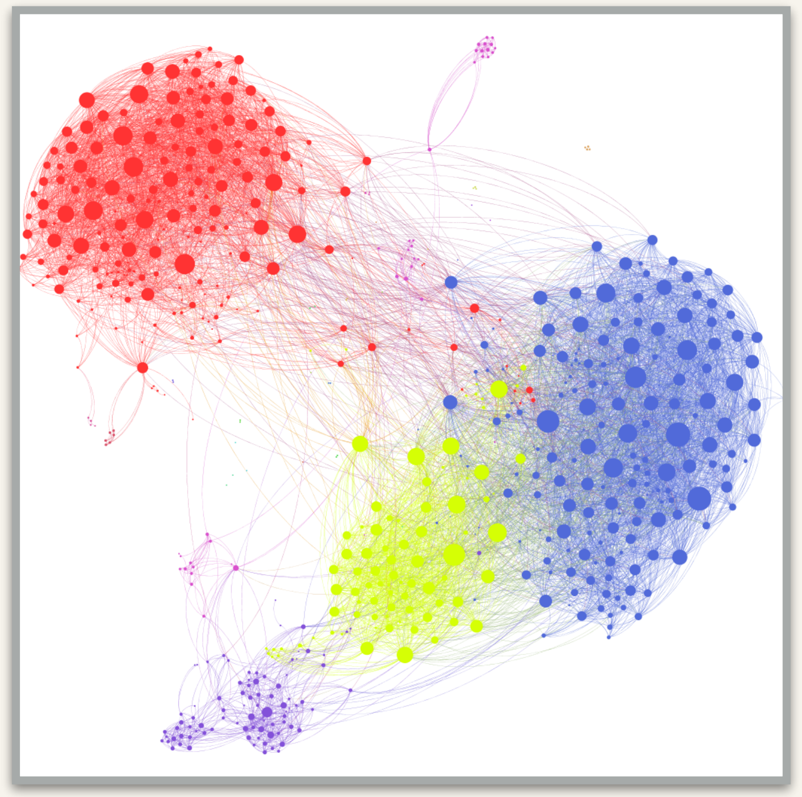
- Design codes without worrying too much about your datasets

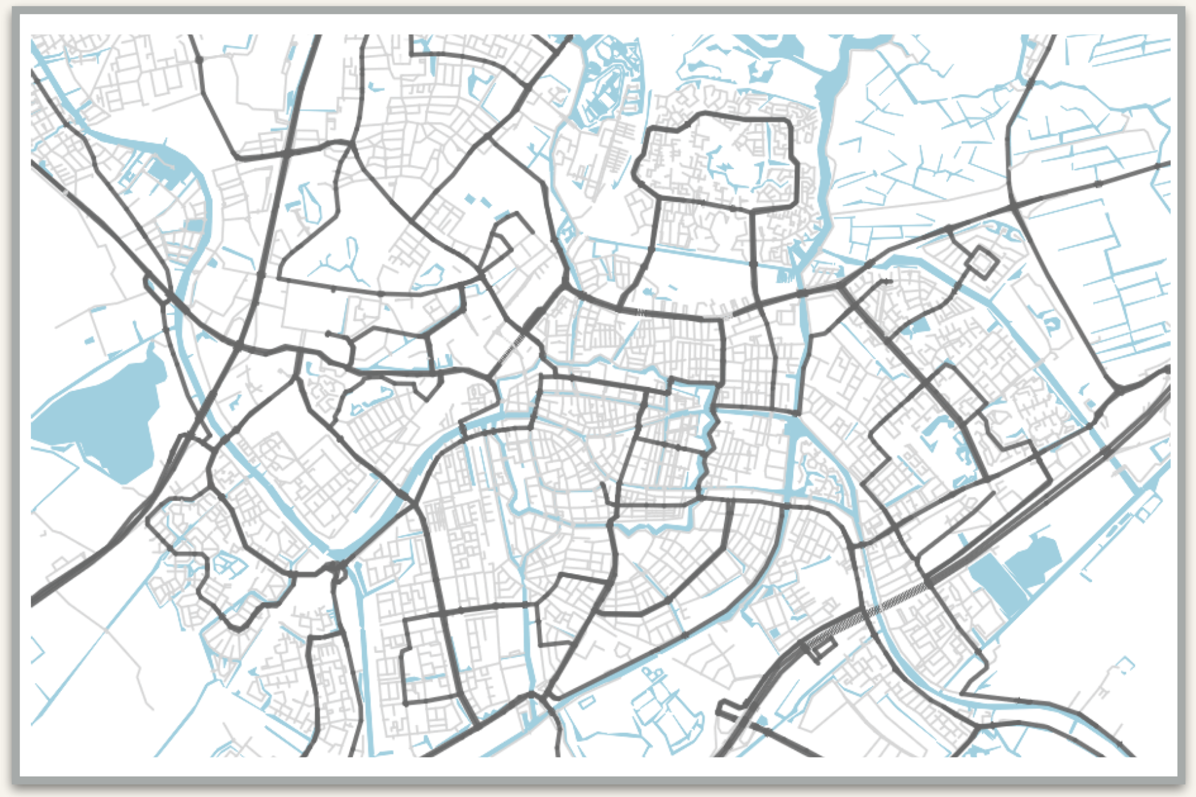
Robustness to bad inputs

- Perform well even on new classes of graphs
- Understand how they will scale on larger graphs

Work-efficiency matters in practice

- Work-efficient algorithms can be much faster than work-inefficient algorithms





Up to 9x faster using a work-efficient k-core algorithm (described in this talk)

Graph Systems: examples

Pregel
PowerGraph
PowerLyra
Parallel BGL
GraphLab
Green-Marl
GraphMat
Ringo
SNAP
GraphIt
Ligra
Julienne
GBBS
STAPL

GraphX (Spark)
ASPIRE
GoFFish
Presto
GraphChi
Blogel
GraM
Giraph
PAGE
MOCgraph
GrapH
LightGraph
Gluon
Graphine

Sage
Graphite
GraFBoost
X-Stream
TurboGraph
TurboGraph++
Ligra+
MMap
PathGraph
GridGraph
NXgraph
Chaos
FlashGraph
Graphene

GraphMat
EmptyHeaded
Congra
CongraPlus
Laika
SociaLite
Graphphi
TuFast
Maiter
LCC-Graph
TopoX
Gluon-Async
GraphA
L-PowerGraph

………

Graph Systems: examples

Pregel
PowerGraph
PowerLyra
ParallelBGL
GraphLab
Green-Marl
GraphMat
Ringo
SNAP
GraphIt
Ligra
Julienne
GBBS
STAPL

GraphX (Spark)
ASPIRE
GoFFish
Presto
GraphChi
Blogel
GraM
Giraph
PAGE
MOCgraph
GraphH
LightGraph
Gluon
Graphine

Sage
Graphite
GraFBoost
X-Stream
TurboGraph
TurboGraph++
Ligra+
MMap
PathGraph
GridGraph
NXgraph
Chaos
FlashGraph
Graphene

GraphMat
EmptyHeaded
Congra
CongraPlus
Laika
SociaLite
Graphphi
TuFast
Maiter
LCC-Graph
TopoX
Gluon-Async
GraphA
L-PowerGraph

....

Unfortunately existing graph systems typically study a very small set of simple problems, such as BFS.

Can we solve a broad set of static graph problems on very large graphs?

Theoretically-Efficient Parallel Graph Algorithms can be Fast and Scalable [D, Blelloch, Shun, SPAA'18 Best Paper]

- Introduce the Graph-Based Benchmark Suite (GBBS) for graph problems with over 20 important problems
- GBBS algorithms achieve state-of-the-art results on the largest publicly available graphs

Connectivity Problems

Low-Diameter Decomposition
Connectivity
Spanning Forest
Biconnectivity
Minimum Spanning Forest
Strongly Connected Components

Subgraph Problems

k-Core Decomposition
k-Truss Decomposition
Apx. Densest Subgraph
Triangle Counting
Higher-Clique Counting

Covering Problems

Maximal Ind. Set
Maximal Matching
Apx. Set Cover
Graph Coloring

Shortest Path Problems

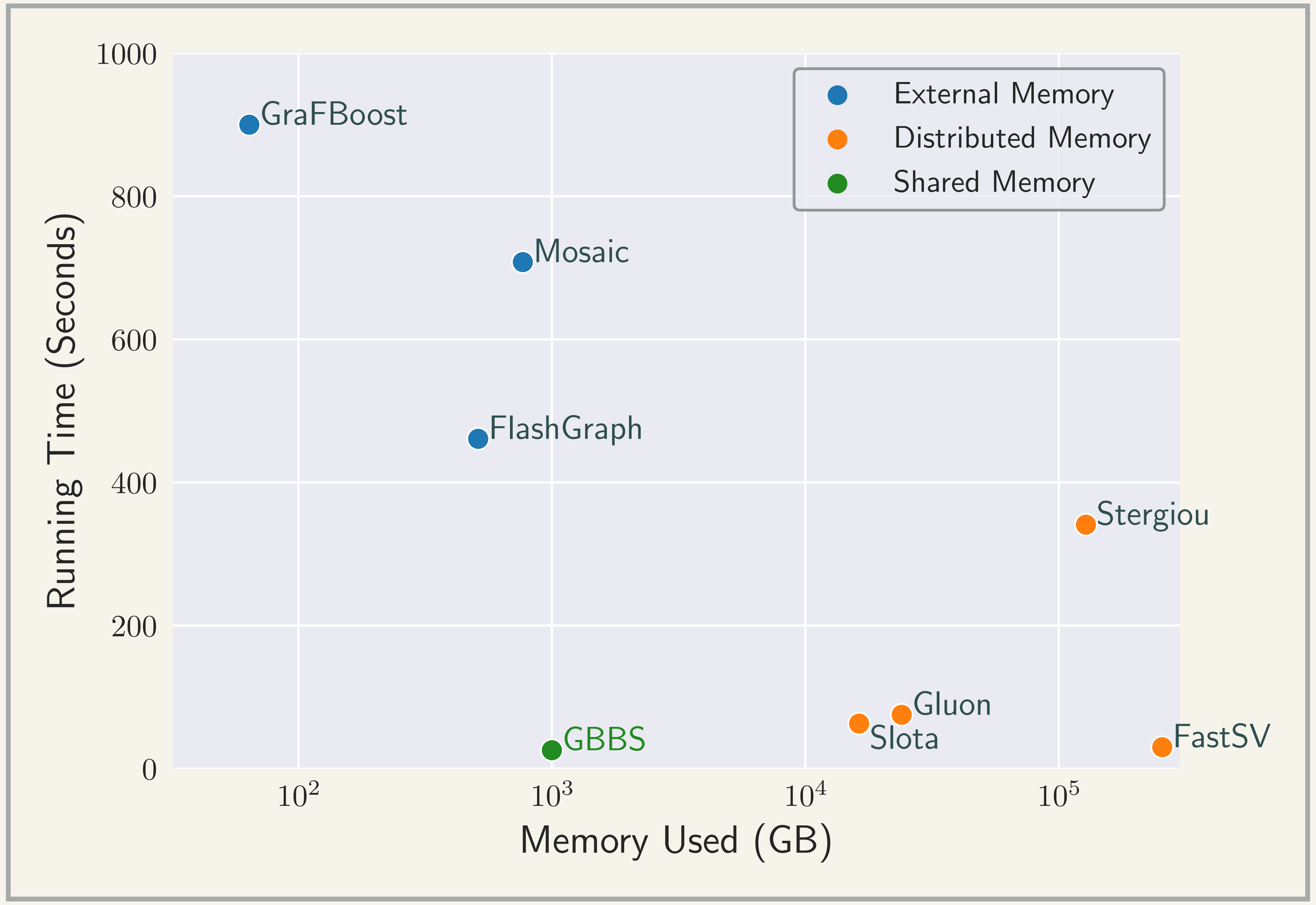
Breadth-First Search
Betweenness Centrality
Bellman-Ford
General Weight SSSP
Integral Weight SSSP
SS Widest Path
k-Spanner

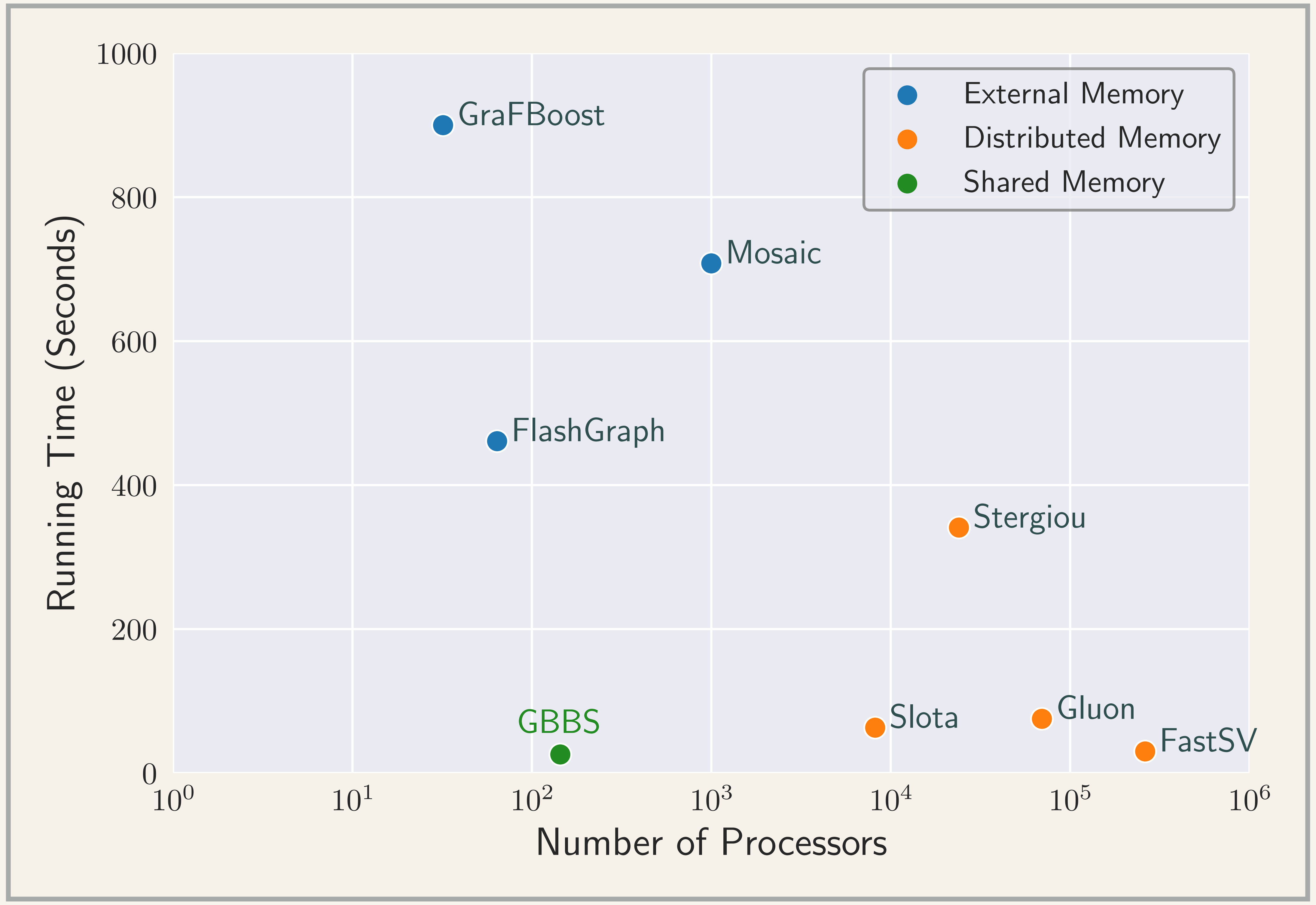
Eigenvector Problems

PageRank
Personalized PageRank
Personalized SimRank

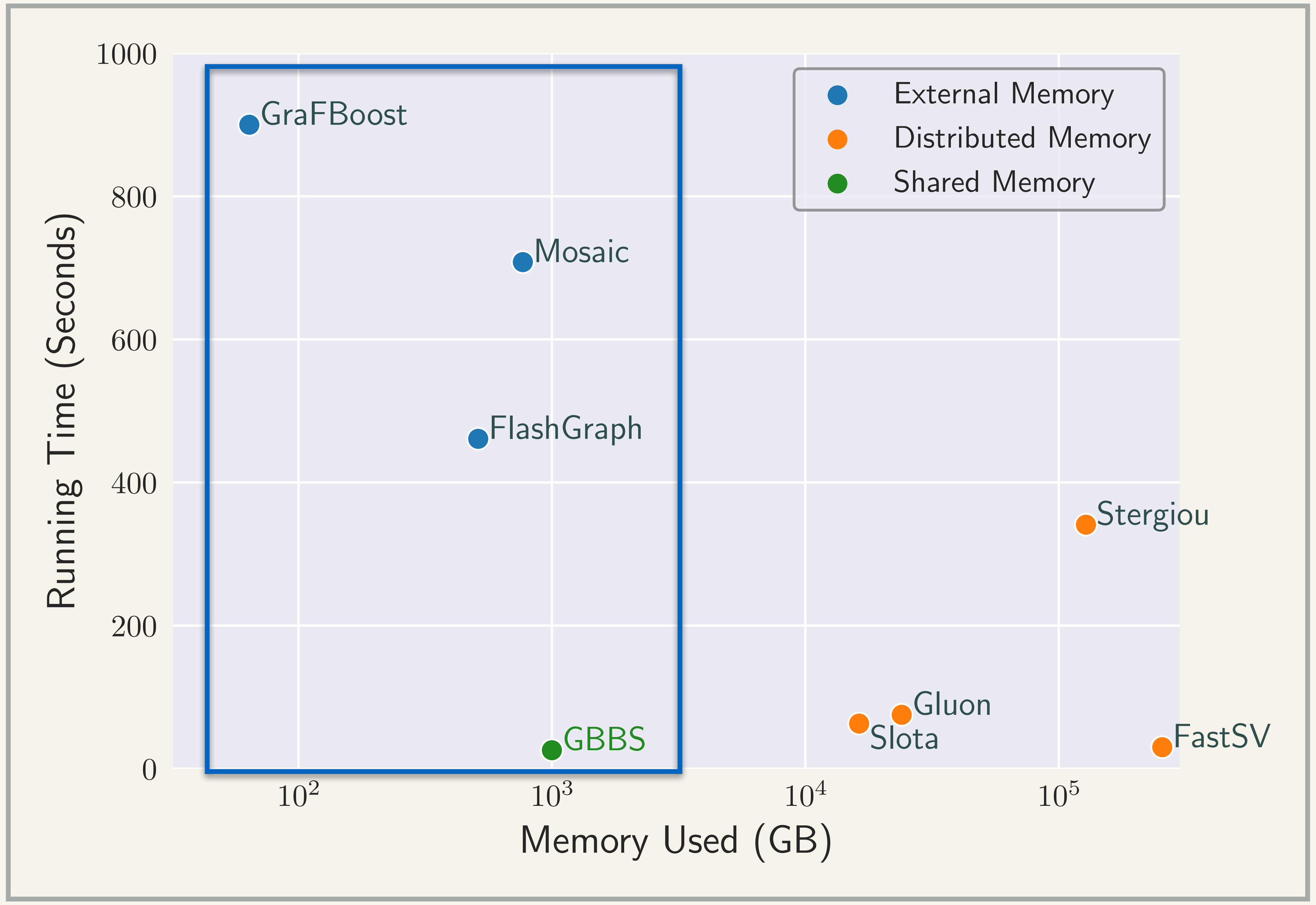
github.com/paralg/gbbs

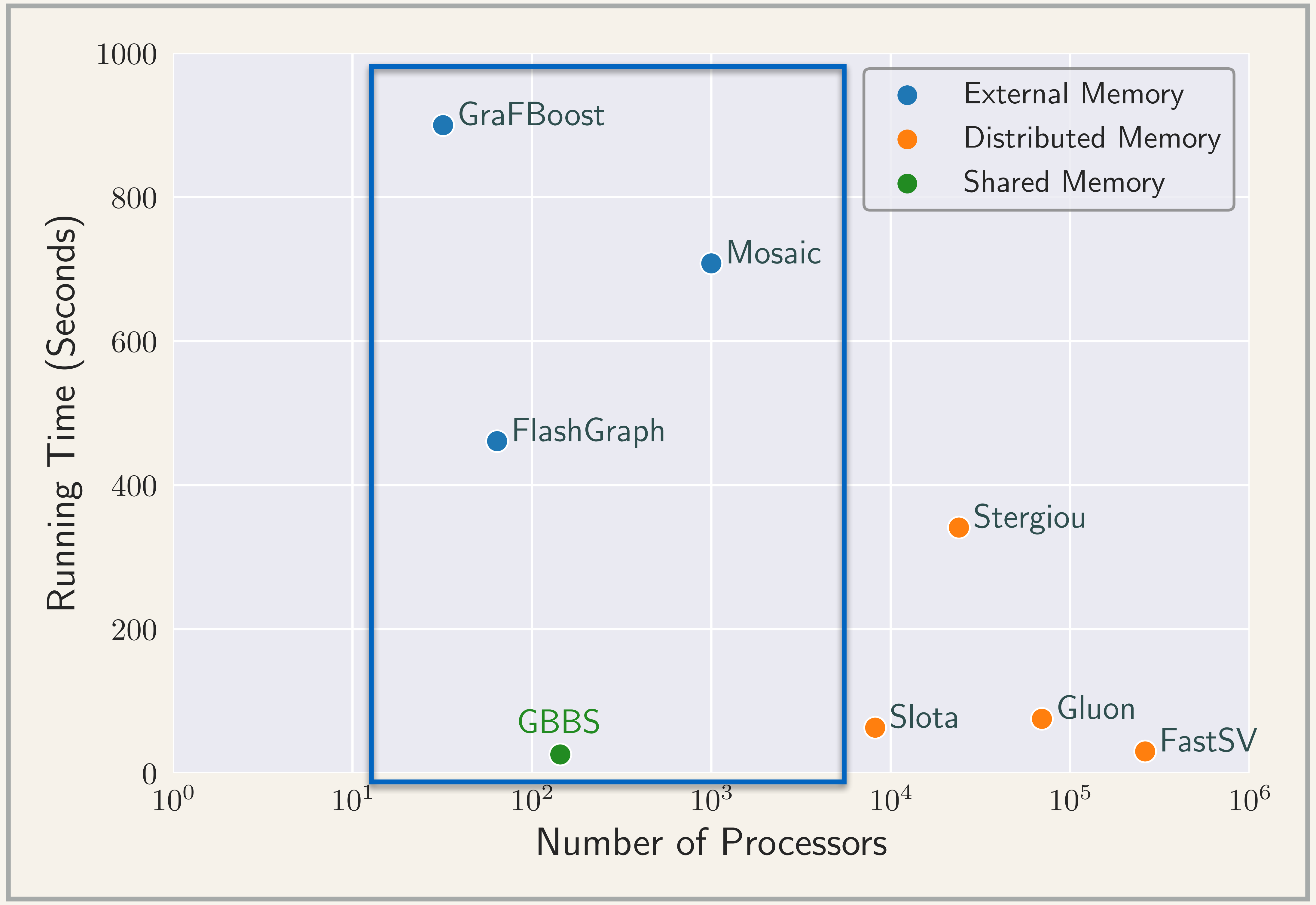
Benchmarking Connectivity on WebDataCommons Graph





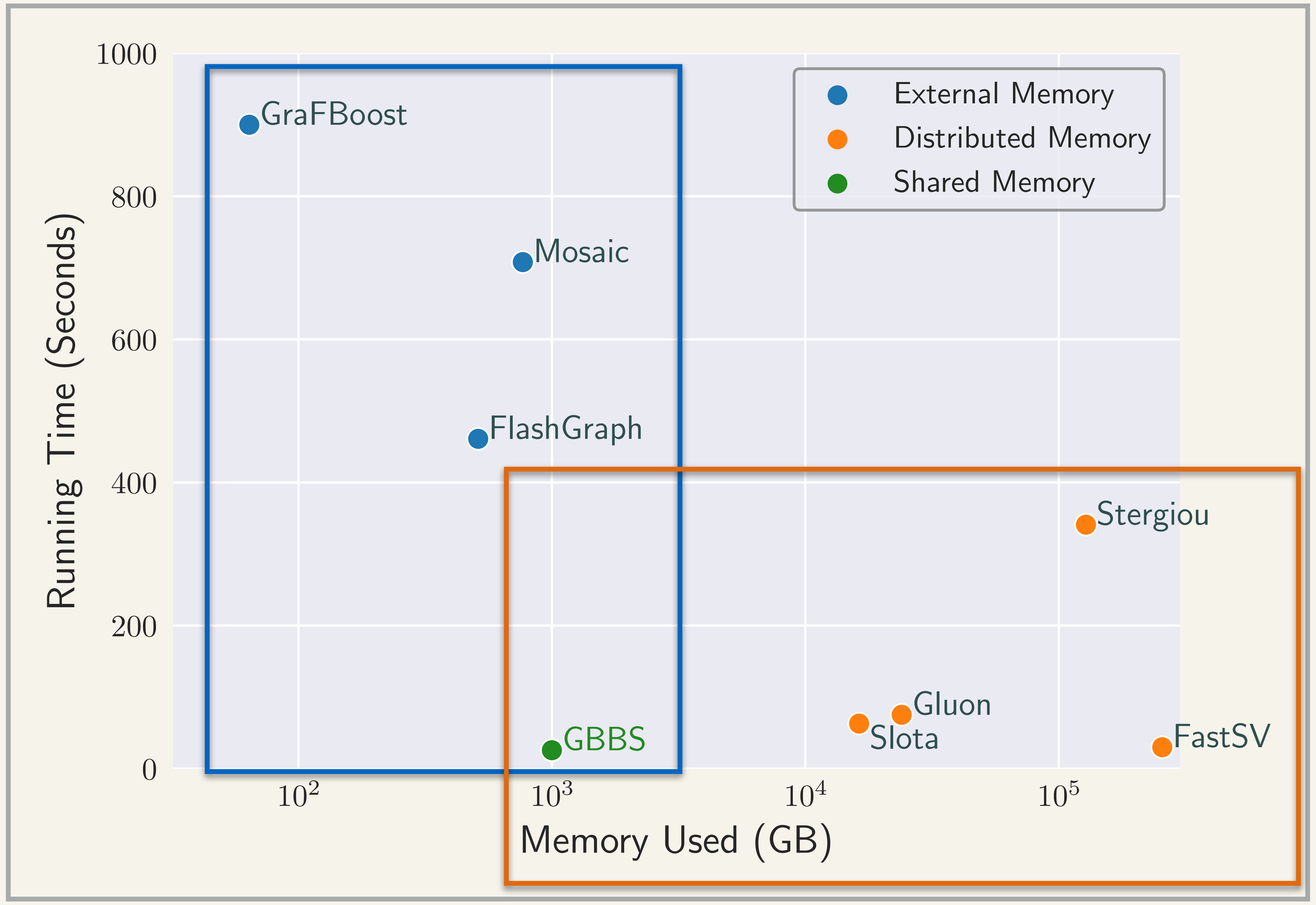
Benchmarking Connectivity on WebDataCommons Graph

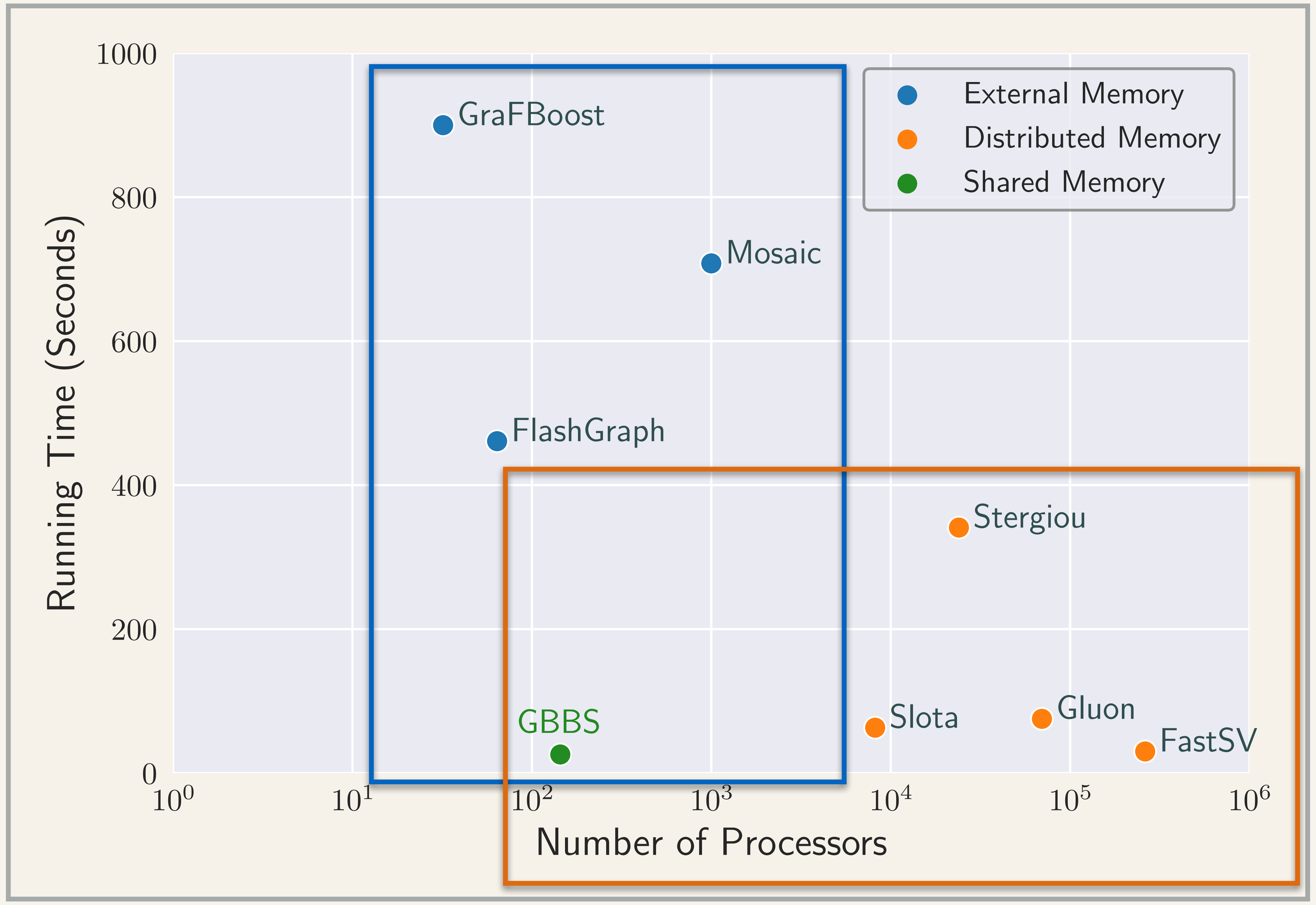




Outperform external memory results by orders of magnitude using comparable hardware.

Benchmarking Connectivity on WebDataCommons Graph

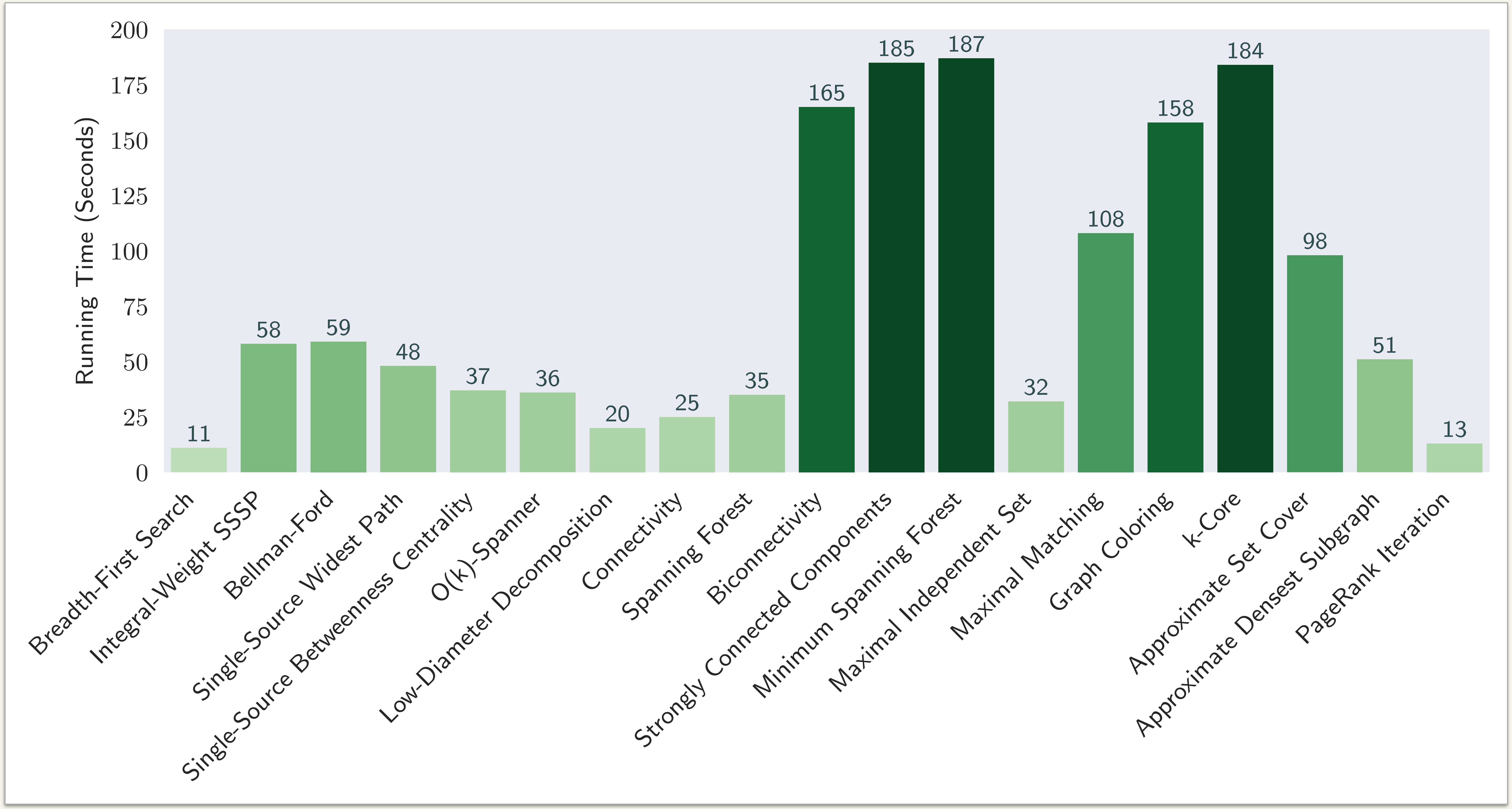




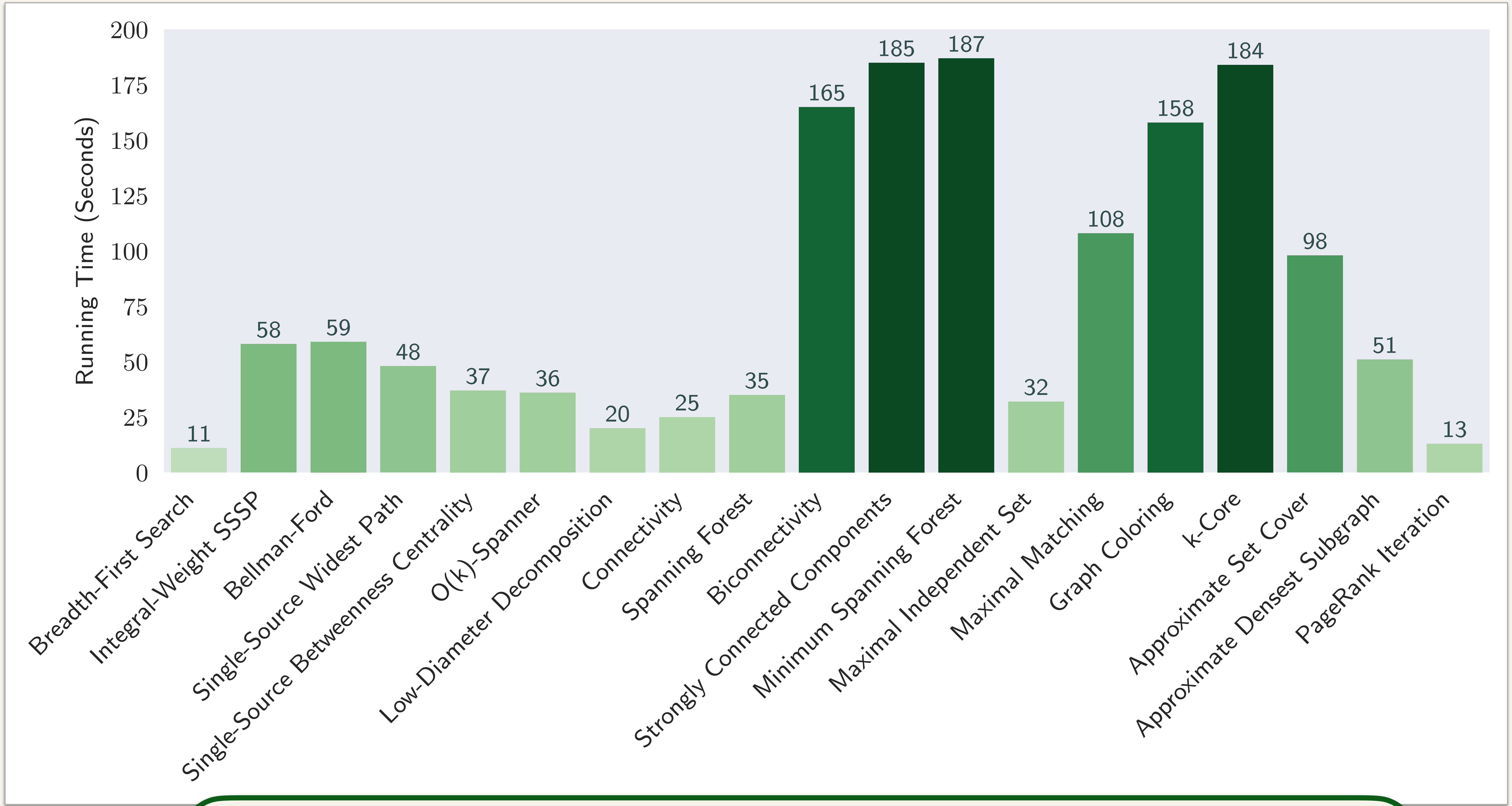
Outperform external memory results by orders of magnitude using comparable hardware.

Outperform distributed memory results using orders of magnitude less hardware.

GBBS can analyze O(100B) edge graphs in seconds to minutes



GBBS can analyze O(100B) edge graphs in seconds to minutes



A broad set of fundamental graph problems can be solved on a graph with over 200 billion edges in 3 minutes

Work and Depth of GBBS Results

† : in expectation $*$: whp

Problem	**Work**	**Depth**
Breadth-First Search (BFS)	$O(m)$	$\tilde{O}(\text{diam}(G))$
Integral-Weight SSSP (weighted BFS)	$O(m)^\dagger$	$\tilde{O}(\text{diam}(G))*$
General-Weight SSSP (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Widest Path (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Betweenness Centrality (BC)	$O(m)$	$\tilde{O}(\text{diam}(G))$
O(k)-Spanner	$O(m)$	$\tilde{O}(k \log n)*$
Low-Diameter Decomposition (LDD)	$O(m)$	$O(\log^2 n)*$
Connectivity (CC)	$O(m)^\dagger$	$O(\log^3 n)*$
Spanning Forest	$O(m)^\dagger$	$O(\log^3 n)*$
Biconnectivity	$O(m)^\dagger$	$O(\max(\text{CC}, \text{BFS}))$
Strongly Connected Components (SCC)	$O(m \log n)^\dagger$	$\tilde{O}(\text{diam}(G))*$
Minimum Spanning Forest (MSF)	$O(m \log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	$O(m)^\dagger$	$O(\log^2 n)*$
Maximal Matching (MM)	$O(m)^\dagger$	$O(\log^2 n)*$
Graph Coloring	$O(m)$	$O(\log n + L \log \Delta)$
k-core	$O(m)^\dagger$	$O(\rho \log n)*$
Approximate Set Cover	$O(m)^\dagger$	$O(\log^3 n)*$
Triangle Counting (TC)	$O(m^{3/2})$	$O(\log n)$
Approximate Densest Subgraph	$O(m)$	$O(\log^2 n)$
PageRank Iteration	$O(n + m)$	$O(\log n)$

Work and Depth of GBBS Results

† : in expectation $*$: whp

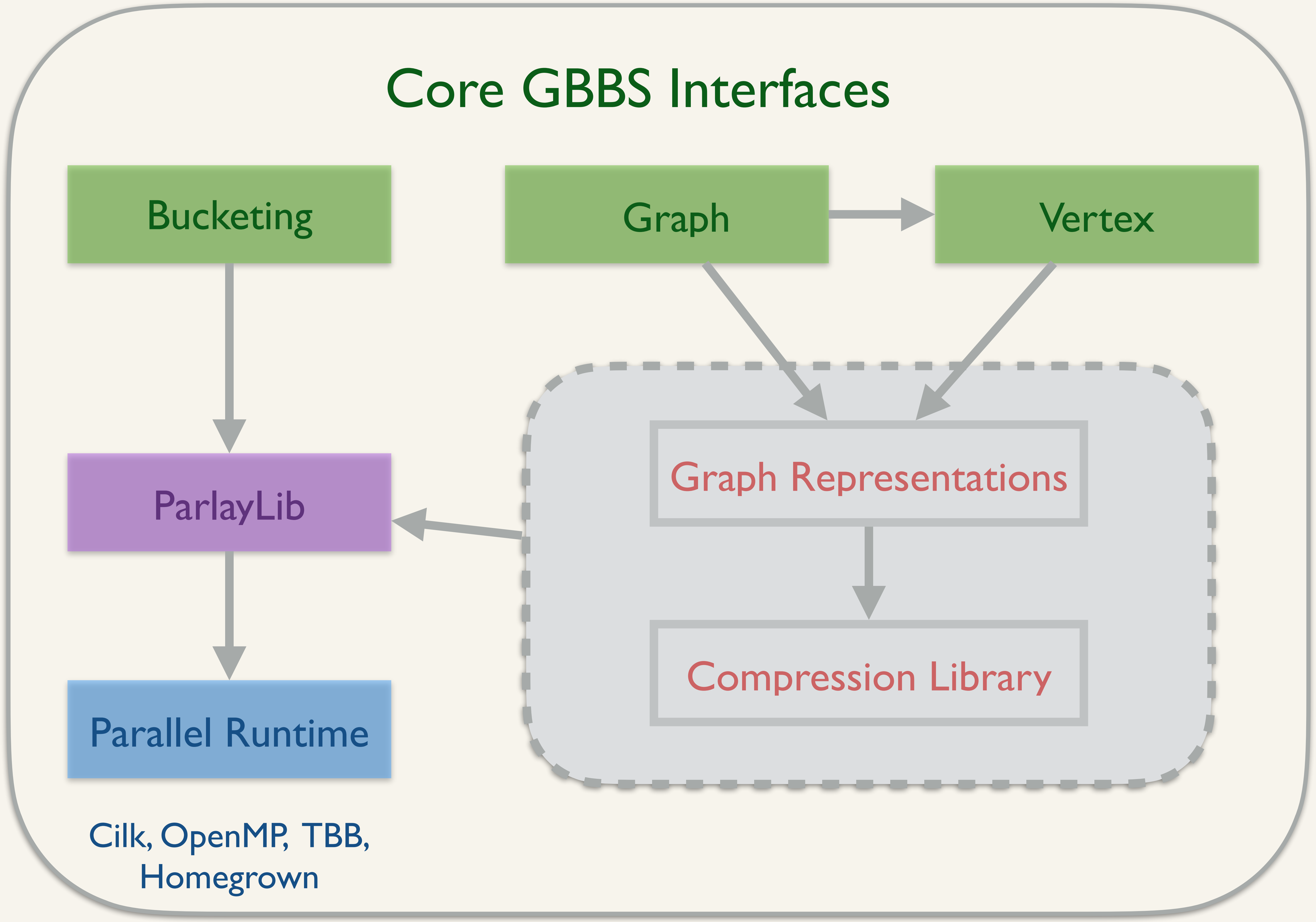
Problem	Work	Depth
Breadth-First Search (BFS)	$O(m)$	$\tilde{O}(\text{diam}(G))$
Integral-Weight SSSP (weighted BFS)	$O(m)^\dagger$	$\tilde{O}(\text{diam}(G))^*$
General-Weight SSSP (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Widest Path (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Betweenness Centrality (BC)	$O(m)$	$\tilde{O}(\text{diam}(G))$
O(k)-Spanner	$O(m)$	$\tilde{O}(k \log n)^*$
Low-Diameter Decomposition (LDD)	$O(m)$	$O(\log^2 n)^*$
Connectivity (CC)	$O(m)^\dagger$	$O(\log^3 n)^*$
Spanning Forest	$O(m)^\dagger$	$O(\log^3 n)^*$
Biconnectivity	$O(m)^\dagger$	$O(\max(\text{CC}, \text{BFS}))$
Strongly Connected Components (SCC)	$O(m \log n)^\dagger$	$\tilde{O}(\text{diam}(G))^*$
Minimum Spanning Forest (MSF)	$O(m \log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	$O(m)^\dagger$	$O(\log^2 n)^*$
Maximal Matching (MM)	$O(m)^\dagger$	$O(\log^2 n)^*$
Approximate Densest Subgraph	$O(m)$	[illegible]
PageRank Iteration	$O(n + m)$	$O(\log n)$

Main Challenge:

How do we build simple and provably-efficient implementations of these algorithms that work on the largest real-world graphs?

GBBS Library

- High-level graph processing interface in the lineage of *Ligra* [SB'12]



GBBS Library

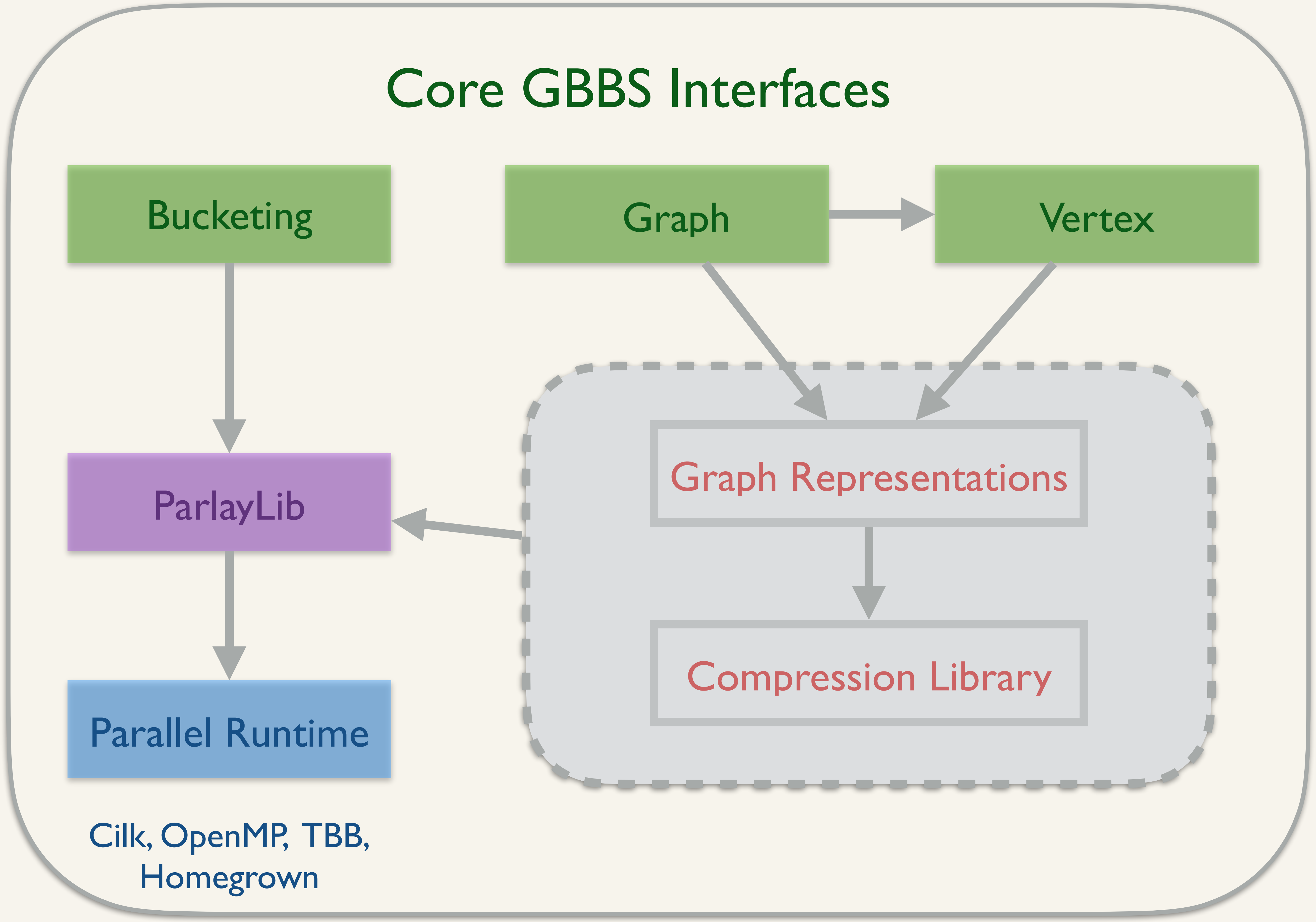
- High-level graph processing interface in the lineage of *Ligra* [SB'12]
- Provides many useful primitives

Vertex Operations

- Map
- Reduce
- Filter
- Pack
- Intersect

Graph Operations

- Filter
- Pack
- Contract



GBBS Library

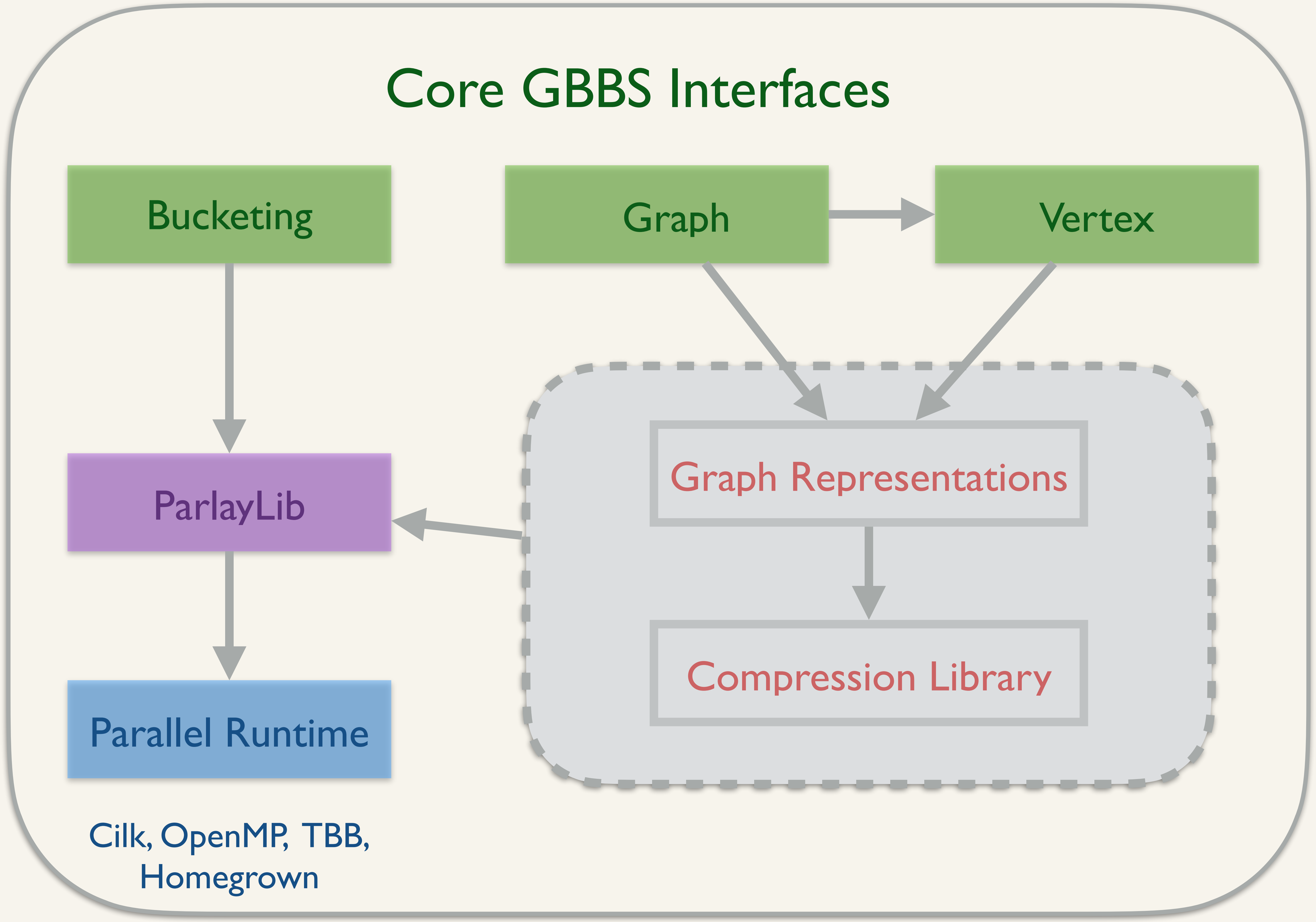
- High-level graph processing interface in the lineage of *Ligra* [SB'12]
- Provides many useful primitives

Vertex Operations

- Map
- Reduce
- Filter
- Pack
- Intersect

Graph Operations

- Filter
- Pack
- Contract



- Compressed graph representations

Graph	|*V*|	|*E*|	*Size (CSR)*	*Compressed*	*Bytes/edge*
WDC Hyperlink	3.5B	128B	1080GB	446GB	1.74
WDC Hyperlink (Sym)	3.5B	225B	928 GB	351GB	1.56

k-Core Decomposition

k-Core Decomposition

k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

k-Core Decomposition

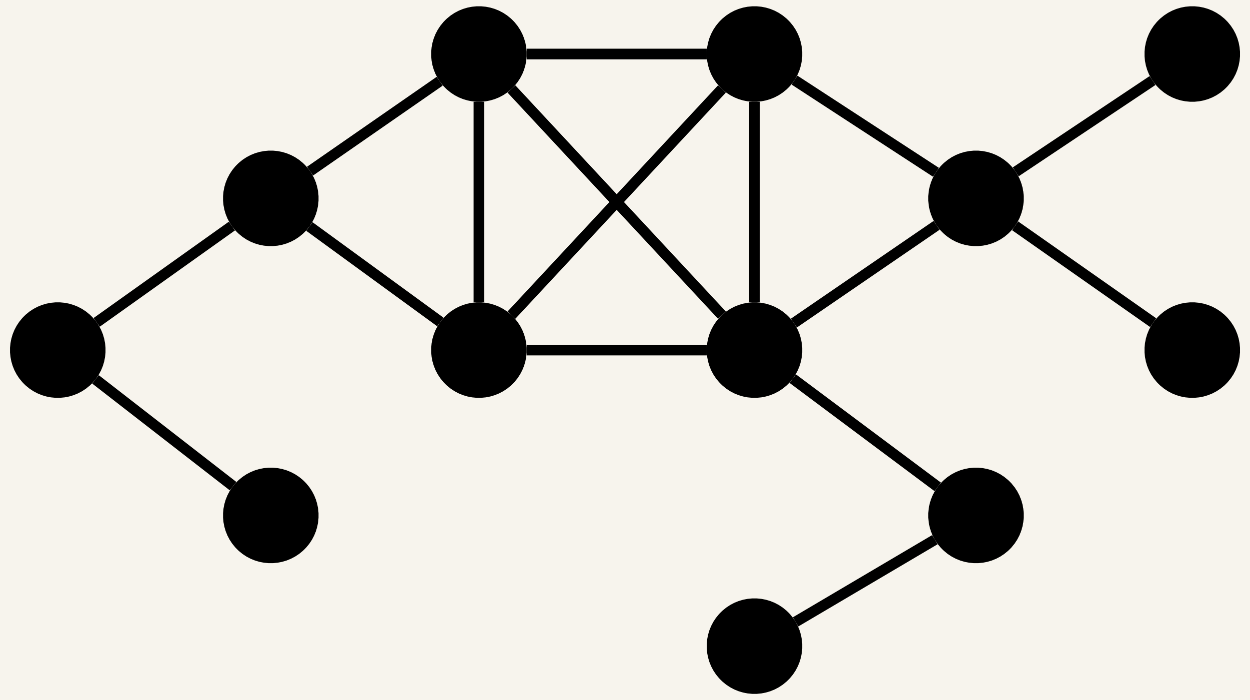
k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

coreness : largest k-core that a given vertex participates in

k-Core Decomposition

k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

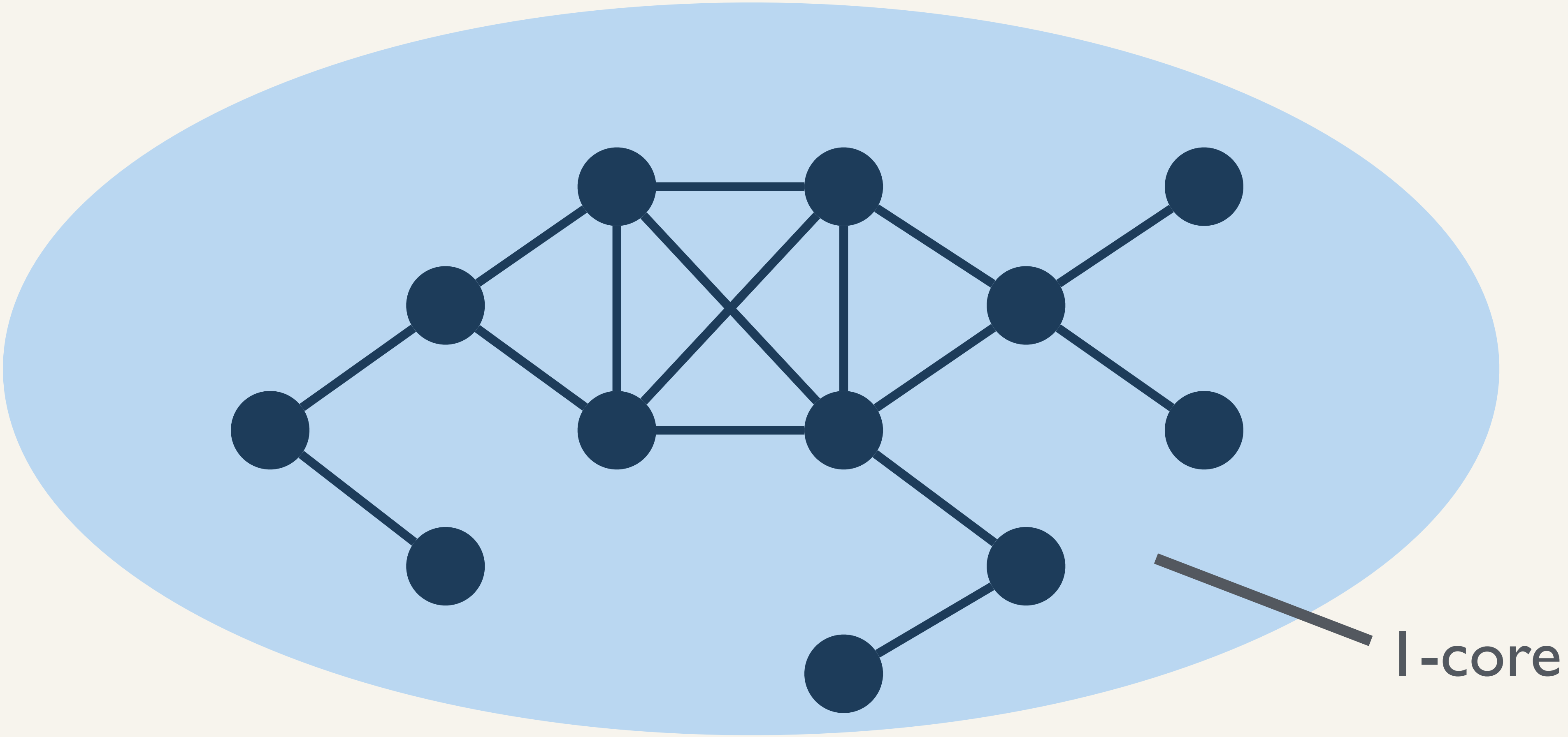
coreness : largest k-core that a given vertex participates in



k-Core Decomposition

k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

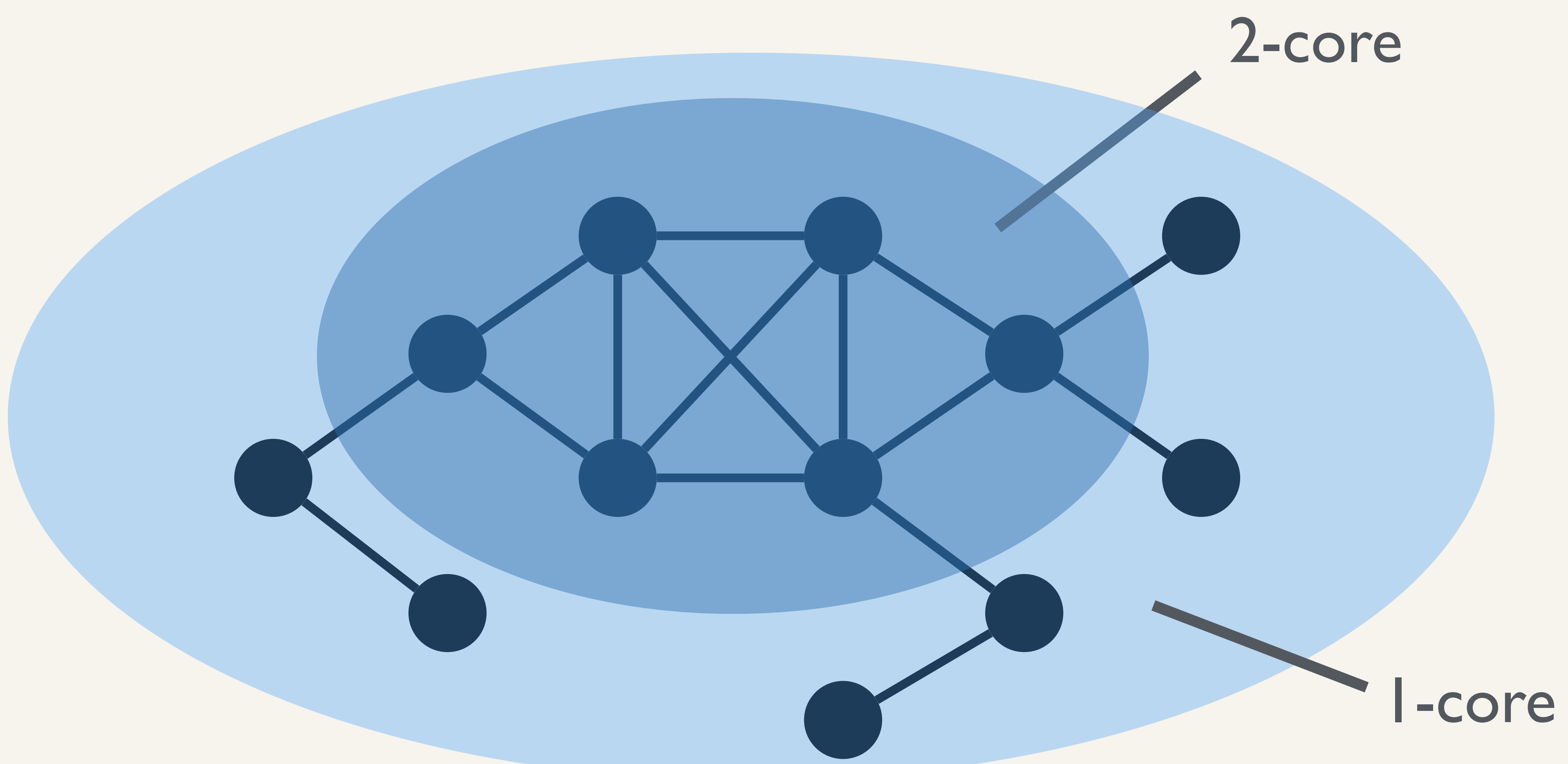
coreness : largest k-core that a given vertex participates in



k-Core Decomposition

k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

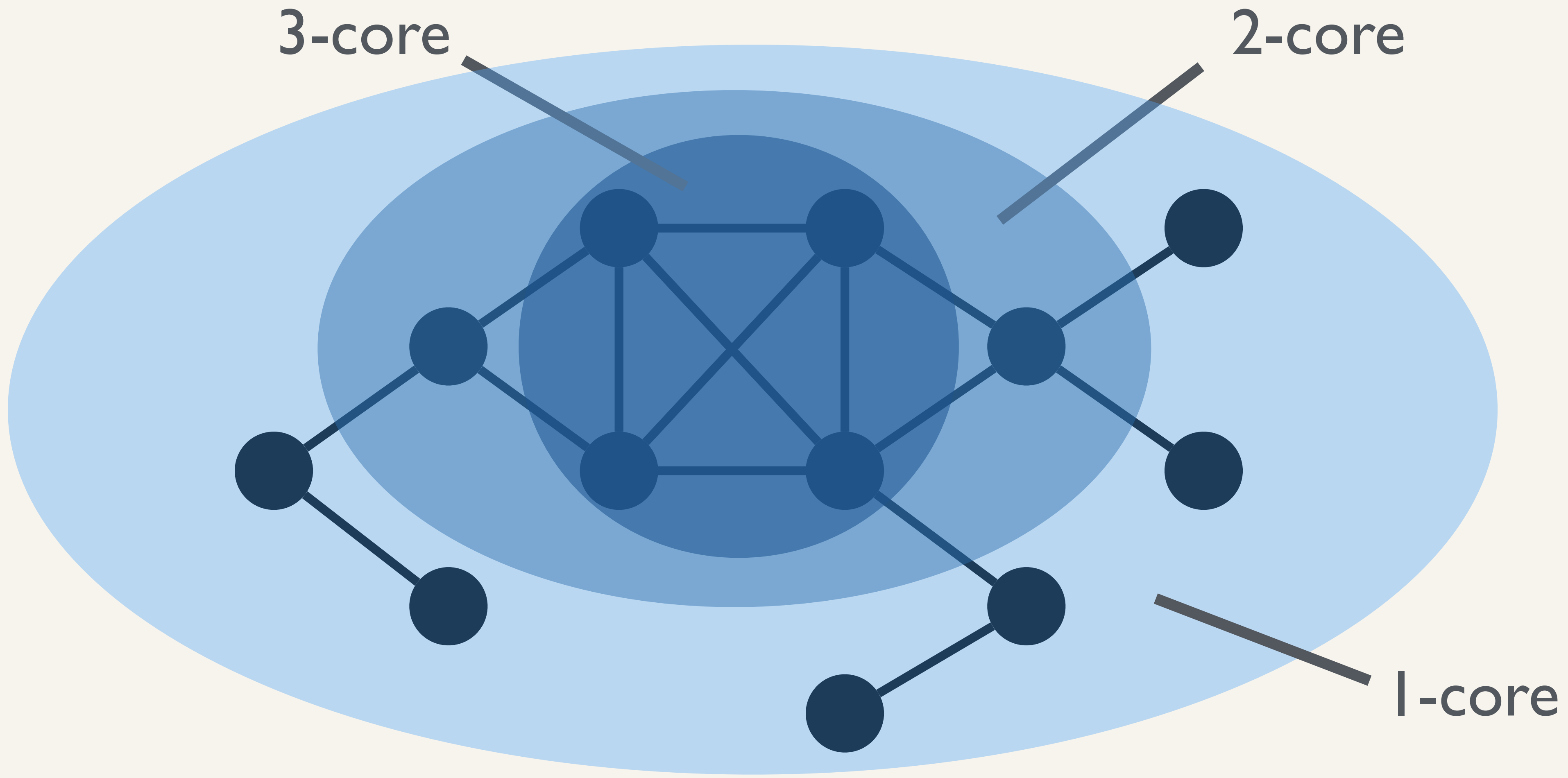
coreness : largest k-core that a given vertex participates in



k-Core Decomposition

k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

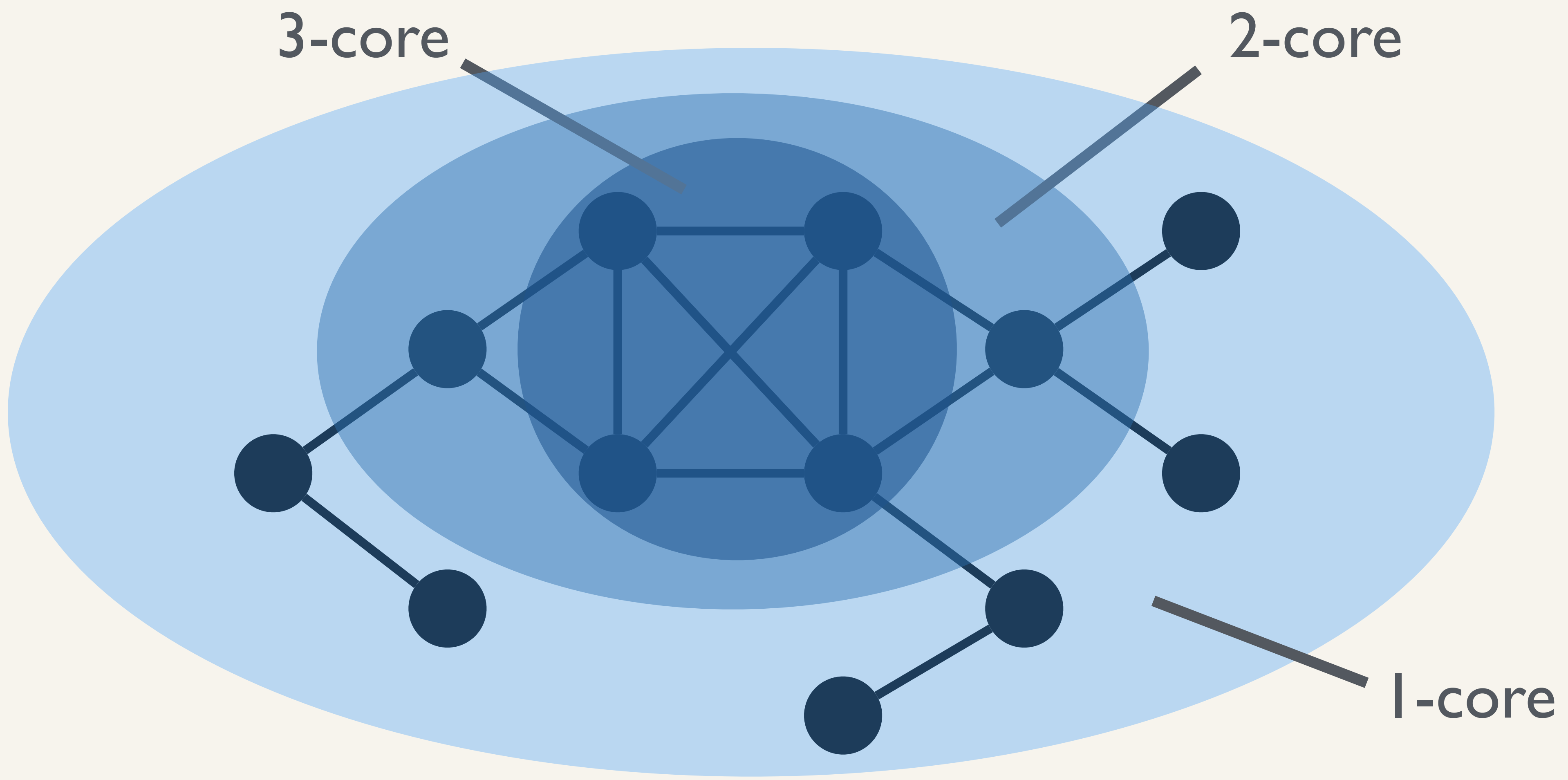
coreness : largest k-core that a given vertex participates in



k-Core Decomposition

k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

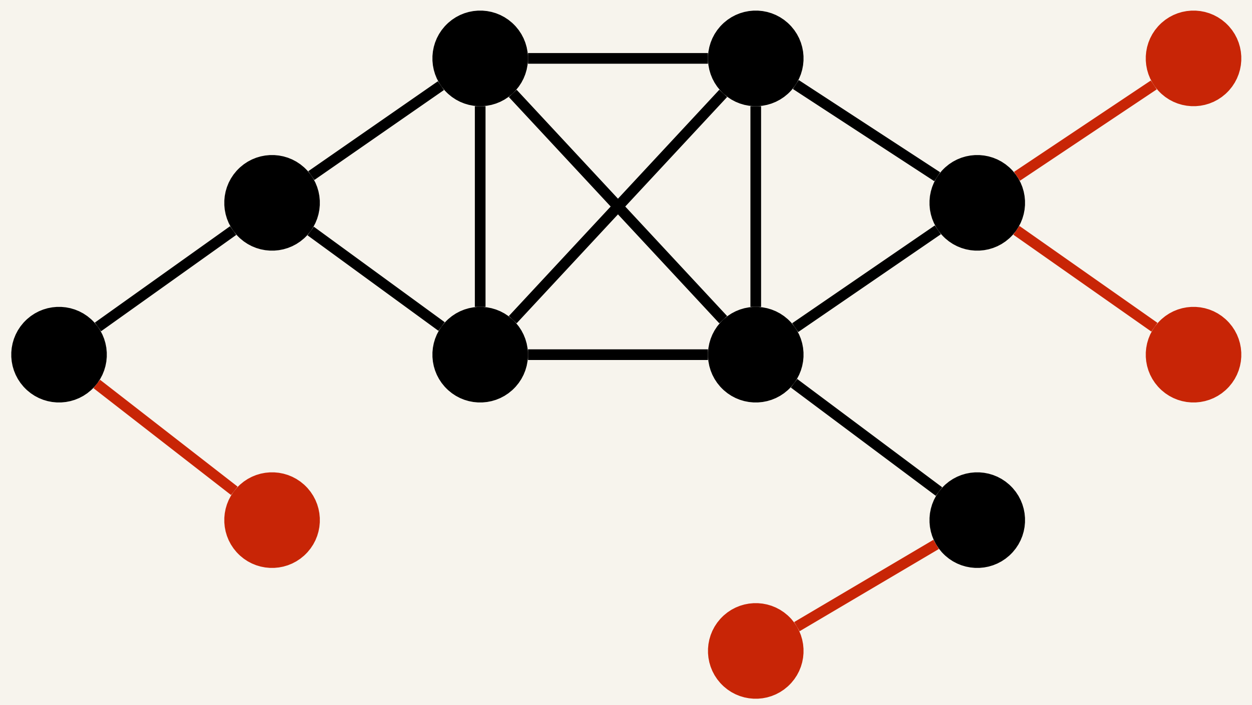
coreness : largest k-core that a given vertex participates in



Widely used in network analysis tasks such as unsupervised clustering of social and biological networks

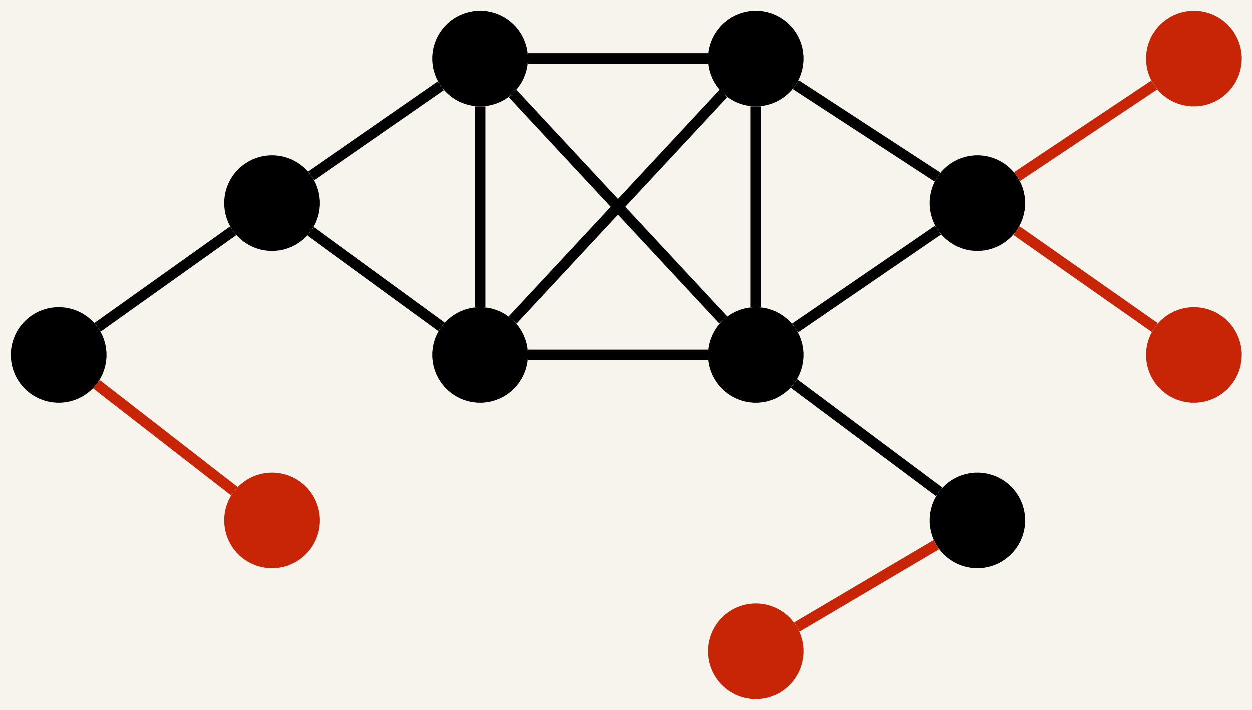
The Peeling Algorithm

$k = 1$

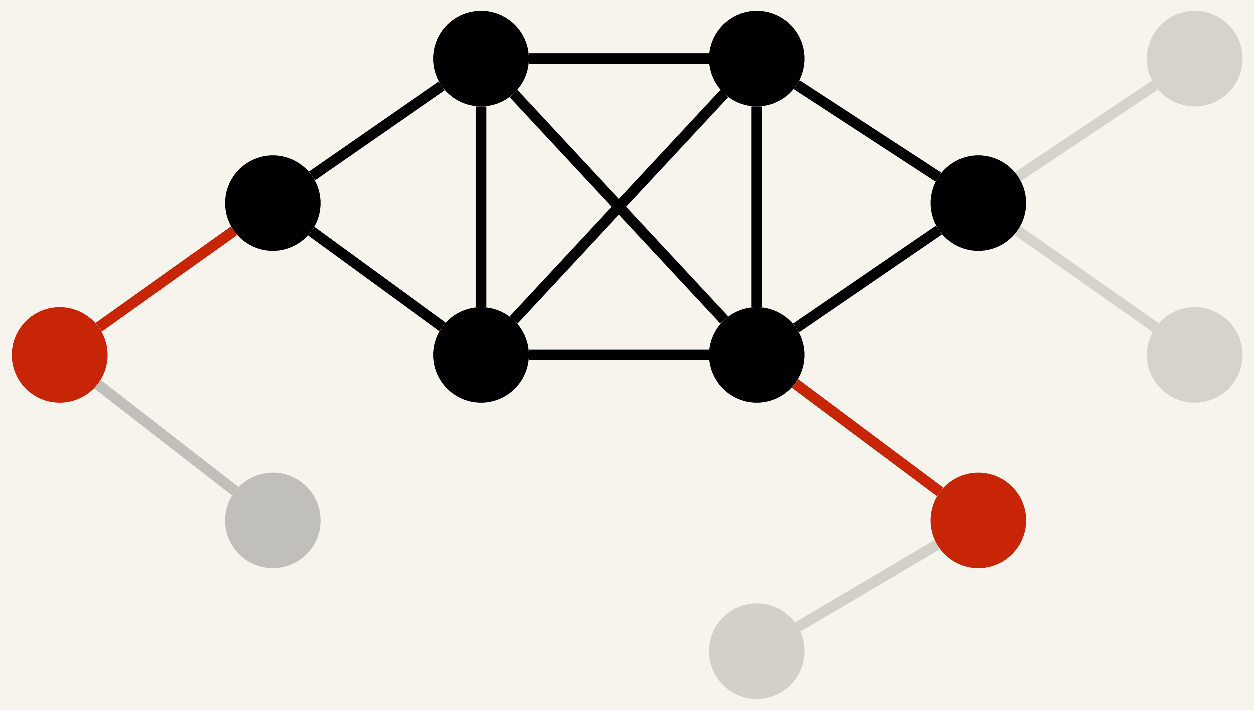


The Peeling Algorithm

$k = 1$

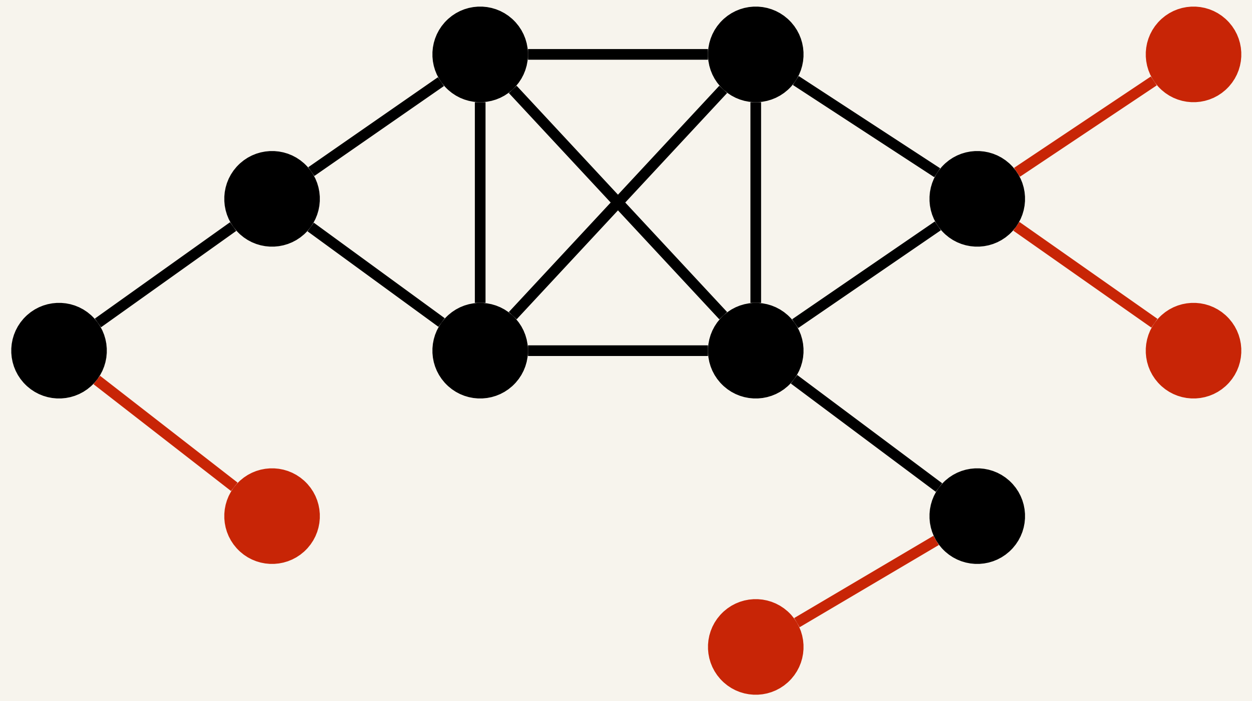


$k = 1$

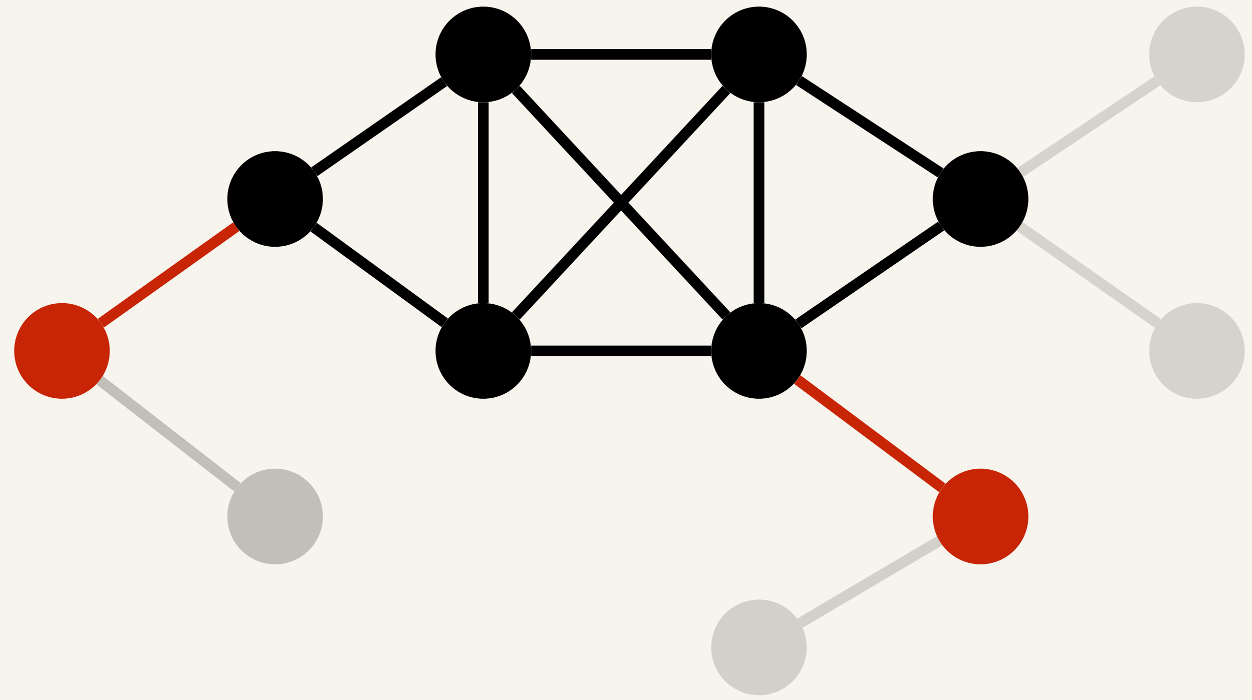


The Peeling Algorithm

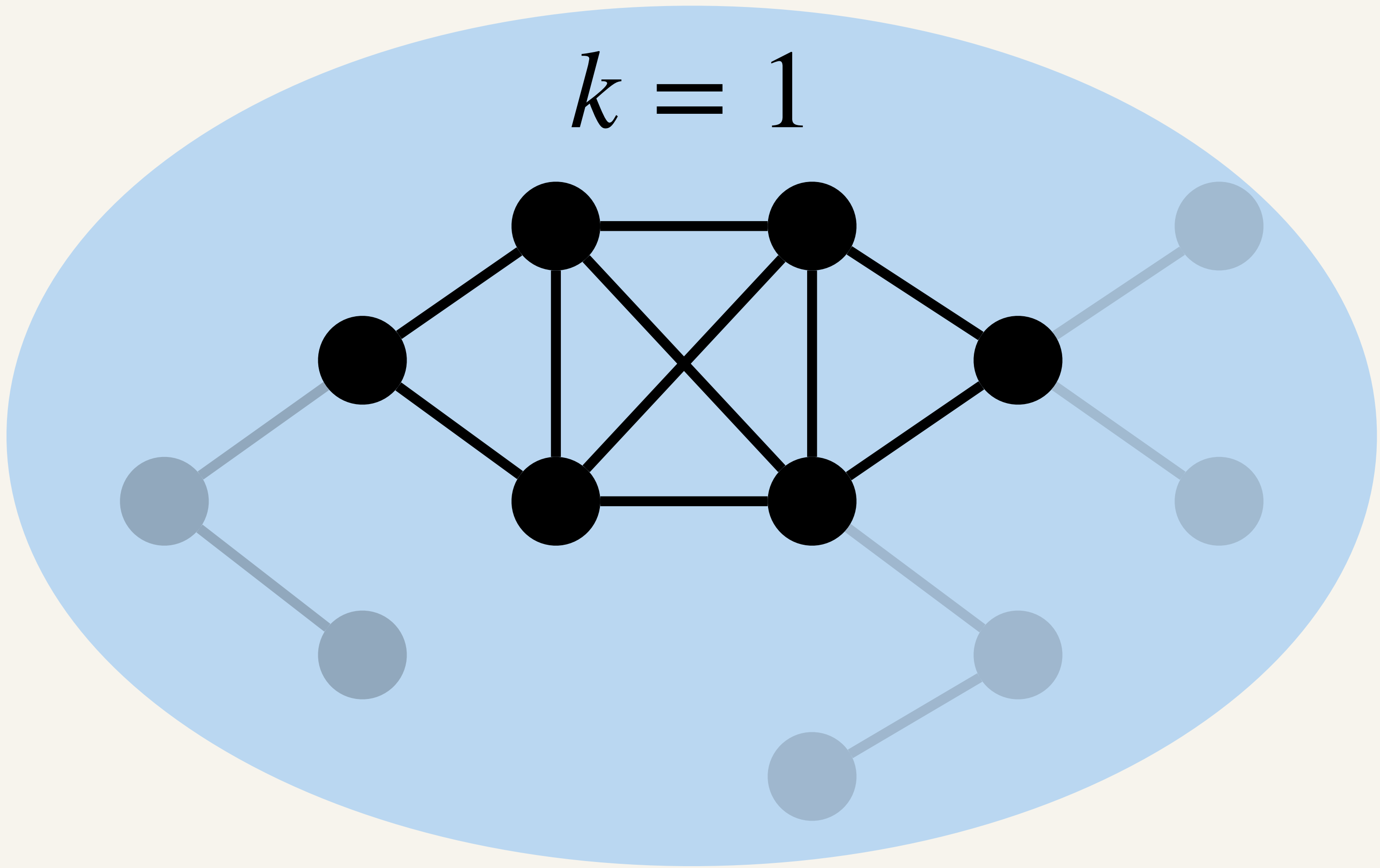
$k = 1$



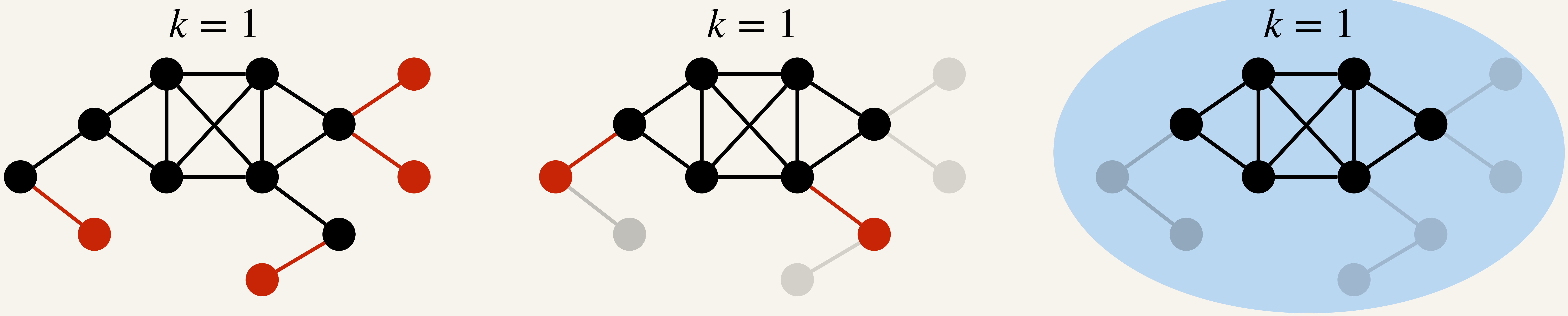
$k = 1$



$k = 1$

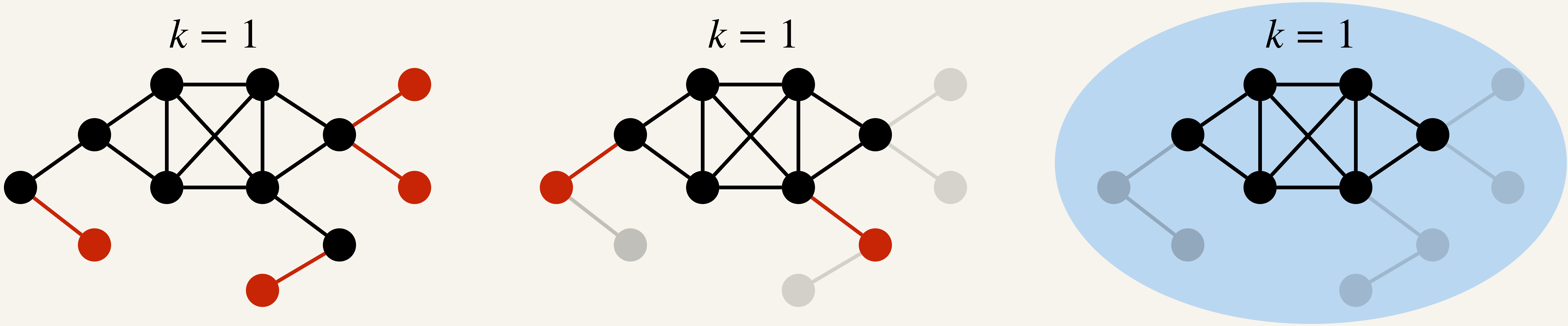


The Peeling Algorithm



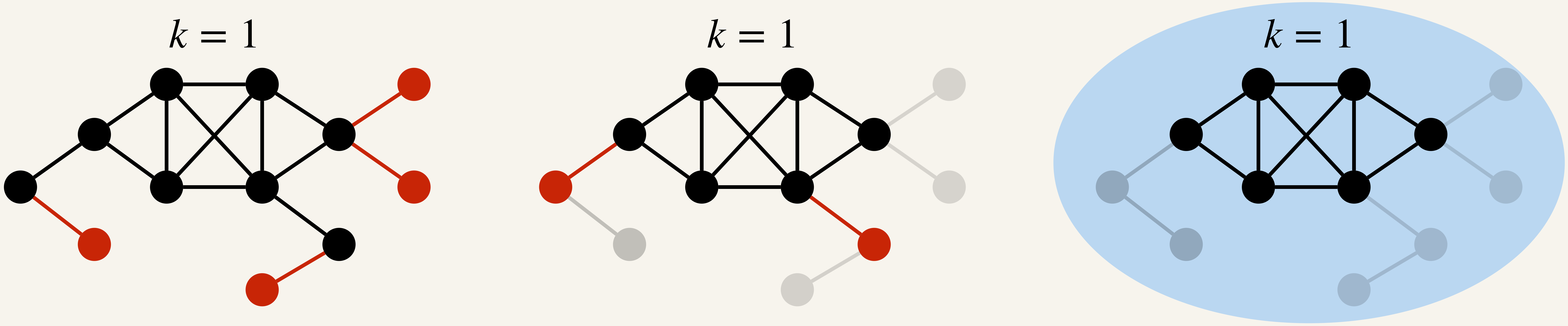
- Current degree of remaining vertices decreases as vertices are *peeled* from the graph

The Peeling Algorithm



- Current degree of remaining vertices decreases as vertices are *peeled* from the graph
- Once a vertex's current degree is less than or equal to the current core number, it gets peeled

The Peeling Algorithm



- Current degree of remaining vertices decreases as vertices are *peeled* from the graph
- Once a vertex's current degree is less than or equal to the current core number, it gets peeled

All vertices "below threshold" can be peeled in parallel
Our contribution is to give a general interface for bucketing

A Work-Efficient k-core Decomposition Algorithm

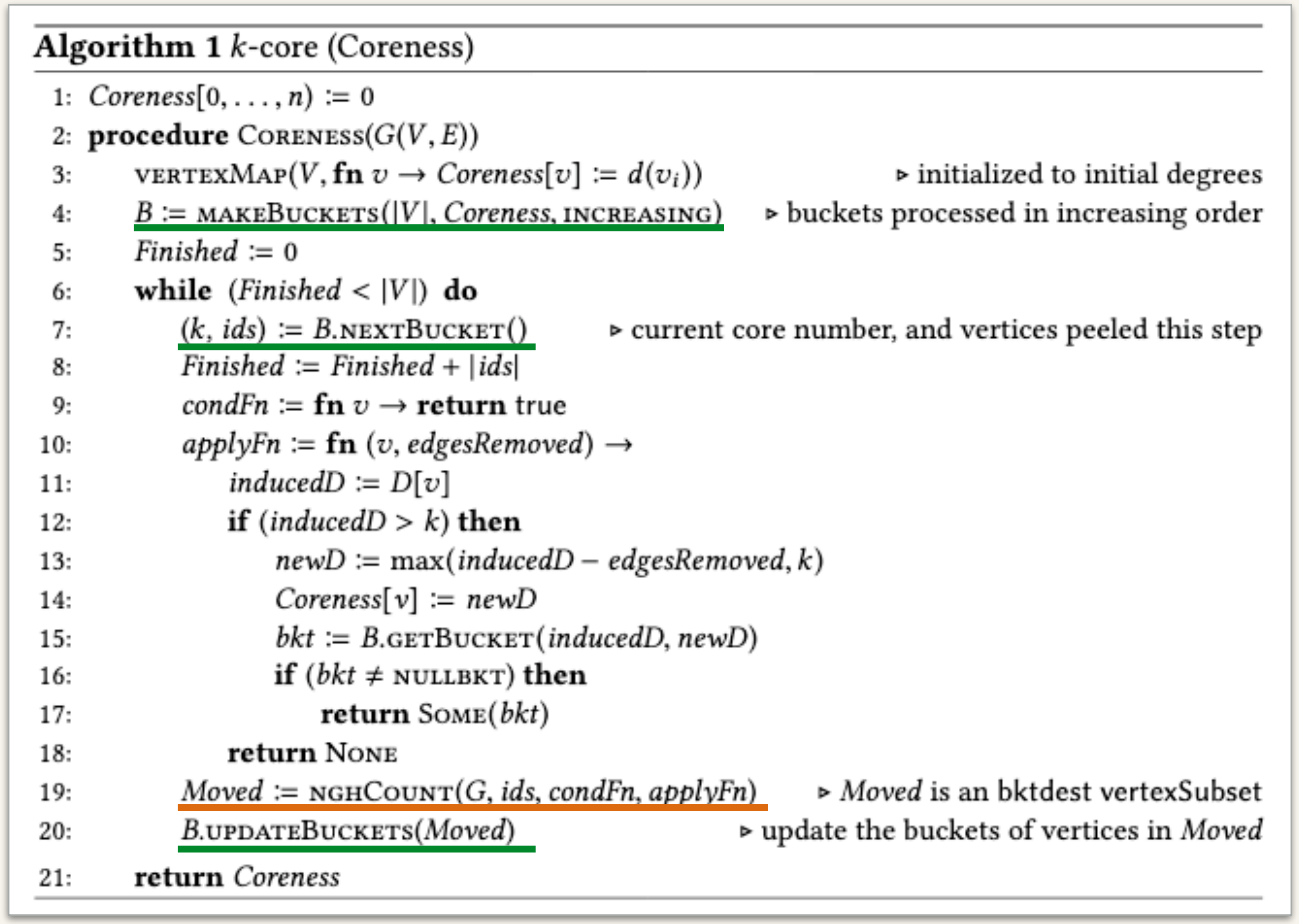
GBBS Algorithm

- Actual code in GBBS is under 50 lines of C++
- Parallel cost:

$O(m+n)$ expected work

$O(\rho \log n)$ depth whp

where ρ is the number of peeling rounds



Algorithm 1 *k*-core (Coreness)

```
Coreness$[0, \dots, n) := 0$
procedure CORENESS($G(V, E)$)
    VERTEXMAP($V$, fn $v \to$ Coreness$[v] := d(v_i)$)        ▹ initialized to initial degrees
    $B :=$ MAKEBUCKETS($|V|$, Coreness, INCREASING)        ▹ buckets processed in increasing order
    Finished $:= 0$
    while (Finished $< |V|$) do
        $(k, ids) := B$.NEXTBUCKET()        ▹ current core number, and vertices peeled this step
        Finished $:=$ Finished $+ |ids|$
        condFn $:=$ fn $v \to$ return true
        applyFn $:=$ fn $(v, edgesRemoved) \to$
            inducedD $:= D[v]$
            if (inducedD $> k$) then
                newD $:= \max(inducedD - edgesRemoved, k)$
                Coreness$[v] :=$ newD
                bkt $:= B$.GETBUCKET(inducedD, newD)
                if (bkt $\neq$ NULLBKT) then
                    return SOME(bkt)
            return NONE
        Moved $:=$ NGHCOUNT($G$, ids, condFn, applyFn)        ▹ Moved is an bktdest vertexSubset
        $B$.UPDATEBUCKETS(Moved)        ▹ update the buckets of vertices in Moved
    return Coreness
```

A Work-Efficient k-core Decomposition Algorithm

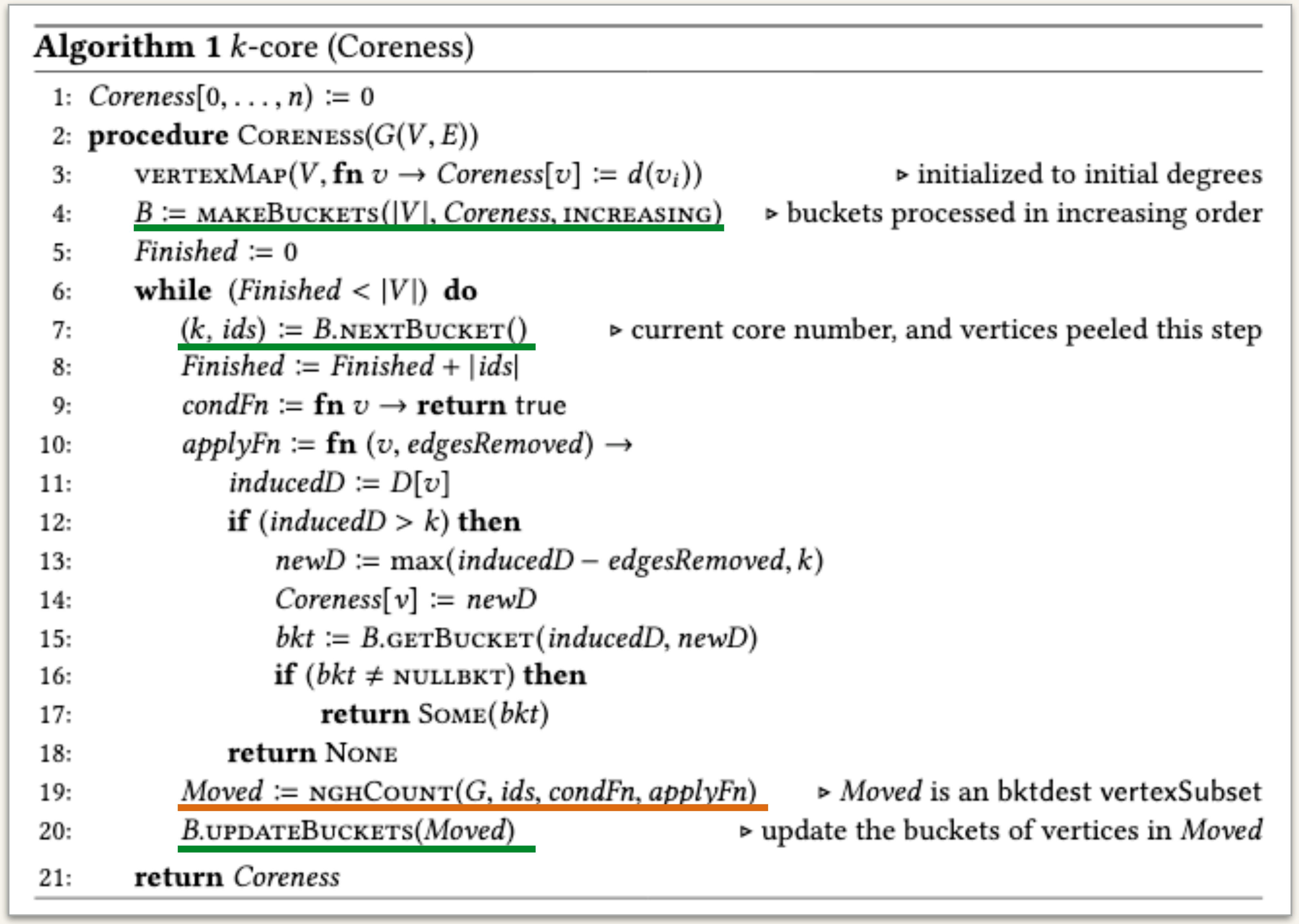
GBBS Algorithm

- Actual code in GBBS is under 50 lines of C++
- Parallel cost:

$O(m+n)$ expected work

$O(\rho \log n)$ depth whp

where ρ is the number of peeling rounds

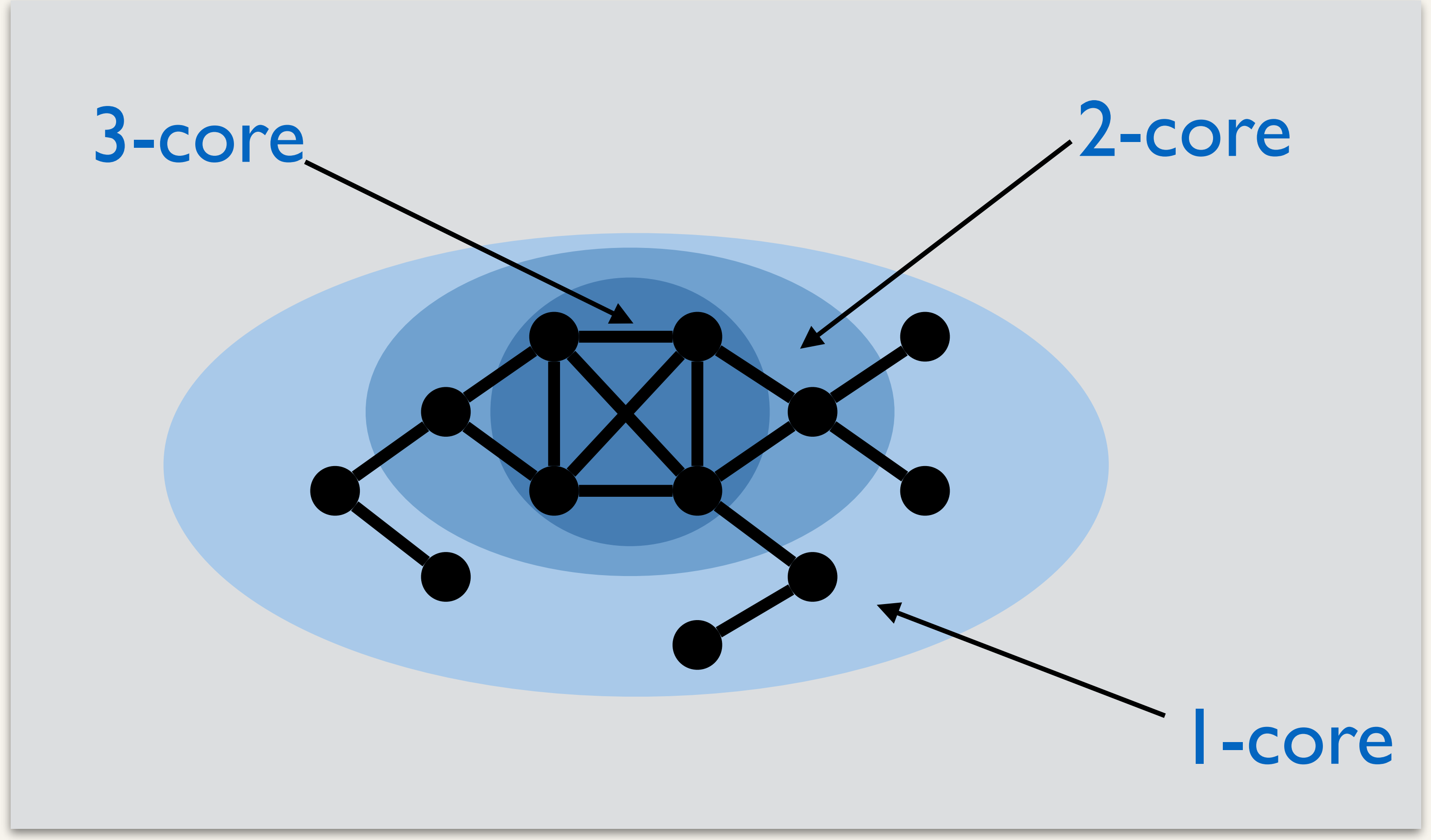


Algorithm 1 *k*-core (Coreness)

```
Coreness[0, ..., n) := 0
procedure CORENESS($G(V, E)$)
    VERTEXMAP($V$, fn $v \to Coreness[v] := d(v_i)$)        ▹ initialized to initial degrees
    $B$ := MAKEBUCKETS($|V|$, Coreness, INCREASING)        ▹ buckets processed in increasing order
    Finished := 0
    while (Finished < $|V|$) do
        ($k$, ids) := B.NEXTBUCKET()        ▹ current core number, and vertices peeled this step
        Finished := Finished + |ids|
        condFn := fn $v \to$ return true
        applyFn := fn ($v$, edgesRemoved) $\to$
            inducedD := $D[v]$
            if (inducedD > $k$) then
                newD := max(inducedD − edgesRemoved, $k$)
                Coreness[$v$] := newD
                bkt := B.GETBUCKET(inducedD, newD)
                if (bkt $\neq$ NULLBKT) then
                    return SOME(bkt)
            return NONE
        Moved := NGHCOUNT(G, ids, condFn, applyFn)        ▹ Moved is an bktdest vertexSubset
        B.UPDATEBUCKETS(Moved)        ▹ update the buckets of vertices in Moved
    return Coreness
```

Our algorithm is the first work-efficient algorithm for k-core decomposition with non-trivial parallelism

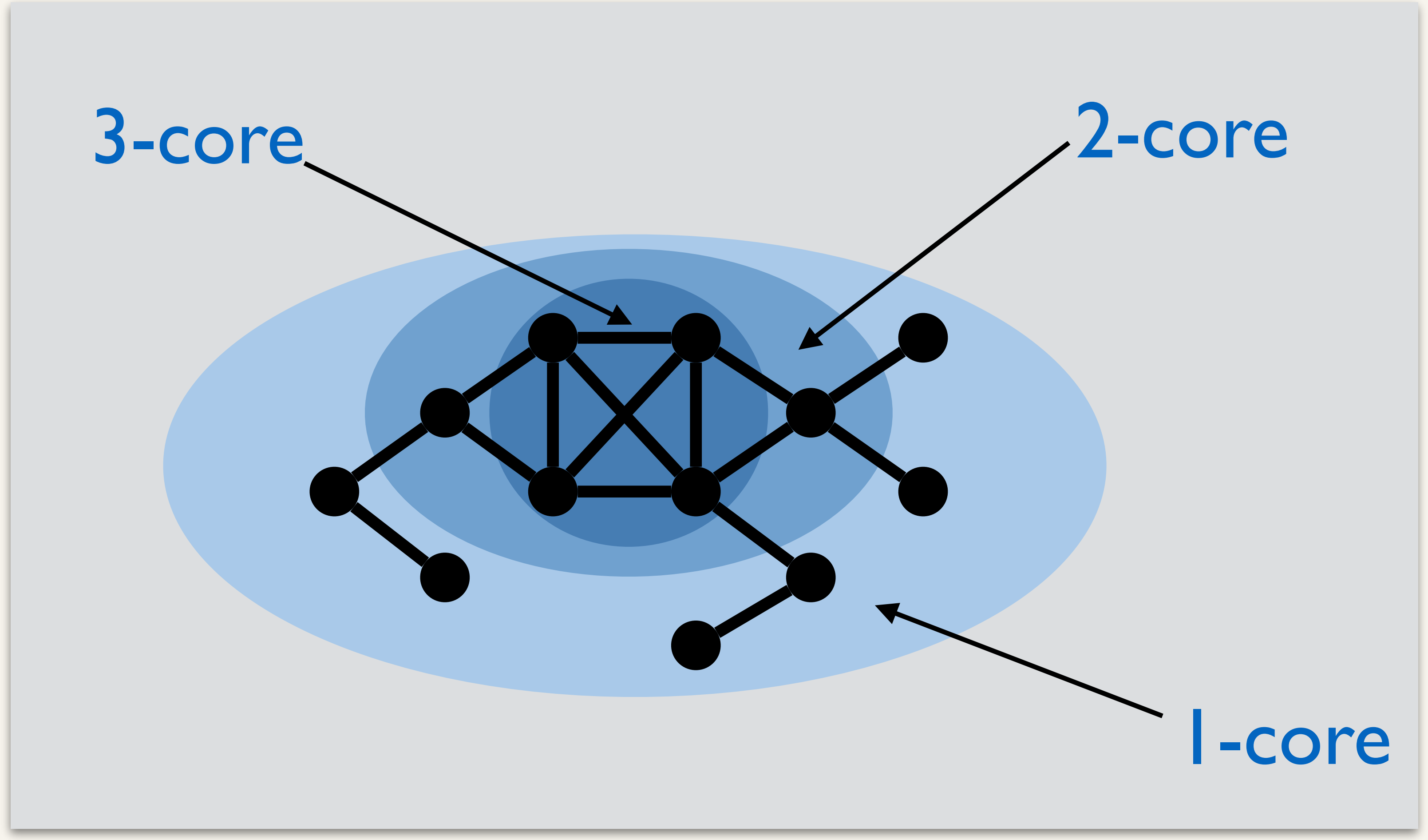
k-Core Decomposition on the WebDataCommons Graph



k-core : maximal connected subgraph of G s.t. all vertices have degree at least k

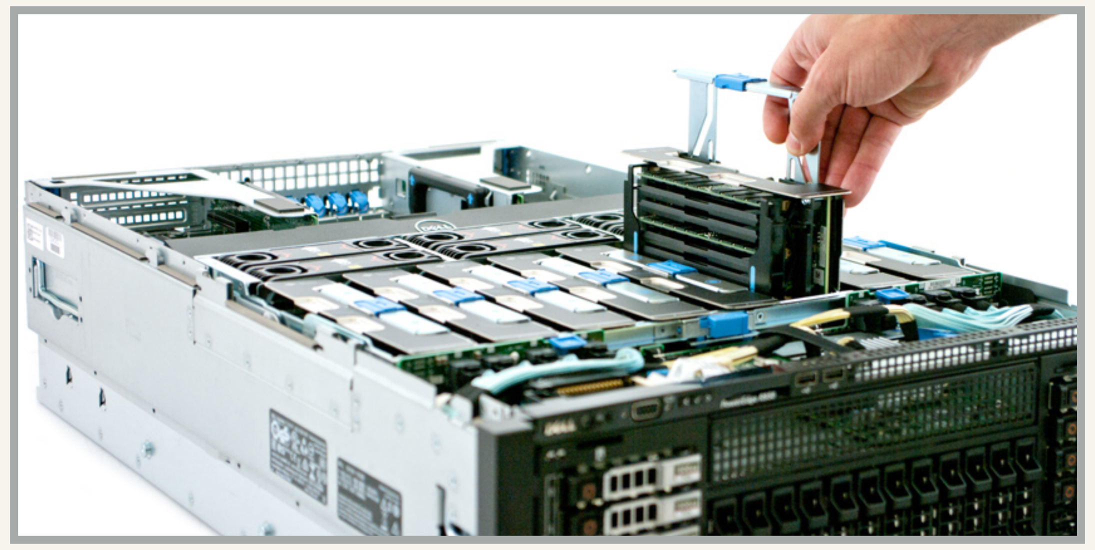
	BlueWaters [SRM'16]
Time	363 seconds
Processors	8192
Memory	16 TB
Quality	Approximate
Cost	Very Expensive

k-Core Decomposition on the WebDataCommons Graph

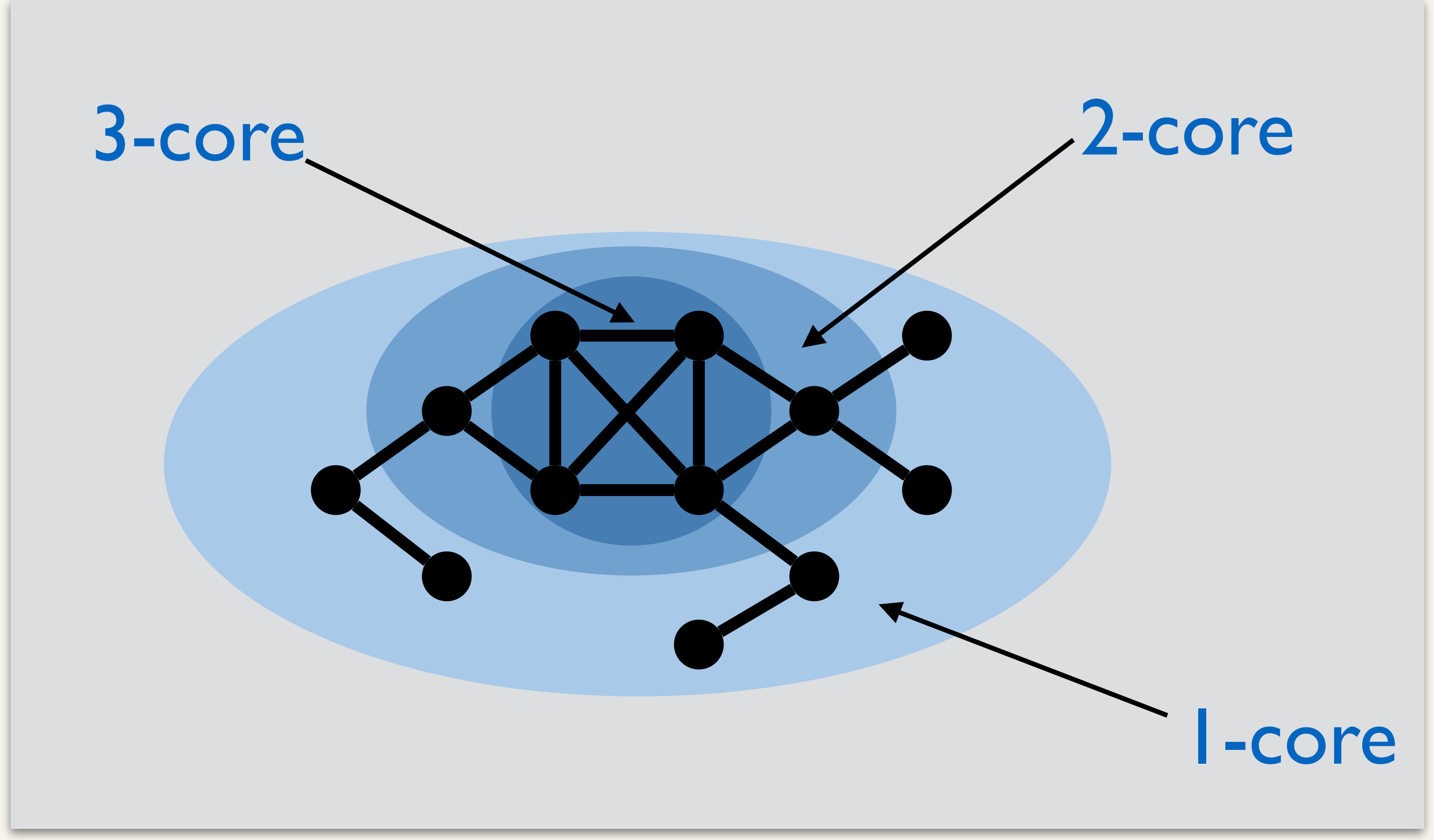


k-core : maximal connected subgraph of G s.t. all vertices have degree at least k

	BlueWaters [SRM'16]	GBBS [**D**BS'18]
Time	363 seconds	184 seconds
Processors	8192	72
Memory	16 TB	1 TB
Quality	Approximate	Exact
Cost	Very Expensive	Highly Affordable



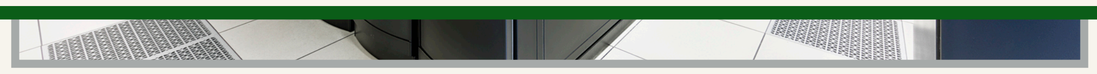
k-Core Decomposition on the WebDataCommons Graph



k-core : maximal connected subgraph of G s.t. all vertices have degree at least k

	BlueWaters [SRM'16]	GBBS [**D**BS'18]
Time	363 seconds	184 seconds
Processors	8192	72
Memory	16 TB	1 TB
Quality	Approximate	Exact
Cost	Very Expensive	Highly Affordable

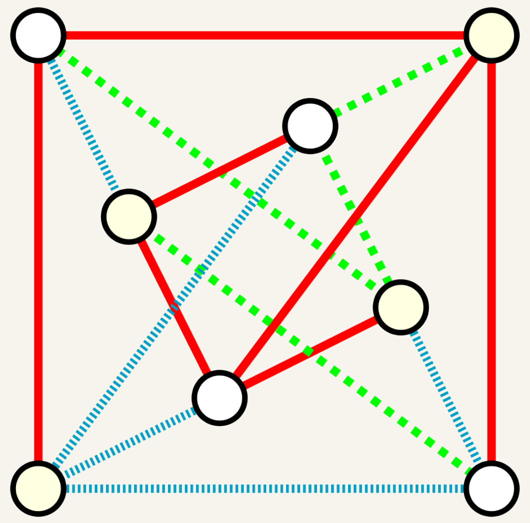
1.95x faster than the approximate distributed result by SRM'16, using 56.8x fewer hyper-threads and 16.3x less memory



GBBS as a Research Repository 5 Years On

Basis for many other parallel graph projects:

- Fast Parallel Graph Connectivity [DHS’21]
- Parallel k-clique enumeration [SDS’21]
- Graph Embedding [QDTPW’21]
- Structural Graph Clustering [TDS’21]
- Batch-Dynamic Graph Orientation [LSYDS’22]



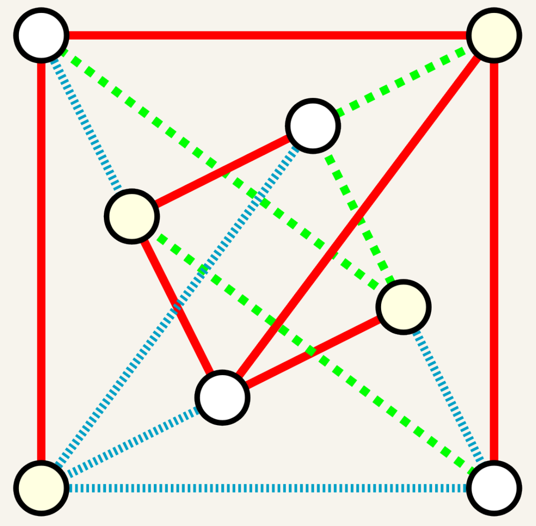
ArbClique

SAGE
Semi-Asymmetric
Graph Engine

GBBS as a Research Repository 5 Years On

Basis for many other parallel graph projects:

- Fast Parallel Graph Connectivity [DHS'21]
- Parallel k-clique enumeration [SDS'21]
- Graph Embedding [QDTPW'21]
- Structural Graph Clustering [TDS'21]
- Batch-Dynamic Graph Orientation [LSYDS'22]

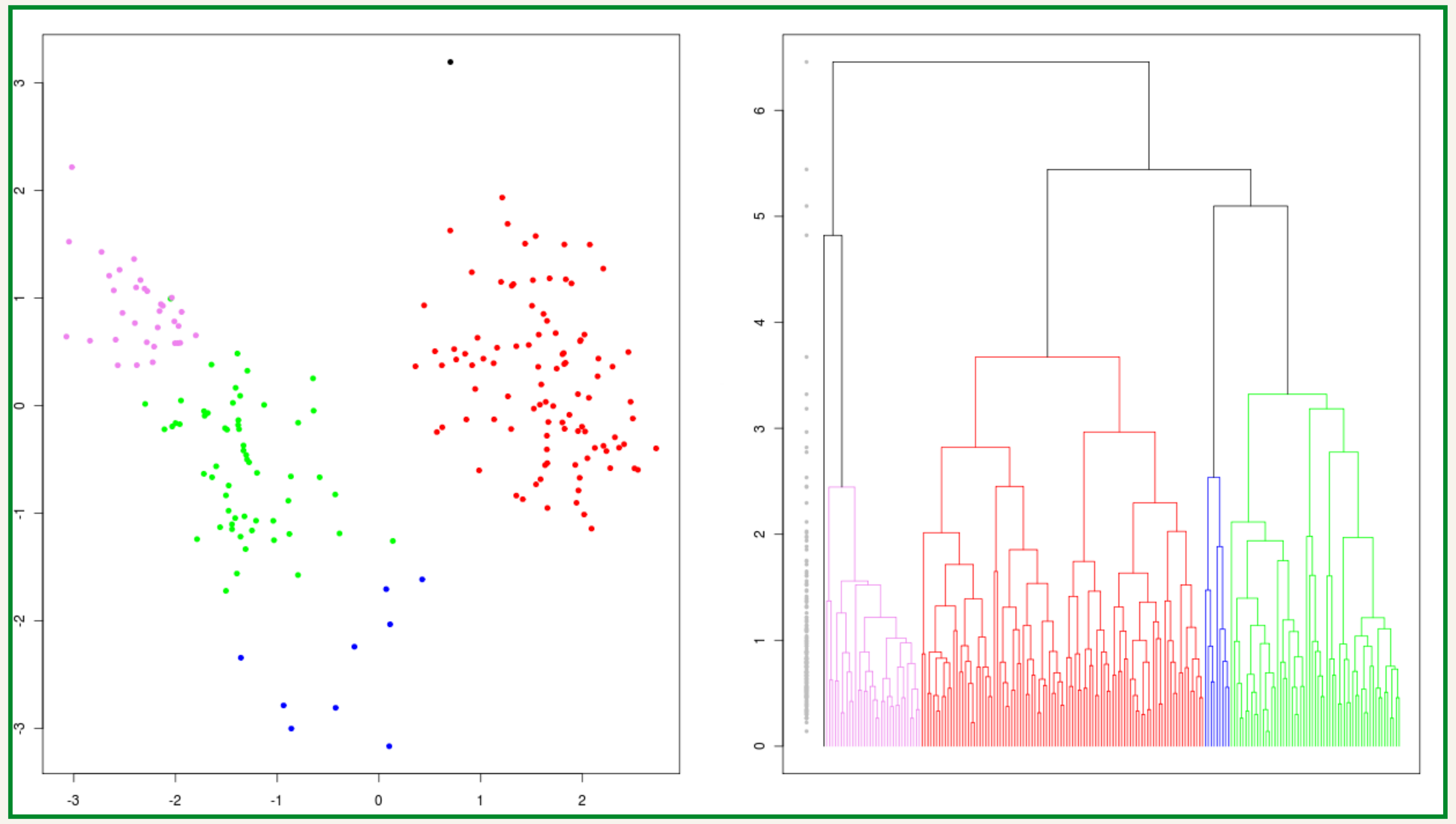


ArbClique

SAGE
Semi-Asymmetric
Graph Engine

Used at Google:

- Fast and scalable implementations of parallel graph clustering algorithms (e.g., Affinity Clustering)
- Being used to develop and evaluate parallel hierarchical agglomerative clustering (HAC) algorithms



Faster k-Means to Accelerate ANNS

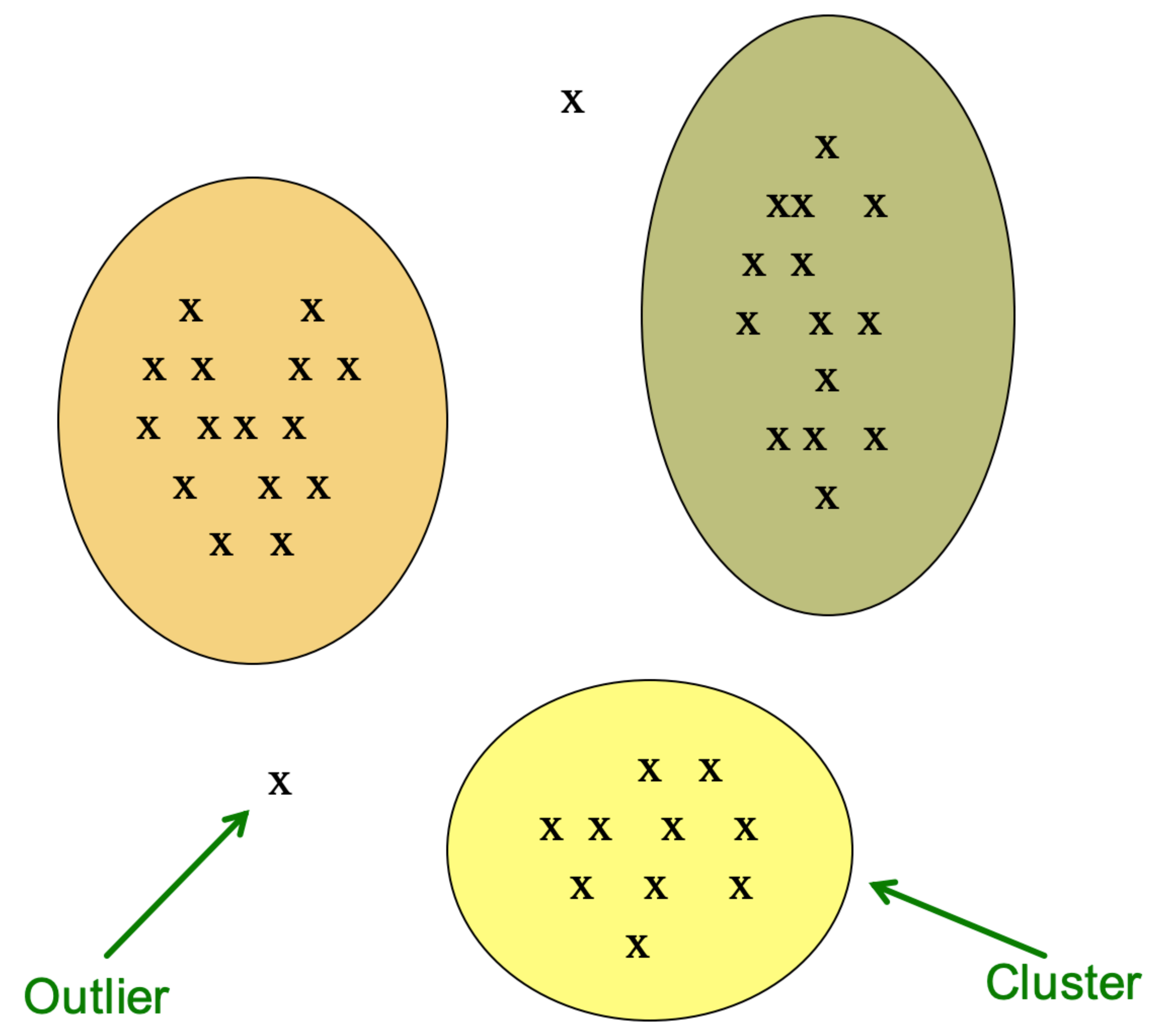
Clustering

- Given a set of points P with a notion of distance between the points, group the points into a number of clusters so that:
  - Members of the same cluster are close / similar to each other
  - Members of different clusters are dissimilar

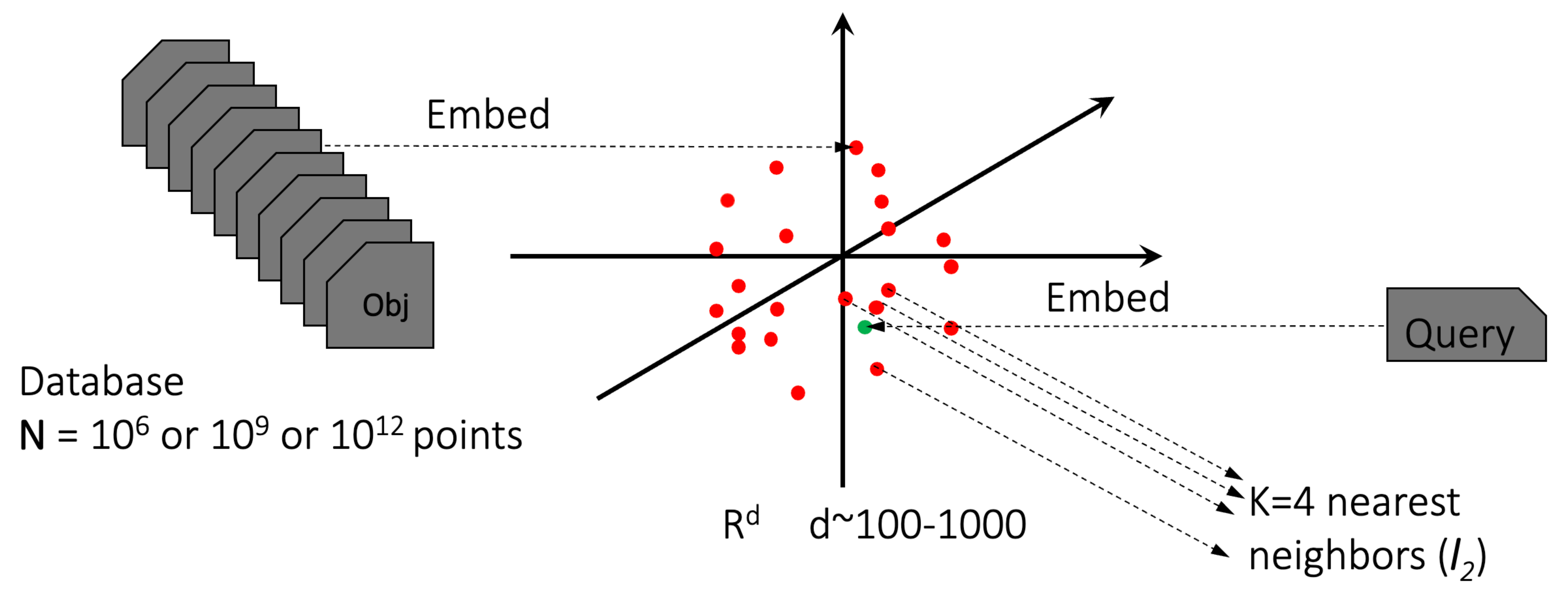
Usually:

- Points are in a high-dimensional space, e.g., $P \in \mathbb{R}^d, d \geq 100$
- Distance is measured using Euclidean distance, but other measures also possible (e.g., Jaccard, edit-distance, etc)

Clustering



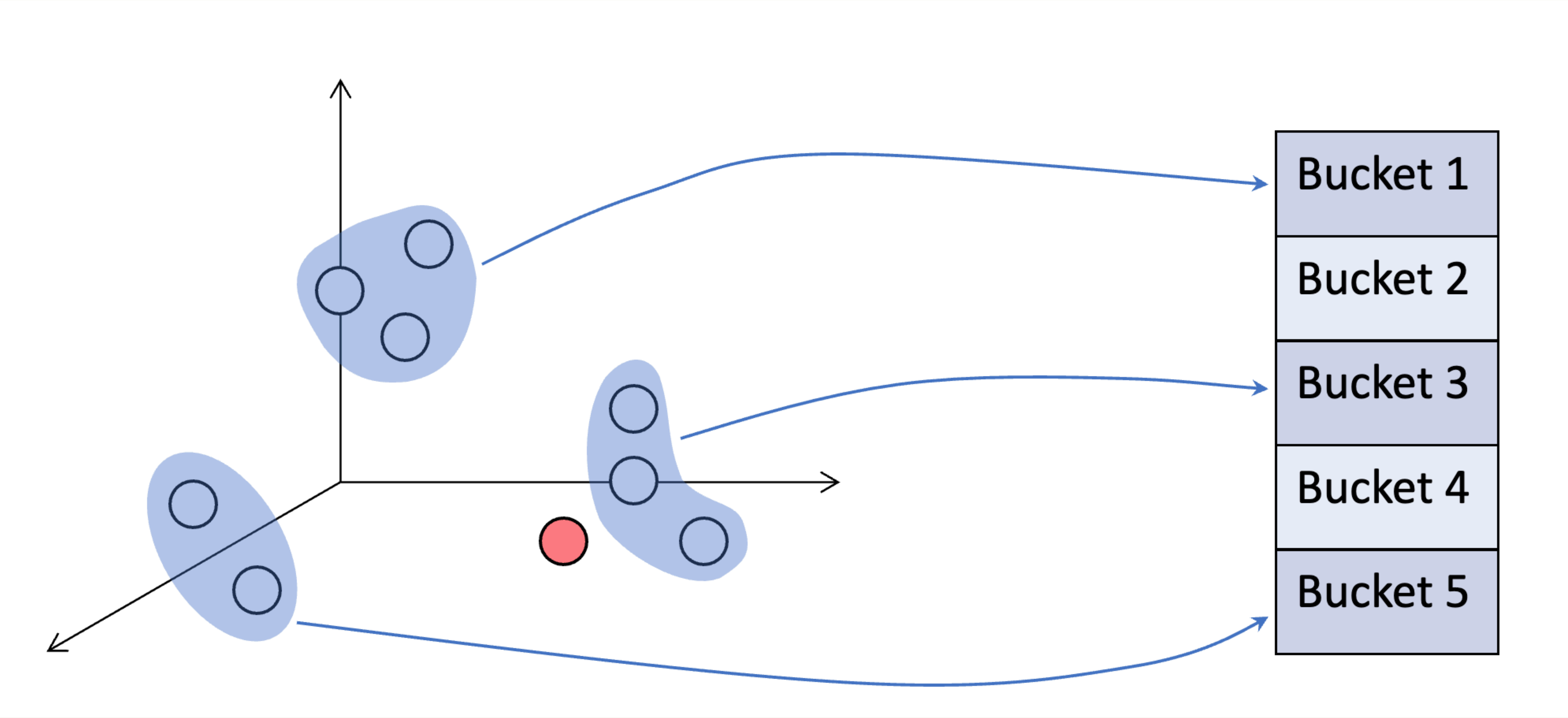
Clustering Problem: Building Bucketing-Based Indexes



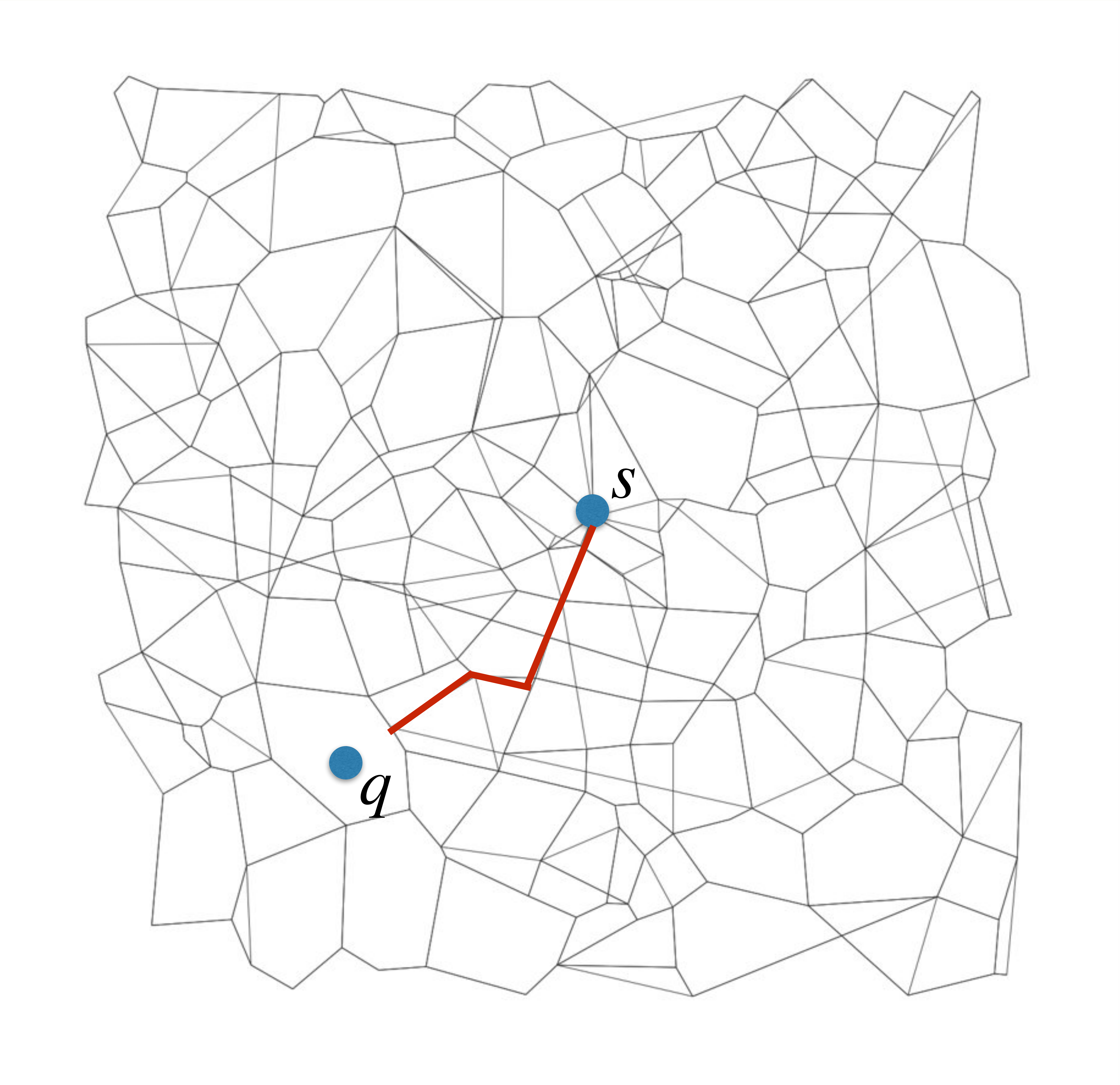
- Exact retrieval requires exhaustive scan in the worst case; settle for approximation instead.
- Measure recall@k: the fraction of output candidates in true top k neighbors

Clustering Problem: Building Bucketing-Based Indexes

- Build:
  - Assign points to one (or more) buckets
  - Nearby points likely to be in the same buckets
- Query:
  - Probe a subset of buckets for the queried point
  - Compare with all points in these buckets and report top-k



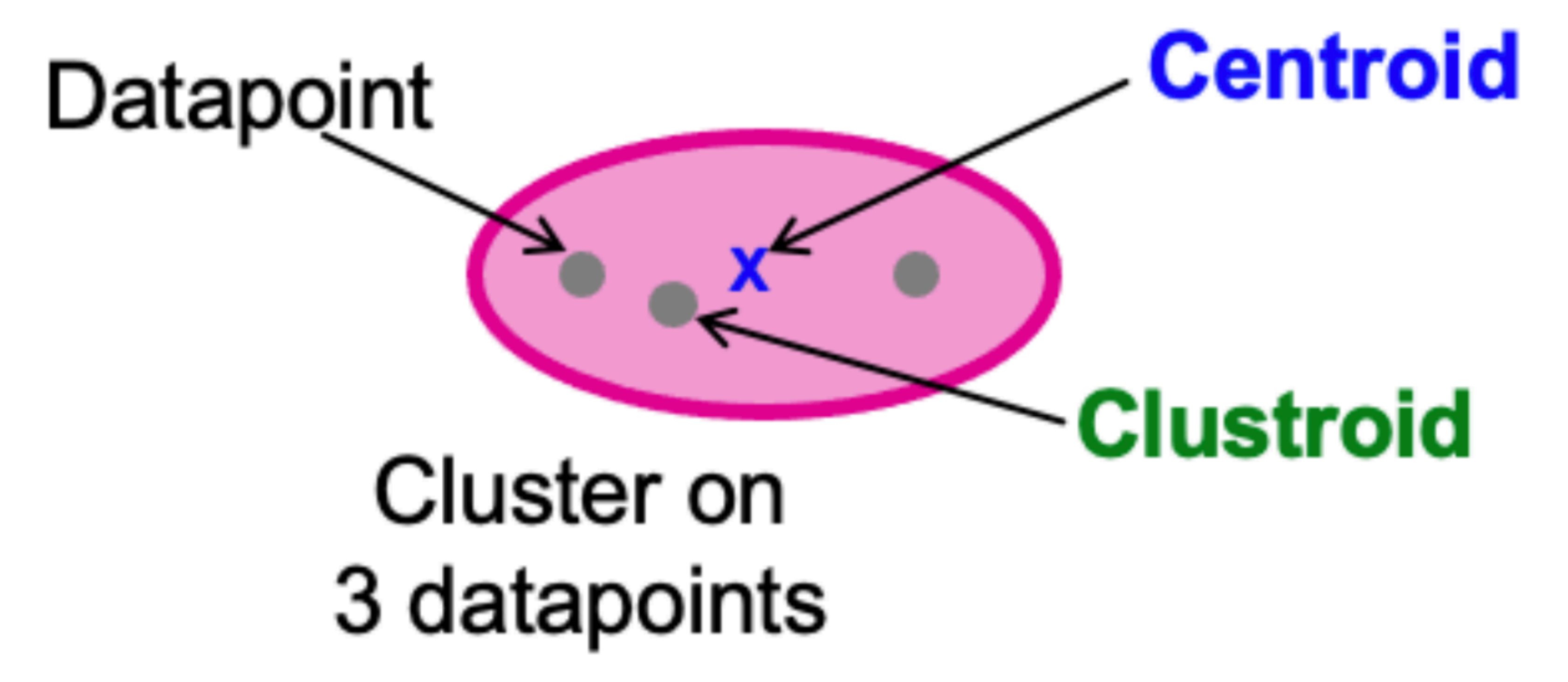
Graph Indexes



- Main ideas:
  - Build graphs with polylog(n) degree
  - Satisfy the "relative neighbor" property (RNGs):
  - Points p, q connected by an edge if there does not exist a third point r that is closer to both p, q than they are to each other

FAISS Index: k-Means Bucketing + Graph over Centroids

- k-Means clustering partitions the data into k convex clusters

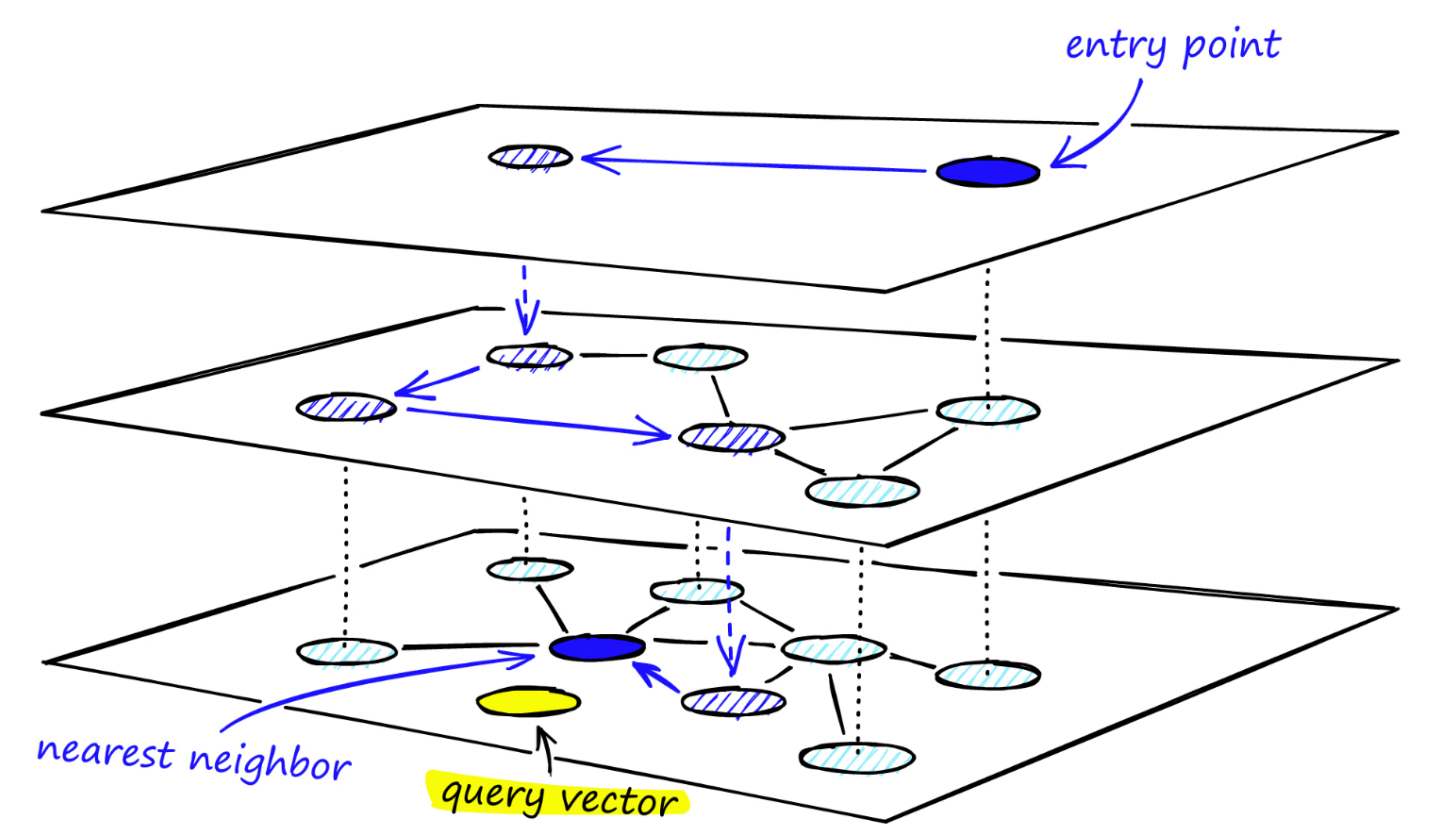


- Idea: run k-means with reasonably large k (e.g., on an $n = 1e9$ point dataset, we might use $k = 1e6$)

FAISS Index: k-Means Bucketing + Graph over Centroids

- Such a large value of k creates an interesting routing problem—given a query q, which buckets (clusters) should we probe?
- Idea: just build another ANN index over the centroids. In this case, a graph index (e.g., HNSW or DiskANN)

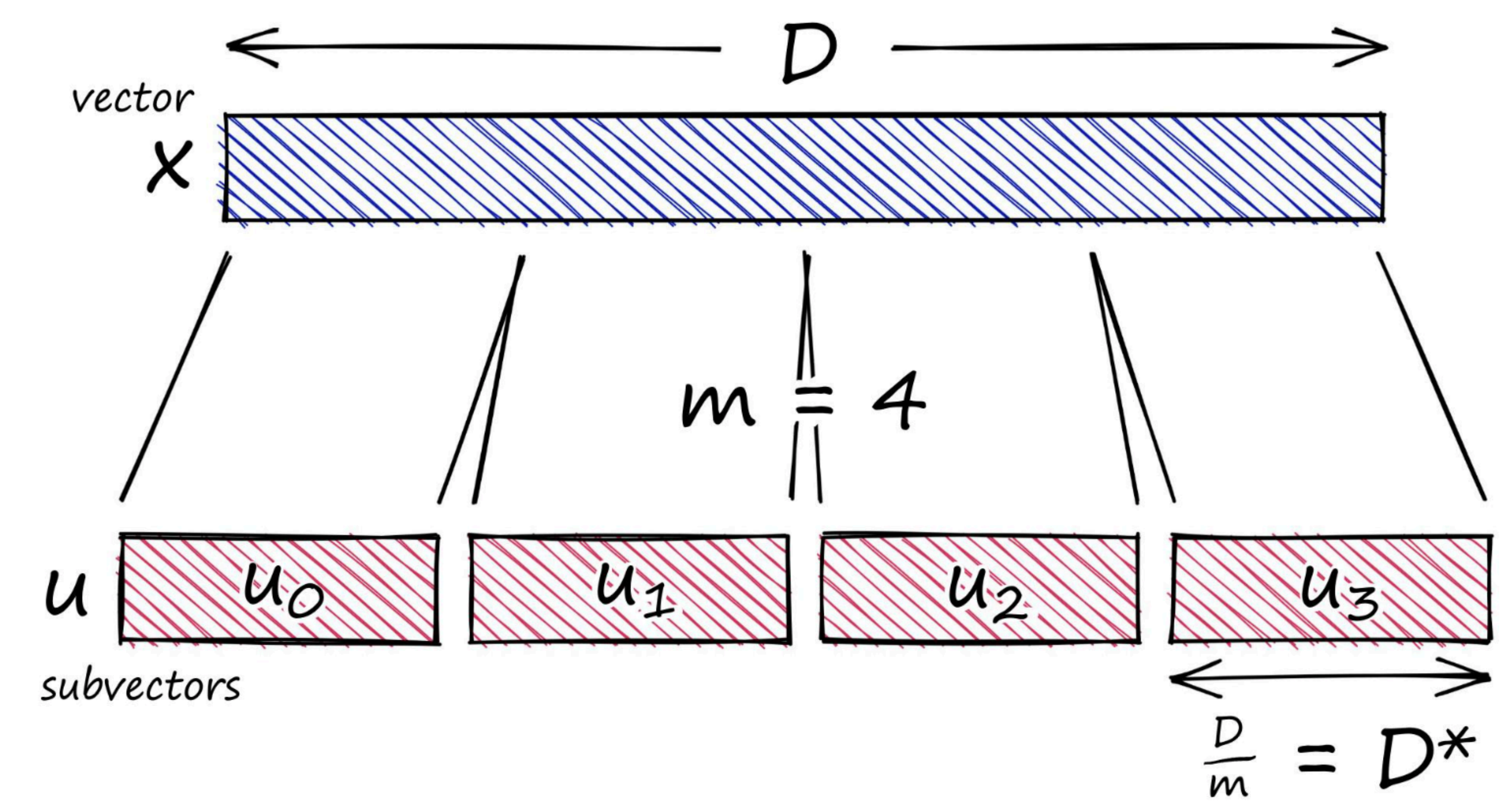
- In practice, we will figure out the k' closest centroids to the query and probe the clusters for these centroids



k-Means for Product Quantization

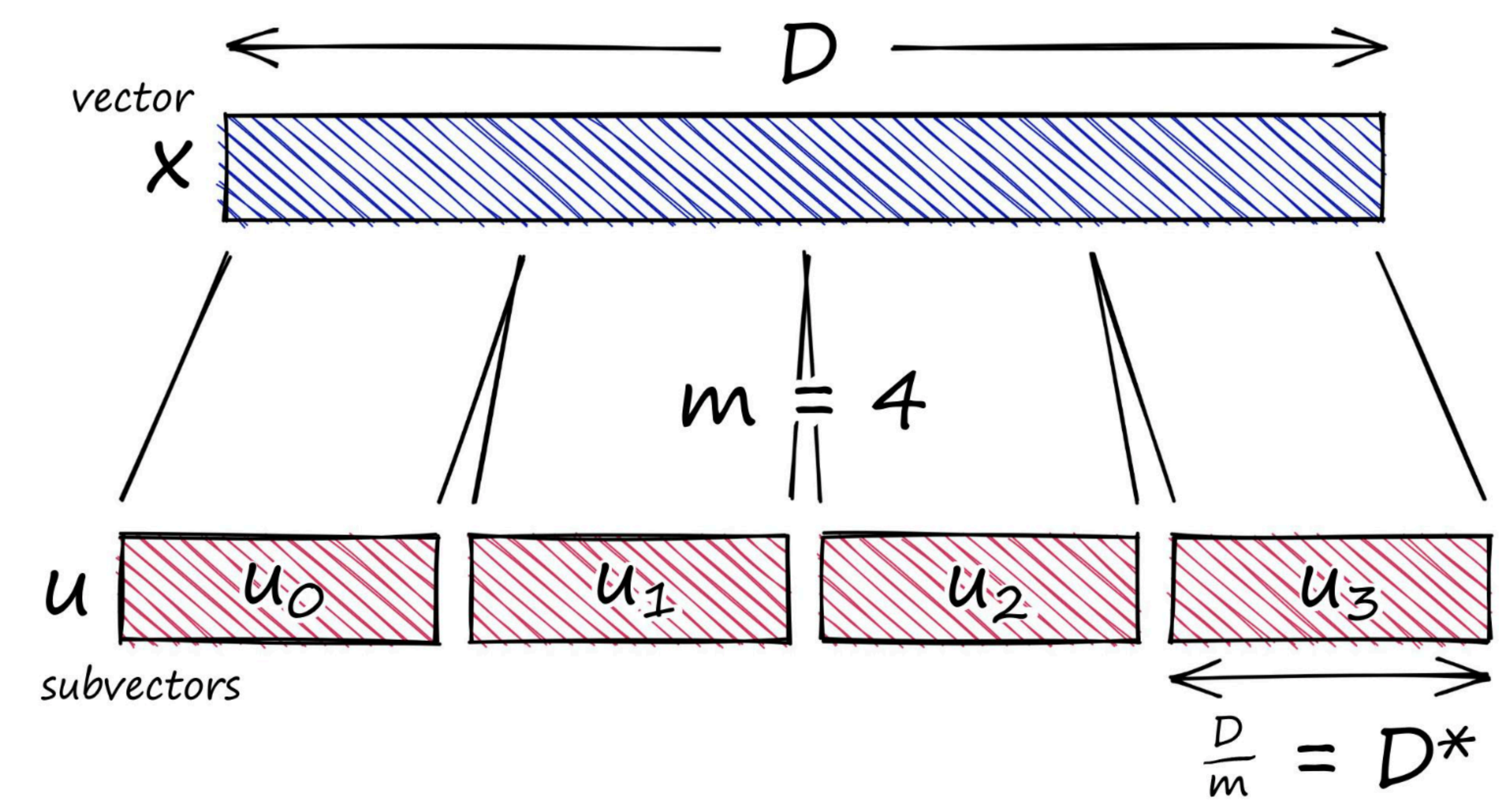
- Vectors in modern applications are large
  - Recent OpenAI text embeddings have ~1600 dimensions. Used to be 8 times larger until recently
  - More dimensions useful in applications, but costly to store and search

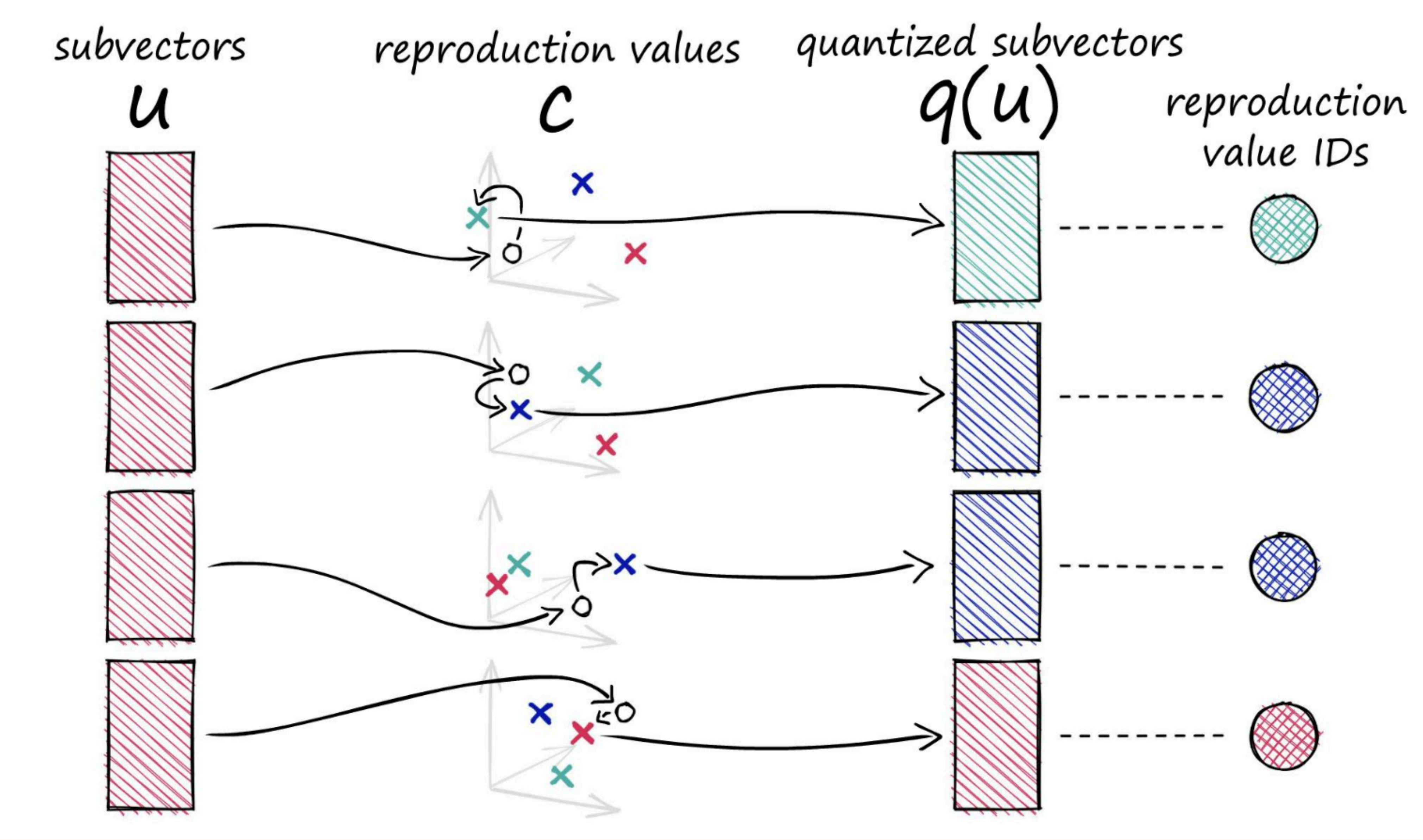
- PQ: main idea
  - $D \rightarrow D*$ dimensions
  - Reduce range of each dimension
    - i.e., use uint8 instead of float



k-Means for Product Quantization

- PQ: main idea
  - $D \rightarrow D^*$ dimensions
  - Reduce range of each dimension
    - i.e., use uint8 instead of float

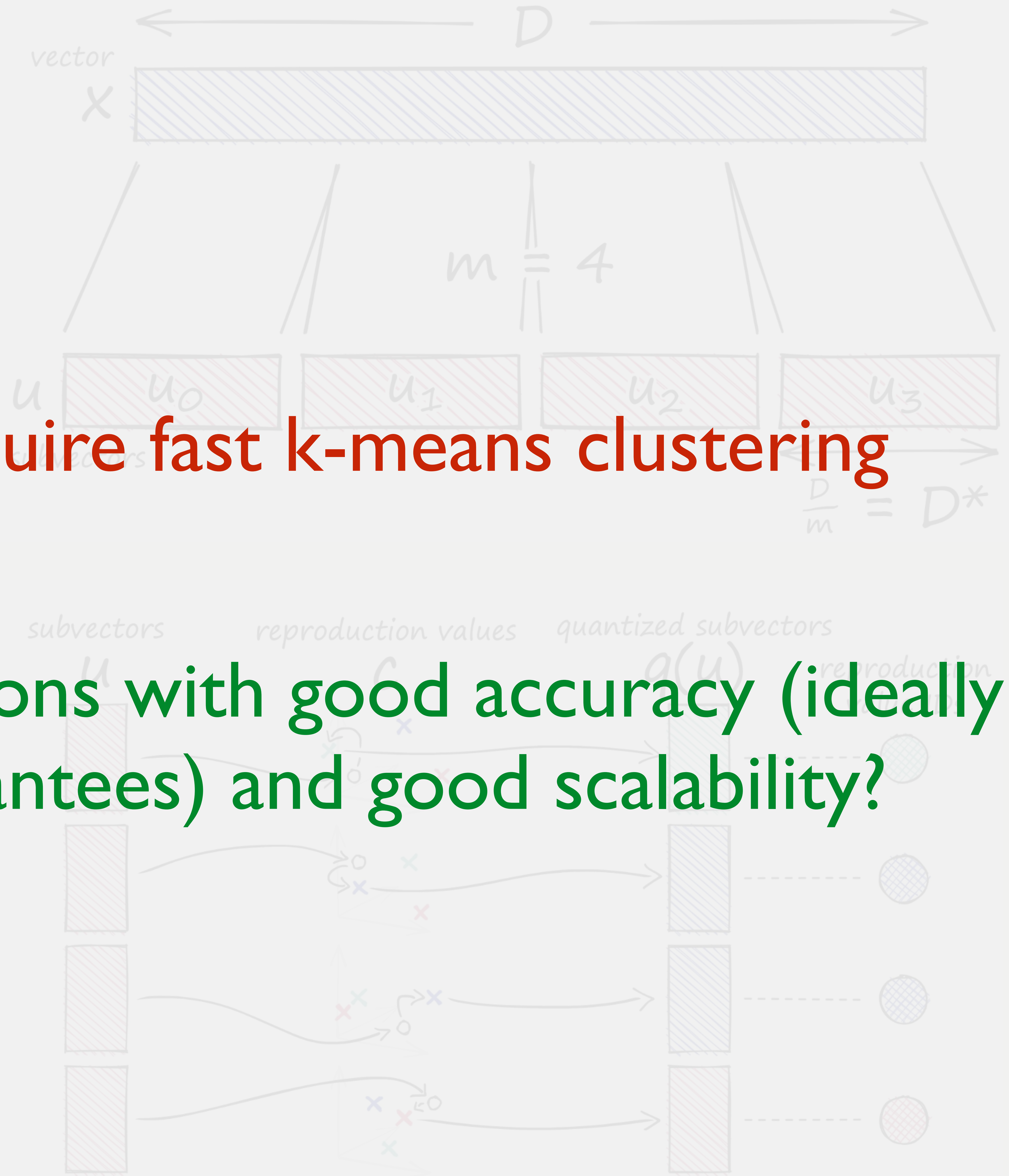




- Range reduction works by using the id of a centroid (say one of $2^8 = 256$ centroids)
- Original point can be approximated by remembering the position of the centroid

k-Means for Product Quantization

- PQ: main idea
  - $D \rightarrow D^*$ dimensions
  - Reduce range of each dimension
    - i.e., use uint8 instead of float
- Range reduction works by using the idea of a centroid (say one of $2^8 = 256$ centroids)
- Original point can be approximated by remembering the position of the centroid



All of these applications require fast k-means clustering

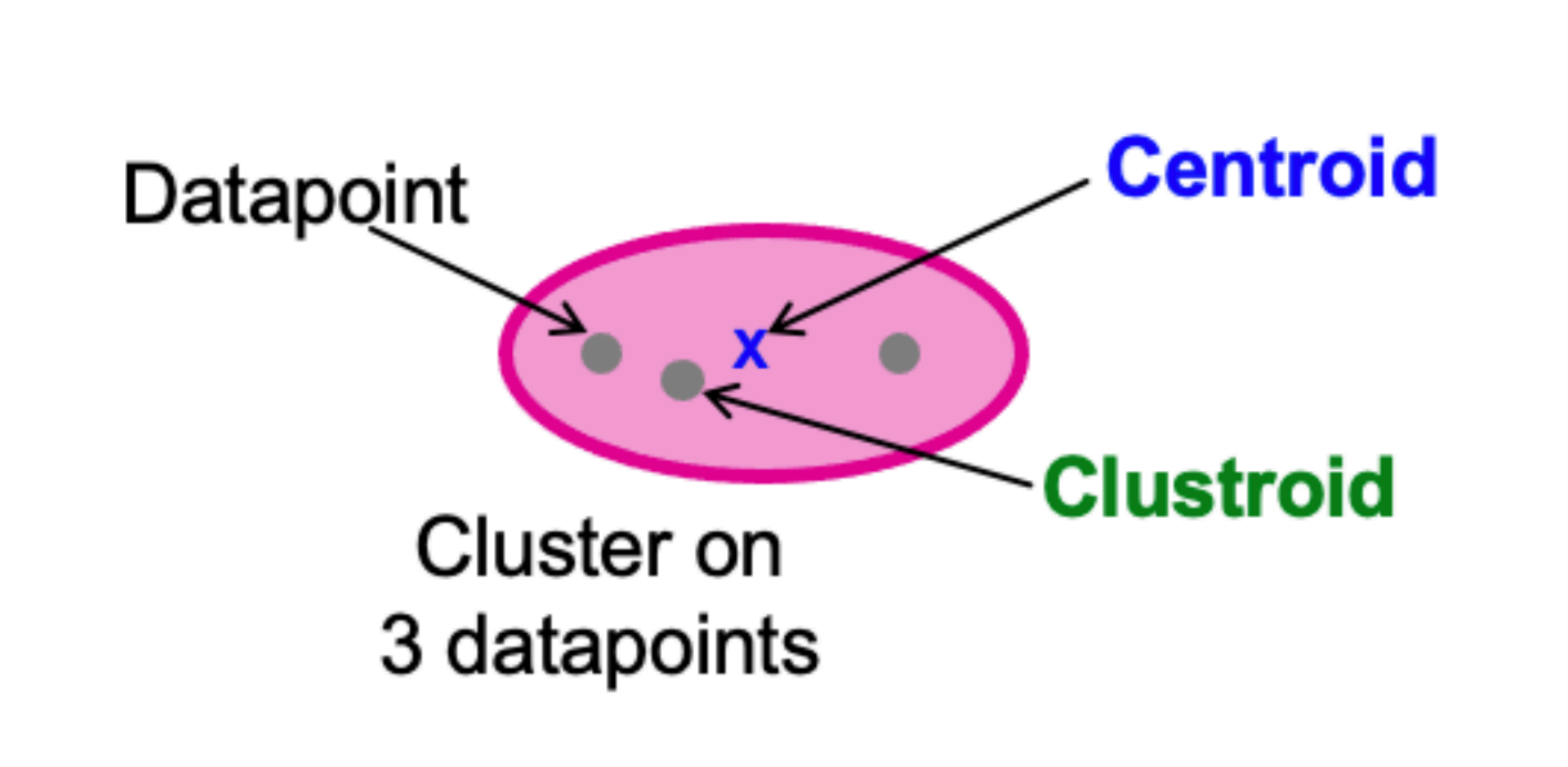
Can we build fast implementations with good accuracy (ideally with some theoretical guarantees) and good scalability?

Our plan: implement a variety of k-means baselines

- k-means objective: partition input points into k clusters $C_1, \ldots, C_k$ minimizing:

$$\sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

$$\mu_i = \text{mean}(C_i) = \frac{1}{|C_i|} \sum_{x \in C_i} x$$



- Related to the idea of minimizing the variance of a cluster (also called "Sum-of-Squared Deviations")

Lloyd's Algorithm

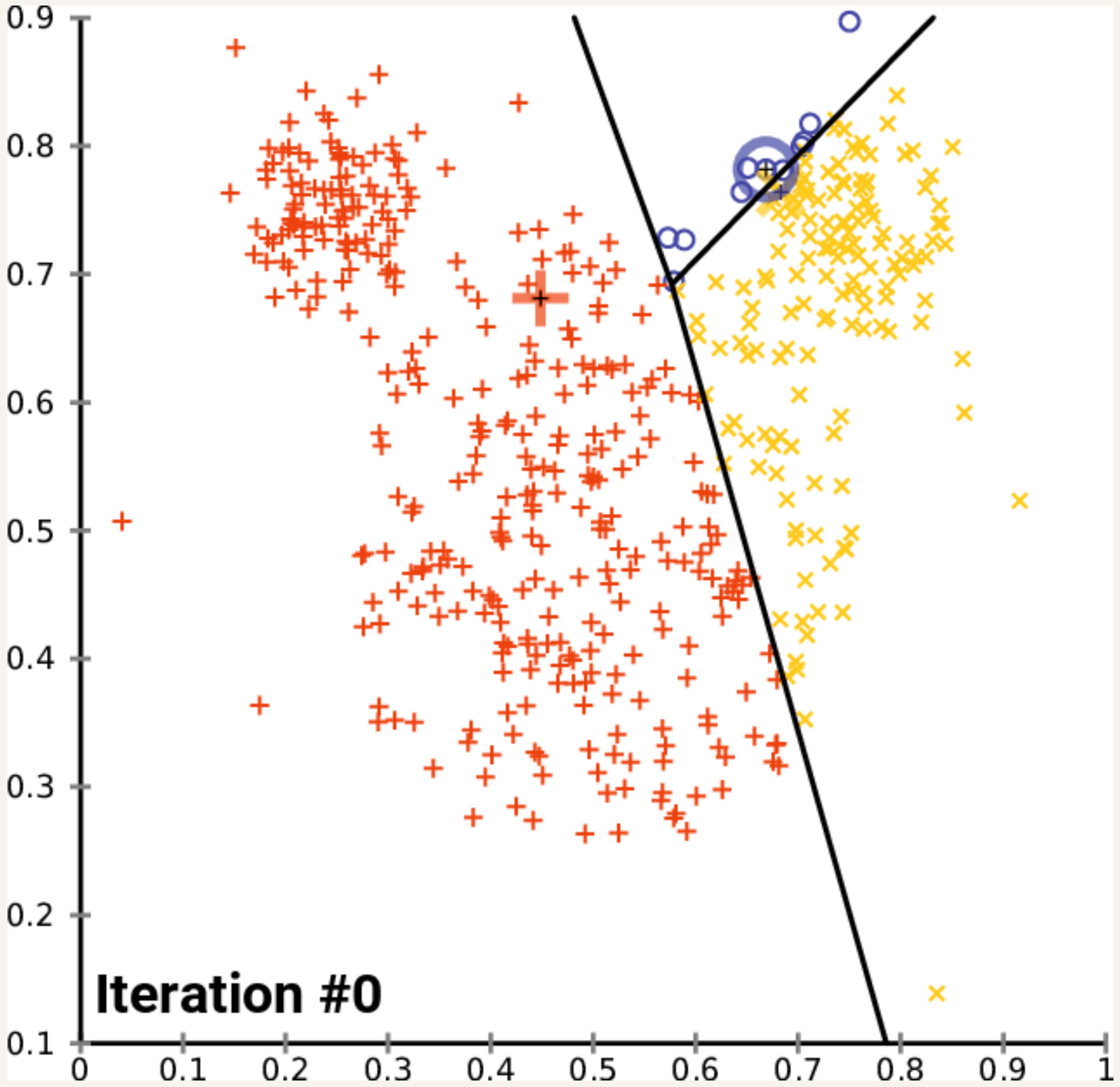
- Lloyd's algorithm (baseline)
- Consists of two steps. Suppose some initial centers $c_1, \ldots, c_k$ given:

(1) Assignment:

  - assign each $p \in P$ to the cluster corresponding to its nearest center

(2) Update:

  - recompute c_i based on the set of points assigned to C_i



Lloyd's Algorithm

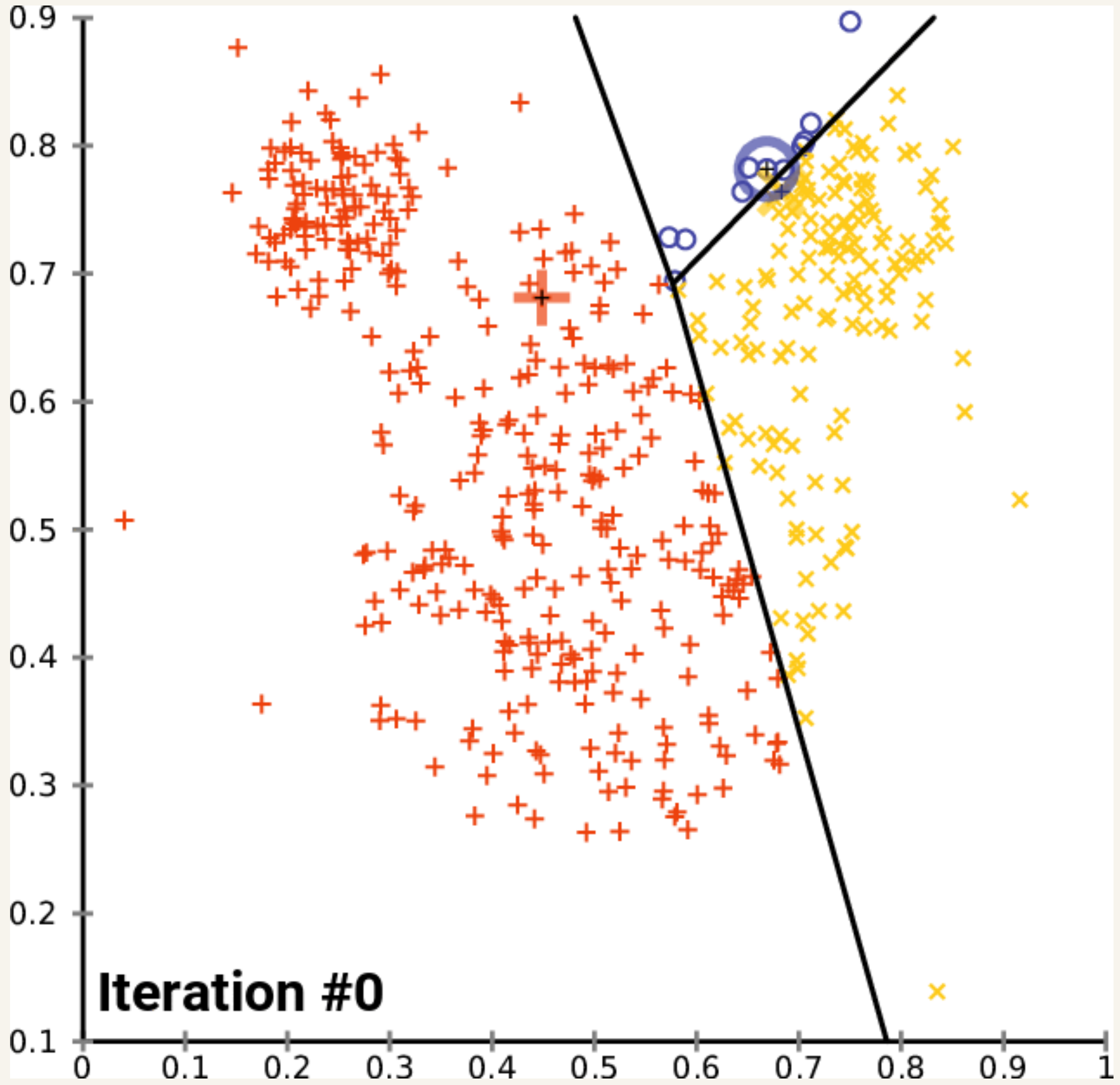
- Lloyd's algorithm (baseline)
- Consists of two steps. Suppose some initial centers $c_1, \ldots, c_k$ given:

(1) Assignment:

  - assign each $p \in P$ to the cluster corresponding to its nearest center

(2) Update:

  - recompute c_i based on the set of points assigned to C_i



Lloyd's Algorithm

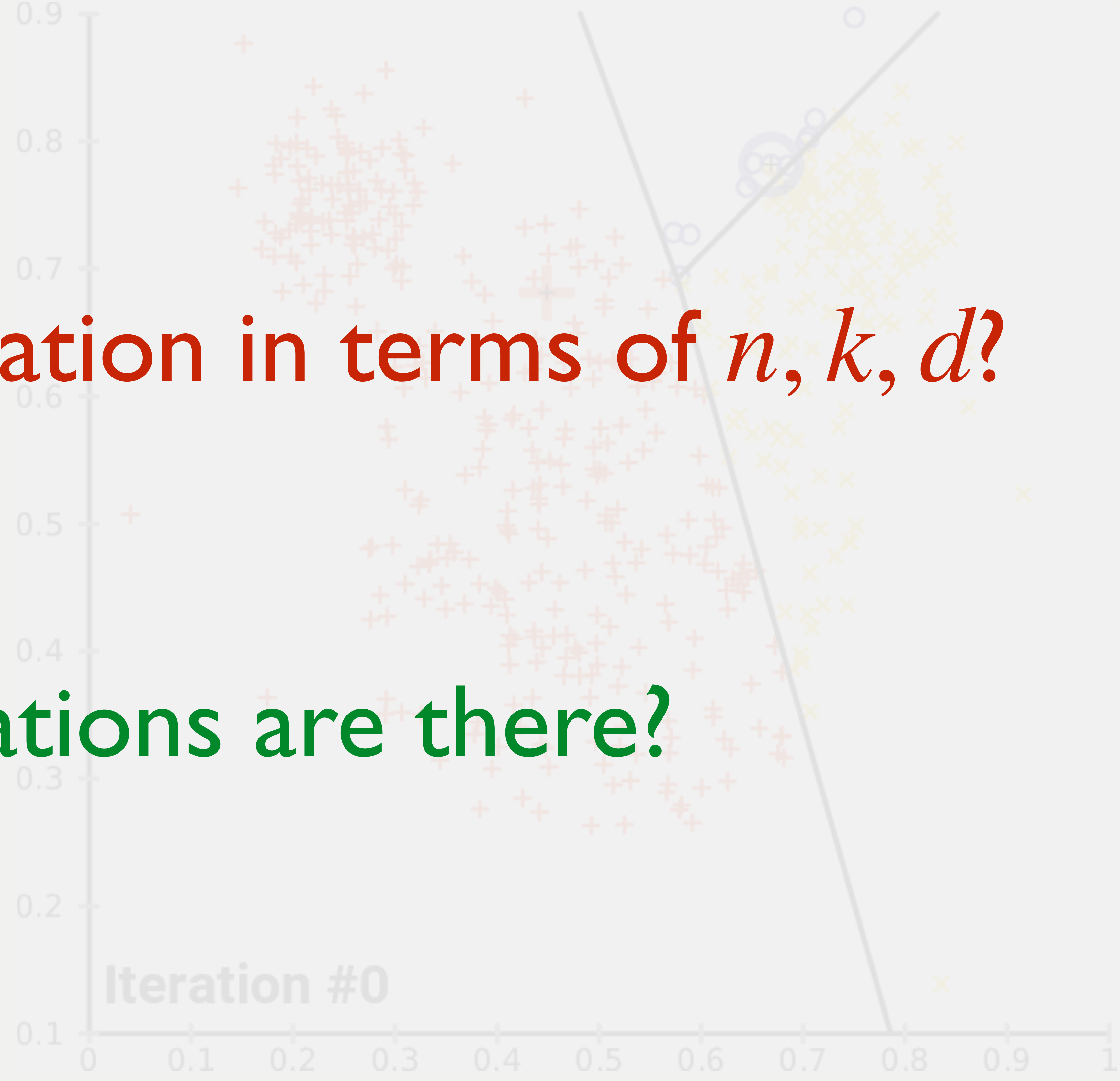
- Lloyd's algorithm (baseline)
- Consists of two steps. Suppose some initial centers $c_1, \ldots, c_k$ given:

(1) Assignment:

  - assign each $p \in P$ to the cluster corresponding to its nearest center

(2) Update:

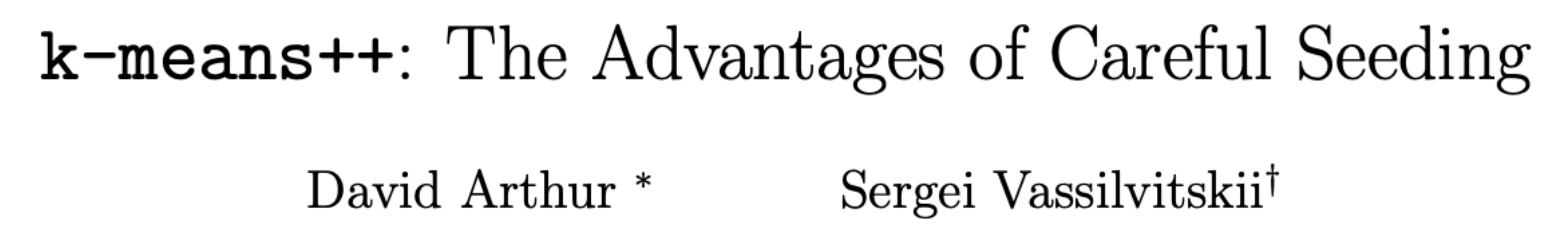
  - recompute c_i based on the set of points assigned to C_i



What is the cost of one Lloyd's iteration in terms of n, k, d?

What potential for optimizations are there?

Better initialization: k-means++

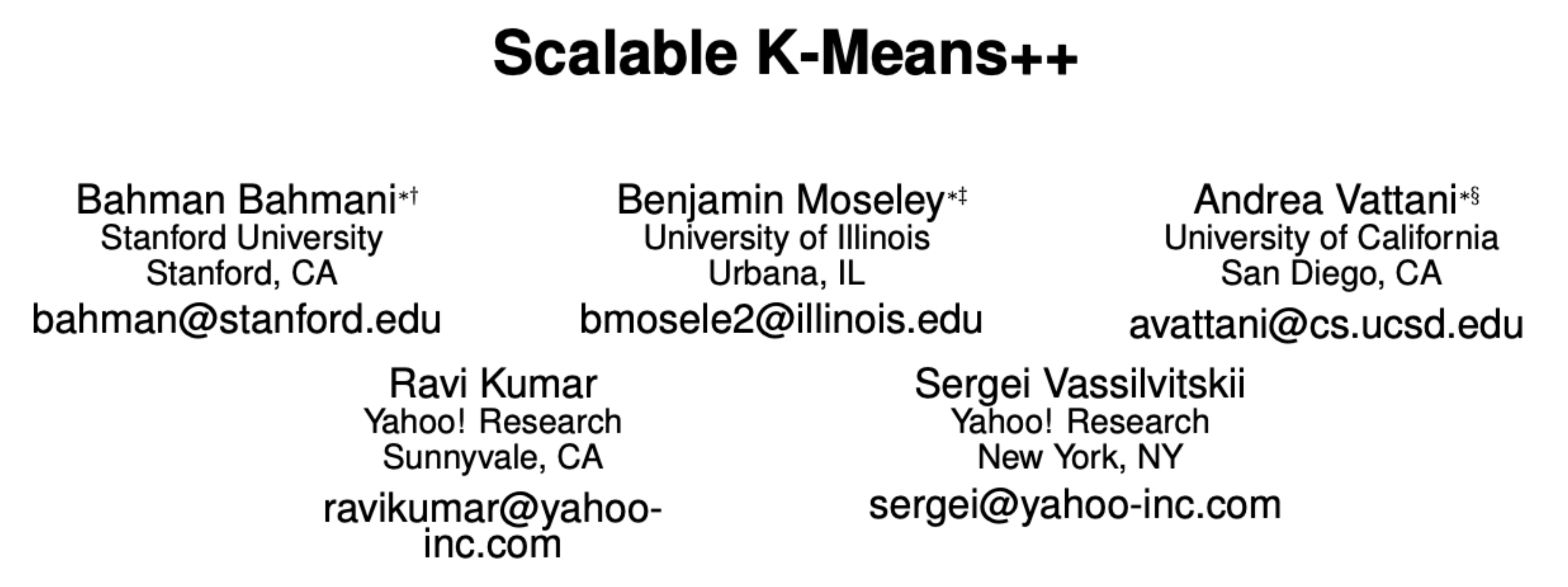

`k-means++`: The Advantages of Careful Seeding

David Arthur [*] Sergei Vassilvitskii[†]

- Instead of picking k random centers initially:
  - Pick one center uniformly at random
  - For each point p not yet elected as a center, compute $D(x)$, the distance between p and its nearest center
  - Sample an unchosen point to be chosen as the center where points are sampled with probability proportional to $D(x)^2$

- Amazingly, can show that the centers that result from this procedure are an $O(\log n)$ approximation of OPT (in expectation)

Scalable initialization: k-means||



Scalable K-Means++

Bahman Bahmani[*†]
Stanford University
Stanford, CA
bahman@stanford.edu

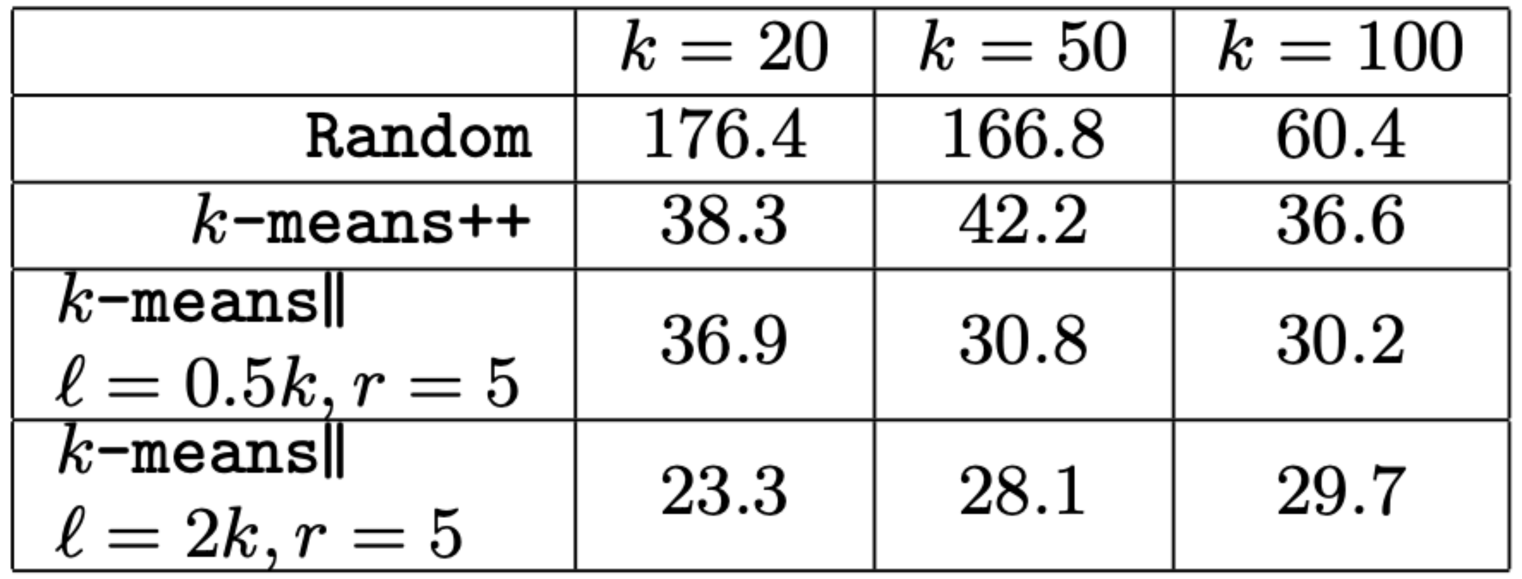
Benjamin Moseley[*‡]
University of Illinois
Urbana, IL
bmosele2@illinois.edu

Andrea Vattani[*§]
University of California
San Diego, CA
avattani@cs.ucsd.edu

Ravi Kumar
Yahoo! Research
Sunnyvale, CA
ravikumar@yahoo-inc.com

Sergei Vassilvitskii
Yahoo! Research
New York, NY
sergei@yahoo-inc.com

- A slightly more complex scheme, but admits more parallelism:
    - Sample $O(k)$ points in each round
    - Repeat for approximately $O(\log n)$ rounds
    - Yields $O(k \log n)$ points that are then reclustered into k initial centers
- Theory: initial $O(k \log n)$ centers give a constant factor approximation of OPT

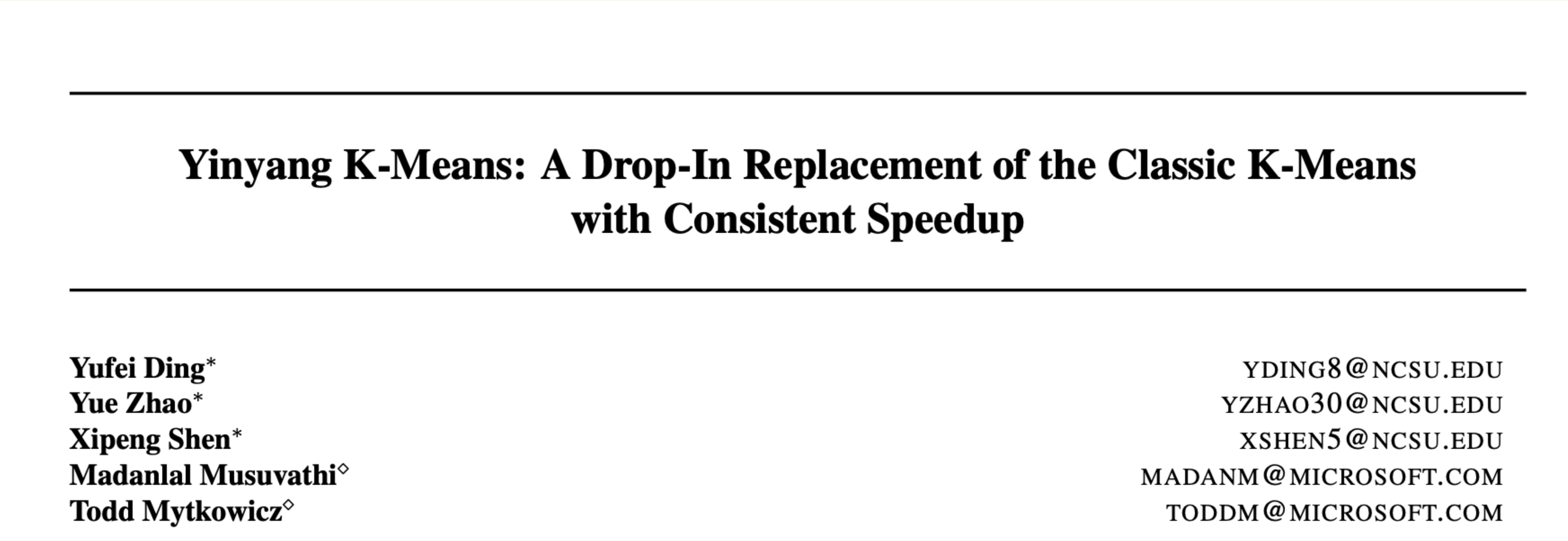


	$k = 20$	$k = 50$	$k = 100$
`Random`	176.4	166.8	60.4
k-`means++`	38.3	42.2	36.6
k-`means`$\|$ $\ell = 0.5k, r = 5$	36.9	30.8	30.2
k-`means`$\|$ $\ell = 2k, r = 5$	23.3	28.1	29.7

Table 6: Number of Lloyd's iterations till convergence (averaged over 10 runs) for Spam.

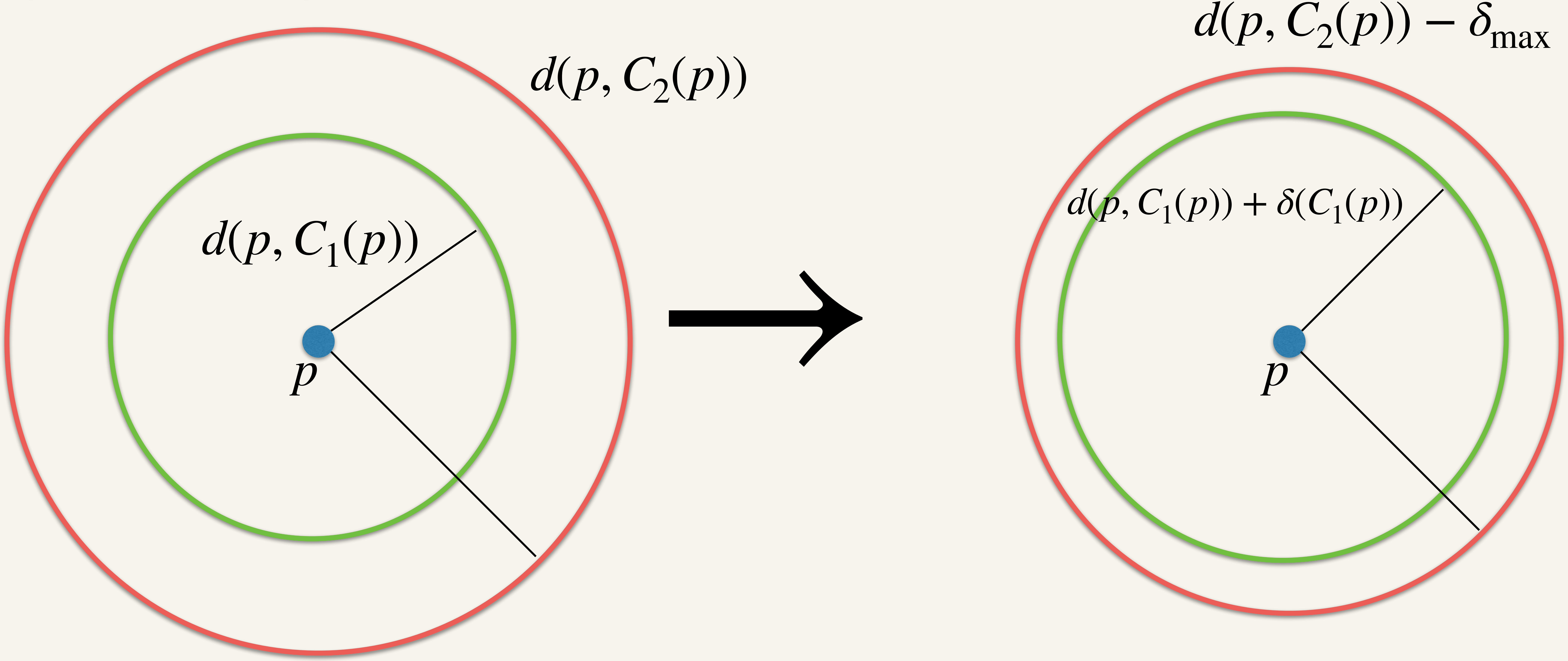
Avoiding distance comparisons

- Costly part of Lloyd iteration is comparing each point p with all k centers (costs $O(nkd)$)
- Idea: use triangle inequality to avoid distance computations for points



Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup

Yufei Ding*	YDING8@NCSU.EDU
Yue Zhao*	YZHAO30@NCSU.EDU
Xipeng Shen*	XSHEN5@NCSU.EDU
Madanlal Musuvathi◇	MADANM@MICROSOFT.COM
Todd Mytkowicz◇	TODDM@MICROSOFT.COM



Project plan:

- Build a highly optimized shared-memory library of k-means implementations
- Evaluate existing algorithms for large n, k, d:
  - $n = 1B$ points
  - $k = 1M$ centers
  - $d \in [100, 1600]$

Evaluate performance on real-world embedding datasets from ANN search applications

(Hopefully) design new algorithms and heuristics to obtain scalability improvements at billion-scale!

Thank you!

Low-Outdegree Orientation

Graph Clustering

Aspen

Julienne

Graph Compression

Dynamic Connectivity

Streaming Graph Processing

GBBS

Euler Tour Trees

Batch-Dynamic Graph Processing

Static Graph Processing

RC-Trees

Sage

Semi-Asymmetric Model

Massively Parallel Computation

Shared-Memory Model

ConnectIt

Adaptive Massively Parallel Computation

Graph-Based HAC

Connectivity and Related Problems

Theoretically-Efficient Parallel Graph Algorithms

laxman@umd.edu