Tutorial: Large-Scale Graph Processing in Shared Memory

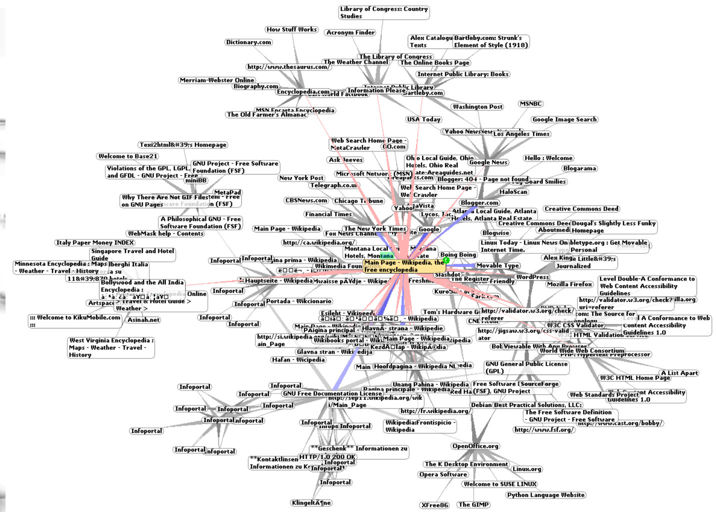
Julian Shun (University of California, Berkeley)
Laxman Dhulipala (Carnegie Mellon University)
Guy Blelloch (Carnegie Mellon University)

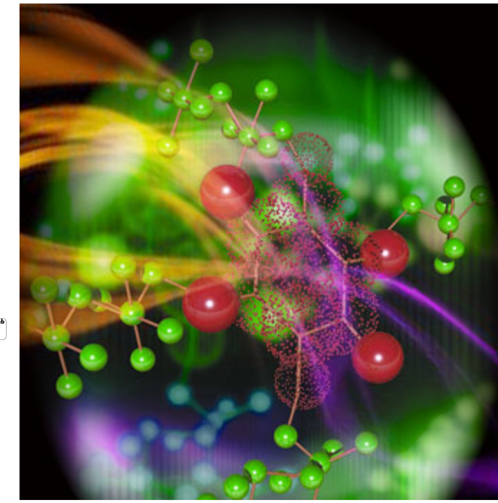
Tutorial Agenda

- 2:00-3:30pm
  - Overview of graph processing and Ligra
  - Walk through installation
  - Do examples in Ligra
- 3:30-4:00pm
  - Break
- 4:00-5:30pm
  - Implementation details of Ligra
  - Overview of other graph processing systems
  - Exercises

Slides available at https://github.com/jshun/ligra/tree/master/tutorial/tutorial.pdf

Graphs are everywhere!





- Can contain billions of vertices and edges!

YAHOO! — 6.6 billion edges

Common Crawl — 128 billion edges

facebook — ~1 trillion edges [VLDB 2015]

Graph processing challenges

- Many random memory accesses, poor locality
- Relatively high communication-to-computation ratio
- Varying parallelism throughout execution
- Race conditions, load balancing

- Need to efficiently analyze large graphs
  - Running time efficiency
  - Space efficiency
  - Programming efficiency

Ligra Graph Processing Framework

EdgeMap

VertexMap

Breadth-first search
Betweenness centrality
Connected components
K-core decomposition
Belief propagation
Maximal independent set
…

Single-source shortest paths
Eccentricity estimation
(Personalized) PageRank
Local graph clustering
Biconnected components
Collaborative filtering
…

Simplicity, Performance, Scalability

Graph Processing Systems

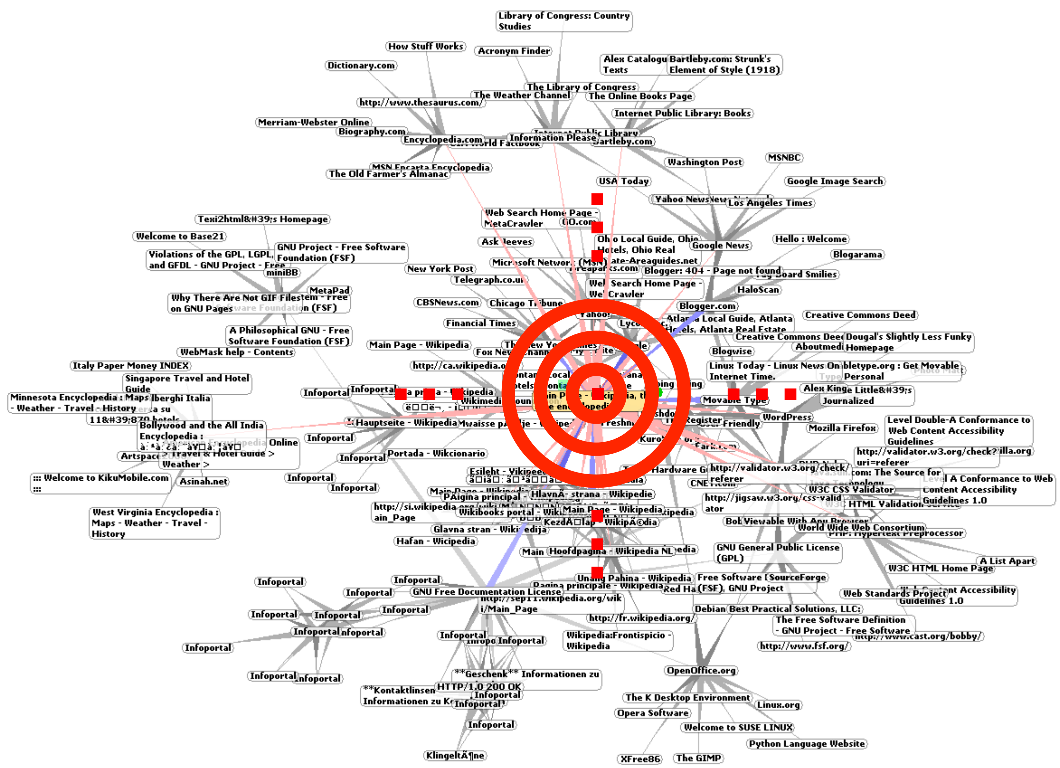
- Existing (at the time Ligra was developed): Pregel/Giraph/GPS, GraphLab, Pegasus, Knowledge Discovery Toolbox, GraphChi, and many others…
- Our system: Ligra - Lightweight graph processing system for shared memory

Takes advantage of "frontier-based" nature of many algorithms (active set is dynamic and often small)

Breadth-first Search (BFS)

- Compute a BFS tree rooted at source *r* containing all vertices reachable from *r*

Applications
Betweenness centrality
Eccentricity estimation
Maximum flow
Web crawlers
Network broadcasting
Cycle detection
…



- Can process each level in parallel
- Race conditions, load balancing

Steps for Graph Traversal

Many graph traversal algorithms do this!

Graph

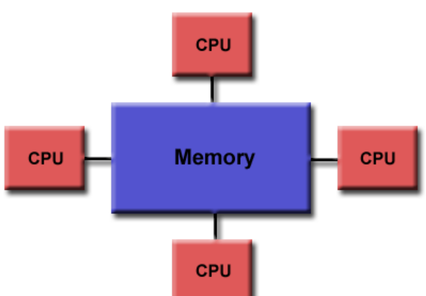
- Operate on a subset of vertices — VertexSubset
- Map computation over subset of edges in parallel
- Return new subset of vertices

} EdgeMap

- Map computation over subset of vertices in parallel — VertexMap

We built the Ligra abstraction for these kinds of computations

Think with flat data-parallel operators



Intel® Cilk™ Plus

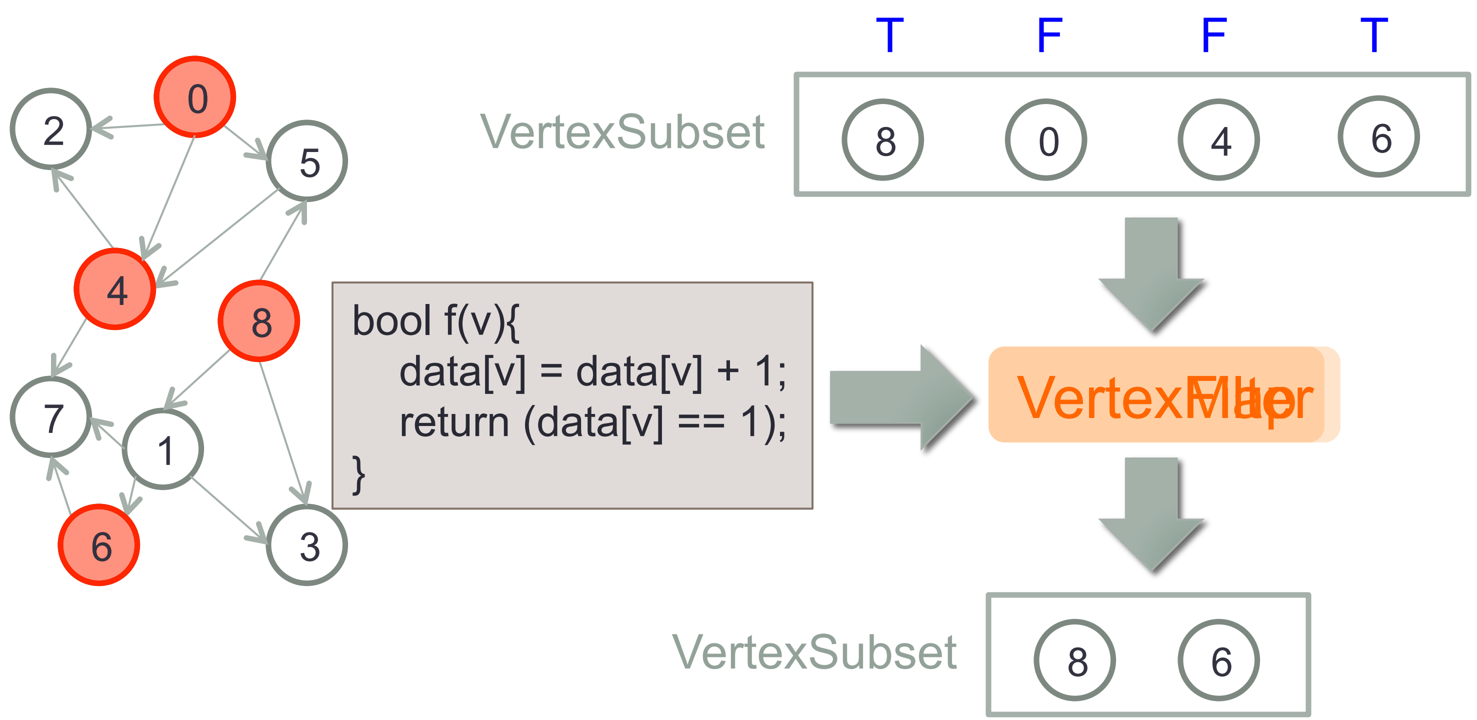
OpenMP

Shared memory parallelism

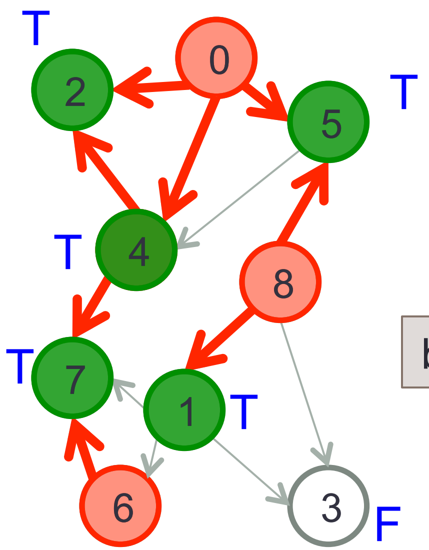
Optimizations:

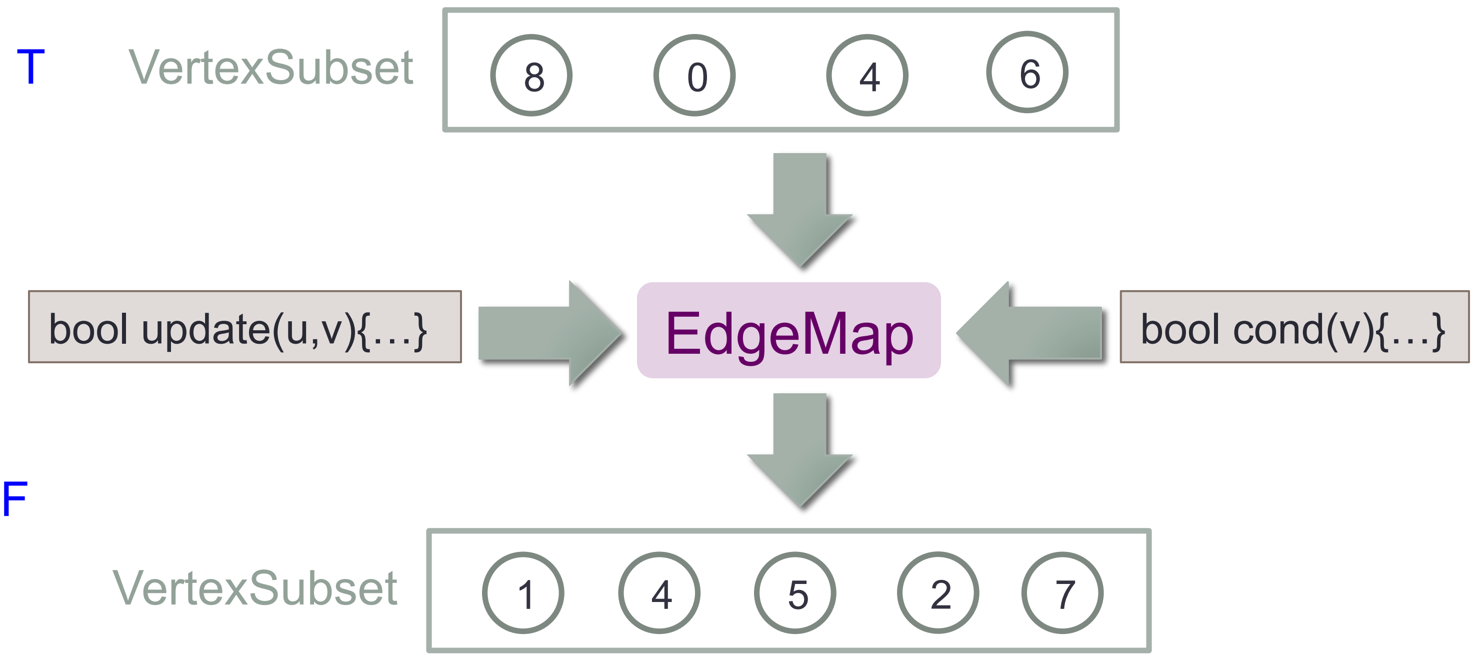
- hybrid traversal
- graph compression

Ligra Framework



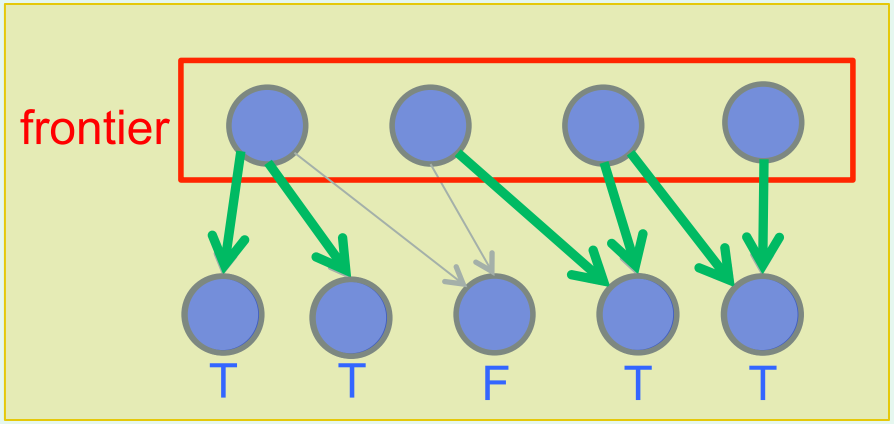
Ligra Framework





Breadth-first Search in Ligra

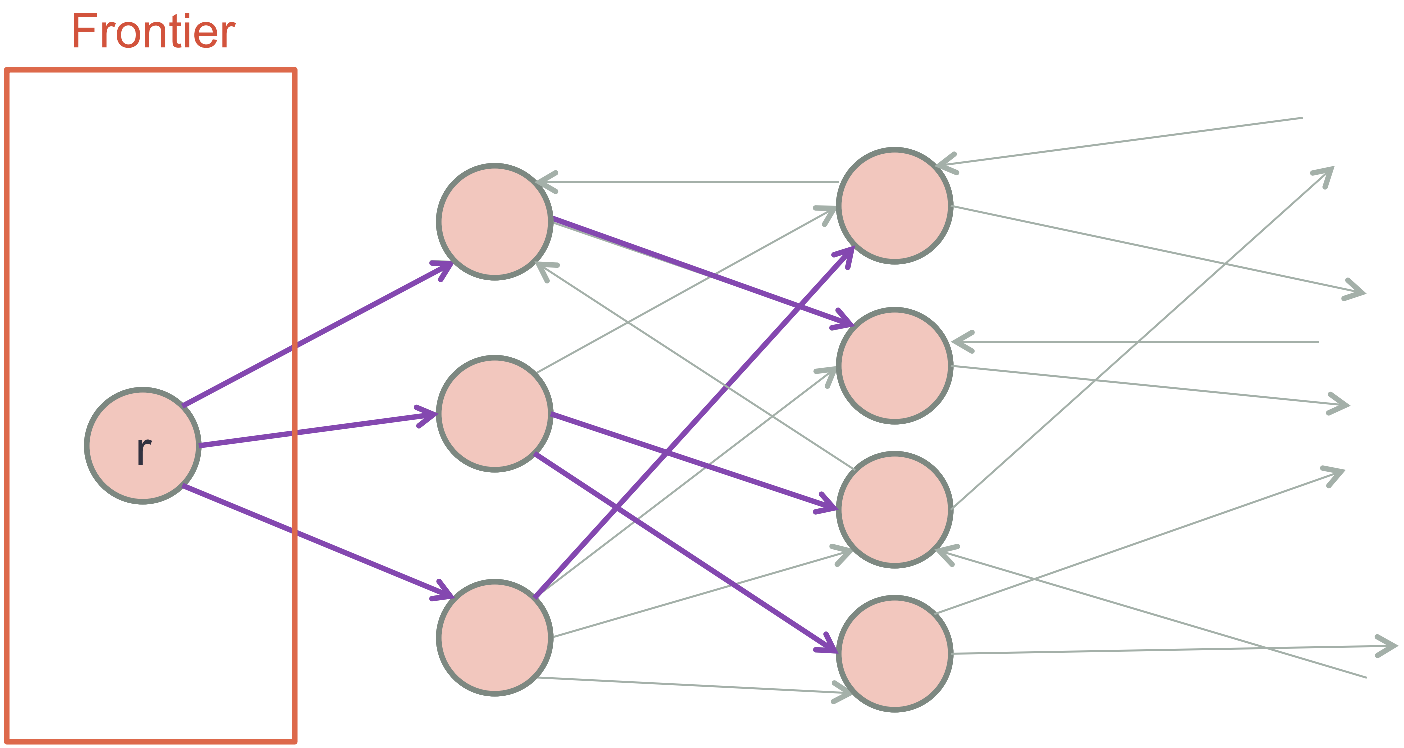
```
parents = {-1, …, -1};   //-1 indicates "unexplored"

procedure UPDATE(s, d):
    return compare_and_swap(parents[d], -1, s);

procedure COND(v):
    return parents[v] == -1;   //checks if "unexplored"

procedure BFS(G, r):
    parents[r] = r;
    frontier = {r};  //VertexSubset
    while (size(frontier) > 0):
        frontier = EDGEMAP(G, frontier, UPDATE, COND);
```



Install and code examples in Ligra

Breadth-first Search (BFS)

- Compute a BFS tree rooted at source *r* containing all vertices reachable from *r*



Connected Components

- Takes an unweighted, undirected graph G = (V, E)
- Returns a label array L such that L[v] = L[w] if and only if v is connected to w

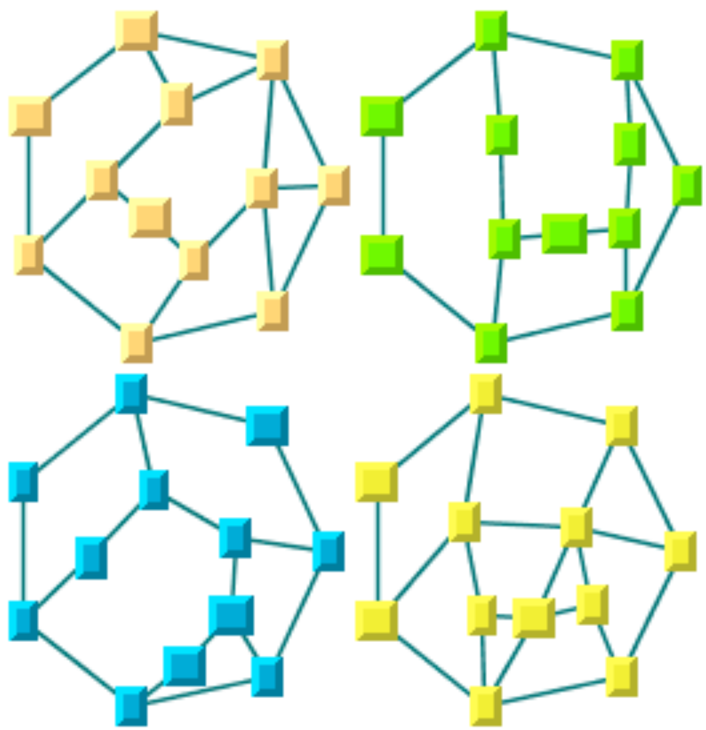
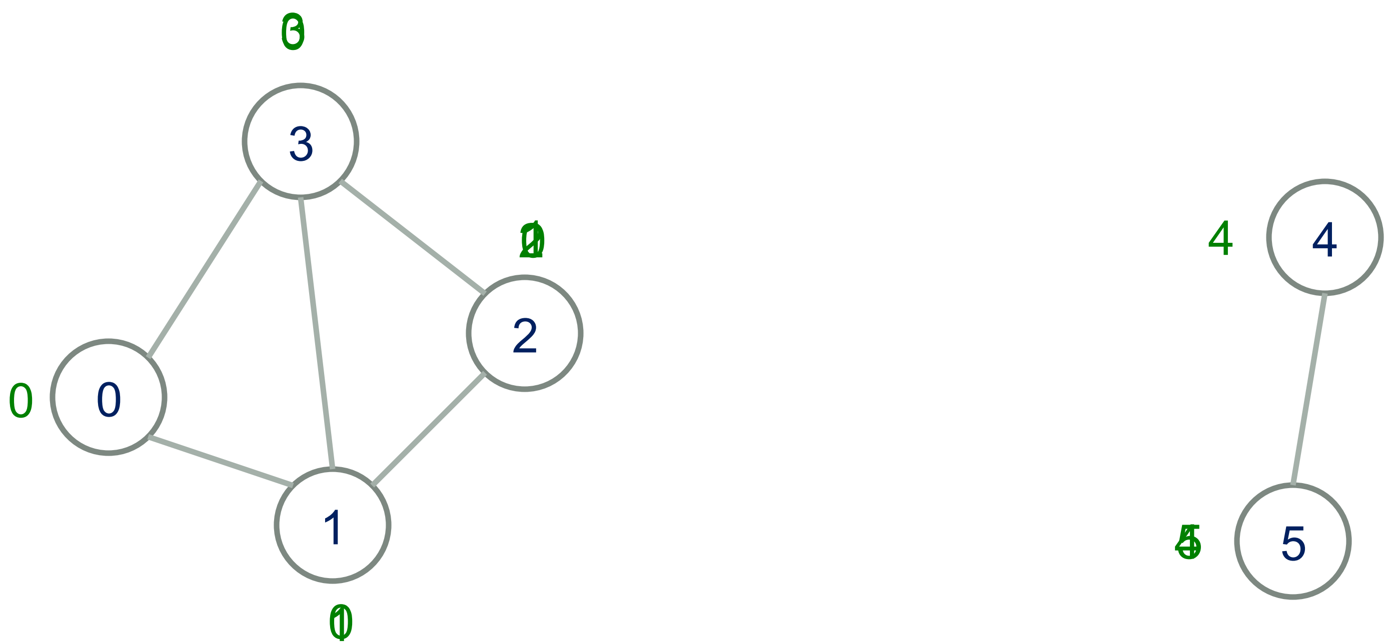


Image Source: docs.roguewave.com

Parallel Label Propagation Algorithm



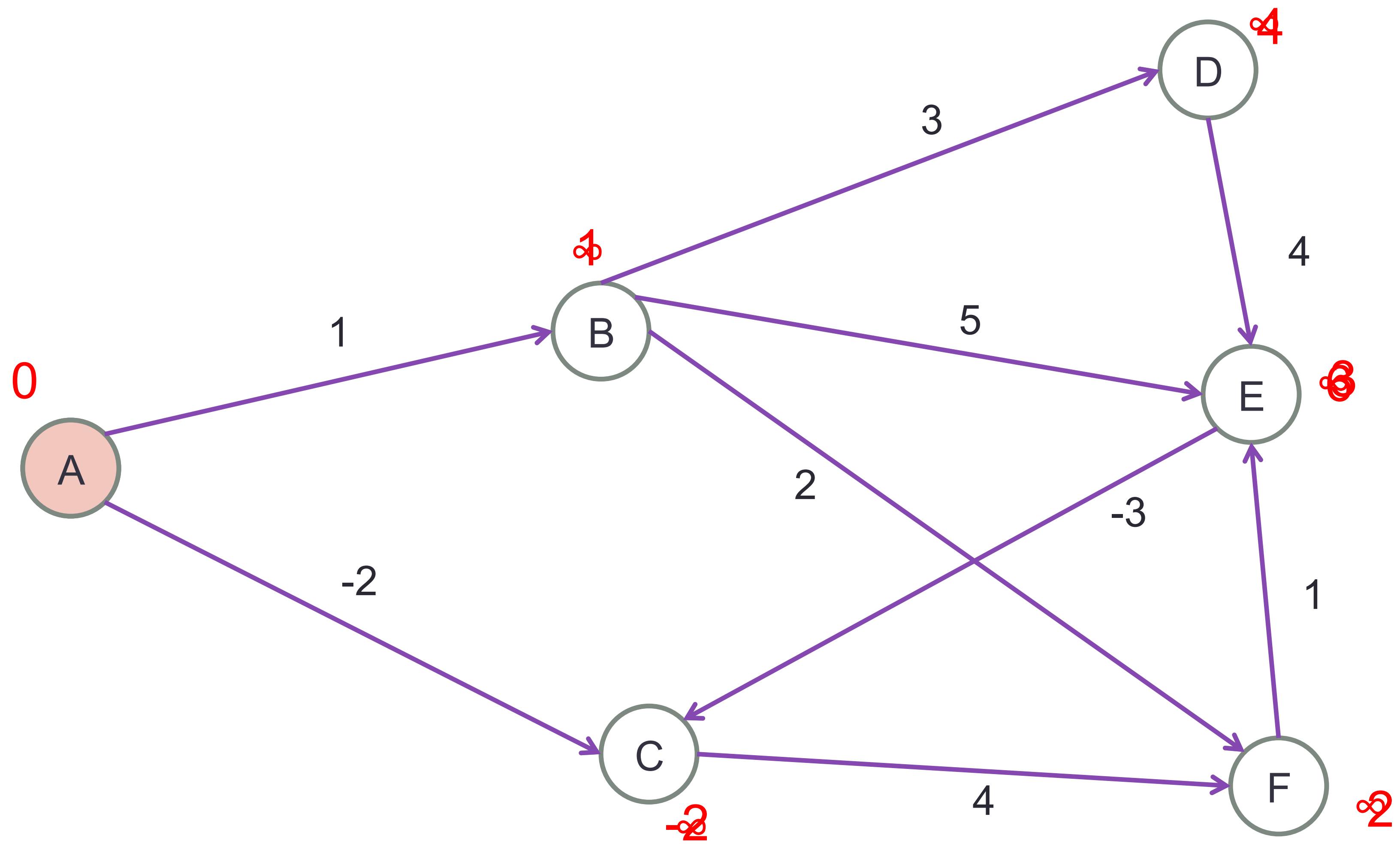
- Processing all vertices in each iteration seems wasteful
  - Optimization: only place vertices who changed on frontier

- Warning: this algorithm is only good for low-diameter graphs

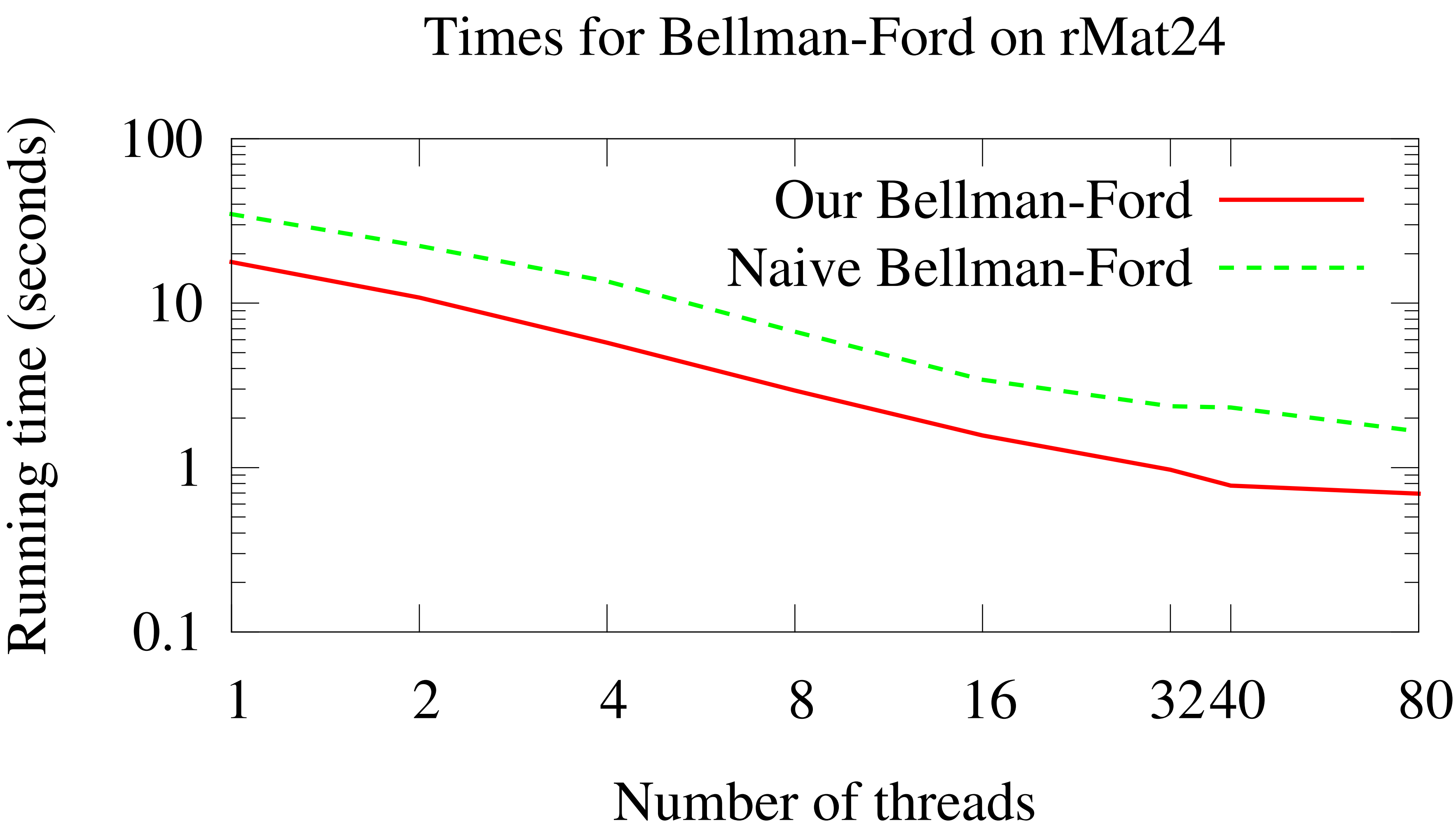
Single-Source Shortest Paths

- Takes a weighted graph G = (V, E, *w*(E)) and starting vertex $r \in V$
- Returns a shortest paths array SP where SP[*v*] stores the shortest path distance from *r* to *v* (∞ if v unreachable from *r*)

Parallel Bellman-Ford Shortest Paths



Parallel Bellman-Ford Performance



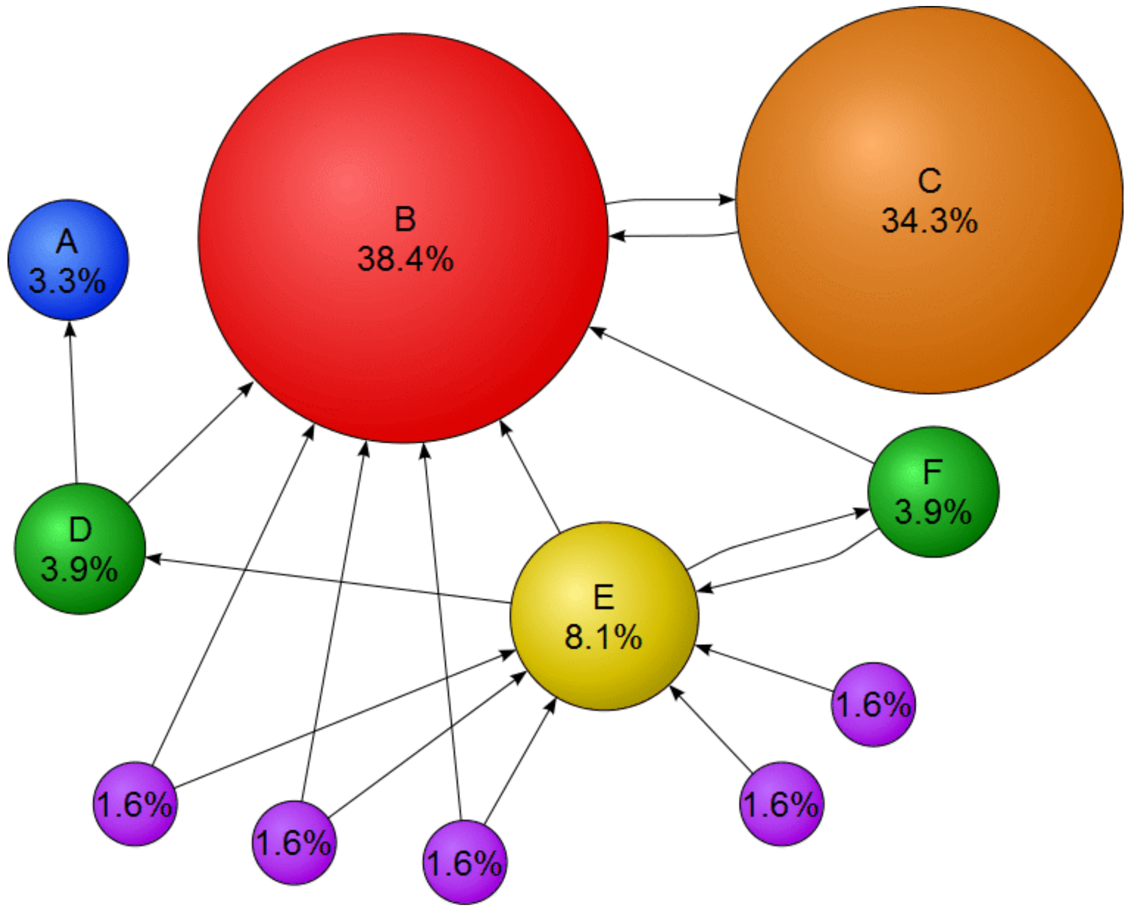
K-core decomposition

- A k-core of a graph is a maximal connected subgraph in which all vertices have degree at least k
- A vertex has core number k if it belongs to a k-core but not a (k+1)-core

- Algorithm: Takes an unweighted, undirected graph G and returns the core number of each vertex in G

```
k = 1
while(G is not empty) {
    while(there exists vertices with degree < k in G) {
            assign a core number of k-1 to all vertices with degree < k;
            remove all vertices with degree < k from G;
    }
    k = k+1;
}
```

PageRank



$$\mathrm{PR}[v] = \frac{1-\gamma}{|V|} + \gamma \sum_{u \in N^{-}(v)} \frac{\mathrm{PR}[u]}{deg^{+}(u)}$$

PageRank in Ligra

```
p_curr = {1/|V|, …, 1/|V|};          p_next = {0, …, 0};          diff = {};

procedure UPDATE(s, d):
    return atomic_increment(p_next[d], p_curr[s] / degree(s));

procedure COMPUTE(i):
    p_next[i] = α · p_next[i] + (1- α) · (1/|V|);
    diff[i] = abs(p_next[i] – p_curr[i]);
    p_curr[i] = 0;
    return 1;

procedure PageRank(G, α, ε):
    frontier = {0, …, |V|-1};
    error = ∞
    while (error > ε):
        frontier = EDGEMAP(G, frontier, UPDATE, COND_true);
        VERTEXMAP(frontier, COMPUTE);
        error = sum of diff entries;
        swap(p_curr, p_next)
    return p_curr;
```

PageRank

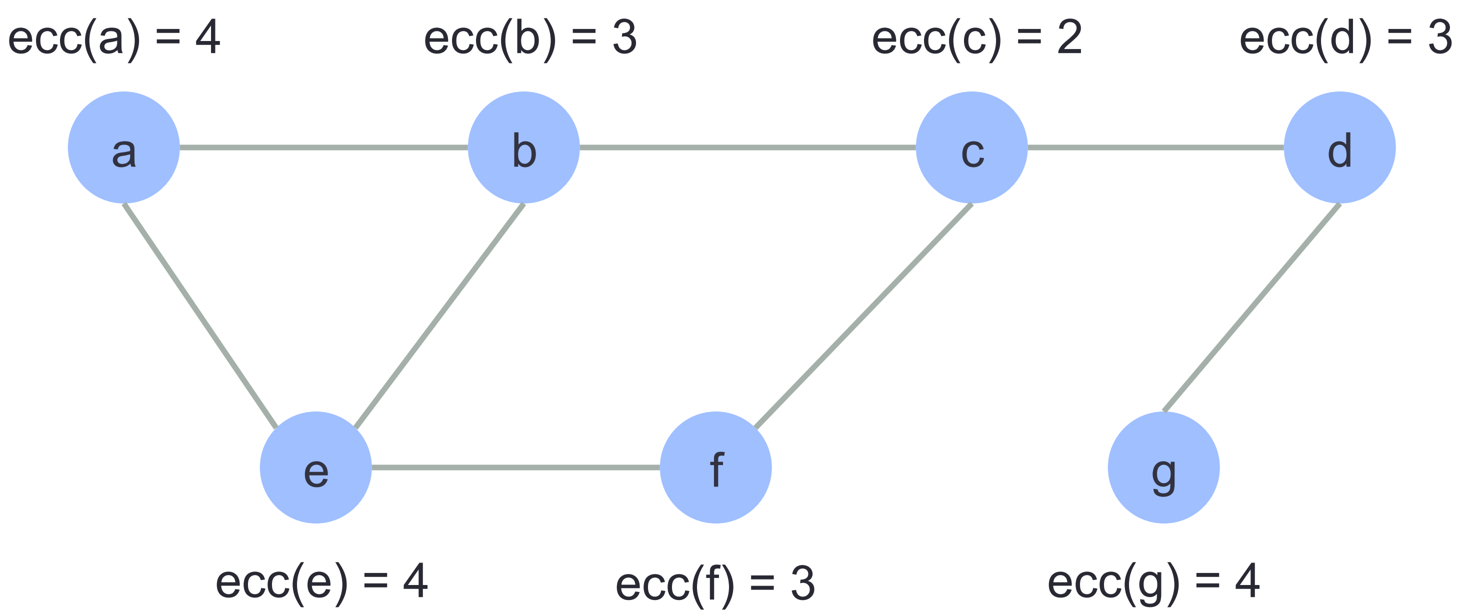
- *Sparse version?*
  - PageRank-Delta: Only update vertices whose PageRank value has changed by more than some Δ-fraction (discussed in GraphLab and McSherry WWW '05)

PageRank-Delta in Ligra

```
PR[i] = {1/|V|, …, 1/|V|};
nghSum = {0, …, 0};
Change = {};                    //store changes in PageRank values

procedure UPDATE(s, d):                //passed to EdgeMap
    return atomic_increment(nghSum[d], Change[s] / degree(s));

procedure COMPUTE(i):                  //now passed to VertexFilter
    Change[i] = α · nghSum[i];
    PR[i] = PR[i] + Change[i];
    return (abs(Change[i]) > Δ);       //check if absolute value of change is big enough

procedure PageRank(G, α, ε):
    …
        frontier = VERTEXFILTER(frontier, COMPUTE);
    …
```

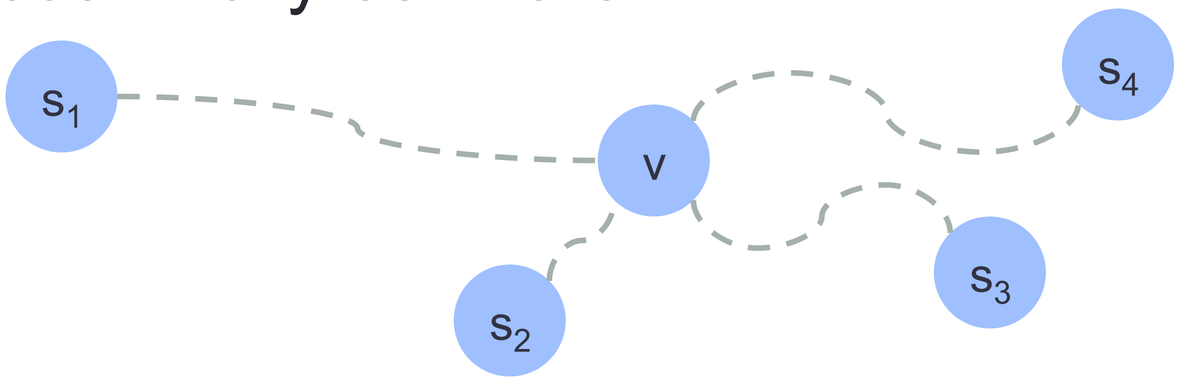
Eccentricity estimation

- Takes an unweighted, undirected graph G = (V, E)
- Returns an estimate of the eccentricity of each vertex where
- The **eccentricity** of a vertex v is the distance to furthest reachable vertex from v



Multiple BFS's

- Run multiple BFS's from a sample of random vertices and use distance from furthest sample as eccentricity estimate

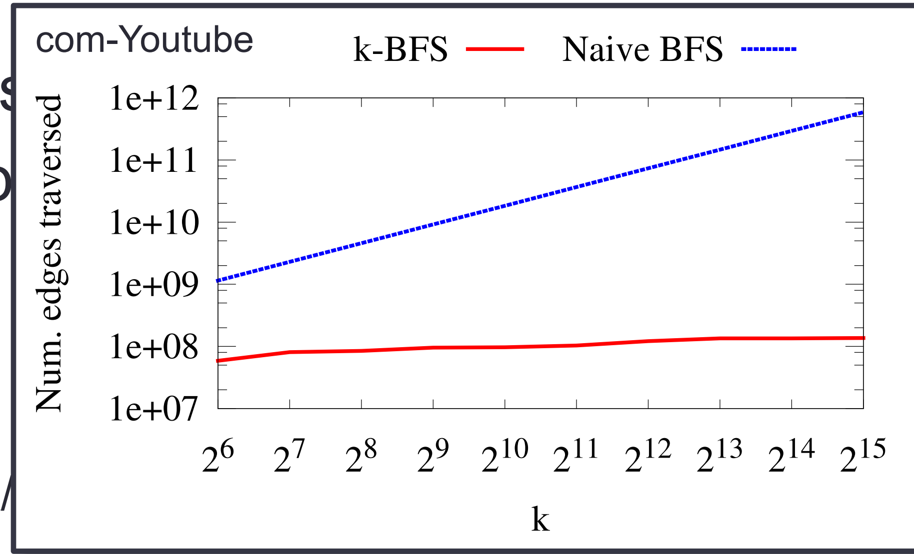


$\hat{ecc}(v) = \max(d(v, s_1), d(v, s_2), d(v, s_3), d(v, s_4))$ for all v

- In practice, need to run two sweeps to get good accuracy [KDD 2015]

Eccentricity estimation implementation

- Run all k BFS's simultaneous
- Take advantage of bit-level p information



v

000000000…000000000
000000000…000000000
⋮
000000000…000000000

k/

Unique bit set for each source

s_1

100000000…000000000
000000000…000000000
⋮
000000000…000000000

s_2

010000000…000000000
000000000…000000000
⋮
000000000…000000000

Eccentricity estimation implementation

- Initial **frontier** = {s_1, s_2, …, s_k}
- d = 0
- While **frontier** not empty: *//Advance all BFS's by 1 level*
  - **nextFrontier** = {}
  - d = d+1
  - For each vertex v in **frontier**:
    - For each neighbor ngh: *//pass "visited" information*
      - Do bitwise-OR of v's words with ngh's words and store in ngh
      - If any of ngh's words changed:
        - $\hat{ecc}$(ngh) = max($\hat{ecc}$(ngh), d) and place ngh on **nextFrontier** if not there
  - **frontier = nextFrontier**

atomic bitwise-OR using compare-and-swap

EdgeMap

- We will implement this example in Ligra for k=64

Ligra Implementation Details

VertexSubset, VertexMap, and VertexFilter

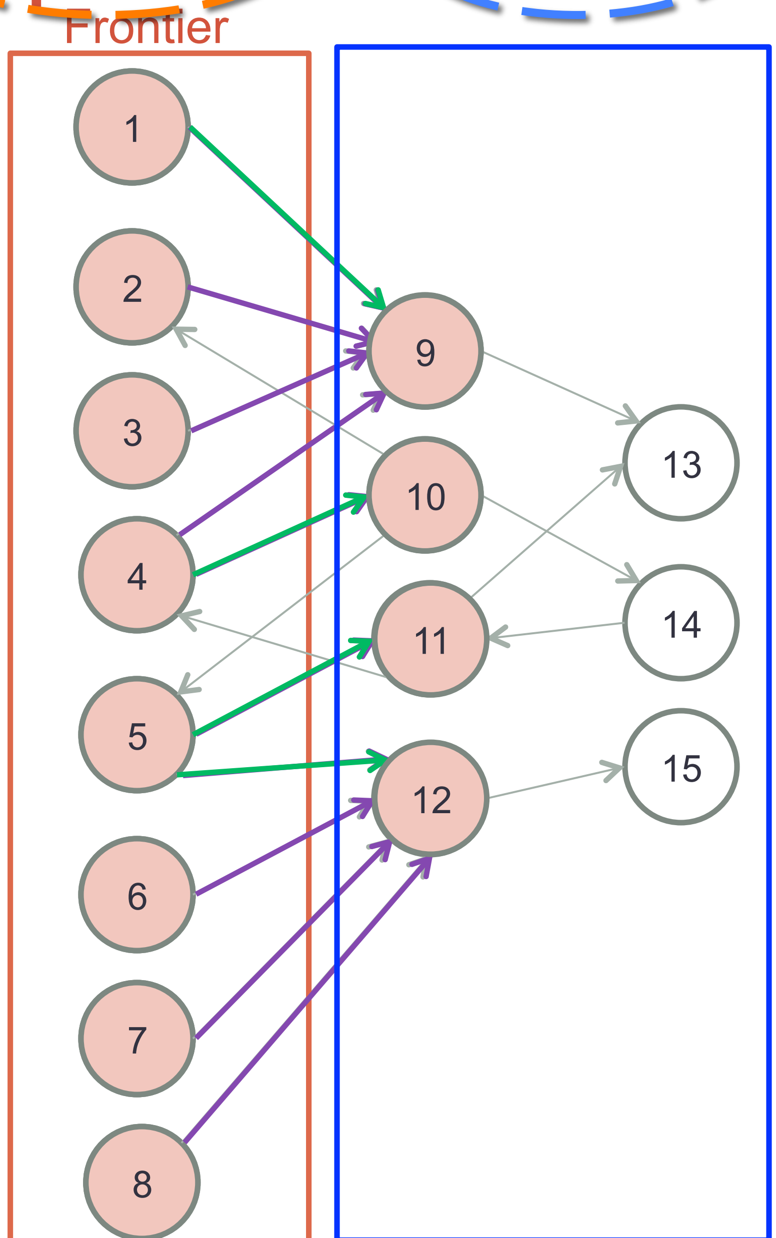
VertexSubset has one of two representations:

- Sparse integer array, e.g. {1, 4, 7}
- Dense boolean array, e.g. {0,1,0,0,1,0,0,1}

```
procedure VERTEXMAP(VertexSubset U, func F):
    parallel foreach v in U:
        F(v); //side-effects application data
```

```
procedure VERTEXFILTER(VertexSubset U, bool func F):
    result = {}
    parallel foreach v in U:
        if(F(v) == 1) then:
            add v to result;
    return result;
```

Sparse or Dense EdgeMap?



- Dense method better when frontier is large and many vertices have been visited
- Sparse (traditional) method better for small frontiers
- Switch between the two methods based on frontier size [Beamer et al. SC '12]

Limited to BFS?

EdgeMap

```
procedure EDGEMAP(G, frontier, Update, Cond):
    if (above threshold) then:
        return EDGEMAP_DENSE(G, frontier, Update, Cond);
    else:
        return EDGEMAP_SPARSE(G, frontier, Update, Cond);
```

Loop through outgoing edges of frontier vertices in parallel

Loop through incoming edges of "unexplored" vertices (in parallel), breaking early if possible

- More general than just BFS!
- Generalized to many other problems
- Users need not worry about this

EdgeMap (sparse version)

- How to represent VertexSubset?
  - Array of integers, e.g. U = {0, 5, 7}

```
procedure EDGEMAP_SPARSE(G, frontier, UPDATE, COND):
    nextFrontier = {};
    parallel foreach v in frontier:
        parallel foreach w in out_neighbors(v):
            if(COND(w) == 1 and UPDATE(v, w) == 1) then:
                add w to nextFrontier;
    remove duplicates from nextFrontier;
    return nextFrontier;
```

```
parents = {-1, …, -1};          //-1 indicates "unvisited"

procedure UPDATE(s, d):
    return compare_and_swap(parents[d], -1, s);

procedure COND(i):
    return parents[i] == -1; //checks if "unvisited"

procedure BFS(G, r):
    parents[r] = r;
    frontier = {r}; //vertexSubset
    while (size(frontier) > 0):
        frontier = EDGEMAP(G, frontier, UPDATE, COND);
```

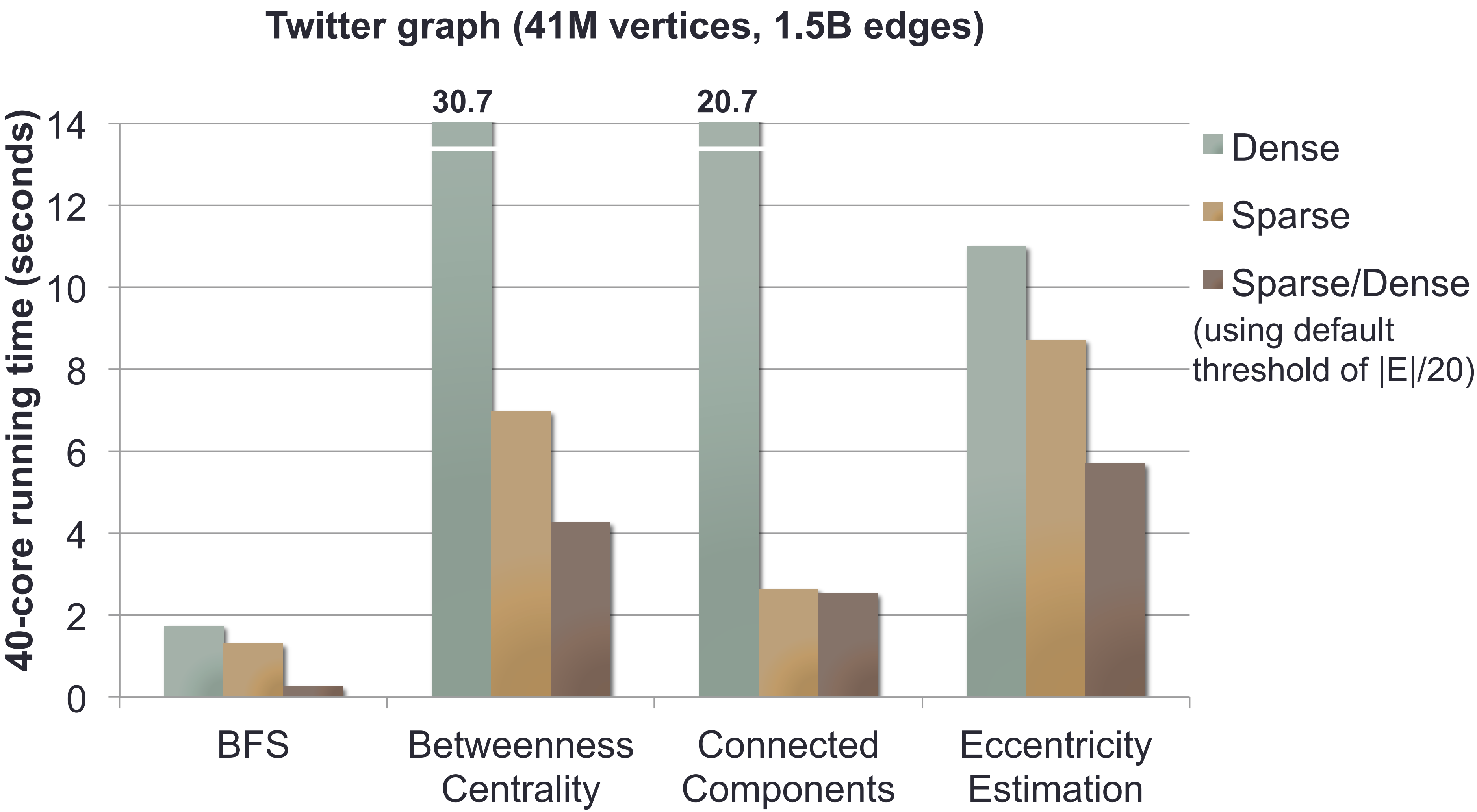
EdgeMap (dense version)

- How to represent dense VertexSubset?
  - Byte array, e.g. U = {1, 0, 0, 0, 0, 1, 0, 1}

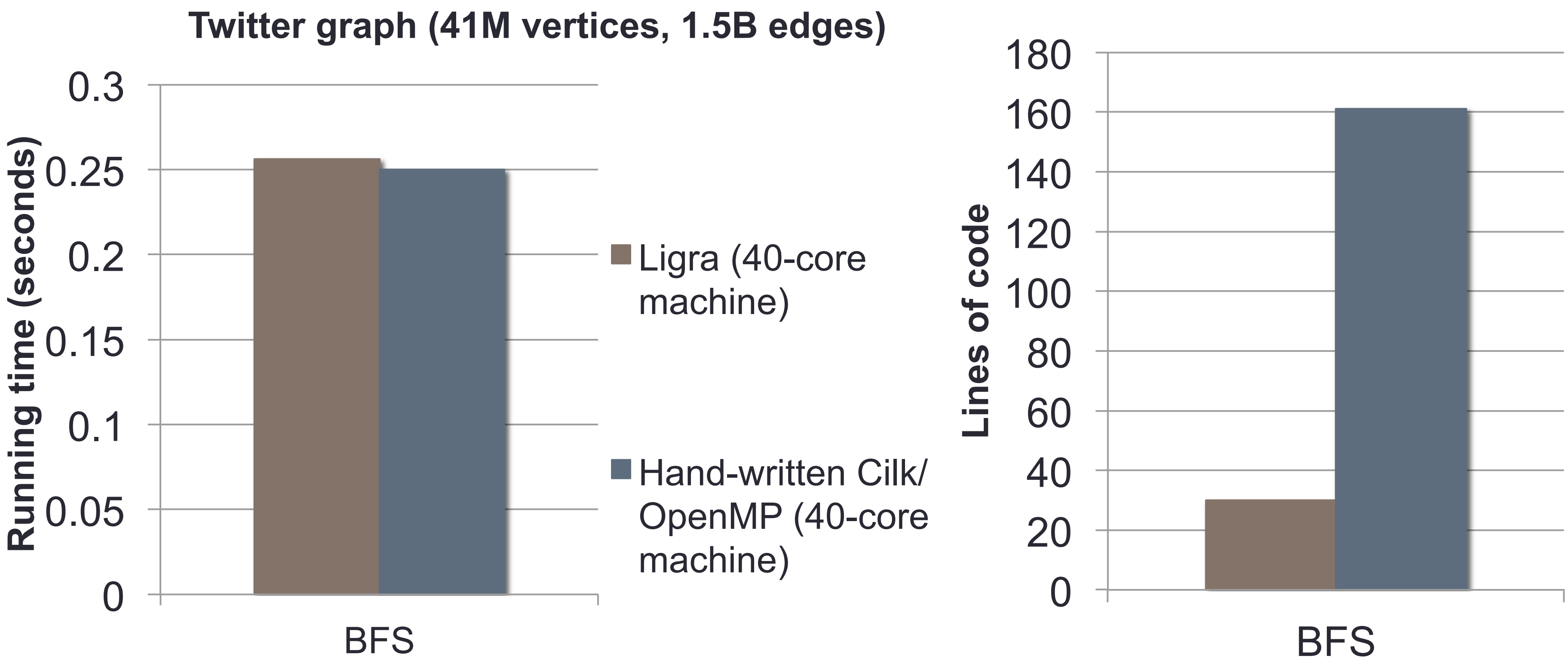
```
procedure EDGEMAP_DENSE(G, frontier, UPDATE, COND):
    nextFrontier = {0, …, 0};
    parallel foreach v in G:
        if (COND(v) == 1) then:
            foreach ngh in in_neighbors(v):
                if ngh in frontier and UPDATE(ngh, v) == 1 then:
                    nextFrontier[v] = 1;
                if (COND(v) == 0) then:
                    break;
    return nextFrontier;
```

```
parents = {-1, …, -1};          //-1 indicates "unvisited"

procedure UPDATE(s, d):
    return compare_and_swap(parents[d], -1, s);

procedure COND(i):
    return parents[i] == -1; //checks if "unvisited"

procedure BFS(G, r):
    parents[r] = r;
    frontier = {r}; //vertexSubset
    while (size(frontier) > 0):
        frontier = EDGEMAP(G, frontier, UPDATE, COND);
```

Frontier-based approach enables sparse/dense traversal



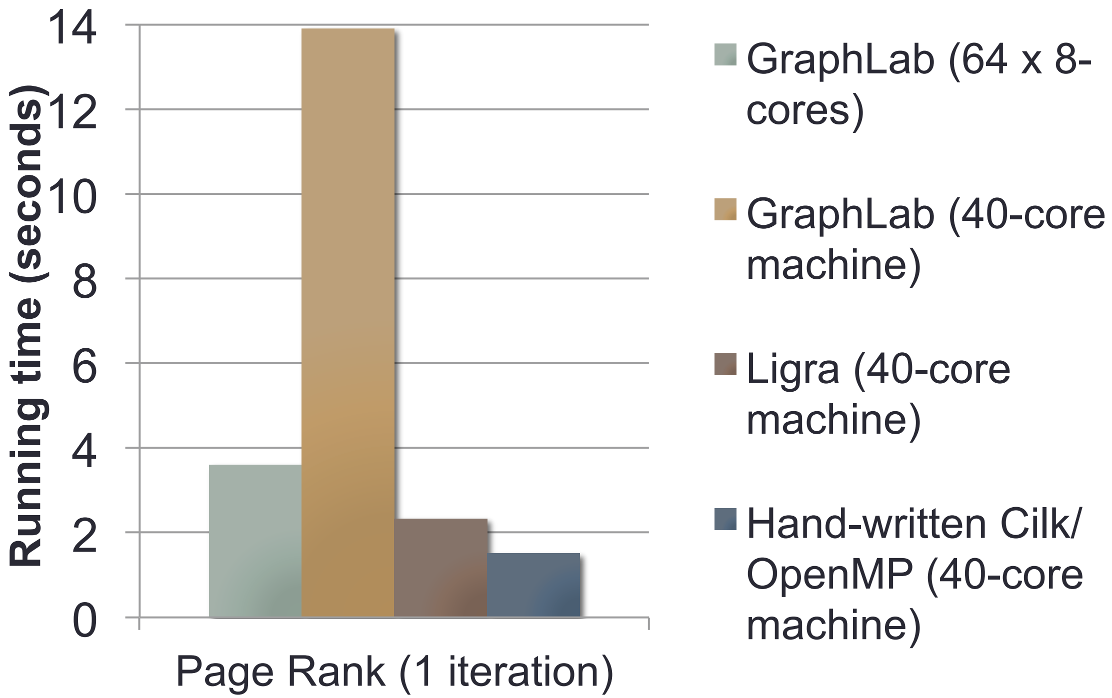
Ligra BFS Performance

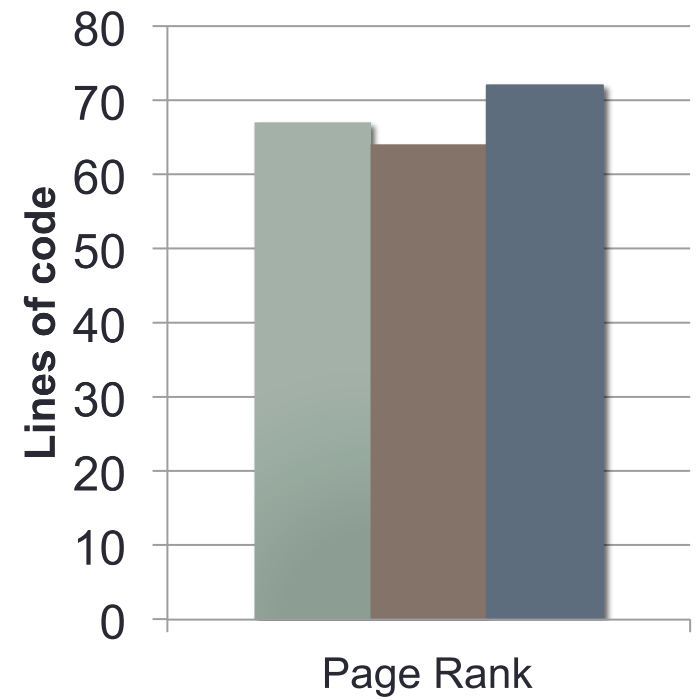


- Comparing against direction-optimizing code by Beamer et al.

Ligra PageRank Performance

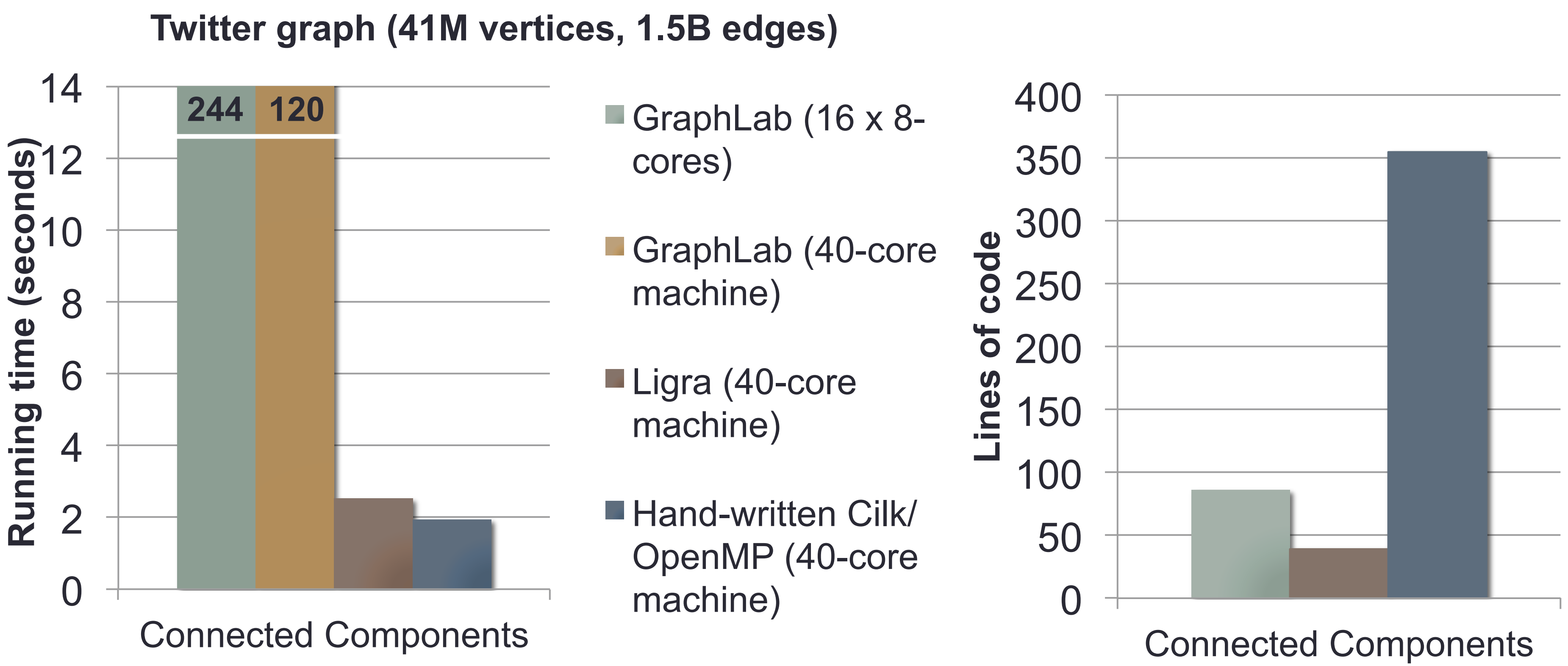
Twitter graph (41M vertices, 1.5B edges)





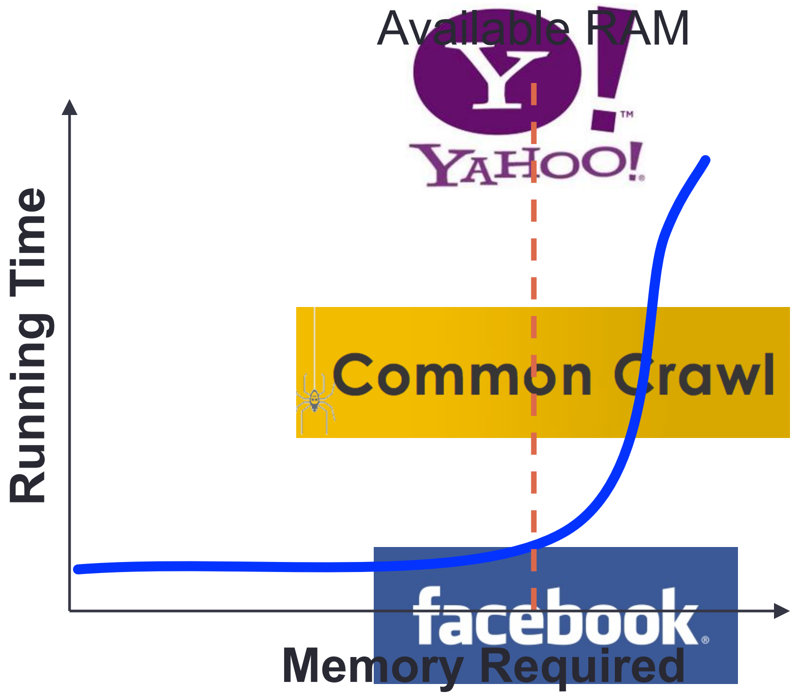
- Easy to implement “sparse” version of PageRank in Ligra

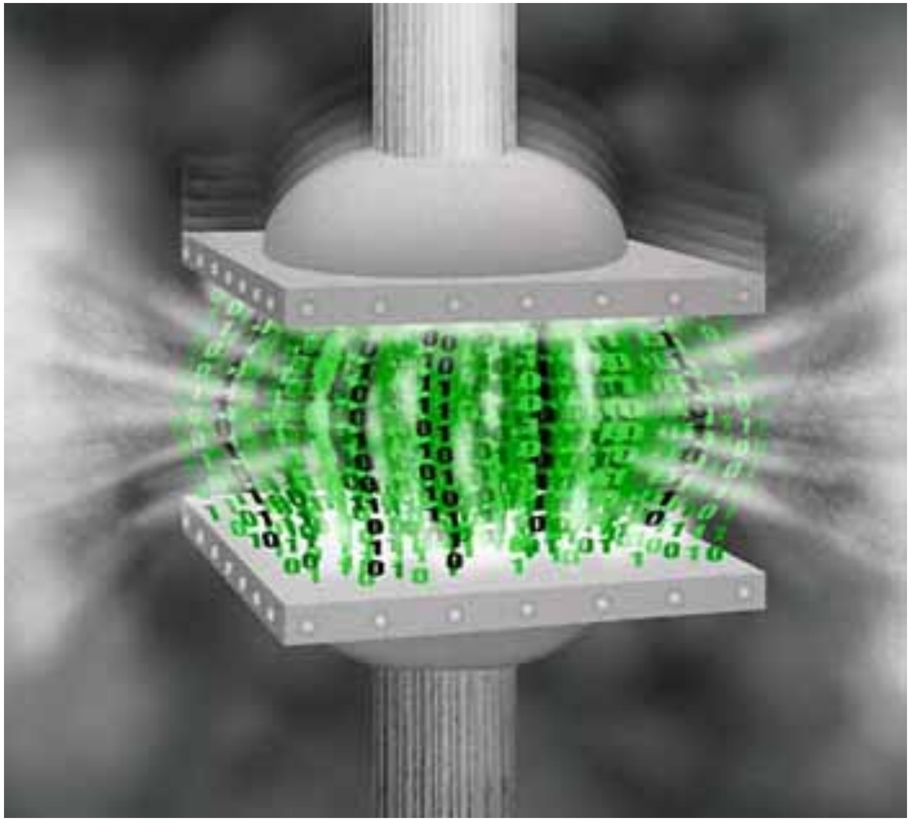
Ligra Connected Components Performance



- Performance close to hand-written code
- Faster than existing high-level frameworks at the time
- Shared-memory graph processing is very efficient
  - Several shared-memory graph processing systems subsequently developed: Galois [SOSP '13], X-stream [SOSP '13], PRISM [SPAA '14], Polymer [PPoPP '15], Ringo [SIGMOD '15], GraphMat [VLDB '15]

Large Graphs

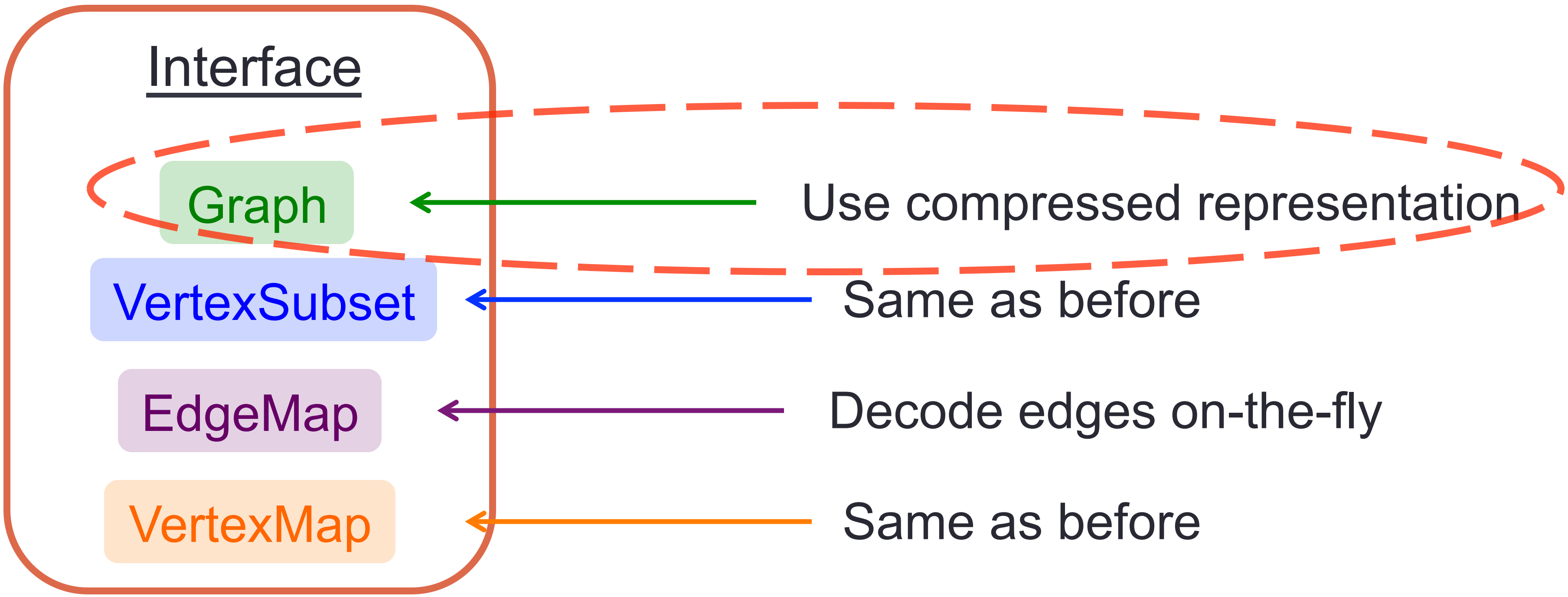




- Most can fit on commodity shared memory machine
- *What if you don't have that much memory?*

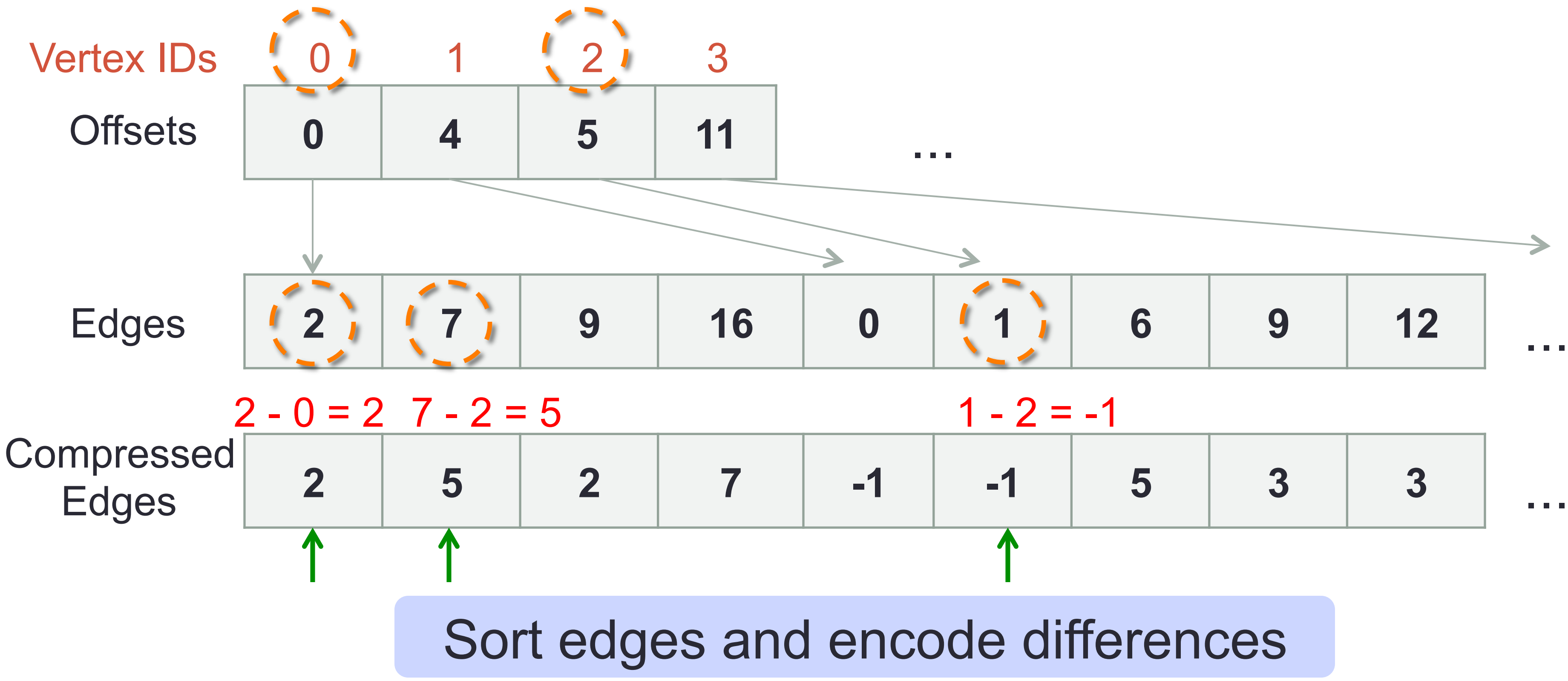
Ligra+: Adding Graph Compression to Ligra

Ligra+: Adding Graph Compression to Ligra



- Same interface as Ligra
- All changes hidden from the user!

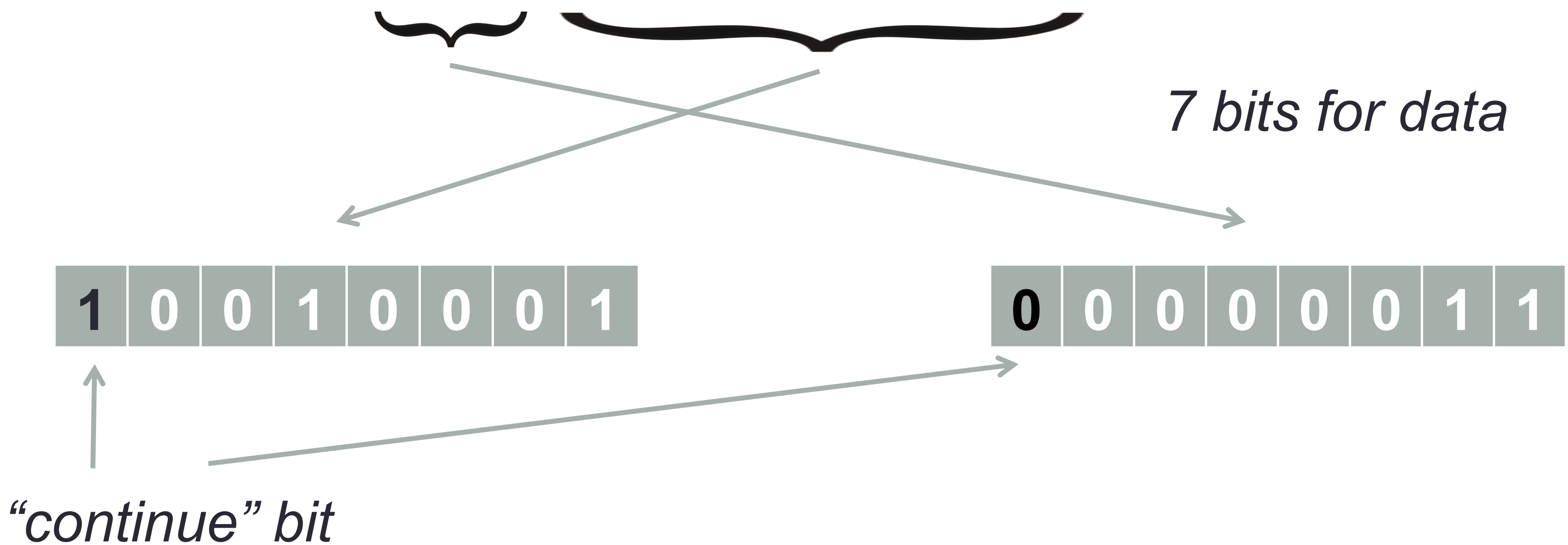
Graph representation



- Graph reordering to improve locality
  - Goal: give neighbors IDs close to vertex ID
  - BFS, DFS, METIS, our own separator-based algorithm

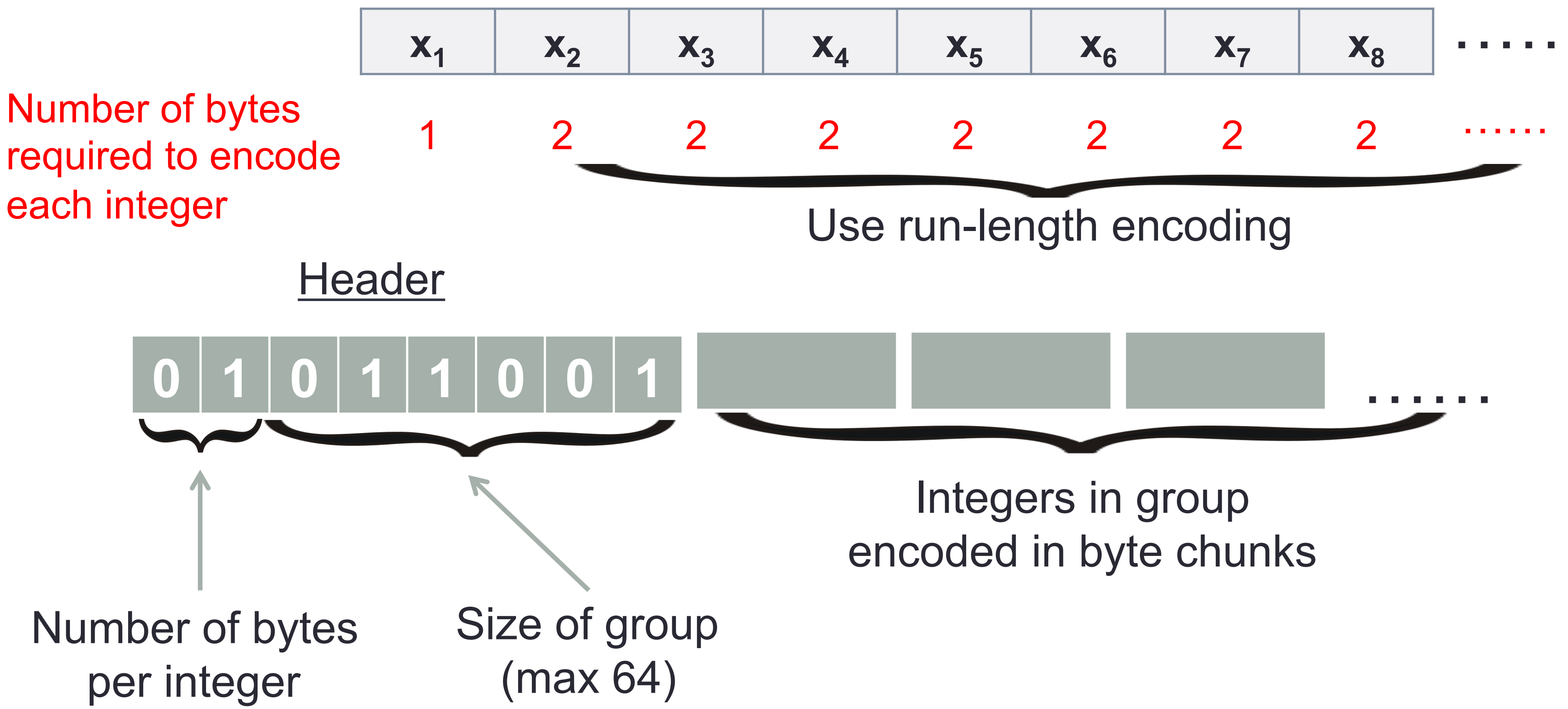
Variable-length codes

- k-bit codes
  - Encode value in chunks of k bits
  - Use k-1 bits for data, and 1 bit as the “continue” bit
- Example: encode “401” using 8-bit (byte) code
- In binary: 1 1 0 0 1 0 0 0 1



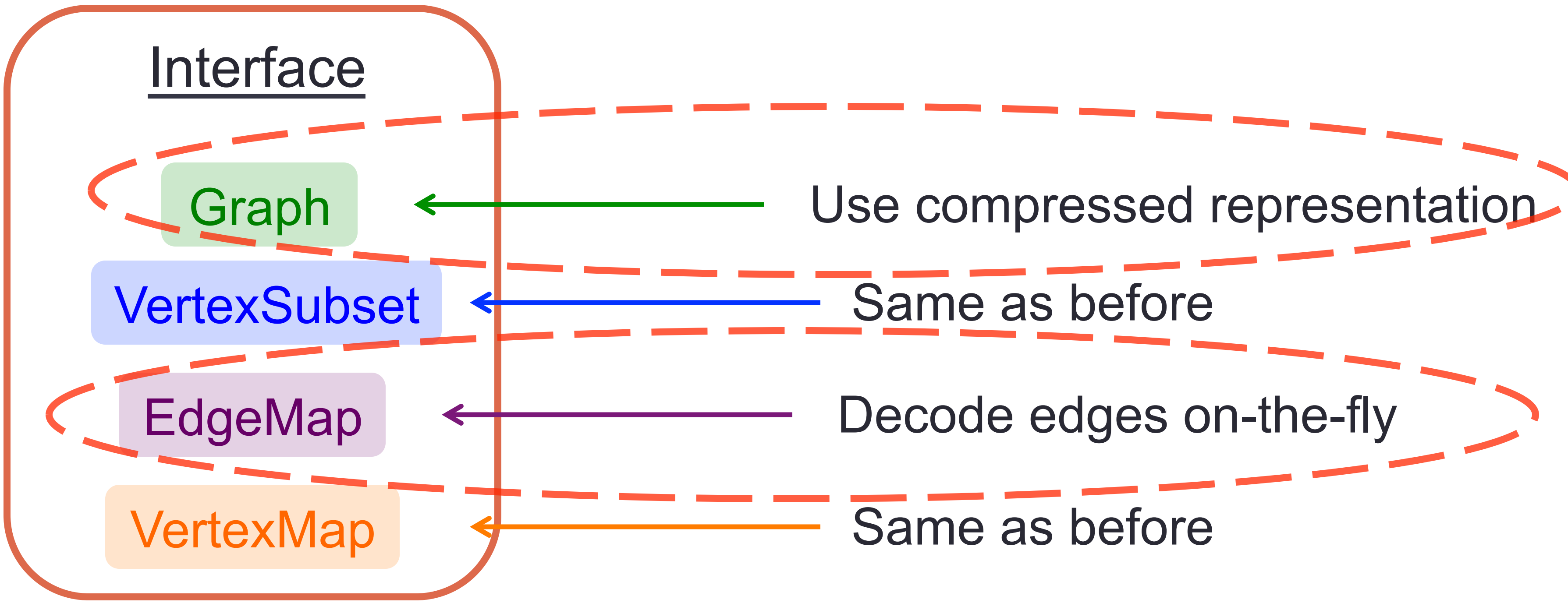
Encoding optimization

- Another idea: get rid of “continue” bits



- Increases space, but makes decoding cheaper (no branch misprediction from checking “continue” bit)

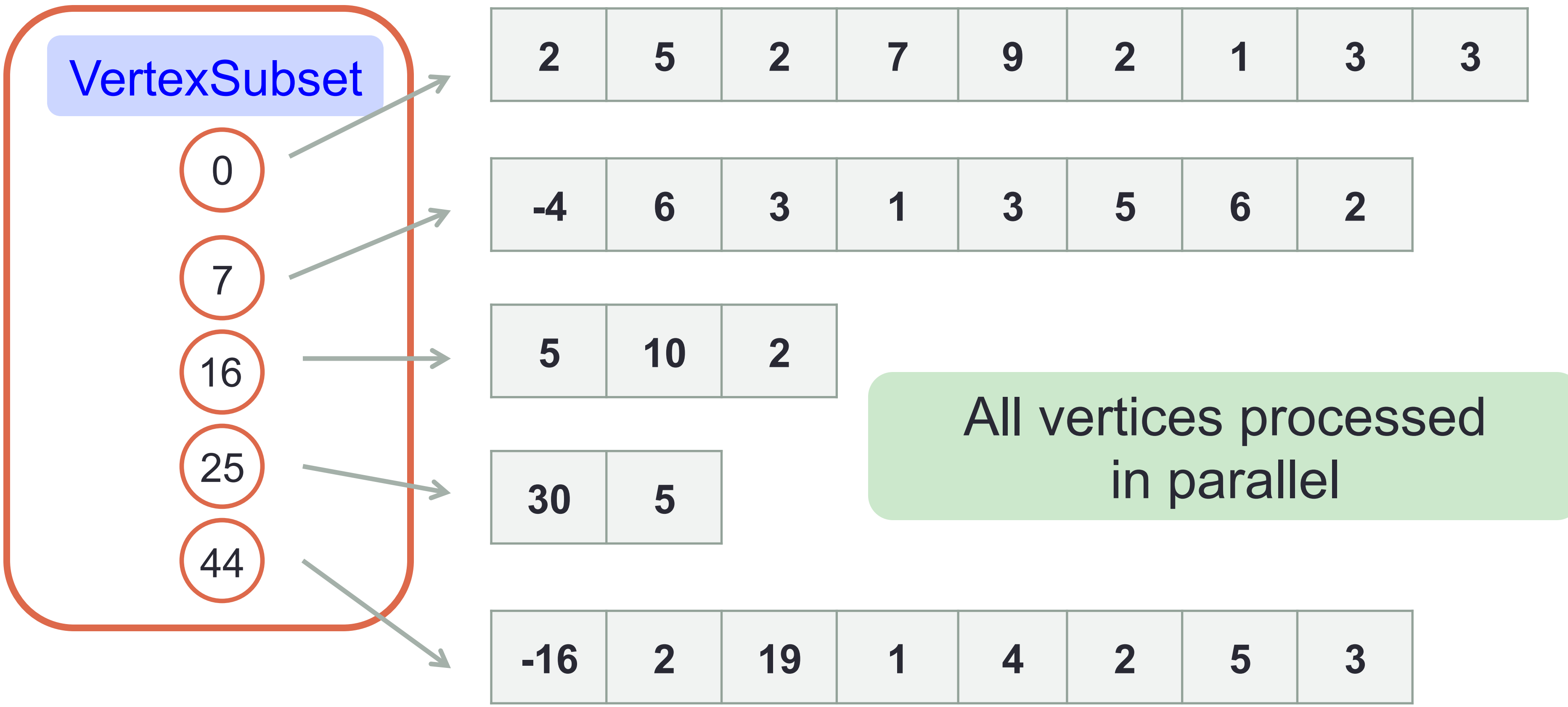
Ligra+: Adding Graph Compression to Ligra



- Same interface as Ligra
- All changes hidden from the user!

Modifying EdgeMap

- Processes outgoing edges of a subset of vertices



What about high-degree vertices?

Handling high-degree vertices

High-degree vertex

-1	2	4	3	16	2	1	5	8	19	4	1	23	14	12	1	9	10	3	5	…

Chunks of size T

…

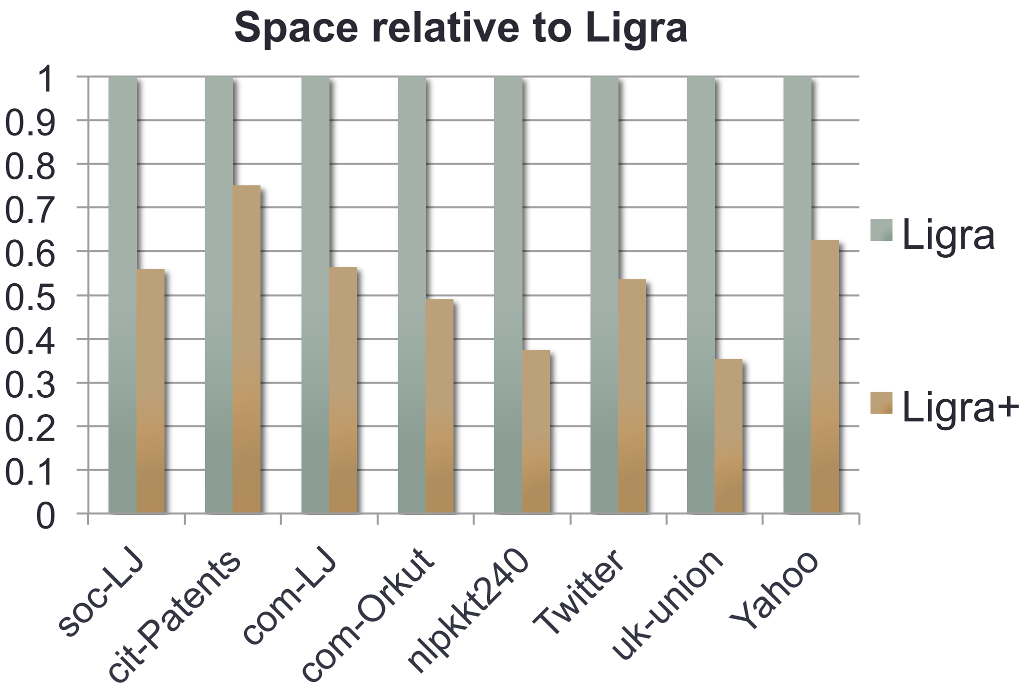
-1	2	4	3	16	2	27	5	8	19	4	1	87	14	12	1	9	10	…

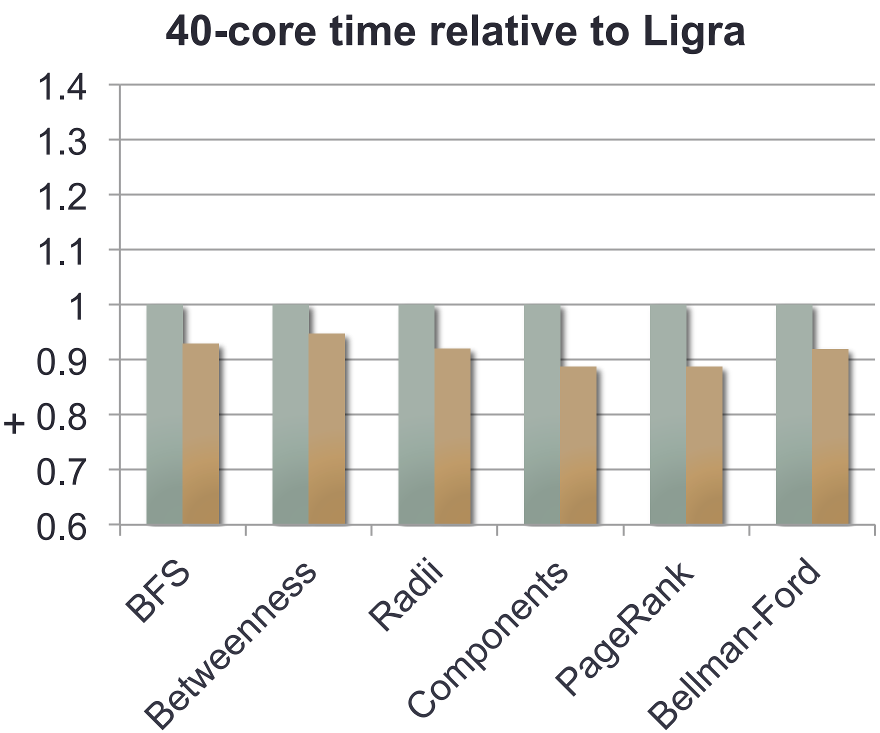
Encode first entry relative to source vertex

- We chose T=1000
- Similar performance and space usage for a wide range of T

All chunks can be decoded in parallel!

Ligra+: Adding Graph Compression to Ligra





- Using 8-bit codes with run-length encoding
- Cost of decoding on-the-fly?
- Memory bottleneck a bigger issue as graph algorithms are memory-bound

Demo on compressed graphs in Ligra

Other Graph Processing Systems

Many existing frameworks

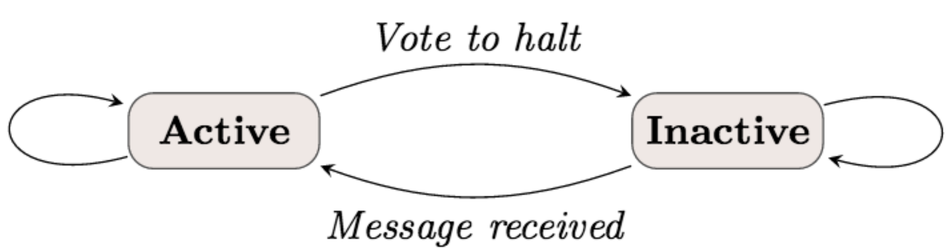
- Pregel/Giraph/GPS, GraphLab/PowerGraph, PRISM, Pegasus, Knowledge Discovery Toolbox/CombBLAS, GraphChi, GraphX, Galois, X-Stream, Gunrock, GraphMat, Ringo, TurboGraph, FlashGraph, Grace, PathGraph, Polymer, GoFFish, Blogel, LightGraph, MapGraph, PowerLyra, PowerSwitch, XDGP, Signal/Collect, PrefEdge, Parallel BGL, KLA, Grappa, Chronos, Green-Marl, GraphHP, P++, LLAMA, Venus, Cyclops, Medusa, NScale, Neo4J, Trinity, GBase, HyperGraphDB, Horton, GSPARQL, Titan, and many others…
- Cannot list everything here. For more information, see:
  - "Systems for Big Graphs", Khan and Elnikety VLDB 2014 Tutorial
  - "Trade-offs in Large Graph Processing: Representations, Storage, Systems and Algorithms", Ajwani et al. WWW 2015 Tutorial
  - "A Survey of Parallel Graph Processing Frameworks", Doekemeijer and Varbanescu 2014
  - "Thinking like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing", McCune et al. 2015

Pregel

- “Think like a vertex”
- Distributed-memory, uses message passing
- Bulk synchronous model

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

- Vertices can be either “active” or “inactive”

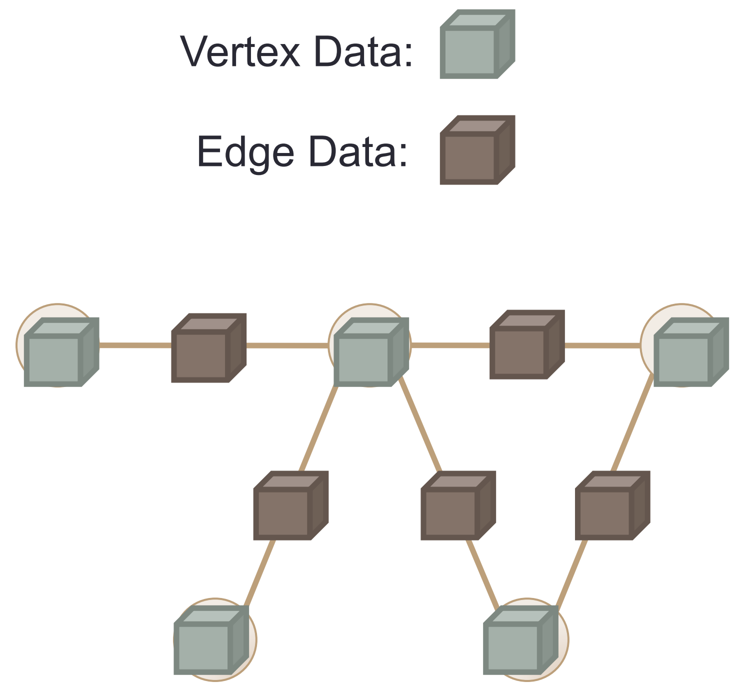


Pregel: A System for Large-Scale Graph Processing, Malewicz et al. SIGMOD 2010

GraphLab/PowerGraph

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

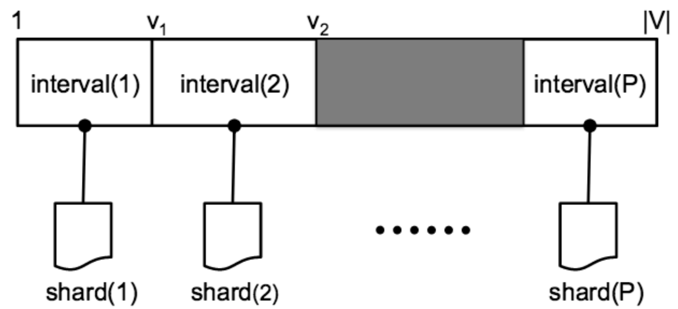
- "Think like a vertex"
- Vertices define Gather, Apply, and Scatter functions
- Data on edges as well as vertices
- Scheduler processes "active" vertices
  - Supports asynchronous execution
    - Useful for some machine learning applications
  - Different levels of consistency
  - Different scheduling orders
- Current version for distributed memory (original version was for shared memory)

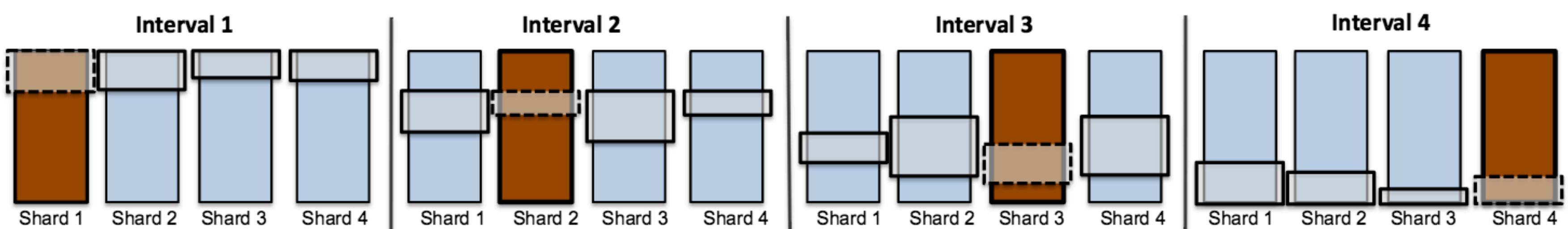


GraphLab: A New Framework for Parallel Machine Learning, Low et al. UAI 2010
PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, Gonzalez et al. OSDI 2012

GraphChi

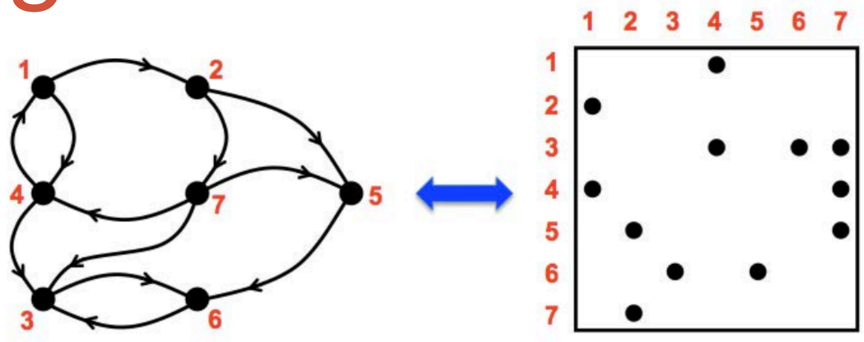
- "Think like a vertex"
  - Define Update function for vertices
- Optimized for disk-based execution
- Divides edges into "shards"
  - Each shard has a range of target IDs
  - Each shard sorted by source ID
- Parallel sliding windows method for efficient execution
  - One shard in memory, other shards read in streaming fashion
  - About O((V+E)/B) I/O's per iteration





GraphChi: Large-Scale Graph Computation on Just a PC, Kyrola et al. OSDI 2012

Linear algebra abstraction



- PEGASUS: Framework using matrix-vector abstraction in MapReduce [Kang et al. ICDM 2009]
- Knowledge Discovery Toolbox/CombBLAS: Uses matrix-vector and matrix-matrix routines for implementing graph algorithms; uses Python frontend, C++/MPI backend [Lugowski et al. SDM 2012]

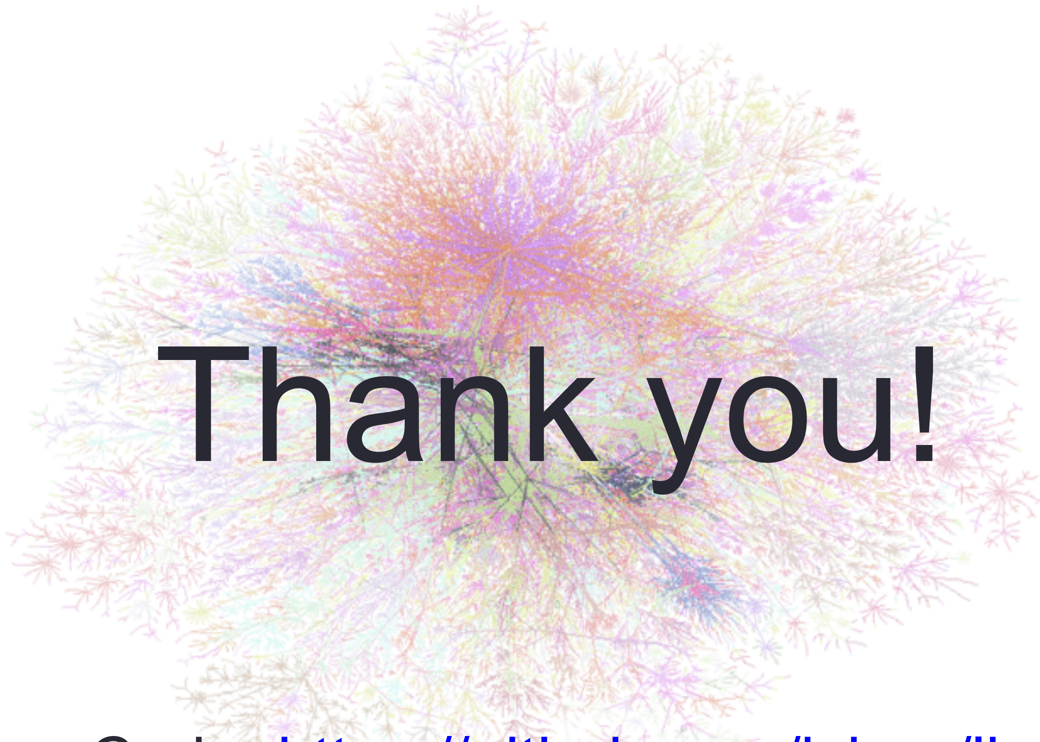
Some other shared memory systems

- X-Stream [SOSP 2013]
  - Edge-centric abstraction
  - Supports disk-based execution
- Galois [SOSP 2013]
  - Parallel programming framework based on dynamic sets with various schedulers
  - Implements graph abstractions of Ligra, X-Stream, and GraphLab
- PRISM [SPAA 2014]
  - Deterministic version of GraphLab using graph coloring for scheduling; implemented in Cilk
- Polymer [PPoPP 2015]
  - NUMA-aware implementation of Ligra's abstraction
- GraphMat [VLDB 2015]
  - Uses optimized matrix-vector routines to implement graph algorithms
- Gunrock [PPoPP 2016]
  - Abstraction for frontier-based computations on GPUs

Exercises

Exercises

- Implement a version of BFS that gives a deterministic BFS tree (i.e., the Parents array is deterministic)
- Implement a version of BFS that uses a bit-vector to check the "visited" status before updating the Parents array



Thank you!

Code: https://github.com/jshun/ligra/

References
Ligra: A Lightweight Graph Processing Framework for Shared Memory, PPoPP 2013.
Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+, DCC 2015.
An Evaluation of Parallel Eccentricity Estimation Algorithms on Undirected Real-World Graphs, KDD 2015.