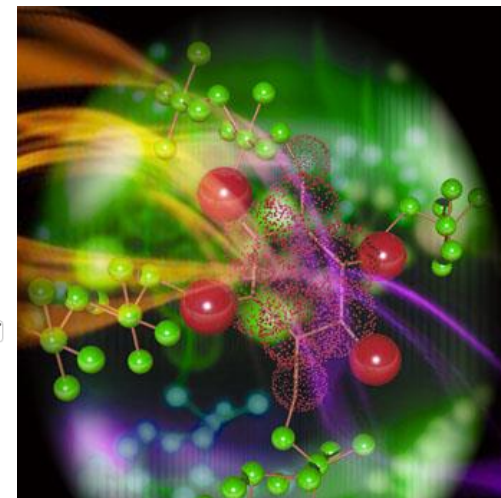
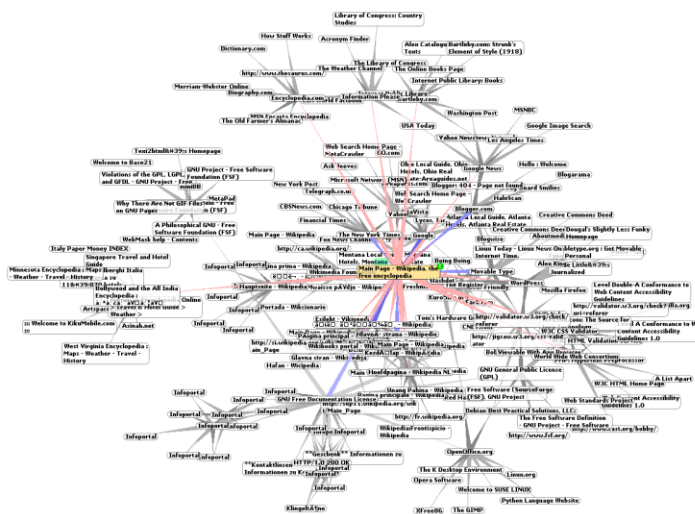


A Framework for Processing Large Graphs in Shared Memory

Julian Shun

Based on joint work with Guy Blelloch and Laxman Dhulipala

What are graphs?

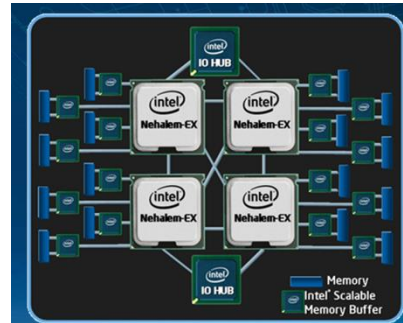


Graph Data is Everywhere!

- Can contain up to billions of vertices and edges
- Need simple, efficient, and scalable ways to analyze them

Efficient Graph Processing

- Use parallelism



- Design efficient algorithms

Breadth-first search
Betweenness centrality
Connected components
...

Single-source shortest paths
Eccentricity estimation
(Personalized) PageRank
...

- Write/optimize code for each application
- Build a general framework

Ligra Graph Processing Framework

EdgeMap

VertexMap

Breadth-first search
Betweenness centrality
Connected components
Triangle counting
K-core decomposition
Maximal independent set
Set cover

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

Simplicity, Performance, Scalability

Graph Processing Systems

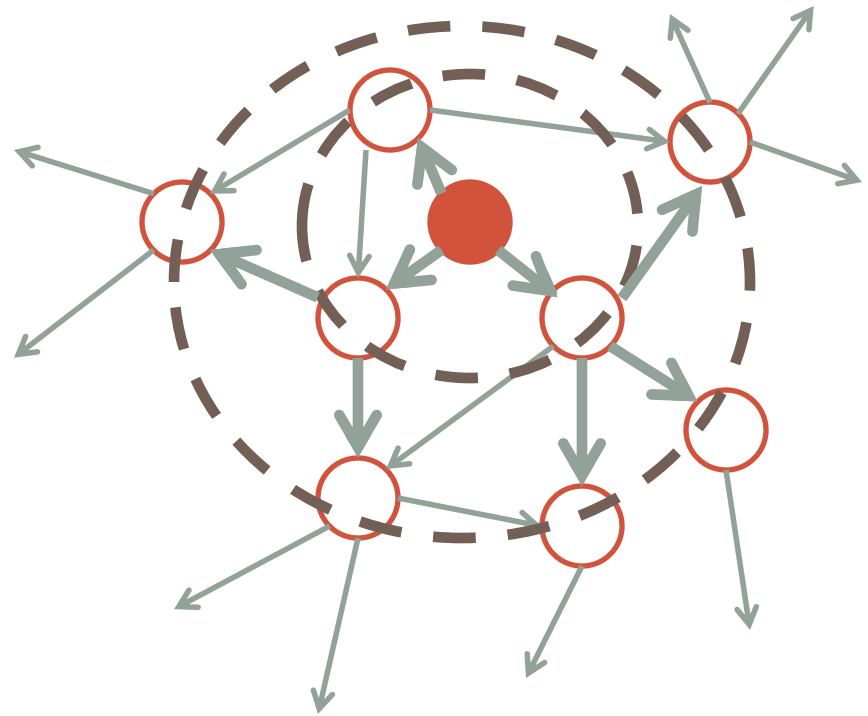
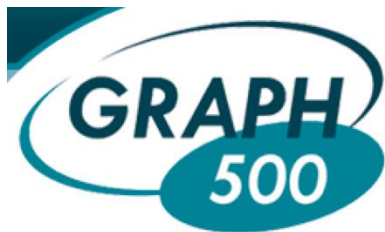
- Existing: Pregel/Giraph/GPS, GraphLab, Pegasus, Knowledge Discovery Toolbox, GraphChi, etc.
- Our system: **Ligra** - Lightweight graph processing system for shared memory

*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 frontier in parallel
- Race conditions, load balancing

Steps for Graph Traversal

Many graph traversal algorithms do this!

Graph

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

VertexSubset

EdgeMap

VertexMap

*We built the **Ligra** abstraction for these kinds of computations*

Think with flat data-parallel operators

*Abstraction enables optimizations
(hybrid traversal and graph compression)*

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”*

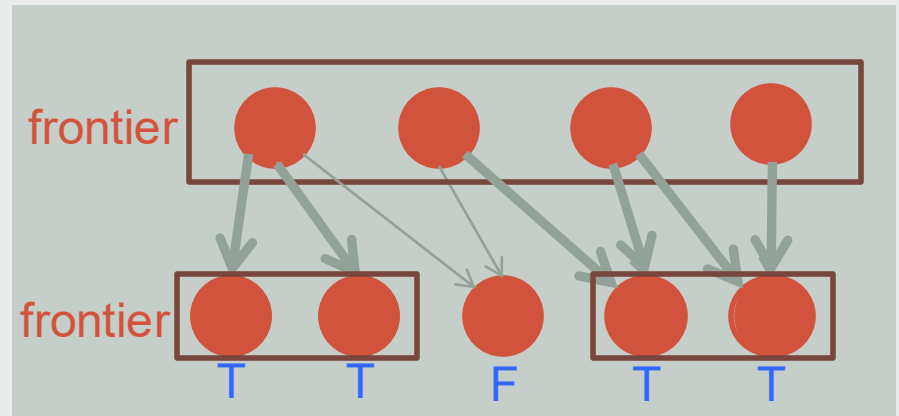
procedure **BFS**(G, r):

 parents[r] = r;

 frontier = {r}; *// VertexSubset*

 while (size(frontier) > 0):

 frontier = **EDGEMAP**(G, frontier, **UPDATE**, **COND**);



Actual BFS code in Ligra

```
#include "ligra.h"

struct BFS_F {
    intT* Parents;
    BFS_F(intT* _Parents) : Parents(_Parents) {}
    inline bool update (intT s, intT d) { //Update
        if(Parents[d] == -1) { Parents[d] = s; return 1; }
        else return 0;
    }
    inline bool updateAtomic (intT s, intT d){ //atomic version of Update
        return (CAS(&Parents[d],(intT)-1,s));
    }
    //cond function checks if vertex has been visited yet
    inline bool cond (intT d) { return (Parents[d] == -1); }
};

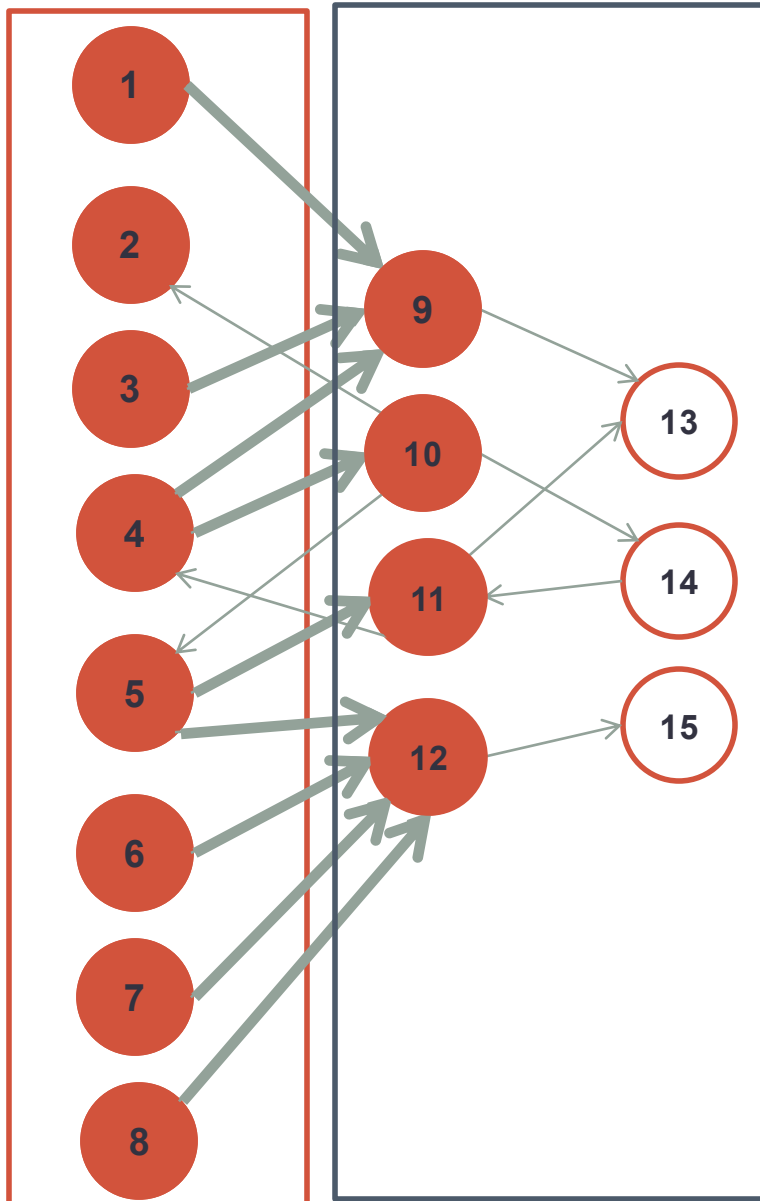
template <class vertex>
void Compute(graph<vertex> GA, intT start) {
    intT n = GA.n;
    //creates Parents array, initialized to all -1, except for start
    intT* Parents = newA(intT,GA.n);
    parallel_for(intT i=0;i<GA.n;i++) Parents[i] = -1;
    Parents[start] = start;

    vertexSubset Frontier(n,start); //creates initial frontier

    while(!Frontier.isEmpty()){ //loop until frontier is empty
        vertexSubset output = edgeMap(GA, Frontier, BFS_F(Parents));
        Frontier.del();
        Frontier = output; //set new frontier
    }
    Frontier.del();
    free(Parents);
}
```

(Sparse) or Dense EdgeMap?

Frontier

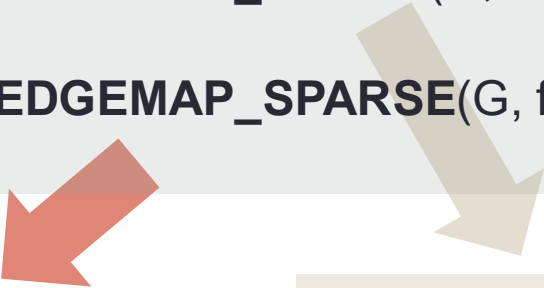


- Dense method better when frontier is large and many vertices have been visited
- Sparse (traditional) method better for small frontiers
- Switch between the two methods based on frontier size [Beamer et al. SC '12]

Limited to BFS?

EdgeMap

```
procedure EDGEMAP(G, frontier, Update, Cond):  
  if (size(frontier) + sum of out-degrees > threshold) then:  
    return EDGEMAP_DENSE(G, frontier, Update, Cond);  
  else:  
    return EDGEMAP_SPARSE(G, frontier, Update, Cond);
```



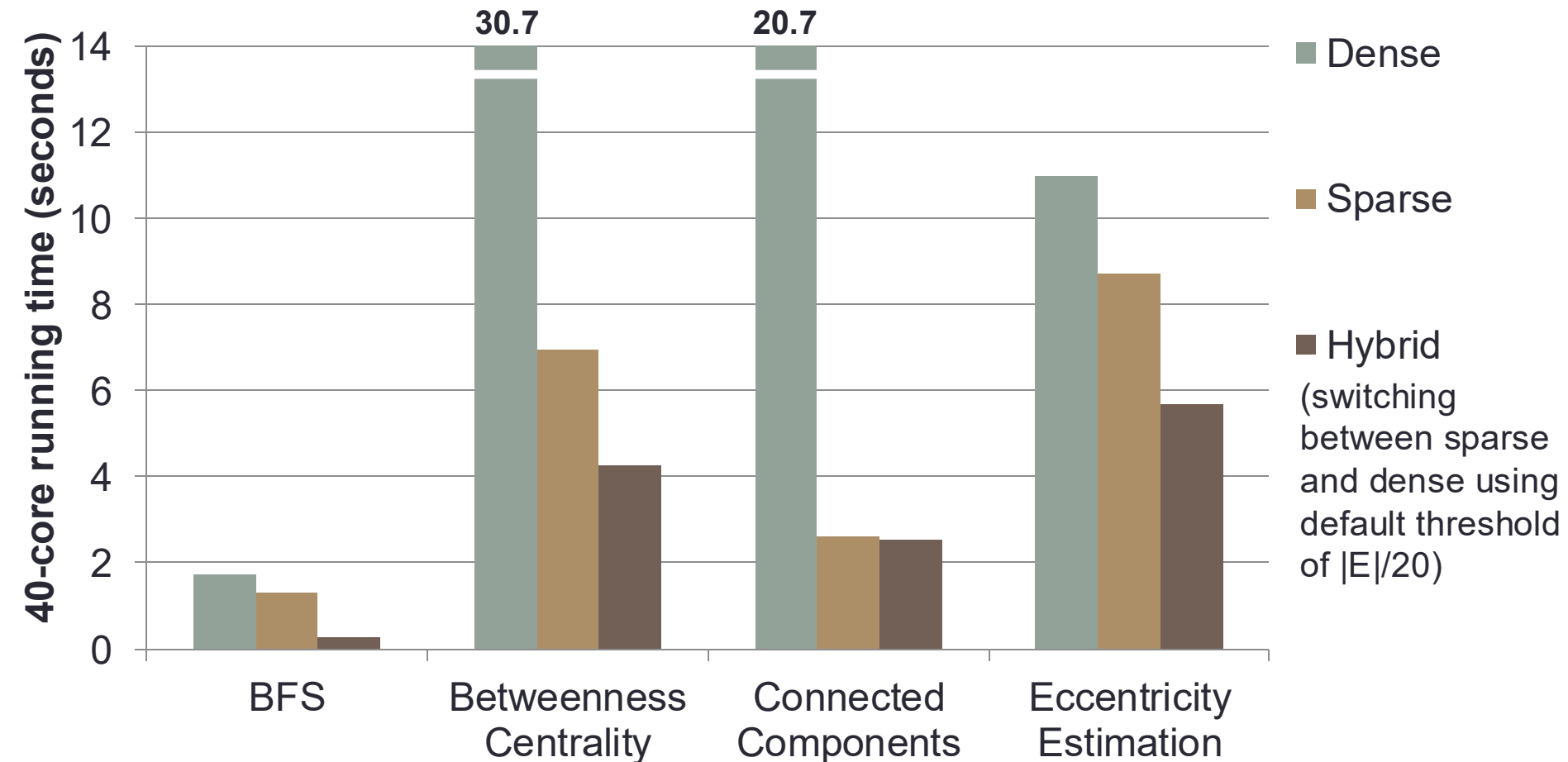
Loop through outgoing edges of frontier vertices in parallel

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

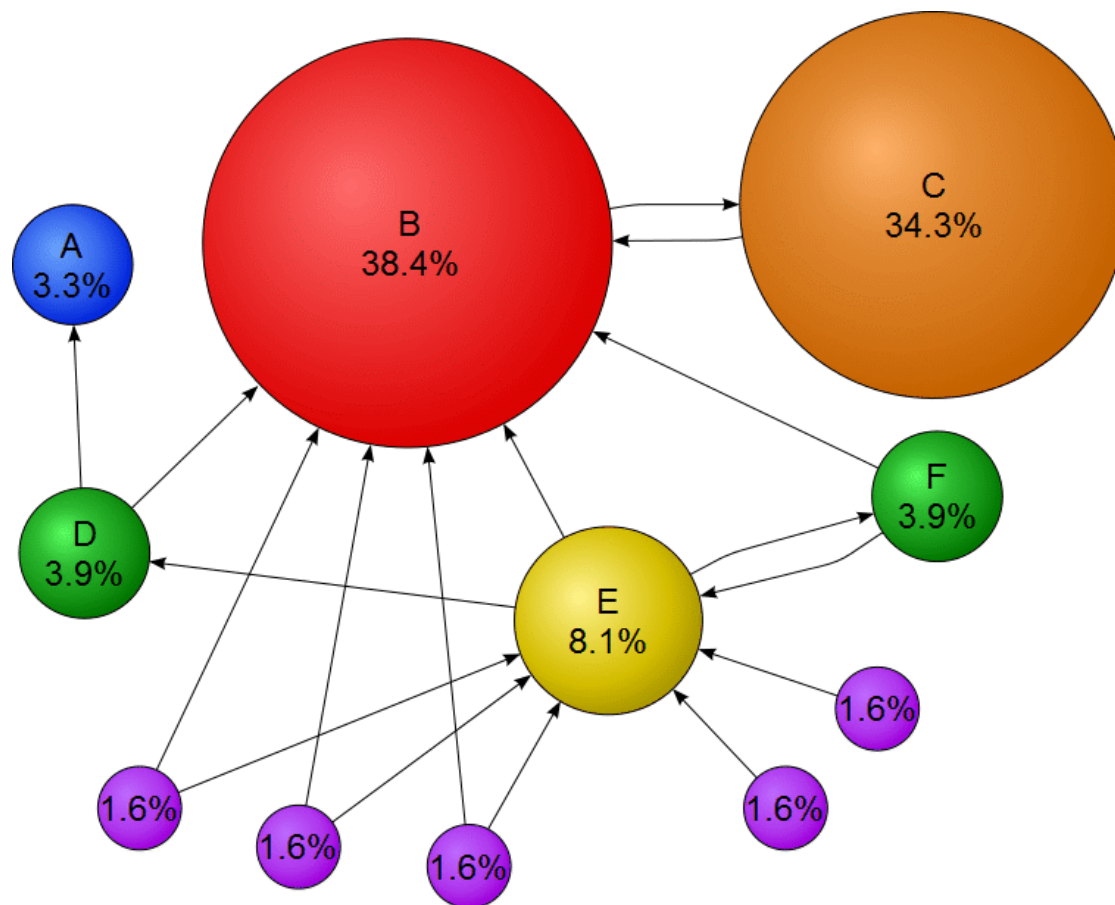
- **More general than just BFS!**
- Generalized to many other problems
 - For example, betweenness centrality, connected components, sparse PageRank, shortest paths, eccentricity estimation, graph clustering, k-core decomposition, set cover, etc.
- Users need not worry about this

Frontier-based approach enables hybrid traversal

Twitter graph (41M vertices, 1.5B edges)

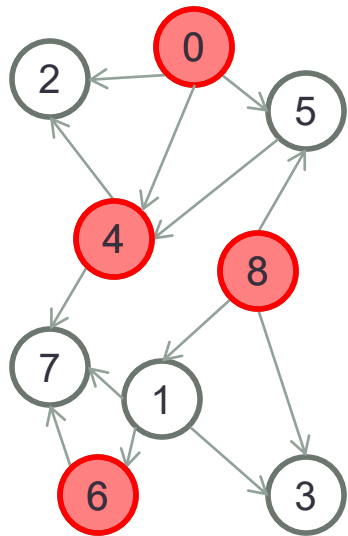


PageRank

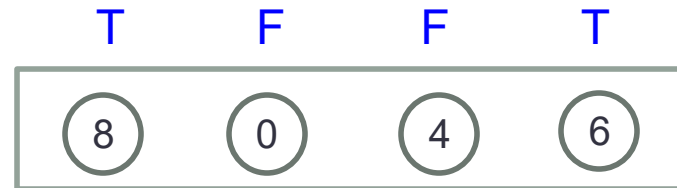


$$PR[v] = \frac{1 - \gamma}{|V|} + \gamma \sum_{u \in N^{-}(v)} \frac{PR[u]}{deg^{+}(u)}$$

VertexMap



VertexSubset



```
bool f(v){  
    data[v] = data[v] + 1;  
    return (data[v] == 1);  
}
```



VertexMap

VertexSubset



PageRank in Ligra

```
p_curr = {1/|V|, ..., 1/|V|};    p_next = {0, ..., 0};    diff = {};    error = ∞;
```

```
procedure UPDATE(s, d):
```

```
    atomic_increment(p_next[d], p_curr[s] / degree(s));
```

```
    return 1;
```

```
procedure COMPUTE(i):
```

```
    p_next[i] =  $\alpha \cdot p\_next[i] + (1 - \alpha) \cdot (1/|V|)$ ;
```

```
    diff[i] = abs(p_next[i] - p_curr[i]);
```

```
    p_curr[i] = 0;
```

```
    return 1;
```

```
procedure PageRank(G,  $\alpha$ ,  $\epsilon$ ):
```

```
    frontier = {0, ..., |V|-1};
```

```
    while (error >  $\epsilon$ ):
```

```
        frontier = EDGEMAP(G, frontier, UPDATE, CONDtrue);
```

```
        frontier = 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 PowerGraph and McSherry WWW '05)

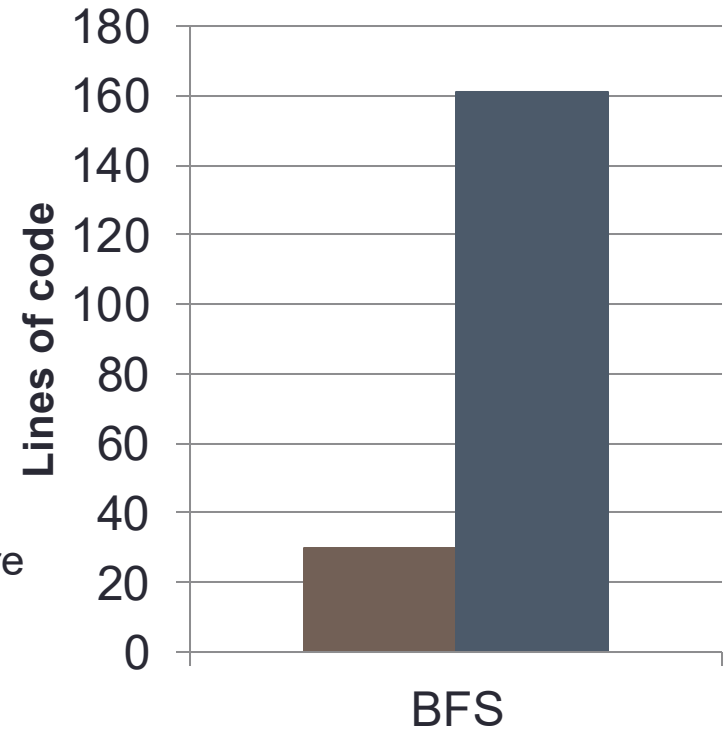
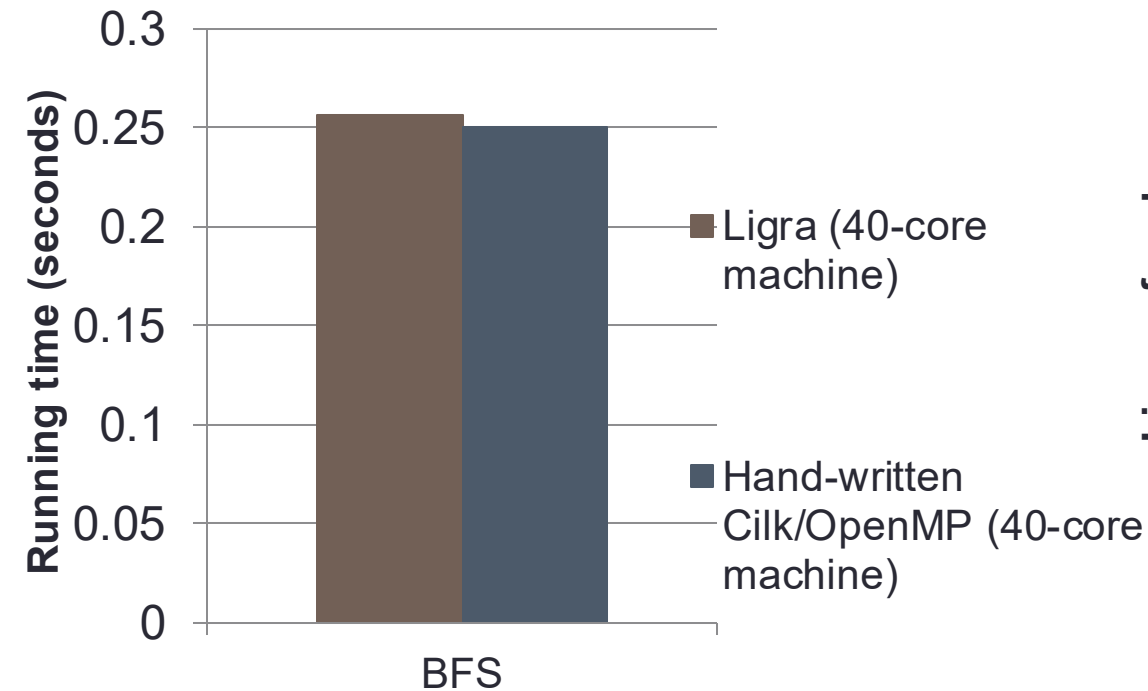
PageRank-Delta in Ligra

```
PR[i] = {1/|V|, ..., 1/|V|};  
nghSum = {0, ..., 0};  
Change = {}; //store changes in PageRank values  
  
procedure UPDATE(s, d): //passed to EdgeMap  
    atomic_increment(nghSum[d], Change[s] / degree(s));  
    return 1;  
  
procedure COMPUTE(i): //passed to VertexMap  
    Change[i] =  $\alpha \cdot$  nghSum[i];  
    PR[i] = PR[i] + Change[i];  
    return (abs(Change[i]) >  $\Delta$ ); //check if absolute value of change is big enough
```

Performance of Ligra

Ligra BFS Performance

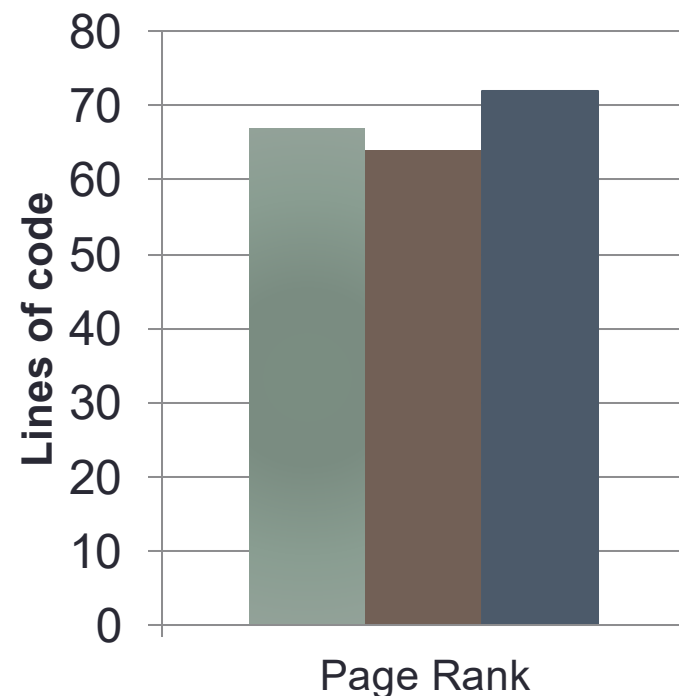
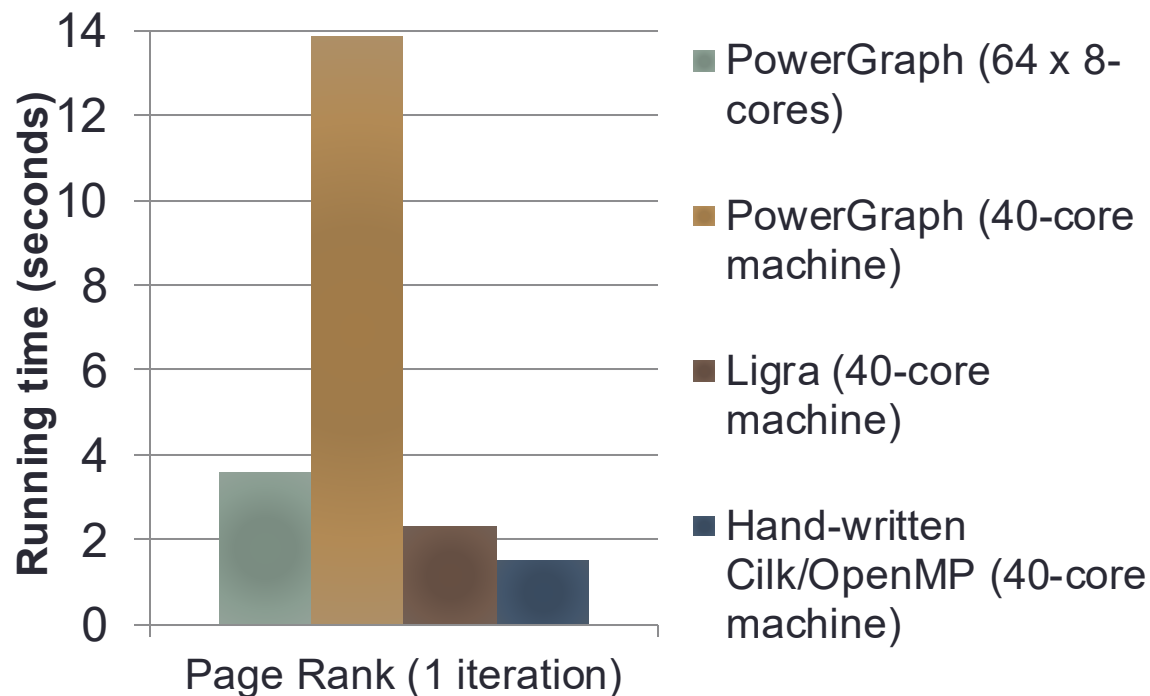
Twitter graph (41M vertices, 1.5B edges)



- Comparing against hybrid traversal BFS code by Beamer et al.

Ligra PageRank Performance

Twitter graph (41M vertices, 1.5B edges)



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

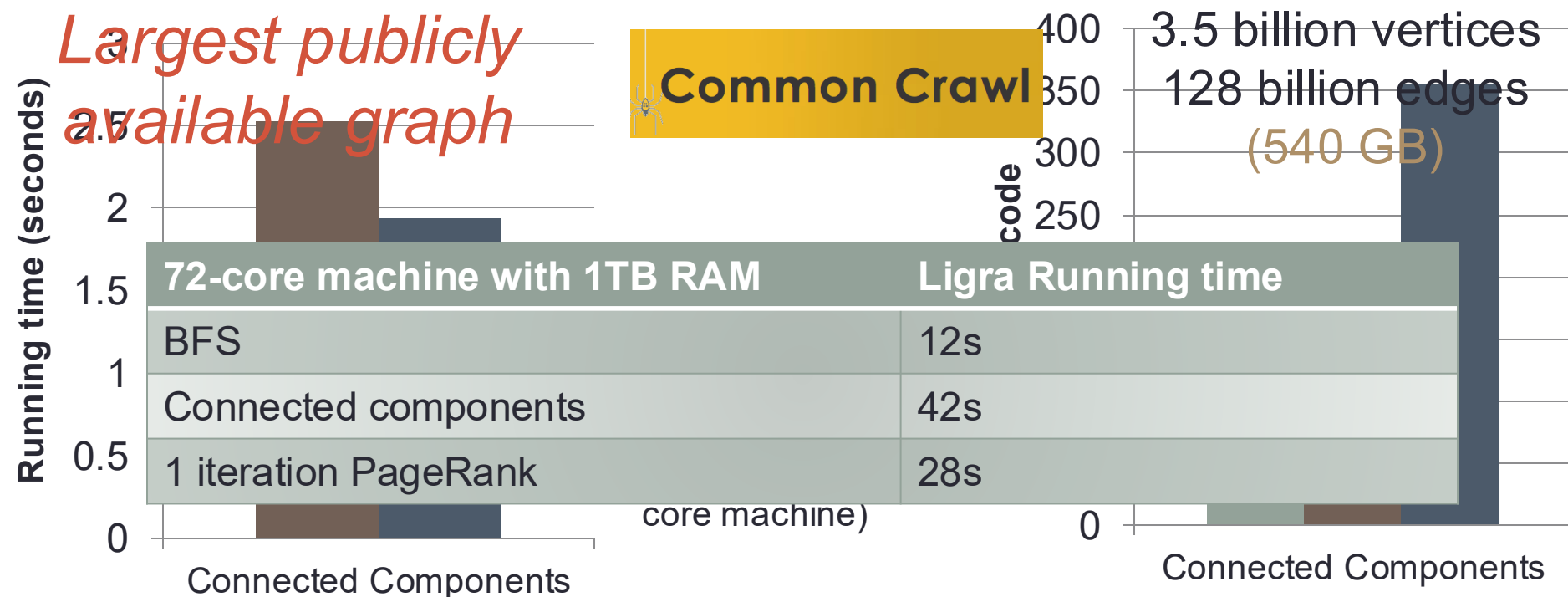
Connected Components Performance

Twitter graph (41M vertices, 1.5B edges)

Largest publicly available graph

Common Crawl

3.5 billion vertices
128 billion edges
(540 GB)



- Ligra's performance is close to hand-written code
- Faster than best existing system
- Subsequent systems have used Ligra's abstraction and hybrid traversal idea, e.g., Galois [SOSP '13], Polymer [PPoPP '15], Gunrock [PPoPP '16], Gemini [OSDI '16], GraphGrind [ICS '17], Grazelle [PPoPP '18]

Large Graphs

Amazon EC2

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
x1e.xlarge	4	12	122	1 x 120 SSD	\$0.834 per Hour
x1e.2xlarge	8	23	244	1 x 240 SSD	\$1.668 per Hour
x1e.4xlarge	16	47	488	1 x 480 SSD	\$3.336 per Hour
x1e.8xlarge	32	91	976	1 x 960	\$6.672 per Hour
x1e.16xlarge	64	179	1952	1 x 1920 SSD	\$13.344 per Hour
x1e.32xlarge	128	340	3904	2 x 1920 SSD	\$26.688 per Hour

- Most can fit on commodity shared memory machine

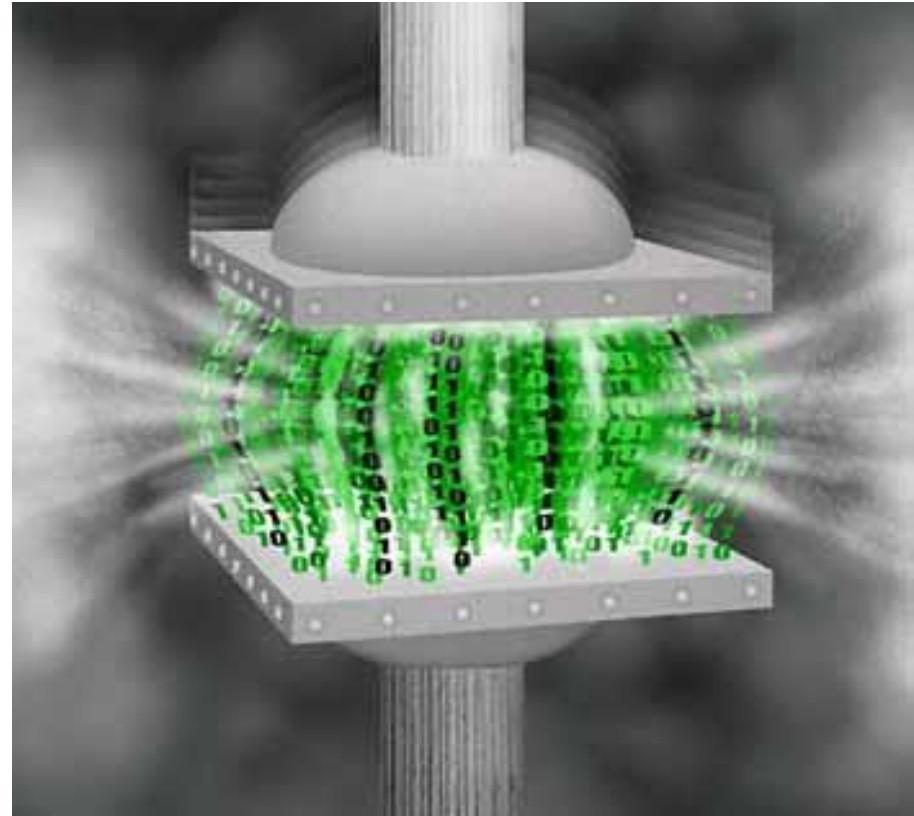
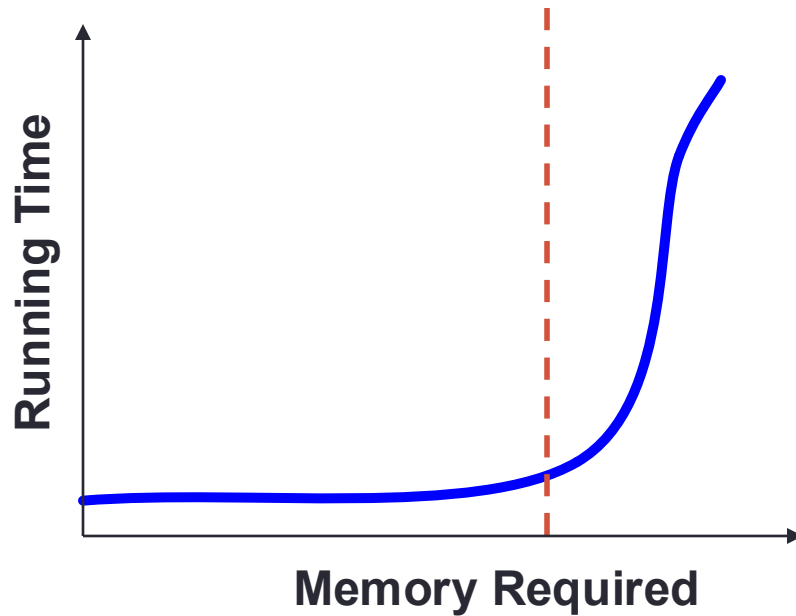


Example

Dell PowerEdge R930:

Up to 96 cores and 6 TB of RAM

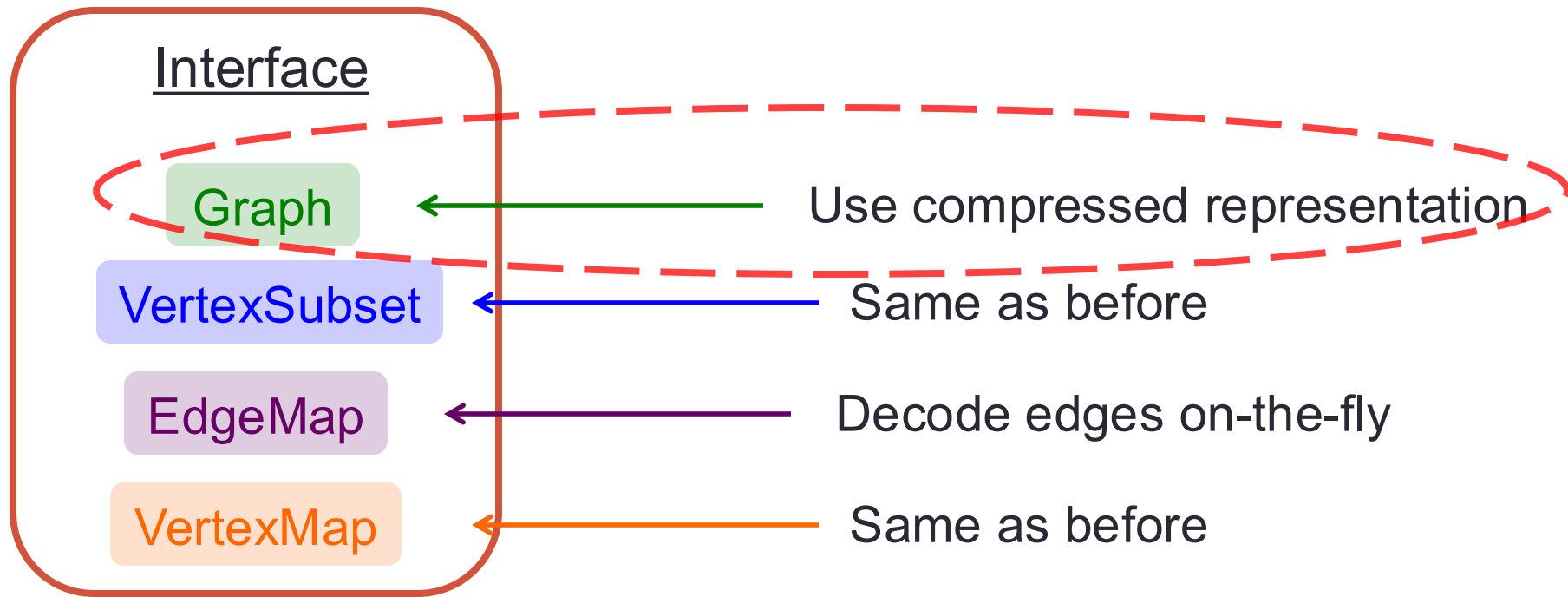
What if you don't have or can't afford that much memory?



Graph Compression

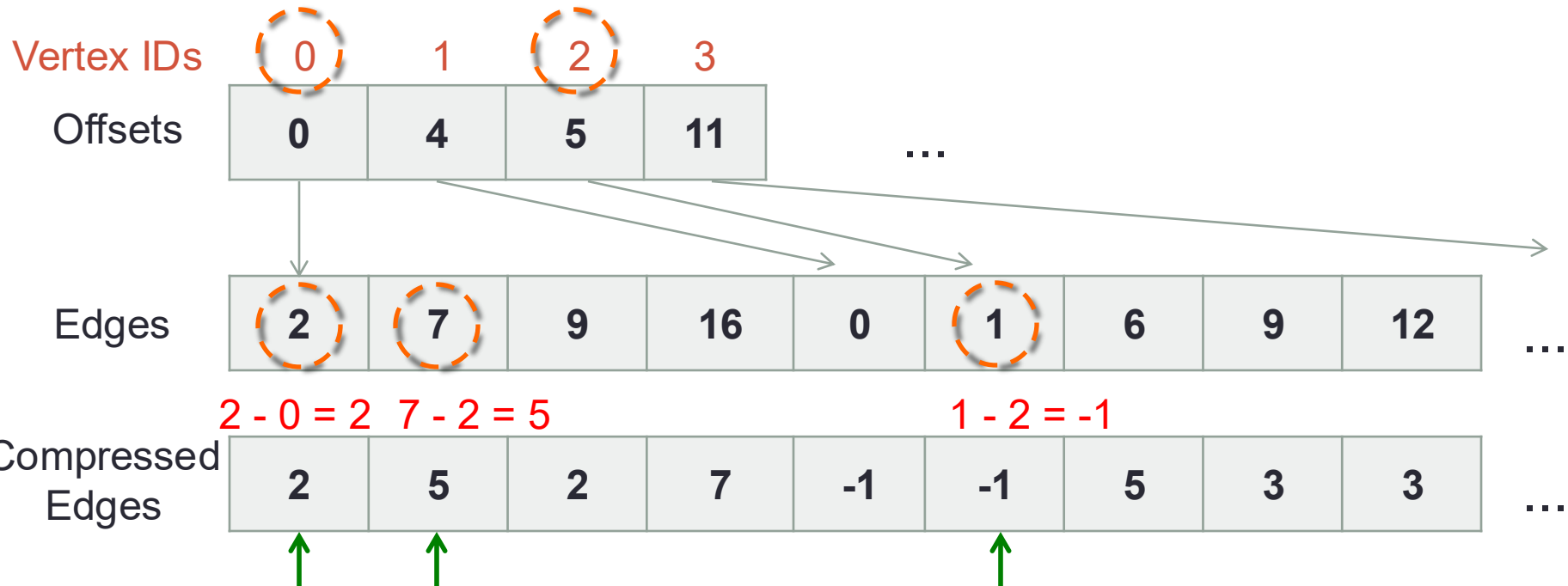
Ligra+: Adding Graph Compression to Ligra

Ligra+: Adding Graph Compression to Ligra



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

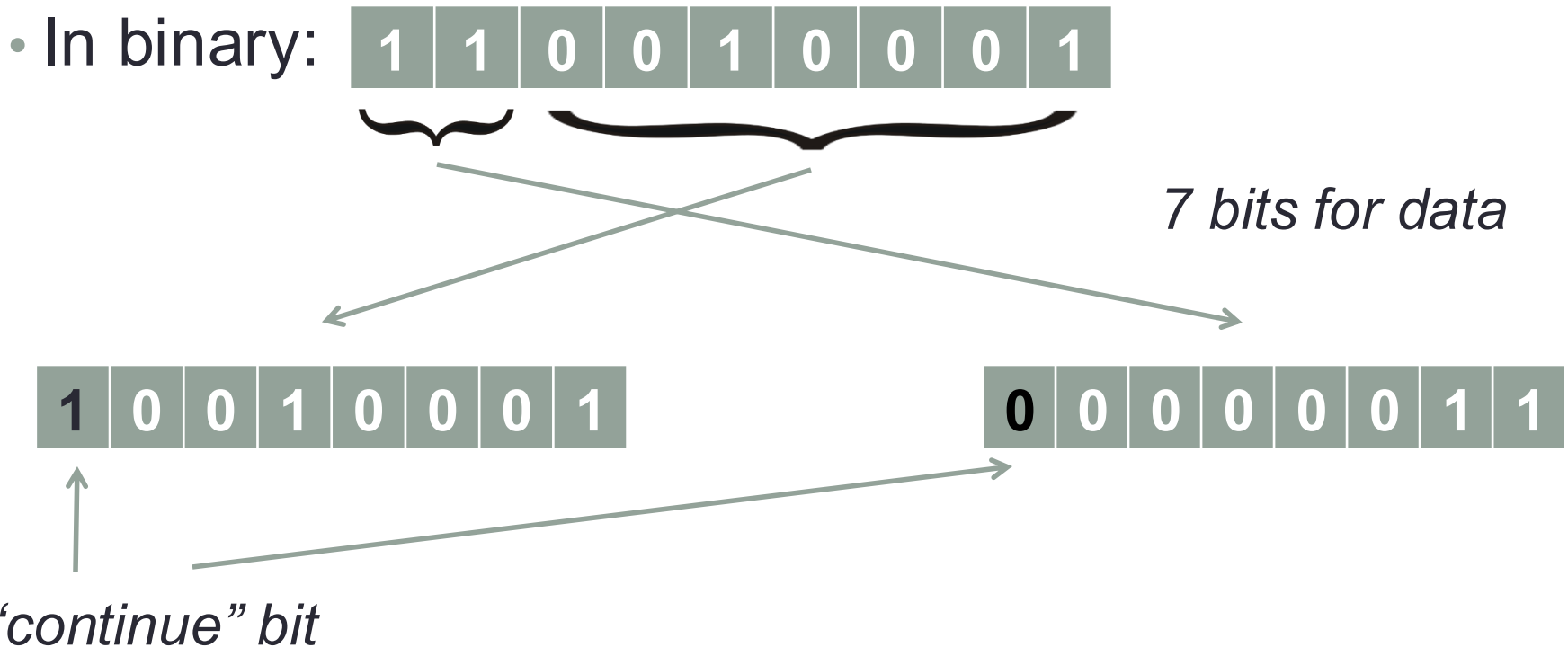
Graph representation



Sort edges and encode differences

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



Encoding optimization

- Another idea: get rid of “continue” bits



Number of bytes
required to encode
each integer

1

2

2

2

2

2

2

2

.....

Use run-length encoding

Header



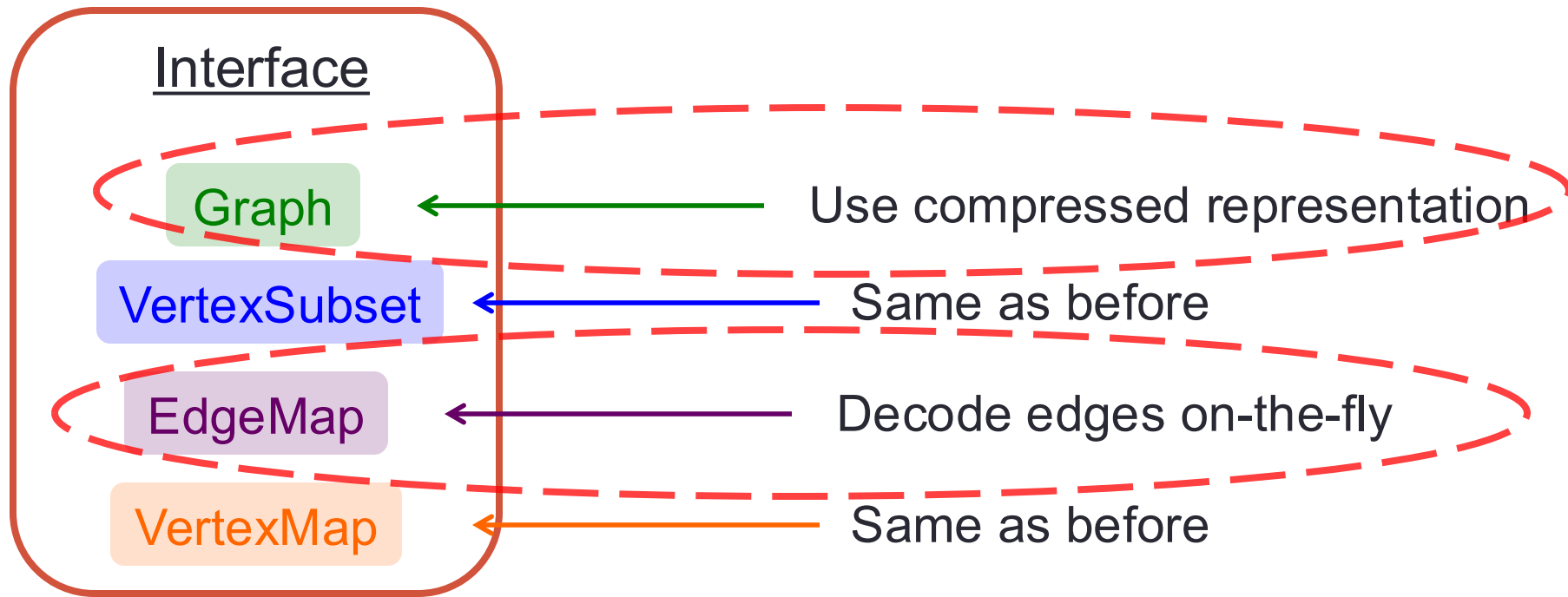
Integers in group
encoded in byte chunks

Number of bytes
per integer

Size of group
(max 64)

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

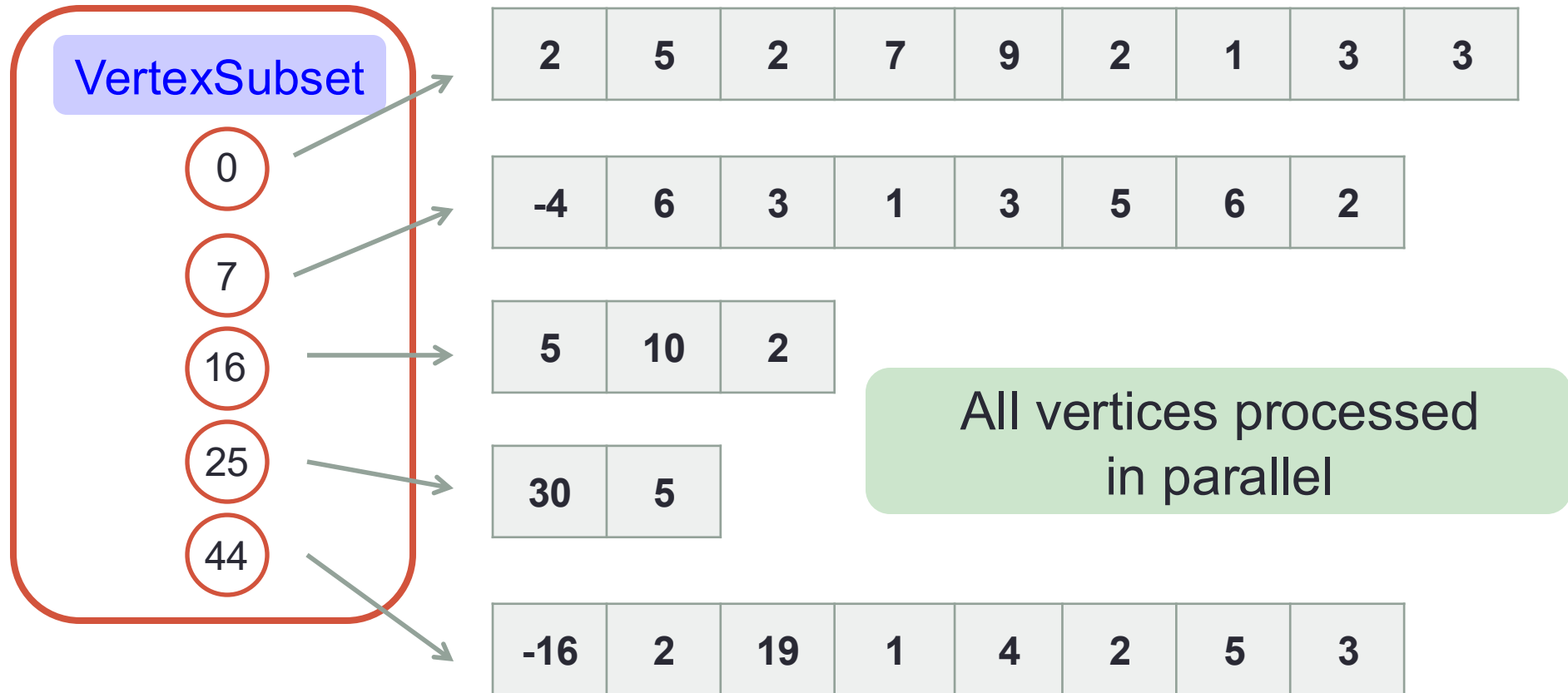
Ligra+: Adding Graph Compression to Ligra



- 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

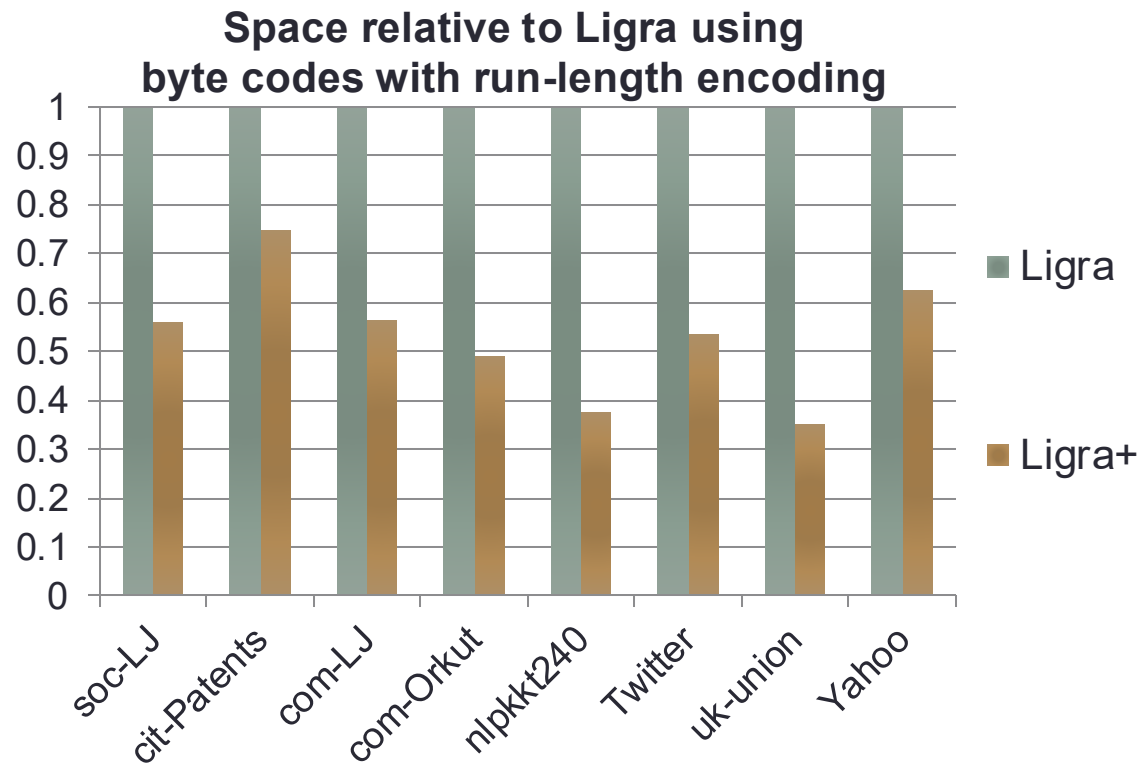
-1	2	4	3	16	2	27	5	8	19	4	1	87	14	12	1	9	10	...
----	---	---	---	----	---	----	---	---	----	---	---	----	----	----	---	---	----	-----

Encode first entry relative to source vertex

All chunks can be decoded in parallel!

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

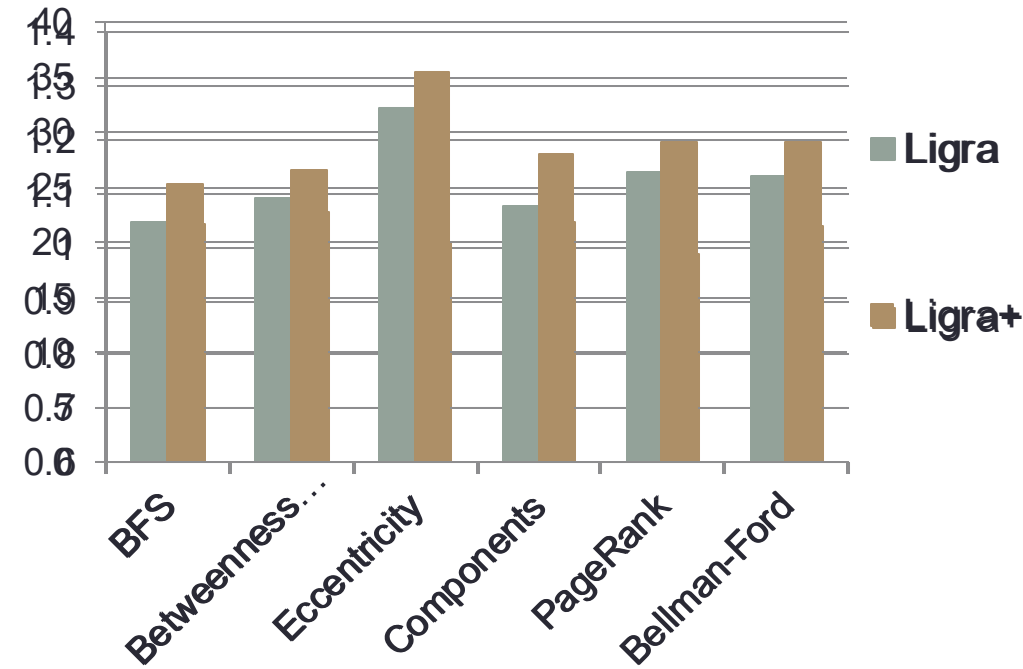
Ligra+ Space Savings



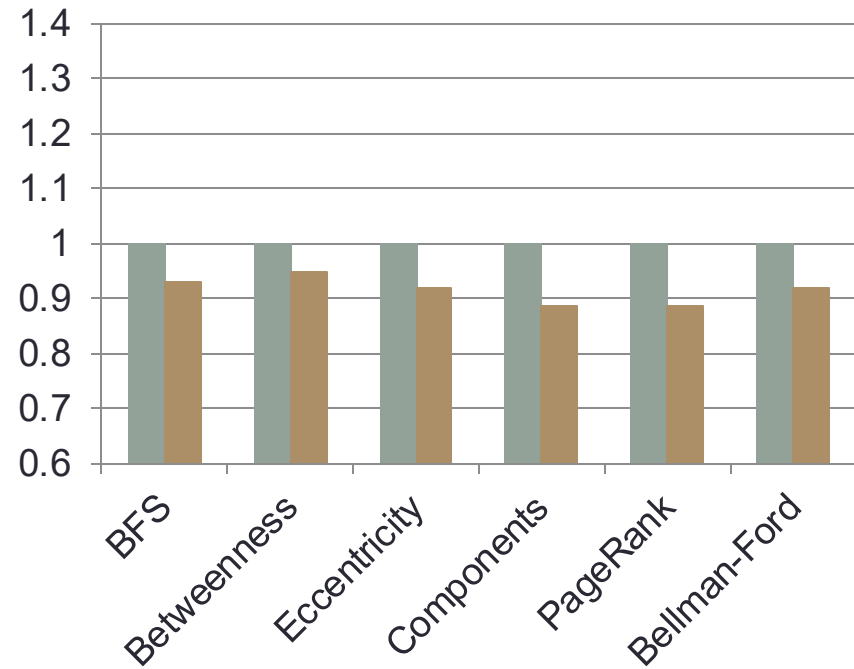
- Space savings of about 1.3—3x
- Could use more sophisticated schemes to further reduce space, but more expensive to decode
- Cost of decoding on-the-fly?

Ligra+ Performance

Single-core relative time to Ligra



40-core time relative to Ligra



- Cost of decoding on-the-fly?
- Memory subsystem is a scalability bottleneck in parallel as these graph algorithms are memory-bound
- Ligra+ decoding gets better parallel speed up

Ligra Summary

VertexSubset

VertexMap

EdgeMap

*Optimizations: Hybrid traversal
and graph compression*

Breadth-first search
Betweenness centrality
Connected components
Triangle counting
K-core decomposition
Maximal independent set
...

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

Simplicity, Performance, Scalability



Thank you!

J. Shun and G. E. Blelloch. *Ligra: A Lightweight Graph Processing Framework for Shared Memory*, Principles and Practice of Parallel Programming, 2013.

J. Shun, L. Dhulipala and G. E. Blelloch. *Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+*, Data Compression Conference, 2015.

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