# Big Data on Small Machines

Guy Blelloch
with: Aapo Kyrola, Julian Shun
Carnegie Mellon

# Question

- There has been a lot of emphasis on large clusters, and tools such as Map Reduce for "big data"
- But, **when might it be better to use a smaller machine such as a laptop, desktop, or rack mounted multi-core server**?
- Many potential advantages
  - More energy efficient
  - More cost effective
  - Easier to program
  - Easier to administer, reliability

# In This Talk

1. Graph analysis using multi-core servers (LIGRA, with Julian Shun, PPOPP '13)
2. Graph analysis using a laptops and disks (GraphChi, with Aapo Kyrola, OSDI '12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA '04)

# Why use Large Clusters

1. Data does not fit in memory of modest machines
2. Modest machines are not fast enough to process the data.

# **BIG** Data

**Sloan Sky Survey**: $7 \times 10^{13}$ bytes/year now
**$7 \times 10^{15}$** bytes/year in 2016

**Large Hadron Collider**:
150 million sensors x 40 million samples/sec
= **$6 \times 10^{16}$** samples/year

**Wallmart Database**: **$2.5*10^{15}$** bytes (predicted)
10 Billion Transactions/year

**YouTube** : 1.2 x 10^8 videos x $2 \times 10^7$ mbytes/video
= **$3 \times 10^{15}$** bytes

**Genome**: $4 \times 10^9$ bp/human x $4 \times 10^9$ humans = **$10^{19}$ bytes**

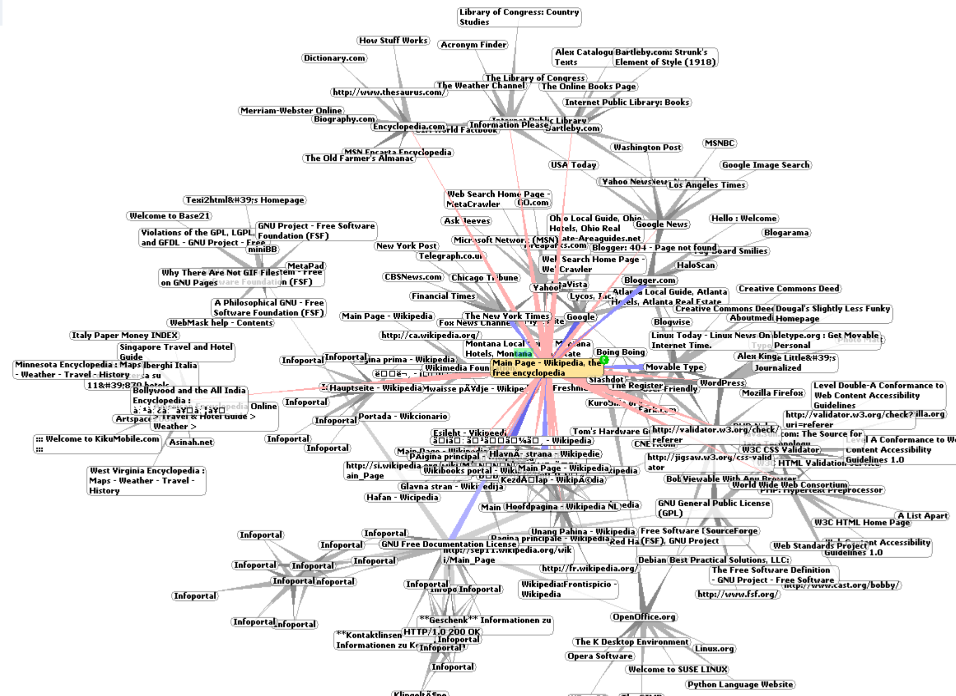
**But most analysis does not have to look at all the data**

# Large Data (Graphs)

- Web graph (centered around Wikipedia)

# Large Data (Graphs)

| | Edges | Uncompressed | Compressed |
|---|---|---|---|
| Publicly Available | 6 Billion | 60 GB | 6 GB |
| Twitter | 25 Billion | 250 GB | 25 GB |
| Facebook | 140 Billion | 1.4 TB | 140 GB |
| Web Graph (usefull) | 200 Billion? | 2 TB | 200 Gbytes |



# Large Text

- Jstor : 2 Million Docs – 100Gbytes
- PubMed 20 Million Docs – 100Gbytes
- Library of congress: $3 \times 10^7$ volumes x $10^5$/vol
  3 TB (compressed)

* All numbers estimated

# Machines

| | Cost | Main Memory | Secondary Memory |
|---|---|---|---|
| Laptop | $1K | 10 GB | 1 TB |
| Desktop | $4K | 100 GB | 10 TB |
| Server | $20K | 1 TB | 100 TB |
| 100 node Cluster | $200K | 10 TB | 1000 TB |

Twitter Graph : 250 Gbytes

Compressed Twitter Graph: 25 Gbytes

# Machines

| | Cost | Main Memory | Secondary Memory |
|---|---|---|---|
| Laptop | $1K | 10 GB | 1 TB |
| Desktop | $4K | 100 GB | 10 TB |
| Server | $20K | 1 TB | 100 TB |
| 100 node Cluster | $200K | 10 TB | 1000 TB |

Twitter Graph : 250 Gbytes

Compressed Twitter Graph: 25 Gbytes

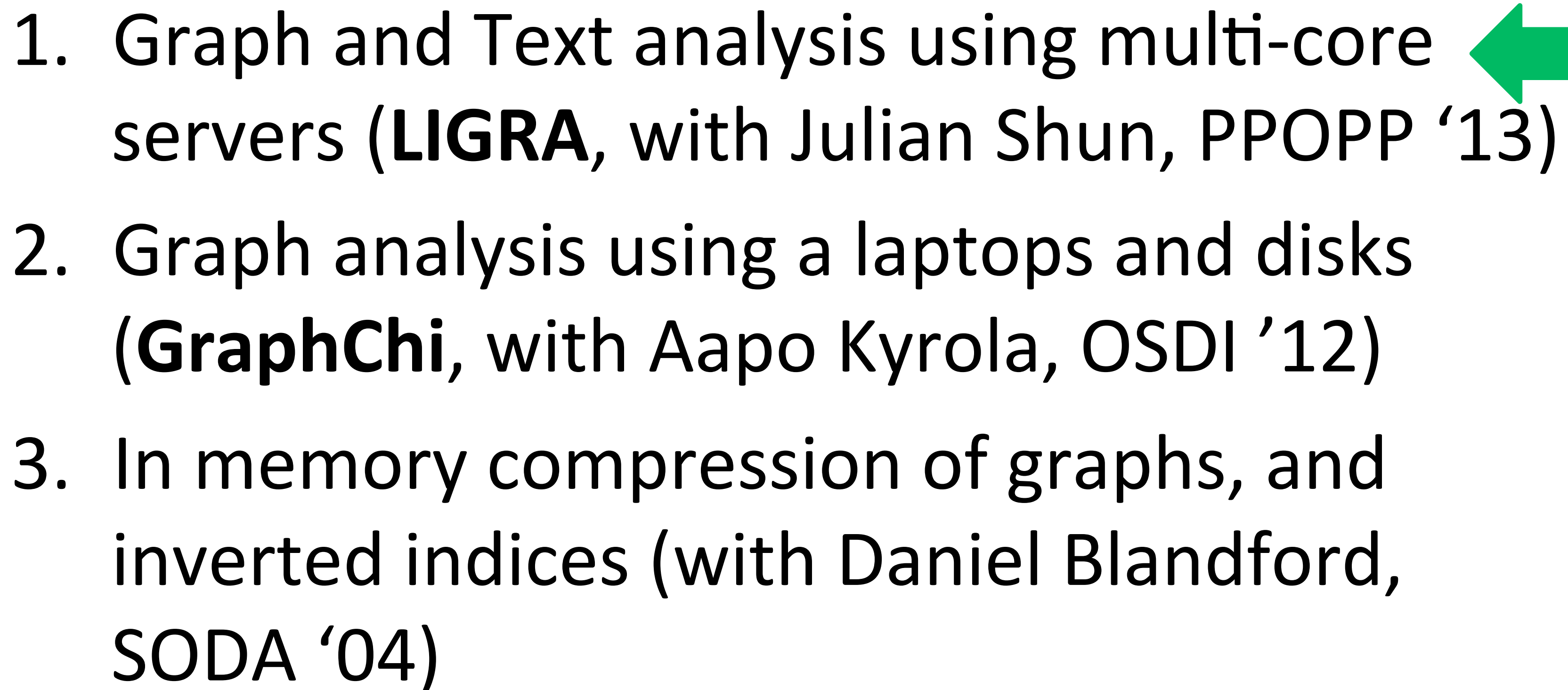
# Machines

| | Cost | Main Memory | Secondary Memory | Cores |
|---|---|---|---|---|
| Laptop | $1K | 10 GB | 1 TB | 4 |
| Desktop | $4K | 100 GB | 10 TB | 16 |
| Server | $20K | 1 TB | 100 TB | 64 |
| 100 node Cluster | $200K | 10 TB | 1000 TB | 800 |

Twitter Graph : 250 Gbytes

Compressed Twitter Graph: 25 Gbytes

# In This Talk



1. Graph and Text analysis using multi-core servers (**LIGRA**, with Julian Shun, PPOPP ‘13)
2. Graph analysis using a laptops and disks (**GraphChi**, with Aapo Kyrola, OSDI ’12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA ‘04)

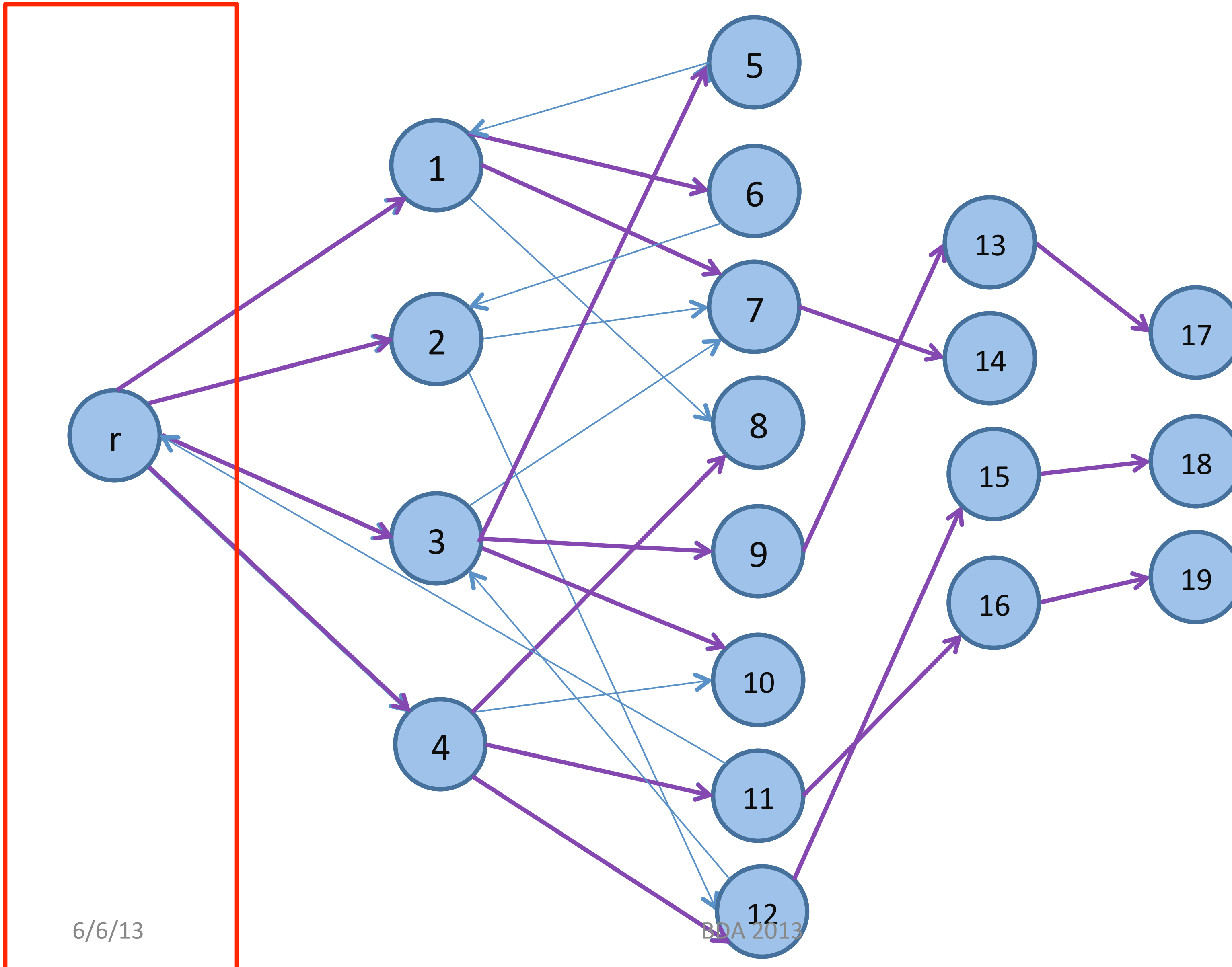
# Ligra

- Lightweight graph processing system for shared memory multicore machines
  - Lightweight: modest amount of code (about 1500 lines including comments)
  - Simple: framework only has 2 routines and one data structure
    - This is enough for a wide class of graph traversal algorithms!
  - Parallel: designed for shared memory systems
  - Efficient: outperform other graph systems by orders of magnitude, up to 39x speedup on 40 cores
  - Designed for synchronous computations

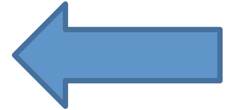
# Ligra Interface

- Vertex subset: represents a subset of vertices
  - Important for algorithms that process only a subset of vertices each iteration
  - Can keep around multiple subsets for same graph
  - Can use one subset for multiple graphs
- Vertex map: maps user boolean function over vertex subset
- Edge map: maps user boolean function over out-edges of vertex subset
- Note: system does not store vertex/edge data. User-defined vertex/edge data can be modified by the function passed to map

# Parallel Breadth First Search (BFS)



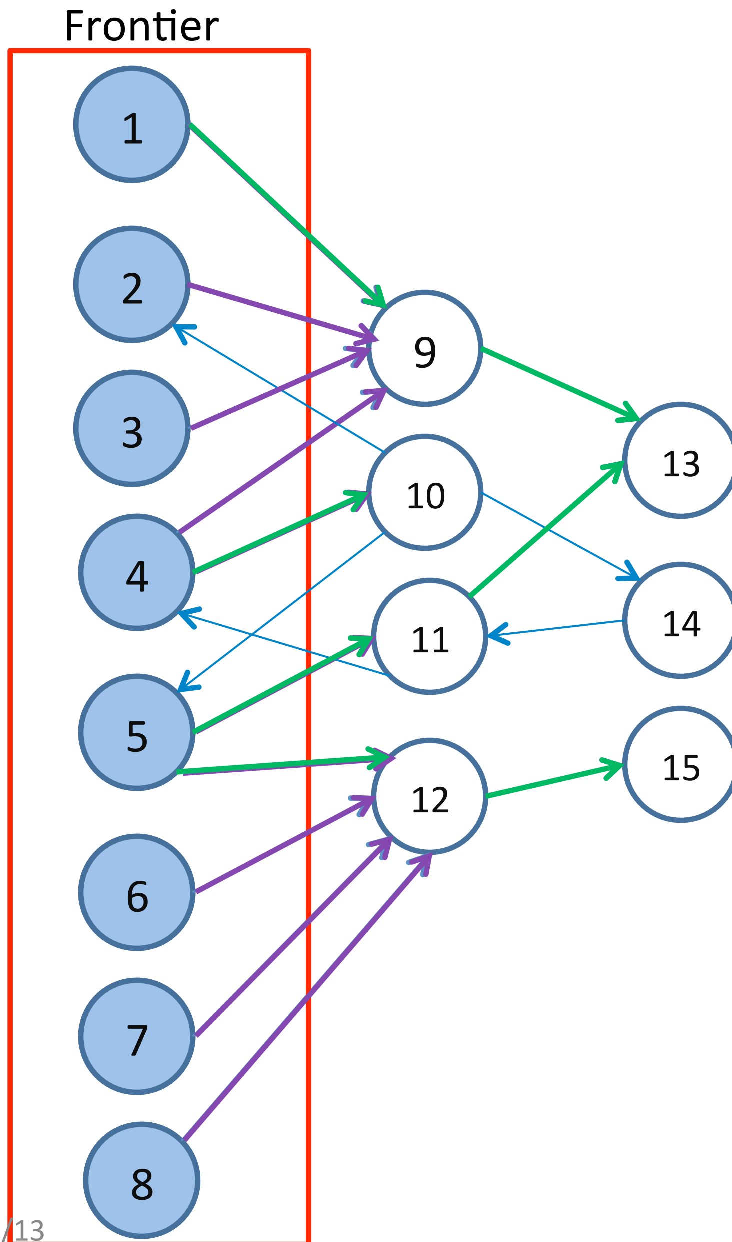
# Breadth-first search in Ligra



```
 1: Parents = {−1, . . . , −1}                          ▷ initialized to all -1's
 2:
 3: procedure UPDATE(s, d)
 4:     return (CAS(&Parents[d], −1 , s ))
 5:
 6: procedure COND(i)
 7:     return (Parents[i] == −1)
 8:
 9: procedure BFS(G, r)                                  ▷ r is the root
10:     Parents[r] = r
11:     Frontier = {r}          ▷ vertexSubset initialized to contain only r
12:     while (SIZE(Frontier) ≠ 0) do
13:         Frontier = EDGEMAP(G, Frontier, UPDATE, COND)
```

- compare-and-swap = CAS; takes three arguments (*addr, oldval, newval*) and atomically updates value at *addr* to *newval* if its value was *oldval*. Returns true if updated, false otherwise.
- Line 4 attempts to sets vertex *s* to be vertex *d*''s parent if unvisited
- Cond is used to check if unvisited (-1 means unvisited)
- EdgeMap takes frontier, outputs next frontier

# Two methods for BFS

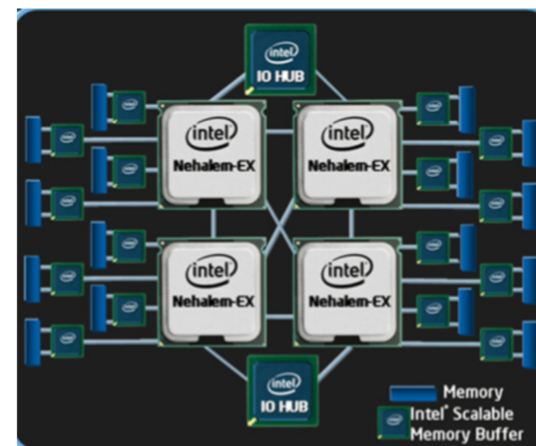



Idea due to Beamer, Asanovic and Patterson (2012):

- 1st (Sparse) method better for small frontiers
- 2nd (Dense) method better when frontier is large and many vertices have been visited
- Switch between the two approaches based on frontier size

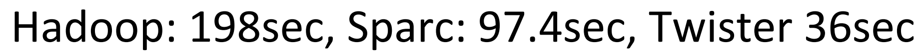
# Experiments

- Used a variety of artificial and real-world graphs
  - Largest is Yahoo web graph with 1.4 billion vertices and 6.6 billion edges
- Implementations in Cilk Plus (extension to C++), 1500 lines of code for the system
- Using 40-core Intel Nehalem based machine
- Good speedup, up to 39x (PageRank)



# Comparison to other graph processing systems

| | **Ligra: 40 core Performance** | **vs.** | **Machines** | **Performance** |
|---|---|---|---|---|
| Breadth first search | 2.5B edges/sec | KDT | 64 x 4 core Intel Nehalem | 473M edges/sec |
| Approximate betweenness centrality | 526M edges/sec | KDT | 24 x 12 core AMD Opteron processors | 125M edges/sec |
| Page Rank (15 lines of code) | 2.91 sec/iteration for 1.5B edge Twitter graph | Powergraph | 8 x 64-core machines | 3.6 sec/iteration for 1.5B edge Twitter graph |
| Shortest Paths | 1.68B edges in under 2 seconds | Pregel | 300 multicore commodity PCs | 1B edges in 20 seconds |


Hadoop: 198sec, Sparc: 97.4sec, Twister 36sec

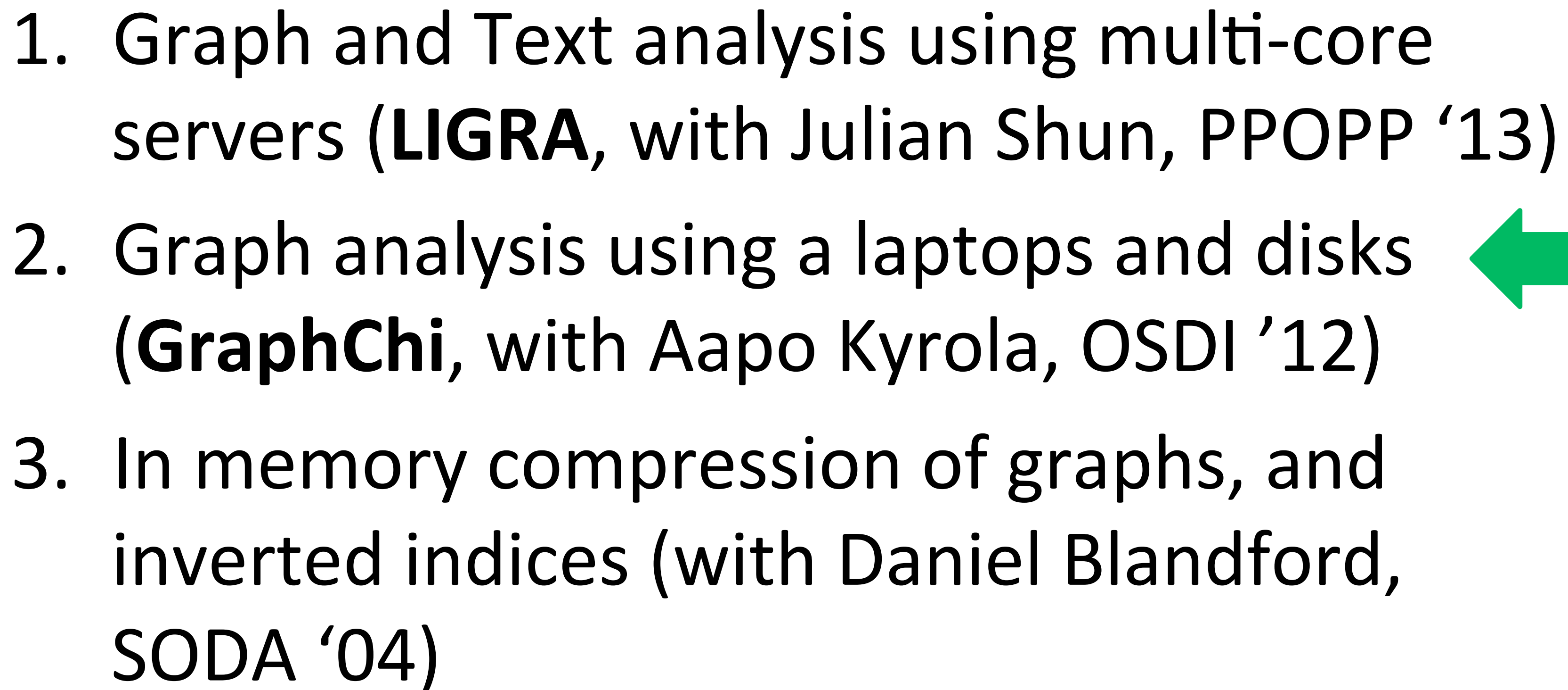
# Ligra - Conclusions

- Lightweight: framework is only about 1500 lines of code including comments
- Simple: Leads to simple implementations of graph traversal algorithms
  - Abstract main components of graph traversal algorithms into two functions
- Efficient: Up to orders of magnitude faster than those of other graph processing systems
  - Not much slower than highly-optimized application-specific code

# Ligra - Conclusions

- Limitations:
  - Does not work well with dynamic graphs
  - Not for asynchronous computations
- Future work:
  - Address limitations
  - Extending to GPUs

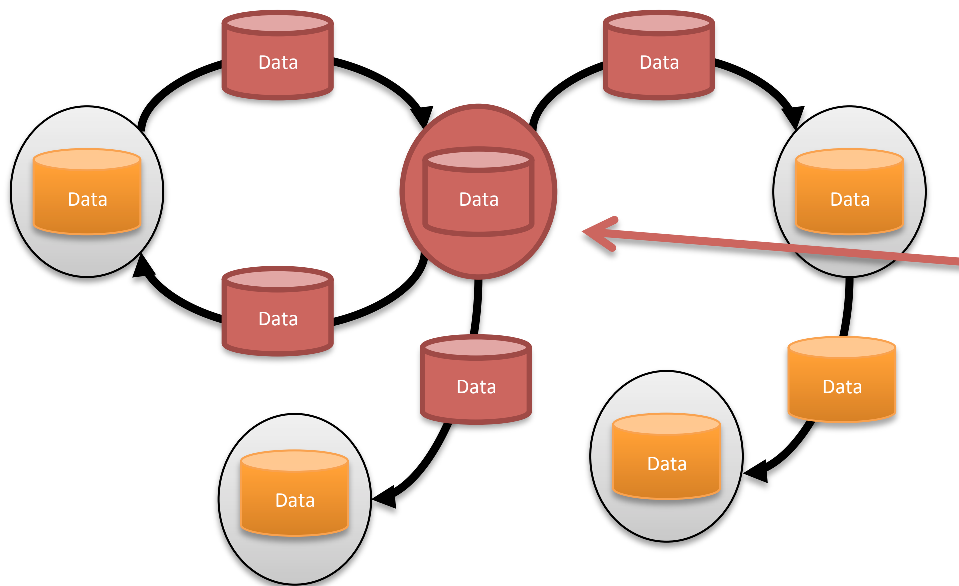
# In This Talk



1. Graph and Text analysis using multi-core servers (**LIGRA**, with Julian Shun, PPOPP ‘13)
2. Graph analysis using a laptops and disks (**GraphChi**, with Aapo Kyrola, OSDI ’12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA ‘04)

# Vertex-centric Programming

- "Think like a vertex"
- Popularized by the Pregel and GraphLab projects
  - Historically, systolic computation and the Connection Machine




**MyFunc**(vertex)
{ // modify neighborhood }

# Disk-Based Graph Algorithms

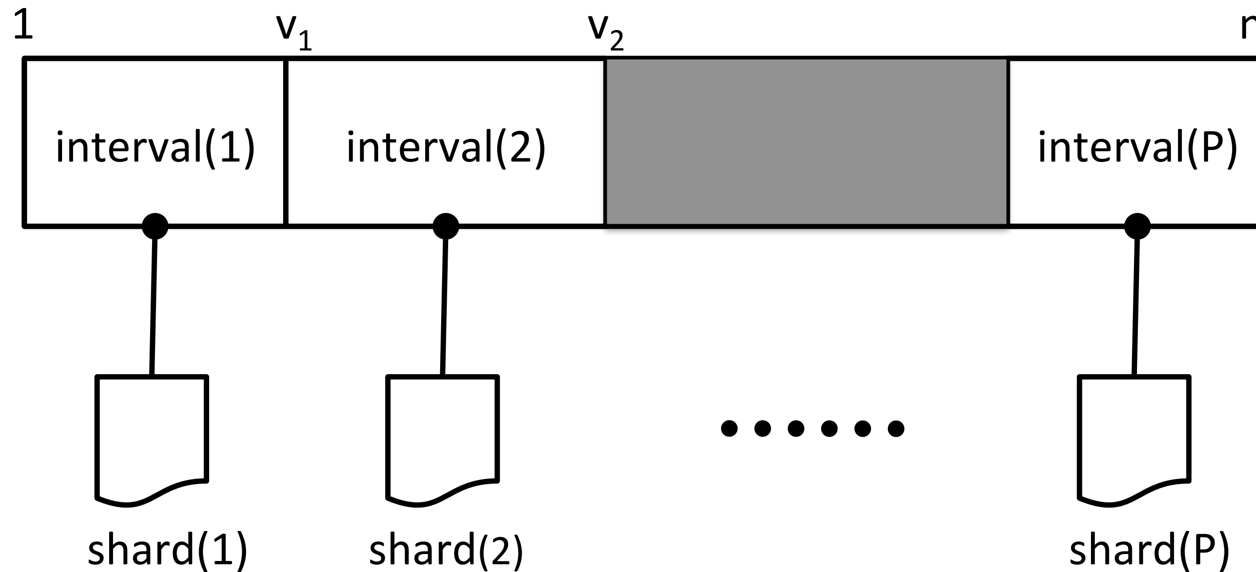
**Main Challenge**: Random Access

5ms Disk, .05ms SSD, .0001ms memory

**GraphChi Solution**: Parallel Sliding Windows

# PSW: Shards and Intervals

- Vertices are numbered from 1 to n
  - **P** intervals, each associated with a **shard** on disk.
  - **sub-graph =** interval of vertices

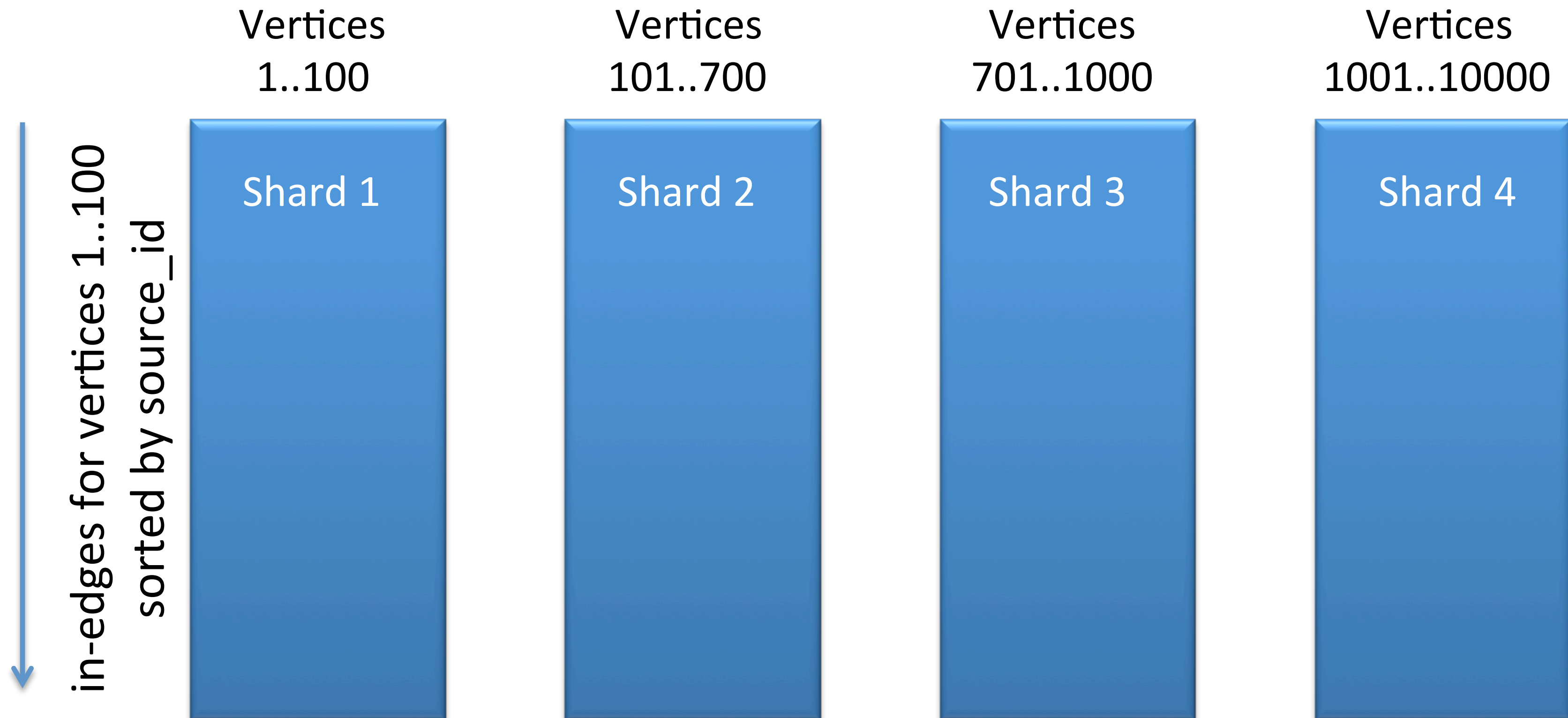


roystoncartoons.com

“I can’t see the point of it.”

# PSW: Layout

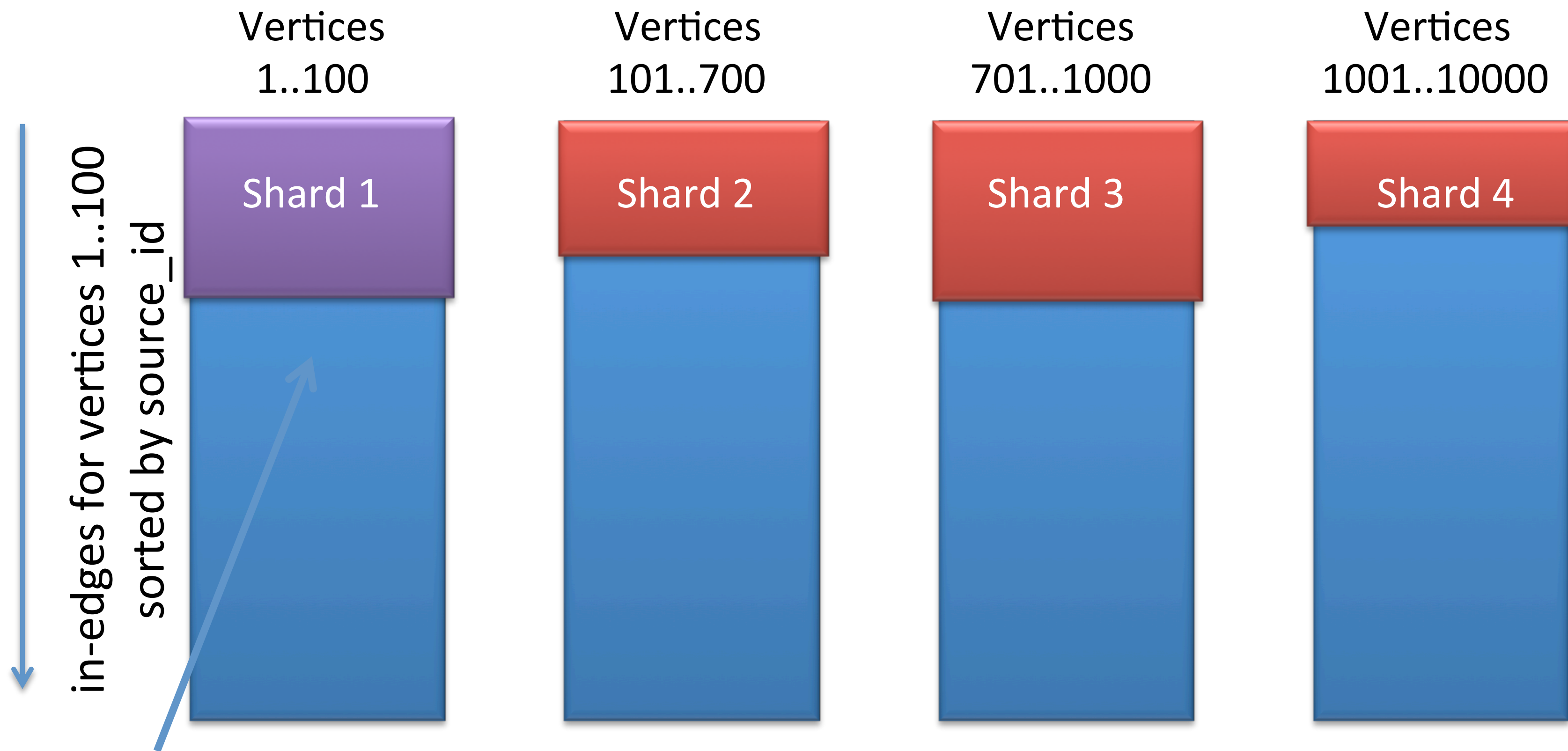
Shard: in-edges for **interval** of vertices; sorted by source-id




Shards small enough to fit in memory; balance size of shards

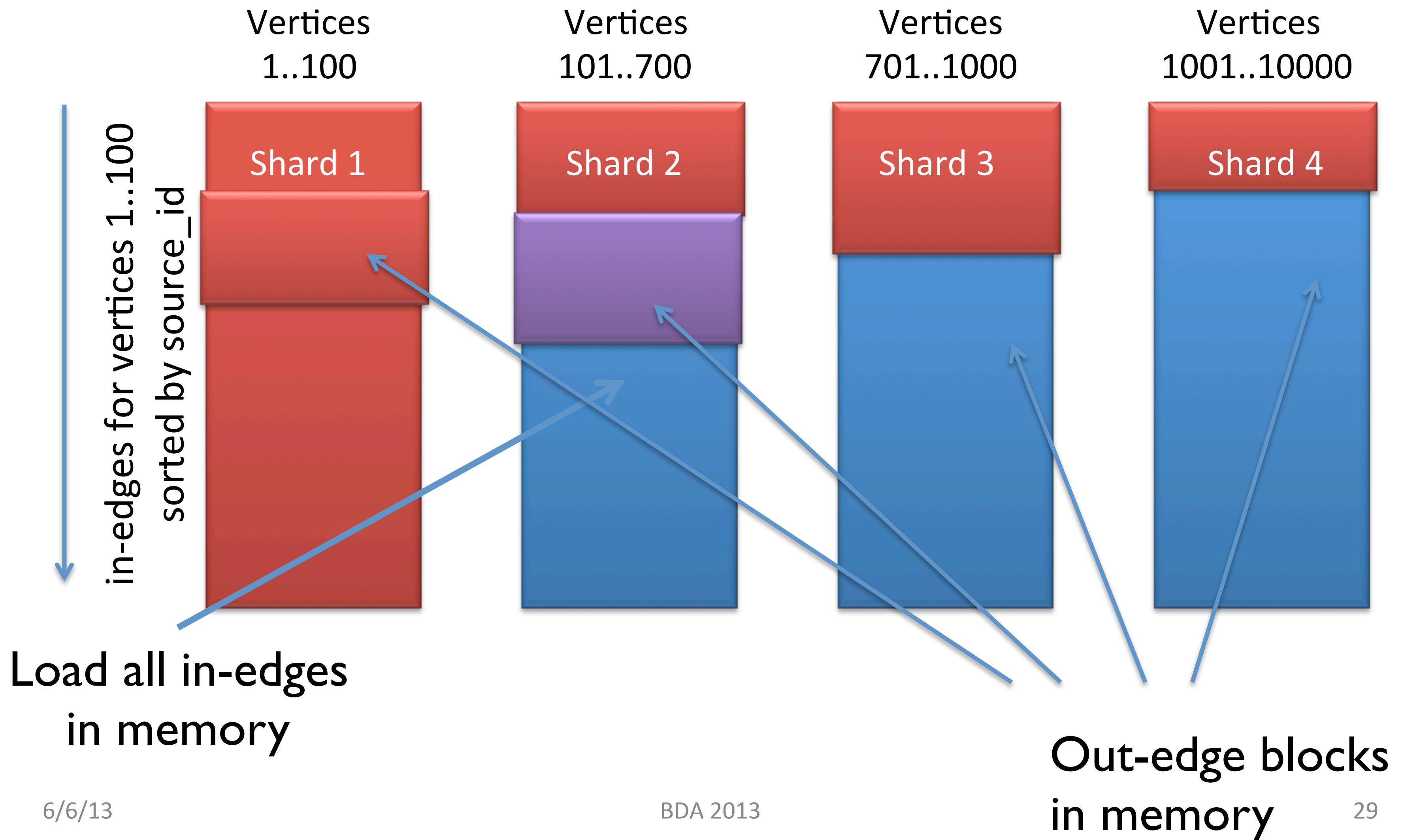
# PSW: Loading Sub-graph

**Load subgraph for vertices 1..100**



# PSW: Loading Sub-graph

**Load subgraph for vertices 101..700**



# GraphChi

- C++ implementation: 8,000 lines of code
  - Java-implementation also available (~ 2-3x slower), with a Scala API.
- Several optimizations to PSW (see paper).

Source code and examples: http://graphchi.org

# Evaluation: Is PSW expressive enough?

**Graph Mining**

- Connected components
- Approx. shortest paths
- Triangle counting
- Community Detection

**SpMV**

- PageRank
- Generic

**Recommendations**

- Random walks

**Collaborative Filtering (by Danny Bickson)**

- ALS
- SGD
- Sparse-ALS
- SVD, SVD++
- Item-CF

*+ many more*

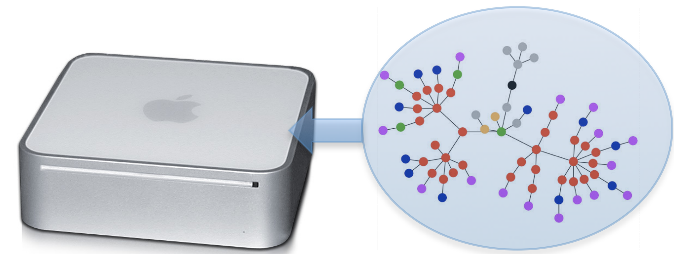
**Probabilistic Graphical Models**

- Belief Propagation

Algorithms implemented for GraphChi (Oct 2012)

# Experiment Setting

- Mac Mini (Apple Inc.)
  - 8 GB RAM
  - 256 GB SSD, 1TB hard drive
  - Intel Core i5, 2.5 GHz



- Experiment graphs:

| Graph | Vertices | Edges | P (shards) | Preprocessing |
|---|---|---|---|---|
| live-journal | 4.8M | **69M** | 3 | 0.5 min |
| netflix | 0.5M | **99M** | 20 | 1 min |
| twitter-2010 | 42M | **1.5B** | 20 | 2 min |
| uk-2007-05 | 106M | **3.7B** | 40 | 31 min |
| uk-union | 133M | **5.4B** | 50 | 33 min |
| yahoo-web | 1.4B | **6.6B** | 50 | 37 min |

OSDI'12

# PowerGraph Comparison

- **PowerGraph / GraphLab 2** outperforms previous systems by a wide margin on natural graphs.
- With 64 more machines, 512 more CPUs:
  - **Pagerank**: 40x faster than GraphChi
  - **Triangle counting**: 30x faster than GraphChi.

vs.

GraphChi

GraphChi has state-of-the-art performance / CPU.

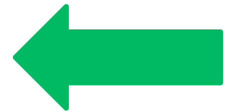
# Hybrid Approach

Possible approach for graphs with metadata:

- Use a server with e.g. 1 TB mem, 100TB Disk
- In memory for main graph
- Disk for meta data

# In This Talk

1. Graph and Text analysis using multi-core servers (LIGRA, with Julian Shun, PPOPP '13)
2. Graph analysis using a laptops and disks (GraphChi, with Aapo Kyrola, OSDI '12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA '04)



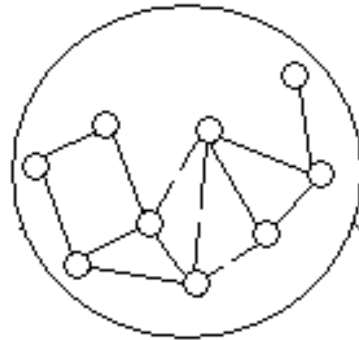
# Compressing Graphs

- **Goal**: To represent large graphs compactly while supporting queries efficiently
  - e.g., adjacency and neighbor queries
  - want to do significantly better than adjacency lists (e.g. a factor of 10 less space, about the same time)
- **Applications**:
  - Large web graphs
  - Large meshes
  - Phone call graphs

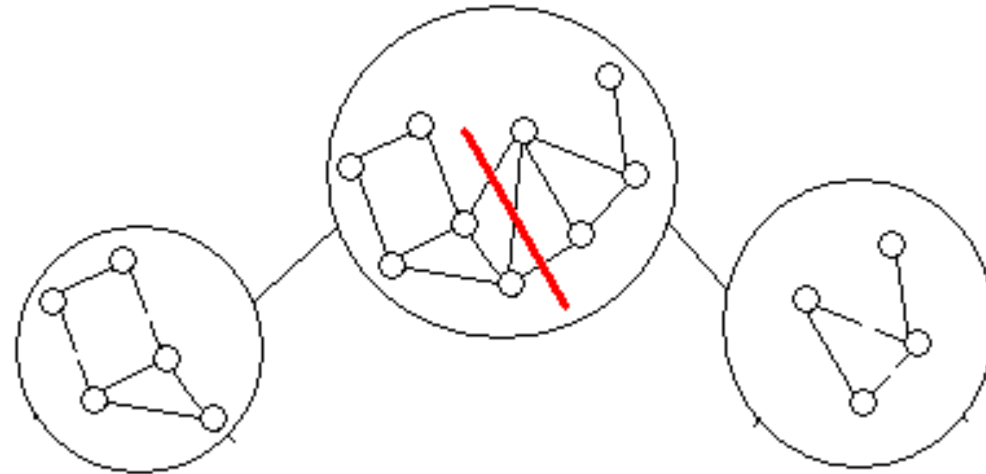
# Main Ideas

- Number vertices so adjacent vertices have similar numbers
  - Use separators to do this
- Use difference coding on adjacency lists
- Use efficient data structure for indexing

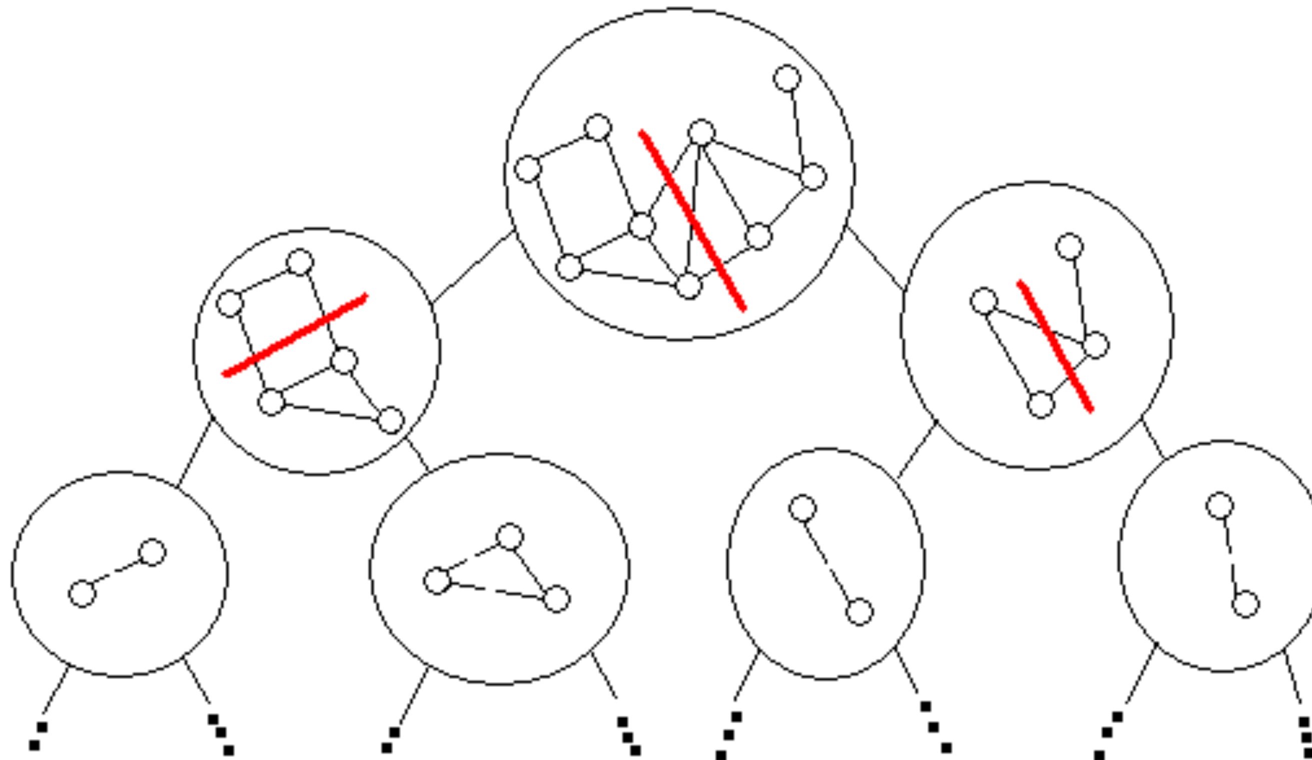
# Renumbering with Edge Separators



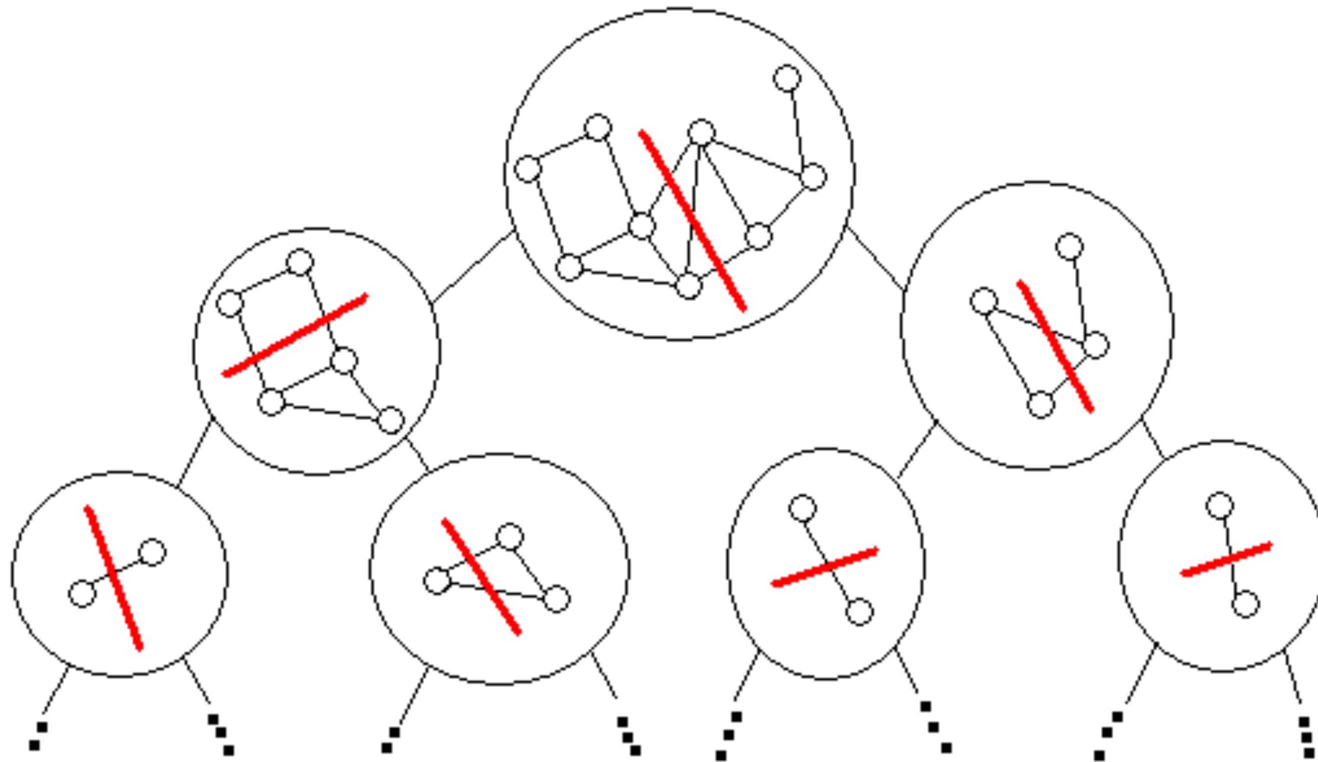
# Renumbering with Edge Separators



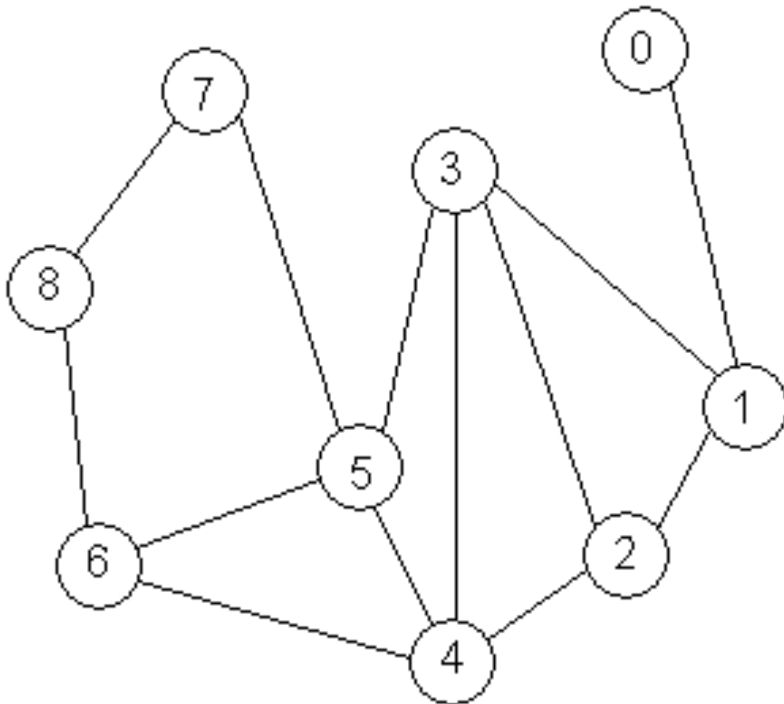
# Renumbering with Edge Separators



# Renumbering with Edge Separators



# Compressed Adjacency Tables




| # | D | Neighbors | Differences |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 3 | 0 2 3 | -1 2 1 |
| 2 | 3 | 1 3 4 | -1 2 1 |
| 3 | 4 | 1 2 4 5 | -1 1 2 1 |
| 4 | 4 | 2 3 5 6 | -2 1 2 1 |
| 5 | 4 | 3 4 6 7 | -2 1 2 1 |
| 6 | 3 | 4 5 8 | -2 1 3 |
| 7 | 2 | 5 8 | -2 3 |
| 8 | 2 | 6 7 | -2 1 |

# Log-sized Codes

- **Log-sized code**: Any prefix code that takes O(log(d)) bits to represent an integer d.
- Gamma code, delta code, skewed Bernoulli code

**Example: Gamma code**

Prefix: unary code for $\lfloor \log d \rfloor$

Suffix: binary code for $d-2^{\lfloor \log d \rfloor}$

(binary code for d, except leading 1 is implied)

| Decimal | Gamma | |
|---|---|---|
| 1 | 1 | |
| 2 | 01 | 0 |
| 3 | 01 | 1 |
| 4 | 001 | 00 |
| 5 | 001 | 01 |
| 6 | 001 | 10 |
| 7 | 001 | 11 |
| 8 | 0001 | 000 |

# Theorem (edge separators)

- Any class of graphs that allows $O(n^c)$ edge separators can be compressed to $O(n)$ bits with $O(1)$ access time using:
  - Difference coded adjacency lists
  - $O(n)$-bit indexing structure

# Performance: Adjacency Table

| | dfs | | metis-cf | | bu-bpq | | bu-cf | |
|---|---|---|---|---|---|---|---|---|
| | $T_d$ | Space | $T/T_d$ | Space | $T/T_d$ | Space | $T/T_d$ | Space |
| auto | 0.79 | 9.88 | 153.11 | 5.17 | 7.54 | 5.90 | 14.59 | 5.52 |
| feocean | 0.06 | 13.88 | 388.83 | 7.66 | 17.16 | 8.45 | 34.83 | 7.79 |
| m14b | 0.31 | 10.65 | 181.41 | 4.81 | 8.16 | 5.45 | 15.32 | 5.13 |
| ibm17 | 0.44 | 13.01 | 136.43 | 6.18 | 11.0 | 6.79 | 20.25 | 6.64 |
| ibm18 | 0.48 | 11.88 | 129.22 | 5.72 | 9.5 | 6.24 | 17.29 | 6.13 |
| CA | 0.76 | 8.41 | 382.67 | 4.38 | 14.61 | 4.90 | 35.21 | 4.29 |
| PA | 0.43 | 8.47 | 364.06 | 4.45 | 13.95 | 4.98 | 33.02 | 4.37 |
| googleI | 1.4 | 7.44 | 186.91 | 4.08 | 12.71 | 4.18 | 40.96 | 4.14 |
| googleO | 1.4 | 11.03 | 186.91 | 6.78 | 12.71 | 6.21 | 40.96 | 6.05 |
| lucent | 0.04 | 7.56 | 390.75 | 5.52 | 19.5 | 5.54 | 45.75 | 5.44 |
| scan | 0.12 | 8.00 | 280.25 | 5.94 | 23.33 | 5.76 | 81.75 | 5.66 |
| **Avg** | | 10.02 | **252.78** | 5.52 | **13.65** | 5.86 | **34.54** | 5.56 |

Time is to create the structure, normalized to time for DFS

# Performance: Overall

| Graph | Array | | List | | bu-cf/semi | |
|---|---|---|---|---|---|---|
| | time | space | time | space | time | space |
| auto | 0.24 | 34.2 | 0.61 | 66.2 | 0.51 | 7.17 |
| feocean | 0.04 | 37.6 | 0.08 | 69.6 | 0.09 | 11.75 |
| m14b | 0.11 | 34.1 | 0.29 | 66.1 | 0.24 | 6.70 |
| ibm17 | 0.15 | 33.3 | 0.40 | 65.3 | 0.34 | 7.72 |
| ibm18 | 0.14 | 33.5 | 0.38 | 65.5 | 0.32 | 7.33 |
| CA | 0.34 | 43.4 | 0.56 | 75.4 | 0.58 | 11.66 |
| PA | 0.19 | 43.3 | 0.31 | 75.3 | 0.32 | 11.68 |
| googleI | 0.24 | 37.7 | 0.49 | 69.7 | 0.45 | 7.86 |
| googleO | 0.24 | 37.7 | 0.50 | 69.7 | 0.51 | 9.90 |
| lucent | 0.02 | 42.0 | 0.04 | 74.0 | 0.05 | 11.87 |
| scan | 0.04 | 43.4 | 0.06 | 75.4 | 0.08 | 12.85 |

**time is for one DFS**

# Conclusions

- For many applications of "large data", data can fit in the memory of a server or disk of a laptop.
- Speed can be improved on a single multicore server over a distributed system, and significantly more energy efficient.
- Code can be simpler and more general
- Disk algorithms are likely the most energy efficient, so good for high bandwidth embarassigly parallel applications

# Thank you!