Big Data on Smallish Machines

Guy Blelloch
with: Dan Blandford, 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**?
- Potential advantages
  - (much) More energy efficient
  - More cost effective
  - Easier to program, especially for general purpose
  - Easier to administer, more reliable

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: 7×10^{13} bytes/year now

7×10^{15} bytes/year in 2016

Large Hadron Collider:

150 million sensors x 40 million samples/sec

= **6×10^{16}** samples/year

Wallmart Database: **$2.5*10^{15}$** bytes (predicted)

10 Billion Transactions/year

YouTube : 1.2 x 10^8 videos x 2×10^7 mbytes/video

= **3×10^{15}** bytes

Genome: 4×10^9 bp/human x 4×10^9 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: 3×10^7 volumes x 10^5/vol
  1 TB (compressed)

* All numbers estimated

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

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

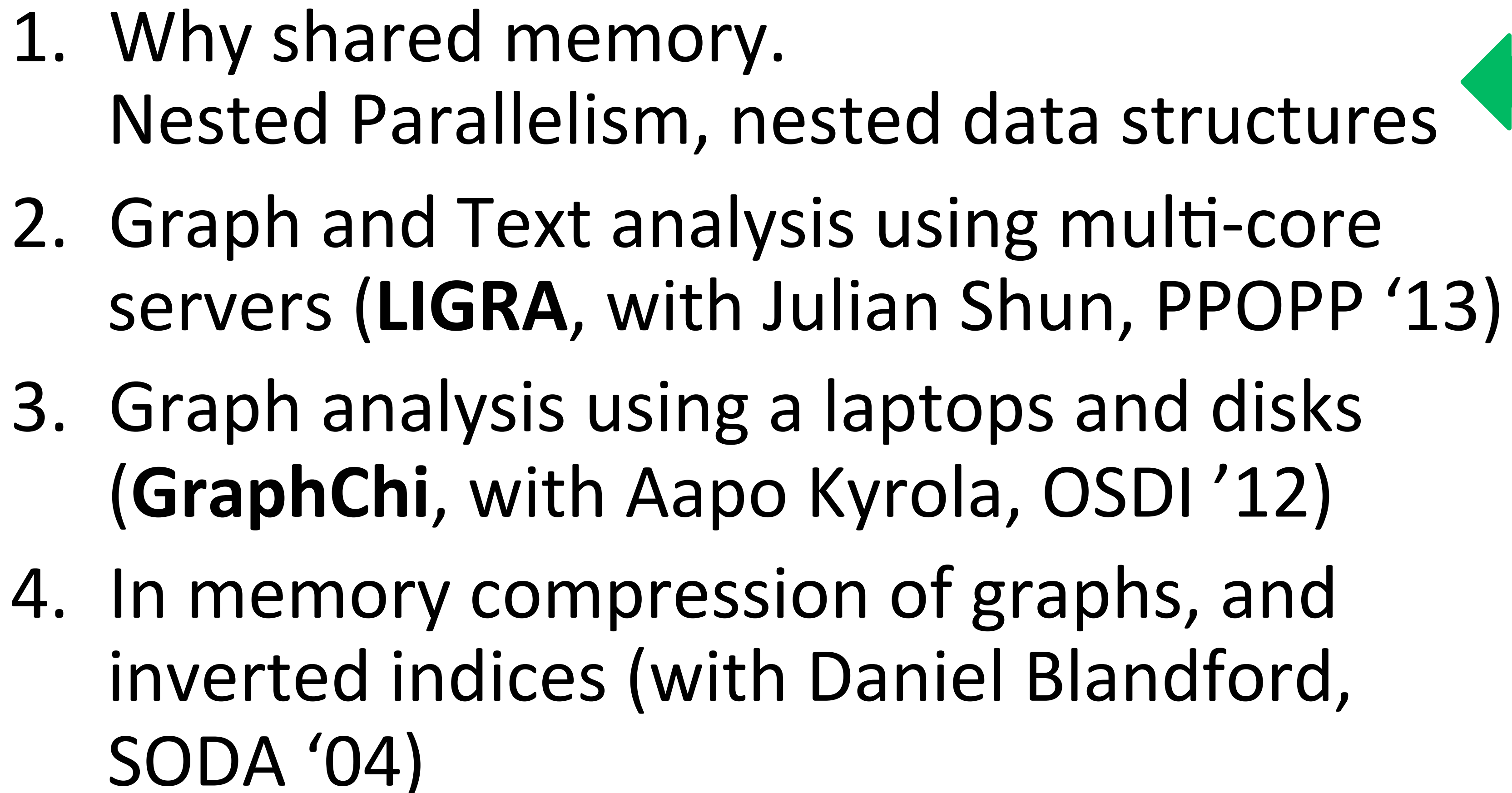
Machines

	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



Why Shared Memory

General Purpose

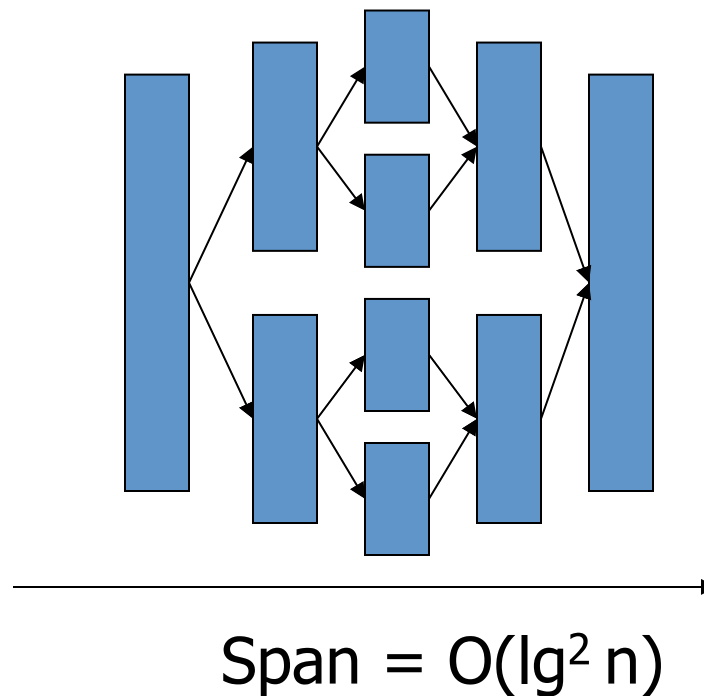
- Nested parallelism
- Pipelined parallelism
- Pointer-based structures
  - Various trees, hypergraphs, combinations of trees and graphs,
- Large knowledge-base of algorithms
- Simple cost models
- All reasonably easy

Parallel Quicksort

```
function quicksort(S) =
if (#S <= 1) then S
else let
  a = S[rand(#S)];
  S1 = {e in S | e < a};
  S2 = {e in S | e == a};
  S3 = {e in S | e > a};
  R = {quicksort(v) : v in [S1, S3]};
in R[0] ++ S2 ++ R[1];
```

Quicksort Complexity

Computational DAG



Work = O(n log n)

Q_1 = O(n/B log (n/M))

Span = O($\lg^2$ n)

Complexities are w.h.p.

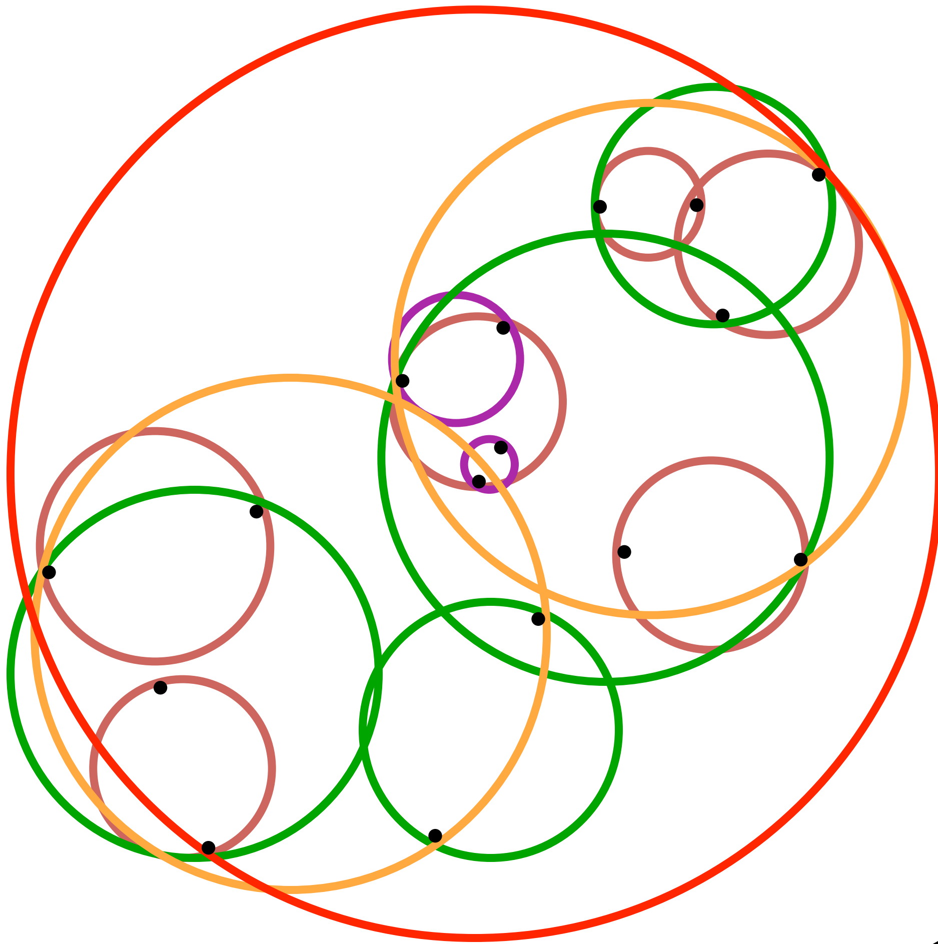
Classification and Regression Trees

D is a set of attribute-value vectors

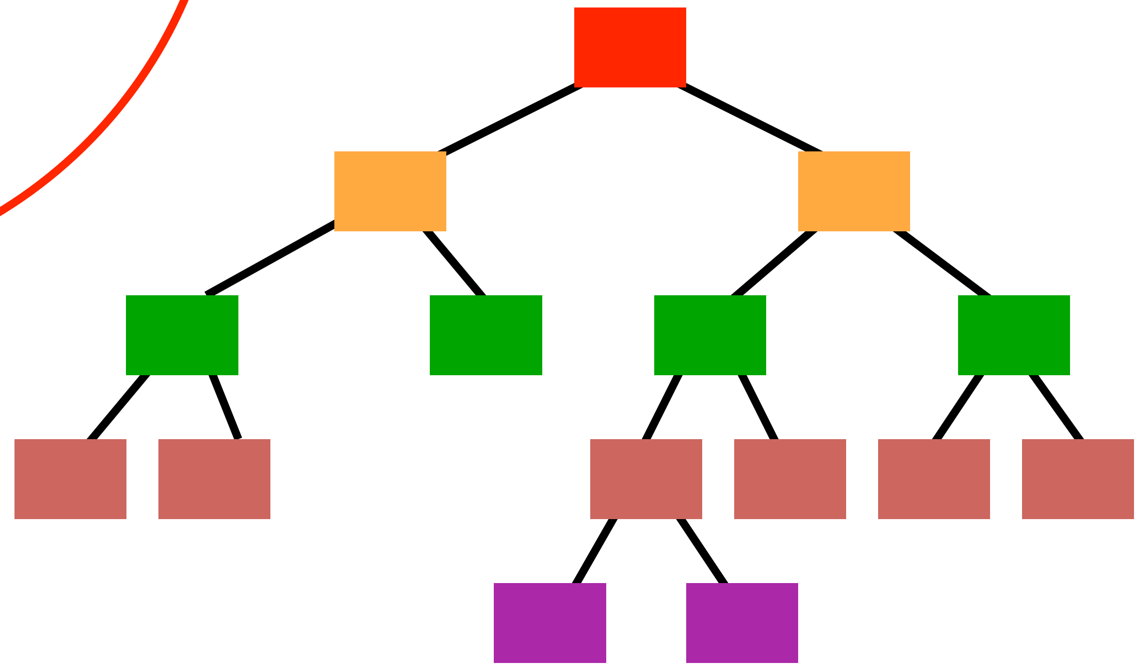
```
function CART(D) =
if done(D) then D
else let
  e = {condEntropy({d[i],d[n] : d in D}
       : i in [0:n]};
  P = split(D,min_index(e));
in TreeNode({CART(p) : p in P},i);
```

Other Tree Based Codes:

- proximity problems : nearest neighbors, metric trees
- Kernel density estimation
- Gaussian process regression
- n-point correlation

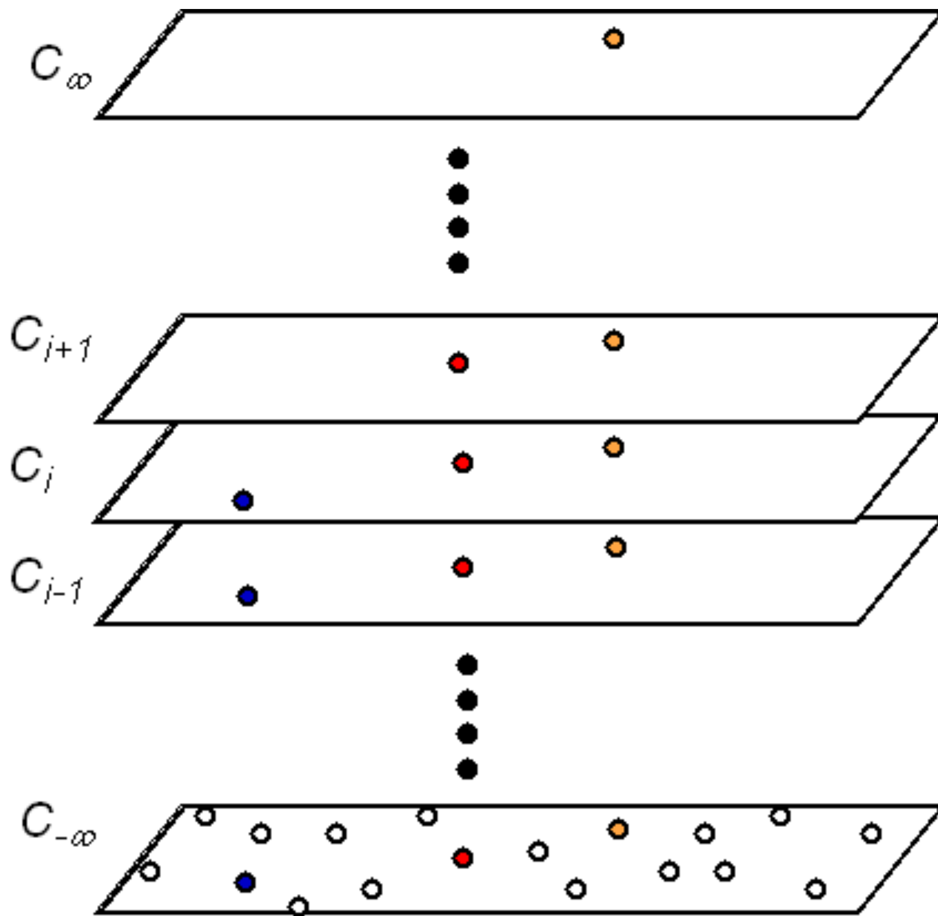


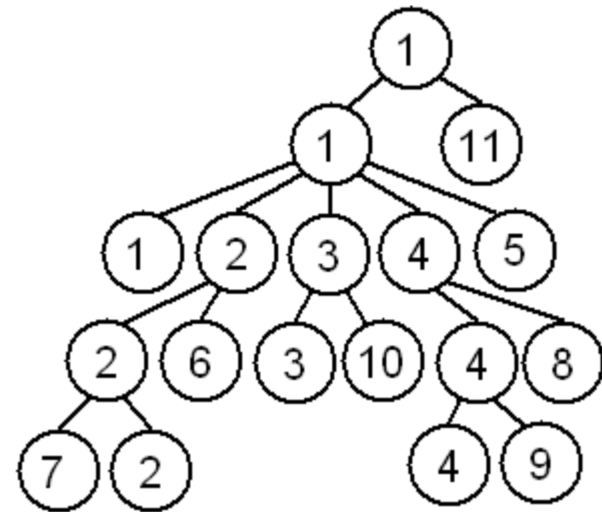
Ball Tree



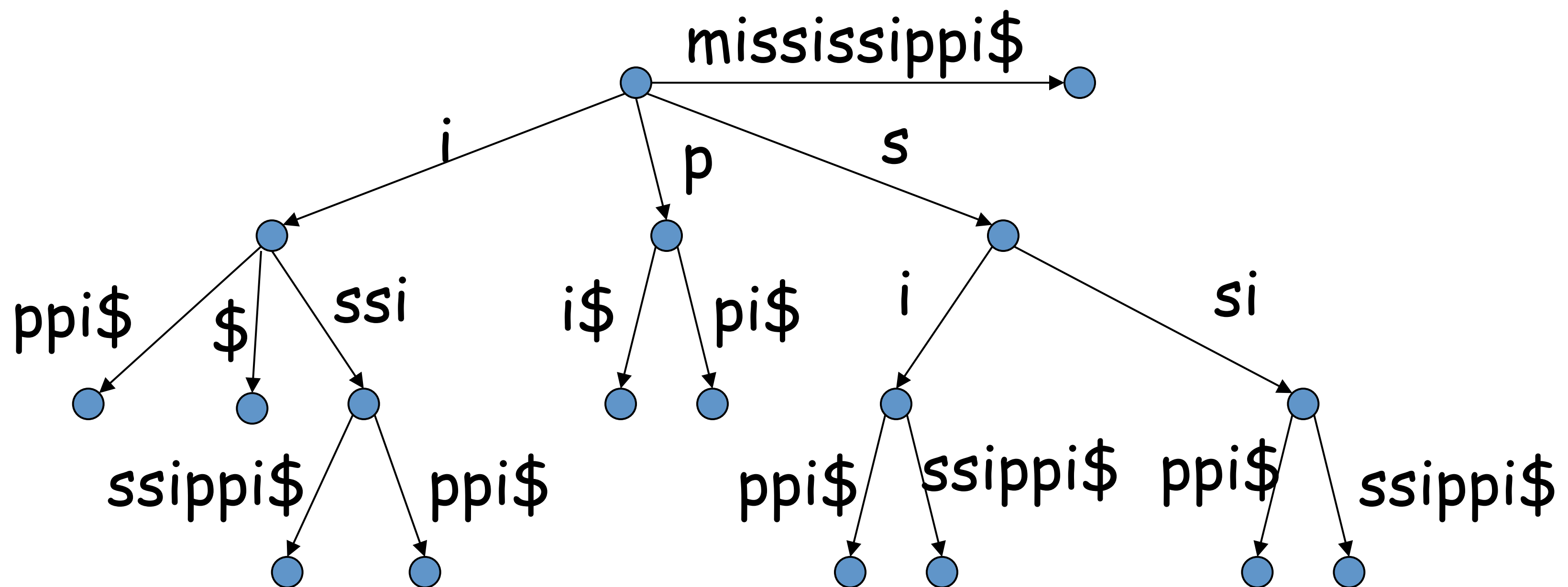
Slide from Alex Gray

Cover Trees





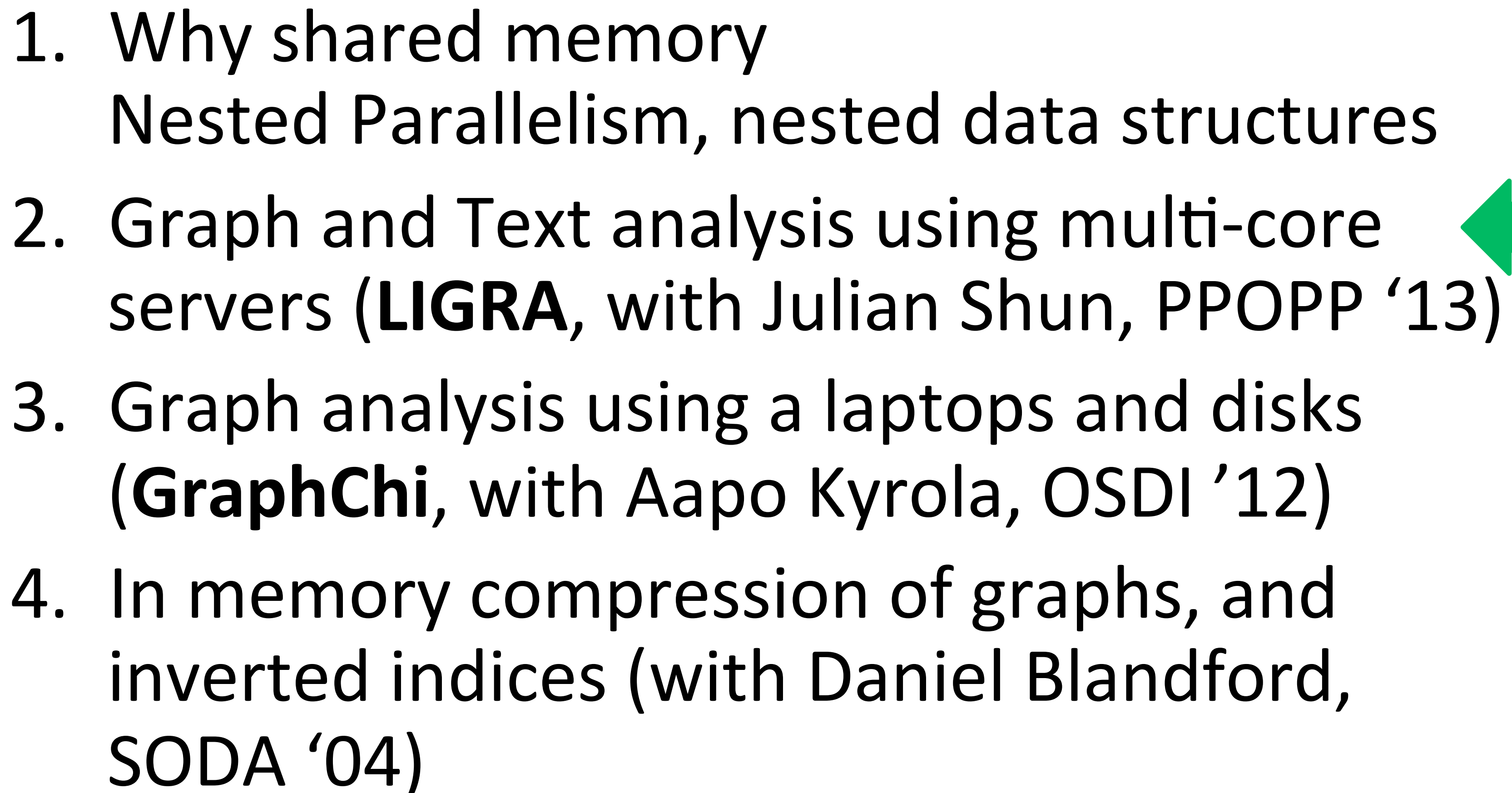
Suffix Trees



24x speedup on 40 cores over best sequential

- consists of many components of different types

In This Talk



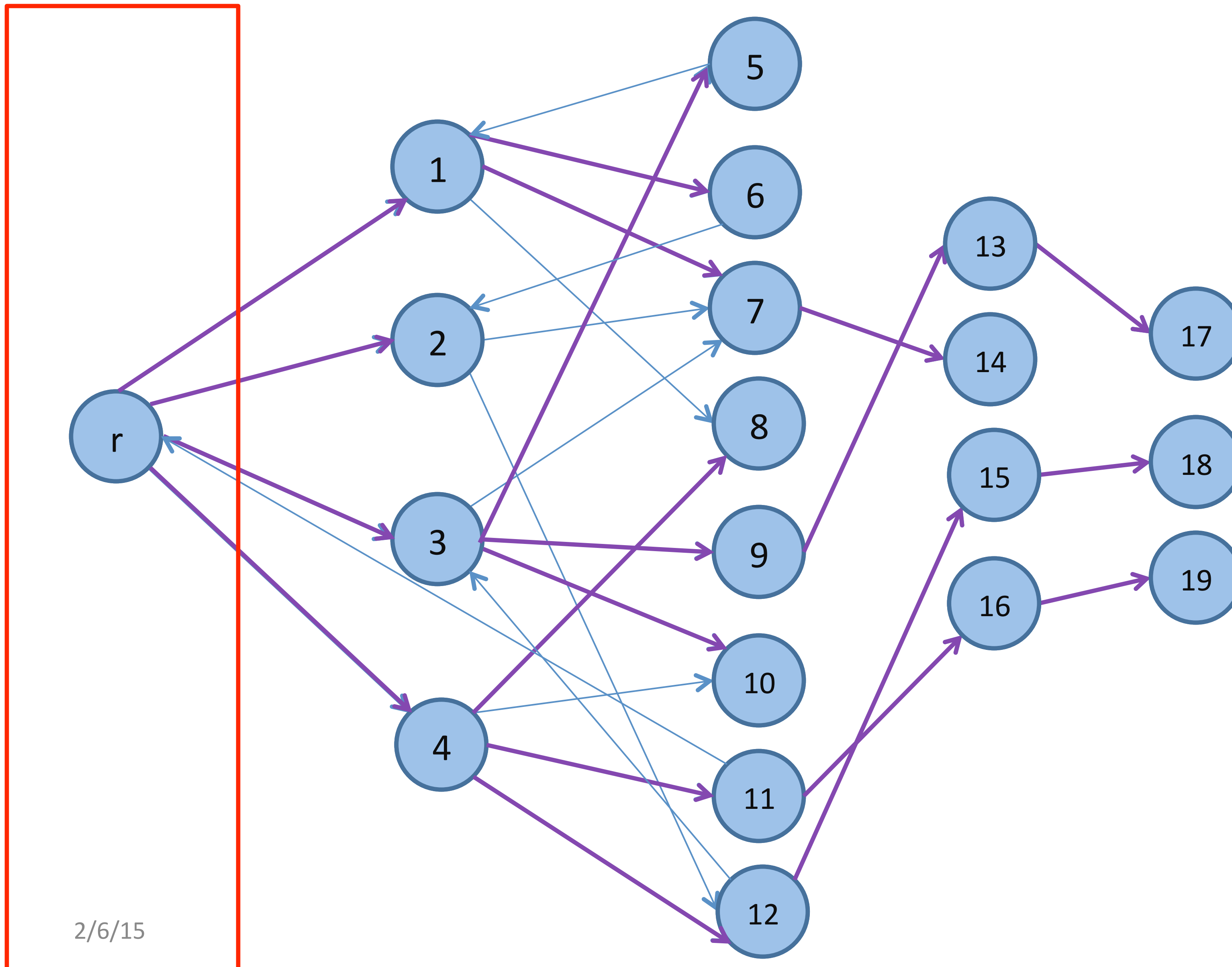
Ligra

- 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 most other graph systems by orders of magnitude, up to 39x speedup on 40 cores

Ligra Interface

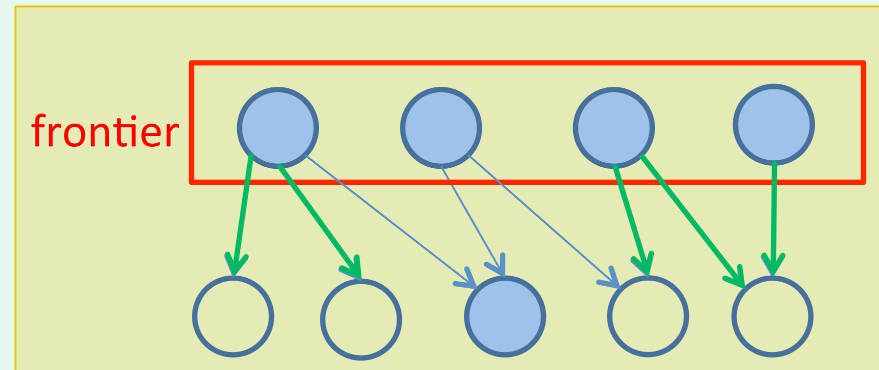
- Vertex subset: represents a subset of vertices
  - Important for algorithms that process only a subset of vertices each iteration
  - Can keep around multiple subsets for same graph
  - Can use one subset for multiple graphs
- Vertex map: maps user boolean function over vertex subset
- Edge map: maps user boolean function over out-edges of vertex subset

Parallel Breadth First Search (BFS)

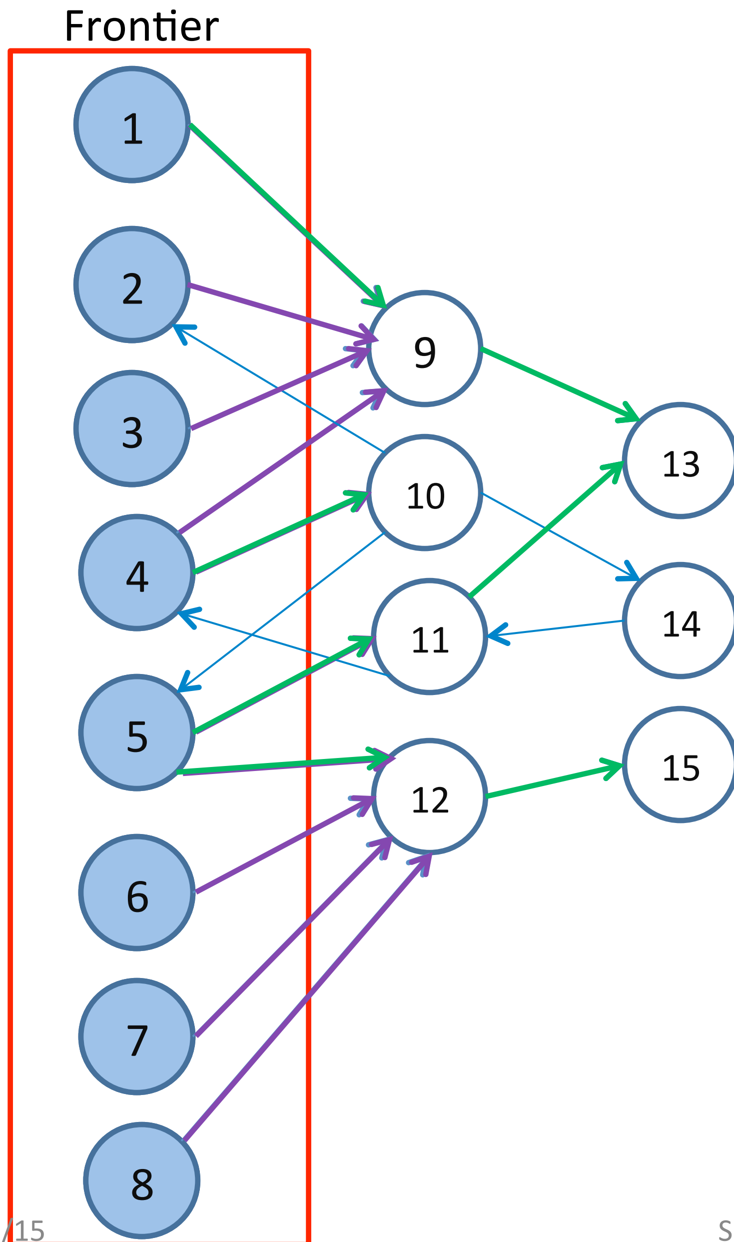


Breadth-first search in Ligra

```
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};
    while (size(frontier) > 0):
        frontier = EDGEMAP(G, frontier, UPDATE, COND);
```



Two methods for BFS



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

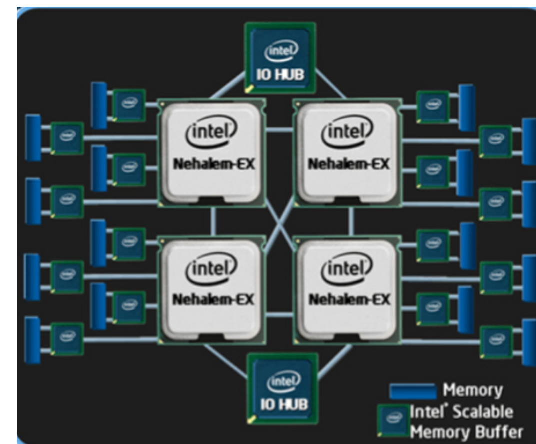
- 1st (Sparse) method better for small frontiers
- 2nd (Dense) method better when frontier is large and many vertices have been visited
- Switch between the two approaches based on frontier size

Comparison to other systems

Ligra	**Pregel/GPS/Giraph**	**GraphLab/ PowerGraph/ Grace**	**Pegasus (MapReduce)/ Knowledge Discovery Toolbox (KDT)**
Shared memory	Distributed memory	Shared/ Distributed	Distributed memory
Synchronous	Synchronous	Asynchronous	Synchronous
Vertex subsets Vertex map Edge map	Vertex operations	Vertex operations	Matrix-vector operations

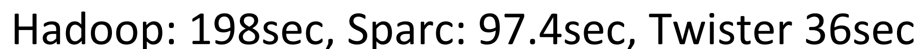
Experiments

- 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



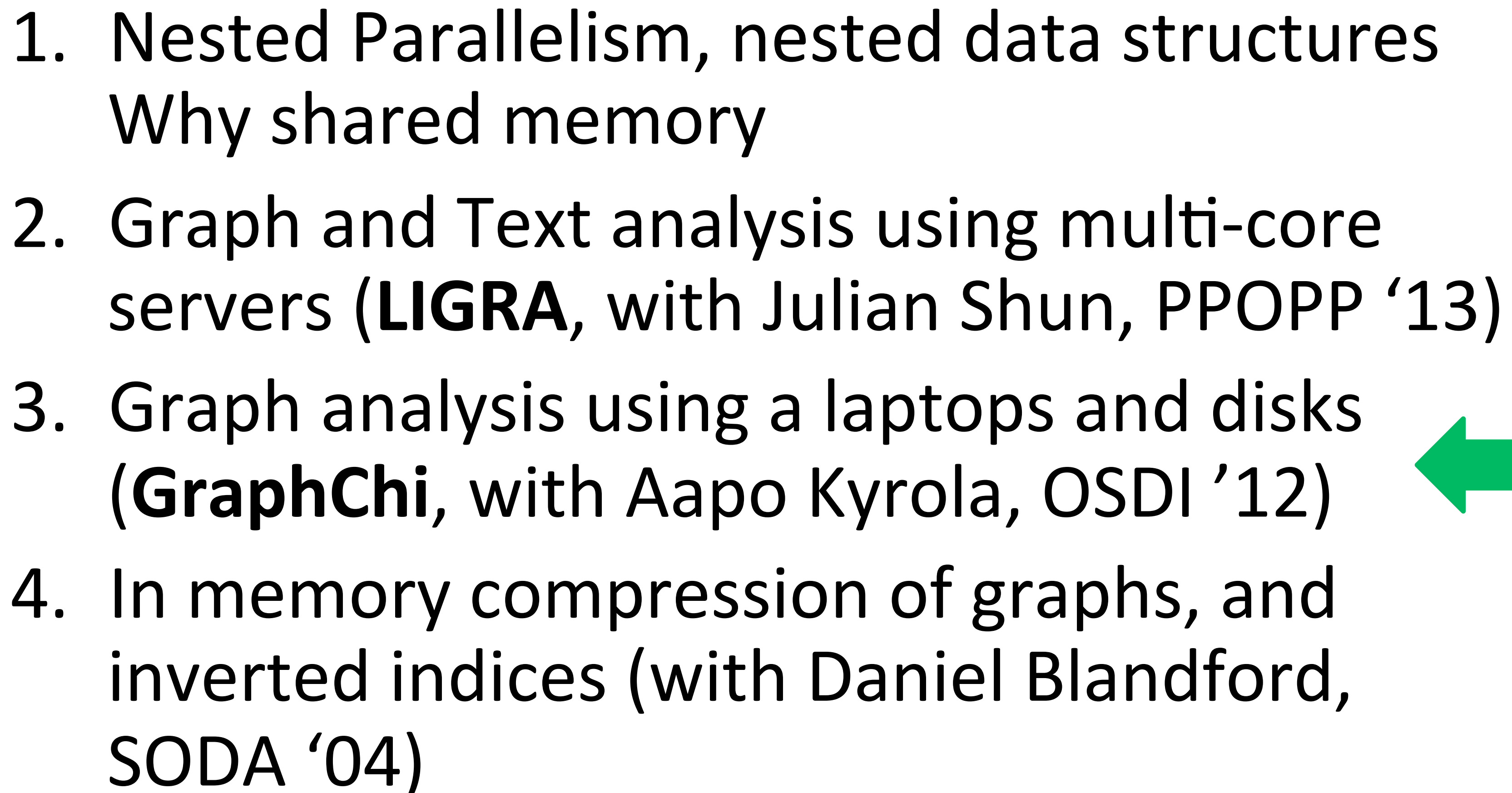
Ligra - Conclusions

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

Disk-Based Graph Algorithms

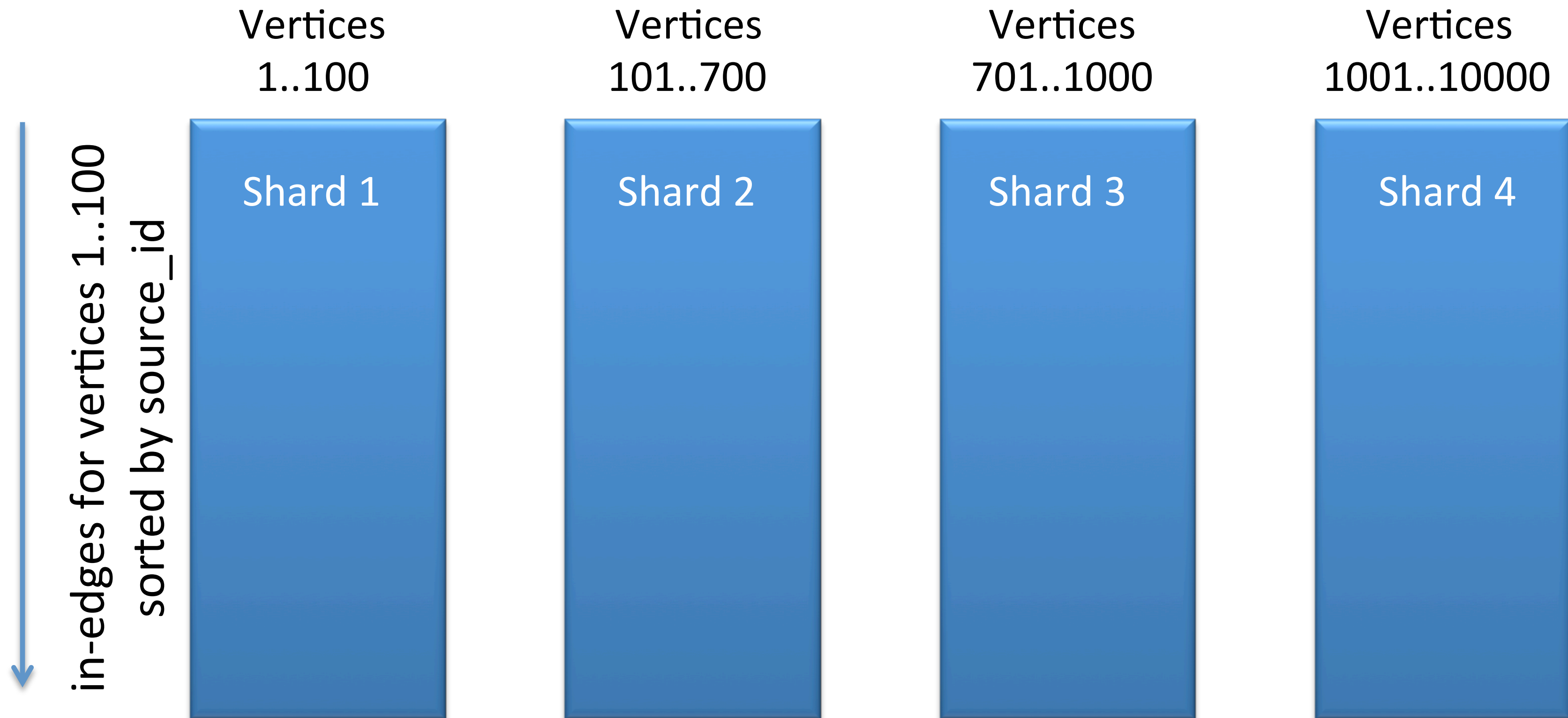
<u>Main Challenge</u>: Random Access

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

<u>GraphChi Solution</u>: Parallel Sliding Windows, based on Graphlab-like interface (vertex centric)

PSW: Layout

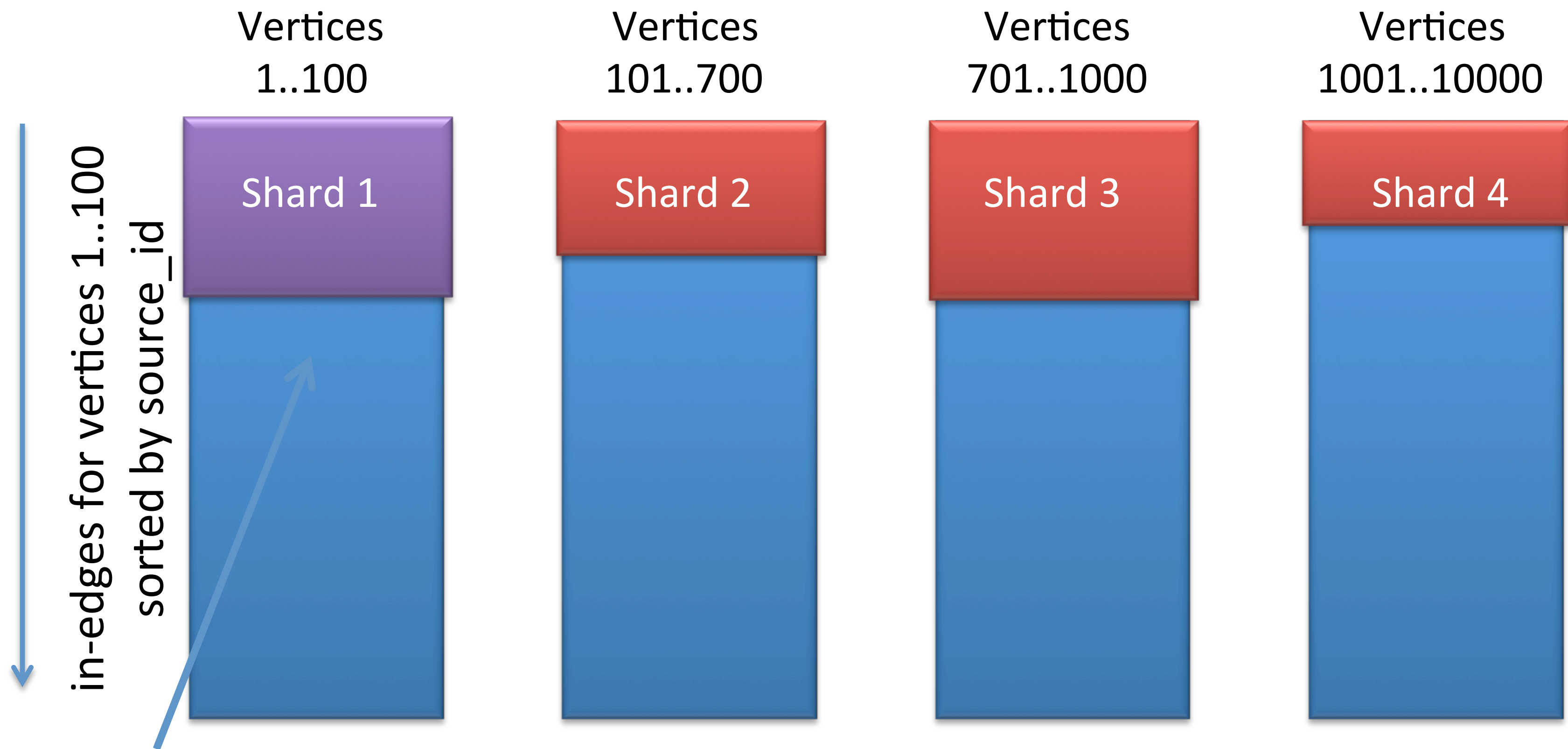
Shard: in-edges for **interval** of vertices; sorted by source-id



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

PSW: Loading Sub-graph

Load subgraph for vertices 1..100

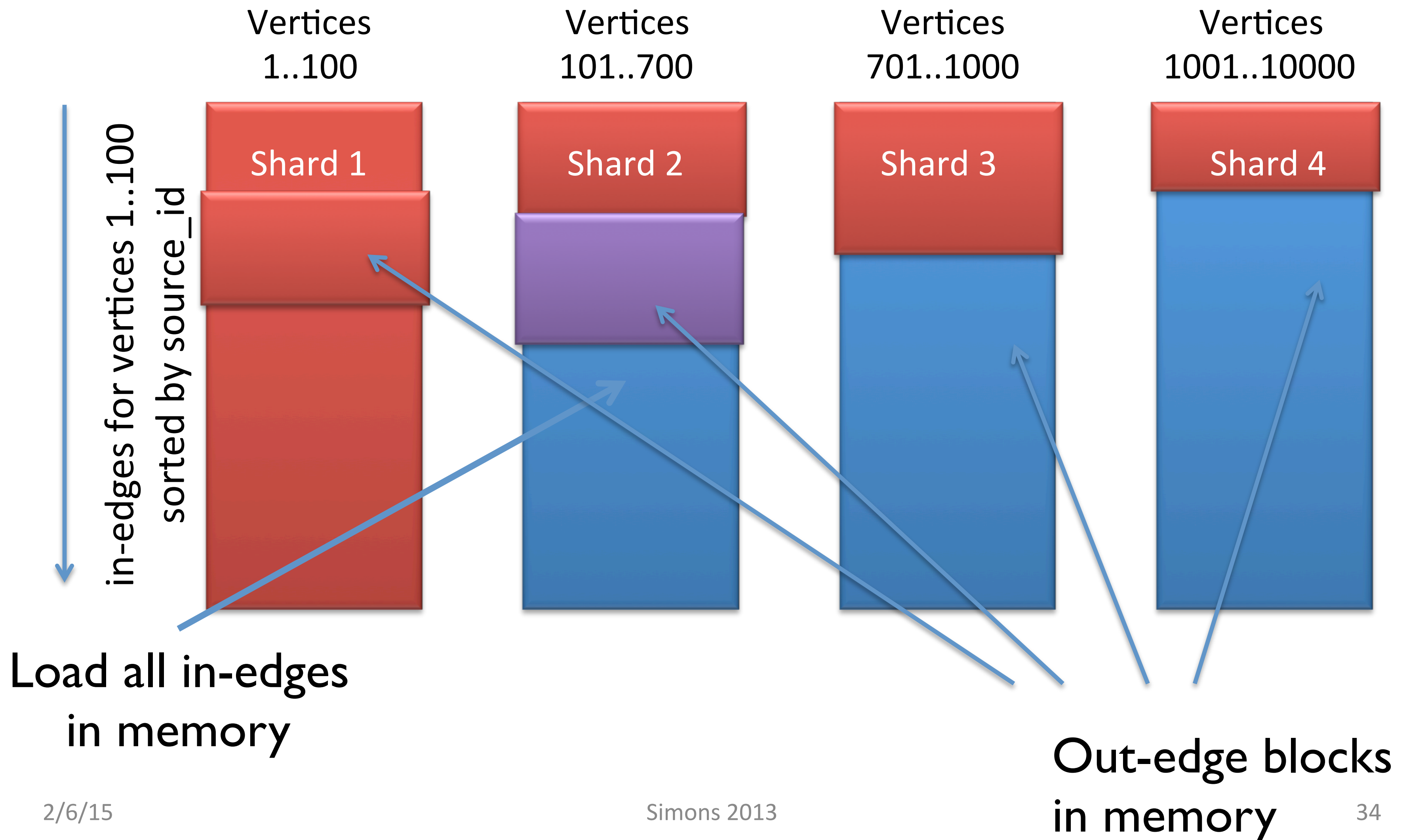


Load all in-edges in memory

What about out-edges?
Arranged in sequence in other shards

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

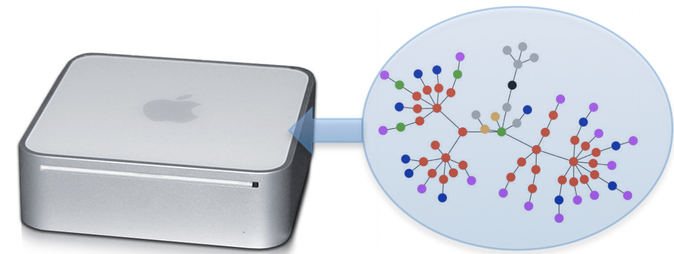
Probabilistic Graphical Models

- 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:

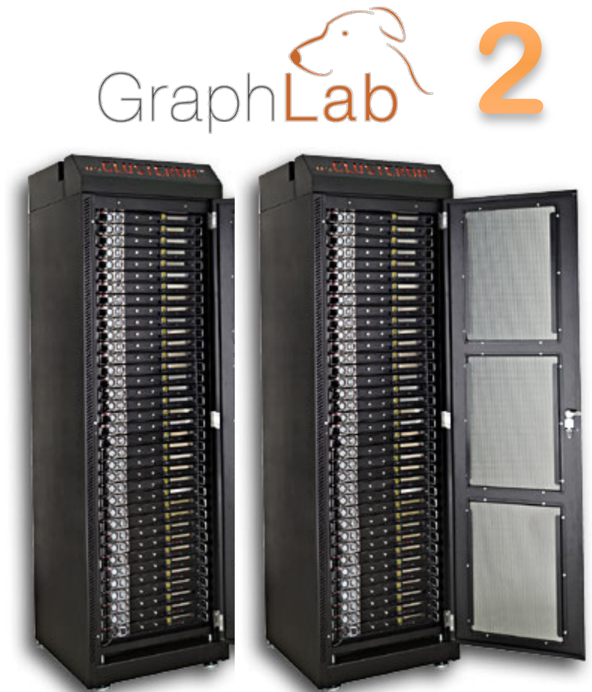


Graph	Vertices	Edges	P (shards)	Preprocessing
live-journal	4.8M	**69M**	3	0.5 min
netflix	0.5M	**99M**	20	1 min
twitter-2010	42M	**1.5B**	20	2 min
uk-2007-05	106M	**3.7B**	40	31 min
uk-union	133M	**5.4B**	50	33 min
yahoo-web	1.4B	**6.6B**	50	37 min

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.

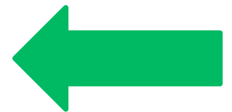
Hybrid Approach

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

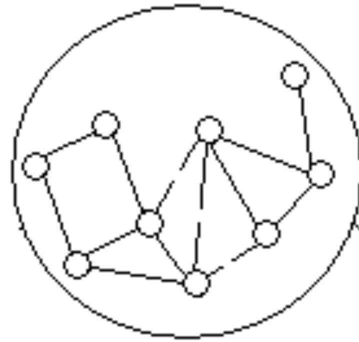
1. Nested Parallelism, nested data structures
   Why shared memory
2. Graph and Text analysis using multi-core servers (**LIGRA**, with Julian Shun, PPOPP '13)
3. Graph analysis using a laptops and disks (**GraphChi**, with Aapo Kyrola, OSDI '12)
4. In memory compression of graphs, and inverted indices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



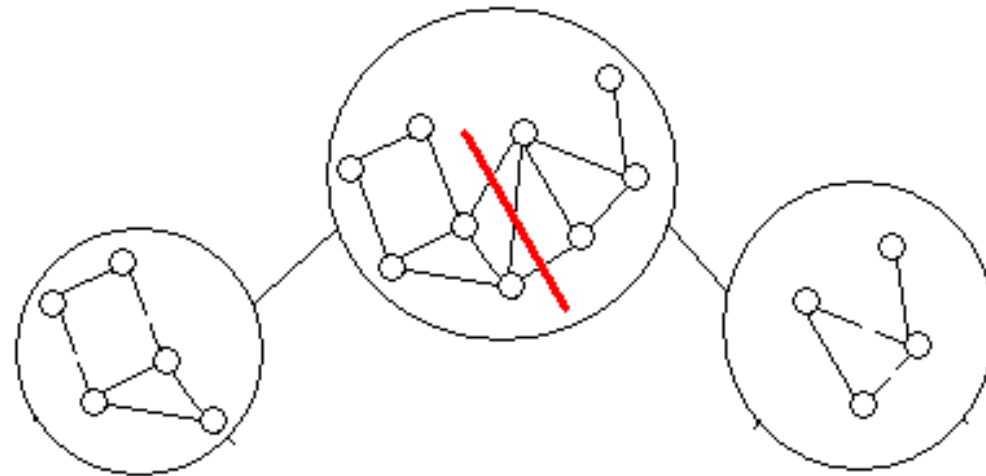
Compressing Graphs

- **<u>Goal</u>:** 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)
- **<u>Main idea</u>**:
  - Renumber vertices using separators
  - Difference encode the edges

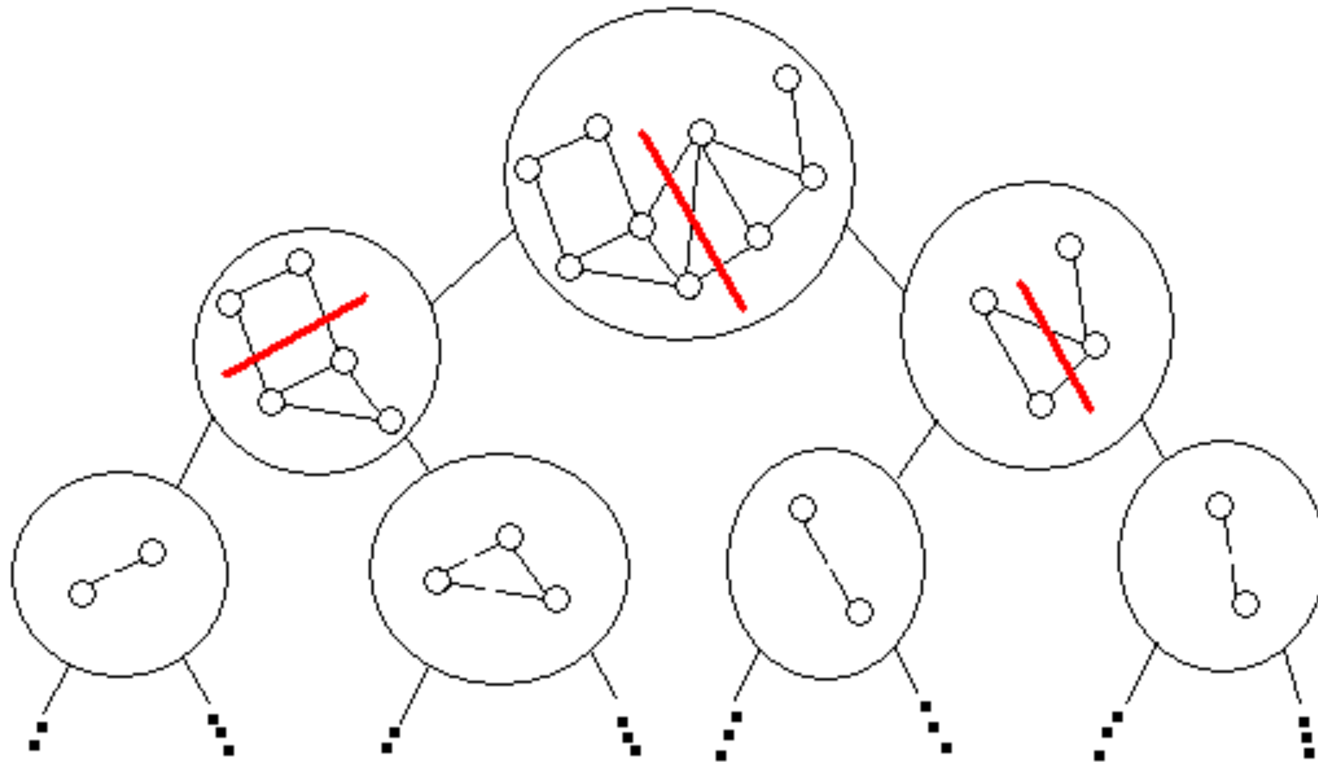
Renumbering with Edge Separators



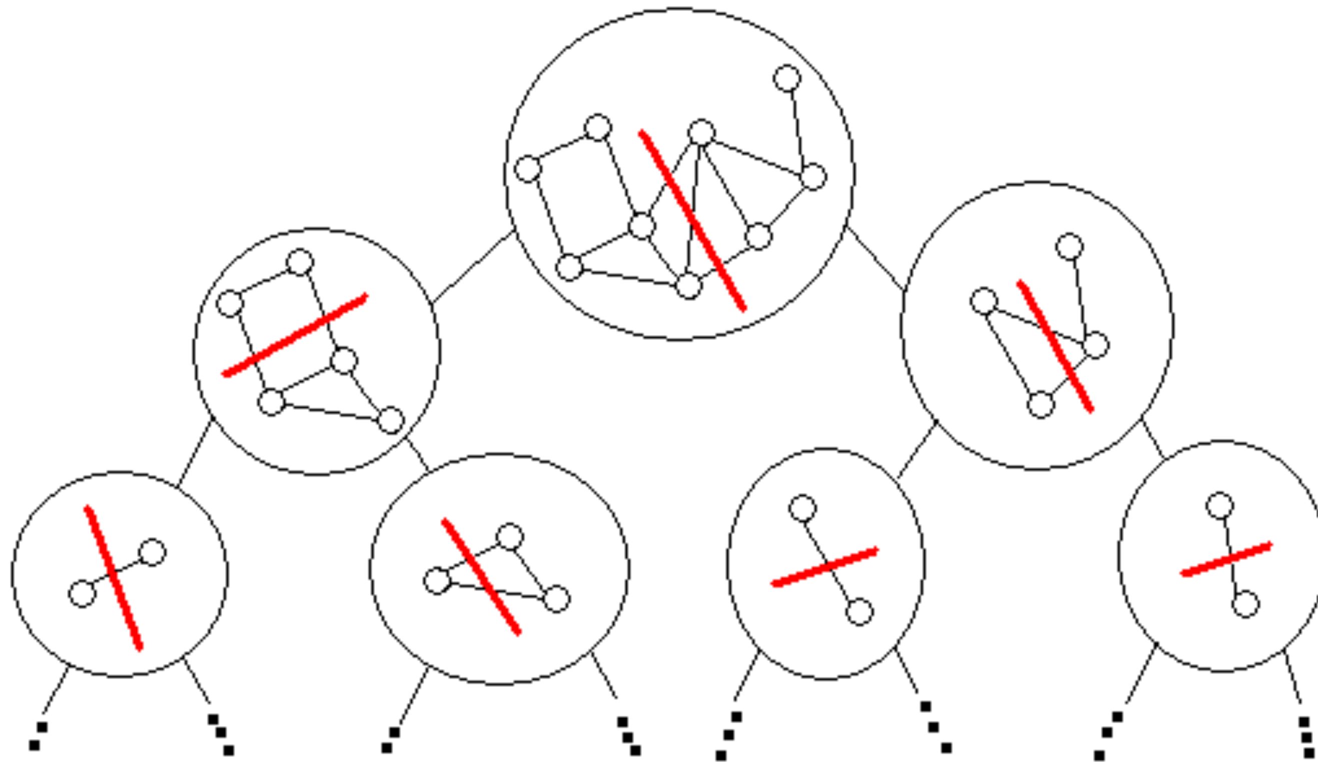
Renumbering with Edge Separators



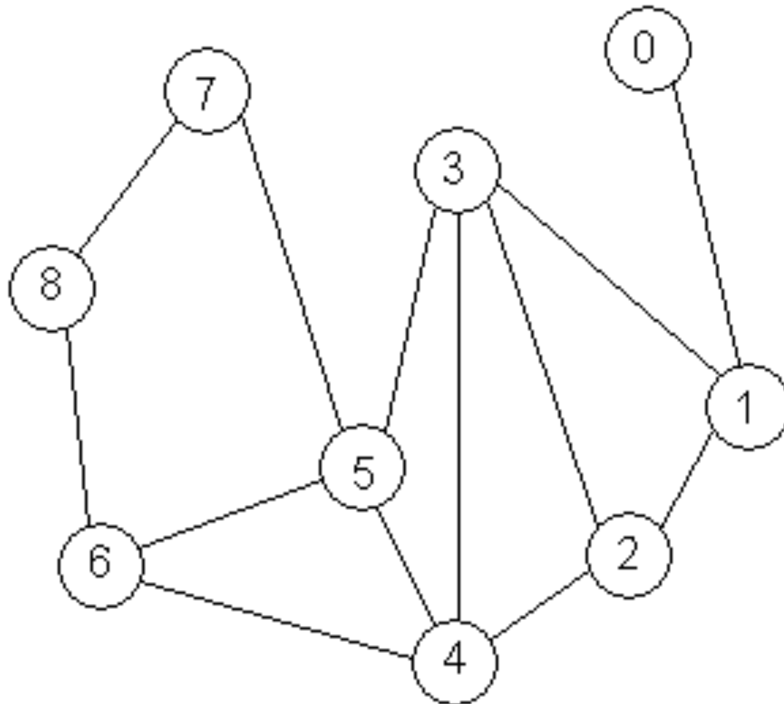
Renumbering with Edge Separators



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

Theorem: If $O(n^c)$, $c < 1$ separators, guarantees $O(n)$ total bits.

Bits per edge for various graphs

	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

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!

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