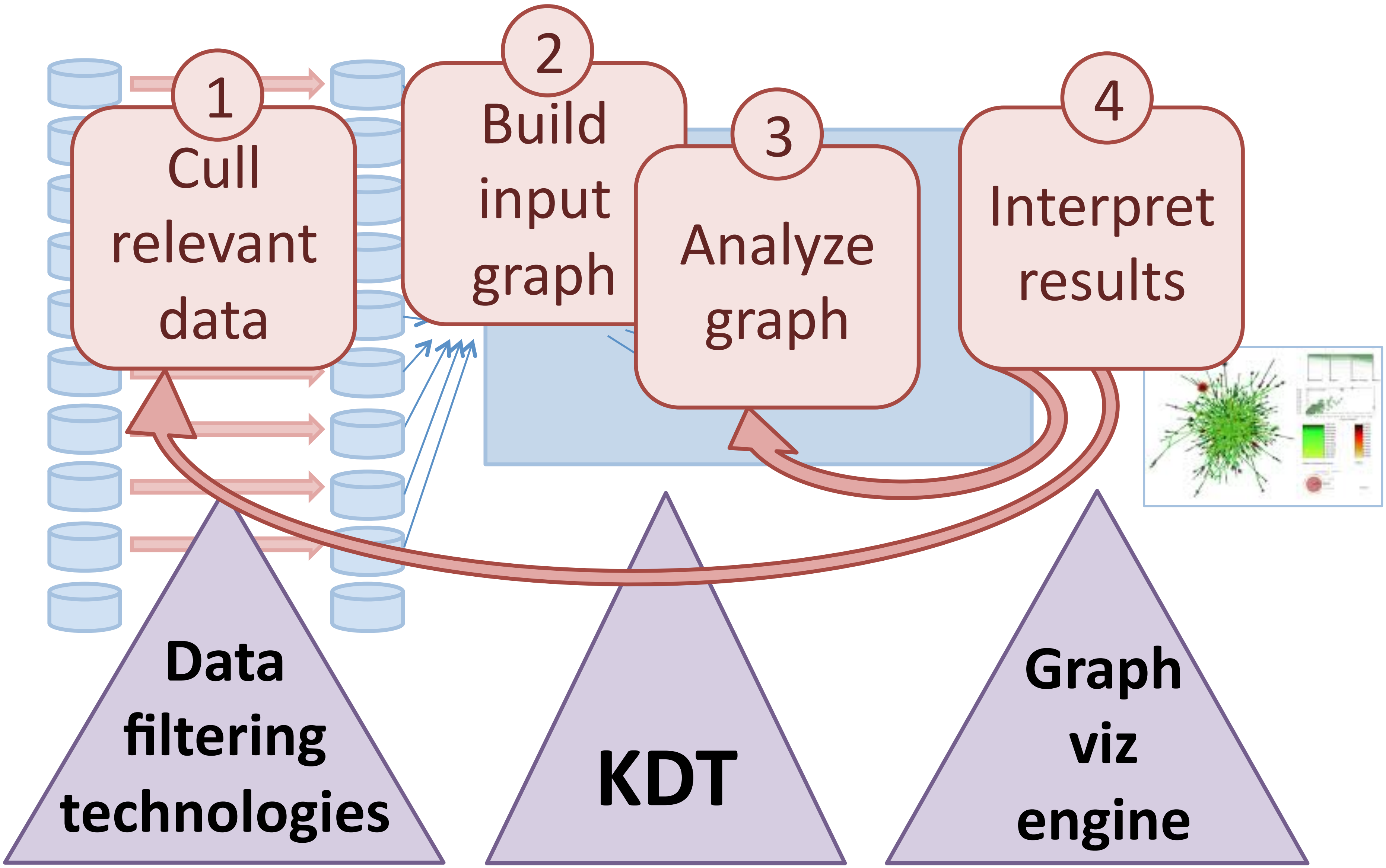
Adam Lugowski

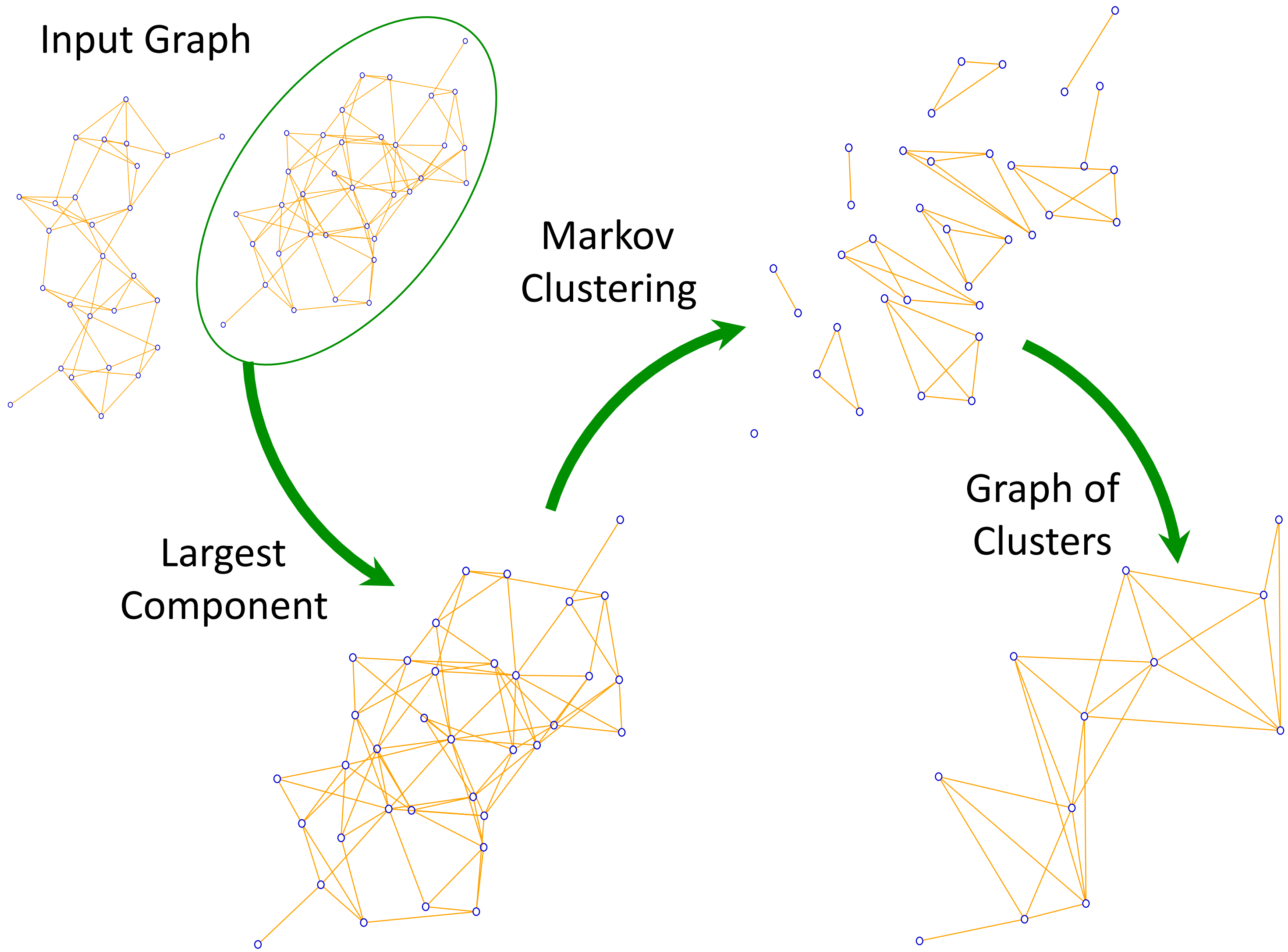
Knowledge Discovery Toolbox

kdt.sourceforge.net

Our users: Domain Experts

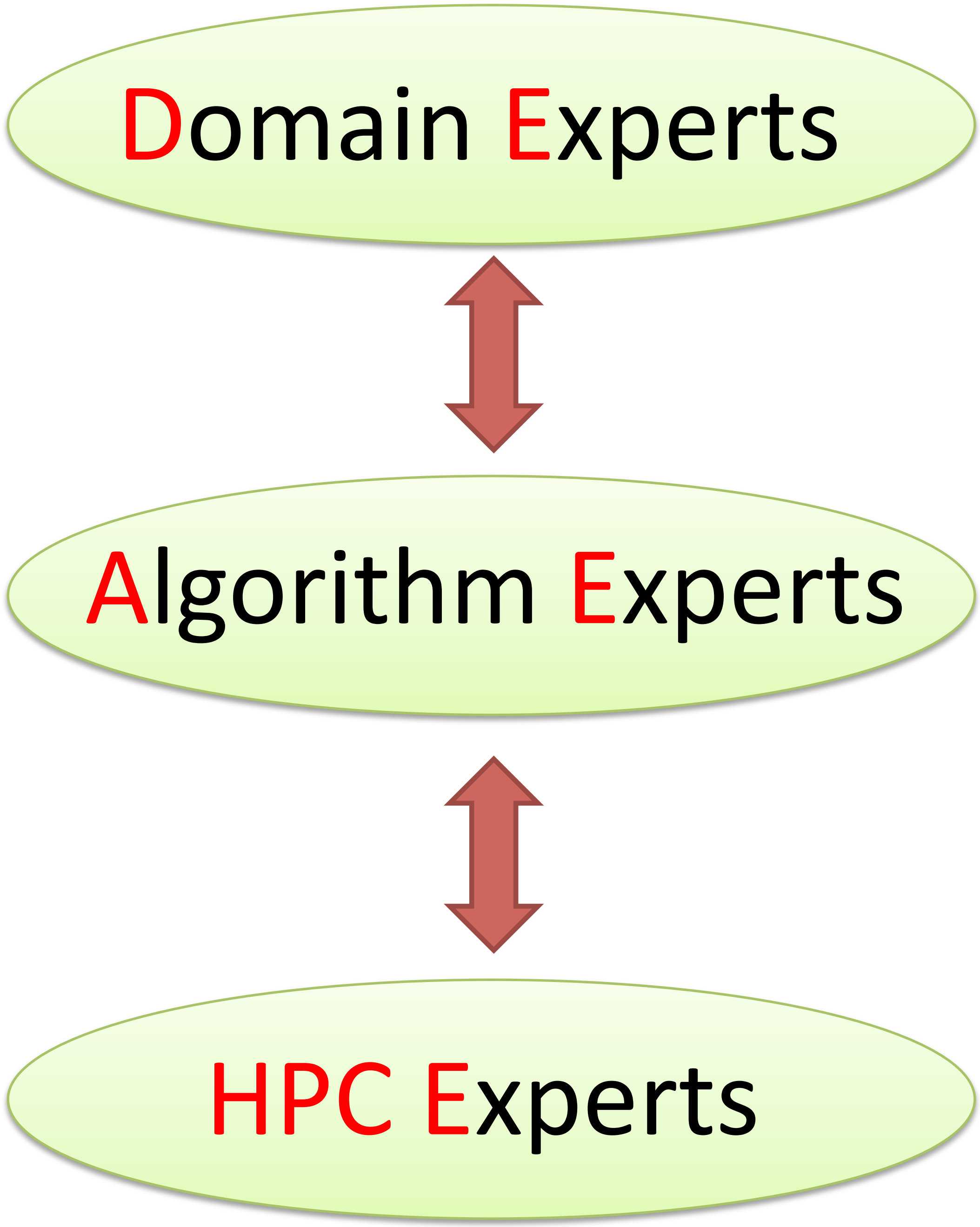


Example workflow



How to target Domain Experts?

- Conceptually simple
- Customizable
- High Performance


Domain Experts
Algorithm Experts
HPC Experts

Complex methods

centrality('approxBC')
pageRank

cluster('Markov')
contract

. . .

Building blocks

DiGraph

- bfsTree,neighbor
- degree,subgraph
- load,UFget
- +, -, sum, scale

Mat

- SpMV
- SpGEMM
- load, eye
- reduce, scale
- +, []

Vec

- max, norm,sort
- abs, any, ceil
- range, ones
- +,-,*,/,>,==,&,[]

Underlying infrastructure (Combinatorial BLAS)

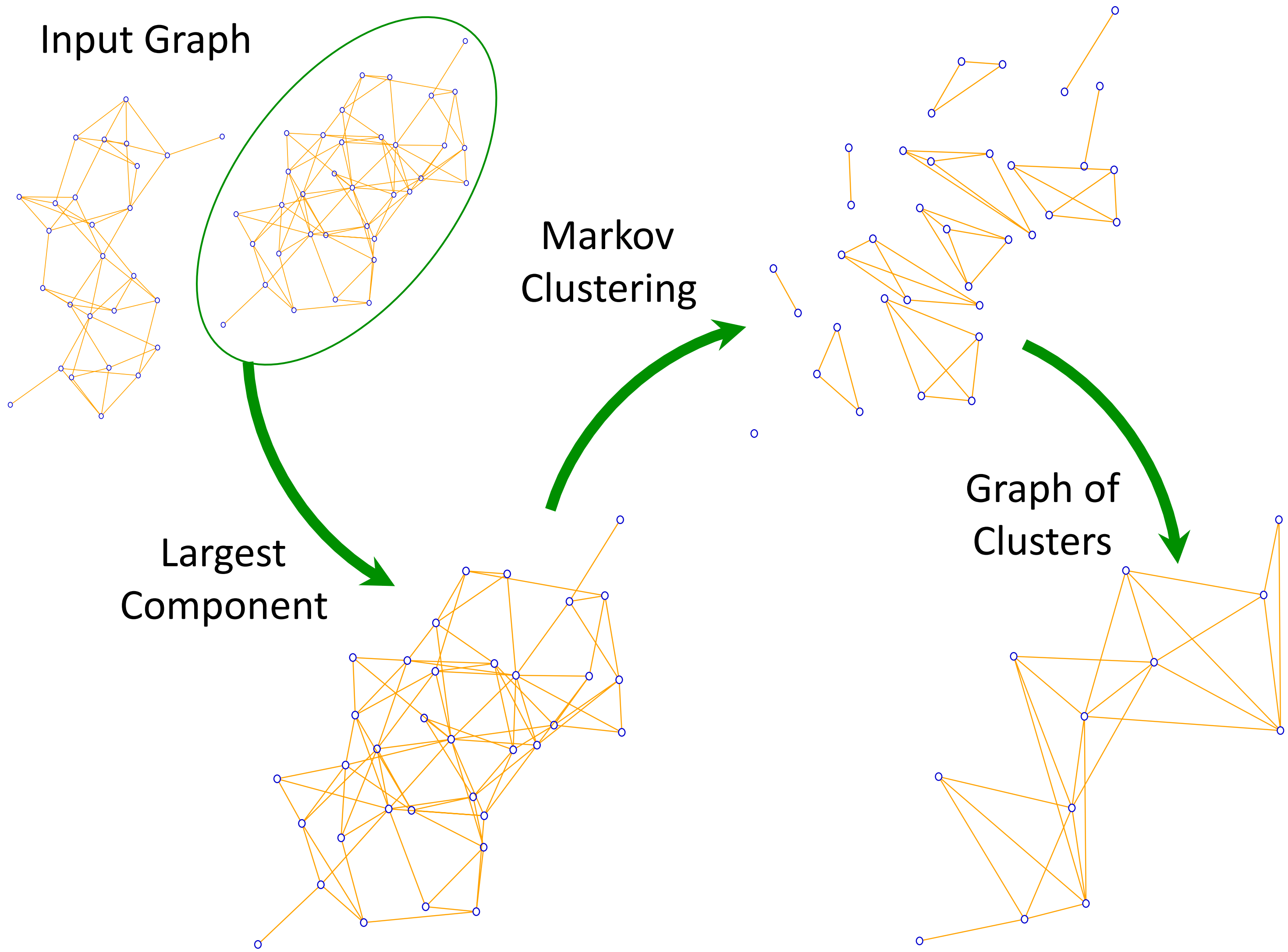
- SpMV, SpMV_SemiRing
- SpGEMM, SpGEMM_SemiRing

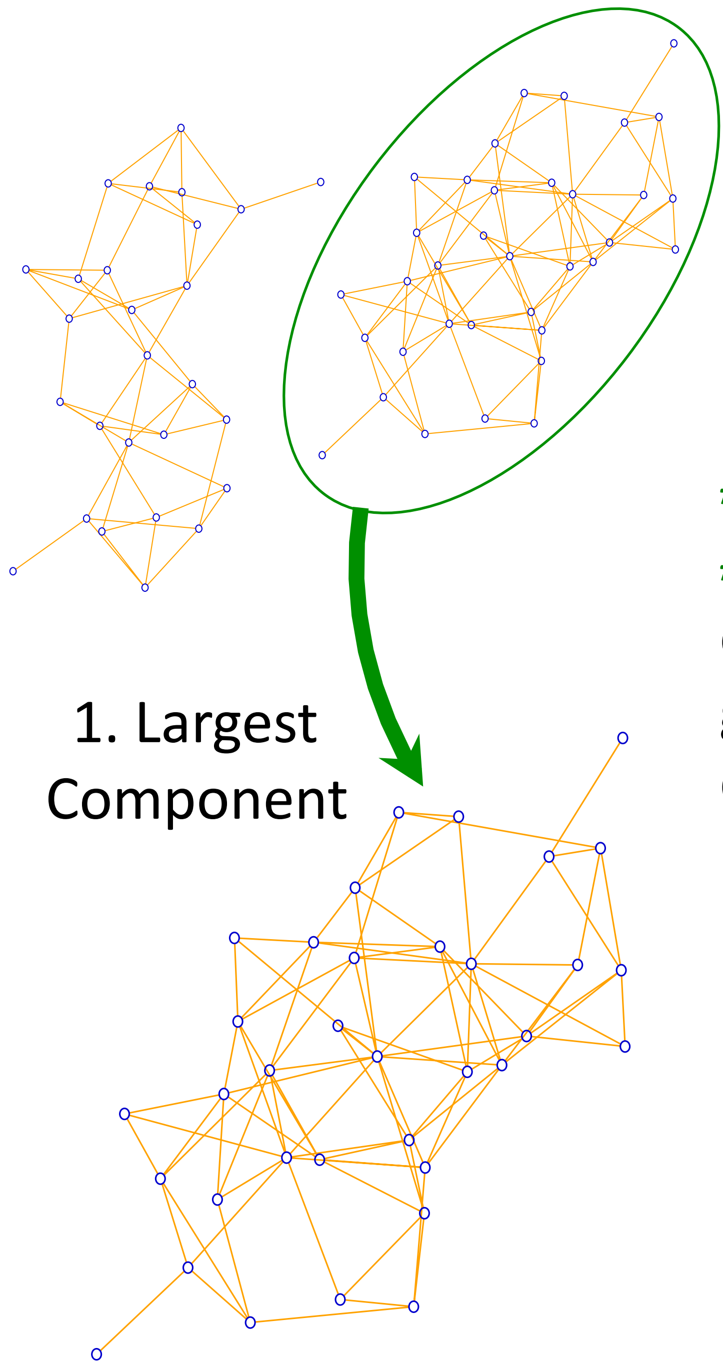
Sparse-matrix classes/methods
(*e.g.*, Apply, EWiseApply, Reduce)

Why (sparse) adjacency matrices?

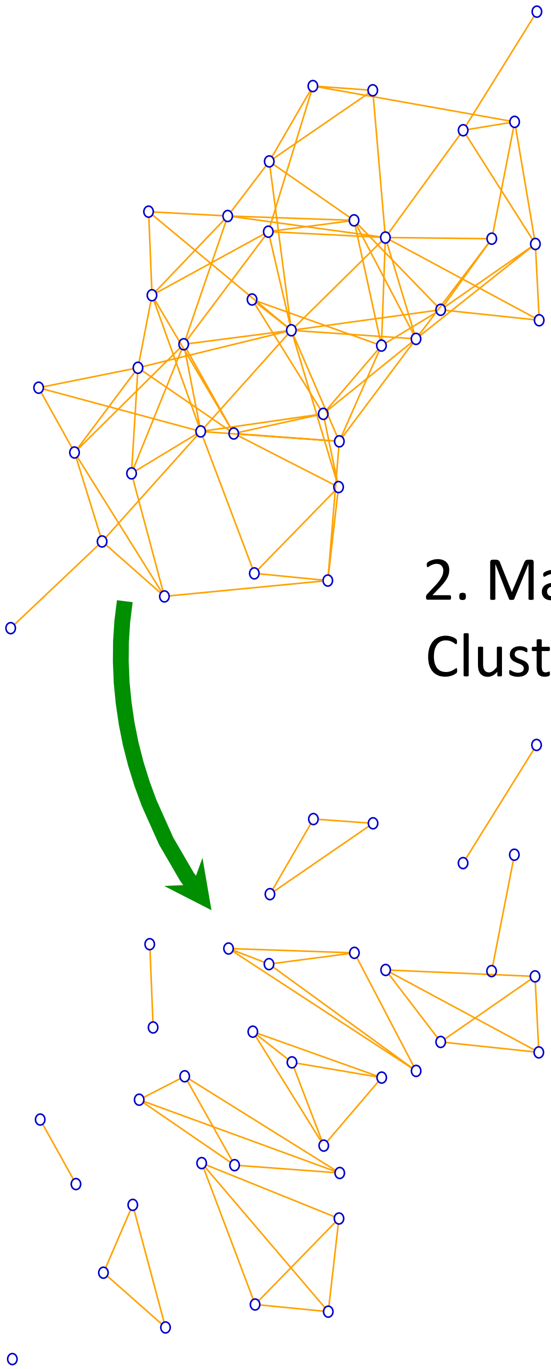
Traditional graph computations	Graphs in the language of linear algebra
Data driven, unpredictable communication	Fixed communication patterns
Irregular and unstructured, poor locality of reference	Operations on matrix blocks exploit memory hierarchy
Fine grained data accesses, dominated by latency	Coarse grained parallelism, bandwidth limited

Example workflow



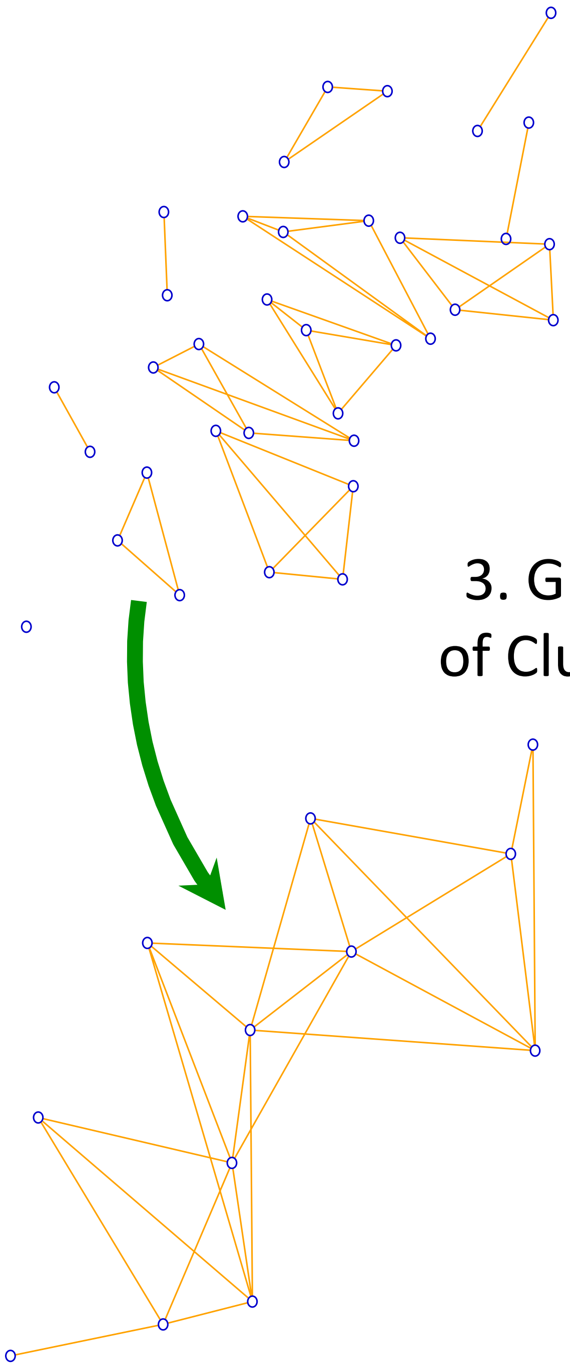



```
# the variable bigG contains the input graph
# find and select the giant component
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)
```



2. Markov
Clustering

 # cluster the graph
clus = G.cluster('Markov')



3. Graph
of Clusters

 # contract the clusters

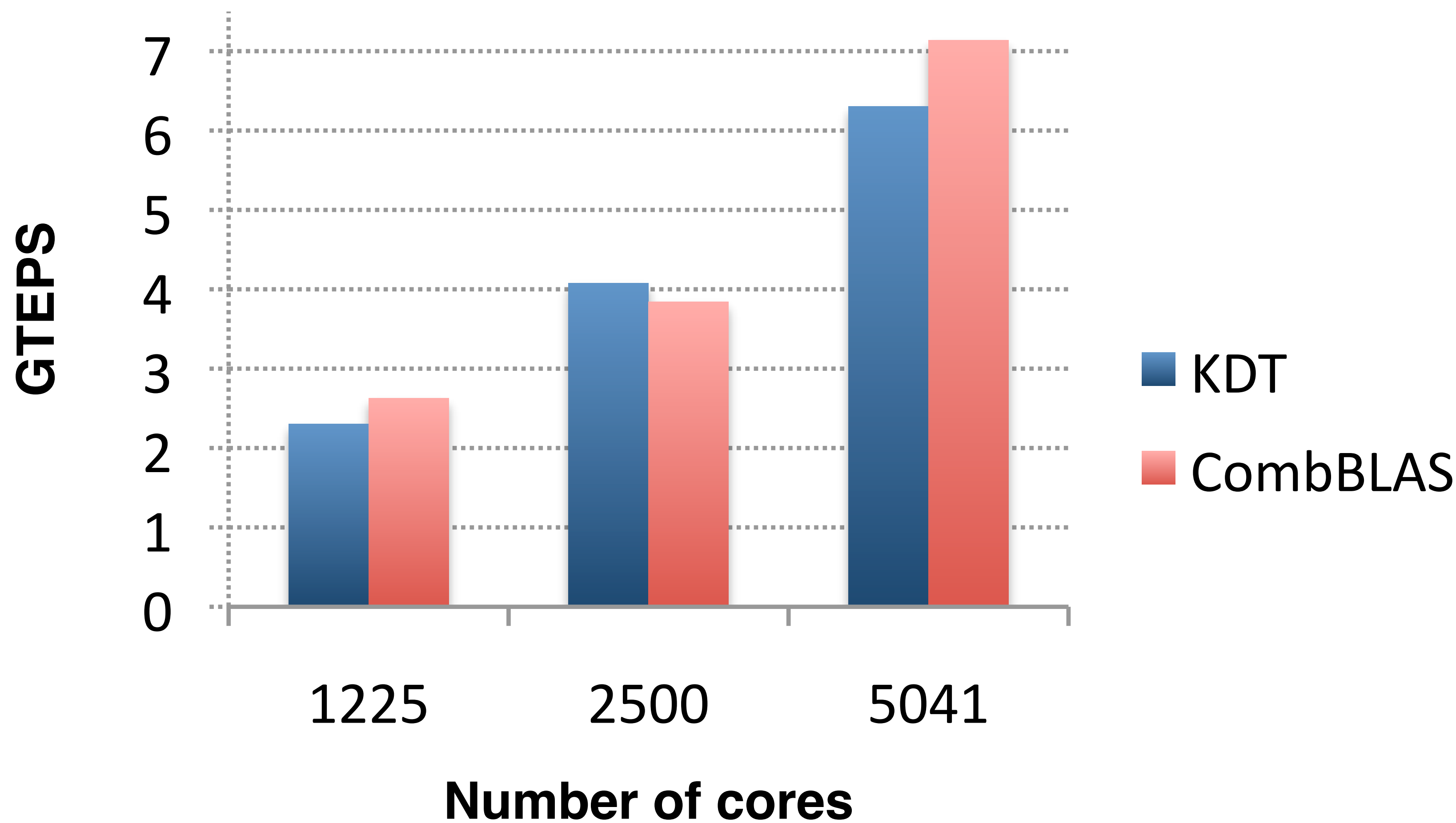
```
smallG = G.contract(clus)
```

Example workflow KDT code

```
# the variable bigG contains the input graph
# find and select the giant component
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

# cluster the graph
clus = G.cluster('Markov')

# contract the clusters
smallG = G.contract(clus)
```

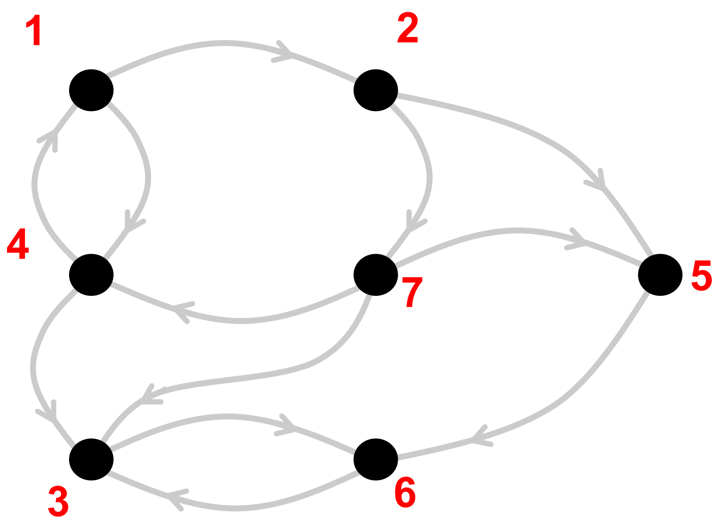
BFS on a Scale 29 RMAT graph

(500M vertices, 8B edges)



Machine: NERSC's Hopper

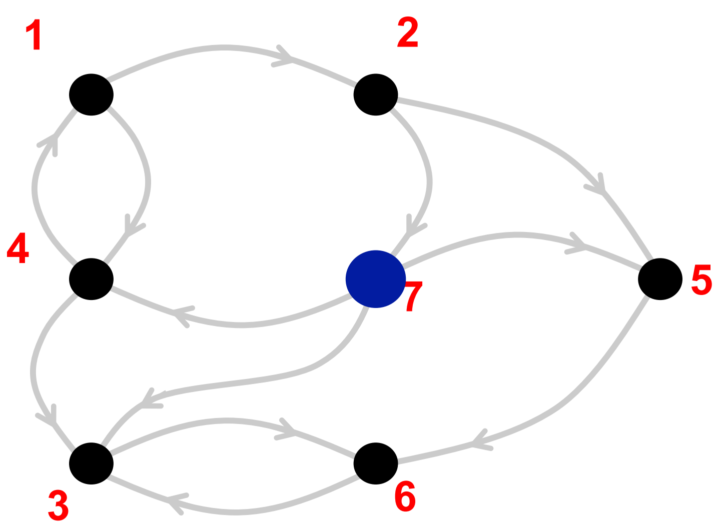
Breadth-First Search



G

			1			
1						
			1		1	1
1						1
	1					1
		1		1		
	1					

Breadth-First Search



G

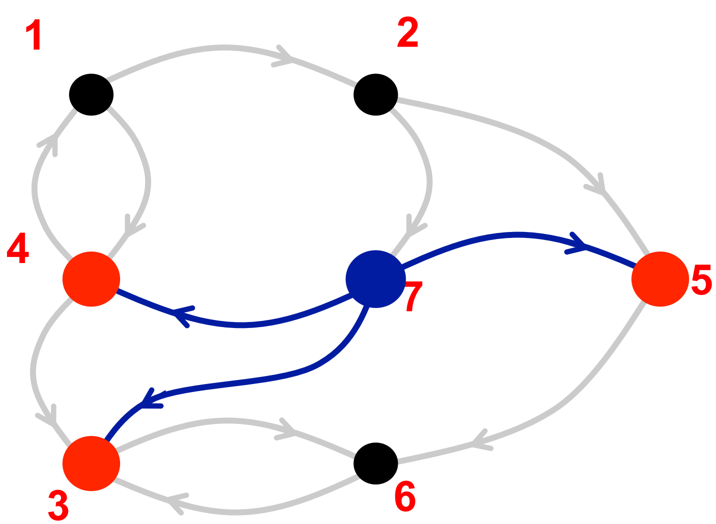
			1			
1						
			1		1	1
1						1
	1					1
		1		1		
	1					

f_{in}

7

distance 1 from vertex 7

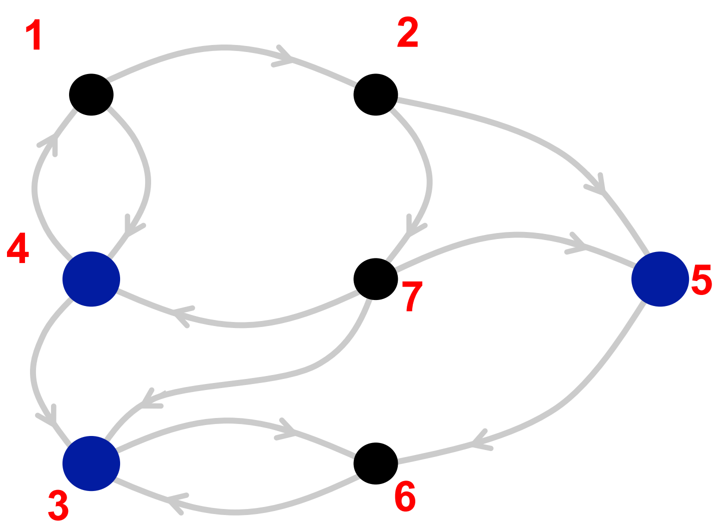
Breadth-First Search



$$
\overset{G}{\begin{bmatrix}
 & & & 1 & & & \\
1 & & & & & & \\
 & & & 1 & & 1 & 1 \\
1 & & & & & & 1 \\
 & 1 & & & & & 1 \\
 & & 1 & & 1 & & \\
 & 1 & & & & &
\end{bmatrix}}
\times
\overset{f_{in}}{\begin{bmatrix} \\ \\ \\ \\ \\ \\ 7 \end{bmatrix}}
=
\overset{f_{out}}{\begin{bmatrix} \\ \\ 7 \\ 7 \\ 7 \\ \\ \end{bmatrix}}
$$

distance 1 from vertex 7

Breadth-First Search



G

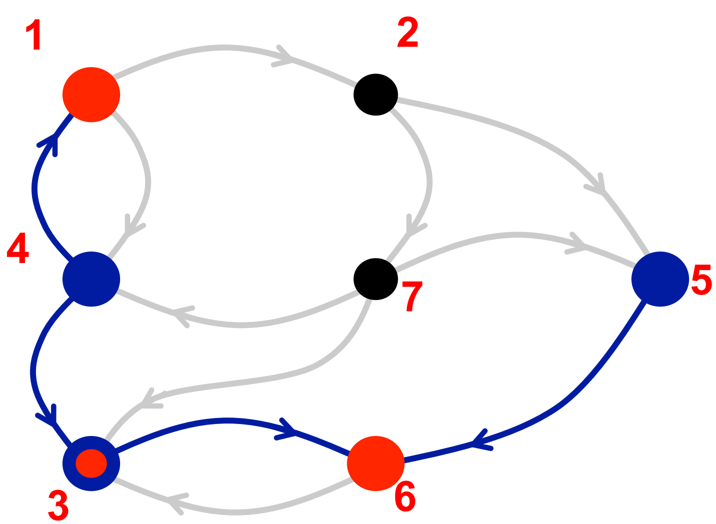
			1			
1						
			1		1	1
1						1
	1					1
		1		1		
	1					

f_{in}

3
4
5

distance 2 from vertex 7

Breadth-First Search



G

			1			
1						
			1		1	1
1						1
	1					1
		1		1		
	1					

×

f_{in}

3
4
5

=

f_{out}

4
4
5

distance 2 from vertex 7

KDT BFS routine

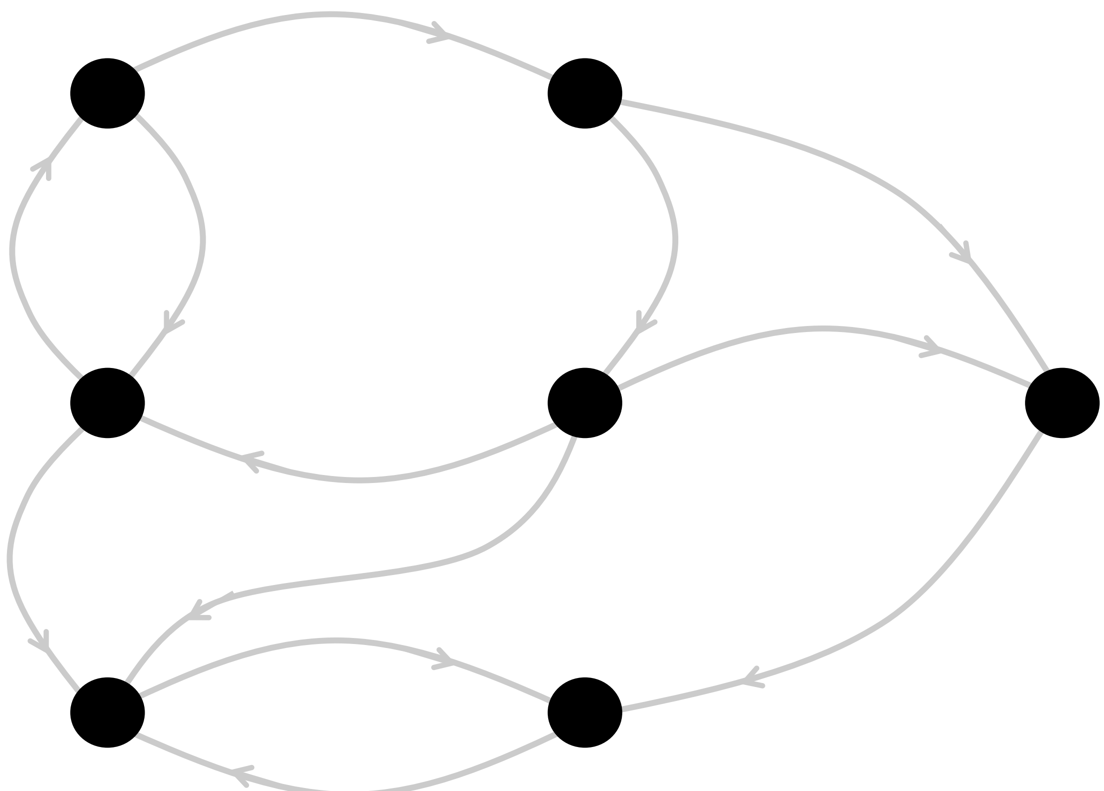
```
# initialization
parents = Vec(self.nvert(), -1, sparse=False)
frontier = Vec(self.nvert(), sparse=True)
parents[root] = root
frontier[root] = root # 1st frontier is just the root
# the semiring mult and add ops simply return the 2nd arg
semiring = sr((lambda x,y: y), (lambda x,y: y))

# loop over frontiers
while frontier.nnn() > 0:
    frontier.spRange() # frontier[i] = i
    self.e.SpMV(frontier, semiring=semiring, inPlace=True)

    # remove already discovered vertices from the frontier.
    frontier.eWiseApply(parents, op=(lambda f,p: f),
                doOp=(lambda f,p: p == -1), inPlace=True)
    # update the parents
    parents[frontier] = frontier
```

BFS comparison with PBGL

Core Count (Machine)	Code	Problem Size		
		Scale 19	Scale 22	Scale 24
4 (Neumann)	PBGL	3.8	2.5	2.1
	KDT	8.9	7.2	6.4
16 (Neumann)	PBGL	8.9	6.3	5.9
	KDT	33.8	27.8	25.1
128 (Carver)	PBGL		25.9	39.4
	KDT		237.5	262.0
256 (Carver)	PBGL		22.4	37.5
	KDT		327.6	473.4

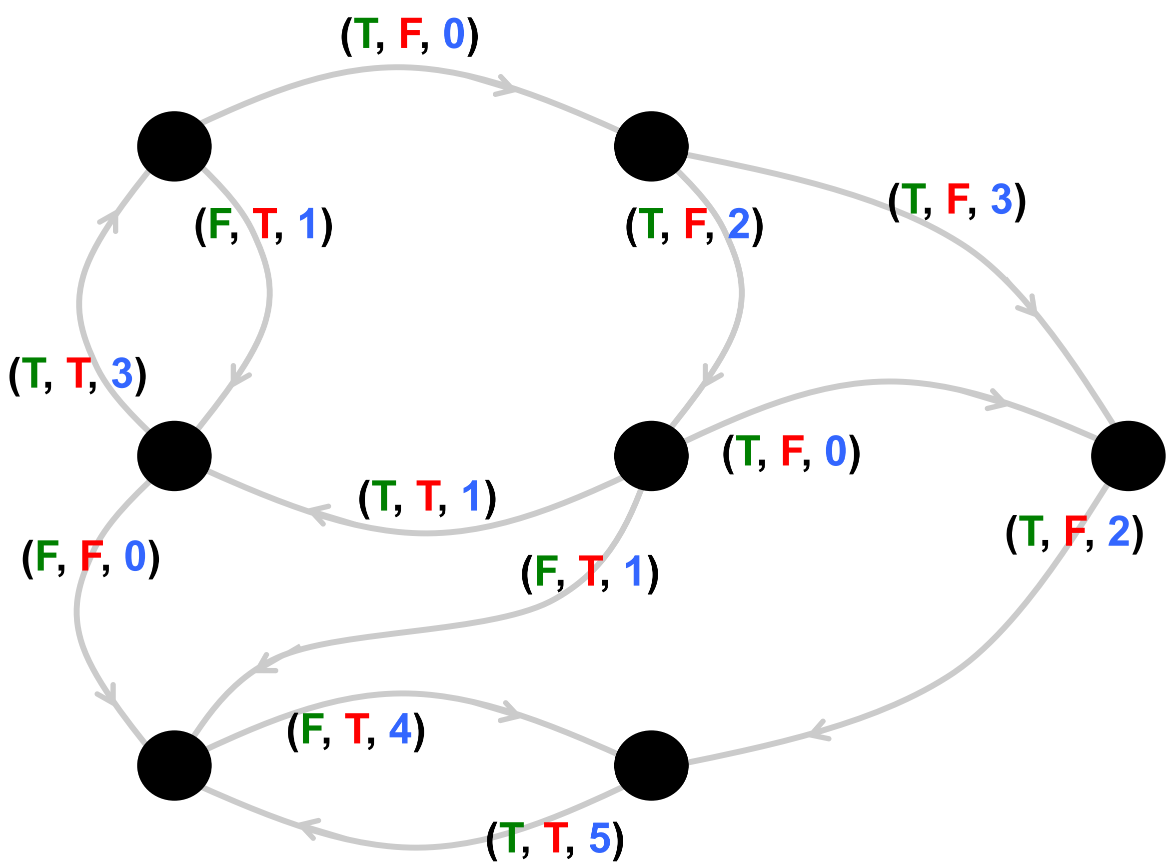
Performance comparison of KDT and PBGL breadth-first search. The reported numbers are in MegaTEPS, or 10^6 traversed edges per second. The graphs are Graph500 RMAT graphs with 2^{scale} vertices and $16*2^{scale}$ edges.

Plain graph



Connectivity only.

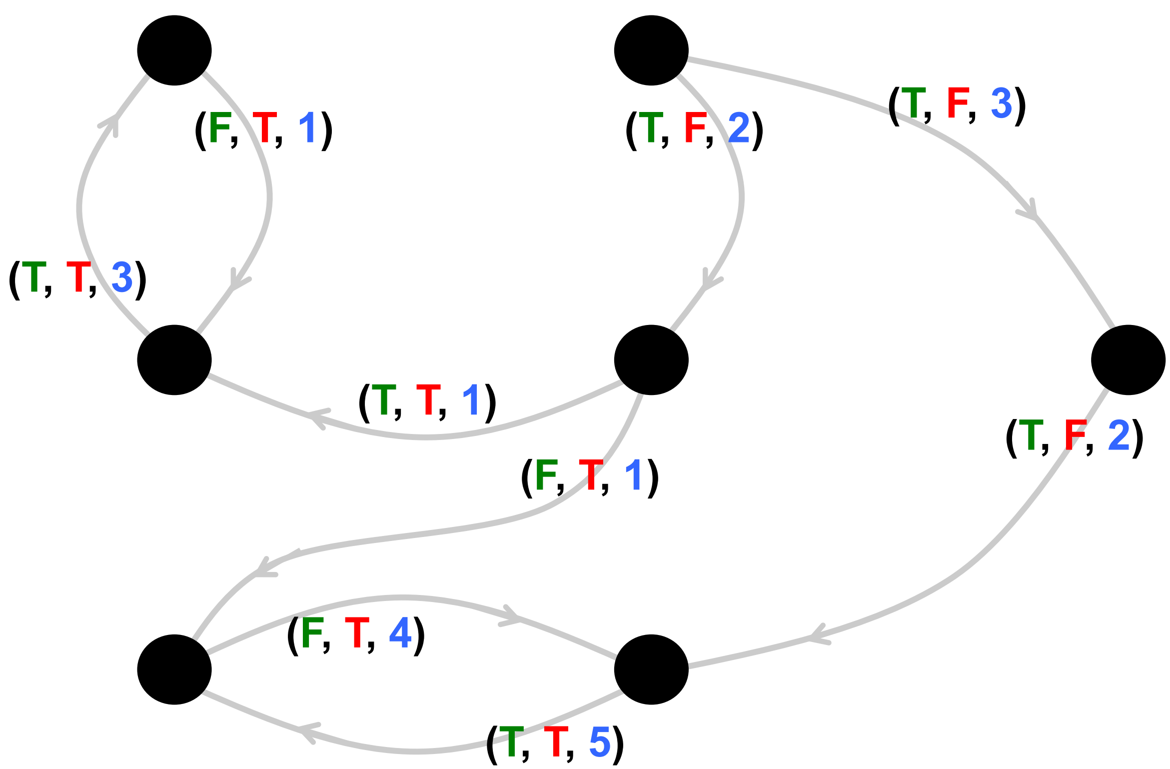
Edge Attributes (semantic graph)




```
class edge_attr:
    isText
    isPhoneCall
    weight
```

Edge Attribute Filter

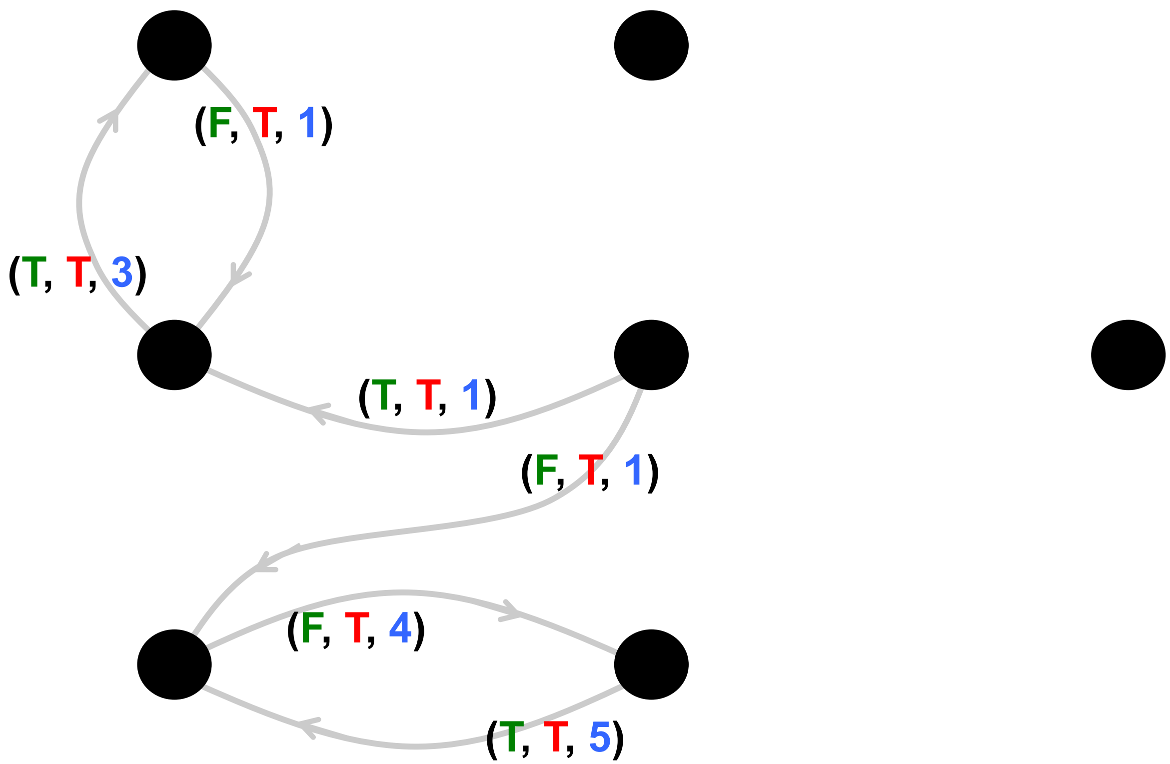
```
G.addEFilter(
lambda e: e.weight > 0)
```




```
class edge_attr:
    isText
    isPhoneCall
    weight
```

Edge Attribute Filter Stack

```
G.addEFilter(
lambda e: e.weight > 0)
G.addEFilter(
lambda e: e.isPhoneCall)
```




```
class edge_attr:
    isText
    isPhoneCall
    weight
```

Filter implementation details

- Filter defined as a unary predicate
  - operates on edge or vertex <u>value</u>
  - written in Python
  - predicates checked in order they were added
- Each KDT object maintains a stack of filter predicates
  - all operations respect filter
    - enables filter-ignorant algorithm design
    - enables algorithm designers to use filters

Two filter modes

- On-The-Fly filters
  - predicate checked each time an operation touches vertex or edge

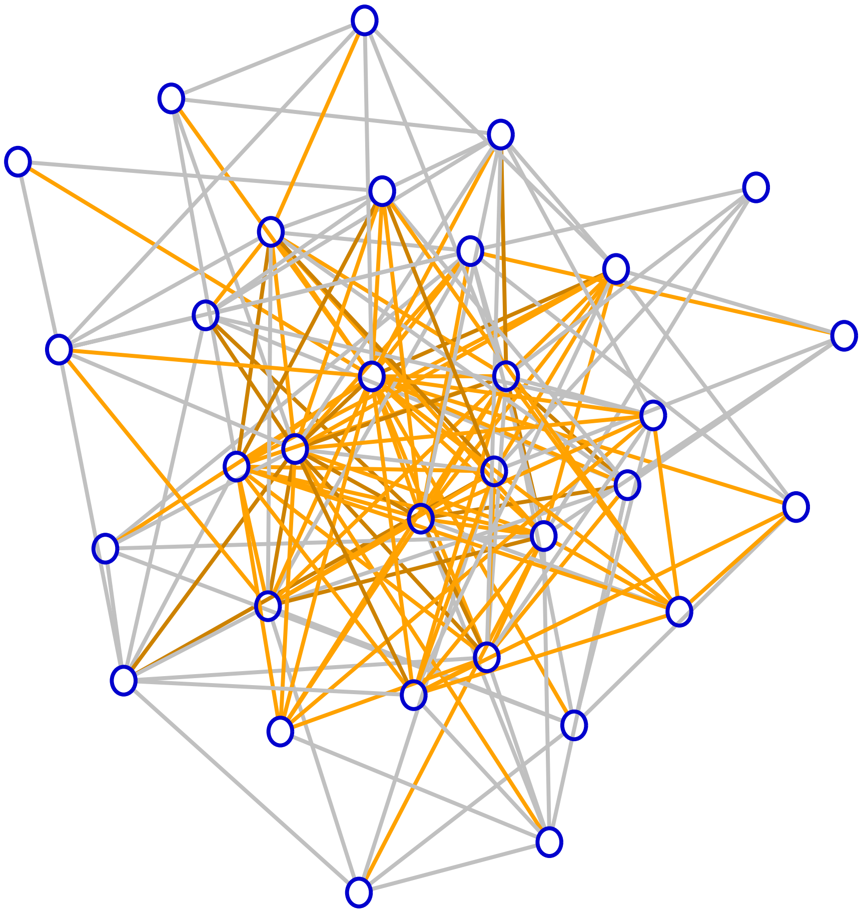
- Materialized filters
  - make copy of graph which excludes filtered elements
    - predicate checked only once for each element

Performance of On-The-Fly filter vs. Materialized filter

- For restrictive filter
  - OTF can be cheaper since fewer edges are touched
    - corpus can be huge, but only traverse small pieces

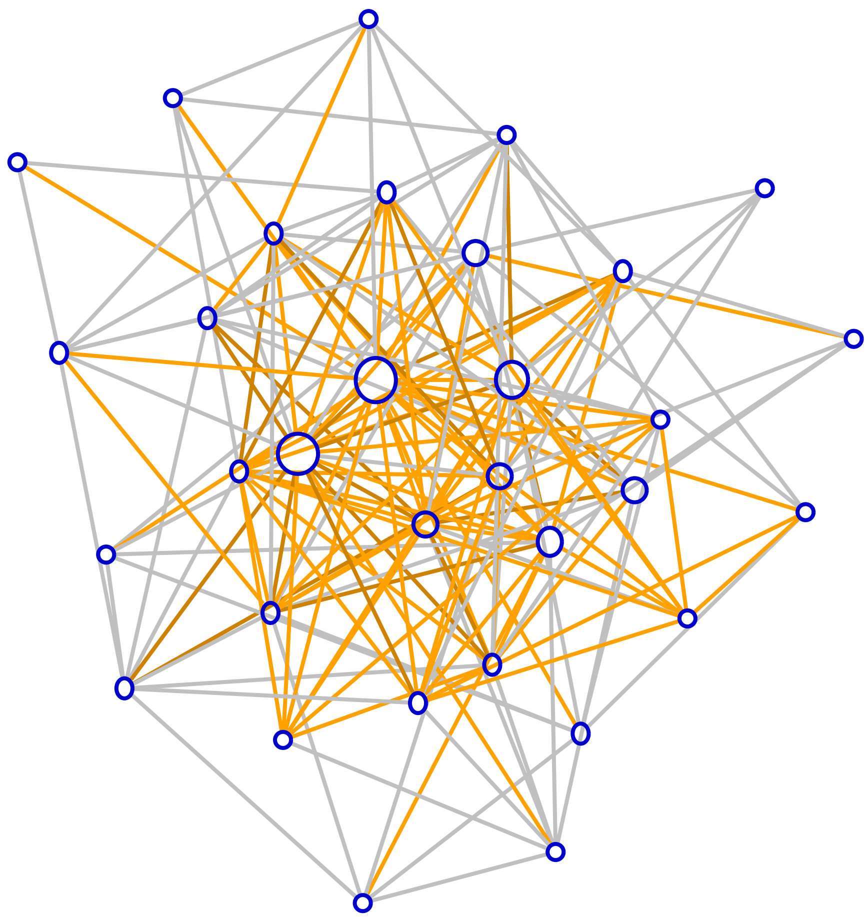
- For non-restrictive filter
  - OTF Saves space (no need to keep two large copies)
  - OTF Makes each operation more computationally expensive

texts and phone calls



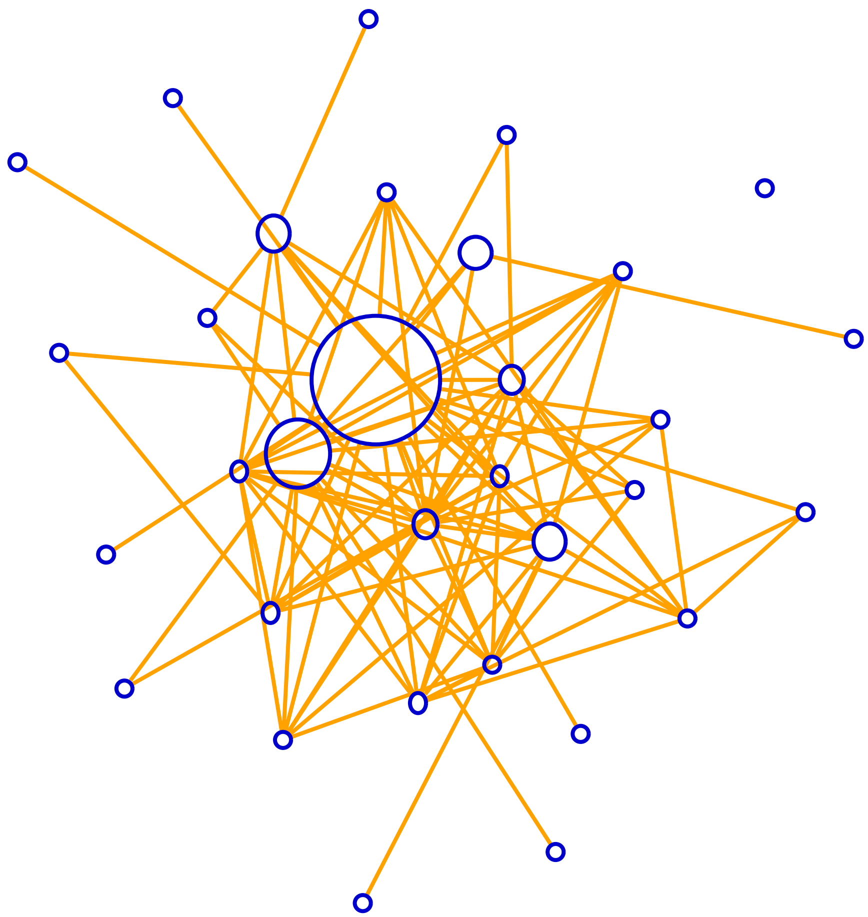
```
# draw graph
draw(G)

# Each edge has this attribute:
class edge_attr:
    isText
    isPhoneCall
    weight
```

Betweenness Centrality



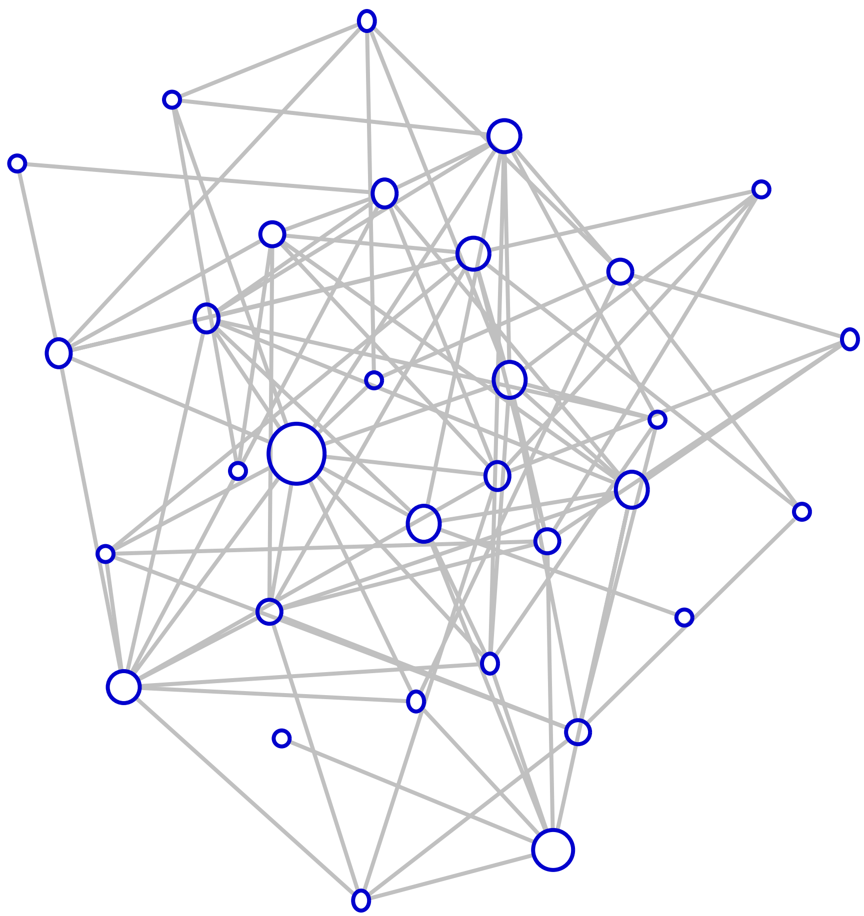
```
bc = G.centrality("approxBC")
# draw graph with node sizes
# proportional to BC score
draw(G, bc)
```

Betweenness Centrality on texts



```
# BC only on text edges
G.addEFilter(
      lambda e: e.isText)
bc = G.centrality("approxBC")
# draw graph with node sizes
# proportional to BC score
draw(G, bc)
```

Betweenness Centrality on calls



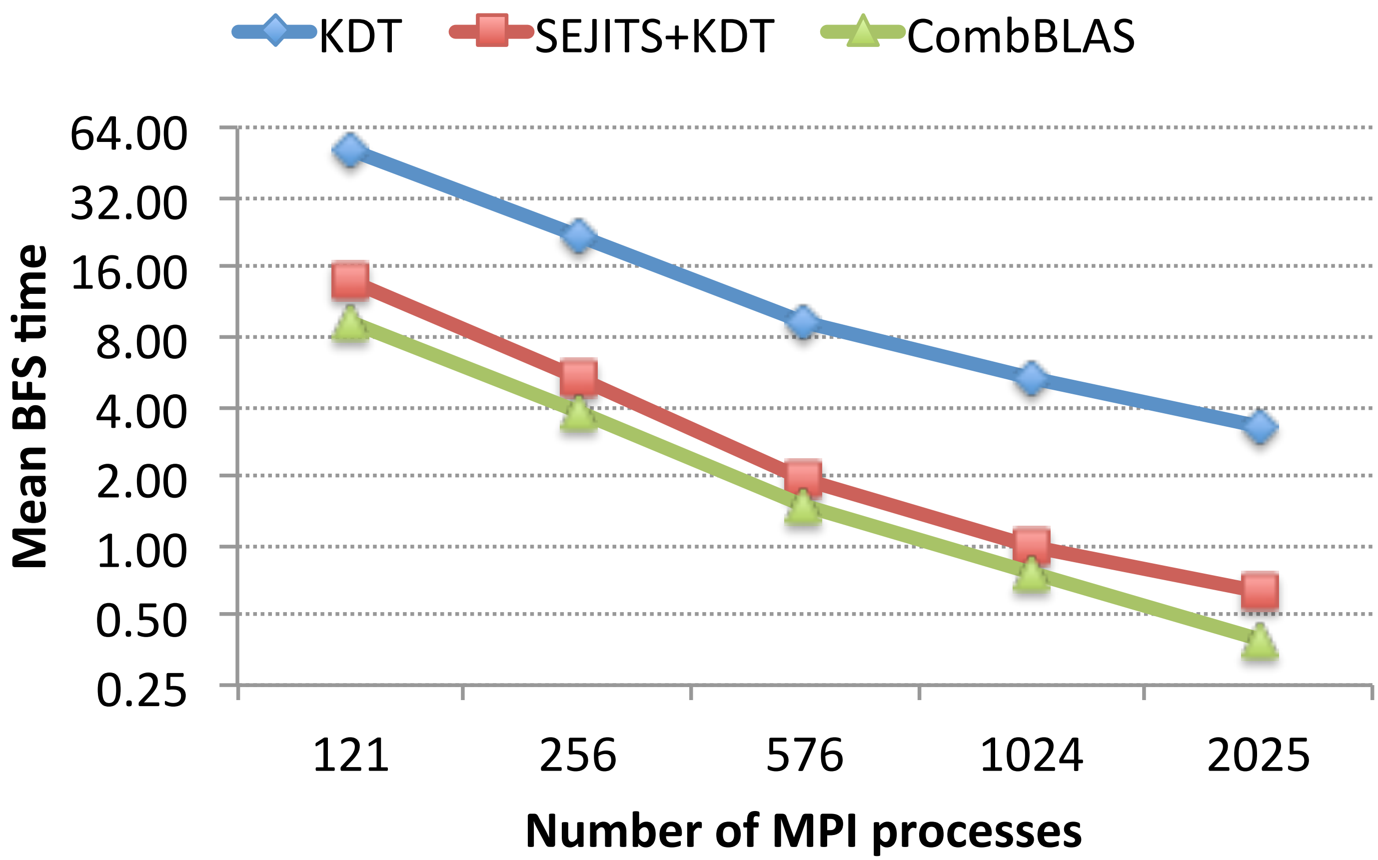
```
# BC only on phone call edges
G.addEFilter(
      lambda e: e.isPhoneCall)
bc = G.centrality("approxBC")
# draw graph with node sizes
# proportional to BC score
draw(G, bc)
```

SEJITS

The way to make Python fast is to not use Python.
-- Me

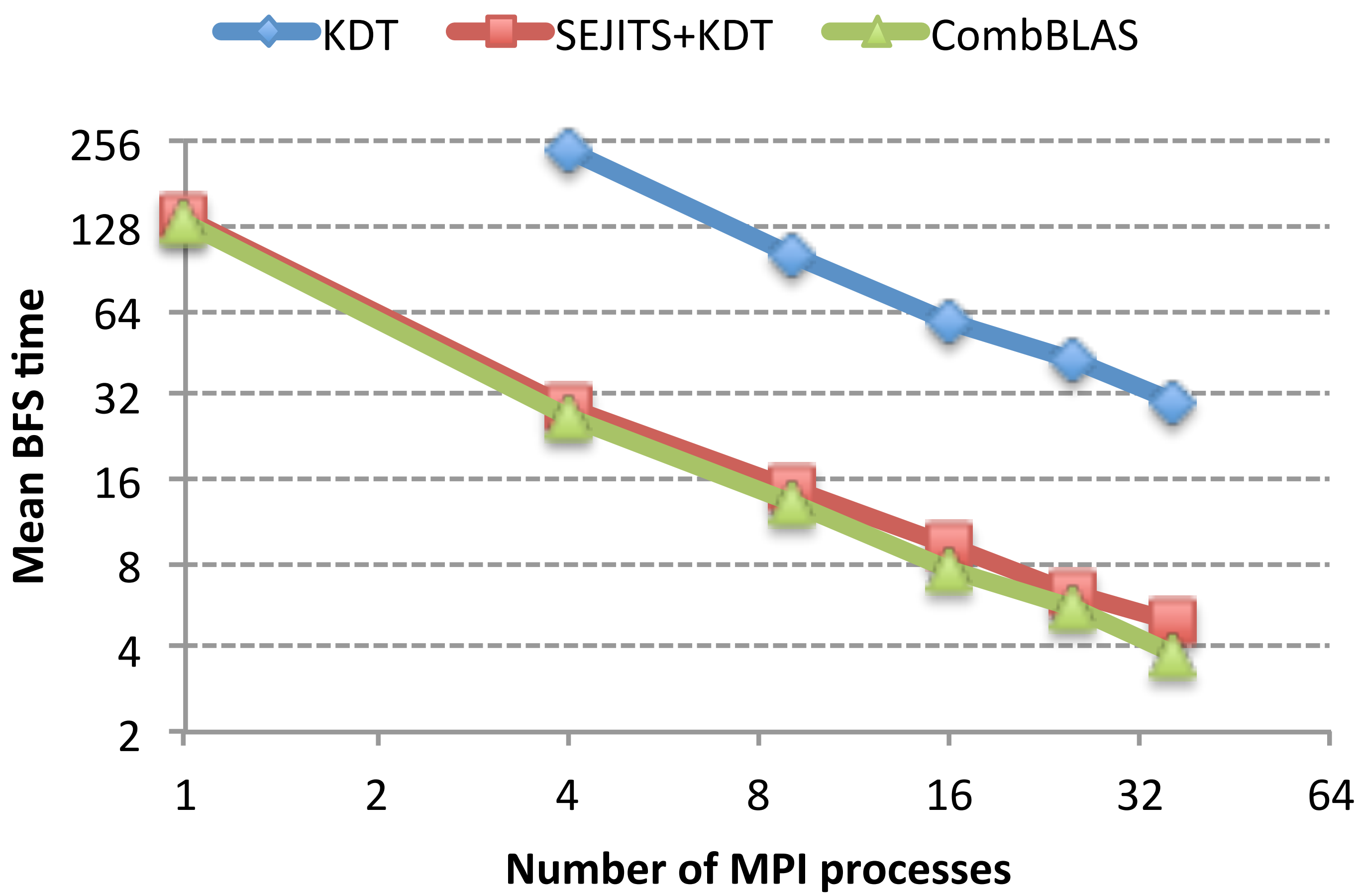
- Selective Embedded Just-In-Time Specialization
  1. Take Python code
  2. Translate it to equivalent C++ code
  3. Compile with GCC
  4. Call compiled version instead of Python version

BFS with SEJITS



Time (in seconds) for a single BFS iteration on Scale 25 RMAT (33M vertices, 500M edges) with 10% of elements passing filter. Machine is NERSC's Hopper.

BFS with SEJITS

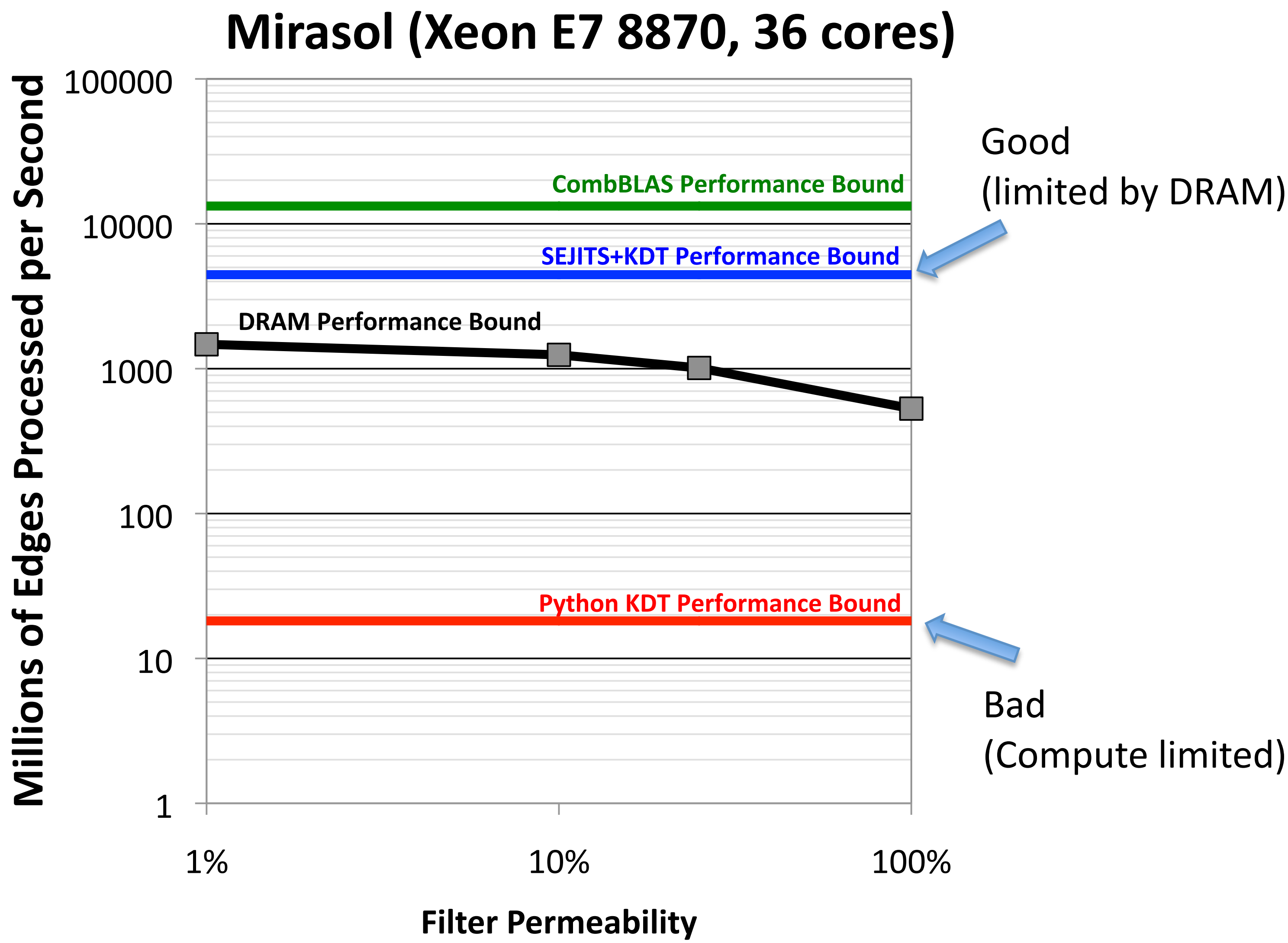


Time (in seconds) for a single BFS iteration on Scale 23 RMAT (8M vertices, 130M edges) with 10% of elements passing filter. Machine is Mirasol.

Roofline

- A way to find what your bottleneck is
- MEASURE and PLOT potential limiting factors in your exact system and program
  - compute power
  - RAM stream speed
  - RAM random access speed
  - disk
  - etc
- Your Roofline is the minimum of your plots

KDT + SEJITS Roofline



Is MapReduce any good for graphs?

The prospect of the entire graph traversing the cloud fabric for each MapReduce job is disturbing.

- Jonathan Cohen

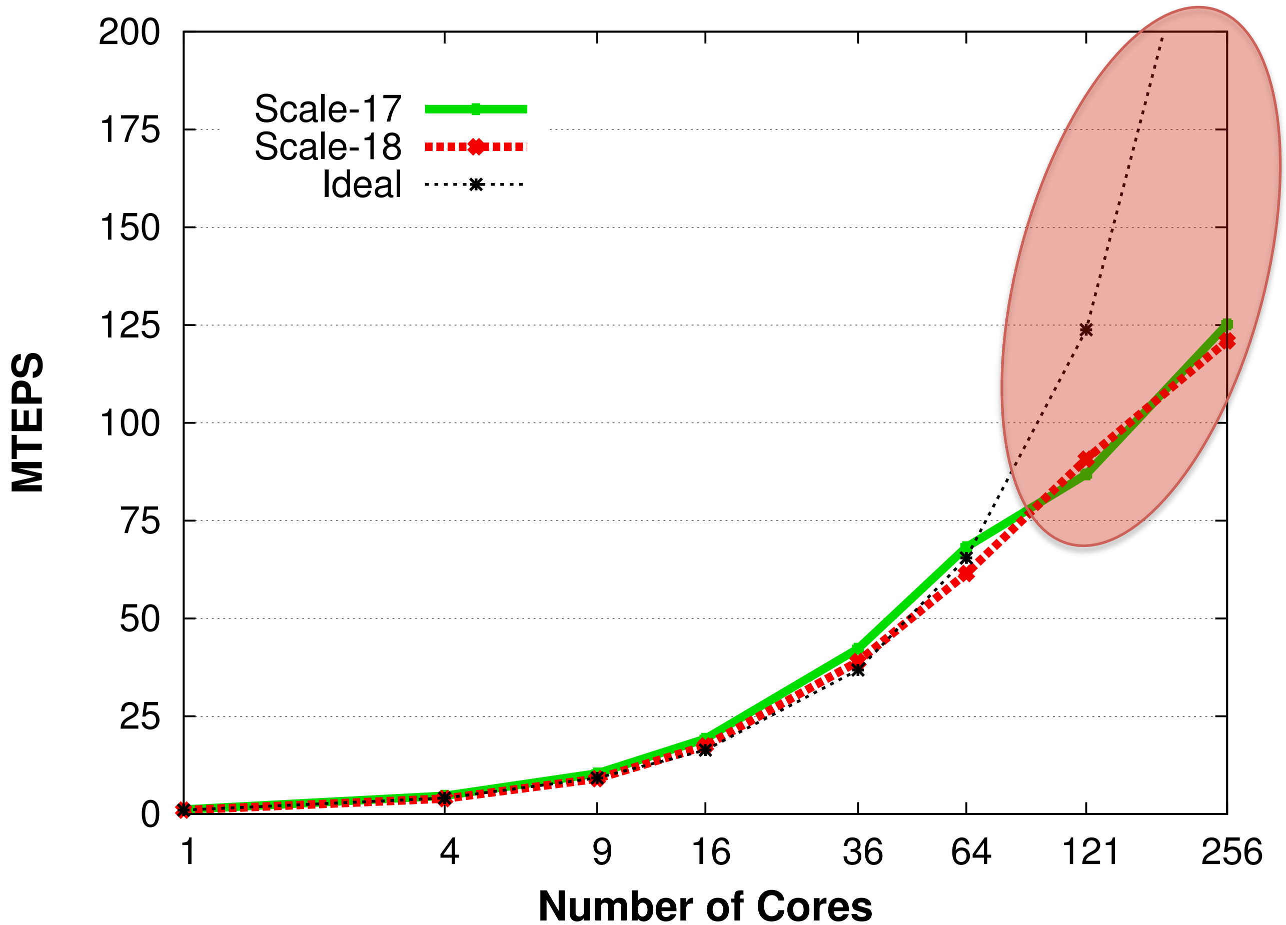
PageRank comparison with Pegasus

MapReduce-based

Core Count	Task Count	Code	Problem Size: Scale 19	Scale 21
-	4	Pegasus	2h 35m 10s	6h 06m 10s
4	-	KDT	55s	7m 12s
-	16	Pegasus	33m 09s	4h 40m 08s
16	-	KDT	13s	1m 34s

Performance comparison of KDT and Pegasus PageRank ($\varepsilon = 10^{-7}$). The graphs are Graph500 RMAT graphs. The machine is Neumann, a 32-core shared memory machine with HDFS mounted in a ramdisk.

A Scalability limit for matrix-matrix multiplication: sqrt(p)



Million Traversed Edges Per Second in Betweenness Centrality computation. BC algorithm is composed of multiple BFS searches batched together into matrices and using SpGEMM for traversals.