Three Goals in Parallel Graph Computations: High Performance, High Productivity, and Reduced Communication

Aydın Buluç
Berkeley Lab (LBNL)
October 21, 2013

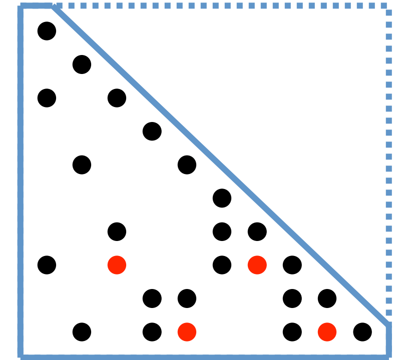
Acknowledgments (past & present)

- Krste Asanovic (UC Berkeley)
- Grey Ballard (UC Berkeley)
- Scott Beamer (UC Berkeley)
- Jim Demmel (UC Berkeley)
- Erika Duriakova (UC Dublin)
- Armando Fox (UC Berkeley)
- John Gilbert (UCSB)
- Laura Grigori (INRIA)
- Shoaib Kamil (MIT)
- Ben Lipshitz (UC Berkeley)
- Adam Lugowski (UCSB)
- Kamesh Madduri (Penn State)
- Lenny Oliker (Berkeley Lab)
- Dave Patterson (UC Berkeley)
- Steve Reinhardt (YarcData)
- Oded Schwartz (UC Berkeley)
- Edgar Solomonik (UC Berkeley)
- Veronika Strnadova (UCSB)
- Sivan Toledo (Tel Aviv Univ)
- Sam Williams (Berkeley Lab)

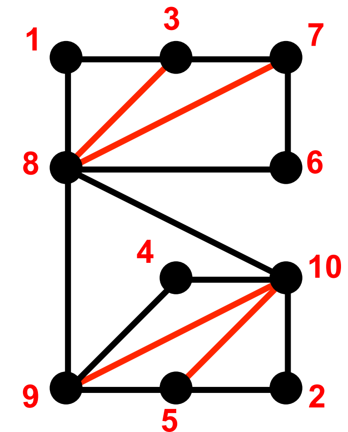
This work is funded by:

U.S. DEPARTMENT OF ENERGY | Office of Science

Graph abstractions in Computer Science



Compiler optimization:
Control flow graph : graph dominators
Register allocation: graph coloring



$G^+(A)$
[chordal]

Scientific computing:
Preconditioning: support graphs, spanning trees
Sparse direct solvers: chordal graphs
Parallel computing: graph separators

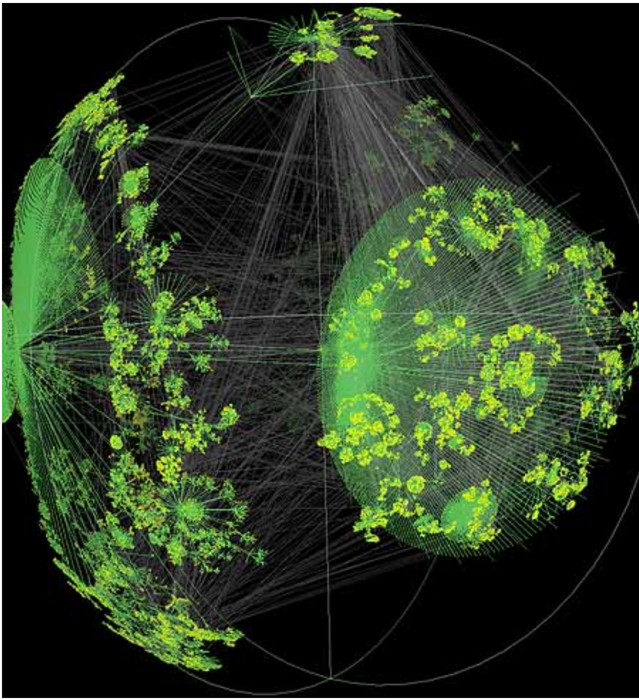
Computer Networks:
Routing: shortest path algorithms
Web crawling: graph traversal
Interconnect design: Cayley graphs

Large graphs are everywhere

Internet structure

Social interactions

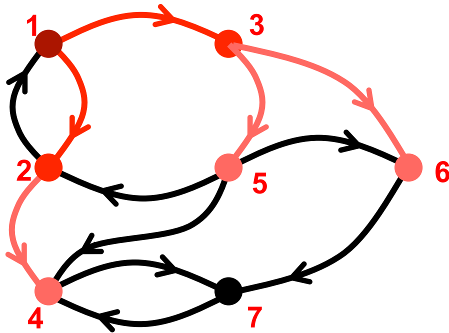
Scientific datasets: biological, chemical, cosmological, ecological, …

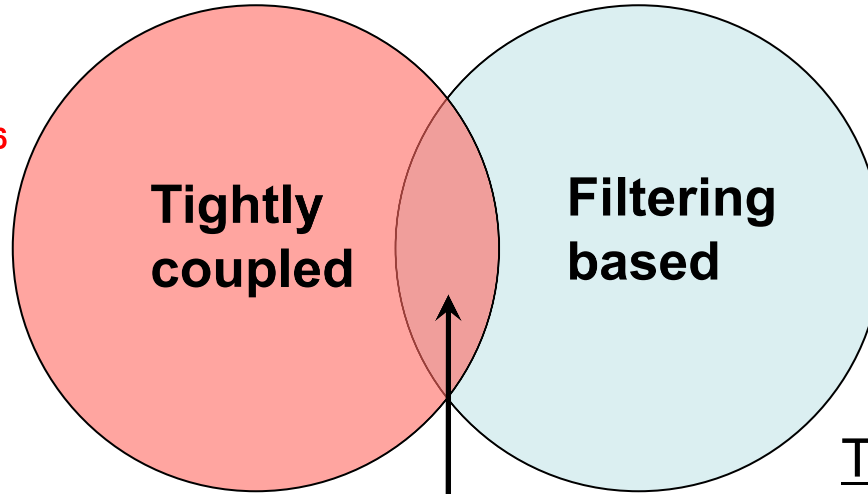


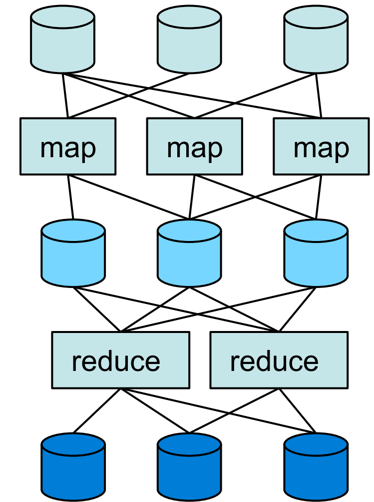
WWW snapshot, courtesy Y. Hyun

Yeast protein interaction network, courtesy H. Jeong

Types of graph computations







Tool: Graph Traversal

Examples:

- Centrality
- Shortest paths
- Network flows
- Strongly Connected Components

Fuzzy intersection

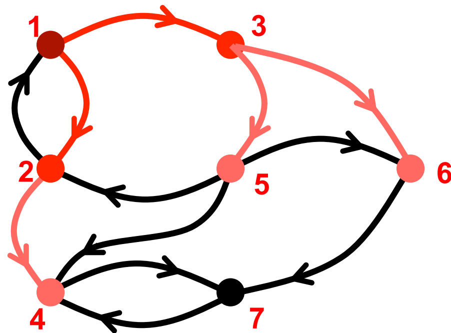
Examples: Clustering, Algebraic Multigrid

Tools: Map/Reduce, SPARQL engines

Examples:

- Loop and multi edge removal
- Triangle/Rectangle enumeration

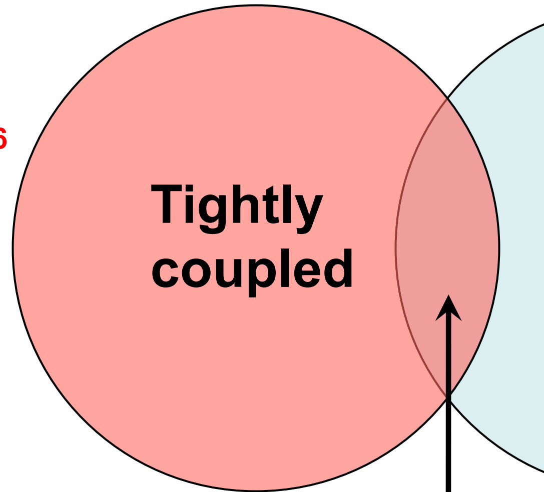
Types of graph computations

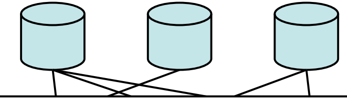


Tool: Graph Traversal

Examples:

- Centrality
- Shortest paths
- Network flows
- Strongly Connected Components





The left diagram is the focus of this talk.

Challenges :

- Difficult to parallelize
- Very low arithmetic intensity
- Unpredictable data access patterns

Fuzzy intersection

Examples: Clustering, Algebraic Multigrid

Examples:

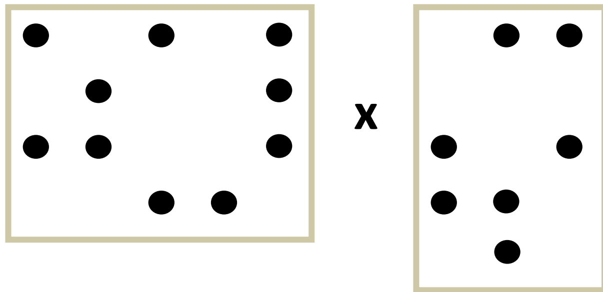
- Loop and multi edge removal
- Triangle/Rectangle enumeration

Part 1: High performance

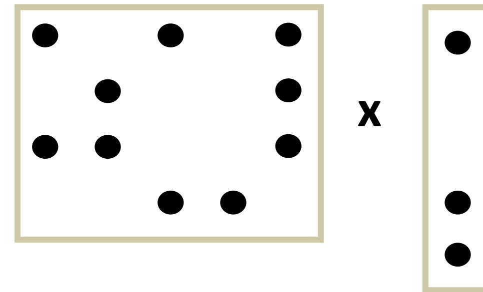
Crux: Linear-algebraic primitives

Linear-algebraic primitives for graphs

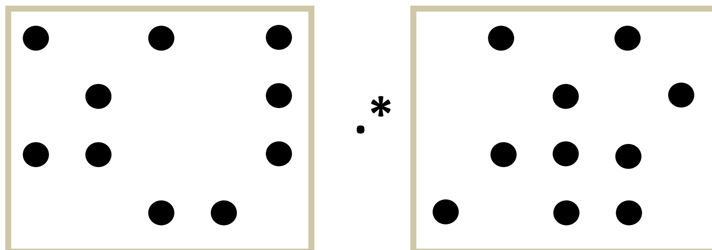
Sparse matrix-sparse matrix multiplication



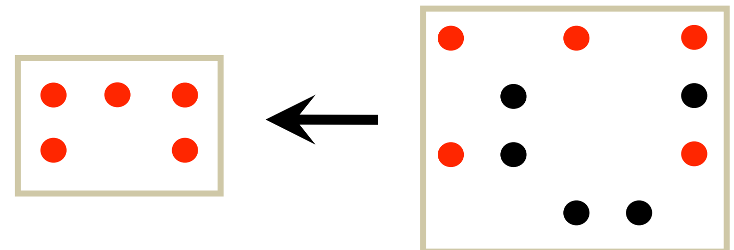
Sparse matrix-sparse vector multiplication



Element-wise operations



Sparse matrix indexing



The Combinatorial BLAS implements these, and more, on arbitrary semirings, e.g. (×, +), (and, or), (+, min)

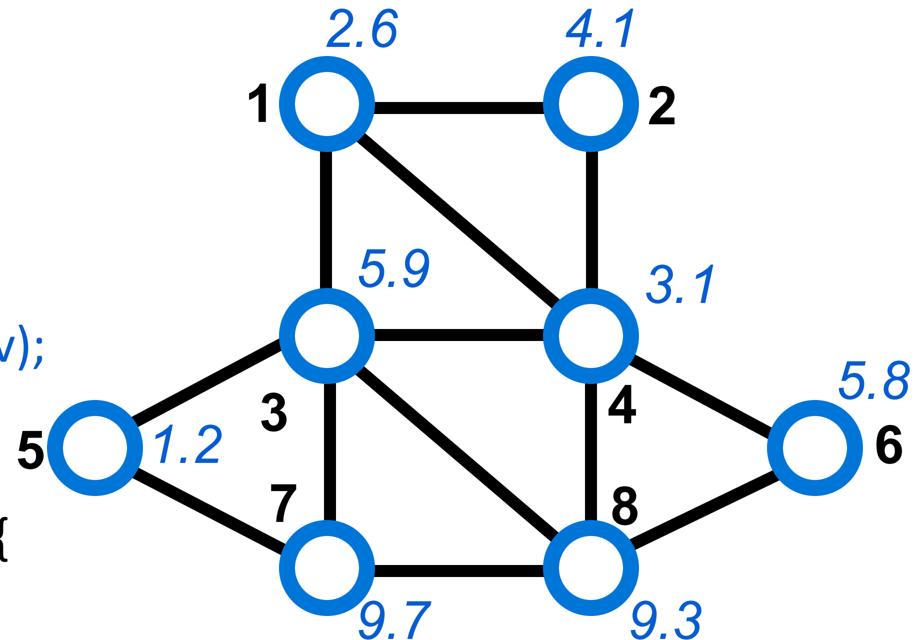
Some Combinatorial BLAS functions*

* Grossly simplified (parameters, semiring)

Function	Parameters	Returns	Math Notation
SpGEMM	- sparse matrices **A** and **B** - unary functors	sparse matrix	**C** = op(**A**) * op(**B**)
SpM{S/D}V	- sparse matrix **A** - sparse/dense vector **x**	sparse/dense vector	**y** = **A** * **x**
SpEWiseX	- sparse matrices or vectors - binary functor and predicate	in place or sparse matrix/vector	**C** = **A** .* **B**
Reduce	- sparse matrix **A** and functors	dense vector	**y** = sum(**A**, op)
SpRef	- sparse matrix **A** - index vectors **p** and **q**	sparse matrix	**B** = **A**(**p**,**q**)
SpAsgn	- sparse matrices **A** and **B** - index vectors **p** and **q**	none	**A**(**p**,**q**) = **B**
Scale	- sparse matrix **A** - dense matrix or vector **X**	none	check manual
Apply	- any matrix or vector **X**	none	op(**X**)

Parallel, randomized maximal independent set algorithm

1. S = empty set; C = V;
2. while C is not empty {
3. label each v in C with a random r(v);
4. for all v in C in parallel {
5. if r(v) < min(r(neighbors of v)) {
6. move v from C to S;
7. remove neighbors of v from C;
8. }
9. }
10. }



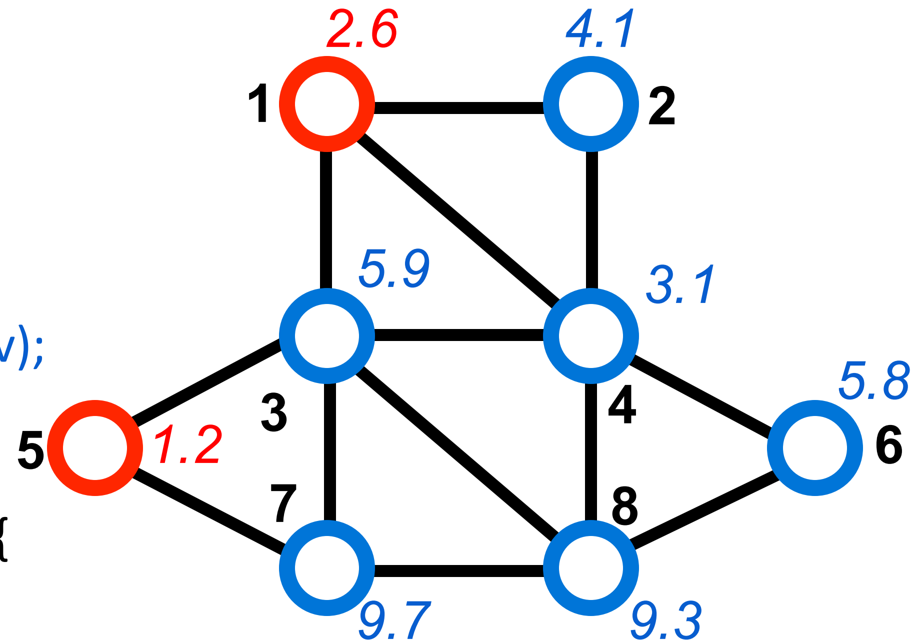
S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

M. Luby. "A Simple Parallel Algorithm for the Maximal Independent Set Problem". *SIAM Journal on Computing* **15** (4): 1036–1053, 1986

Parallel, randomized maximal independent set algorithm

1. S = empty set; C = V;
2. while C is not empty {
3.   label each v in C with a random r(v);
4.   for all v in C in parallel {
5.   if r(v) < min(r(neighbors of v)) {
6.   move v from C to S;
7.   remove neighbors of v from C;
8.   }
9.   }
10. }

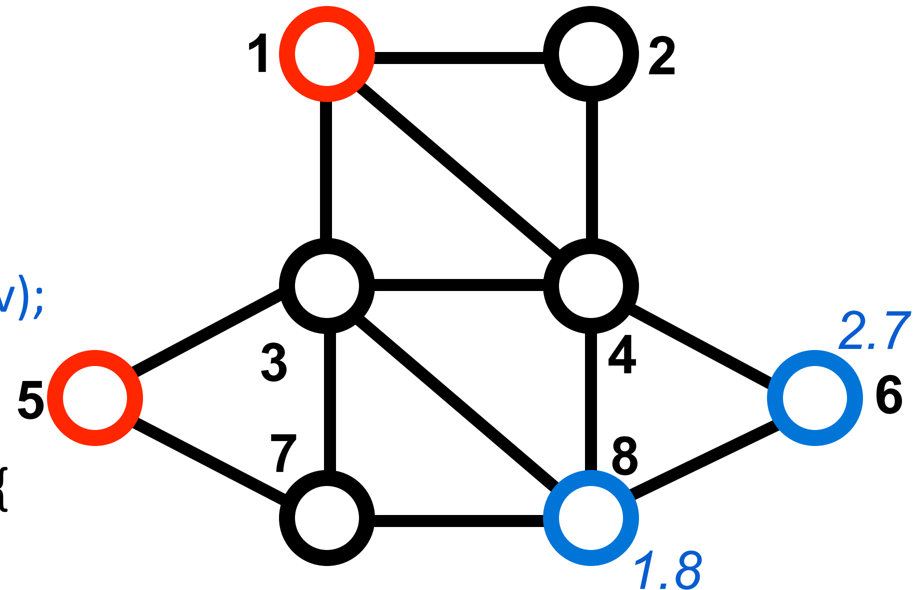


S = { 1, 5 }

C = { 6, 8 }

Parallel, randomized maximal independent set algorithm

1. S = empty set; C = V;
2. while C is not empty {
3. label each v in C with a random r(v);
4. for all v in C in parallel {
5. if r(v) < min(r(neighbors of v)) {
6. move v from C to S;
7. remove neighbors of v from C;
8. }
9. }
10. }

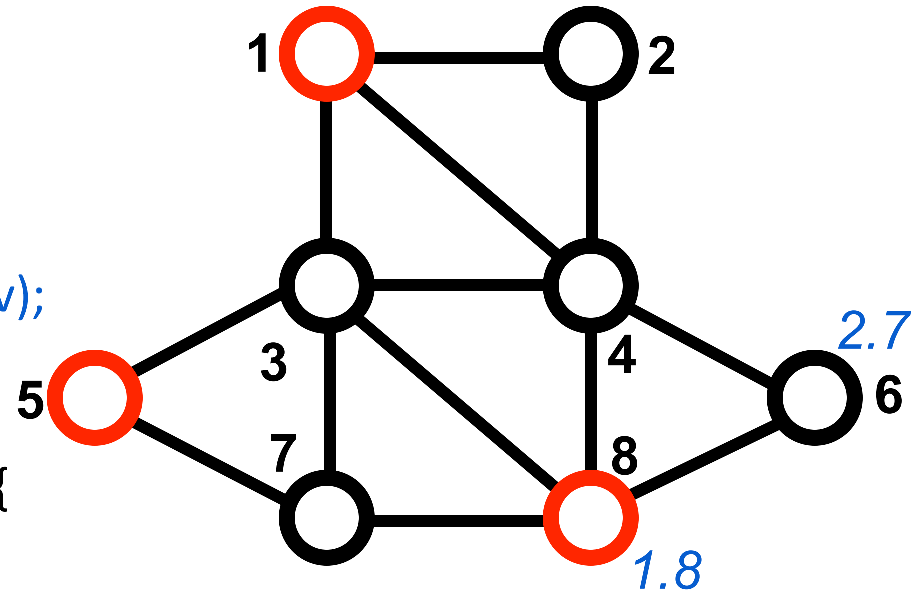


S = { 1, 5 }

C = { 6, 8 }

Parallel, randomized maximal independent set algorithm

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```



S = { 1, 5, 8 }

C = { }

<u>Theorem</u>: This algorithm "very probably" finishes within O(log n) rounds.

Luby's algorithm in KDT

Input: A^T (transpose of Graph's adjacency matrix)
Output: MIS (maximal independent set)

```
while (candidates.nnz()>0):  # number of nonzeros

    candidates.Apply (rand) # label each vertex in candidates with random value

    # find the smallest random value among a vertex's neighbors
    minadj = A^T.SpSMV (candidates, semiring(min,select2nd))

    # Add vertices to S if its --own value-- is smallest among neighbors
    newS = minadj.SpEWiseX (candidates, op=[]{return 1}, predicate=is2ndSmaller,…)

    # newS are no longer candidates, so remove them
    candidates. SpEWiseX (newS, op=[]{return 1}, …)  # in place

    sadj = A^T.SpSMV (newS, semiring(select2nd,select2nd)) # find neighbors of newS

    # remove sadj from candidate as well; they can't be part of the MIS
    candidates. SpEWiseX (sadj, op=[]{return 1}, …)    # in place

    MIS += newS # add new_S to MIS
```

vectors are sparse : often faster than $\Theta(m)$

Graph algorithm comparison (LA: linear algebra)

Slide inspiration: Jeremy Kepner (MIT LL)

Algorithm (Problem)	Canonical Complexity	LA-Based Complexity	Critical Path (for LA)
Breadth-first search	$\Theta(m)$	$\Theta(m)$	$\Theta(diameter)$
Betweenness Centrality (unweighted)	$\Theta(mn)$	$\Theta(mn)$	$\Theta(diameter)$
All-pairs shortest-paths (dense)	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n)$
Prim (MST)	$\Theta(m+n \log n)$	$\Theta(n^2)$	$\Theta(n)$
Borůvka (MST)	$\Theta(m \log n)$	$\Theta(m \log n)$	$\Theta(\log^2 n)$
Edmonds-Karp (Max Flow)	$\Theta(m^2n)$	$\Theta(m^2n)$	$\Theta(mn)$
Greedy MIS (MIS)	$\Theta(m+n \log n)$	$\Theta(mn+n^2)$	$\Theta(n)$
Luby (MIS)	$\Theta(m+n \log n)$	$\Theta(m \log n)$	$\Theta(\log n)$

Majority of selected algorithms can be represented with array-based constructs with equivalent complexity.

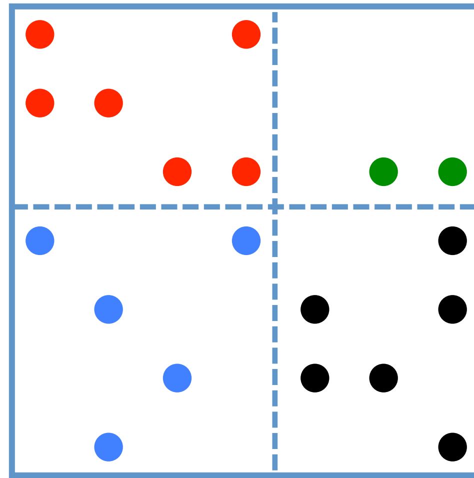
($n = |V|$ and $m = |E|$)

The case for sparse matrices

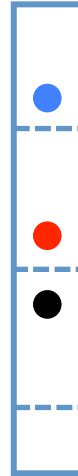
Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

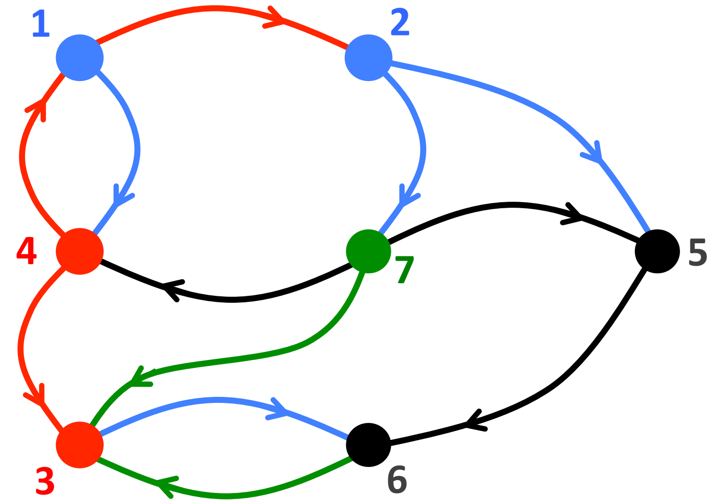
Traditional graph computations	Graphs in the language of linear algebra
Data driven, unpredictable communication.	Fixed communication patterns
Irregular and unstructured, poor locality of reference	Operations on matrix blocks exploit memory hierarchy
Fine grained data accesses, dominated by latency	Coarse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

2D Parallel BFS algorithm (variants top Graph500)



x





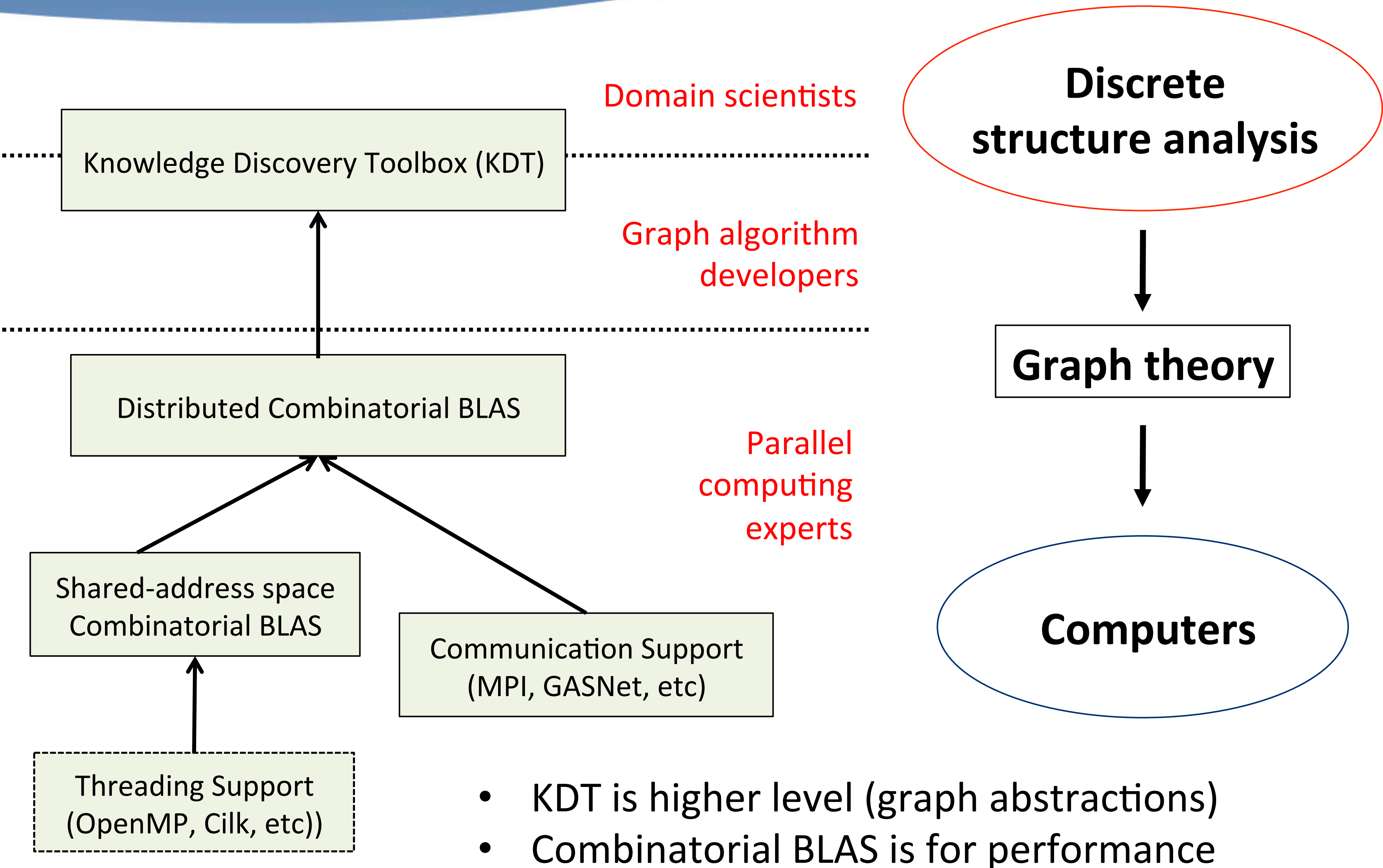
ALGORITHM:

1. Gather vertices in *processor column* **[communication]**
2. Find owners of the current frontier's adjacency [computation]
3. Exchange adjacencies in *processor row* **[communication]**
4. Update distances/parents for unvisited vertices. [computation]

Part 2: High productivity

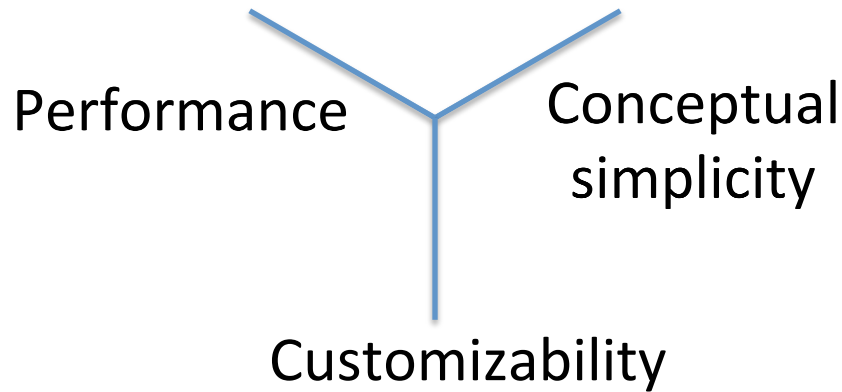
Crux: Very high-level languages and selective just-in-time translation using domain-specific languages.

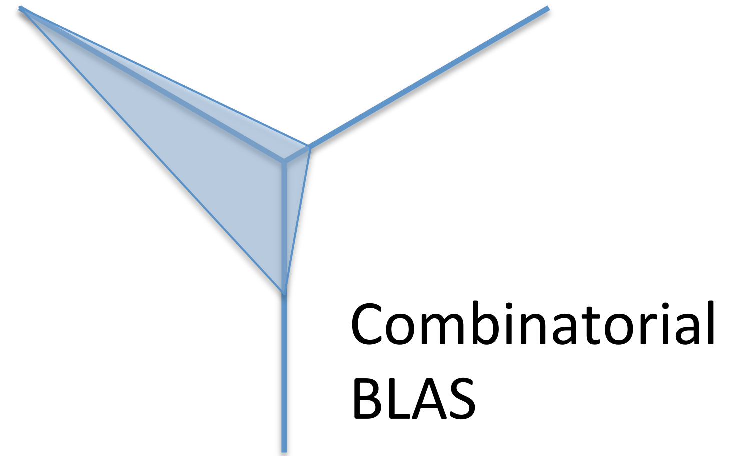
Parallel Graph Analysis Software

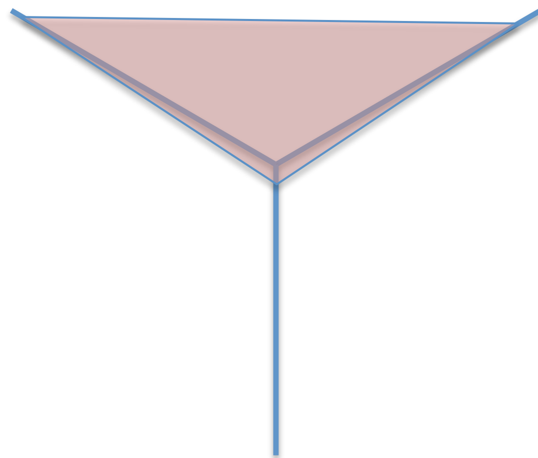


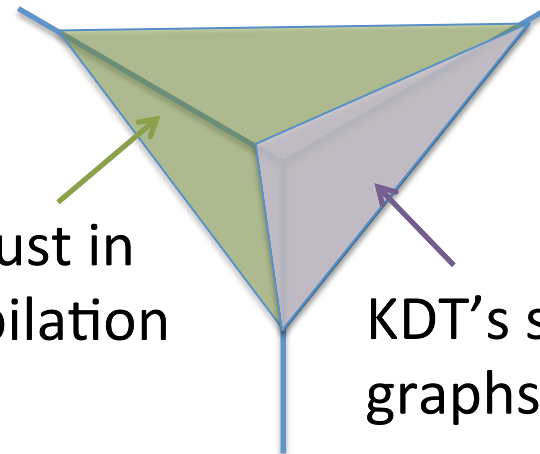
- KDT is higher level (graph abstractions)
- Combinatorial BLAS is for performance

How to target "domain experts"?









Attributed semantic graphs and filters

Example:

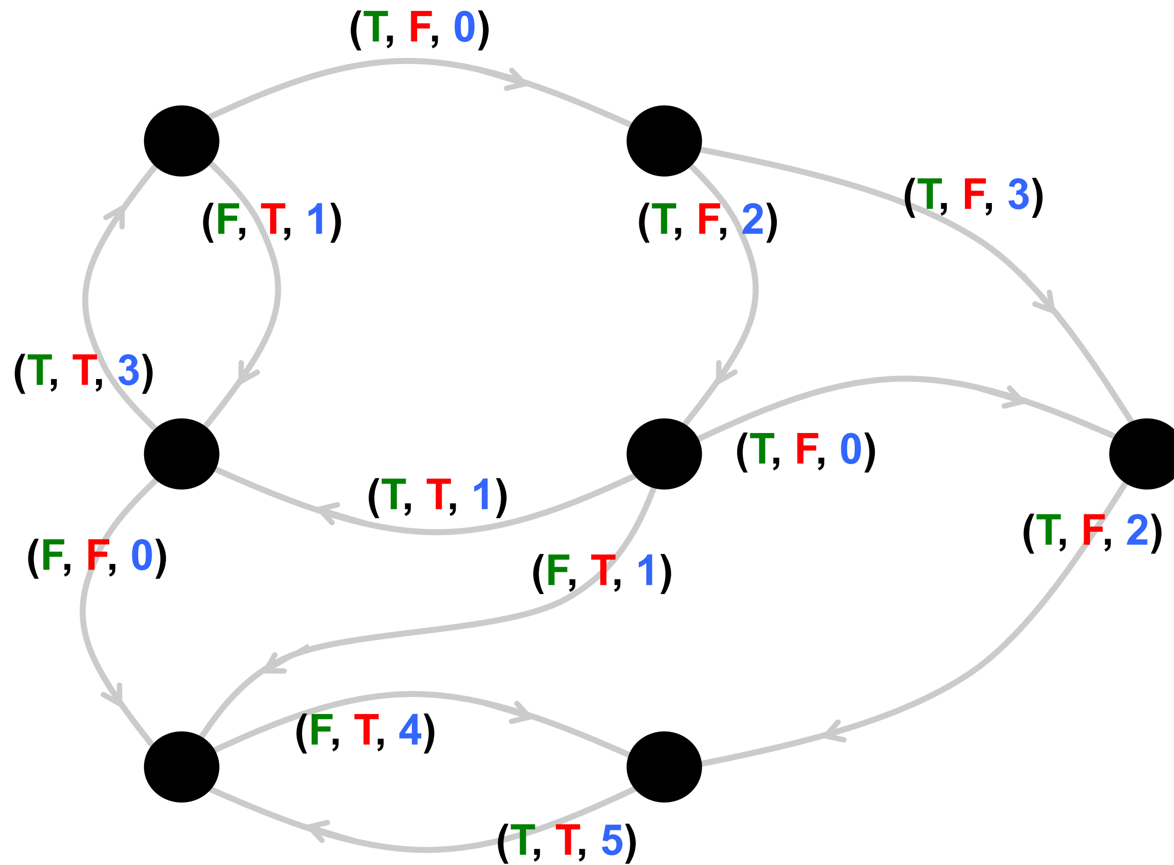
- Vertex types: Person, Phone, Camera, Gene, Pathway
- Edge types: PhoneCall, TextMessage, CoLocation, Sequence Similarity
- Edge attributes: StartTime, EndTime

- Calculate centrality just for emails among engineers sent between times sTime and eTime

```
def onlyEngineers(self):
    return self.position == Engineer

def timedEmail (self, sTime, eTime):
    return ((self.type == email) and
           (self.Time > sTime) and
           (self.Time < eTime))

start = dt.now() - dt.timedelta(days=30)
end = dt.now()

# G denotes the graph
G.addVFilter(onlyEngineers)
G.addEFilter(timedEmail(start, end))

# rank via centrality based on recent
email transactions among engineers
bc = G.rank('approxBC')
```

Lugowski, Alber, **B.**, Gilbert, Reinhardt, Teng, and Waranis. A flexible open-source toolbox for scalable complex graph analysis. *SIAM Conference on Data Mining (SDM)*,2012

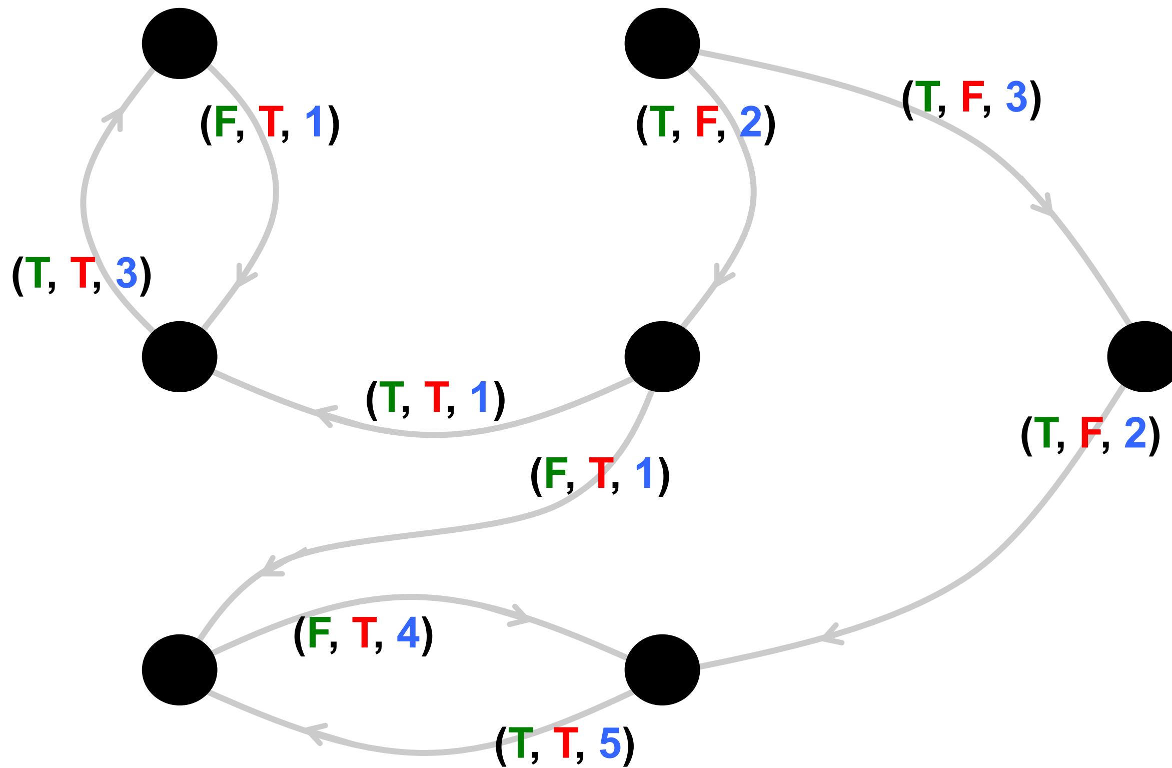
Edge filter illustration




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

Edge filter illustration

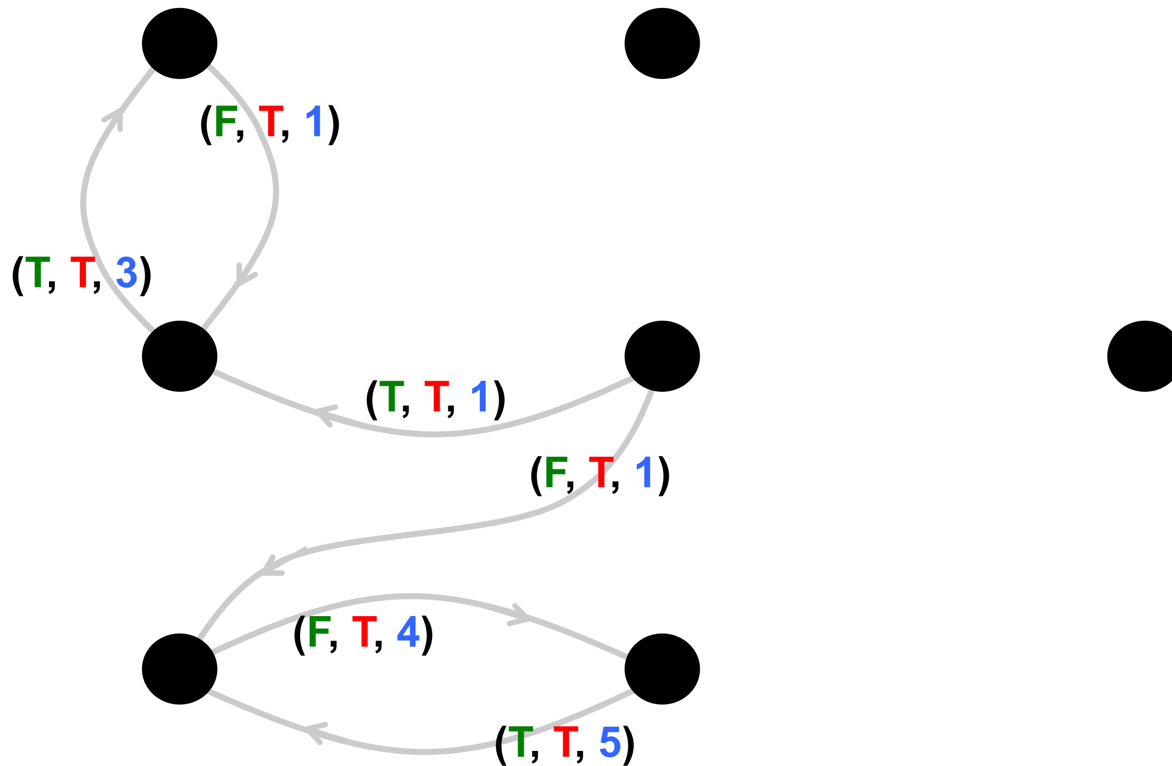
```
G.addEFilter(lambda e: e.weight > 0)
```




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

Edge filter illustration

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



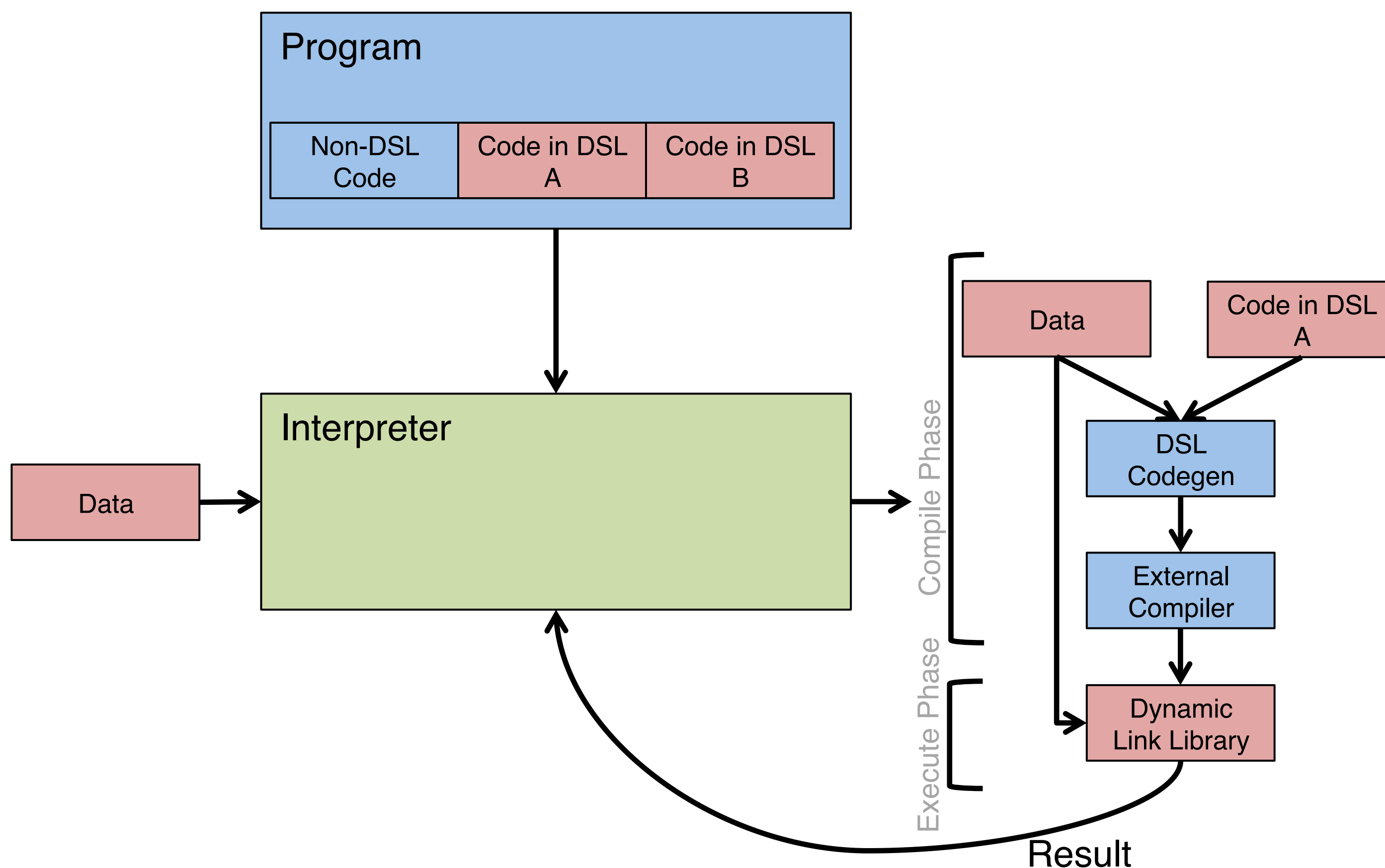

```
class edge_attr:
    isText
    isPhoneCall
    weight
```

Problems with Customizing in KDT

- Filtering on attributed semantic graphs is slow
  - In plain KDT, filters are pure Python functions.
  - Requires a per-vertex or per-edge upcall into Python
  - Can be as slow as 80X compared to pure C++

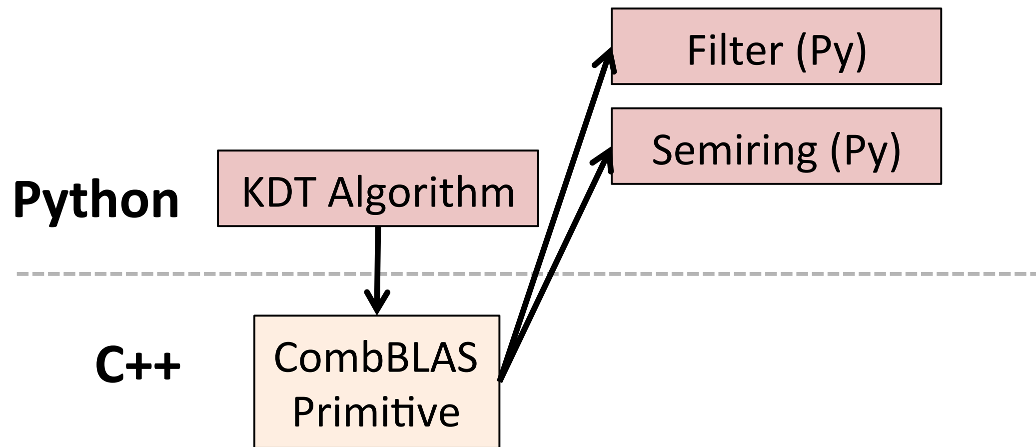
- Adding new graph algorithms to KDT is slow
  - A new graph algorithm = composing linear algebraic primitives + customizing the *semiring* operation
  - *Semirings* in Python; similar performance bottleneck

Review: Selective Embedded Just In Time Specialization (SEJITS)

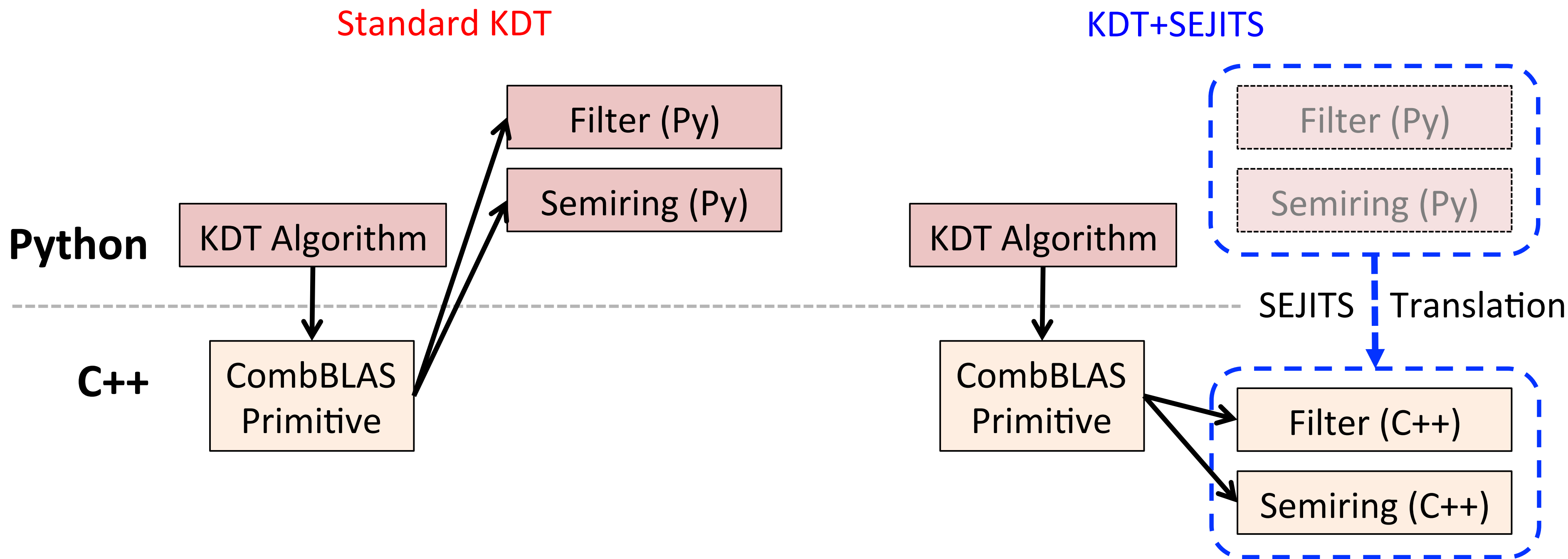


Catanzaro, Kamil, Lee, Asanovic, Demmel, Keutzer, Shalf, Yelick, Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. *PMEA,* 2009

SEJITS for filter/semiring acceleration



SEJITS for filter/semiring acceleration

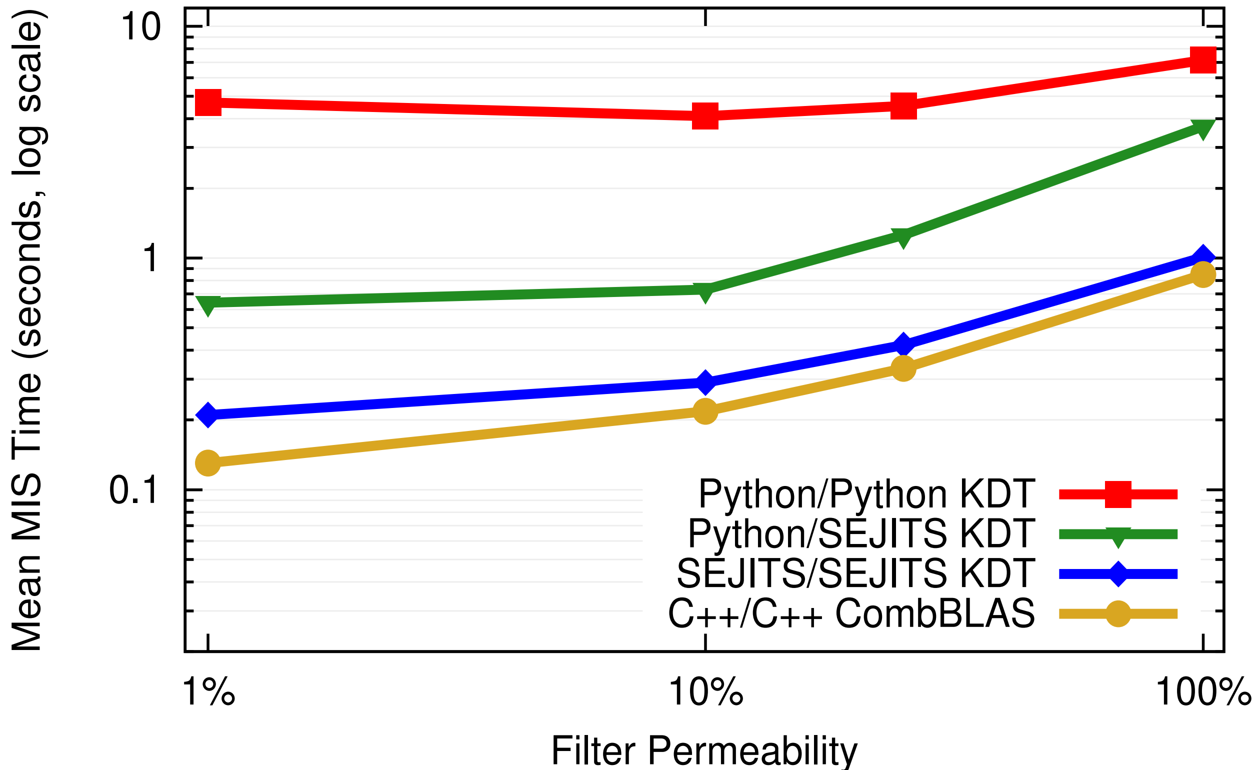


Embedded DSL: Python for the whole application

- Introspect, translate Python to equivalent C++ code
- Call compiled/optimized C++ instead of Python

B., Duriakova, Gilbert, Fox, Kamil, Lugowski, Oliker, Williams. High-Performance and High-Productivity Analysis of Filtered Semantic Graphs, *IPDPS*, 2013

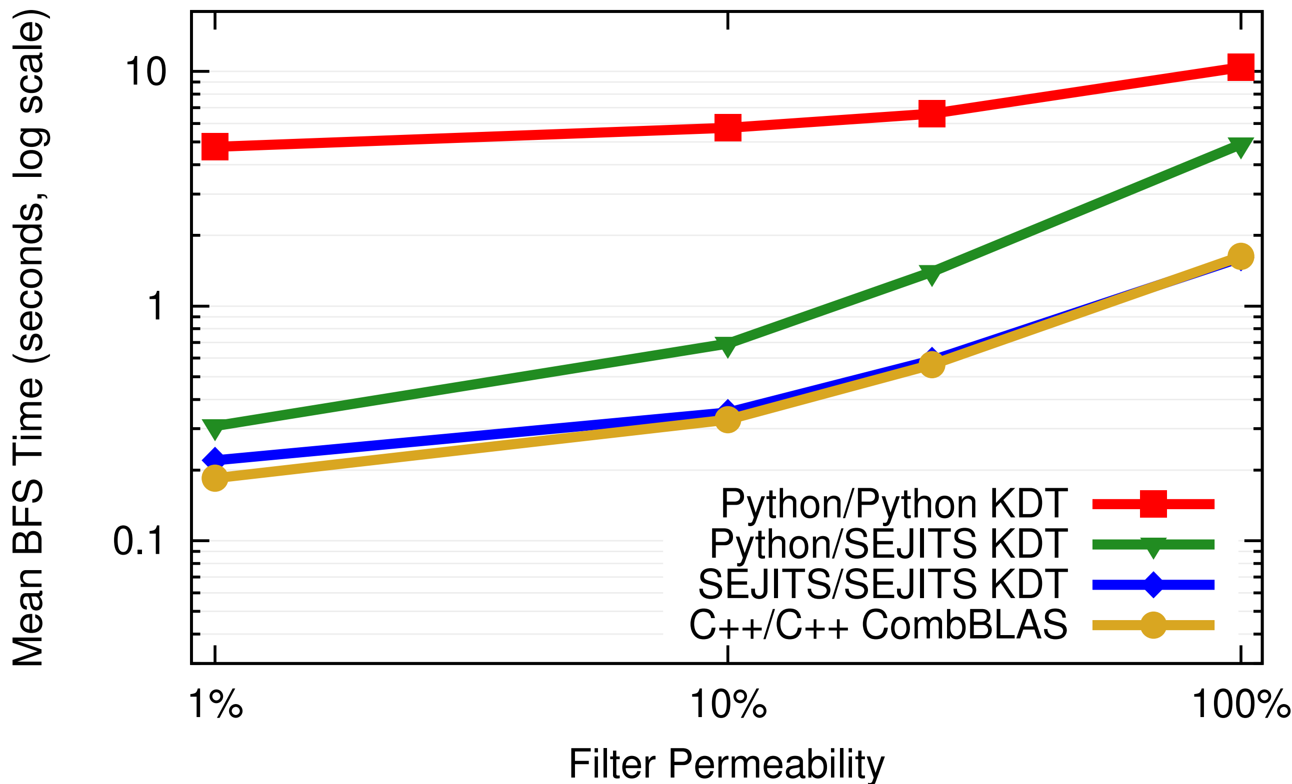
SEJITS+KDT multicore performance



- MIS= Maximal Independent Set
- 36 cores of Mirasol (Intel Xeon E7-8870)
- Erdős-Rényi (Scale 22, edgefactor=4)

Synthetic data with weighted randomness to match filter permeability
Notation: [semiring impl] / [filter impl]

SEJITS+KDT cluster performance



- Breadth-first search
- 576 cores of Hopper (Cray XE6 at NERSC with AMD Opterons)
- R-MAT (Scale 25, edgefactor=16, symmetric)

A ***roofline model*** for shows how SEJITS moves KDT analytics from being Python ***compute bound*** to being ***bandwidth bound***.

Part 3: Minimal Communication

Crux: Communication-avoiding versions of higher-level primitives

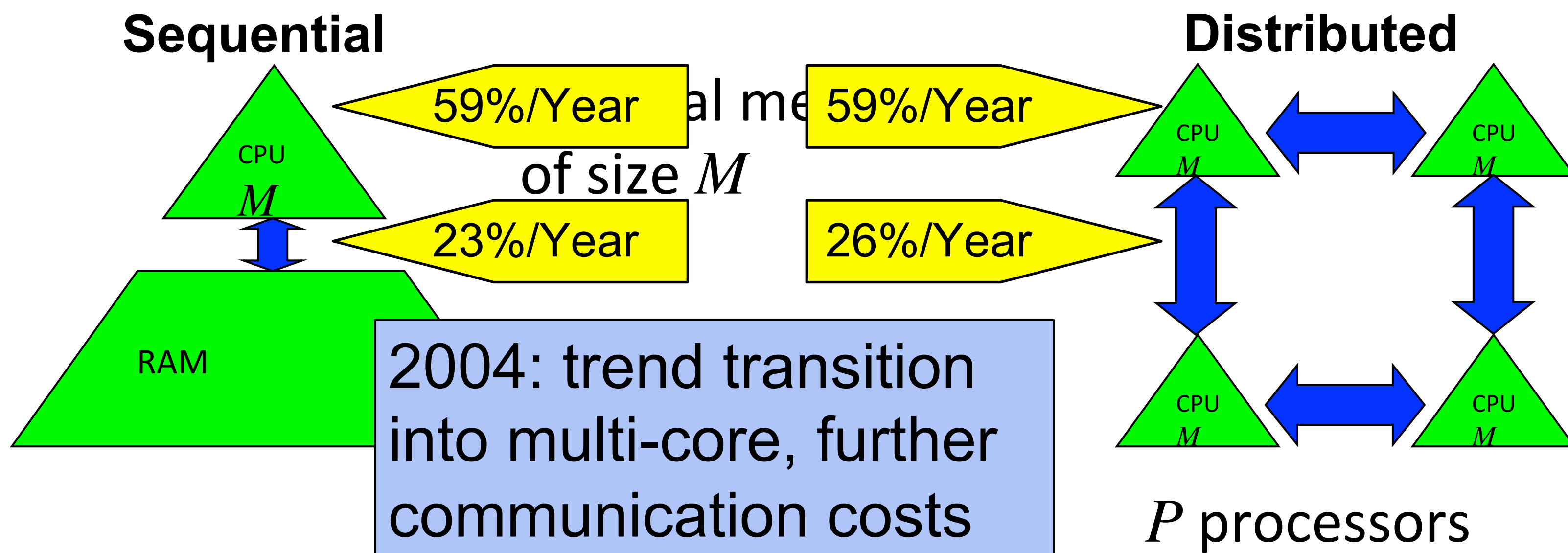
Cost Models for Time and Energy

Two kinds of costs:

- Arithmetic (FLOPs)
- Communication: moving data

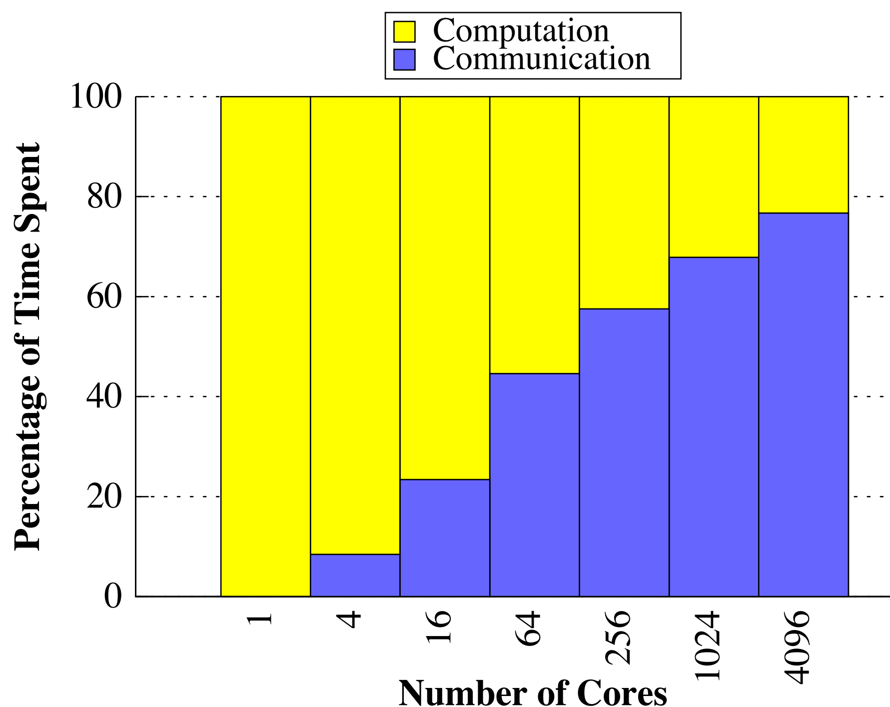
Develop faster algorithms: minimize communication (to lower bound if possible)

$$Running\ time = \gamma \cdot \#FLOPs + \beta \cdot \#Words + (\alpha \cdot \#Messages)$$



Communication crucial for graphs

- Often no surface to volume ratio.
- Very little data reuse in existing algorithmic formulations *
- Already heavily communication bound



2D sparse matrix-matrix multiply emulating:

- Graph contraction
- AMG restriction operations

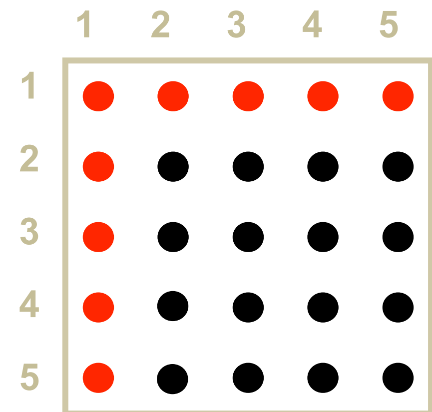
Scale 23 R-MAT (scale-free graph) **times** order 4 restriction operator

Cray XT4, Franklin, NERSC

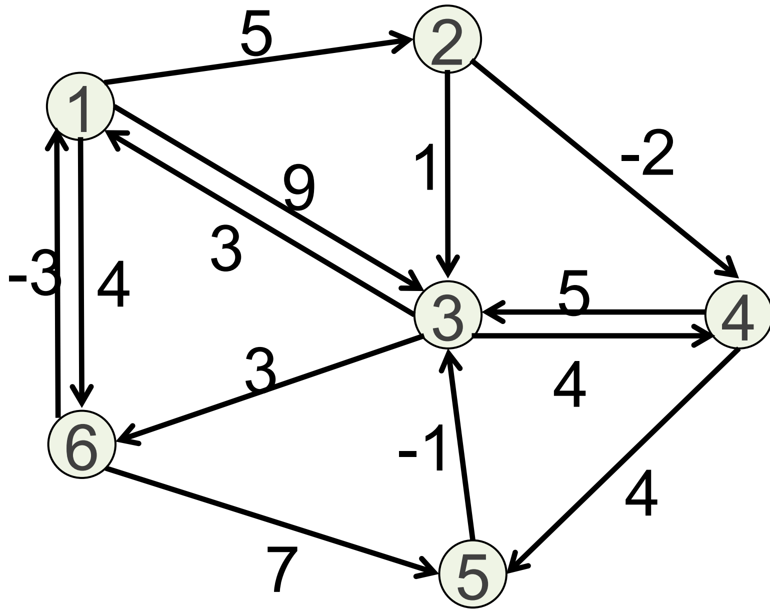
All-pairs shortest-paths problem

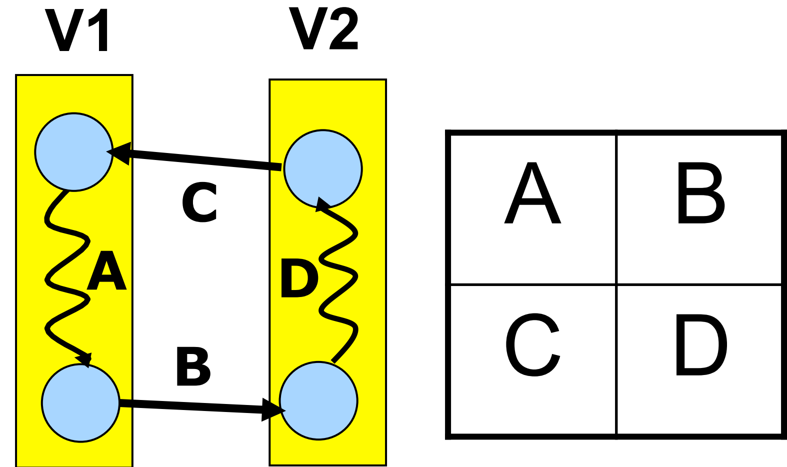
- <u>Input:</u> Directed graph with “costs” on edges
- Find least-cost paths between all reachable vertex pairs
- Classical algorithm: Floyd-Warshall

```
for k=1:n    // the induction sequence
    for i = 1:n
        for j = 1:n
            if(w(i→k)+w(k→j) < w(i→j))
                w(i→j):= w(i→k) + w(k→j)
```



- It turns out a previously overlooked **recursive version** is more parallelizable than the triple nested loop

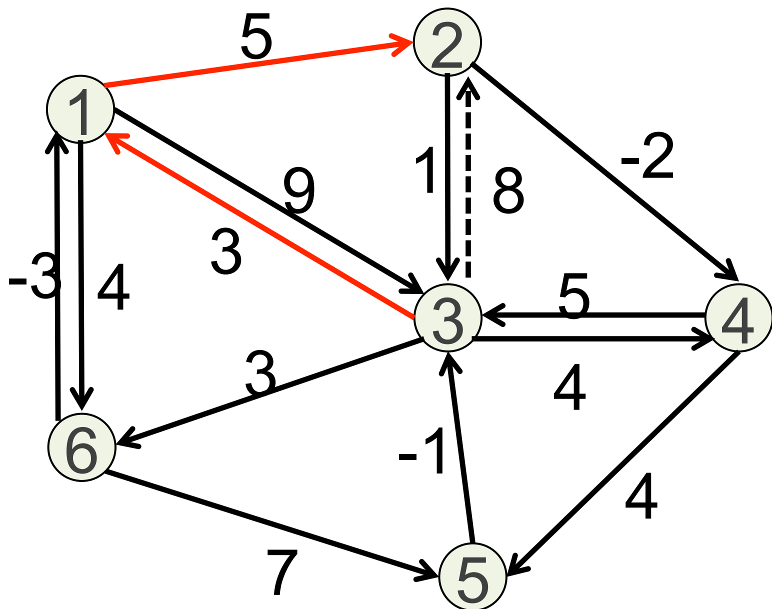




$$\left[\begin{array}{c|cc|ccc} 0 & 5 & 9 & \infty & \infty & 4 \\ \hline \infty & 0 & 1 & -2 & \infty & \infty \\ 3 & \infty & 0 & 4 & \infty & 3 \\ \hline \infty & \infty & 5 & 0 & 4 & \infty \\ \infty & \infty & -1 & \infty & 0 & \infty \\ -3 & \infty & \infty & \infty & 7 & 0 \end{array}\right]$$

+ is "min", **×** is "add"

```
A = A*;        % recursive call
B = AB;  C = CA;
D = D + CB;
D = D*;        % recursive call
B = BD;  C = DC;
A = A + BC;
```



$$\begin{bmatrix} \infty \\ 3 \end{bmatrix} \begin{bmatrix} 5 & 9 \end{bmatrix} = \begin{bmatrix} \infty & \infty \\ 8 & 12 \end{bmatrix}$$

C **B**

The cost of 3-1-2 path

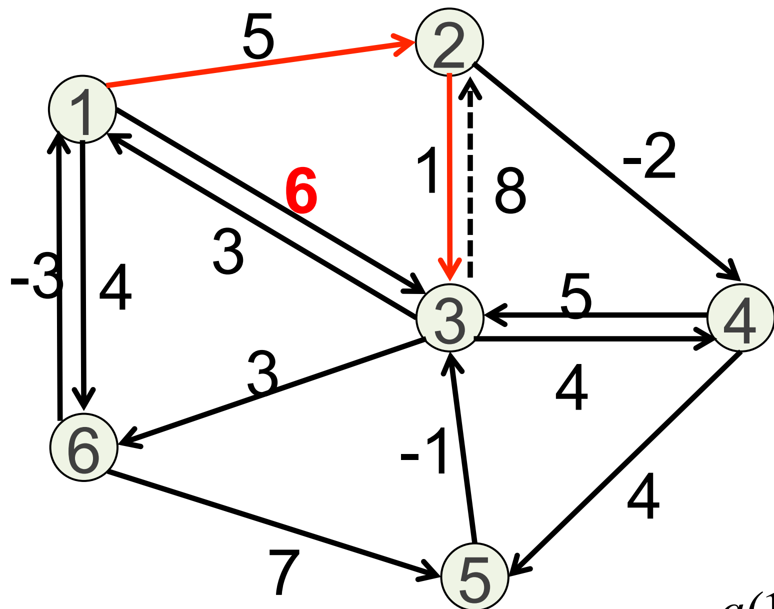
$$a(3,2) = a(3,1) + a(1,2) \xrightarrow{then} \Pi(3,2) = \Pi(1,2)$$

$$\begin{bmatrix} 0 & 5 & 9 & \infty & \infty & 4 \\ \infty & 0 & 1 & -2 & \infty & \infty \\ 3 & \mathbf{8} & 0 & 4 & \infty & 3 \\ \infty & \infty & 5 & 0 & 4 & \infty \\ \infty & \infty & -1 & \infty & 0 & \infty \\ -3 & \infty & \infty & \infty & 7 & 0 \end{bmatrix}$$

Distances

$$\Pi = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & \mathbf{1} & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 \end{bmatrix}$$

Parents



D = D*: no change

$$\begin{bmatrix} 5 & 9 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 8 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 6 \end{bmatrix}$$

B **D**

Path: 1-2-3

$$a(1,3) = a(1,2) + a(2,3) \xrightarrow{then} \Pi(1,3) = \Pi(2,3)$$

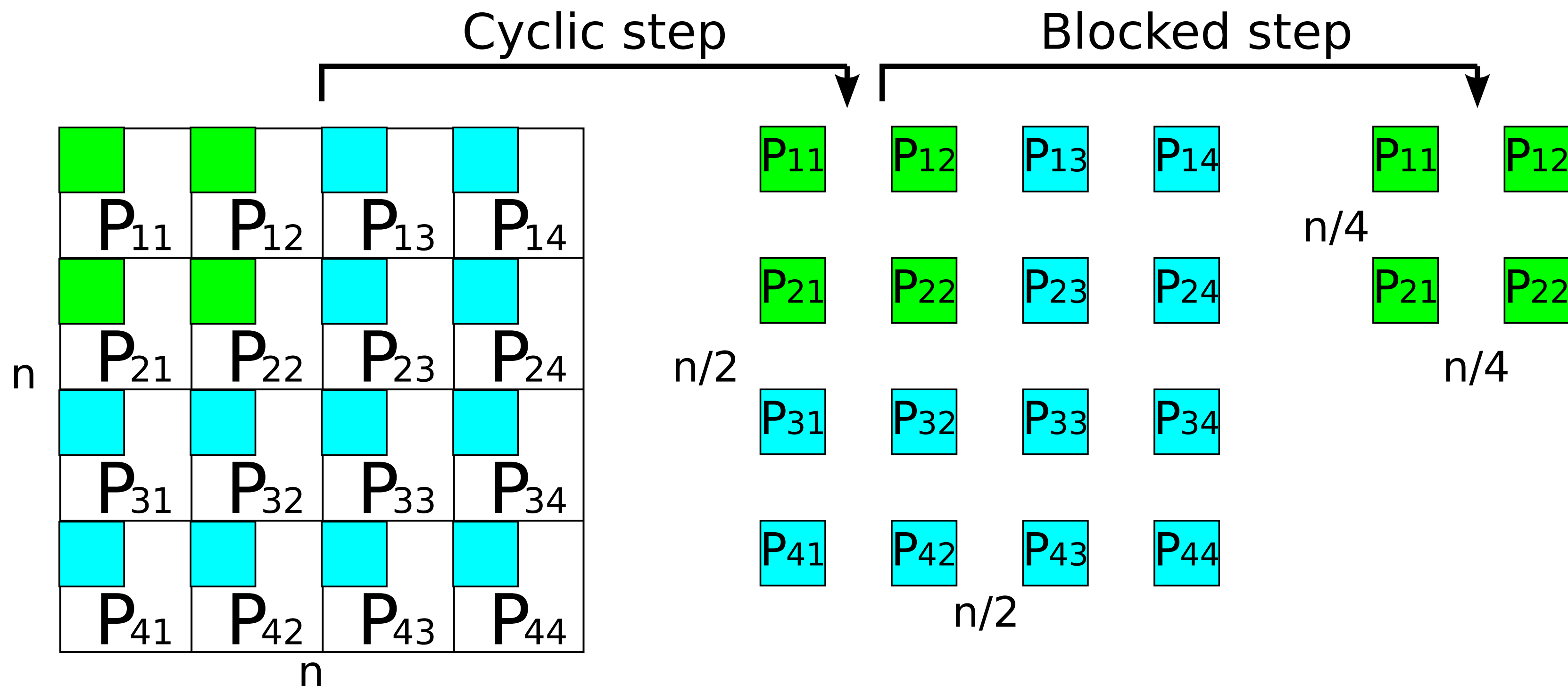
$$\begin{bmatrix} 0 & 5 & \mathbf{6} & \infty & \infty & 4 \\ \infty & 0 & 1 & -2 & \infty & \infty \\ 3 & 8 & 0 & 4 & \infty & 3 \\ \infty & \infty & 5 & 0 & 4 & \infty \\ \infty & \infty & -1 & \infty & 0 & \infty \\ -3 & \infty & \infty & \infty & 7 & 0 \end{bmatrix}$$

Distances

$$\Pi = \begin{bmatrix} 1 & 1 & \mathbf{2} & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 \end{bmatrix}$$

Parents

Communication-avoiding APSP on distributed memory



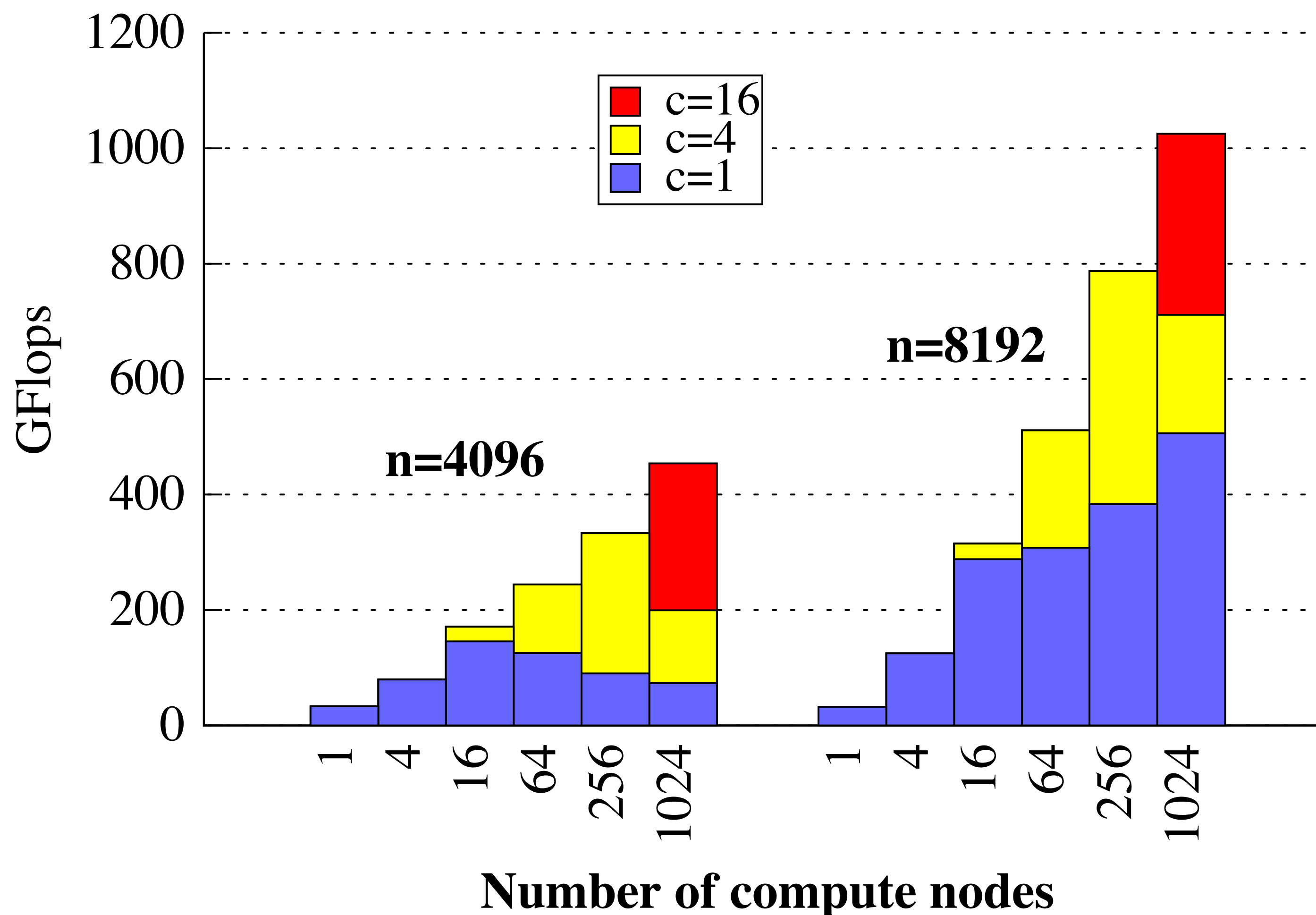
Bandwidth: $W_{\text{bc-2.5D}}(n,p) = O(n^2/\sqrt{cp})$

Latency: $S_{\text{bc-2.5D}}(p) = O\left(\sqrt{cp}\log^2(p)\right)$

c: number of replicas

Optimal for any memory size !

Communication-avoiding APSP on distributed memory

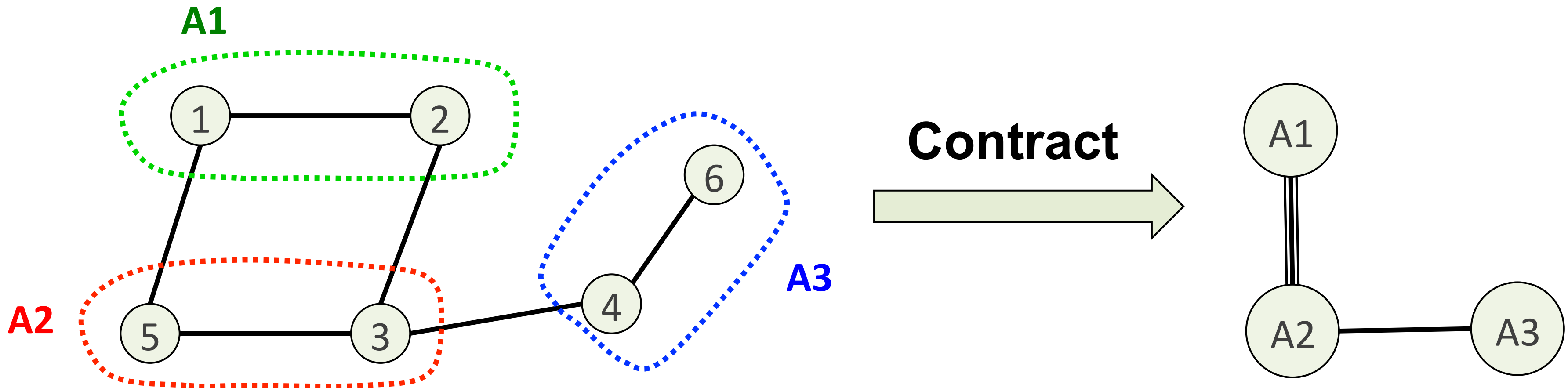


Solomonik, **B.**, and J. Demmel. “Minimizing communication in all-pairs shortest paths”, Proceedings of the IPDPS. 2013.

Questions?

Large-scale
bounds graph Communication
dense Inherent Betweenness multiplication
centrality matrix-matrix poor BFS Direction-optimizing
algebra all attaining algebraic algorithm Input
language Floyd-Warshall quantum contraction Semiring GEMM pairs Sparsity-independent
reachable assigning multiply lower Classical distributed Communication-avoiding
energy graphs number non-zero abstraction
less compare sparse between Find Allows communication-optimal
structure reduces cycle data extensive glues
paths edges parallel bandwidth detection bottom-up primitives
replicas enable holds
majority search New Directed standard matrix discovered
extraction chemistry shortest-paths Significant saves all-pairs
least-cost clustering computations scalable Cij match gap
processors Breadth-first independent memory
costs reuse RMAT linear Erdos-Renyi algorithms
multigrid subgraph particular
approach APSP computation sparsity grid-points

Graph contraction via sparse triple product



	1	2	3	4	5	6
1	1	1				
2			1		1	
3				1		1

x

	1	2	3	4	5	6
1		●			●	
2	●		●			
3		●		●	●	
4			●			●
5	●		●			
6				●		

x

1		
1		
	1	
		1
	1	
		1

=

	●	
●		●
	●	