UCSB

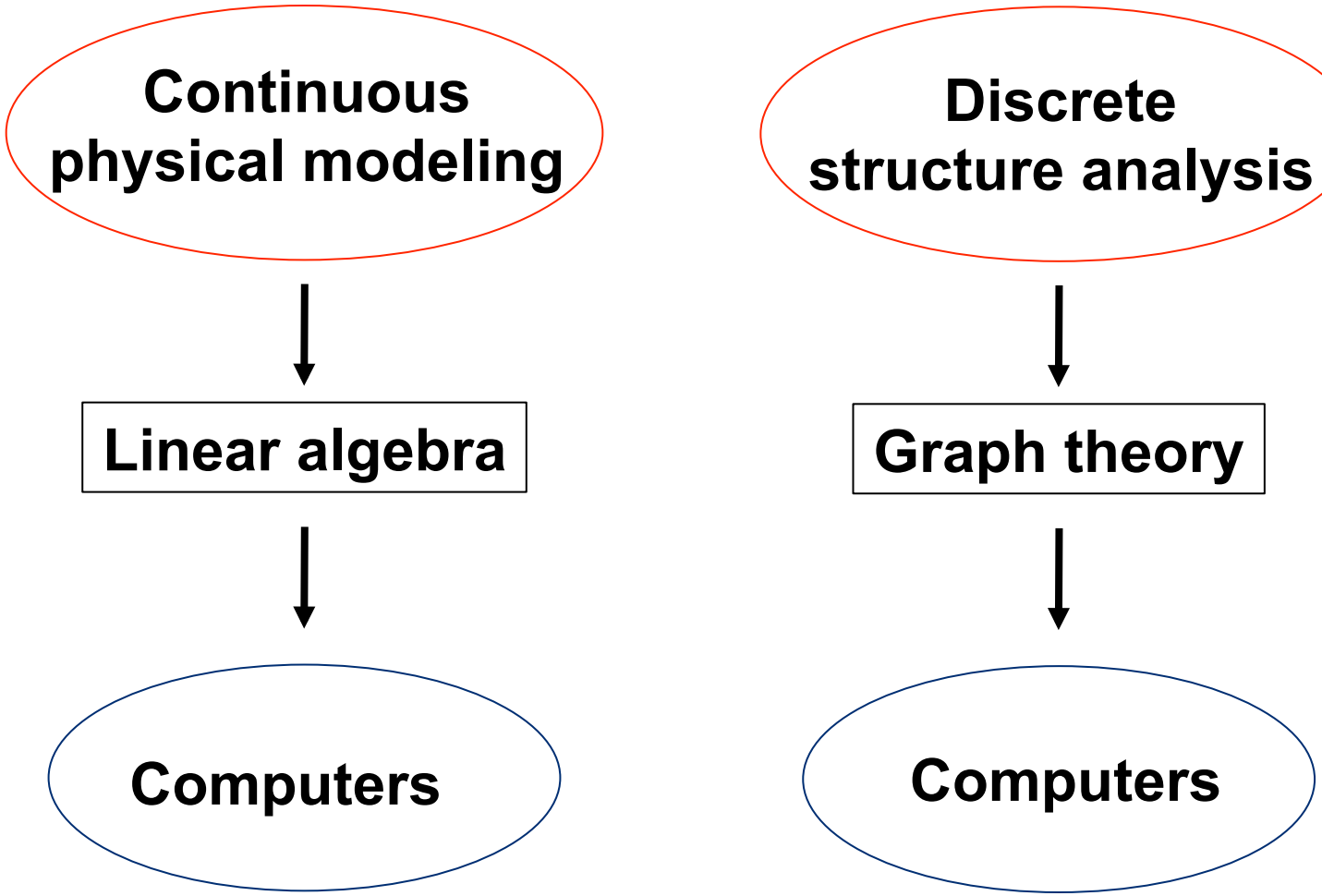
Graph Algorithms in the Language of Linear Algebra

John R. Gilbert
University of California, Santa Barbara

CS 240A presentation adapted from:
Intel Non-Numeric Computing Workshop
January 17, 2014

Support: Intel, Microsoft, DOE Office of Science, NSF

The middleware of scientific computing



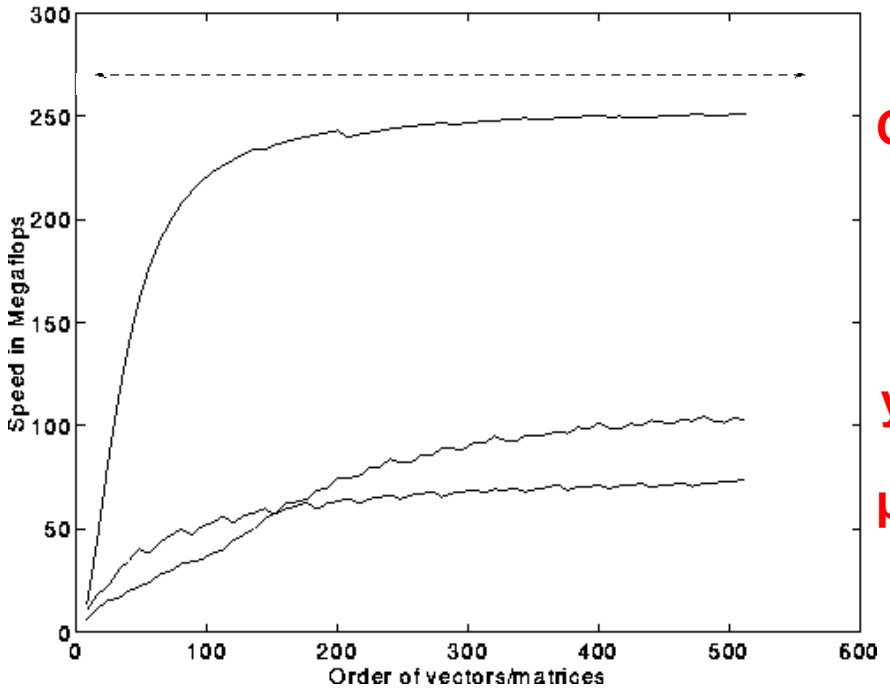
UCSB

The challenge of the software stack

- By analogy to numerical scientific computing. . .

- What should the combinatorial BLAS look like?

Basic Linear Algebra Subroutines (BLAS): Ops/Sec vs. Matrix Size



C = A*B

y = A*x

μ = x^T y

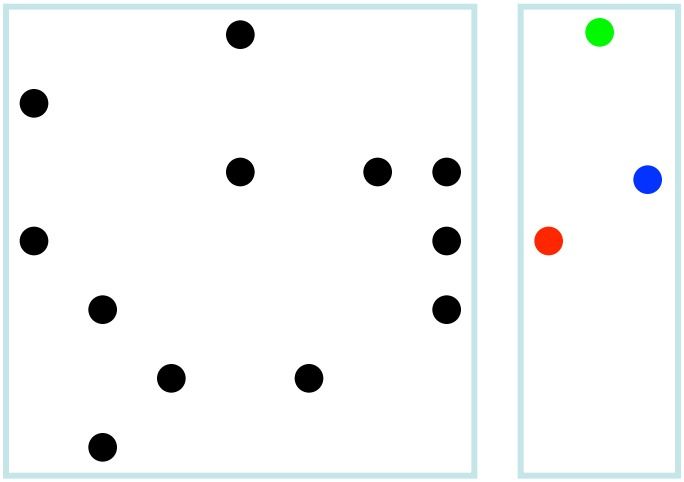
UCSB

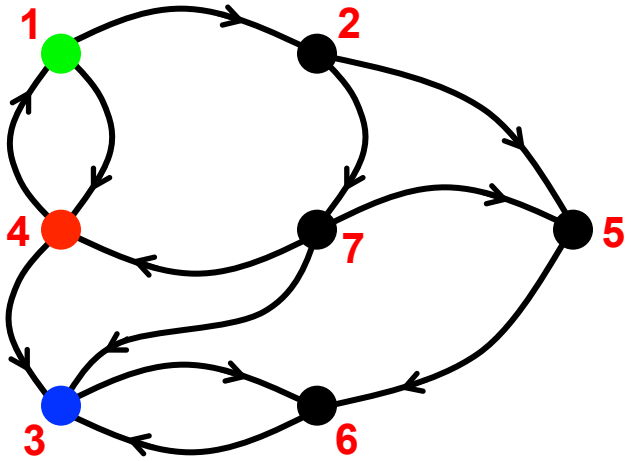
Outline

- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Standards for graph algorithm primitives

UCSB

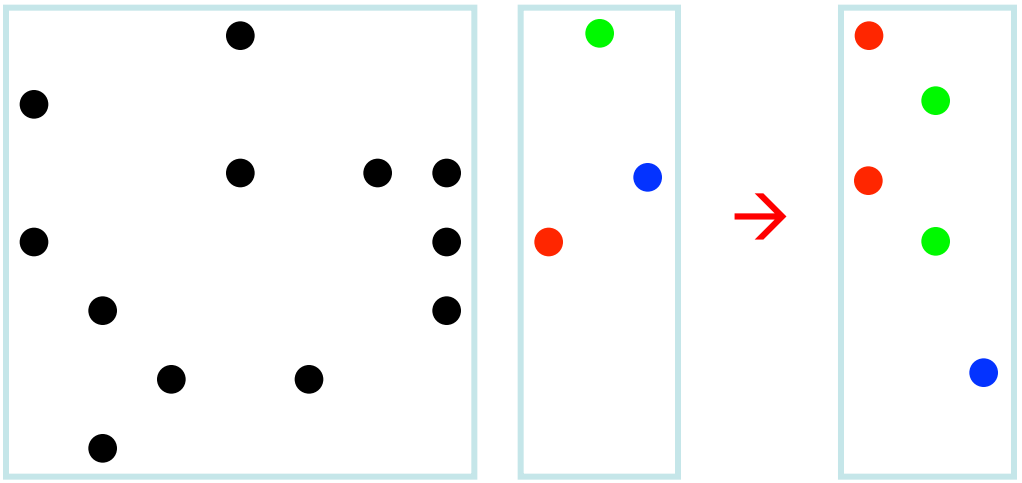
Multiple-source breadth-first search

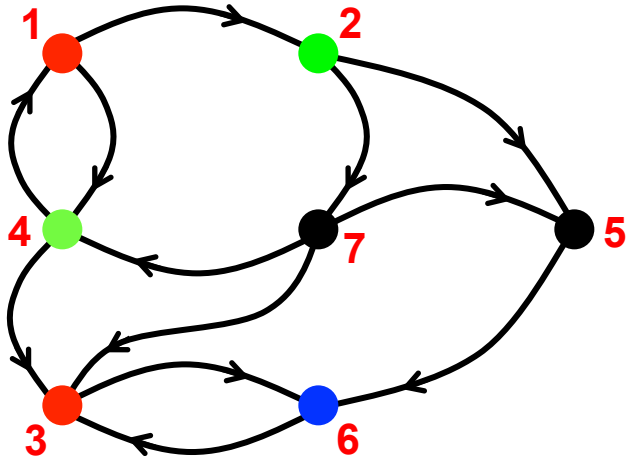




UCSB

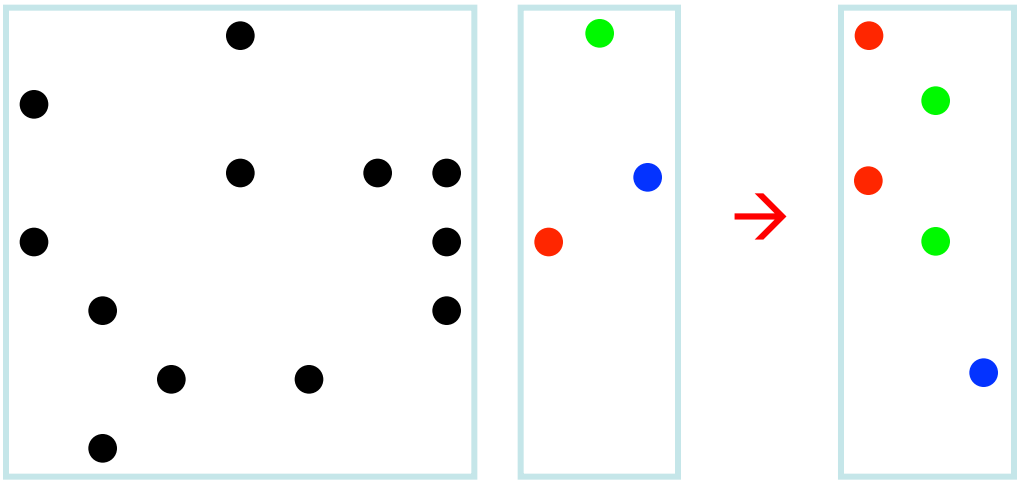
Multiple-source breadth-first search

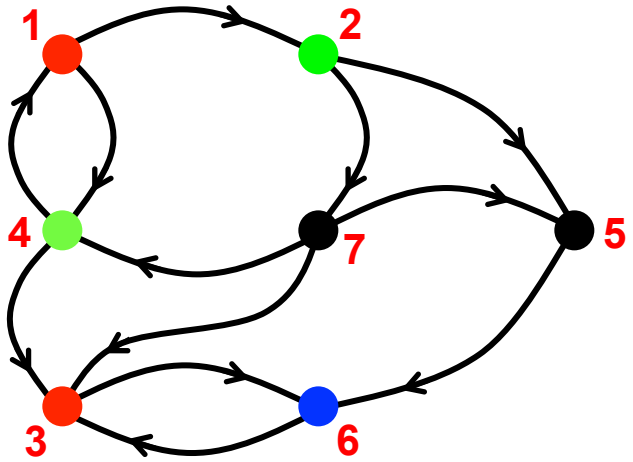




UCSB

Multiple-source breadth-first search

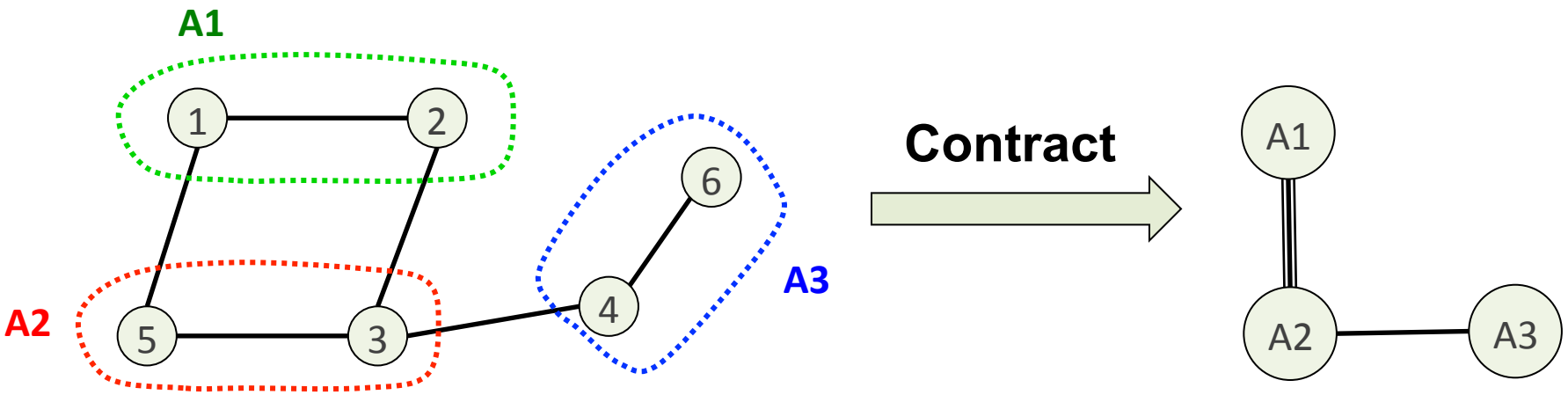


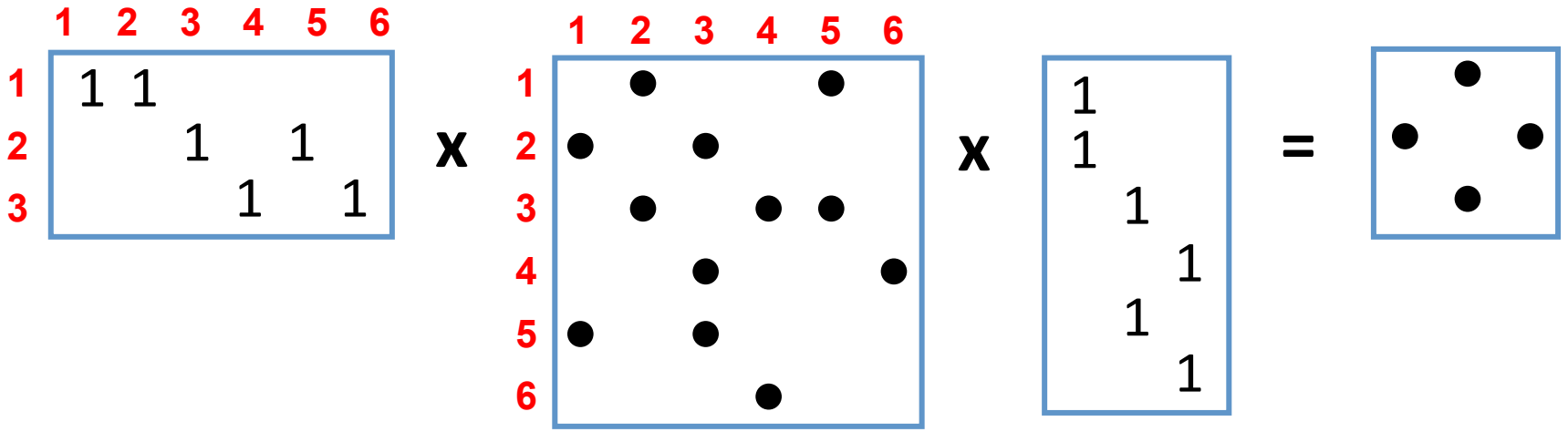


- Sparse array representation => space efficient
- Sparse matrix-matrix multiplication => work efficient
- Three possible levels of parallelism: searches, vertices, edges

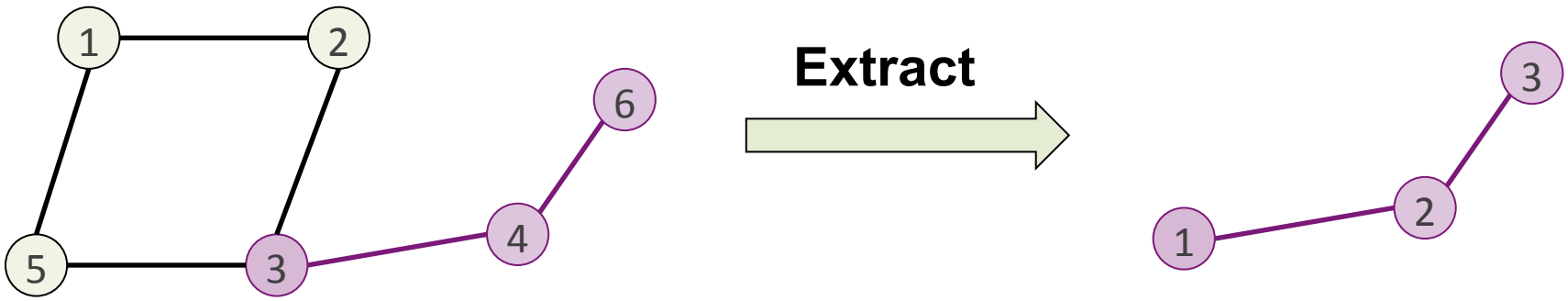
UCSB

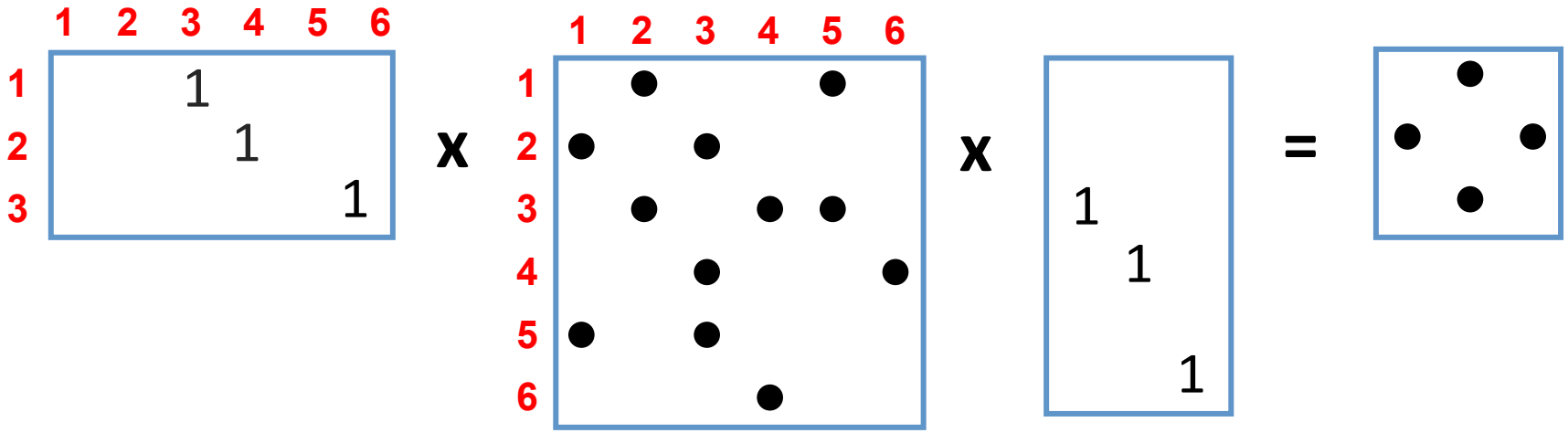
Graph contraction via sparse triple product



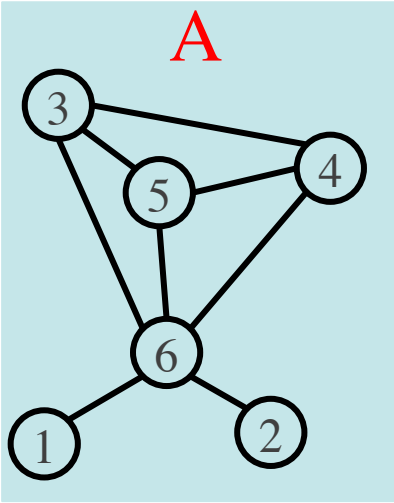


Subgraph extraction via sparse triple product





Counting triangles (clustering coefficient)

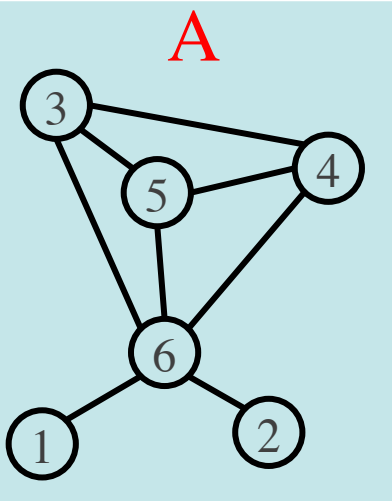


Clustering coefficient:

- Pr (wedge i-j-k makes a triangle with edge i-k)
- 3 * # triangles / # wedges
- 3 * **4** / **19** = **0.63** in example
- may want to compute for each vertex j

UCSB

Counting triangles (clustering coefficient)

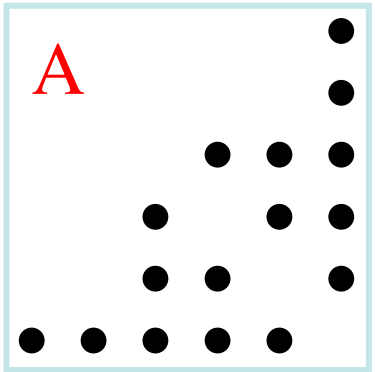


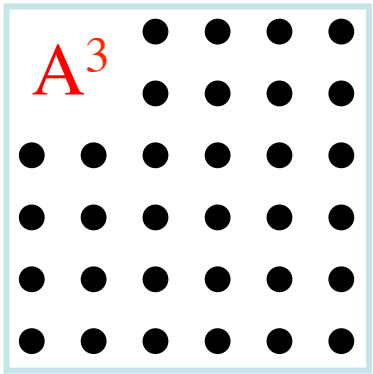
Clustering coefficient:

- Pr (wedge i-j-k makes a triangle with edge i-k)
- 3 * # triangles / # wedges
- 3 * **4** / **19** = **0.63** in example
- may want to compute for each vertex j

Inefficient way to count triangles with matrices:

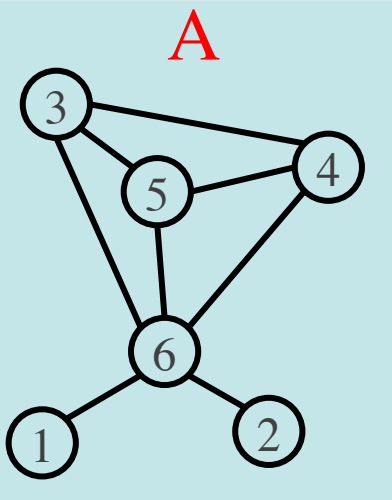
- A = adjacency matrix
- # triangles = trace(A^3) / 6
- but A^3 is likely to be pretty dense





UCSB

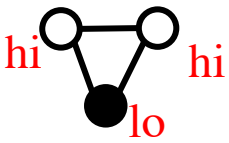
Counting triangles (clustering coefficient)



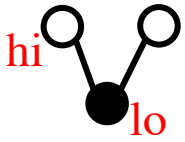
Clustering coefficient:

- Pr (wedge i-j-k makes a triangle with edge i-k)
- 3 * # triangles / # wedges
- 3 * **4** / **19** = **0.63** in example
- may want to compute for each vertex j

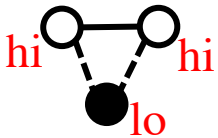
Cohen's algorithm to count triangles:



- Count triangles by lowest-degree vertex.



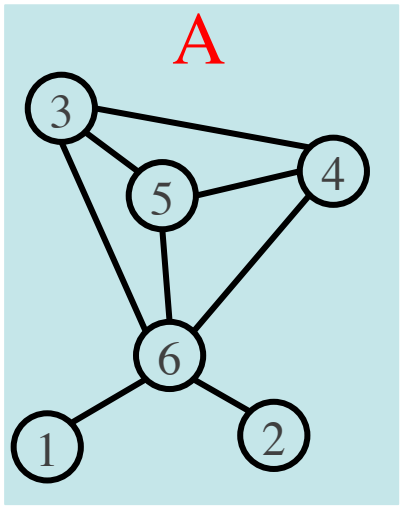
- Enumerate "low-hinged" wedges.



- Keep wedges that close.

UCSB

Counting triangles (clustering coefficient)

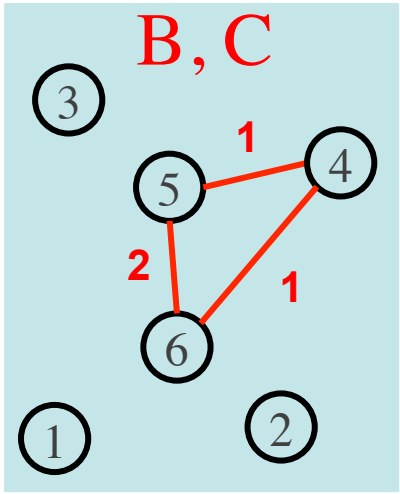


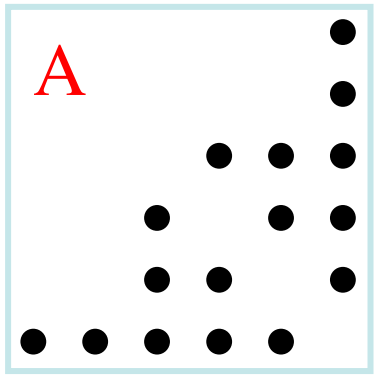
$A = L + U$ (hi->lo + lo->hi)

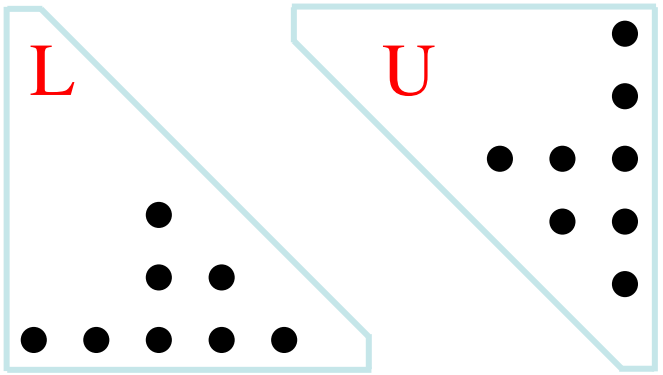
$L \times U = B$ (wedge, low hinge)

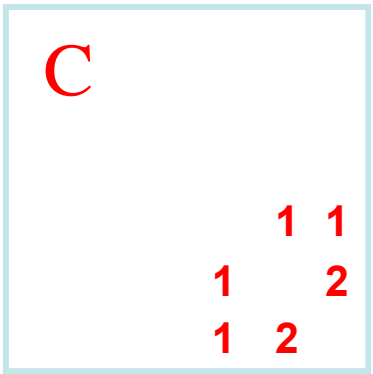
$A \wedge B = C$ (closed wedge)

$sum(C)/2 =$ **4 triangles**









UCSB

Betweenness centrality *[Robinson 2008]*

$b = \text{BETWEENNESSCENTRALITY}(G = A : \mathbb{B}^{N_V \times N_V})$

```
1  b = 0
2  for 1 ≤ r ≤ N_V
3        do
4           d = 0
5           S = 0
6           p = 0, p_r = 1
7           f = a_{r,:}
8           while f ≠ 0
9                 do
10                   d = d + 1
11                   p = p + f
12                   s_{d,:} = f
13                   f = fA × ¬p
14          while d ≥ 2
15                do
16                   w = s_{d,:} × (1 + u) ÷ p
17                   w = Aw
18                   w = w × s_{d−1,:} × p
19                   u = u + w
20                   d = d − 1
21          b = b + u
```

Variables:

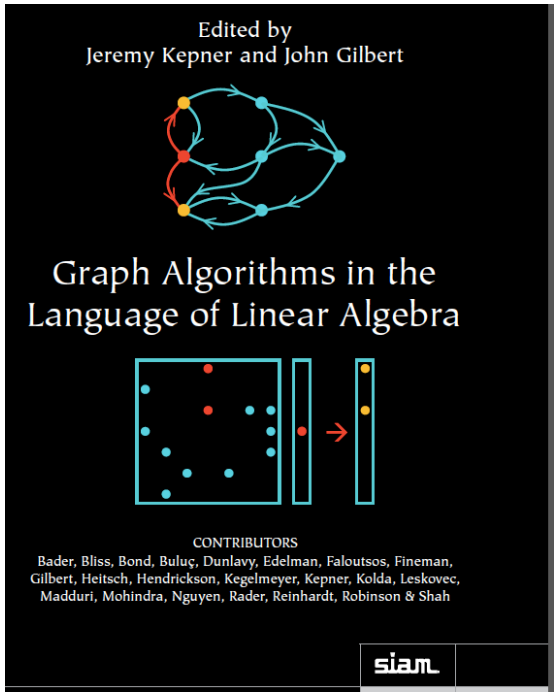
			Storage:
A :	sparse adjacency matrix	$B^{S(NxN)}$	O(M+N)
f :	sparse fringe vector	$Z^{S(N)}$	O(N)
p :	shortest path vector	Z^{N}	O(N)
S :	sparse depth matrix	$B^{S(NxN)}$	O(N)
u :	centrality update vector	R^{N}	O(N)

Storage: O(M+N)

Time: $O(MN + N^2)$

Graph algorithms in the language of linear algebra

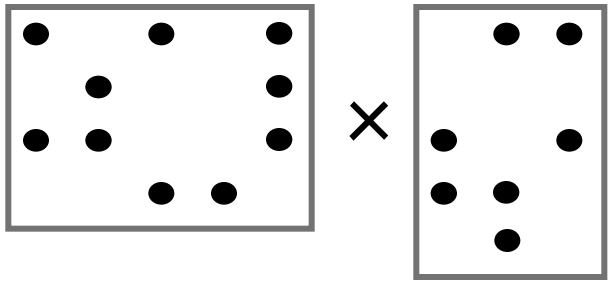
- Kepner et al. study [2006]: fundamental graph algorithms including min spanning tree, shortest paths, independent set, max flow, clustering, …
- SSCA#2 / centrality [2008]
- Basic breadth-first search / Graph500 [2010]
- Beamer et al. [2013] direction-optimizing breadth-first search, implemented in CombBLAS



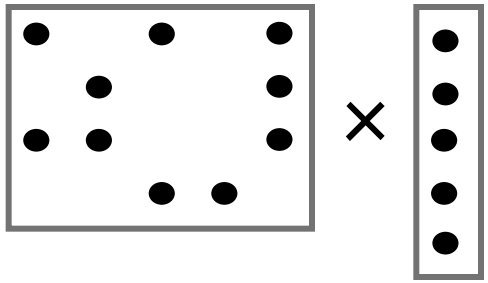
UCSB

Sparse array-based primitives

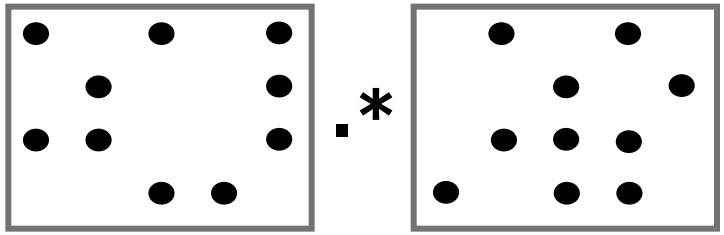
Sparse matrix-matrix multiplication (SpGEMM)



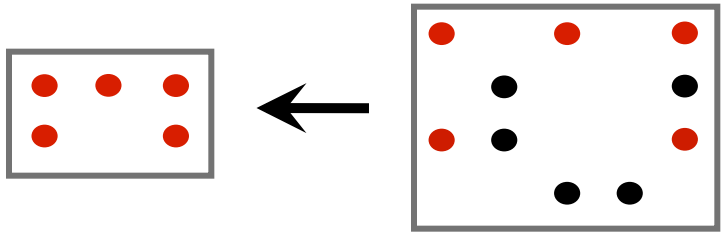
Sparse matrix-dense vector multiplication



Element-wise operations



Sparse matrix indexing



Matrices over various semirings: (+ . x), (min . +), (or . and), …

UCSB

The case for sparse matrix graph primitives

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

Matrices over semirings

- E.g. matrix multiplication **C = AB** (or matrix/vector):

$$C_{i,j} = A_{i,1} \times B_{1,j} + A_{i,2} \times B_{2,j} + \cdots + A_{i,n} \times B_{n,j}$$

- Replace scalar operations × and + by
  - ⊗ : associative, distributes over ⊕
  - ⊕ : associative, commutative
- Then $C_{i,j} = A_{i,1} \otimes B_{1,j} \oplus A_{i,2} \otimes B_{2,j} \oplus \cdots \oplus A_{i,n} \otimes B_{n,j}$
- Examples: ×.+ ; and.or ; +.min ; . . .
- *Same data reference pattern and control flow*

UCSB

Examples of semirings in graph algorithms

(R, +, x) Real Field	Standard numerical linear algebra
({0,1}, |, &) Boolean Semiring	Graph traversal
(R U {∞}, min, +) Tropical Semiring	Shortest paths
(R U {∞}, min, x)	Select subgraph, or contract nodes to form quotient graph
(edge/vertex attributes, vertex data aggregation, edge data processing)	Schema for user-specified computation at vertices and edges

Jon Berry challenge problems for GALA (all final project possibilities)

- Clustering coefficient (triangle counting)
- Connected components (bully algorithm)
- Maximum independent set (NP-hard)
- Maximal independent set (Luby algorithm)
- Single-source shortest paths
- Special betweenness (for subgraph isomorphism)

Fast Approximate Neighborhood Function (S. Vigna) (final project possibility)

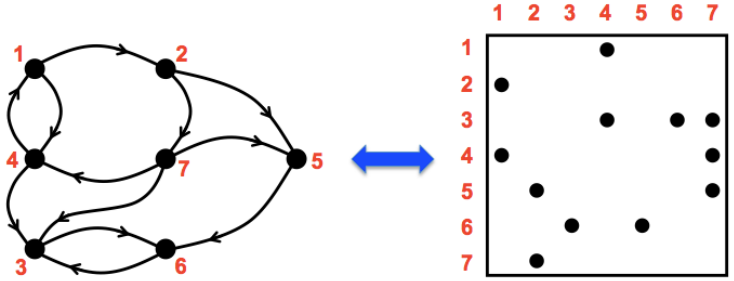
- Distribution of distances between vertices in a graph
  - For each vertex v, how many vertices at distance k?
  - What's the average distance (or closeness) between vertices?
- Expensive to compute exactly
- Nifty but simple data structure gives good approximation fast
- Could be implemented as sparse matrix times sparse vector, with the nifty data structure in the semiring
- E.g., using Combinatorial BLAS library
- Link to paper on course web site

Outline

- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Standards for graph algorithm primitives

Combinatorial BLAS

gauss.cs.ucsb.edu/~aydin/CombBLAS



An extensible distributed-memory library offering a small but powerful set of linear algebraic operations specifically targeting graph analytics.

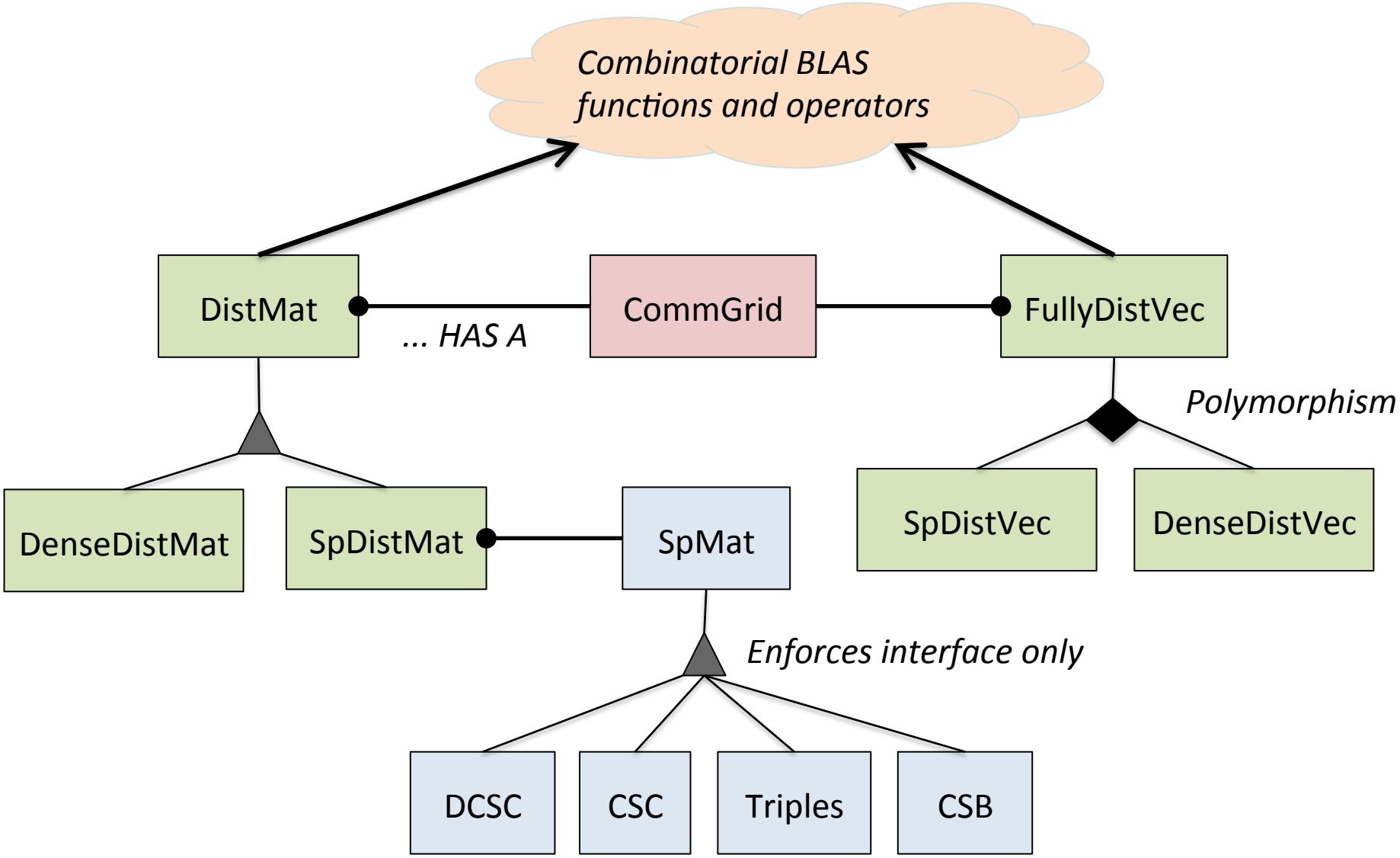
- Aimed at graph algorithm designers/programmers who are not expert in mapping algorithms to parallel hardware.
- Flexible templated C++ interface.
- Scalable performance from laptop to 100,000-processor HPC.

- Open source software.
- Version 1.4.0 released January 16, 2014.

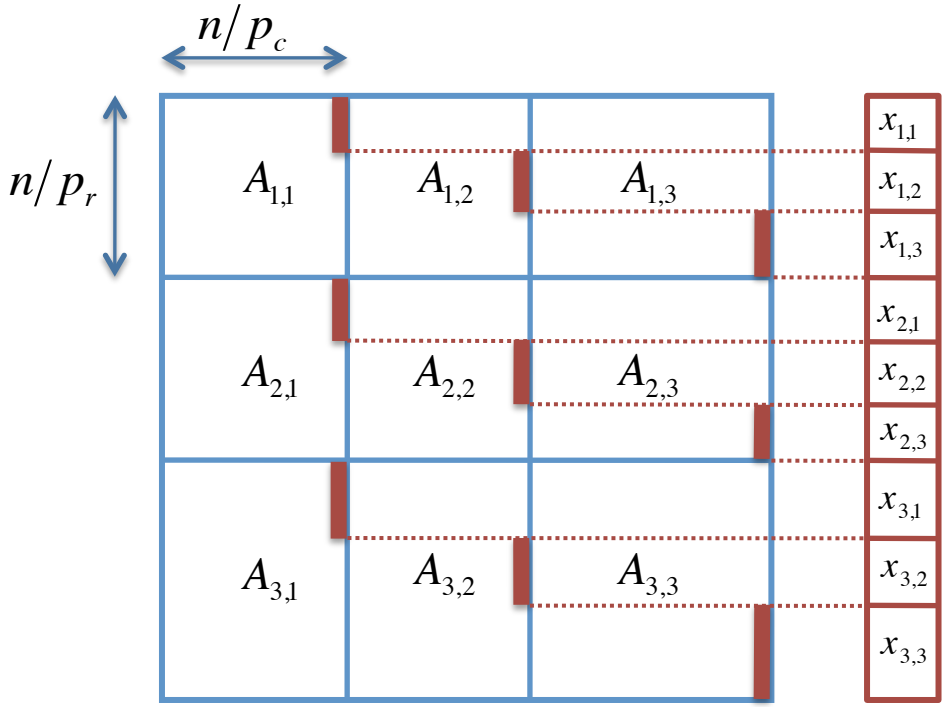
Combinatorial BLAS: Functions

Function	Applies to	Parameters		Returns	Matlab Phrasing
SpGEMM	Sparse Matrix (as friend)	**A**, **B**: trA: trB:	sparse matrices transpose **A** if true transpose **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
SpMV	Sparse Matrix (as friend)	**A**: **x**: trA:	sparse matrices sparse or dense vector(s) transpose **A** if true	Sparse or Dense Vector(s)	$\mathbf{y} = \mathbf{A} * \mathbf{x}$
SpEWiseX	Sparse Matrices (as friend)	**A**, **B**: notA: notB:	sparse matrices negate **A** if true negate **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
Reduce	Any Matrix (as method)	dim: *binop*:	dimension to reduce reduction operator	Dense Vector	sum(**A**)
SpRef	Sparse Matrix (as method)	**p**: **q**:	row indices vector column indices vector	Sparse Matrix	$\mathbf{B} = \mathbf{A}(\mathbf{p}, \mathbf{q})$
SpAsgn	Sparse Matrix (as method)	**p**: **q**: **B**:	row indices vector column indices vector matrix to assign	none	$\mathbf{A}(\mathbf{p}, \mathbf{q}) = \mathbf{B}$
Scale	Any Matrix (as method)	**rhs**:	any object (except a sparse matrix)	none	Check guiding principles 3 and 4
Scale	Any Vector (as method)	**rhs**:	any vector	none	none
Apply	Any Object (as method)	*unop*:	unary operator (applied to non-zeros)	None	none

Combinatorial BLAS: Distributed-memory reference implementation



2D layout for sparse matrices & vectors



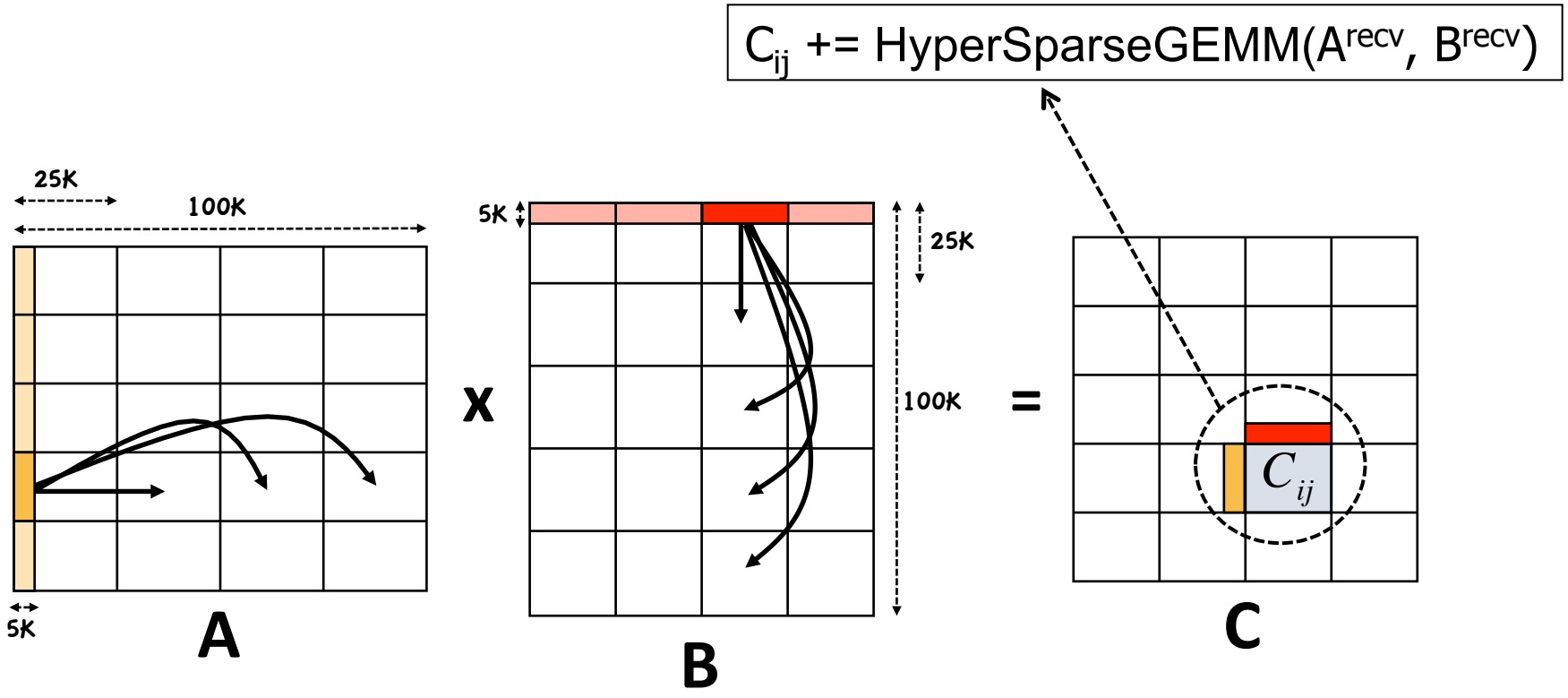
Matrix/vector distributions, interleaved on each other.

Default distribution in **Combinatorial BLAS**.

Scalable with increasing number of processes

- 2D matrix layout wins over 1D with large core counts and with limited bandwidth/compute
- 2D vector layout sometimes important for load balance

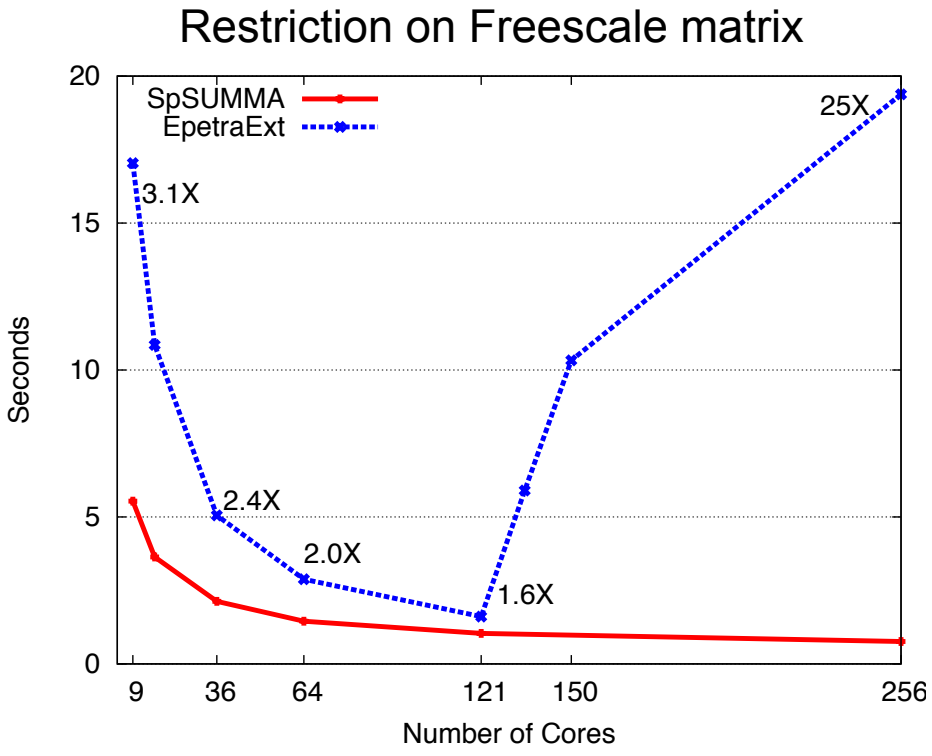
Parallel sparse matrix-matrix multiplication algorithm

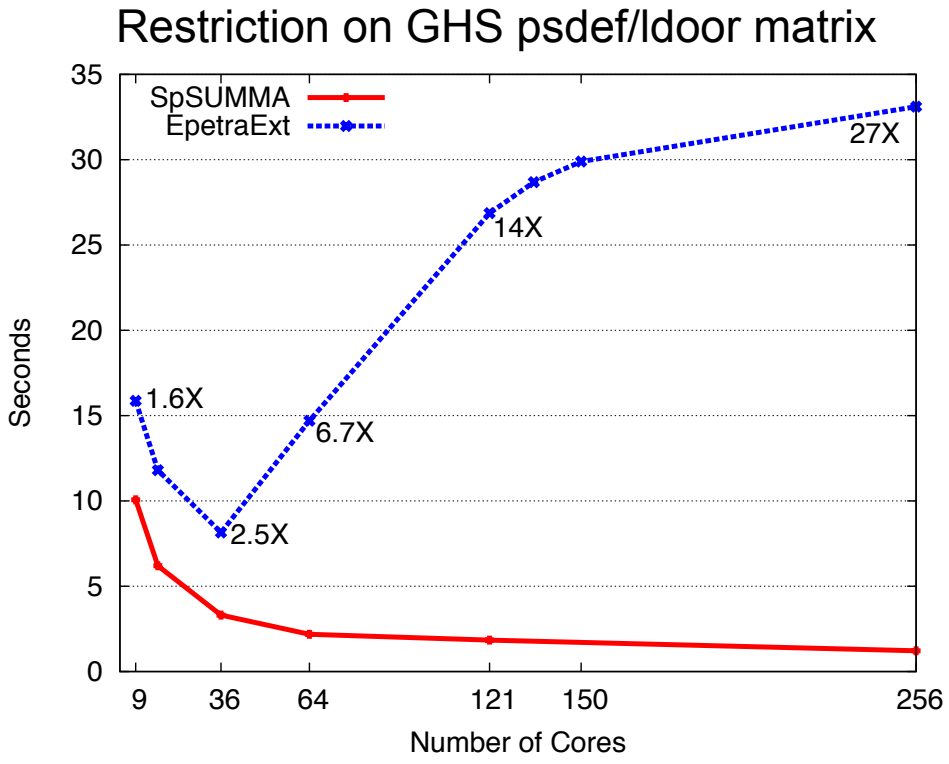


2D algorithm: Sparse SUMMA (based on dense SUMMA)

General implementation that handles rectangular matrices

1D vs. 2D scaling for sparse matrix-matrix multiplication



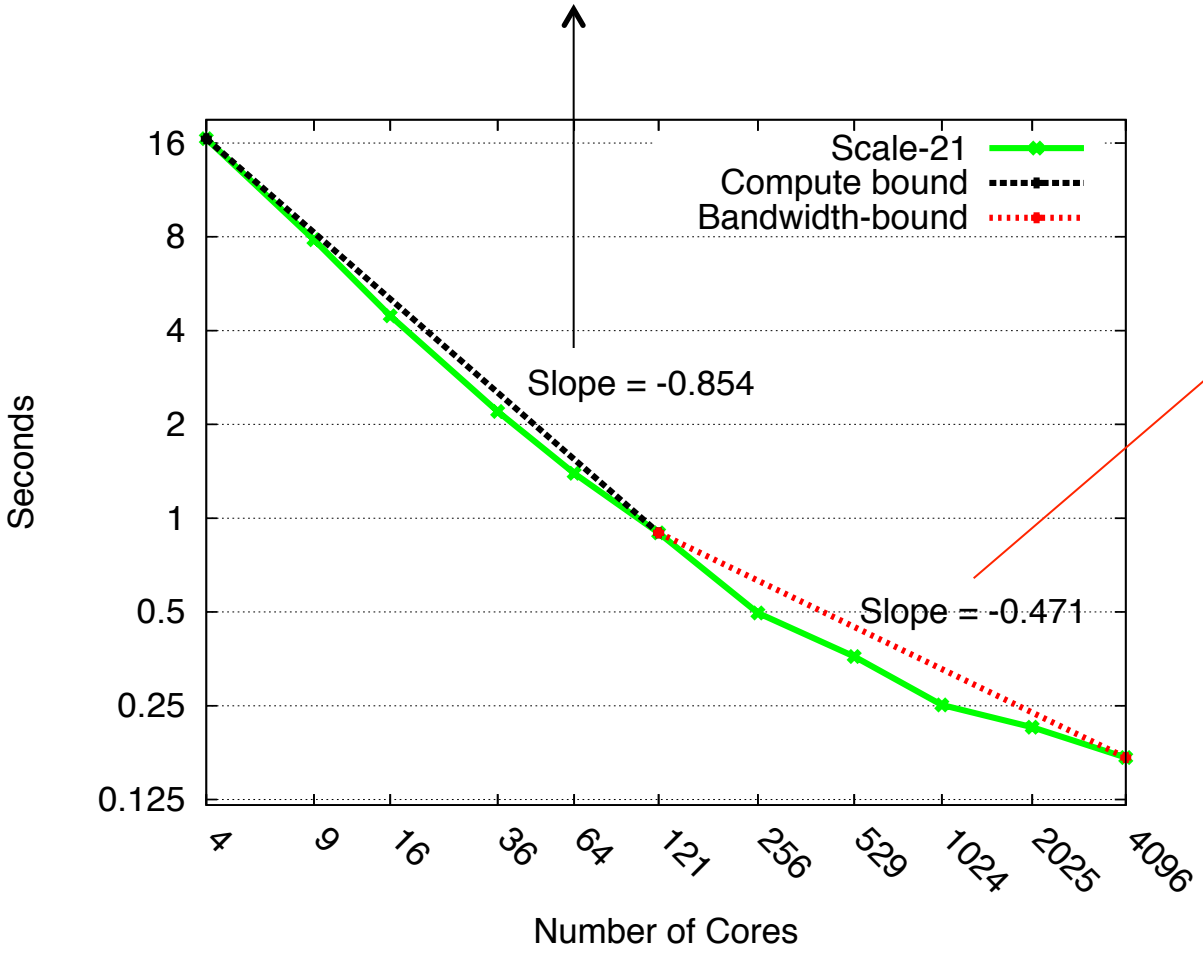


SpSUMMA = 2-D data layout (Combinatorial BLAS)

EpetraExt = 1-D data layout

Scaling to more processors…

Almost linear scaling until bandwidth costs starts to dominate

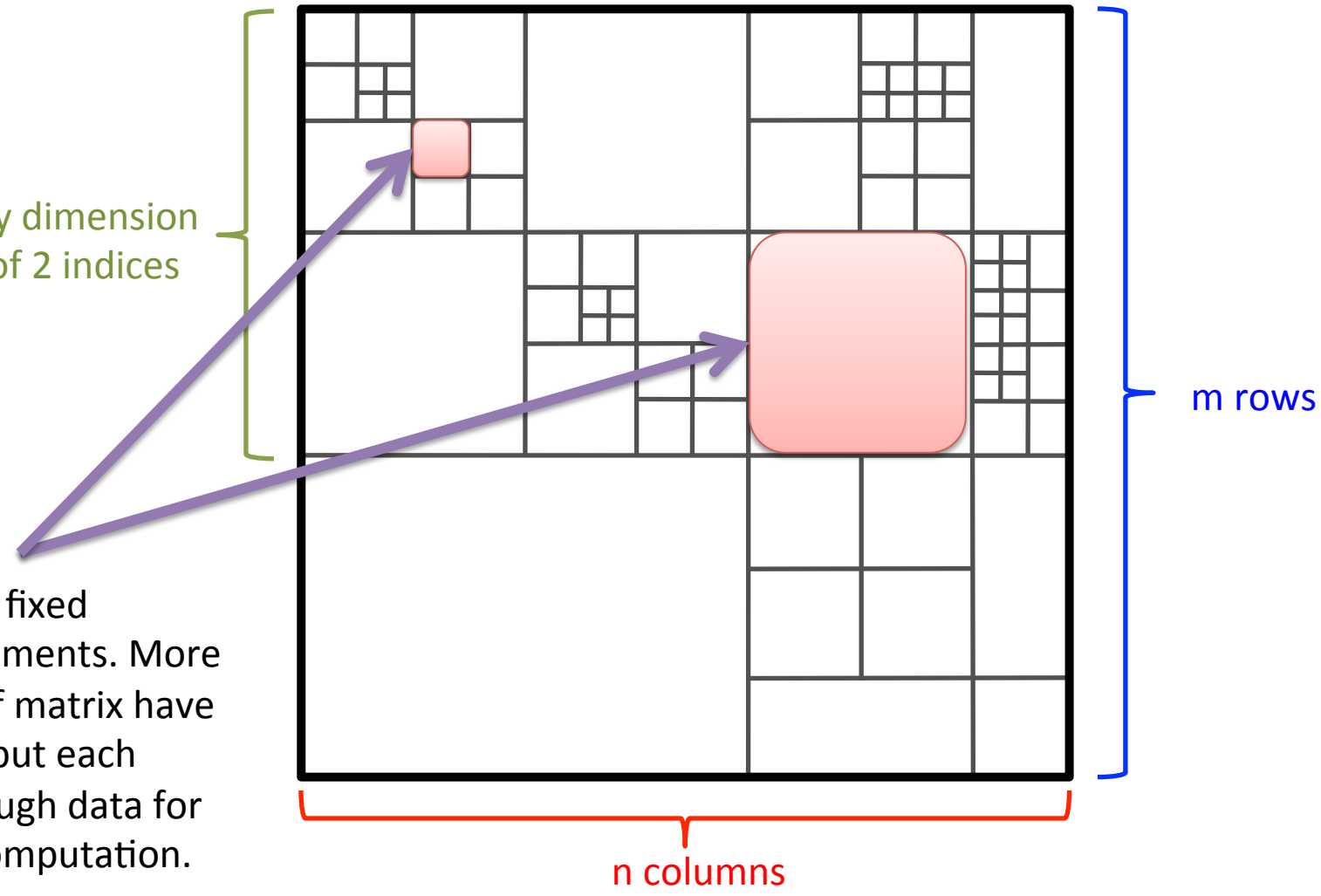


Scaling proportional to √p afterwards

$$T_{comp} = O(n)$$

$$T_{comm} = O(n\sqrt{p})$$

Work In Progress:
QuadMat Shared-memory data structure

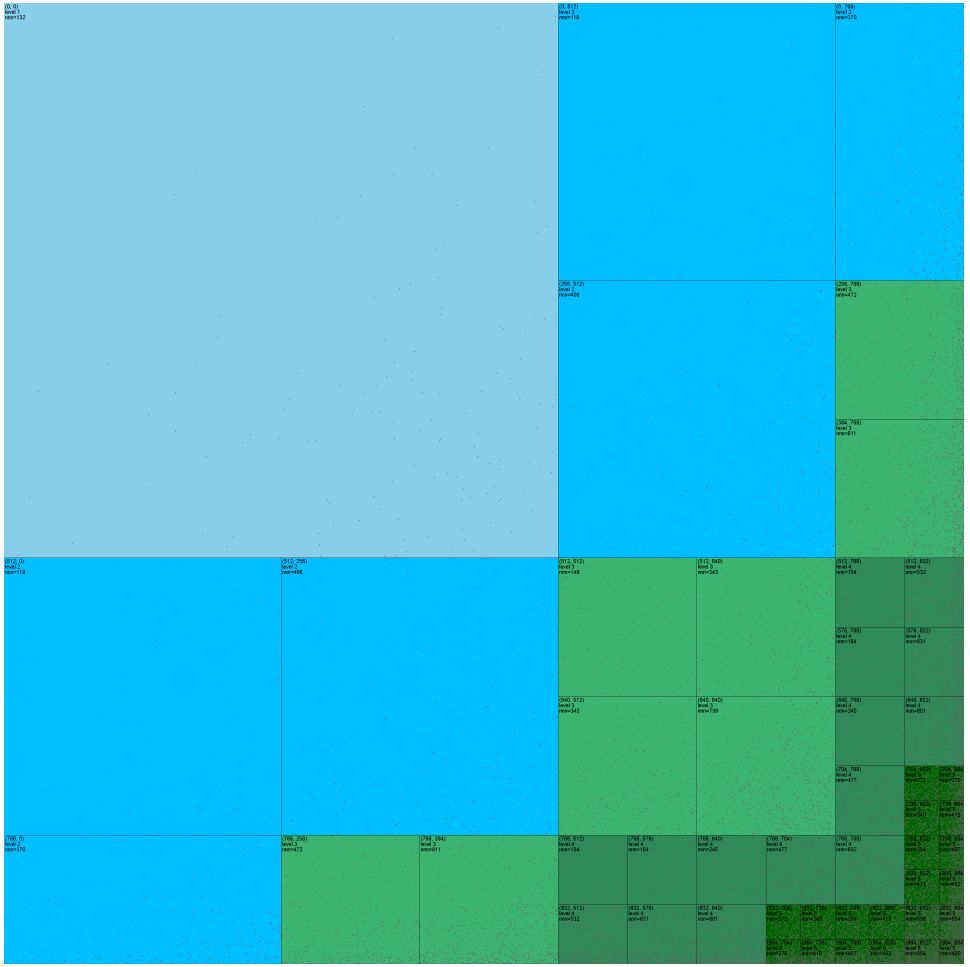


Blocks store a fixed number of elements. More dense parts of matrix have more blocks, but each block has enough data for meaningful computation.

Example

Scale 10 RMAT
(887x887, 21304 non-nulls)
up to 1024 non-nulls per block
Sorted by degree

Blue blocks: uint16_t indices
Green blocks: uint8_t indices
Each (tiny) dot is a non-null



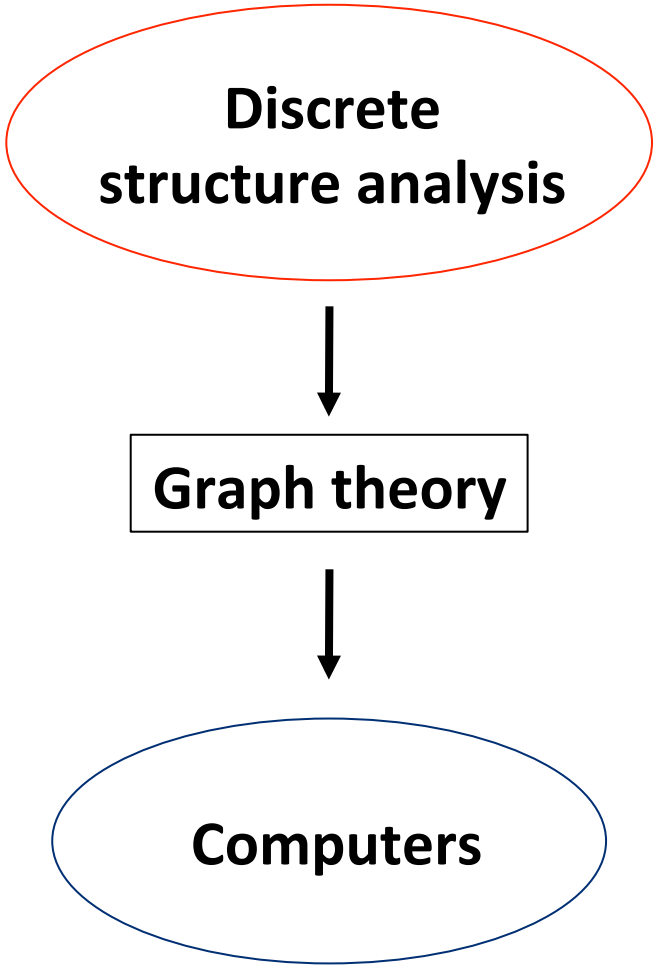
Implementation in templated C++

- Parallelized with TBB's task scheduler.
  - *Continuation passing* style.
- Leaf operations are instantiated templates with statically-known data types.
  - No virtual functions or other barriers to compiler optimization. Note: slow compilation times due to number of permutations of block types.
- Only pay for what you use.
  - Sorts are optional.
  - Overhead for variable-length data only if used.

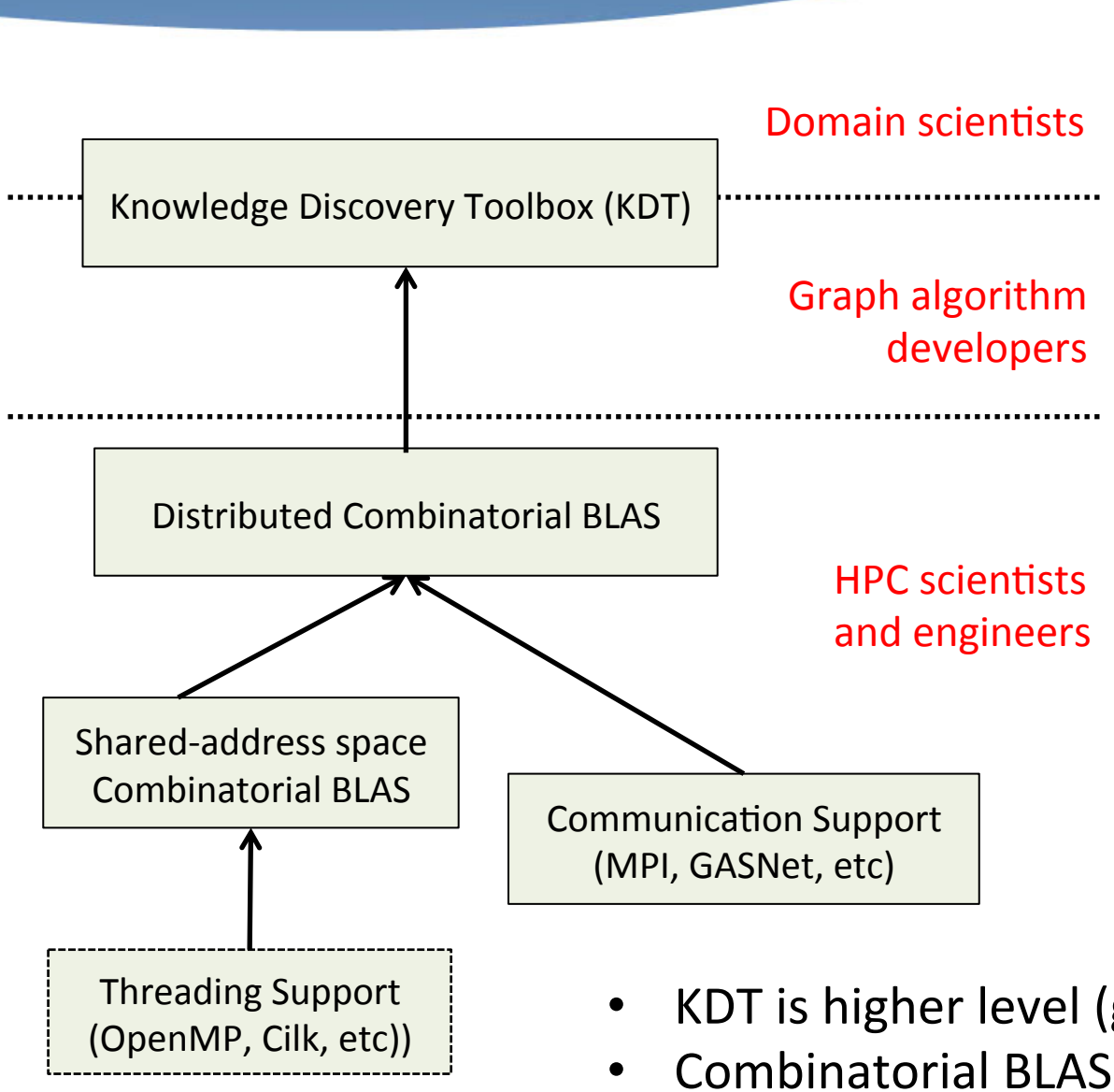
Outline

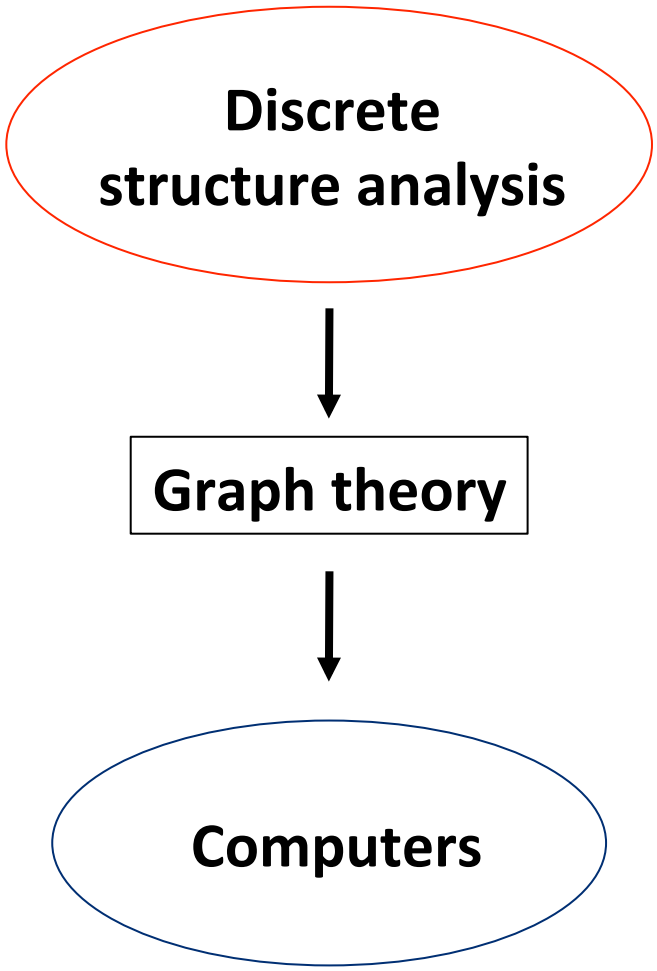
- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Standards for graph algorithm primitives

Parallel graph analysis software



Parallel graph analysis software



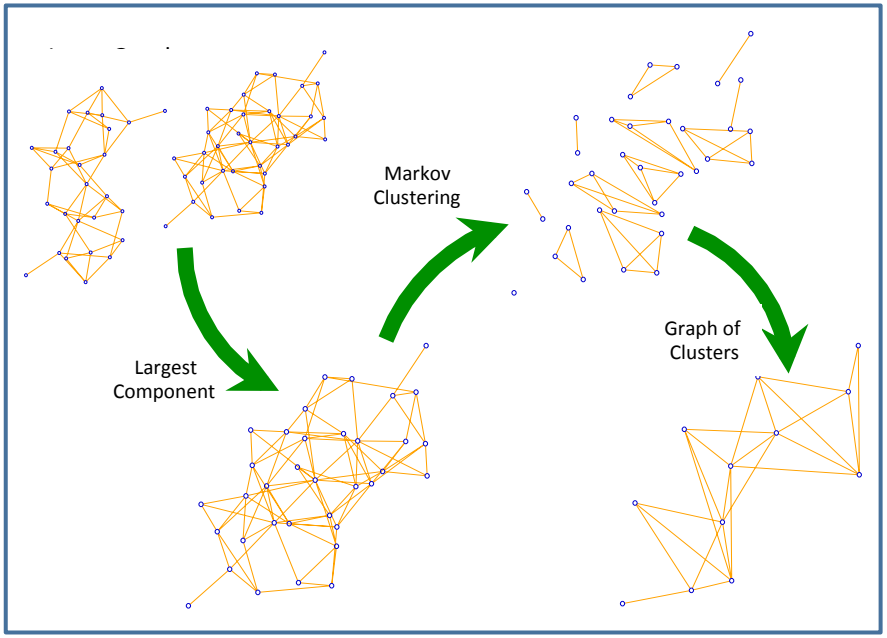


- KDT is higher level (graph abstractions)
- Combinatorial BLAS is for performance

Domain expert vs. graph expert

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering / components
- Centrality / authority:

  betweenness centrality, PageRank



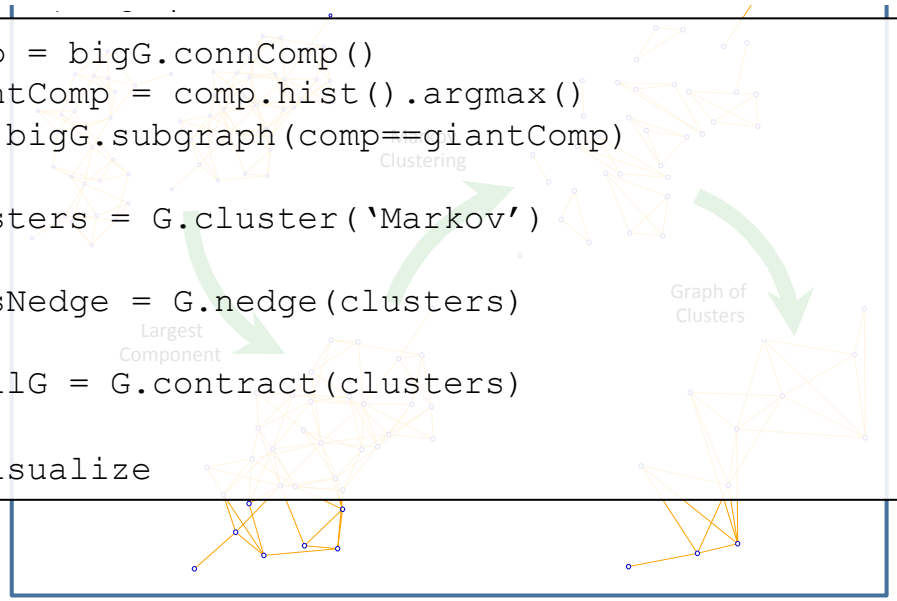
- Hypergraphs and sparse matrices
- Graph primitives (*e.g.*, `bfsTree()`)
- SpMV / SpGEMM on semirings

Domain expert vs. graph expert

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering / components
- Centrality / authority:

  betweenness centrality, PageRank

```
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

clusters = G.cluster('Markov')

clusNedge = G.nedge(clusters)

smallG = G.contract(clusters)

# visualize
```

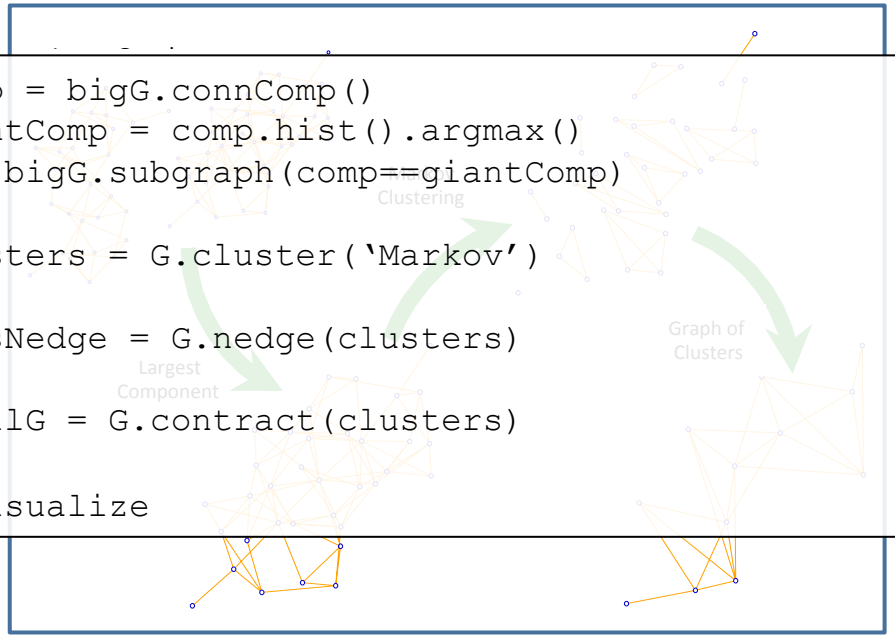


- Hypergraphs and sparse matrices
- Graph primitives (*e.g.*, `bfsTree()`)
- SpMV / SpGEMM on semirings

Domain expert vs. graph expert

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering / components
- Centrality / authority:

  betweenness centrality, PageRank




```
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

clusters = G.cluster('Markov')

clusNedge = G.nedge(clusters)

smallG = G.contract(clusters)

# visualize
```

- Hypergraphs and sparse matrices
- Graph primitives (*e.g.*, `bfsTree()`)
- SpMV / SpGEMM on semirings

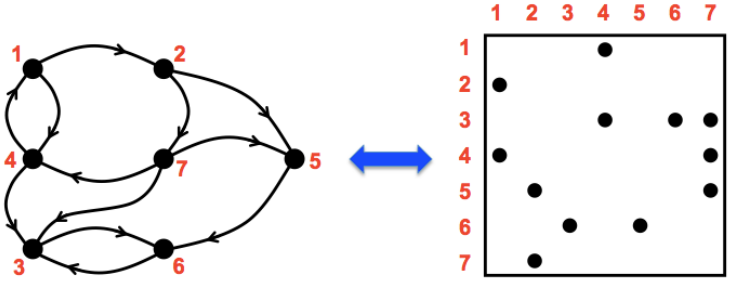
```
                [...]
L = G.toSpParMat()
d = L.sum(kdt.SpParMat.Column)
L = -L
L.setDiag(d)
M = kdt.SpParMat.eye(G.nvert()) - mu*L
pos = kdt.ParVec.rand(G.nvert())
for i in range(nsteps):
    pos = M.SpMV(pos)
```

Knowledge

Discovery

Toolbox

http://kdt.sourceforge.net/



A general graph library with operations based on linear algebraic primitives

- Aimed at domain experts who know their problem well but don't know how to program a supercomputer
- Easy-to-use Python interface
- Runs on a laptop as well as a cluster with 10,000 processors

- Open source software (New BSD license)
- V3 release April 2013 (V4 expected spring 2014)

A few KDT applications

Markov Clustering

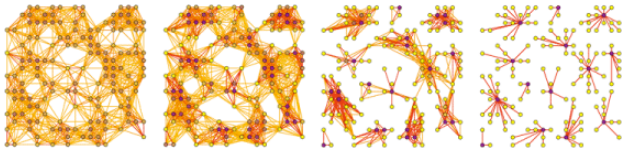
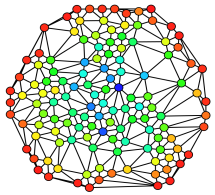


image courtesy Stijn van Dongen

Markov Clustering (MCL) finds clusters by postulating that a random walk that visits a dense cluster will probably visit many of its vertices before leaving.

We use a Markov chain for the random walk. This process is reinforced by adding an inflation step that uses the Hadamard product and rescaling.

Betweenness Centrality



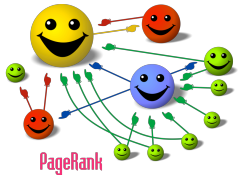
$$C_B(v) = \sum_{\substack{s \neq v \neq t \in V \\ s \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

image courtesy Claudio Rocchini

Betweenness Centrality says that a vertex is important if it appears on many shortest paths between other vertices. An exact computation requires a BFS for every vertex. A good approximation can be achieved by sampling starting vertices.

PageRank

PageRank says a vertex is important if other important vertices link to it.

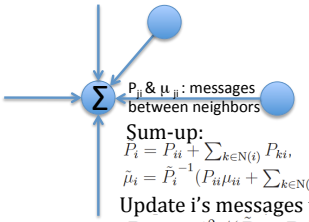


courtesy Felipe Micaroni Lalli

Each vertex (webpage) votes by splitting its PageRank score evenly among its out edges (links). This broadcast (an SpMV) is followed by a normalization step (ColWise). Repeat until convergence.

PageRank is the stationary distribution of a Markov Chain that simulates a "random surfer".

Belief Propagation



Gaussian belief propagation (GaBP) is an iterative algorithm for solving the linear system of equations Ax = b, where A is symmetric positive definite.
GaBP assumes each variable follows a normal distribution. It iteratively calculates the precision P and mean value μ of each variable; the converged mean-value vector approximates the actual solution.

Attributed semantic graphs and filters

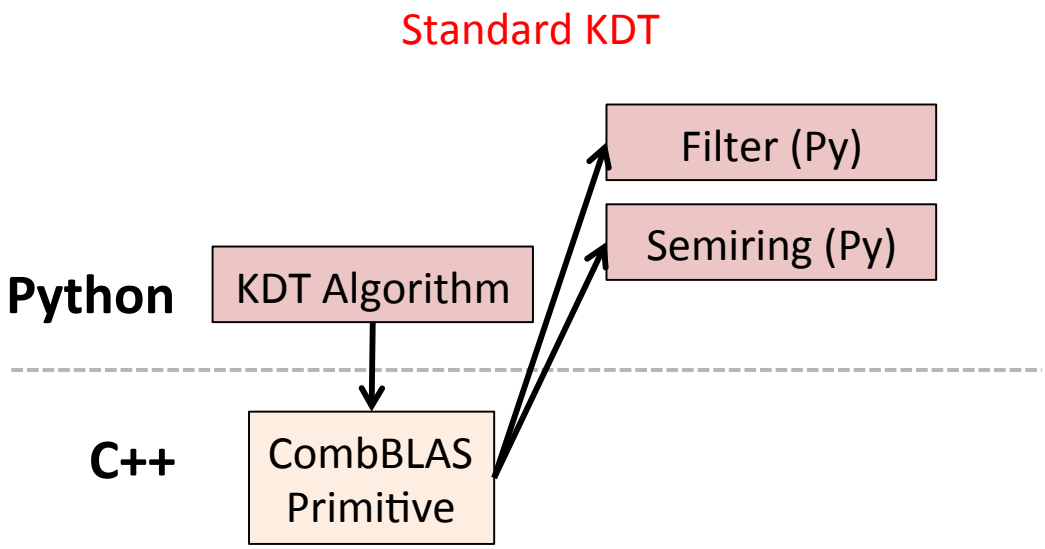
Example:

- Vertex types: Person, Phone, Camera, Gene, Pathway
- Edge types: PhoneCall, TextMessage, CoLocation, SequenceSimilarity
- Edge attributes: Time, Duration

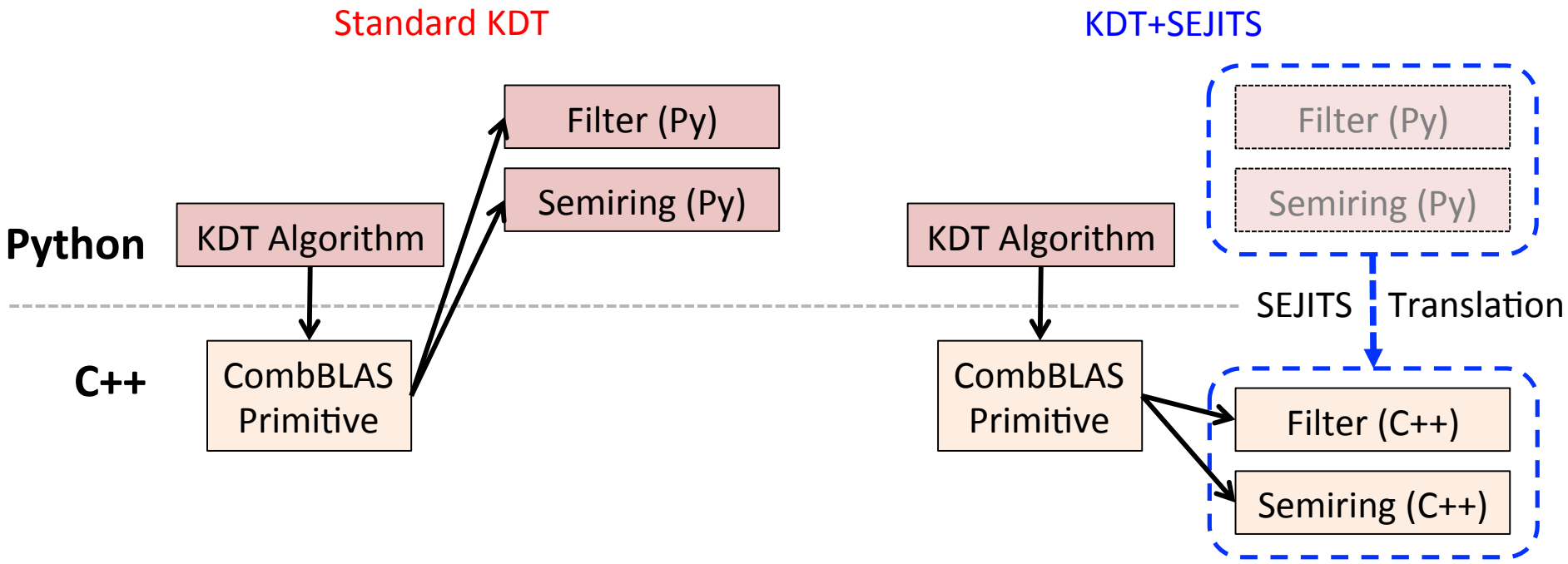
- Calculate centrality just for emails among engineers sent between given start and end times

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

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


G.addVFilter(onlyEngineers)
G.addEFilter(timedEmail(start, end))

# rank via centrality based on recent
email transactions among engineers

bc = G.rank('approxBC')
```

SEJITS for filter/semiring acceleration



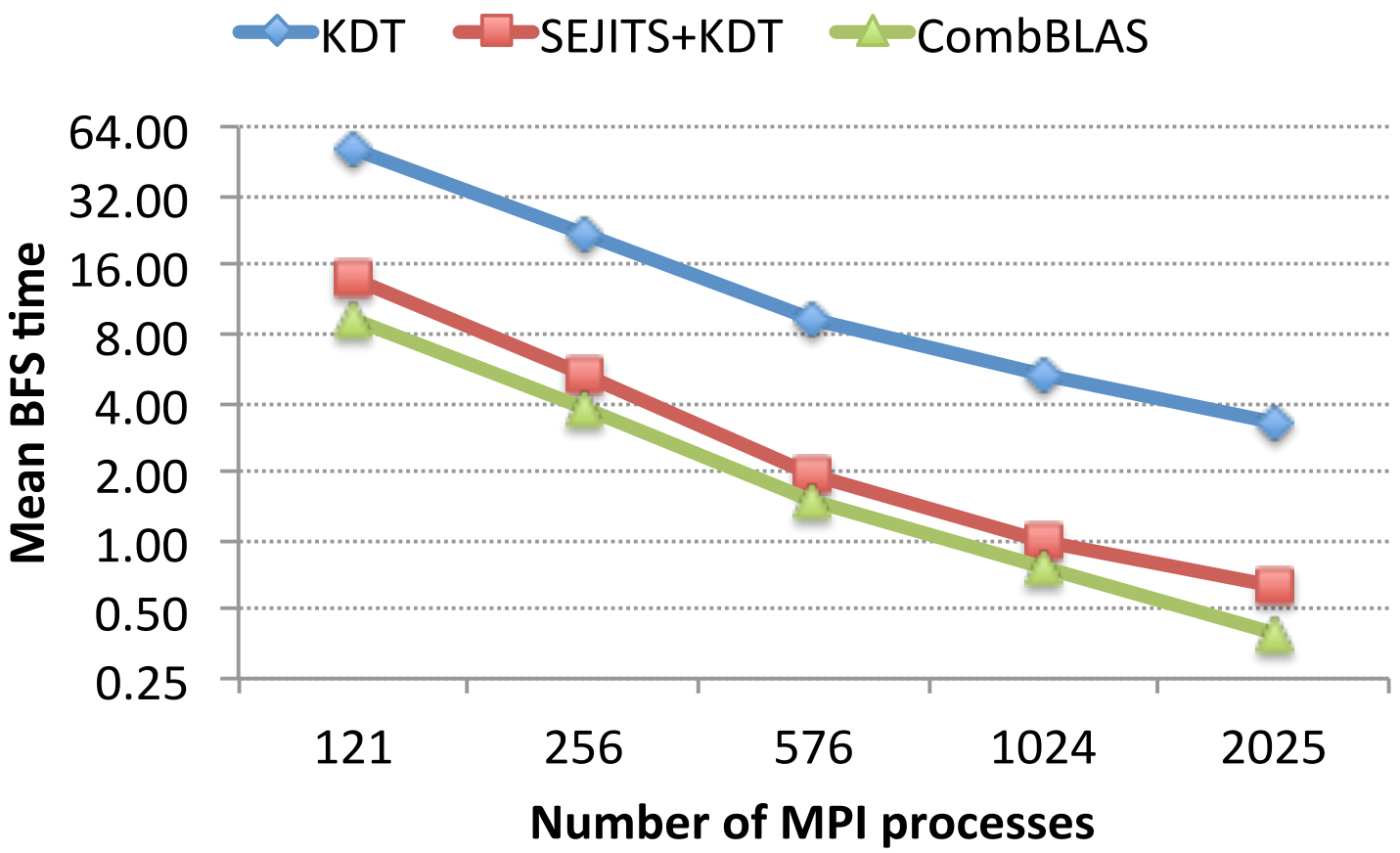
SEJITS for filter/semiring acceleration



Embedded DSL: Python for the whole application

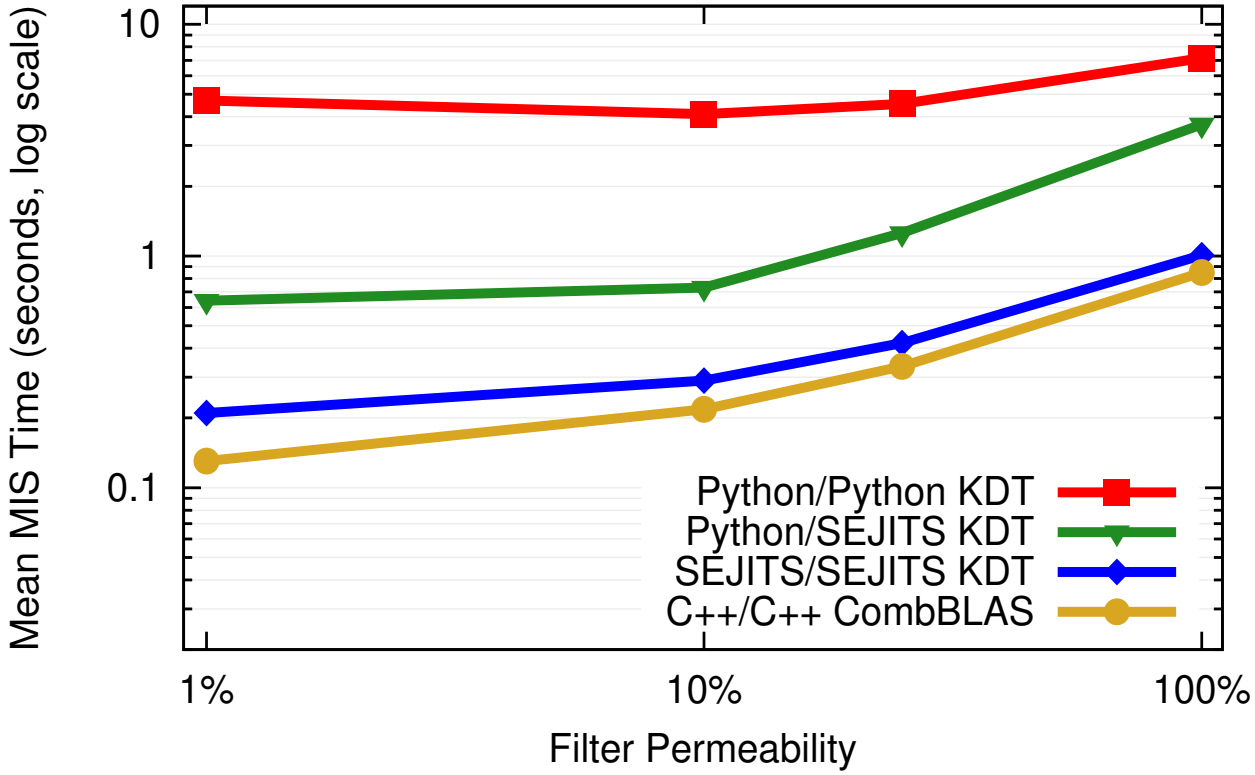
- Introspect, translate Python to equivalent C++ code
- Call compiled/optimized C++ instead of Python

Filtered 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.

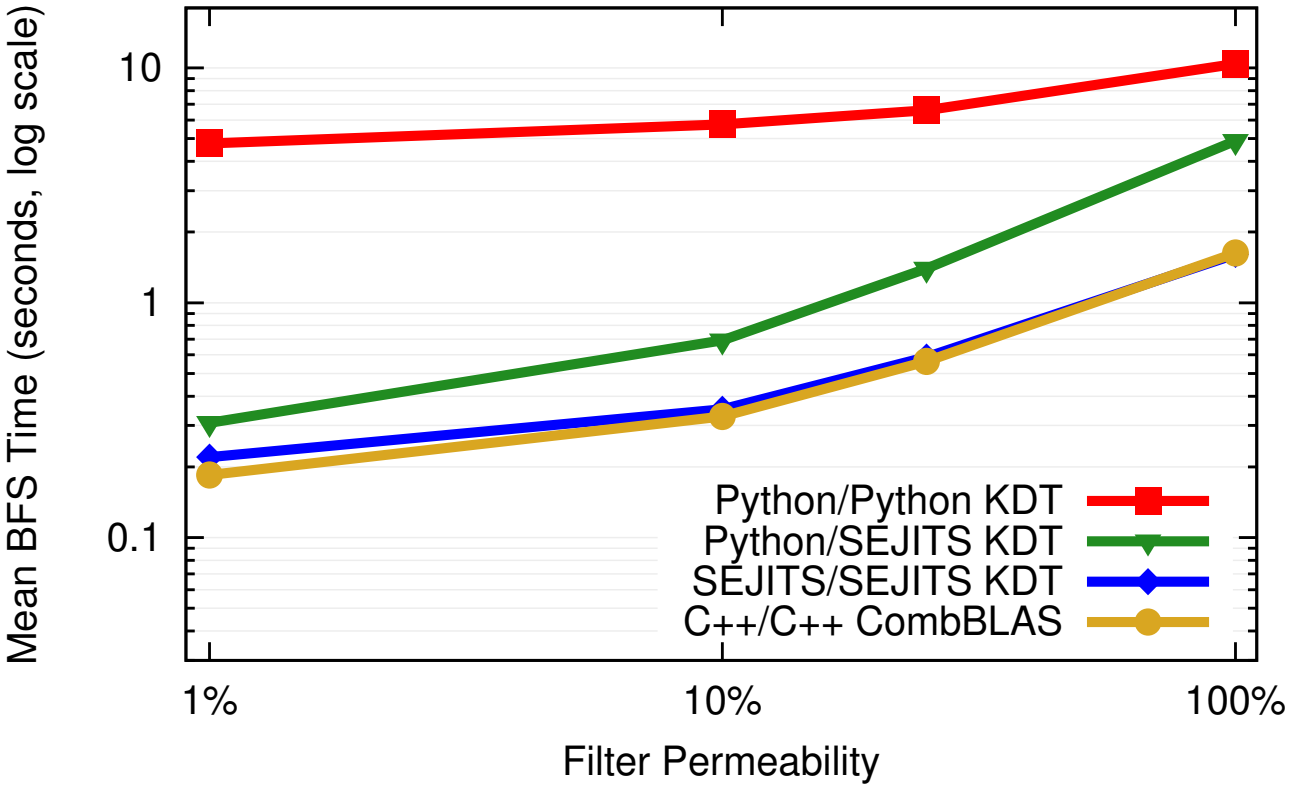
SEJITS+KDT multicore performance



- 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]

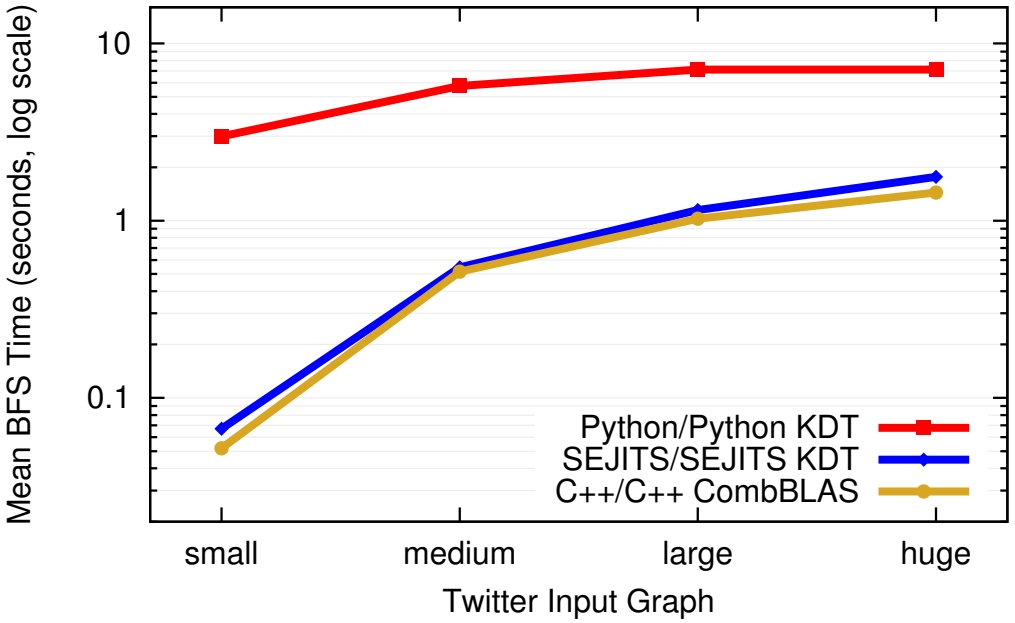
SEJITS+KDT cluster performance



- 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***.

SEJITS+KDT real graph performance



- Breadth-first search
- 16 cores of Mirasol (Intel Xeon E7-8870)

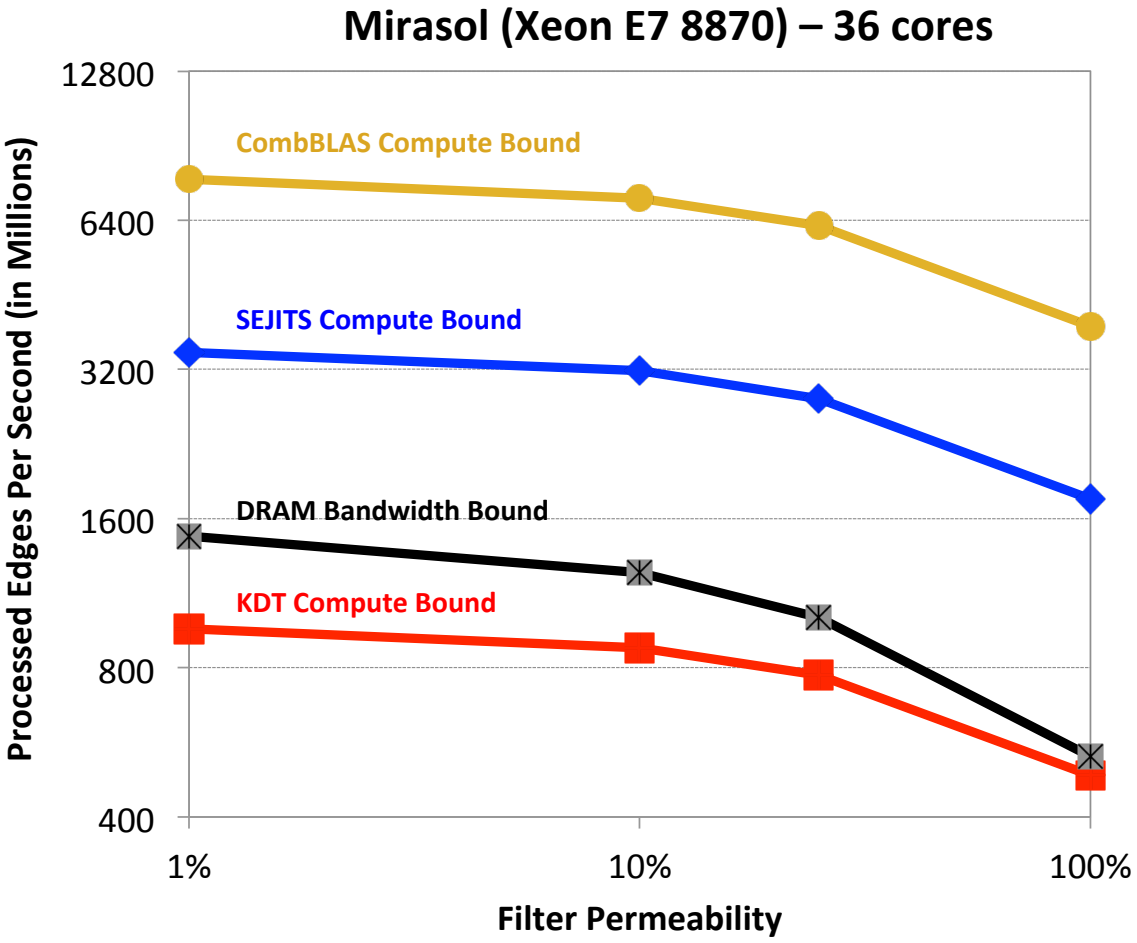
SIZES (VERTEX AND EDGE COUNTS) OF DIFFERENT COMBINED TWITTER GRAPHS.

Label	Vertices (millions)	Edges (millions)		
		Tweet	Follow	Tweet&follow
Small	0.5	0.7	65.3	0.3
Medium	4.2	14.2	386.5	4.8
Large	11.3	59.7	589.1	12.5
Huge	16.8	102.4	634.2	15.6

STATISTICS ABOUT THE LARGEST STRONGLY CONNECTED COMPONENTS OF THE TWITTER GRAPHS

	Vertices	Edges traversed	Edges processed
Small	78,397	147,873	29.4 million
Medium	55,872	93,601	54.1 million
Large	45,291	73,031	59.7 million
Huge	43,027	68,751	60.2 million

Roofline analysis: Why does SEJITS+KDT work?



Even with SEJITS, there are run-time overheads with function calls via pointers.

How is it so close to the Combinatorial BLAS performance?

Because once we are bandwidth bound, additional complexity does not hurt.

Outline

- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Standards for graph algorithm primitives

UCSB

The (original) BLAS

The Basic Linear Algebra Subroutines
had a revolutionary impact
on computational linear algebra.

BLAS 1	vector ops	Lawson, Hanson, Kincaid, Krogh, 1979	LINPACK
BLAS 2	matrix-vector ops	Dongarra, Du Croz, Hammarling, Hanson, 1988	LINPACK on vector machines
BLAS 3	matrix-matrix ops	Dongarra, Du Croz, Hammarling, Hanson, 1990	LAPACK on cache based machines

- Experts in mapping algorithms to hardware tune BLAS for specific platforms.
- Experts in numerical linear algebra build software on top of the BLAS to get high performance "for free."

Today every computer, phone, etc. comes with `/usr/lib/libblas`

Can we define and standardize the “Graph BLAS”?

- No, it is not reasonable to define a universal set of graph algorithm building blocks:
  - Huge diversity in matching algorithms to hardware platforms.
  - No consensus on data structures and linguistic primitives.
  - Lots of graph algorithms remain to be discovered.
  - Early standardization can inhibit innovation.
- Yes, it is reasonable to define a common set of graph algorithm building blocks … for Graphs as Linear Algebra:
  - Representing graphs in the language of linear algebra is a mature field.
  - Algorithms, high level interfaces, and implementations vary.
  - But the core primitives are well established.

Standards for Graph Algorithm Primitives

Tim Mattson (Intel Corporation), David Bader (Georgia Institute of Technology), Jon Berry (Sandia National Laboratory), Aydin Buluc (Lawrence Berkeley National Laboratory), Jack Dongarra (University of Tennessee), Christos Faloutsos (Carnegie Melon University), John Feo (Pacific Northwest National Laboratory), John Gilbert (University of California at Santa Barbara), Joseph Gonzalez (University of California at Berkeley), Bruce Hendrickson (Sandia National Laboratory), Jeremy Kepner (Massachusetts Institute of Technology), Charles Leiserson (Massachusetts Institute of Technology), Andrew Lumsdaine (Indiana University), David Padua (University of Illinois at Urbana-Champaign), Stephen Poole (Oak Ridge National Laboratory), Steve Reinhardt (Cray Corporation), Mike Stonebraker (Massachusetts Institute of Technology), Steve Wallach (Convey Corporation), Andrew Yoo (Lawrence Livermore National Laboratory)

Abstract— It is our view that the state of the art in constructing a large collection of graph algorithms in terms of linear algebraic operations is mature enough to support the emergence of a standard set of primitive building blocks. This paper is a position paper defining the problem and announcing our intention to launch an open effort to define this standard.

Graph Algorithm Building Blocks workshop: GABB @ IPDPS May 2014

Sparse array attribute survey

Function	Graph BLAS	Comb BLAS	Sparse BLAS	STINGER	D4M	SciDB	Tensor Toolbox	Julia	GraphLab
Version		1.3.0	2006	r633	2.5	13.9	2.5	0.2.0	2.2
Language	any	C++	F,C,C++	C	Matlab	C++	Matlab, C++	Julia	C++
Dimension	2	1, 2	2	1, 2, 3	2	1 to 100	2, 3	1,2	2
Index Base	0 or 1	0	0 or 1	0	1	±N	1	1	0
Index Type	uint64	uint64	int	int64	double, string	int64	double	any int	uint64
Value Type	?	user	single, double, complex	int64	logical, double, complex, string	user	logical, double, complex	user	user
Null	0	user	0	0	≤0	null	0	0	int64(-1)
Sparse Format	?	tuple	undef	linked list	dense, csc, tuple	RLE	dense, csc	csc	csr/csc
Parallel	?	2D block	none	block	arbitrary	N-D block, cyclic w/ overlap	none	N-D block, cyclic w/ overlap	Edge based w/ vertex split
+ operations *** operations**	user? user?	user user	+ *	user user	+,*,max,min, ∩,∪	user user	+ *	user user	user user

Matrix times matrix over semiring

Inputs

matrix **A**: $\mathbb{S}^{MxN}$ (sparse or dense)

matrix **B**: $\mathbb{S}^{NxL}$ (sparse or dense)

Optional Inputs

matrix **C**: $\mathbb{S}^{MxL}$ (sparse or dense)

scalar "add" function $\oplus$

scalar "multiply" function $\otimes$

transpose flags for **A**, **B**, **C**

Outputs

matrix **C**: $\mathbb{S}^{MxL}$ (sparse or dense)

Implements $\mathbf{C} \oplus= \mathbf{A} \oplus.\otimes \mathbf{B}$

for j = 1 : N

$\mathbf{C}(i,k) = \mathbf{C}(i,k) \oplus (\mathbf{A}(i,j) \otimes \mathbf{B}(j,k))$

If input **C** is omitted, implements

$\mathbf{C} = \mathbf{A} \oplus.\otimes \mathbf{B}$

Transpose flags specify operation on $\mathbf{A}^T$, $\mathbf{B}^T$, and/or $\mathbf{C}^T$ instead

Notes

$\mathbb{S}$ is the set of scalars, user-specified

$\mathbb{S}$ defaults to IEEE double float

$\oplus$ defaults to floating-point +

$\otimes$ defaults to floating-point *

Specific cases and function names:

SpGEMM: sparse matrix times sparse matrix

SpMSpV: sparse matrix times sparse vector

SpMV: Sparse matrix times dense vector

SpMM: Sparse matrix times dense matrix

Sparse matrix indexing & assignment

Inputs

matrix **A**: $\mathbb{S}^{MxN}$ (sparse)

matrix **B**: $\mathbb{S}^{|p|x|q|}$ (sparse)

vector $p \subseteq \{1, \ldots, M\}$

vector $q \subseteq \{1, \ldots, N\}$

Optional Inputs

none

Outputs

matrix **A**: $\mathbb{S}^{MxN}$ (sparse)

matrix **B**: $\mathbb{S}^{|p|x|q|}$ (sparse)

SpRef Implements $\mathbf{B} = \mathbf{A}(p,q)$

```
for i = 1 : |p|
  for j = 1 : |q|
    B(i,j) = A(p(i),q(j))
```

SpAsgn Implements $\mathbf{A}(p,q) = \mathbf{B}$

```
for i = 1 : |p|
  for j = 1 : |q|
    A(p(i),q(j)) = B(i,j)
```

Notes

$\mathbb{S}$ is the set of scalars, user-specified

$\mathbb{S}$ defaults to IEEE double float

$|p|$ = length of vector p

$|q|$ = length of vector q

Specific cases and function names

SpRef: get sub-matrix

SpAsgn: assign to sub-matrix

Element-wise operations

Inputs

matrix **A**: $\mathbb{S}^{MxN}$ (sparse or dense)

matrix **B**: $\mathbb{S}^{MxN}$ (sparse or dense)

Optional Inputs

matrix **C**: $\mathbb{S}^{MxN}$ (sparse or dense)

scalar "add" function $\oplus$

scalar "multiply" function $\otimes$

Outputs

matrix **C**: $\mathbb{S}^{MxN}$ (sparse or dense)

Implements $\mathbf{C} \oplus= \mathbf{A} \otimes \mathbf{B}$

```
for i = 1 : M
  for j = 1 : N
    C(i,j) = C(i,j) ⊕ (A(i,j) ⊗ B(i,j))
```

If input **C** is omitted, implements

$\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$

Notes

$\mathbb{S}$ is the set of scalars, user-specified

$\mathbb{S}$ defaults to IEEE double float

$\oplus$ defaults to floating-point +

$\otimes$ defaults to floating-point *

Specific cases and function names:

SpEWiseX: matrix elementwise

M=1 or N=1: vector elementwise

Scale: when **A** or **B** is a scalar

Apply/Update

Inputs

matrix **A**: $\mathbb{S}^{MxN}$ (sparse or dense)

Optional Inputs

matrix **C**: $\mathbb{S}^{MxN}$ (sparse or dense)

scalar "add" function $\oplus$

unary function f()

Outputs

matrix **C**: $\mathbb{S}^{MxN}$ (sparse or dense)

Implements $\mathbf{C} \oplus= f(\mathbf{A})$

for i = 1 : M
  for j = 1 : N
    if **A**(i,j) ≠ 0
      C(i,j) = **C**(i,j) $\oplus$ f(**A**(i,j))

If input **C** is omitted, implements
  C = f(**A**)

Notes

$\mathbb{S}$ is the set of scalars, user-specified

$\mathbb{S}$ defaults to IEEE double float

$\oplus$ defaults to floating-point +

Specific cases and function names:

Apply: matrix apply

M=1 or N=1: vector apply

Conclusion

- It helps to look at things from two directions.
- Sparse arrays and matrices yield useful primitives and algorithms for high-performance graph computation.
- Graphs in the language of linear algebra are sufficiently mature to support a standard set of BLAS.