UCSB

Sparse Matrices for High-Performance Graph Computation

John R. Gilbert
University of California, Santa Barbara

with Aydin Buluc, LBNL; Armando Fox, UCB; Shoaib Kamil, MIT; Adam Lugowski, UCSB; Lenny Oliker, LBNL, Sam Williams, LBNL

ENS Lyon
September 25, 2012

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

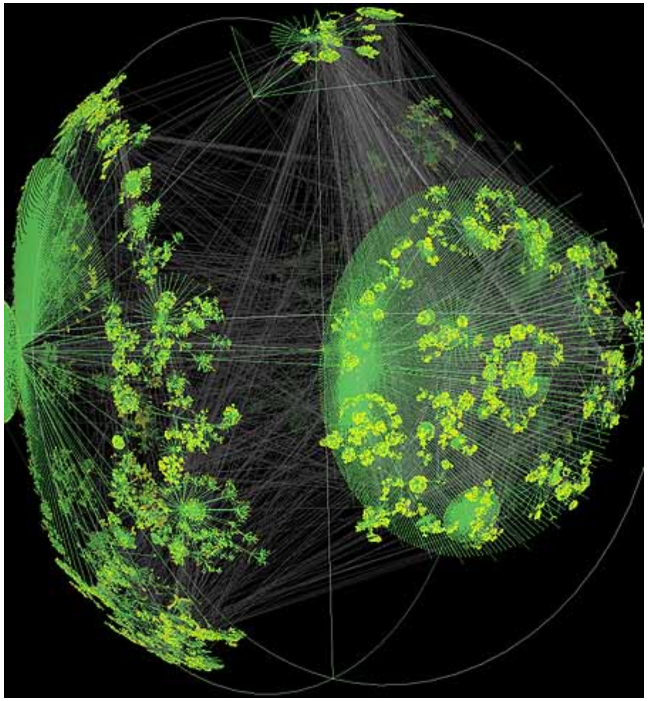
Outline

- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Specialization: getting the best of both worlds

Large graphs are everywhere…

- Internet structure
- Social interactions

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

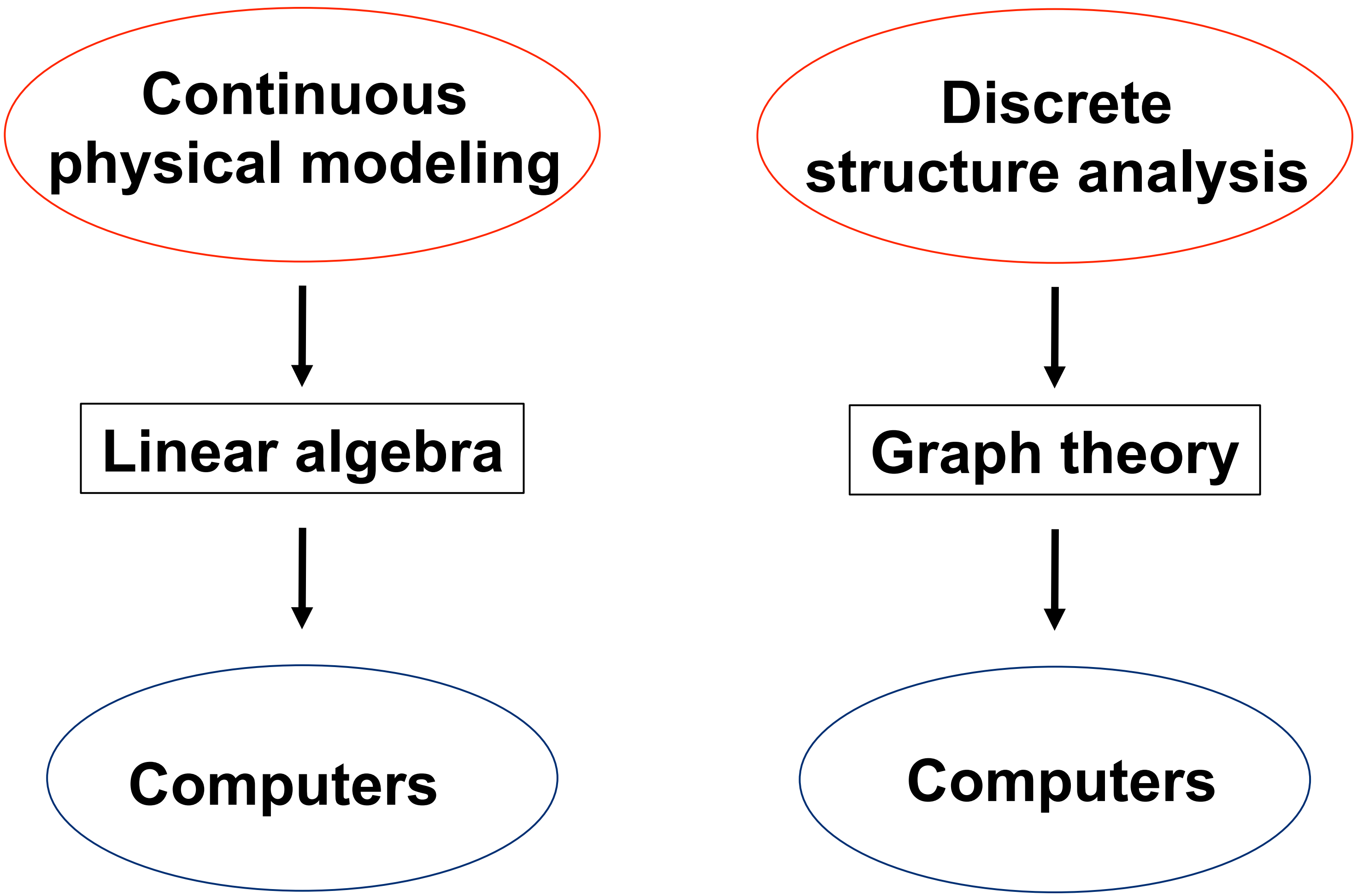


WWW snapshot, courtesy Y. Hyun

Yeast protein interaction network, courtesy H. Jeong

UCSB

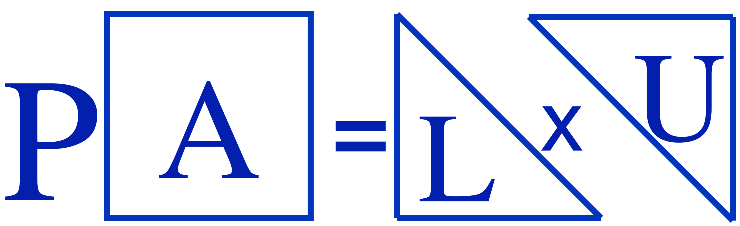
An analogy?


Continuous physical modeling
Linear algebra
Computers
Discrete structure analysis
Graph theory
Computers

Top 500 List (June 2012)

Top500 Benchmark:

Solve a large system
of linear equations
by Gaussian elimination

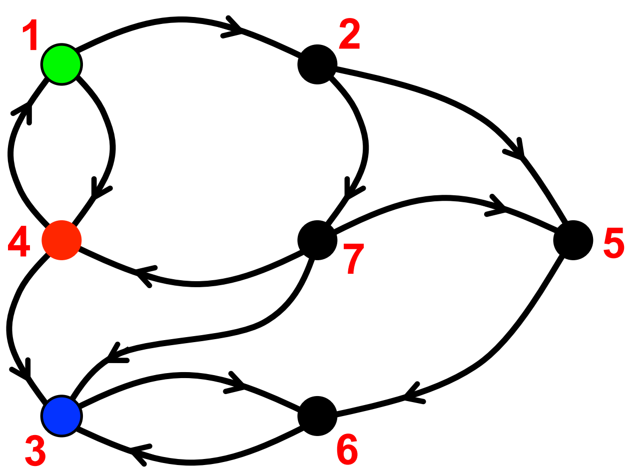


Rank	Site	Computer/Year Vendor	Cores	R_{max}
1	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom / 2011 IBM	1572864	16324.75
2	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	705024	10510.00
3	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	786432	8162.38
4	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR / 2012 IBM	147456	2897.00
5	National Supercomputing Center in Tianjin China	**Tianhe-1A** - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 / 2010 NUDT	186368	2566.00
6	DOE/SC/Oak Ridge National Laboratory United States	**Jaguar** - Cray XK6, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA 2090 / 2009 Cray Inc.	298592	1941.00
7	CINECA Italy	**Fermi** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	163840	1725.49
8	Forschungszentrum Juelich (FZJ) Germany	**JuQUEEN** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	131072	1380.39
9	CEA/TGCC-GENCI France	**Curie thin nodes** - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2012 Bull	77184	1359.00
10	National Supercomputing Centre in Shenzhen (NSCS) China	**Nebulae** - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 / 2010 Dawning	120640	1271.00
11	NASA/Ames Research Center/NAS United States	**Pleiades** - SGI Altix ICE X/8200EX/8400EX, Xeon 54xx 3.0/5570/5670/E5-2670 2.93/2.6 /3.06/3.0 Ghz, Infiniband QDR/FDR / 2011 SGI	125980	1243.00

Graph 500 List (June 2012)

Graph500 Benchmark:

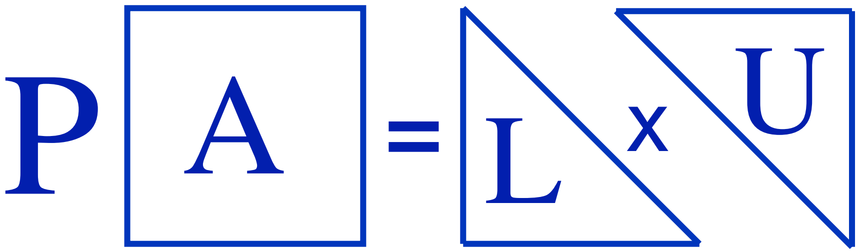
Breadth-first search in a large power-law graph



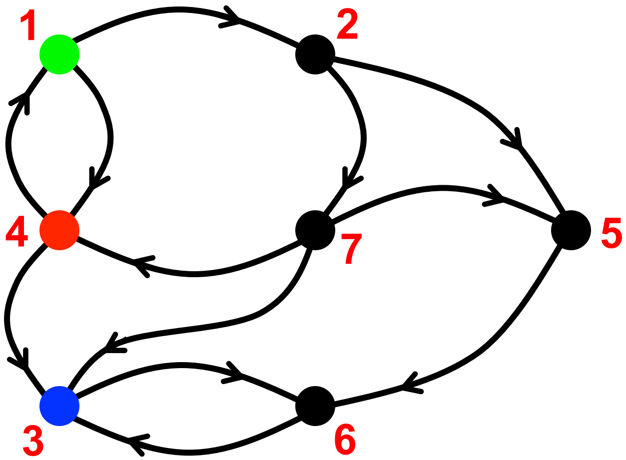
Rank	Installation Site	Machine	Number of nodes	Number of cores	Problem scale	GTEPS
1	DOE/SC/Argonne National Laboratory	Mira/BlueGene/Q	32768	524288	38	3541
1	LLNL	Sequoia/Blue Gene/Q	32768	524288	38	3541
2	DARPA Trial Subset, IBM Development Engineering	Power 775, POWER7 8C 3.836 GHz	1024	32768	35	508.05
3	Information Technology Center, The University of Tokyo	Oakleaf-FX (Fujitsu PRIMEHPC FX 10)	4800	76800	38	358.1
4	GSIC Center, Tokyo Institute of Technology	HP Cluster Platform SL390s G7 (three Tesla cards per node)	1366	16392	35	317.09
5	Brookhaven National Laboratory	BLUE GENE/Q	1024	16384	34	294.293
6	DOE/SC/Argonne National Laboratory	Vesta/BlueGene/Q	1024	16384	34	292.363
7	NASA-Ames / Parallel Computing Lab, Intel Labs	Pleiades - SGI ICE-X, dual plane hypercube FDR infiniband, E5-2670 "sandybridge"	1024	16384	34	270.33
8	NERSC/LBNL	XE6	4817	115600	35	254.074
9	NNSA and IBM Research, T.J. Watson	NNSA/SC Blue Gene/Q Prototype II	4096	65536	32	236
10	GSIC Center, Tokyo Institute of Technology	TSUBAME 2.0 (CPU only)	1366	16392	36	202.68

Floating-Point vs. Graphs, June 2012

16.3 Petaflops



3.54 Terateps



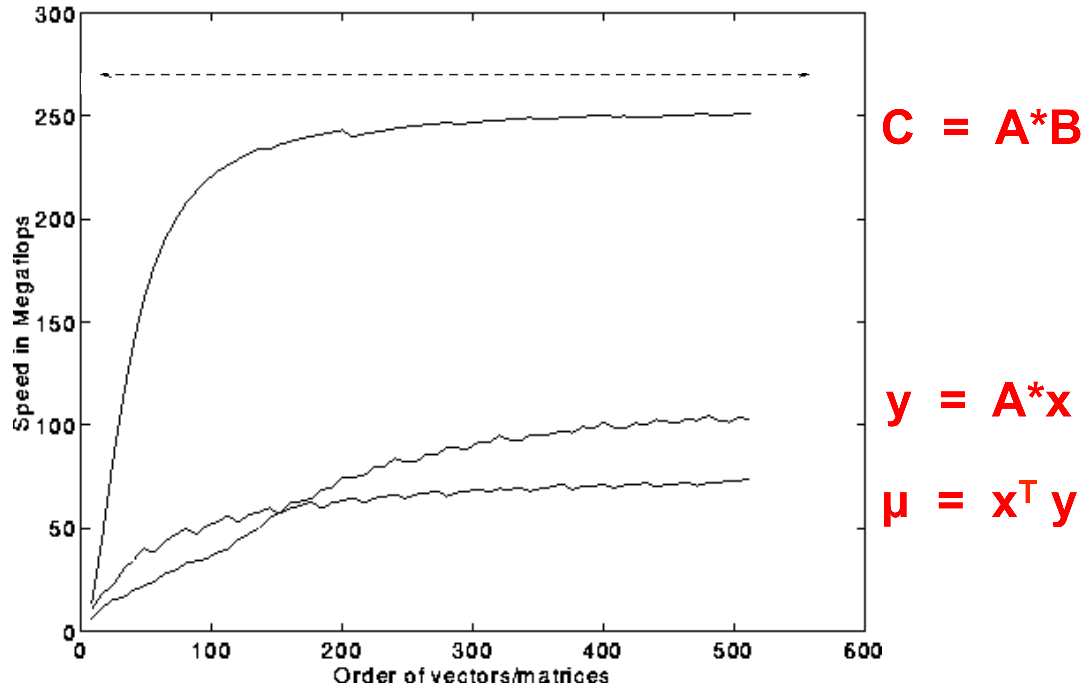
16.3 Peta / 3.54 Tera is about 4600

The challenge of the software stack

- By analogy to numerical scientific computing. . .

- What should the combinatorial BLAS look like?

Basic Linear Algebra Subroutines (BLAS):
Speed (MFlops) vs. Matrix Size (n)

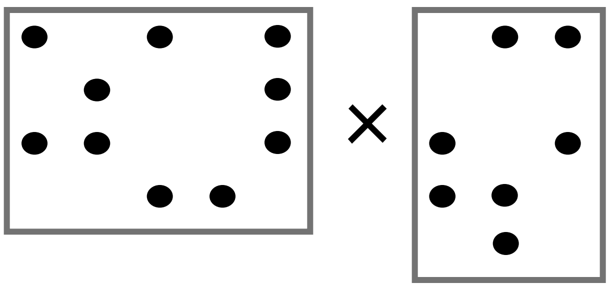


Outline

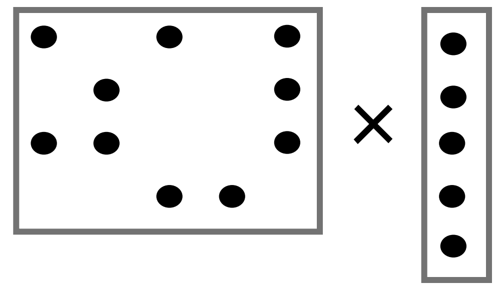
- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Specialization: getting the best of both worlds

Sparse array-based primitives

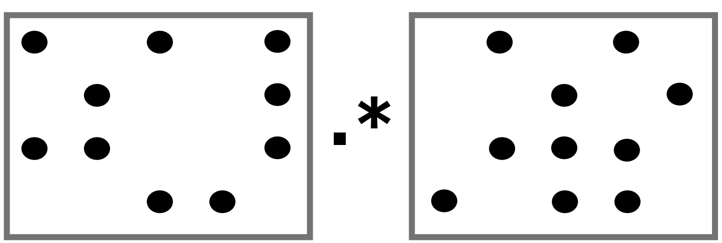
Sparse matrix-matrix multiplication (SpGEMM)



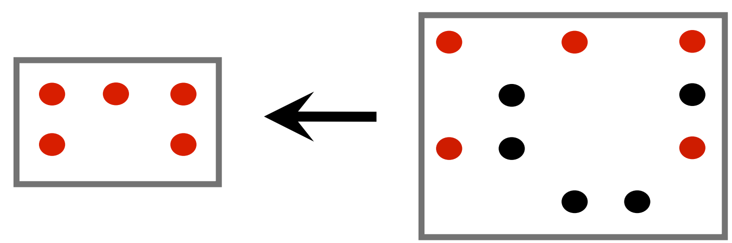
Sparse matrix-dense vector multiplication



Element-wise operations

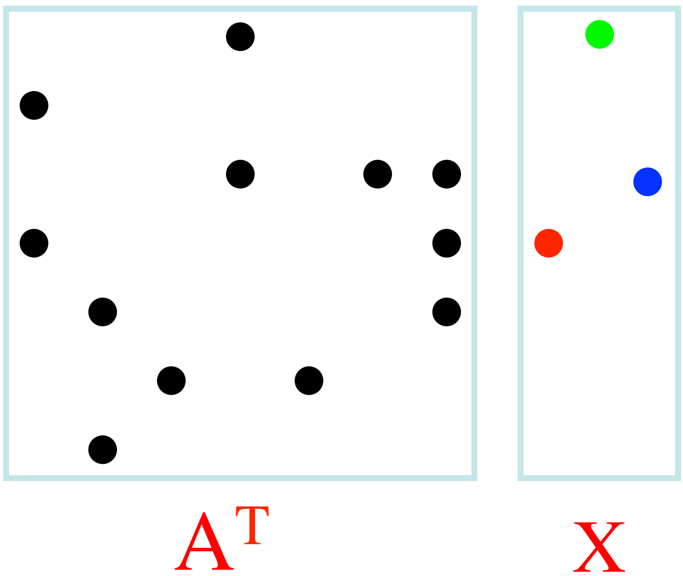


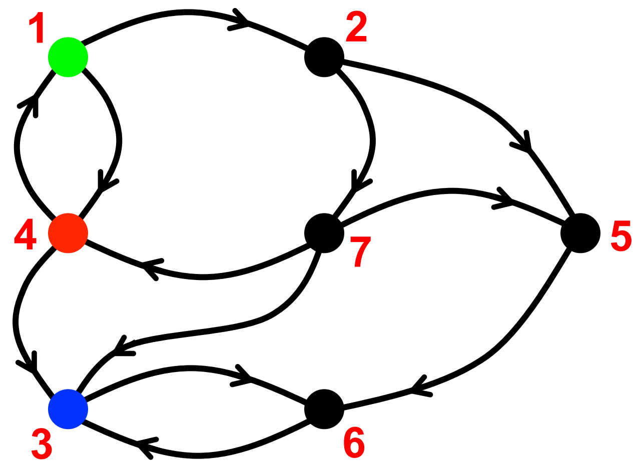
Sparse matrix indexing



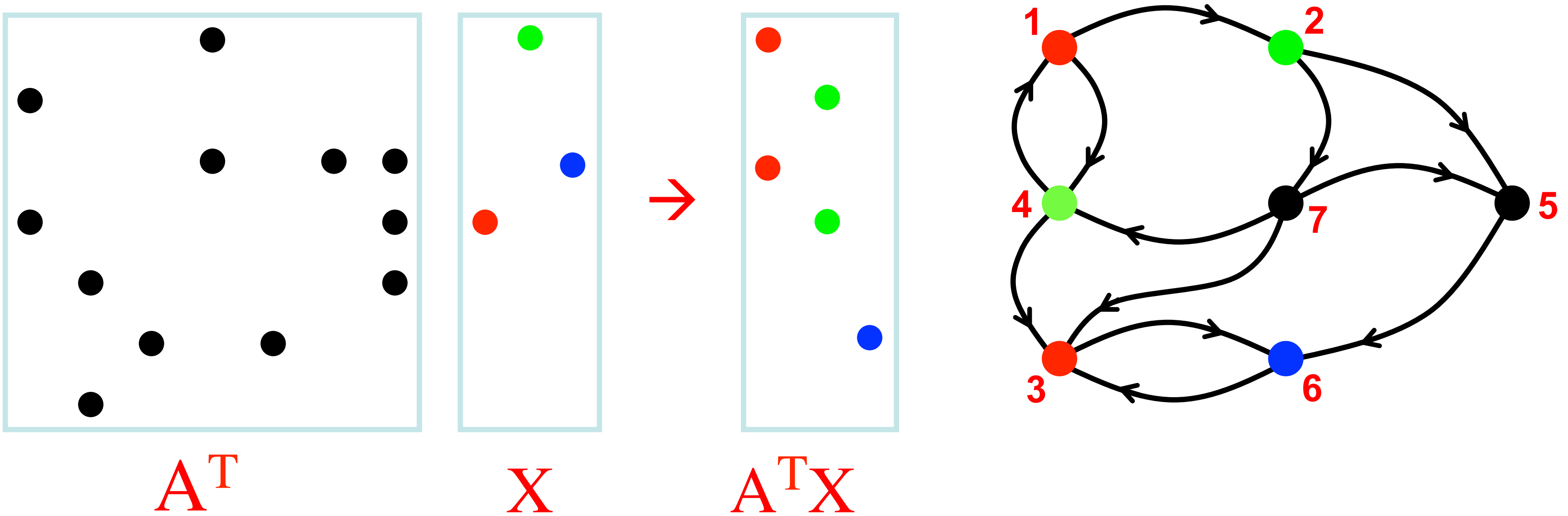
Matrices on various semirings: (x, +) , (and, or) , (+, min) , …

Multiple-source breadth-first search





Multiple-source breadth-first search

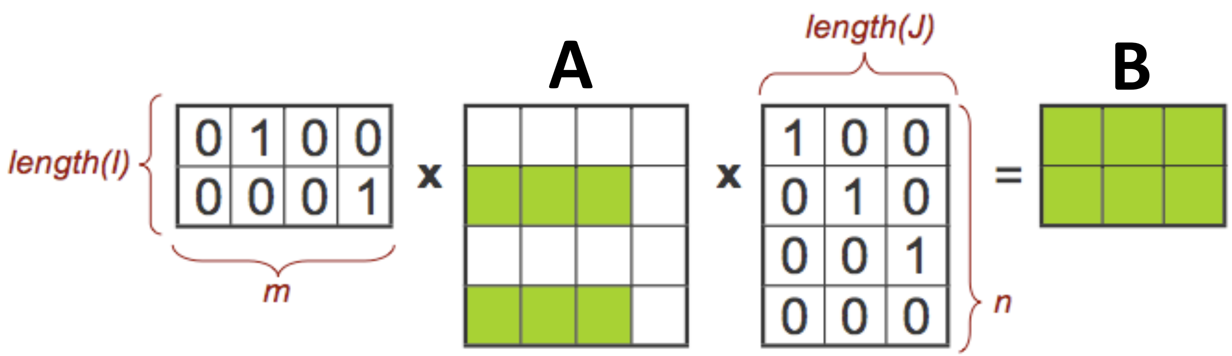


- Sparse array representation => space efficient
- Sparse matrix-matrix multiplication => work efficient
- Three possible levels of parallelism: searches, vertices, edges

Indexing sparse arrays in parallel
(extract subgraphs, coarsen grids, etc.)

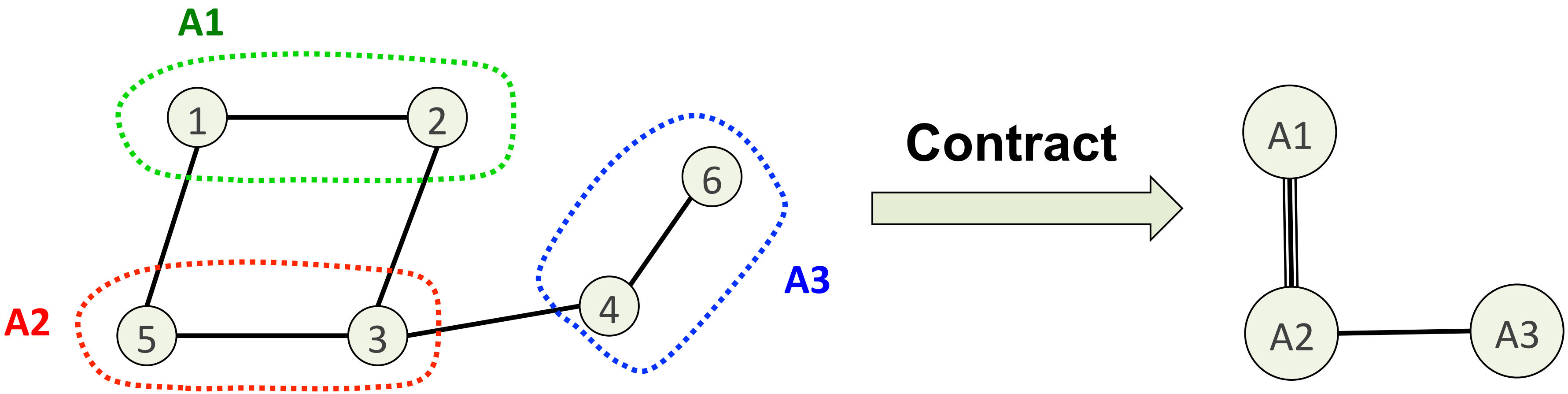
```
SpRef:     B = A(I,J)
SpAsgn:    B(I,J) = A
SpExpAdd:  B(I,J) += A
```

A,B: sparse matrices
I,J: vectors of indices



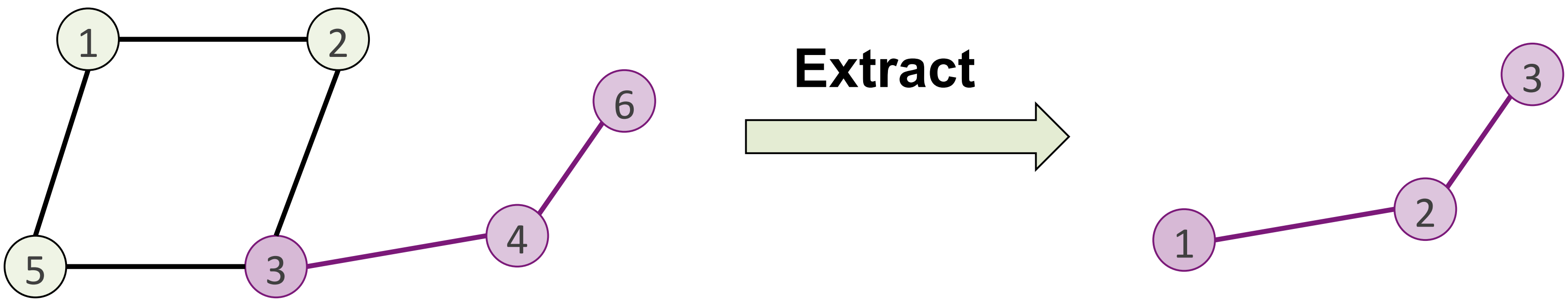
SpRef using mixed-mode sparse matrix-matrix multiplication (**SpGEMM**). Ex: B = A([2,4], [1,2,3])

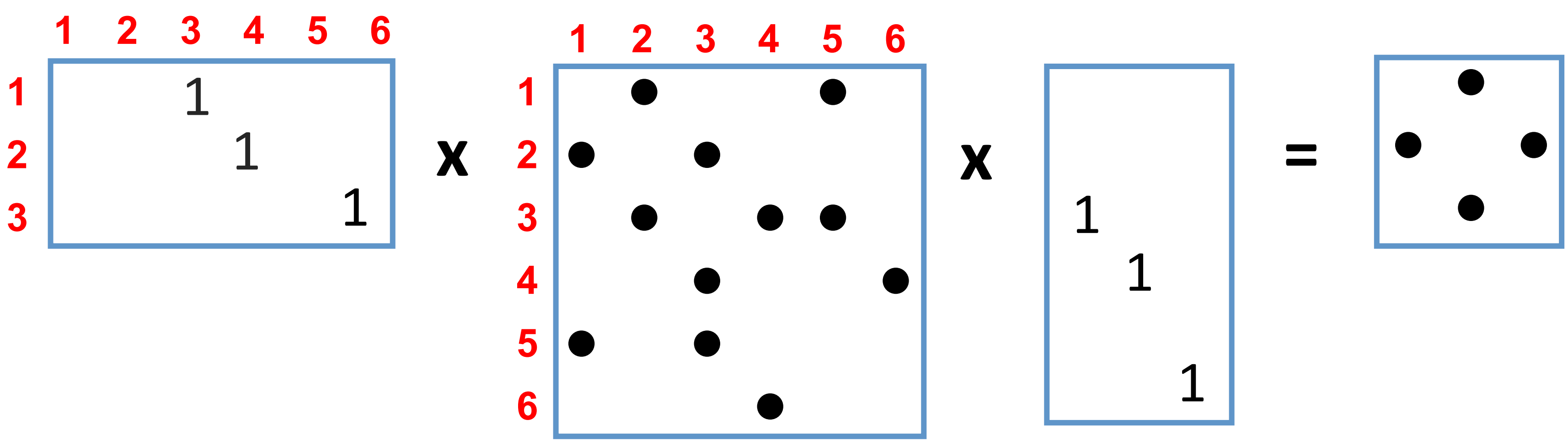
Graph contraction via sparse triple product



$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 1 & 1 & & & & \\
2 & & & 1 & & 1 & \\
3 & & & & 1 & & 1 \\
\hline
\end{array}
\;\times\;
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & & \bullet & & & \bullet & \\
2 & \bullet & & \bullet & & & \\
3 & & \bullet & & \bullet & \bullet & \\
4 & & & \bullet & & & \bullet \\
5 & \bullet & & \bullet & & & \\
6 & & & & \bullet & & \\
\hline
\end{array}
\;\times\;
\begin{bmatrix}
1 & & \\
1 & & \\
 & 1 & \\
 & & 1 \\
 & 1 & \\
 & & 1
\end{bmatrix}
\;=\;
\begin{bmatrix}
 & \bullet & \\
\bullet & & \bullet \\
 & \bullet &
\end{bmatrix}
$$

Subgraph extraction via sparse triple product





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

UCSB

Outline

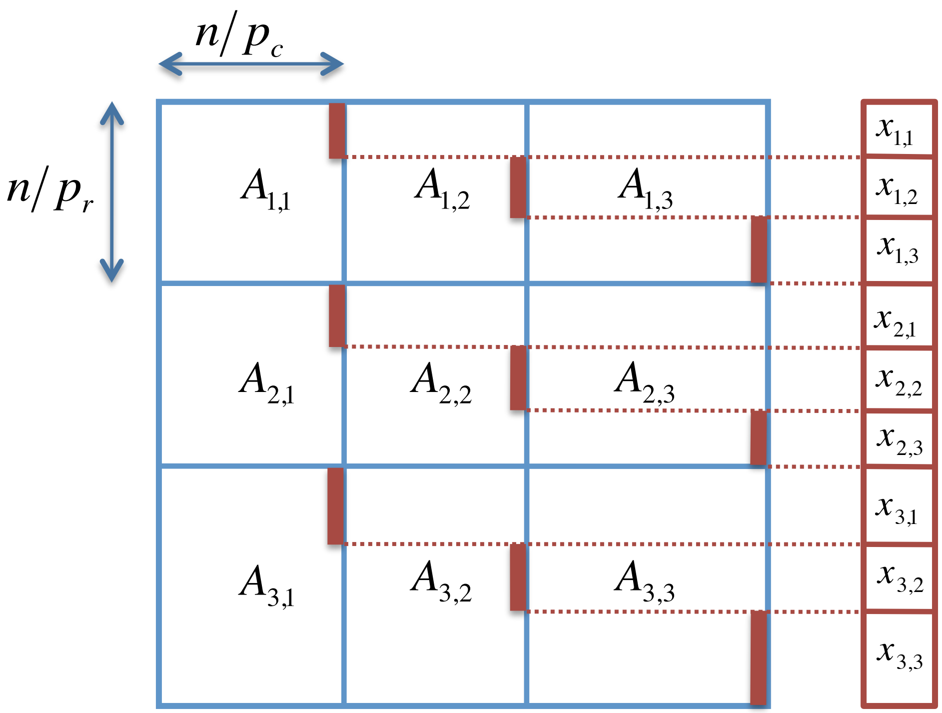
- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Specialization: getting the best of both worlds

UCSB

Some 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

2D layout for sparse matrices & vectors



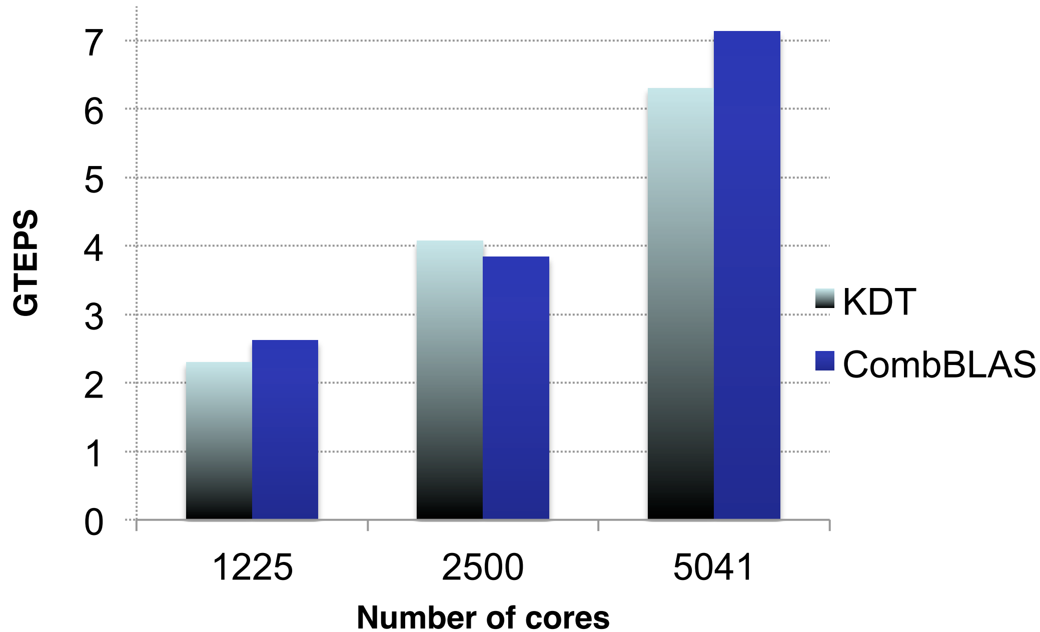
Matrix/vector distributions, interleaved on each other.

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

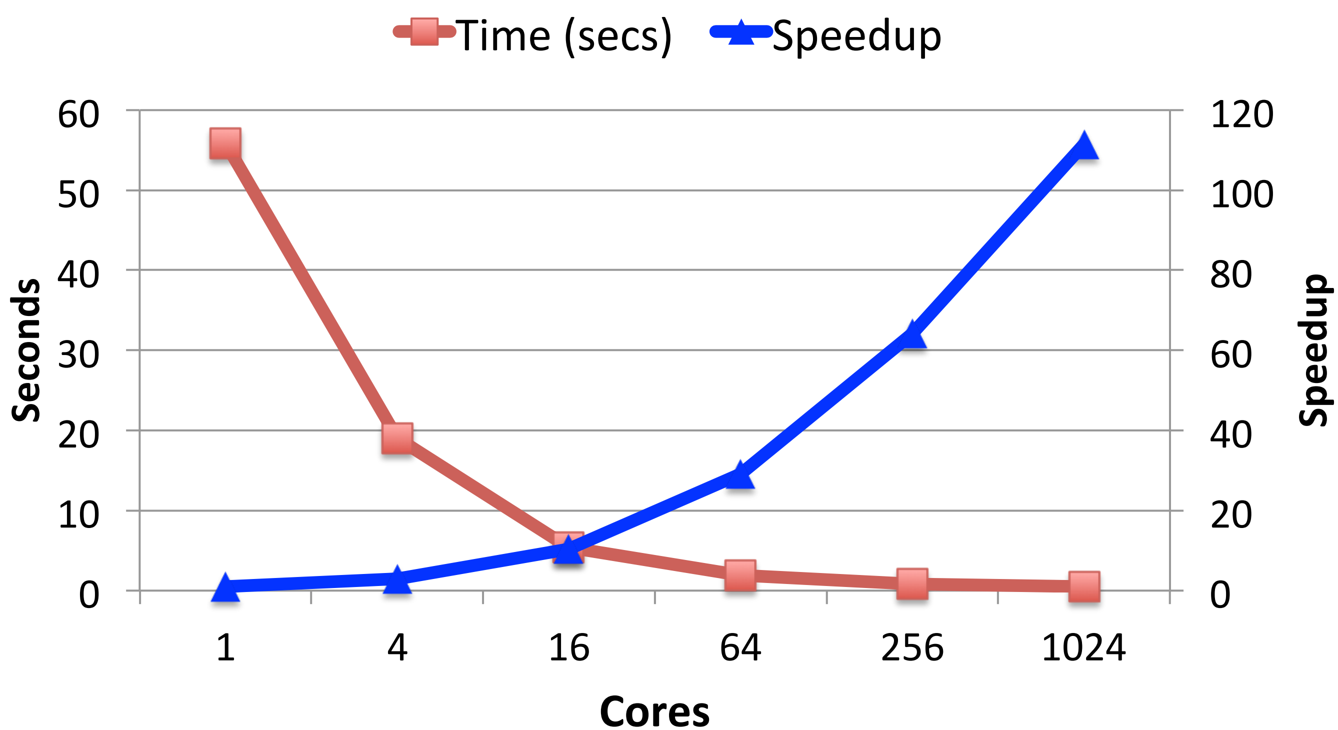
BFS in “vanilla” MPI Combinatorial BLAS

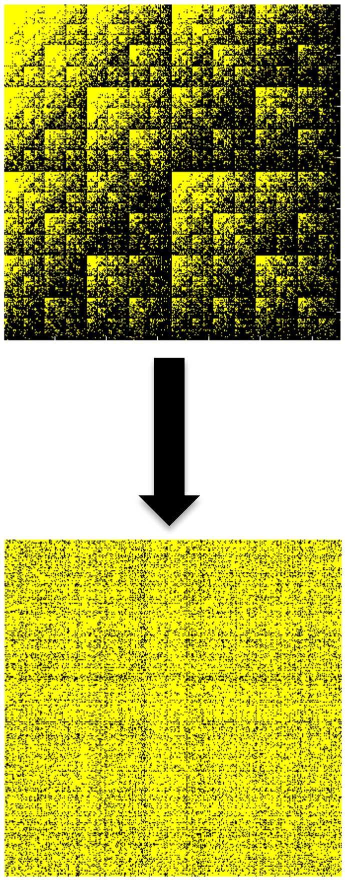


- Graph500 benchmark at scale 29, C++ (or KDT) calling CombBLAS
- NERSC “Hopper” machine (Cray XE6)

UCSB

Strong scaling of vertex relabeling

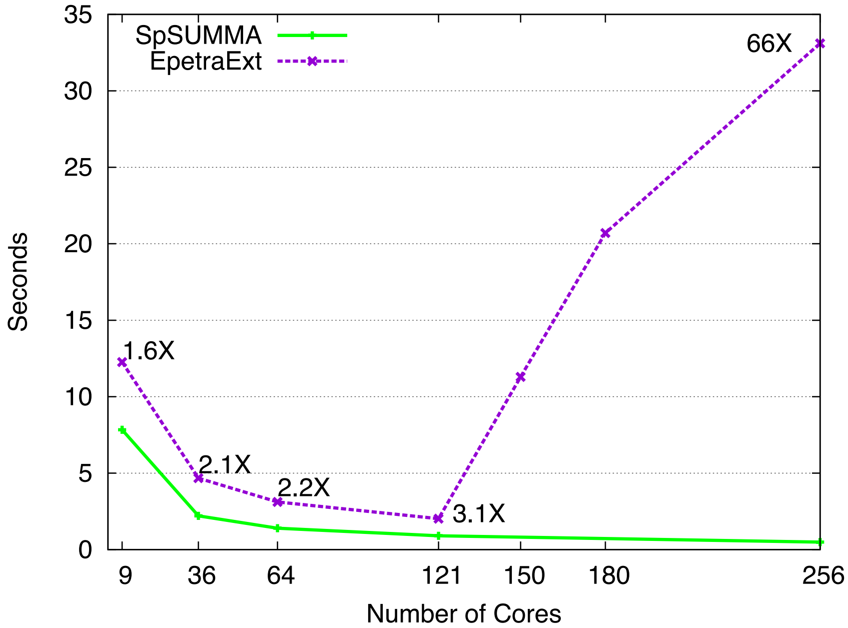




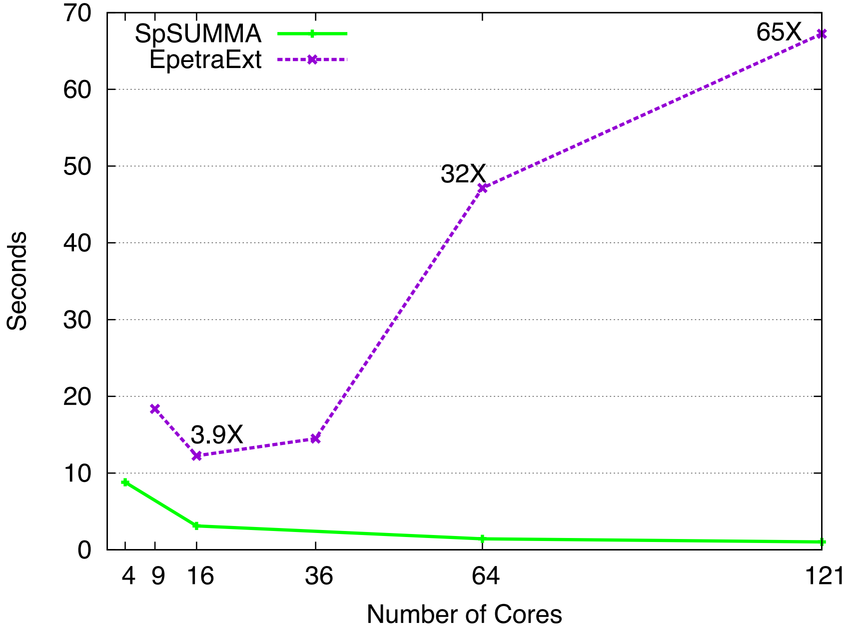
symmetric permutation ⇔ relabeling graph vertices

- RMAT Scale 22; edge factor=8; a=.6, b=c=d=.4/3
- Franklin/NERSC, each node is a quad-core AMD Budapest

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



(a) R-MAT × R-MAT product (scale 21).



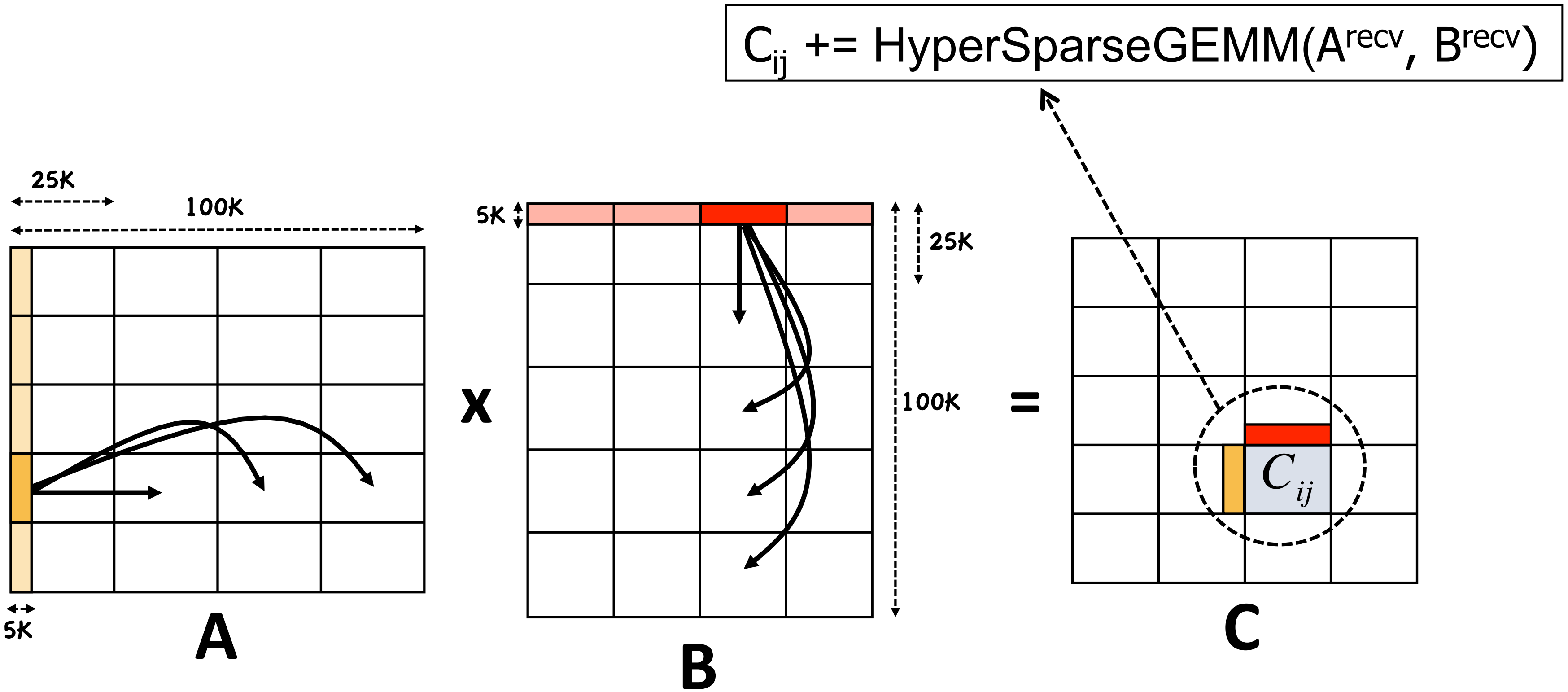
(b) Multiplication of an R-MAT matrix of scale 23 with the restriction operator of order 8.

SpSUMMA = 2-D data layout (Combinatorial BLAS)
EpetraExt = 1-D data layout (Trilinos)

In practice, 2D algorithms have the potential to scale, but not linearly

$$T_{comm}(2\mathrm{D}) = \alpha p\sqrt{p} + \beta c n\sqrt{p} \qquad T_{comp}(optimal) = c^2 n$$

Parallel sparse matrix-matrix multiplication algorithms



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

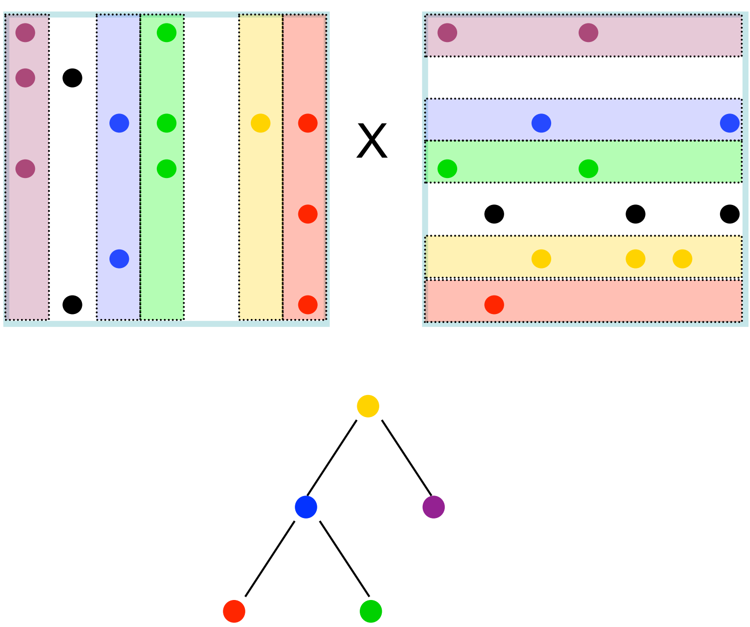
General implementation that handles rectangular matrices

Sequential “hypersparse” kernel

Operates on the strictly O(nnz) DCSC data structure

Sparse outer-product formulation with multi-way merging

Efficient in parallel, i.e. T(1) ≈ p T(p)



Time complexity:

$$O(flops \cdot \lg ni + nzc(A) + nzr(B))$$

- independent of dimension

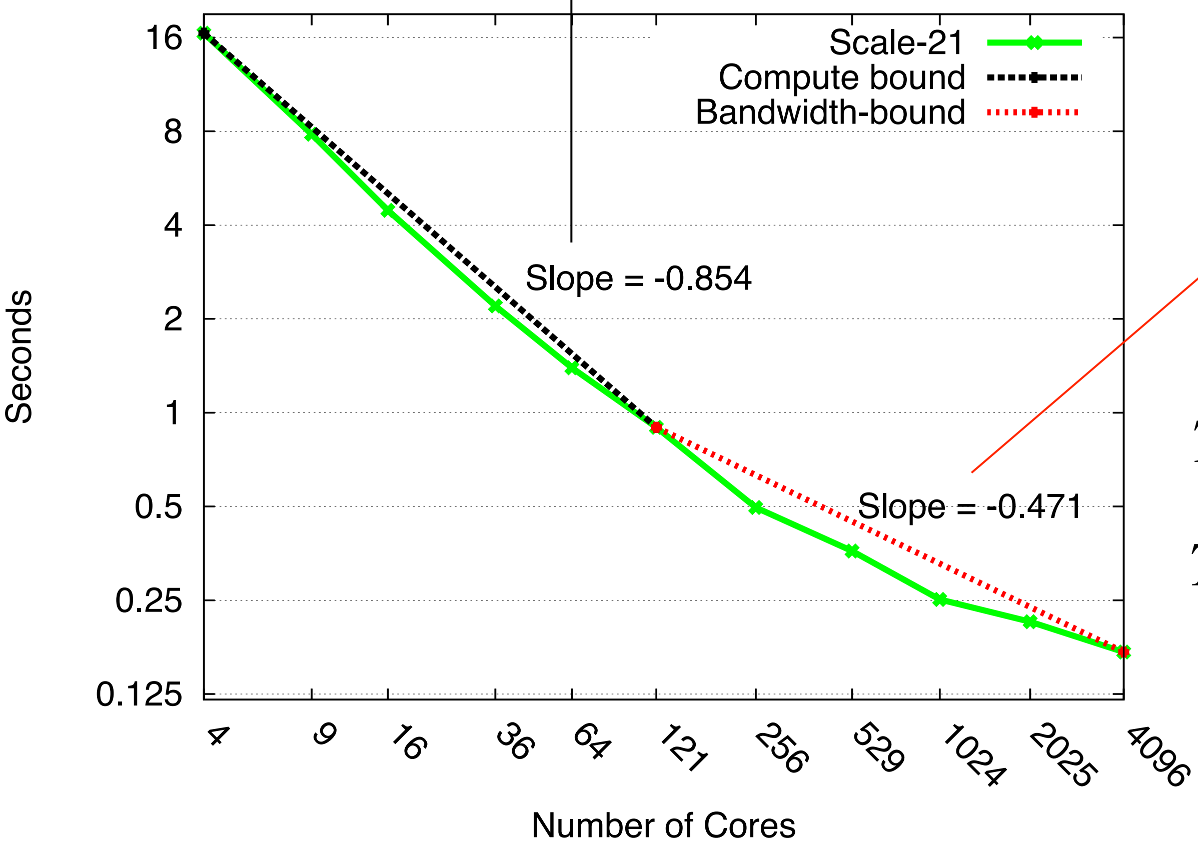
Space complexity:

$$O(nnz(A) + nnz(B) + nnz(C))$$

- independent of flops

Square sparse matrix multiplication

Almost linear scaling until bandwidth costs starts to dominate



Scaling proportional to √p afterwards

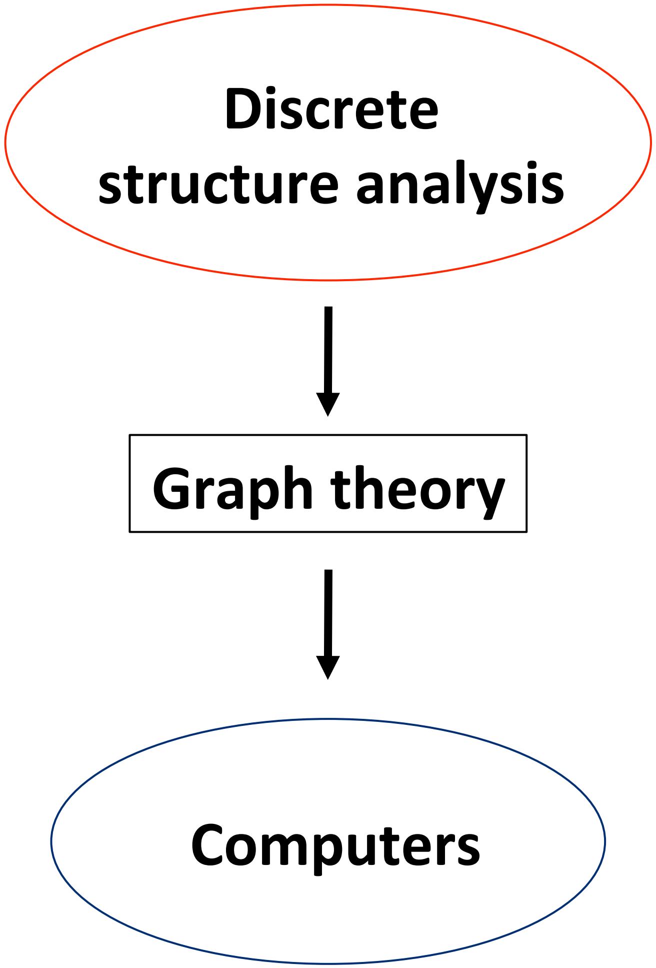
$$T_{comm} = \alpha p\sqrt{p} + \beta c n\sqrt{p}$$

$$T_{comp}(optimal) = c^2 n$$

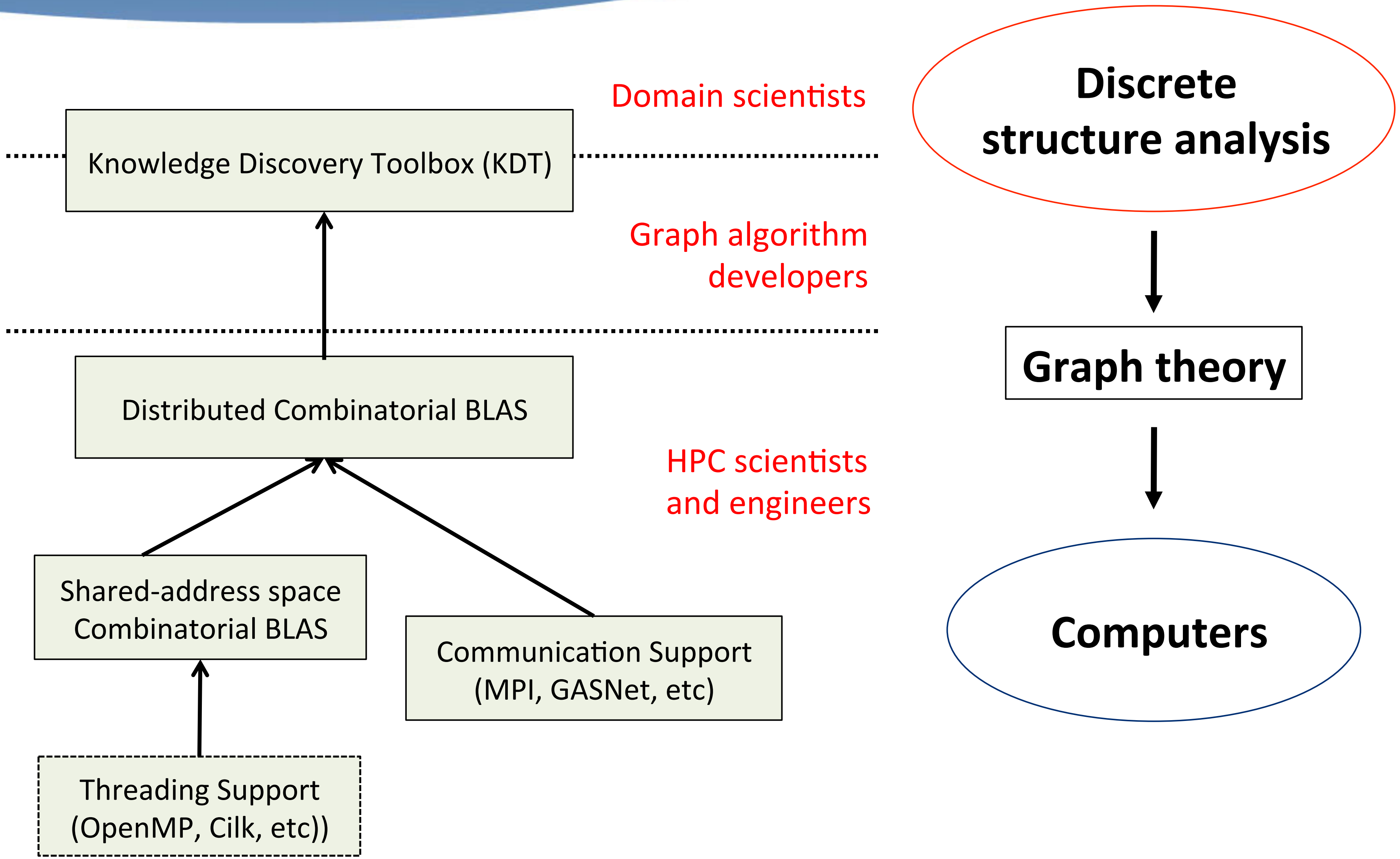
Outline

- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Specialization: getting the best of both worlds

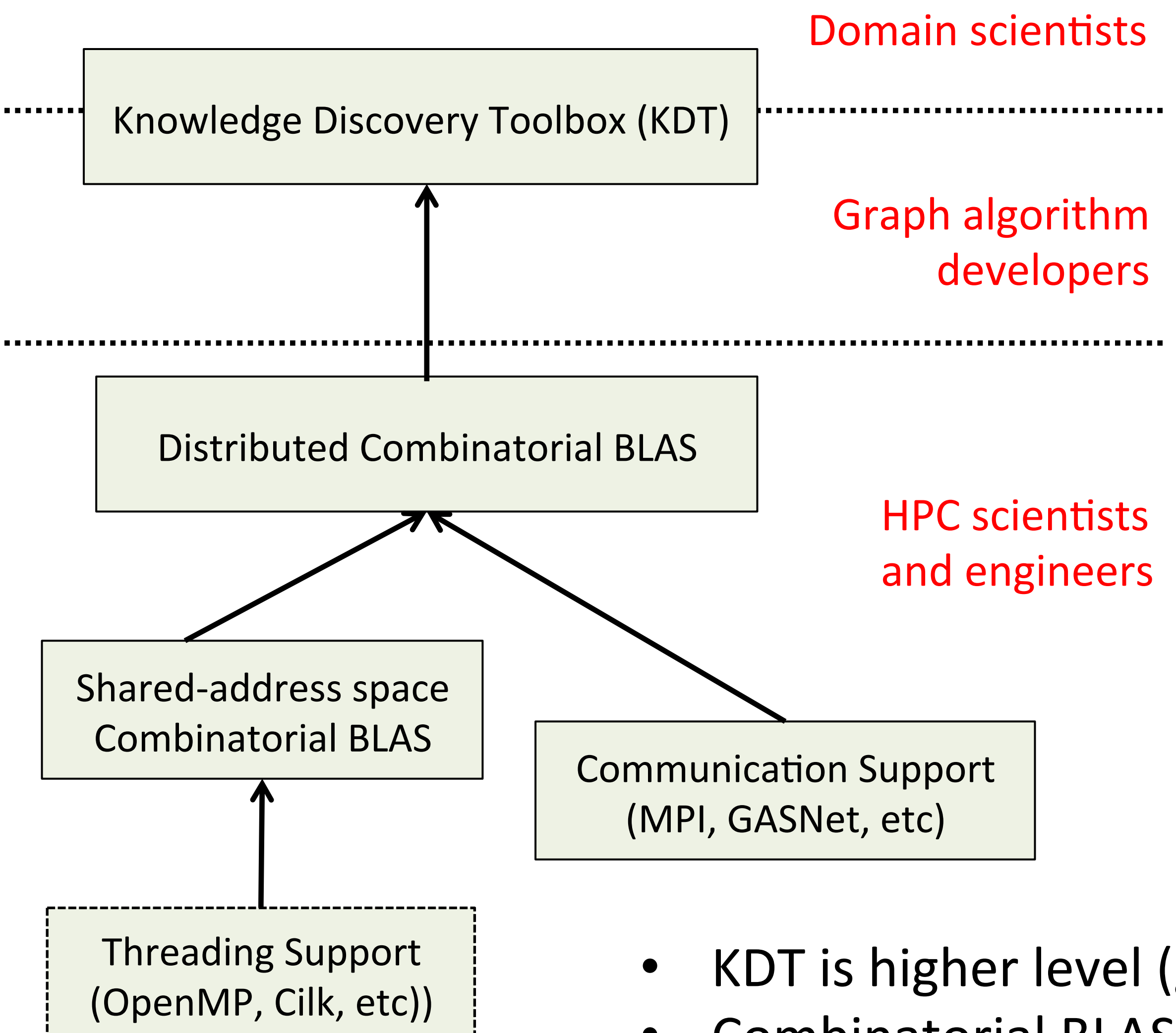
Parallel Graph Analysis Software

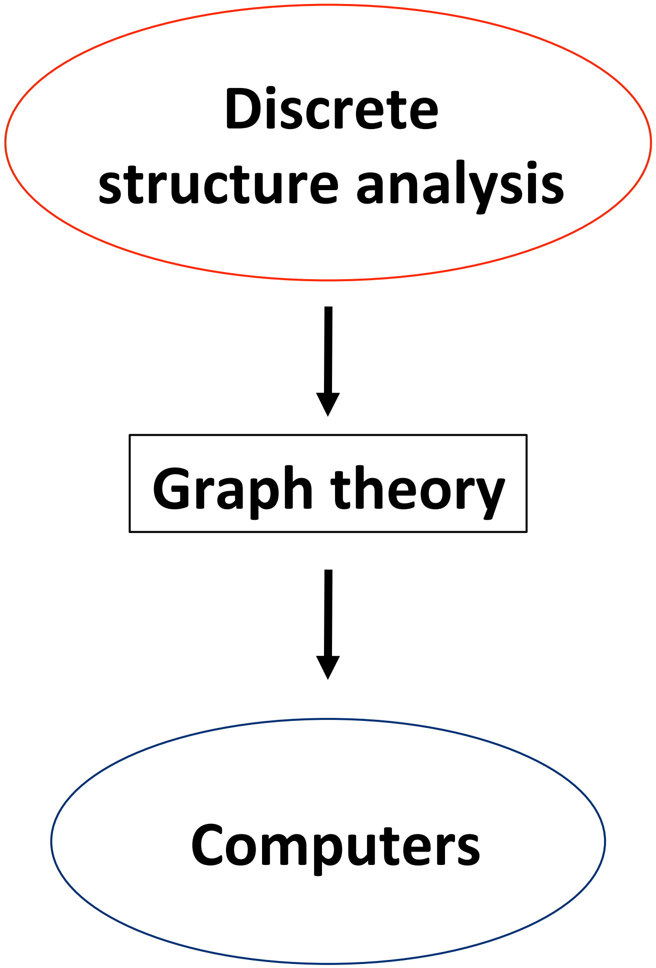


Parallel Graph Analysis Software



Parallel Graph Analysis Software

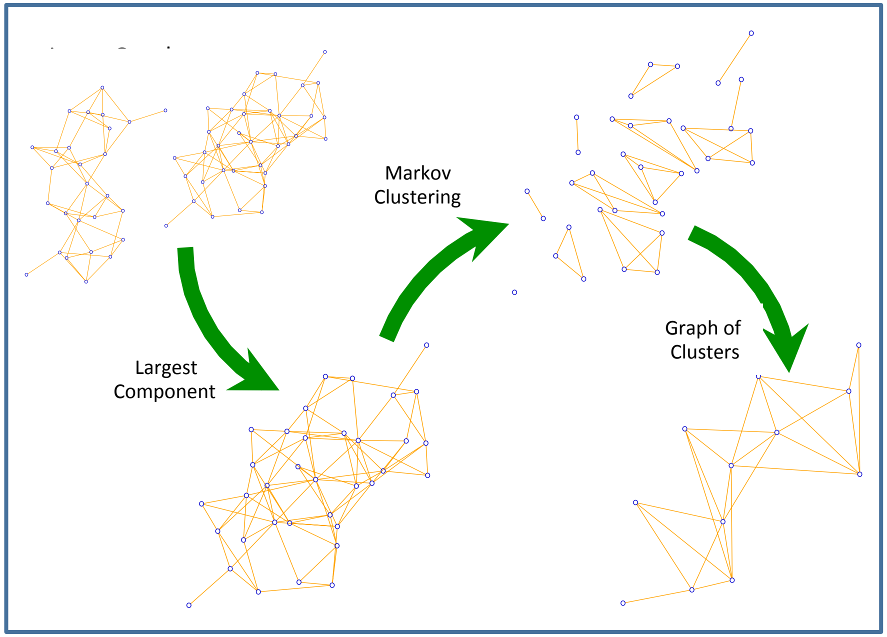




- KDT is higher level (graph abstractions)
- Combinatorial BLAS is for performance

Domain expert vs. Graph expert

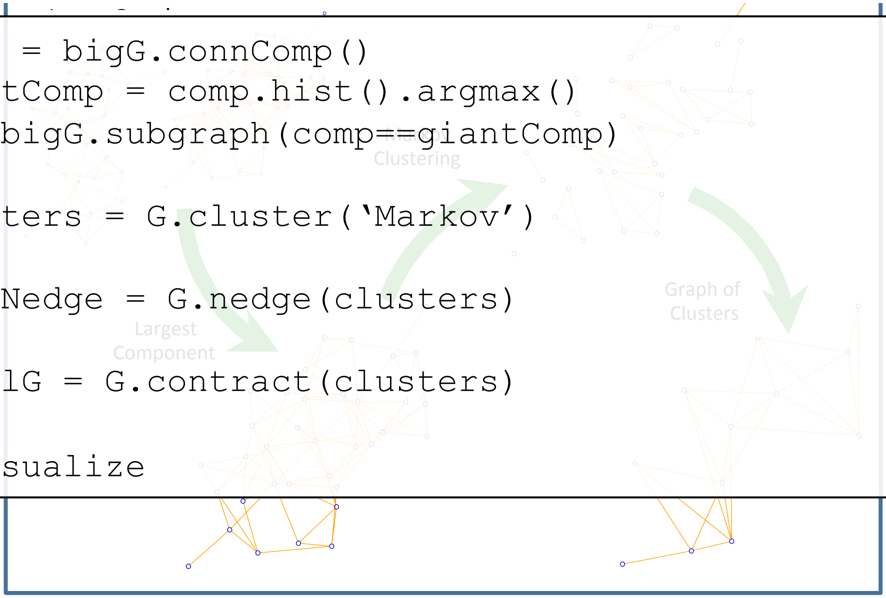
- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering / components
- Centrality / authority: betweenness centrality, PageRank



- 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



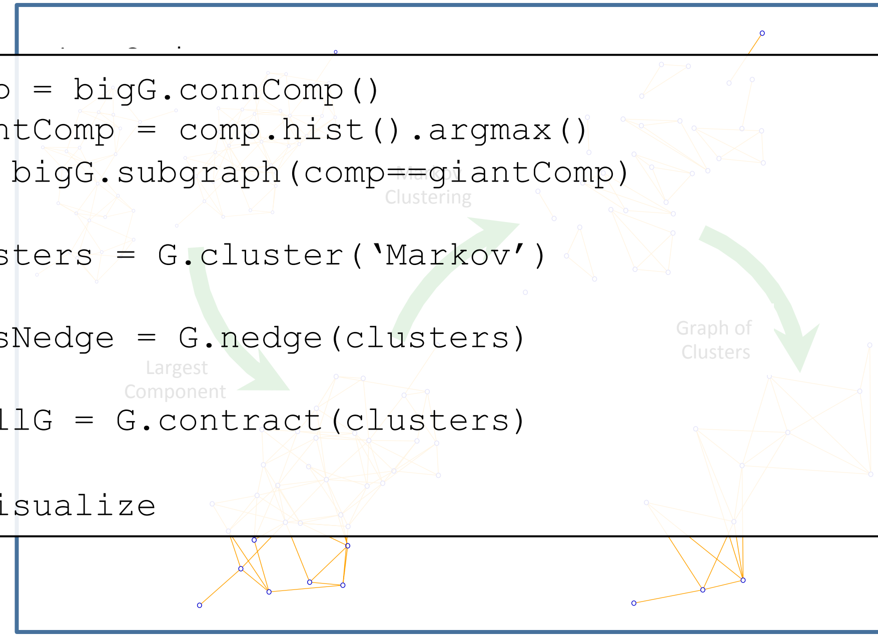

```
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

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



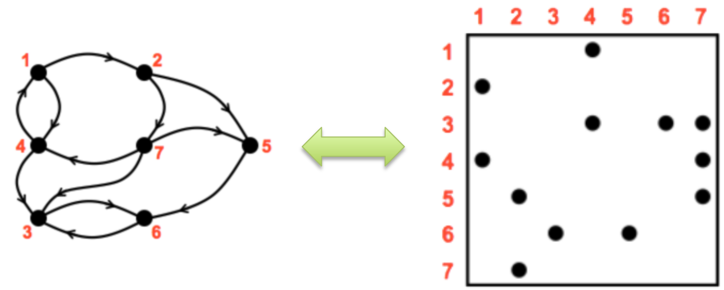

```
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

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)
- V0.2 released March 2012

A few KDT applications

Markov Clustering

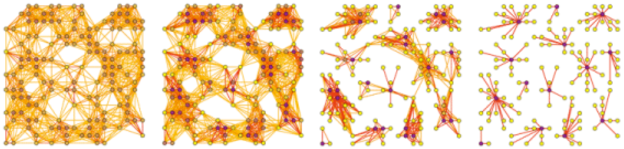
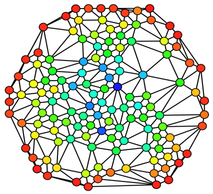


image courtesy Stijn van Dongen

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



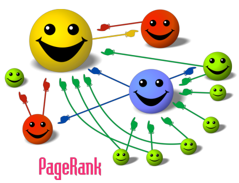
$$C_B(v) = \sum_{\substack{s \neq v \neq t \in V \\ s \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

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.

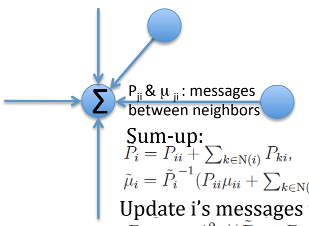


courtesy Felipe Micaroni Lalli

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



Sum-up:

$\tilde{P}_i = P_{ii} + \sum_{k \in \mathrm{N}(i)} P_{ki},$

$\tilde{\mu}_i = \tilde{P}_i^{-1}(P_{ii}\mu_{ii} + \sum_{k \in \mathrm{N}(i)} P_{ki}\mu_{ki}), \forall i$

Update i's messages to its neighbors

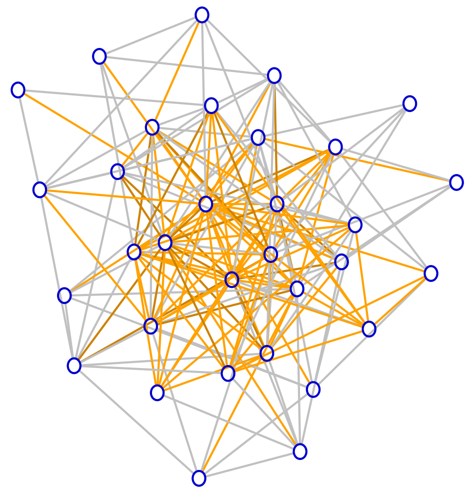
$P_{ij} = -A_{ij}^2/(\tilde{P}_i - P_{ji}),$

$\mu_{ij} = (\tilde{P}_i\tilde{\mu}_i - P_{ji}\mu_{ji})/A_{ij}.$

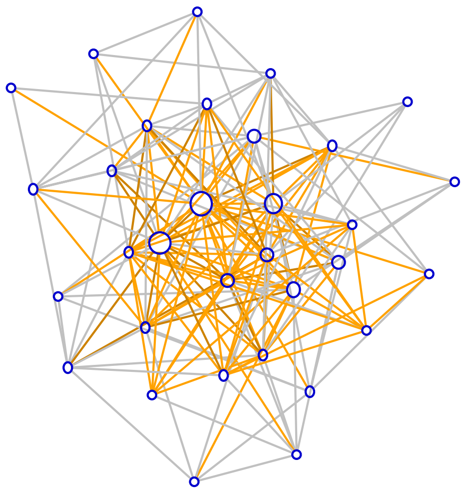
Gaussian belief propagation (GaBP) is an iterative algorithm for solving the linear system of equations Ax = b, where A is symmetric positive definite.
GaBP assumes each variable follows a normal distribution. It iteratively calculates the precision P and mean value μ of each variable; the converged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

The need for filters

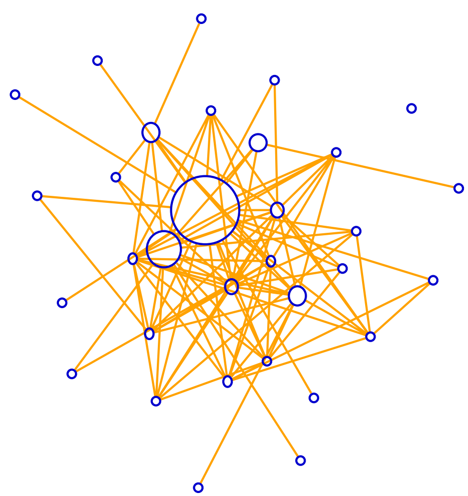
Graph of text & phone calls



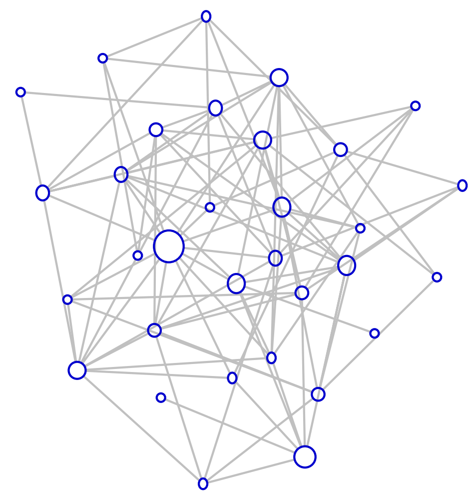
Betweenness centrality



Betweenness centrality on text messages



Betweenness centrality on phone calls



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')
```

Filter options and implementation

- Filter defined as unary predicates, checked in order they were added
- **Each KDT object** maintains **a stack of filter predicates**
- All operations respect filters, enabling **filter-ignorant algorithm design**

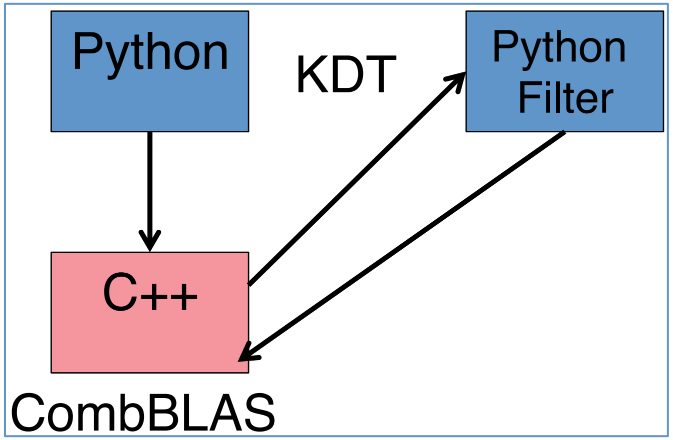
On-the-fly filters	Materialized filters
Edges are retained	Edges are pruned on copy
Check predicate on each edge/vertex traversal	Check predicate once on materialization
Cheap but done on each run	Expensive but done once

Outline

- Motivation
- Sparse matrices for graph algorithms
- CombBLAS: sparse arrays and graphs on parallel machines
- KDT: attributed semantic graphs in a high-level language
- Specialization: getting the best of both worlds

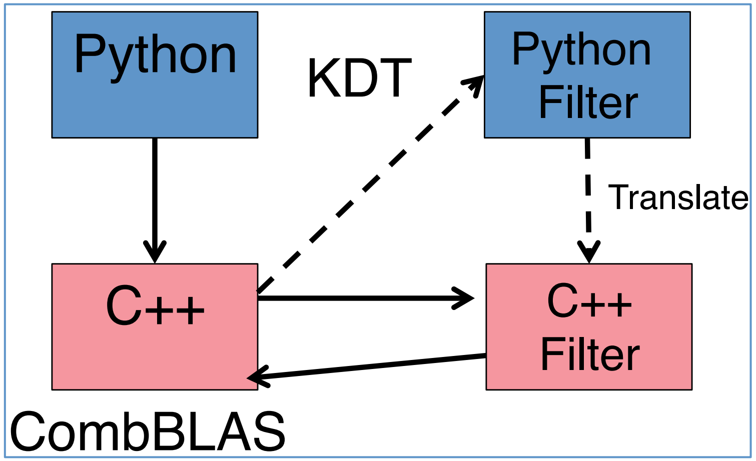
UCSB

On-the-fly filter performance issues



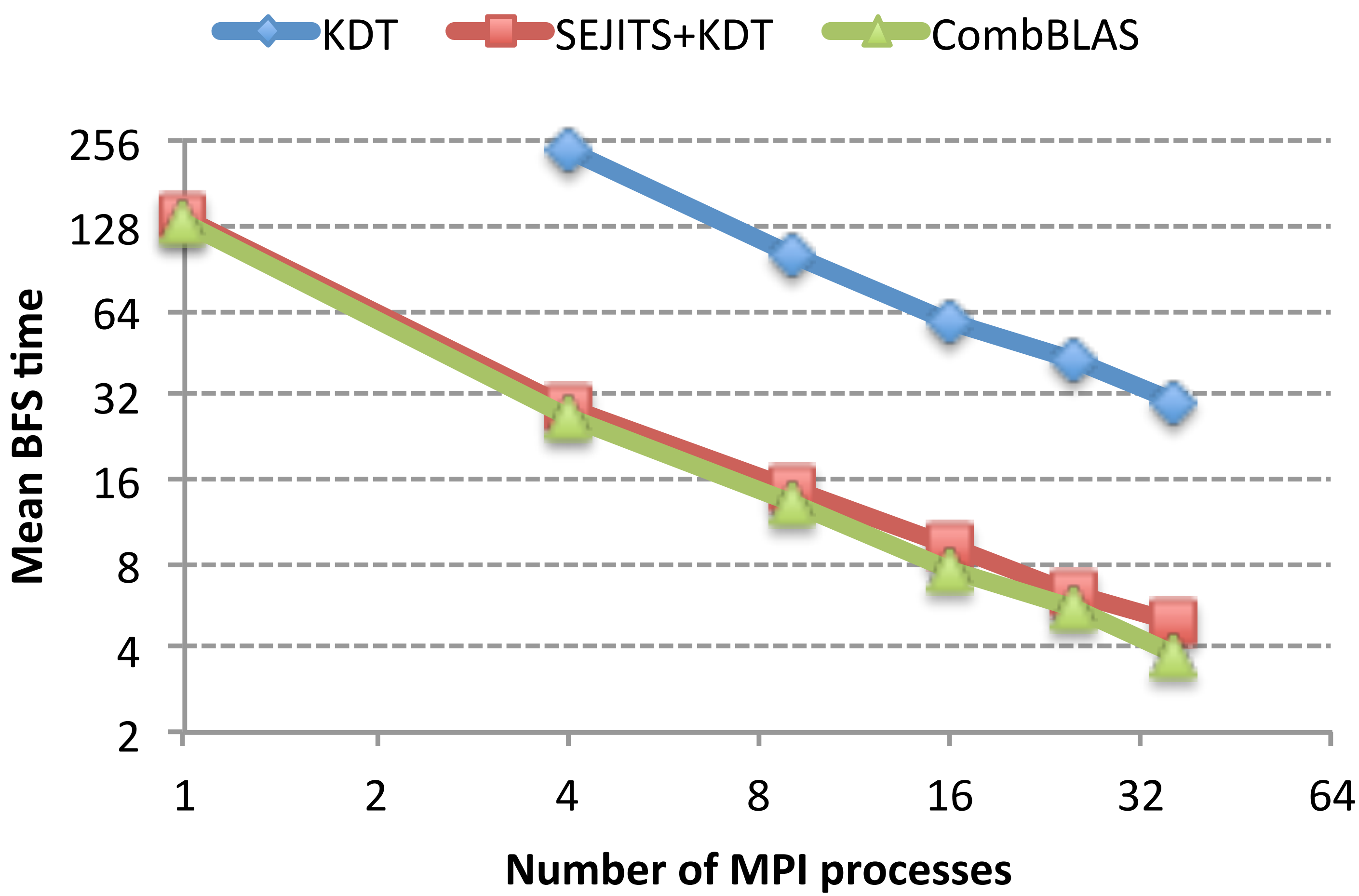
- Write filters as semiring ops in C, wrap in Python
  - Good performance but hard to write new filters
- Write filters in Python and call back from CombBLAS
  - Flexible and easy but runs slow
- The issue is local computation, not parallelism or comms
- All user-written semirings face the same issue.

Solution: Just-in-time specialization (SEJITS)



- On first call, translate Python filter or semiring op to C++
- Compile with GCC
- Call the compiled code thereafter
- (Lots of details omitted ….)

Filtered 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.

Conclusion

- Sparse arrays and matrices supply useful primitives, and algorithms, for high-performance graph computation.
- A CSC moral: Things are always clearer when you look at them from two directions.
- Just-in-time specialization is key to performance of flexible user-programmable graph analytics.

kdt.sourceforge.net/

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

UCSB