Berkeley
UNIVERSITY OF CALIFORNIA

Sparse Matrices Beyond Solvers - Graphs, Biology, and Machine Learning

Aydın Buluç
Computational Research Division, LBNL
EECS Department, UC Berkeley

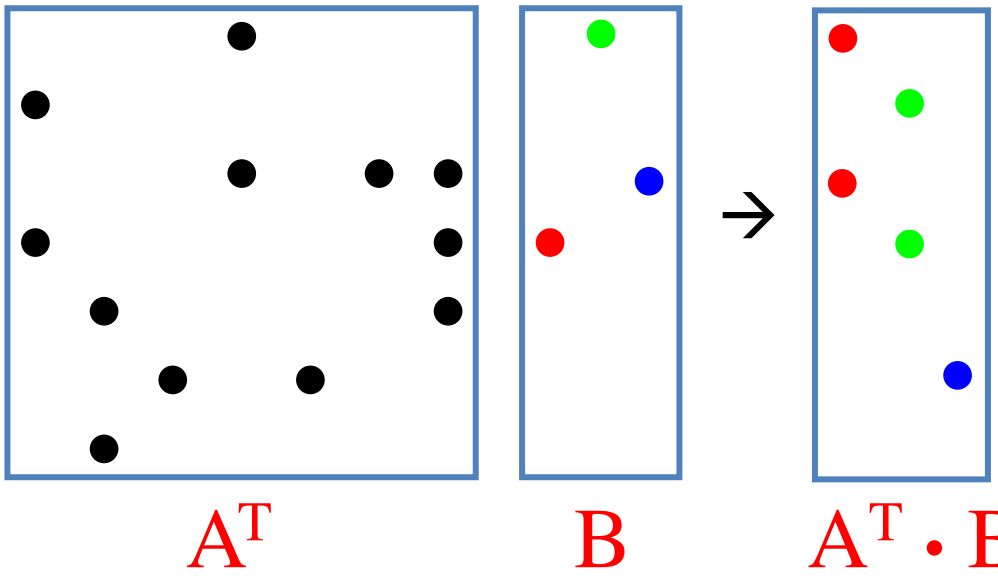
Fast Code Seminar, MIT
June 22, 2020

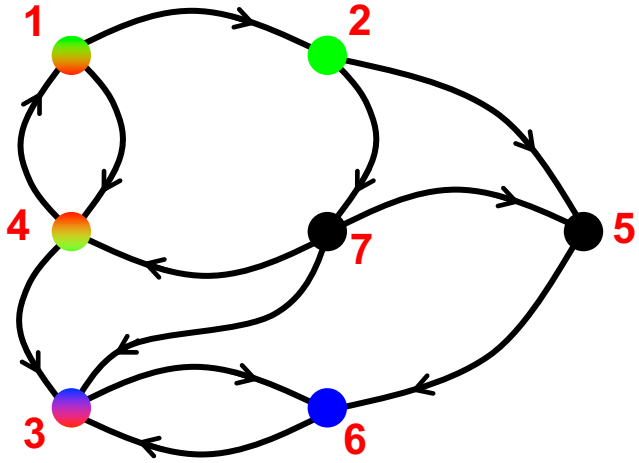
Sparse Matrices

"I observed that most of the coefficients in our matrices were zero; i.e., the nonzeros were 'sparse' in the matrix, and that typically the triangular matrices associated with the forward and back solution provided by Gaussian elimination would remain sparse if pivot elements were chosen with care"

- Harry Markowitz, describing the 1950s work on portfolio theory that won the 1990 Nobel Prize for Economics

Graphs in the language of matrices

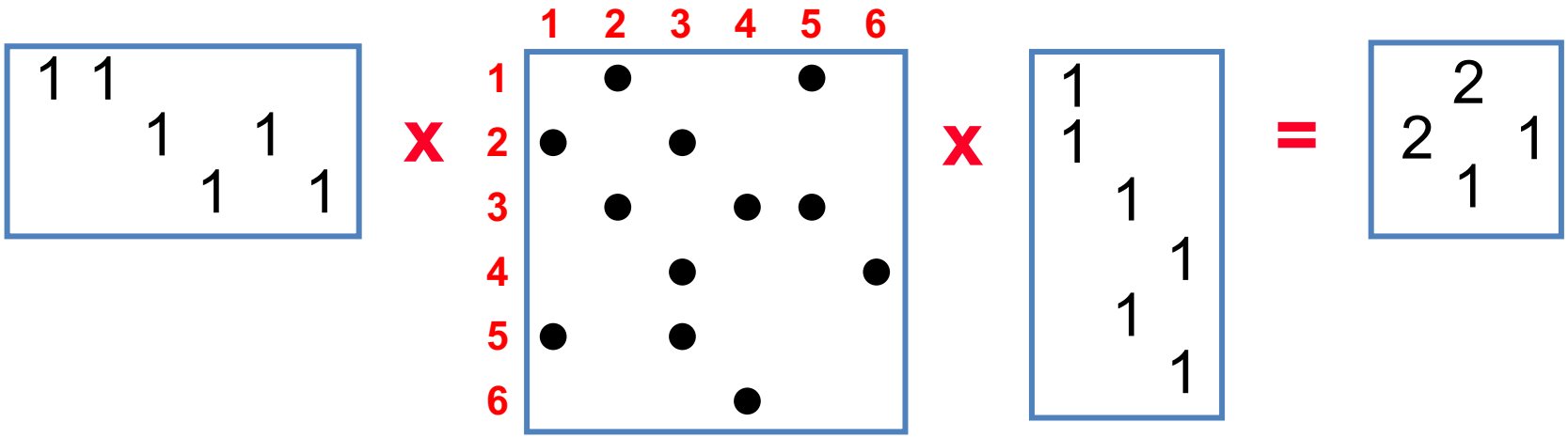


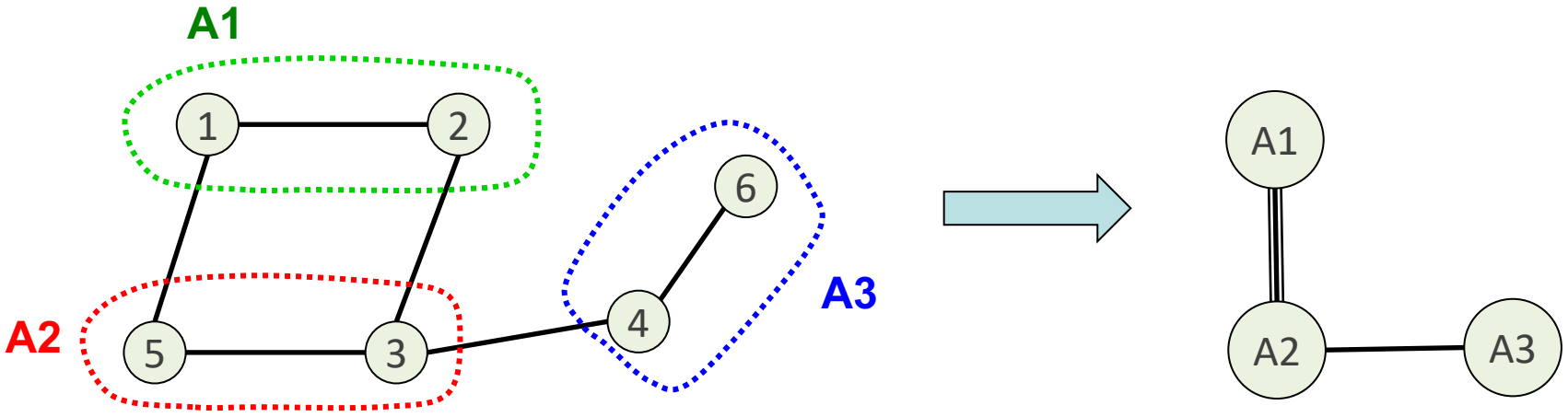


- Sparse array representation => space efficient
- Sparse matrix-matrix multiplication => work efficient
- Three possible levels of parallelism: searches, vertices, edges
- Highly-parallel implementation for Betweenness Centrality*

*: A measure of influence in graphs, based on shortest paths

Graph coarsening via sparse matrix-matrix products





Aydin Buluç and John R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal of Scientific Computing (SISC), 2012.*

The GraphBLAS effort

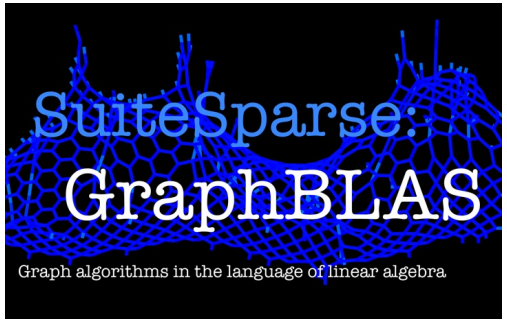
Standards for Graph Algorithm Primitives

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.

- The GraphBLAS Forum: http://graphblas.org
- Graphs: Architectures, Programming, and Learning (GrAPL @IPDPS): http://hpc.pnl.gov/grapl/

SuiteSparse::GraphBLAS



- From Tim Davis (Texas A&M)
- First conforming implementation of C API
- Features [1]:
  - 960 semirings built in; also user-defined semirings
  - Fast incremental updates using non-blocking mode and “zombies”
  - Several sparse data structures & polyalgorithms under the hood
- Already multithreaded [2]
- Performance on graph benchmarks (e.g. triangles, k-truss) comparable to highly-tuned custom C code
- Included in Debian and Ubuntu Linux distributions
- Used as computational engine in commercial RedisGraph product

[1] Davis, Timothy A. "Algorithm 1000: SuiteSparse: GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra." ACM Transactions on Mathematical Software (TOMS) 45.4 (2019): 44.
[2] Aznaveh, Mohsen, et al. "Parallel GraphBLAS with OpenMP." CSC20, SIAM Workshop on Combinatorial Scientific Computing. SIAM. 2020.

GraphBLAS C API Spec (http://graphblas.org)

- **Goal:** A crucial piece of the GraphBLAS effort is to translate the mathematical specification to an actual Application Programming Interface (API) that
  i. is faithful to the mathematics as much as possible, and
  ii. enables efficient implementations on modern hardware.
- **Impact:** All graph and machine learning algorithms that can be expressed in the language of linear algebra
- **Innovation:** Function signatures (e.g. mxm, vxm, assign, extract), parallelism constructs (blocking v. non-blocking), fundamental objects (masks, matrices, vectors, descriptors), a hierarchy of algebras (functions, monoids, and semiring)

```
GrB_info GrB_mxm(GrB_Matrix          *C,      // destination
                 const GrB_Matrix     Mask,
                 const GrB_BinaryOp   accum,
                 const GrB_Semiring   op,
                 const GrB_Matrix     A,
                 const GrB_Matrix     B
                [, const Descriptor   desc]);
```

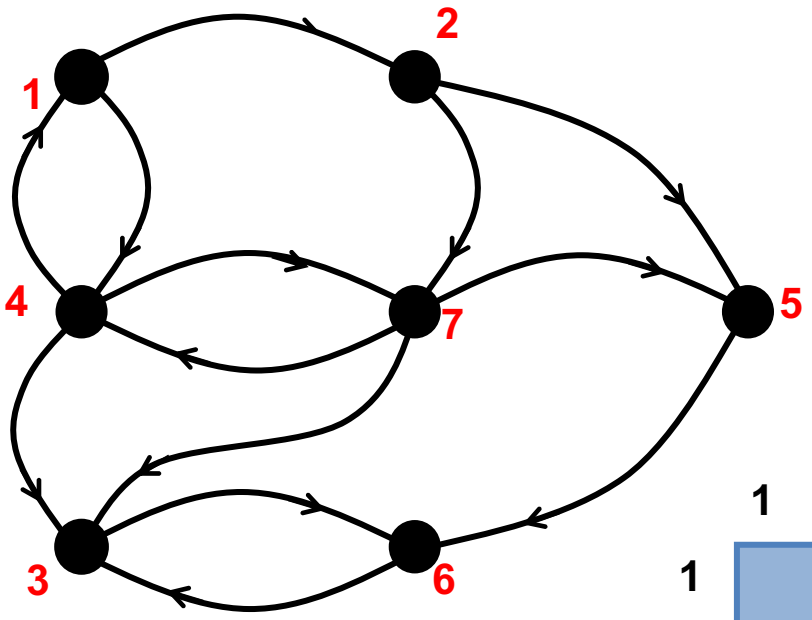
$$C(\neg M) \oplus= A^T \oplus.\otimes B^T$$

A.Buluç, T. Mattson, S. McMillan, J. Moreira, C. Yang. "The GraphBLAS C API Specification", version 1.3.0

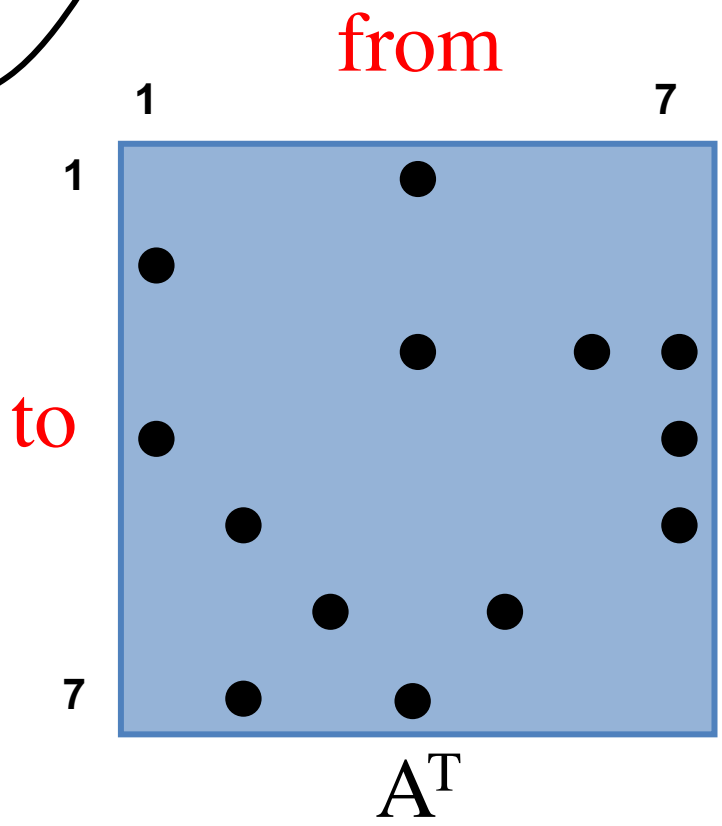
Examples of semirings in graph algorithms

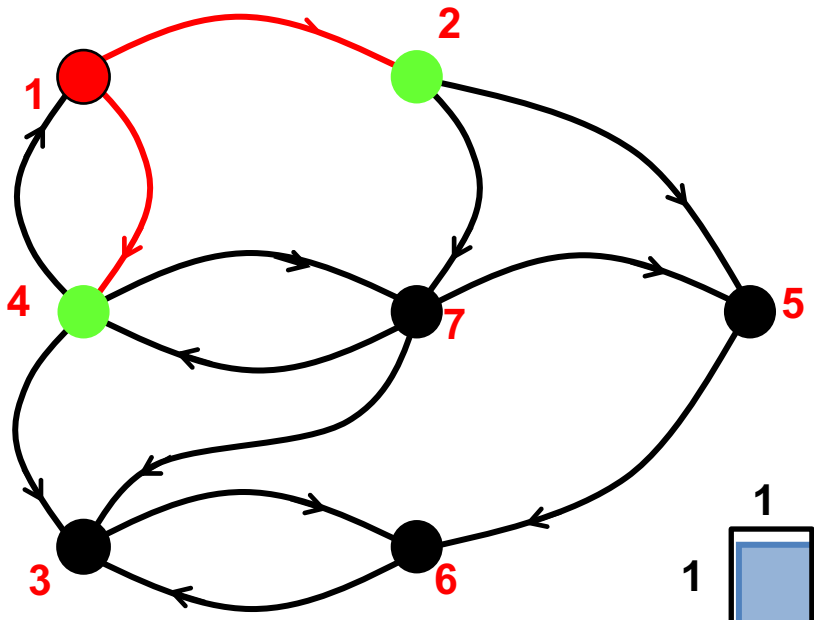
Real field: **(R, +, x)**	Classical numerical linear algebra
Boolean algebra: **({0 1}, |, &)**	Graph connectivity
Tropical semiring: **(R U {∞}, min, +)**	Shortest paths
(S, select, select)	Select subgraph, or contract nodes to form quotient graph
(edge/vertex attributes, vertex data aggregation, edge data processing)	Schema for user-specified computation at vertices and edges
(R, max, +)	Graph matching &network alignment
(R, min, times)	Maximal independent set

- **Shortened semiring notation: (Set, Add, Multiply)**. Both identities omitted.
- **Add:** Traverses edges, **Multiply:** Combines edges/paths at a vertex
- Neither add nor multiply needs to have an inverse.
- Both **add** and **multiply** are **associative**, **multiply distributes** over **add**



Breadth-first search in the language of matrices

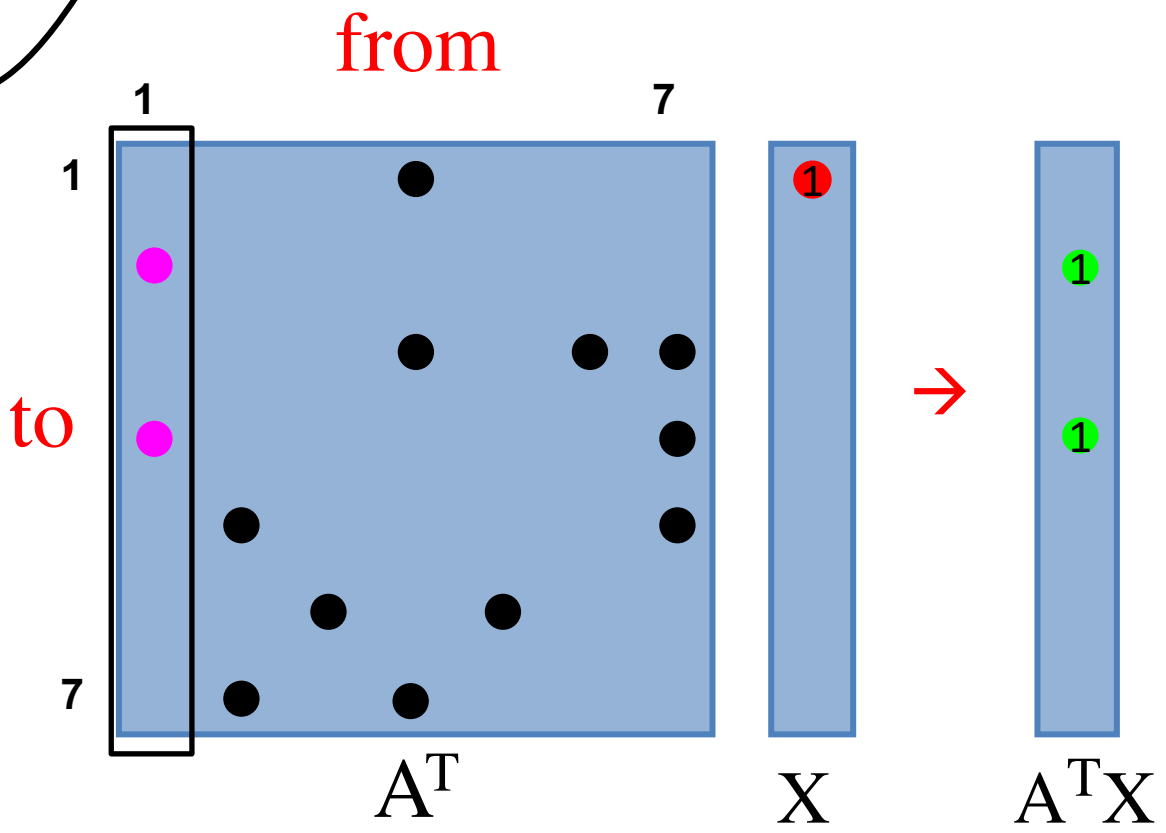




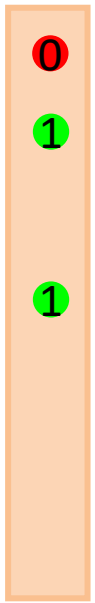
Particular semiring operations:

Multiply: select2nd

Add: minimum



parents:



Input sparsity

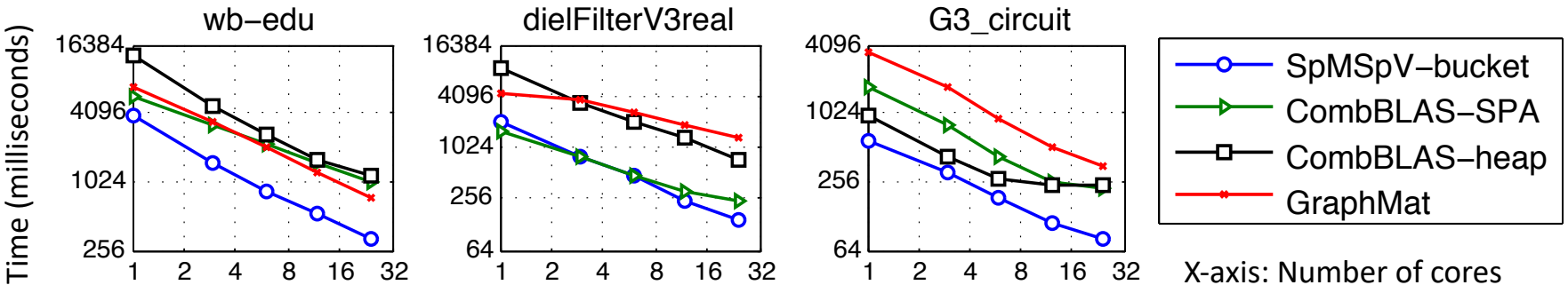
- What was the cost of that $A^T x$ in the previous slide?
- If x is dense, it is O(nnz(A)) = O(m) where m=#edges
- **If x is sparse**, it is

$$\sum_{i:x_i \neq 0} nnz(A_{i:})$$

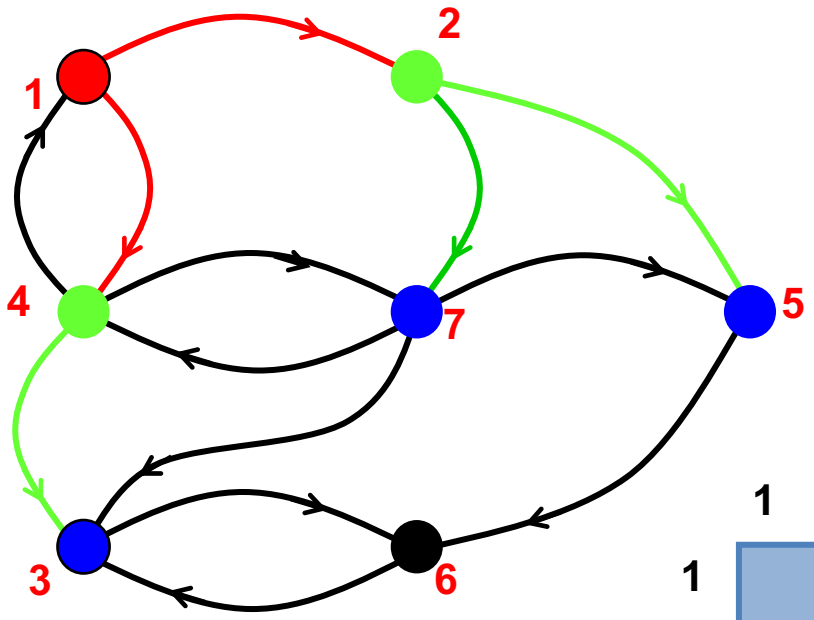
- ***Over all iterations*** of BFS, the cost sums up to O(nnz(A)), because no x_i appears twice in the input.
- Note that this is ***optimal*** for conventional (top-down) BFS
- Many people outside the community miss this observation and ***mistakenly think*** SpMV based BFS is suboptimal by a factor of the graph diameter.

A work-efficient parallel algorithm for sparse matrix-sparse vector multiplication (SpMSpV)

- **Moral:** You should use this algorithm for exploiting input (vector) sparsity in SpMV in multicore and many-core architectures
- **Algorithmic innovation:**
  - Attains work-efficiency by arranging necessary columns of the matrix into buckets where each bucket is processed by a single thread
  - Avoids synchronization by row-wise partitioning of the matrix on the fly
- **Performance:**
  - First ever work-efficient algorithm for SpMSpV that attains up to 15x speedup on a 24-core Intel Ivy Bridge processor and up to 49x speedup on a 64-core KNL processor
  - Up to an order of magnitude faster than its competitors, especially for sparser vector

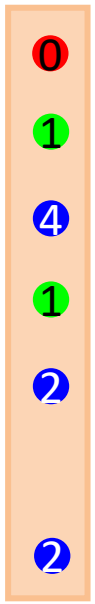


A.Azad, A. Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. IPDPS'17



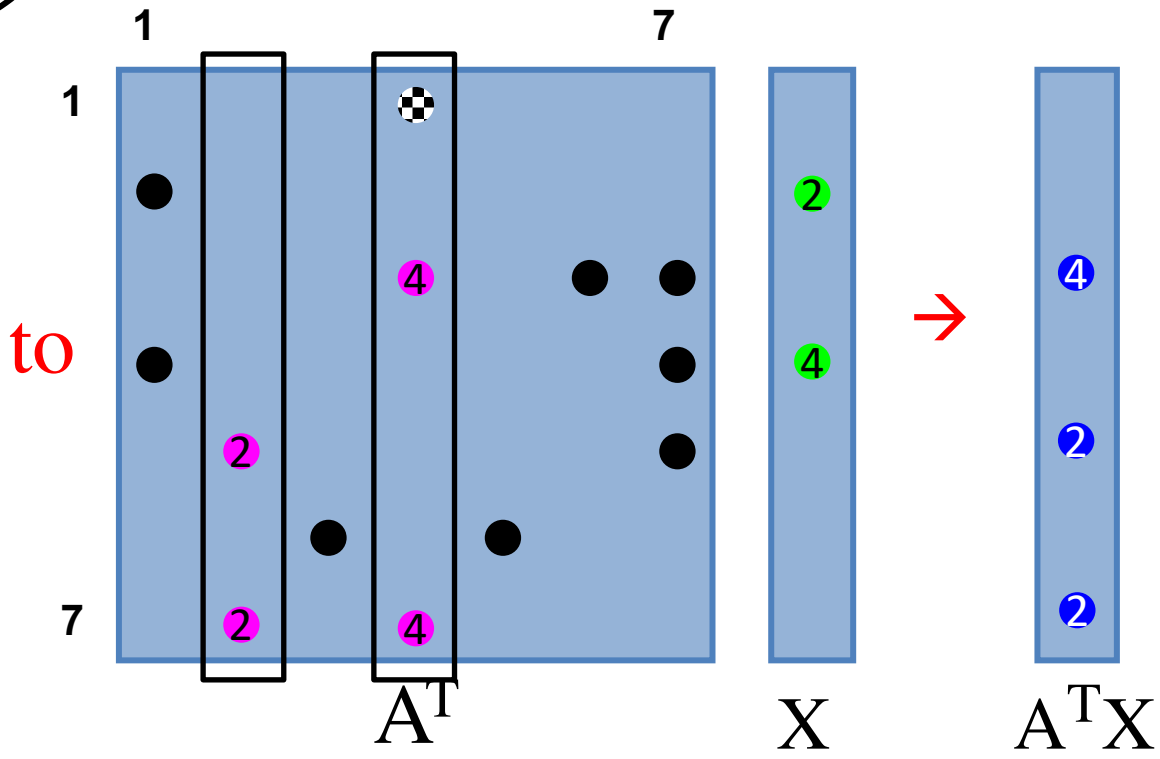
Select vertex with
minimum label as parent

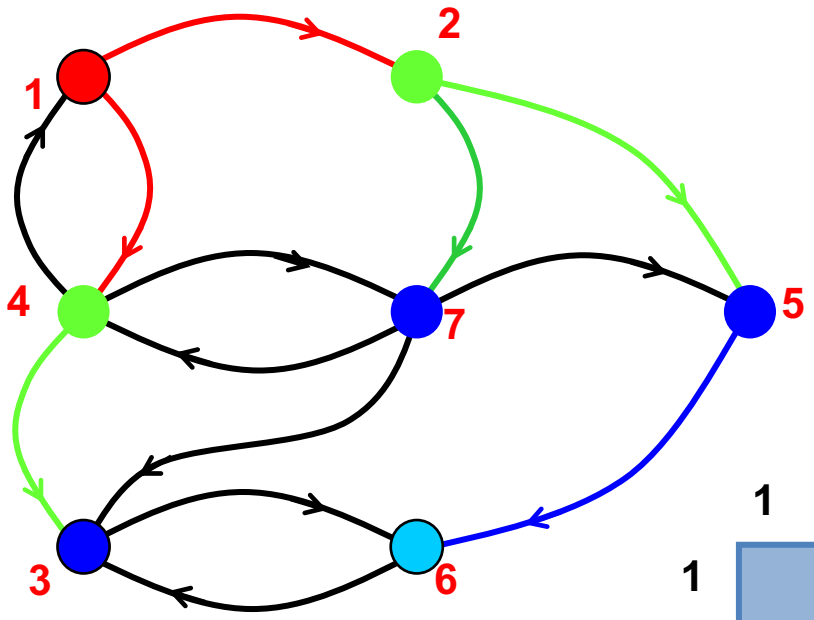
parents:



from

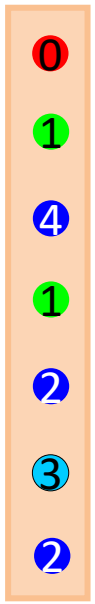
to

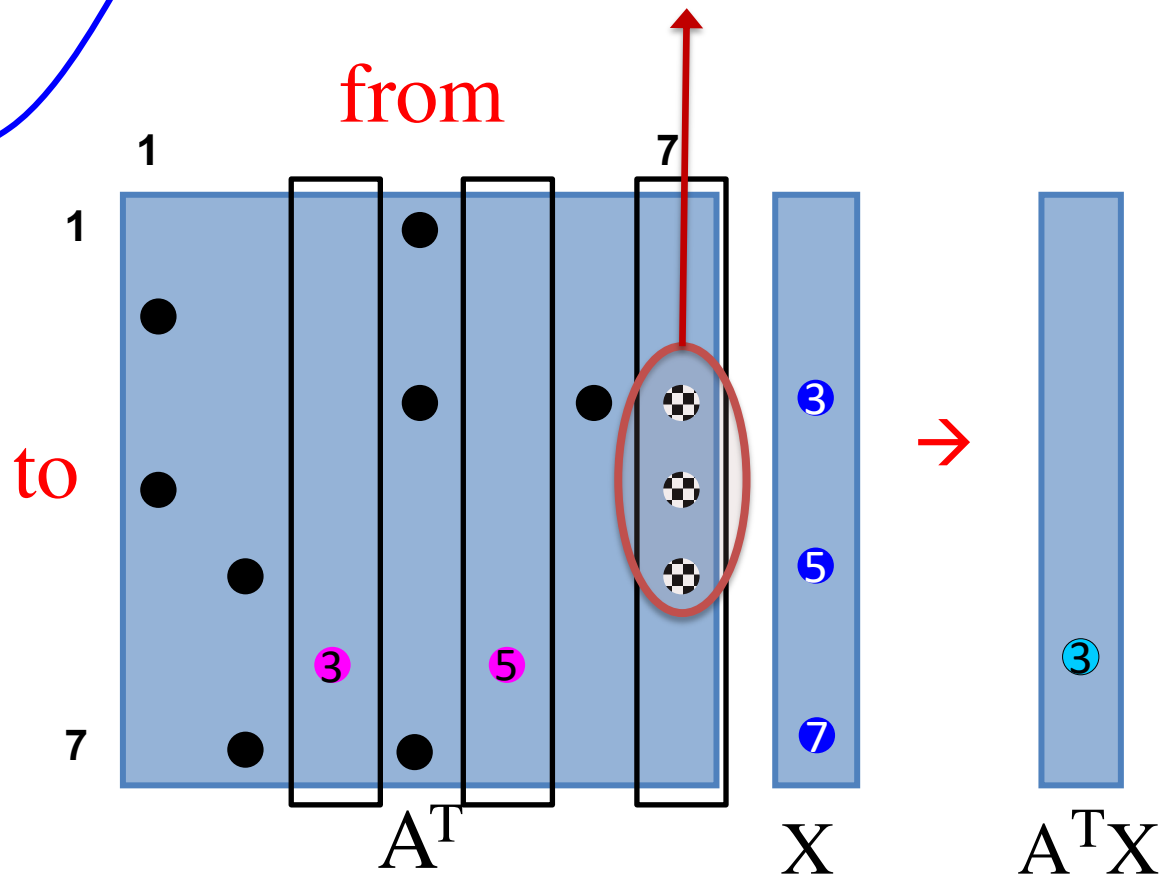


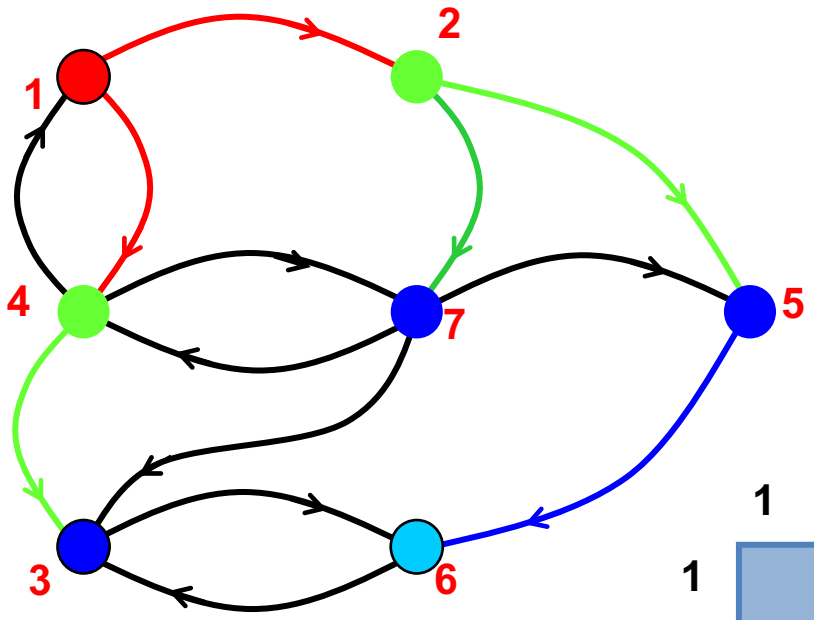


- Masks avoid formation of temporaries and can enable automatic direction optimization
- These footballs are nonzeros that are ***masked out*** by the parents array

parents:

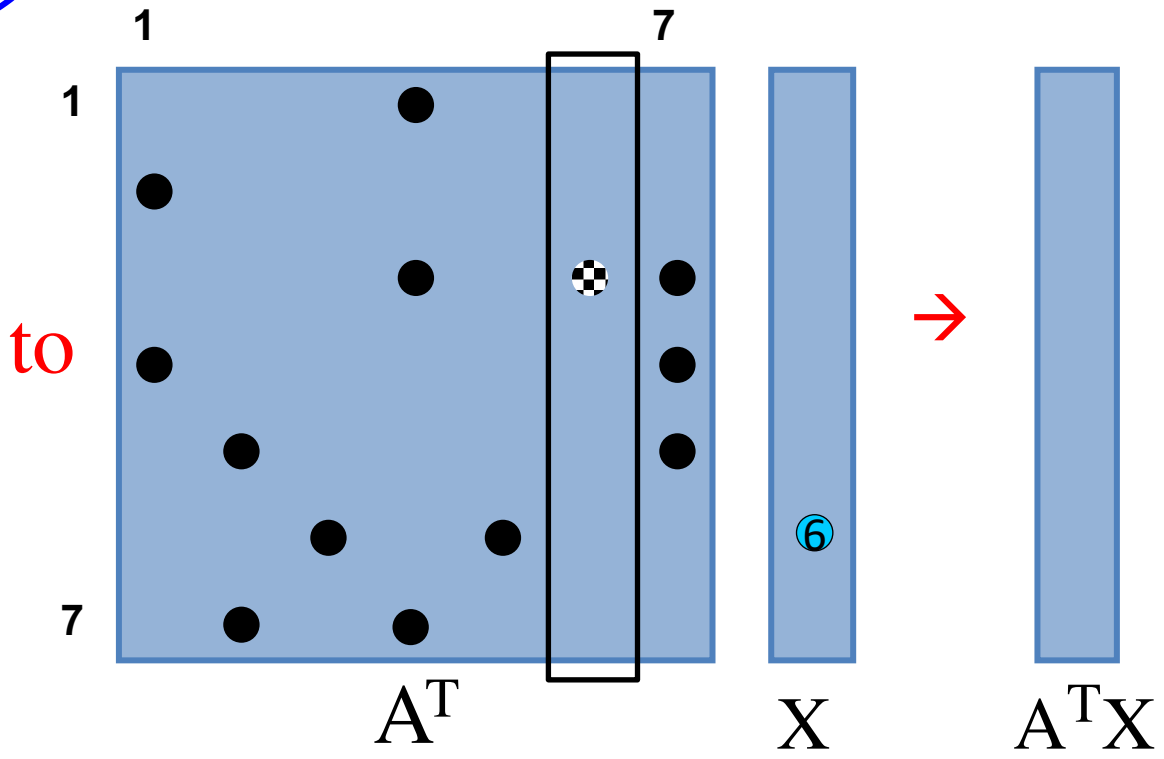






from

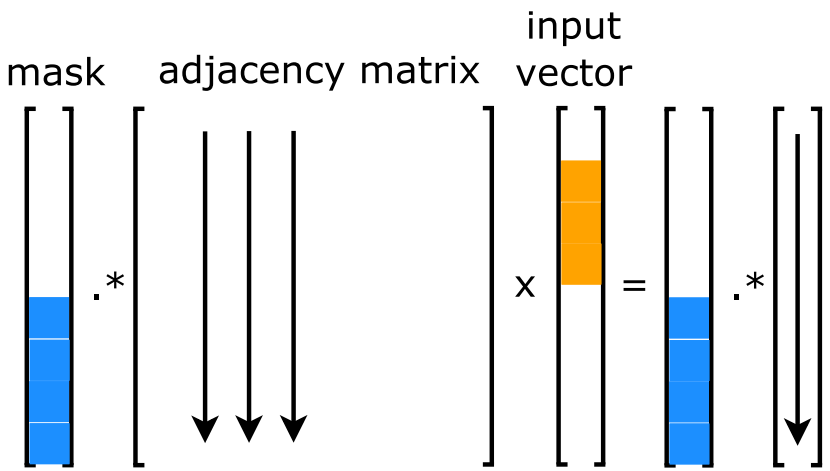
to



Output sparsity via masks

- The actual operation is $x = A^T x$.* p

  p is the parents array and .* is elementwise multiplication
- At first, our vision was limited: we only thought about eliminating temporaries in GrB_mxv
- But it was important enough to motivate the inclusion of masks into the GraphBLAS spec, though in limited form



Column-based matvec w/ mask

Idea was to run the same column-based algorithm, but checking against a mask before writing to output

BFS in GraphBLAS with Masks

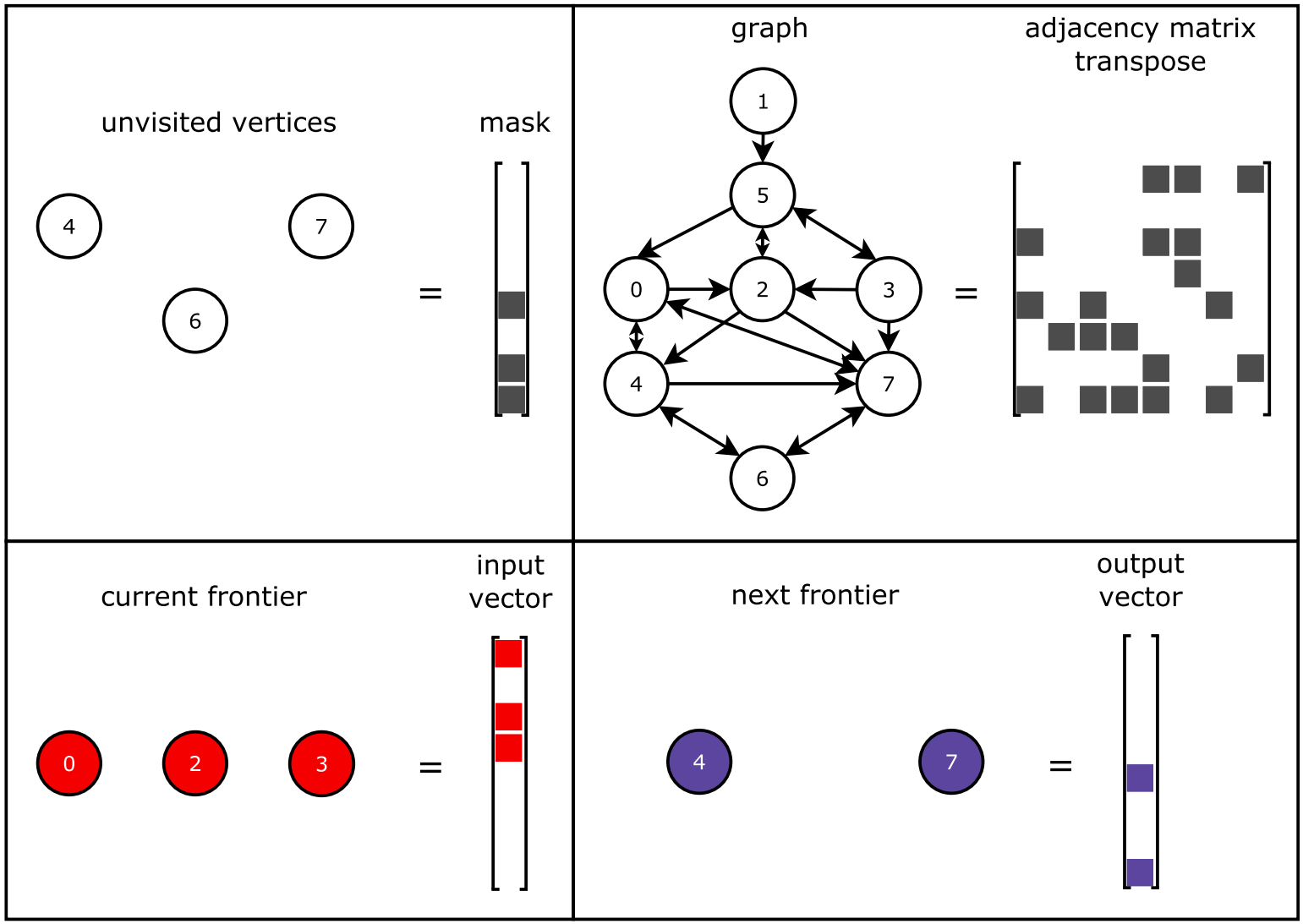
```
GrB_Vector q;                                   // vertices visited in each level
GrB_Vector_new(&q,GrB_BOOL,n);                  // Vector<bool> q(n) = false
GrB_Vector_setElement(q,(bool)true,s);          // q[s] = true, false everywhere else

GrB_Monoid Lor;                                 // Logical-or monoid
GrB_Monoid_new(&Lor,GrB_LOR,false);

GrB_Semiring Boolean;                           // Boolean semiring
GrB_Semiring_new(&Boolean,Lor,GrB_LAND);

GrB_Descriptor desc;                            // Descriptor for vxm
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP);     // invert the mask
GrB_Descriptor_set(desc,GrB_OUTP,GrB_REPLACE);// clear the output before assignment

GrB_UnaryOp apply_level;
GrB_UnaryOp_new(&apply_level,return_level,GrB_INT32,GrB_BOOL);

/*
 * BFS traversal and label the vertices.
 */
level = 0;
GrB_Index nvals;
do {
  ++level;                                      // next level (start with 1)
  GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,apply_level,q,GrB_NULL);  // v[q] = level
  GrB_vxm(q,*v,GrB_NULL,Boolean,q,A,desc);      // q[!v] = q ||.&& A ; finds all the
                                                // unvisited successors from current q
  GrB_Vector_nvals(&nvals, q);
} while (nvals);                                // if there is no successor in q, we are done.
```

Push-pull ≡ column-row matvec

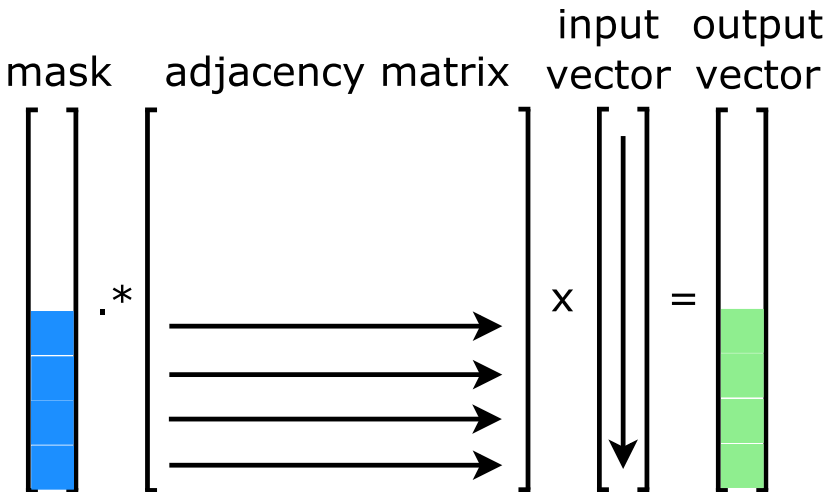
This is a story on how languages (and in this case APIs) change our thinking and drive our creative process

- Carl Yang and I pondered quite a bit on whether it was possible to implement direction optimization in the language of matrices *
- Push-pull (also known as direction optimization) was just about running a row- vs. column-based matvec
- But it wouldn't be competitive it its pure form because you were pulling from every vertex, not just unexplored ones.
- A year or so later, GraphBLAS had "masks"
- Now it was totally obvious how to make push-pull competitive in GraphBLAS

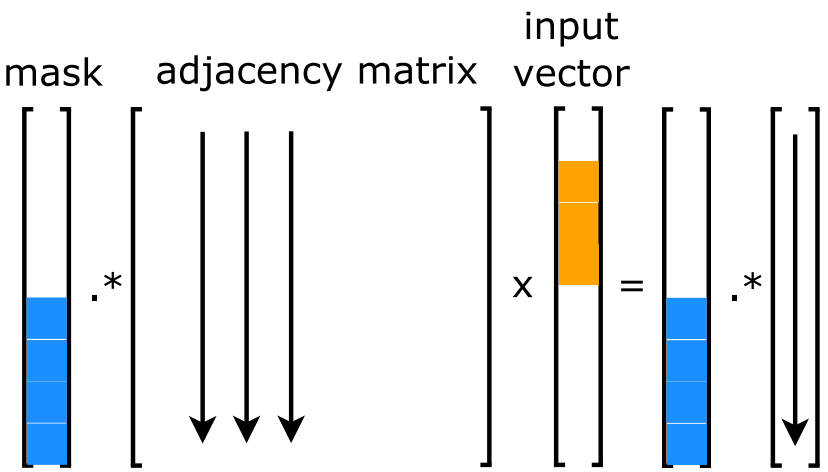
Enter “masks”


unvisited vertices
mask
4
7
6
=
graph
adjacency matrix
transpose
1
5
0
2
3
4
7
6
=
current frontier
input
vector
0
2
3
=
next frontier
output
vector
4
7
=

Masks make “pull” implementable competitively in GraphBLAS



Row-based matvec w/ mask



Column-based matvec w/ mask

- **Pull** is better for sufficiently sparse masks; **push** otherwise
- **Claim**: “direction optimization” would have been discovered automatically by the GraphBLAS runtime if we designed the interface back half a decade ago.

Yang, C., Buluc, A. and Owens, J.D., Implementing Push-Pull Efficiently in GraphBLAS. *ICPP’18*

GraphBLAST

- First “high-performance” GraphBLAS implementation on the GPU
- Optimized to take advantage of both input and output sparsity
- Automatic direction-optimization through the use of masks
- Competitive with fastest GPU (Gunrock) and CPU (Ligra) codes
- Outperforms multithreaded SuiteSparse::GraphBLAS

Design principles:

1. Exploit input sparsity => direction-optimization
2. Exploit output sparsity => masking
3. Proper load-balancing => key for GPU implementations

Extensively evaluated on (more implemented, google for github repo)

- Breadth-first-search (BFS)
- Single-source shortest-path (SSSP)
- PageRank (PR)
- Triangle counting (TC)

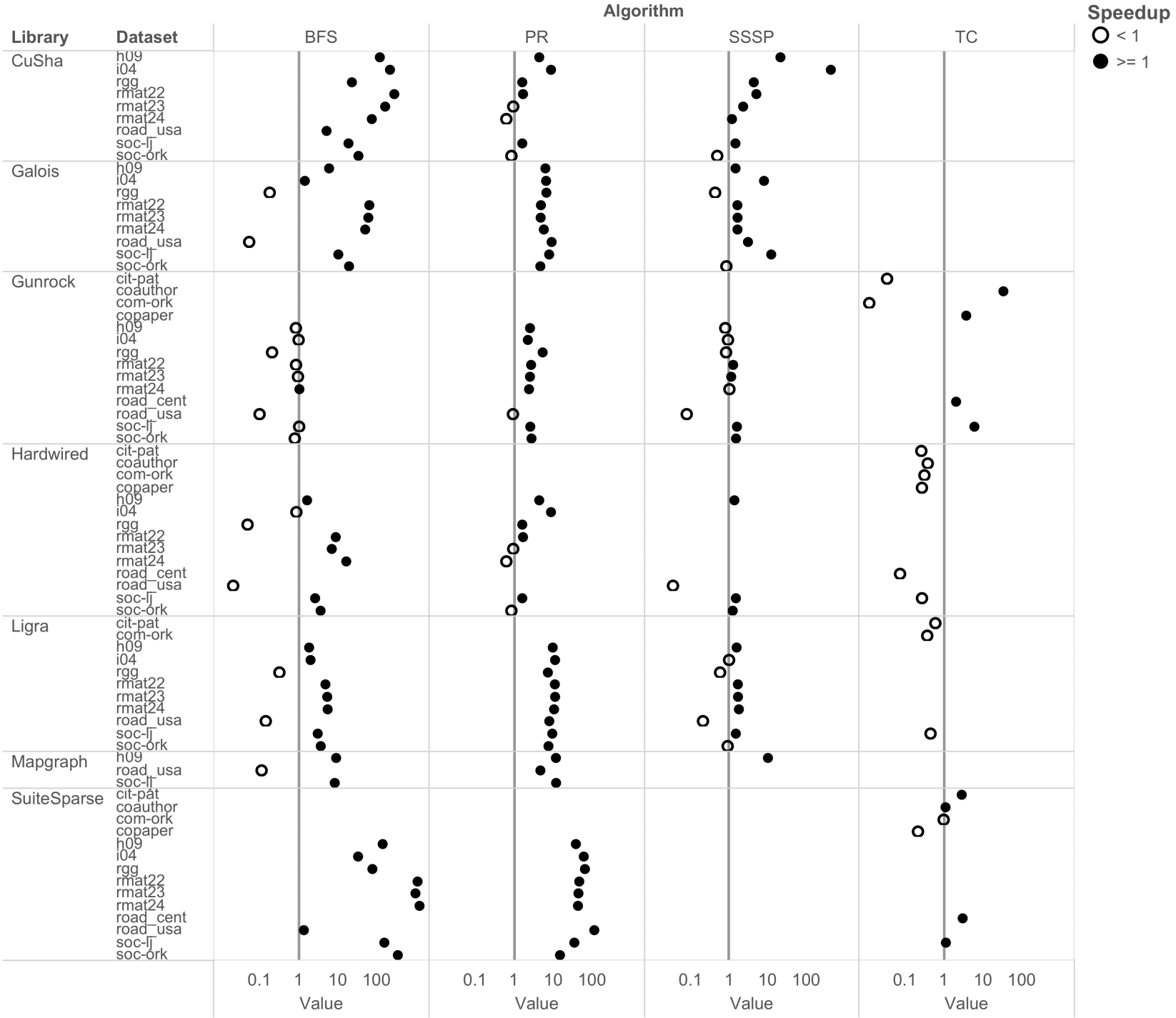
https://github.com/gunrock/graphblast

Yang, B., Owens, “GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU”, arXiv

GraphBLAST syntax

```cpp
#include <graphblas/graphblas.hpp>

void bfs(Vector<float>*       v,
         const Matrix<float>* A,
         Index                s,
         Descriptor*          desc) {
  Index A_nrows;
  A->nrows(&A_nrows);
  float d = 1.f;

  Vector<float> f1(A_nrows);
  Vector<float> f2(A_nrows);
  std::vector<Index> indices(1, s);
  std::vector<float> values(1, 1.f);
  f1.build(&indices, &values, 1, GrB_NULL);

  v->fill(0.f);
  float c = 1.f;
  while (c > 0) {
    // Assign level d at indices f1 to visited vector v
    graphblas::assign(v, &f1, GrB_NULL, d, GrB_ALL, A_nrows, desc);
    // Set mask to use structural complement (negation)
    desc->toggle(GrB_MASK);
    // Multiply frontier f1 by transpose of matrix A using visited vector v as mask
    // Semiring: Boolean semiring (see Table 4)
    graphblas::vxm(&f2, v, GrB_NULL, LogicalOrAndSemiring<float>(), &f1, A, desc);
    // Set mask to not use structural complement (negation)
    desc->toggle(GrB_MASK);
    f2.swap(&f1);
    // Check how many vertices of frontier f1 are active, stop when number reaches 0
    // Monoid: Standard addition (see Table 4)
    graphblas::reduce(&c, GrB_NULL, PlusMonoid<float>(), &f1, desc);
    d++;
  }
}
```

- To avoid have many different descriptors that are different minimally, GraphBLAST introduces the convenience function desc::toggle.
- If the value for field is currently set to default, desc::toggle will set it to the non-default value and vice-versa


Algorithm
Speedup
< 1
>= 1
Library
Dataset
BFS
PR
SSSP
TC
CuSha
Galois
Gunrock
Hardwired
Ligra
Mapgraph
SuiteSparse
h09
i04
rgg
rmat22
rmat23
rmat24
road_usa
soc-lj
soc-ork
cit-pat
coauthor
com-ork
copaper
road_cent
0.1
1
10
100
Value

Kernel methods in Machine Learning

A **kernel** is a function that	Implicitly transforms raw data into high-dimensional feature vectors via a **feature map**; and then	Returns an **inner product** between the feature vectors. Must be **positive-definite.**
A **kernel** is useful for	**Factor out** knowledge on data representation from downstream algorithms,	Exploit **infinite dimensionality and nonlinear** feature spaces.
Kernels are used in	Support vector machine (SVM), Gaussian process regression (GPR), Kernel principal component analysis (kPCA), etc.	

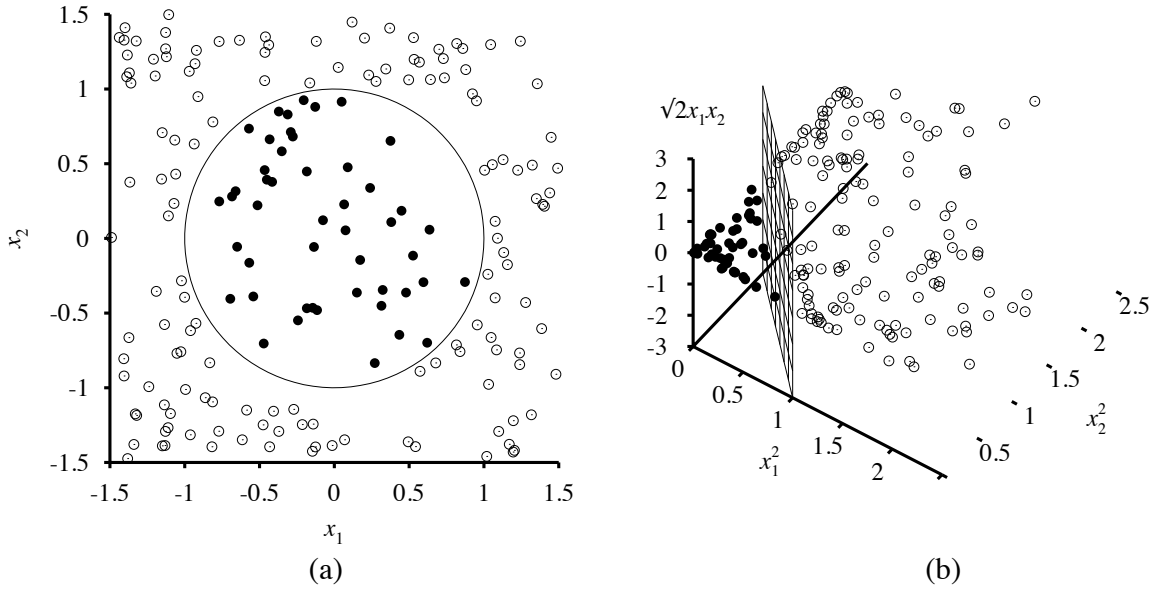
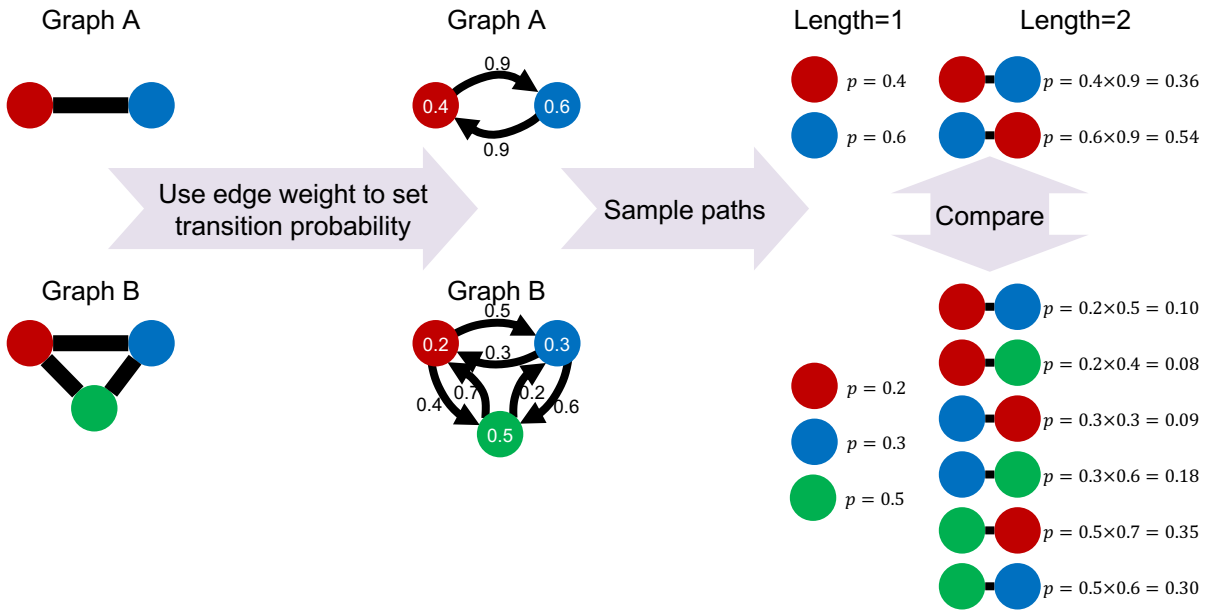


Figure source: Russell & Norvig

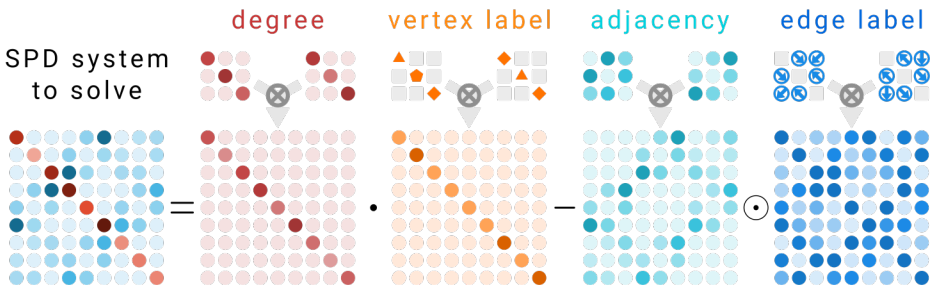
The circular decision boundary in 2D (a) becomes a linear boundary in 3D (b) using the following transformation: $\phi(x_1, x_2) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$

Marginalized Graph Kernels

The inner product between two graphs is the statistical average of the inner product of simultaneous random walk paths on the two graphs.



$$K(G, G') = \mathbf{p}_\times^\mathsf{T} \left(\mathbf{D}_\times \mathbf{V}_\times^{-1} - \mathbf{A}_\times \odot \mathbf{E}_\times\right)^{-1} \mathbf{D}_\times \mathbf{q}_\times$$

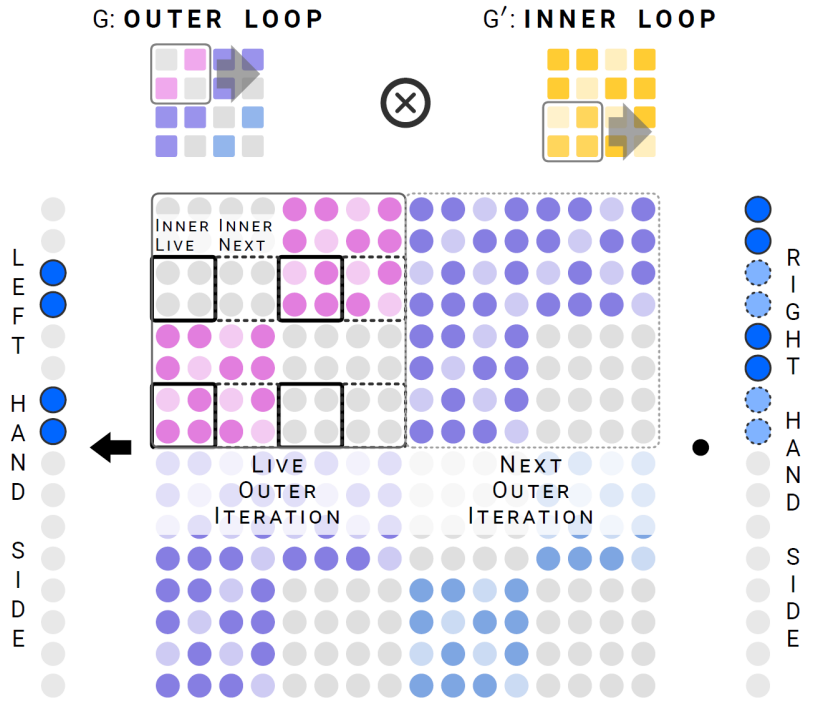


The marginalized graph kernel in linear algebra form represents a modified graph Laplacian

Solving the Graph Kernel PSD system

Streaming Kronecker matrix-vector multiplication

- Regenerates the product linear system on the fly by streaming 8-by-8 tiles.
- Tiles staged in shared memory.
- Trade FLOPS for GB/s, but asymptotic arithmetic complexity stays the same.

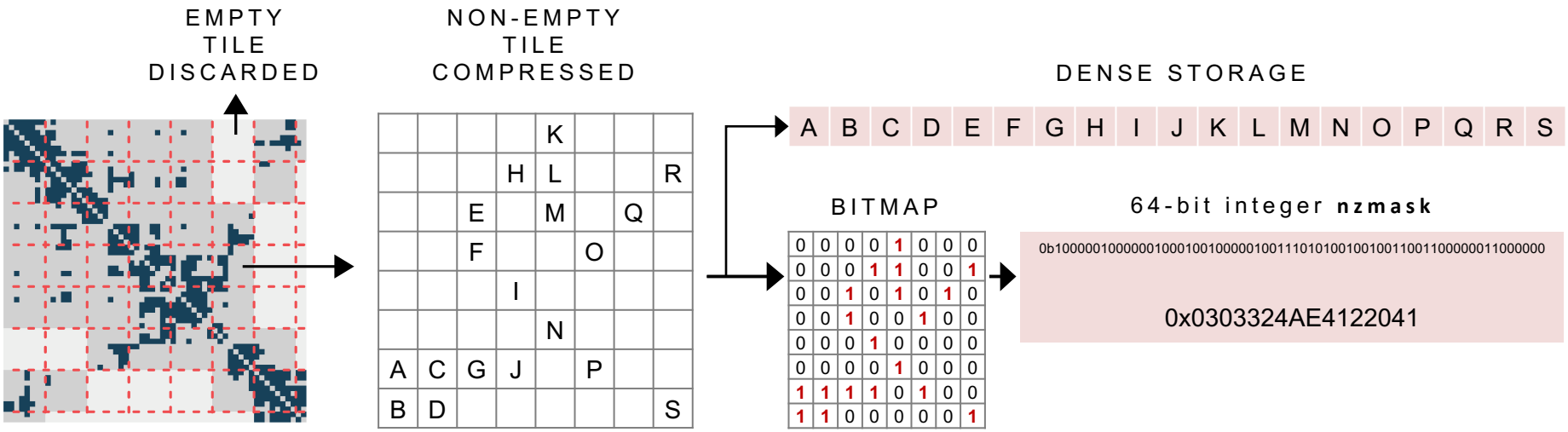



```
1  function CG4GK(d,d′,v,v′,A,A′,E,E′, q,q′)
2      M ← diag[(d ⊗ d′) ⊙ (v ⊗κ v′)^−1]                    |+|
3      x ← 0                                                |+|
4      r ← (d ⊗ d′) · (q ⊗ q′)                              |⋅|
5      z ← v ⊗κ v′                                          |+|
6      p ← z                                                |+|
7      ρ ← r^T z                                            |⋅|
8      repeat
9          a ← (d ⊗ d′) ⊙ (v ⊗κ v′)^−1 · p                  |⋅|
10             +(A ⊗ A′) ⊙ (E ⊗κ E′) · p                    |⋅|
11         α ← ρ/(p^T a)                                    |⋅|
12         x ← x + αp                                       |+|
13         r ← r − αa                                       |+|
14         z ← M^−1 r                                       |+|
15         ρ′ ← r^T z                                       |⋅|
16         β ← ρ′/ρ
17         p ← z + βp                                       |+|
18         ρ ← ρ′
19     until r^T r < ϵ
20     return x
```

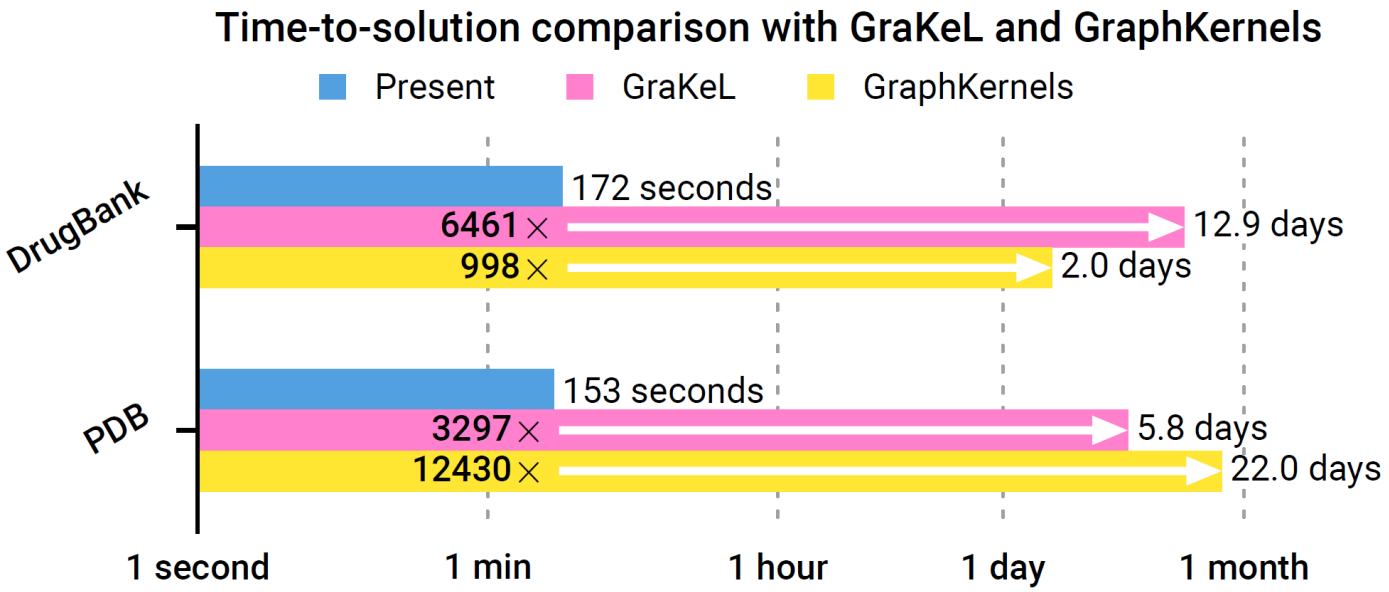
Exploiting Sparsity

- Most discrete systems have natural sparsity (e.g. not all atoms are connected).
- 2-level sparsity exploitation:
  i. Outer level: retain only non-empty tiles
  ii. Inner level: use bitmap + compact storage format
- Packing into compact format: on CPU as a preprocessing step
- Unpacking for Streaming Kronxv: on GPU using bit magic + warp intrinsics
- Partition-based graph ordering reduces # non-empty tiles

  ☛ Cost easily amortized because we reorder each graph, not their product



Performance of the Graph Kernel

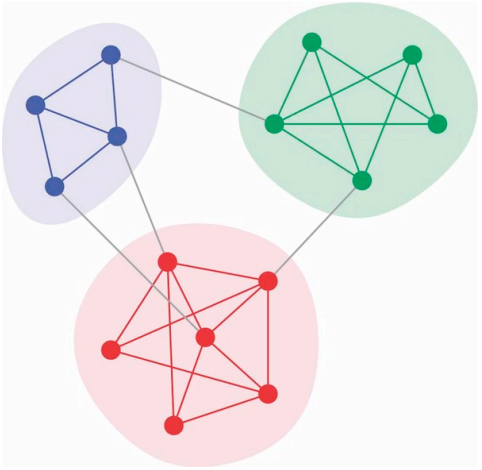


GraKeL: Cython, multi-threading
GraphKernels: Python, no parallelization

Yu-Hang Tang, Oguz Selvitopi, Doru Popovici, and Aydin Buluç. A high-throughput solver for marginalized graph kernels on GPU. In Proceedings of the IPDPS, 2020.

The Markov Cluster Algorithm (MCL)

Widely popular and successful algorithm for discovering clusters (e.g. protein families) in protein interaction and protein sequence similarity networks

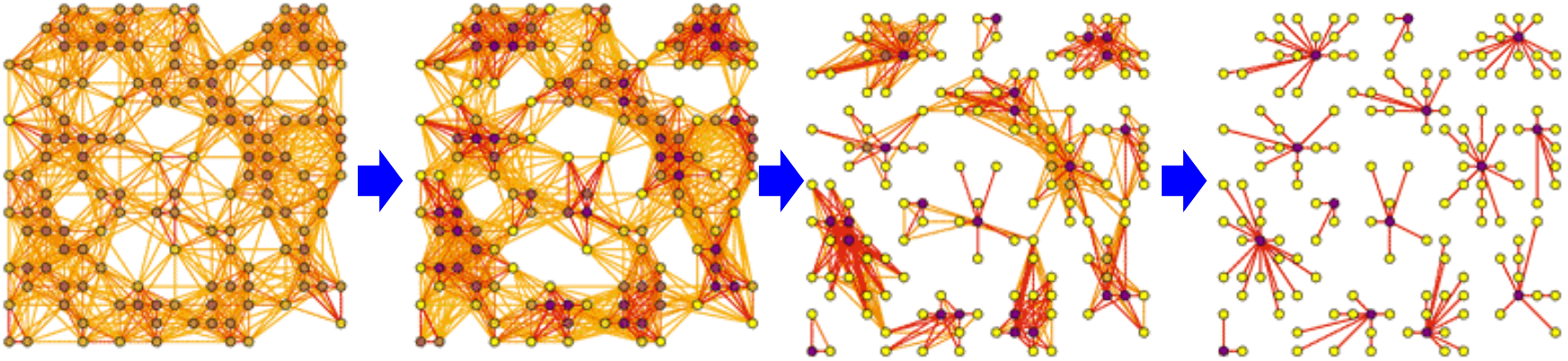


The number of **edges or higher-length paths** between two arbitrary nodes in a cluster is greater than the number of paths between nodes from different clusters

Random walks on the graph will frequently remains within a cluster

The algorithm **computes the probability** of random walks through the graph and **removes lower probability terms** to form clusters

The Markov Cluster Algorithm (MCL)



Initial network | Iteration 1 | Iteration 2 | Iteration 3

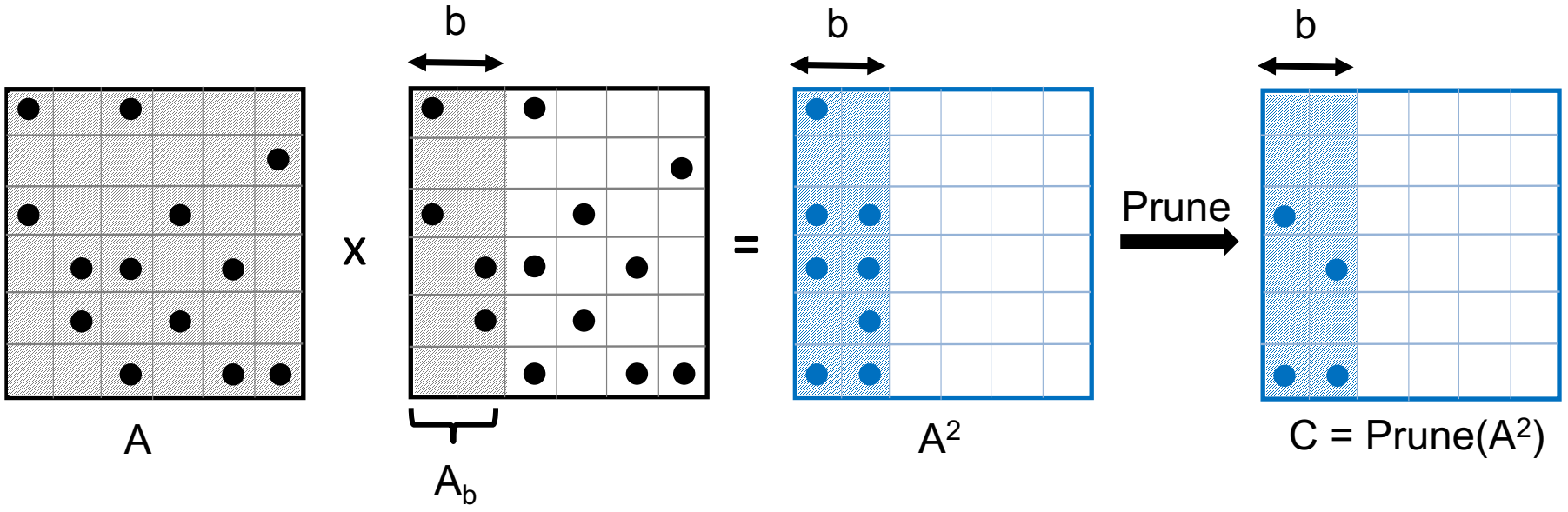
At each iteration:

Step 1 (Expansion): Squaring the matrix while pruning (a) small entries, (b) denser columns

Naïve implementation: sparse matrix-matrix product (SpGEMM), followed by column-wise top-K selection and column-wise pruning

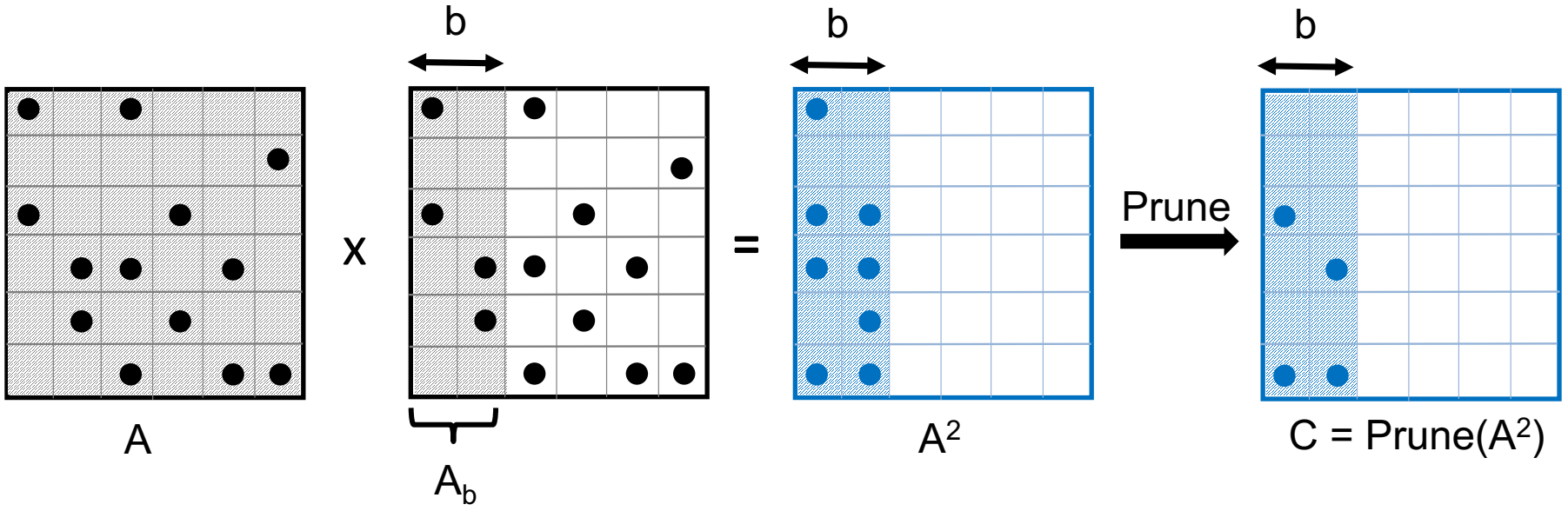
Step 2 (Inflation) : taking powers entry-wise

A combined expansion and pruning step



- b: number of columns in the output constructed at once
  - Smaller b: less parallelism, memory efficient (b=1 is equivalent to sparse matrix-sparse vector multiplication used in MCL)
  - Larger b: more parallelism, memory intensive

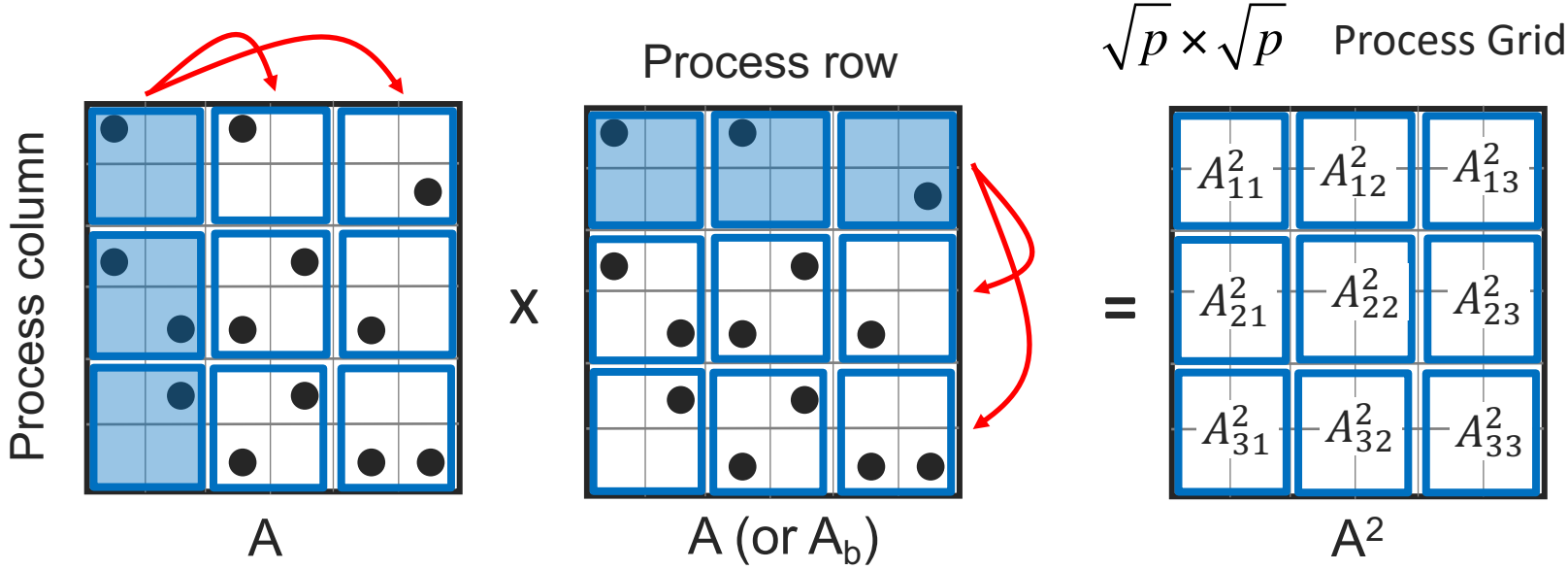
A combined expansion and pruning step



- b: number of columns in the output constructed at once
  - HipMCL selects b dynamically as permitted by the available memory
  - The algorithm works in h=N/b phases where N is the number of columns (vertices in the network) in the matrix

HipMCL: High-performance MCL

- MCL process is both **computationally expensive** and **memory hungry**, limiting the sizes of networks that can be clustered
- HipMCL overcomes such limitation via **sparse parallel algorithms**.
- **Up to 1000X times faster** than original MCL with same accuracy.



A. Azad, G. Pavlopoulos, C. Ouzounis, N. Kyrpides, A. Buluç; HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks, *Nucleic Acids Research, 2018*

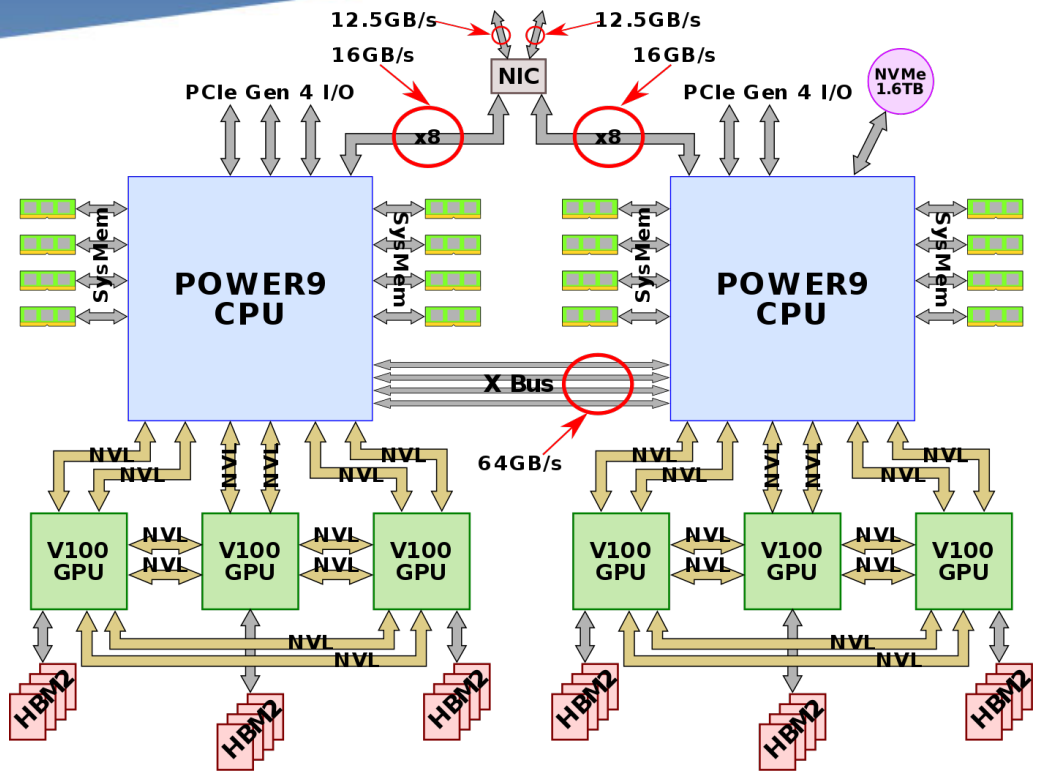
HipMCL on large networks

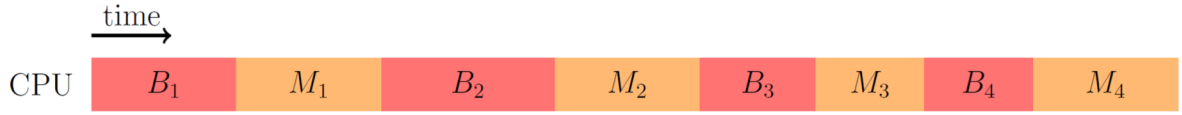
Data	Proteins	Edges	#Clusters	HipMCL time	platform
Isolate-1	47M	7 B	1.6M	1 hr	1024 nodes Edison
Isolate-2	69M	12 B	3.4M	1.66 hr	1024 nodes Edison
Isolate-3	70M	68 B	2.9M	2.41 hr	2048 nodes Cori KNL
MetaClust50	282M	37B	41.5M	3.23 hr	2048 nodes Cori KNL

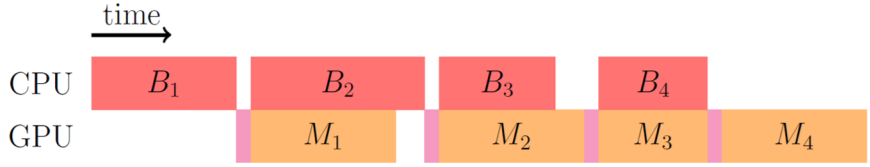
MCL can not cluster these networks

HipMCL on Supercomputers with accelerators

- Recent top supercomputers are all accelerated (e.g. with GPUs)
- This is what a ORNL Summit node looks like
- There are 4608 such nodes in the system
- Challenges: (1) Utilizing all GPUs, (2) hiding the communication



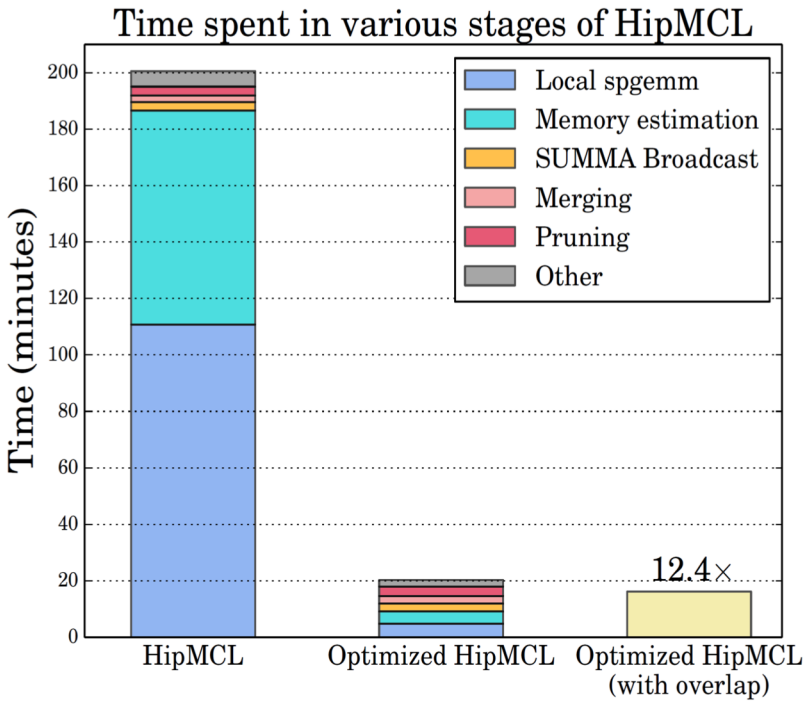


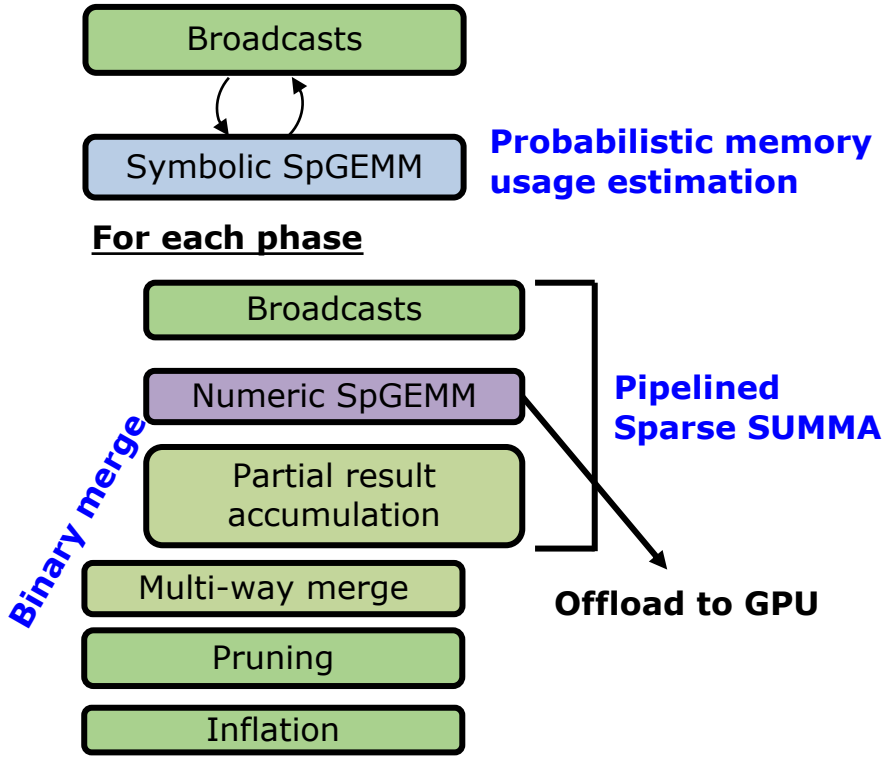


Pipelined Sparse SUMMA

Joint CPU-GPU distributed memory expansion of MCL algorithm

HipMCL on Supercomputers with accelerators





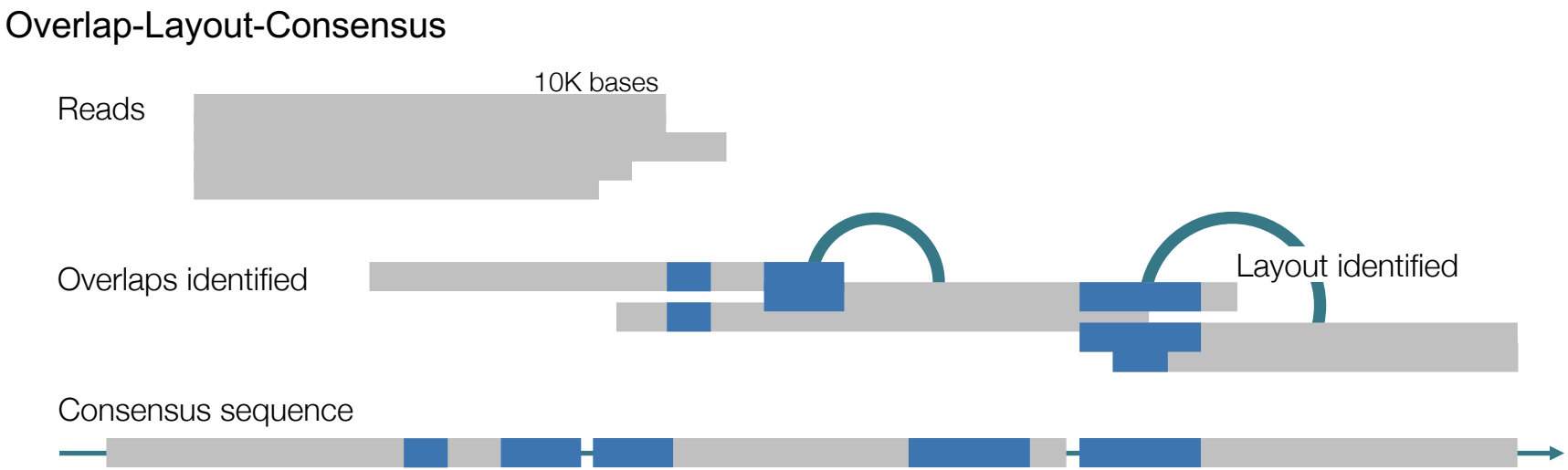
Other changes to HipMCL for the CPU-GPU workflow:

- *Randomized memory estimation algorithm* avoids symbolic phase
- New *eager binary merging* reduces memory footprint
- Integration of a much faster hash-based CPU SpGEMM algorithm

O. Selvitopi, M.T. Hussain, A. Azad, and A. Buluç. Optimizing high performance Markov clustering for pre-exascale architectures. IPDPS, 2020

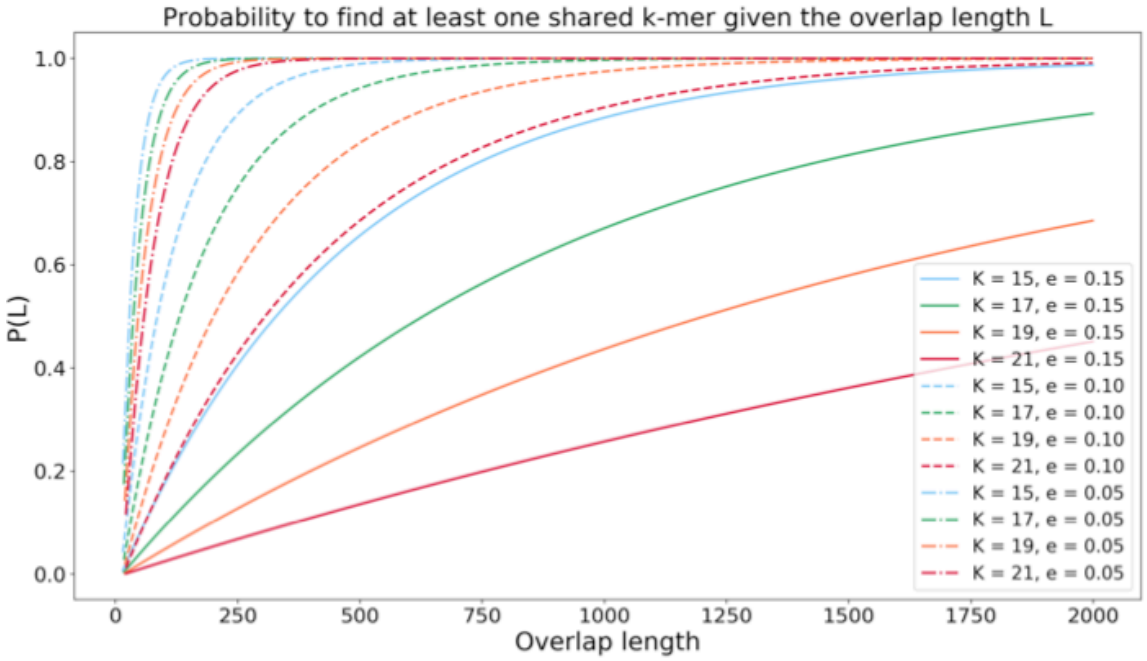
SpGEMM for DNA read overlapping

- **Long reads** from PacBio and Oxford Nanopore have the potential to revolutionize de-novo assembly
- **Overlap-Consensus-Layout** paradigm is more suitable than de Bruijn graph paradigm.
- **Overlapping** is the most computationally expensive step.

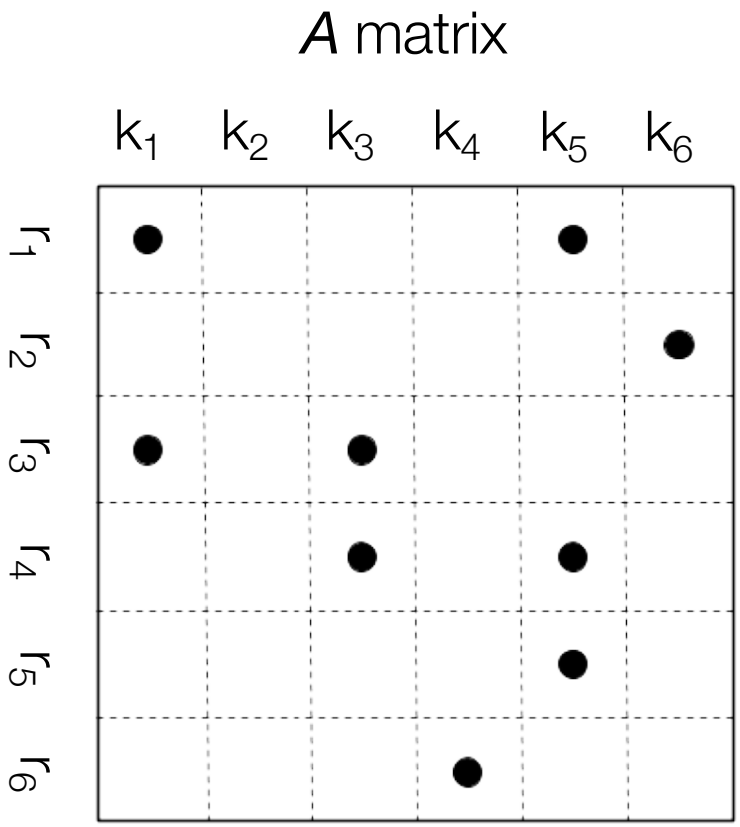


SpGEMM for DNA read overlapping

- We need to quickly determine pairs of reads that are *likely to* overlap, without resorting to $O(n^2)$ comparisons
- If two reads do not share any subsequence of length k (aka a k-mer) for a reasonably short k, then they are unlikely to overlap



SpGEMM for DNA read overlapping



- Suppose you have counted k-mers and only retained *reliable* k-mers
- Now you can generate this ***read-by-kmer*** sparse matrix **A**
- These are all linear time computations so far

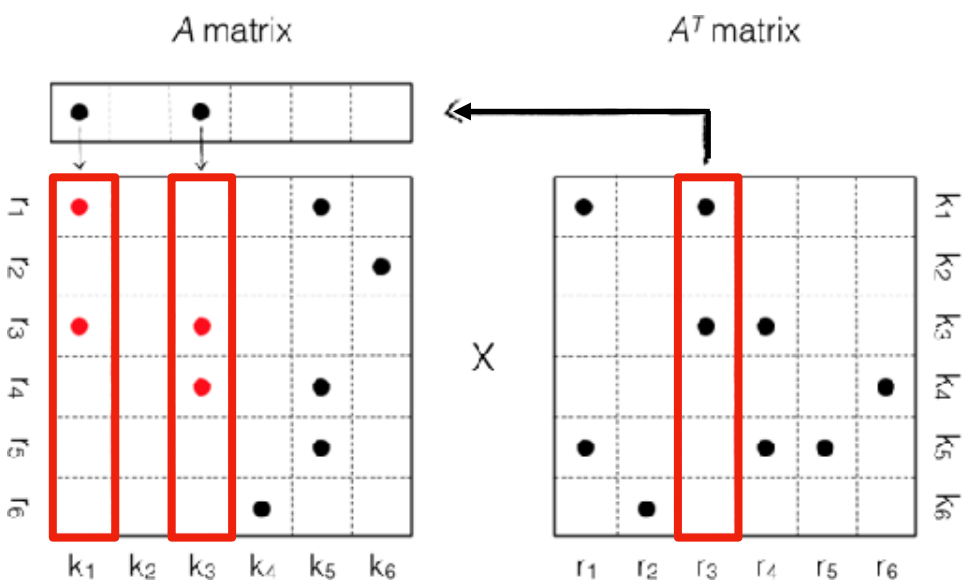
r_i = i^{th} read

k_j = j^{th} reliable ***k***-mer

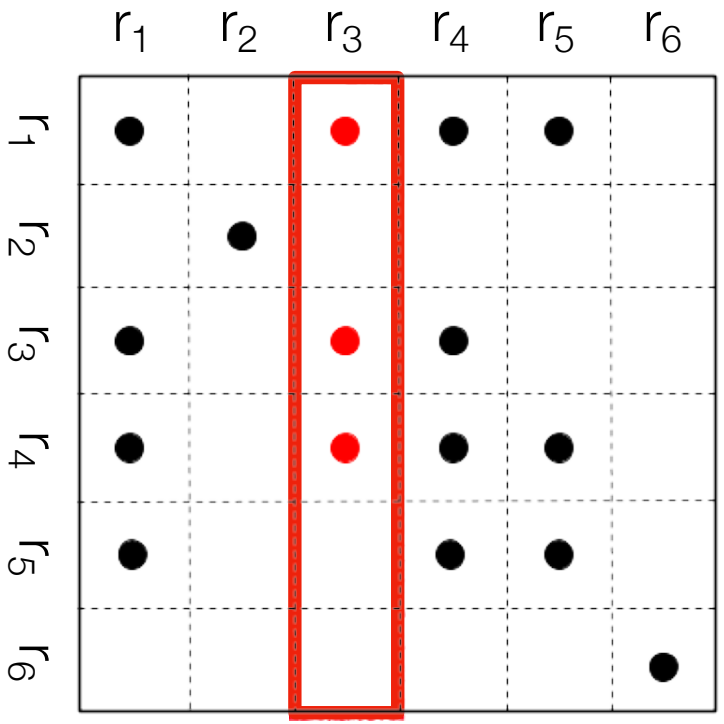
A(i,j) = presence of j^{th} reliable ***k***-mer in i^{th} read, plus its position

Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Kathy Yelick, Aydın Buluç, BELLA: Berkeley Efficient Long-read to Long-Read Overlapper and Aligner, Biorxiv, 2018

SpGEMM for DNA read overlapping



Read-by-read overlap matrix: AA^T



$AA^T(i,j)$ = # shared k-mers between reads i and j, plus their positions in the reads

Use any fast SpGEMM algorithm, as long as it runs on ***arbitrary semirings***

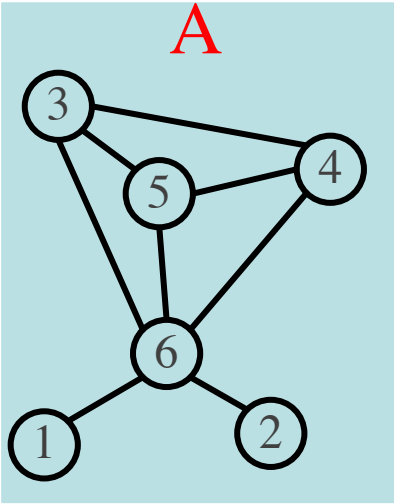
Acknowledgments

Ariful Azad, Tim Davis, Marquita Ellis, John Gilbert, Giulia Guidi, Jeremy Kepner, Nikos Krypides, Tim Mattson, Scott McMillan, Jose Moreira, John Owens, Georgios Pavlopoulos, Dan Rokhsar, Oguz Selvitopi, Yu-Hang Tang, Carl Yang, Kathy Yelick.

- My Research Team: http://passion.lbl.gov
- Our (new) Youtube Channel: http://shorturl.at/lpFRY
- The GraphBLAS Forum: http://graphblas.org

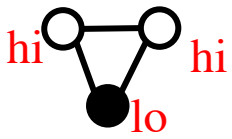
Extra Slides

Counting triangles

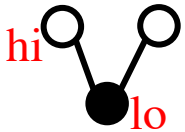


Thanks to triangle counting, we knew or sensed that something smarter algorithmically could be done than just eliminating temporaries.

Cohen's algorithm to count triangles:



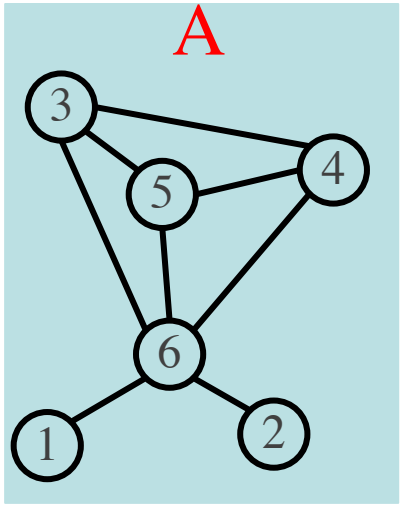
- Count triangles by lowest-degree vertex.



- Enumerate "low-hinged" wedges.

- Keep wedges that close.

Counting triangles

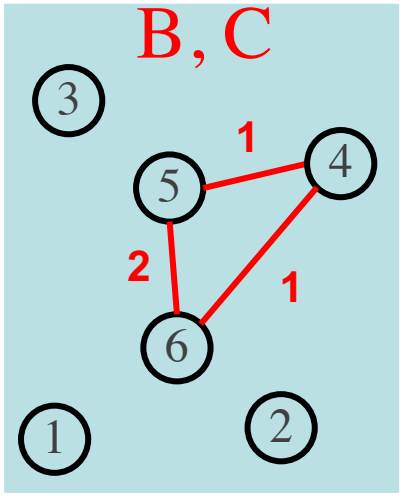


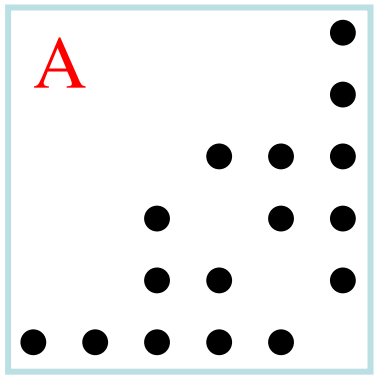
$A = L + U$ (hi->lo + lo->hi)

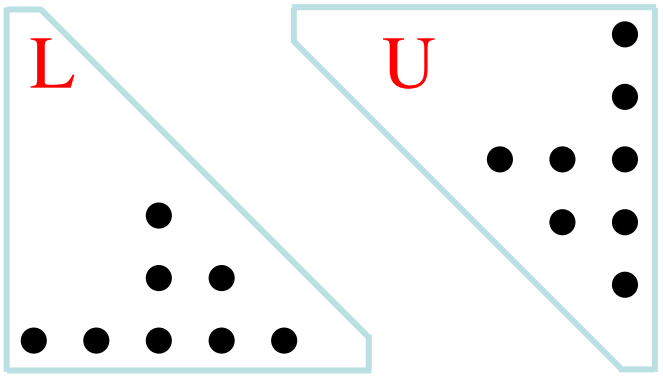
$L \times U = B$ (wedge, low hinge)

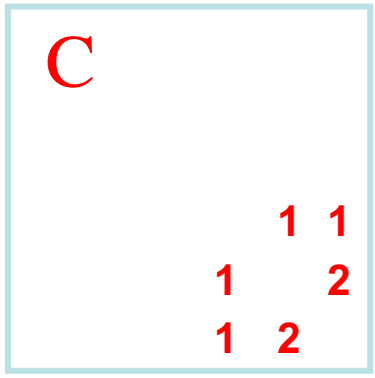
$A \wedge B = C$ (closed wedge)

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





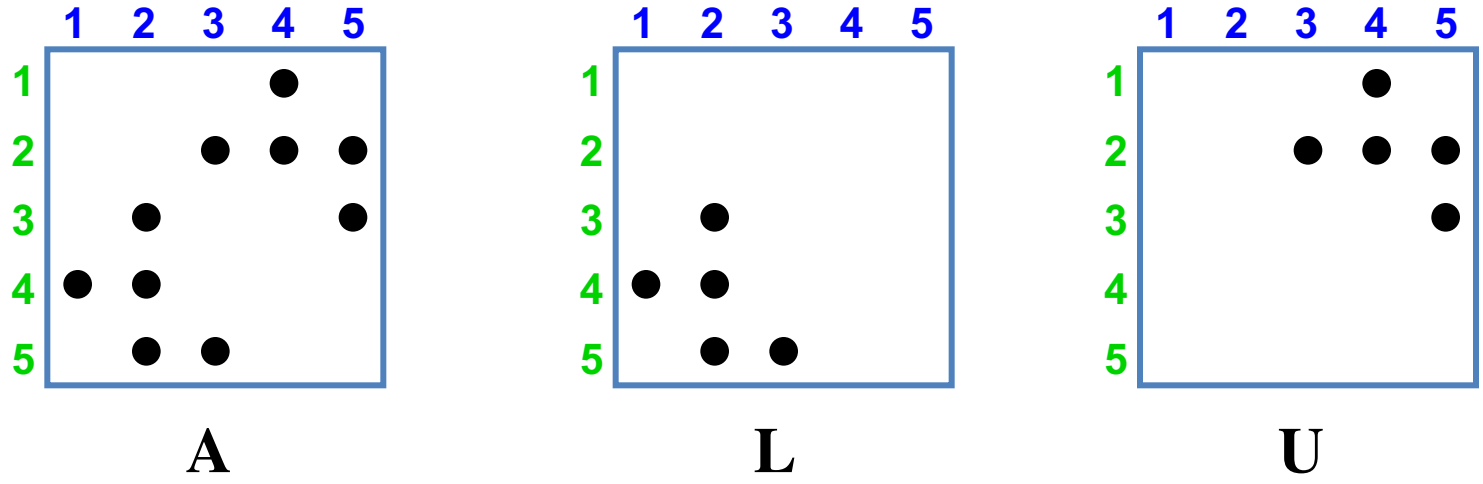




Azad, B., Gilbert. "Parallel triangle counting and enumeration using matrix algebra". *IPDPSW, 2015*

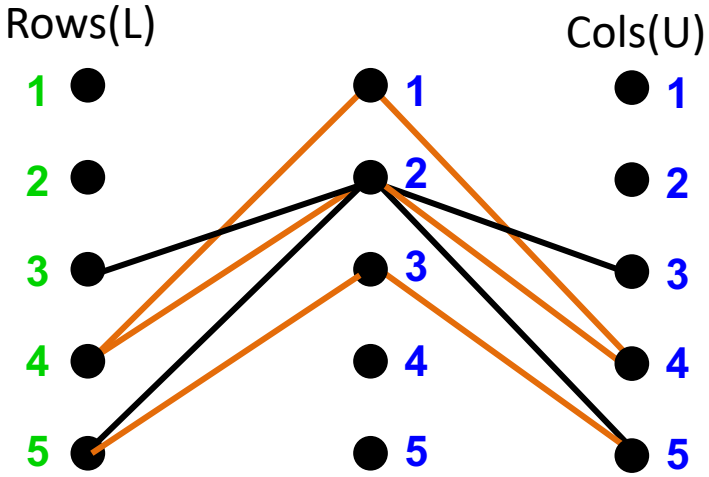
The first paper that has the word "Masked SpGEMM" in it

Exploiting Masks to avoid computation



Triangle counting example: **B = (LU) ^ A** ⇔ **mxm (&B, A, GrB_NULL, Int32AddMul, L, U)**

- Orange edges can not contribute to the output, so drop them before computation
- If the mask is really sparse, just run the inner product SpGEMM on mask nonzeros
- The inner product algorithm is *equivalent to the set intersection algorithm* Andrew L. wanted (at HPEC'17) GraphBLAS to beat



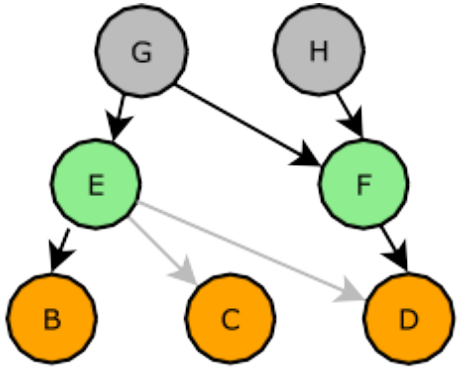
Triangle Counting in GraphBLAS

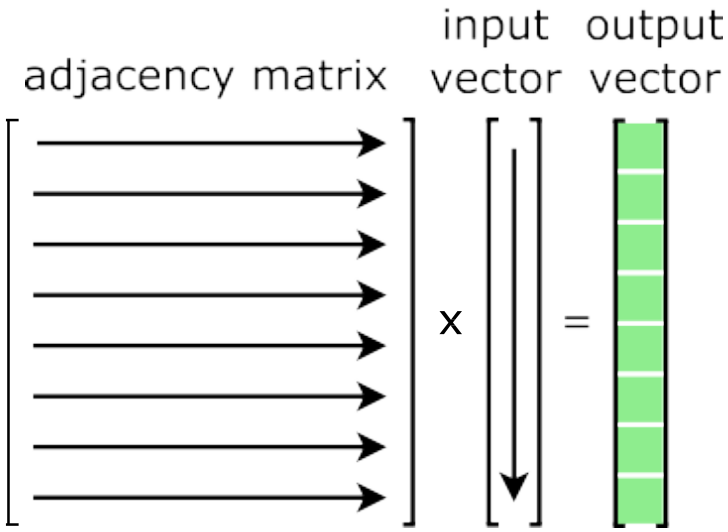
```
/*
 * Given, L, the lower triangular portion of n x n adjacency matrix A (of and
 * undirected graph), computes the number of triangles in the graph.
 */
uint64_t triangle_count(GrB_Matrix L)                  // L: NxN, lower-triangular, bool
{
  GrB_Index n;
  GrB_Matrix_nrows(&n, L);                             // n = # of vertices

  GrB_Matrix C;
  GrB_Matrix_new(&C, GrB_UINT64, n, n);

  GrB_Monoid UInt64Plus;                               // integer plus monoid
  GrB_Monoid_new(&UInt64Plus,GrB_PLUS_UINT64,0ul);

  GrB_Semiring UInt64Arithmetic;                       // integer arithmetic semiring
  GrB_Semiring_new(&UInt64Arithmetic,UInt64Plus,GrB_TIMES_UINT64);

  GrB_Descriptor desc_tb;                              // Descriptor for mxm
  GrB_Descriptor_new(&desc_tb);
  GrB_Descriptor_set(desc_tb,GrB_INP1,GrB_TRAN); // transpose the second matrix

  GrB_mxm(C, L, GrB_NULL, UInt64Arithmetic, L, L, desc_tb); // C<L> = L *.+ L'

  uint64_t count;
  GrB_reduce(&count, GrB_NULL, UInt64Plus, C, GrB_NULL);     // 1-norm of C

  GrB_free(&C);                          // C matrix no longer needed
  GrB_free(&UInt64Arithmetic);           // Semiring no longer needed
  GrB_free(&UInt64Plus);                 // Monoid no longer needed
  GrB_free(&desc_tb);                    // descriptor no longer needed

  return count;
}
```

http://graphblas.org

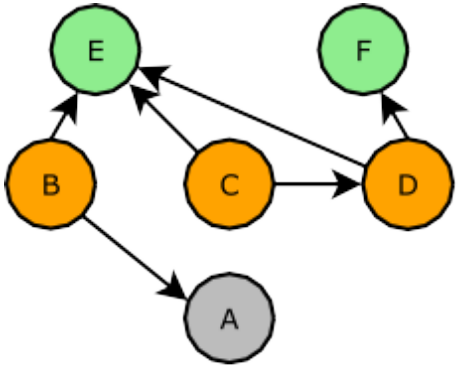
Push-pull ≡ column-row matvec

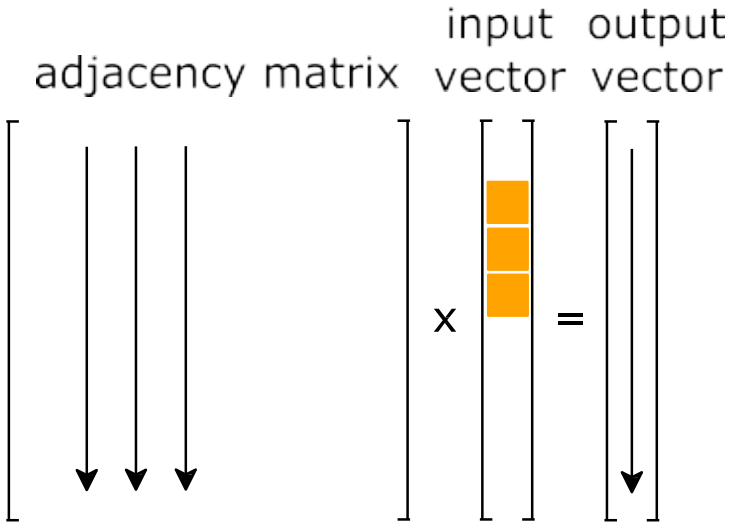
Pull





Push



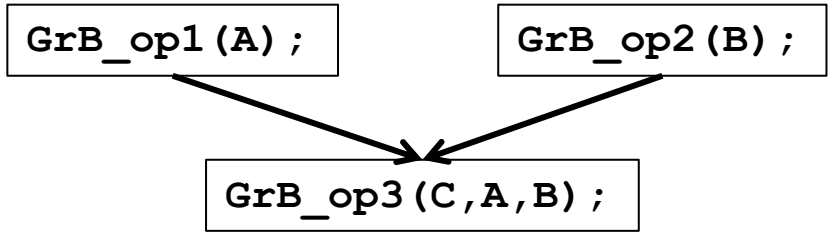


Yang, C., Buluc, A. and Owens, J.D., Implementing Push-Pull Efficiently in GraphBLAS. *ICPP'18*

GraphBLAS Execution modes

- A GraphBLAS program defines a DAG of operations.
- Objects are defined by the sequence of GraphBLAS method calls, but object's value is not assured until a GraphBLAS method queries its state.
- This gives an implementation flexibility to optimize the execution (fusing methods, replacing method sequences by more efficient ones, etc.)

```
GrB_op1(A);
GrB_op2(B);
GrB_op3(C,A,B);
```



- The GraphBLAS execution runs in one of two modes:
  - **Blocking mode** … executes methods in program order with each method completing before the next is called
  - **Non-Blocking mode** … methods launched in order. Complete in any order consistent with the DAG. Objects do not exit in fully defined state until queried.

Opportunities in GraphBLAS' non-blocking mode

- Suppose you are solving a linear system on the Kronecker product graph
- Actually happens when you are computing similarity between two graphs
- Using "graph kernels" enable machine learning on graph structures data, such as proteins and other molecules.

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} \doteq \begin{pmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \dots & a_{1,m}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \dots & a_{2,m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}\mathbf{B} & a_{n,2}\mathbf{B} & \dots & a_{n,m}\mathbf{B} \end{pmatrix}$$

$N \times M$ $K \times L$

$N*K \times M*L$

- The Kronecker product itself has huge memory footprint and lots of redundancy (NK+ML dimension but NKML apparent values)

Opportunities in GraphBLAS' non-blocking mode

- The only way to write this in GraphBLAS or any other library we know of:

```
GrB_kronecker(C, …, A, B, …); // C=A⊗B
GrB_mxv(y, …, C, x, …); // y=C x
```

- What we would rather call:

```
GrB_kronxv(y, …, A, B, x …); // y= (A⊗B) x
```

- But that would result in API bloat and would lead us to a rabbit hole.
- There are many other examples:
  - KFAC (optimization method for deep learning),
  - Triple matrix product (graph contraction and AMG restriction),
  - Triangle counting (who needs the list of triangles when all we need is the count)

- **Solution**: **A JIT that performs automatic operator fusion**