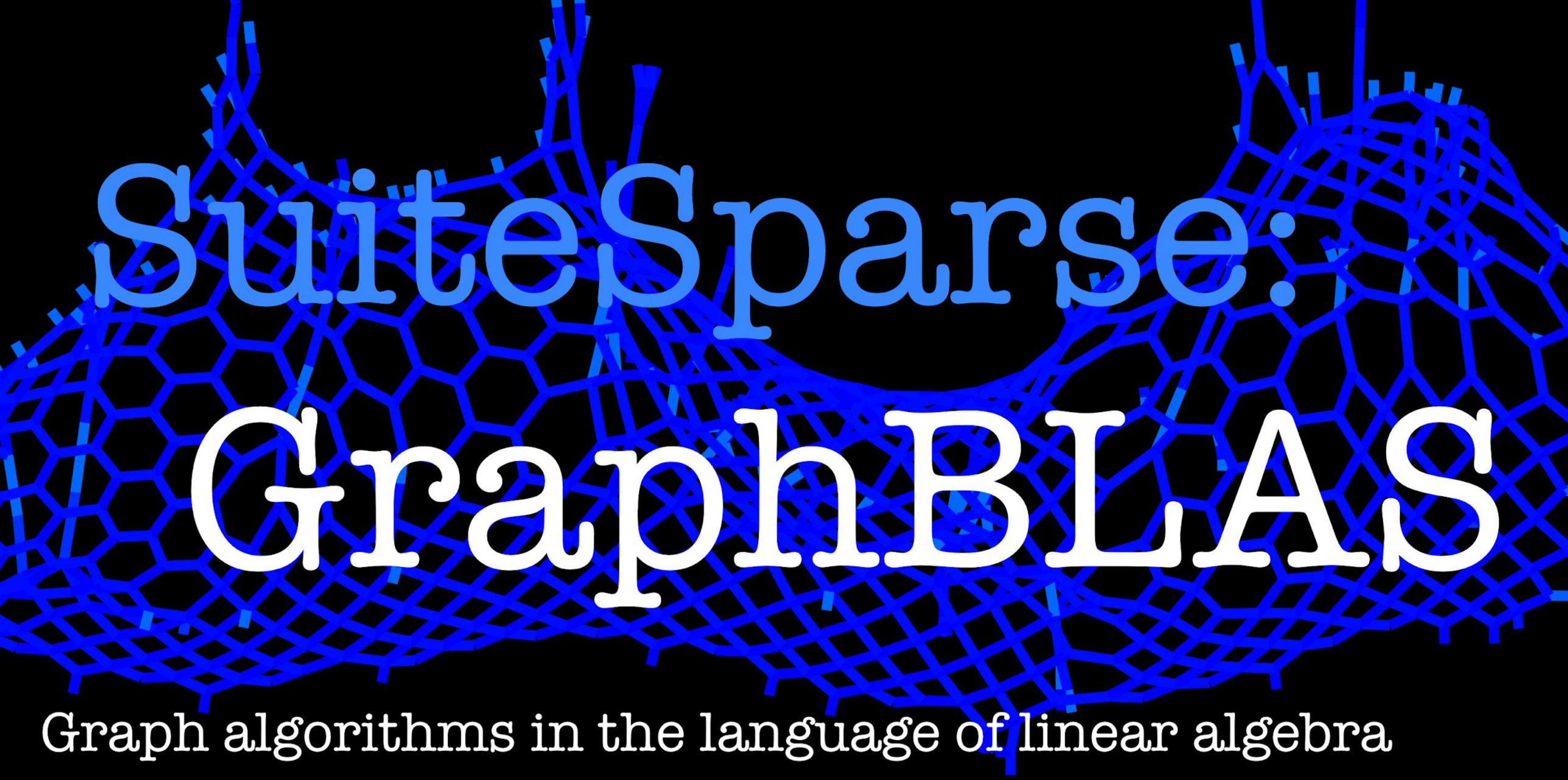


SuiteSparse: GraphBLAS

Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University

Talk given at SIAM CSE21, March 2021

SuiteSparse:GraphBLAS, a Parallel Implementation of the GraphBLAS API

function name	description	GraphBLAS notation
`GrB_mxm`	matrix-matrix mult.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix mult.	$\mathbf{w}'\langle\mathbf{m}'\rangle = \mathbf{w}' \odot \mathbf{u}'\mathbf{A}$
`GrB_mxv`	matrix-vector mult.	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise, set-intersection	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise, set-union	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{i},\mathbf{j})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
`GrB_assign`	assign submatrix	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i},\mathbf{j}) = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$ $\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GxB_subassign`	assign submatrix	$\mathbf{C}(\mathbf{i},\mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$ $\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
`GxB_select`	apply select op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A},\mathbf{k})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u},\mathbf{k})$
`GrB_reduce`	reduce to vector reduce to scalar	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:,j)]$ $s = s \odot [\oplus_{ij} \mathbf{A}(i,j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$
`GxB_kron`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A},\mathbf{B})$

caveat: GraphBLAS notation under revision

GrB_Matrix and Vector: opaque data structures

8 internal formats: 4 formats, each held by row or column:

sparse: compressed sparse vector, like MATLAB.
a matrix is a dense vector of n sparse vectors
good general-purpose format

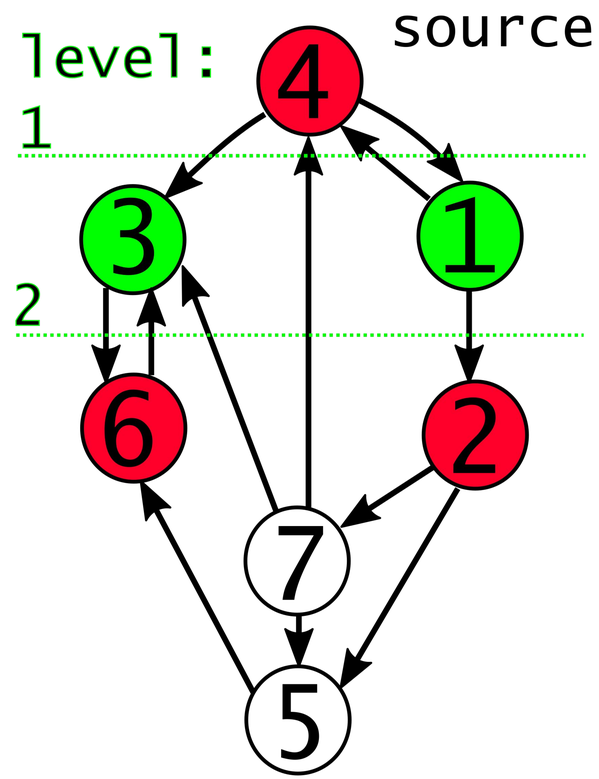
hypersparse: a sparse vector of sparse vectors
can support huge graphs (n = 2^{60}).
also good for extracting subgraphs.

bitmap: a dense matrix for the values, with an extra dense boolean matrix to describe the pattern.
good for vectors and tall-and-thin matrices.

full: just like LAPACK, but also int, bool, user-defined...

Parallel algorithms via OpenMP (CUDA in progress):
highly specialized, depending on the data structures.
C<M>=A*B: 79 different parallel algorithms, for each of 1,498 built-in semirings, plus a generic one for user-defined operators. 79*1499 = over 118 thousand C=A*B kernels.

So what can you do with these parallel kernels? Lots!



Breadth-first-search (initialization):

```
q = {source} ; parent = [ size n, all zero ]
parent (source) = source
```

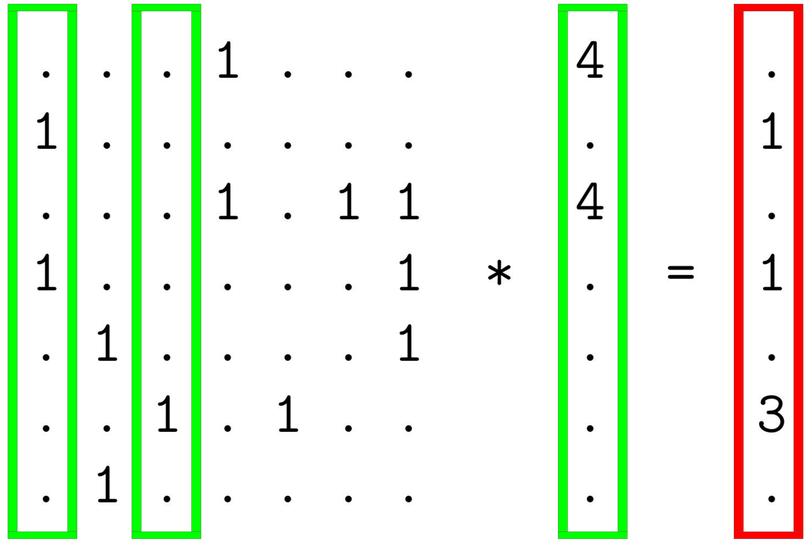
Traditional BFS:

```
while (q not empty)
    for each i in frontier q
        for each edge (i,j)
            if (j not yet seen)
                add j to next q
                parent (j) = i
                flag j as seen
```

GraphBLAS BFS: using the ANY-SECONDI semiring

```
while (q not empty)
    q<¬parent> = A'*q          masked parallel matvec
    parent<q> = q              masked parallel assignment
```

SECONDI multiplier: z = A(i,k)*q(k) = k, the parent node id
additive operator: ANY function: any(x,y) = x or y, arbitrary choice



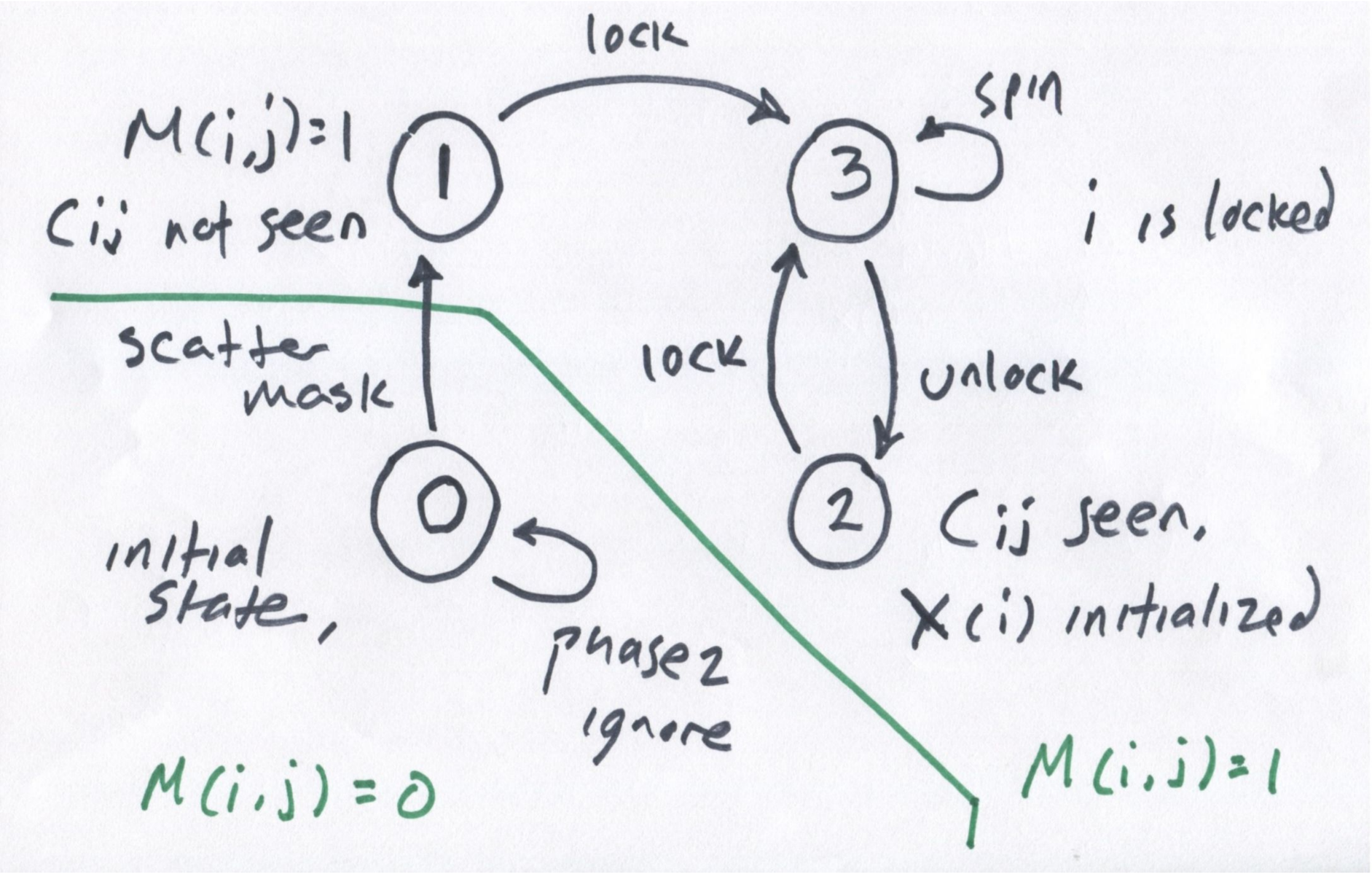
Parallel matrix-matrix multiply

- masked dot product: `C<M>=A'*B`
- unmasked dot product: `C=A'*B` or `C<¬M>A'*B`
- saxpy-style, `C=A*B`, `C<M>=A*B`, or `C<¬M>=A*B`. Mix of 4 kinds of tasks:
  - coarse Gustavson: `C(:,j1:j2) = A*B(:,j1:j2)` with O(n) workspace
  - fine Gustavson: `C(:,j) = A*B(:,j)` with many threads, atomics and shared O(n) workspace
  - coarse Hash: `C(:,j1:j2) = A*B(:,j1:j2)` with O(f) workspace, f = max "flops" for any C(:,j)
  - fine Hash: `C(:,j) = A*B(:,j)` with many threads, uses atomics and shared O(f) hash space
  - all four tasks in any `C=A*B`

A*B all variants: total 9K lines of code,
not including 320K lines of generated code for 1,499 semirings

References:
Nagasaka, Matsuoka, Azad, Buluç, "High Performance Sparse matrix-matrix products on Intel KNL and multicore architectures", ICPP'18.
Gustavson, Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition, ACM TOMS, 1978.

Fine Hash/Gustavson task with mask C<M>=A*B:



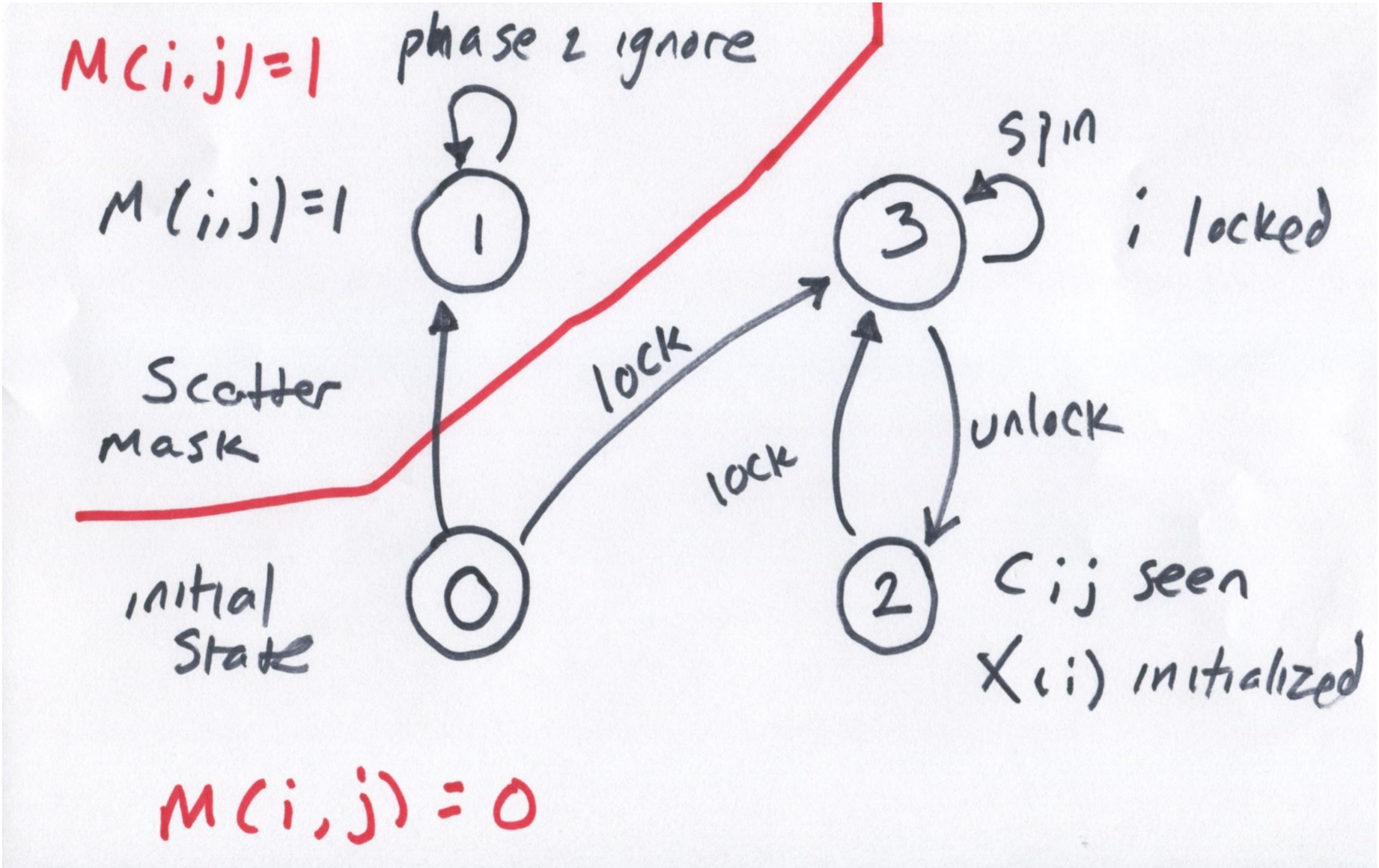
Each thread given a range `i1:i2` of rows of B:

`C(:,j)+=A*B(i1:i2,j)`

Each hash entry contains a row index i and 2-bit atomic state.

Fine Hash tasks:
Phase1: scatter M into hash
Phase2: numerical work
Phase3/4: count C(:,1:m)
Phase5: gather from hash

Fine Hash/Gustavson task: with negated mask M



Parallel assignment: **C<M>(i,j)=A C<M>(i,j)+= A**

- A blizzard of combinations:
  - mask: present or not, complemented or not, structural or not
  - replace option: true or false
  - accumulator: present or not
  - A: matrix or scalar
  - S: constructed or not
  - C, M, A: sparse/hypersparse/bitmap/full, by row/col
- Algorithms:
  - some use S = C(i,j), symbolic extraction. Given C(I,J)=A where I and J are vectors of indices.
  - C(I(2),J(3)) = A(2,3), then S(2,3) = position of C(I(2),J(3)) in the data structure for C.
  - Allows for C [S (x,y)] = A (x,y) assignment for some row x and column y.
  - some algorithms do not use S and thus do not construct it.

Parallel assignment: C<M>(I,J)=A, using S

About 40 different algorithms. Most are 2-pass. For example: C<M>(I,J)=A, with S:

- sort I and J index lists, if needed, and remove duplicates; permute A if changed
- S = C (I,J), a parallel structural extraction, does not use the mask M.
- Symbolic analysis: construct parallel tasks for 1st and 2nd passes
- First pass: Iterate through all of set union of (A,S), like A+S.
  - For each entry found in set union A+S, lookup M. If false, skip it. Otherwise:
  - if both A and S present: assign C[S(i,j)] = A (i,j), updating the existing value
  - if A present but not S: C[S(i,j)] = A(i,j) must be added to C as a new entry: pending tuple (count them)
  - if S present but not A: C[S(i,j)] must be deleted: mark it for deletion (a zombie)
- Middle pass: cumulative sum of all pending tuple counts, for all tasks
- Second pass: repeat the algorithm, but only insert pending tuples into the pile

MATLAB: native sparse matrices vs @GrB objects

@GrB vs MATLAB syntax	@GrB advantages / limitations	@GrB speedup relative to MATLAB native on 20 cores
`C = A*B`	@GrB: any semiring, any mask MATLAB: just plus-times caveat: Tim D wrote both	Up to 30x
`C(I,J) = A`	Same syntax, more types: sparse int8, int16, …, single complex, ...	2x to 1000x
`C = sparse (2^60, 2^60)` `C = GrB (2^60, 2^60)`	MATLAB: too big @GrB: no problem; hypersparse	
`C(M) = A`	MATLAB mask: same syntax, @GrB much faster	MATLAB: > one week + GraphBLAS: 7 seconds, 100,000x speedup

Intel® Xeon® E5-2698 v4 CPU with 20 cores and 40 threads, with gcc 5.4.0. Caveat: MATLAB R2018a, I need to upgrade.

Performance of BFS:

time in seconds, NVIDIA DGX Station (Intel Xeon, 20 hardware cores, 40 threads)

	Urand	Kron	Twitter	Web	Road
GAP	0.58	0.31	0.22	0.34	0.25
SuiteSparse	1.22	0.52	0.33	0.66	3.32

Performance of Betweenness Centrality:

	Urand	Kron	Twitter	Web	Road
GAP	46.4	31.5	10.8	3.0	1.5
SuiteSparse	32.7	23.6	9.25	8.2	34.4

GAP, by Scott Beamer: 6 parallel kernels, fastest method in most cases; but difficult code to write, not a user library.
SuiteSparse:GraphBLAS: also parallel, simple to write, sometimes faster; easy code to write, able to write “any” algorithm

Performance of PageRank:

time in seconds, NVIDIA DGX Station (Intel Xeon, 20 hardware cores, 40 threads)

	Urand	Kron	Twitter	Web	Road
GAP	25.3	19.8	15.2	5.1	1.0
SuiteSparse	22.2	22.2	17.2	9.3	1.3

Performance of Triangle Counting:

	Urand	Kron	Twitter	Web	Road
GAP	21.8	374.1	79.6	22.2	0.03
SuiteSparse	34.0	918.0	239.6	34.7	0.23

GAP: about tied with GraphBLAS for PageRank. About 3x faster than GraphBLAS for TC.
SuiteSparse: not yet fully exploiting non-blocking mode, so L=tril(A); C<L>=L'*L ; nt=sum(C) constructs C then sums it up.

Performance of Connected Components:

time in seconds, NVIDIA DGX Station (Intel Xeon, 20 hardware cores, 40 threads)

	Urand	Kron	Twitter	Web	Road
GAP	1.7	0.53	0.23	0.22	0.05
SuiteSparse	4.5	3.4	1.5	2.0	1.0

Performance of Single-Source Shortest Paths:

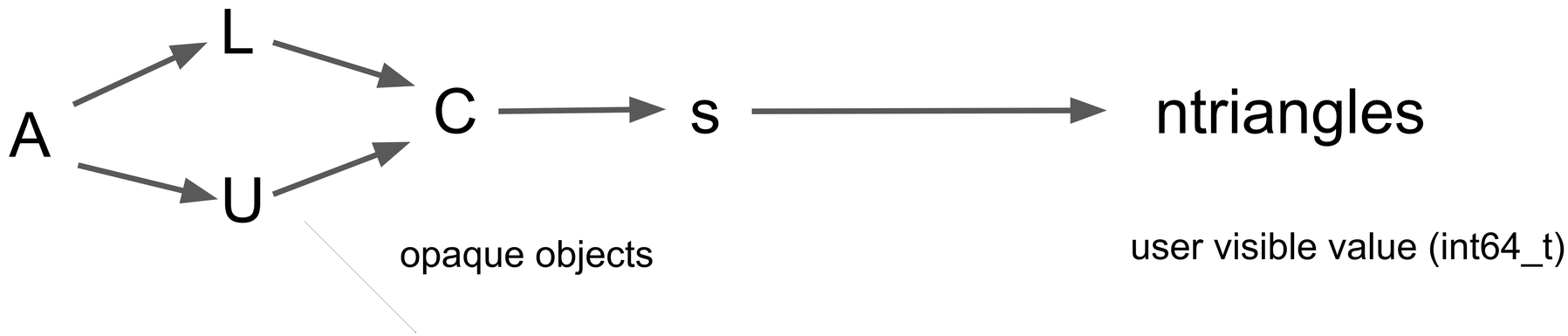
	Urand	Kron	Twitter	Web	Road
GAP	7.2	4.9	2.0	0.81	0.21
SuiteSparse	25.5	17.4	8.5	9.6	46.8

SuiteSparse: parallel code, easy to write, but typically 3x to 4x slower than the GAP, still worse for the Road graph, for Connected Components and Single-Source-Shortest-Paths.

GraphBLAS non-blocking mode

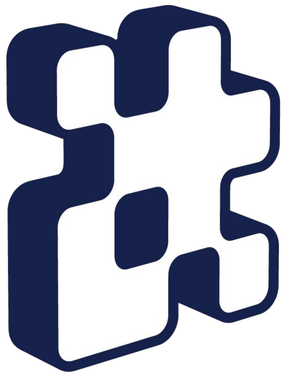
```
GxB_select (&L, … GxB_TRIL, A, ...) ;                              // L=tril(A,-1)
GxB_select (&U, … GxB_TRIU, A, ...) ;                              // U=triu(A,1)
GrB_mxm (C, L, NULL, GxB_PLUS_PAIR_INT64, L, U, GrB_DESC_ST1) ;    // C<L>=L*U'
GrB_reduce (s, NULL, GrB_PLUS_INT64_MONOID, C, NULL) ;             // s=sum(C) as GrB_Scalar
GrB_free (&C) ;  GrB_free (&L) ; GrB_free (&U) ;   // C, L, U now known to be temporary
GrB_extractElement (&ntriangles, s) ;              // ntriangles as int64_t
```

- non-blocking API allows intermediate matrices to not be instantiated
- allows for depenency DAG with fusion and lazy evaluation
- no need to form L, U, and C
- not yet exploited in SuiteSparse:GraphBLAS. In progress.

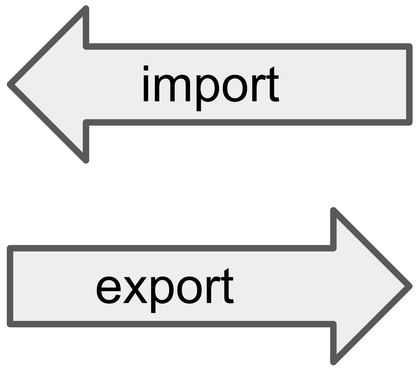


Matrix-based API vs Vertex-centered API

matrix-based API



GRAPHBLAS



vertex-centered
end user can write their own parallel vertex/edge-based code, or use a vertex/edge-centered library

it’s not ***either-or***, it’s ***both-and***

See Roger Pearce’s talk, this session

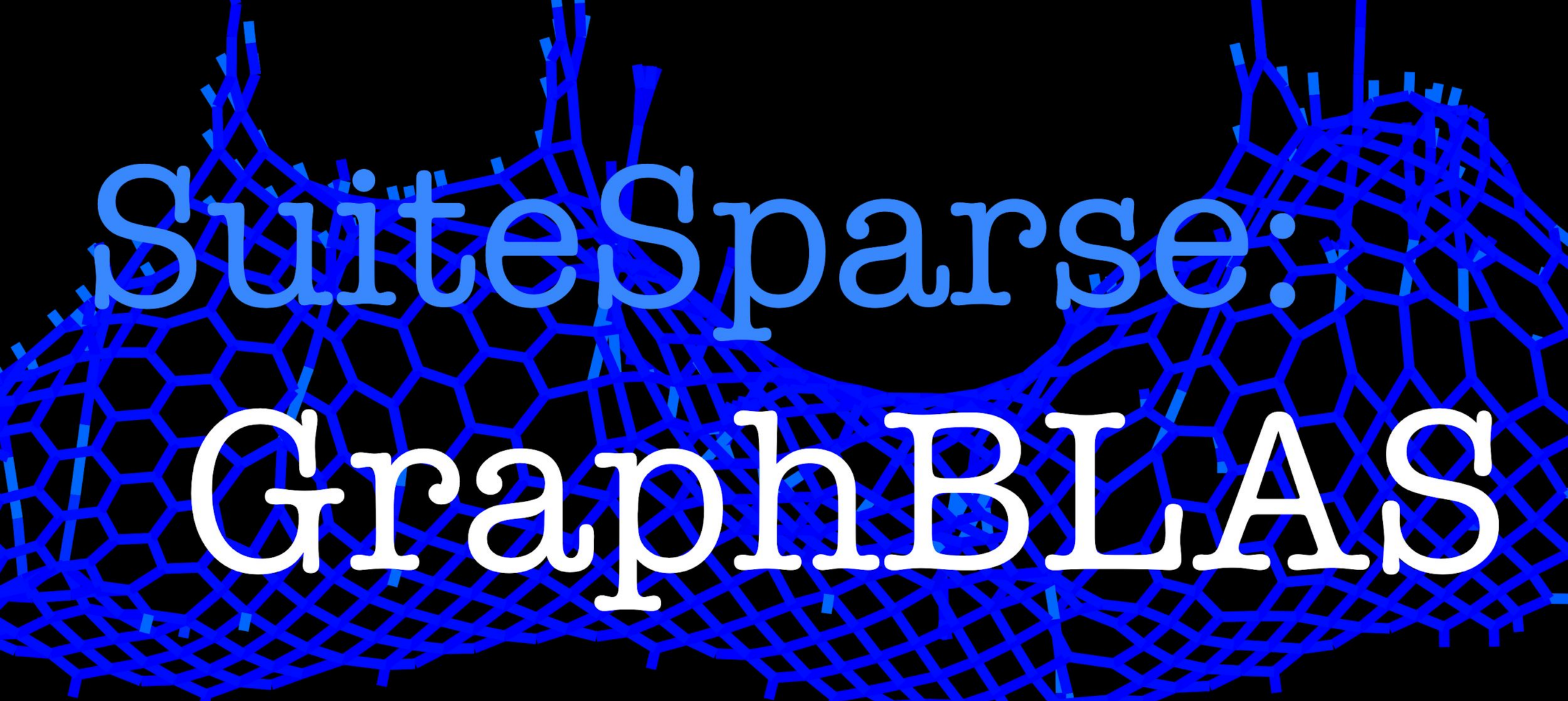
In summary: GraphBLAS strengths & limitations

Strengths:

- avoids "`for all j in Adj(i) …`" loops; akin to triply-nested loops vs C=A*B
- simple high-level API; bulk operations give lots of power to underlying implementation
- typically simple algorithms; most parallel graph algorithms can be expressed in linear algebra
- non-blocking mode in API: can fuse kernels and skip instantiating intermediate results
- SuiteSparse:GraphBLAS:
  - some asynchronous features can be expressed (`ANY` monoid)
  - no loss of performance in Python vs C API; nearly same in MATLAB
  - parallel performance can rival or even beat highly-tuned graph libraries

Limitations:

- no "`for all j in Adj(i) …`" loops, but can work side-by-side with vertex-centered libraries
- some algorithms hard to express (Depth-First-Search, Afforest CC, ...)
- SuiteSparse:GraphBLAS: non-blocking mode: just zombies, pending tuples, & lazy sort so far
- fully asynchronous methods hard to express (PageRank with Gauss-Seidel, for example) … but might be possible to extend the API



Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University