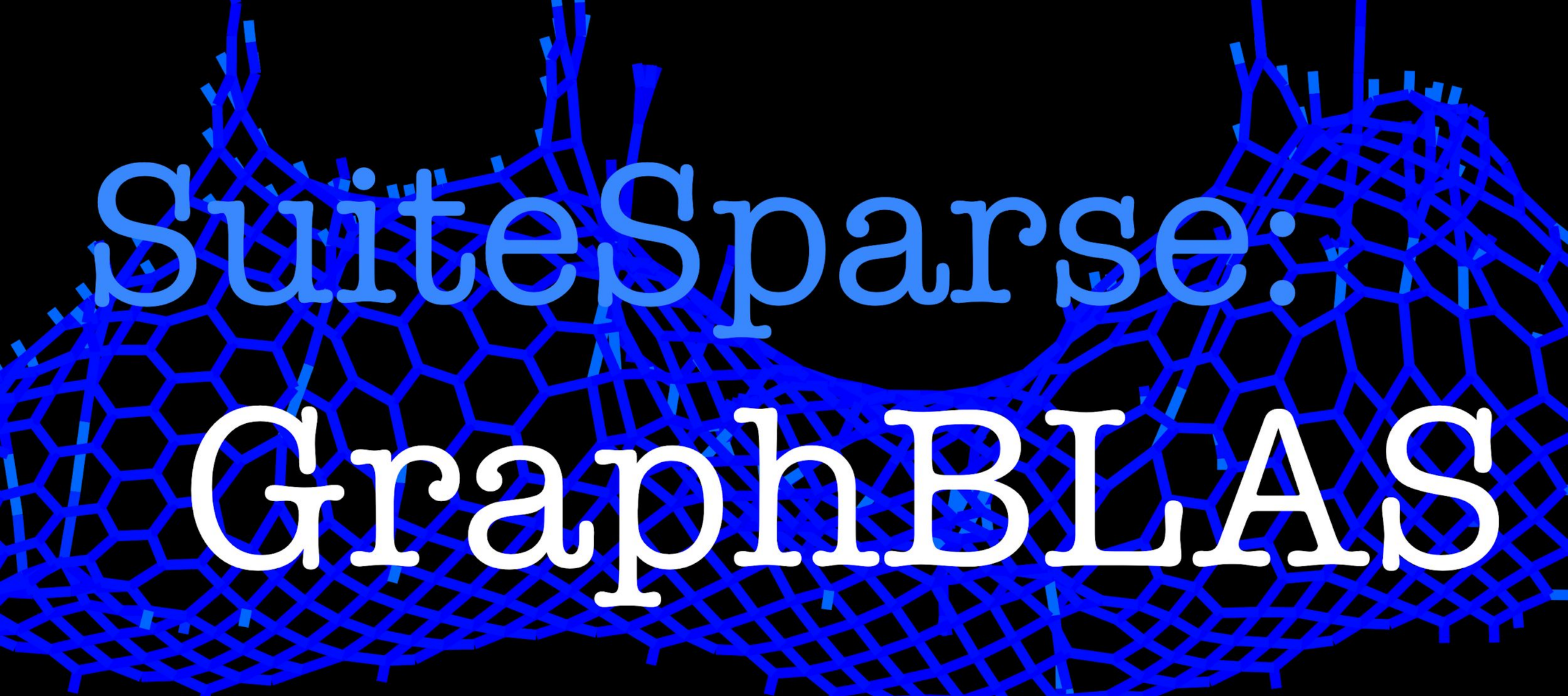


SuiteSparse: GraphBLAS

Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University

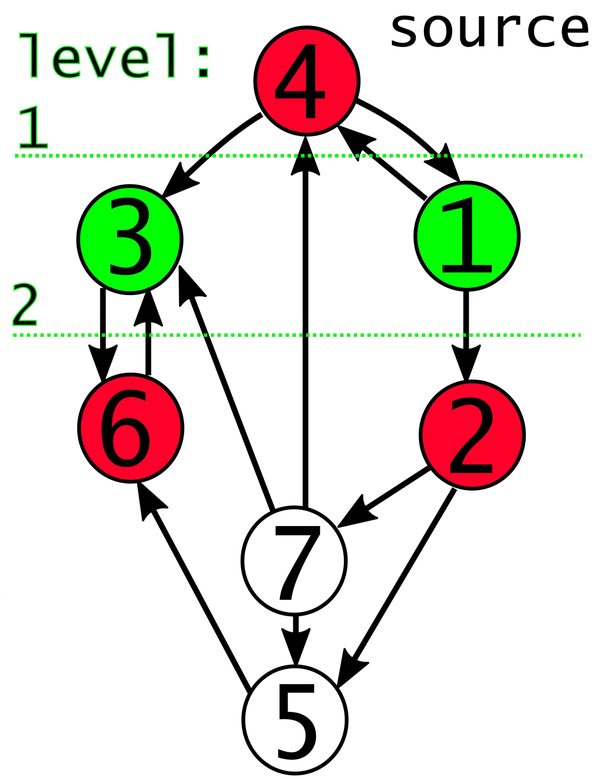
In conclusion … GraphBLAS: easy to code + high performance

Breadth-first-search (initialization):

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

Traditional BFS:

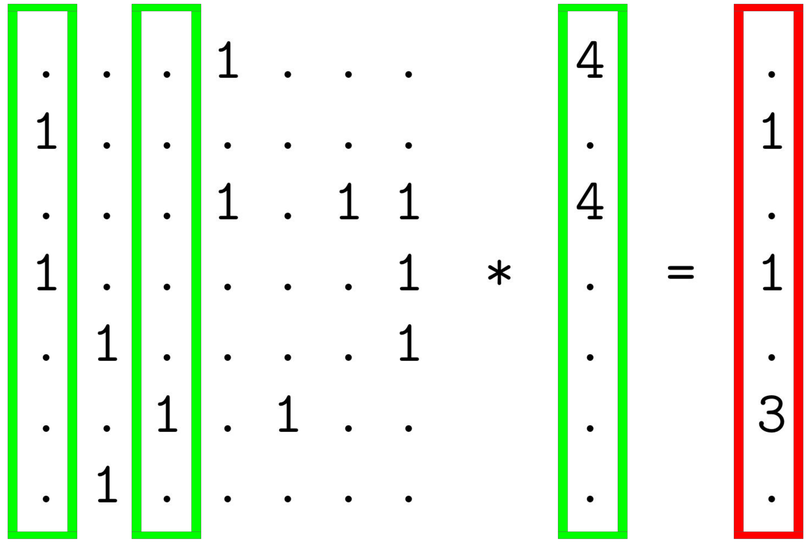
```
while (q not empty)
    for each i in frontier q
        for each edge (i,j)
            if (j not yet seen)
                add j to next q
                parent (j) = i
                flag j as seen
```



GraphBLAS BFS: using the ANY-SECONDI semiring

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



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	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A},\mathbf{B})$

Semirings for graph algorithms:

`ANY-SECONDI`: breadth-first-search

`MIN-PLUS`: single source shortest path

and many thousands more

Graph algorithms are simple to write:

in C, MATLAB, Python

Yet with good performance: close to highly-tuned specialized kernels

Appears in:

RedisGraph: commercial graph database

Linux distros

to appear in Anaconda Python

GraphBLAS in Python and MATLAB

Tim Mattson (Intel), Michel Pelletier (Graphegon, Inc.),
Tim Davis (Texas A&M)

- GraphBLAS: graph algorithms via linear algebra on sparse adjacency matrices, in different semirings
- Bulk, high-level operations, well suited for Python and MATLAB
- SuiteSparse:GraphBLAS + Python and MATLAB

For example: Matrix multiply, with a mask, accumulator

- GraphBLAS mathematical notation: **C<M>** $\otimes$**= A** min.$\oplus$ **B**
- C: `GrB_mxm (C, M, GrB_TIMES_FP32, GrB_MIN_PLUS_FP32, A, B, NULL) ;`
- Python: `A.mxm (B, out=C, mask=M, accum=FP32.TIMES, semiring=FP32.MIN.PLUS)`
- MATLAB: `C = GrB.mxm (C, M, '*', A, 'min.+', B) ;`
- But simpler in Python and MATLAB when using fewer parameters

PageRank in Python and MATLAB

```python
def pagerank(A, d, damp=0.85, itermax=100):
    # A: input graph, d: vector of out-degree for A
    # damp: damping factor, itermax: max iterations
    n = A.nrows
    # initial PageRank: all nodes have rank 1/n
    r = Vector.dense(FP32, n, 1.0 / n)
    t = Vector.sparse(FP32, n)
    # scale the out-degree vector by the damping factor
    d.assign_scalar(damp, accum=FP32.DIV)
    tfactor = (1 - damp) / n
    for i in range(itermax):
        # swap t and r every iteration
        temp = t ; t = r ; r = temp
        w = t / d
        r[:] = tfactor
        # r += A' * (t./d)
        A.mxv(w, out=r, accum=FP32.PLUS,
              semiring=FP32.PLUS_SECOND, desc=TransposeA)
        t -= r
        # halt if norm (r-t,1) < 1e-4
        t.apply(FP32.ABS, out=t)
        rdiff = t.reduce_float()
        if rdiff < 1e-4:
            break
    return r
```

```
function r = pagerank (A, d, damp, itermax)
% r = PageRank of graph A, where A is n-by-n,
% damp: damping factor, itermax: max iterations
n = size (A, 1) ;
% initial PageRank: all nodes have rank 1/n
r = GrB.ones (n, 1, 'single') / n ;
if (nargin < 2)
    % compute the row degree of each node:
    d = GrB (GrB.entries (A, 'row', 'degree'), 'single') ;
end
if (nargin < 3) damp = 0.85 ; end
d = d / damp ;
if (nargin < 4) itermax = 100 ; end
% teleport factor:
tfactor = cast ((1 - damp) / n, 'single') ;
desc.in0 = 'transpose' ;
for iter = 1:itermax
    t = r ;
    % r (:) = tfactor ;
    r = GrB.expand (tfactor, r) ;
    % r += A' * (t./d)
    r = GrB.mxm (r, '+', A, '+.2nd', t./d, desc) ;
    % halt if norm (r-t,1) < 1e-4
    if (GrB.normdiff (r, t, 1) < 1e-4) break ; end
end
```

Fig. 1: PageRank in Python and MATLAB

Betweenness Centrality in Python and MATLAB

```python
def betweenness(sources, A, AT):
    # input: set of source nodes; matrix A and transpose AT.
    # output is n-by-1: centrality(i) = score for node i.
    # initialize frontier and path count matrix (paths):
```

```matlab
function centrality = betweenness (sources, A, AT)
% input: set of source nodes; matrix A and transpose AT.
% output is n-by-1: centrality(i) = score for node i.
% initialize frontier and path count matrix (paths):
```

...

```python
    # betweenness centrality computation phase:
    bc = Matrix.dense(FP32, ns, n, 1)
    W = Matrix.sparse(FP32, ns, n)
    for i in range(depth - 1, 0, -1):
        # update from successors, mask with kth frontier:
        # W<S[k]> = bc ./ path
        bc.emult(paths, FP32.DIV, out=W,
                 mask=S[i], desc=Replace)
        # W<S[k-1]> = W*A'
        W.mxm(AT, out=W, mask=S[i-1],
              semiring=FP32.PLUS_FIRST, desc=Replace)
        # bc += W .* paths
        W.emult(paths, FP32.TIMES, out=bc, accum=FP32.PLUS)

    # centrality (i) = sum (bc(:,i)) - ns, for all nodes i:
    centrality = Vector.dense (FP32, n, -ns)
    bc.reduce_vector(accum=FP32.PLUS, out=centrality,
                     desc=TransposeA)
    return centrality
```

```matlab
% betweenness centrality computation phase:
bc = GrB (ns, n, 'single', 'by row') ;
bc (:,:) = 1 ;
W = GrB (ns, n, 'single', 'by row') ;
for k = depth:-1:2
    % update from successors, mask with kth frontier:
    % W<S{k}> = bc ./ path
    W = GrB.emult (W, S{k}, '/', bc, paths, drs) ;
    % W<S{k-1}> = W*A'
    W = GrB.mxm (W, S{k-1}, '+.1st', W, AT, drs) ;
    % bc += W .* paths
    bc = GrB.emult (bc, '+', W, '*', paths) ;
end

% centrality (i) = sum (bc(:,i)) - ns, for all nodes i:
centrality = GrB (n, 1, 'single', 'by col') ;
centrality (:) = -ns ;
centrality = GrB.vreduce (centrality, '+', '+', bc,
    struct ('in0', 'transpose') ;
```

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`	Same syntax, *way* faster	even faster than 1000x

Intel® Xeon® E5-2698 v4 CPU with 20 cores and 40 threads, with gcc 5.4.0.