SuiteSparse: GraphBLAS

Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University

47th Univ of Arkansas Spring Lecture Series, May 2022

GraphBLAS: Graph algorithms in the language of linear algebra

graphblas.org

Steering Committee:
D. Bader (NJIT)
A. Buluç (LBNL)
J. Kepner (MIT Lincoln Lab)
T. Mattson (Intel)
H. MeyerHenke (KIT)

LAGraph Working Group:
D. Bader (NJIT)
T. Davis (A&M)
J. Kitchen (Anaconda)
R. Lipman (Redis Labs)
J. Moreira (IBM)
S. McMillan (CMU)
M. Pelletier (Graphegon)
G. Szárnyas (CWI, Netherlands)
E. Welch (NVIDIA)

Julia Interface:
W. Kimmerer (MIT)
V. Shah (Julia Computing)
A. Mehndiratta
M. Zgubic

C API Committee:
A. Buluç (LBNL)
T. Mattson (Intel)
S. McMillan (CMU)
J. Moreira (IBM)
B. Brock (UC Berkeley)

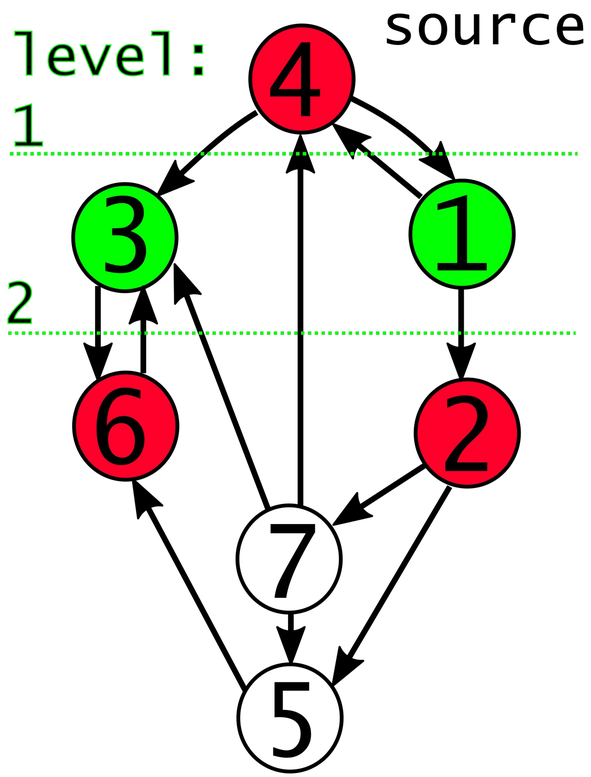
LAGraph co-authors:
A. Azad (IU)
M. Aznaveh (A&M)
J. Chen (A&M)
T. Davis (A&M)
B. Hegyi (Budapest BME)
S. Kolodziej (A&M)
T. Low (CMU)
T. Mattson (Intel)
S. McMillan (CMU)
G. Szárnyas (CWI, Netherlands)
Y. Zhang (SOKENDAI, Japan)

Python Interface:
E. Welch (NVIDIA)
J. Kitchen (Anaconda)
M. Pelletier (Graphegon)

(on G)

SuiteSparse:GraphBLAS:
T. Davis (A&M)

GraphBLAS: Graph algorithms in the language of linear algebra



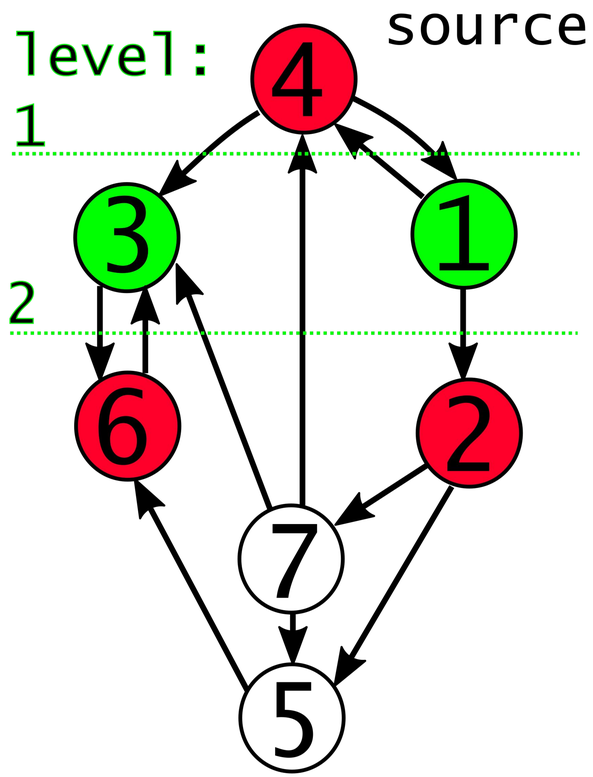
GraphBLAS: Graph algorithms in the language of linear algebra

Breadth-first-search (initialization):

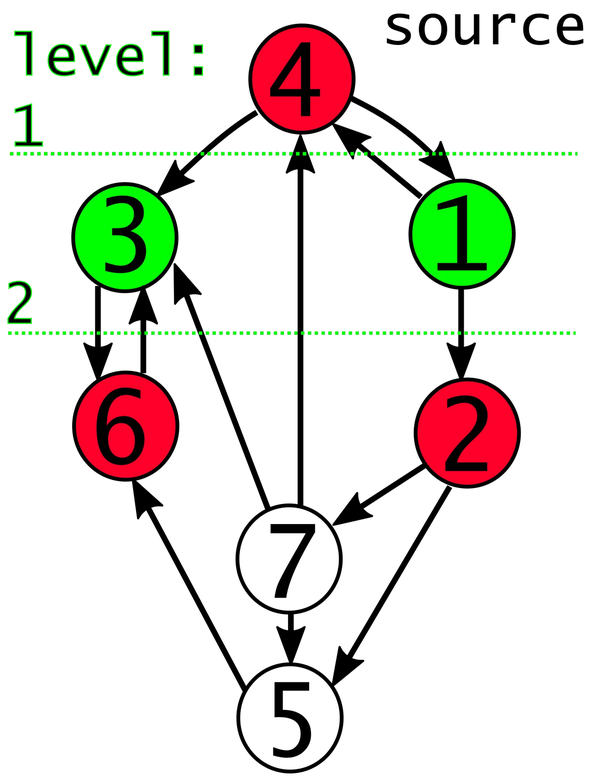
```
q = {source} ; parent = [ size n, all zero ]
parent (source) = source
```

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: Graph algorithms in the language of linear algebra



Breadth-first-search (initialization):

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

Traditional BFS:

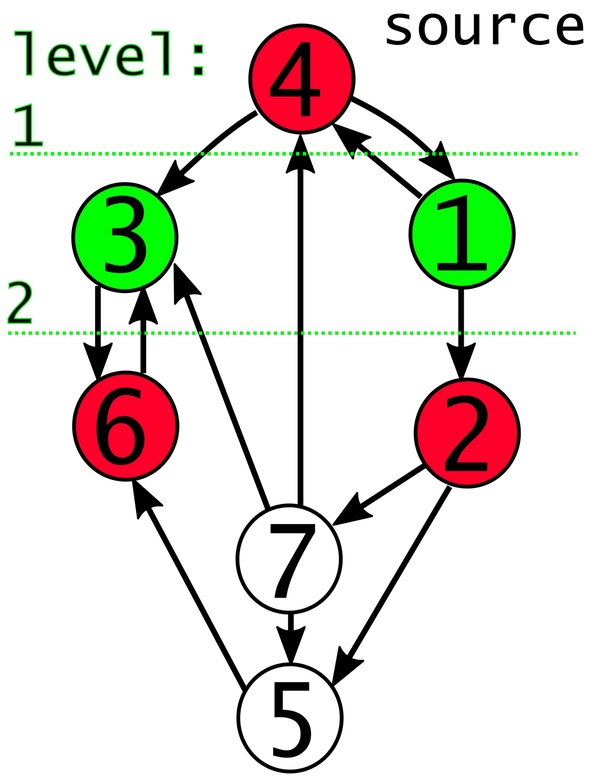
```
while (q not empty)
    for each i in frontier q
        for each edge (i,j)
            if (j not yet seen)
                add j to next q
                parent (j) = i
                flag j as seen
```

GraphBLAS BFS: using the `ANY-SECONDI` semiring

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

$$\begin{bmatrix} . & . & . & 1 & . & . & . \\ 1 & . & . & . & . & . & . \\ . & . & . & 1 & . & 1 & 1 \\ 1 & . & . & . & . & . & 1 \\ . & 1 & . & . & . & . & 1 \\ . & . & 1 & . & 1 & . & . \\ . & 1 & . & . & . & . & . \end{bmatrix} * \begin{bmatrix} 4 \\ . \\ 4 \\ . \\ . \\ . \\ . \end{bmatrix} = \begin{bmatrix} . \\ 1 \\ . \\ 1 \\ . \\ 3 \\ . \end{bmatrix}$$

GraphBLAS: Graph algorithms in the language of linear algebra



Breadth-first-search (initialization):

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

Traditional BFS:

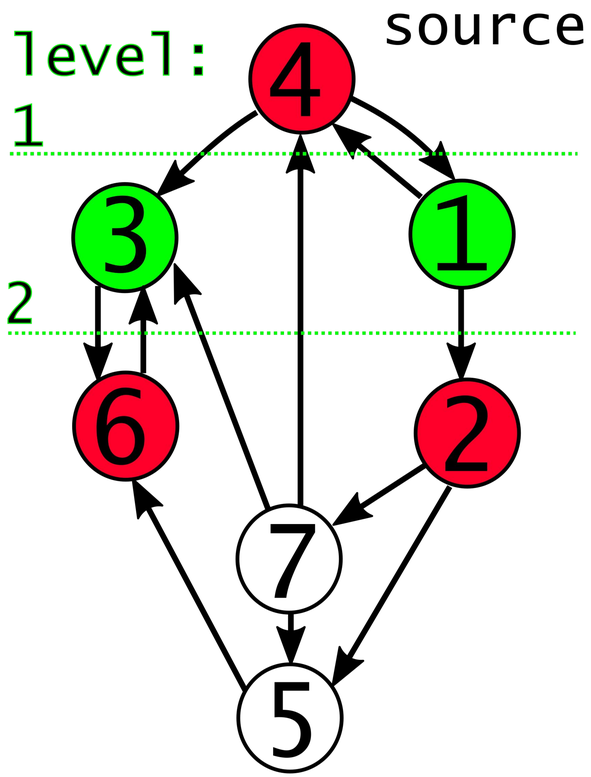
```
while (q not empty)
    for each i in frontier q
        for each edge (i,j)
            if (j not yet seen)
                add j to next q
                parent (j) = i
                flag j as seen
```

GraphBLAS BFS: using the `ANY-SECONDI` semiring

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

$$\begin{bmatrix} . & . & . & 1 & . & . & . \\ 1 & . & . & . & . & . & . \\ . & . & . & 1 & . & 1 & 1 \\ 1 & . & . & . & . & . & 1 \\ . & 1 & . & . & . & . & 1 \\ . & . & 1 & . & 1 & . & . \\ . & 1 & . & . & . & . & . \end{bmatrix} * \begin{bmatrix} 4 \\ . \\ 4 \\ . \\ . \\ . \\ . \end{bmatrix} = \begin{bmatrix} . \\ 1 \\ . \\ 1 \\ . \\ 3 \\ . \end{bmatrix}$$

GraphBLAS: Graph algorithms in the language of linear algebra



Breadth-first-search (initialization):

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

Traditional BFS:

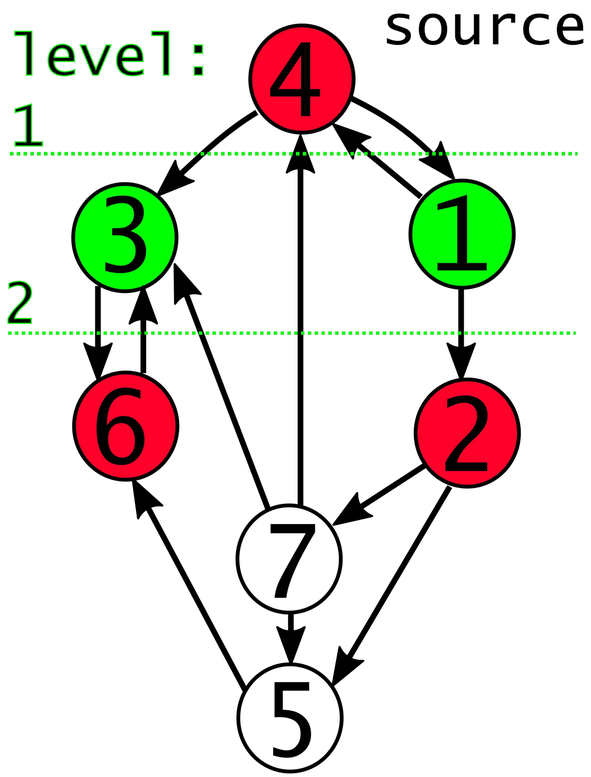
```
while (q not empty)
    for each i in frontier q
        for each edge (i,j)
            if (j not yet seen)
                add j to next q
                parent (j) = i
                flag j as seen
```

GraphBLAS BFS: using the ANY-SECONDI semiring

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

$$\begin{bmatrix} . & . & . & 1 & . & . & . \\ 1 & . & . & . & . & . & . \\ . & . & . & 1 & . & 1 & 1 \\ 1 & . & . & . & . & . & 1 \\ . & 1 & . & . & . & . & 1 \\ . & . & 1 & . & 1 & . & . \\ . & 1 & . & . & . & . & . \end{bmatrix} * \begin{bmatrix} 4 \\ . \\ 4 \\ . \\ . \\ . \\ . \end{bmatrix} = \begin{bmatrix} . \\ 1 \\ . \\ 1 \\ . \\ 3 \\ . \end{bmatrix}$$

GraphBLAS: Graph algorithms in the language of linear algebra



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

$$\begin{bmatrix} . & . & . & 1 & . & . & . \\ 1 & . & . & . & . & . & . \\ . & . & . & 1 & . & 1 & 1 \\ 1 & . & . & . & . & . & 1 \\ . & 1 & . & . & . & . & 1 \\ . & . & 1 & . & 1 & . & . \\ . & 1 & . & . & . & . & . \end{bmatrix} * \begin{bmatrix} 4 \\ . \\ 4 \\ . \\ . \\ . \\ . \end{bmatrix} = \begin{bmatrix} . \\ 1 \\ . \\ 1 \\ . \\ 3 \\ . \end{bmatrix}$$

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,+)`: single source shortest path

and many more

Graph algorithms are simple to write:

in C, MATLAB, Python, Julia

Yet with good performance: close to highly-tuned specialized kernels (some methods as fast as the GAP benchmark, others)

Non-blocking mode:

intermediate results need not be materialized

Appears in:

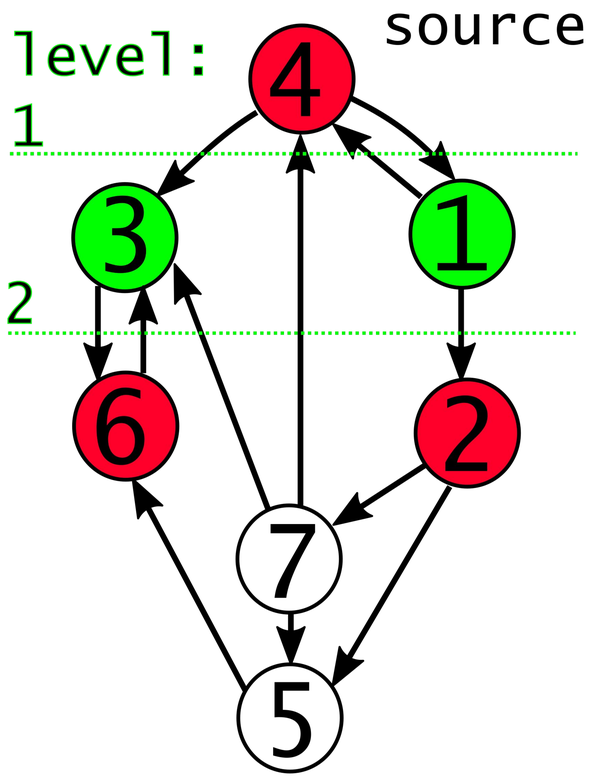
RedisGraph, Linux distros

Anaconda Python, MATLAB, Julia,...

SuiteSparse:GraphBLAS implementation

- 16 matrix formats:
  - **sparse**, O(n+e) memory: dense vector of sparse vectors
  - **hypersparse**, O(e) memory: sparse vector of sparse vectors
  - **bitmap**, O(mn) memory: dense matrix with dense boolean matrix
  - **full**, O(mn) memory: a dense matrix with all entries present
  - each of these by row or by column
  - each can be iso-valued
- data types: bool, signed/unsigned integer (8, 16, 32, 64), float, double, complex
- user-defined types and operators
- iso-valued matrices: all entries in the structure have the same value
- lazy modifications:
  - pending tuples: lazy insertions
  - zombies: lazy deletions
  - jumbled vectors: lazy sort

GraphBLAS: a few more algorithms



Pagerank:

```
while (...)
    r += A' * (r ./ d) ; // PLUS-SECOND semiring
```

Triangle counting: API non-blocking allows C not to be materialized

```
L = tril (A,-1) ;
U = triu (A,1) ;
C<L> = L*U' ;
# triangles = sum (C)
```

K-truss: C(i,j) = # of triangles incident on edge (i,j)

```
C = A
while (C changing)
    C<C> = C*C ;
    C = select (C >= k-2)
```

Sparse deep neural network: 20 minutes to write, 100x speedup over baseline

```
for each layer k
    Y = select ((Y +.* W{k}) +.+ Bias{k}) > 0) ;
    M = Y > 32 ;
    Y<M> = 32 ;
```

Single-source shortest path (Δ-stepping): pathlen = sssp (A, source, Δ)

```
pathlen (src) = 0
reach (src) = true
s (src) = true
AL = select (A <= Δ)
AH = select (A > Δ)
for step = 1:n
    tmasked<reach,replace> = pathlen
    tmasked = select (tmasked < step*Δ)
    while (tmasked not empty)
        tReq = AL'*tmasked using the (min,+) semiring
        s<tmasked> = true
        tless = (tReq .< pathlen) using set intersection
        if (tless is empty) break
        reach<tless> = true
        tmasked<tless> = select (tReq < step*Δ)
        pathlen<tless> = tReq
    end while
    tmasked<s,replace> = pathlen
    tReq = AH'*tmasked using the (min,+) semiring
    tless = (tReq .< pathlen) using set intersection
    pathlen = min (pathlen, tReq)
    reach<tless> = true
    reach<s> = empty
    if (reach is empty) break
    clear s
end for
```

- splits adjacency matrix into AL (light edges) and AH (heavy edges)
- inner loop: a single bucket, handles paths of length < (step*Δ)
- outer loop: advances to the next bucket
- new shortest paths found via the (min,+) semiring

GraphBLAS in Python, Julia, and MATLAB

Tim Mattson (Intel), Michel Pelletier (Graphegon, Inc.),
Will Kimmerer (MIT), 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, Julia, and MATLAB
- SuiteSparse:GraphBLAS + Python, Julia, and MATLAB

For example: Matrix multiply, with a mask, accumulator

- GraphBLAS mathematical notation: **C<M>** ⊗= **A** min.⊕ **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)`
- Julia: `C = *(min,+)(A,B)`
  `mul(A,B,(min,plus),mask=M,accum=*)`
- MATLAB: `C = GrB.mxm (C, M, '*', A, 'min.+', B) ;`

GraphBLAS in Julia and Python

M. Pelletier, W. Kimmerer, TD, T. Mattson

TABLE II
GRAPHBLAS OPERATIONS IN PYTHON AND JULIA

Python Function Syntax	Python Operator Syntax	Julia Function Syntax	Julia Operator Syntax
`A.min_plus(B,out=C)`	`with FP32.min_plus: C=A@B`	`C = mul(A,B,(min, plus))`	`C = ⋆(min,+)(A,B)`
`A.eadd(B,out=C,mask=M)`	`with Mask(M): C=A|B`	`C = eadd(A,B,mask=M)`	`C = ⊕(A,B,mask=M)`
`A.emult(B,out=C,accum=FP32.min)`	`with Accum(FP32.min): C=A&B`	`emul!(C,A,B,accum=/)`	`C./=A .⋆ B`
`A.assign(B,slice(1,2),slice(3,4)`	`A[1:2,3:4]=B`	`setindex!(A,1:2,3:4,B)`	`A[1:2,3:4]=B`
`A.assign(B,mask=C)`	`A[C]=B`	`setindex!(A,B,mask=C)`	`A[mask=C]=v`
`A.assign_row(1,v)`	`C[1]=v`	`setindex!(A,1,:,v)`	`A[1,:]=v`
`A.assign_col(1,v)`	`C[:,1]=v`	`setindex!(A,:,1,v)`	`A[:,1]=v`
`A.extract_row(1,out=v)`	`v=C[1]`	`v = getindex(A,1,:)`	`v = A[1,:]`
`A.extract_col(1,out=v)`	`v=C[:,1]`	`v = getindex(A,:,1)`	`v = A[:,1]`
`A.assign_scalar(42,slice(1,2))`	`A[1:2]=42`	`setindex!(A,1:2,:,42)`	`A[1:2,:] = 42`
`A.apply(FP64.abs)`	`A.abs()`	`map(abs,A)`	`abs.(A)`
`A.reduce_vector(FP64.min)`	N/A	`reduce(min,A,dims=1)`	N/A
`A.reduce_float(FP64.min)`	N/A	`reduce(min,A)`	N/A

PageRank in Julia and Python

M. Pelletier, W. Kimmerer, TD, T. Mattson

```python
def PR(A, dout, damping=0.85, itermax=100, tol=1e-4)
    # A: transpose of adjacency matrix
    # dout: vector of out-degree of A
    n = A.nrows
    t = Vector.sparse(FP64, n)
    r = Vector.dense(FP64, n, fill=1.0 / n)
    d = dout / damping
    dmin = Vector.dense(FP64, n, fill=1.0 / damping)
    d.eadd(dmin, FP64.max, out=d)
    teleport = (1 - damping) / n
    for i in range(1, itermax):
        t, r = r, t
        w = t / d
        r[:] = teleport
        A.plus_second(w, out=r, accum=FP64.plus)
        t -= r
        t.abs(out=t)
        if t.reduce_float() <= tol:
            break
    return r
```

```julia
function PR(A, d, damping=0.85, itermax=100, tol=1e-4)
    # A: transpose of adjacency matrix
    # d: vector of out-degree for A
    n = size(A, 1)
    t = GBVector{Float64}(n)
    r = GBVector(n, 1.0 / n)
    d = d ./ damping
    dmin = GBVector(n, 1.0 / n)
    eadd!(d, d, dmin, max)
    teleport = (1 - damping) / n
    for j in 1:itermax
        t, r = r, t
        w = t ./ d
        r[:] = teleport
        mul!(r, A, w, (+, second), accum=+)
        eadd!(t, t, r, -)
        map!(abs, t)
        if reduce(+, t) <= tol
            break
        end
    end
    return r
end
```

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):
    n = A.nrows
    ns = len(sources)
    paths      = Matrix.dense (FP32, ns, n, 0)
    frontier   = Matrix.sparse(FP32, ns, n)
    for i, s in enumerate(sources):
        paths[   i, sources[i]] = 1
        frontier[i, sources[i]] = 1



    # S[k] is the pattern of the frontier at level k:
    S = []
    # first frontier:  frontier<!paths> = frontier*A
    frontier.mxm(A, out=frontier, mask=paths,
        semiring=FP32.PLUS_FIRST, desc=descriptor.oocr)

    # breadth-first search stage:
    for depth in range(n):
        # S[depth] = pattern of frontier, as a bool matrix
        s = Matrix.sparse(BOOL, ns, n)
        frontier.apply(BOOL.ONE, out=s)
        S.append(s)
        # accumulate path counts: paths += frontier
        paths.assign_matrix(frontier, accum=FP32.PLUS)
        # update frontier: frontier<!paths> = frontier*A
        frontier.mxm(A, out=frontier, mask=paths,
            semiring=FP32.PLUS_FIRST, desc=descriptor.oocr)
        if frontier.nvals == 0:
            break

```

```boldface
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):
n = size (A,1) ;
ns = length (sources) ;
paths = GrB (ns, n, 'single', 'by row') ; paths (:,:) = 0 ;
frontier = GrB (ns, n, 'single', 'by row') ;
for i = 1:ns
    paths (i, sources (i)) = 1 ;
    frontier (i, sources (i)) = 1 ;
end
drc = struct ('out', 'replace', 'mask', 'complement') ;
drs = struct ('out', 'replace', 'mask', 'structural') ;
% S{k} is the pattern of the frontier at level k:
S = cell (1, n) ;
% first frontier:  frontier<!paths> = frontier*A
frontier = GrB.mxm (frontier, paths, '+.1st', ...
    frontier, A, drc) ;

% breadth-first search stage:
for depth = 1:n
    % S {depth} = pattern of frontier, as a logical matrix
    S {depth} = spones (frontier, 'logical') ;
    % accumulate path counts: paths += frontier
    paths = GrB.assign (paths, '+', frontier) ;
    % update frontier: frontier<!paths> = frontier*A
    frontier = GrB.mxm (frontier, paths, '+.1st', ...
        frontier, A, drc) ;
    if (GrB.entries (frontier) == 0)
        break ;
    end
end

```

...

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):
```

```mathematica
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
```

```mathematica
% 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') ;
```

```
const NodeID* g_out_start = g.out_neigh(0).begin();
#pragma omp parallel
{
  NodeID depth = 0;
  QueueBuffer<NodeID> lqueue(queue);
  while (!queue.empty()) {
    depth++;
    #pragma omp for schedule(dynamic, 64) nowait
    for (auto q_iter = queue.begin(); q_iter < queue.end(); q_iter++) {
      NodeID u = *q_iter;
      for (NodeID &v : g.out_neigh(u)) {
        if ((depths[v] == -1) &&
            (compare_and_swap(depths[v], static_cast<NodeID>(-1), depth))) {
          lqueue.push_back(v);
        }
        if (depths[v] == depth) {
          succ.set_bit_atomic(&v - g_out_start);
          #pragma omp atomic
          path_counts[v] += path_counts[u];
        }
      }
    }
    lqueue.flush();
    #pragma omp barrier
    #pragma omp single
    {
```

Performance of BFS:

time in seconds, NVIDIA DGX Station (Intel Xeon, 20 hardware cores, 40 threads). Draft GraphBLAS v5.

	Urand	Kron	Twitter	Web	Road
GAP	0.58	0.31	0.22	0.34	0.25
SuiteSparse	1.20	0.51	0.35	0.69	3.54

Performance of Betweenness Centrality:

	Urand	Kron	Twitter	Web	Road
GAP	46.4	31.5	10.8	3.0	1.5
SuiteSparse	31.7	23.1	9.25	6.4	34.6

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	27.8	21.4	17.2	9.4	1.4

Performance of Triangle Counting:

	Urand	Kron	Twitter	Web	Road
GAP	21.8	374.1	79.6	22.2	0.03
SuiteSparse	34.0	930.0	242.6	34.7	0.20

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	3.3	1.5	2.0	0.97

Performance of Single-Source Shortest Paths:

	Urand	Kron	Twitter	Web	Road
GAP	7.2	4.9	2.0	0.81	0.21
SuiteSparse	25.3	17.2	8.0	9.4	46.0

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.

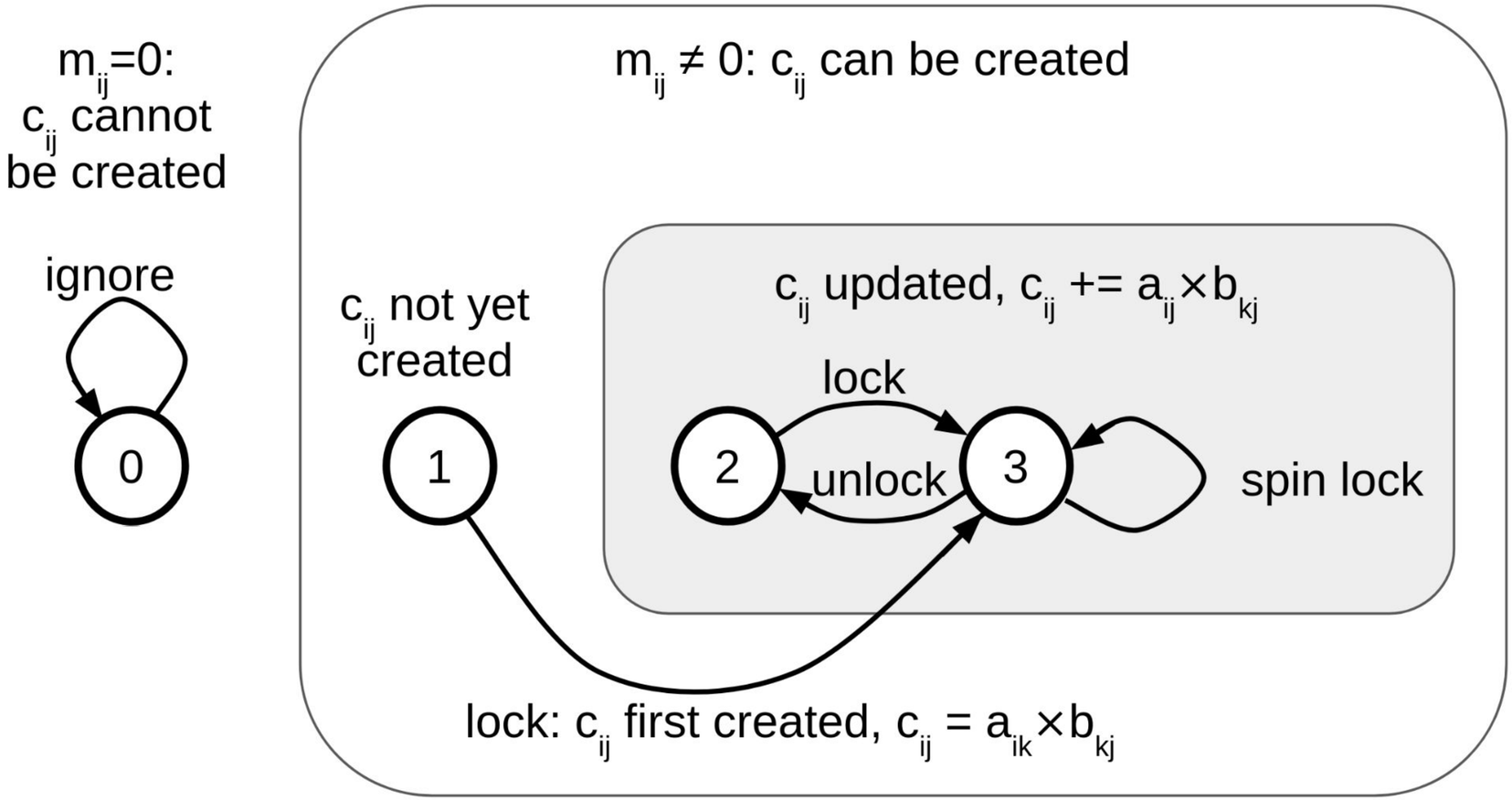
Parallel matrix-matrix multiply

- 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 10K lines of code,
not including 320K lines of generated code for 1,498 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

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` (sparse times sparse)	@GrB: any semiring, any mask MATLAB: just plus-times caveat: Tim D wrote them all	30x in MATLAB 2020b, 1x versus GrB v3.3.3 in R2021a, but another 2x to 8x with v7.0.4
`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, $O(e^2)$ @GrB much faster: O(e log e)	MATLAB: 5 days GraphBLAS: 0.8 seconds 500,000x speedup

MATLAB: native sparse matrices vs @GrB objects

@GrB vs MATLAB syntax	@GrB advantages / limitations	@GrB speedup relative to MATLAB native on 20 cores
`C = S*F` `(sparse-times-full)`	@GrB: any semiring, any mask MATLAB: just plus-times	up to 10x
`C = F*S` `(full-times-sparse)`	ditto	up to 17x
`C = A'`		up to 10x
`C=A+B` `(sparse plus sparse)`		up to 10x

Intel® Xeon® E5-2698 v4 CPU with 20 cores and 40 threads, with gcc 9.4. MATLAB R2021a, released March 2021

GraphBLAS versus the Intel MKL sparse library

computation	format	MKL method	MKL time (sec)		SuiteSparse	speedup	
			1st	2nd	time (sec)	1st	2nd
`y+=S*x`	S by row	`mkl_sparse_d_mv`	2.54	1.27	1.21	2.10	1.05
`y+=S*x`	S by col	`mkl_sparse_d_mv`	7.22	7.22	1.98	3.65	3.65
`C+=S*F`	S by row, F by row	`mkl_sparse_d_mm`	2.95	1.90	1.98	1.49	**.96**
`C+=S*F`	S by row, F by col	`mkl_sparse_d_mm`	6.12	4.99	1.48	4.13	3.37
`C+=S*F`	S by col, F by row	`mkl_sparse_d_mm`	28.82	28.82	13.78	2.09	2.09
`C+=S*F`	S by col, F by col	`mkl_sparse_d_mm`	78.82	5.17	9.38	8.40	**.55**
`C=S+B`	S by row	`mkl_sparse_d_add`	30.77	30.77	1.44	21.37	21.37
`C=S'+B`	S by row	`mkl_sparse_d_add`	102.09	27.30	16.29	6.26	1.67
`C=S'`	S by row	`mkl_sparse_convert_csr`	77.27	77.27	14.80	5.22	5.22

Table 4. SuiteSparse vs MKL 2022 with the GAP-Twitter matrix

LAGraph: graph algorithm library based on GraphBLAS (not yet v1.0)

- 6 polished, stable algorithms (the GAP benchmark):
  - Breadth-first search
  - Betweenness-centrality
  - PageRank
  - Connected Components
  - Single-source Shortest-Path
  - Triangle Counting

- stable utilities
  - malloc/calloc/realloc/free wrappers
  - create/destroy the LAGraph_Graph
  - compute properties: degree, A', # diag entries
  - delete properties
  - display graph
  - Matrix Market file I/O (very slow)
  - Sorting
  - thread control
  - timing
  - type management

- many experimental algorithms:
  - K-truss
  - All K-truss
  - Bellman-Ford single-source shortest path
  - Maximal independent set
  - Triangle Centrality
  - Community detection w/ label propagation
  - Deep Neural Network Inference
  - Strongly Connected Components
  - Minimum Spanning Forest
  - Local Clustering Coefficient
  - K-core
  - Counting all size-4 graphlets
  - Triangle polling
  - Fiedler vector
- experimental utilities:
  - random matrix, vector generators
  - Binary matrix file I/O (very fast), serialize/deserialize, parallel LZ4 comp.

SuiteSparse:GraphBLAS connections

- RedisGraph: Property graph database built using SuiteSparse:GraphBLAS: 1500 github stars, > 100K docker downloads, 1 million github pulls
  - Apple: used internally
  - Google: hosted on the Google Cloud
  - Amazon AWS
  - IBM: “We tried several graph database technologies and we really found that RedisGraph is the one that gave us the speed to solve instant real-time problems, yielding a minimum 5x improvement in query speed”
  - MDmetrix: managing and understanding medical data for better outcomes. For example: solving the opioid crisis by drastically reducing morphine prescriptions, while maintaining pain control
- Anaconda Python
- every Linux distro
- MATLAB C=A*B, 30x speedup
- Julia: the recommended package for core sparse linear algebra, also enables complete access to all GraphBLAS features (data types, semirings, …) for graph algorithms
- NVIDIA: collaboration to develop CUDA kernels, with a run-time JIT
- GraphStax.ai: discovering toxic chemicals via AI, relying on GraphBLAS
- Linkurious: visualizing RedisGraph data with Ogma
- MIT Lincoln Lab: Center for Applied Internet Data Analysis (CAIDA) Telescope: 40 trillion packets analyzed with GraphBLAS. Hypersparse matrices essential (n = entire internet protocol)

RedisGraph 2.8 Is Out!

Lior Kogan
April 29, 2022

About RedisGraph

RedisGraph is a high-performance, memory-first graph data structure for Redis. RedisGraph supports graph multi-tenancy (it can hold many graphs simultaneously) and can serve multiple clients accessing the graphs simultaneously. It is now available also as part of Redis Stack.

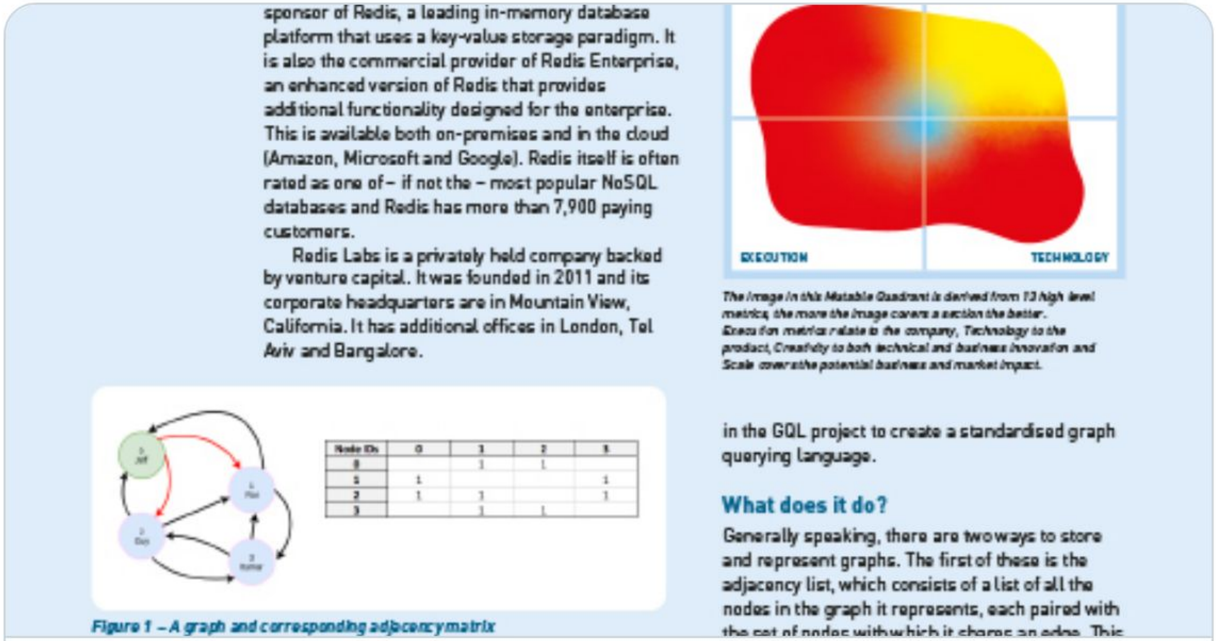
Major new features in RedisGraph 2.8

- Richer graph model
  - Multi-labeled nodes
- Enhanced querying capabilities
  - Enhanced full-text search
  - Supporting more Cypher constructs, functions, and operators
- Performance improvements
  - Indexes over relationship properties
  - Delta Matrices
  - Controllable node creation buffer
  - Benchmarks

Tweet

Bloor Research
@BloorResearch

Despite only being released in 2018, we've already seen RedisGraph from @RedisLabs mature significantly. Take a look at @InfoPhil and @DJSLHoward's latest review, which sees Redis Labs receive a Highly Commended Mutable Award: #MutableBusiness



sponsor of Redis, a leading in-memory database platform that uses a key-value storage paradigm. It is also the commercial provider of Redis Enterprise, an enhanced version of Redis that provides additional functionality designed for the enterprise. This is available both on-premises and in the cloud (Amazon, Microsoft and Google). Redis itself is often rated as one of – if not the – most popular NoSQL databases and Redis has more than 7,900 paying customers.

Redis Labs is a privately held company backed by venture capital. It was founded in 2011 and its corporate headquarters are in Mountain View, California. It has additional offices in London, Tel Aviv and Bangalore.

The image in this Mutable Quadrant is derived from 13 high level metrics; the more the image covers a section the better. Execution metrics relate to the company, Technology to the product, Creativity to both technical and business innovation and Scale covers the potential business and market impact.

in the GQL project to create a standardised graph querying language.

What does it do?

Generally speaking, there are two ways to store and represent graphs. The first of these is the adjacency list, which consists of a list of all the nodes in the graph it represents, each paired with

Figure 1 – A graph and corresponding adjacency matrix

InBrief: RedisGraph
RedisGraph is the graph database module for Redis where, by "module" the company means functionality embedded into the product as opposed to …

Search Twitter

Relevant people

Bloor Res... @BloorRe... Following
A leading independent research and analyst house for over 30 years. Pioneering the #MutableBusiness story - a permanent state of reinvention for businesses.

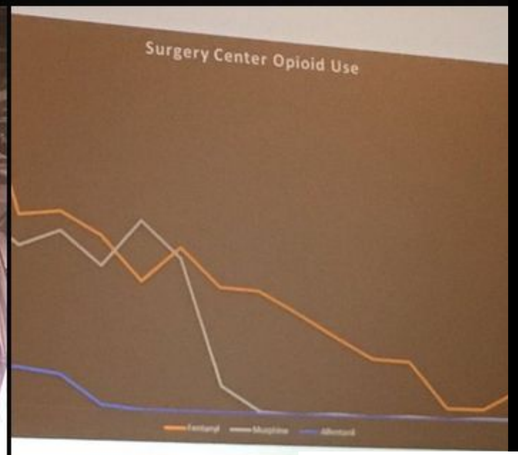
Redis L... @RedisLabs Following
Redis Enterprise in-memory database platform for real-time applications

Philip Howard @InfoPhil Follow

Feb 2017: 1st line of SuiteSparse:GraphBLAS
Nov 2017: GraphBLAS v1, sequential
July 2018: RedisGraph v1
July 2019: GraphBLAS v3, OpenMP
Oct 2020: RedisGraph Award by Bloor Research
Jan 2021: GraphBLAS v4, up to 200x faster than v1
Jan 2022: ~100KLOC, ~40KLOC test suite

Dave Nielsen @davenielsen · Jan 14, 2020

Matthew Goos @m4g005 explains how @MDmetrix uses RedisGraph to track medical treatments so hospitals can reduce overmedicating patients and reduce drug addiction #RedisDay #Seattle



@g_korland

"ThinkData has seen immense performance improvements since adopting RedisGraph, including a 62% decrease in total request time for functions such as text-query catalog searching."
— Brendan Stennett, CTO, ThinkData
@bthinkdata

redislabs.com/case-studies/t...

#redis #redisgraph

10:12 AM · Oct 20, 2020 · Twitter for Android

fiverr.

"RedisGraph offered a great mix of maturity, feature completeness, interoperability with the languages that we use, price, and performance."

— *Brendan Stennett, CTO, ThinkData*

THINKDATA WORKS

LINKURIOUS

Solutions Product Why Linkurious Resources

Product

Visualize your RedisGraph data with Ogma

January 4, 2022

"We've been using Redis for over a decade and it has been great to see the success our teams are achieving with it. With the simplicity and scale that RedisGraph is able to deliver with its multi-tenant architecture and the compactness of the data storage, we've been able to reduce our infrastructure footprint quite considerably. With support for the Cypher Query Language, the switch from our existing graph implementation in favor of RedisGraph was straightforward. Now, with the expected increase of the speed of our queries, we can get the insights we rely on faster than ever."

Emre Sokullu
CEO and Founder

GROUPS

IBM

Sidy

KINEVIZ GraphXR

GRAPHILEON

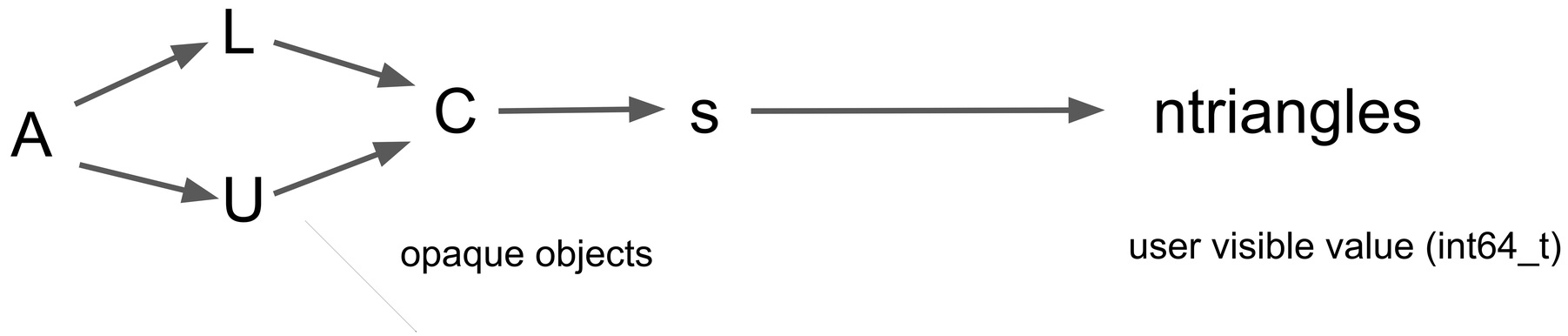
Work in progress, and future work

- LAGraph: graph algorithms using GraphBLAS (nearing v1.0)
- GPU acceleration (CUDA, with J. Eaton and C. Nolet, NVIDIA)
- Julia integration (just announced v0.7), replacing Julia SparseArrays
- further Python integration
- RedisGraph: already lots of users. Future: faster, more features
- JIT for faster user-defined types and operations
- aggressive non-blocking mode, kernel fusion
- x=A\b over a field
- more built-in types (FP16, complex integers, …)
- faster kernels (GrB_mxm for sampled dense-dense matrix multiply)
- matrices with shallow components
- …

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.



GrB_mxm for sampled dense-dense matrix product (SDDMM)

- `C = S .* (A*B')` where S is sparse, A and B are dense, and "`.*`" denotes the Hadamard product
- Required by factor analysis algorithms in machine learning

```
// C = S, a deep copy
GrB_dup (&C, S) ;
// C<struct(C)> *= (A*B')
GrB_mxm (C, C, GrB_TIMES_FP32, GrB_PLUS_TIMES_FP32, A, B, GrB_DESC_ST0) ;
```

- the accumulator is eWiseAdd (set union) not eWiseMult (set intersection) but here they are the same
- but could be faster by exploiting the properties of the problem
- C can be modified in place; no change of sparsity structure (because of the accumulator)
- the Mask M is aliased to C, and used structurally: a powerful property
  - for example: `C<struct(C)> = 1` sets all entries of C to 1, converting it to iso-valued in O(1) time
- needs a specific kernel in SuiteSparse:GraphBLAS for best performance
- kernel fusion of dup and mxm could save some work (aggressive non-blocking exploit), but a special case for GrB_mxm would be the biggest performance gain, and is low-hanging fruit (easy algorithm!)

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 vs Julia API; nearly same in MATLAB
  - faster than the Intel MKL sparse, vastly faster than MATLAB (conventional semiring)
  - 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)

SuiteSparse: GraphBLAS

Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University