SuiteSparse: GraphBLAS

Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University

Smart Grid Center Seminar, Texas A&M, June 8, 2022
GraphBLAS: Graph algorithms in the language of linear algebra

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

SuiteSparse:GraphBLAS:
T. Davis (A&M)
### Figure 4.1.

Visualization of the Texas 2000 network (top) and underneath, the same network with vertex sizes proportional to their betweenness (left), closeness (middle), and eigenvector (right) scores. The table (bottom) lists the top 5 rated vertices under each notion of centrality.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Node</th>
<th>Area</th>
<th>Node</th>
<th>Area</th>
<th>Node</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sterling City 3</td>
<td>WEST</td>
<td>Eastland</td>
<td>NCENT</td>
<td>Fort Worth 10</td>
<td>NCENT</td>
</tr>
<tr>
<td>2</td>
<td>Eastland</td>
<td>NCENT</td>
<td>Abilene 2</td>
<td>WEST</td>
<td>Fort Worth 16</td>
<td>NCENT</td>
</tr>
<tr>
<td>3</td>
<td>Blackwell 2</td>
<td>WEST</td>
<td>De Leon</td>
<td>NCENT</td>
<td>Dallas 8</td>
<td>NCENT</td>
</tr>
<tr>
<td>4</td>
<td>Abilene 2</td>
<td>WEST</td>
<td>Clyde</td>
<td>NCENT</td>
<td>Euless</td>
<td>NCENT</td>
</tr>
<tr>
<td>5</td>
<td>De Leon</td>
<td>NCENT</td>
<td>Blackwell 2</td>
<td>WEST</td>
<td>Gordon</td>
<td>NCENT</td>
</tr>
</tbody>
</table>
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
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

Breadth-first-search (initialization):
q = {source} ; parent = [ size n, all zero ]
parent (source) = source

GraphBLAS BFS: using the ANY-SECOND_D semiring
while (q not empty)
    q<parent> = A'*q // masked matvec
    parent<q> = q // masked assignment

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 BFS: using the ANY-SECONDI semiring
while (q not empty)
  q<parent> = A'*q      // masked matvec
  parent<q> = q           // masked assignment

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 BFS: using the ANY–SECOND semiring
while (q not empty)
    q< parent> = A’ *q      // masked matvec
    parent<q> = q           // masked assignment

level: source

level: 1

level: 2

level: 3

level: 4

level: 5

level: 6

level: 7

source

1

2

3

4

5

6

7

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 BFS: using the \textcolor{green}{\textbf{ANY}}—\textcolor{red}{\textbf{SECONDI}} semiring
while (q not empty)
  q\leftarrow parent = A' * q \quad // masked matvec
  parent(q) = q \quad // masked assignment

\textcolor{red}{\textbf{SECONDI}} multiplier: \( z = A(i,k)*q(k) = k \), the parent node id
additive operator: \textcolor{green}{\textbf{ANY}} function: any(x,y) = x or y, arbitrary choice
### Table of Functions and Descriptions

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

### Semirings for graph algorithms:

- **ANY**, **SECOND**: 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:

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

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

```c
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
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

```c
for each layer k {
    Y = select (((Y +.* W{k}) ++ Bias{k}) > 0) ;
    M = Y > 32 ;
    Y<M> = 32 ;
}
Single-source shortest path (Δ-stepping): \( \text{pathlen} = \text{sssp} (A, \text{source}, \Delta) \)

\[
\begin{align*}
\text{pathlen (src)} &= 0 \\
\text{reach (src)} &= \text{true} \\
\text{s (src)} &= \text{true} \\
\text{AL} &= \text{select } (A \leq \Delta) \\
\text{AH} &= \text{select } (A > \Delta) \\
\text{for } \text{step} = 1:n \\
\text{tmasked<reach,replace>} &= \text{pathlen} \\
\text{tmasked} &= \text{select } (\text{tmasked} < \text{step} \times \Delta) \\
\text{while (tmasked not empty)} \\
\text{tReq} &= \text{AL}' \times \text{tmasked using the (min,+)} \text{ semiring} \\
\text{s<tmasked>} &= \text{true} \\
\text{tless} &= (\text{tReq} .< \text{pathlen}) \text{ using set intersection} \\
\text{if (tless is empty) break} \\
\text{reach<tless>} &= \text{true} \\
\text{tmasked<tless>} &= \text{select } (\text{tReq} < \text{step} \times \Delta) \\
\text{pathlen<tless>} &= \text{tReq} \\
\text{end while} \\
\text{tmasked<s,replace>} &= \text{pathlen} \\
\text{tReq} &= \text{AH}' \times \text{tmasked using the (min,+)} \text{ semiring} \\
\text{tless} &= (\text{tReq} .< \text{pathlen}) \text{ using set intersection} \\
\text{pathlen} &= \min (\text{pathlen}, \text{tReq}) \\
\text{reach<tless>} &= \text{true} \\
\text{reach<s>} &= \text{empty} \\
\text{if (reach is empty) break} \\
\text{clear s} \\
\text{end for}
\end{align*}
\]

- 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: \( \mathbf{C} \odot \mathbf{M} \odot \mathbf{A} \odot \mathbf{B} \)
- C: \( \text{GrB}_\text{mxm} (C, M, \text{GrB\_TIMES\_FP32}, \text{GrB\_MIN\_PLUS\_FP32}, A, B, \text{NULL}) \);
- Python: \( \text{A.mxm} (B, \text{out}=C, \text{mask}=M, \text{accum}=\text{FP32\_TIMES}, \text{semiring}=\text{FP32\_MIN\_PLUS}) \)
- Julia: \( \text{C} = \ast \text{(min, +)}(A, B) \)
  \( \text{mul}(A, B, \text{(min, plus)}, \text{mask}=M, \text{accum}=\ast) \)
- MATLAB: \( \text{C} = \text{GrB.mxm} (C, M, \text{"\'*\''}, A, \text{"min.+''}, B) \);
# GraphBLAS in Julia and Python

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

## Table II

<table>
<thead>
<tr>
<th>Python Function Syntax</th>
<th>Python Operator Syntax</th>
<th>Julia Function Syntax</th>
<th>Julia Operator Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.min_plus(B, out=C)</td>
<td>with FP32.min_plus: C=A*B</td>
<td>C = mul(A,B,(min, plus))</td>
<td>C = *(min,+) (A,B)</td>
</tr>
<tr>
<td>A.eadd(B, out=C, mask=M)</td>
<td>with Mask(M): C=A</td>
<td>B</td>
<td>C = eadd(A,B,mask=M)</td>
</tr>
<tr>
<td>A.emult(B, out=C, accum=FP32.min)</td>
<td>with Accum(FP32.min): C=A*B</td>
<td>emul!(C,A,B,accum=/)</td>
<td>C./=A .* B</td>
</tr>
<tr>
<td>A.assign_col(1,v)</td>
<td>C[1,:]=v</td>
<td>setindex!(A,:,1,v)</td>
<td>A[:,1]=v</td>
</tr>
<tr>
<td>A.extract_row(1, out=v)</td>
<td>v=C[1]</td>
<td>v = getindex(A,1,:)</td>
<td>v = A[1,:]</td>
</tr>
<tr>
<td>A.extract_col(1, out=v)</td>
<td>v=C[:,1]</td>
<td>v = getindex(A,:,1)</td>
<td>v = A[: ,1]</td>
</tr>
<tr>
<td>A.apply(FP64.abs)</td>
<td>A.abs()</td>
<td>map(abs,A)</td>
<td>abs.(A)</td>
</tr>
<tr>
<td>A.reduce_vector(FP64.min)</td>
<td>N/A</td>
<td>reduce(min,A, dims=1)</td>
<td>N/A</td>
</tr>
<tr>
<td>A.reduce_float(FP64.min)</td>
<td>N/A</td>
<td>reduce(min,A)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
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
```

```python
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
    return r
```
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.
    n = A.shape[0]
    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 = Matrix.sparse(A, out=frontier, mask=paths,
                             semiring=FP32.PLUS_FIRST, descr=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 = Matrix.sparse(A, out=frontier, mask=paths,
                                 semiring=FP32.PLUS_FIRST, descr=descriptor.oocr)
        if frontier.nvals == 0:
            break
```

```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.
    n = size (A,1) ;
    ns = length (sources) ;
    paths = GrB (ns, n, 'single', 'by row') ;
    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.irmx (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.irmx (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: betweenness(i) = score for node i.
    # initialize frontier and path count matrix (paths):

    # 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.mmX(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:
    betweenness = Vector.dense (FP32, n, -ns)
    bc.reduce_vector(accum=FP32.PLUS, out=betwecnss, desc=TransposeA)

    return betweenness
```

```matlab
function betweenness = 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):

    % 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.mmX (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.

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>0.58</td>
<td>0.31</td>
<td>0.22</td>
<td>0.34</td>
<td>0.25</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>1.20</td>
<td>0.51</td>
<td>0.35</td>
<td>0.69</td>
<td>3.54</td>
</tr>
</tbody>
</table>

### Performance of Betweenness Centrality:

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>46.4</td>
<td>31.5</td>
<td>10.8</td>
<td>3.0</td>
<td>1.5</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>31.7</td>
<td>23.1</td>
<td>9.25</td>
<td>6.4</td>
<td>34.6</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>25.3</td>
<td>19.8</td>
<td>15.2</td>
<td>5.1</td>
<td>1.0</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>27.8</td>
<td>21.4</td>
<td>17.2</td>
<td>9.4</td>
<td>1.4</td>
</tr>
</tbody>
</table>

### Performance of Triangle Counting:

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>21.8</td>
<td>374.1</td>
<td>79.6</td>
<td>22.2</td>
<td>0.03</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>34.0</td>
<td>930.0</td>
<td>242.6</td>
<td>34.7</td>
<td>0.20</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>1.7</td>
<td>0.53</td>
<td>0.23</td>
<td>0.22</td>
<td>0.05</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>4.5</td>
<td>3.3</td>
<td>1.5</td>
<td>2.0</td>
<td>0.97</td>
</tr>
</tbody>
</table>

### Performance of Single-Source Shortest Paths:

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>7.2</td>
<td>4.9</td>
<td>2.0</td>
<td>0.81</td>
<td>0.21</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>25.3</td>
<td>17.2</td>
<td>8.0</td>
<td>9.4</td>
<td>46.0</td>
</tr>
</tbody>
</table>

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 $C^{<\neg M>} = A' * B$
- saxpy-style, $C = A*B$, $C^{<M>} = A*B$, or $C^{<\neg M>} = A*B$. 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, $f = \max \text{“flops”}$ 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:
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:
Phase 1: scatter $M$ into hash
Phase 2: numerical work
Phase 3/4: count $C(:,1:m)$
Phase 5: gather from hash
Parallel assignment: \[ \text{C}<M> (i,j) = A \quad \text{C}<M> (i,j) += A \]

- 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
GraphBLAS versus the Intel MKL sparse library

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

Table 4. SuiteSparse vs MKL 2022 with the GAP-Twitter matrix
## MATLAB: native sparse matrices vs @GrB objects

<table>
<thead>
<tr>
<th>@GrB vs MATLAB syntax</th>
<th>@GrB advantages / limitations</th>
<th>@GrB speedup relative to MATLAB native on 20 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C = A \times B$</td>
<td>@GrB: any semiring, any mask</td>
<td>30x in MATLAB 2020b, 1x versus GrB v3.3.3 in R2021a, but another 2x to 8x with v7.1.0</td>
</tr>
<tr>
<td>(sparse times sparse)</td>
<td>MATLAB: just plus-times caveat: Tim D wrote them all</td>
<td></td>
</tr>
<tr>
<td>$C(I,J) = A$</td>
<td>Same syntax, more types: sparse int8, int16, …, single complex, …</td>
<td>2x to 1000x</td>
</tr>
<tr>
<td>$C = \text{sparse } (2^{60}, 2^{60})$</td>
<td>MATLAB: too big</td>
<td></td>
</tr>
<tr>
<td>$C = \text{GrB } (2^{60}, 2^{60})$</td>
<td>@GrB: no problem; hypersparse</td>
<td></td>
</tr>
<tr>
<td>$C(M) = A$</td>
<td>MATLAB mask: same syntax, $O(e^2)$</td>
<td>MATLAB: 5 days 500,000x speedup</td>
</tr>
<tr>
<td></td>
<td>@GrB much faster: $O(e \log e)$</td>
<td>GraphBLAS: 0.8 seconds</td>
</tr>
</tbody>
</table>
## MATLAB: native sparse matrices vs @GrB objects

<table>
<thead>
<tr>
<th>computation</th>
<th>MATLAB (seconds)</th>
<th>SuiteSparse (seconds)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=S*x</td>
<td>22.80</td>
<td>2.40</td>
<td>9.49</td>
</tr>
<tr>
<td>y=x*S</td>
<td>16.16</td>
<td>1.16</td>
<td>13.92</td>
</tr>
<tr>
<td>C=S*F</td>
<td>30.61</td>
<td>9.71</td>
<td>3.15</td>
</tr>
<tr>
<td>C=F*S</td>
<td>26.40</td>
<td>1.52</td>
<td>17.32</td>
</tr>
<tr>
<td>C=S'</td>
<td>224.73</td>
<td>22.69</td>
<td>9.91</td>
</tr>
<tr>
<td>C=S+B</td>
<td>15.56</td>
<td>1.51</td>
<td>10.31</td>
</tr>
<tr>
<td>C=S(p,q)</td>
<td>95.62</td>
<td>15.95</td>
<td>6.00</td>
</tr>
</tbody>
</table>

Table 3. SuiteSparse vs MATLAB R2021a 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, more to come
- 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)
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
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

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

"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
GROU.PS
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
- $x = A \backslash b$ 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 dependency DAG with fusion and lazy evaluation
- no need to form L, U, and C
- not yet exploited in SuiteSparse:GraphBLAS. In progress.

```
A  \rightarrow L \rightarrow C \rightarrow s \rightarrow ntriangles
```

opaque objects

user visible value (int64_t)
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
// 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