SuiteSparse:GraphBLAS: a parallel implementation of the GraphBLAS specification

Tim Davis, Texas A&M University

June 29, 2020

SIAM MDS'20: Linear Algebraic Tools for Graph Computation

SuiteSparse:GraphBLAS matrix data structure

- both CSC (compressed sparse column form) and CSR. Default is CSR.
- sparse ($O(n + |A|)$ space): a dense vector of sparse vectors
- or hypersparse ($O(|A|)$): a sparse vector of sparse vectors
- **ZOMBIES!**: an entry marked for deletion (negated row index so binary search still OK)
- *pending tuples*: an entry in an ordered list, waiting to inserted
- values: just about anything (each scalar of fixed size)
- no value given to the implicit entry

Parallel algorithms (assume CSC)

- matrix-matrix and matrix-vector multiply. Three primary methods:
  - saxpy-style (Gustavson and hash)
  - hash method (Nagasaka, Matsuoka, Azad, Buluç (2018); extended here with a concurrent data structure via atomics)
  - dot, with mask: $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}\mathbf{B}'$
  - dot, without mask but $\mathbf{C}$ dense (in-place)
- dot-style with mask: each M_{ij}
- saxpy-style: 4 kinds of tasks, all can be used in a single $\mathbf{C} = \mathbf{A}\mathbf{B}$. Tasks selected based on amount of work per column, and number of threads. Let f be the flop count.
- Parallel assignment: `C(I,J)=A`, with **ZOMBIES!** ... and pending tuples too.

Parallel matrix multiply (assume CSC)

Four kinds of saxpy tasks, each with 3 variants: no mask, M, and $\neg M$.

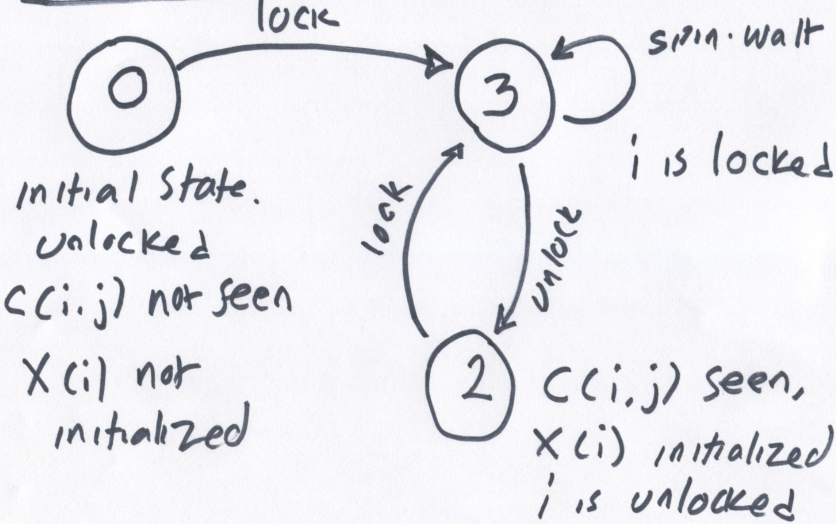
- coarse Gustavson task: `C(:,j1:j2) = A * B (:,j1:j2)` by one thread with $O(n)$ workspace, owns columns `j1:j2`. Time is $O(n+f)$.
- fine Gustavson task: multiple threads cooperate on a single column: `C(:,j) = A * B (:,j)`. Threads share a single $O(n)$ workspace; using atomics. Time is $O(n/p+f/p)$.
- coarse hash task: `C(:,j1:j2) = A * B (:,j1:j2)`, by one thread using $O(h)$ workspace, $h << n$, where $h=4f$. Time is $O(f)$ assuming few collisions.
- fine hash task: multiple threads cooperate on a single column: `C(:,j) = A * B (:,j)`. Threads share $O(h)$ workspace, $h << n$, where $h=4f$. Time is $O(f/p)$ assuming few collisions.

Fine Gustavson task: with no mask

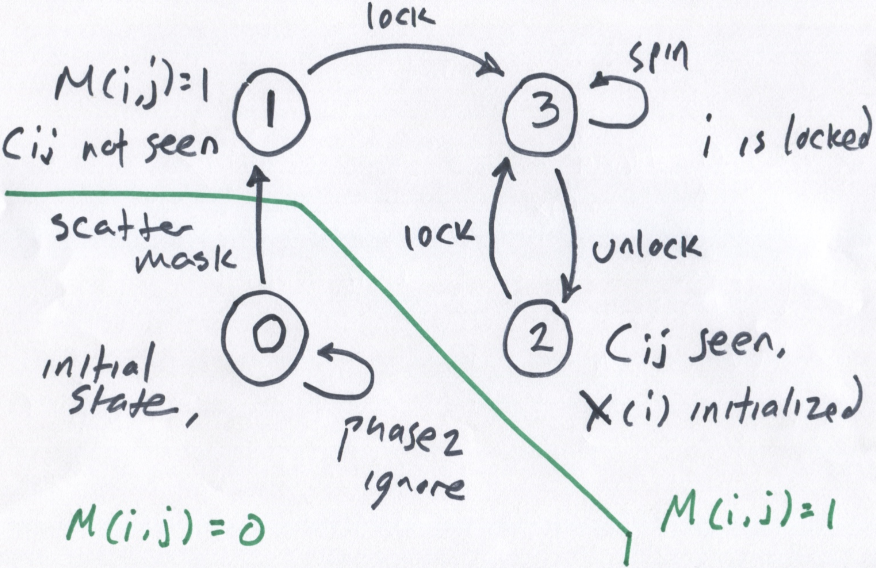
To compute both pattern and values of `C`.

- concurrent data structure: two arrays of size n: `X` for numerical values, `F` for state (`int8`) for n concurrent finite-state machines. `F` is calloc'd so starts as zero.
- phase1: scatter the mask (no mask for this case however).
- phase2: scatter/sum values in `X`, using machine `F`. Using atomics. Time is $O(f)$ assuming no spin-wait.
- phase3: count `c(j) = nnz(C(:,j))` for each `j`. Time is $O(n/p)$ with p threads for a single column `j`.
- phase4: `Cp = cumsum(c)` for column pointers, for all `j` Time is $O(n/p)$ for entire matrix.
- phase5: gather from `X` to create values and pattern of `C(:,j)`.

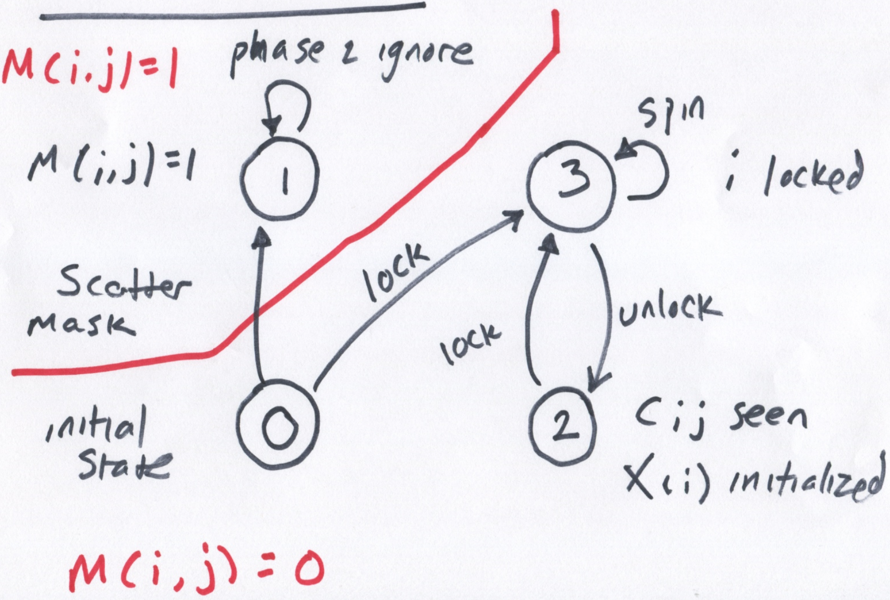
Fine Gustavson task : no mask



Fine Gustavson task: with mask M



Fine Gustavson task : with !M



Fine hash tasks

- F has size $4f$, 64-bit integers: `F[k]=(h,f)`.
- `f`: lowest 2 bits: for 4-state finite-state machine
- `h`: upper 62 bits: $\mathtt{h}$ = row index $i + 1$ occupying the hash entry `F[k]`
- `F[k]=(h,f)` can be read/swapped/written in a single atomic operation
- The four states:
  - `h=0`, `f=0`: unlocked, unoccuppied.
  - `h=i+1`, `f=1`: unlocked, occuppied by row i. `C(i,j)` not seen, or ignored. `X(k)` not initialized.
  - `h=i+1`, `f=2`: unlocked, occuppied by row i. `C(i,j)` seen. `X(k)` initialized.
  - `h=anything`, `f=3`: locked, occuppied by something.
- simple hash function; linear probing if first entry occuppied.
- hash table size based on flop count f for that column `j`.

Parallel matrix-matrix performance

`C=A*A` for matrix `ND/nd3k` ($n = 9000$, with 3.3 million entries): All results on 20-core Intel Xeon. MATLAB R2018a.

time in seconds:

method	single	double	single complex	double complex
MATLAB	-	3.76	-	7.90
GrB:1	3.12	3.35	4.63	4.92
GrB:20	0.22	0.24	0.33	0.31
GrB:40	0.18	0.21	0.27	0.29

speedup:

vs GrB:1	17.3	16.0	17.1	17.0
vs MATLAB	20.9	18.0	29.3	27.3

Parallel matrix - sparse vector performance

`C=A*x` for `Freescale2` ($n = 3$ million, 14.3 million entries). `x` is 50% nonzero:

time in seconds:

method	single	double	single complex	double complex
MATLAB	-	0.228	-	0.288
GrB:1	0.090	0.096	0.121	0.174
GrB:20	0.010	0.014	0.015	0.024
GrB:40	0.011	0.014	0.015	0.029

speedup:

vs GrB:1	8.2	6.9	8.1	6.0
vs MATLAB	20.7	16.3	19.2	9.9

Parallel C(I,J)=A

- 128 variants: $\mathbf{C(i,j)}\langle\mathbf{M}\rangle = \mathbf{C(i,j)} \odot \mathbf{A}$
  - **M** present, or not
  - mask complemented, or not
  - mask structural, or not
  - REPLACE option enabled, or not
  - accumlator $\odot$ present, or not
  - **A**: scalar or matrix
  - **S** matrix constructed: see below
- implemented with 30 main kernels, including 7 special cases:
  - `C=x` with scalar `x`
  - `C=A`
  - `C+=x` when `C` is dense
  - `C+=A` when `C` is dense
  - `C<M>=x`, when `C` is dense
  - `C<A>=A`, when `C` is dense
  - `C<M,struct>=A`, when `A` is dense and `C` is empty

Parallel C(I,J)=A, simple case

Basic case: no mask, no accumulator, no replace option, mask not complemented, `I` and `J` arbitray lists of indices, `C` and `A` sparse, ...

- 1st pass: structural extraction to compute `S`
  - extract `S=C(I,J)` where `S(i,j)` is the not *value* of `C(I[i],J[j])`, but its *position* in the data structure for `C`.
  - `S` and `A` have the same size.
  - `S(i,j)` tells where `A(i,j)` goes. Let `p=S(i,j)`
- 2nd pass: do updates, make zombies, count pending tuples
  - `S(i,j)` and `A(i,j)` both present. update: `C(p) = A(i,j)`. A **ZOMBIE** might come back to life!
  - `S(i,j)` not present; `A(i,j)` present. `A(i,j)` becomes a *pending tuple*, waiting to be inserted. Lazy.
  - `S(i,j)` present; `A(i,j)` *not* present. Entry at `C(p)` must be deleted ... becomes a **ZOMBIE!**
- cumulative sum of pending tuples found per thread
- 3rd pass: each thread puts its pending tuples in common list

Parallel C=A(I,J) performance

`A` is square, n=100 million, 1 billion entries.
`C = A(I,J)`, where `I=randperm(n,n/10)`, also `J`.
Note: GrB algorithm not presented in this talk. Results here to compare with next slide on `C(I,J)=A`.

	threads	time (sec)	speedup vs MATLAB	speedup vs GrB:1
MATLAB	1	8.87	1	1.07
GrB	1	9.48	0.94	1
GrB	5	2.70	3.28	3.51
GrB	10	1.90	4.66	4.98
GrB	20	1.48	5.98	6.39
GrB	40	1.35	6.56	7.01

Parallel C(I,J)=A performance

Same `A`, `I`, and `J` as last slide.
`A(I,J) = 2*A(I,J)` (no change to pattern).

	threads	time (sec)	speedup vs MATLAB	speedup vs GrB:1
MATLAB	1	> 24 hours	-	-
GrB	1	32.02	$> 2,700$	1
GrB	5	8.88	$> 10,000$	3.60
GrB	10	5.91	$> 15,000$	5.42
GrB	20	4.78	$> 18,000$	6.69
GrB	40	4.43	$> 20,000$	7.32

GrB: about 3x the time for `C=A(I,J)` but this expression starts with that; remainder is `A(I,J)=2*C`.
MATLAB: still running after 24+ hours. GrB using same syntax, via MATLAB `@GrB` interface.

Parallel C(I,J)=A performance

Same `A`, `I`, and `J` as last slide.
`A(I,J) = 2*A(I+1,J+1)` (changes pattern).

	threads	time (sec)	speedup vs MATLAB	speedup vs GrB:1
MATLAB	1	-	-	0
GrB	1	55.25	-	1
GrB	5	14.79	-	3.74
GrB	10	9.59	-	5.76
GrB	20	7.47	-	7.39
GrB	40	7.04	-	7.84

MATLAB: first experiment still running after 24 hours.

Parallel performance: betweenness centrality

Dominated by matrix-matrix multiply (one matrix 4-by-n and dense)

time in seconds

	threads			
matrix	1	10	20	40
kron	1076.9	137.7	74.6	42.3
urand	1405.8	107.2	70.3	63.2
twitter	328.1	35.1	17.5	13.0
web	91.3	13.4	8.3	7.8
road	52.5	51.9	54.1	61.4

speedup

kron	1	7.8	14.4	25.5
urand	1	13.1	20.0	22.2
twitter	1	9.3	18.7	25.2
web	1	6.8	11.0	11.7
road	1	1.0	1.0	0.9

Parallel performance: pagerank

Domininated by matrix-vector multiply (dense vector).

time in seconds

	threads			
matrix	1	10	20	40
kron	372.7	40.8	22.2	21.8
urand	378.9	42.2	27.7	27.8
twitter	286.5	32.0	17.9	17.3
web	90.0	10.9	8.9	8.9
road	12.9	1.8	1.4	1.4

speedup

kron	1	9.1	16.8	17.1
urand	1	9.0	13.7	13.6
twitter	1	9.0	16.0	16.6
web	1	8.3	10.1	10.1
road	1	7.2	9.2	9.2