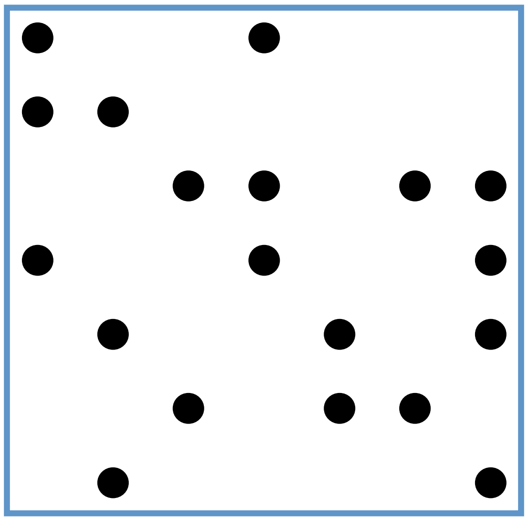
Parallel Sparse Matrix Indexing and Assignment

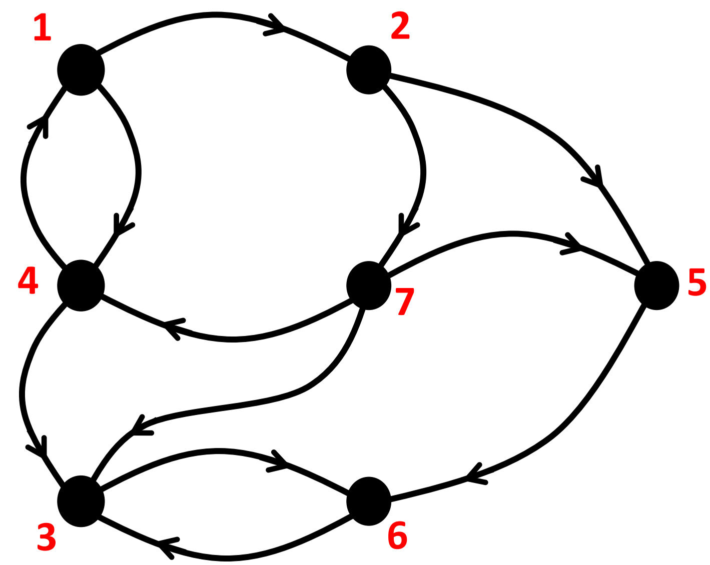
Aydın Buluç
Lawrence Berkeley National Laboratory

John R. Gilbert
University of California, Santa Barbara

Sparse adjacency matrix and graph



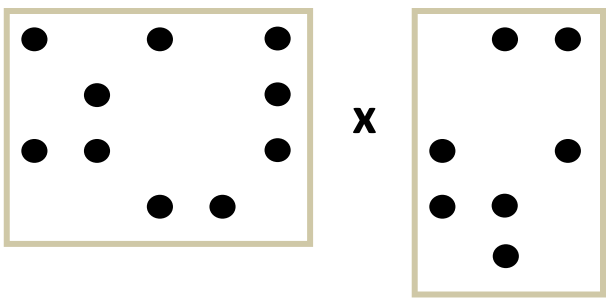
A^T



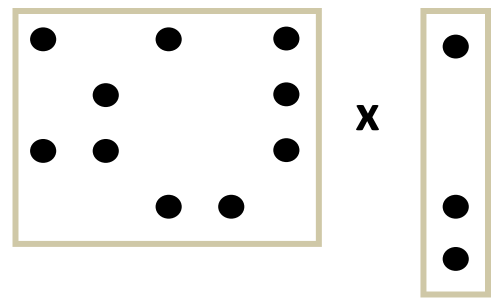
- Every graph is a sparse matrix and vice-versa
- Adjacency matrix: sparse array w/ nonzeros for graph edges
- Storage-efficient implementation from sparse data structures

Linear-algebraic primitives for graphs

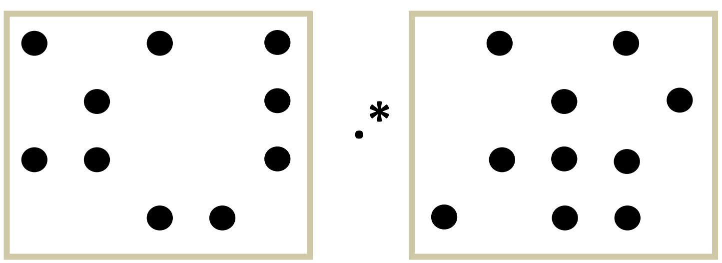
Sparse matrix-matrix Multiplication (SpGEMM)



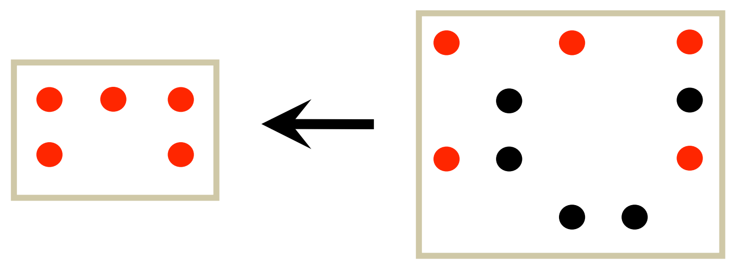
Sparse matrix-sparse vector multiplication



Element-wise operations



Sparse Matrix Indexing



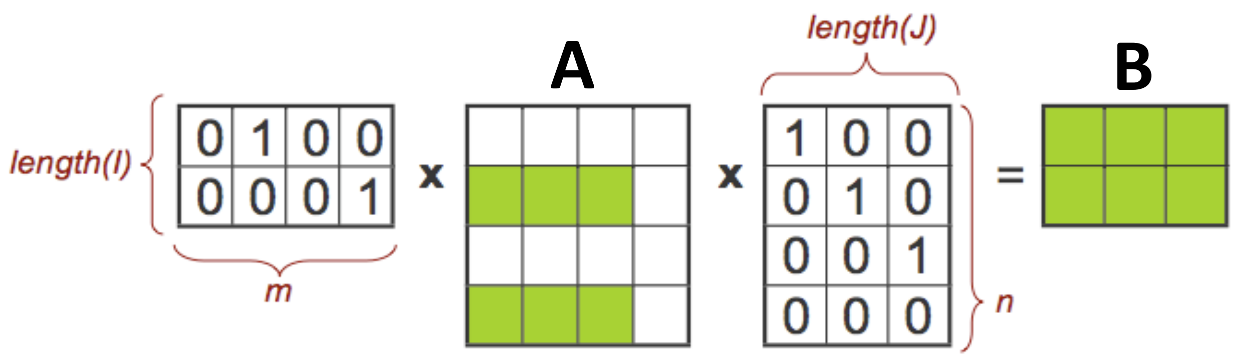
Matrices on semirings, e.g. ($\times$, +), (and, or), (+, min)

Indexed reference and assignment

Matlab internal names: **subsref, subsasgn**
For sparse special case, we use: **SpRef, SpAsgn**

SpRef:	`B = A(I,J)`
SpAsgn:	`B(I,J) = A`

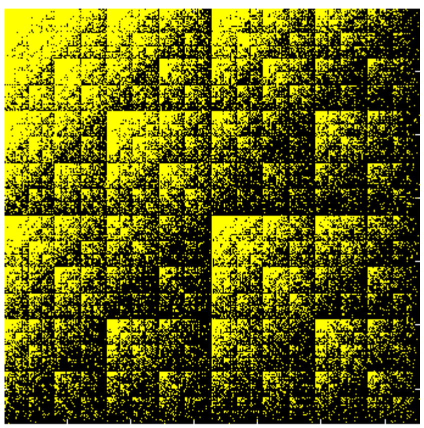
`A,B`: sparse matrices
`I,J`: vectors of indices



SpRef using mixed-mode sparse matrix-matrix multiplication (**SpGEMM**). Ex: B = A([2,4], [1,2,3])

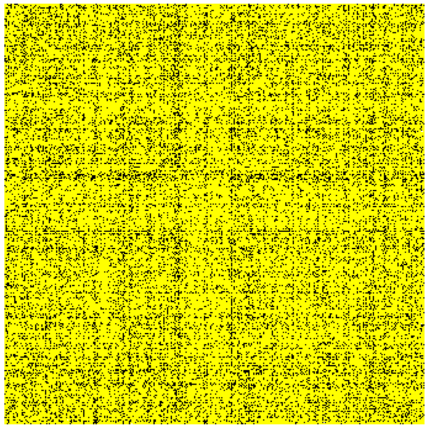
Why are **SpRef/SpAsgn** important?

Subscripting and colon notation:
⇒ Batched and vectorized operations
⇒ High Performance and parallelism.



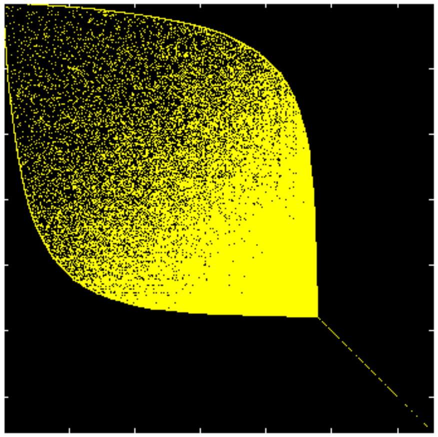
A=rmat(15)

− Load balance hard
± Some locality



A(r,r) : r random

+ Load balance easy
− No locality

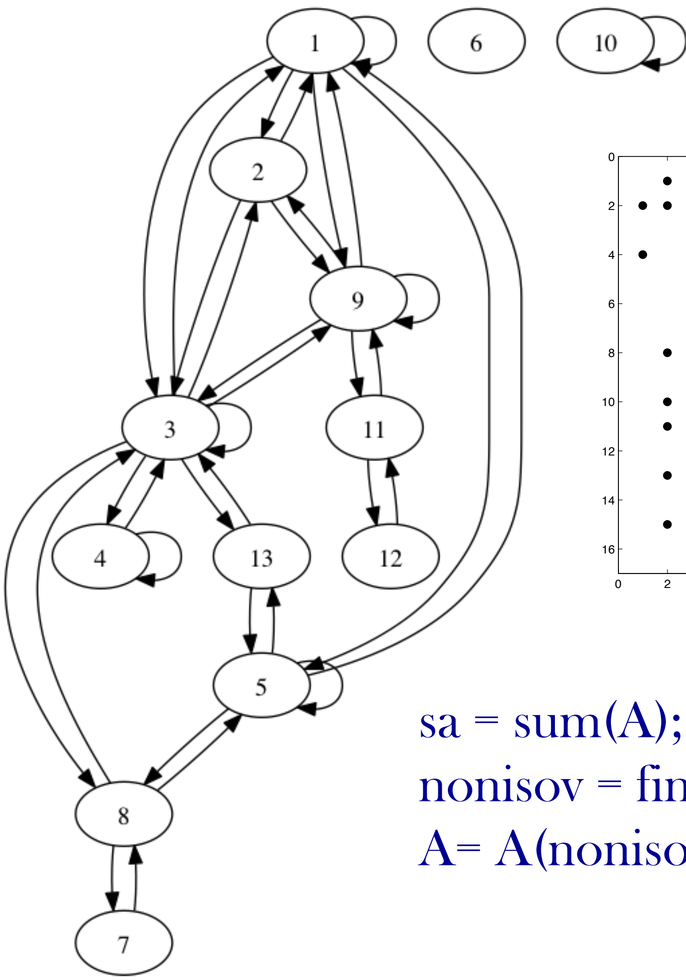


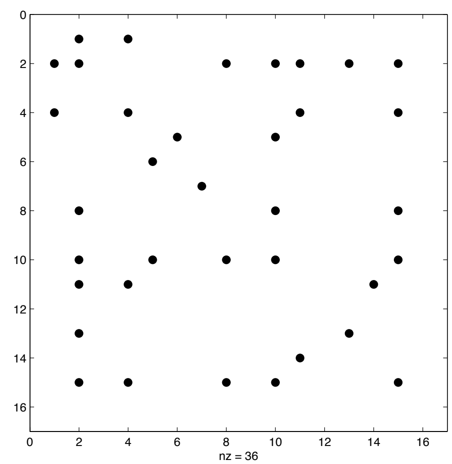
A(r,r) : r=symrcm(A)

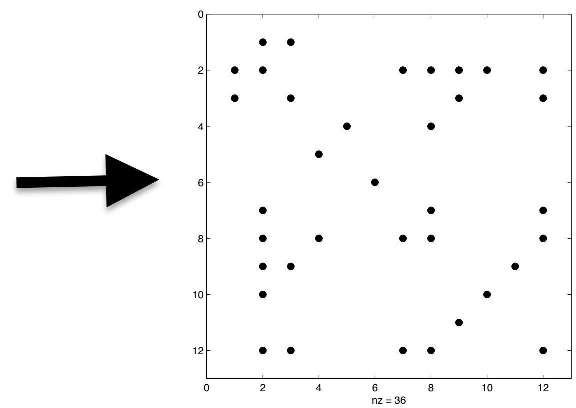
− Load balance hard
+ Good locality

More applications

Prune isolated vertices; plug-n-play way (Graph 500)



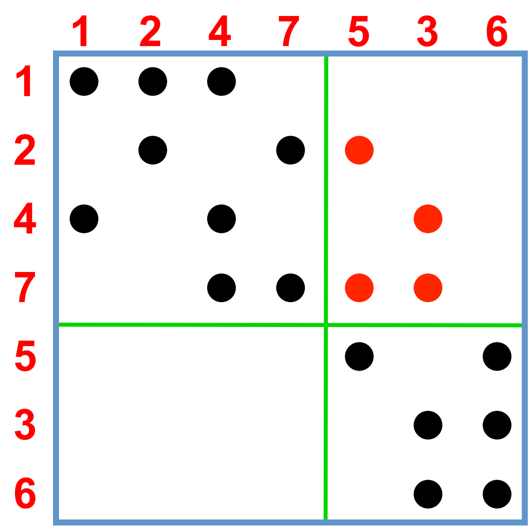




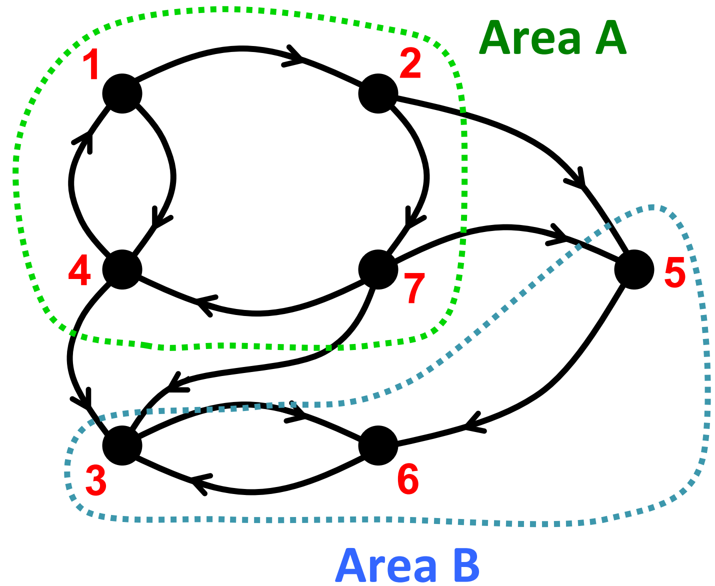

```
sa = sum(A);  // A is symmetric, for undirected graph
nonisov = find(sa>0);
A= A(nonisov, nonisov);   // keep only connected vertices
```

More applications

Extracting (induced) subgraphs



PAP^T



- Per-area analysis on power grids
- Subroutine for recursive algorithms on graphs

Sequential algorithms

```
function B = spref(A,I,J)
R = sparse(1:length(I),I,1,length(I),size(A,1));
Q = sparse(J,1:length(J),1,size(A,2),length(J));
B = R*A*Q;
```

$$T_{spref} = flops(R \cdot A) + flops(RA \cdot Q) = nnz(R \cdot A) + nnz(RA \cdot Q) = O(nnz(A))$$

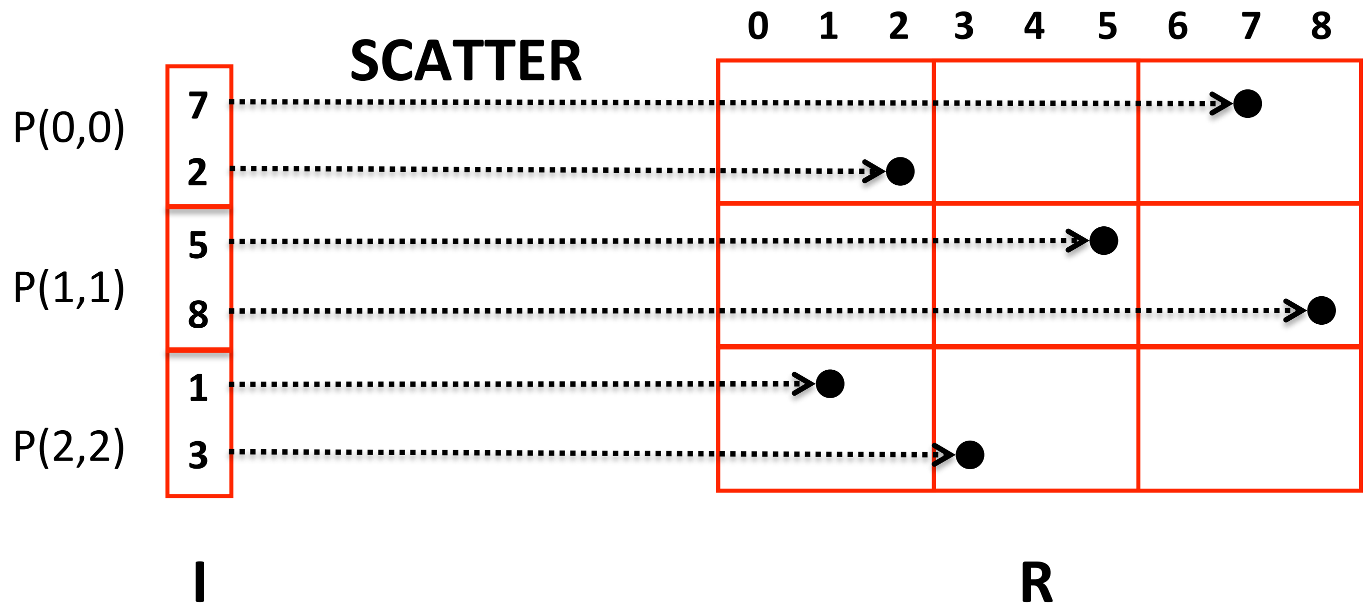
```
function C = spasgn(A,I,J,B)
[ma,na] = size(A);
[mb,nb] = size(B);
R = sparse(I,1:mb,1,ma,mb);
Q = sparse(1:nb,J,1,nb,na);
S = sparse(I,I,1,ma,ma);
T = sparse(J,J,1,na,na);
C = A + R*B*Q - S*A*T;
```

$$A + \begin{pmatrix} 0 & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 & 0 \\ 0 & A(I,J) & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$T_{spasgn} = O(nnz(A))$$

Parallel algorithm for SpRef

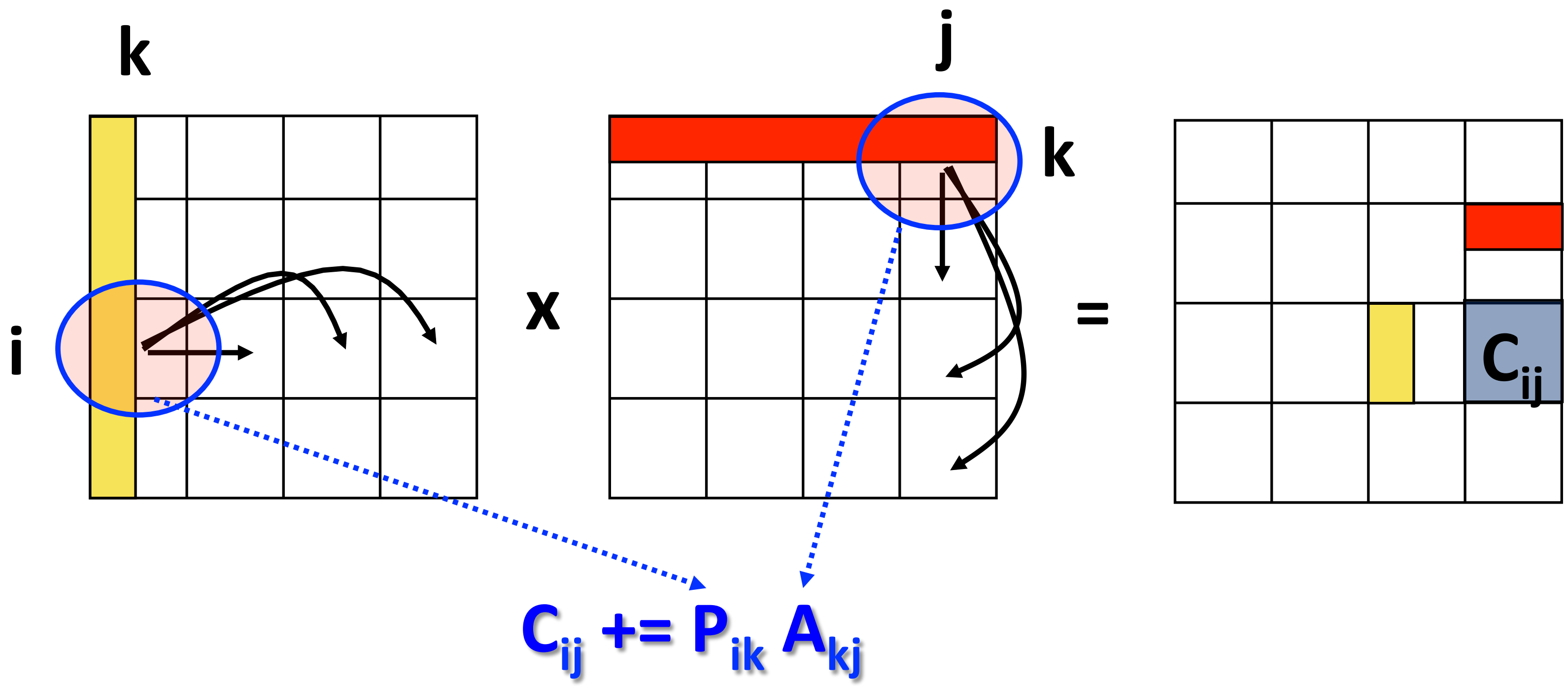
1. Forming R from I in parallel, on a 3x3 processor grid



- Vector distributed only on diagonal processors; for illustration.
- Full (2D) vector distribution: SCATTER → ALLTOALLV
- Forming Q^T from J is identical, followed by Q=Q^T.Transpose()

Parallel algorithm for SpRef

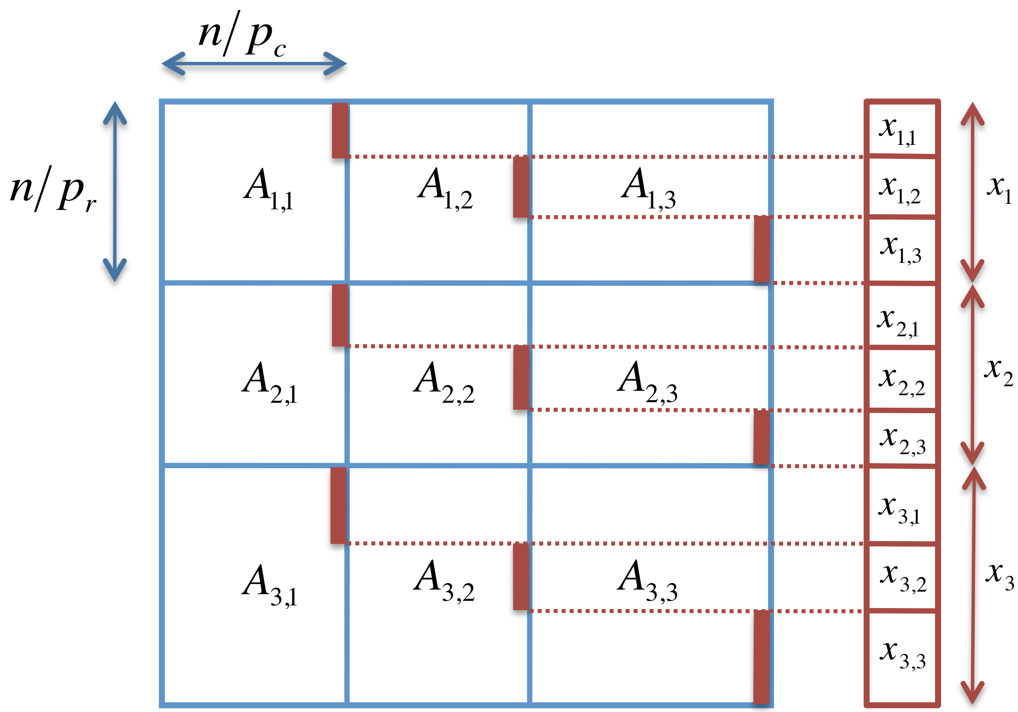
2. **SpGEMM** using memory-efficient Sparse SUMMA.



Minimize temporaries by:

- Splitting local matrix, and broadcasting multiple times
- Deleting P (and A if in-place) immediately after forming C=P*A

2D vector distribution



Matrix/vector distributions, interleaved on each other.

Default distribution in **Combinatorial BLAS**.

- Performance change is marginal (dominated by **SpGEMM**)
- Scalable with increasing number of processes
- No significant load imbalance

Complexity analysis

SpGEMM:

$$T_{comp} \approx \Theta\left(\frac{nnz(A)}{p} \cdot \log\left(\frac{length(I)}{p} + \frac{length(J)}{p} + \sqrt{p}\right)\right)$$

$$T_{comm} = \Theta\left(\alpha \cdot \sqrt{p} + \beta \cdot \frac{nnz(A)}{\sqrt{p}}\right)$$

Dominated by **SpGEMM**

Bottleneck: bandwidth costs

Speedup: $\Theta\left(\sqrt{p}\right)$

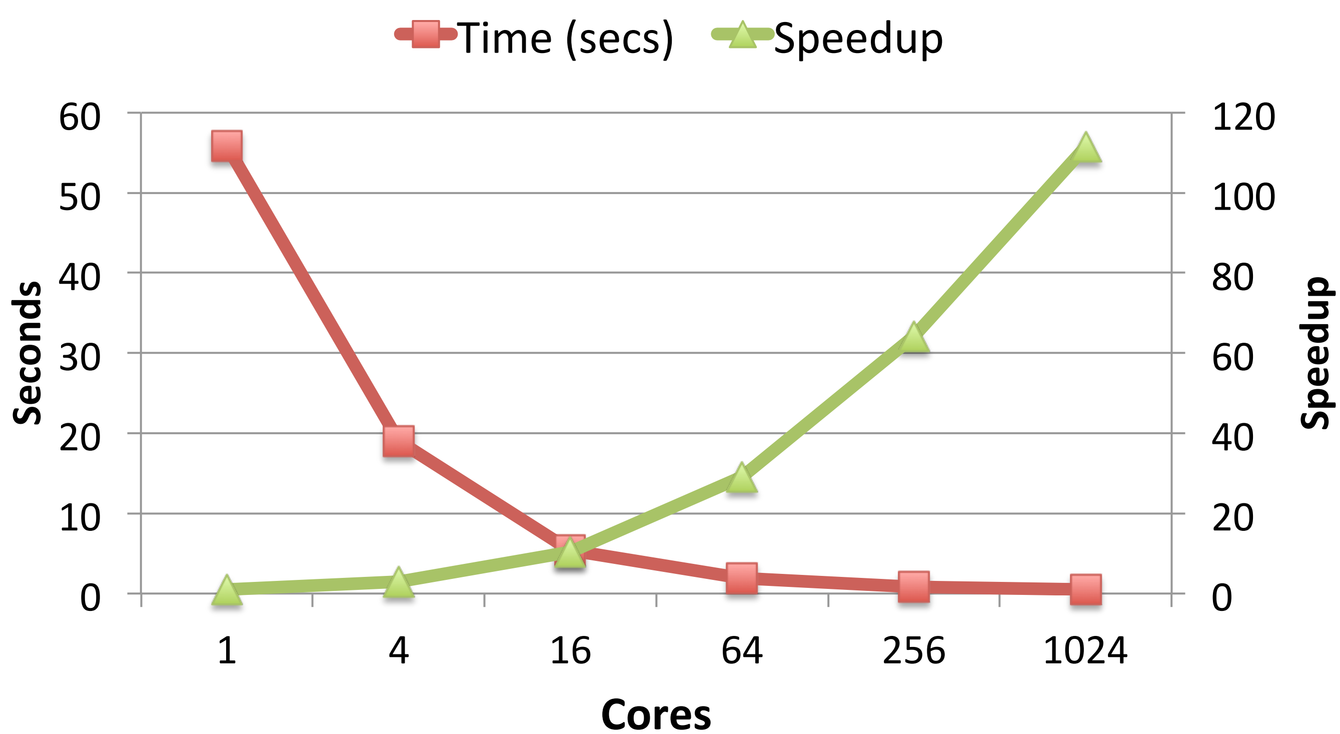
Matrix formation:

$$\Theta\left(\alpha \cdot \log(p) + \beta \cdot \frac{length(I) + length(J)}{\sqrt{p}}\right)$$

Assumptions:

- The triple product is evaluated from left to right: B=(R*A)*Q
- Nonzeros uniformly distributed to processors (chicken-egg?)

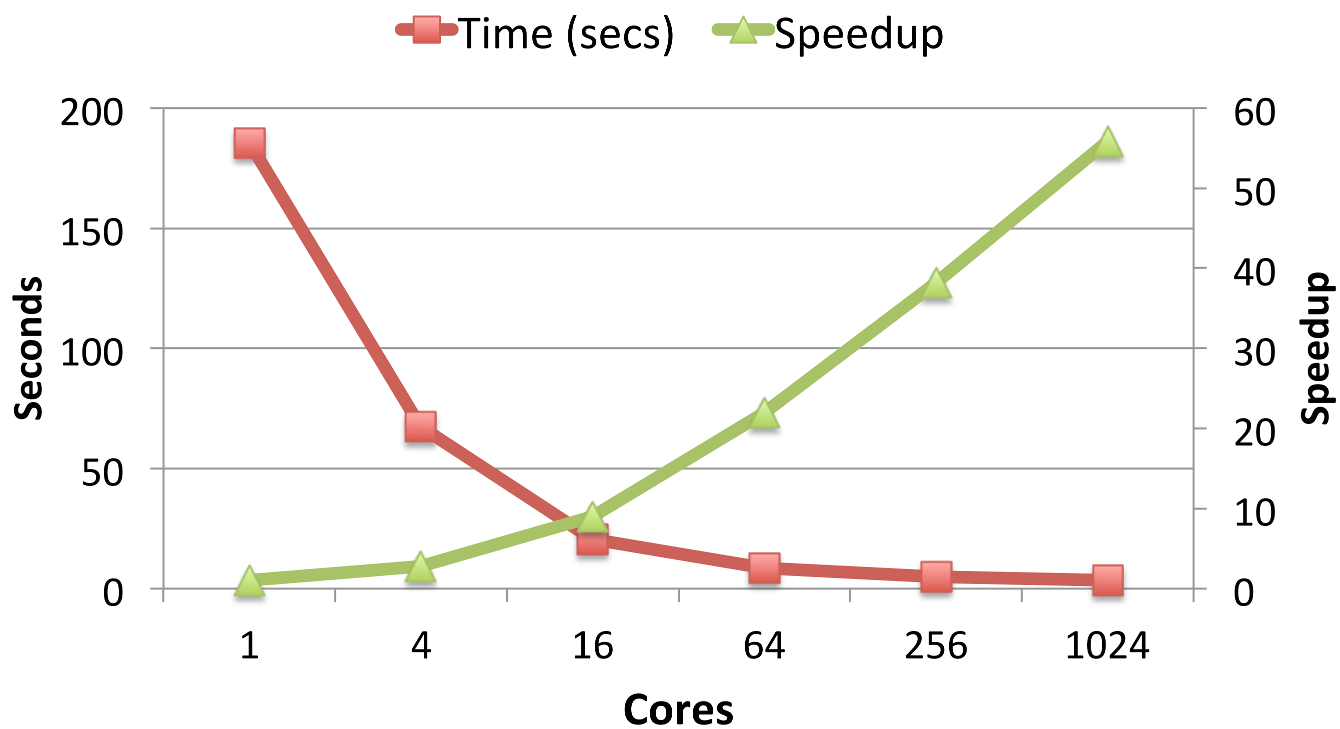
Strong scaling of **SpRef**



random symmetric permutation ⇔ relabeling graph vertices

- RMAT Scale 22; edge factor=8; a=.6, b=c=d=.4/3
- Franklin/NERSC, each node is a quad-core AMD Budapest

Strong scaling of SpRef



Extracts 10 random (induced) subgraphs, each with |V|/10 vert.

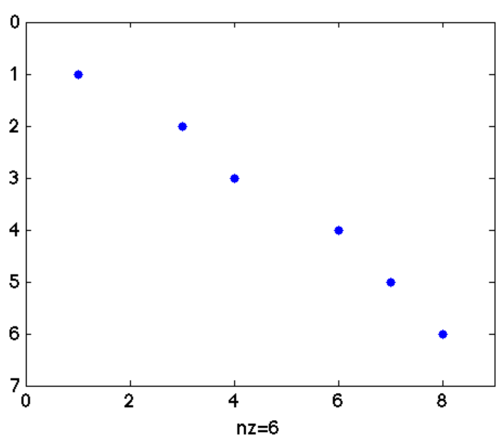
Higher span → Decreased parallelism → Lower speedup

Conclusions

- Parallel algorithms for **SpRef** and **SpAsgn**
- Systemic algorithm structure imposed by **SpGEMM**
- Analysis made possible for the general case
- Good strong scaling for 1000-way concurrency
- Many applications on sparse matrix and graph world.

Caveat: Avoid load imbalance by indexing non-monotonically

$$I = \begin{bmatrix} 1 \\ 3 \\ 4 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$



$$I = \begin{bmatrix} 7 \\ 3 \\ 8 \\ 1 \\ 6 \\ 4 \end{bmatrix}$$