Parallel algorithms for sparse matrix product, indexing, and assignment

Aydın Buluç
Lawrence Berkeley National Laboratory
February 8, 2012

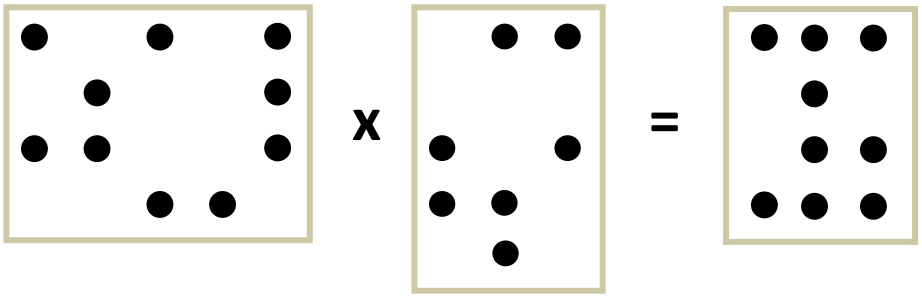
Joint work with John R. Gilbert (UCSB)

With inputs from G. Ballard, J. Demmel, O. Schwartz, E. Solomonik.

Linear-algebraic primitives

Sparse matrix-matrix multiplication (SpGEMM)

```
C = A * B
```

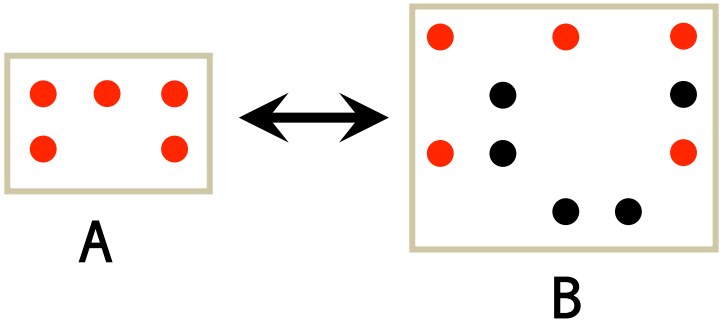


Sparse matrix indexing (SpRef)

```
A = B(I,J)
```

Sparse matrix assignment (SpAsgn)

```
B(I,J) = A
```

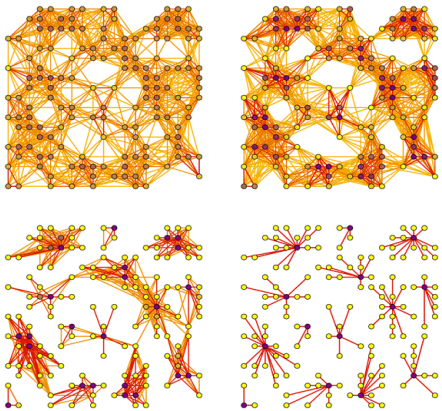


`A,B:` sparse matrices with arbitrary nonzero distribution

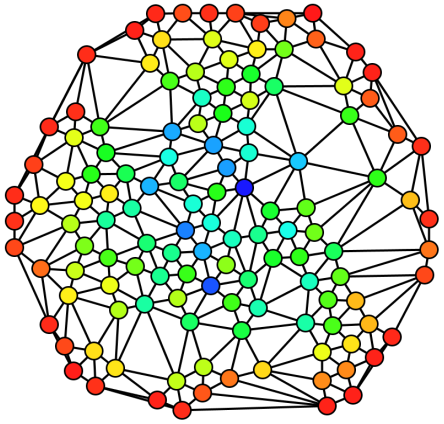
`I,J:` vectors of indices (arbitrary order and length)

Applications of Sparse GEMM

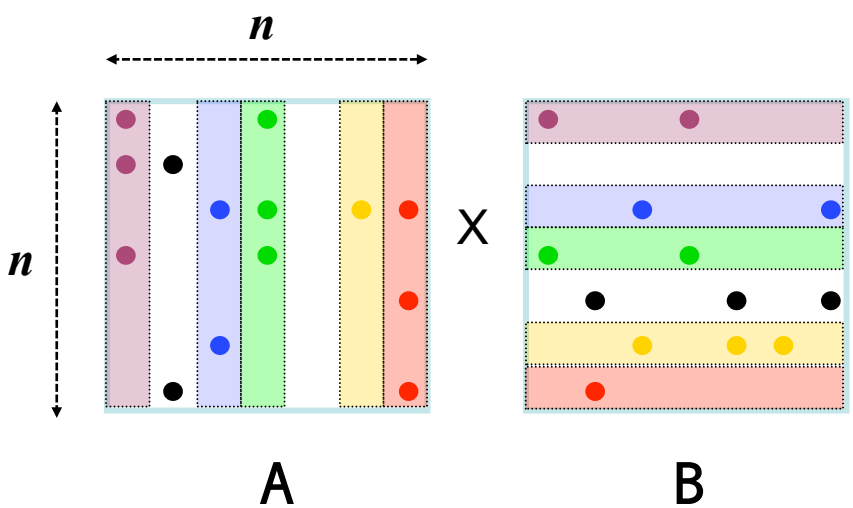
- *Graph clustering* (*MCL*, peer pressure)
- Subgraph / submatrix indexing
- Shortest path calculations
- *Betweenness centrality*
- Graph contraction
- Cycle detection
- Multigrid interpolation & restriction
- Colored intersection searching
- Applying constraints in finite element computations
- Context-free parsing ...



```
Loop until convergence
A = A * A;                % expand
A = A .^ 2;               % inflate
sc =  1./sum(A, 1);
A = A * diag(sc);         % renormalize
A(A < 0.0001) = 0;        % prune entries
```



Some terminology



n =7
nnz(A) =14, nnz(B) =12
flops =22
nzc(A) =6, nzr(B) =6
ni = 5

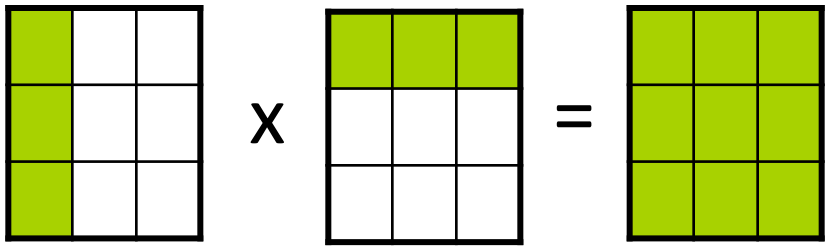
nnz: number of nonzeros
flops: number of nonzero arithmetic operations required
nzr: number of rows that are not entirely zero
nzc: number of columns that are not entirely zero
ni: number of indices i for which A(:,i)≠0 and B(i,:)≠0

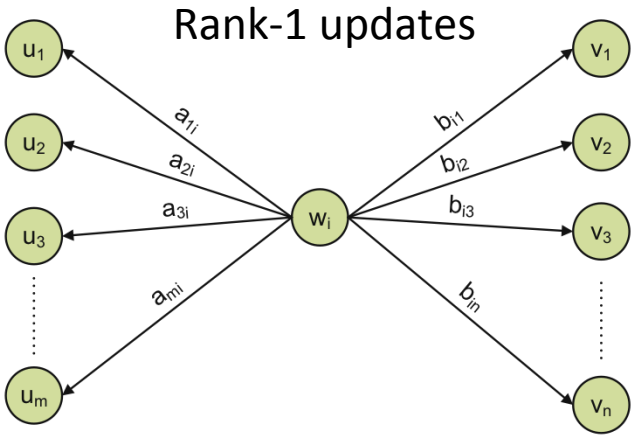
If nonzeros of A and B are i.i.d with ***d*** nonzeros per row/column, then $\boldsymbol{nnz = dn}$ and $\boldsymbol{flops = d^2n}$

Organizations of matrix multiplication

1) <u>outer product:</u>

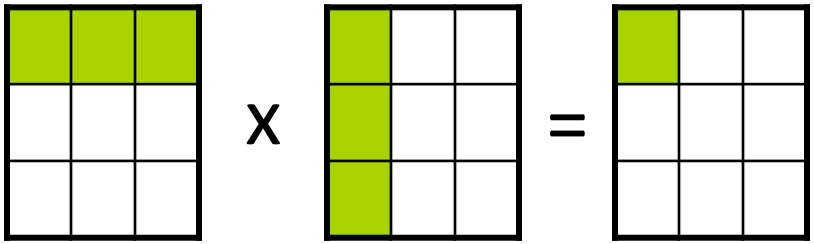
```
for k = 1:n
  C = C + A(:, k) * B(k, :)
```

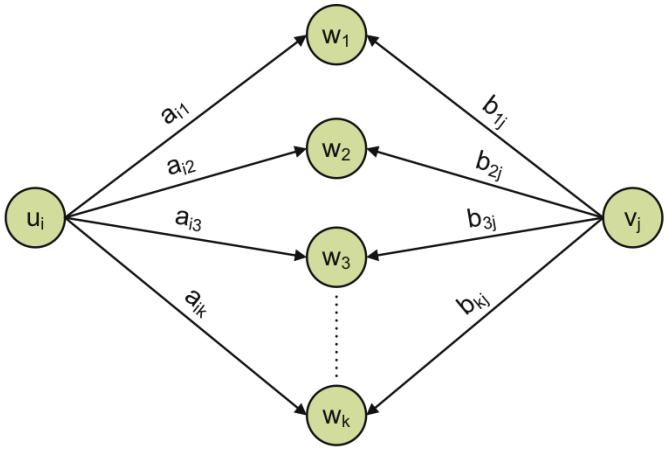




2) <u>inner product:</u>

```
for i = 1:n
  for j = 1:n
    C(i, j) = A(i, :) * B(:, j)
```

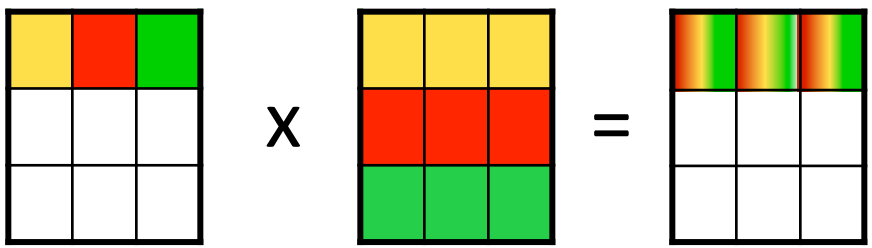


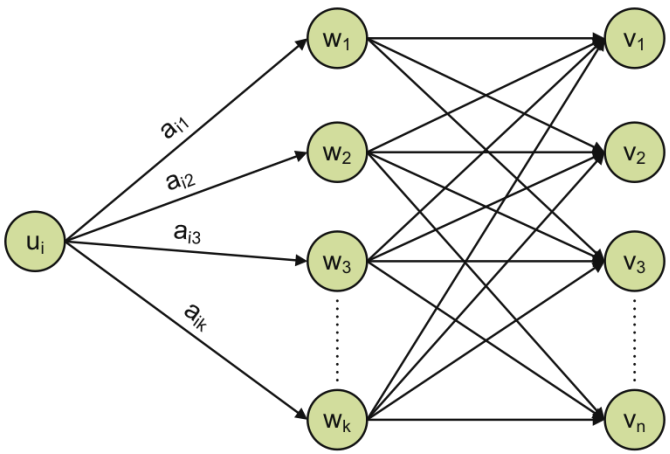


Organizations of matrix multiplication

3) Row-by-row formulation:

```
for i = 1:n
  forall k s.t. A(i,k) ≠ 0
    C(i,:) = C(i,:) + A(i,k) * B(k,:)
```





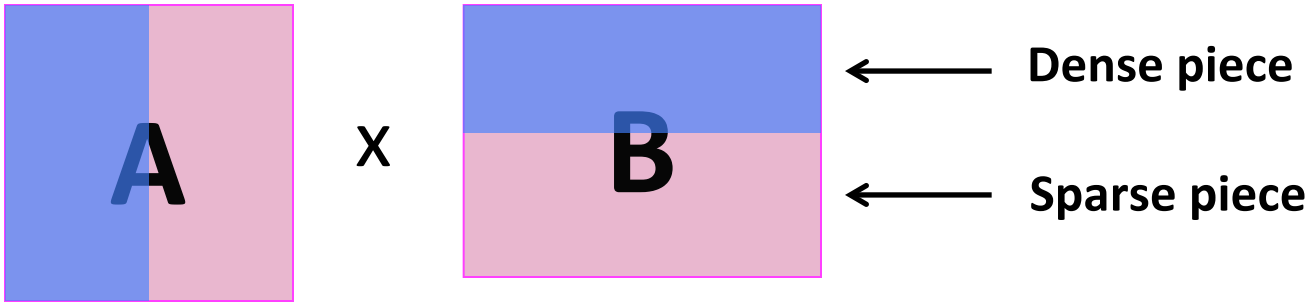
- ***Complexity: O(n+nnz+flops)***
- Due to Gustavson (1978), implemented in Matlab and CSparse.
- Fastest general purpose algorithm in serial.
- "flops-optimal" for ***flops > n, nnz***

Alternative algorithm on a ring

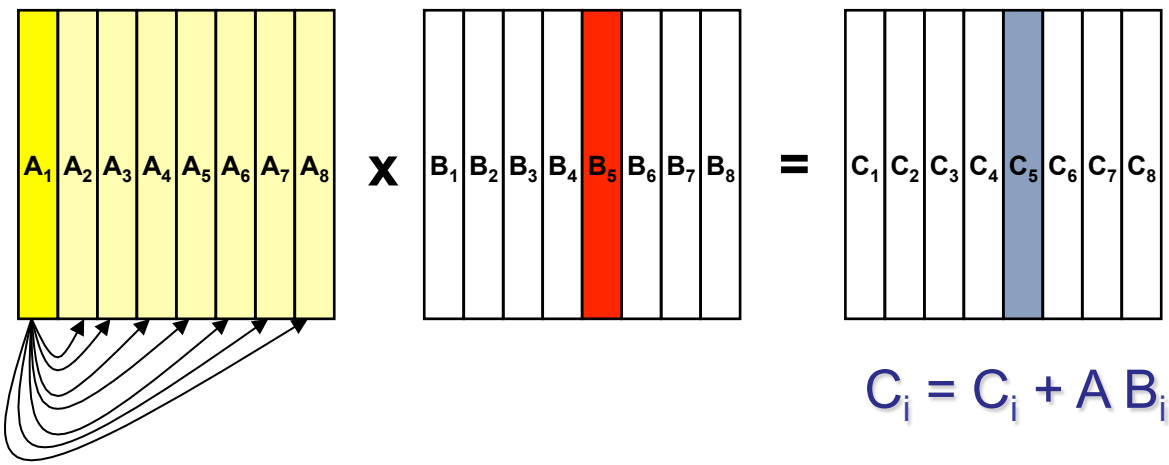
Yuster & Zwick (2005): $nnz^{0.7}n^{1.2}+n^{2+o(1)}$

1. Perform outer-product of dense rows and columns using fast (Strassen-like) dense rectangular matrix multiplication.
2. Use sparse algorithm for remaining rows and columns.

- "Worst-case optimal", but worst case only happens for $nnz(C)=n^2$
- Not "flops optimal", hence only suitable when output is dense.
- Many applications require a classical (semiring) matrix multiply.

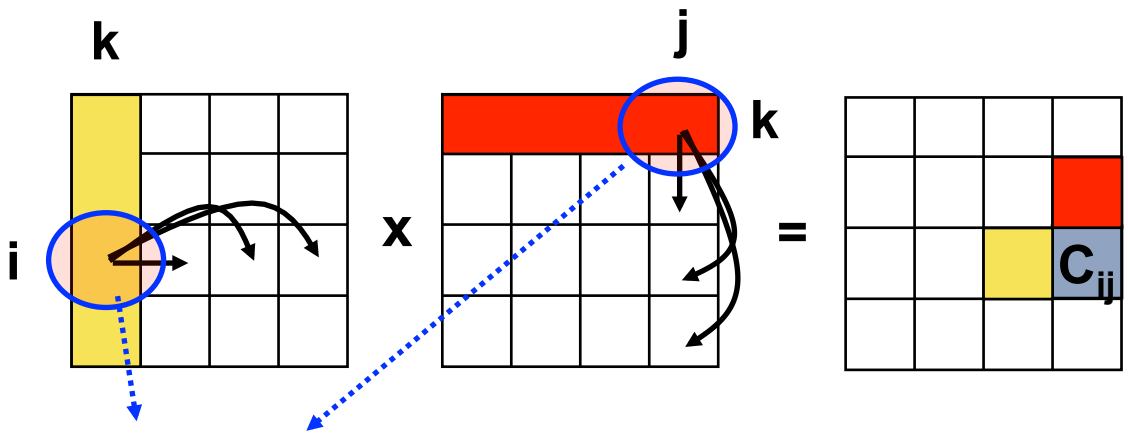


Two versions of Sparse GEMM



$C_i = C_i + A\,B_i$

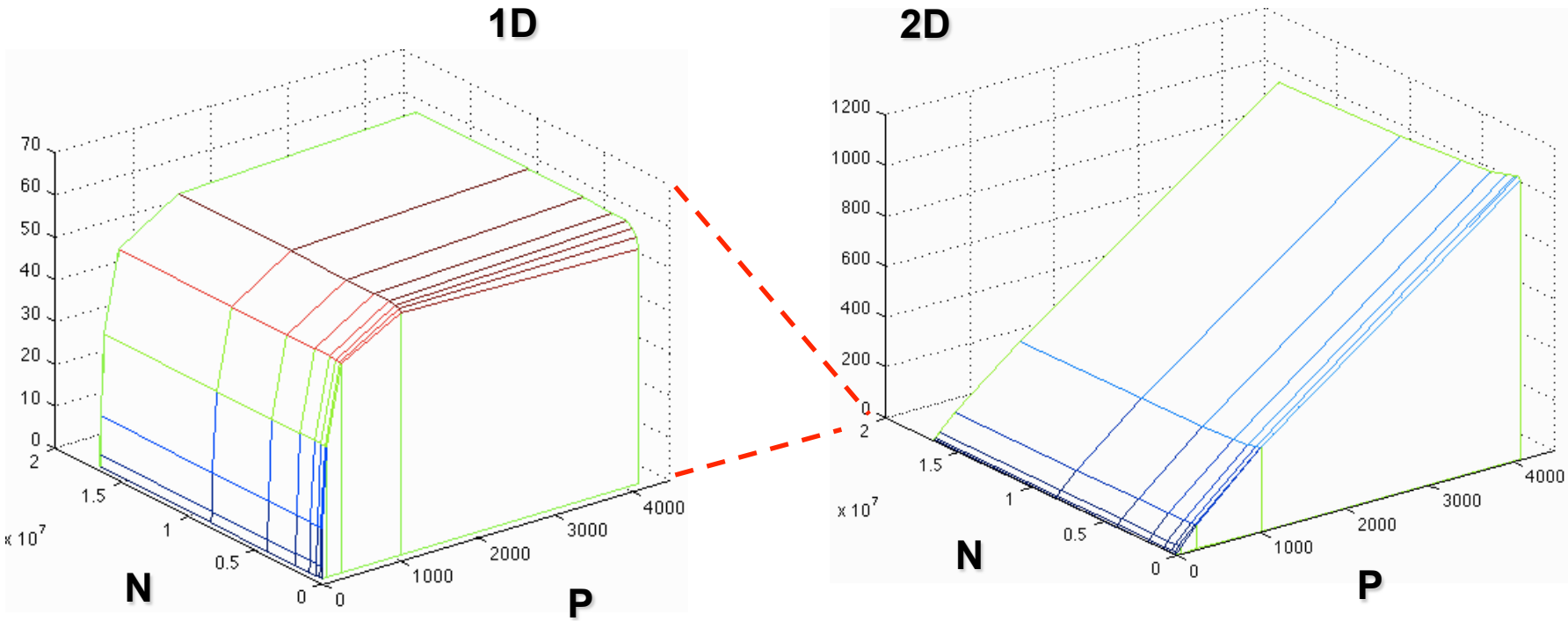
1D block-column distribution



$C_{ij} \mathrel{+}= A_{ik}\,B_{kj}$

Checkerboard (2D block) distribution

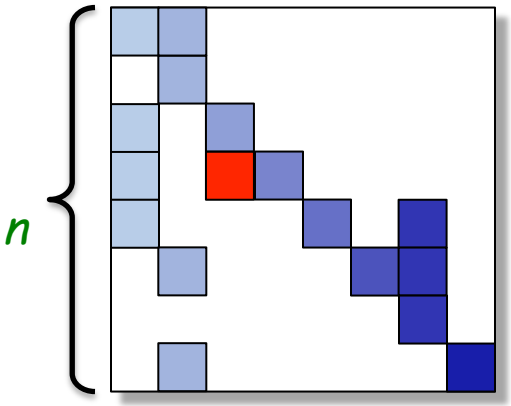
Projected speedup of Sparse 1D & 2D



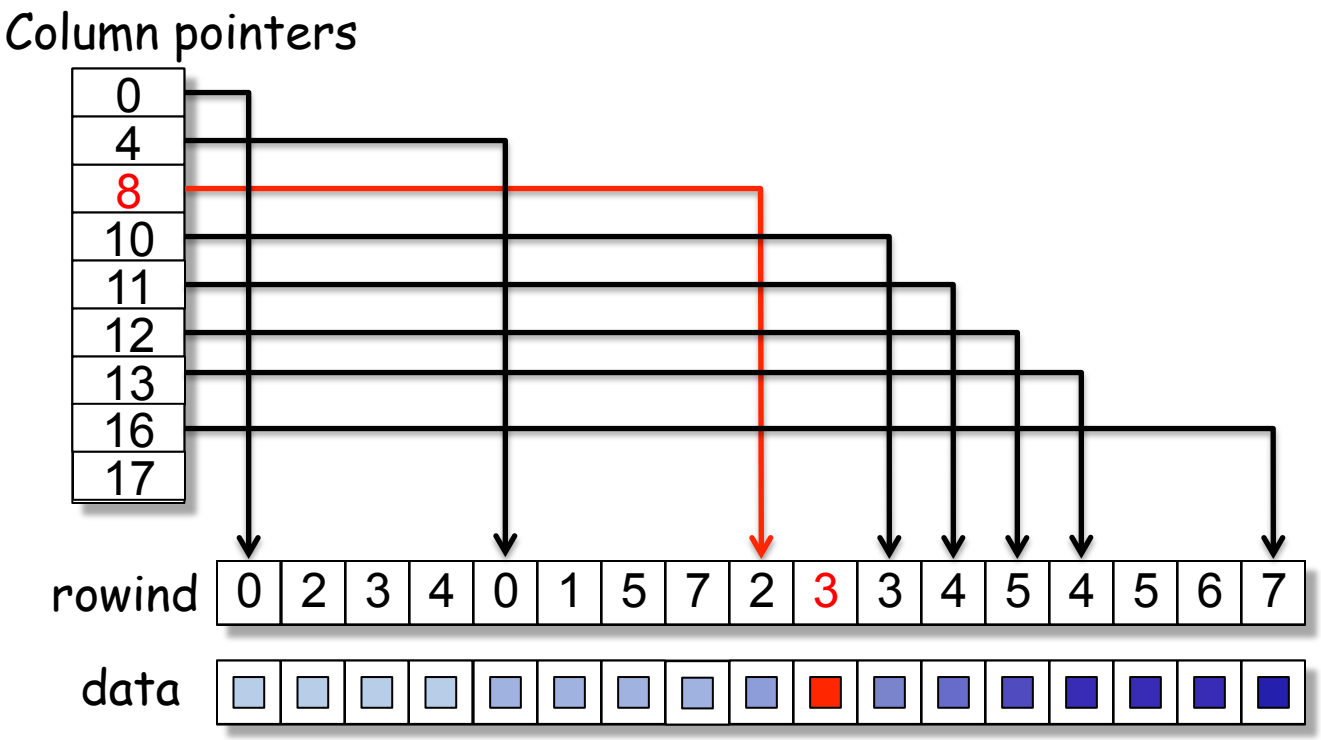
In practice, 2D algorithms have <u>the potential</u> to scale, but not linearly

$$E = \frac{W}{p(T_{comp} + T_{comm})} = \frac{d^2 n}{\beta d n \sqrt{p} + d^2 n \lg\left(d^2 n / p\right) + \alpha p \sqrt{p}}$$

Compressed Sparse Columns (CSC): A Standard Layout



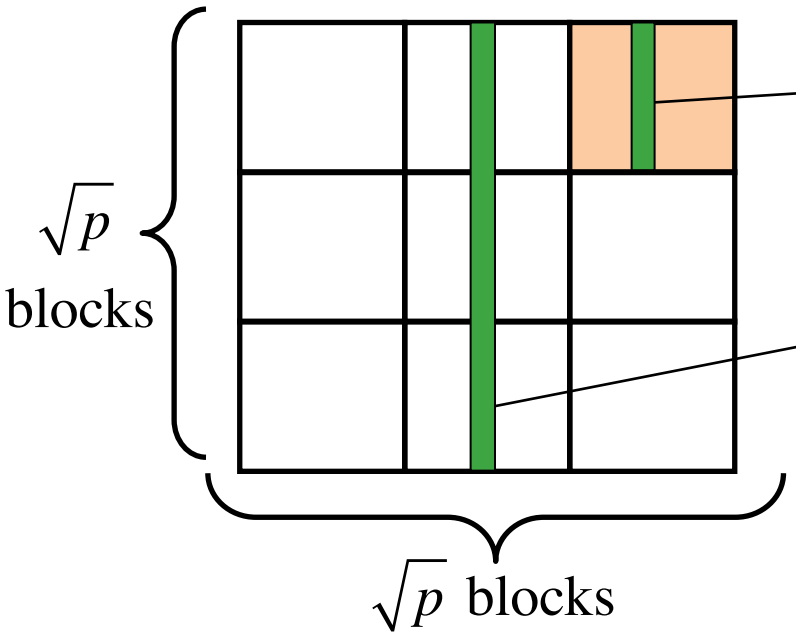
$n \times n$ matrix with
nnz nonzeroes



- Stores entries in column-major order
- Dense collection of *"sparse columns"*
- Uses $O(n + nnz)$ storage.

Submatrix storage in 2D

Submatrices are "*hypersparse*" (*i.e. nnz << n*)



$$nnz' = \frac{d}{\sqrt{p}} \rightarrow 0$$

Average of ***d*** nonzeros per column

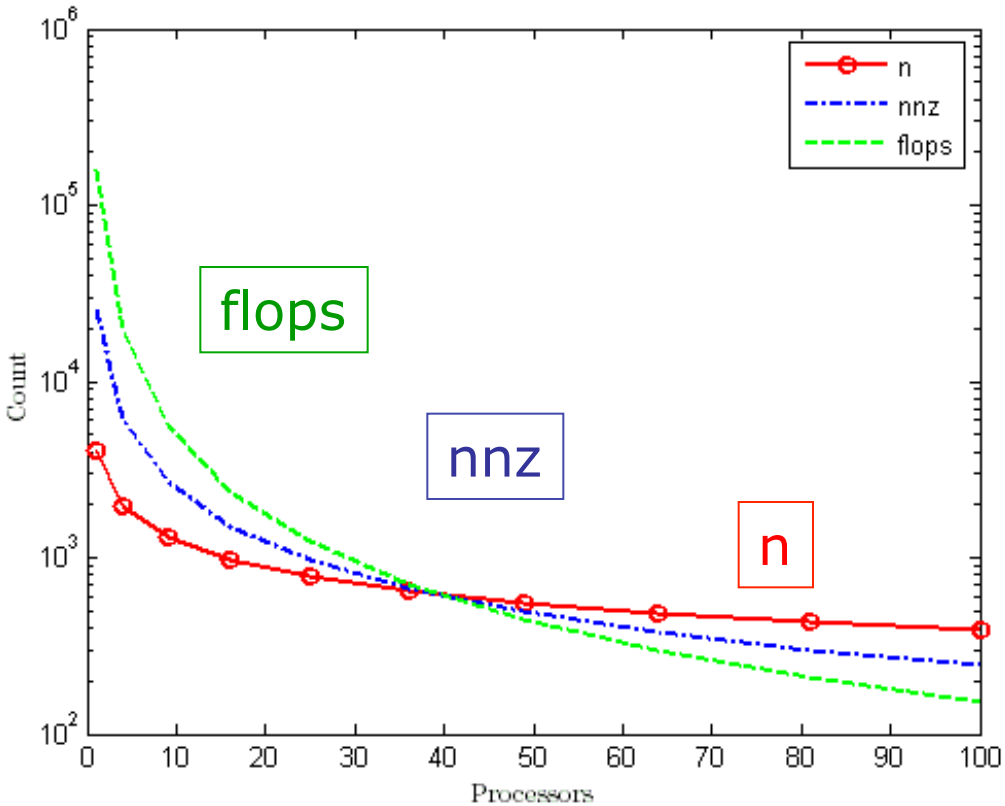
<u>Total Storage</u>:

$$O(n + nnz) \rightarrow O(n\sqrt{p} + nnz)$$

- A data structure or algorithm that depends on matrix dimension n (e.g. CSR or CSC) is asymptotically too wasteful for submatrices
- Use doubly-compressed (DCSC) data structures or compressed sparse blocks (CSB) instead.

Complexity measure trends with increasing p in 2D

Gustavson's algorithm is *O(nnz+ flops+n)*



$$n'(\text{dimension}) \approx \frac{n}{\sqrt{p}}$$

$$nnz'(\text{data size}) \approx \frac{nnz}{p}$$

$$flops'(\text{work}) \approx \frac{flops}{p\sqrt{p}}$$

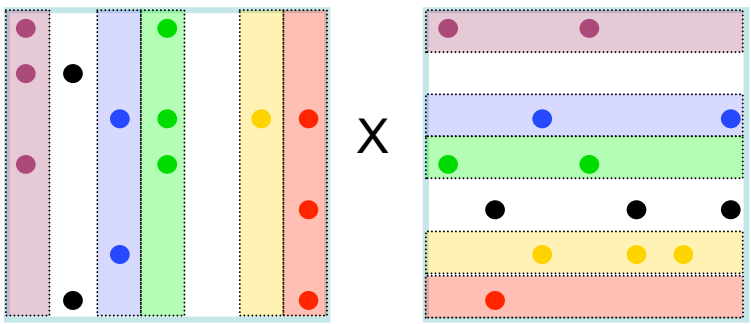
When multiplying two R-MAT matrices with nnz/n =8
(column/row nonzero counts follow a power law)

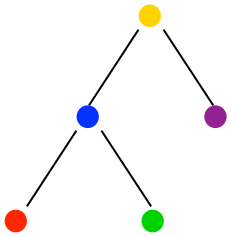
Sequential “hypersparse” kernel

Operates on the strictly O(nnz) DCSC data structure

Sparse outer-product formulation with multi-way merging

Efficient in parallel, i.e. T(1) ≈ p T(p)





Time complexity:

$$O(flops \cdot \lg ni + nzc(A) + nzr(B))$$

- independent of dimension

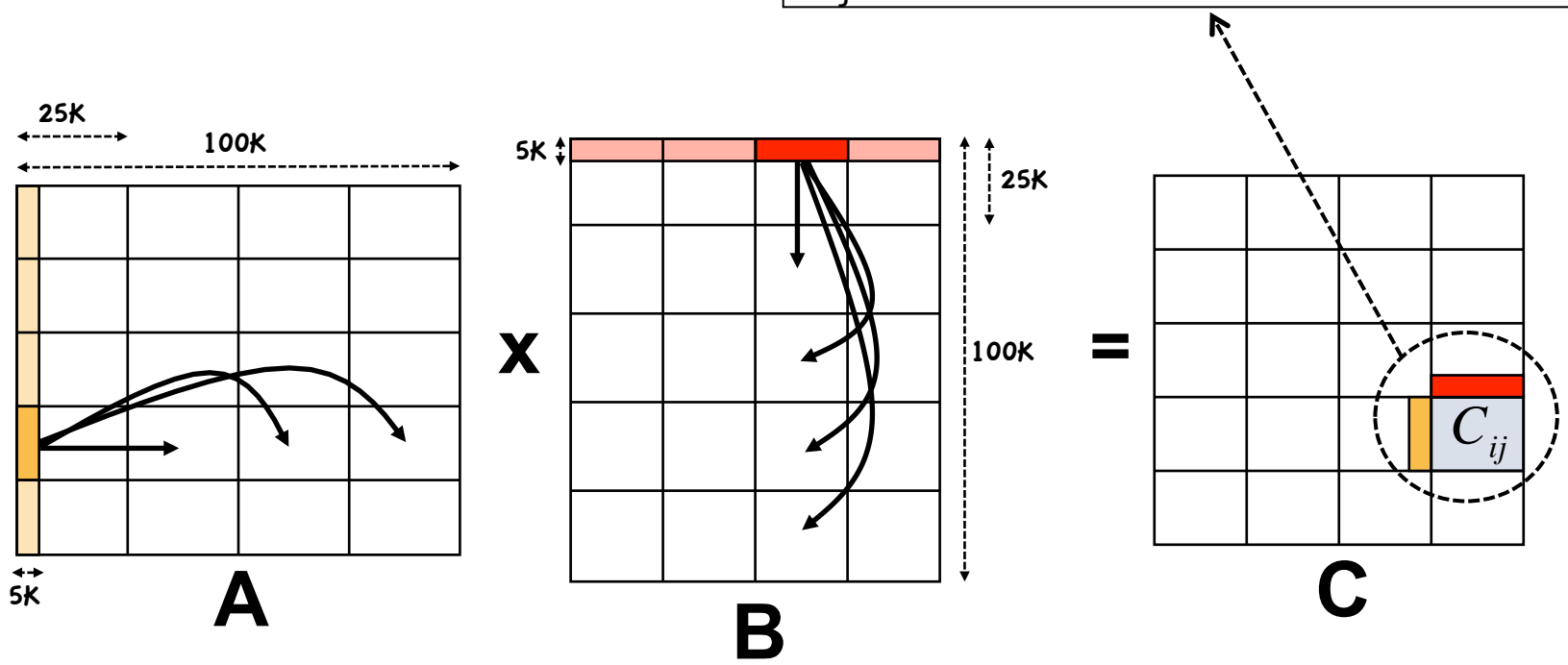
Space complexity:

$$O(nnz(A) + nnz(B) + nnz(C))$$

- independent of flops

2D algorithm: Sparse SUMMA

C_{ij} += HyperSparseGEMM(A^{recv}, B^{recv})



Based on dense SUMMA

General implementation that handles rectangular matrices

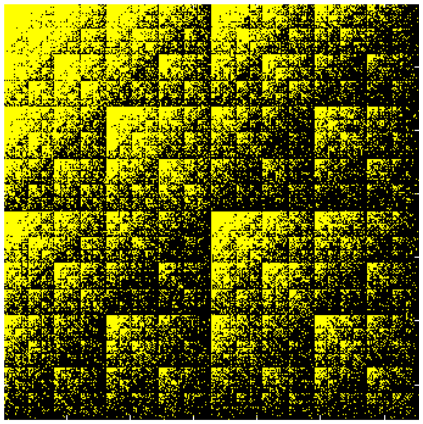
Experimental details

Platform: NERSC Franklin, Cray XT4 with quad-core processors.

Test cases:

1. Square sparse matrix multiplication
2. Multiplication with the restriction operator
3. Tall skinny right-hand-side matrix

Main matrix generator: R-MAT with 8 nonzeros per column



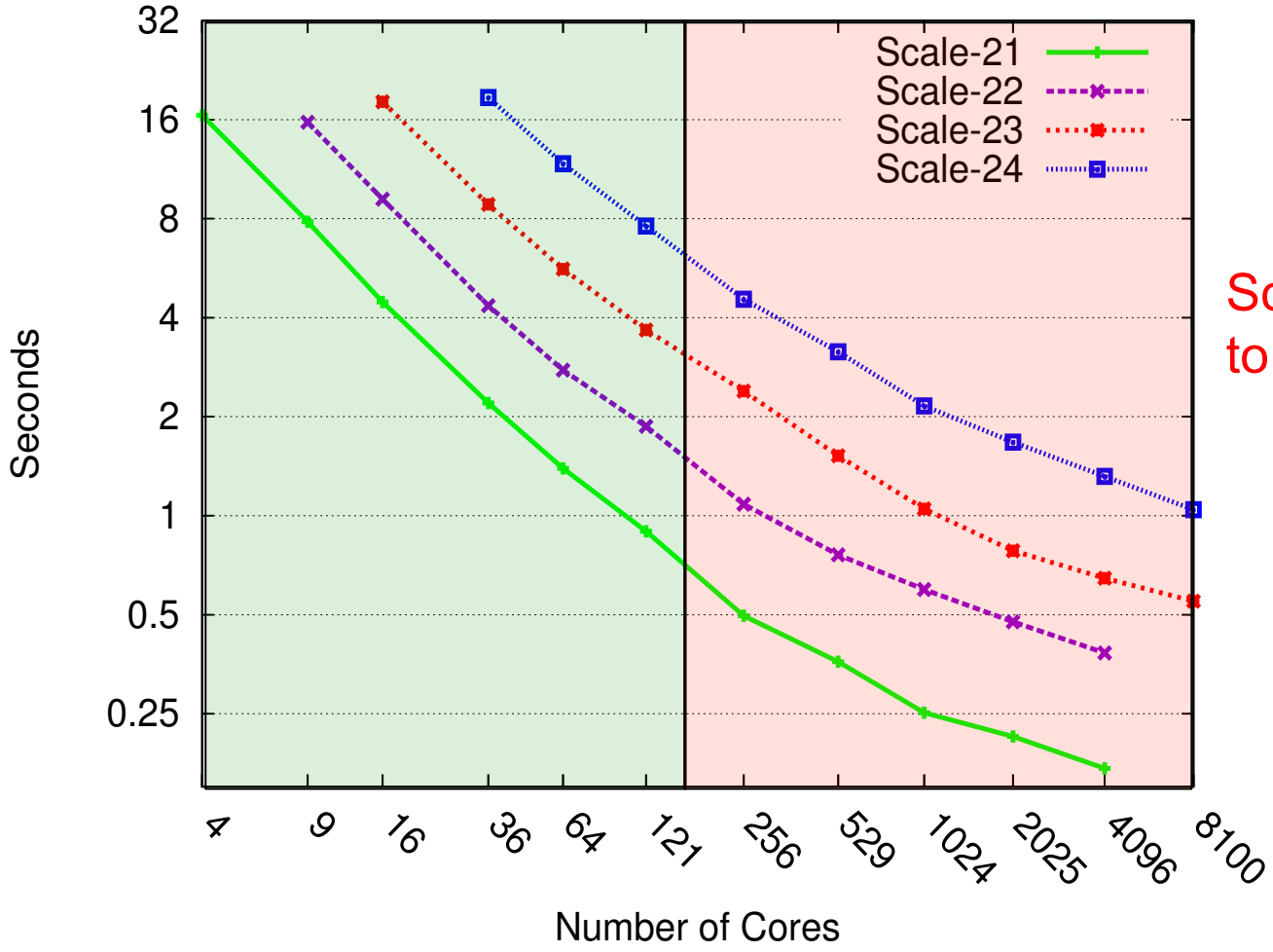
Spy plot of R-MAT matrix
(yellows denote nonzeros)

Scale N matrix is 2^N by 2^N

Double precision arithmetic

Square sparse matrix multiplication

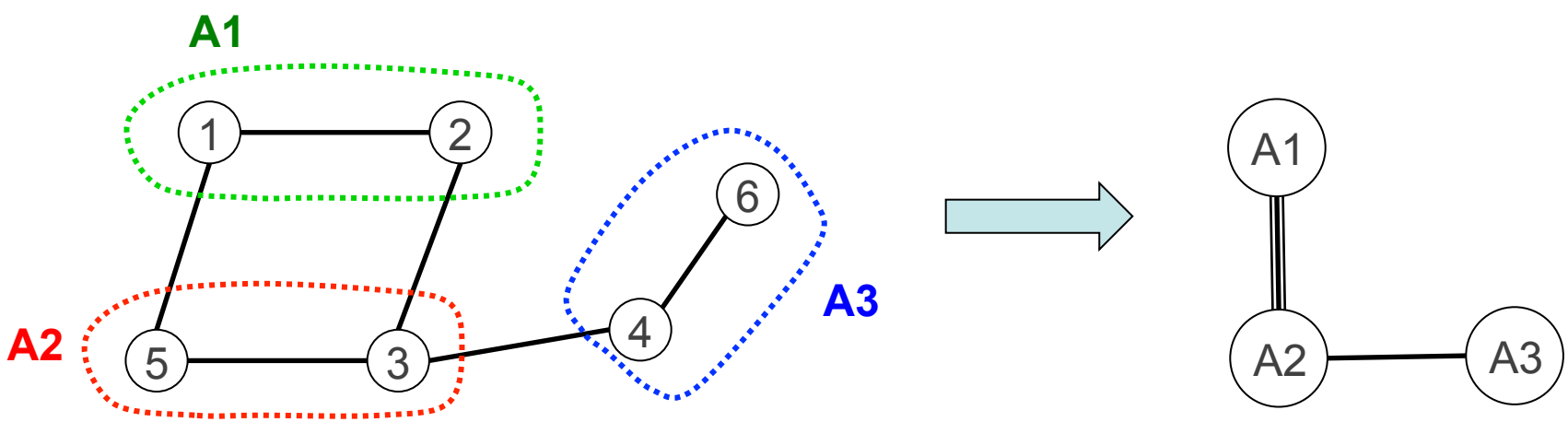
Linear scaling until bandwidth costs starts to dominate



Scaling proportional to √p afterwards

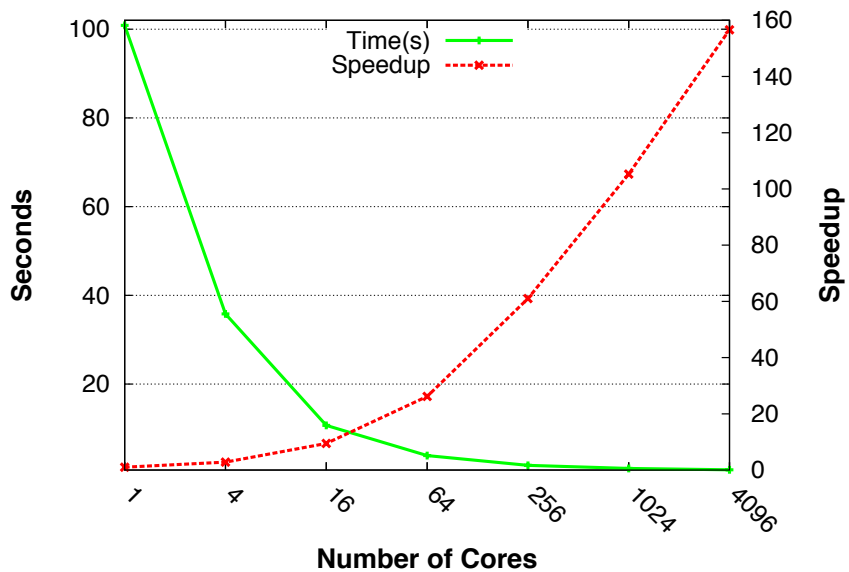
Multiplication with the restriction operator

$$
\begin{bmatrix} 1 & 1 & & & & \\ & & 1 & & 1 & \\ & & & 1 & & 1 \end{bmatrix}
\times
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline
1 & & \bullet & & & \bullet & \\
2 & \bullet & & \bullet & & & \\
3 & & \bullet & & \bullet & \bullet & \\
4 & & & \bullet & & & \bullet \\
5 & \bullet & & \bullet & & & \\
6 & & & & \bullet & &
\end{array}
\times
\begin{bmatrix} 1 & & & & & \\ 1 & & & & & \\ & 1 & & & & \\ & & & 1 & & \\ & 1 & & & & \\ & & & & 1 & \end{bmatrix}
=
\begin{bmatrix} & \bullet & \\ \bullet & & \bullet \\ & \bullet & \end{bmatrix}
$$

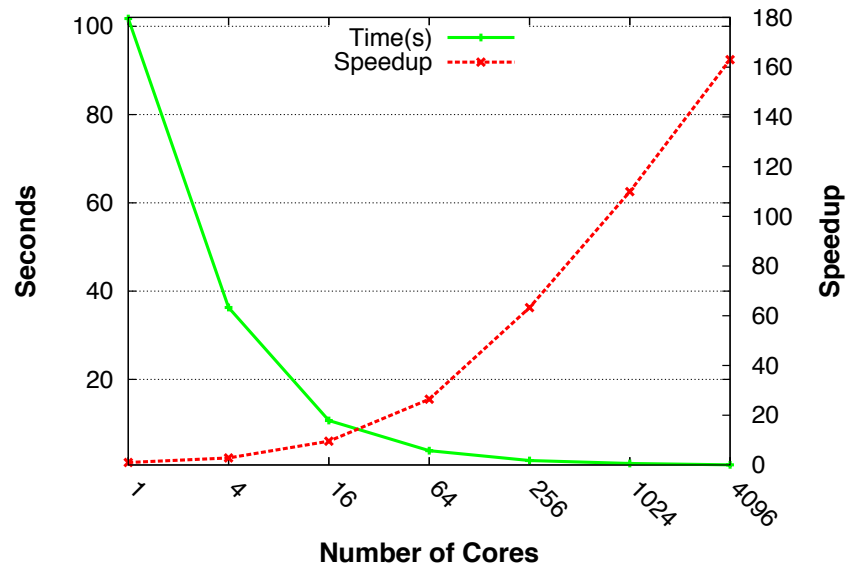


Multiplication with the restriction operator

The full restriction operation of order 8 applied to a scale 23 R-MAT



(a) Left to right evaluation: $(\mathbf{SA})\mathbf{S}^{\mathsf{T}}$

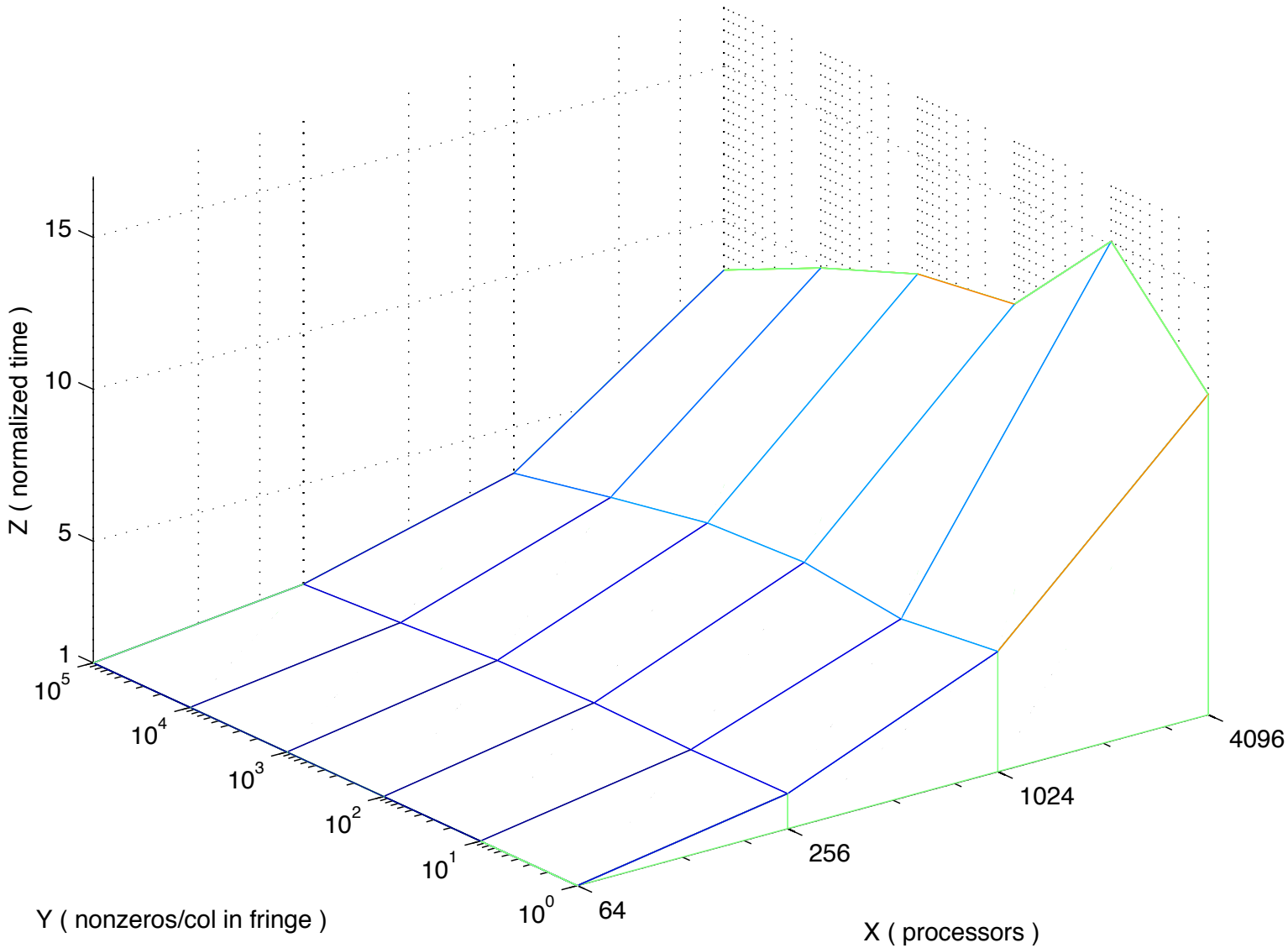


(b) Right to left evaluation: $\mathbf{S}(\mathbf{AS}^{\mathsf{T}})$

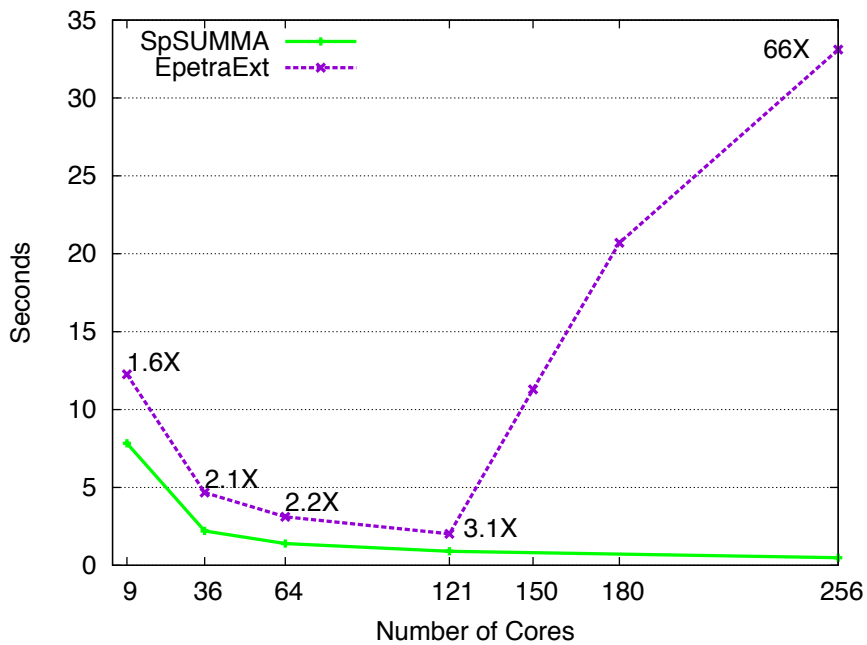
Restriction order does NOT affect performance since

- Our algorithms are dimension oblivious
- The expected nnz(C) is not affected.

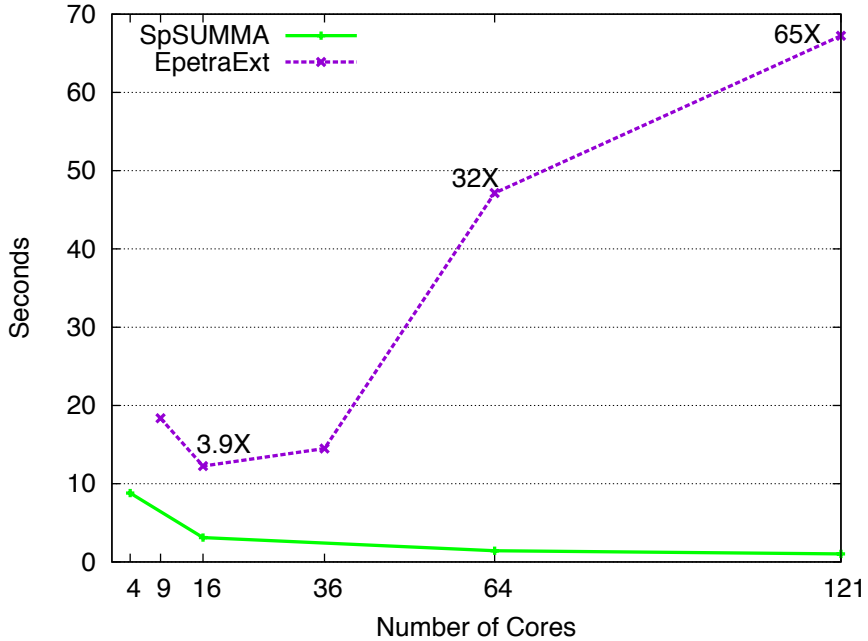
Tall skinny right-hand-side matrix



Comparison of SpGEMM implementations



(a) R-MAT × R-MAT product (scale 21).

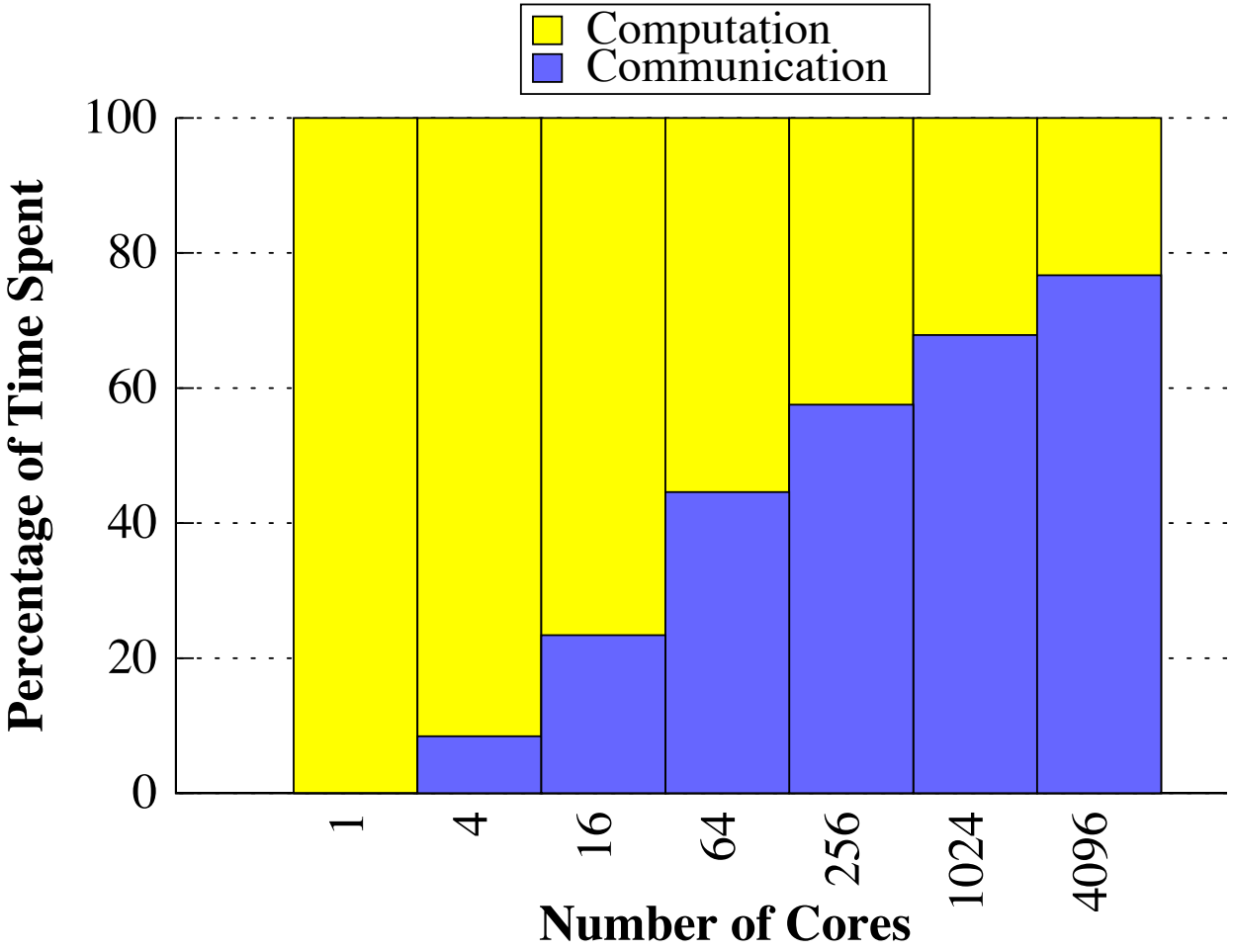


(b) Multiplication of an R-MAT matrix of scale 23 with the restriction operator of order 8.

SpSUMMA = 2-D data layout (Combinatorial BLAS)

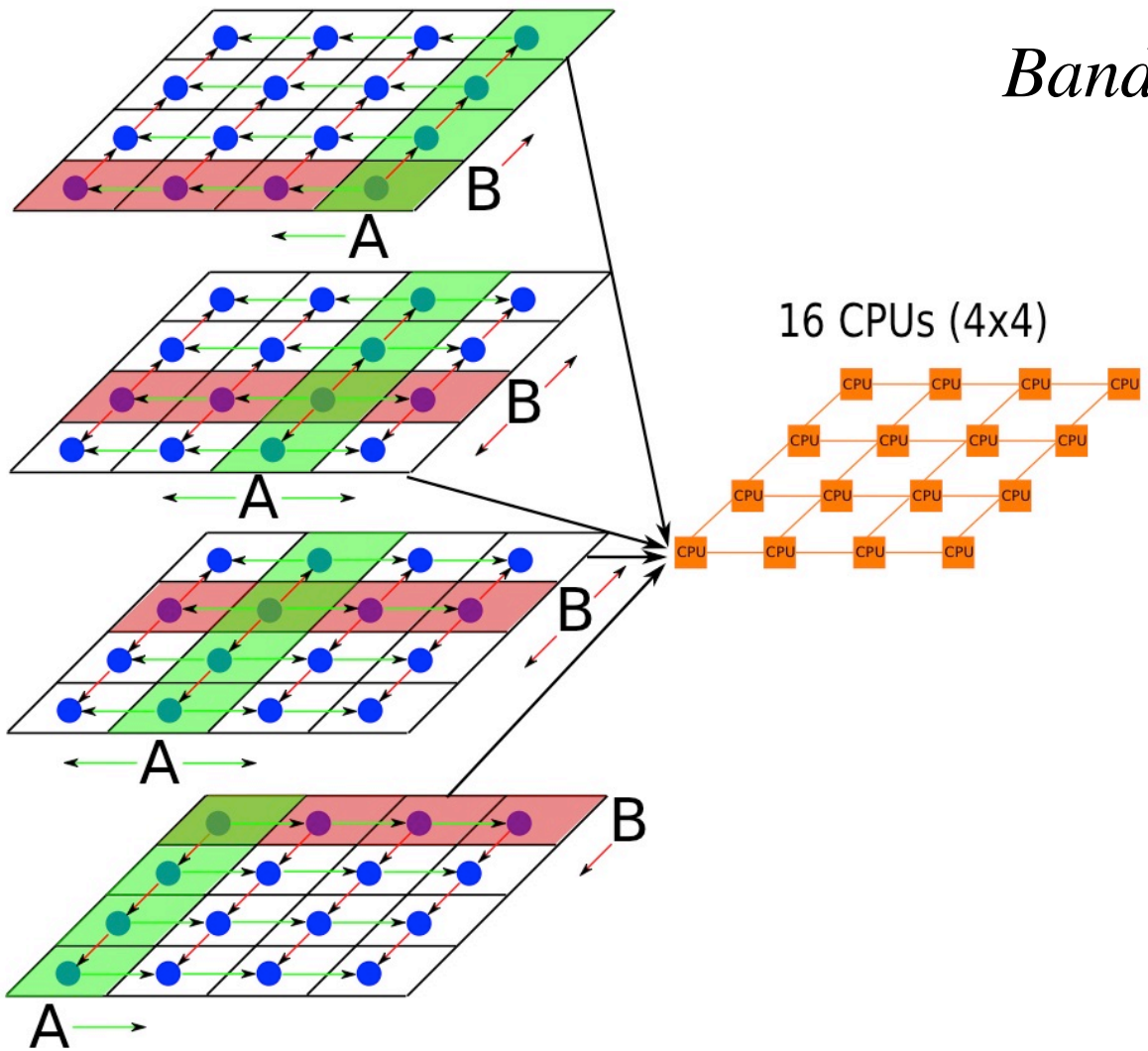
EpetraExt = 1-D data layout (Trilinos)

Need to reduce communication



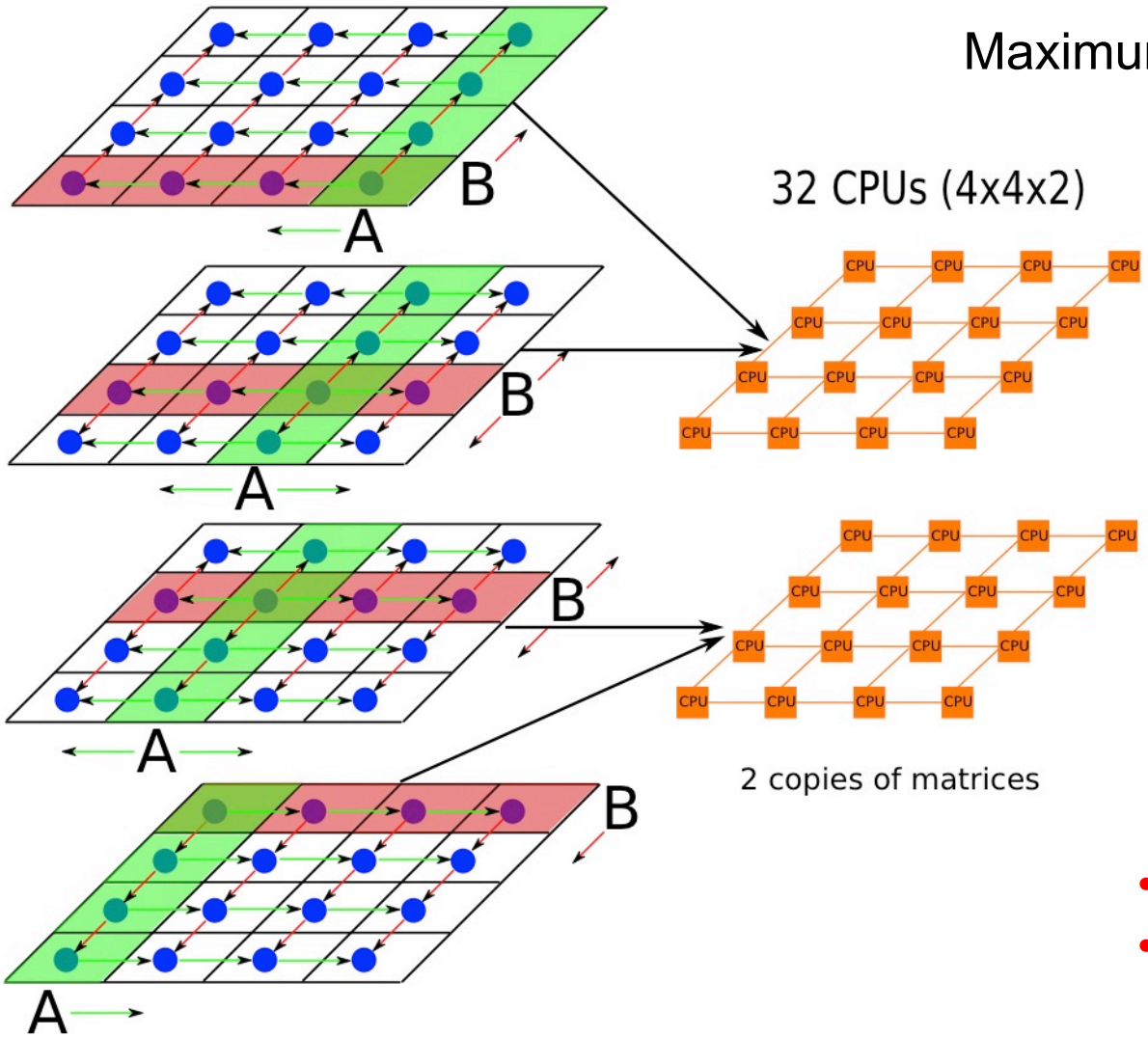
- Normalized communication/computation breakdown
- Scale 23 R-MAT times restriction operator of order 4

Remember the 2D algorithm



$$Bandwidth = \Theta(\frac{dn}{\sqrt{p}})$$

Generalize SUMMA to 2.5D



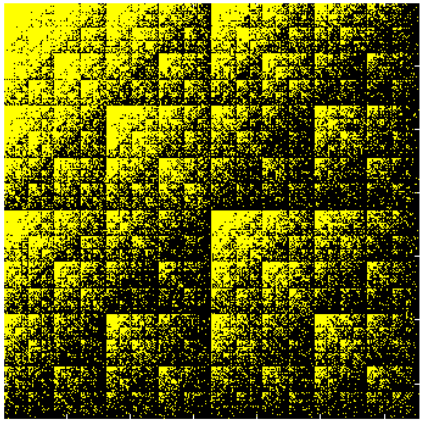
Maximum replicas: $c \leq \frac{\sqrt[3]{p}}{d^{2/3}}$

$$Bandwidth: \Theta\left(\frac{d^{4/3} n}{p^{2/3}}\right)$$

- **Better scaling with p**
- **Worse with d**

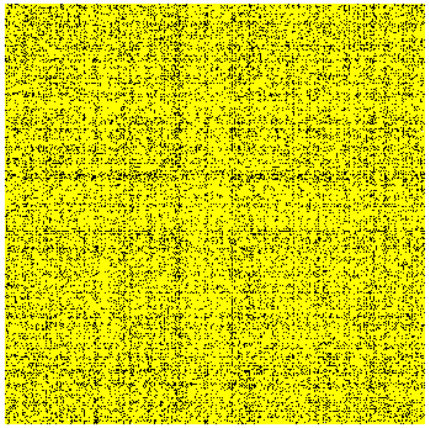
Why are SpRef/SpAsgn important?

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



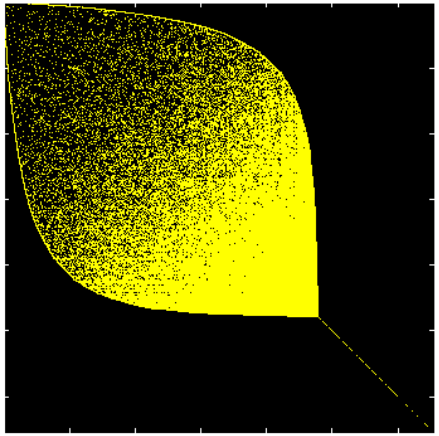
A=rmat(15)

− Load balance hard
± Some locality



A(r,r) : r random

+ Load balance easy
− No locality

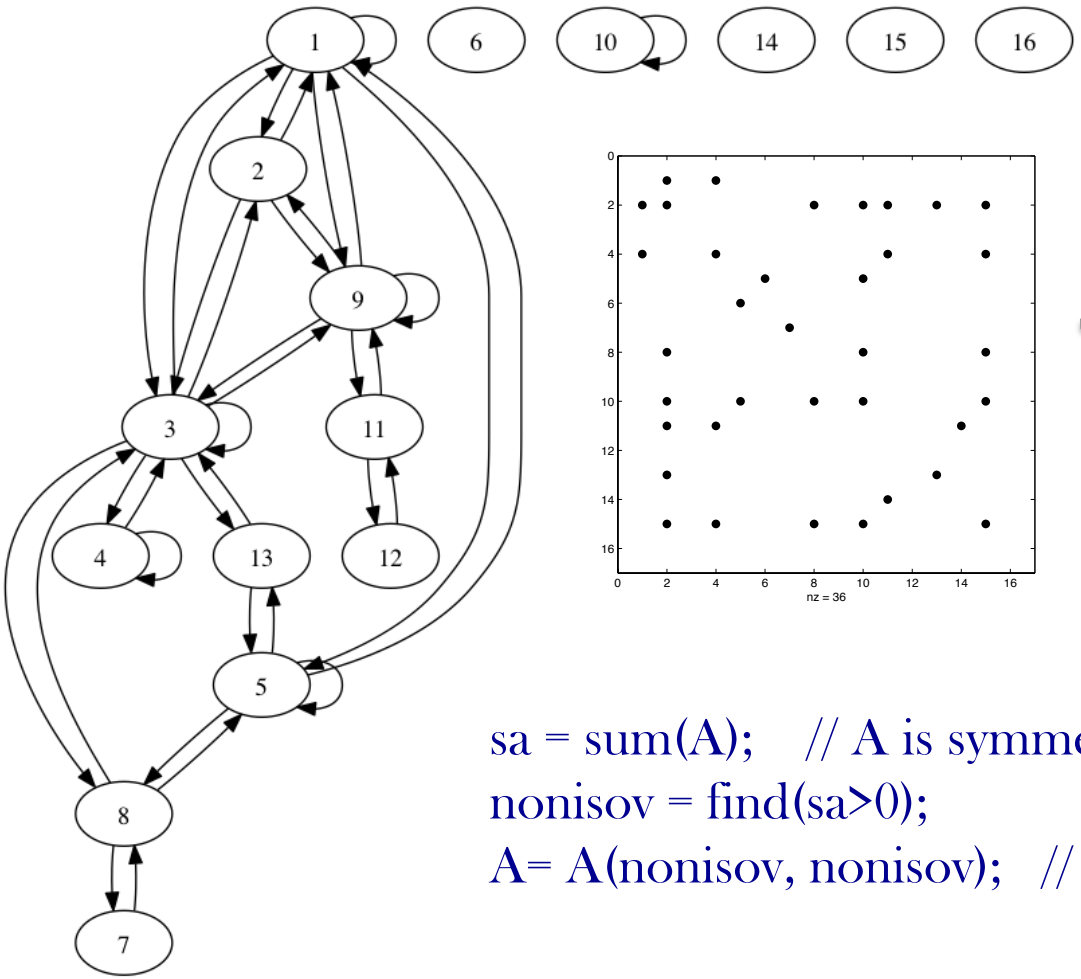


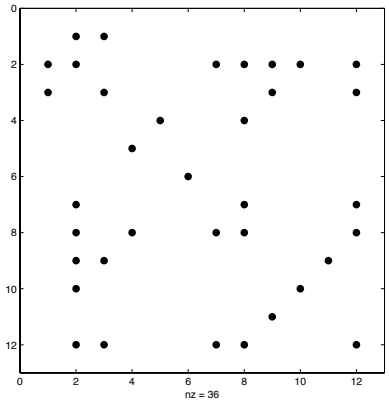
A(r,r) : r=symrcm(A)

− Load balance hard
+ Good locality

More applications of SpRef

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





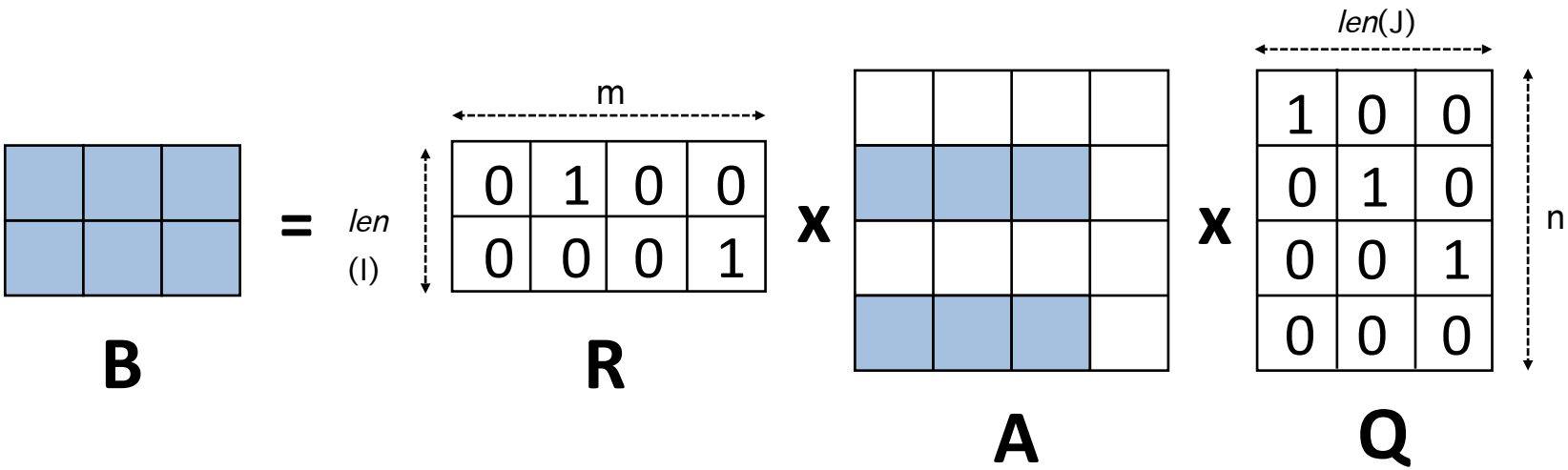

```
sa = sum(A);    // A is symmetric, for undirected graph
nonisov = find(sa>0);
A= A(nonisov, nonisov);   // prune isolated vertices
```

Indexing sparse arrays in parallel

Used for **extracting subgraphs**, **coarsening grids**, **relabeling vertices**, etc.

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

Sequential SpRef and SpAsgn

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

$$O(nnz(A))$$

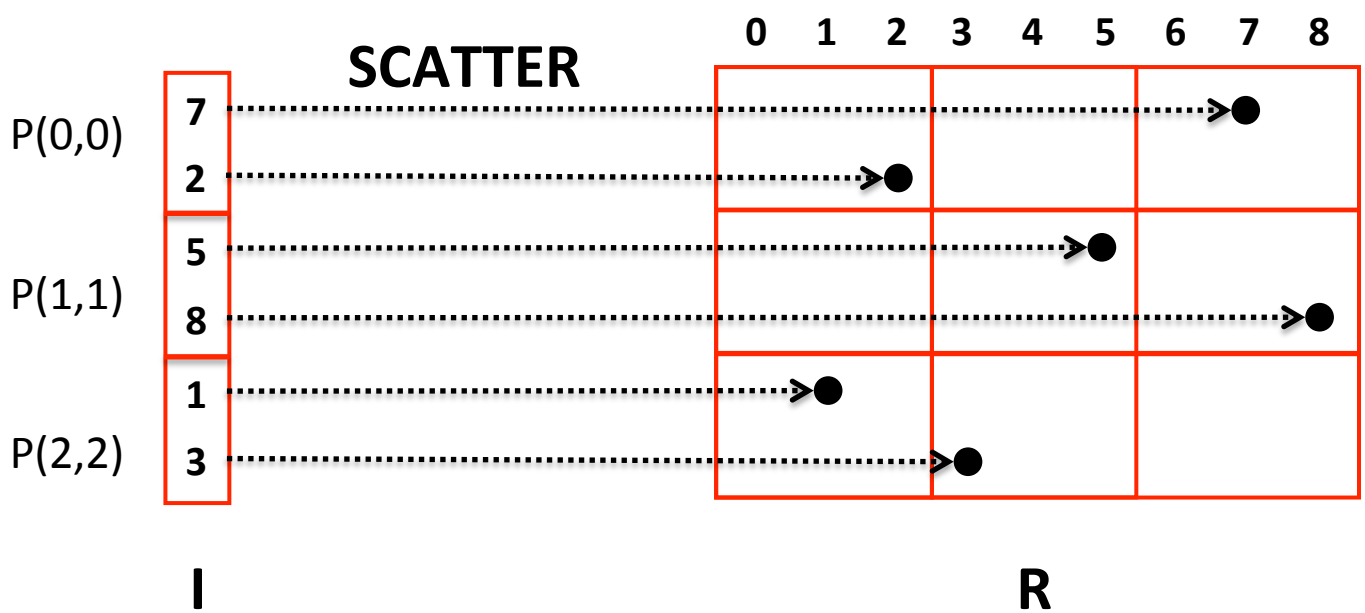
```
function C = spasgn(A,I,J,B)
  [ma,na] = size(A);
  [mb,nb] = size(B);
  R = sparse(I,1:mb,1,ma,mb);
  Q = sparse(1:nb,J,1,nb,na);
  S = sparse(I,I,1,ma,ma);
  T = sparse(J,J,1,na,na);
  C = A + R*B*Q - S*A*T;
```

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

$$O(nnz(A) + nnz(B) + len(I) + len(J))$$

Parallel algorithm for SpRef

Step 1: Form **R** from I in parallel, on a 3x3 processor grid



Forming **Q**:
Similar row-wise communication, followed by **Q.Transpose()**

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

Parallel algorithm for **SpRef**

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

Minimize temporaries by:

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

$$T_{comp} = O\left(\frac{len(I)+len(J)+nnz(A)}{\sqrt{p}} + \frac{nnz(A)}{p}\cdot\log\left(\frac{len(I)+len(J)}{p}+\sqrt{p}\right)\right)$$

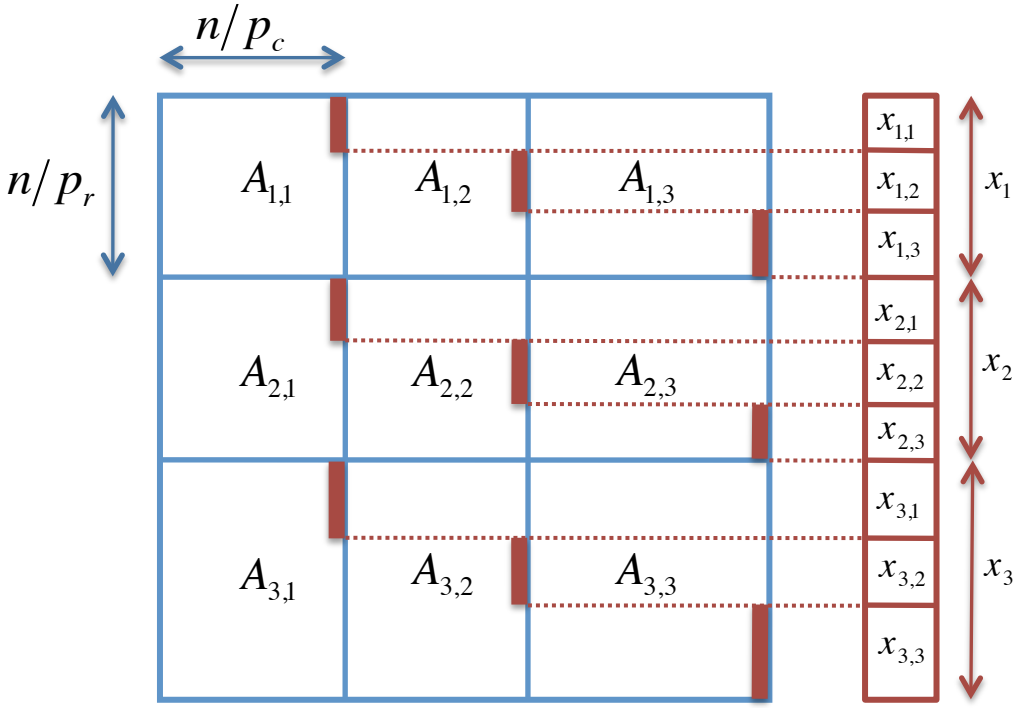
$$T_{comm} = \Theta\left(\alpha\cdot\sqrt{p}+\beta\cdot\frac{nnz(A)}{\sqrt{p}}\right)$$

Dominated by **SpGEMM**

Bottleneck: bandwidth costs

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

2D vector distribution

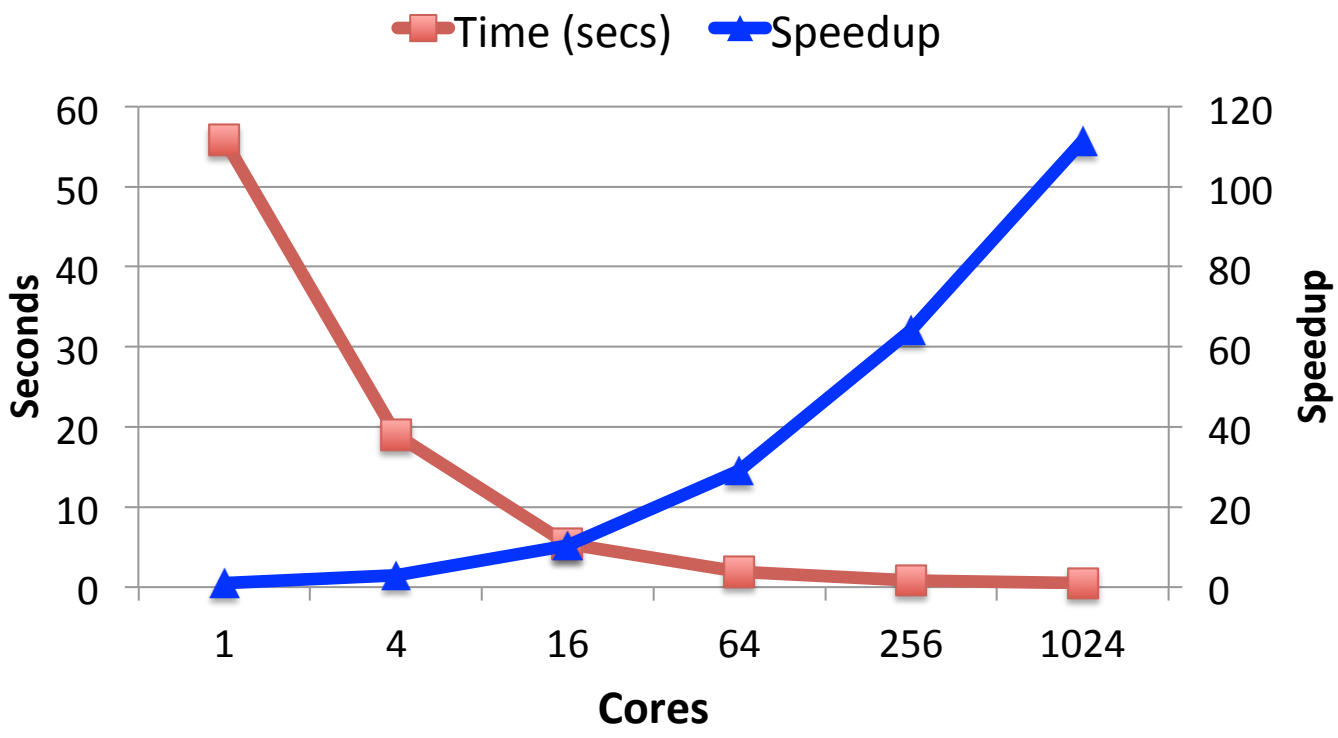


Matrix/vector distributions, interleaved on each other.

Default distribution in **Combinatorial BLAS**.

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

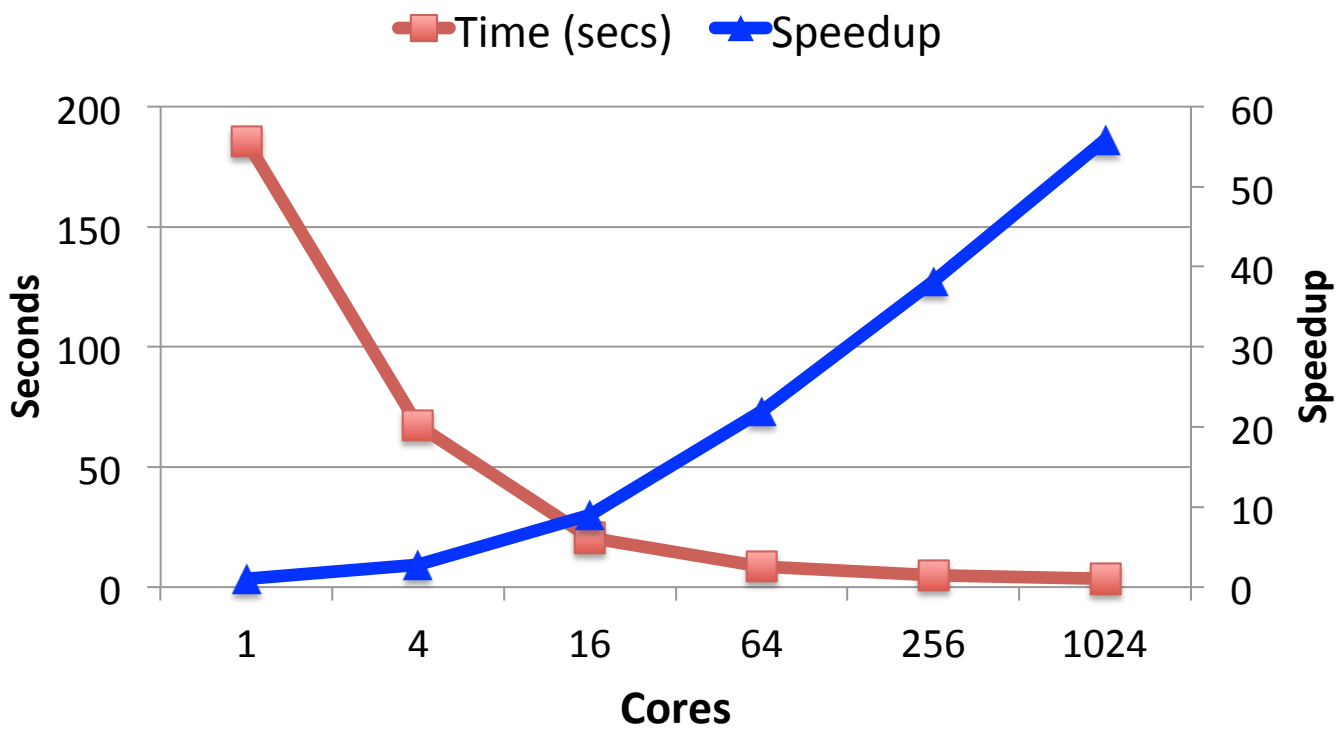
Strong scaling of SpRef



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



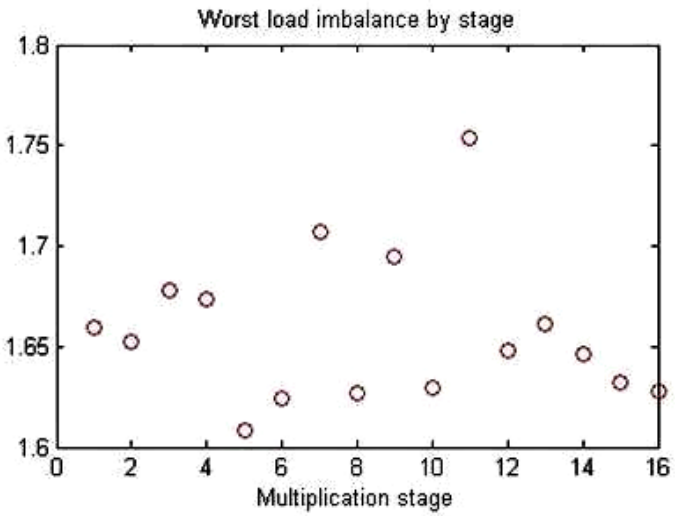
Extracts 10 random (induced) subgraphs, each with |V|/10 vert.
Higher span → Decreased parallelism → Lower speedup

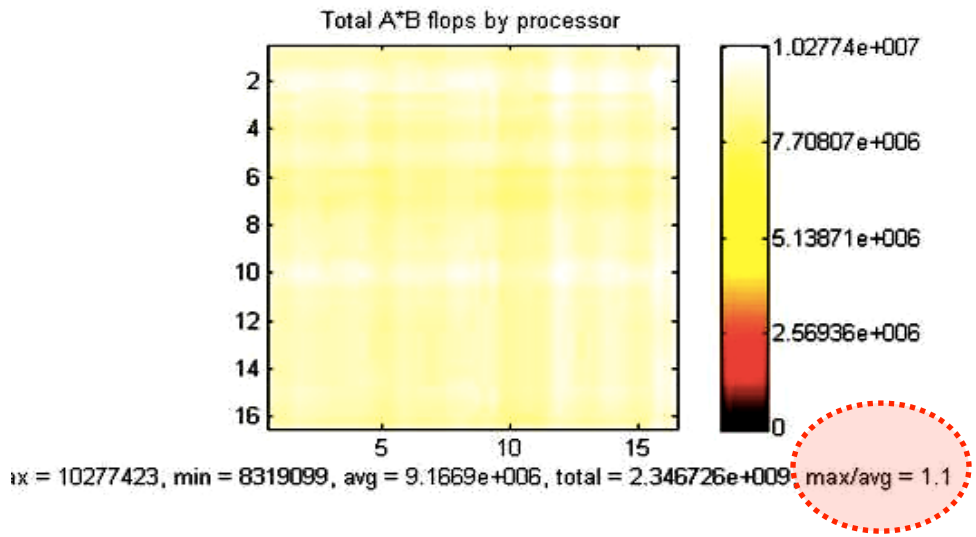
Conclusions

- Flexible and scalable **SpGEMM** algorithm: Sparse SUMMA
- Parallel algorithms for **SpRef** and **SpAsgn**
- Systemic algorithm structure imposed by **SpGEMM**
- Complexity analysis made possible for the general case
- Good strong scaling for 1000-way concurrency
- Many applications on sparse matrix and graph world
- Freely available within **Combinatorial BLAS**.
- All presented primitives are ultimately communication-bound

Future Work

- Competitive implementation of the 2.5D algorithm
- Lower bounds for communication
- Direct support for chain products.
- Robust asynchronous implementation.





References

All primitives incorporated into the Combinatorial BLAS:
http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/index.html

B., Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 2011.

Sparse Matrix Indexing: B., Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. http://arxiv.org/abs/1109.3739

Hypersparsity in 2D decomposition, sequential kernel: B., Gilbert, On the representation and multiplication of hypersparse matrices, *IPDPS'08*

Parallel analysis of Sparse GEMM: B., Gilbert, Challenges and advances in parallel sparse matrix-matrix multiplication, *ICPP'08*

Some Combinatorial BLAS functions

Function	Applies to	Parameters		Returns	Matlab Phrasing
SpGEMM	Sparse Matrix (as friend)	**A**, **B**: trA: trB:	sparse matrices transpose **A** if true transpose **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
SpMV	Sparse Matrix (as friend)	**A**: **x**: trA:	sparse matrices sparse or dense vector(s) transpose **A** if true	Sparse or Dense Vector(s)	$\mathbf{y} = \mathbf{A} * \mathbf{x}$
SpEWiseX	Sparse Matrices (as friend)	**A**, **B**: notA: notB:	sparse matrices negate **A** if true negate **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
Reduce	Any Matrix (as method)	dim: *binop*:	dimension to reduce reduction operator	Dense Vector	sum(**A**)
SpRef	Sparse Matrix (as method)	**p**: **q**:	row indices vector column indices vector	Sparse Matrix	$\mathbf{B} = \mathbf{A}(\mathbf{p}, \mathbf{q})$
SpAsgn	Sparse Matrix (as method)	**p**: **q**: **B**:	row indices vector column indices vector matrix to assign	none	$\mathbf{A}(\mathbf{p}, \mathbf{q}) = \mathbf{B}$
Scale	Any Matrix (as method)	**rhs**:	any object (except a sparse matrix)	none	Check guiding principles 3 and 4
Scale	Any Vector (as method)	**rhs**:	any vector	none	none
Apply	Any Object (as method)	*unop*:	unary operator (applied to non-zeros)	None	none