

# Lecture 8: Sparse Linear Algebra

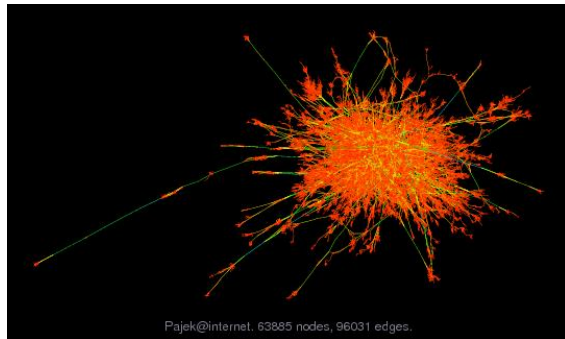
# Outline for today

---

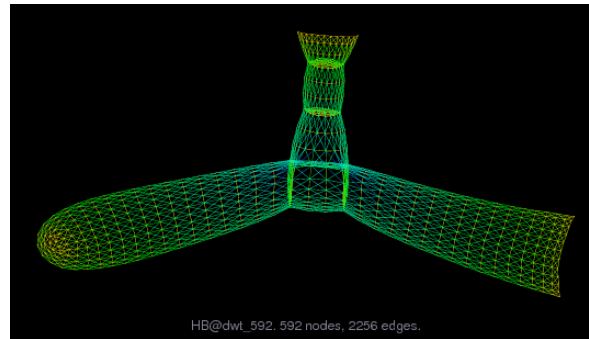
- Sparse matrix formats and basic SpMV
  - Sequential optimizations
  - Distributed memory optimizations
- Higher-level kernels
  - Sparse Matrix Multiply
  - Matrix powers computations
- Iterative solvers - Krylov subspace methods
  - Communication-Avoiding Krylov solvers

# Sparse matrices are everywhere

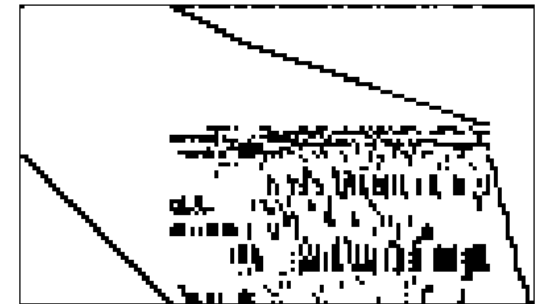
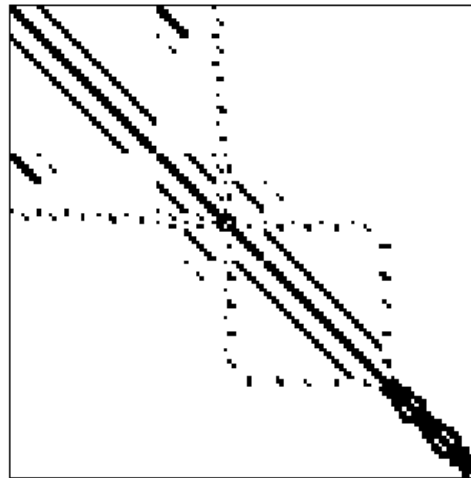
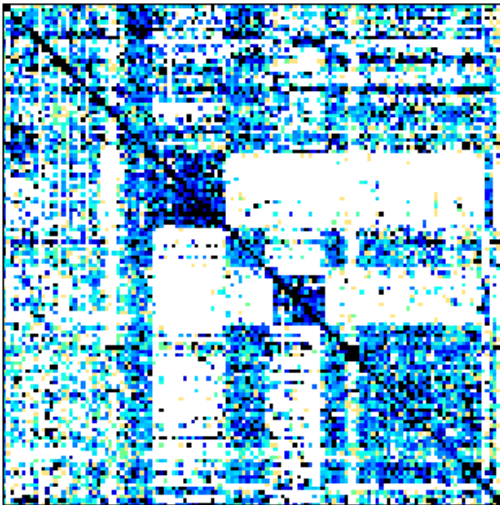
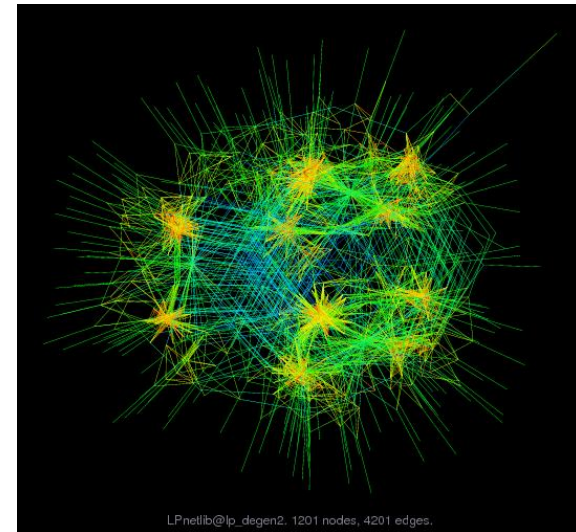
Internet connectivity



Structural design



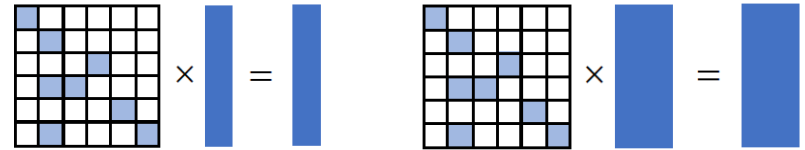
Linear Programming



# Sparse Matrix Computations

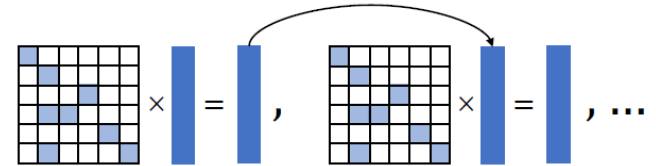
- Sparse matrix-(dense)vector multiplication (SpMV) or sparse-matrix-multiple (dense) vector multiplication

- Solving linear systems
- Eigenvalue problems
- Optimization algorithms
- Machine learning, etc.



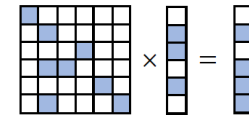
- Repeated SpMV/SPMM ( $A_k x$ )

- Transitive closure on graphs
- Linear relaxation
- Pagerank, Krylov basis computation



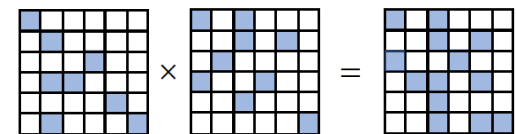
- Sparse matrix-sparse-vector (SpMSpV)

- E.g., graph algorithms: breadth-first search, bipartite graph matching, and maximal independent sets



- Sparse matrix-sparse matrix (SpGEMM)

- E.g., graph algorithms
- Common special case:  $A * A^T$



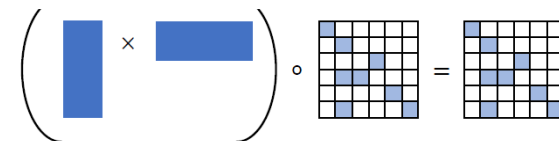
- Sparse matrix-dense matrix (SpDM<sup>3</sup>)

- Machine learning

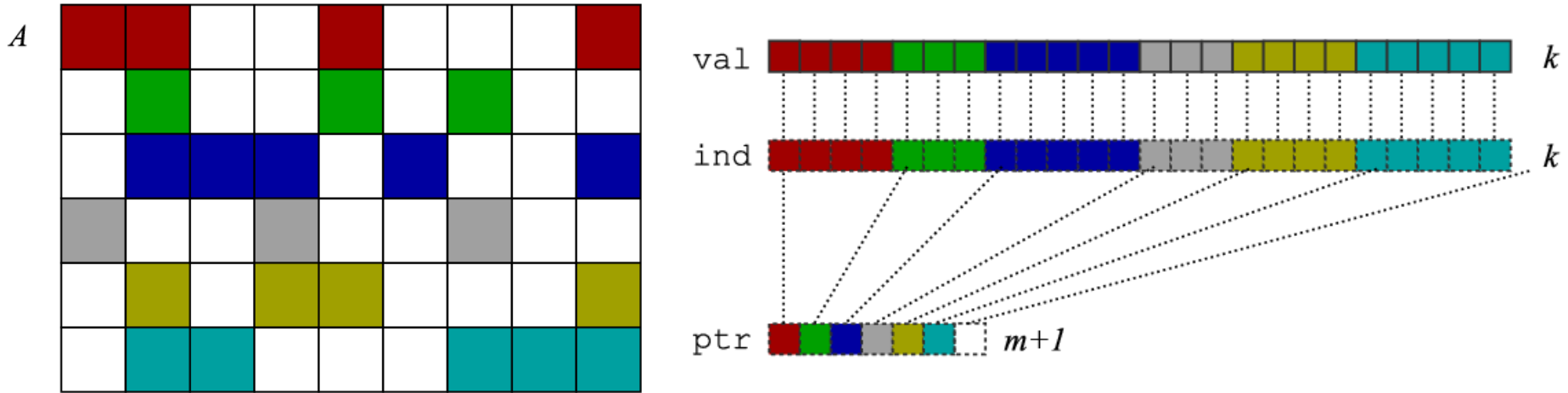


- Sampled Dense-Dense Matrix Multiplication

- Machine learning



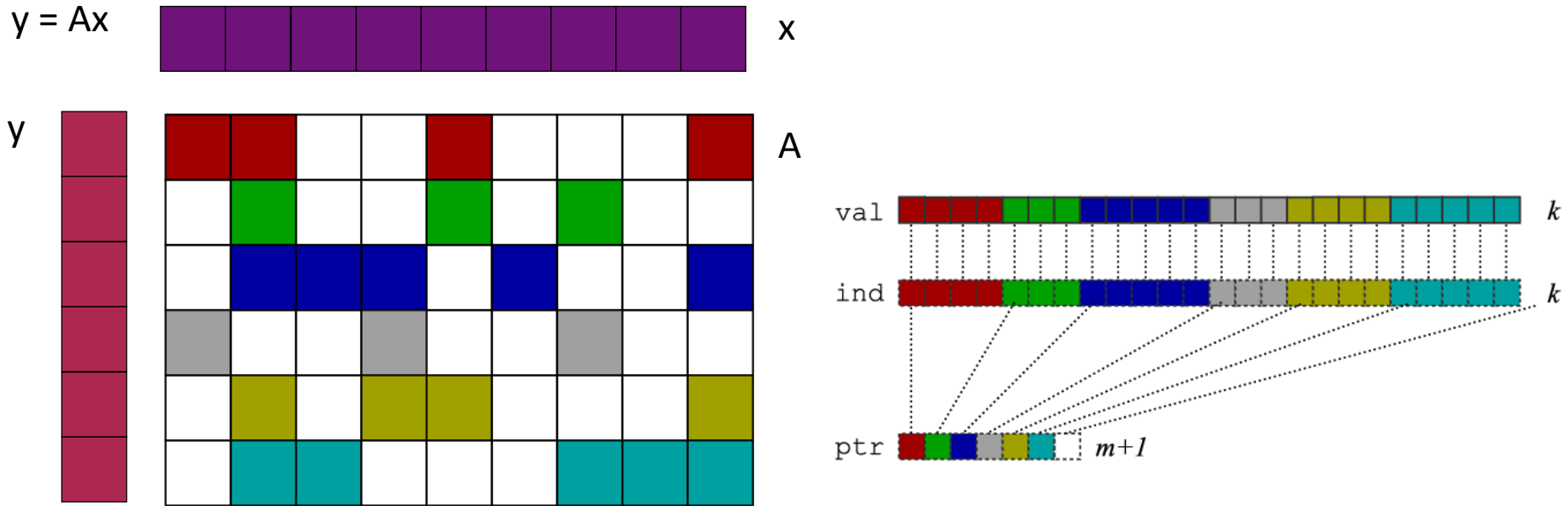
# Compressed Sparse Row (CSR) Storage



- CSR has:
  - Array of the nonzero values (val) of size  $\text{nnz} = \text{number of nonzeros}$
  - Array of the column indices for each value of size  $\text{nnz}$
  - Array of row start pointers of size  $n = \text{number of rows}$
- Other common formats (plus blocking)
  - Compressed sparse column (CSC)
  - Coordinate (COO): row + column index per nonzero (easy to build)

**And many more  
specialized ones!**

# SpMV with Compressed Sparse Row (CSR)



Matrix-vector multiply kernel:  $y(i) \leftarrow y(i) + A(i,j)*x(j)$

for each row  $i$

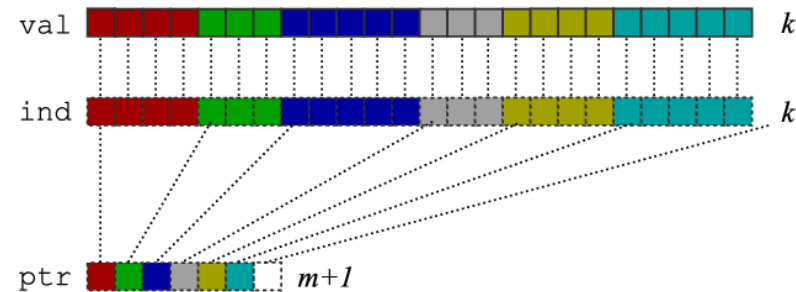
for  $k=ptr[i]$  to  $ptr[i+1]-1$  do

$y[i] = y[i] + val[k]*x[ind[k]]$

# SpMV with Compressed Sparse Row (CSR)

Matrix-vector multiply kernel:  $y(i) \leftarrow y(i) + A(i,j)*x(j)$

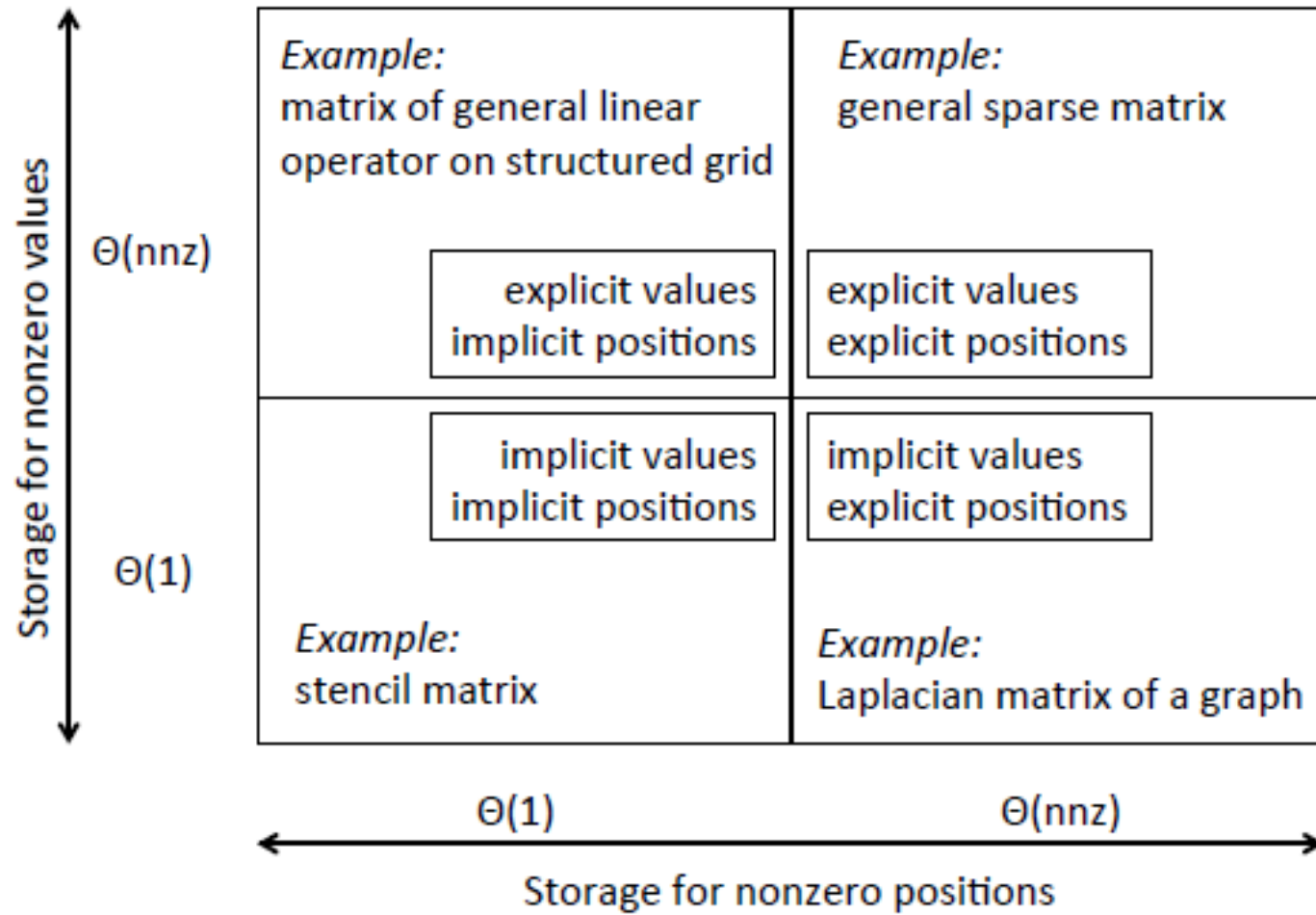
```
for each row i
  for k=ptr[i] to ptr[i+1]-1 do
    y[i] = y[i] + val[k]*x[ind[k]]
```



Possible optimizations:

- 1) Unroll the k loop  $\rightarrow$  need # non-zeros per row
- 2) Hoist `y[i]`  $\rightarrow$  OK absent aliasing
- 3) Eliminate `ind[i]`  $\rightarrow$  need to know non-zero pattern
- 4) Reuse elements of `x`  $\rightarrow$  need good non-zero pattern

# Sparse matrix representations

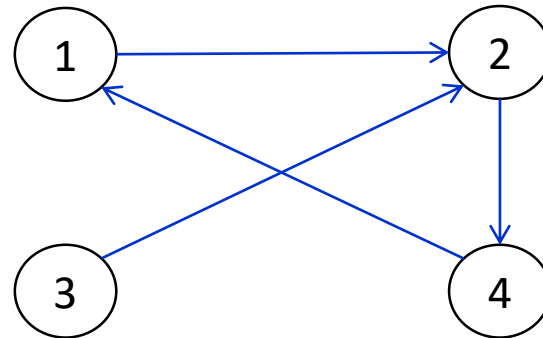




# SpMV dependency graph

- Graph of A:  $G(A)=(V,E)$ 
  - Directed graph with vertices  $V=\{1,\dots,n\}$
  - Edges  $(i,j) \in E \subseteq V \times V$
  - $(i,j) \in E$  iff  $A(i,j) \neq 0$
  - $\text{nnz} = |E|$

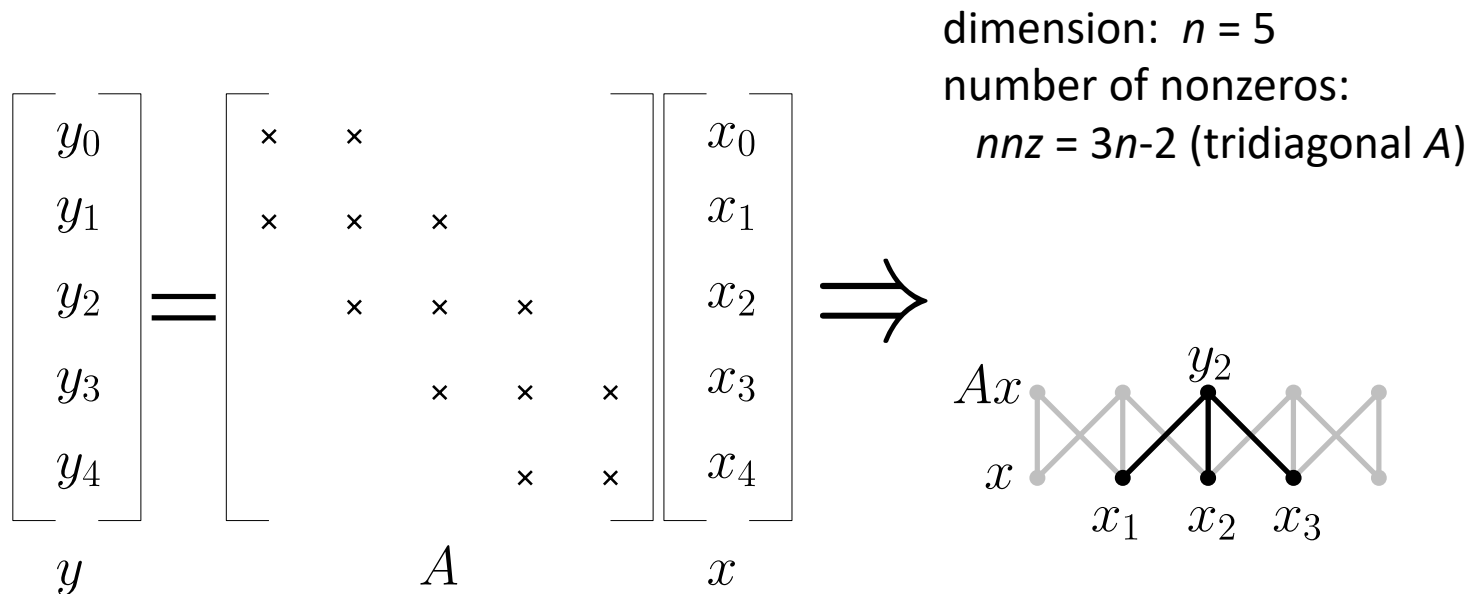
$$\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & 0 & \times \\ 0 & \times & \times & 0 \\ \times & 0 & 0 & \times \end{bmatrix}$$



# Lower bounds and optimal algorithms - sequential

- First, sequential case (assume explicit values/indices)
- Flops:  $\Omega(\text{nnz})$
- Bandwidth (words moved):  $\Omega(\text{nnz})$ 
  - lower bound for the explicit case follows from the fact that  $W = \Omega(\text{nnz})$  words must be moved between slow and fast memory (of size  $M$ )
  - this many nonzero values and/or positions must be read to apply  $A$ .
- Latency ( $\#$  messages):  $\Omega(\text{nnz}/M)$ 
  - Since we allow messages of size between 1 and  $M$ , the latency lower bounds are a factor of  $M$  smaller

# SpMV Arithmetic Intensity (1)



	SpMV
floating point operations	$2 \cdot nnz$
floating point words moved	$nnz + 2 \cdot n$

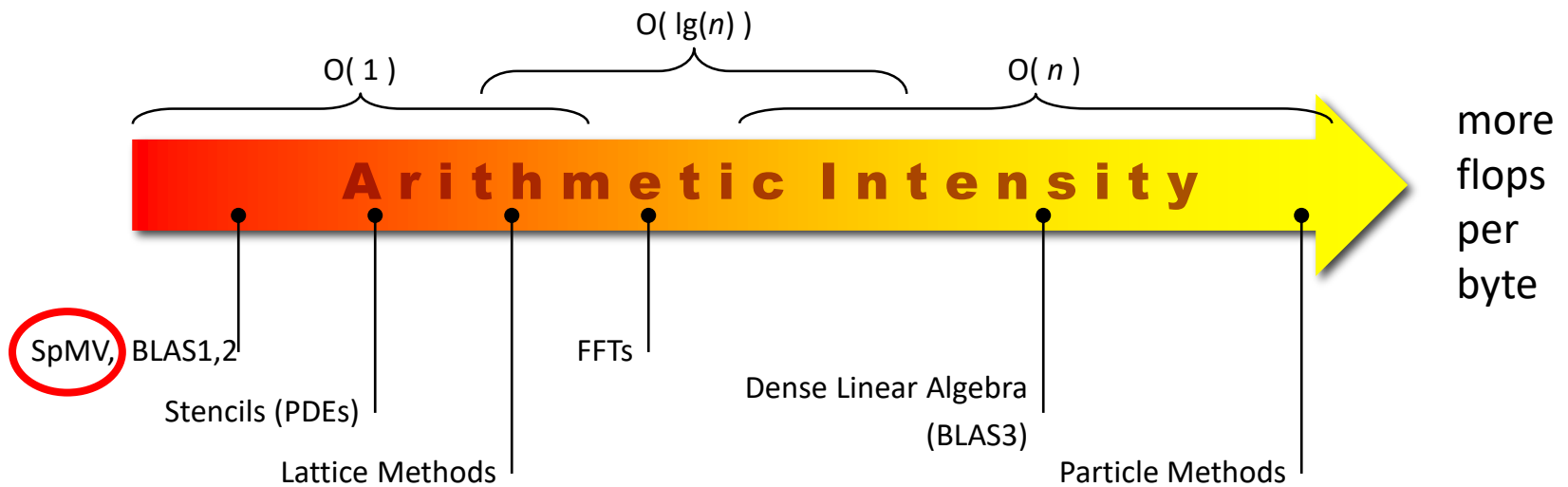
Assumption:  $A$  is invertible

$\Rightarrow$  nonzero in every row

$\Rightarrow nnz \geq n$

overcounts flops by up to  $n$  (diagonal  $A$ )

# SpMV Arithmetic Intensity (2)

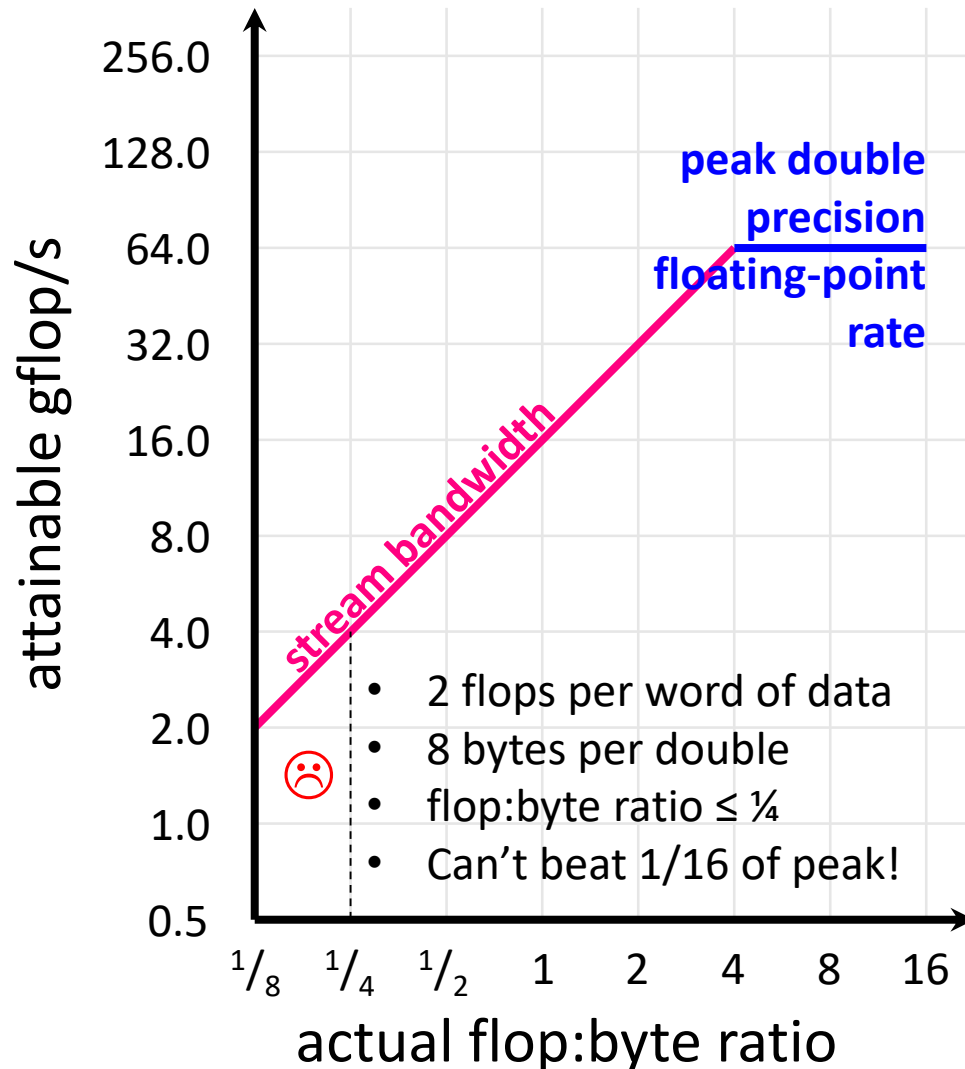


- Arithmetic intensity := Total flops / Total DRAM bytes

$$\frac{\text{flops}}{\text{words}} \approx 2 \times \frac{nnz}{nnz + 2n} \xrightarrow{nnz=W(n)} 2$$

	SpMV
flops	$2 \cdot nnz$
words moved	$nnz + 2 \cdot n$
arith. intensity	2

# SpMV Arithmetic Intensity (3)



“Roofline model”

[Williams, Waterman,  
Patterson, CACM, 2009]

How to do more  
flops per byte?

Reuse data ( $x, y, A$ )  
across multiple SpMVs

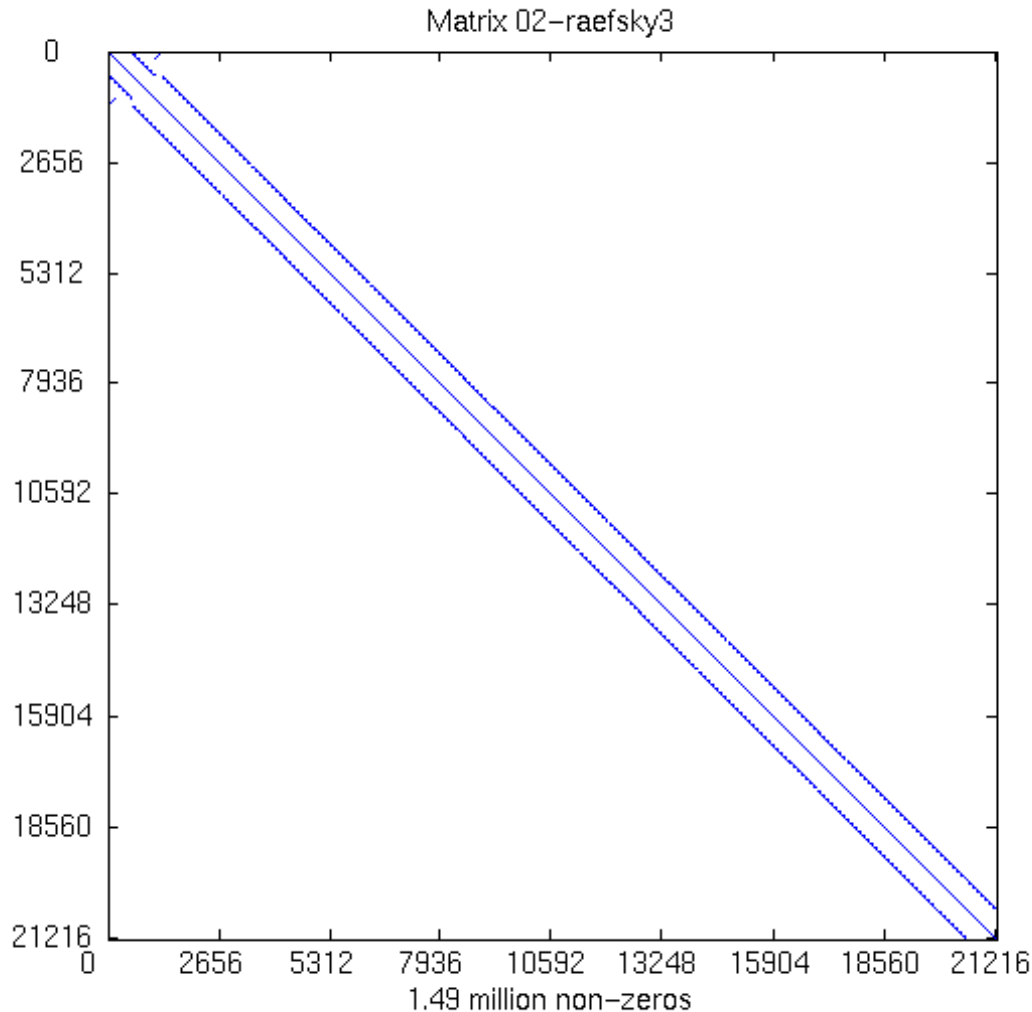
# Lower bounds and optimal algorithms - sequential

- SpMV will be **communication bound**
  - Each nonzero  $A_{ij}$ , or its position, is only needed once, so there is **no reuse** of these values.
  - Thus, if the nonzeros, or their positions, do not fit in cache, then they can be accessed at no faster rate than main memory bandwidth.
  - More importantly, at most two floating-point operations – a multiply and, perhaps, an add – are performed for each  $A_{ij}$  read from memory
- performance generally bounded above by peak memory bandwidth
- no more than 10% of peak flop rate on commodity hardware

# Optimization techniques

- Register blocking - considering small, dense blocks of  $A$  as 'nonzeros' rather than the nonzero elements themselves
  - helps exploit re-use of vector entries, and also reduces the number of indices needing to be read from memory
  - see [Vuduc, 2003], [Vuduc et al., 2005]
- Cache blocking
  - see [Nishtala, Vuduc, Demmel, Yelick, 2007]
- Reordering - Reorder the sparse matrix to concentrate elements around the diagonal (e.g., reverse Cuthill–McKee ordering)
  - can improve spatial locality of the vector accesses, potentially reducing the latency cost

# Changing Matrix Format: Blocking

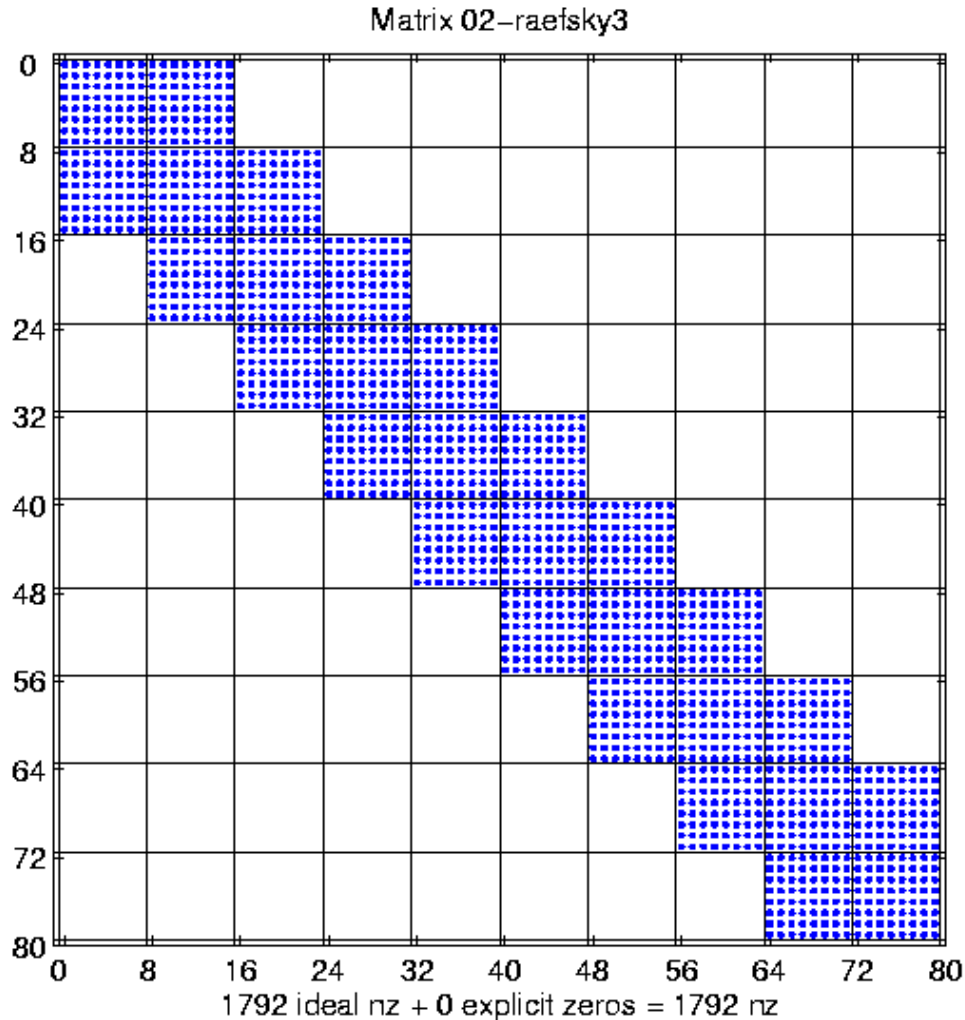


- $n = 21200$
- $\text{nnz} = 1.5 \text{ M}$
- kernel: SpMV
- Source: NASA structural analysis problem

<https://sparse.tamu.edu/>



# Changing Matrix Format: Blocking



- $n = 21200$
- $\text{nnz} = 1.5 \text{ M}$
- kernel: SpMV
- Source: NASA structural analysis problem
- **8x8** dense substructure

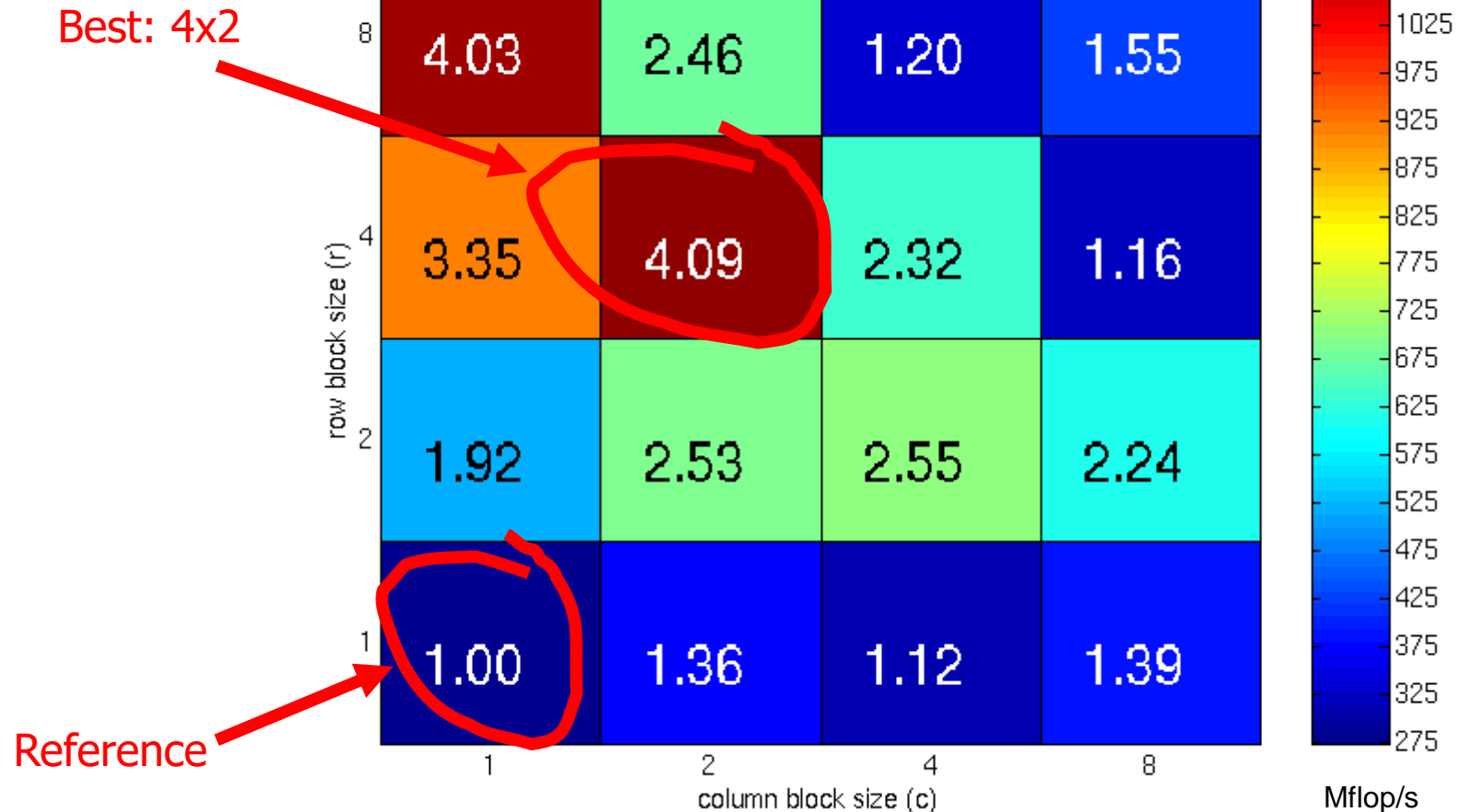
# Taking advantage of block structure in SpMV

- Bottleneck is time to get matrix from memory
  - Only 2 flops for each nonzero in matrix
  - Fetching at  $\sim 1$  int (column index) + 1 float (value) for 2 flops
- Don't store each nonzero with index, instead store each nonzero r-by-c block with 1 column index
  - As  $r \times c$  grows, storage drops by up to 2x, for all 32-bit quantities
  - Time to fetch matrix from memory decreases
- Change both data structure and algorithm
  - Need to pick  $r$  and  $c$
  - Need to change algorithm accordingly
- In example, is  $r=c=8$  best choice?
  - Minimizes storage, so looks like a good idea...
- Consider best case: dense matrix in sparse format

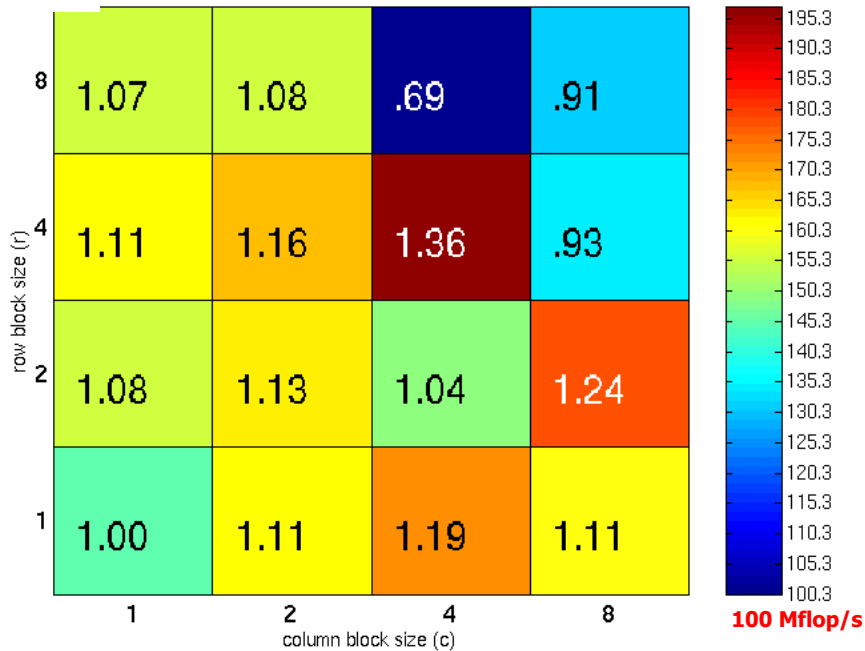
# The Need for Search

## Itanium 2 processor

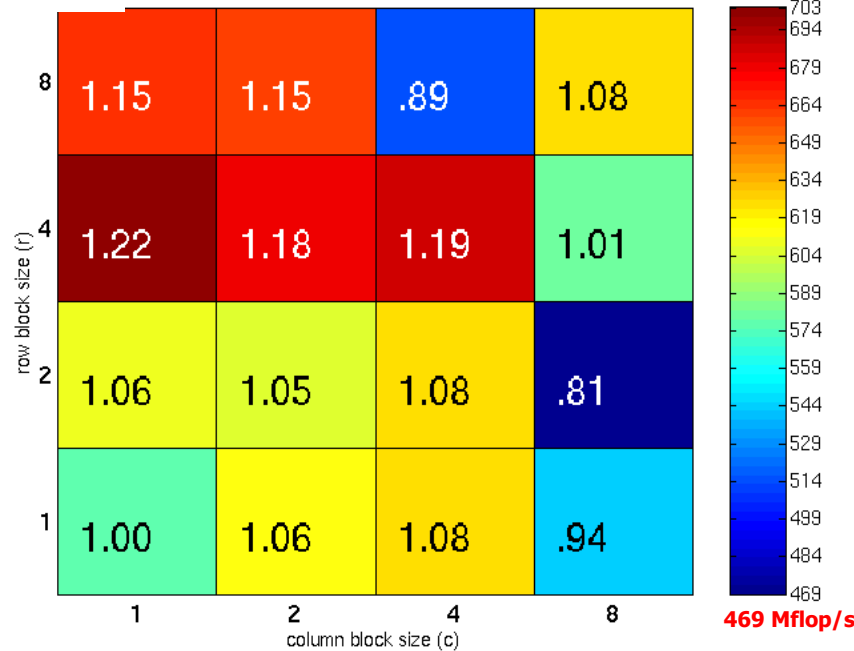
Matrix #02-raefsky3.rua on Itanium 2 (900 MHz) [Ref=274.3 Mflop/s]



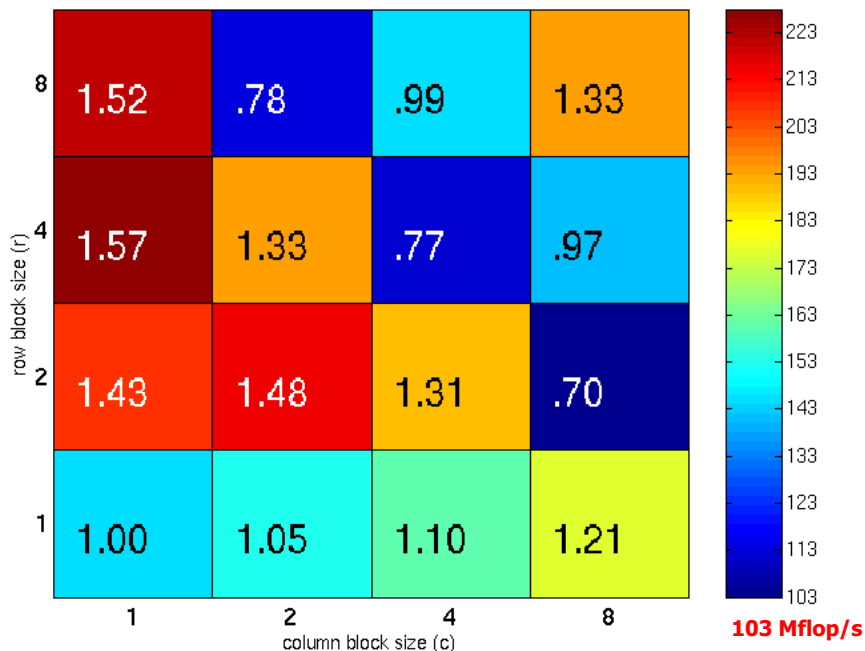
**Power3** Performance: raefsky3.rua [ref=144.7 Mflop/s; 375 MHz Power3, IBM xlc v6]



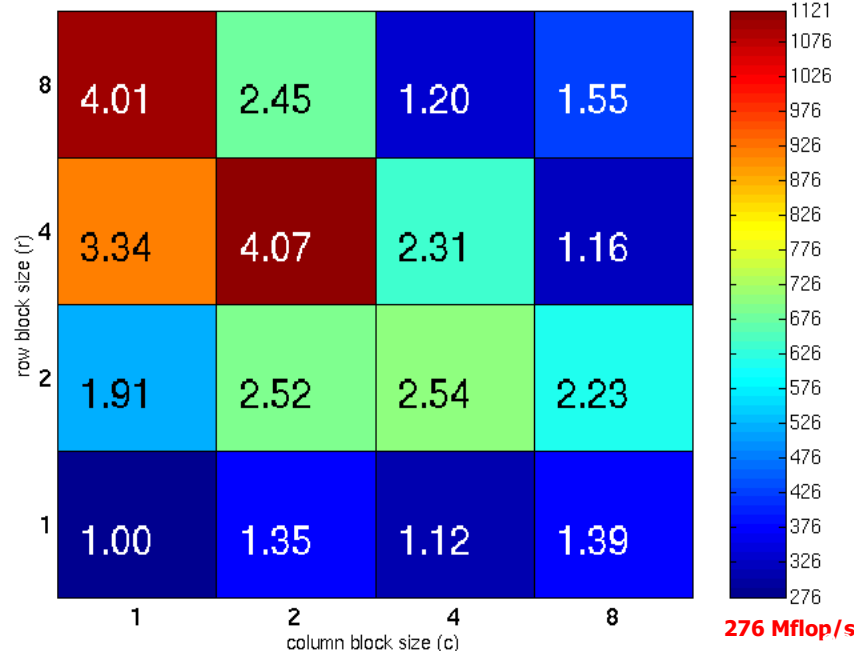
**Power4** Performance: raefsky3.rua [ref=576.9 Mflop/s; 1.3 GHz Power4, IBM xlc v6]



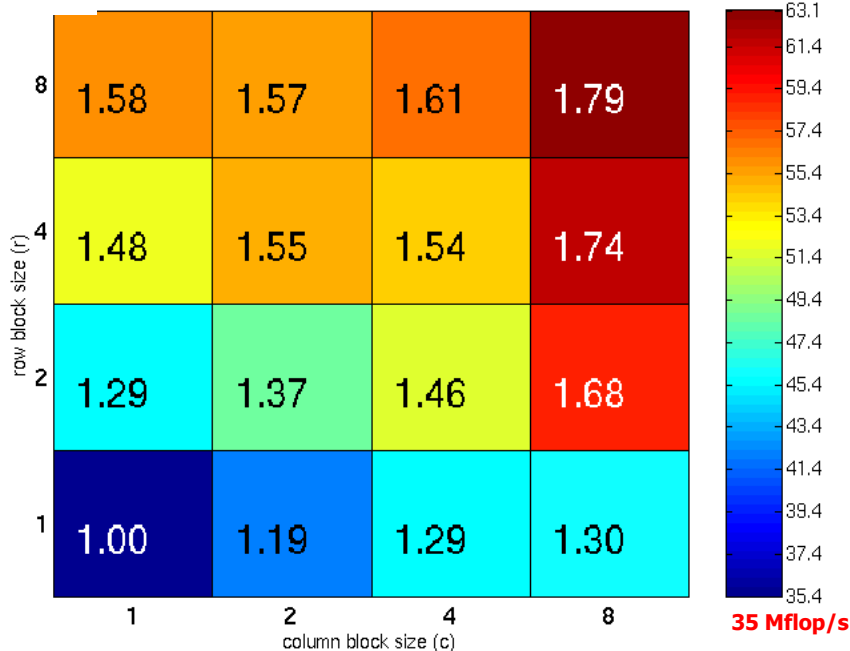
**Itanium 1** Performance: raefsky3.rua [ref=145.8 Mflop/s; 800 MHz Itanium, Intel C v7]



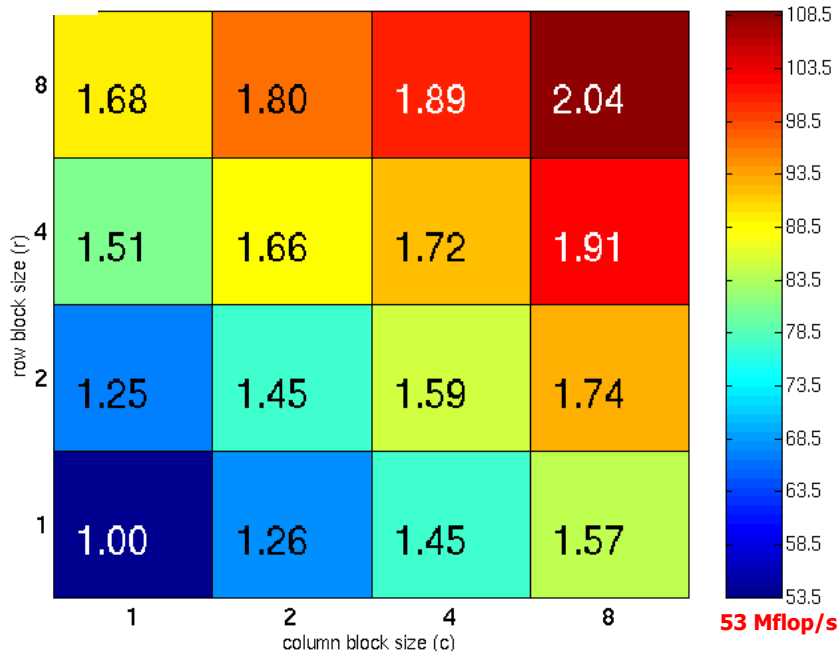
**Itanium 2** Performance: raefsky3.rua [ref=275.3 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]



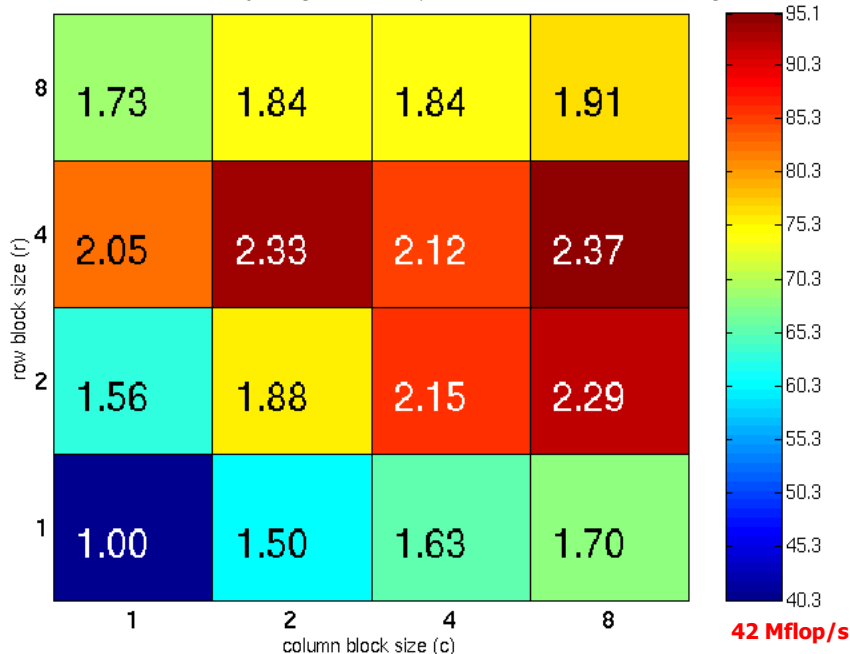
**Ultra 2i** performance: raefsky3.rua [ref=35.3 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]



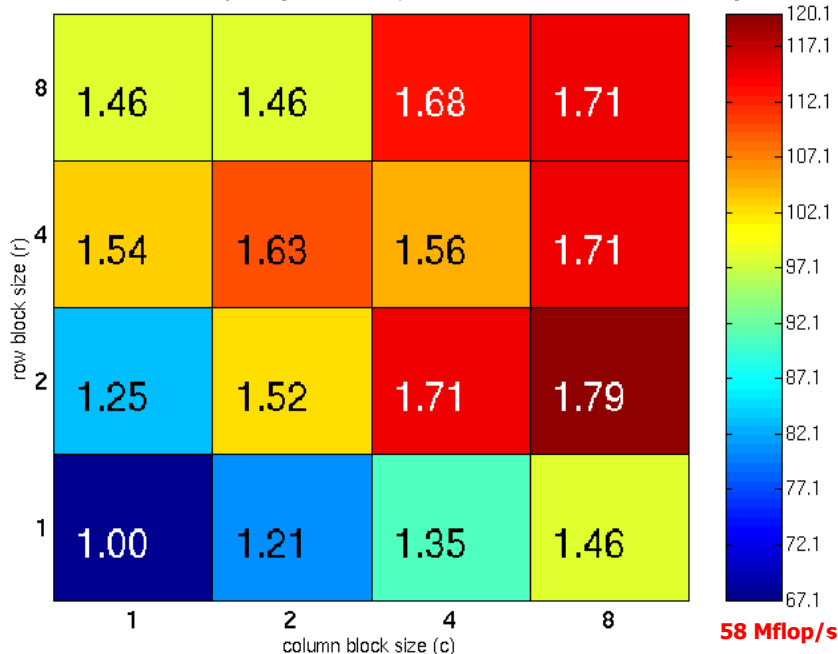
**Ultra 3** performance: raefsky3.rua [ref=53.5 Mflop/s; 900 MHz Ultra 3, Sun CC v6]



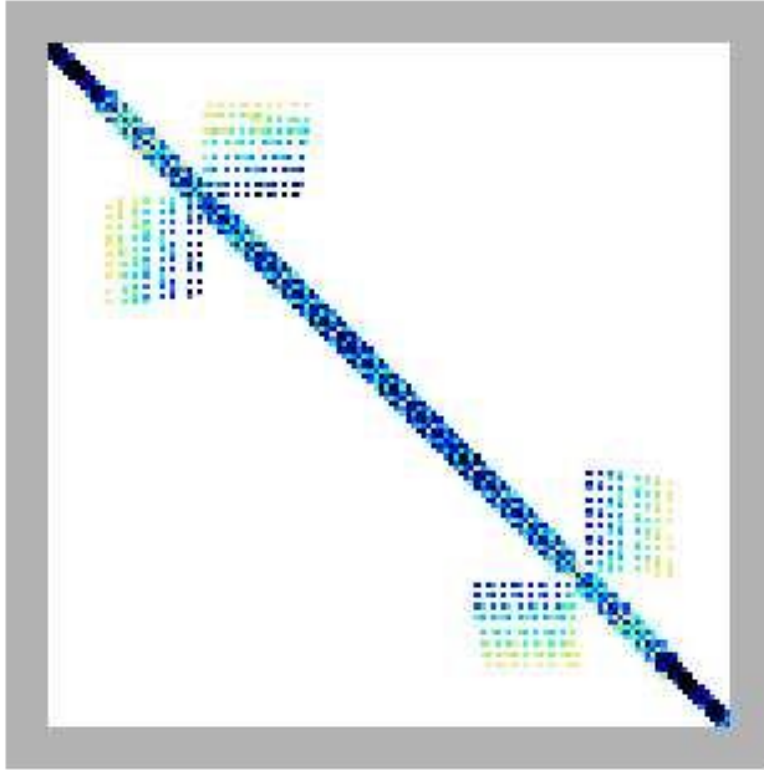
**Pentium III** performance: raefsky3.rua [ref=40.2 Mflop/s; 500 MHz Pentium III, Intel C v7.0]



**Pentium III-M** performance: raefsky3.rua [ref=67.1 Mflop/s; 800 MHz Pentium III-M, Intel C v7.0]

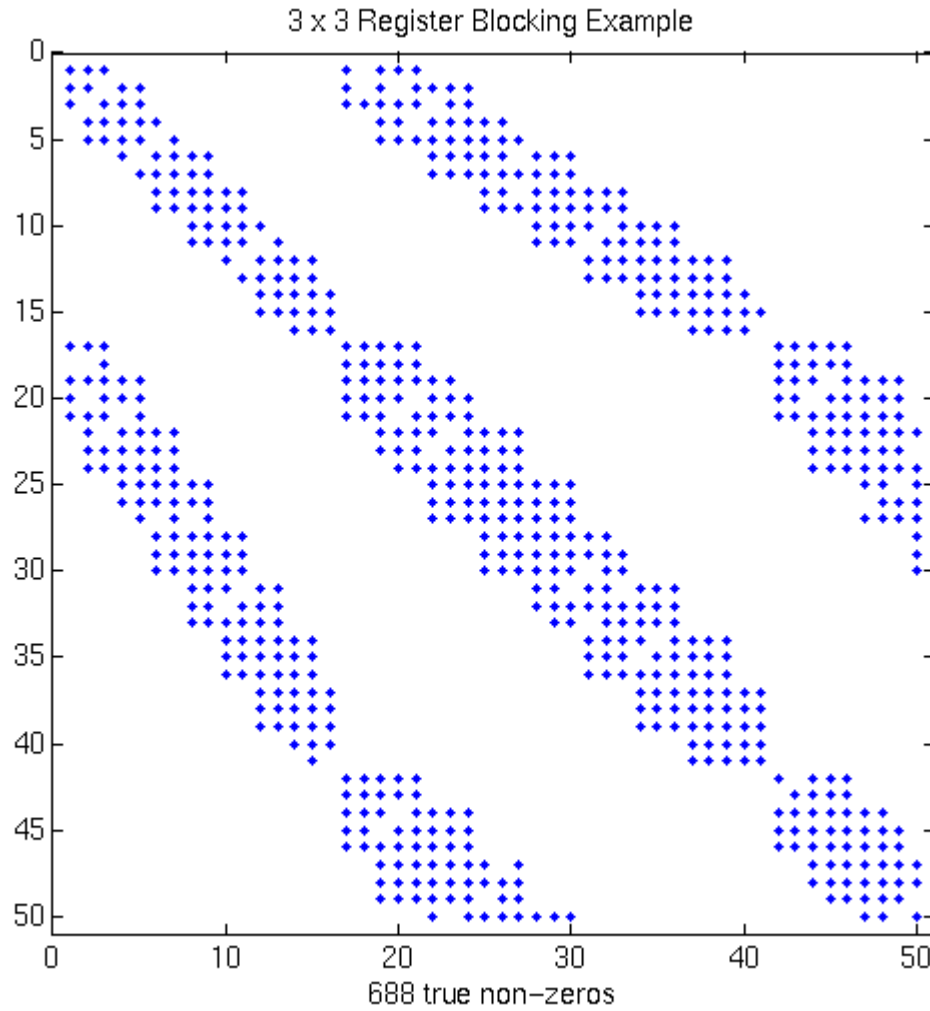


# But most matrices don't block so easily



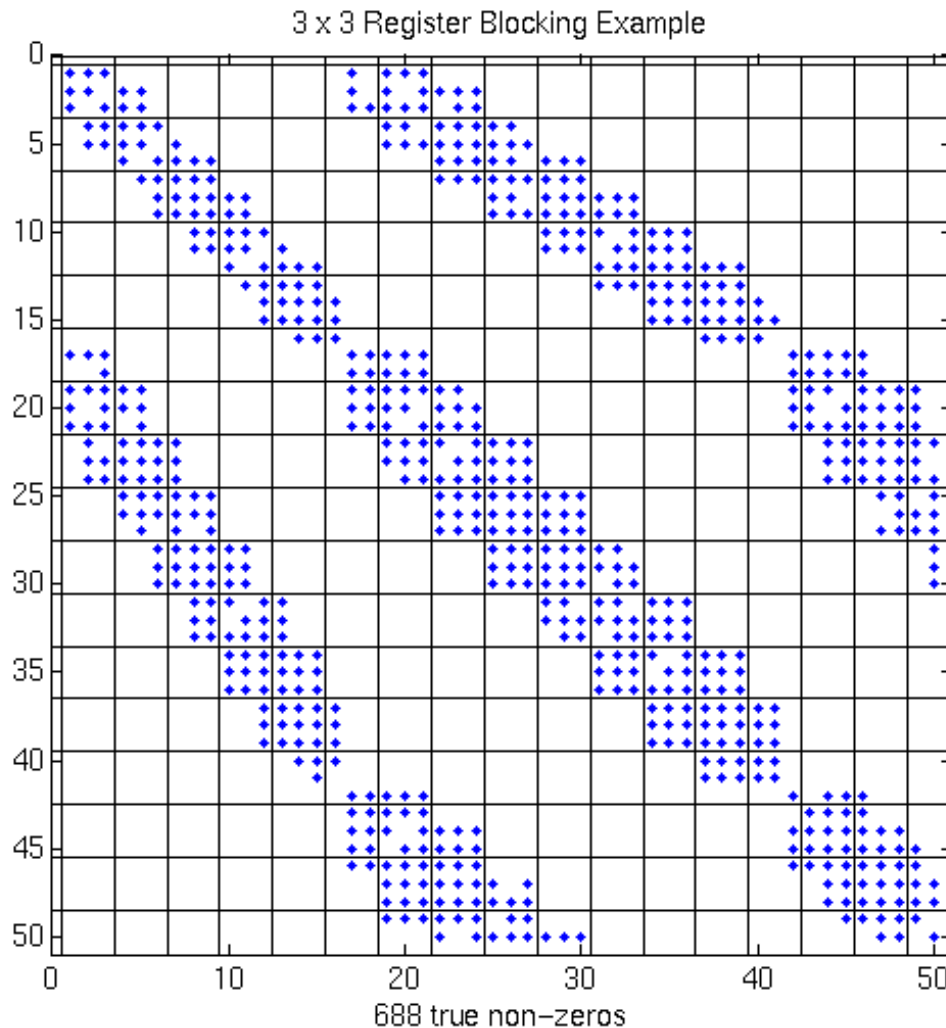
- FEM Fluid dynamics problems
- More complicated non-zero structure in general
- $N = 16614$
- $NNZ = 1.1M$

# Zoom in to top corner



- More complicated non-zero structure
- $N = 16614$
- $NNZ = 1.1M$

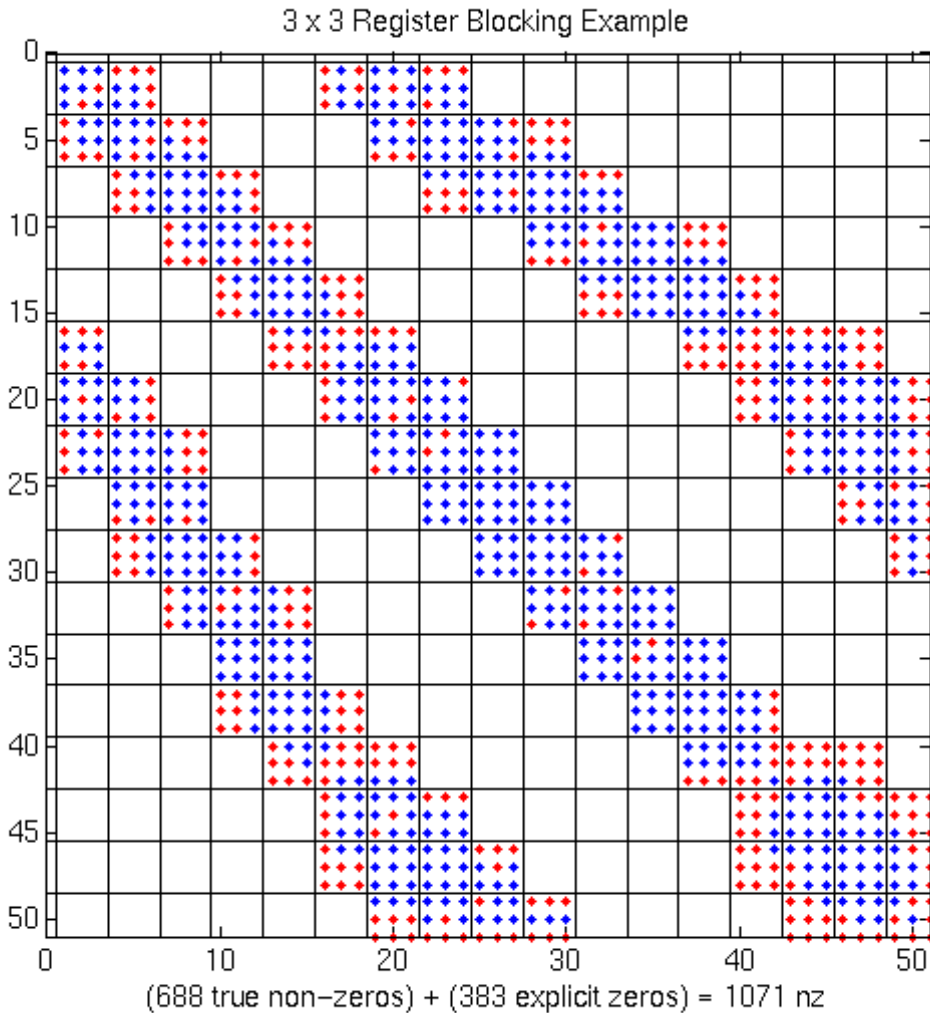
# 3x3 blocks look natural, but...



- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Many cell are not full
- $N = 16614$
- $NNZ = 1.1M$



# Extra work can improve efficiency

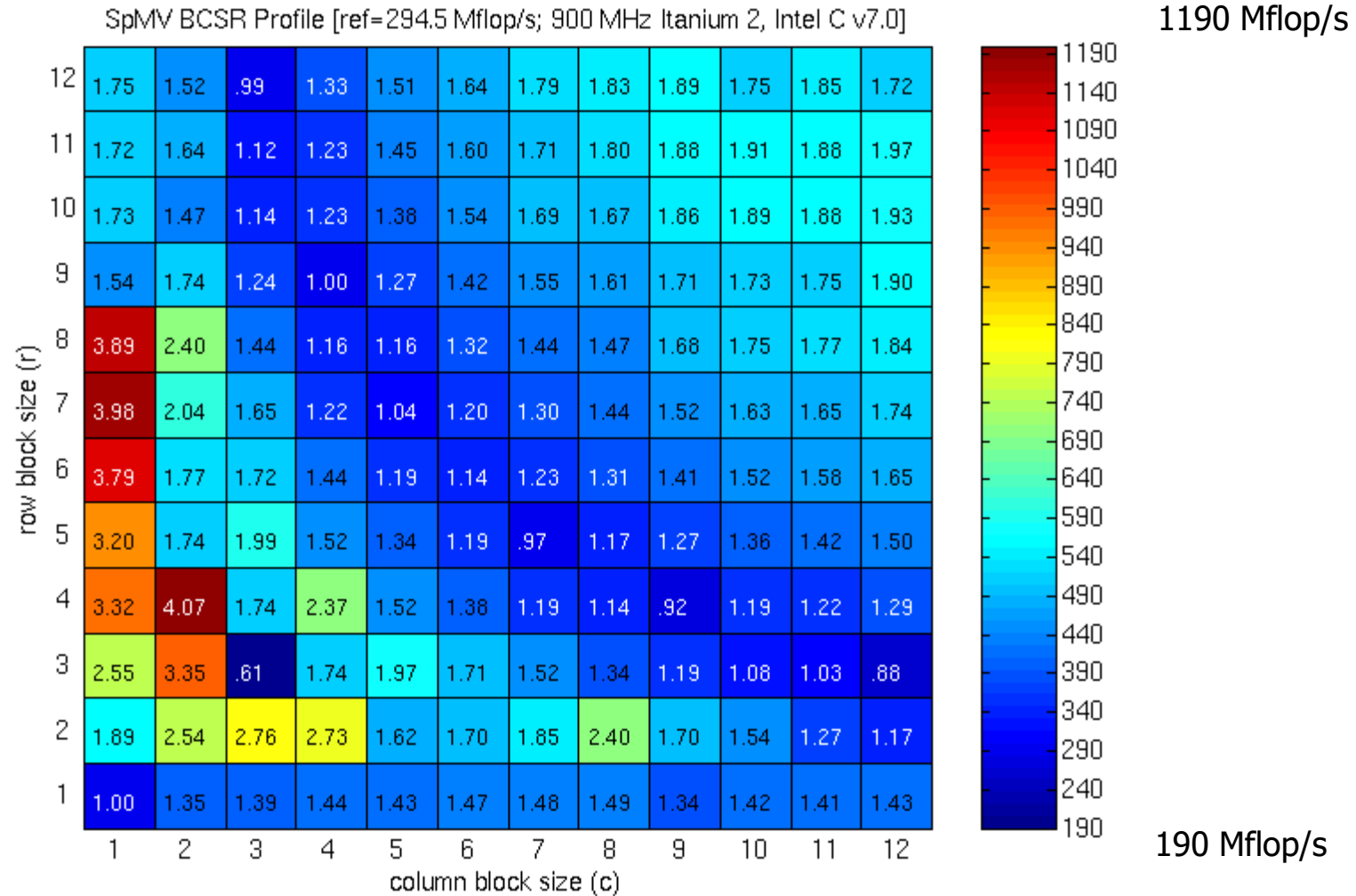


- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Add explicit zeros: 1.5x “fill overhead”
  - Unroll loops
- More work but can be faster

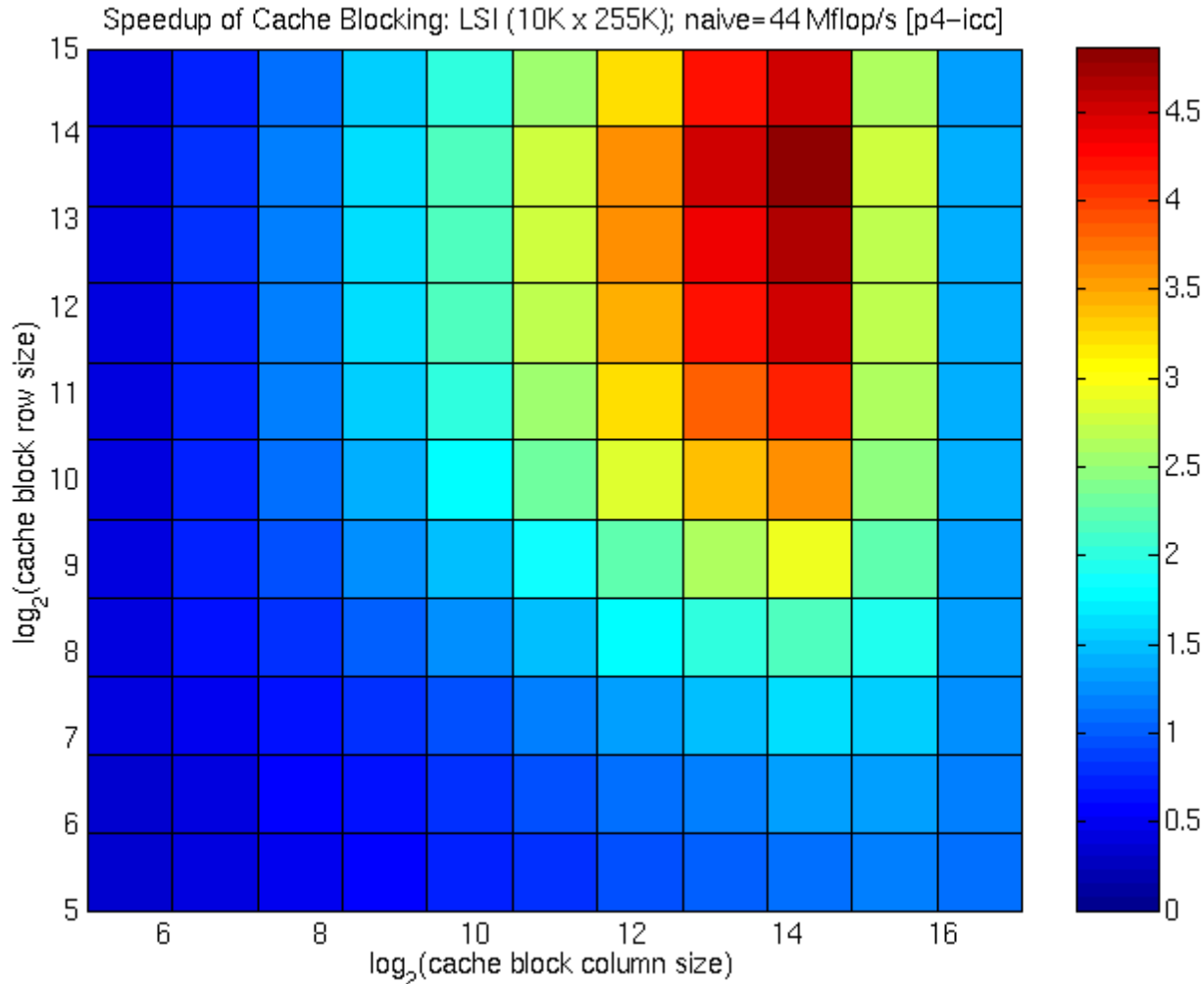
# Automatic Register Block Size Selection

- Selecting the  $r \times c$  block size
  - Off-line benchmark of “register profile”
    - Precompute  $\text{Mflops}(r,c)$  using *dense  $A$  in sparse format (blocked sparse row)* for each  $r \times c$
    - Once per machine/architecture
  - Run-time “search”
    - Sample  $A$  to estimate  $\text{Fill}(r,c)$  for each  $r \times c$
  - Run-time heuristic model
    - Choose  $r, c$  to minimize  $\text{time} \approx \text{Fill}(r,c) / \text{Mflops}(r,c)$

# Register Profile: dense matrix in sparse format



# Cache Blocking on LSI Matrix: Pentium 4



**A**  
10k x 255k  
3.7M non-zeros

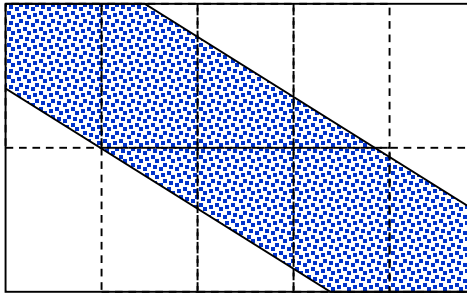
**Baseline:**  
44 Mflop/s

**Best block size  
& performance:**  
16k x 16k  
210 Mflop/s

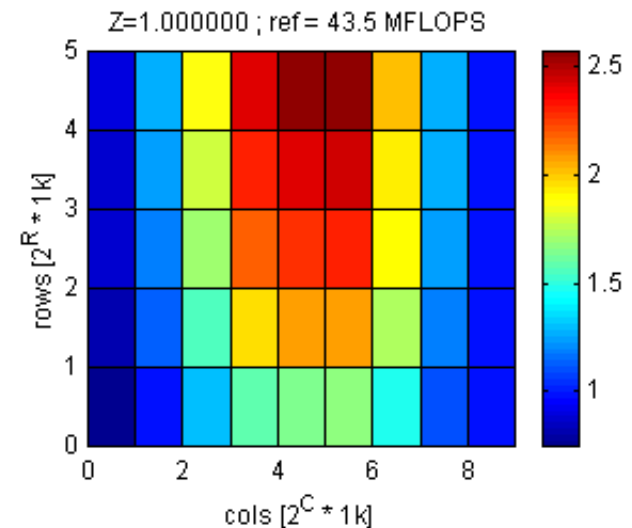
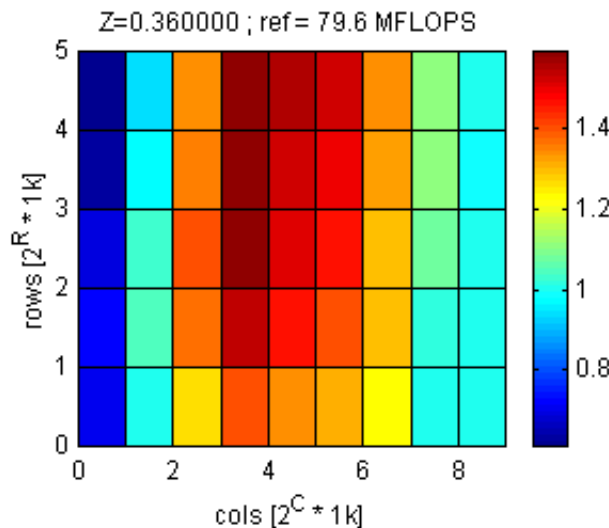
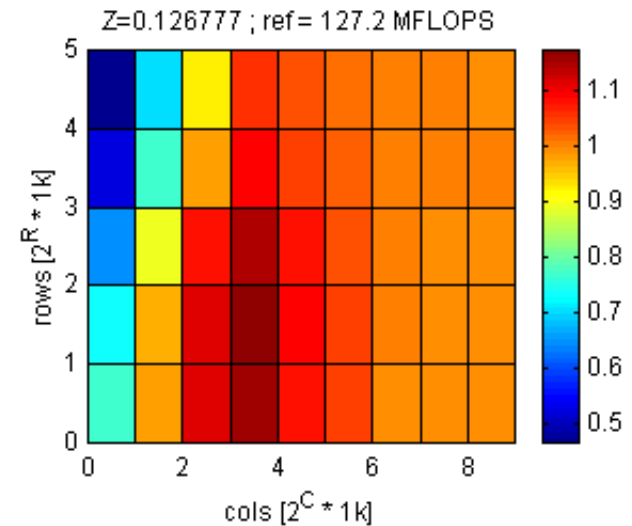
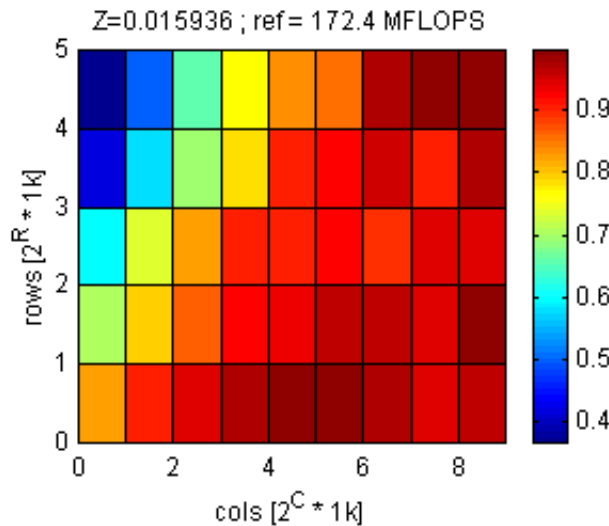
*Nishtala, et al (2007). When  
cache blocking of sparse matrix  
vector multiply works and why.*

# Cache Blocking on Random Matrices: Itanium

Speedup on four banded random matrices.



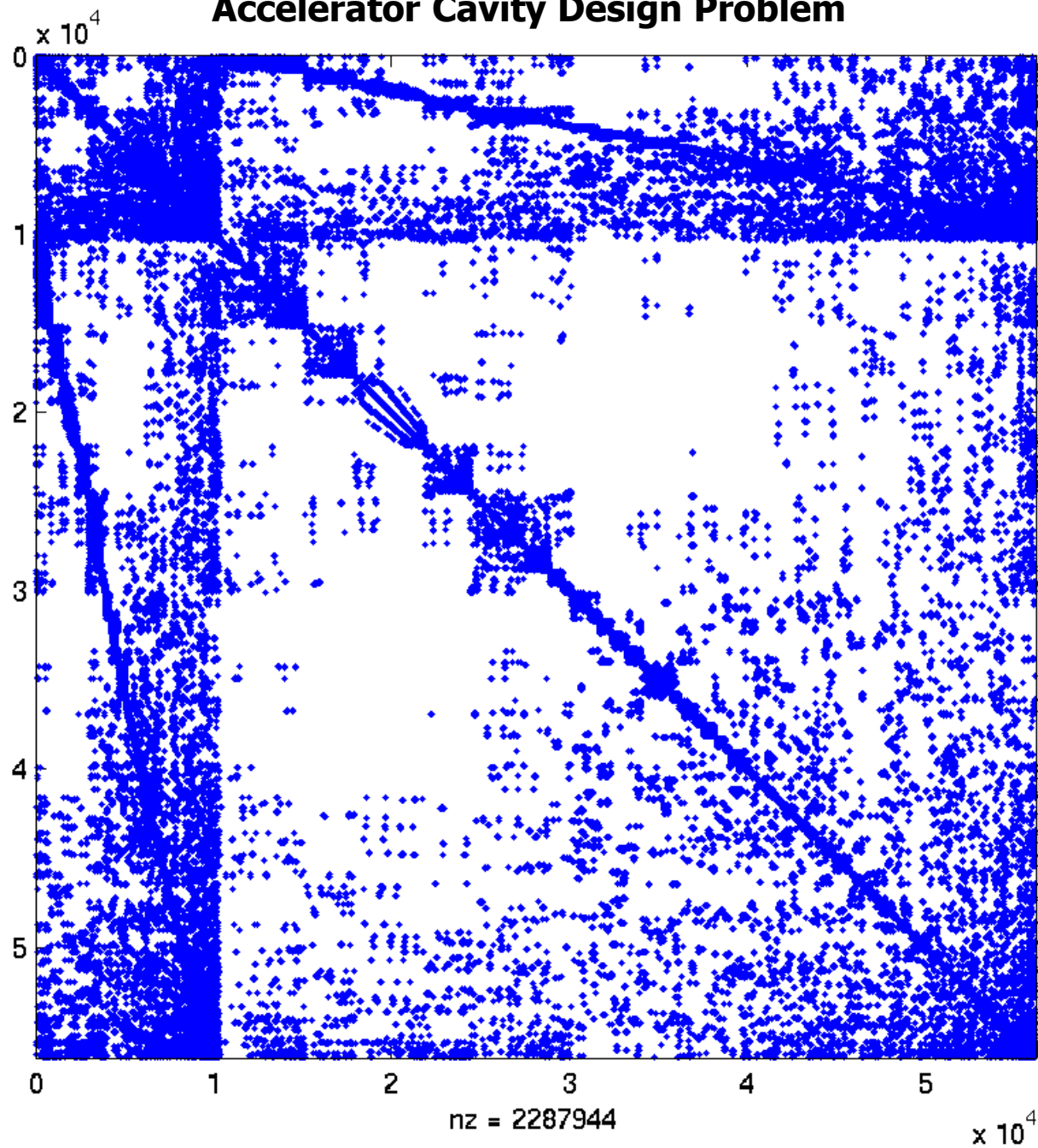
*Nishtala, et al (2007). When cache blocking of sparse matrix vector multiply works and why.*



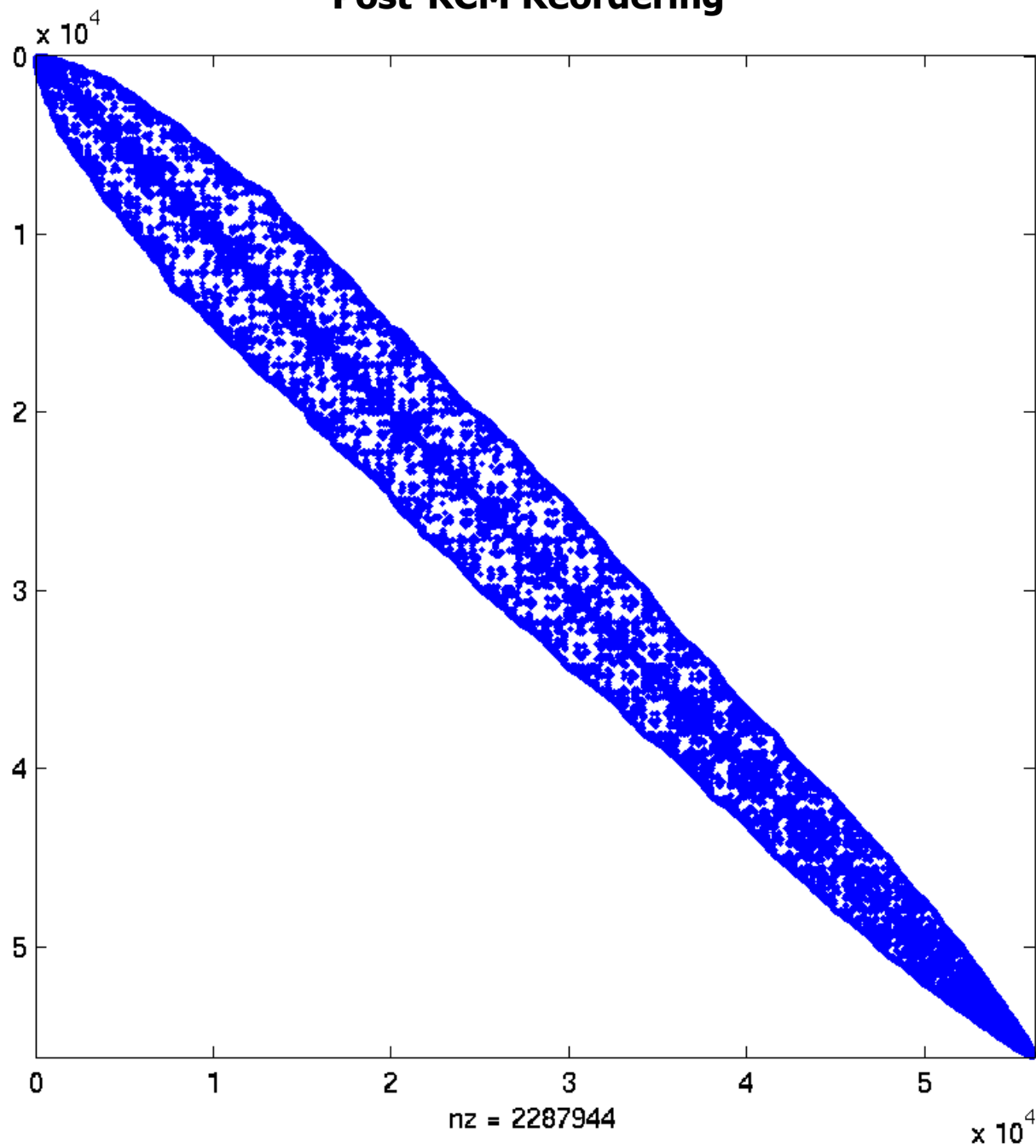
# Matrix Reordering: Example

- Application: accelerator cavity design
- Optimizations:
  - Reordering, to create more dense blocks
    - Reverse Cuthill-McKee ordering to reduce bandwidth
      - Do Breadth-First-Search, number nodes in reverse order visited
  - Traveling Salesman Problem-based ordering to create blocks
    - Nodes = columns of  $A$
    - $\text{Weights}(u, v)$  = no. of nonzeros  $u, v$  have in common
    - Tour = ordering of columns
    - Choose maximum weight tour
    - See [Pinar & Heath '97]

# Accelerator Cavity Design Problem

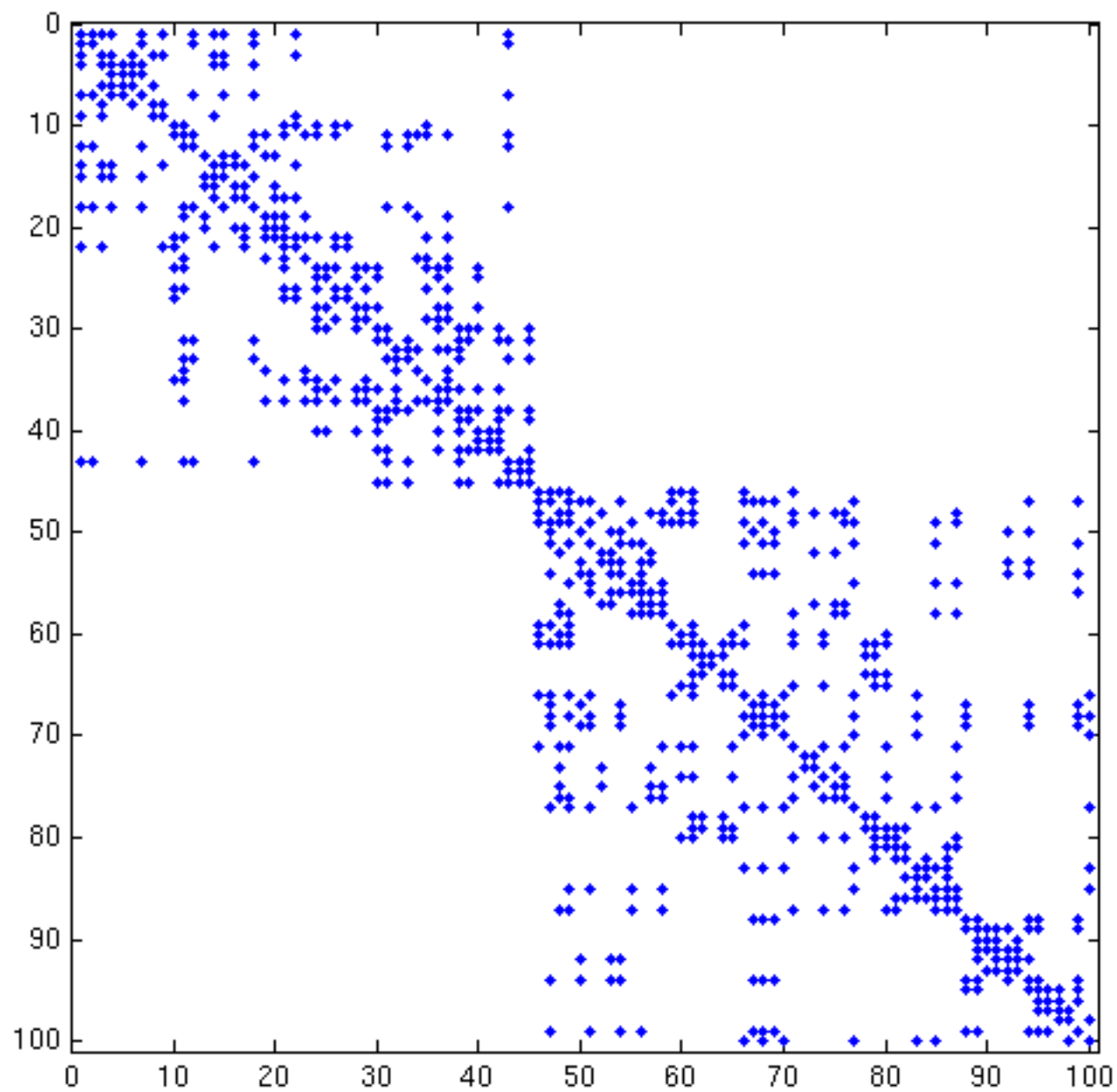


## Post-RCM Reordering

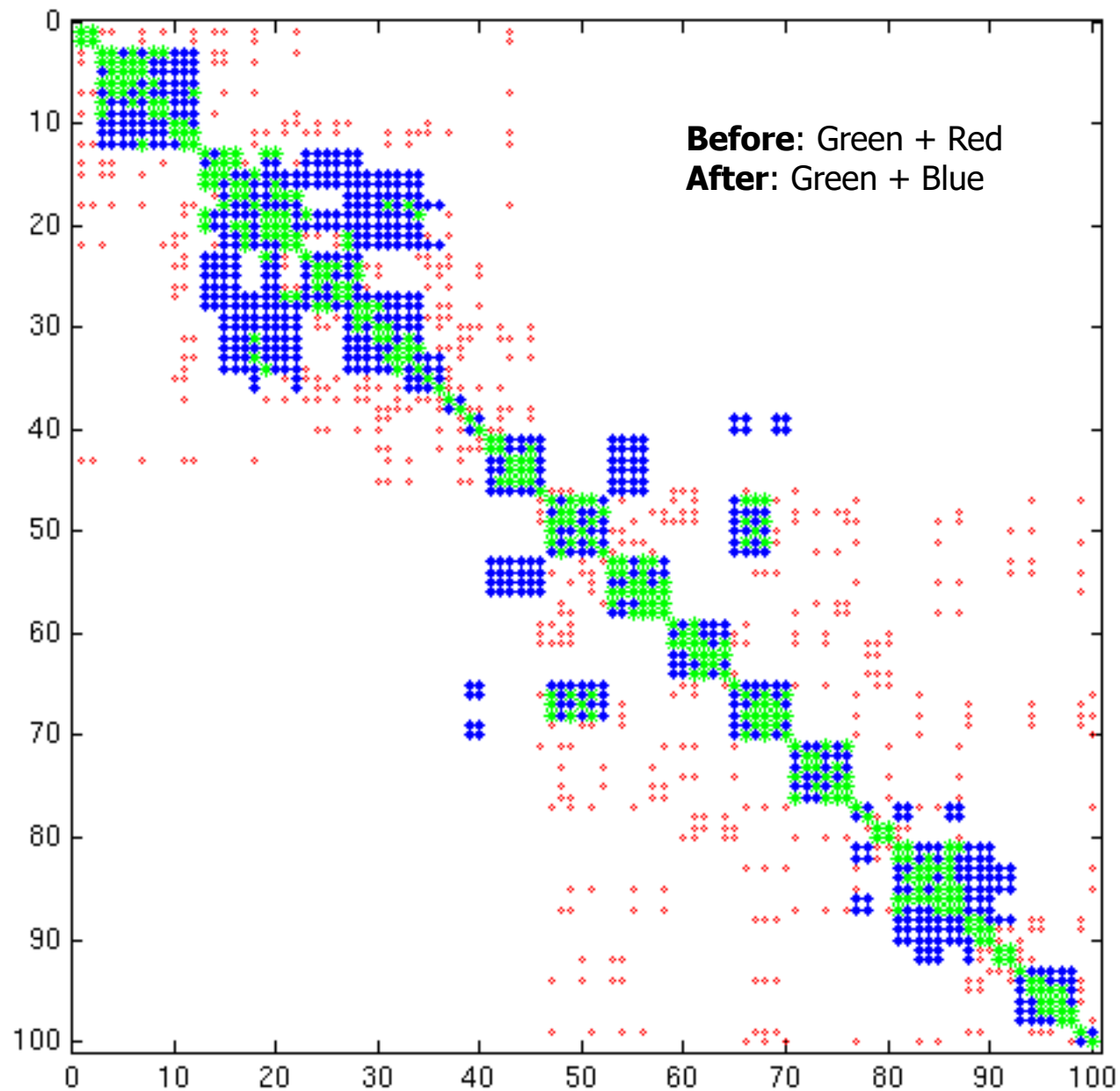




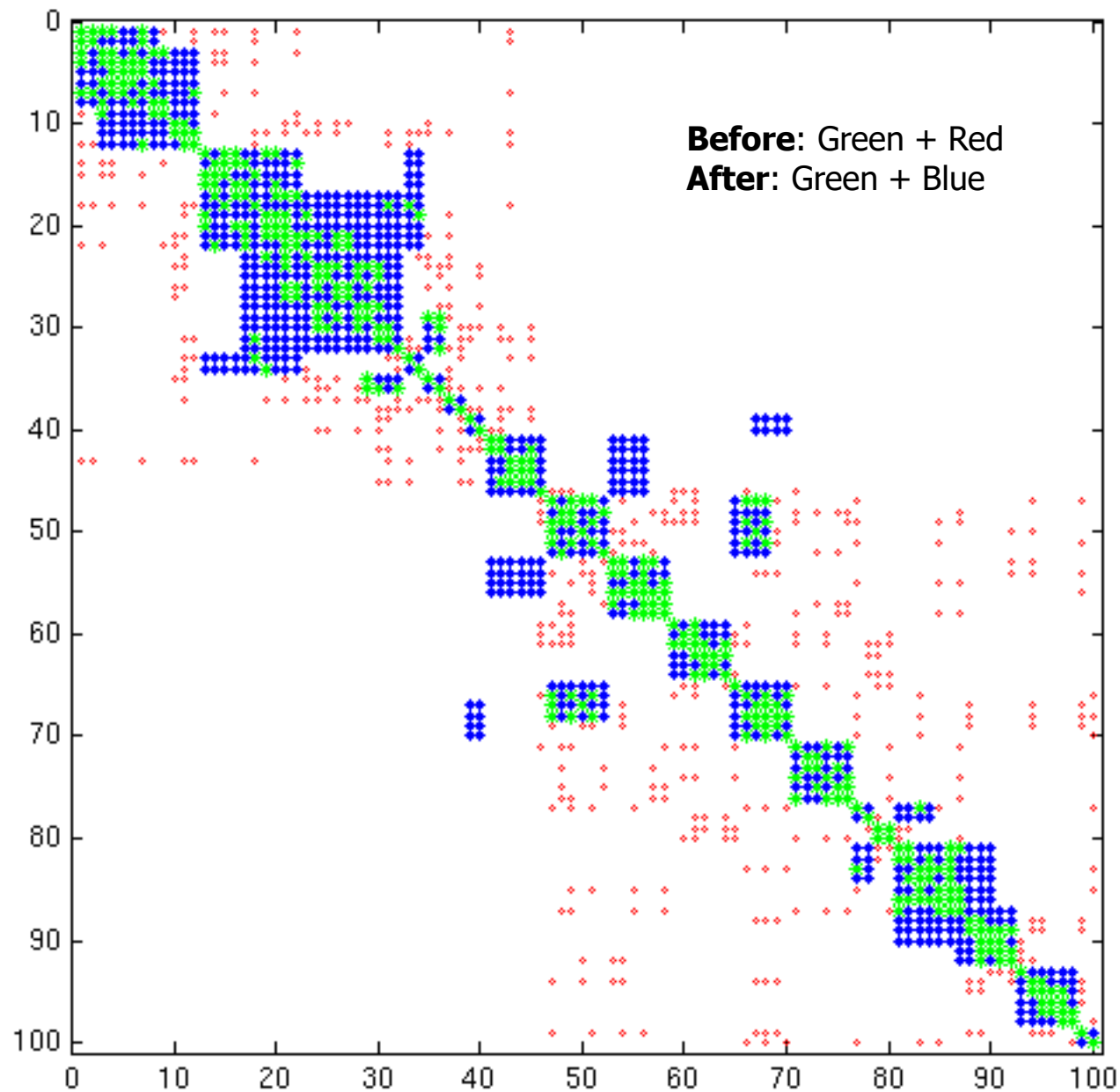
**100x100 Submatrix Along Diagonal**



# “Microscopic” Effect of RCM Reordering



# “Microscopic” Effect of Combined RCM+TSP Reordering

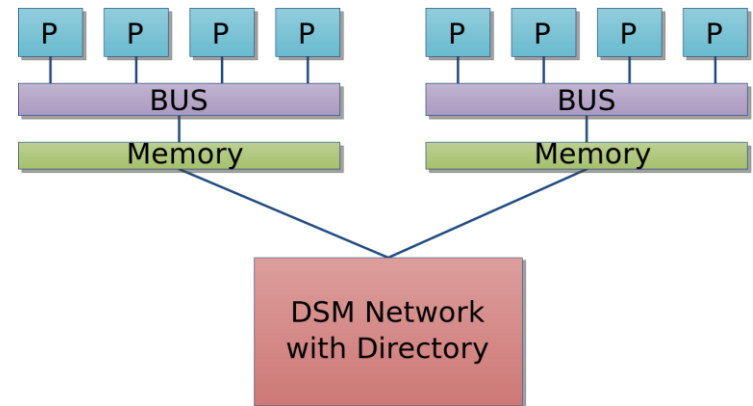


# How do permutations affect algorithms?

- $A$  = original matrix,  $A_P = P_R A P_C$  ( $A$  with permuted rows, columns)
- SpMV: permute  $x$  ( $x_P = P_C^T x$ ), multiply  $y_P = (P_R A P_C)(P_C^T x)$ , permute  $y$  ( $y = P_R^T y_P$ )
- Faster way to solve  $Ax = b$ 
  - Solve  $A_P x_P = P_R b$  for  $x_P$ , using SpMV with  $A_P$ , then let  $x = P_C x_P$
  - Only need to permute vectors twice, not twice per iteration
- Faster way to solve  $Ax = \lambda x$ 
  - $A$  and  $A_P$  have same eigenvalues, no vectors to permute!
  - $A_P x_P = \lambda x_P$  implies  $Ax = \lambda x$  where  $x = P_C x_P$

# Shared-Memory Multicore Optimizations

- NUMA - Non-Uniform Memory Access
  - pin submatrices to memories close to cores assigned to them



- Prefetch – values, indices, and/or vectors
  - use exhaustive search on prefetch distance
- Matrix Compression – not just register blocking (BCSR)
  - 32 or 16-bit indices, Block Coordinate format for submatrices
- Cache-blocking
  - 2D partition of matrix, so needed parts of x,y fit in cache

# Distributed-memory parallel SpMV

- Harder to make general statements about performance:
  - Many ways to partition  $x$ ,  $y$ , and  $A$  processors
  - Communication, computation, and load-balance are partition-dependent
- A parallel SpMV involves 1 or 2 rounds of messages
  - (Sparse) collective communication, costly synchronization
    - Latency-bound (hard to saturate network bandwidth)
  - Scatter entries of  $x$  and/or gather entries of  $y$  across network
- $k$  SpMVs cost  $O(k)$  rounds of messages

# Lower bounds and optimal algorithms - parallel

- First require some notion of initial data layout, load balance and/or local memory capacity
- Classical algorithm: every processor  $j$  owns matrix  $A^{(j)}$  and computes  $y^{(j)} = A^{(j)}x$
- $A = \sum_{j=1}^P A^{(j)}$  is a sum of matrices with disjoint nonzero structures.
- vectors  $x, y$  are distributed across the  $P$  processors, and their layout, along with the splitting of  $A$ , determines the communication cost
  - zero or more entries of  $x$  are communicated
  - zero or more entries of  $y$  are computed by a reduction over the (sparse) vectors  $y^{(j)}$
- assume a load-balanced parallelization among  $P \geq 2$  processors, where at least two processors perform at least  $\text{nnz}/P$  flops.

# Hypergraph model

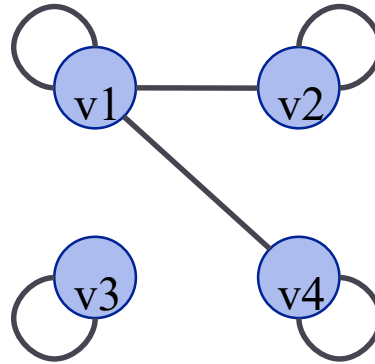
- Hypergraph: generalization of a graph where “edge” can connect more than 2 vertices
- Communication costs for parallel SpMV without data replication (implicit or explicit storage) can be **exactly modelled** by a hypergraph constructed from the computation's DAG
  - see [Catalyurek and Aykanat, 2001]
- Vertices represent matrix nonzeros and the hyperedges contain the vertices adjacent to incoming (resp. outgoing) edges, of each vertex in the graph of  $A$ .
- vertex partition = parallelization of the classical SpMV computations
  - induced hyperedge cut corresponds to interprocessor communication for that parallelization.
- By varying the metric applied to the cut, one can exactly measure communication volume (number of words moved) or synchronization (number of messages between processors) on a distributed-memory machine.
- Various heuristics are applied to find approximate solutions to these NP-hard partitioning problems in practice and mature software packages are available: see, for example, Devine *et al.* (2006).



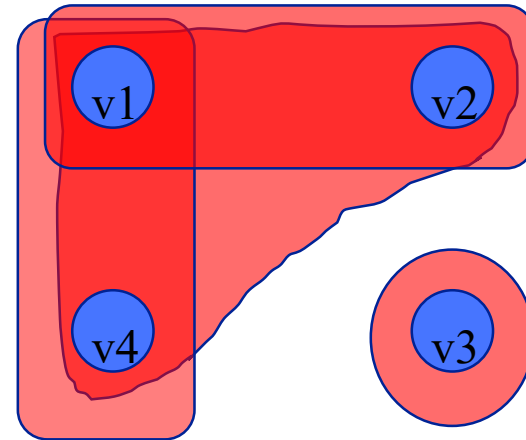
# Hypergraph Model for Communication in SpMV

$$\begin{bmatrix} \times & \times & 0 & \times \\ \times & \times & 0 & 0 \\ 0 & 0 & \times & 0 \\ \times & 0 & 0 & \times \end{bmatrix}$$

**Pattern Matrix**



**Graph Representation**

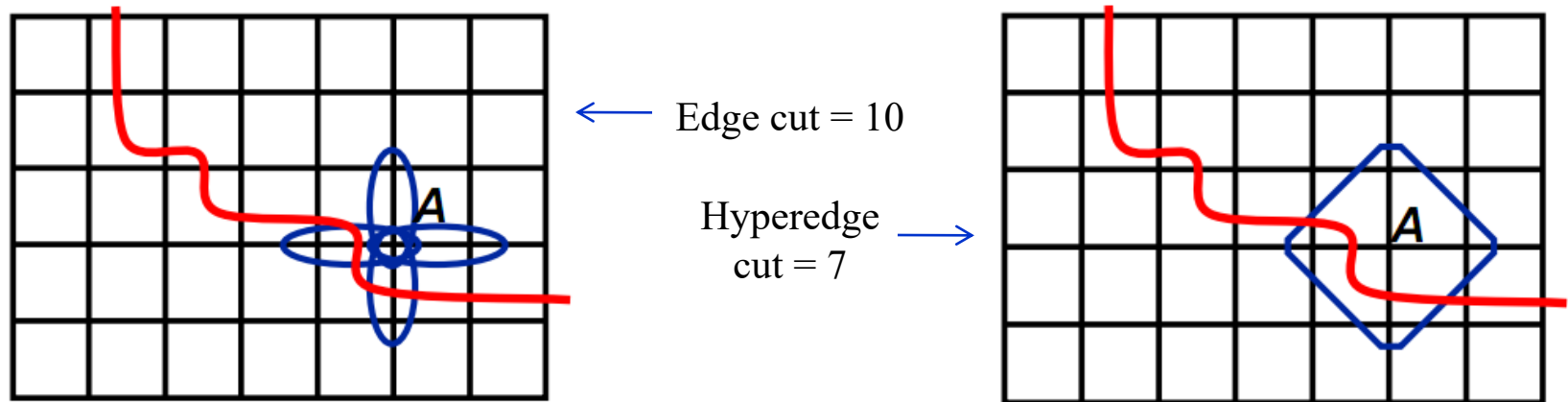


**Hypergraph Representation  
(Column-Net Model)**

- Hypergraph model for row-wise partition (similar for column-wise)
  - Hyperedge for each column, vertex for each row. Vertex  $v_i$  is connected to hyperedge  $e_j$  if  $A(i, j) \neq 0$
- Benefits over graph model:
  - Natural representation of nonsymmetric matrices
  - Cost of hyperedge cut for a given partition is exactly equal to the number of words moved in SpMV operation with the same partition of  $A$

# Graph vs. Hypergraph Partitioning

Consider a 2-way partition of a 2D mesh:



The cost of communicating vertex A is 1 – we can send the value in one message to the other processor

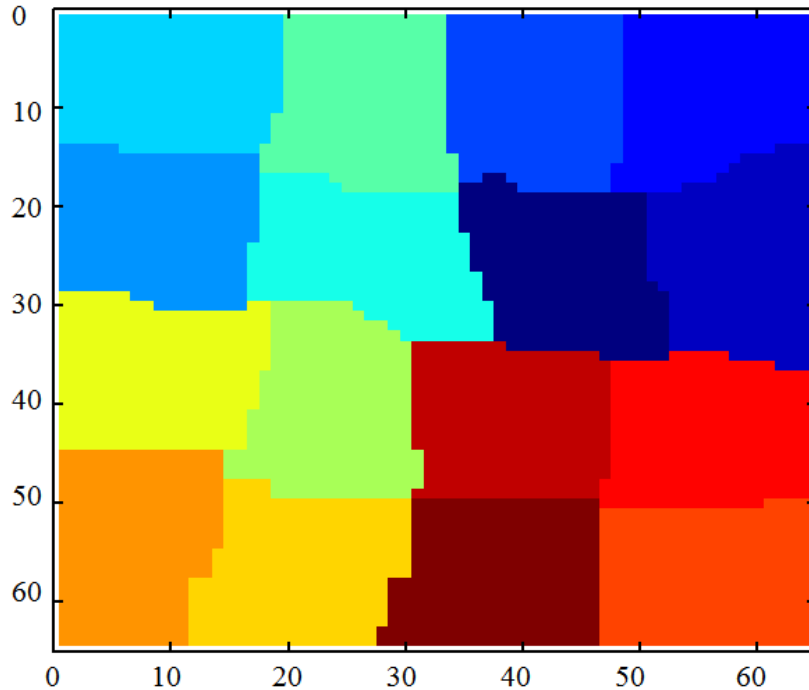
According to the graph model, however the vertex A contributes 2 to the total communication volume, since 2 edges are cut.

The hypergraph model accurately represents the cost of communicating A (one hyperedge cut, so communication volume of 1).

Unlike graph partitioning model, the hypergraph partitioning model gives exact communication volume (minimizing cut = minimizing communication)

# Example: Hypergraph vs. Graph Partitioning

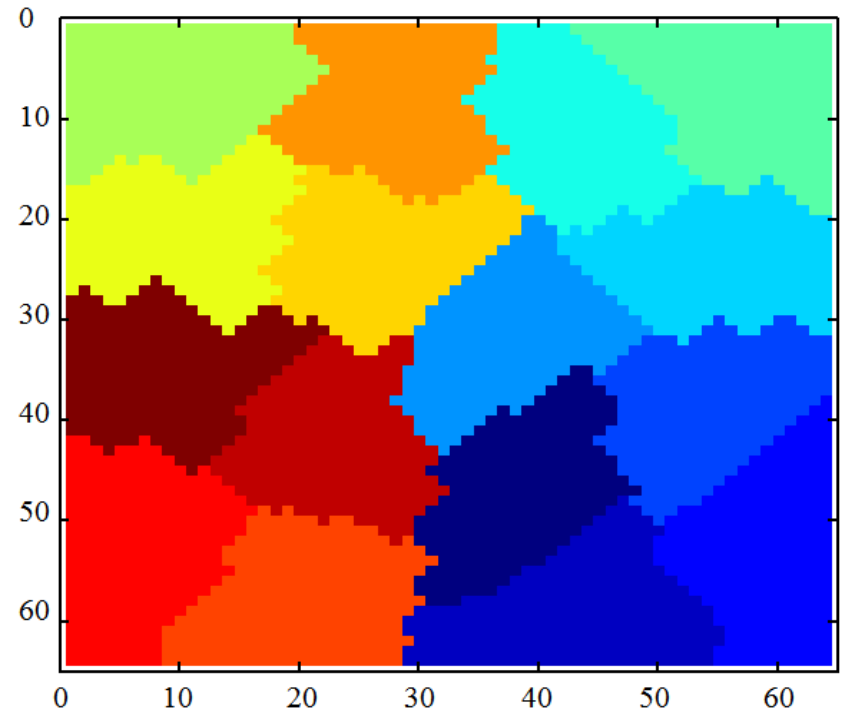
2D Mesh, 5-pt stencil,  $n = 64$ ,  $p = 16$



Graph Partitioning (Metis)

Total Comm. Vol = 777

Max Vol per Proc = 69



Hypergraph Partitioning (PaToH)

Total Comm. Vol = 719

Max Vol per Proc = 59

# Takeaway messages

---

- Tuning for modern processors is hard
- Sparse matrices: tuning harder
- SpMV: low Computational Intensity
- Usual low-level tuning (prefetch, etc.) have some benefit
- Reordering (including graph partitioning) improves locality
- But SpMV will likely still be bandwidth limited

# Is tuning SpMV all we can do?

- Iterative methods all depend on it
- But speedups are limited
  - Just 2 flops per nonzero
  - Communication costs dominate
- Can we beat this bottleneck?
- Need to look at next level in stack:
  - What do algorithms that use SpMV do?
  - Can we reorganize them to avoid communication?
- Only way significant speedups will be possible

# Combining multiple SpMV

(1)  $k$  independent SpMVs

$$[y_0, y_1, \dots, y_k] = A \cdot [x_0, x_1, \dots, x_k]$$

(2)  $k$  dependent SpMVs

$$\begin{aligned} [x_1, x_2, \dots, x_k] &= A \cdot [x_0, x_1, \dots, x_{k-1}] \\ &= [Ax_0, A^2x_0, \dots, A^kx_0] \end{aligned}$$

(3)  $k$  dependent SpMVs,  
in-place variant

$$x = A^k x$$

What if we can amortize cost  
of reading  $A$  over  $k$  SpMVs ?

- ( $k$ -fold reuse of  $A$ )

(1) used in:

- Block Krylov methods
- Krylov methods for multiple systems ( $AX = B$ )

(2) used in:

- $s$ -step Krylov methods/  
Communication-avoiding Krylov  
methods  
...to compute  $k$  Krylov basis vectors

Def. *Krylov space* (given  $A, x, s$ ):

$$K_s(A, x) := \text{span}(x, Ax, \dots, A^s x)$$

(3) used in:

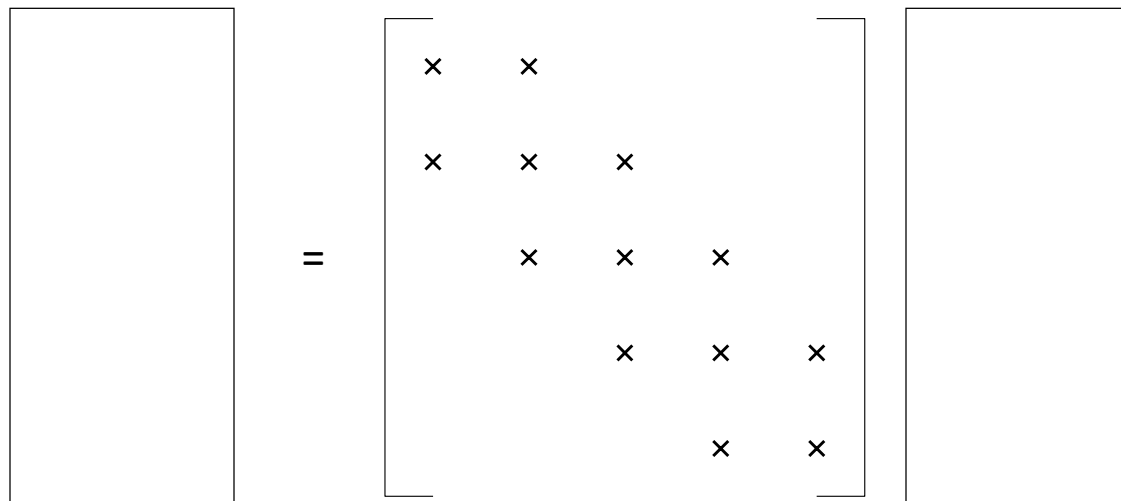
- multigrid smoothers, power  
method

# (1) $k$ independent SpMV (SpMM)

$$[y_0, y_1, \dots, y_k] = A \cdot [x_0, x_1, \dots, x_k]$$

SpMM optimization:

- Compute row-by-row
- Stream  $A$  only once



	1 SpMV	$k$ independent SpMVs	$k$ independent SpMVs (using SpMM)
flops	$2 \cdot nnz$	$2k \cdot nnz$	$2k \cdot nnz$
words moved	$nnz + 2n$	$k \cdot nnz + 2kn$	$1 \cdot nnz + 2kn$
arith. intensity, $nnz = \omega(n)$	2	2	$2k$

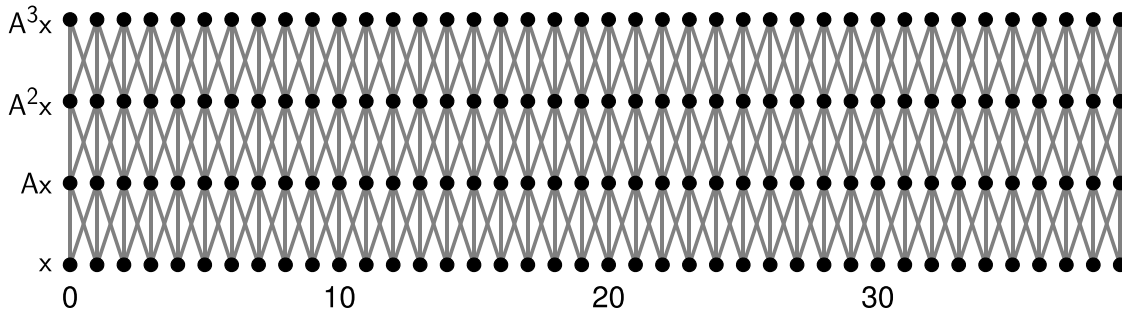
## (2) $k$ dependent SpMVs ( $A^k x$ )

$$\begin{aligned} [x_1, x_2, \dots, x_k] &= A \cdot [x_0, x_1, \dots, x_{k-1}] \\ &= [Ax_0, A^2 x_0, \dots, A^k x_0] \end{aligned}$$

Naïve algorithm (no reuse):

$A^k x$  ( $A^k x$ ) optimization:

- Must satisfy data dependencies while keeping working set in cache

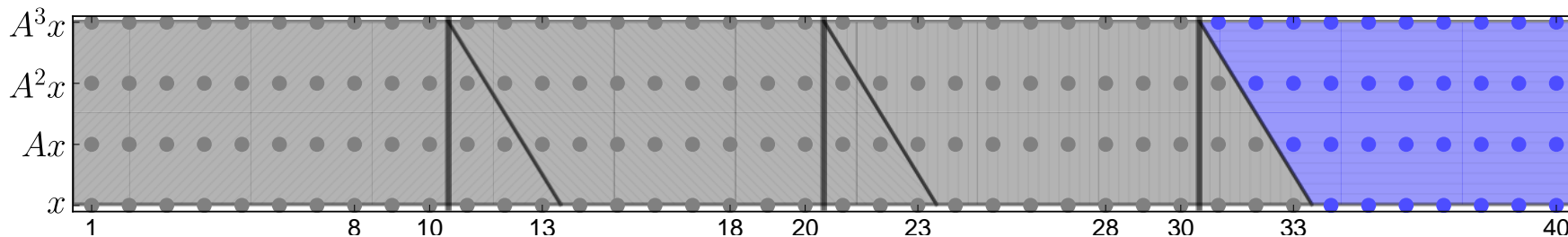


	1 SpMV	$k$ dependent SpMVs	$k$ dependent SpMVs (using $A^k x$ )
flops	$2 \cdot nnz$	$2k \cdot nnz$	$2k \cdot nnz$
words moved	$nnz + 2n$	$k \cdot nnz + 2kn$	$1 \cdot nnz + (k+1)n$
arith. intensity, $nnz = \omega(n)$	2	2	$2k$



## (2) $k$ dependent SpMVs ( $Akx$ )

**Akx algorithm** (reuse nonzeros of  $A$ ):



	1 SpMV	$k$ dependent SpMVs	$k$ dependent SpMVs (using Akx)
flops	$2 \cdot nnz$	$2k \cdot nnz$	$2k \cdot nnz$
words moved	$nnz + 2n$	$k \cdot nnz + 2kn$	$1 \cdot nnz + (k+1)n$
arith. intensity, $nnz = \omega(n)$	2	2	$2k$

### (3) $k$ dependent SpMV, in-place ( $Akx$ , last-vector-only)

$$x = A^k x$$

Last-vector-only  $Akx$  optimization:

- Reuses matrix **and vector**  $k$  times, instead of once.
- Overwrites intermediates without memory traffic
- Attains  $O(k)$  reuse, even when  $nnz < n$ 
  - eg,  $A$  is a stencil (implicit values and structure)

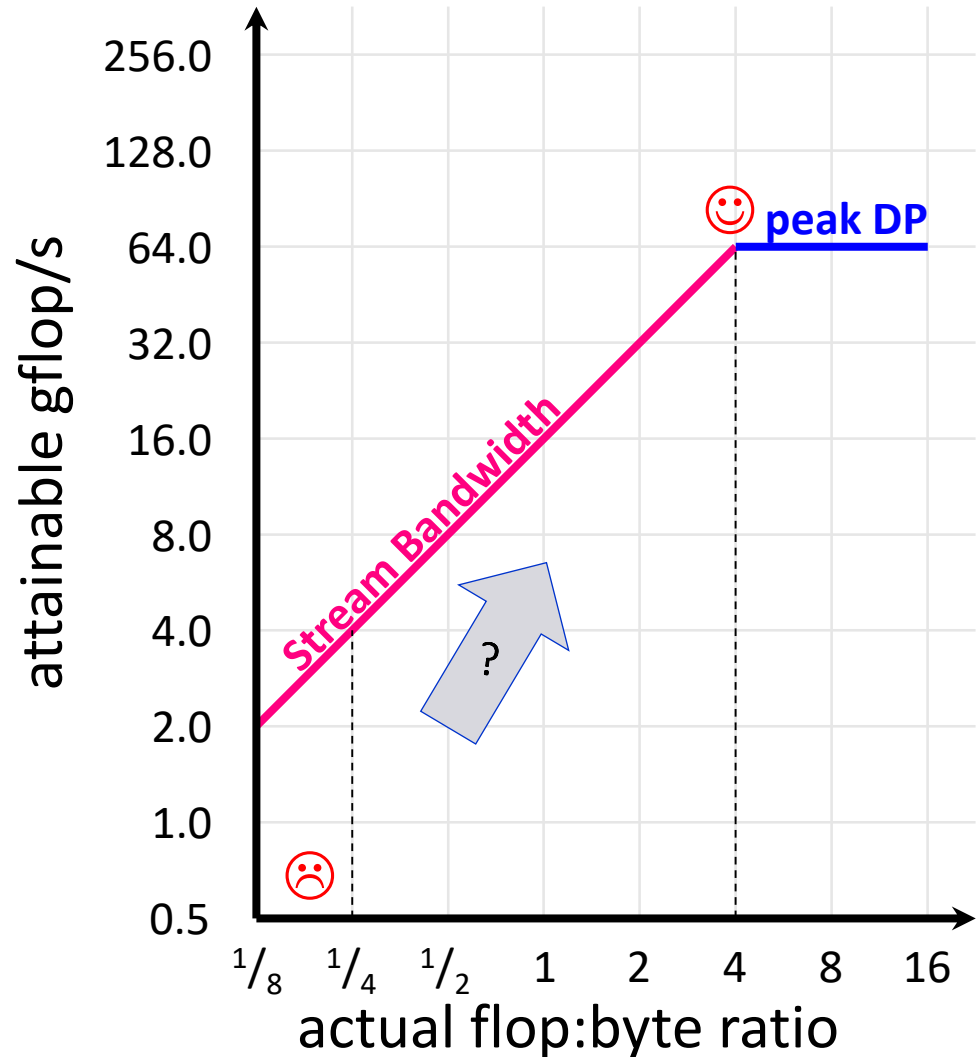
	1 SpMV	$k$ dependent SpMVs, in-place	$Akx$ , last-vector-only
flops	$2 \cdot nnz$	$2k \cdot nnz$	$2k \cdot nnz$
words moved	$nnz + 2n$	$k \cdot nnz + 2kn$	$1 \cdot nnz + 2n$
arith. intensity	2	2	$2k$

# Combining multiple SpMV's (summary of sequential results)

Problem	flops	words moved	optimization	words moved
SpMV	$2 \cdot nnz$	$nnz + 2n$	-	-
$k$ independent SpMV's	$2k \cdot nnz$	$k \cdot nnz + 2kn$	SpMM	$nnz + 2kn$
$k$ dependent SpMV's	$2k \cdot nnz$	$k \cdot nnz + 2kn$	Akx	$nnz + (k+1)n$
$k$ dependent SpMV's, in-place	$2k \cdot nnz$	$k \cdot nnz + 2kn$	Akx, last-vector-only	$nnz + 2n$

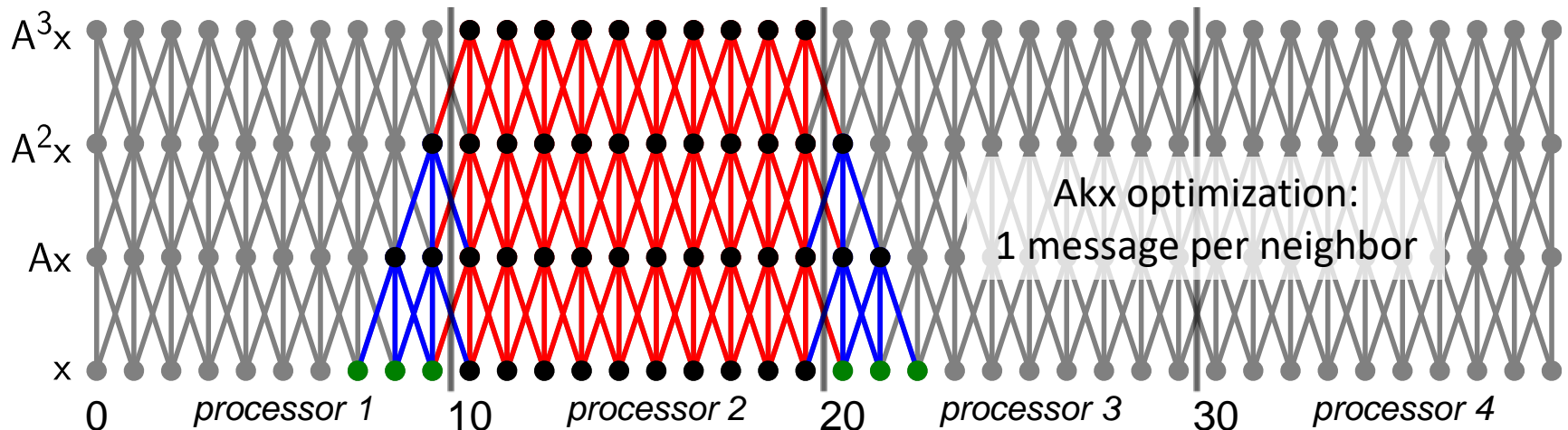
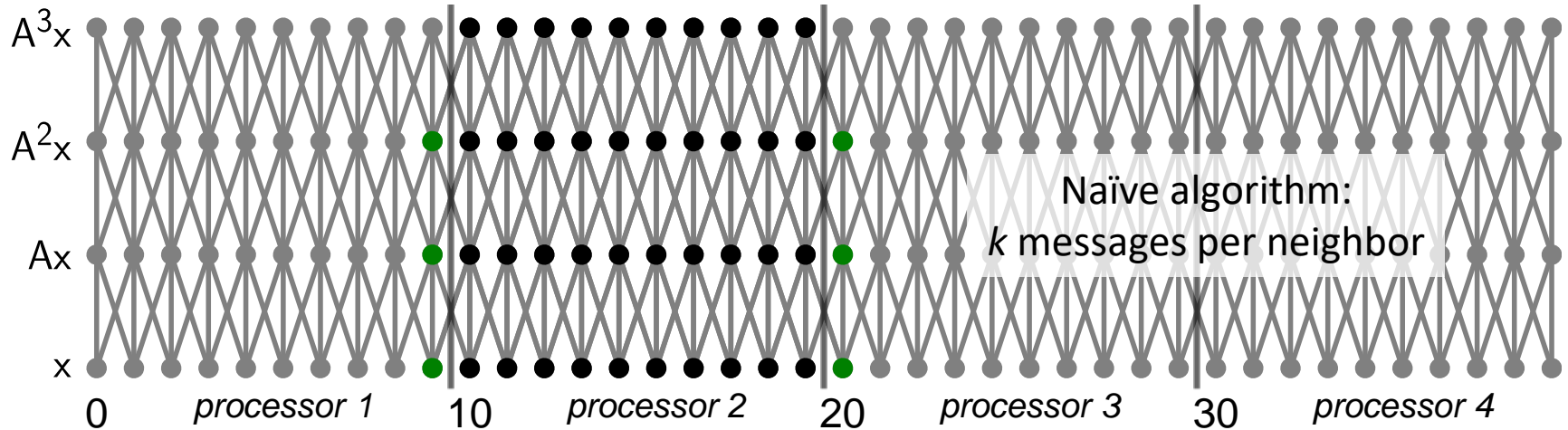
# Avoiding Serial Communication

- Reduce compulsory misses by reusing data:
  - more efficient use of memory
  - decreased bandwidth cost (A<sub>kx</sub>, asymptotic)
- Must also consider *latency* cost
  - How many cachelines?
  - depends on contiguous accesses
- When  $k$  is large  $\Rightarrow$  compute-bound?
- In practice, complex performance tradeoffs.
  - Autotune to find best  $k$

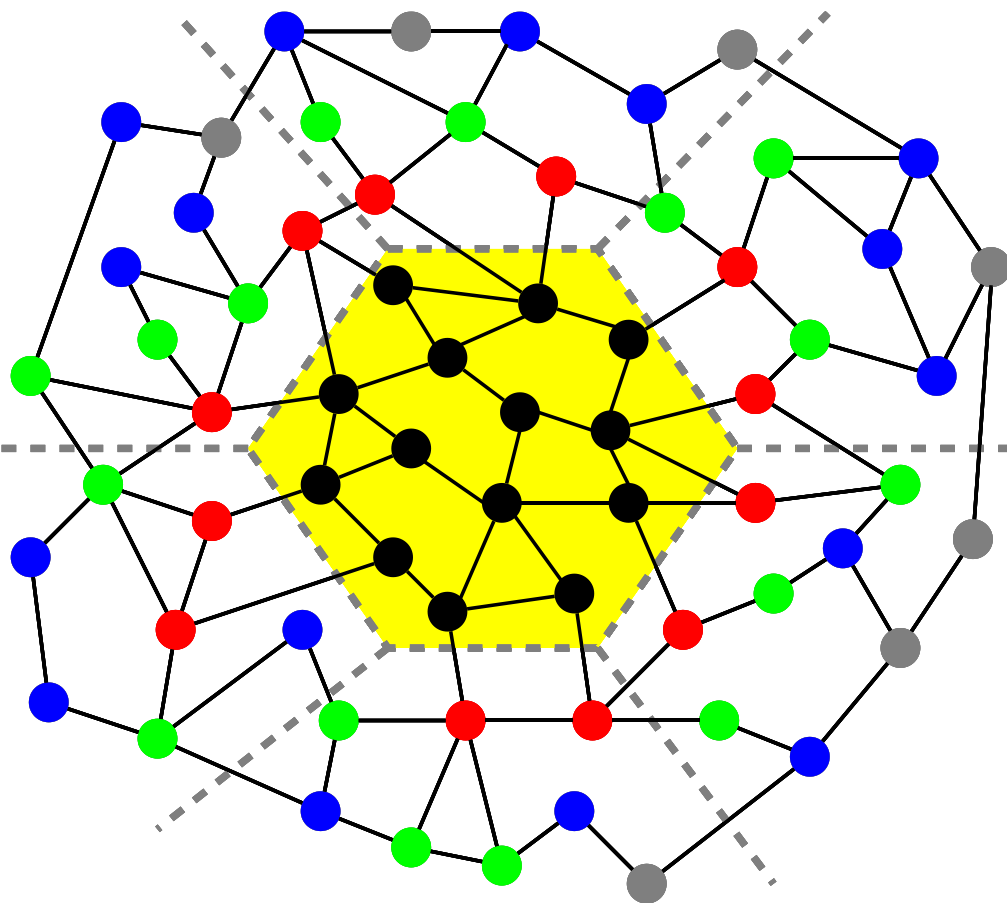


# Distributed-memory parallel Akx

Example: tridiagonal matrix,  $k = 3$ ,  $n = 40$ ,  $p = 4$



# Matrix Powers Kernel on a General Matrix



- *Need hypergraph partitioning*
- *For implicit memory management (caches) uses a TSP algorithm for layout*

See paper by Demmel, Hoemman, Mohiyuddin, Yelick, 2011

- Saves communication for “well partitioned” matrices
  - Serial memory bandwidth:  $O(1)$  moves of data vs.  $O(k)$
  - Parallel message latency:  $O(1)$  messages vs.  $O(k)$

# Example costs for model problem

- Assume 1D 3-point stencil
- $n$  points (rows/cols), partitioned evenly among  $p$  processors
  - Assume matrix is partitioned rowwise
  - Assume  $k < n/p$
- Entries in table meant in big-O sense

	Naive Akx	CA-Akx
Flops	$kn/p$	$kn/p + k^2$
Words Moved	$k$	$k$
Messages	$k$	1

# Tuning space for Akx

- **DLP optimizations:**
  - vectorization
- **ILP optimizations:**
  - Software pipelining
  - Loop unrolling
  - Eliminate branches, inline functions
- **TLP optimizations:**
  - Explicit SMT
- **Memory system optimizations:**
  - NUMA-aware affinity
  - Software prefetching
  - TLB blocking
- **Memory traffic optimizations:**
  - Streaming stores (cache bypass)
  - Array padding
  - Cache blocking
  - Index compression
  - Blocked sparse formats
  - Stanza encoding
- **Distributed memory optimizations:**
  - Topology-aware sparse collectives
  - Hypergraph partitioning
  - Dynamic load balancing
  - Overlapped communication and computation
- **Algorithmic variants:**
  - Compositions of distributed-memory parallel, shared memory parallel, sequential algorithms
  - Streaming or explicitly buffered workspace
  - Explicit or implicit cache blocks
  - Avoiding redundant computation/storage/traffic
  - Last-vector-only optimization
  - Remove low-rank components (blocking covers)
  - Different polynomial bases  $p_j(A)$
- **Other:**
  - Preprocessing optimizations
  - Extended precision arithmetic
  - Scalable data structures (sparse representations)
  - Dynamic value and/or pattern updates



# General polynomial bases for Krylov subspaces

- Given  $A$ ,  $x$ ,  $k > 0$ , compute

$$[p_1(A)x, p_2(A)x, \dots, p_k(A)x]$$

where  $p_j(A)$  is a degree- $j$  polynomial in  $A$ .

- Thus far we considered the special case of the monomials:

$$[Ax, A^2x, \dots, A^kx]$$

# Krylov subspace methods

- **Linear systems**  $Ax = b$ , eigenvalue problems, singular value problems, least squares, etc.
- Best for:  $A$  large & very sparse, stored implicitly, or only approximation needed

- **Krylov Subspace Method** is a projection process onto the Krylov subspace

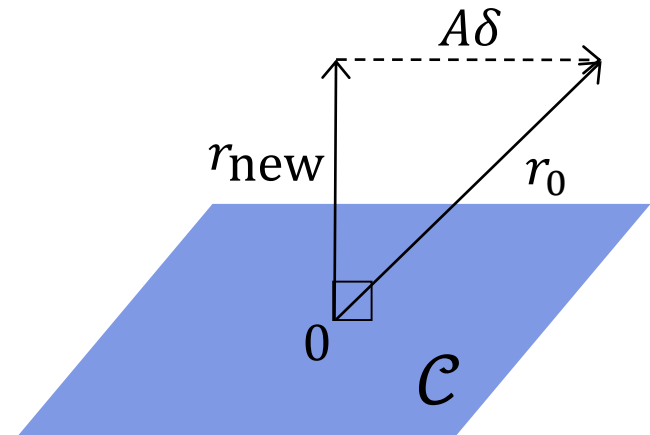
$$\mathcal{K}_i(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \dots, A^{i-1}r_0\}$$

where  $A$  is an  $N \times N$  matrix and  $r_0 = b - Ax_0$  is a length- $N$  vector

- In each iteration,
  - Add a dimension to the Krylov subspace
    - Forms nested sequence of Krylov subspaces

$$\mathcal{K}_1(A, r_0) \subset \mathcal{K}_2(A, r_0) \subset \dots \subset \mathcal{K}_i(A, r_0)$$

- Orthogonalize (with respect to some  $\mathcal{C}_i$ )
- Select approximate solution  $x_i \in x_0 + \mathcal{K}_i(A, r_0)$   
using  $r_i = b - Ax_i \perp \mathcal{C}_i$



- Ex: Lanczos/**Conjugate Gradient (CG)**, Arnoldi/Generalized Minimum Residual (GMRES), Biconjugate Gradient (BICG), BICGSTAB, GKL, LSQR, etc.

# The conjugate gradient method

$A$  is symmetric positive definite,  $\mathcal{C}_i = \mathcal{K}_i(A, r_0)$

$$r_i \perp \mathcal{K}_i(A, r_0) \iff \|x - x_i\|_A = \min_{z \in x_0 + \mathcal{K}_i(A, r_0)} \|x - z\|_A$$

$$\Rightarrow r_{N+1} = 0$$

Connection with Lanczos

- With  $v_1 = r_0 / \|r_0\|$ ,  $i$  iterations of Lanczos produces  $N \times i$  matrix  $V_i = [v_1, \dots, v_i]$ , and  $i \times i$  tridiagonal matrix  $T_i$  such that

$$AV_i = V_i T_i + \delta_{i+1} v_{i+1} e_i^T, \quad T_i = V_i^* A V_i$$

- CG approximation  $x_i$  is obtained by solving the reduced model

$$T_i y_i = \|r_0\| e_1, \quad x_i = x_0 + V_i y_i$$

- Connections with orthogonal polynomials, Stieltjes problem of moments, Gauss-Cristoffel quadrature, others (see 2013 book of Liesen and Strakoš)

$\Rightarrow$  CG (and other Krylov subspace methods) are highly nonlinear

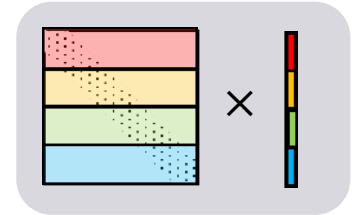
- Good for convergence, bad for ease of finite precision analysis

# Communication in CG

Projection process in terms of communication:

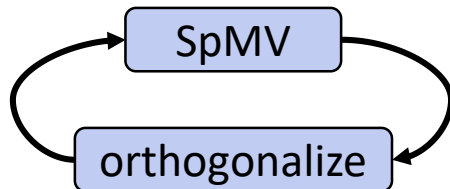
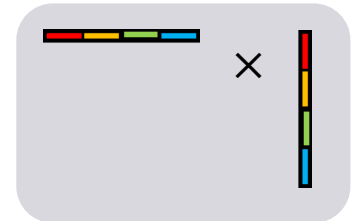
“Add a dimension to  $\mathcal{K}_i$ ”

- Sparse matrix-vector multiplication (SpMV)
  - Must communicate vector entries w/ neighboring processors (**P2P communication**)



“Orthogonalize with respect to  $\mathcal{C}_i$ ”

- Inner products
  - **global synchronization** (MPI\_Allreduce)
  - all processors must exchange data and wait for *all* communication to finish before proceeding



**Dependencies between communication-bound kernels  
in each iteration limit performance!**

# Communication in HSCG

$$r_0 = b - Ax_0, \quad p_0 = r_0$$

for  $i = 1:nmax$

$$\alpha_{i-1} = \frac{r_{i-1}^T r_{i-1}}{p_{i-1}^T A p_{i-1}}$$

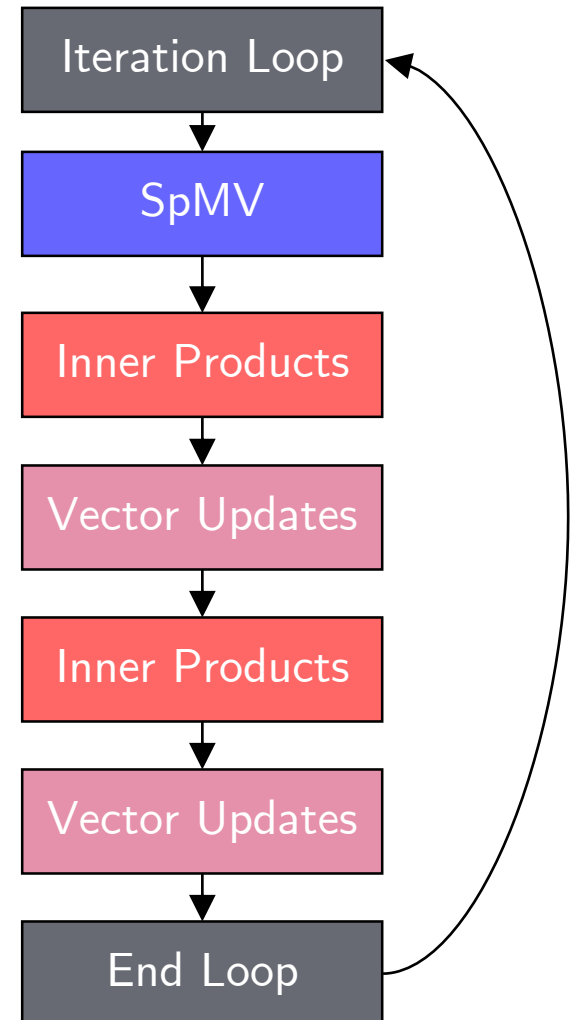
$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1} A p_{i-1}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$p_i = r_i + \beta_i p_{i-1}$$

end



# Communication in HSCG

$$r_0 = b - Ax_0, \quad p_0 = r_0$$

for  $i = 1:nmax$

$$\alpha_{i-1} = \frac{r_{i-1}^T r_{i-1}}{p_{i-1}^T A p_{i-1}}$$

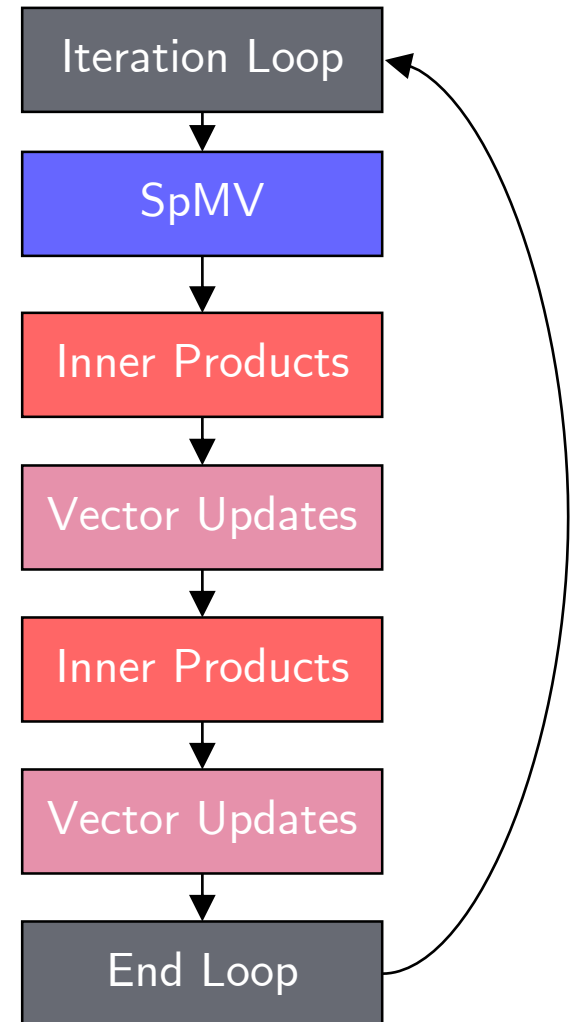
$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1} A p_{i-1}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$p_i = r_i + \beta_i p_{i-1}$$

end



# Communication in HSCG

$r_0 = b - Ax_0, \quad p_0 = r_0$   
for  $i = 1:nmax$

$$\alpha_{i-1} = \frac{r_{i-1}^T r_{i-1}}{p_{i-1}^T A p_{i-1}}$$

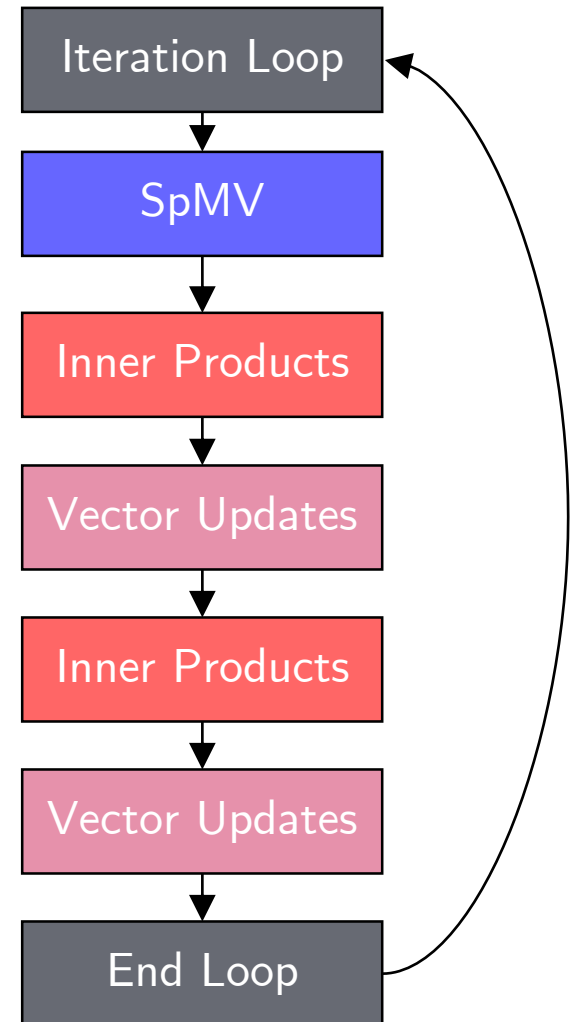
$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1} A p_{i-1}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$p_i = r_i + \beta_i p_{i-1}$$

end



# Communication in HSCG

$r_0 = b - Ax_0, \quad p_0 = r_0$   
for  $i = 1:nmax$

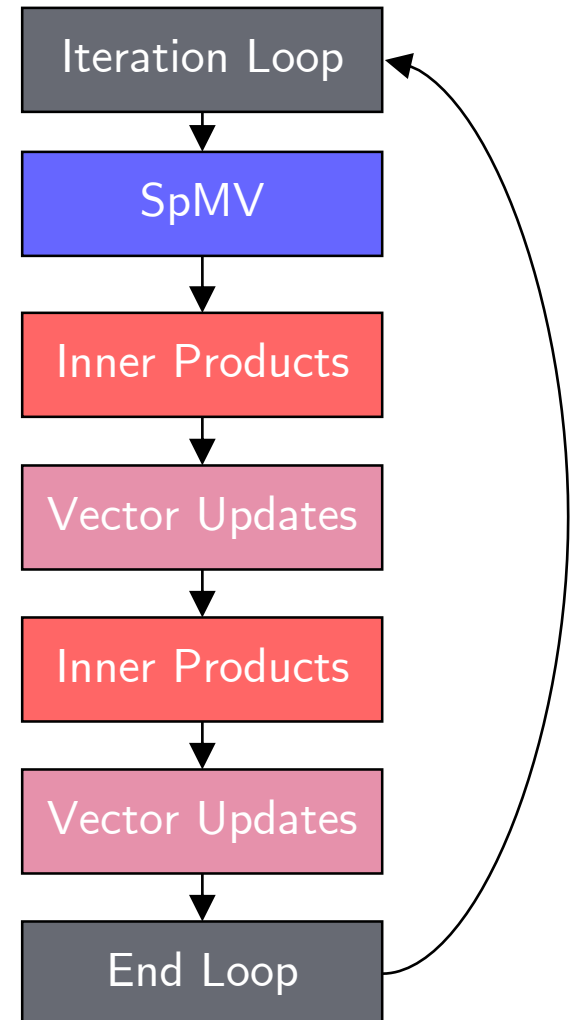
$$\alpha_{i-1} = \frac{r_{i-1}^T r_{i-1}}{p_{i-1}^T A p_{i-1}}$$

$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$
$$r_i = r_{i-1} - \alpha_{i-1} A p_{i-1}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$p_i = r_i + \beta_i p_{i-1}$$

end





# Communication in HSCG

$r_0 = b - Ax_0, \quad p_0 = r_0$   
for  $i = 1:nmax$

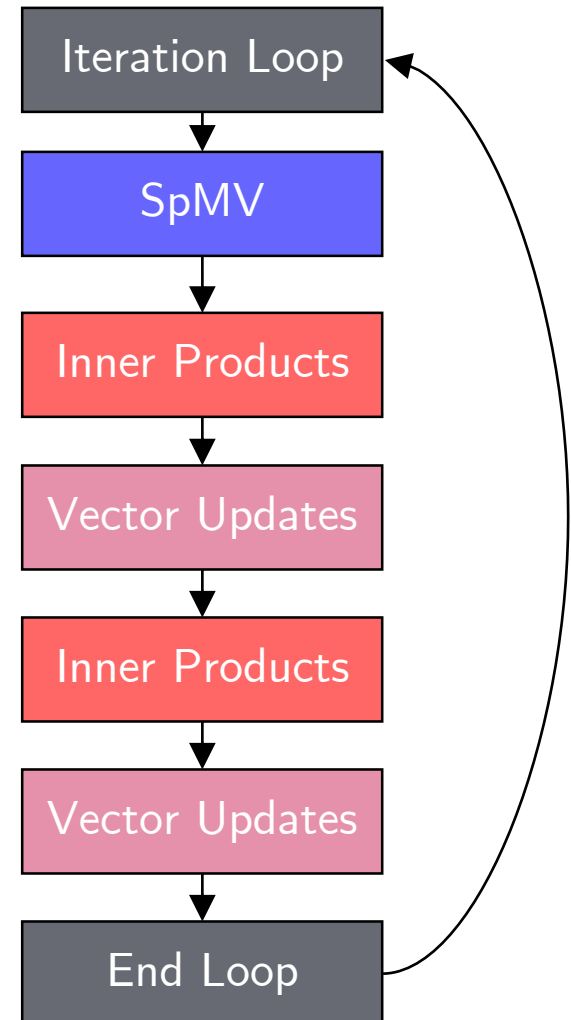
$$\alpha_{i-1} = \frac{r_{i-1}^T r_{i-1}}{p_{i-1}^T A p_{i-1}}$$

$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$
$$r_i = r_{i-1} - \alpha_{i-1} A p_{i-1}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$p_i = r_i + \beta_i p_{i-1}$$

end



# Communication in HSCG

$r_0 = b - Ax_0, \quad p_0 = r_0$   
for  $i = 1:nmax$

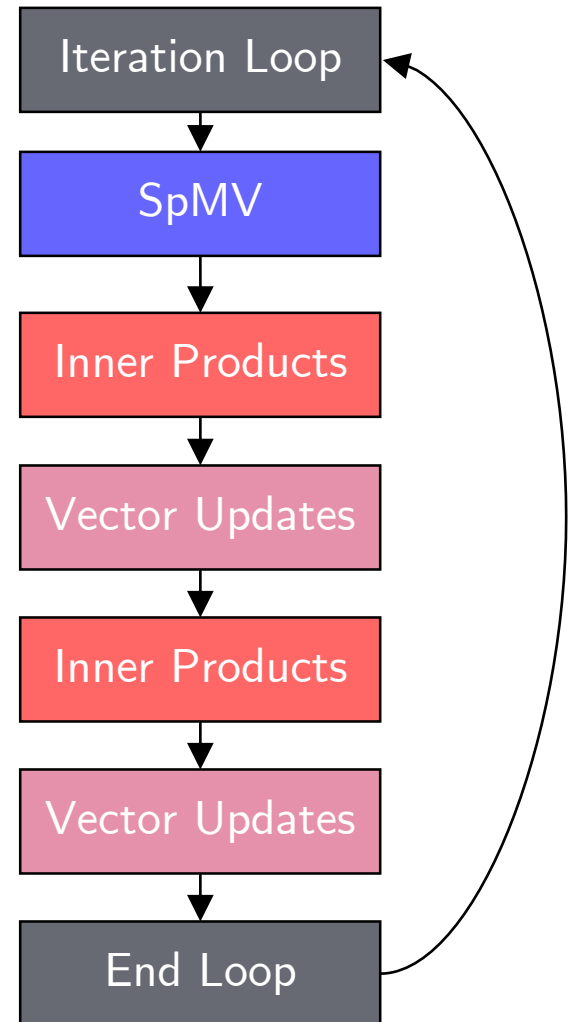
$$\alpha_{i-1} = \frac{r_{i-1}^T r_{i-1}}{p_{i-1}^T A p_{i-1}}$$

$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$
$$r_i = r_{i-1} - \alpha_{i-1} A p_{i-1}$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$p_i = r_i + \beta_i p_{i-1}$$

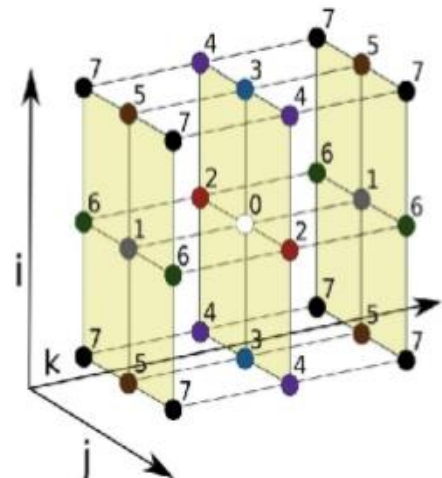
end



# HPCG Benchmark

## Model Problem Description

- Synthetic discretized 3D PDE (FEM, FVM, FDM).
- Single DOF heat diffusion model.
- Zero Dirichlet BCs, Synthetic RHS s.t. solution = 1.
- Local domain:  $(n_x \times n_y \times n_z)$
- Process layout:  $(np_x \times np_y \times np_z)$
- Global domain:  $(n_x * np_x) \times (n_y * np_y) \times (n_z * np_z)$
- Sparse matrix:
  - 27 nonzeros/row interior.
  - 7 – 18 on boundary.
  - Symmetric positive definite.



# HPCG Results (June 2022)

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	HPCG (TFlop/s)
1	2	<b>Supercomputer Fugaku</b> - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	16004.50
2	4	<b>Summit</b> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	2925.75
3	3	<b>LUMI</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	1935.73
4	7	<b>Perlmutter</b> - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	761,856	70.87	1905.44
5	5	<b>Sierra</b> - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	1795.67

➡ ~3% of peak

# Synchronization-reducing variants

Communication cost has motivated many approaches to reducing synchronization in CG:

- Early work: CG with a single synchronization point per iteration
  - 3-term recurrence CG
  - Using modified computation of recurrence coefficients
  - Using auxiliary vectors
- Pipelined Krylov subspace methods
  - Uses modified coefficients and auxiliary vectors to reduce synchronization points to 1 per iteration
  - Modifications also allow decoupling of SpMV and inner products - enables overlapping
- s-step Krylov subspace methods
  - Compute iterations in blocks of  $s$  using a different Krylov subspace basis
  - Enables one synchronization per  $s$  iterations

# CG with two three-term recurrences (STCG)

- HSCG recurrences can be written as

$$AP_i = R_{i+1}\underline{L}_i, \quad R_i = P_i U_i$$

we can combine these to obtain a 3-term recurrence for the residuals (STCG):

$$AR_i = R_{i+1}\underline{T}_i, \quad \underline{T}_i = \underline{L}_i U_i$$

- First developed by Stiefel (1952/53), also Rutishauser (1959) and Hageman and Young (1981)
- Motivated by relation to three-term recurrences for orthogonal polynomials

$r_0 = b - Ax_0, \quad p_0 = r_0, \quad x_{-1} = x_0, \quad r_{-1} = r_0, \quad e_{-1} = 0$   
for  $i = 1:nmax$

$$q_{i-1} = \frac{(r_{i-1}, Ar_{i-1})}{(r_{i-1}, r_{i-1})} - e_{i-2}$$

$$x_i = x_{i-1} + \frac{1}{q_{i-1}} (r_{i-1} + e_{i-2}(x_{i-1} - x_{i-2}))$$

$$r_i = r_{i-1} + \frac{1}{q_{i-1}} (-Ar_{i-1} + e_{i-2}(r_{i-1} - r_{i-2}))$$

$$e_{i-1} = q_{i-1} \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

end

Can be accomplished with a single synchronization point on parallel computers (Strakoš 1985, 1987)

- Similar approach (computing  $\alpha_i$  using  $\beta_{i-1}$ ) used by D'Azevedo, Eijkhout, Romaine (1992, 1993)

# Chronopoulos and Gear's CG (ChG CG)

- Chronopoulos and Gear (1989)
- Looks like HSCG, but very similar to 3-term recurrence CG (STCG)
- Reduces synchronizations/iteration to 1 by changing computation of  $\alpha_i$  and using an auxiliary recurrence for  $Ap_i$

$$r_0 = b - Ax_0, \quad p_0 = r_0,$$

$$s_0 = Ap_0, \quad \alpha_0 = (r_0, r_0) / (p_0, s_0)$$

for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = Ar_i$$

$$\beta_i = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

$$\alpha_i = \frac{(r_i, r_i)}{(w_i, r_i) - (\beta_i / \alpha_{i-1})(r_i, r_i)}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

end

# Chronopoulos and Gear's CG (ChG CG)

- Chronopoulos and Gear (1989)
- Looks like HSCG, but very similar to 3-term recurrence CG (STCG)
- Reduces synchronizations/iteration to 1 by changing computation of  $\alpha_i$  and using an auxiliary recurrence for  $Ap_i$

```

$$r_0 = b - Ax_0, \quad p_0 = r_0,$$

$$s_0 = Ap_0, \quad \alpha_0 = (r_0, r_0) / (p_0, s_0)$$

$$\text{for } i = 1:n_{\max}$$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = Ar_i$$

$$\beta_i = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

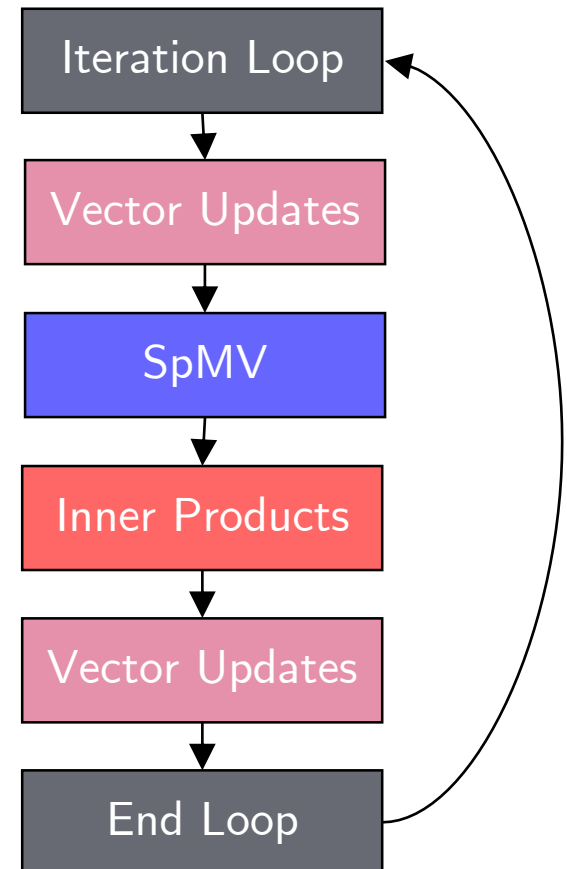
$$\alpha_i = \frac{(r_i, r_i)}{(w_i, r_i) - (\beta_i / \alpha_{i-1})(r_i, r_i)}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

$$\text{end}$$

```





# Chronopoulos and Gear's CG (ChG CG)

- Chronopoulos and Gear (1989)
- Looks like HSCG, but very similar to 3-term recurrence CG (STCG)
- Reduces synchronizations/iteration to 1 by changing computation of  $\alpha_i$  and using an auxiliary recurrence for  $Ap_i$

$r_0 = b - Ax_0, p_0 = r_0,$   
 $s_0 = Ap_0, \alpha_0 = (r_0, r_0)/(p_0, s_0)$   
for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = Ar_i$$

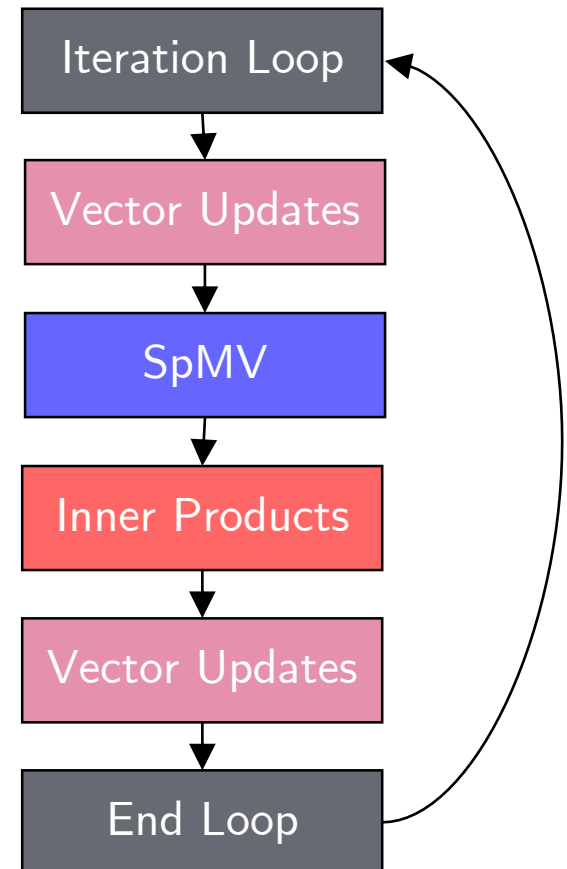
$$\beta_i = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

$$\alpha_i = \frac{(r_i, r_i)}{(w_i, r_i) - (\beta_i/\alpha_{i-1})(r_i, r_i)}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

end



# Chronopoulos and Gear's CG (ChG CG)

- Chronopoulos and Gear (1989)
- Looks like HSCG, but very similar to 3-term recurrence CG (STCG)
- Reduces synchronizations/iteration to 1 by changing computation of  $\alpha_i$  and using an auxiliary recurrence for  $Ap_i$

$r_0 = b - Ax_0, p_0 = r_0,$   
 $s_0 = Ap_0, \alpha_0 = (r_0, r_0)/(p_0, s_0)$   
for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = Ar_i$$

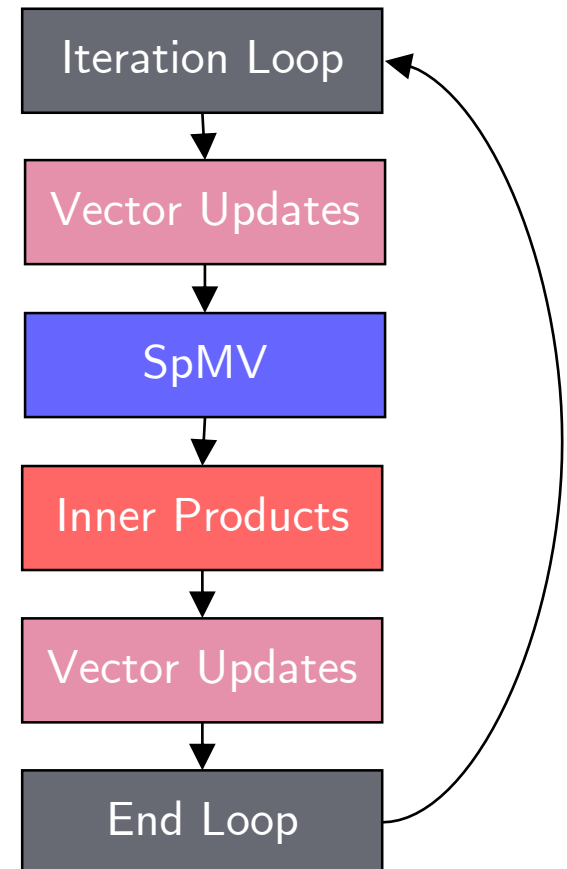
$$\beta_i = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

$$\alpha_i = \frac{(r_i, r_i)}{(w_i, r_i) - (\beta_i/\alpha_{i-1})(r_i, r_i)}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

end



# Chronopoulos and Gear's CG (ChG CG)

- Chronopoulos and Gear (1989)
- Looks like HSCG, but very similar to 3-term recurrence CG (STCG)
- Reduces synchronizations/iteration to 1 by changing computation of  $\alpha_i$  and using an auxiliary recurrence for  $Ap_i$

$r_0 = b - Ax_0, p_0 = r_0,$   
 $s_0 = Ap_0, \alpha_0 = (r_0, r_0)/(p_0, s_0)$   
for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = Ar_i$$

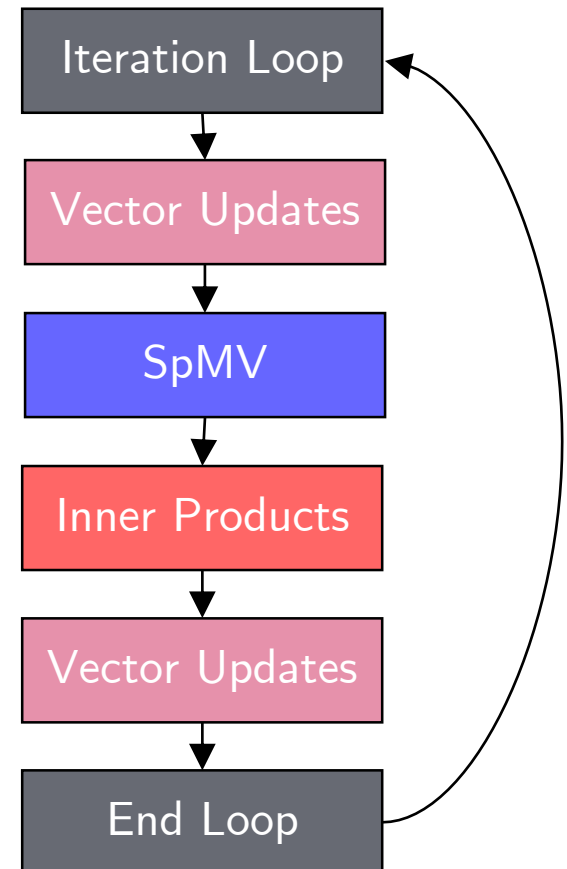
$$\beta_i = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

$$\alpha_i = \frac{(r_i, r_i)}{(w_i, r_i) - (\beta_i/\alpha_{i-1})(r_i, r_i)}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

end



# Chronopoulos and Gear's CG (ChG CG)

- Chronopoulos and Gear (1989)
- Looks like HSCG, but very similar to 3-term recurrence CG (STCG)
- Reduces synchronizations/iteration to 1 by changing computation of  $\alpha_i$  and using an auxiliary recurrence for  $Ap_i$

$r_0 = b - Ax_0, p_0 = r_0,$   
 $s_0 = Ap_0, \alpha_0 = (r_0, r_0)/(p_0, s_0)$   
for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = Ar_i$$

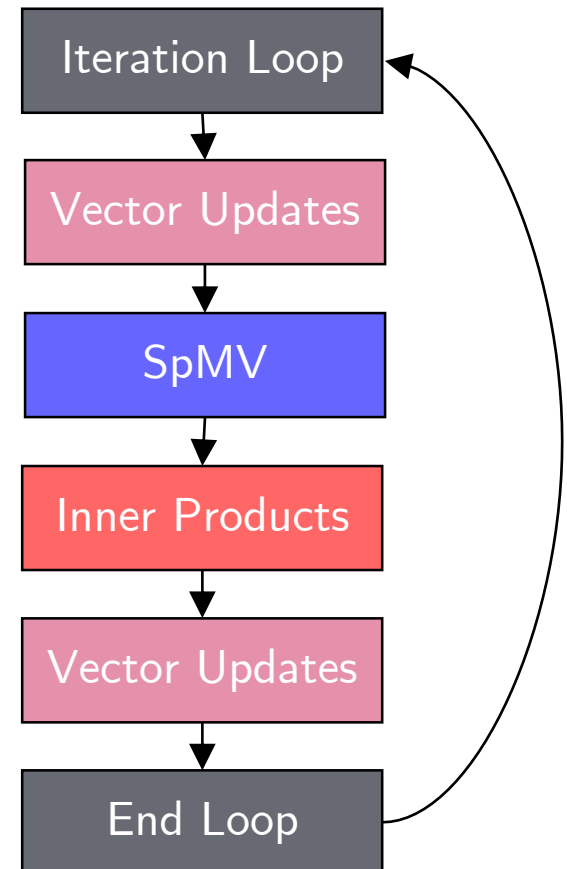
$$\beta_i = \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}$$

$$\alpha_i = \frac{(r_i, r_i)}{(w_i, r_i) - (\beta_i/\alpha_{i-1})(r_i, r_i)}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

end



# Pipelined CG (GVCG)

- Pipelined CG of Ghysels and Vanroose (2014)
- Similar to Chronopoulos and Gear approach
  - Uses auxiliary vector  $s_i \equiv Ap_i$  and same formula for  $\alpha_i$
- Also uses auxiliary vectors for  $Ar_i$  and  $A^2r_i$  to remove sequential dependency between SpMV and inner products
  - Allows the use of nonblocking (asynchronous) MPI communication to *overlap* SpMV and inner products
  - Hides the latency of global communications

# GVCG (Ghysels and Vanroose 2014)

$$r_0 = b - Ax_0, p_0 = r_0$$

$$s_0 = Ap_0, w_0 = Ar_0, z_0 = Aw_0,$$

$$\alpha_0 = r_0^T r_0 / p_0^T s_0$$

for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1} s_{i-1}$$

$$w_i = w_{i-1} - \alpha_{i-1} z_{i-1}$$

$$q_i = Aw_i$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

$$\alpha_i = \frac{r_i^T r_i}{w_i^T r_i - (\beta_i / \alpha_{i-1}) r_i^T r_i}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

$$z_i = q_i + \beta_i z_{i-1}$$

end

# GVCG (Ghysels and Vanroose 2014)

$$r_0 = b - Ax_0, p_0 = r_0$$

$$s_0 = Ap_0, w_0 = Ar_0, z_0 = Aw_0,$$

$$\alpha_0 = r_0^T r_0 / p_0^T s_0$$

for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = w_{i-1} - \alpha_{i-1}z_{i-1}$$

$$q_i = Aw_i$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

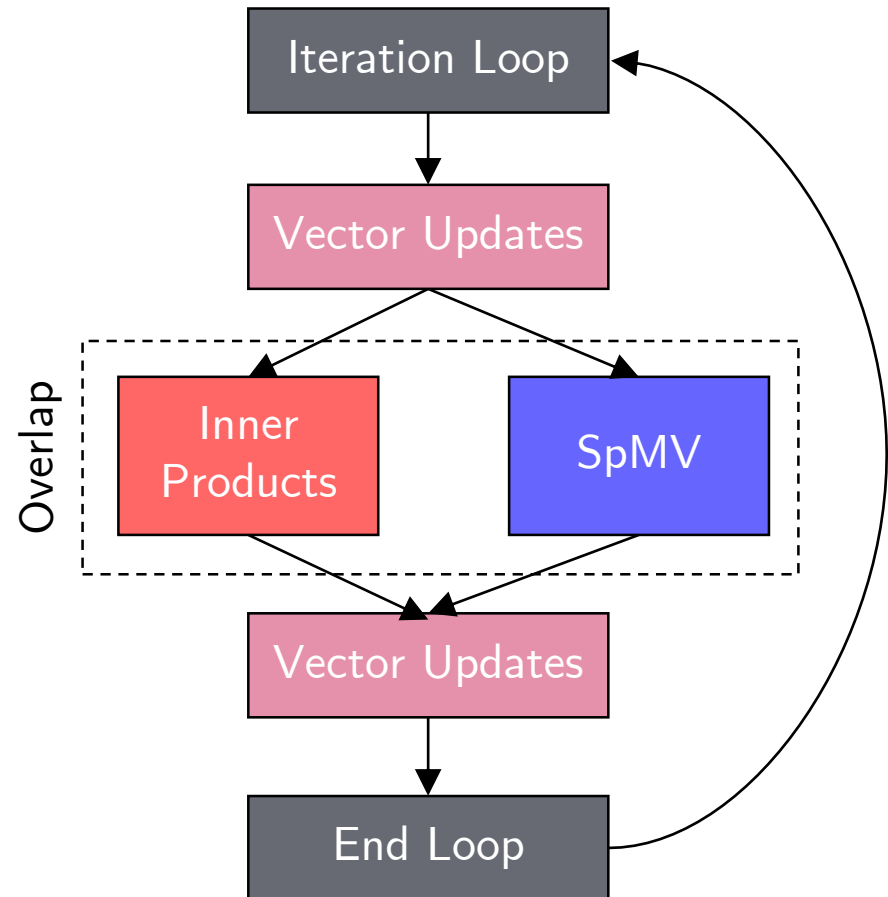
$$\alpha_i = \frac{r_i^T r_i}{w_i^T r_i - (\beta_i / \alpha_{i-1}) r_i^T r_i}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

$$z_i = q_i + \beta_i z_{i-1}$$

end



# GVCG (Ghysels and Vanroose 2014)

$$r_0 = b - Ax_0, p_0 = r_0$$

$$s_0 = Ap_0, w_0 = Ar_0, z_0 = Aw_0,$$

$$\alpha_0 = r_0^T r_0 / p_0^T s_0$$

for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1}s_{i-1}$$

$$w_i = w_{i-1} - \alpha_{i-1}z_{i-1}$$

$$q_i = Aw_i$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

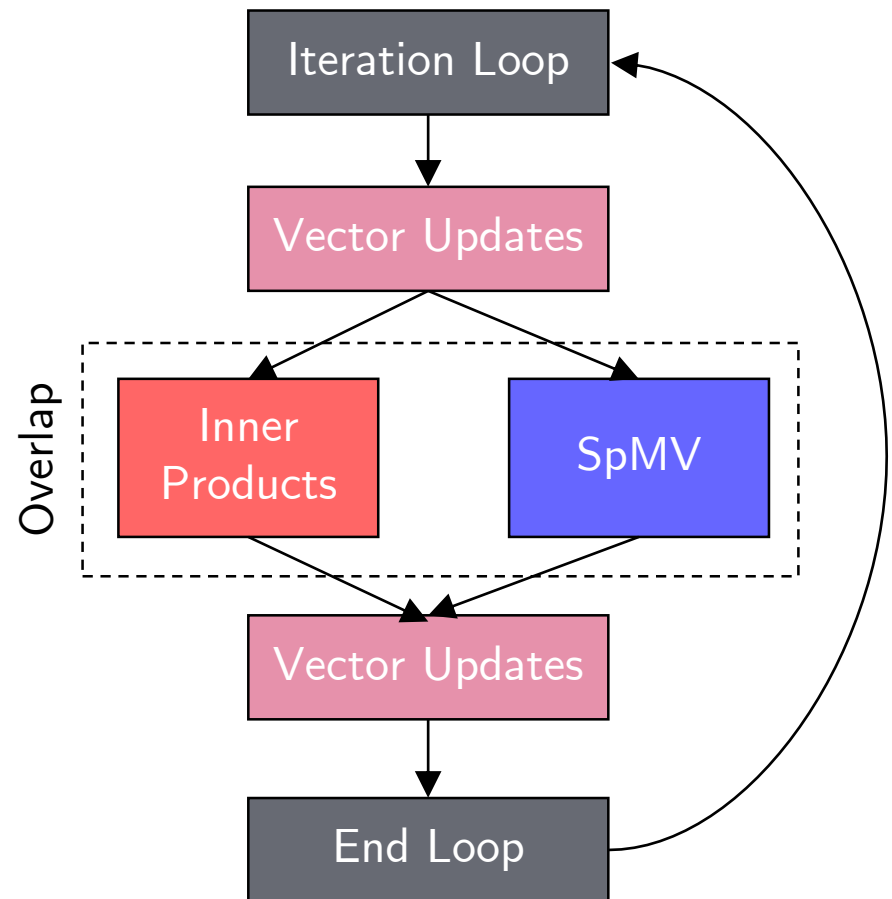
$$\alpha_i = \frac{r_i^T r_i}{w_i^T r_i - (\beta_i / \alpha_{i-1}) r_i^T r_i}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

$$z_i = q_i + \beta_i z_{i-1}$$

end





# GVCG (Ghysels and Vanroose 2014)

$$r_0 = b - Ax_0, p_0 = r_0$$

$$s_0 = Ap_0, w_0 = Ar_0, z_0 = Aw_0,$$

$$\alpha_0 = r_0^T r_0 / p_0^T s_0$$

for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1} s_{i-1}$$

$$w_i = w_{i-1} - \alpha_{i-1} z_{i-1}$$

$$q_i = Aw_i$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

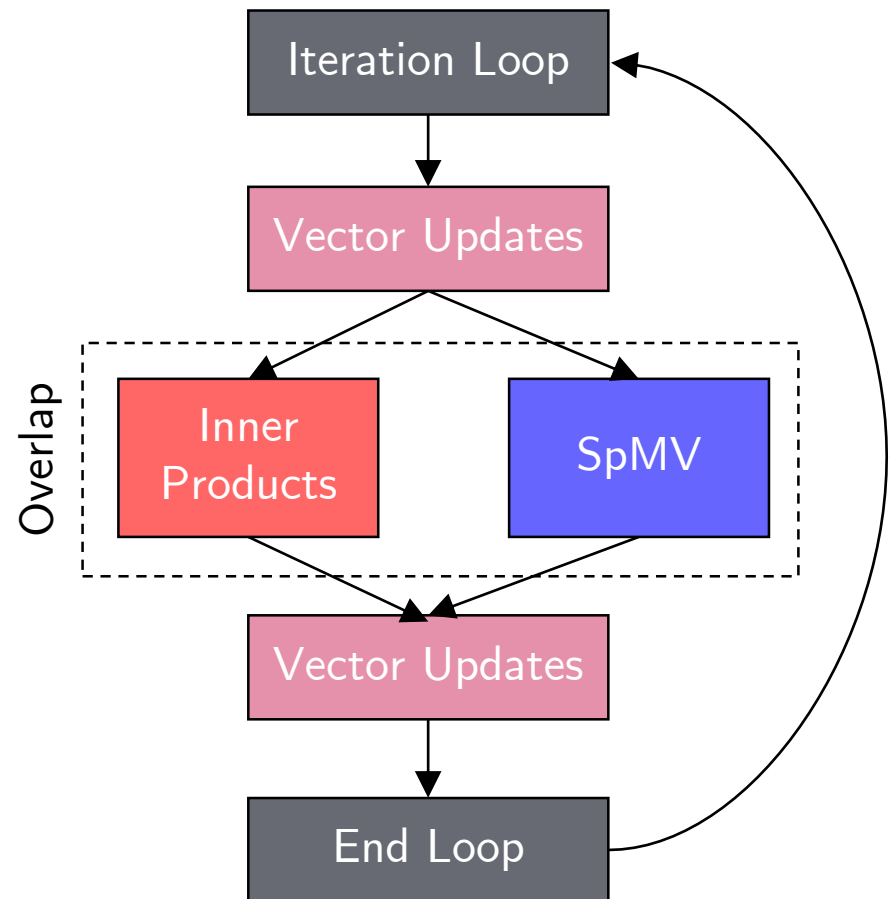
$$\alpha_i = \frac{r_i^T r_i}{w_i^T r_i - (\beta_i / \alpha_{i-1}) r_i^T r_i}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

$$z_i = q_i + \beta_i z_{i-1}$$

end



# GVCG (Ghysels and Vanroose 2014)

$$r_0 = b - Ax_0, p_0 = r_0$$

$$s_0 = Ap_0, w_0 = Ar_0, z_0 = Aw_0,$$

$$\alpha_0 = r_0^T r_0 / p_0^T s_0$$

for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1} s_{i-1}$$

$$w_i = w_{i-1} - \alpha_{i-1} z_{i-1}$$

$$q_i = Aw_i$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

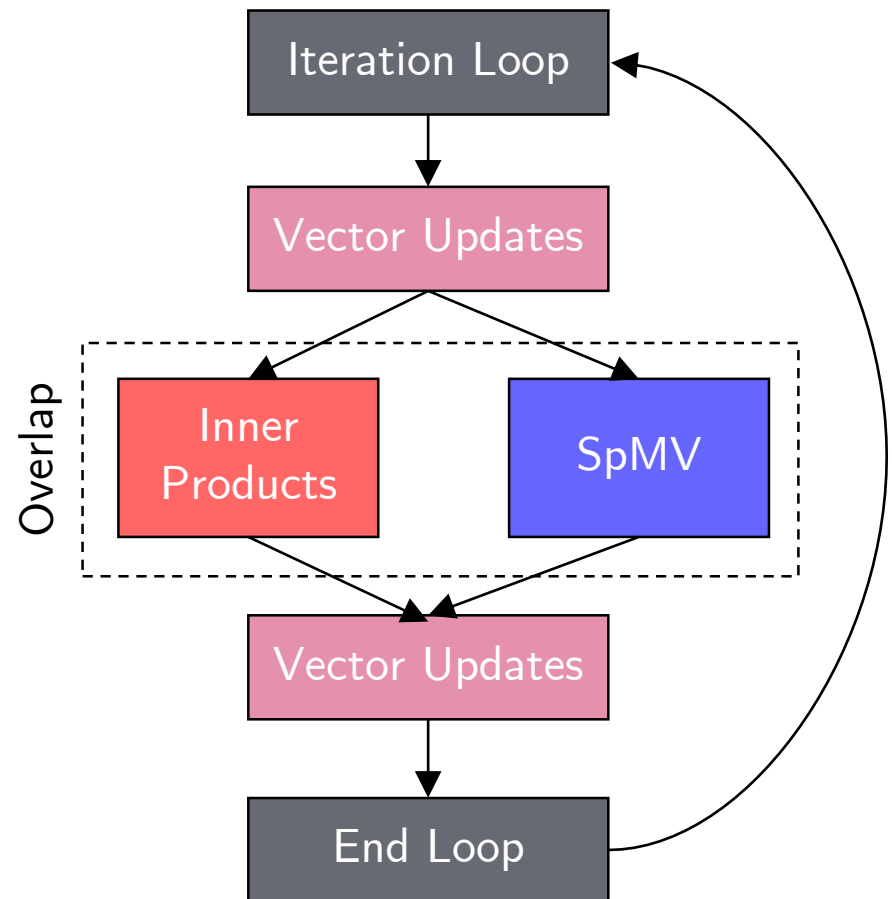
$$\alpha_i = \frac{r_i^T r_i}{w_i^T r_i - (\beta_i / \alpha_{i-1}) r_i^T r_i}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

$$z_i = q_i + \beta_i z_{i-1}$$

end



# GVCG (Ghysels and Vanroose 2014)

$$r_0 = b - Ax_0, p_0 = r_0$$

$$s_0 = Ap_0, w_0 = Ar_0, z_0 = Aw_0,$$

$$\alpha_0 = r_0^T r_0 / p_0^T s_0$$

for  $i = 1:nmax$

$$x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$$

$$r_i = r_{i-1} - \alpha_{i-1} s_{i-1}$$

$$w_i = w_{i-1} - \alpha_{i-1} z_{i-1}$$

$$q_i = Aw_i$$

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}$$

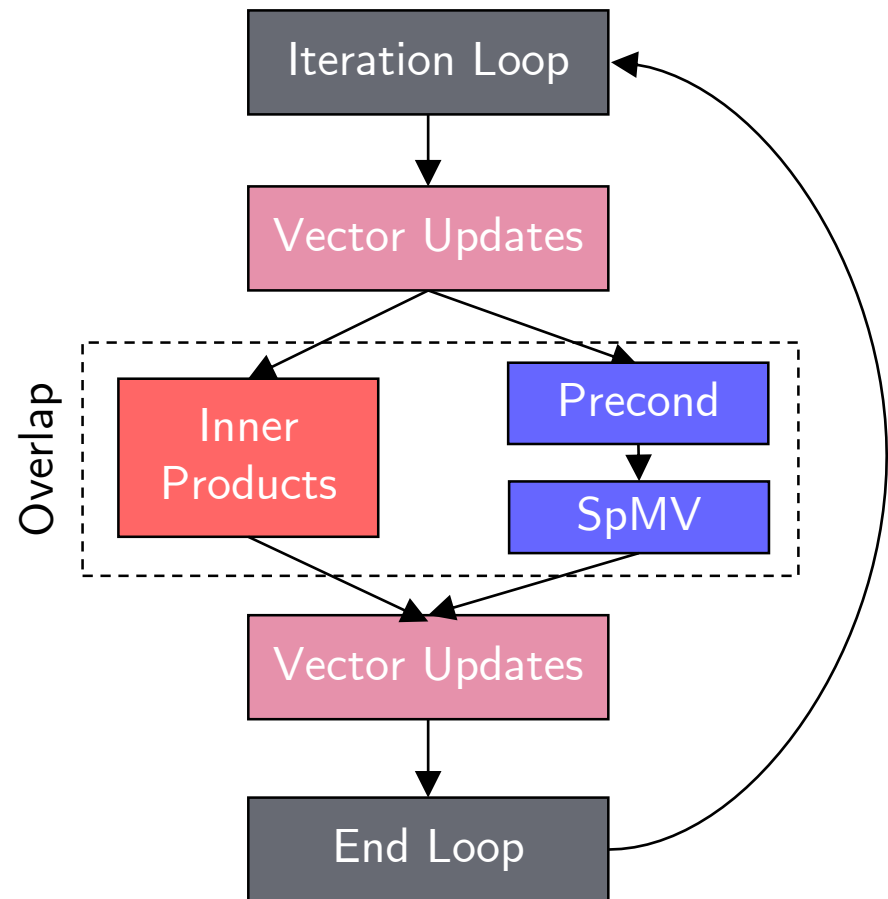
$$\alpha_i = \frac{r_i^T r_i}{w_i^T r_i - (\beta_i / \alpha_{i-1}) r_i^T r_i}$$

$$p_i = r_i + \beta_i p_{i-1}$$

$$s_i = w_i + \beta_i s_{i-1}$$

$$z_i = q_i + \beta_i z_{i-1}$$

end



# s-step CG

- Idea: Compute blocks of  $s$  iterations at once
  - Compute updates in a different basis
  - Communicate every  $s$  iterations instead of every iteration
  - Reduces number of synchronizations per iteration by a factor of  $s$
- An idea rediscovered many times...
- First related work:  $s$ -dimensional steepest descent, least squares
  - Khabaza ('63), Forsythe ('68), Marchuk and Kuznecov ('68)
- Flurry of work on  $s$ -step Krylov methods in '80s/early '90s: see, e.g., Van Rosendale (1983); Chronopoulos and Gear (1989)
- Resurgence of interest in recent years due to growing problem sizes; growing relative cost of communication

# s-step CG

Key observation: After iteration  $i$ , for  $j \in \{0, \dots, s\}$ ,

$$x_{i+j} - x_i, \quad r_{i+j}, \quad p_{i+j} \in \mathcal{K}_{s+1}(A, p_i) + \mathcal{K}_s(A, r_i)$$

## s steps of s-step CG:

**Expand solution space  $s$  dimensions at once**

Compute “basis” matrix  $\mathcal{Y}$  such that  $\text{span}(\mathcal{Y}) = \mathcal{K}_{s+1}(A, p_i) + \mathcal{K}_s(A, r_i)$  according to the recurrence  $\underline{A\mathcal{Y}} = \mathcal{Y}\mathcal{B}$

**Compute inner products between basis vectors in one synchronization**

$$\mathcal{G} = \mathcal{Y}^T \mathcal{Y}$$

**Compute  $s$  iterations of vector updates**

Perform  $s$  iterations of vector updates by updating coordinates in basis  $\mathcal{Y}$ :

$$x_{i+j} - x_i = \mathcal{Y}x'_j, \quad r_{i+j} = \mathcal{Y}r'_j, \quad p_{i+j} = \mathcal{Y}p'_j$$

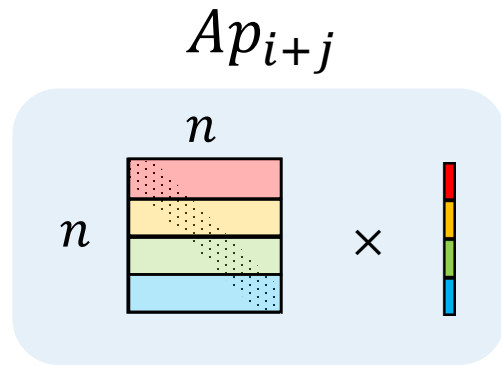
# s-step CG

---

For  $s$  iterations of updates, inner products and SpMV's (in basis  $\mathcal{Y}$ ) can be computed independently by each processor without communication:

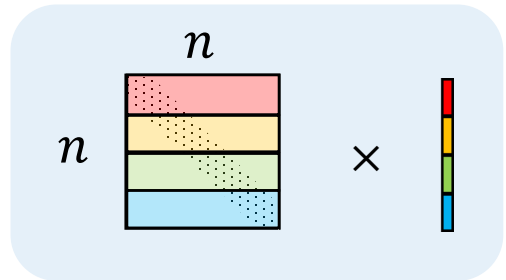
# s-step CG

For  $s$  iterations of updates, inner products and SpMV (in basis  $\mathcal{Y}$ ) can be computed independently by each processor without communication:



# s-step CG

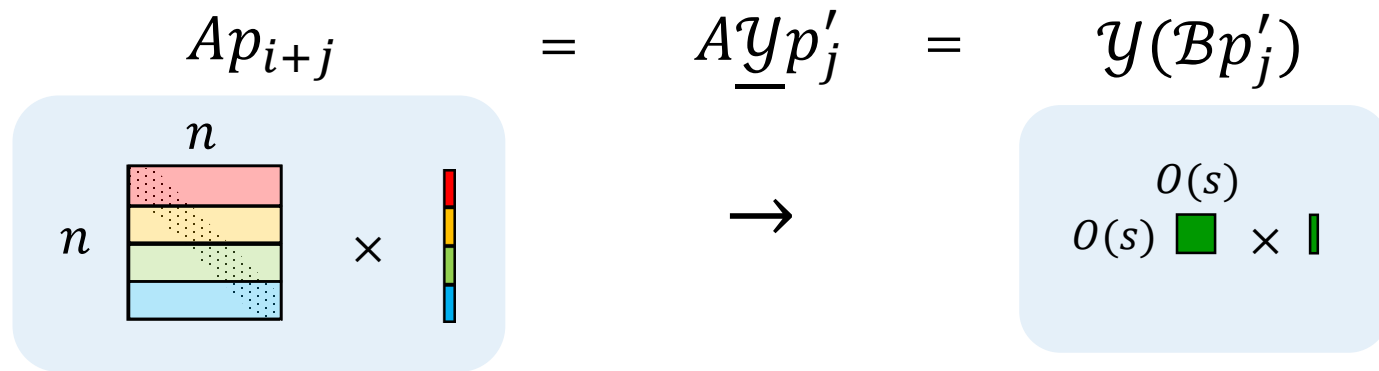
For  $s$  iterations of updates, inner products and SpMV (in basis  $\mathcal{Y}$ ) can be computed independently by each processor without communication:

$$Ap_{i+j} = \underline{A} \mathcal{Y} p'_j$$




# s-step CG

For  $s$  iterations of updates, inner products and SpMV (in basis  $\mathcal{Y}$ ) can be computed independently by each processor without communication:

$$Ap_{i+j} = A\underline{\mathcal{Y}}p'_j = \mathcal{Y}(\mathcal{B}p'_j)$$


The diagram illustrates the computation of  $Ap_{i+j}$  as a product of a matrix  $A$  and a vector  $p'_{i+j}$ . The matrix  $A$  is shown as a block of size  $n$  by  $n$ , divided into four horizontal bands of different colors (red, yellow, green, blue) with a diagonal band of dots. The vector  $p'_{i+j}$  is shown as a vertical bar of size  $n$ , also divided into four colored segments. The result is shown as a product of a matrix of size  $O(s)$  by  $O(s)$  and a vector of size  $O(s)$ . The matrix is shown as a green square and the vector as a green bar.

# s-step CG

For  $s$  iterations of updates, inner products and SpMV (in basis  $\mathcal{Y}$ ) can be computed independently by each processor without communication:

$$\begin{array}{c}
 \begin{array}{c}
 \text{\textit{A}p}_{i+j} \\
 \begin{array}{c} n \\
 \begin{array}{c} n \end{array}
 \end{array}
 \end{array}
 \times
 \begin{array}{c}
 \begin{array}{c} \text{\textit{r}}_{i+j} \end{array}
 \end{array}
 =
 \begin{array}{c}
 \text{\textit{A}}\underline{\text{\textit{Y}}}p'_j \\
 \rightarrow
 \end{array}
 =
 \begin{array}{c}
 \text{\textit{Y}}(\text{\textit{B}}p'_j) \\
 \begin{array}{c} O(s) \\
 O(s) \end{array}
 \end{array}
 \times
 \begin{array}{c}
 \begin{array}{c} \text{\textit{r}}_{i+j} \end{array}
 \end{array}
 \end{array}$$

Diagram illustrating the computation of  $\text{\textit{A}p}_{i+j}$  and  $(\text{\textit{r}}_{i+j}, \text{\textit{r}}_{i+j})$  using the basis  $\mathcal{Y}$ .

The first part shows the matrix-vector product  $\text{\textit{A}p}_{i+j}$  (represented by a block matrix of size  $n \times n$  with colored bands) multiplied by a vector  $\text{\textit{r}}_{i+j}$  (represented by a vertical bar with colored segments).

The second part shows the inner product  $(\text{\textit{r}}_{i+j}, \text{\textit{r}}_{i+j})$  (represented by a horizontal bar with colored segments) multiplied by a vector  $\text{\textit{r}}_{i+j}$  (represented by a vertical bar with colored segments).

The third part shows the transformation  $\text{\textit{A}}\underline{\text{\textit{Y}}}p'_j = \text{\textit{Y}}(\text{\textit{B}}p'_j)$ , where  $\text{\textit{Y}}$  is a matrix of size  $O(s) \times O(s)$  (represented by a green square) and  $p'_j$  is a vector (represented by a vertical bar).

# s-step CG

For  $s$  iterations of updates, inner products and SpMV (in basis  $\mathcal{Y}$ ) can be computed independently by each processor without communication:

$$\begin{aligned}
 & \begin{array}{c} A p_{i+j} \\ \begin{array}{|c|} \hline n \\ \hline \end{array} \\ \begin{array}{|c|} \hline n \\ \hline \end{array} \end{array} \times \begin{array}{|c|} \hline \text{red} \\ \hline \text{yellow} \\ \hline \text{green} \\ \hline \text{blue} \\ \hline \end{array} \rightarrow \begin{array}{c} \mathcal{Y} p'_j \\ \mathcal{Y}(\mathcal{B} p'_j) \\ \begin{array}{|c|} \hline O(s) \\ \hline \end{array} \end{array} \times \begin{array}{|c|} \hline \text{green} \\ \hline \end{array}
 \end{aligned}$$
  

$$\begin{aligned}
 & (r_{i+j}, r_{i+j}) = r_j'^T \mathcal{Y}^T \mathcal{Y} r_j' \\
 & \begin{array}{|c|} \hline \text{red} \\ \hline \text{yellow} \\ \hline \text{green} \\ \hline \text{blue} \\ \hline \end{array} \times \begin{array}{|c|} \hline \text{red} \\ \hline \text{yellow} \\ \hline \text{green} \\ \hline \text{blue} \\ \hline \end{array}
 \end{aligned}$$

# s-step CG

For  $s$  iterations of updates, inner products and SpMV (in basis  $\mathcal{Y}$ ) can be computed independently by each processor without communication:

$$\begin{array}{ccccc}
 Ap_{i+j} & = & A\underline{\mathcal{Y}}p'_j & = & \mathcal{Y}(\mathcal{B}p'_j) \\
 \begin{array}{c} n \\ n \end{array} \begin{array}{|c|} \hline \text{red dotted} \\ \hline \text{yellow dotted} \\ \hline \text{green dotted} \\ \hline \text{blue dotted} \\ \hline \end{array} \times \begin{array}{|c|} \hline \text{red} \\ \hline \text{yellow} \\ \hline \text{green} \\ \hline \text{blue} \\ \hline \end{array} & \rightarrow & \begin{array}{c} O(s) \\ O(s) \end{array} \begin{array}{|c|} \hline \text{green} \\ \hline \end{array} \times \begin{array}{|c|} \hline \text{red} \\ \hline \end{array} \\
 \\
 (r_{i+j}, r_{i+j}) & = & r_j'^T \mathcal{Y}^T \mathcal{Y} r_j' & = & r_j'^T \mathcal{G} r_j' \\
 \begin{array}{|c|} \hline \text{red} \\ \hline \text{yellow} \\ \hline \text{green} \\ \hline \text{blue} \\ \hline \end{array} \times \begin{array}{|c|} \hline \text{red} \\ \hline \text{yellow} \\ \hline \text{green} \\ \hline \text{blue} \\ \hline \end{array} & \rightarrow & \text{red} \times \begin{array}{|c|} \hline \text{green} \\ \hline \end{array} \times \begin{array}{|c|} \hline \text{red} \\ \hline \end{array}
 \end{array}$$

# s-step CG

$$r_0 = b - Ax_0, p_0 = r_0$$

for  $k = 0:nmax/s$

Compute  $\underline{y}_k$  and  $\mathcal{B}_k$  such that  $A\underline{y}_k = y_k \mathcal{B}_k$  and

$$\text{span}(\underline{y}_k) = \mathcal{K}_{s+1}(A, p_{sk}) + \mathcal{K}_s(A, r_{sk})$$

$$\mathcal{G}_k = \underline{y}_k^T \underline{y}_k$$

$$x'_0 = 0, r'_0 = e_{s+2}, p'_0 = e_1$$

for  $j = 1:s$

$$\alpha_{sk+j-1} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}{p'_{j-1}{}^T \mathcal{G}_k \mathcal{B}_k p'_{j-1}}$$

$$x'_j = x'_{j-1} + \alpha_{sk+j-1} p'_{j-1}$$

$$r'_j = r'_{j-1} - \alpha_{sk+j-1} \mathcal{B}_k p'_{j-1}$$

$$\beta_{sk+j} = \frac{r'_j{}^T \mathcal{G}_k r'_j}{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}$$

$$p'_j = r'_j + \beta_{sk+j} p'_{j-1}$$

end

$$[x_{s(k+1)} - x_{sk}, r_{s(k+1)}, p_{s(k+1)}] = \underline{y}_k[x'_s, r'_s, p'_s]$$

end

# s-step CG

$$r_0 = b - Ax_0, p_0 = r_0$$

for  $k = 0:nmax/s$

Compute  $\mathcal{Y}_k$  and  $\mathcal{B}_k$  such that  $A\mathcal{Y}_k = \mathcal{Y}_k\mathcal{B}_k$  and

$$\text{span}(\mathcal{Y}_k) = \mathcal{K}_{s+1}(A, p_{sk}) + \mathcal{K}_s(A, r_{sk})$$

$$\mathcal{G}_k = \mathcal{Y}_k^T \mathcal{Y}_k$$

$$x'_0 = 0, r'_0 = e_{s+2}, p'_0 = e_1$$

for  $j = 1:s$

$$\alpha_{sk+j-1} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}{p'_{j-1}{}^T \mathcal{G}_k \mathcal{B}_k p'_{j-1}}$$

$$x'_j = x'_{j-1} + \alpha_{sk+j-1} p'_{j-1}$$

$$r'_j = r'_{j-1} - \alpha_{sk+j-1} \mathcal{B}_k p'_{j-1}$$

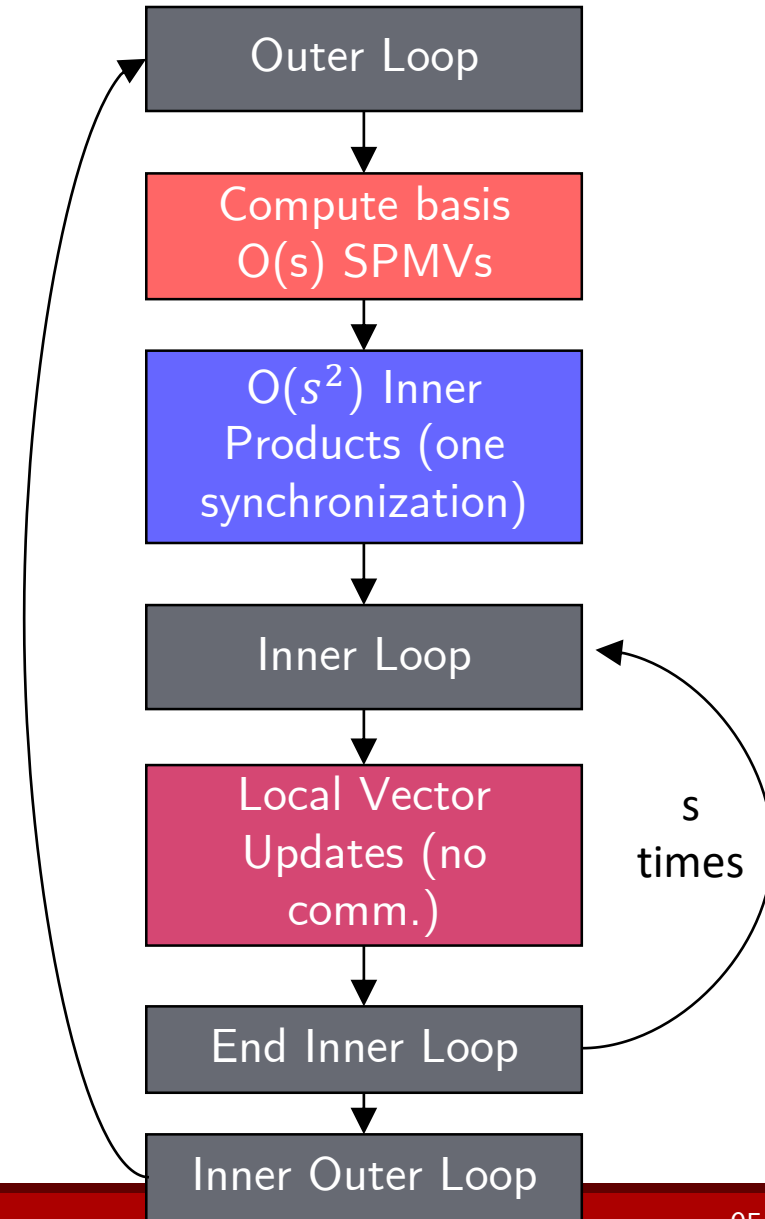
$$\beta_{sk+j} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_j}{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}$$

$$p'_j = r'_j + \beta_{sk+j} p'_{j-1}$$

end

$$[x_{s(k+1)} - x_{sk}, r_{s(k+1)}, p_{s(k+1)}] = \mathcal{Y}_k[x'_s, r'_s, p'_s]$$

end



# s-step CG

$r_0 = b - Ax_0, p_0 = r_0$

for  $k = 0:nmax/s$

Compute  $\underline{y}_k$  and  $\mathcal{B}_k$  such that  $A\underline{y}_k = \underline{y}_k \mathcal{B}_k$  and  
 $\text{span}(\underline{y}_k) = \mathcal{K}_{s+1}(A, p_{sk}) + \mathcal{K}_s(A, r_{sk})$

$\mathcal{G}_k = \underline{y}_k^T \underline{y}_k$

$x'_0 = 0, r'_0 = e_{s+2}, p'_0 = e_1$

for  $j = 1:s$

$$\alpha_{sk+j-1} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}{p'_{j-1}{}^T \mathcal{G}_k \mathcal{B}_k p'_{j-1}}$$

$$x'_j = x'_{j-1} + \alpha_{sk+j-1} p'_{j-1}$$

$$r'_j = r'_{j-1} - \alpha_{sk+j-1} \mathcal{B}_k p'_{j-1}$$

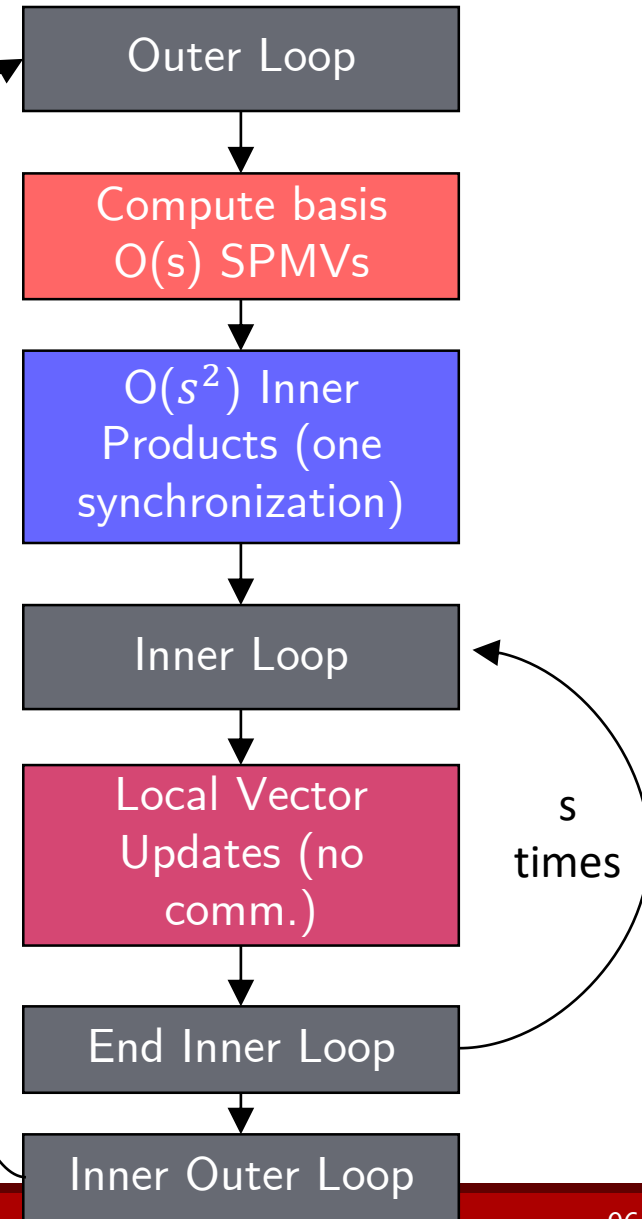
$$\beta_{sk+j} = \frac{r'_j{}^T \mathcal{G}_k r'_j}{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}$$

$$p'_j = r'_j + \beta_{sk+j} p'_{j-1}$$

end

$$[x_{s(k+1)} - x_{sk}, r_{s(k+1)}, p_{s(k+1)}] = \underline{y}_k [x'_s, r'_s, p'_s]$$

end



# s-step CG

$$r_0 = b - Ax_0, p_0 = r_0$$

for  $k = 0:nmax/s$

Compute  $\mathcal{Y}_k$  and  $\mathcal{B}_k$  such that  $A\mathcal{Y}_k = \mathcal{Y}_k\mathcal{B}_k$  and  
 $\text{span}(\mathcal{Y}_k) = \mathcal{K}_{s+1}(A, p_{sk}) + \mathcal{K}_s(A, r_{sk})$

$$\mathcal{G}_k = \mathcal{Y}_k^T \mathcal{Y}_k$$

$$x'_0 = 0, r'_0 = e_{s+2}, p'_0 = e_1$$

for  $j = 1:s$

$$\alpha_{sk+j-1} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}{p'_{j-1}{}^T \mathcal{G}_k \mathcal{B}_k p'_{j-1}}$$

$$x'_j = x'_{j-1} + \alpha_{sk+j-1} p'_{j-1}$$

$$r'_j = r'_{j-1} - \alpha_{sk+j-1} \mathcal{B}_k p'_{j-1}$$

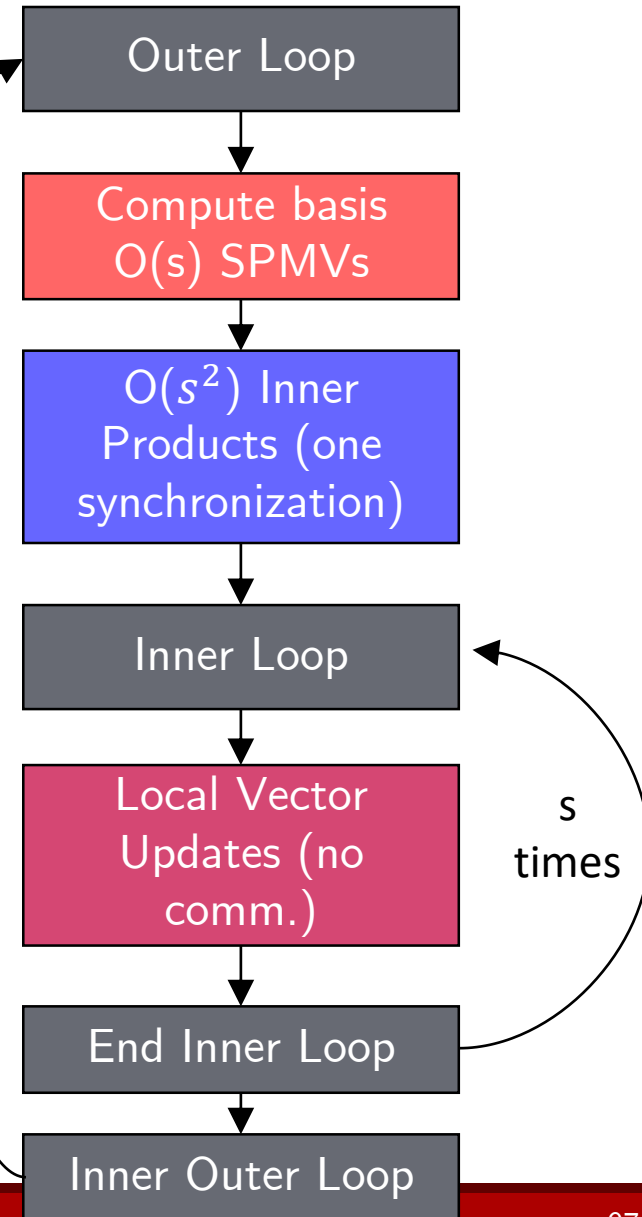
$$\beta_{sk+j} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_j}{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}$$

$$p'_j = r'_j + \beta_{sk+j} p'_{j-1}$$

end

$$[x_{s(k+1)} - x_{sk}, r_{s(k+1)}, p_{s(k+1)}] = \mathcal{Y}_k[x'_s, r'_s, p'_s]$$

end





# s-step CG

$$r_0 = b - Ax_0, p_0 = r_0$$

for  $k = 0:nmax/s$

Compute  $\mathcal{Y}_k$  and  $\mathcal{B}_k$  such that  $A\mathcal{Y}_k = \mathcal{Y}_k\mathcal{B}_k$  and  
 $\text{span}(\mathcal{Y}_k) = \mathcal{K}_{s+1}(A, p_{sk}) + \mathcal{K}_s(A, r_{sk})$

$$\mathcal{G}_k = \mathcal{Y}_k^T \mathcal{Y}_k$$

$$x'_0 = 0, r'_0 = e_{s+2}, p'_0 = e_1$$

for  $j = 1:s$

$$\alpha_{sk+j-1} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}{p'_{j-1}{}^T \mathcal{G}_k \mathcal{B}_k p'_{j-1}}$$

$$x'_j = x'_{j-1} + \alpha_{sk+j-1} p'_{j-1}$$

$$r'_j = r'_{j-1} - \alpha_{sk+j-1} \mathcal{B}_k p'_{j-1}$$

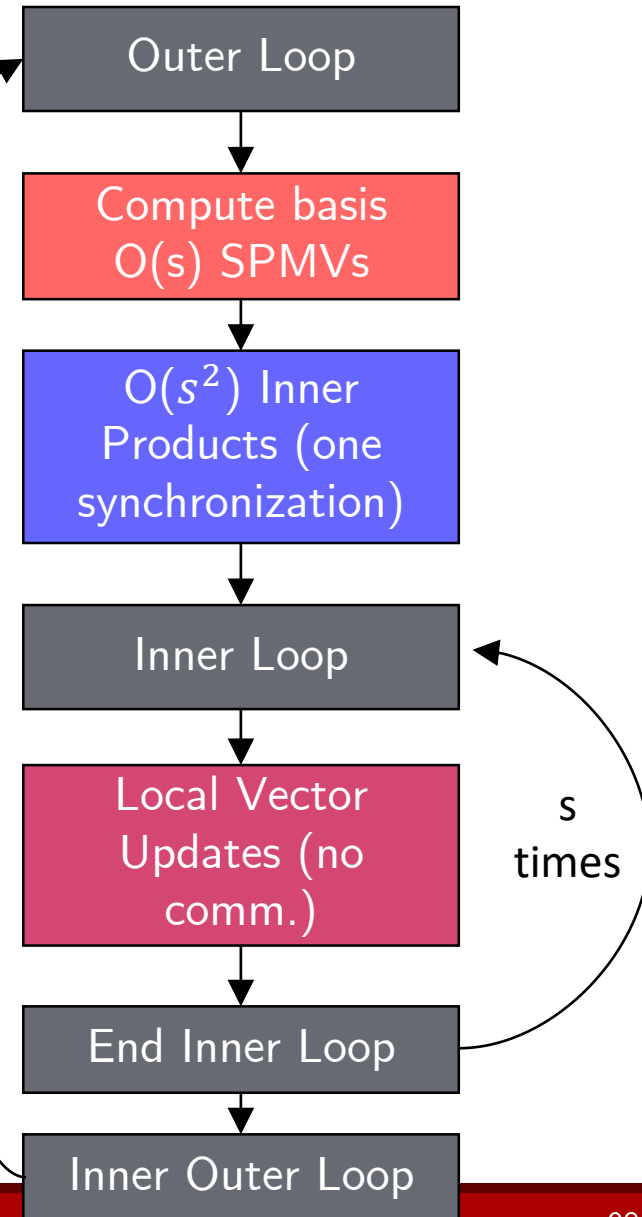
$$\beta_{sk+j} = \frac{r'_{j-1}{}^T \mathcal{G}_k r'_j}{r'_{j-1}{}^T \mathcal{G}_k r'_{j-1}}$$

$$p'_j = r'_j + \beta_{sk+j} p'_{j-1}$$

end

$$[x_{s(k+1)} - x_{sk}, r_{s(k+1)}, p_{s(k+1)}] = \mathcal{Y}_k[x'_s, r'_s, p'_s]$$

end



# CA-CG complexity (1)

Kernel	Computation costs	Communication costs
s dependent SpMV	$2s \cdot \text{nnz}$ flops (1 source vector)	Sequential: <ul style="list-style-type: none"> <li>• Read <math>s</math> vectors of length <math>n</math></li> <li>• Write <math>s</math> vectors of length <math>n</math></li> <li>• Read <math>A</math> <math>s</math> times</li> <li>• bandwidth cost <math>\approx s \cdot \text{nnz} + 2sn</math></li> </ul> Parallel: <ul style="list-style-type: none"> <li>• Distribute 1 source vector <math>s</math> times</li> </ul>
Akx	$4s \cdot \text{nnz}$ flops (2 source vectors)	Sequential: <ul style="list-style-type: none"> <li>• Read 2 vectors of length <math>n</math>,</li> <li>• Write <math>2s-1</math> vectors of length <math>n</math>,</li> <li>• Read <math>A</math> once (both Akx and SpMM optimizations)</li> <li>• bandwidth cost <math>\approx \text{nnz} + (2s+1)n</math></li> </ul> Parallel: <ul style="list-style-type: none"> <li>• Distribute 2 source vectors once</li> <li>• Communication volume and number of messages increase with <math>s</math> (ghost zones)</li> </ul>

# CA-CG complexity (2)

Kernel	Computation costs	Communication costs
2s+1 dot products	Sequential: <ul style="list-style-type: none"> <li>• <math>2(2s+1)n</math> flops</li> <li>• <math>\approx 4ns</math></li> </ul> Parallel: <ul style="list-style-type: none"> <li>• <math>(2s+1)(2n+(p-1))/p</math></li> <li>• <math>\approx 4ns/p</math></li> </ul>	Sequential: <ul style="list-style-type: none"> <li>• Read a vector of length <math>n</math> <math>2s+1</math> times</li> </ul> Parallel: <ul style="list-style-type: none"> <li>• <math>2s+1</math> all-reduce collectives, each with <math>\lg(p)</math> rounds of messages: latency cost <math>\approx 2s \lg(p)</math></li> <li>• 1 word to/from each proc.: bandwidth cost <math>\approx 2s \lg(p)</math></li> </ul>
Gram matrix	Sequential: <ul style="list-style-type: none"> <li>• <math>(2s+1)^2 n</math> flops</li> <li>• <math>\approx 4ns^2</math></li> </ul> Parallel: <ul style="list-style-type: none"> <li>• <math>(2s+1)^2(n/p + (p-1)/(2p))</math></li> <li>• <math>\approx 4ns^2/p</math></li> </ul>	Sequential: <ul style="list-style-type: none"> <li>• Read a matrix of size <math>(2s+1) \times n</math> once</li> </ul> Parallel: <ul style="list-style-type: none"> <li>• 1 all-reduce collective, with <math>\lg(p)</math> rounds of messages: latency cost <math>\approx \lg(p)</math></li> <li>• <math>(2s+1)^2/2</math> words to/from each proc.: bandwidth cost <math>\approx 4s^2 \lg(p)</math></li> </ul>

# CA-CG complexity (3)

Using Gram matrix and coefficient vectors have additional costs for CA-CG:

- Dense work (besides dot products/Gram matrix, i.e., vector updates) does not significantly increase with  $s$ :

$$\text{CG} \approx 6sn = O(sn)$$

$$\text{CA-CG} \approx 3(2s+1)(2s+n) = O(sn)$$

- Sequential memory traffic decreases

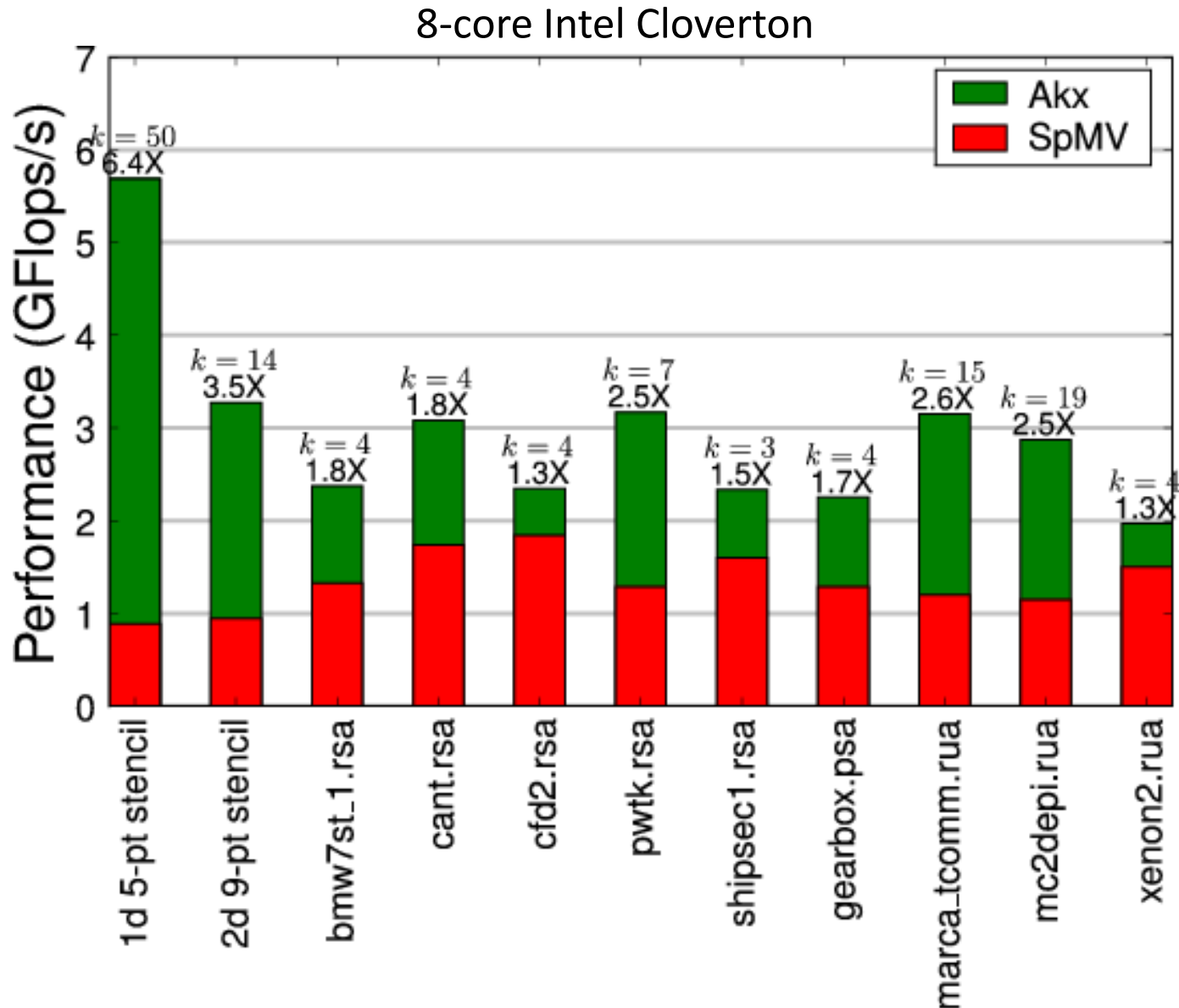
$$\text{CG} \approx 6sn \text{ reads, } 3sn \text{ writes}$$

$$\text{CA-CG} \approx (2s+1)n \text{ reads, } 3n \text{ writes}$$

- Example asymptotic costs for 1D 3-point stencil:

Method	Parallel flops	Parallel bandwidth	Parallel latency
CG, $s$ steps	$s \cdot \text{nnz}/p + sn/p$	$s + s \lg(p)$	$s + s \lg(p)$
CA-CG( $s$ ), 1 step	$s \cdot \text{nnz}/p + s^2 n/p$	$s + s^2 \lg(p)$	$1 + \lg(p)$

# Multicore Speedups

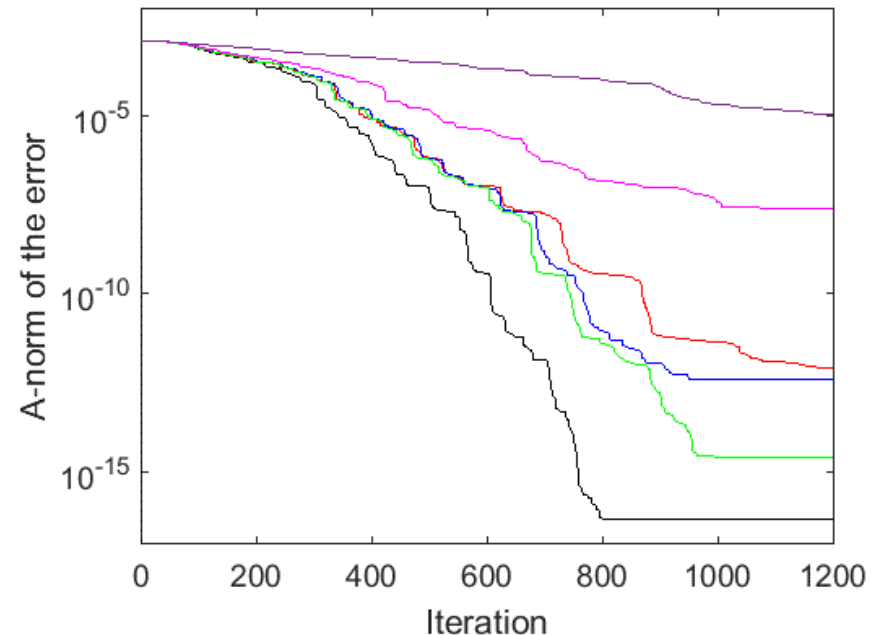


# Optimizing high performance iterative solvers

- Synchronization-reducing variants are designed to reduce the time/iteration
- But this is not the whole story!
- What we really want to minimize is the **runtime, subject to some constraint on accuracy**,

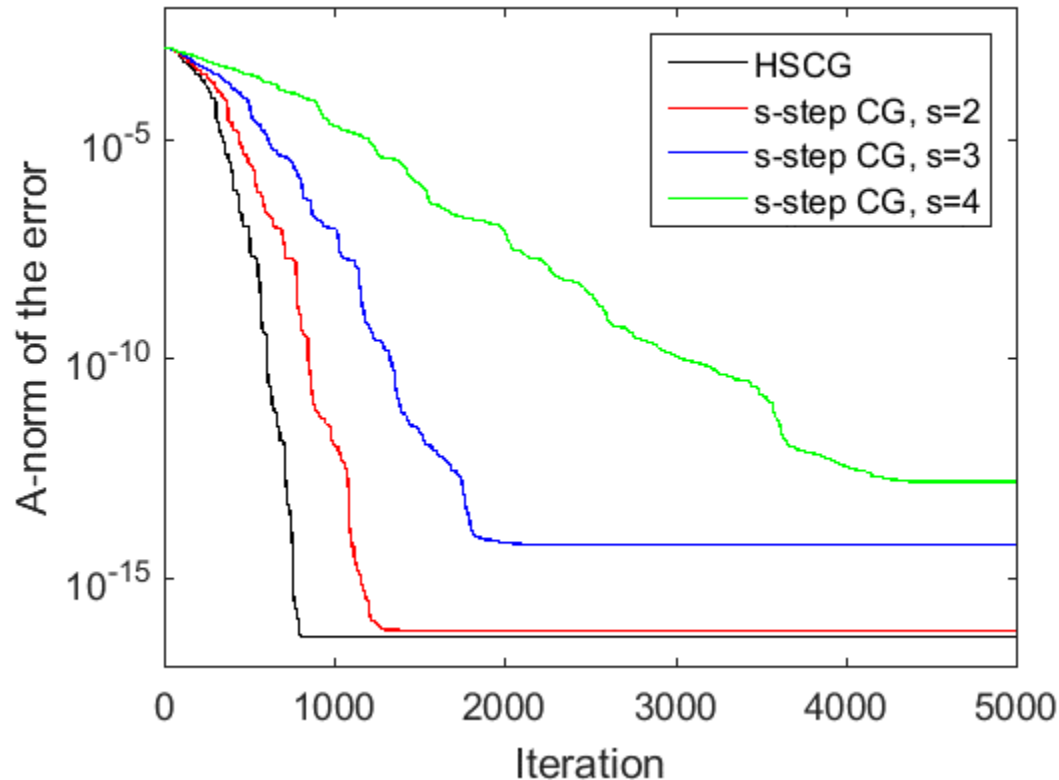
$$\text{runtime} = (\text{time/iteration}) \times (\# \text{ iterations})$$

- Changes to how the recurrences are computed can exacerbate finite precision effects of convergence delay and loss of accuracy
- Crucial that we understand and take into account how algorithm modifications will affect the convergence rate and attainable accuracy!



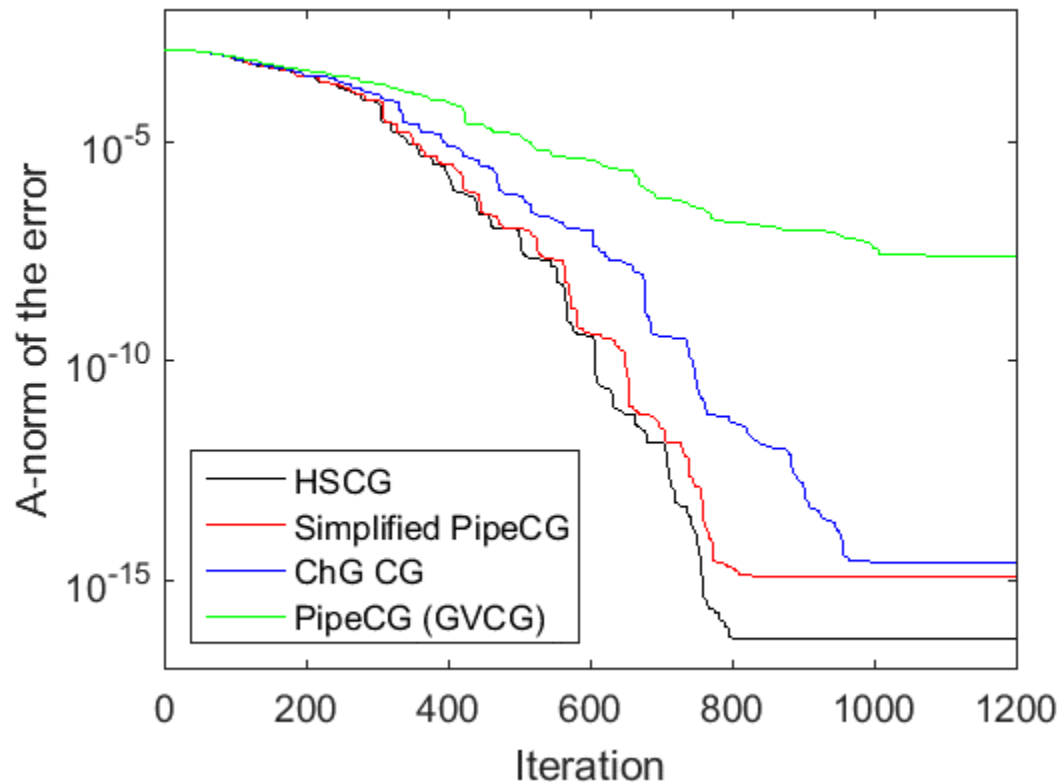
# s-step CG

s-step CG with monomial basis ( $\mathcal{Y} = [p_i, Ap_i, \dots, A^s p_i, r_i, Ar_i, \dots, A^{s-1} r_i]$ )



Can also use other, more well-conditioned bases to improve convergence rate and accuracy (see, e.g. Philippe and Reichel, 2012).

# Simple pipelined CG

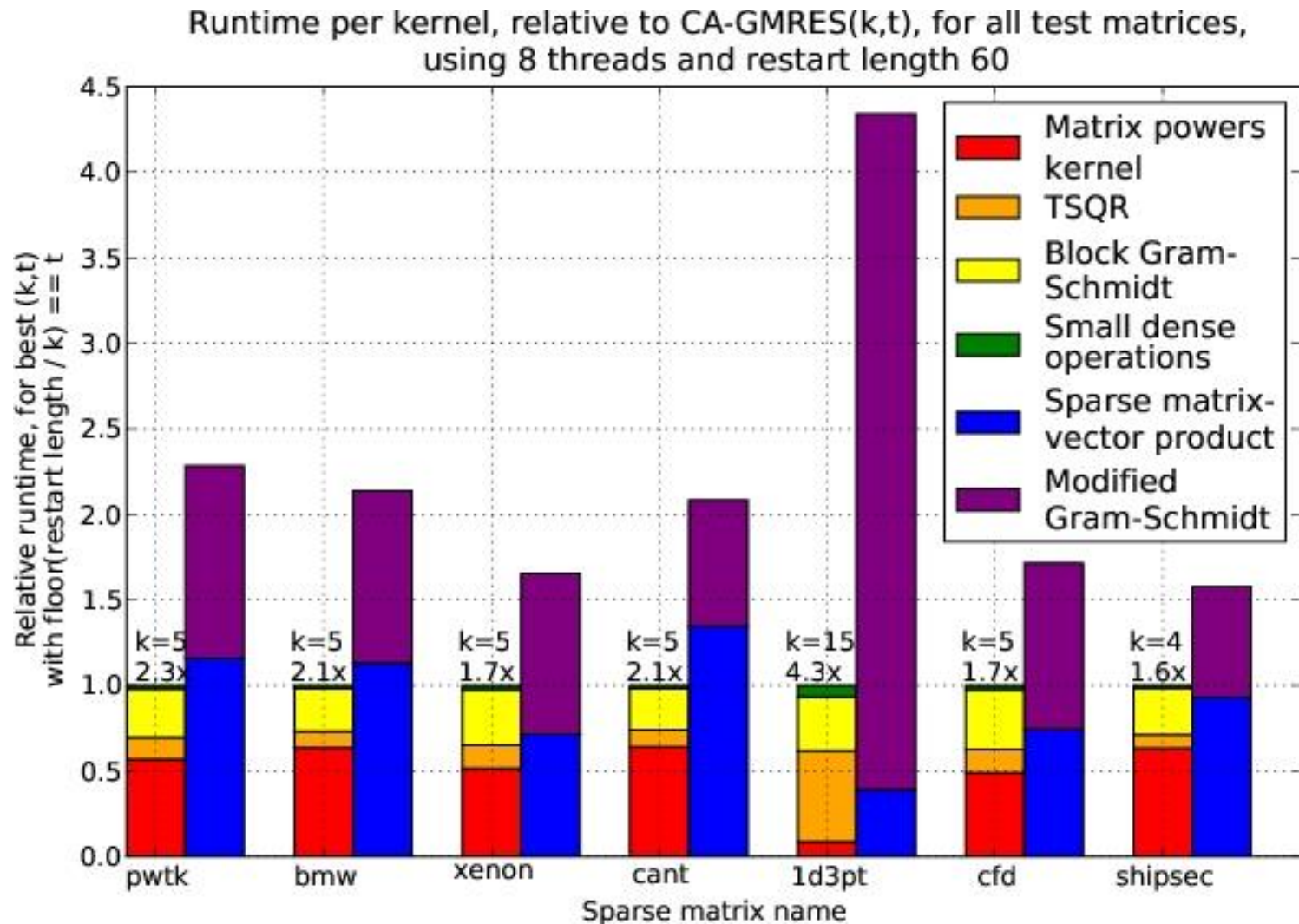


effect of changing formula for recurrence coefficient  $\alpha$  and  
using auxiliary vectors  $s_i \equiv Ap_i$ ,  $w_i \equiv Ar_i$ ,  $z_i \equiv A^2r_i$



# Communication-Avoiding Krylov Method (GMRES)

Performance on 8 core Clovertown



# Preconditioning

- Instead of solving  $Ax = b$ , solve
  - $M^{-1}Ax = M^{-1}b$  (left), or
  - $AM^{-1}y = b$  and  $Mx = y$  (right)
- Where goal is that preconditioned system converges faster


$$\text{Runtime} = (\text{time/iteration}) \times (\# \text{ iterations})$$

- Hard to design preconditioners for communication-avoiding methods
  - Rule-of-thumb: the better the preconditioner, the more communication needed to apply it

# Preconditioners: Current Work

- Current communication-avoiding preconditioners
  - **Diagonal**
  - **Sparse Approximate Inverse (SAI)** –(Mehri, 2014)
  - **CA-ILU(0)** – (Moufawad, Grigori, 2013)
  - **CA-ILU(k)** –(Nataf, Moufawad, Grigori, 2015)
  - **Domain decomposition** – (Yamazaki, Rajamanickam, Boman, Hoemmen, Heroux, Tomov, 2014)
  - **HSS preconditioning** – (Hoemmen, 2010); (Knight, C., Demmel, 2014)
  - **Deflation** – (C., Knight, Demmel, 2014); (Yamazaki et al., 2014)

An active area of ongoing research...

# Other Active Areas of Research

- Pipelined and s-step variants of other Krylov subspace methods
- “Low-synch” variants of Krylov subspace methods
  - “Low synchronization GMRES algorithms”, Swirydowicz et al., 2018, <https://arxiv.org/abs/1809.05805>
- Hypergraph models of communication for other sparse computations
- Other sparse computations:
  - Sparse matrix x sparse matrix
  - Sparse matrix x dense matrix
  - Sparse matrix x sparse vector
  - “Data-sparse” matrices (e.g., hierarchical semiseparable structures)
- Mixed precision approaches