Efficient sparse matrix–vector multiplication

Albert-Jan Yzelman

February, 2012

www.exascience.com

1. Space weather modeling
2. Simulation toolkit
3. Architectural simulation
4. Visualisation

www.exascience.com

1. Space weather modeling
2. Simulation toolkit $\Leftarrow$ includes **solvers**, supporting **kernels**
3. Architectural simulation
4. Visualisation

Central question:

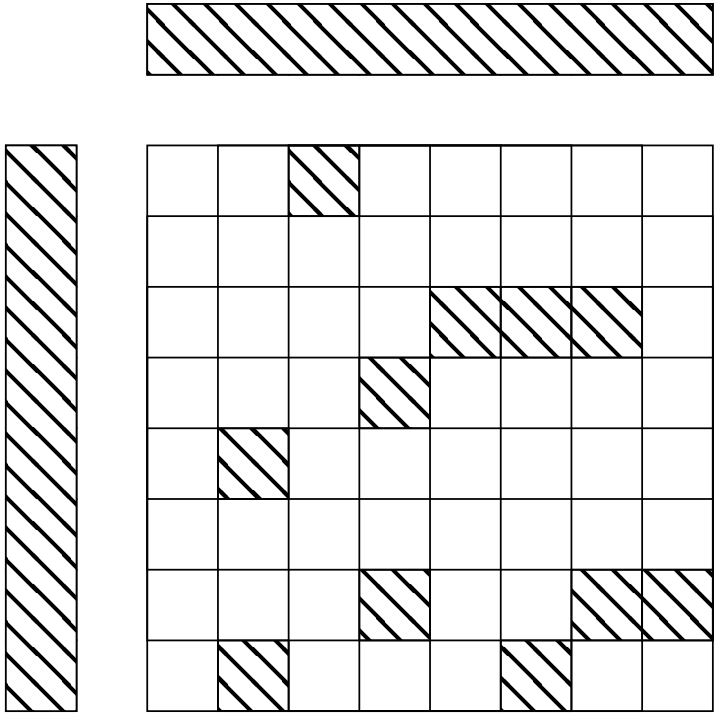
how to calculate

$$y = Ax$$

on a (parallel) computer, as fast as possible?

Initial observations

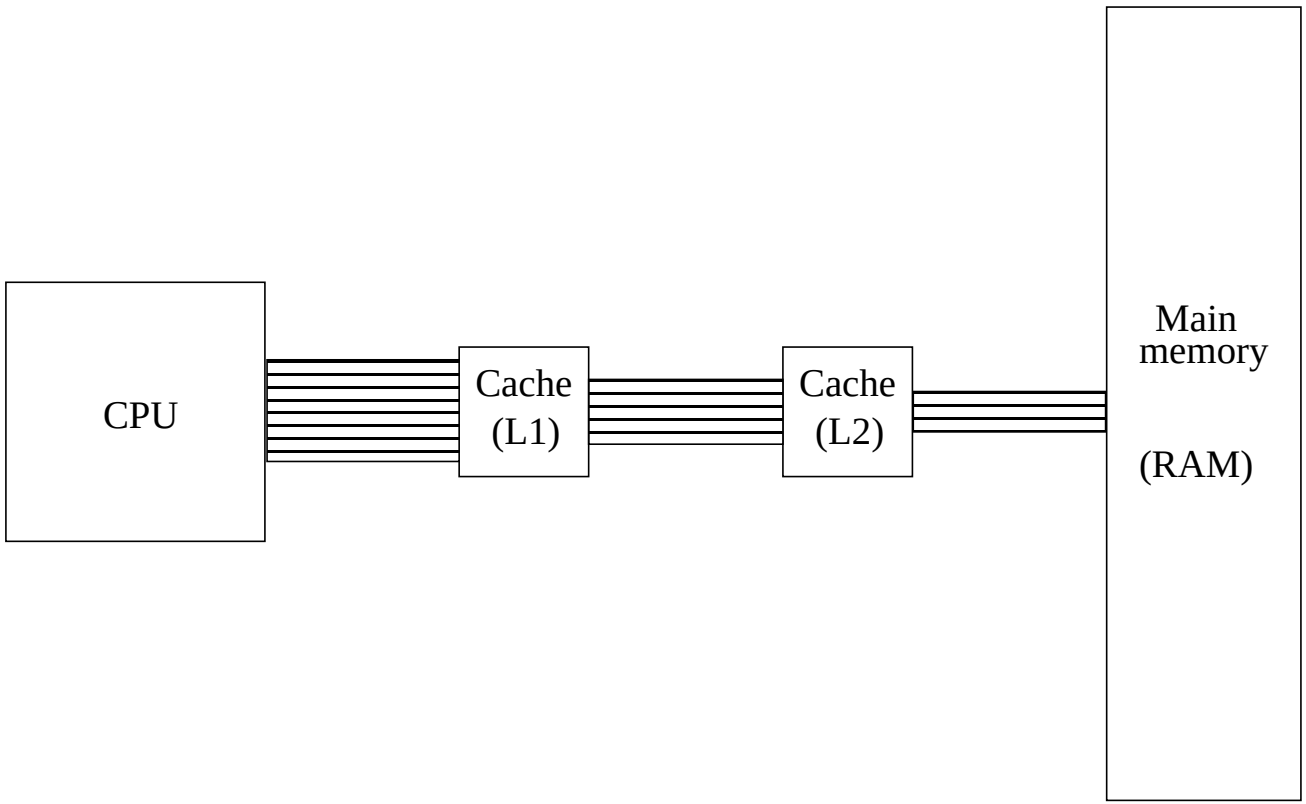
$y = Ax =$



Two main issues:

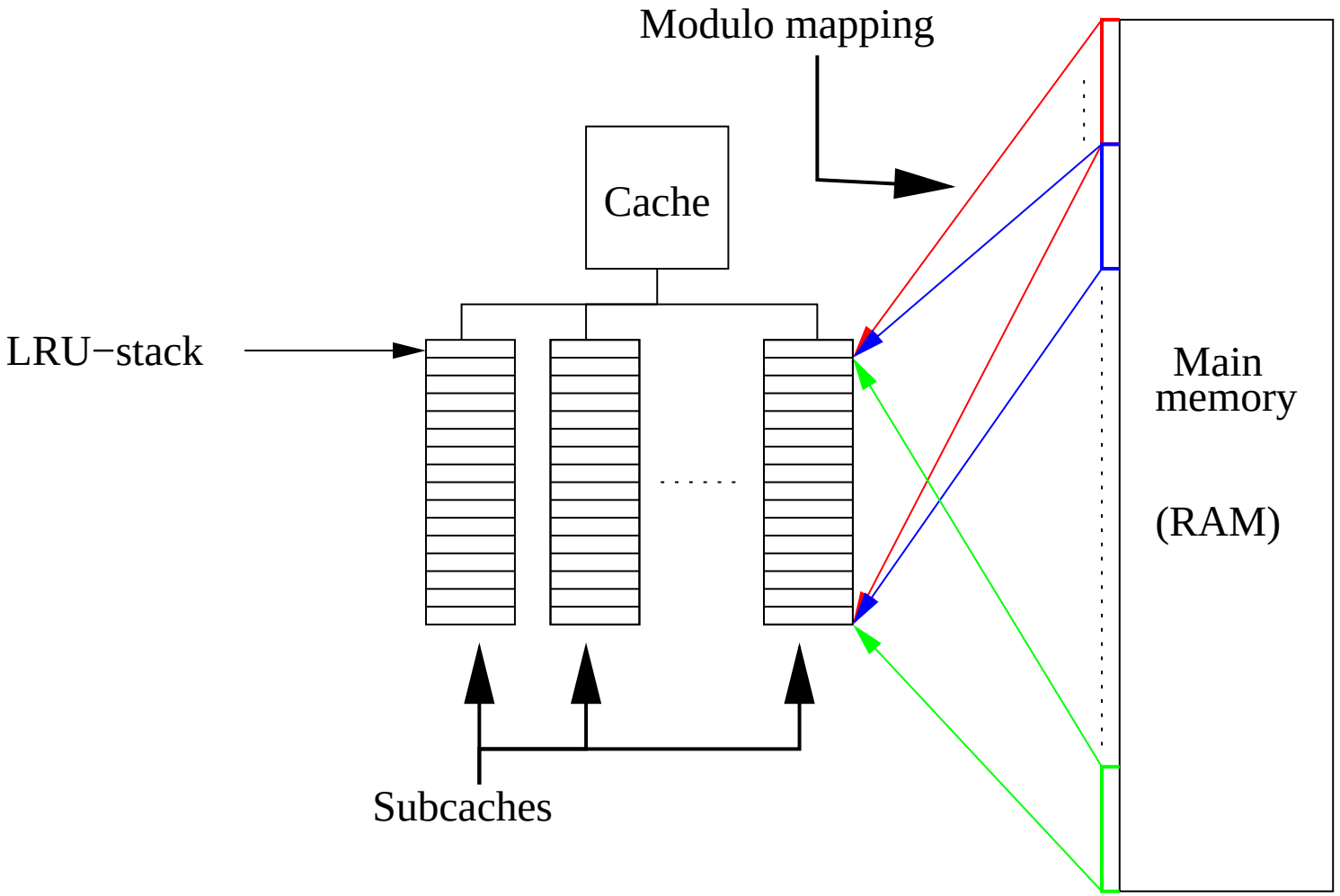
1. bandwidth-limited: two flops versus *at least* three data elements;
2. cache inefficiency: inefficient *vector element reuse*.

The cure and the cause



The cure and the cause

Realistic cache: both modulo-mapping and the LRU policy



The dense SpMV

Dense matrix–vector multiplication

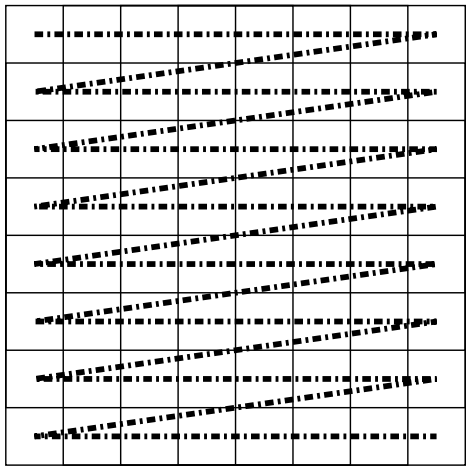
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $L = 5$:

$$\begin{array}{ccccccccccc} x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} & & y_0 \\ & & x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} \\ & \Longrightarrow & & \Longrightarrow & x_0 & \Longrightarrow & a_{00} & \Longrightarrow & y_0 & \Longrightarrow & x_1 \\ & & & & & & x_0 & & a_{00} & & x_0 \\ \hline & & \hline & & \hline & & \hline & & \hline & & \hline \end{array} \quad \Longrightarrow \quad \begin{array}{c} a_{02} \\ y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ \hline x_0 \end{array}$$

The sparse SpMV

Standard data structure: Compressed Row Storage (CRS)

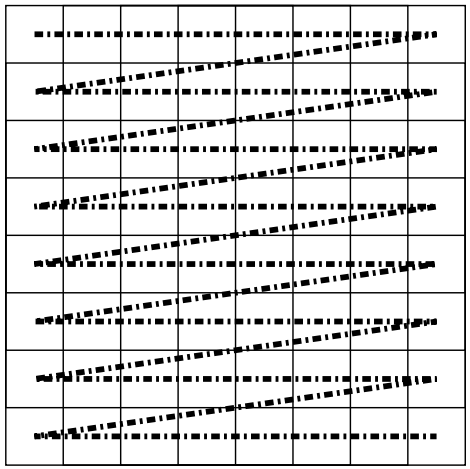


x?

$\Longrightarrow$

The sparse SpMV

Standard data structure: Compressed Row Storage (CRS)



$x_?$ $\quad$ $a_{0?}$ $\quad$ y_0

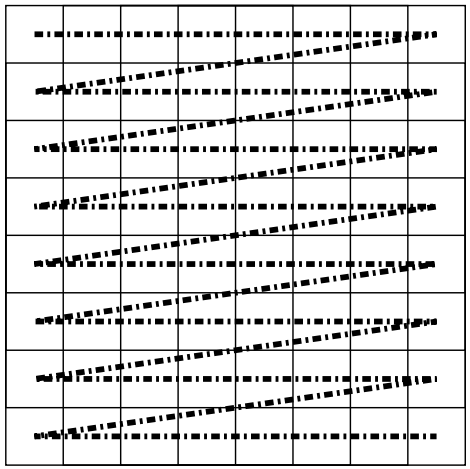
$\quad$ $x_?$ $\quad$ $a_{0?}$

$\quad$ $x_?$

$\Longrightarrow$ $\quad$ $\Longrightarrow$ $\quad$ $\Longrightarrow$

The sparse SpMV

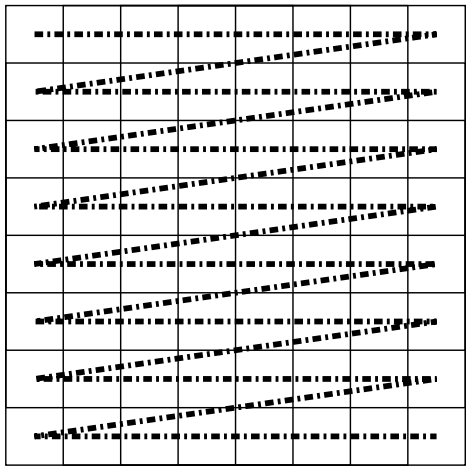
Standard data structure: Compressed Row Storage (CRS)



$$x_? \qquad a_{0?} \qquad y_0 \qquad x_? \qquad \Longrightarrow \qquad x_? \qquad \Longrightarrow \qquad a_{0?} \qquad y_0 \qquad \Longrightarrow \qquad a_{0?} \qquad a_{0?} \qquad \Longrightarrow \qquad x_? \qquad x_? \qquad \Longrightarrow$$

The sparse SpMV

Standard data structure: Compressed Row Storage (CRS)



$$x_? \Longrightarrow \begin{matrix} a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} y_0 \\ a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} x_? \\ y_0 \\ a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} a_{??} \\ x_? \\ y_0 \\ a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} y_? \\ a_{??} \\ x_? \\ y_0 \\ a_{0?} \\ x_? \end{matrix}$$

Memory accesses cannot be predicted!

Solutions

Solutions for efficient SpMV:

1. Parallel multiplication: partitioning
2. Sequential multiplication: reordering

...but are these really different solutions?

The (classic) parallel SpMV: partitioning

1 The (classic) parallel SpMV: partitioning

2 Sequential SpMV

3 Parallel cache-friendly SpMV

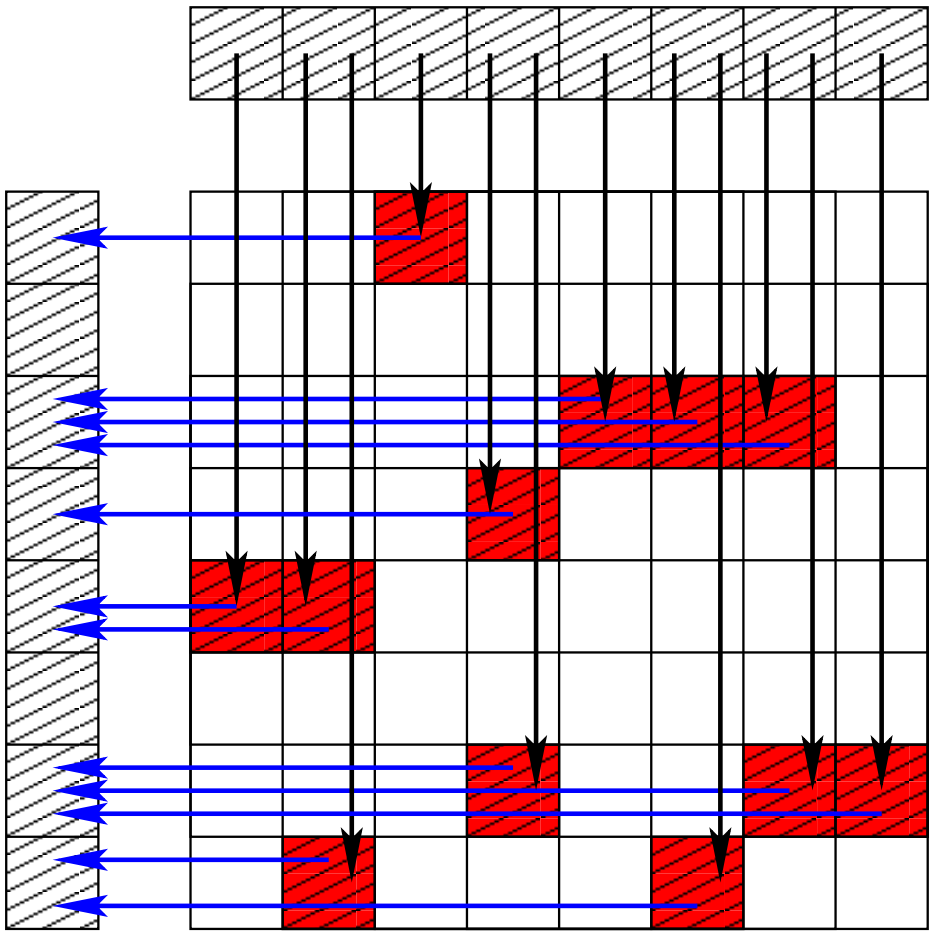
4 Experimental results

Parallel SpMV multiplication

Distribute the nonzeroes of A, then:

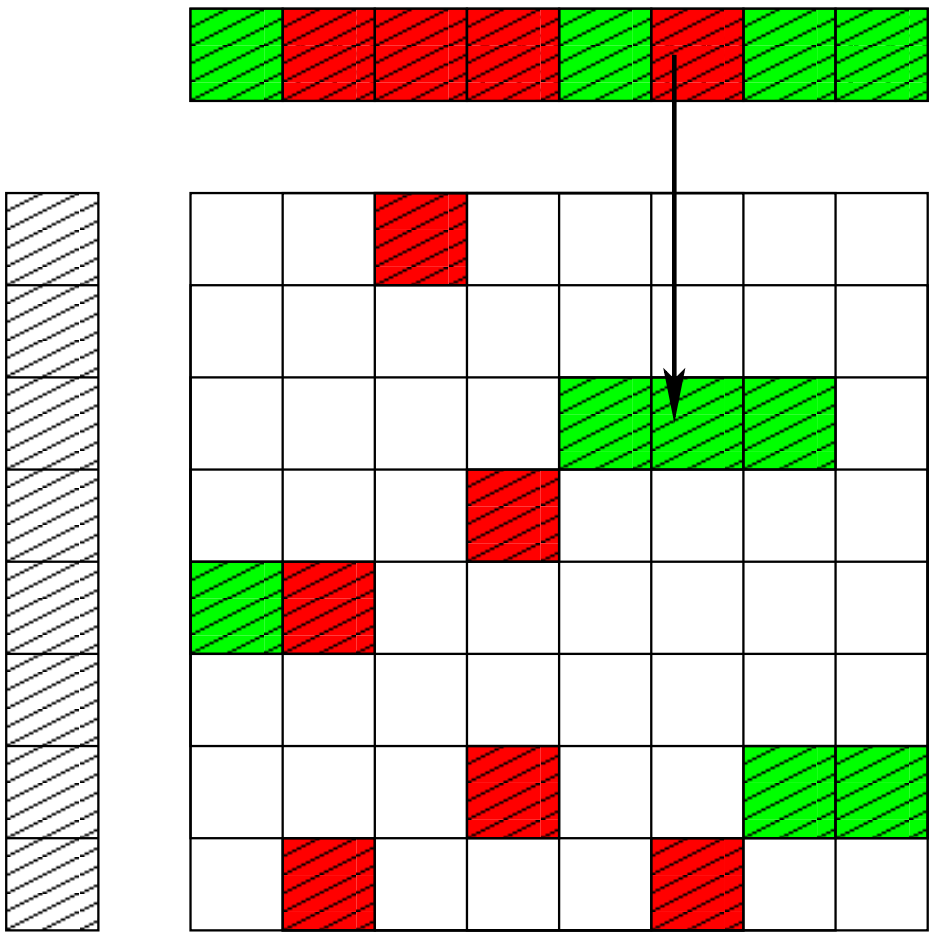
for each nonzero k **from** (local) A

add $x[k.column] \cdot k.value$ **to** $y[k.row]$



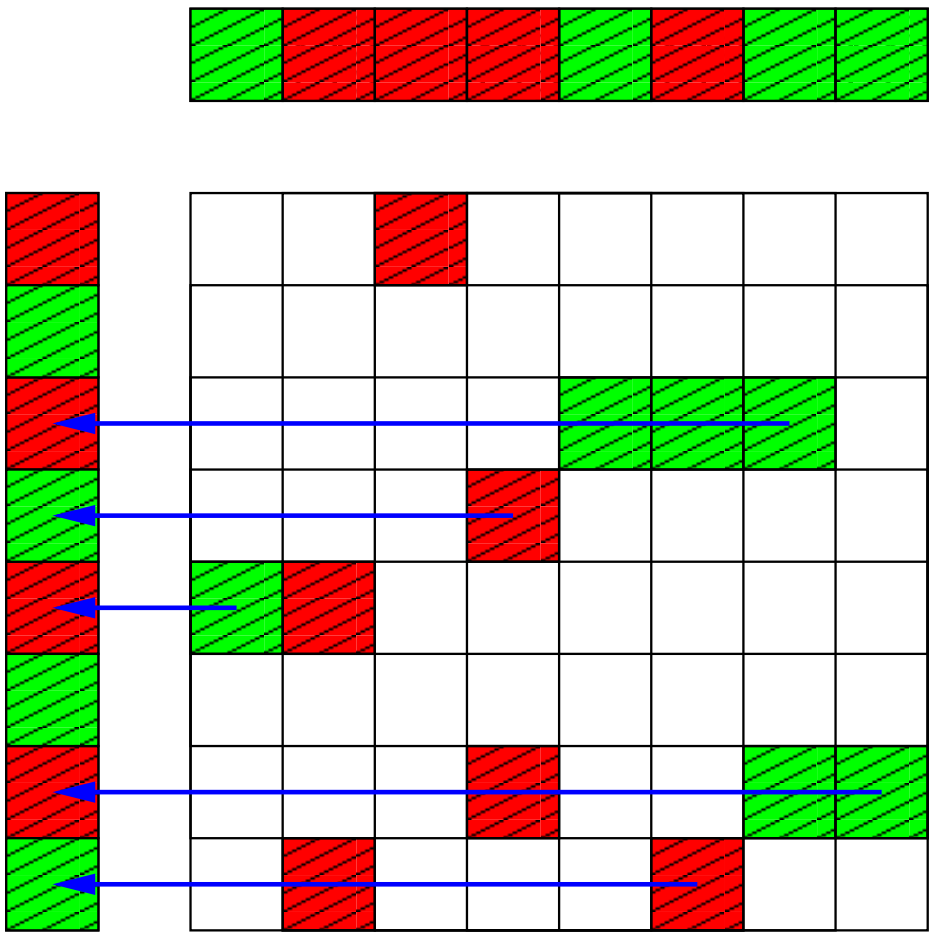
Parallel SpMV multiplication

Step 1 (*fan-out*): not all processors have the elements from x they need; processors need to get the missing items. Here, only one message is needed, x is well-partitioned.



Parallel SpMV multiplication

Step 2 (*mv*): use received elements from x for multiplication.
Step 3 (*fan-in*): send local results to the correct processors;
here, y is distributed cyclically, obviously a bad choice.



Parallel SpMV multiplication

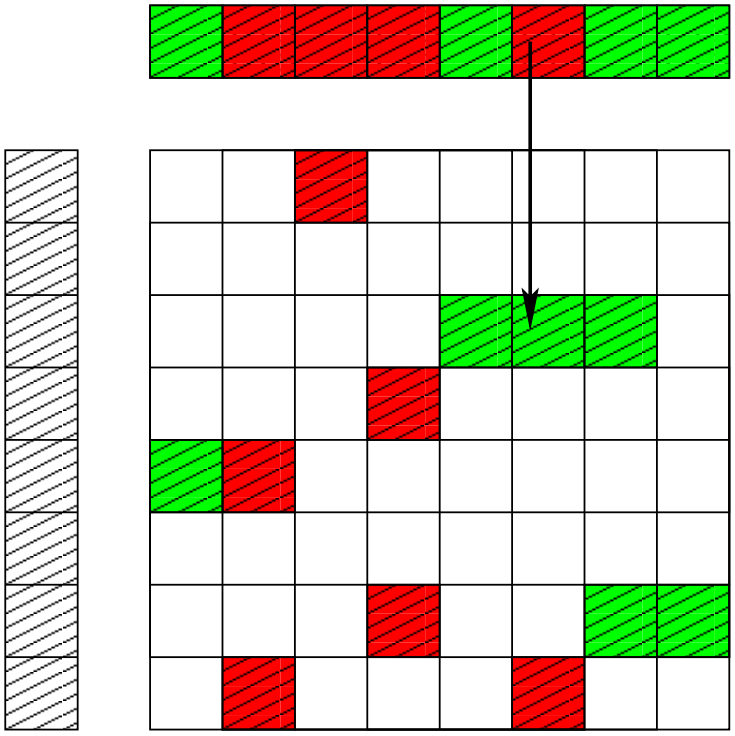
The classical Bulk Synchronous Parallel SpMV multiplication:

1. **for all** nonzeroes k **from** A
   if column of k is not local
   request element from x from the appropriate processor
   synchronise
2. **for all** nonzeroes k **from** A
   do the SpMV for k
   send all non-local row sums to the appropriate processor
   synchronise
3. **add** all incoming row sums to the corresponding $y[i]$

Automatic nonzero partitioning

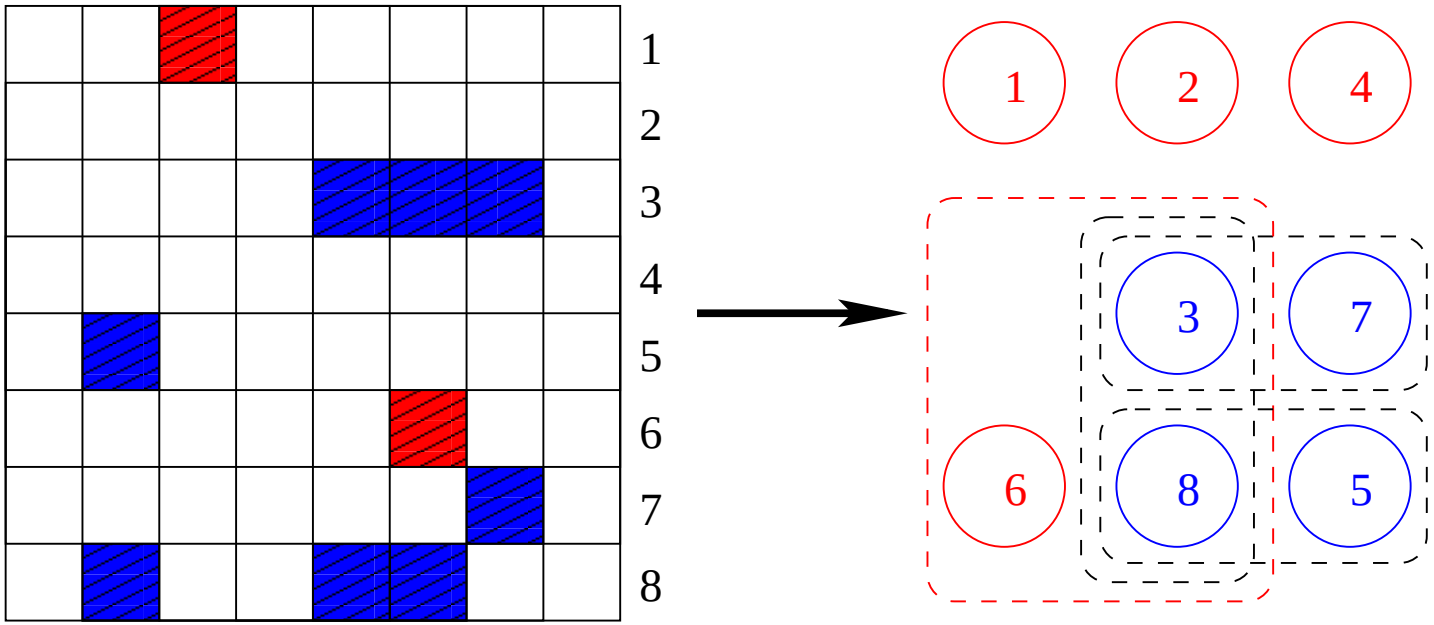
What causes the communication?

- nonzeroes on the same column distributed to different processors: *fan-out* communication



Automatic nonzero partitioning

"Shared" columns: communication during *fan-out*

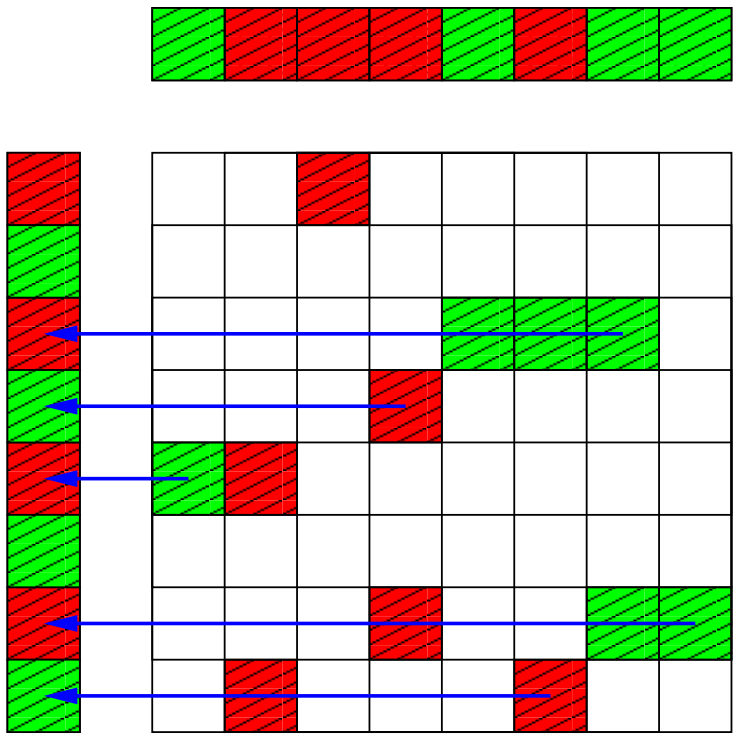


Column-net model; a cut net means a shared column

Automatic nonzero partitioning

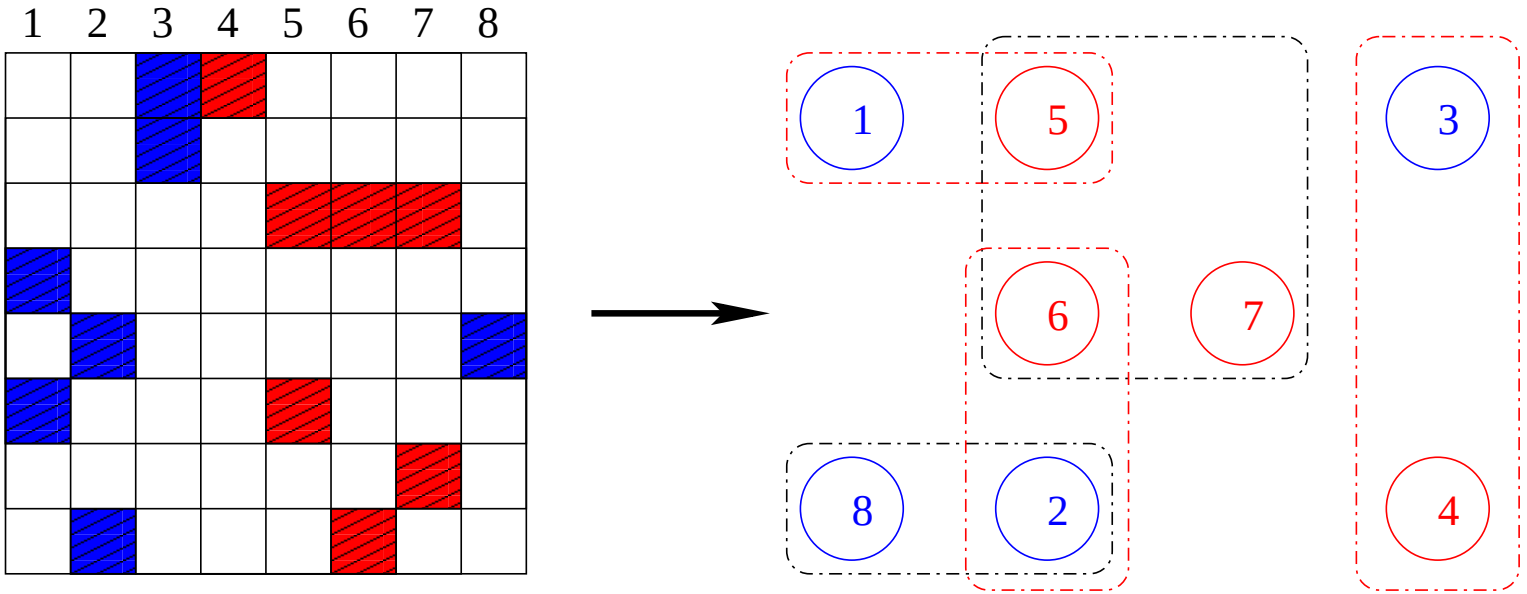
What causes the communication?

- nonzeroes on the same row distributed to different processors: fan-in communication



Automatic nonzero partitioning

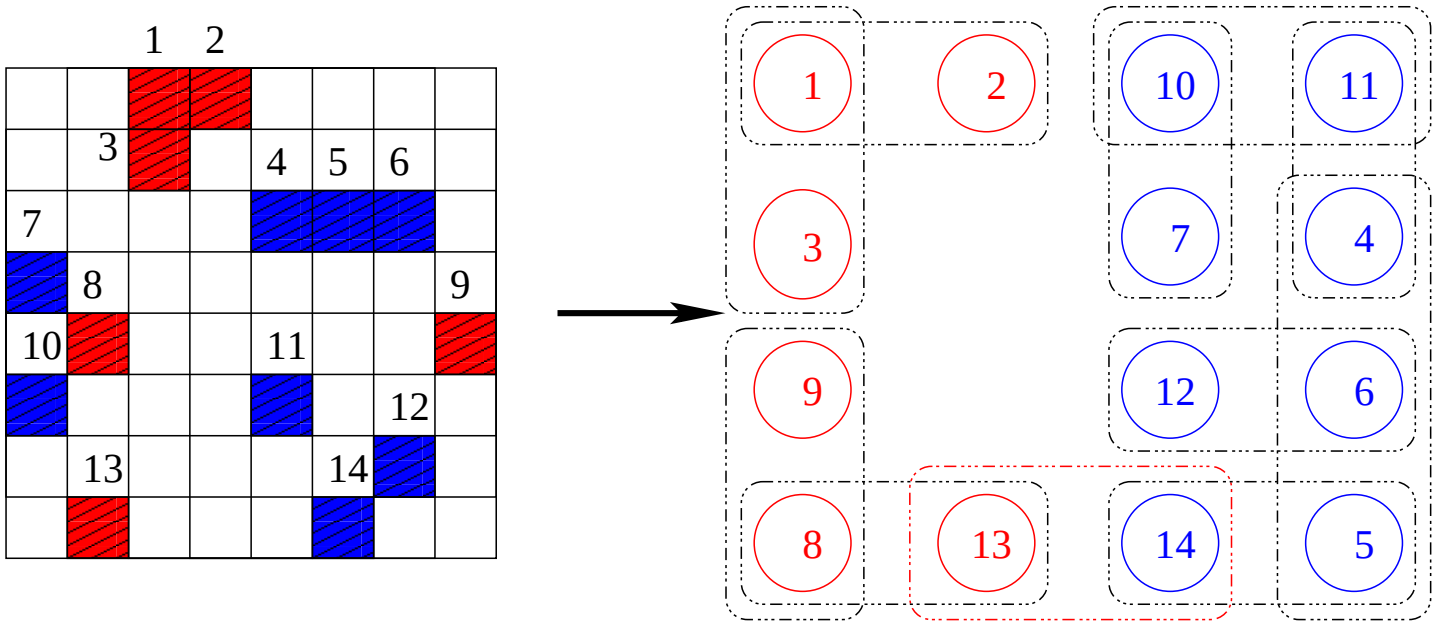
"Shared" rows: communication during fan-in



Row-net model; a cut net means a shared row

Automatic nonzero partitioning

Catch both fan-in and fan-out communication:



Fine-grain model; a cut net means either a shared row or column.

Disadvantage: the hypergraph is larger and more complicated than those corresponding to the previously shown 1D models.

Automatic nonzero partitioning

A cut net n_i means communication. The number of processors involved in processing the ith net, also called the *connectivity* of the ith net, is:

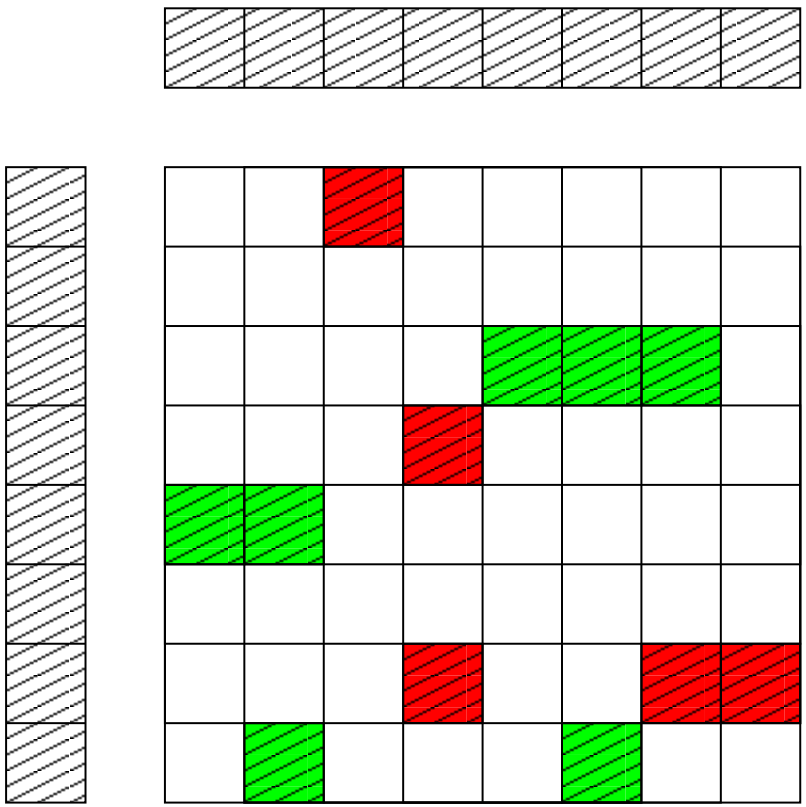
$$\lambda_i = \#\{\mathcal{V}_i \mid \mathcal{V}_i \cap n_i \neq \emptyset\}.$$

The quantity to minimise (under the constraint of load-balance) is:

$$C = \sum_i (\lambda_i - 1)\,.$$

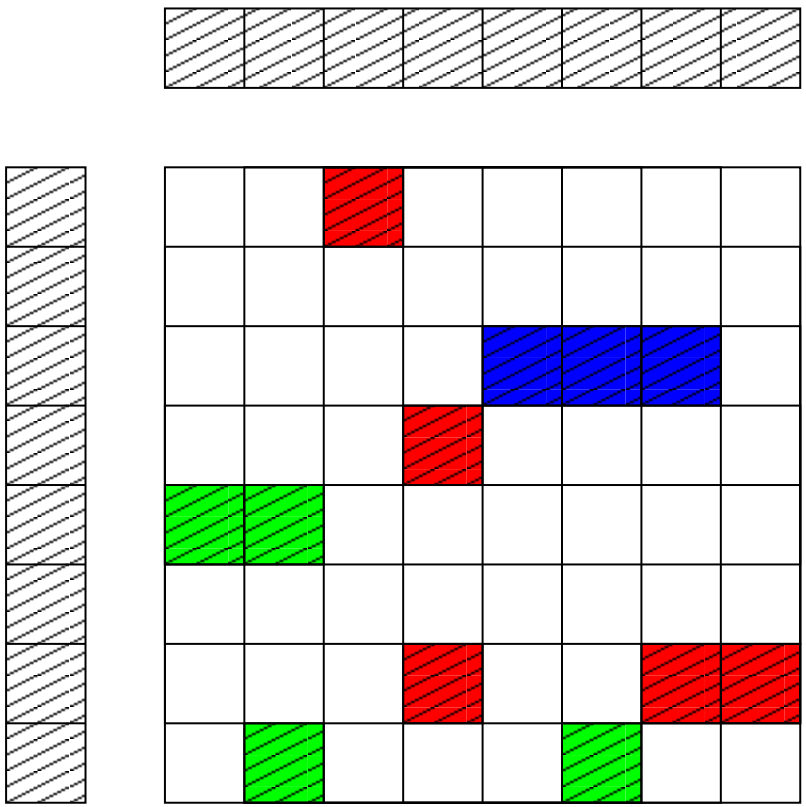
Automatic nonzero partitioning

- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)



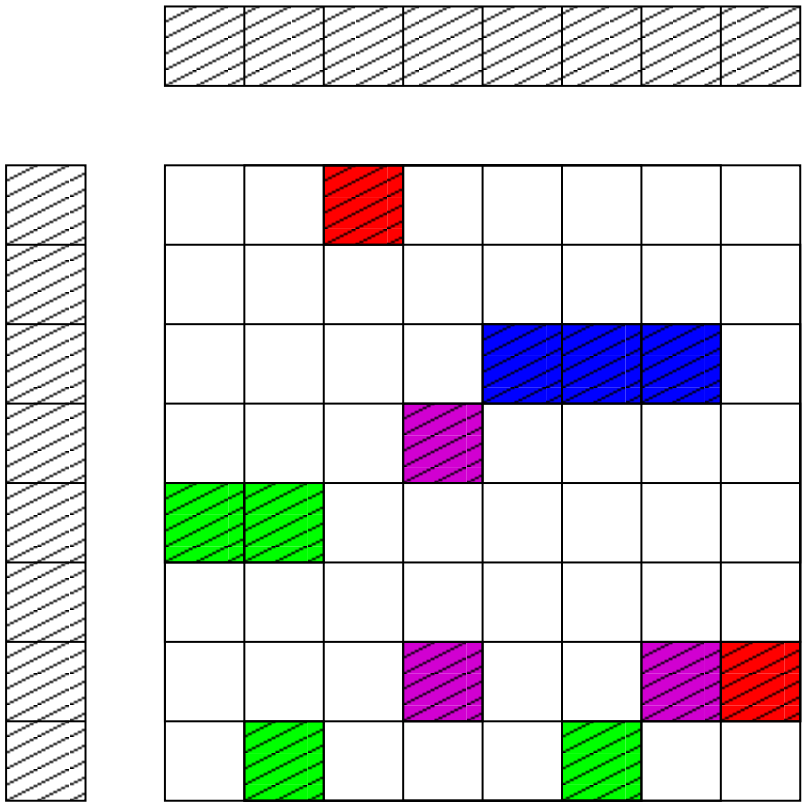
Automatic nonzero partitioning

- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



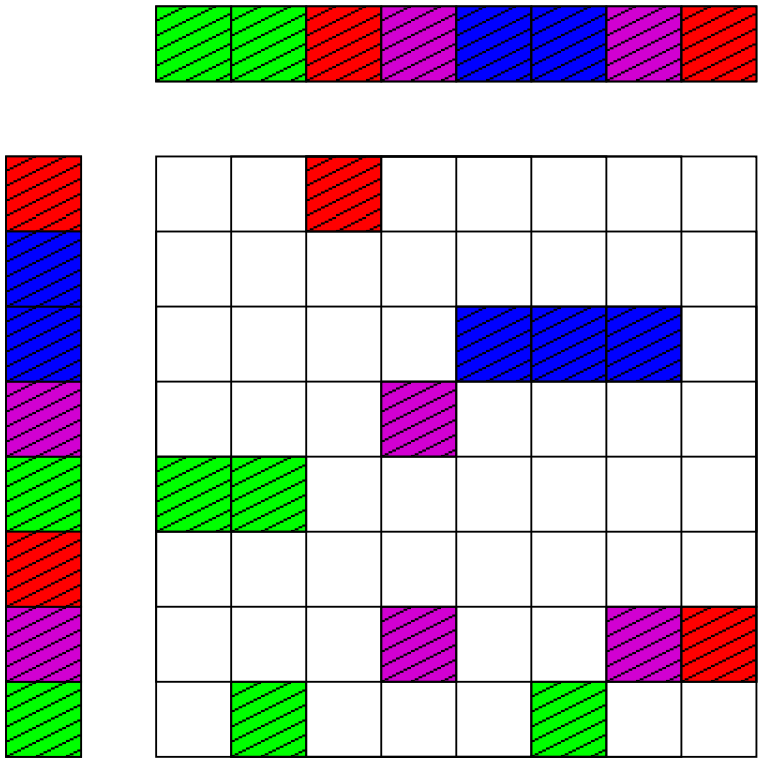
Automatic nonzero partitioning

- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



Automatic nonzero partitioning

- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph in two
- Recursively keep partitioning the vertex parts
- **Partition the vector elements**



Automatic nonzero partitioning

References

Modeling:

Catalyürek & Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel Distributed Systems 10 (1999).

Catalyürek & Aykanat, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, Proc. IPDPS 8th Int'l Workshop on Solving Irregularly Structured Problems in Parallel (2001).

Bisseling & Vastenhouw, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review Vol. 47(1), 2005.

Partitioning:

Kernighan & Lin, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Technical Journal 49 (1970).

Fiduccia & Mattheyses, *A linear-time heuristic for improving network partitions*, Proceedings of the 19th IEEE Design Automation Conference (1982).

Catalyürek & Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool*, Bilkent University, Ankara (1999–present)

Bisseling, Fagginger Auer, van Leeuwen, Meesen, Vastenhouw, Yzelman, *Mondriaan for sparse matrix partitioning*, Utrecht University (2002–present).

Bisseling and Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis, Vol. 21 (2005) pp. 47-65

Sequential SpMV

1 The (classic) parallel SpMV: partitioning

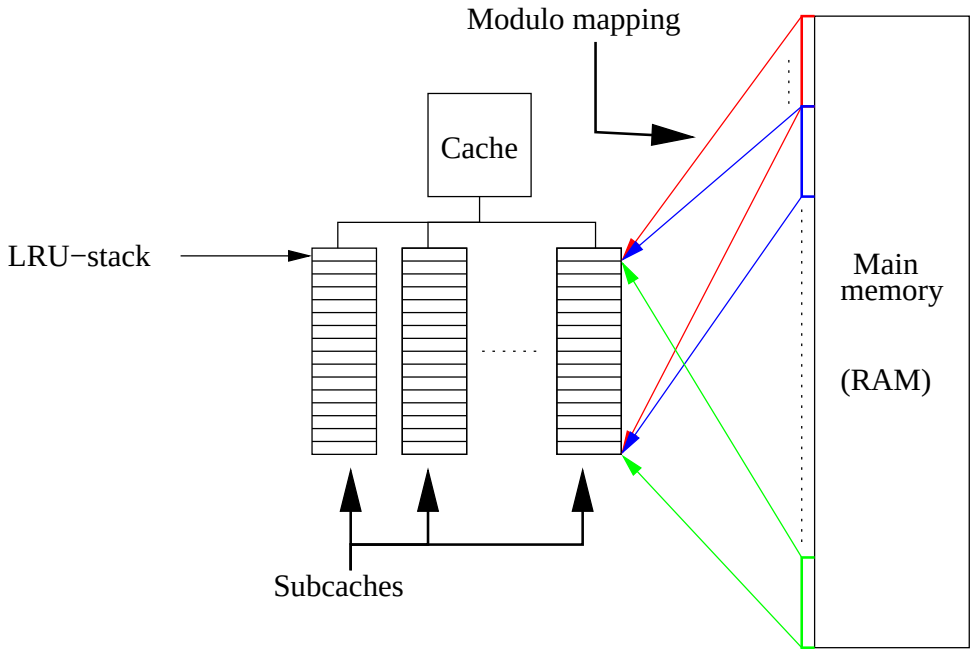
2 Sequential SpMV

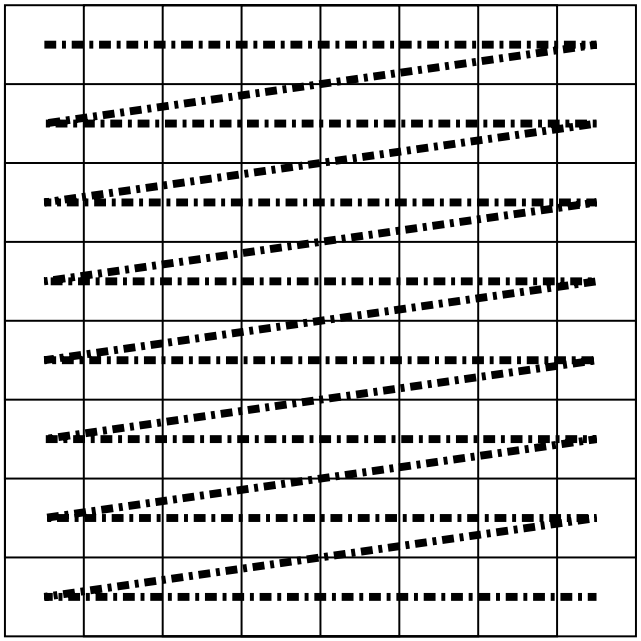
3 Parallel cache-friendly SpMV

4 Experimental results

Assumptions

We assumed a cache which combines modulo-mapping and the LRU policy, and that the sparse matrix is stored in CRS form:





CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

nzs: [4 1 3 2 3 1 2 7 1 1]
col: [0 1 2 2 3 0 3 0 2 3] , $2nnz + (m + 1)$ accesses
row: [0 3 5 7 10]

See e.g.: Barrett et al., *Templates for the solution of linear systems*, SIAM, Philadelphia, 1994

Incremental CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

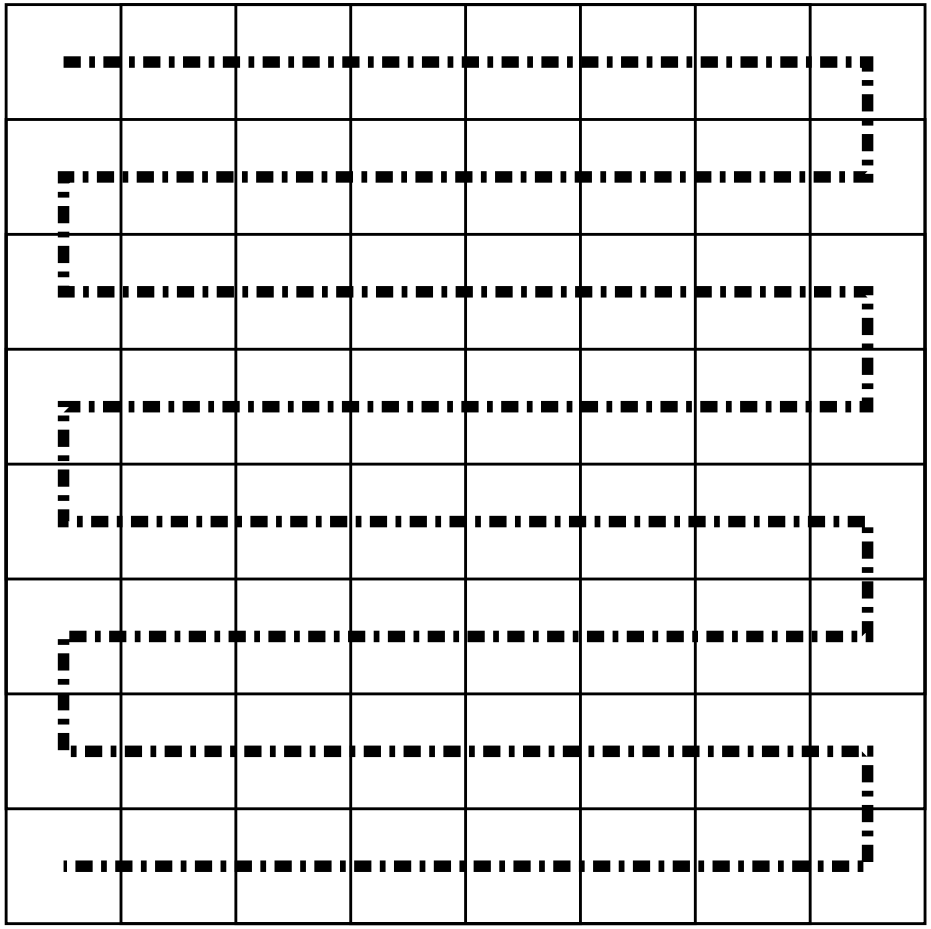
Stored as:

nzs:	[4 1 3 2 3 1 2 7 1 1]
col_increment:	[0 1 1 4 1 1 3 1 2 1] , $2nnz + m$ accesses
row_increment:	[0 1 1 1]

Joris Koster, *Parallel templates for numerical linear algebra*, Master's Thesis, Utrecht University, 2002

Zig-zag CRS

Change the order of CRS (slightly cache-oblivious):



Zig-zag CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

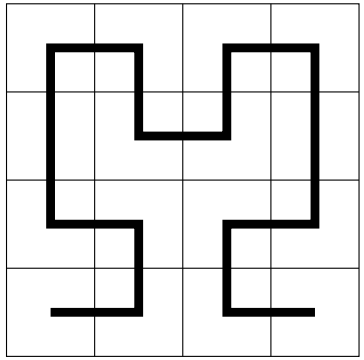
Stored as:

nzs: [4 1 3 3 2 1 2 1 1 7]
col: [0 1 2 3 2 0 3 3 2 0] , $2nnz + (m + 1)$ accesses
row: [0 3 5 7 10]

Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SIAM Journal on Scientific Computing, 2009

Fractal datastructures (triplets)

$$A = \begin{pmatrix} 4 & 1 & 0 & 2 \\ 0 & 2 & 0 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 0 \end{pmatrix}$$



Stored as:

nzs: [7 1 4 1 2 2 3 2 1]
i : [3 2 0 0 1 0 1 2 3] , **3nnz** accesses per nonzero
j : [0 0 0 1 1 3 3 3 2]

Haase, Liebmann and Plank, *A Hilbert-Order Multiplication Scheme for Unstructured Sparse Matrices*, 2005

SBD-based cache-oblivious SpMV

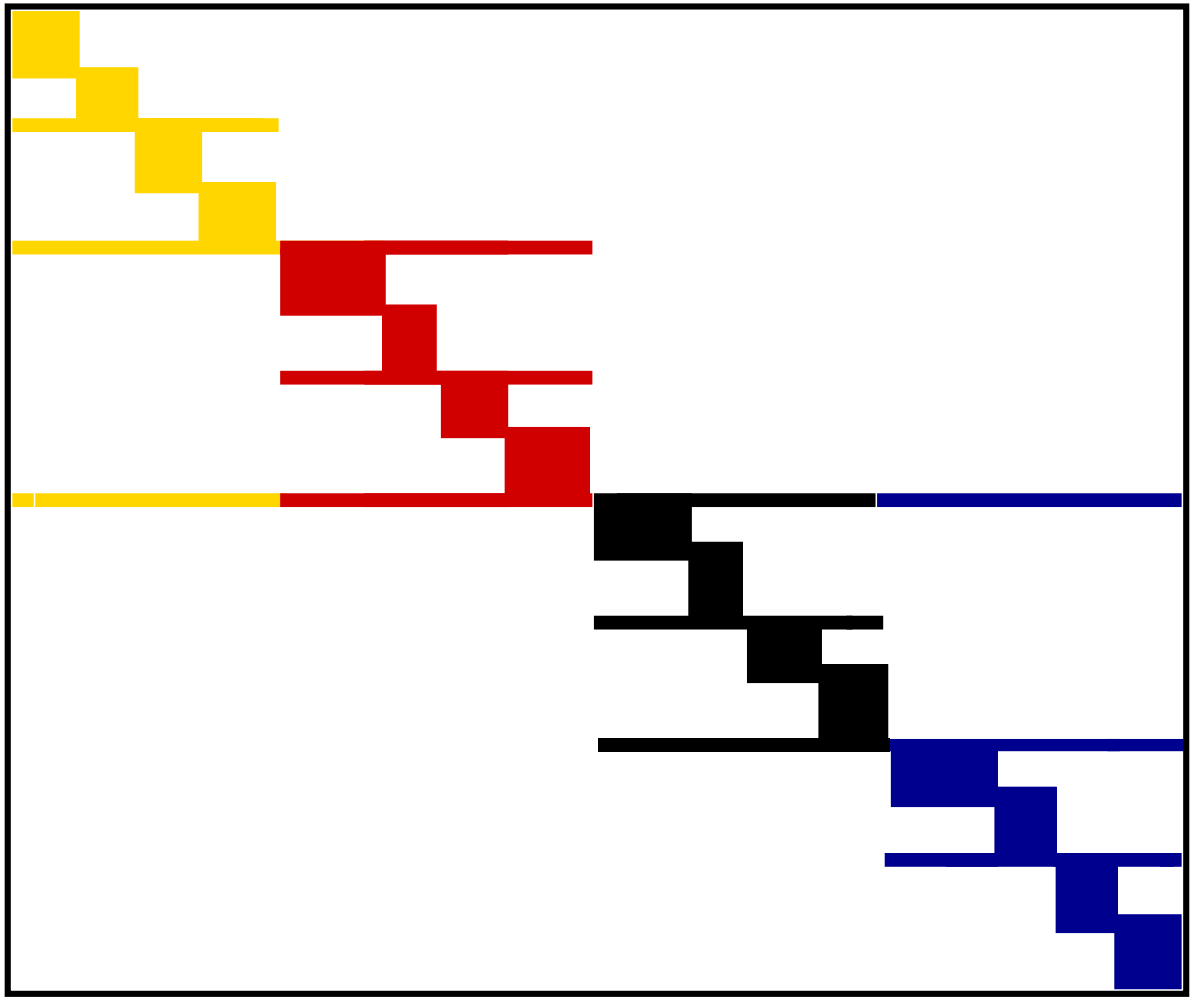
This improves vector element reuse by changing the data structure,

but why not also change the input matrix?

- Assume zig-zag CRS ordering
- Allow only row and column permutations

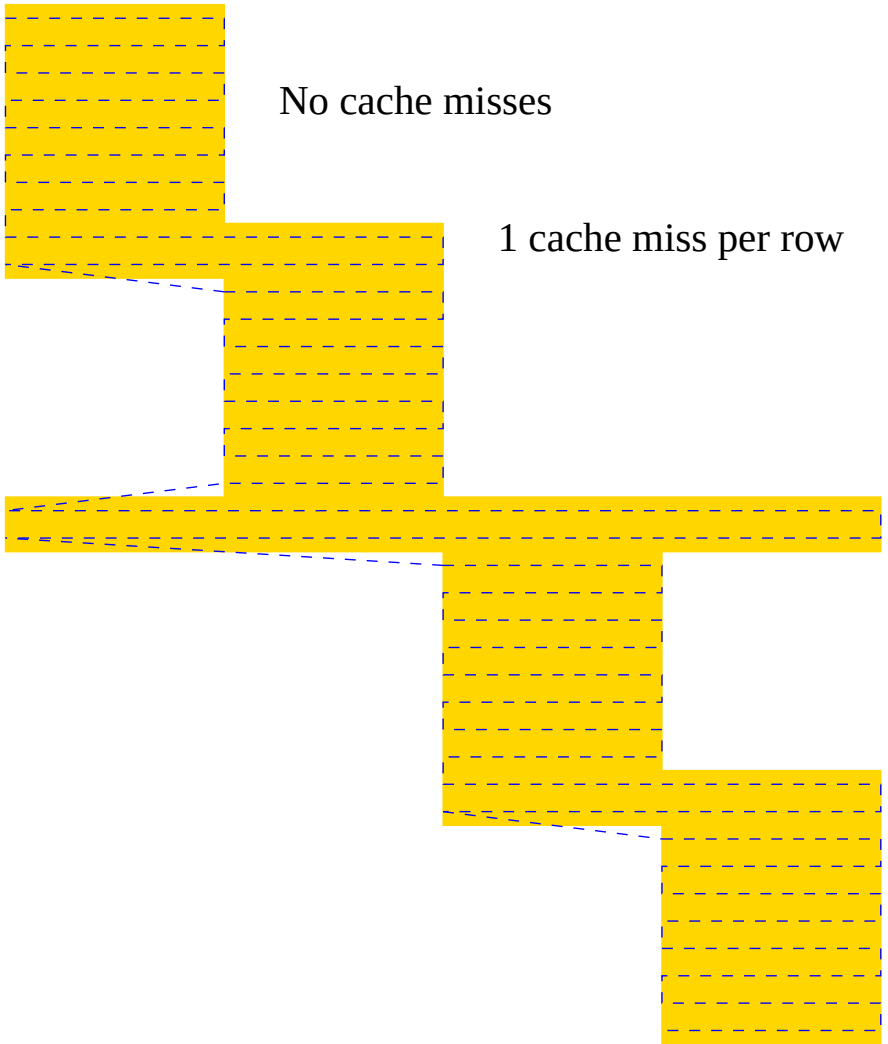
SBD-based cache-oblivious SpMV

Separated Block Diagonal form:



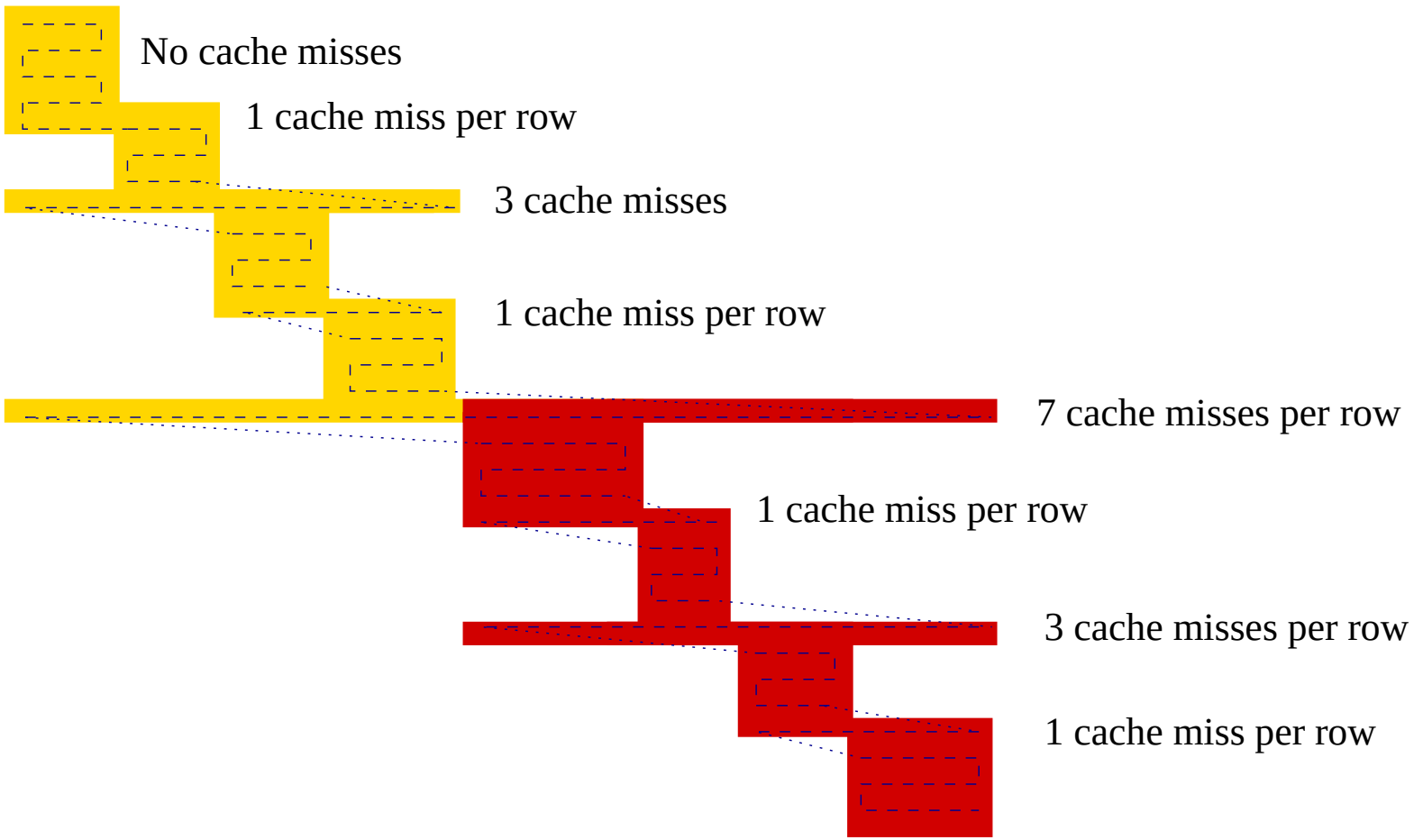
SBD-based cache-oblivious SpMV

Separated Block Diagonal form:



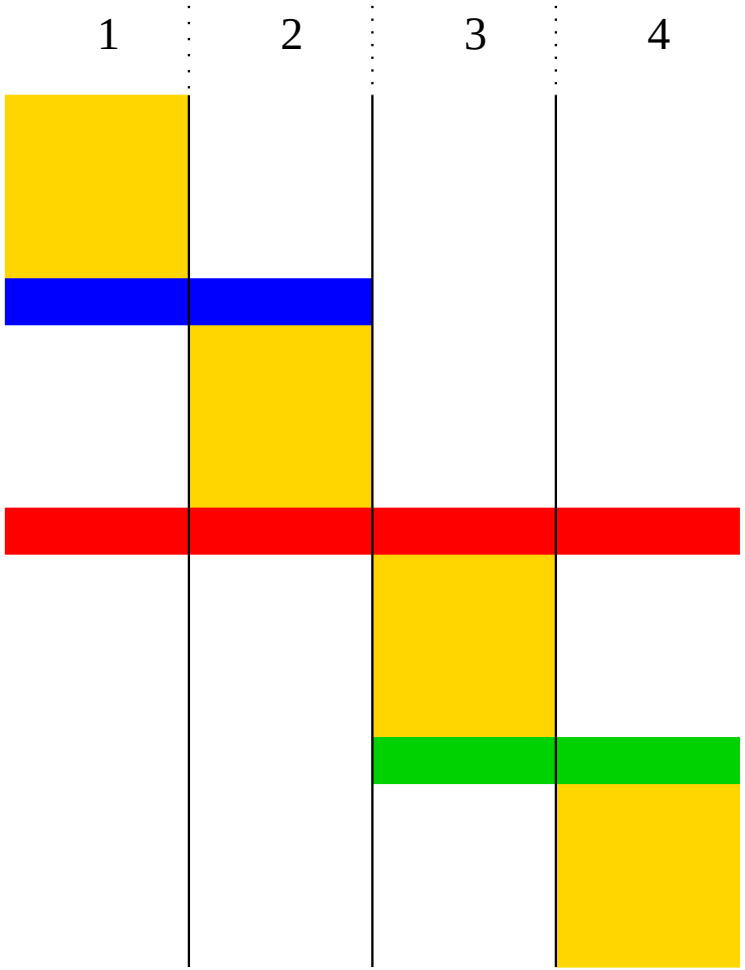
SBD-based cache-oblivious SpMV

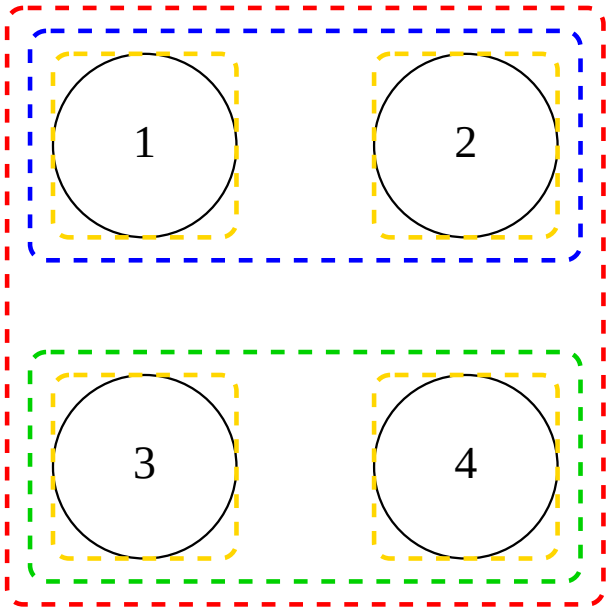
Separated Block Diagonal form:



SBD-based cache-oblivious SpMV

Separated Block Diagonal form:

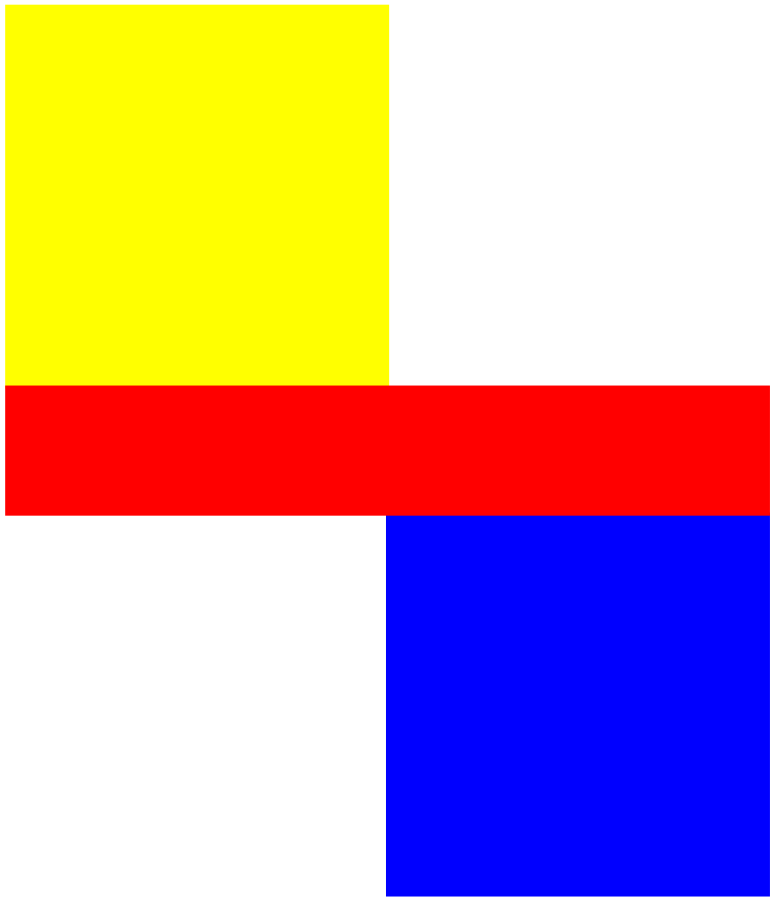




(Upper bound on) cache misses: $\sum_i (\lambda_i - 1)$

SBD-based cache-oblivious SpMV

Use the partitioner to do the matrix permutation:



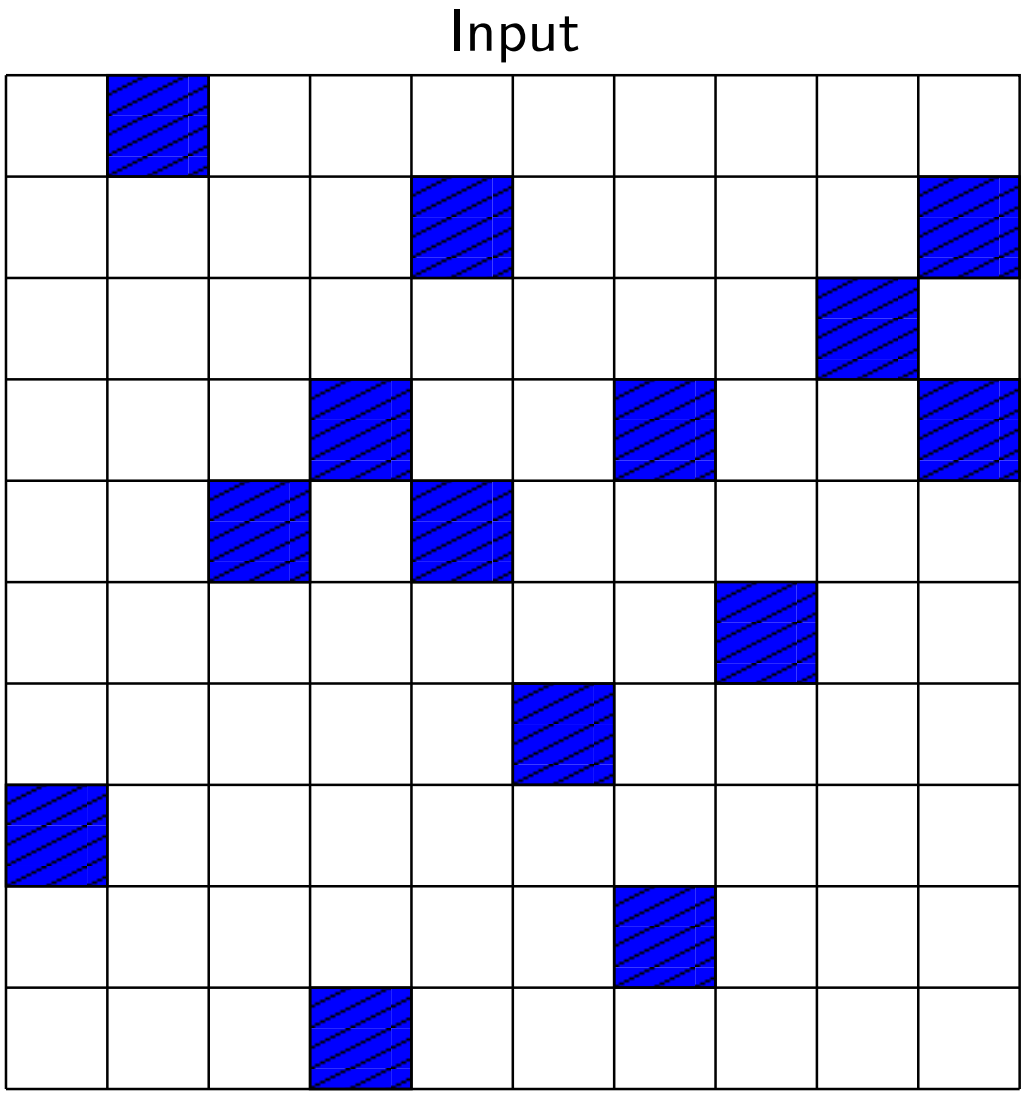
$\mathcal{N}_{-}^{\text{row}}$

$\mathcal{N}_{c}^{\text{row}}$

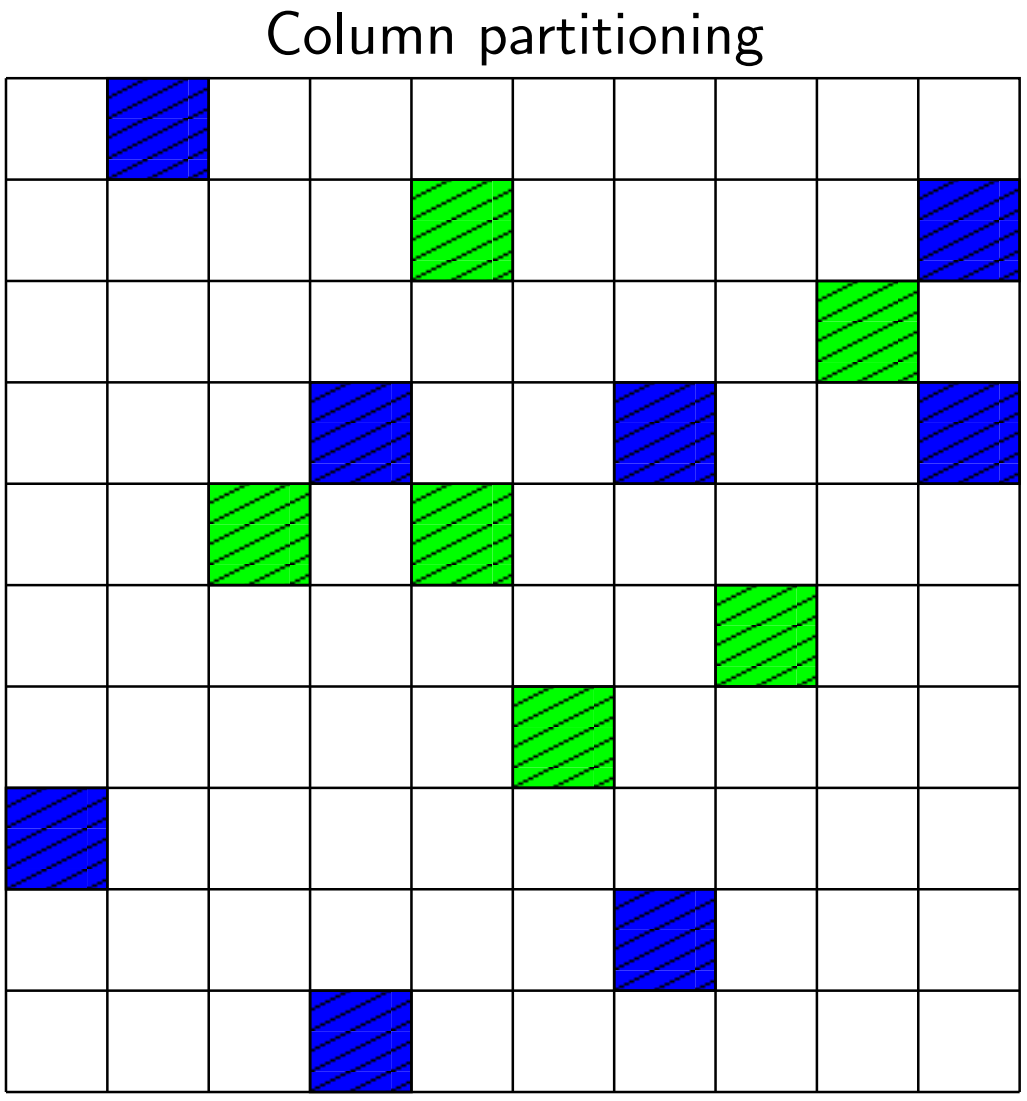
$\mathcal{N}_{+}^{\text{row}}$

$\mathcal{V}_1$ $\mathcal{V}_2$

Permuting to SBD form

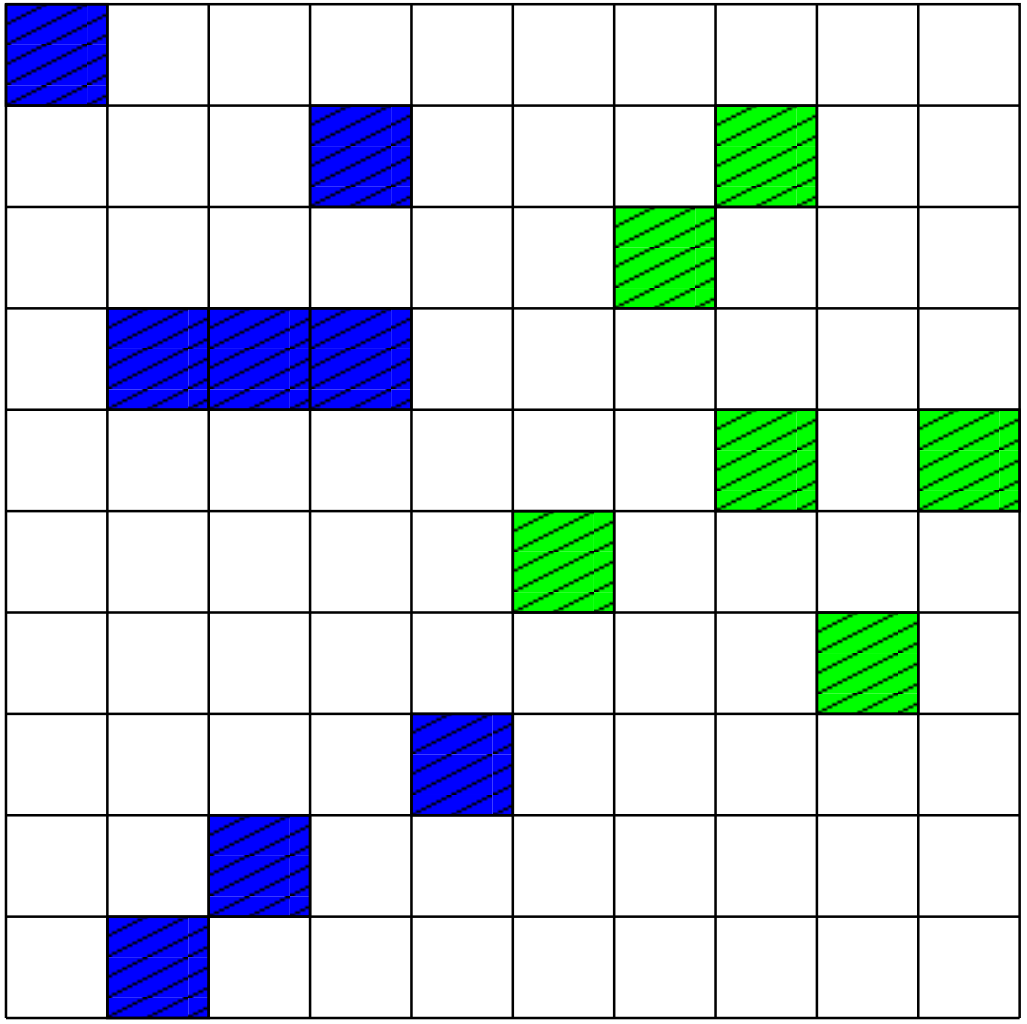


Permuting to SBD form



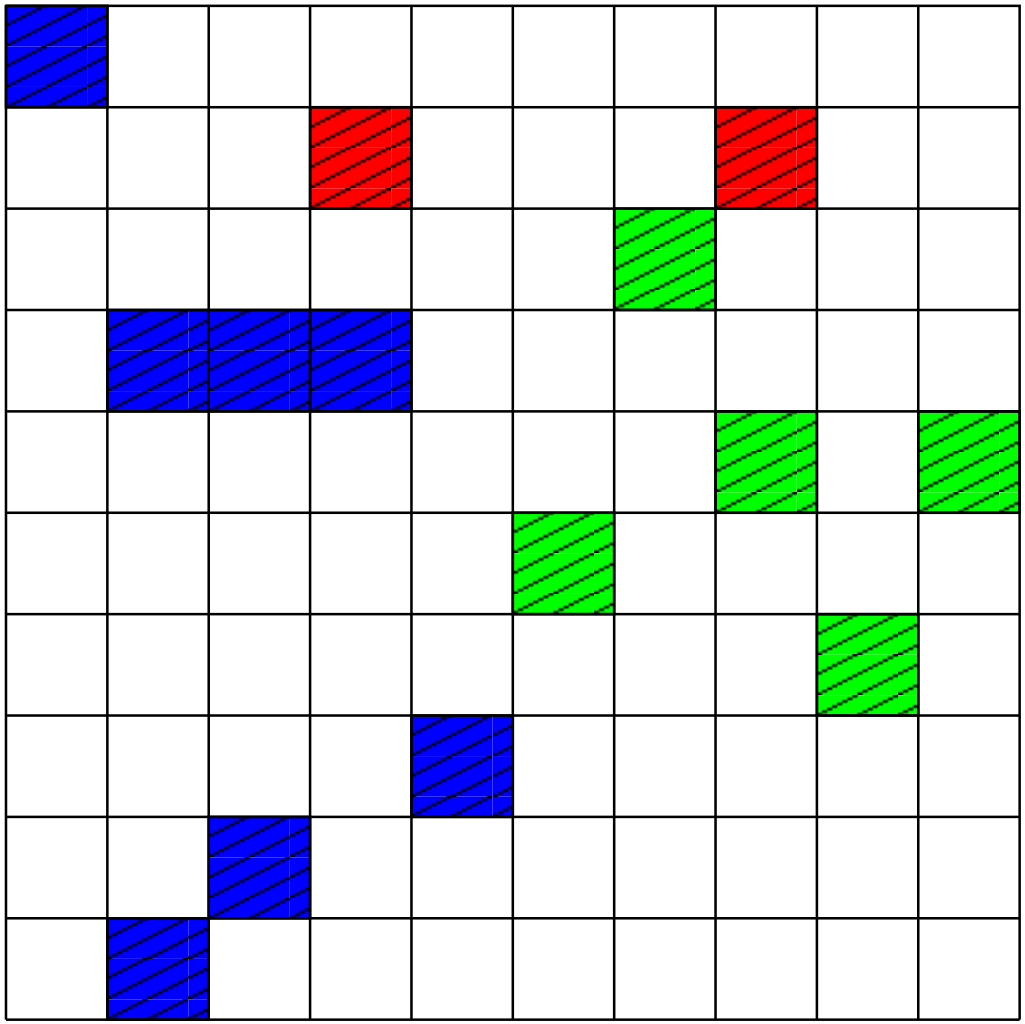
Permuting to SBD form

Column permutation

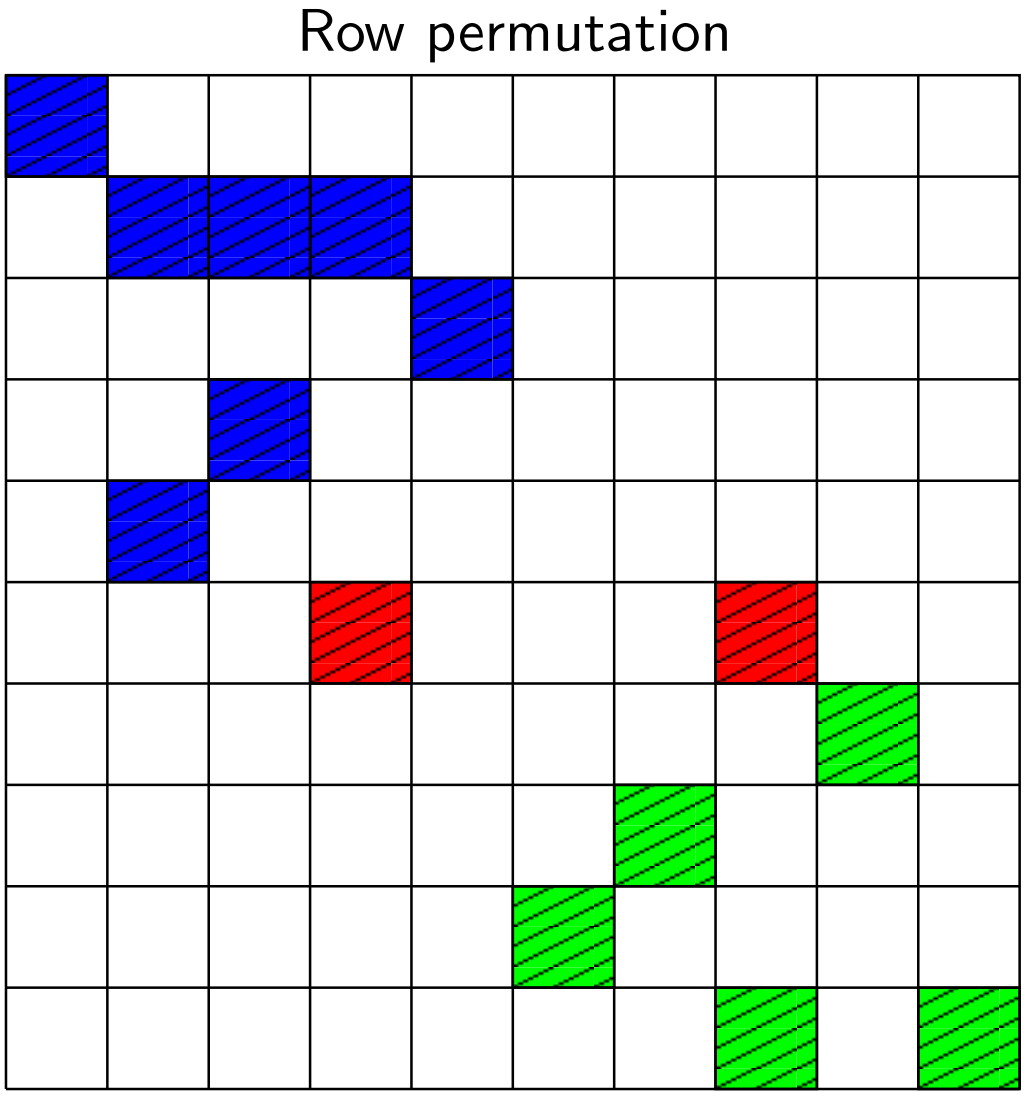


Permuting to SBD form

Mixed row detection

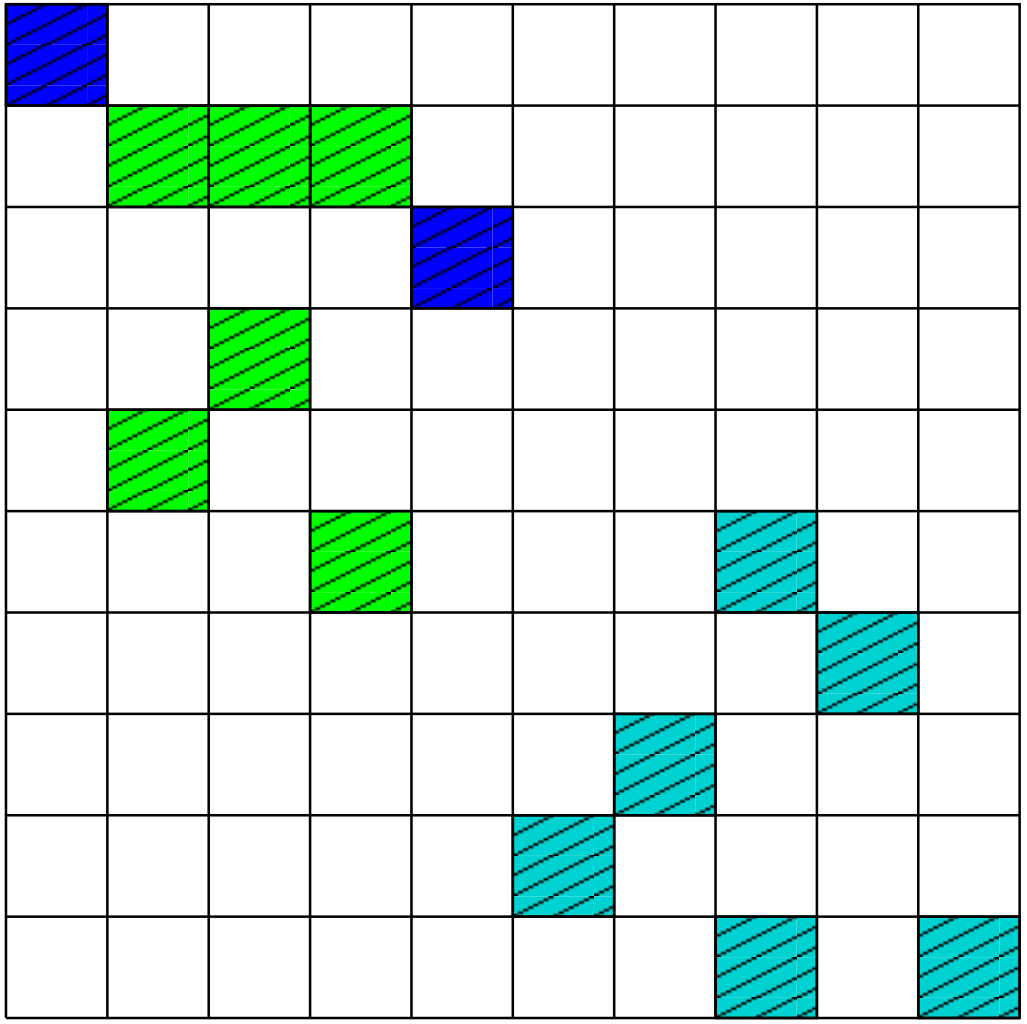


Permuting to SBD form



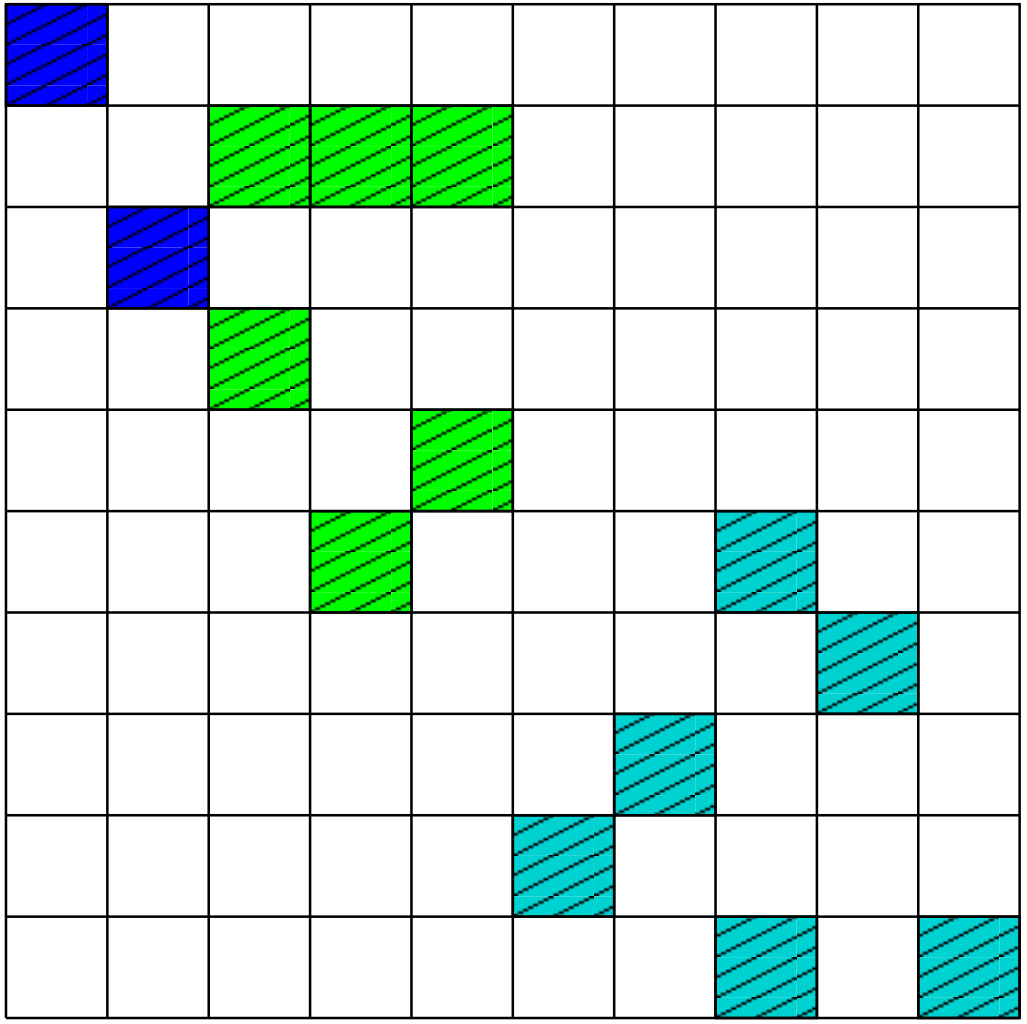
Permuting to SBD form

Column subpartitioning



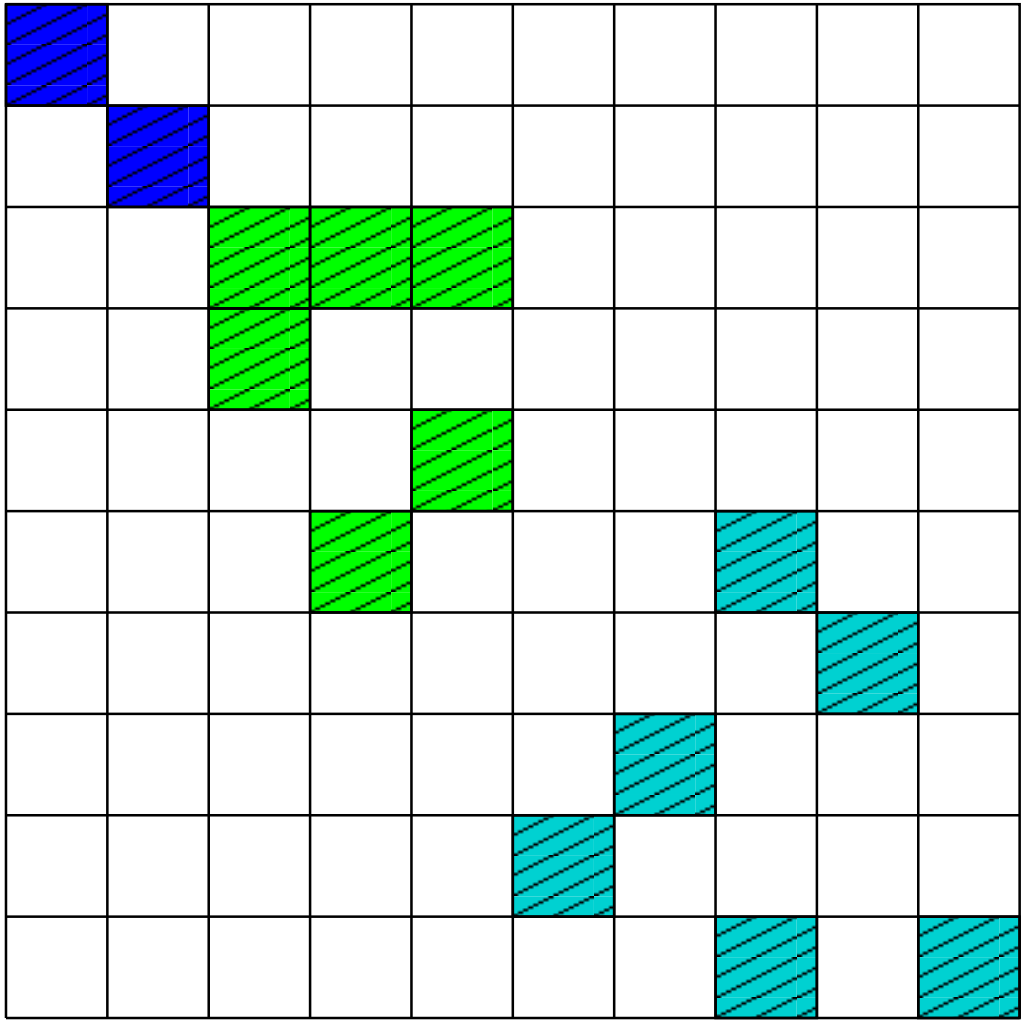
Permuting to SBD form

Column permutation



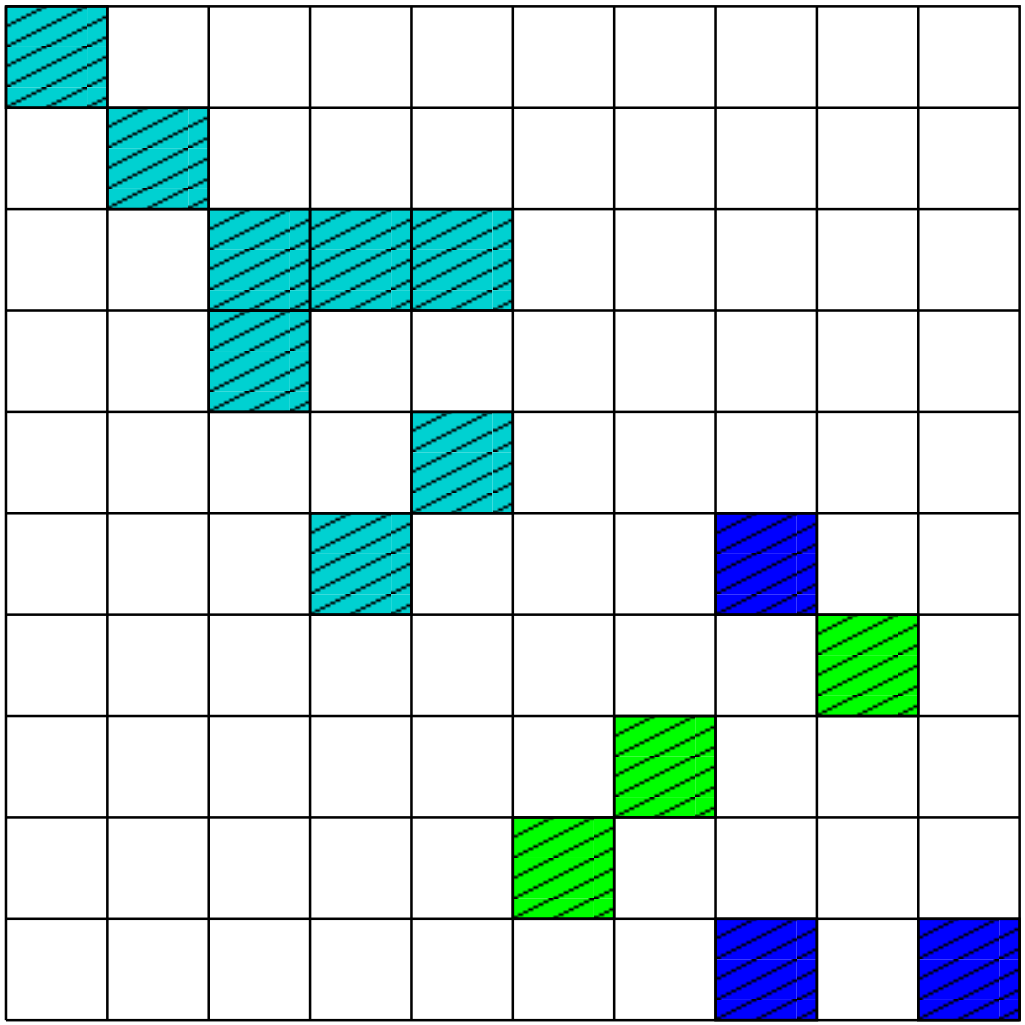
Permuting to SBD form

No mixed rows - row permutation



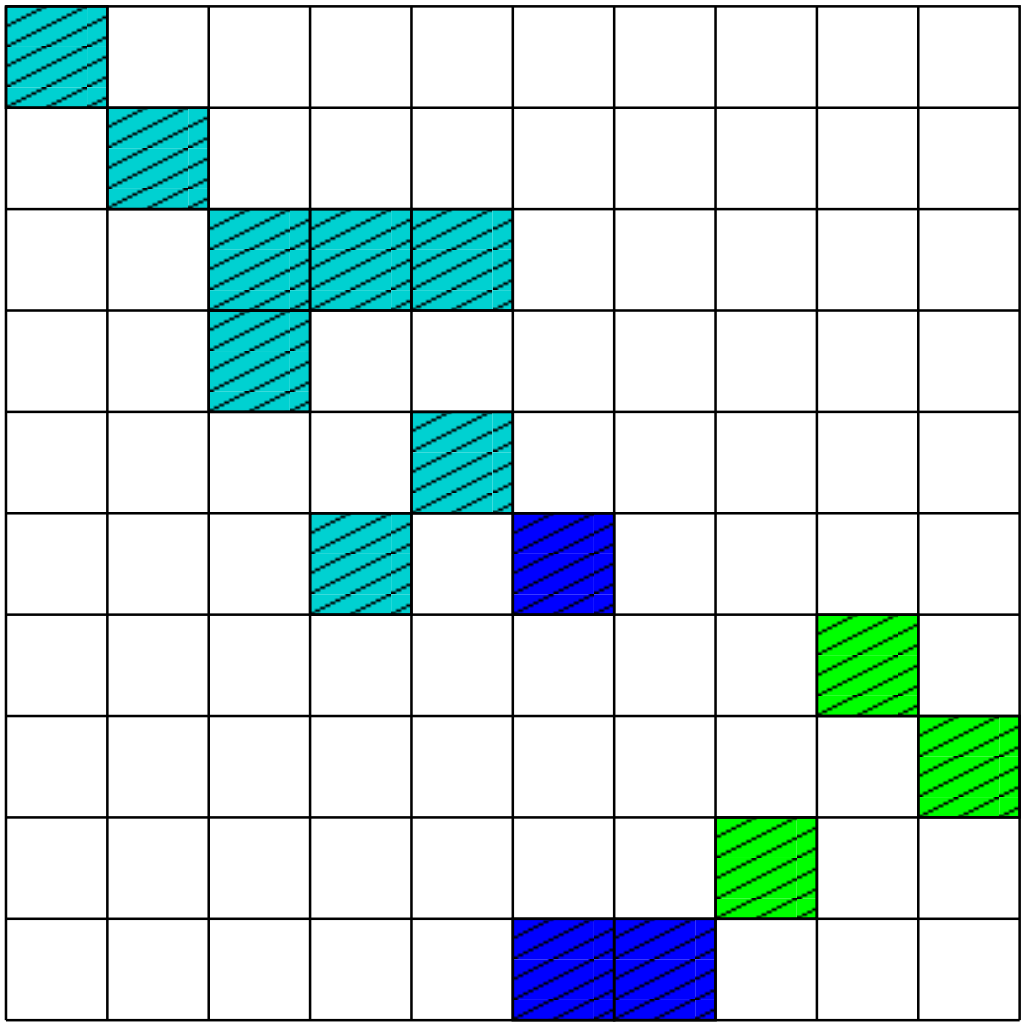
Permuting to SBD form

Continued



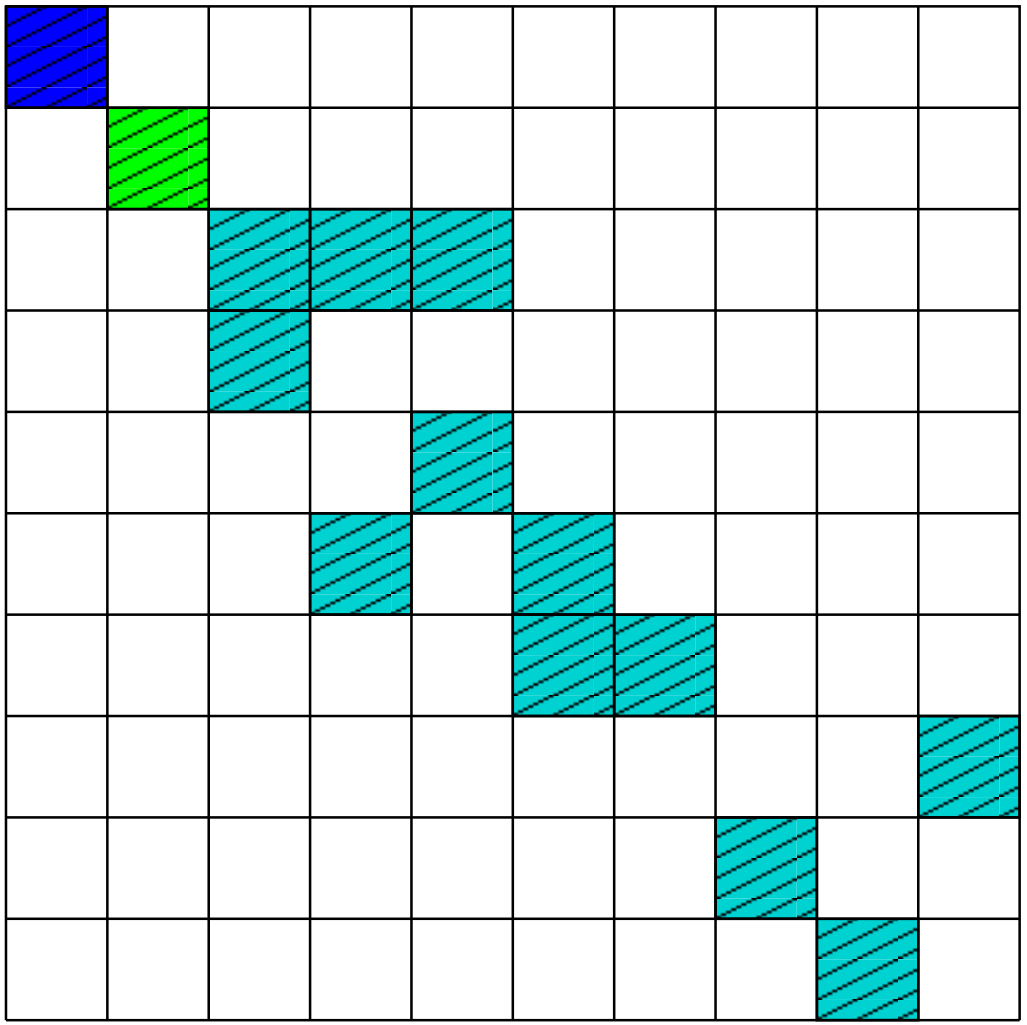
Permuting to SBD form

Continued



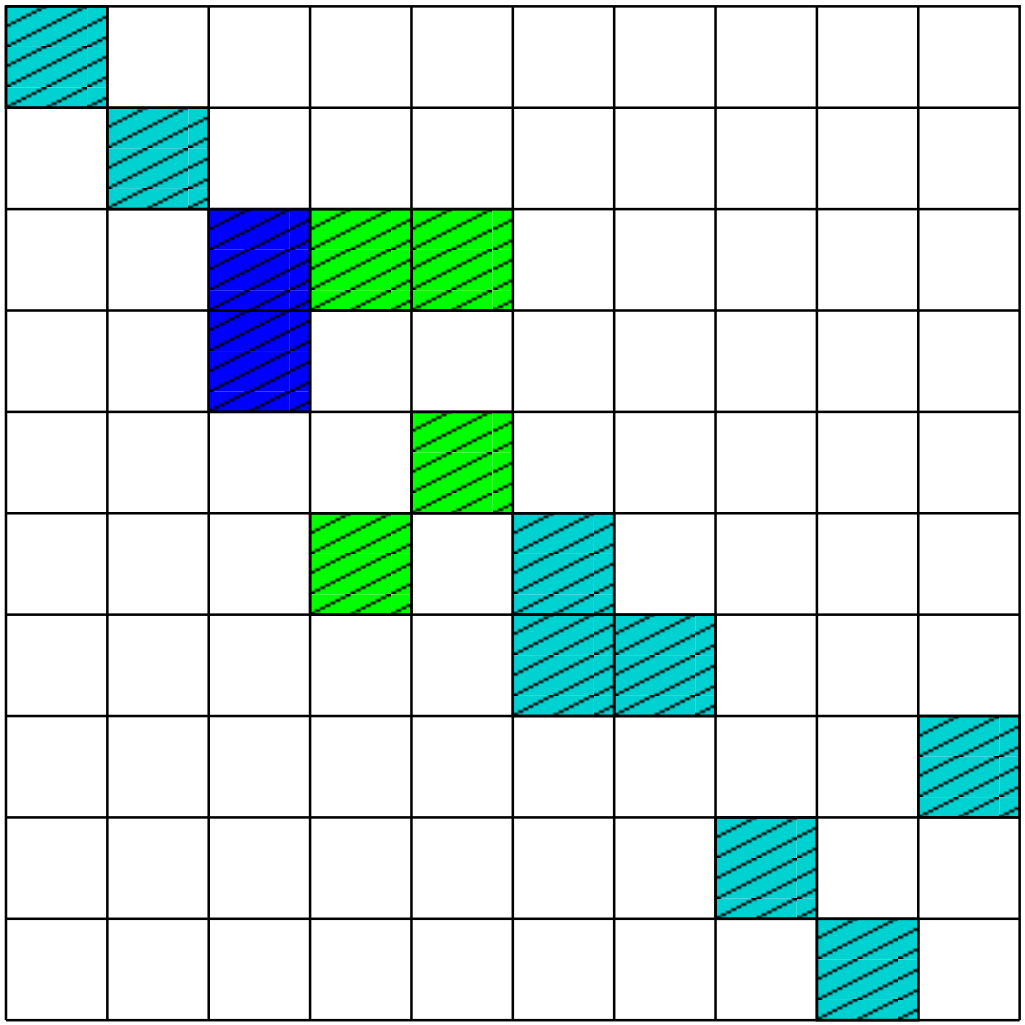
Permuting to SBD form

Continued



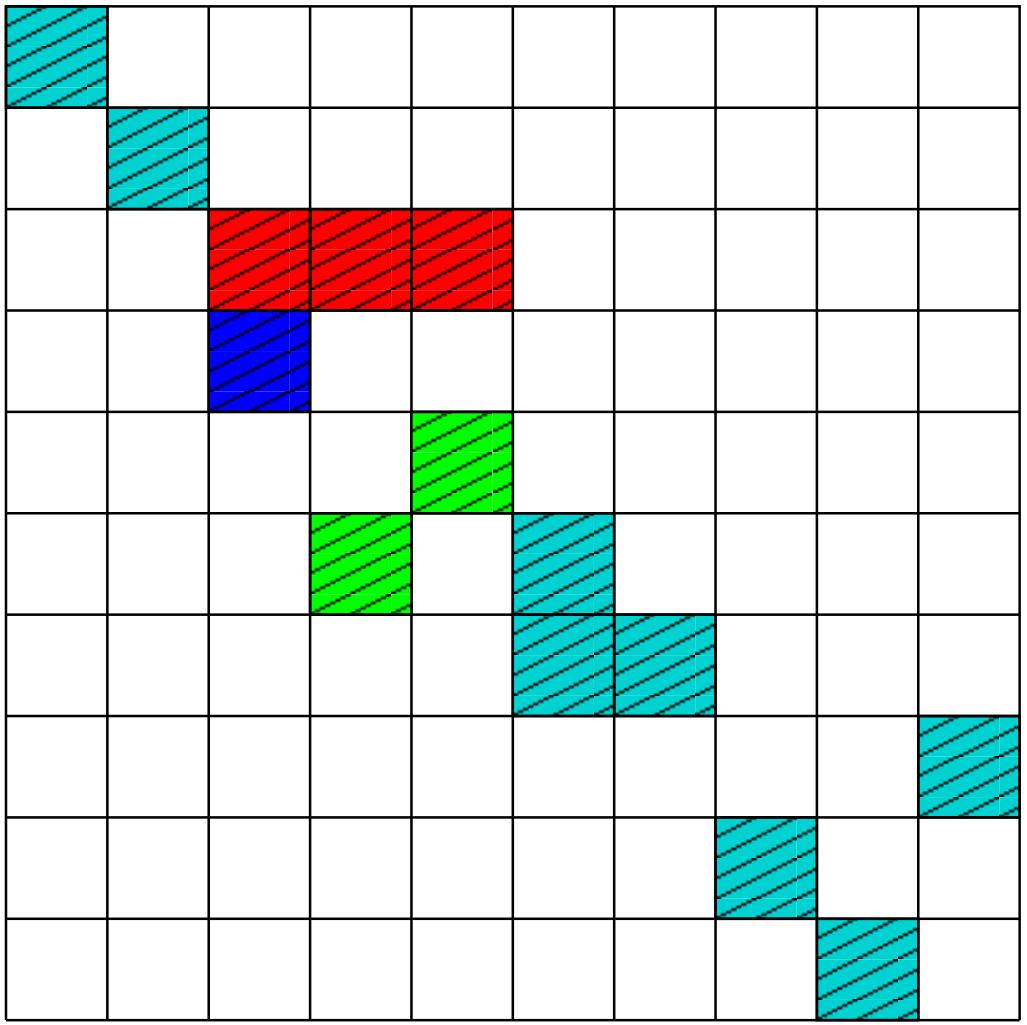
Permuting to SBD form

Continued



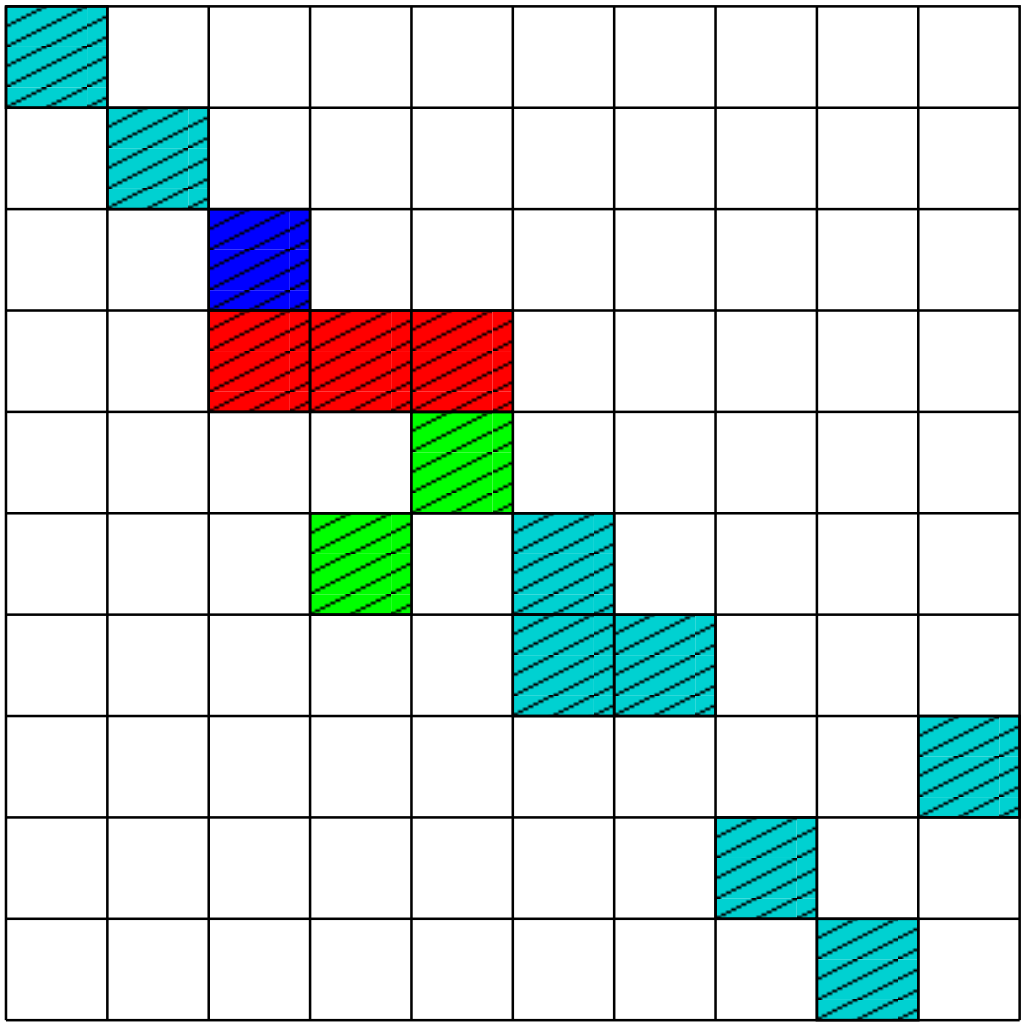
Permuting to SBD form

Continued



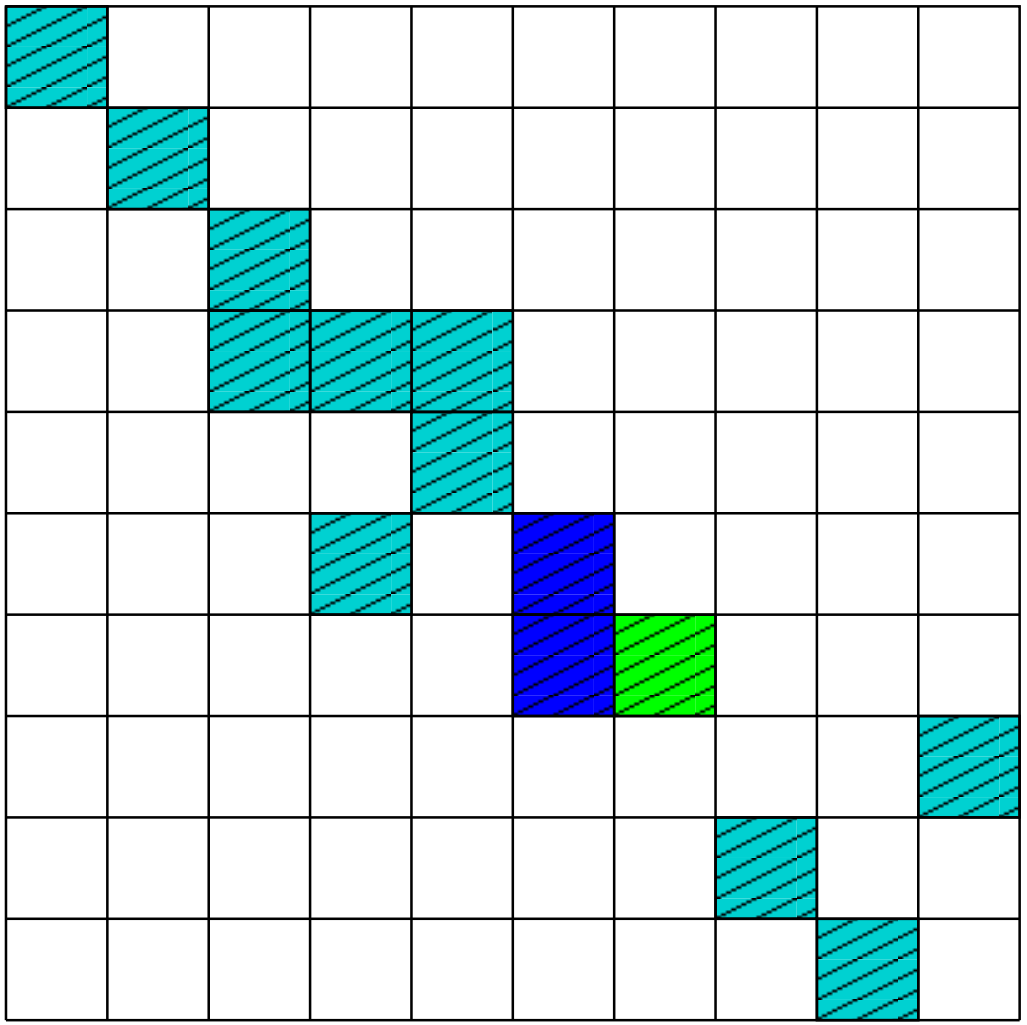
Permuting to SBD form

Continued



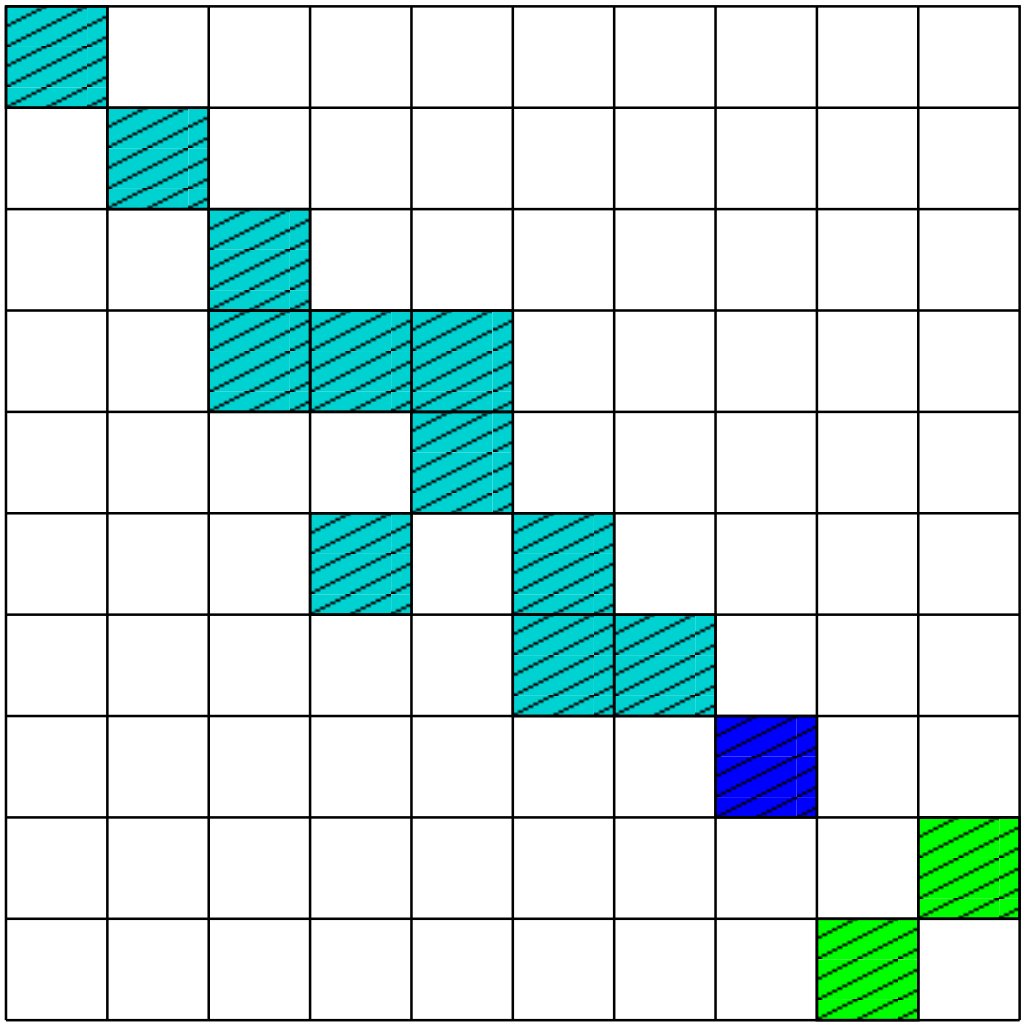
Permuting to SBD form

Continued



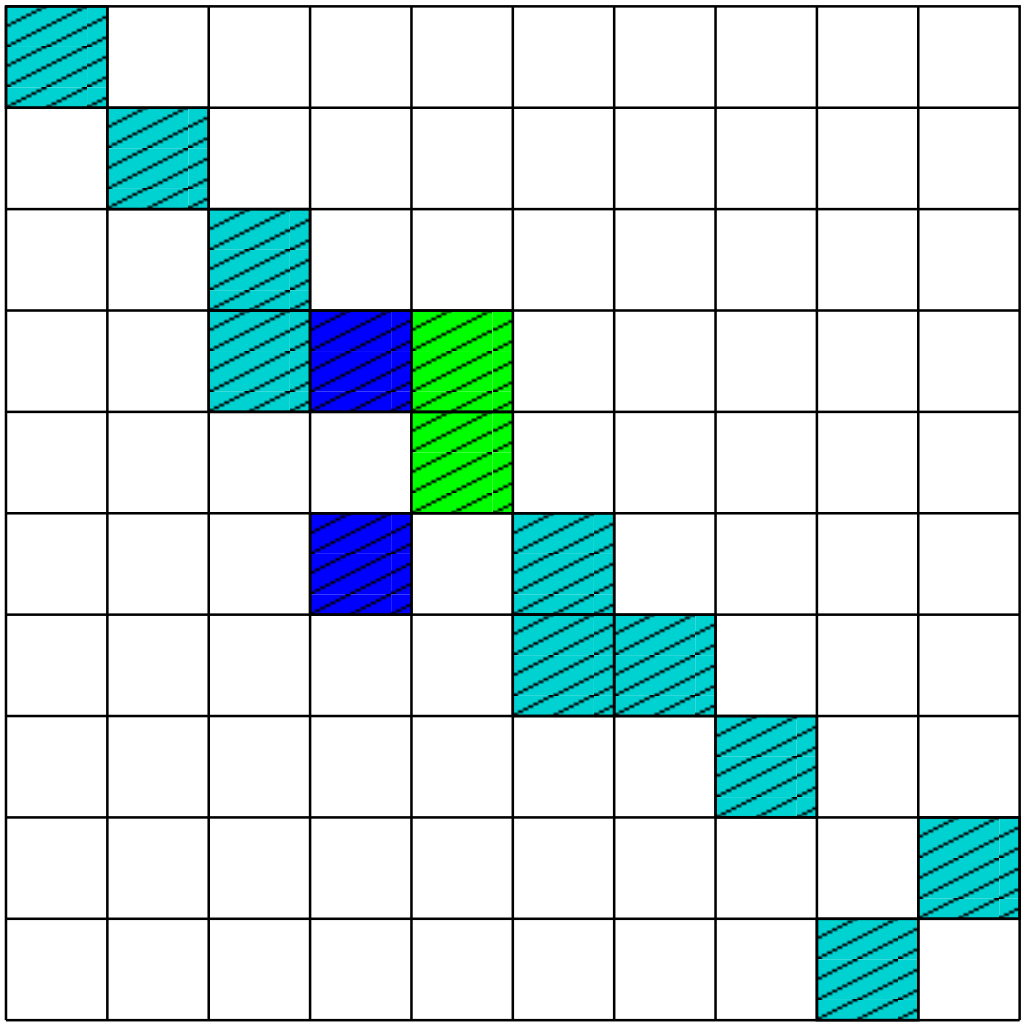
Permuting to SBD form

Continued



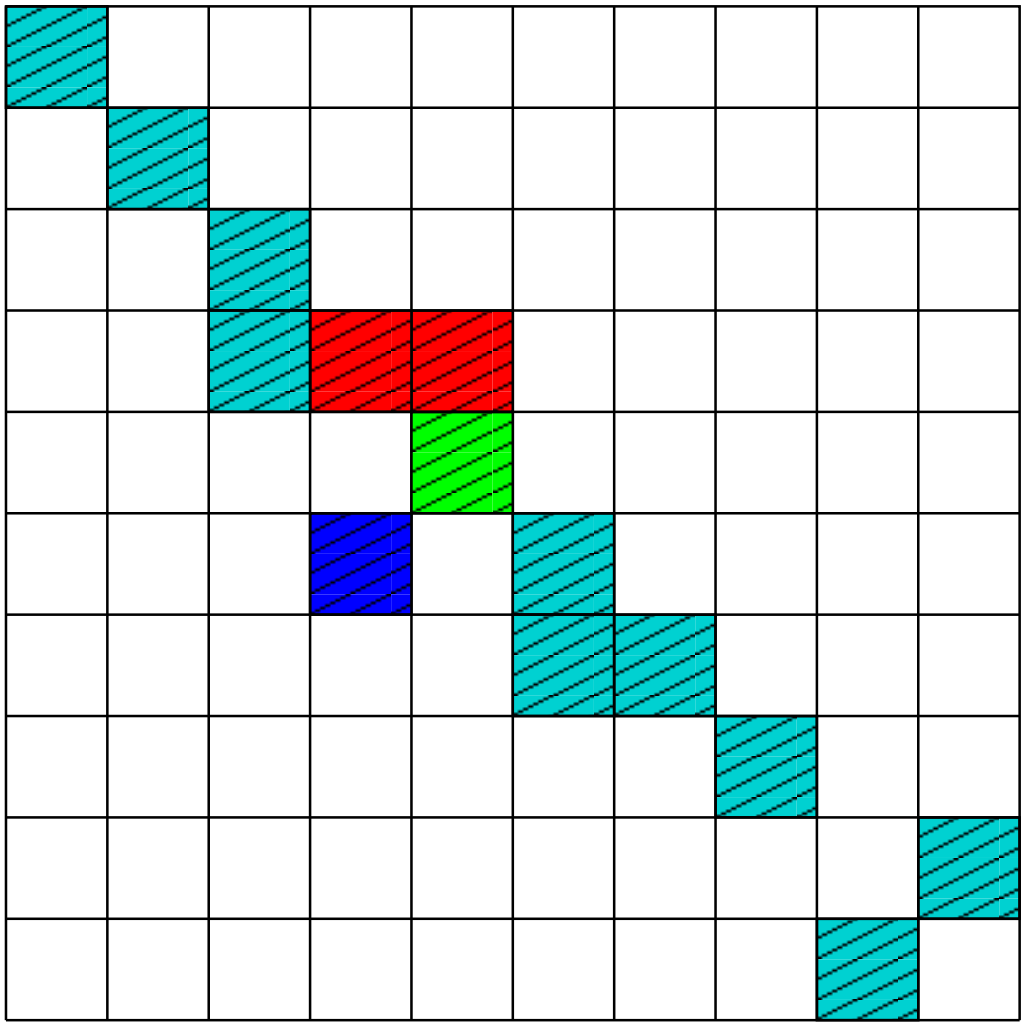
Permuting to SBD form

Continued



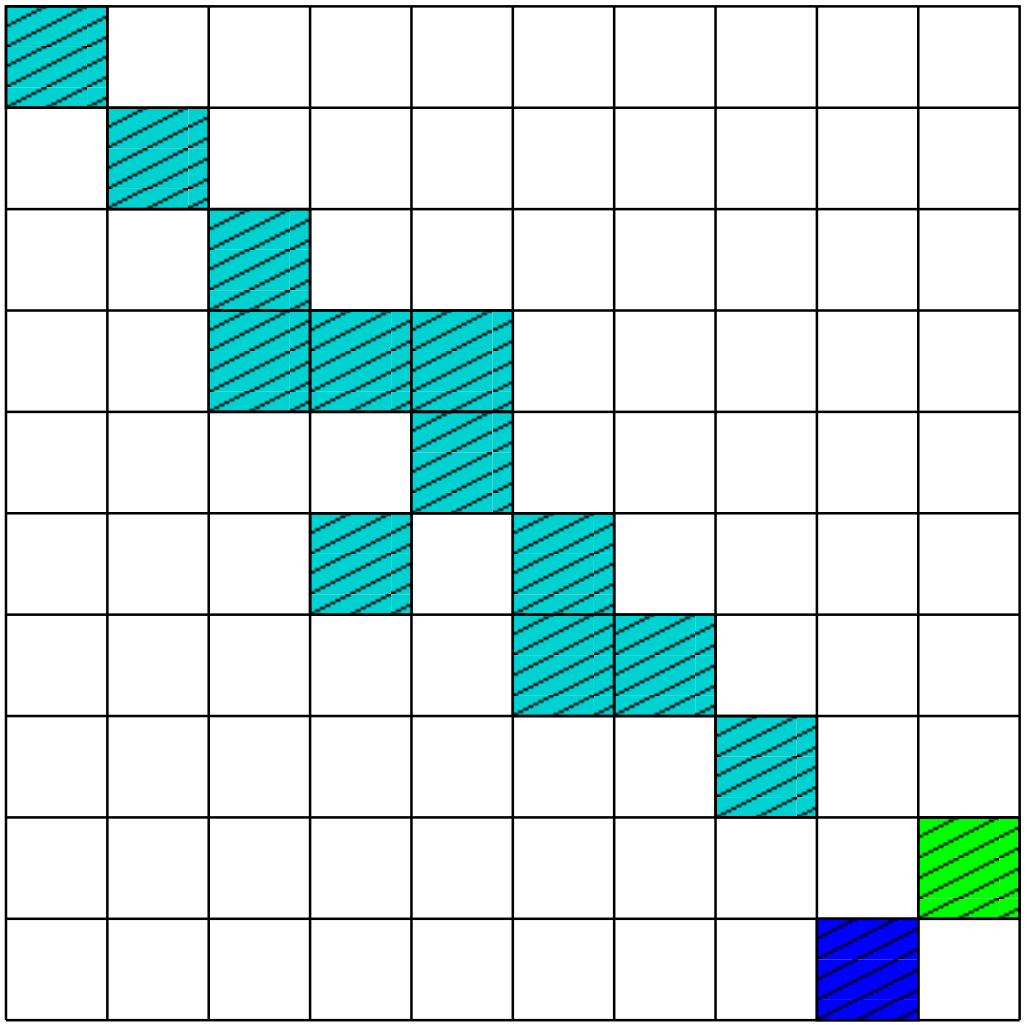
Permuting to SBD form

Continued



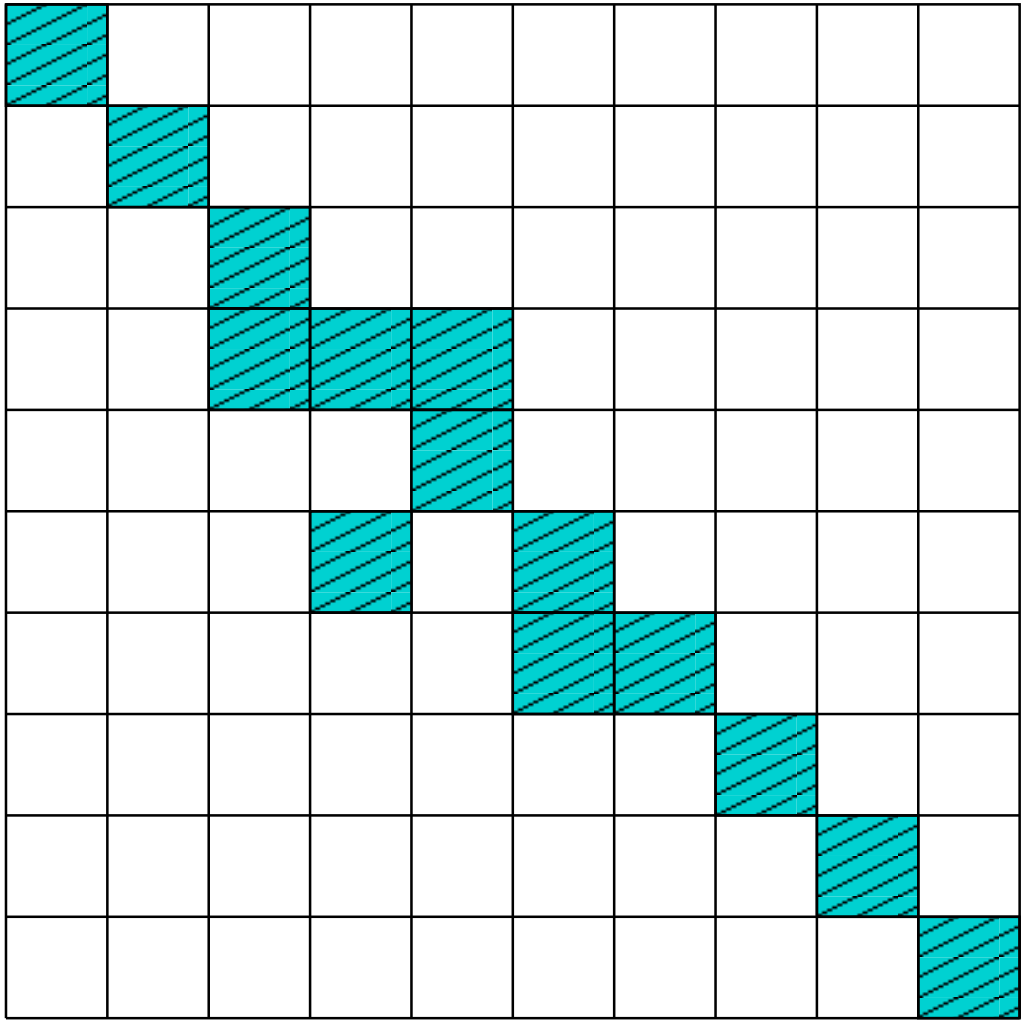
Permuting to SBD form

Continued



Permuting to SBD form

Continued



Reordering parameters

$$\text{Taking } p = \frac{n}{S},$$

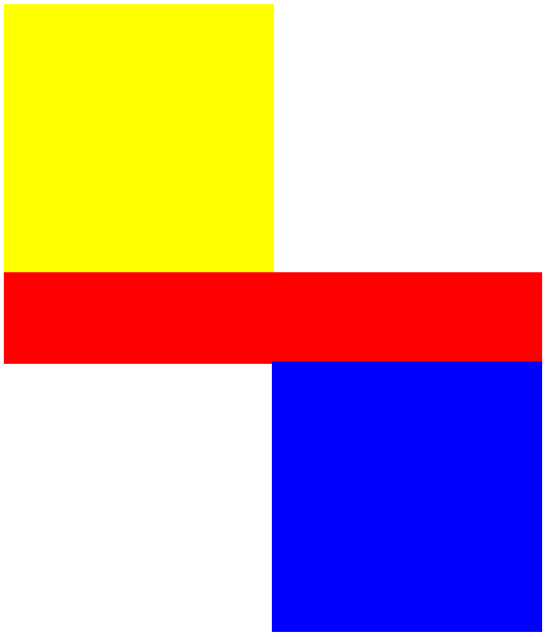
the number of cache misses is strictly bounded by

$$\sum_{i:\ n_i \in \mathcal{N}} (\lambda_i - 1);$$

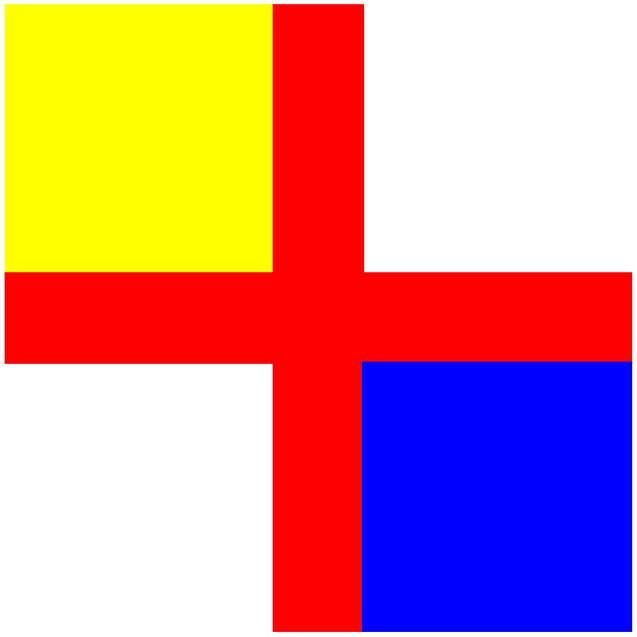
taking $p \to \infty$ *yields a* cache-oblivious *method with the same bound.*

- Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SIAM Journal on Scientific Computing, 2009

Two-dimensional SBD (doubly separated block diagonal)



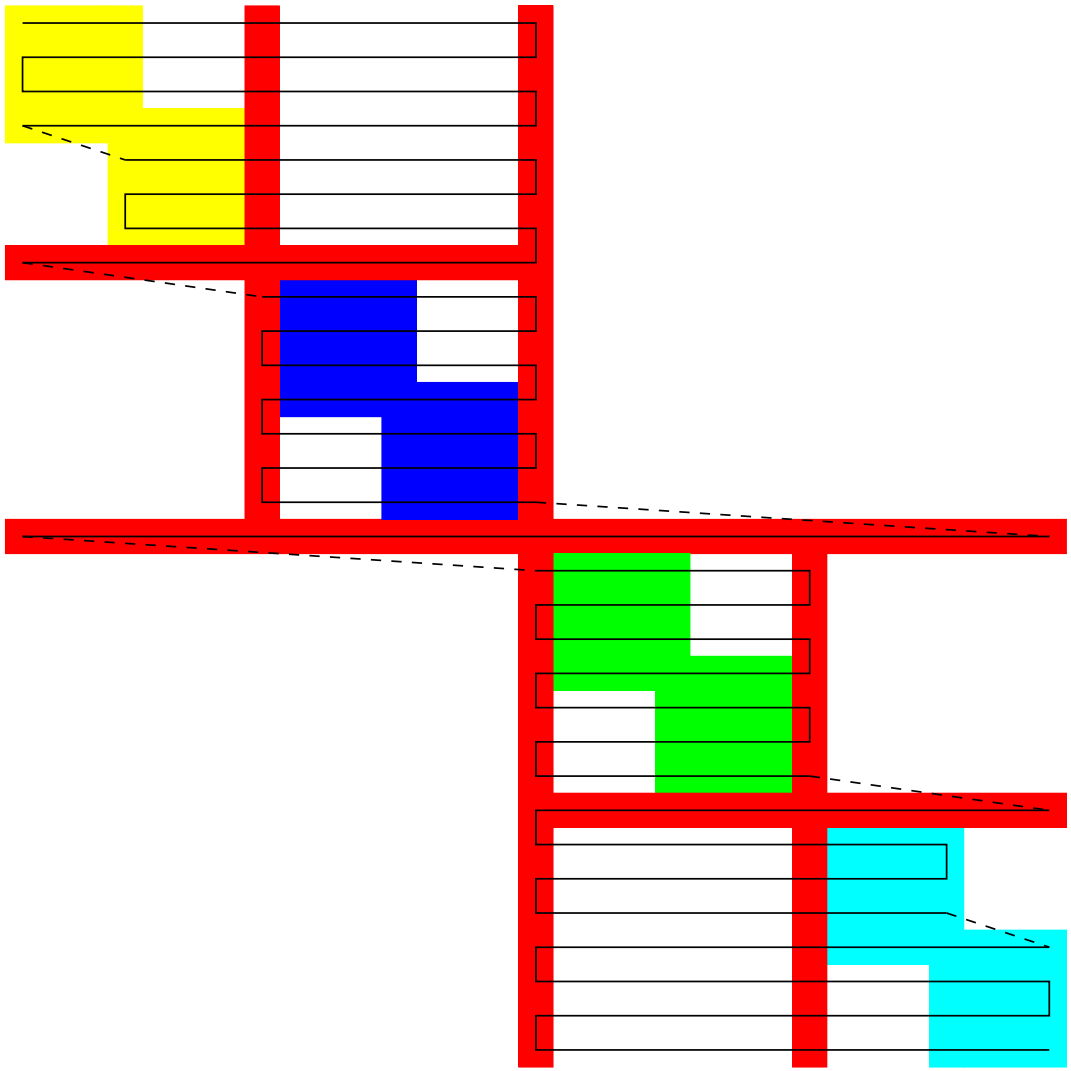
1D



2D

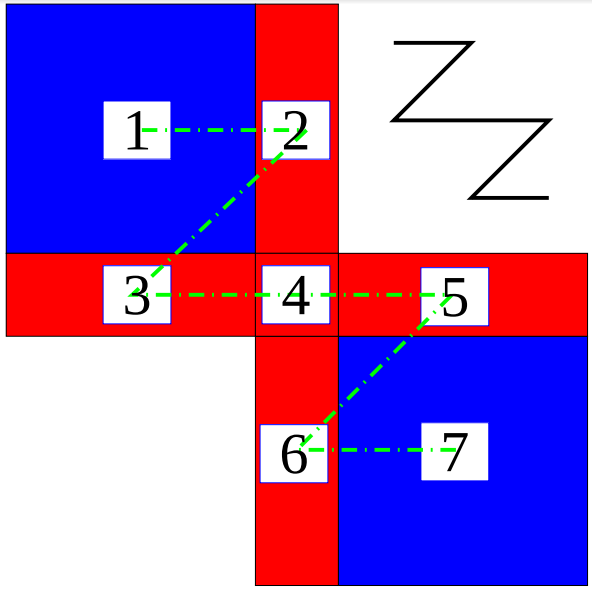
Yzelman and Bisseling, *Two-dimensional cache-oblivious sparse matrix–vector multiplication*, Parallel Computing 37(12), pp. 806–819, 2011

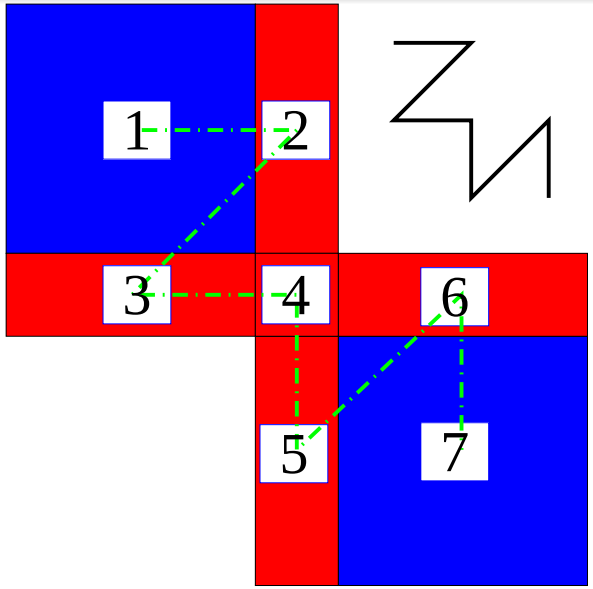
Two-dimensional SBD (doubly separated block diagonal)

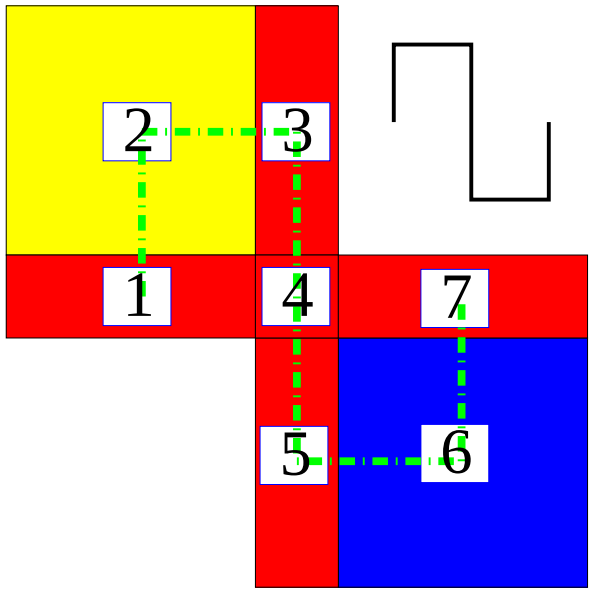


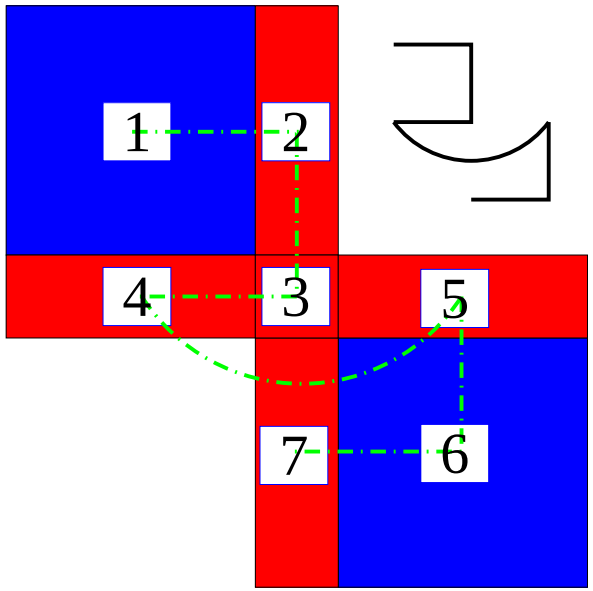
Zig-zag CRS is not suitable for handling 2D SBD!

Two-dimensional SBD; block ordering

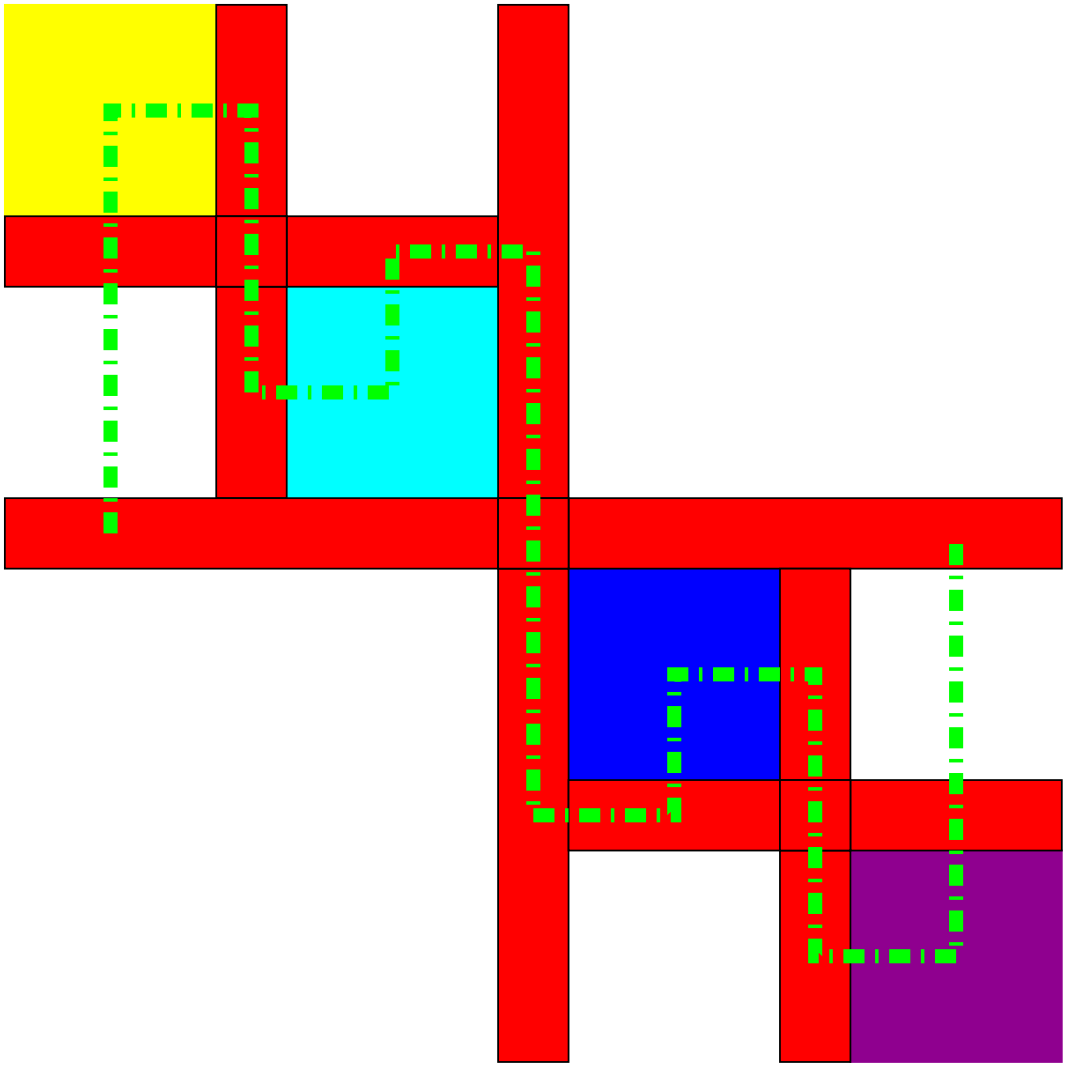








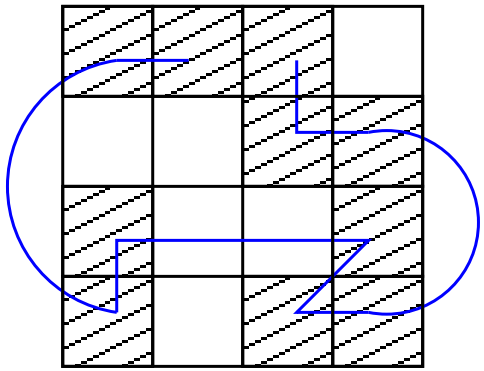
Two-dimensional SBD; block ordering



Bi-directional Incremental CRS (BICRS)

Storage method for *blocked 2D SBD*:

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$



Stored as:

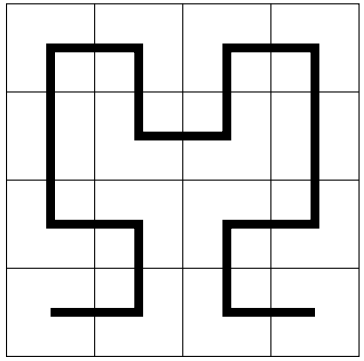
nzs:	[3 2 3 1 1 2 1 7 4 1]
col_increment:	[2 4 1 4 -1 5 -3 4 4 1] ,
row_increment:	[0 1 2 -1 1 -3]

$2nnz + (row\_jumps + 1)$ accesses per nonzero

BICRS and fractal storage

However, BICRS also enables efficient *Hilbert SpMV multiplication*:

$$A = \begin{pmatrix} 4 & 1 & 0 & 2 \\ 0 & 2 & 0 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 0 \end{pmatrix}$$



Stored as:

nzs: [7 1 4 1 2 2 3 2 1]
i : [3 2 0 0 1 0 1 2 3] , **3nnz** accesses per nonzero
j : [0 0 0 1 1 3 3 3 2]

Yzelman and Bisseling, *A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve*, in M. Günther et al. (eds.), *Progress in Industrial Mathematics at ECMI 2010*, Mathematics in Industry 17, 2012

Parallel cache-friendly SpMV

1 The (classic) parallel SpMV: partitioning

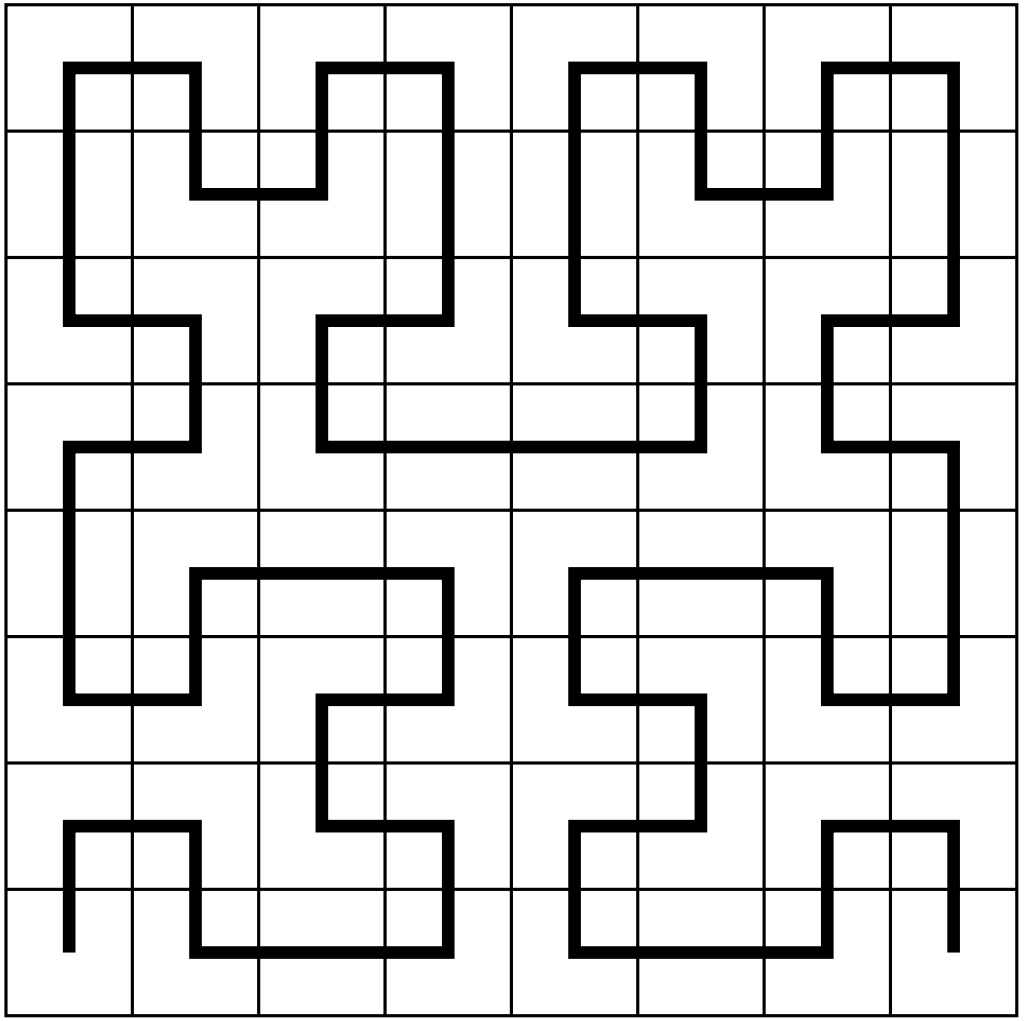
2 Sequential SpMV

3 Parallel cache-friendly SpMV

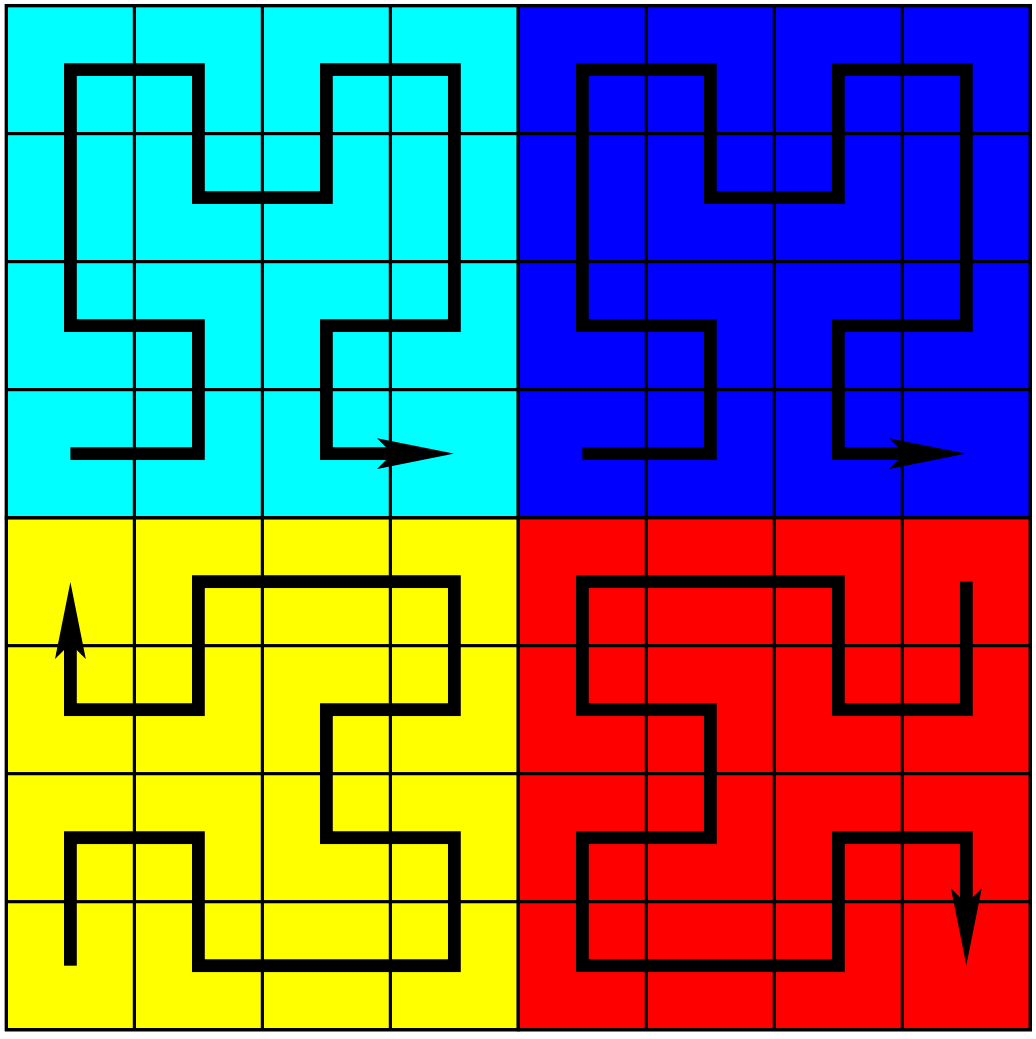
4 Experimental results

Parallel Hilbert-based SpMV

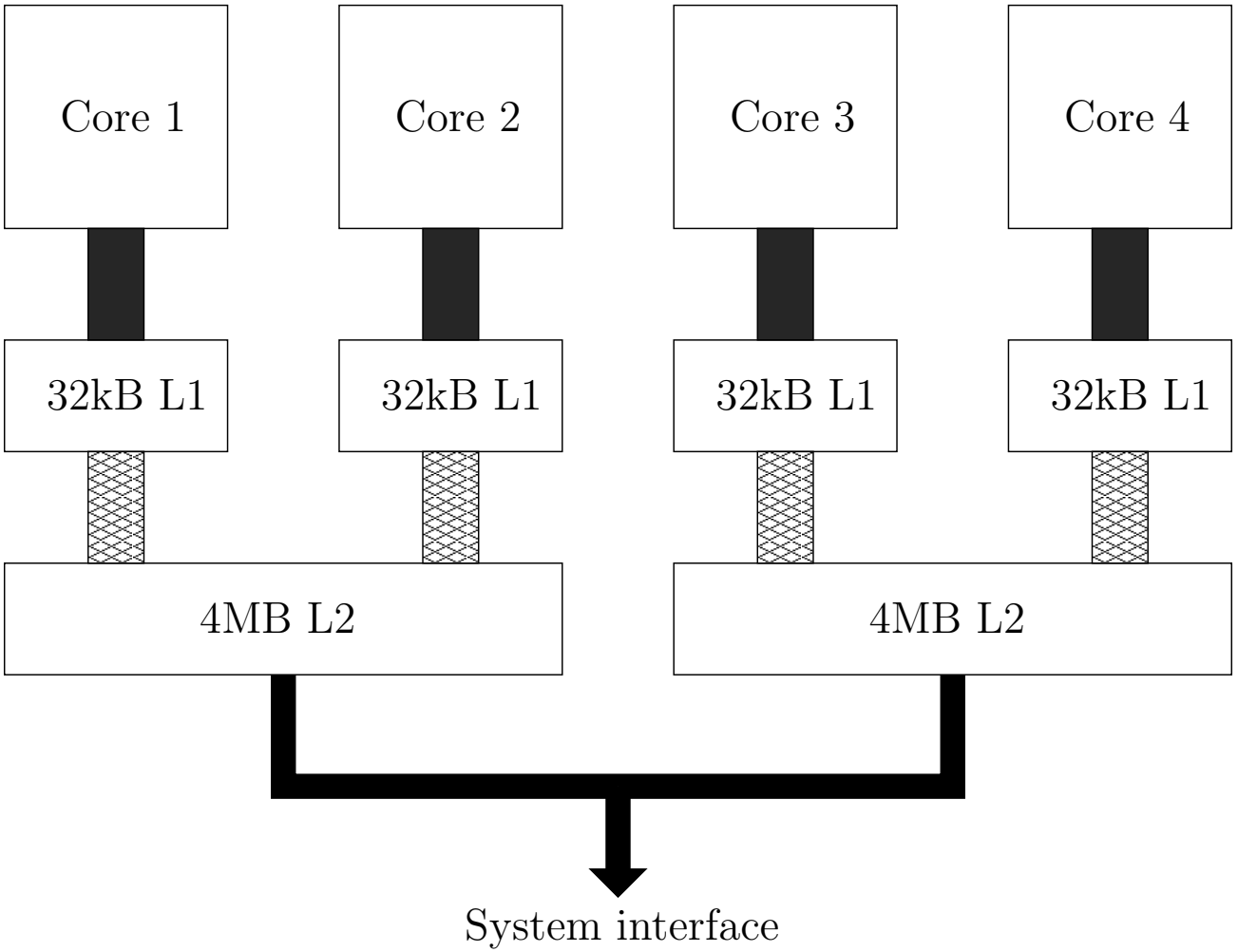
Parallel Hilbert SpMV, a non-distributing (**ND**) scheme:



Parallel Hilbert-based SpMV

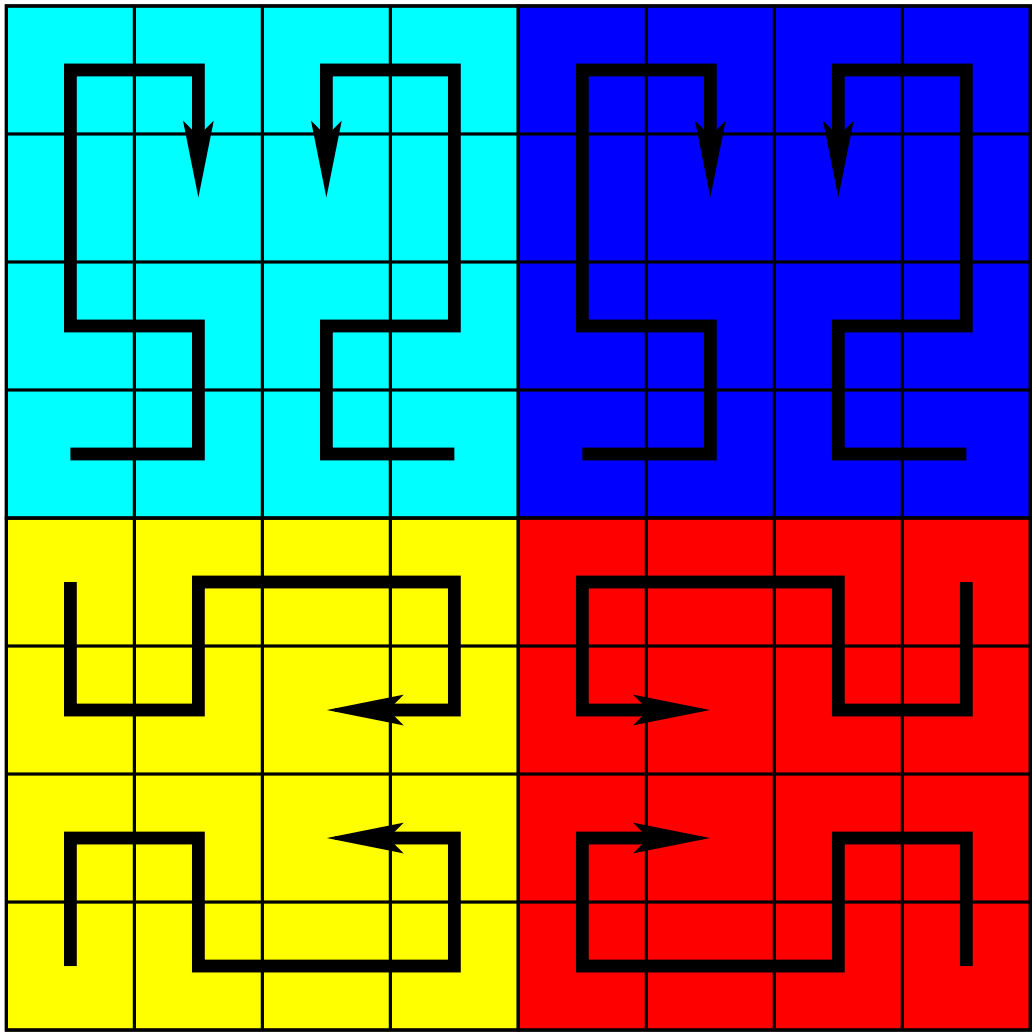


Parallel Hilbert-based SpMV



But what of shared caches, as displayed here? (Intel Q6600)

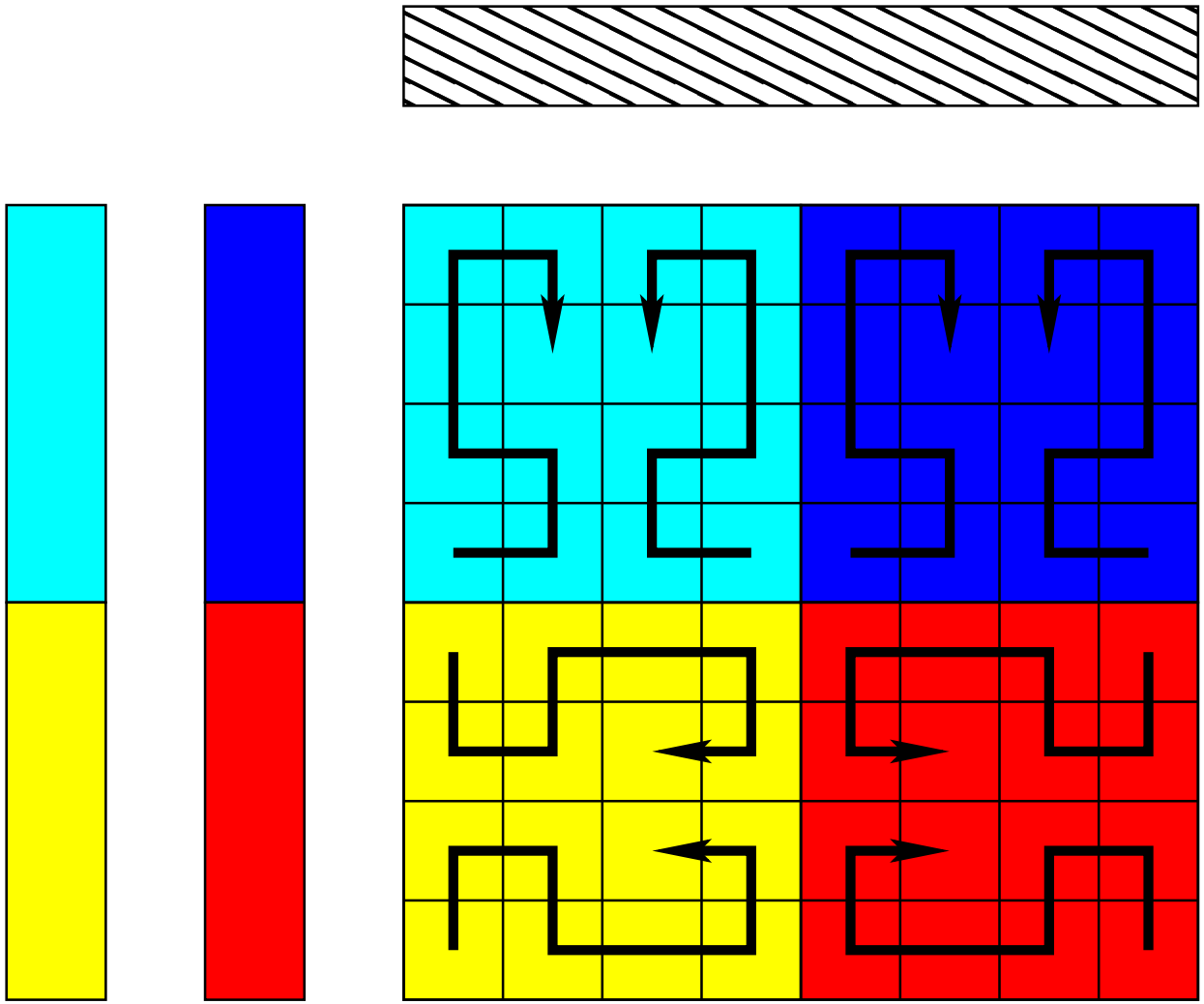
Parallel Hilbert-based SpMV



'Core-oblivious'

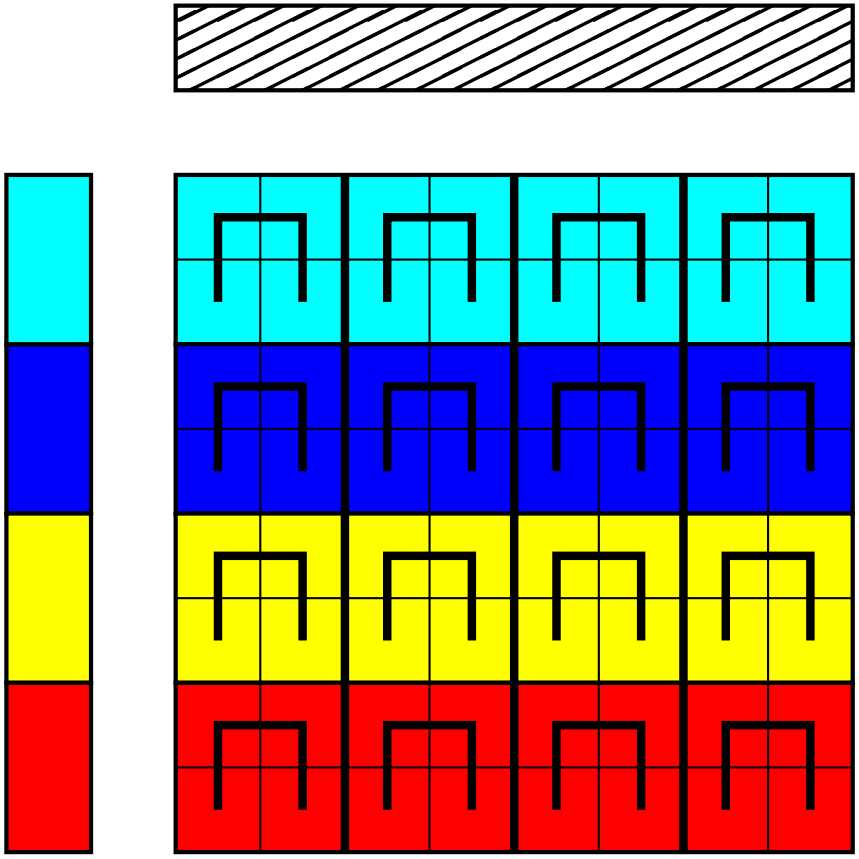
Parallel Hilbert-based SpMV

This method does not scale due to necessary output vector replication:



Parallel Hilbert-based SpMV

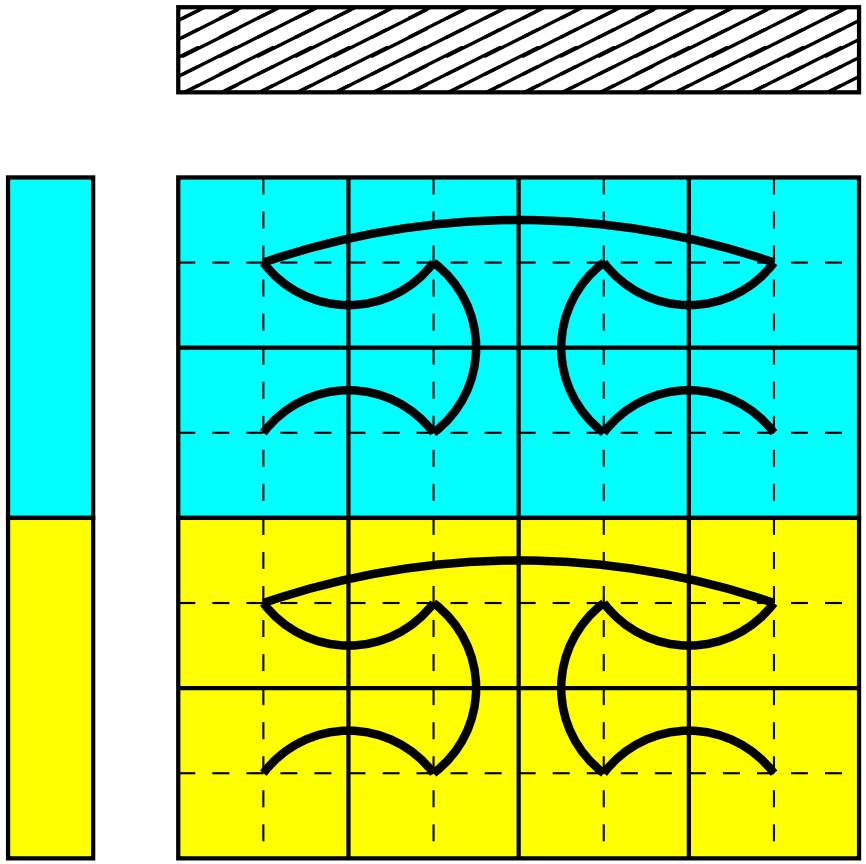
Submatrices provides a scalable (and NUMA-friendly!) solution:



Using space-filling curves within submatrices; using Z-curve yields Compressed Sparse Blocks (**CSB**), Buluç et al., 2009

Parallel Hilbert-based SpMV

Pre-distributing matrix rows scales as well (**1D** Hilbert method):



(Use of space-filling curves to order submatrices; with Z-curve, see Lorton and Wise, 2006; Martone et al., 2010)

Parallel BSP-based SpMV

2-step BSP algorithm for shared-memory NUMA (employs local subvectors for *both* input and output vectors):

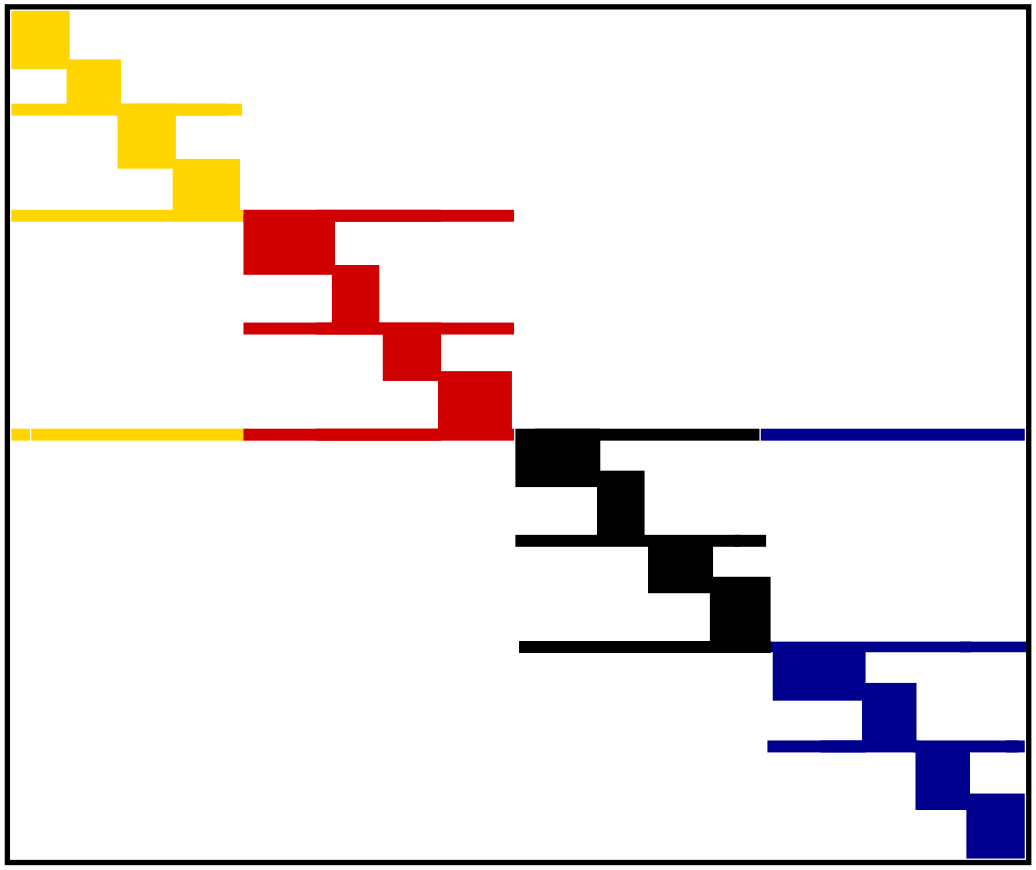
1. **for all** nonzeroes k **from** A
   if column of k is not local
   get element from x from the appropriate processor
   execute local cache-oblivious SpMV kernel
   send all non-local row sums to the appropriate processor
   synchronise
2. **add** all incoming row sums to the corresponding $y[i]$

Yzelman and Bisseling, *An Object-Oriented BSP Library for Multicore Programming*, Concurrency and Computation: Practice and Experience, 2011

Local multiplication can utilise the Hilbert scheme, or

Parallel and cache-oblivious SpMV (2D SBD method)

use both partitioner and reordering output; that is, partition for $p \longrightarrow \infty$, but distribute only over the actual number of processors:



Experimental results

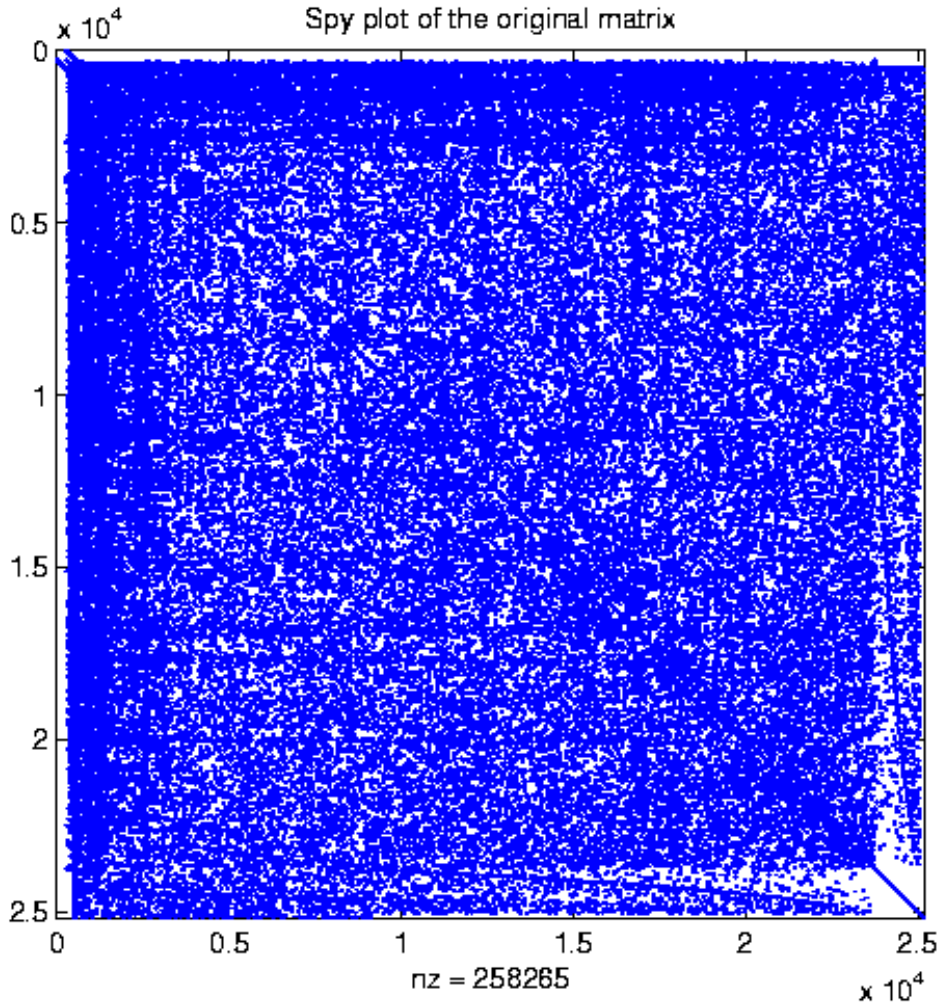
1 The (classic) parallel SpMV: partitioning

2 Sequential SpMV

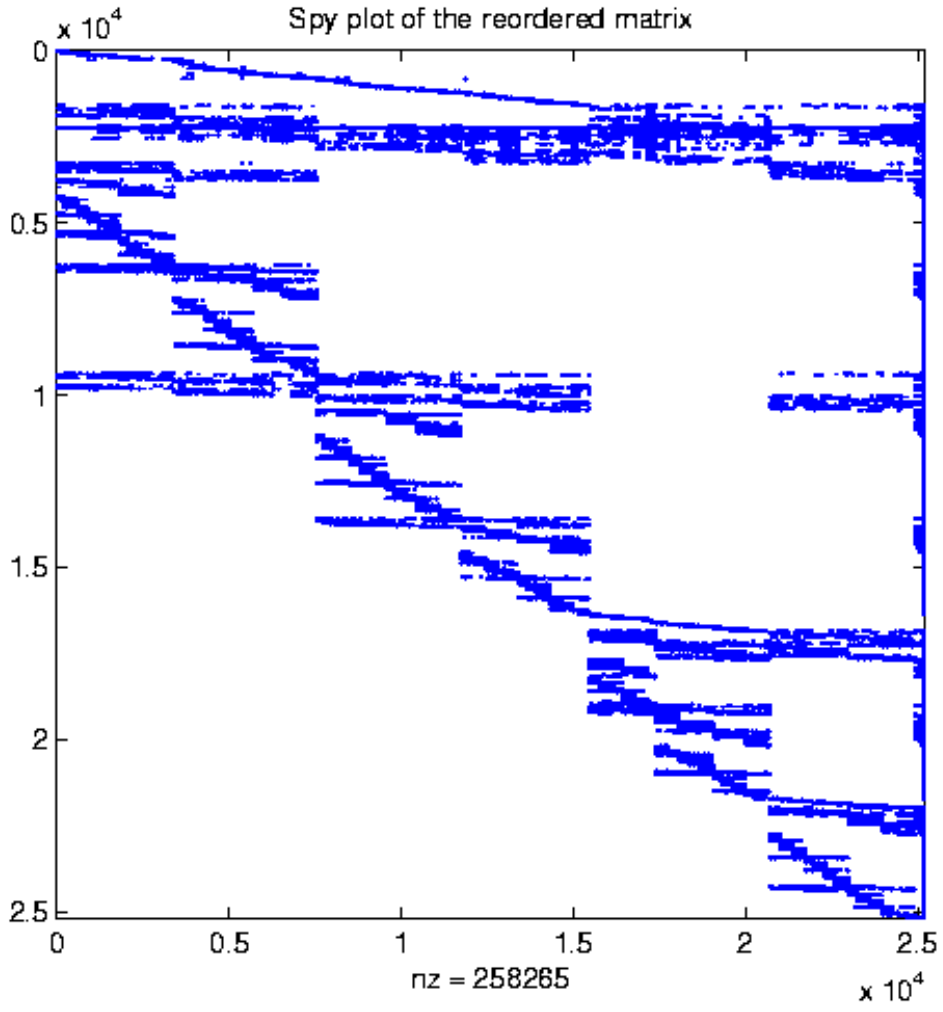
3 Parallel cache-friendly SpMV

4 Experimental results

Chip industry – rhpentium

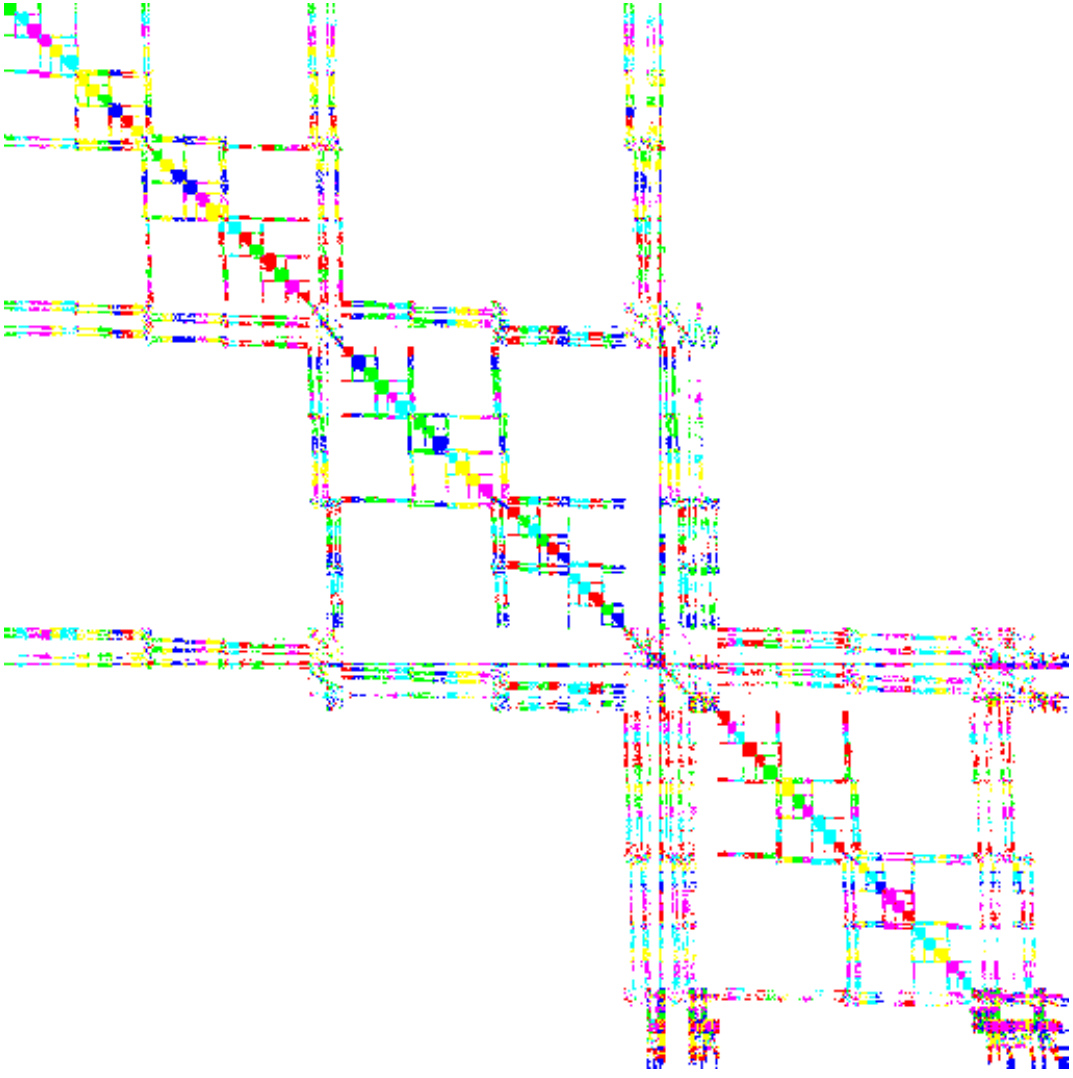


Chip industry – 1D reordering



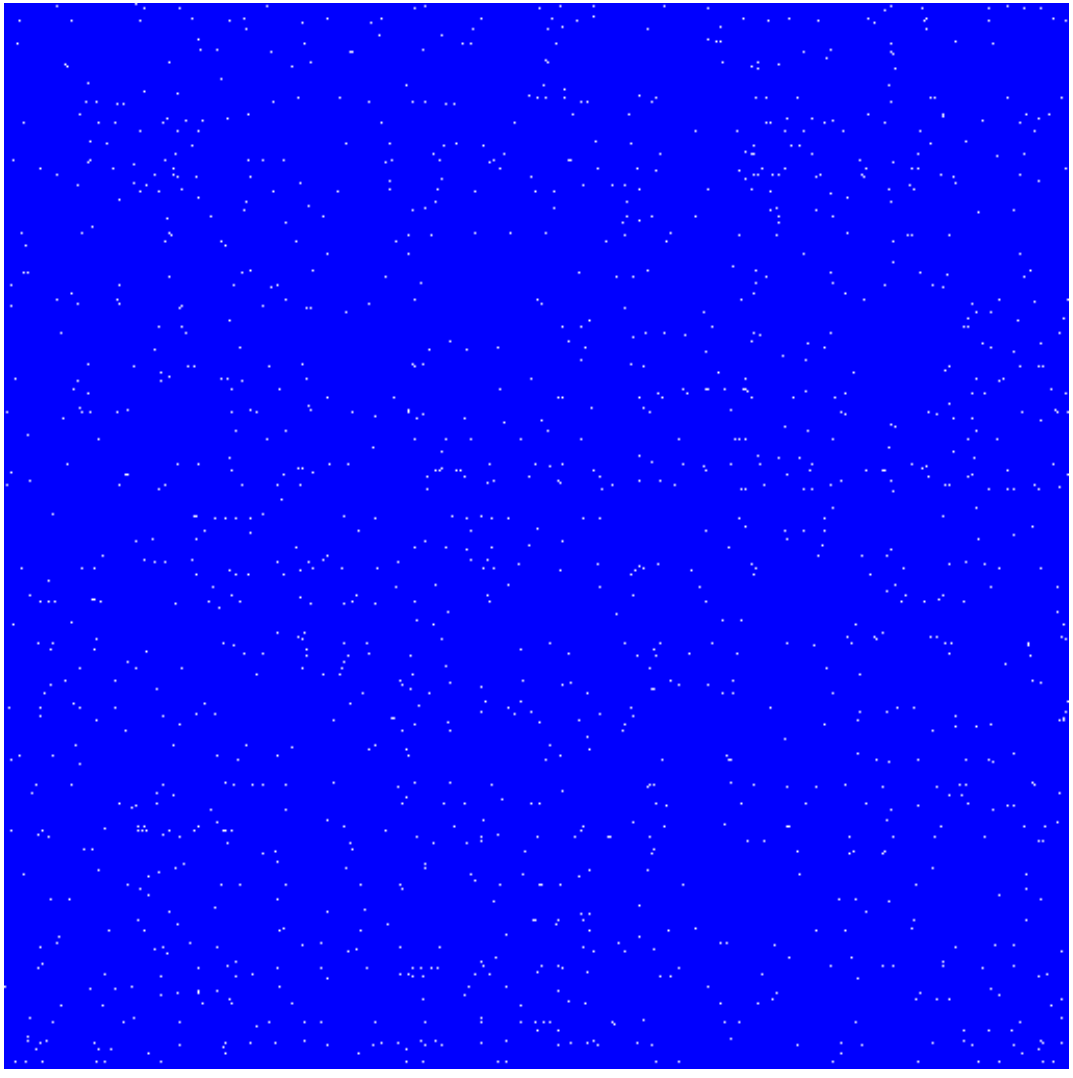
$$p = 100, \quad \epsilon = 0.1$$

Chip industry – 2D reordering

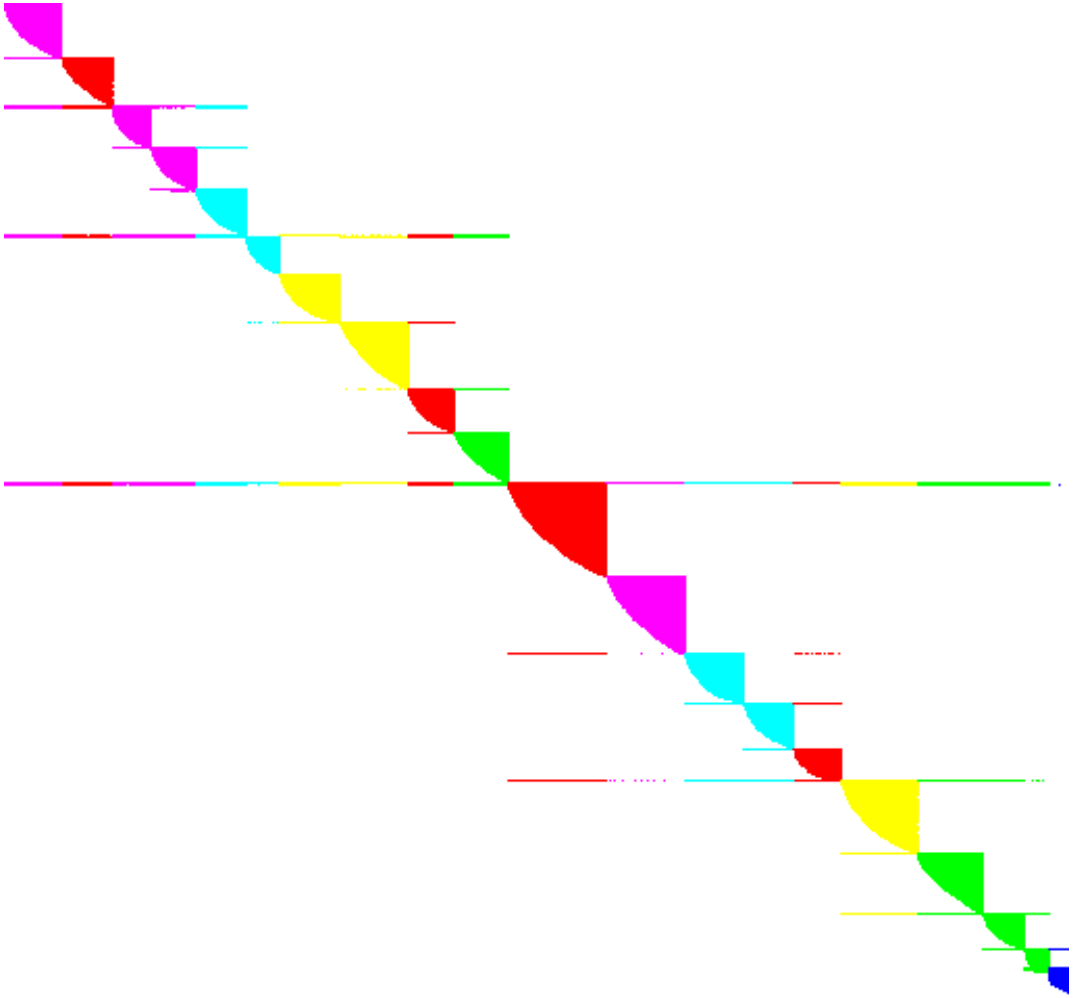


$p = 100, \quad \epsilon = 0.1$

Link matrix

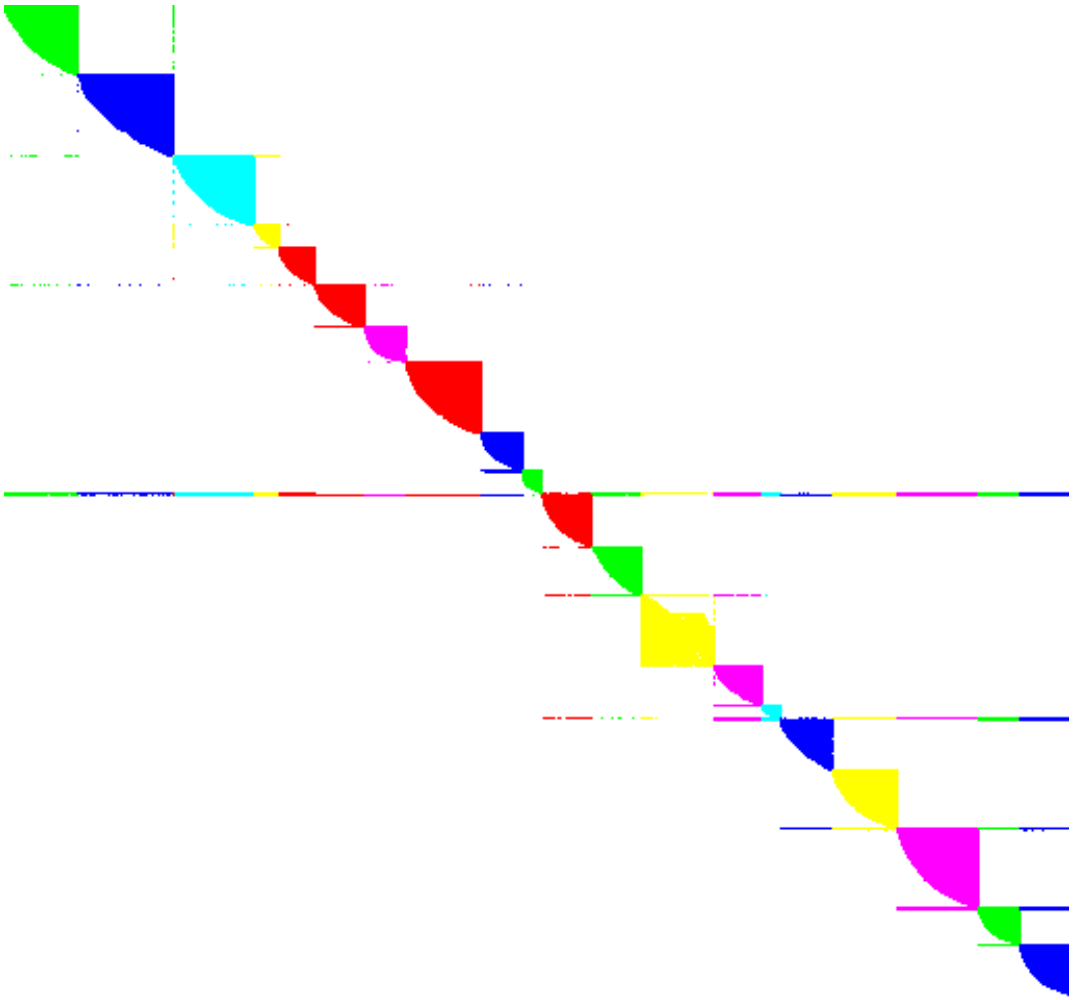


Link matrix – 1D reordering



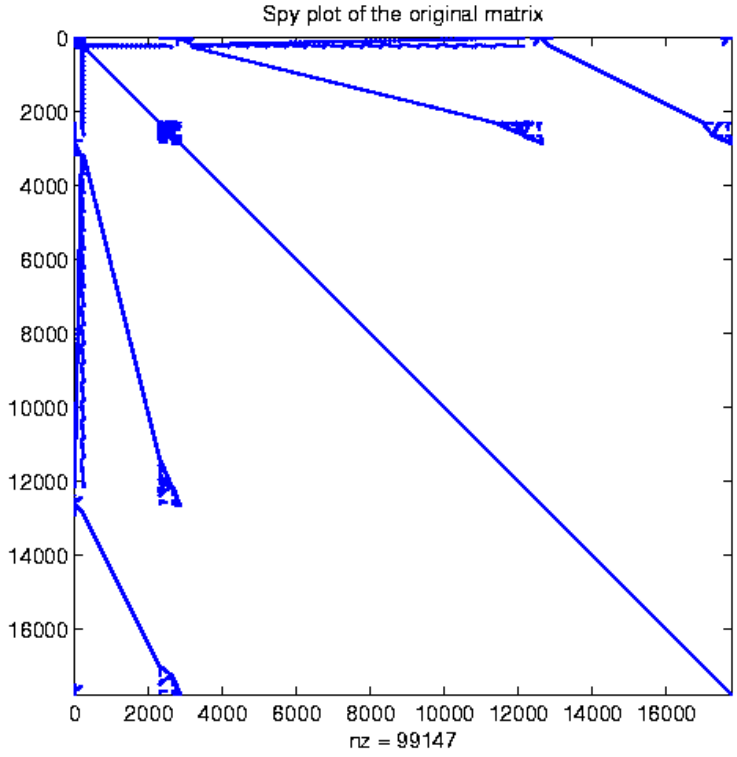
$p = 20, \quad \epsilon = 0.1$

Link matrix – 2D reordering

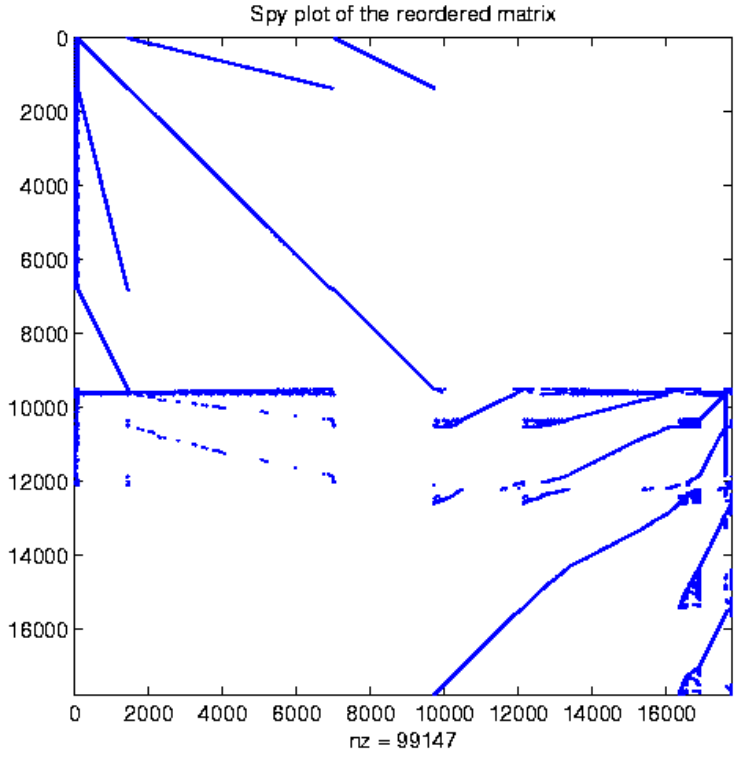


$p = 20, \quad \epsilon = 0.1$

The memplus matrix – 1D reordering

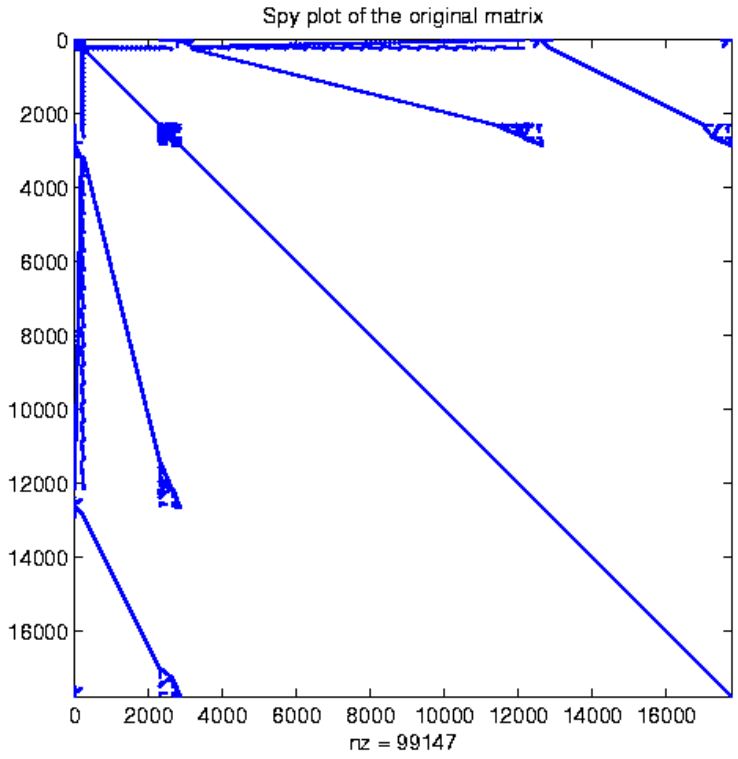


$p = 1$ (original)

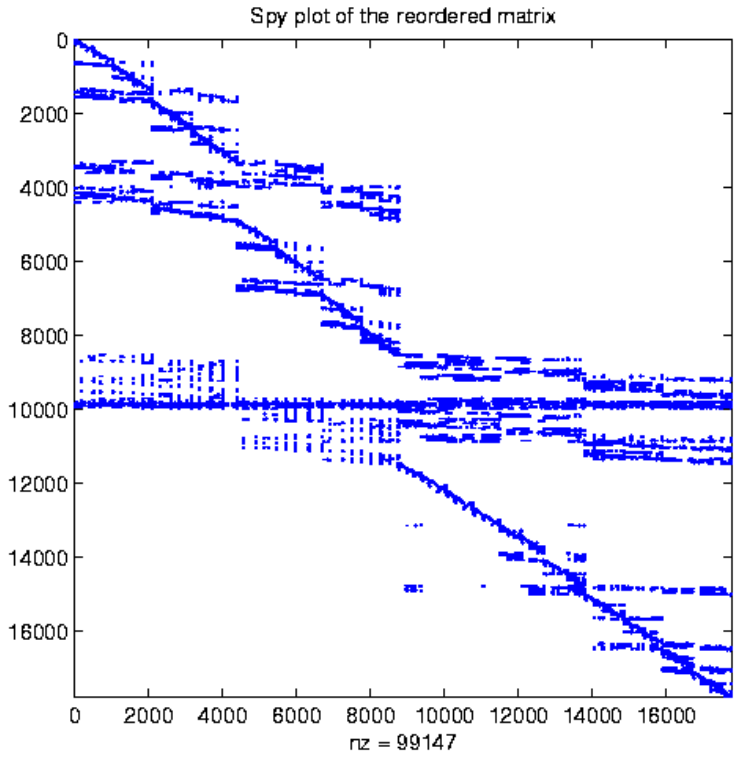


$p = 2, \quad \epsilon = 0.1$

The memplus matrix – 1D reordering



$p = 1$ (original)



$p = 100, \quad \epsilon = 0.1$

Sequential pre-processing and SBD SpMV times

Comparing natural ordering against (doubly) SBD:

Matrix	Reordering time	SpMV time (old/1D/2D)
memplus, $p = 50$:	4 seconds	(0.4 / 0.3 / 0.3 ms.)
rhpentium, $p = 50$:	1 minute	(0.9 / 0.7 / 0.9 ms.)
cage14, $p = 10$:	30 minutes	(111.6 / 130.4 / 130.4 ms.)
wiki2005, $p = 10$:	2 hours	(347.4 / 212.5 / 136.7 ms.)
GL7d18, $p = 10$:	2 hours	(780.3 / 552.5 / 549.5 ms.)
wiki2006, $p = 9$:	21 hours	(745.0 / 495.0 / 311.8 ms.)

Black indicates use of regular (Incremental) CRS, green the use of block ordering, blue the use of the OSKI auto-tuning library.

(reordering on an AMD Opteron 2378, the SpMVs on an Intel Q6600)

Sequential SBD and Hilbert SpMV times

Comparing the sequential Hilbert scheme against (doubly) SBD:

Matrix	Hilbert	SBD (old/1D/2D)
cage14, $p = 10$:	165.2 ms.	(111.6 / 130.4 / 130.4 ms.)
wiki2005, $p = 10$:	253.5 ms.	(347.4 / 212.5 / 136.7 ms.)
GL7d18, $p = 10$:	372.3 ms.	(780.3 / 552.5 / 549.5 ms.)
wiki2006, $p = 9$:	576.7 ms.	(745.0 / 495.0 / 311.8 ms.)

Black indicates use of a regular data structure, green the use of block ordering, blue the use of the OSKI auto-tuning library.

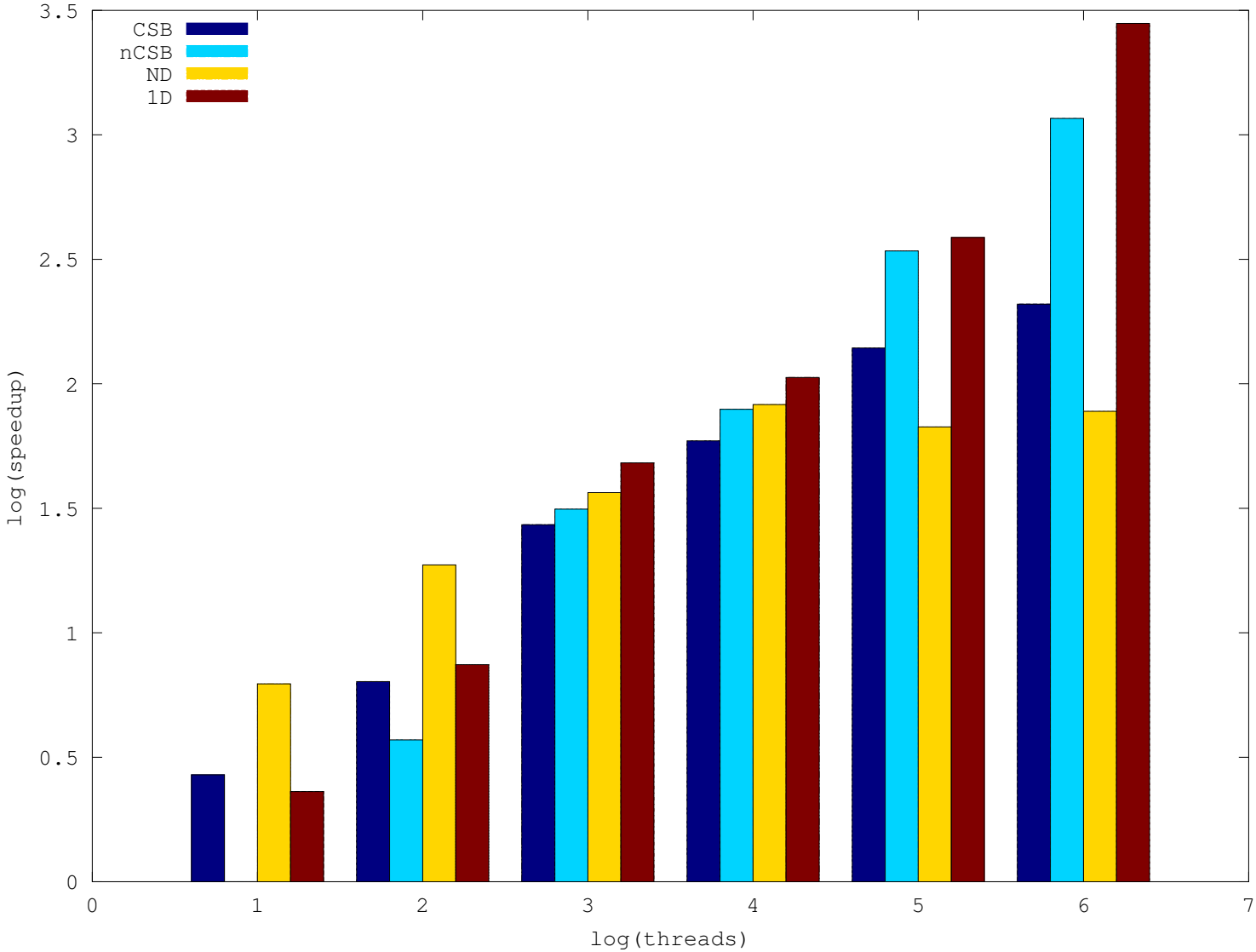
Pre-processing times for the Hilbert scheme is much faster than reordering; wiki2006 takes 24 seconds, the others proportionally less.

Parallel spacefilling-curve based methods

Experiments performed on the wikipedia-2006 matrix, using an 8-socket octacore E7-2830 (Intel Westmere-EX) machine, for a total of 64 physical cores. The memory hierarchy is highly non-uniform.

Speedups are relative to a sequential CRS-based SpMV multiplication, but includes CSB as comparison.

Parallel spacefilling-curve based methods



Parallel 2D SBD method

Partitioning and doubly SBD reordering with $p \longrightarrow \infty$ on two matrices, performed on an Intel Core 2 Q6600 (these are old results using $t - 1$ synchronisations and global vectors, and thus would not be NUMA-friendly):

s3dkt3m2 $t \backslash p$	4	16	32	64
1	17	*16*	18	17
2	17	*16*	18	17
4	20	*18*	22	21

GL7d18 $t \backslash p$	4	16	32	64
1	906	633	492	*486*
2	718	347	345	*285*
4	583	491	398	*385*

Maximum speedup: 3.1x using 2 cores

Thank you for your attention!

Questions?

Albert-Jan N. Yzelman
Deptartment of Computing Science, KU Leuven
ExaScience Laboratory (Intel Labs Europe)

http://people.cs.kuleuven.be/~albert-jan.yzelman
albert-jan.yzelman@cs.kuleuven.be

Software locations:

- albert-jan.yzelman.net/software
- www.math.uu.nl/people/bisseling/Mondriaan
- www.multicorebsp.com

Sequential SpMV times without reordering

	Intel Q6600		AMD 945e	
	s3dkt3m2	GL7d18	s3dkt3m2	GL7d18
Triplet	18	1323	12	466
CRS	14	780	12	437
ICRS	13	856	9	610

All timings are in milliseconds.
s3dkt3m2 is a 90449×90449 structured sparse matrix with about 1.9 million nonzeroes.
GL7d18 is a 1955309×1548650 unstructured sparse matrix with about 35.6 million nonzeroes.

Sequential SpMV times with reordering

	Intel Q6600			
	s3dkt3m2		GL7d18	
Original	10	(OSKI)	780	(CRS)
Hilbert	28	(BICRS)	372	(BICRS)
1D reordering	10	(OSKI)	553	(OSKI)
1D blocking	13	(Block)	613	(BICRS)
2D Mondriaan	11	(Block)	550	(CRS)
2D finegrain	13	(ICRS)	568	(BICRS)

All timings are in milliseconds.

Sequential SpMV times with reordering

	AMD 945e			
	s3dkt3m2		GL7d18	
Original	9	(ICRS)	730	(CRS)
Hilbert	16	(BICRS)	453	(BICRS)
1D reordering	9	(ICRS)	452	(ICRS)
1D blocking	9	(Block)	412	(BICRS)
2D Mondriaan	8	(BICRS)	425	(BICRS)
2D finegrain	9	(Block)	423	(BICRS)

All timings are in milliseconds.

Parallel: distributed-memory architectures

Directly use partitioner output:

Matrix	$p = 1$	$p = 4$	$p = 16$	$p = 64$
cage13	372.2	120.7 (3.0x)	37.1 (10x)	*16.1* (23.1x)
stanford_berkeley	552.6	169.3 (3.2x)	71.2 (7.7x)	*21.4* (25.8x)

Using the BSPonMPI library with the 3-step BSP SpMV multiplication code, on two nodes of 16 IBM Power6+ processors each.

Bisseling, van Leeuwen, Çatalyürek, Fagginger Auer, Yzelman, *Two-dimensional approach to sparse matrix partitioning* in *Combinatorial Scientific Computing* by Schenk and Naumann (eds.), 2011; in press.

Parallel: shared-memory architectures

Directly use partitioner output:

Matrix	$p = 1$	$p = 2$	$p = 3$	$p = 4$
memplus	1.5	2.1 (0.7x)	–	6.0 (0.2x)
cage14	232.8	272.5 (0.8x)	*249.7* (0.9x)	297.1 (0.7x)
wiki2005	564.2	285.3 (1.9x)	*244.5* (2.3x)	255.0 (2.2x)

Using the Java MulticoreBSP library on an Intel Q6600; two superstep algorithm with full synchronisation.

Yzelman and Bisseling, *An Object-Oriented BSP Library for Multicore Programming*, Concurrency and Computation: Practice and Experience, 2011; in press

`http://www.multicorebsp.com`

Combined parallel with reordering

AMD Phenom II 945e:

s3dkt3m2 $t \backslash p$	4	16	32	64
1	*11*	*11*	14	*11*
2	8	*7*	9	8
4	*6*	7	*6*	*6*

GL7d18 $t \backslash p$	4	16	32	64
1	482	373	*352*	372
2	333	376	*236*	357
4	250	200	*199*	237

Maximum speedup for s3dkt3m2: 1.8x using 2 cores
Maximum speedup for GL7d18: 2.4x using 4 cores