Reordering Sparse Matrices for Cache-Oblivious Computations

Albert-Jan Yzelman & Rob Bisseling

July 2010

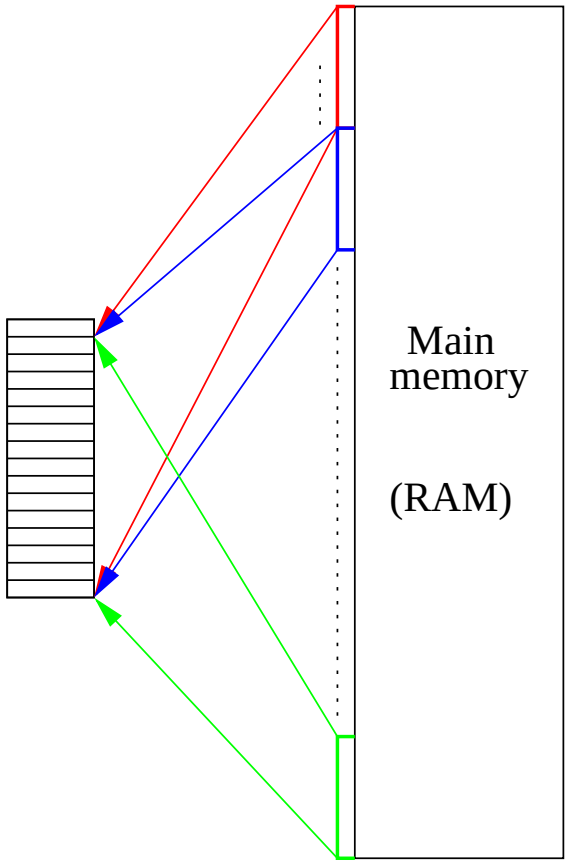
Outline

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

Naive cache

$k = 1$, modulo mapped cache

Dividing main memory (RAM) in stripes of size L_S, the xth line is mapped to the cache line number x mod L:



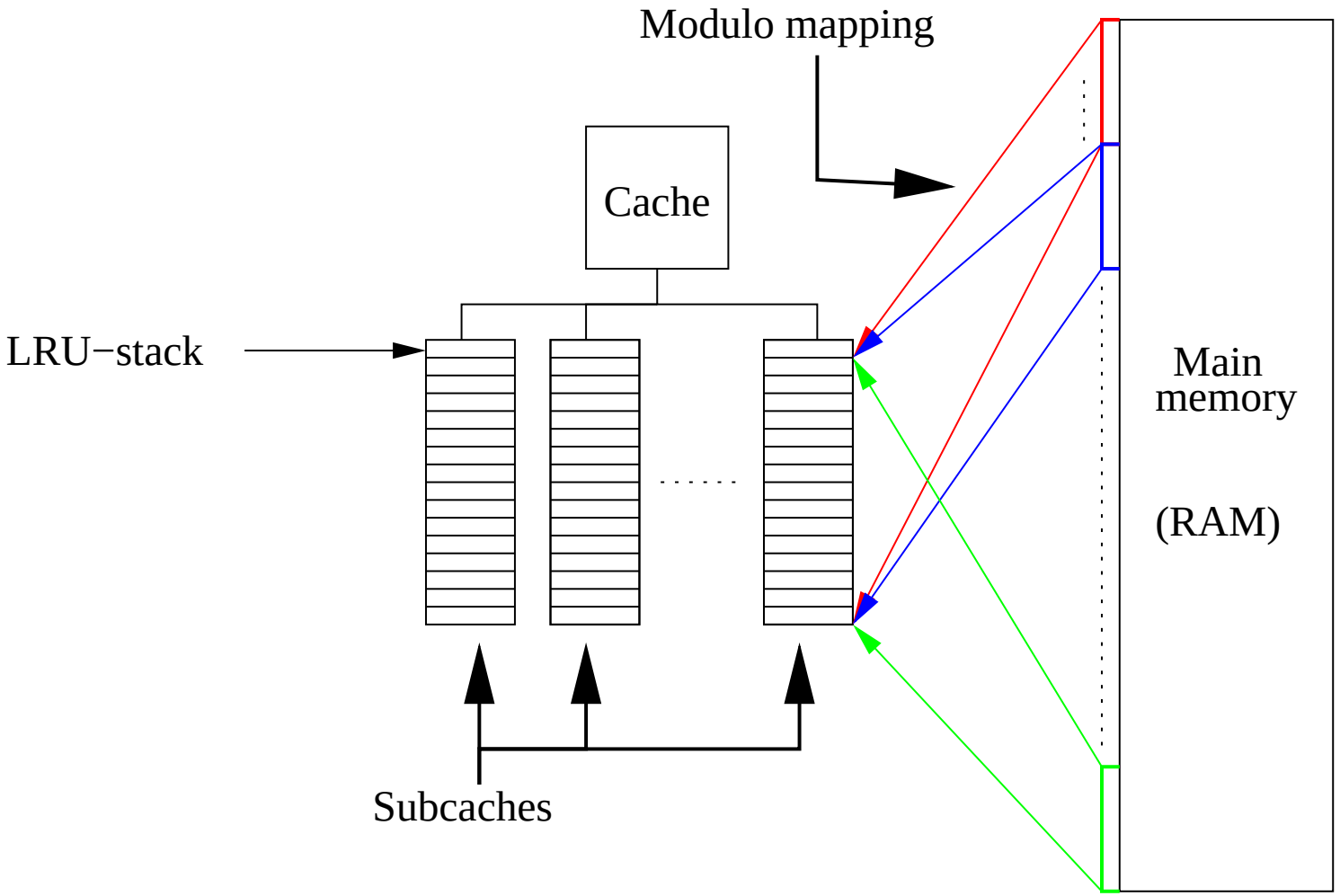
'Ideal' cache

Instead of a naive modulo mapping, we use a smarter policy. We take $k = L = 4$, using 'Least Recently Used (LRU)' policy:

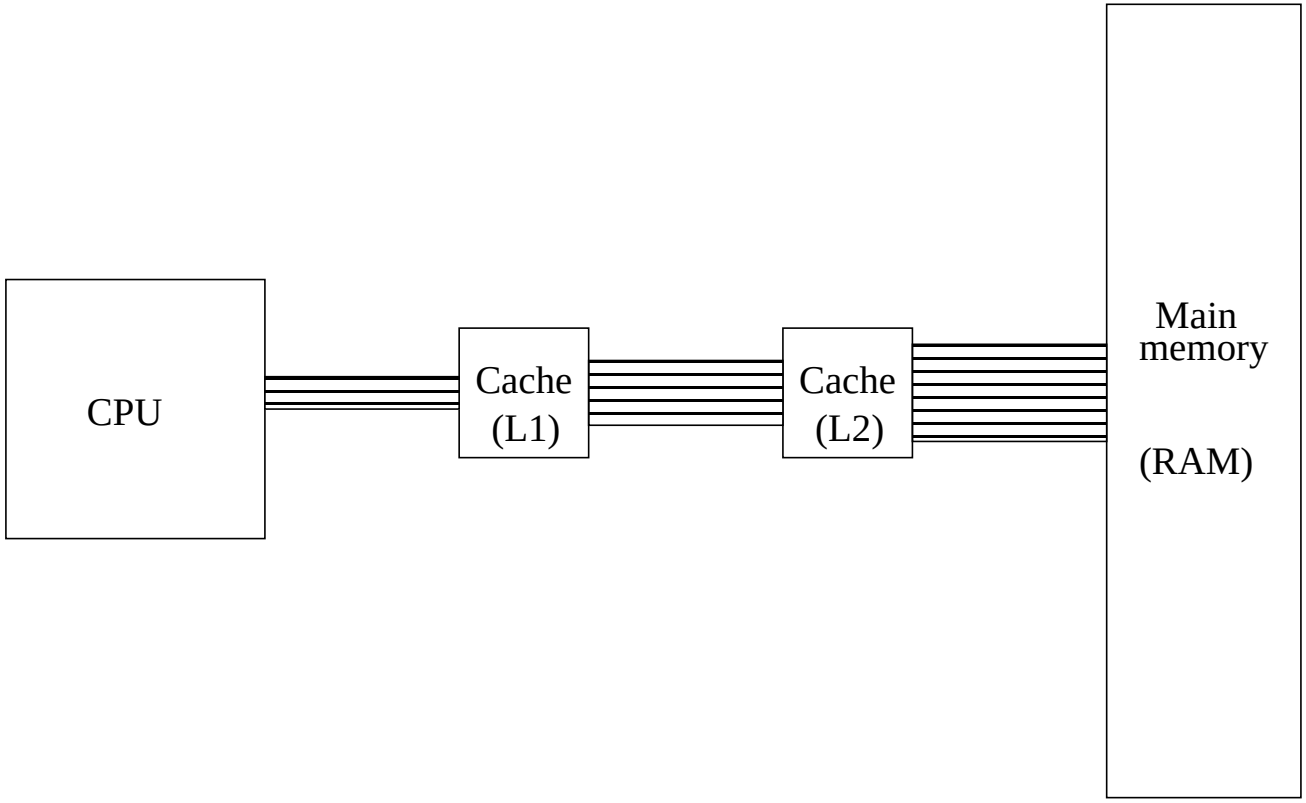
Req. x_1		Req. $x_2, \ldots, x_4$		Req. x_2		Req. x_5
x_1	$\Rightarrow$	x_4	$\Rightarrow$	x_2	$\Rightarrow$	x_5
		x_3		x_4		x_2
		x_2		x_3		x_4
		x_1		x_1		x_3

Realistic cache

$1 < k < L$, combining modulo-mapping and the LRU policy



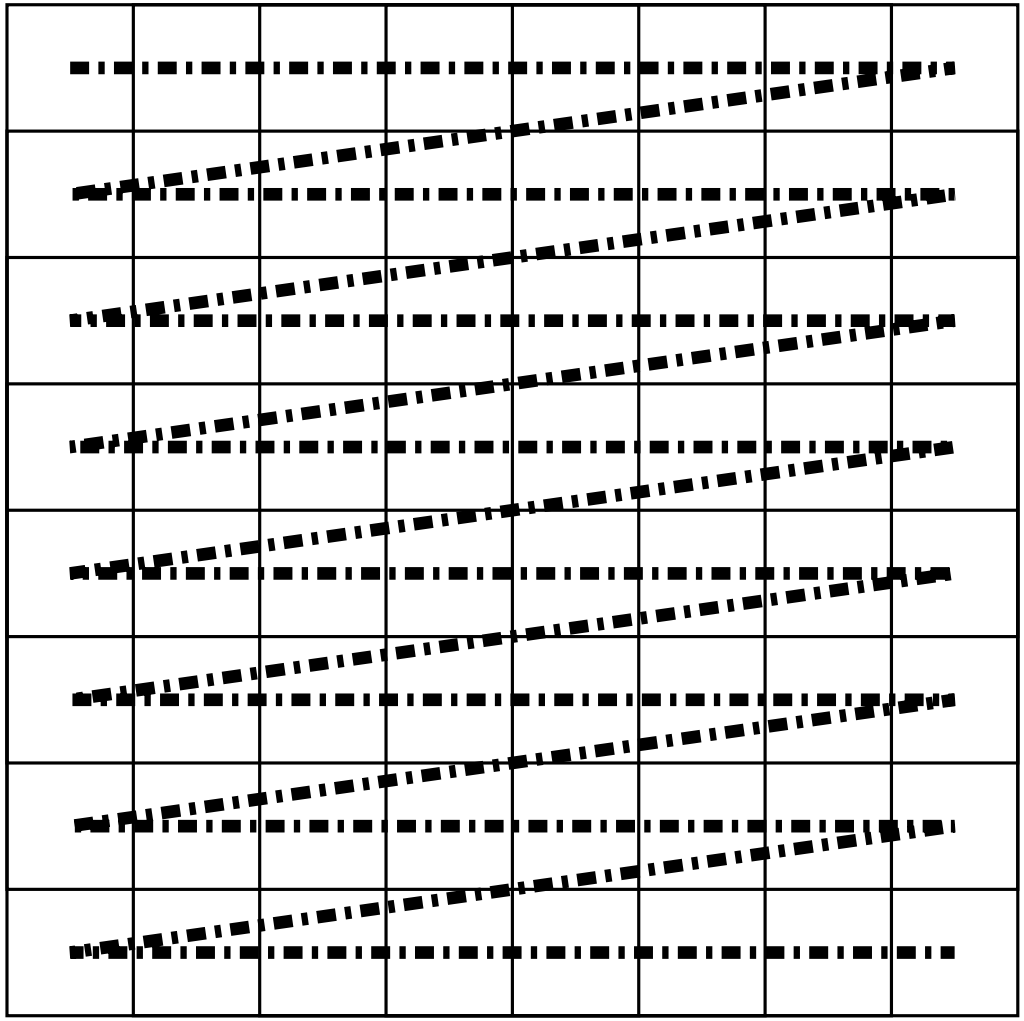
Multilevel caches



	Intel Core2 (Q6600)		AMD Opteron 2378		IBM Power6	
L1:	32kB	$k = 8$	64kB	$k = 2$	64kB	$k = 4$
L2:	4MB	$k = 16$	512kB	$k = 16$	4MB	$k = 8$
L3:	-	-	6MB	$k = 48$	32MB	$k = 16$

Sparse matrix–vector multiplication (SpMV)

Standard datastructure: Compressed Row Storage (CRS)



Sparse matrix–vector multiplication (SpMV)

$x_?$

$\Longrightarrow$

We cannot predict memory accesses in the sparse case, as opposed to the dense case:

$$x_0 \Longrightarrow \begin{matrix} a_{00} \\ x_0 \end{matrix} \Longrightarrow \begin{matrix} y_0 \\ a_{00} \\ x_0 \end{matrix} \Longrightarrow \begin{matrix} x_1 \\ y_0 \\ a_{00} \\ x_0 \end{matrix} \Longrightarrow \begin{matrix} a_{01} \\ x_1 \\ y_0 \\ a_{00} \\ x_0 \end{matrix} \Longrightarrow \begin{matrix} y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ x_0 \end{matrix} \Longrightarrow \begin{matrix} x_2 \\ y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ \hline x_0 \end{matrix}$$

Sparse matrix–vector multiplication (SpMV)

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

We cannot predict memory accesses in the sparse case, as opposed to the dense case:

$$x_0 \quad a_{00} \quad y_0 \qquad \Longrightarrow \qquad x_0 \qquad \Longrightarrow \qquad a_{00} \quad x_1 \quad y_0 \qquad \Longrightarrow \qquad x_0 \qquad \Longrightarrow \qquad y_0 \quad a_{00} \quad x_1 \qquad \Longrightarrow \qquad a_{00} \quad x_1 \quad y_0 \qquad \Longrightarrow \qquad x_1 \quad a_{00} \quad x_0 \qquad \Longrightarrow \qquad x_0 \quad y_0 \quad a_{00} \quad x_1 \quad x_2$$

Sparse matrix–vector multiplication (SpMV)

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

We cannot predict memory accesses in the sparse case, as opposed to the dense case:

$$x_0 \quad \begin{matrix} a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} y_0 \\ a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} x_1 \\ y_0 \\ a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} a_{01} \\ x_1 \\ y_0 \\ a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ x_0 \end{matrix} \quad \Longrightarrow \quad \begin{matrix} x_2 \\ y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ \hline x_0 \end{matrix}$$

Sparse matrix–vector multiplication (SpMV)

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

We cannot predict memory accesses in the sparse case, as opposed to the dense case:

$$x_0 \quad \begin{matrix} a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} y_0 \\ a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} x_1 \\ y_0 \\ a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} a_{01} \\ x_1 \\ y_0 \\ a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ x_0 \end{matrix} \quad \begin{matrix} x_2 \\ y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ \hline x_0 \end{matrix}$$

Cache-friendly data structures

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]

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]

Note: accesses like plain CRS, but requires less instructions for SpMV.

Reference: Joris Koster, *Parallel templates for numerical linear algebra* (Masters Thesis), Utrecht University, 2002.

Blocked CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$, dense blocks: $4, 1, 3$ / $2, 3$ / 1 / 2 / $7, 0, 1, 1$

Stored as:

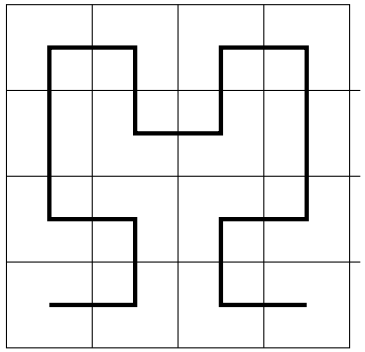
nzs: [4 1 3 2 3 1 2 7 0 1 1]
blk: [0 3 5 6 7 11]
col: [0 2 0 3 0]
row: [0 1 2 4 5]

, $nnz + (2nblk + 1) + (m + 1)$ accesses

Reference: Pinar and Heath, *Improving Performance of Sparse Matrix-Vector Multiplication*, 1999

Fractal datastructures (triplets)

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



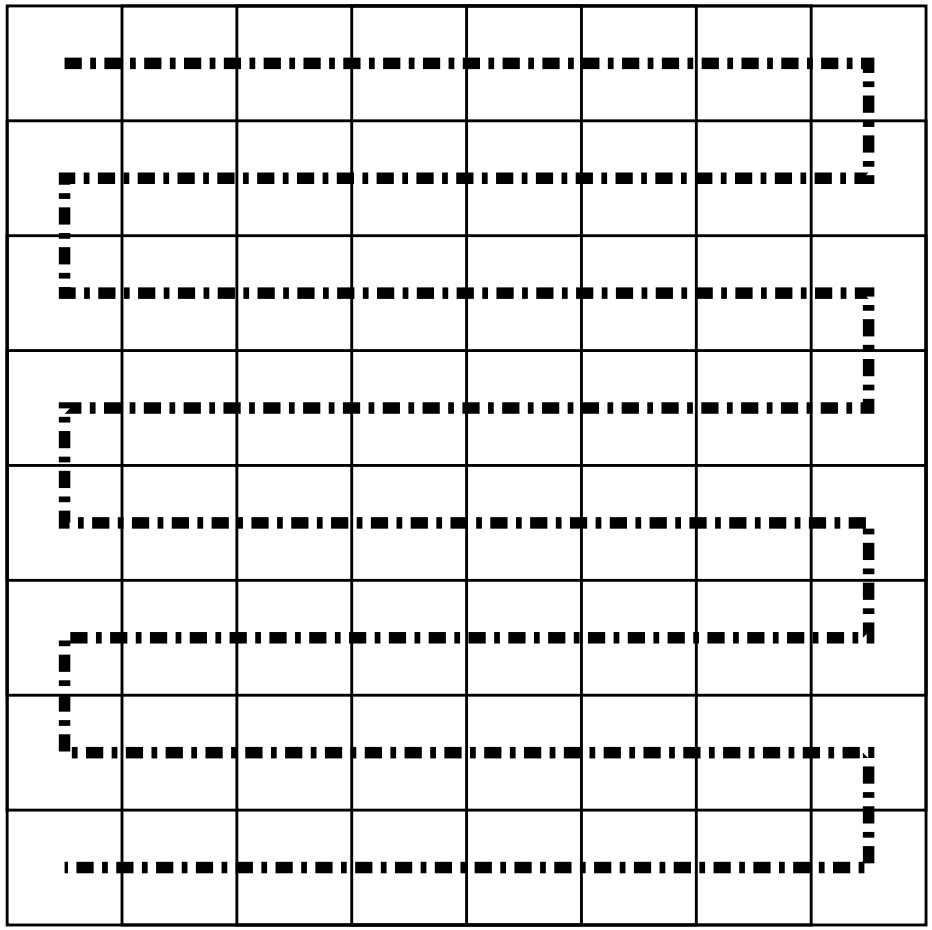
Stored as:

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]

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

Zig-zag CRS

Step 1: Change the order of CRS:



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]

Note: like CRS and ICRS, Zig-zag ICRS can also be constructed.

Reference: Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SISC, 2009

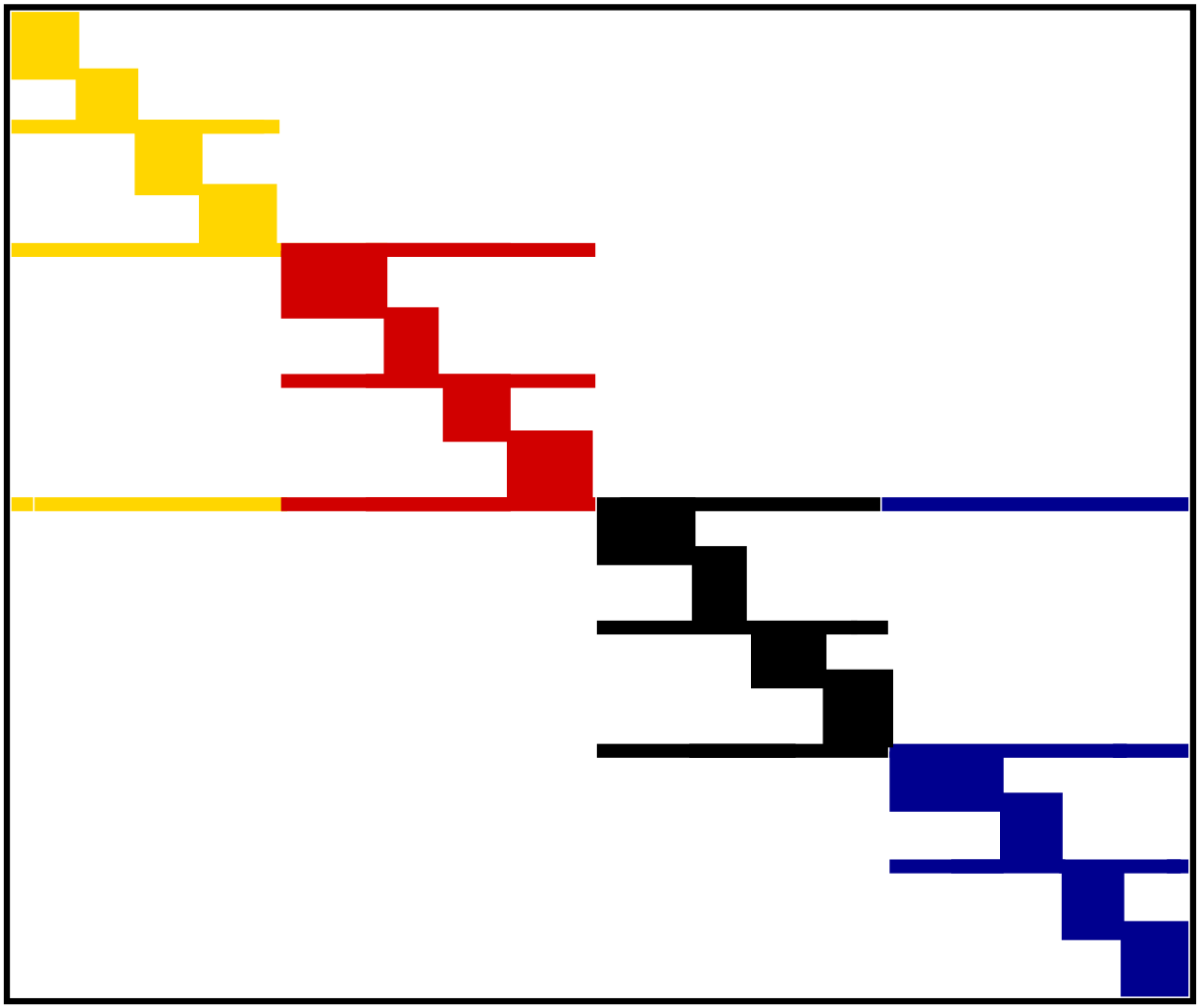
Cache-oblivious sparse matrix structure

Separated Block Diagonal form

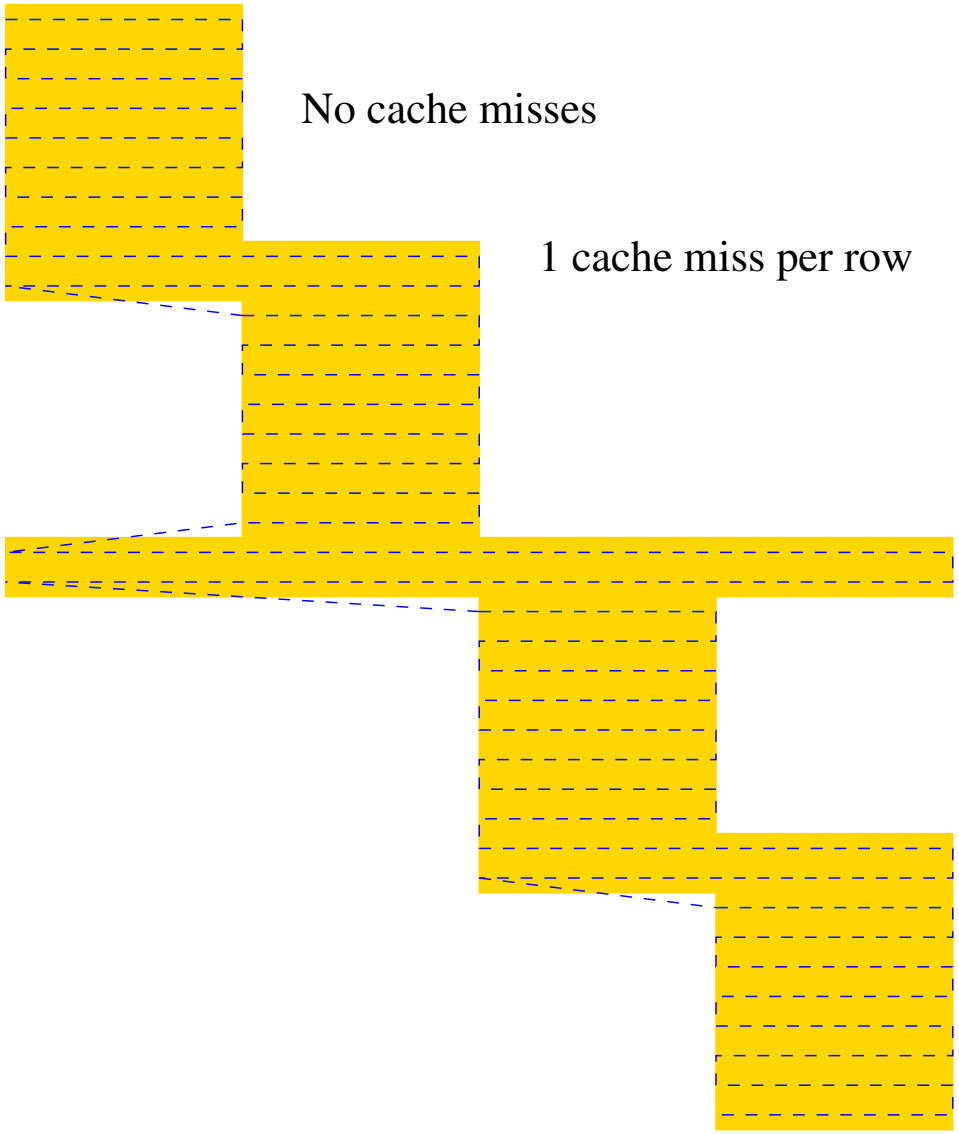
Step 2: change the input matrix structure

- Assuming zig-zag CRS ordering
- Allowing only row and column permutations ($A \Rightarrow PAQ$)

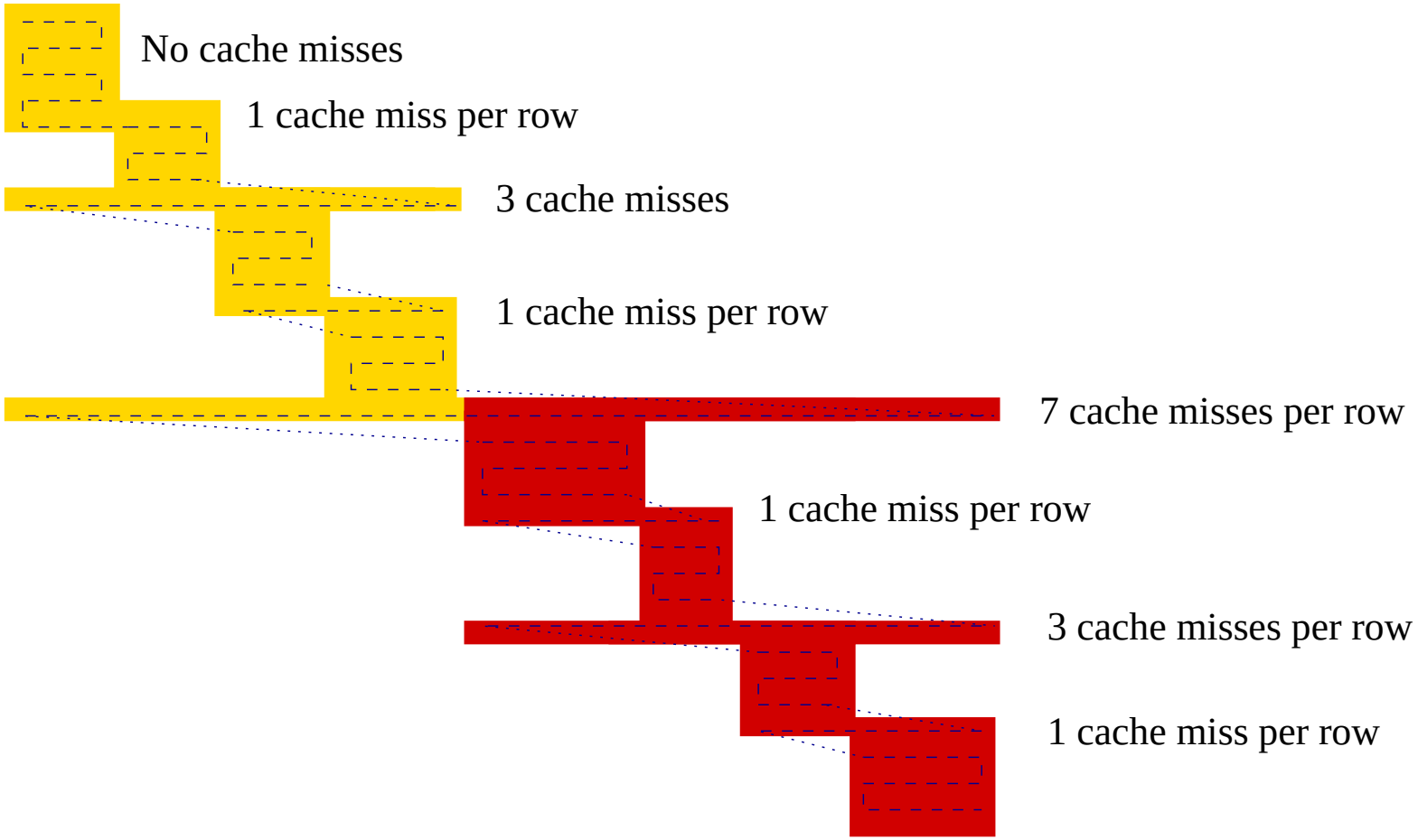
Separated Block Diagonal form



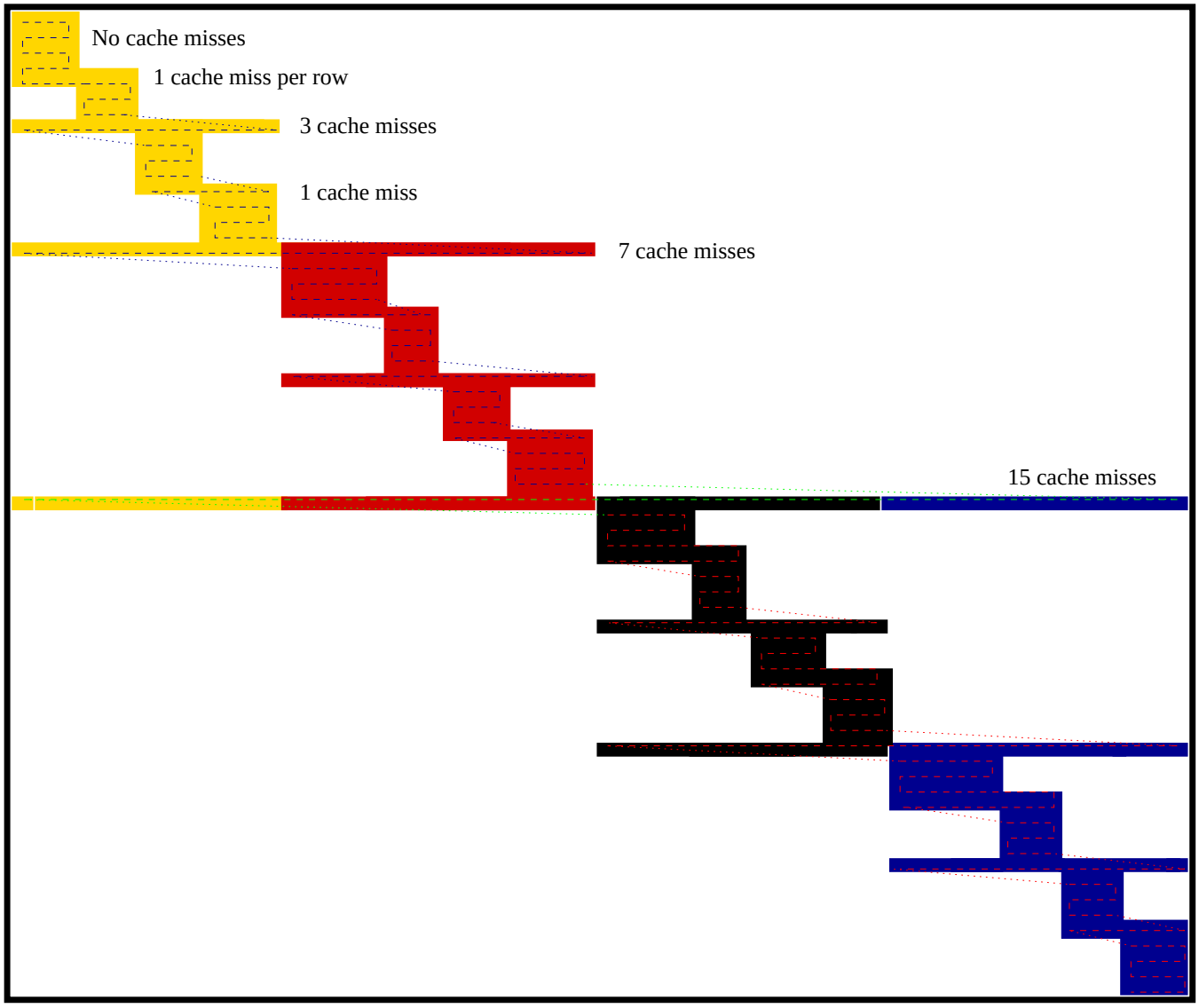
Separated Block Diagonal form



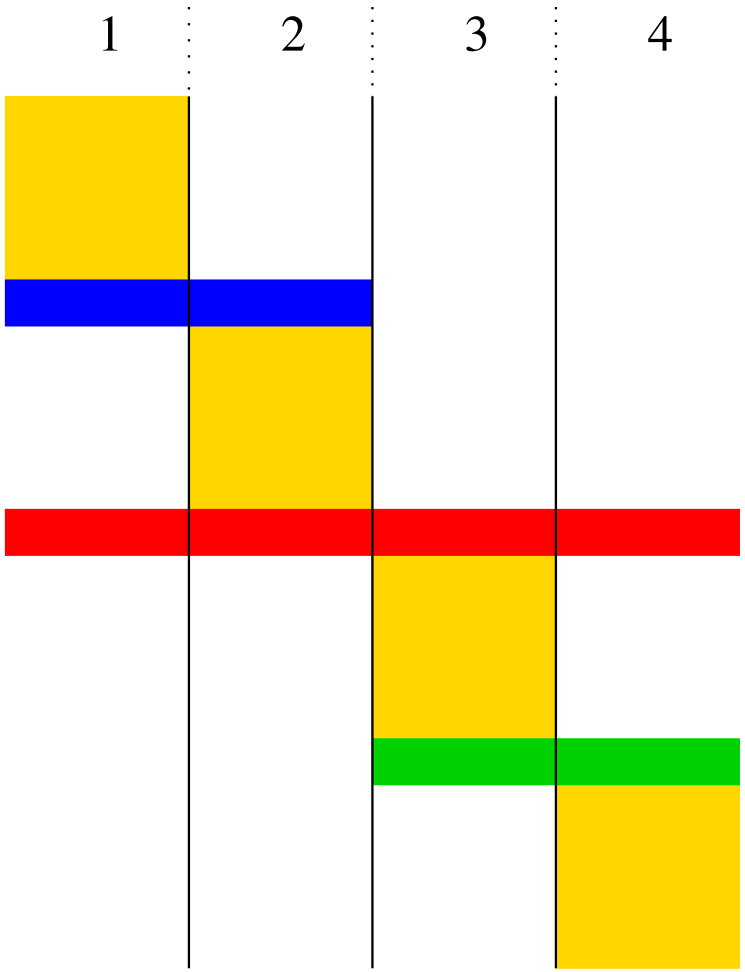
Separated Block Diagonal form

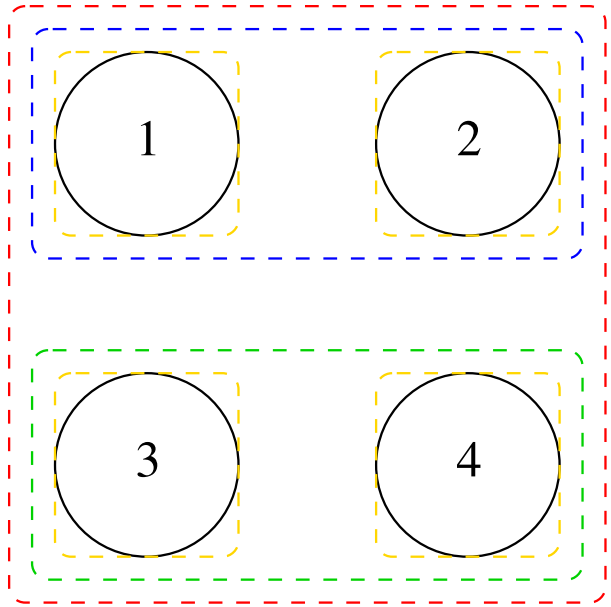


Separated Block Diagonal form



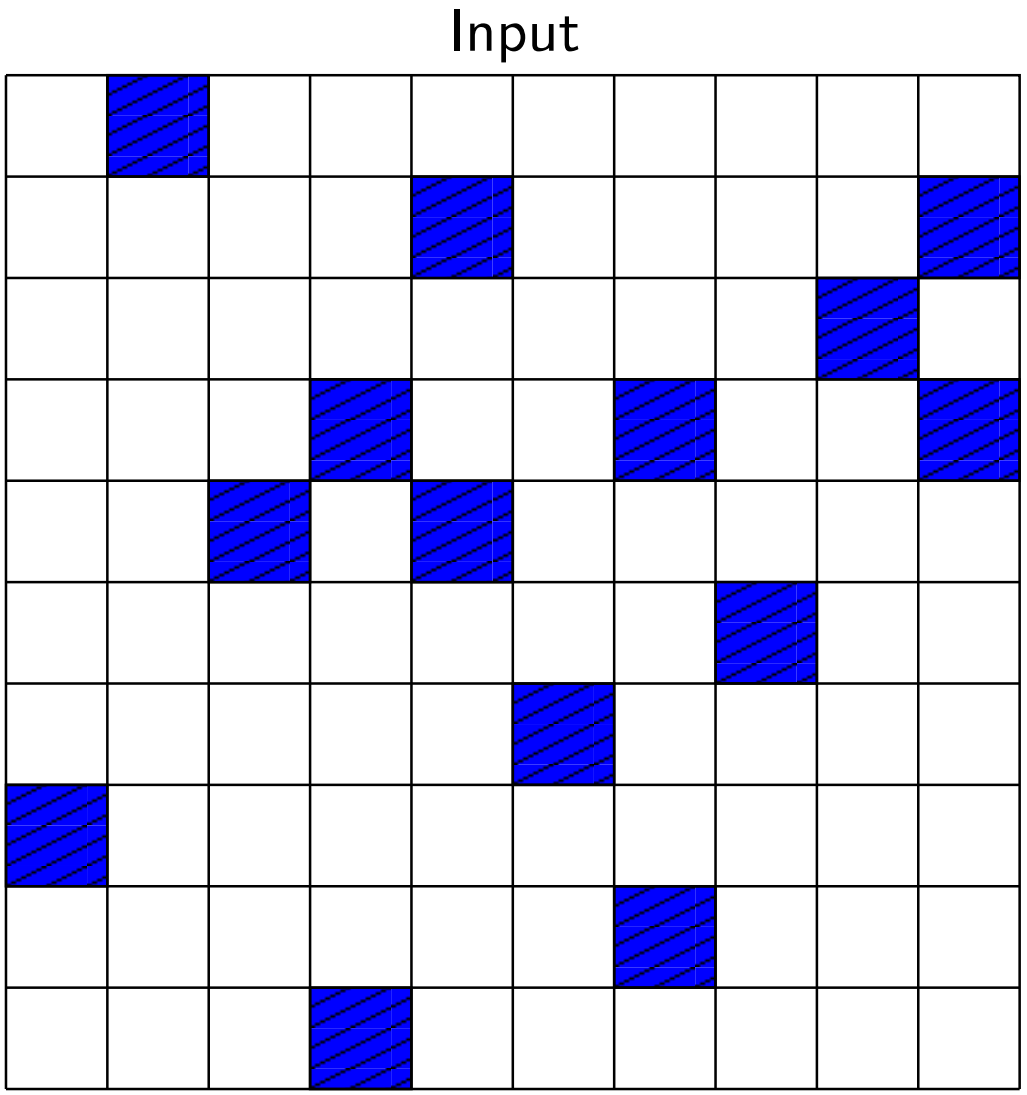
Separated Block Diagonal form



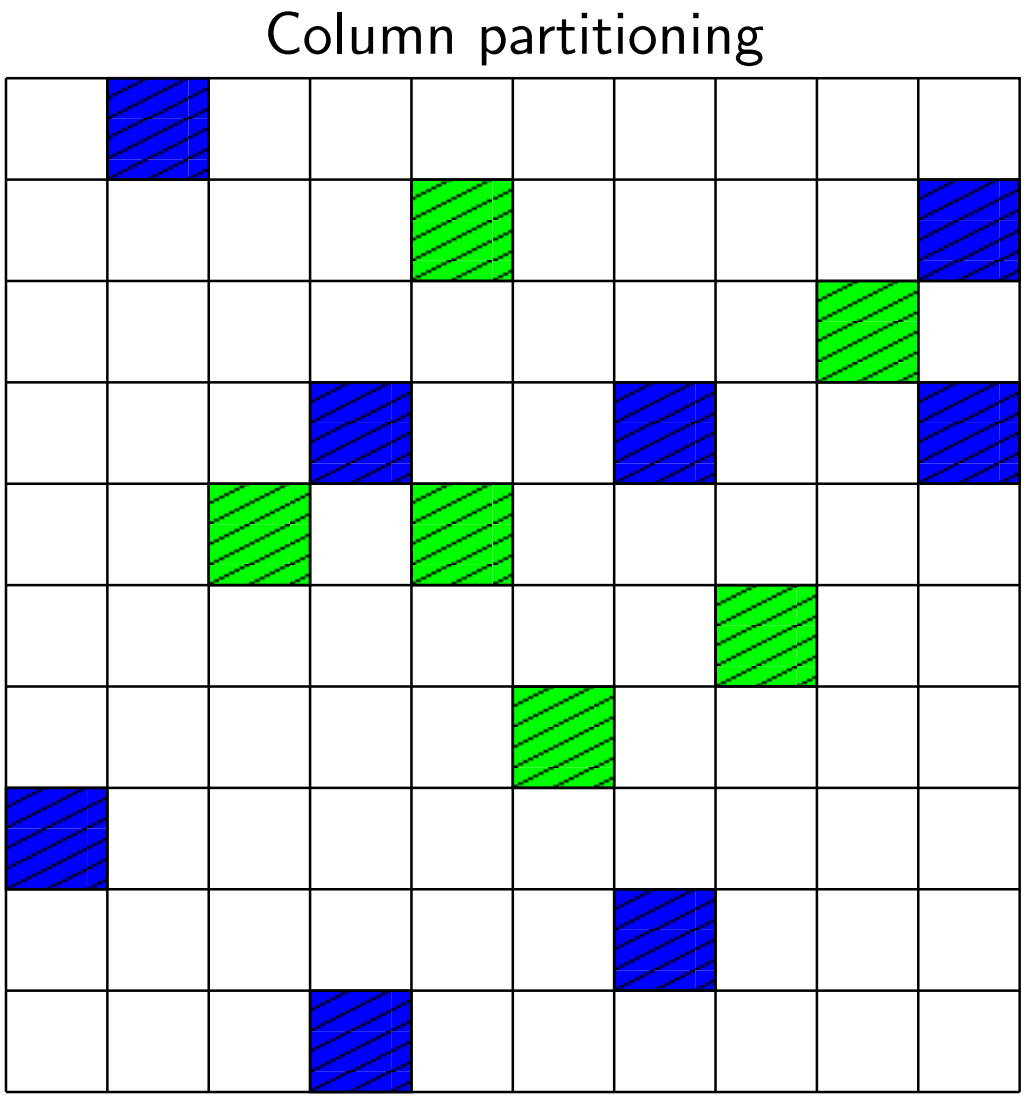


(Upper bound on) the number of cache misses: $\sum_i (\lambda_i - 1)$

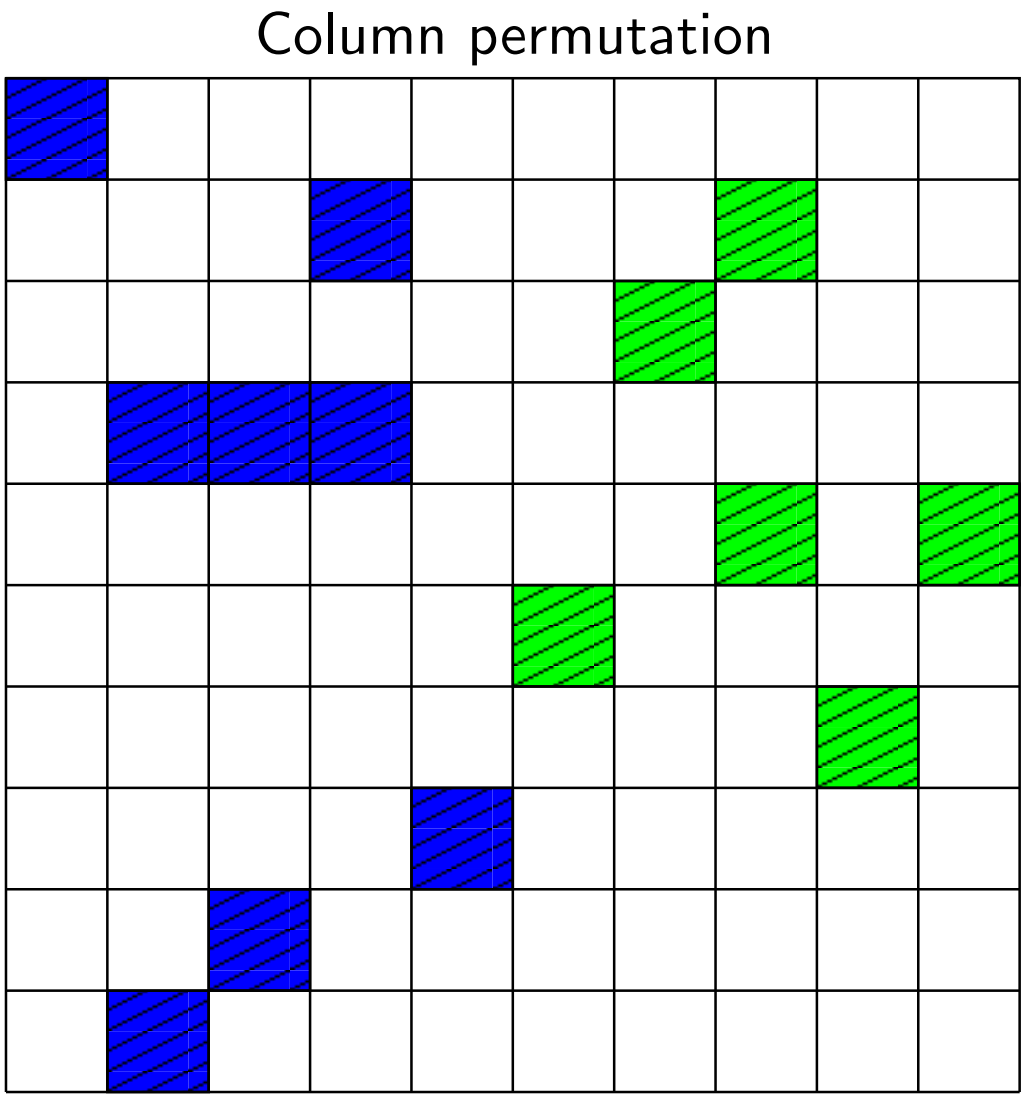
Permuting to SBD form



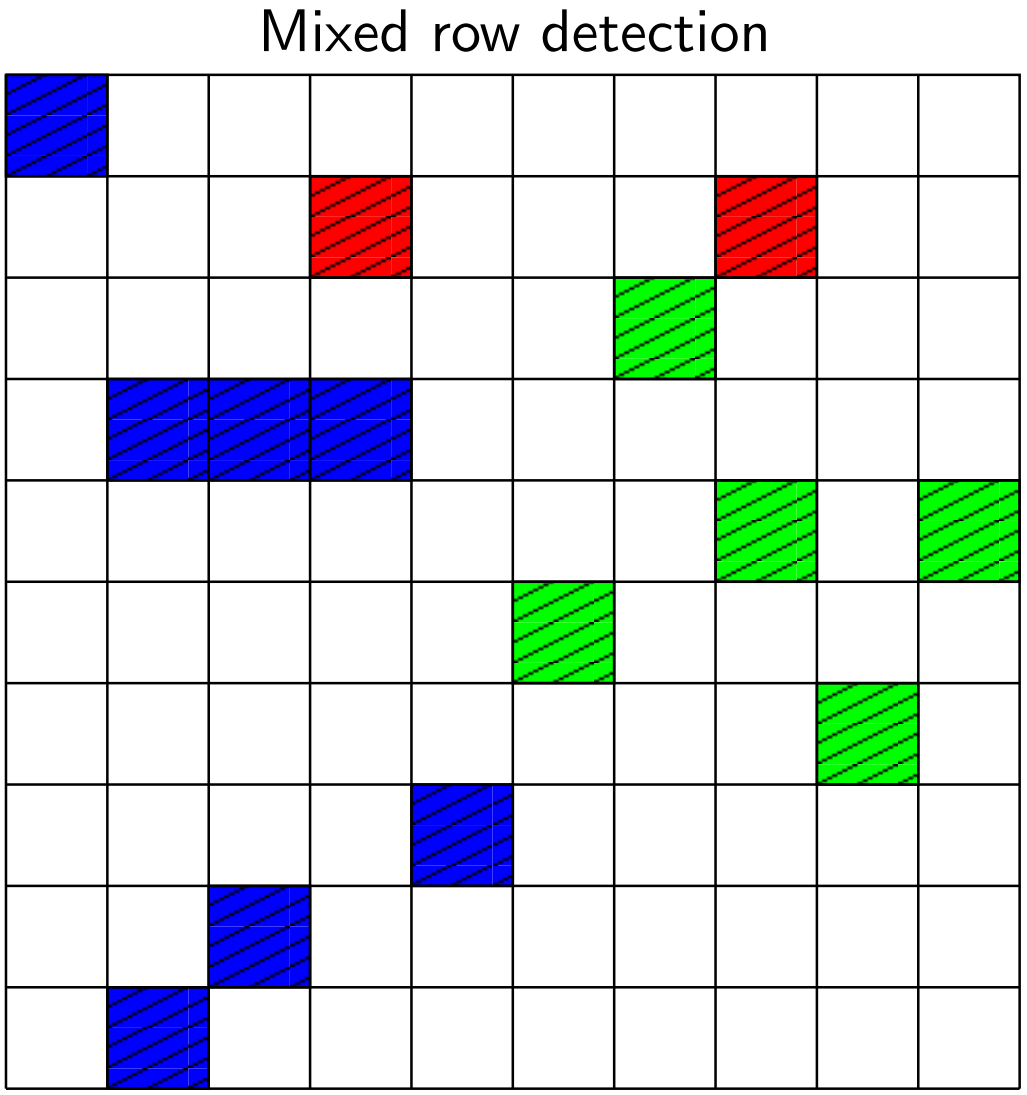
Permuting to SBD form



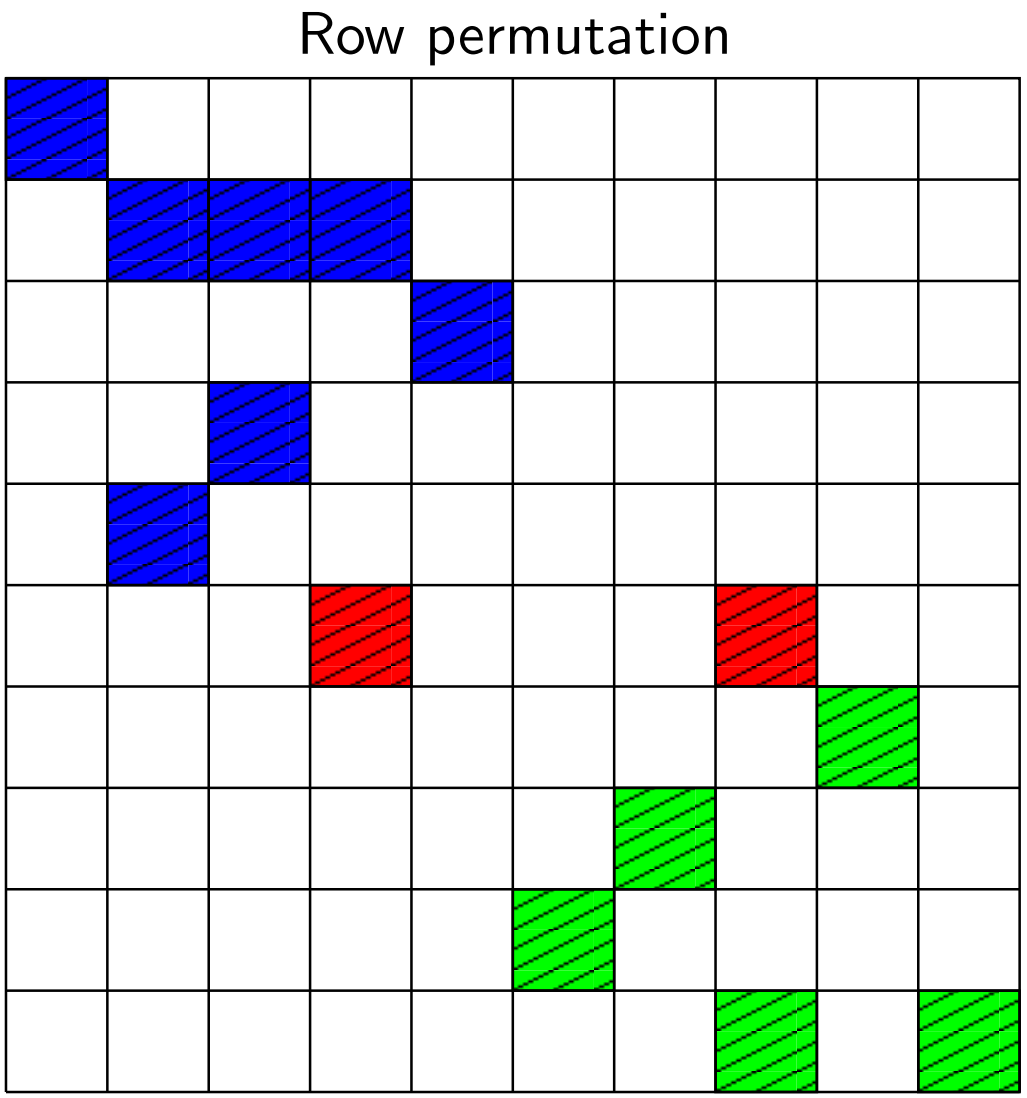
Permuting to SBD form



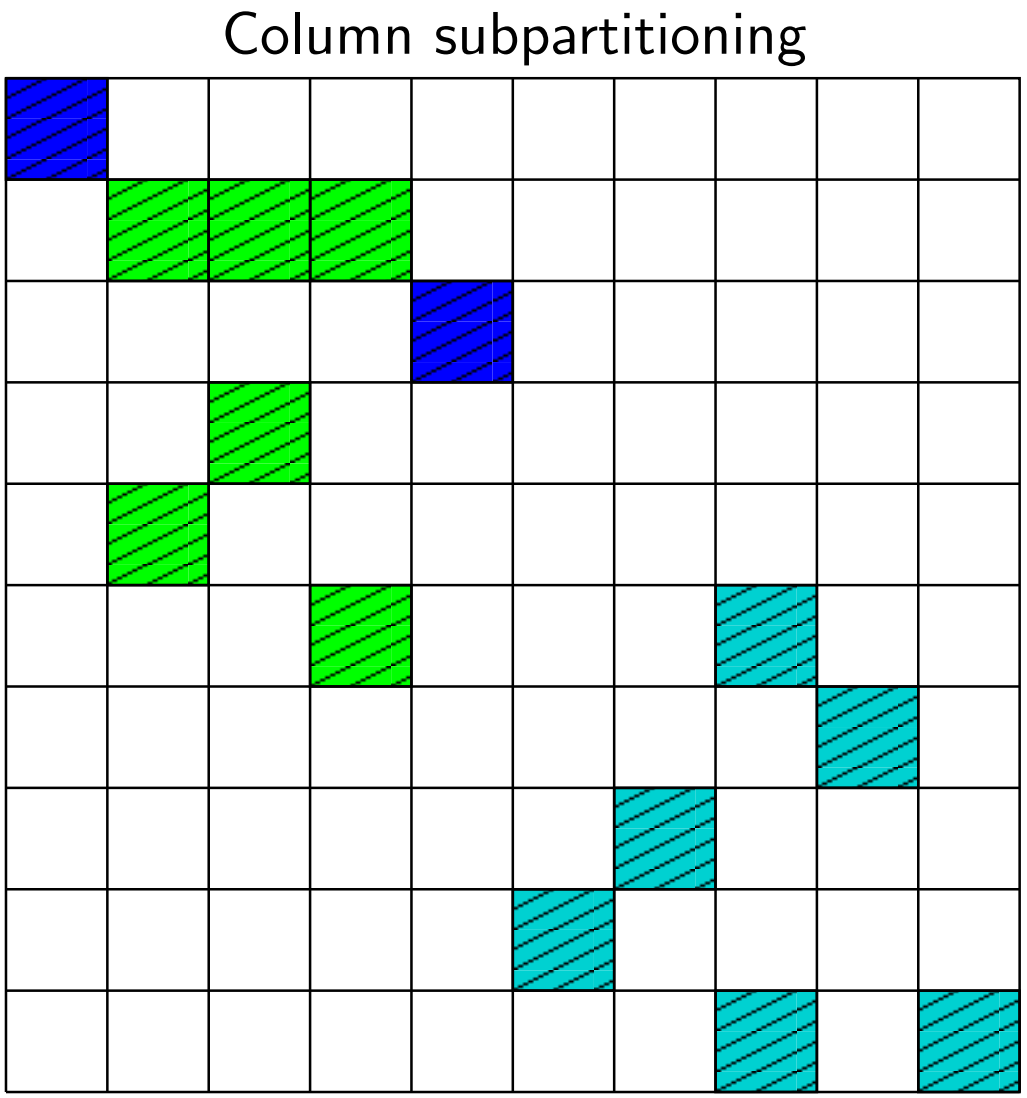
Permuting to SBD form



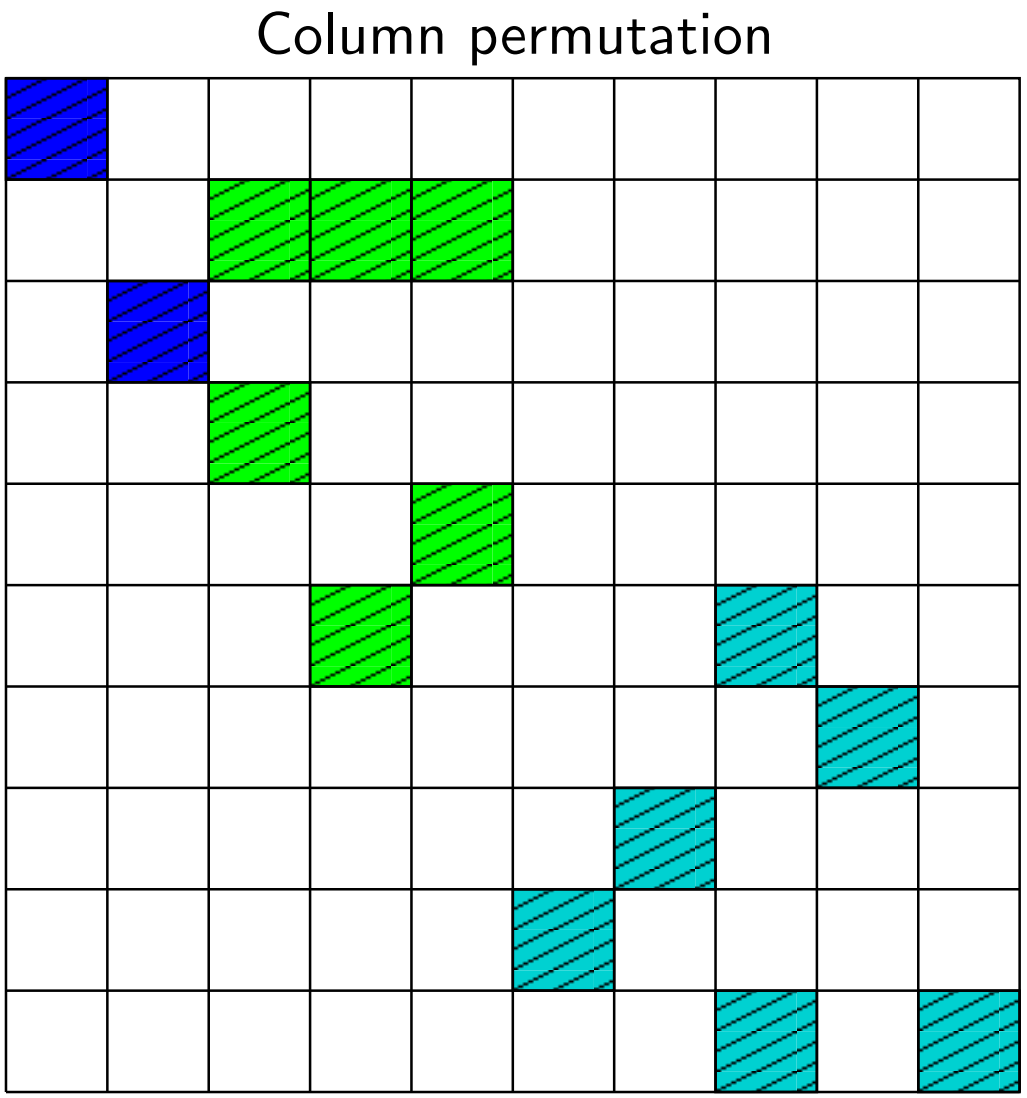
Permuting to SBD form



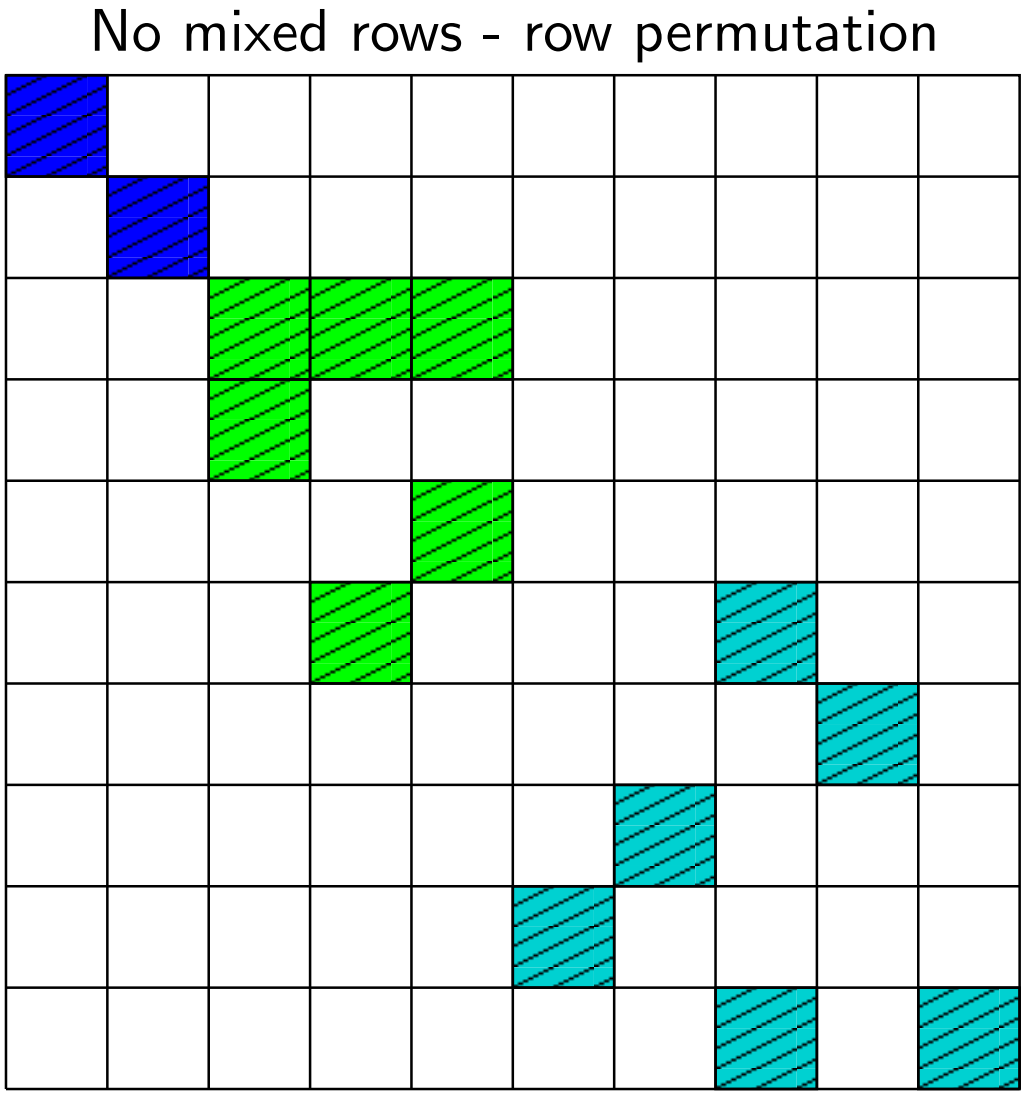
Permuting to SBD form



Permuting to SBD form

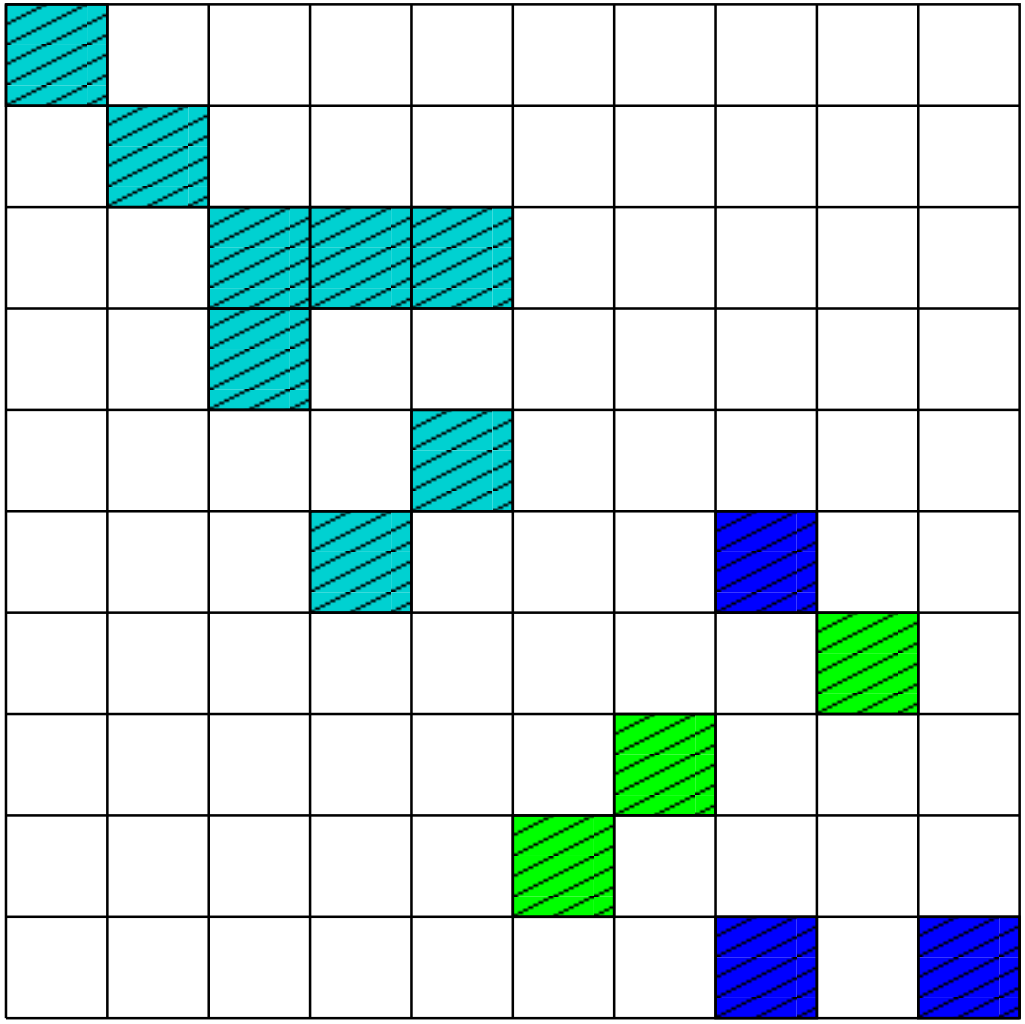


Permuting to SBD form



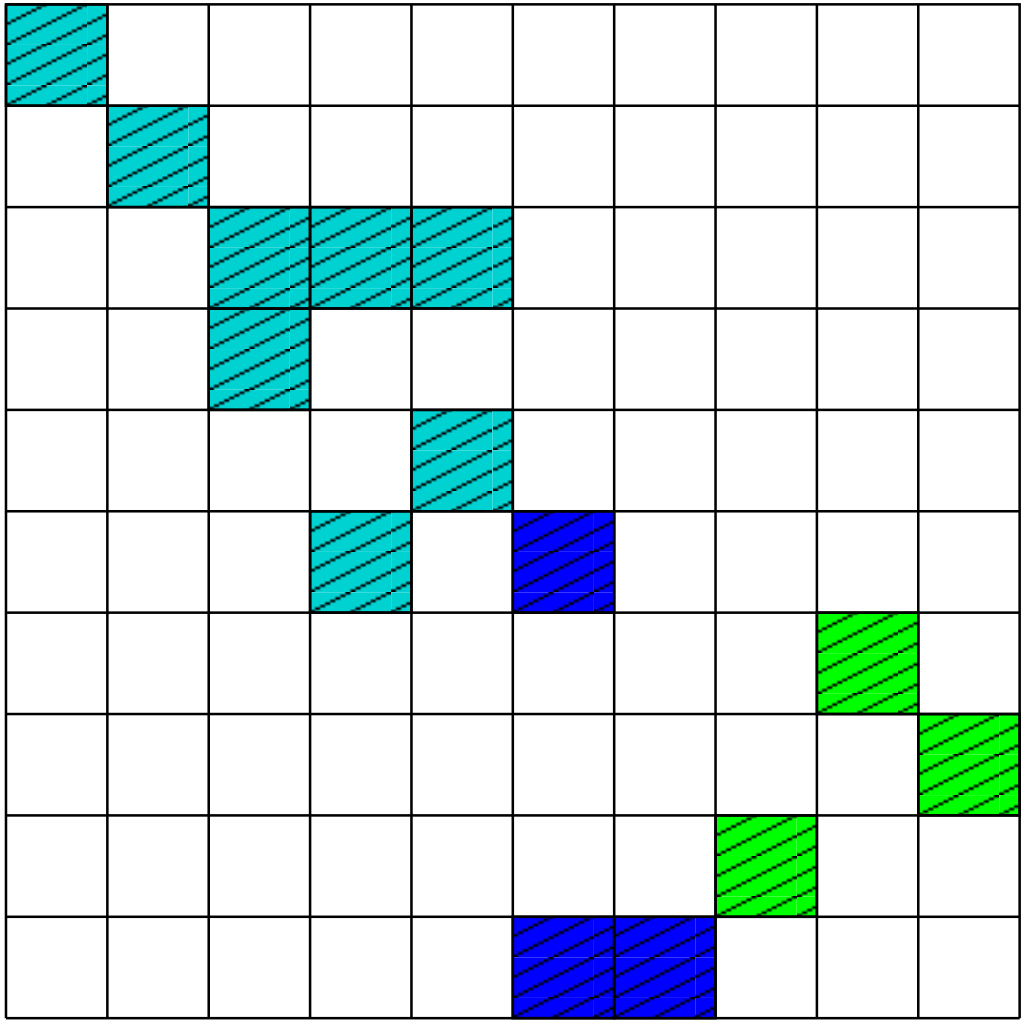
Permuting to SBD form

Continued



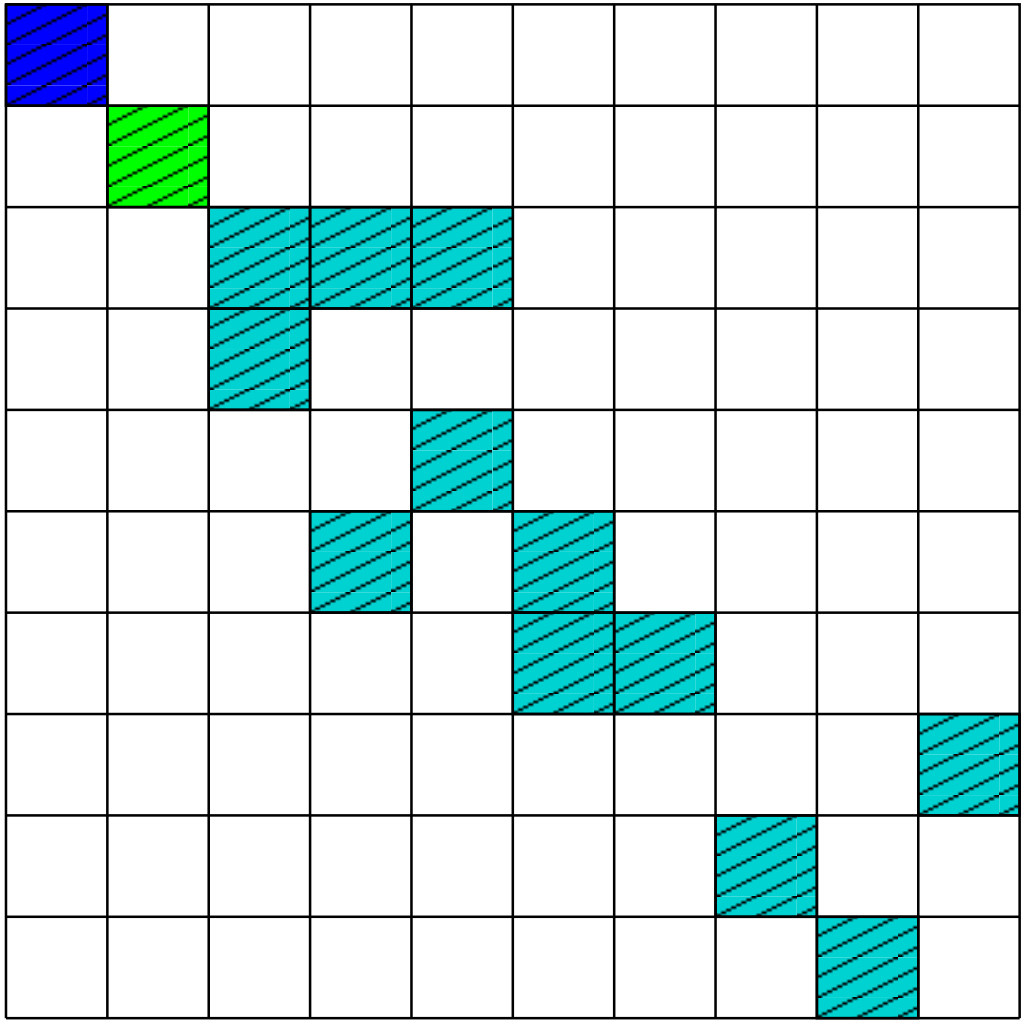
Permuting to SBD form

Continued



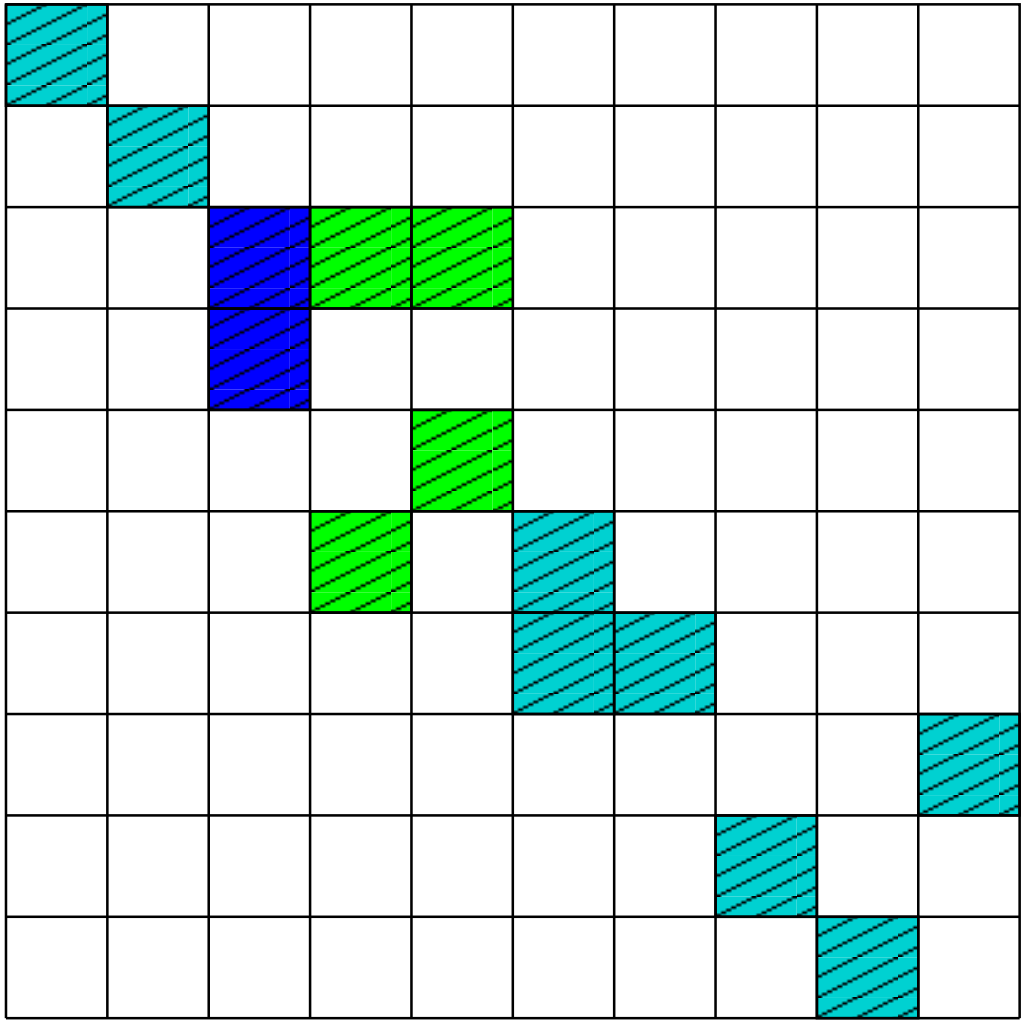
Permuting to SBD form

Continued



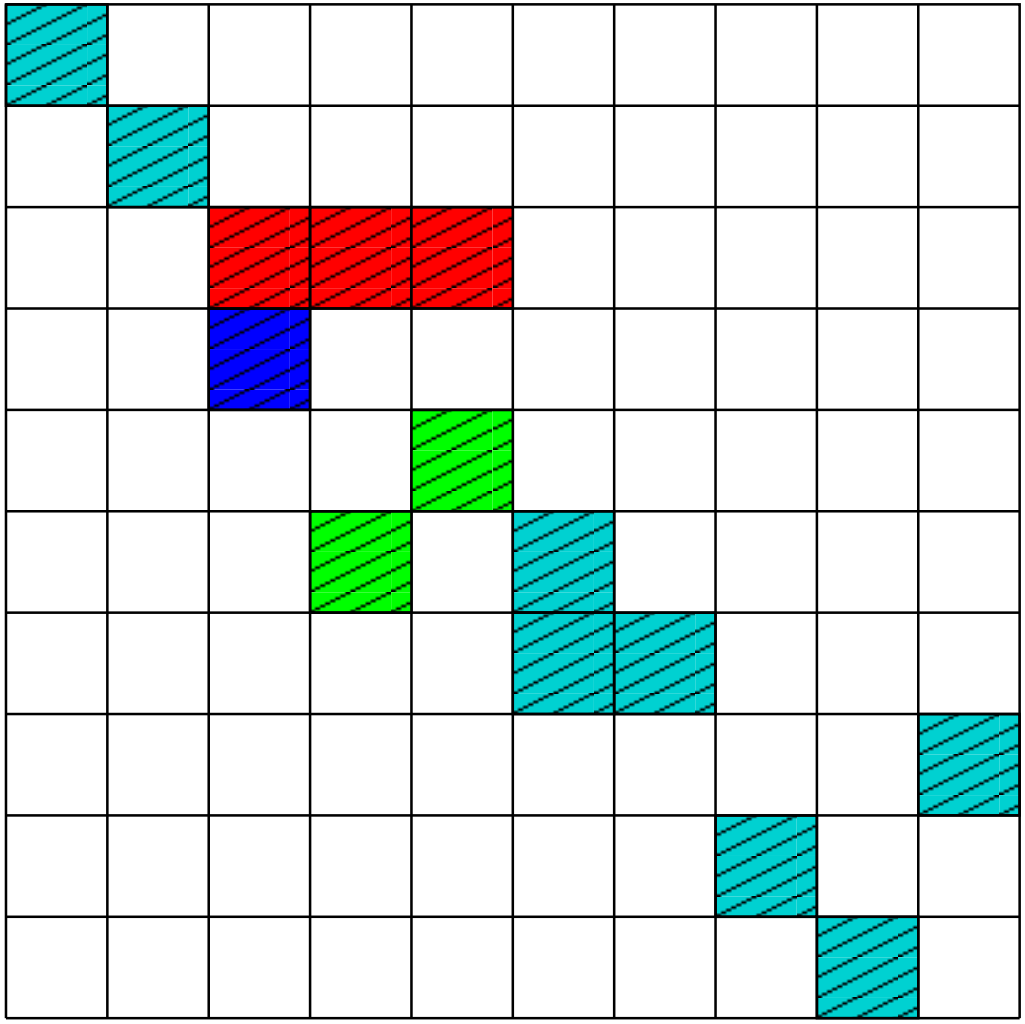
Permuting to SBD form

Continued



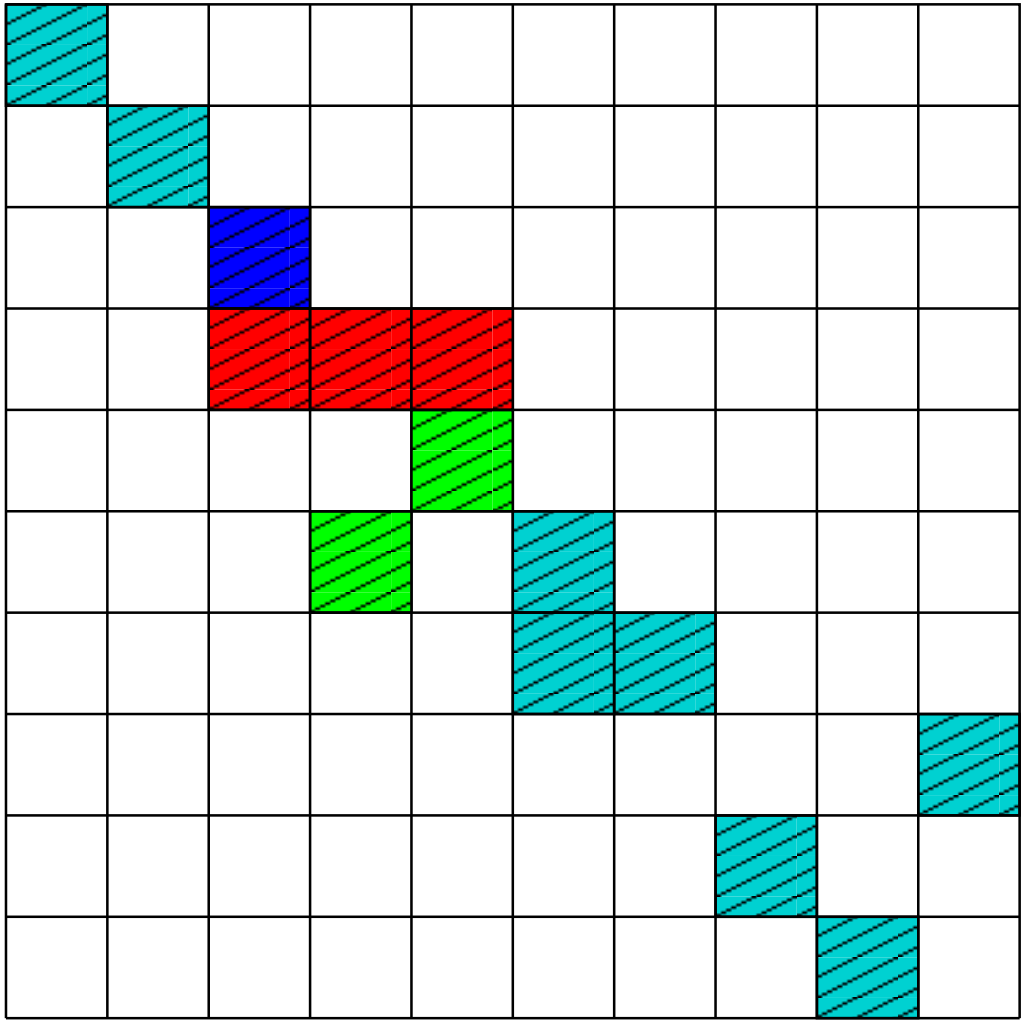
Permuting to SBD form

Continued



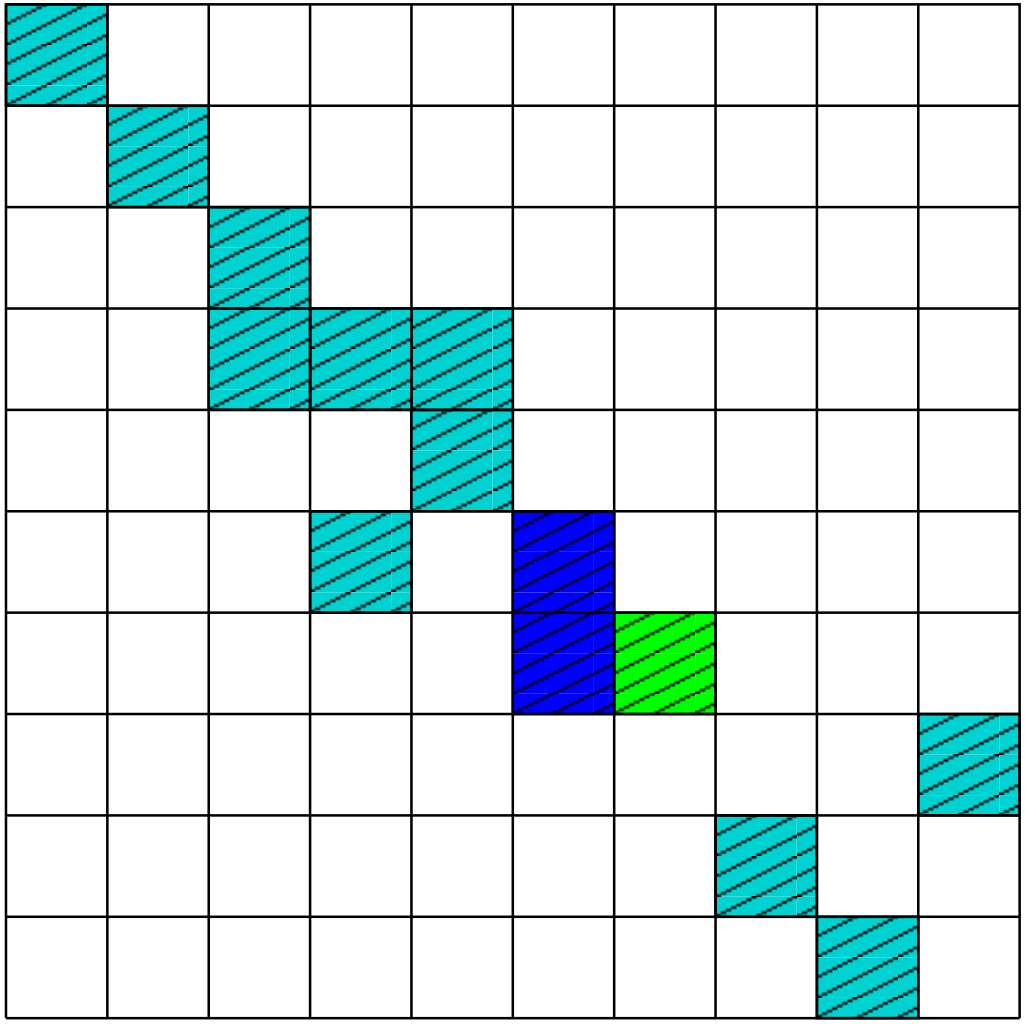
Permuting to SBD form

Continued



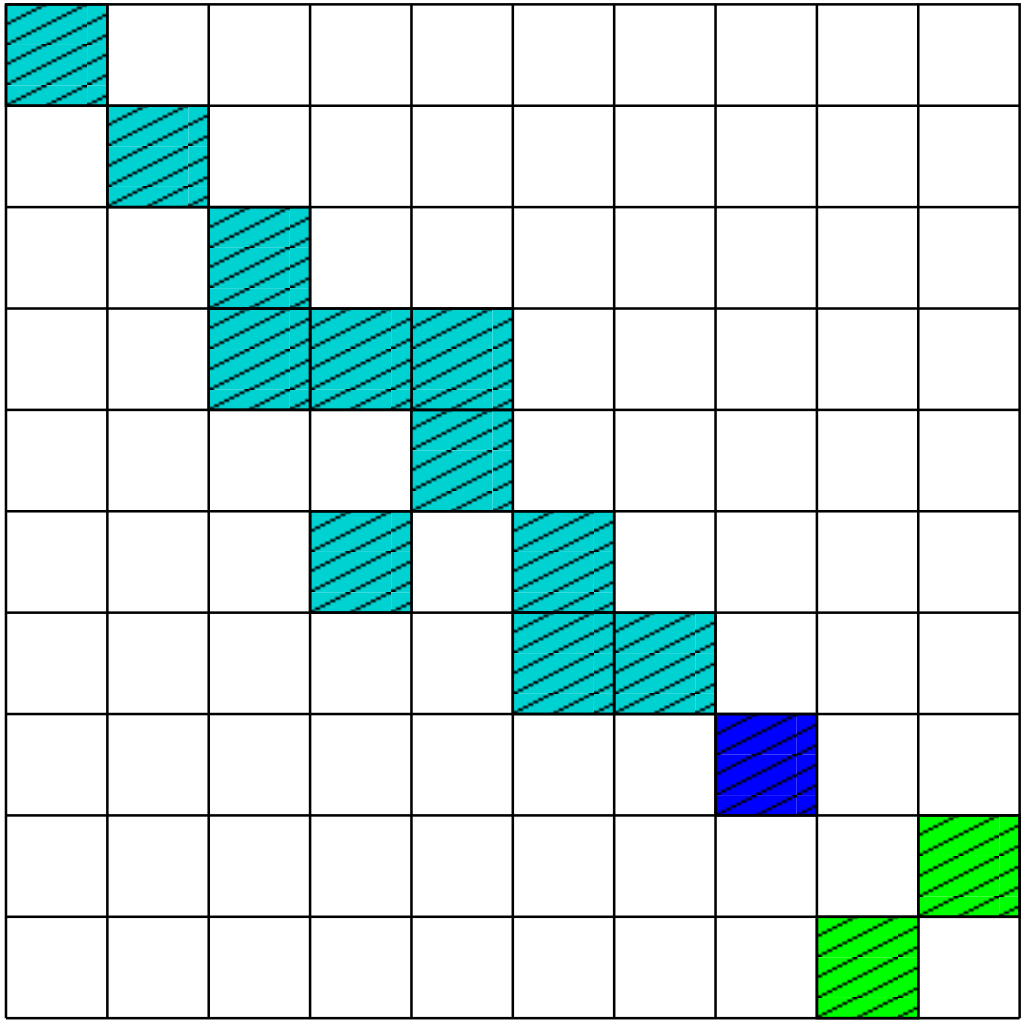
Permuting to SBD form

Continued



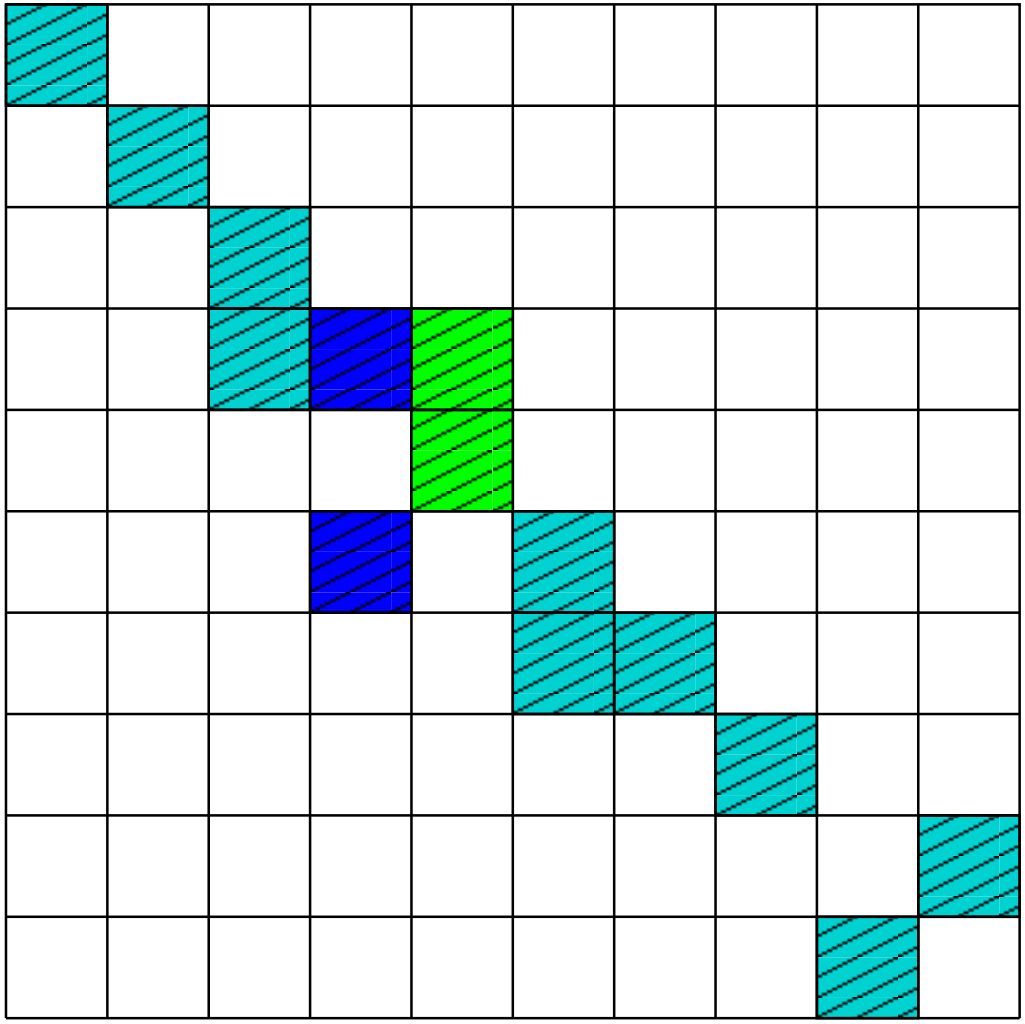
Permuting to SBD form

Continued



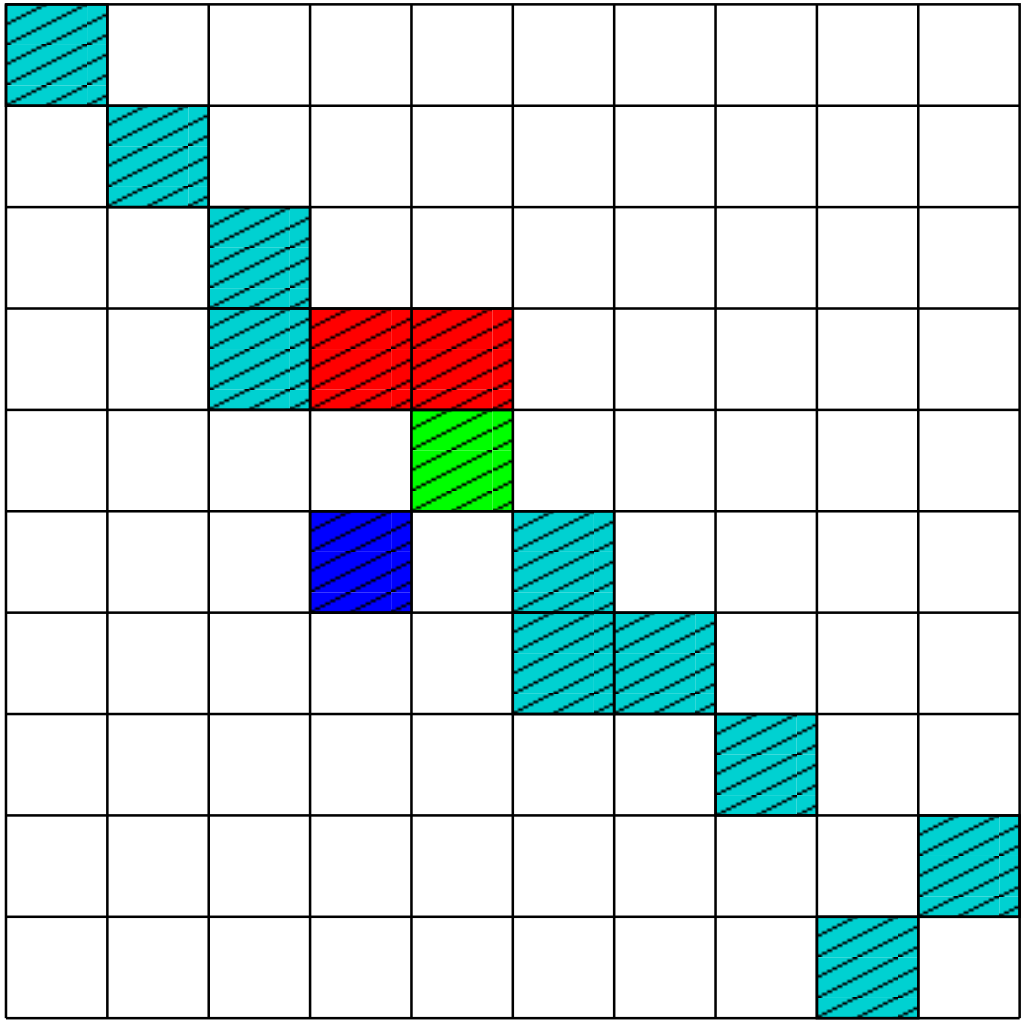
Permuting to SBD form

Continued



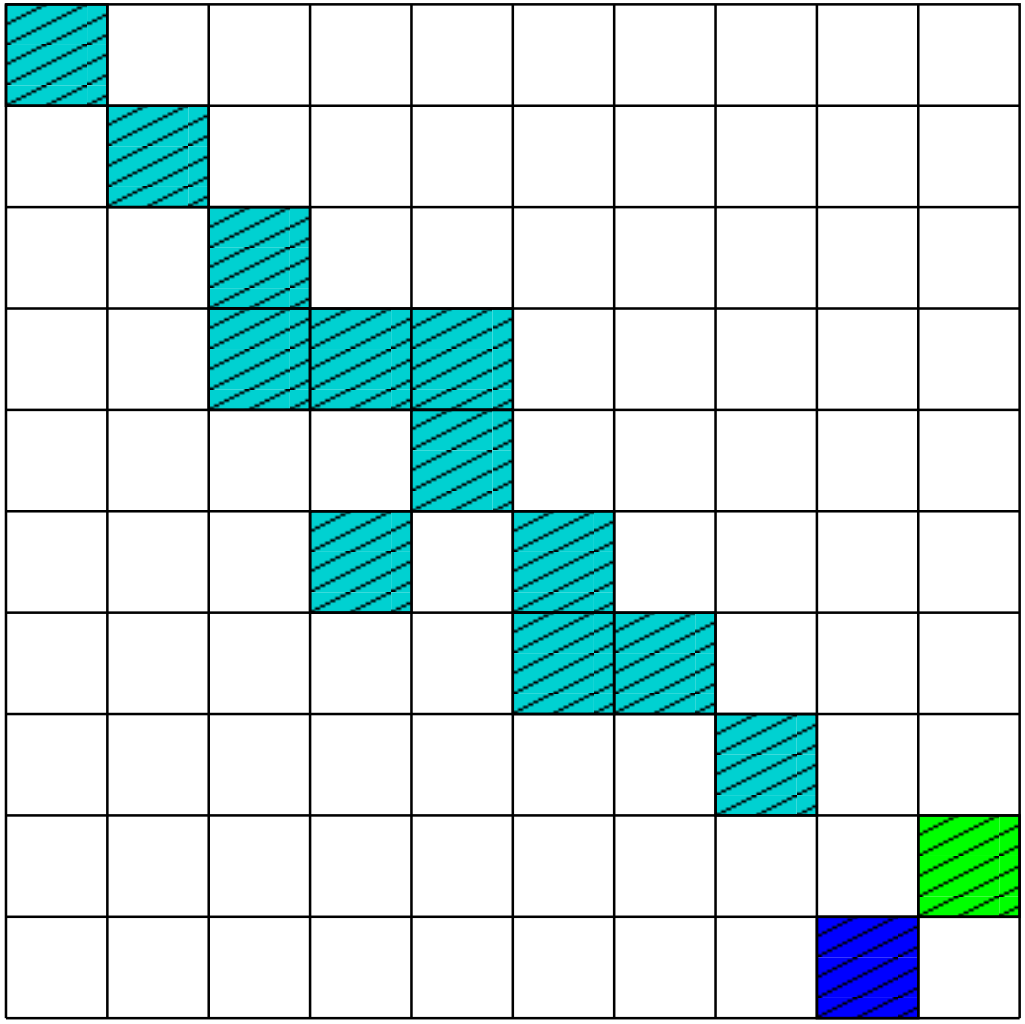
Permuting to SBD form

Continued



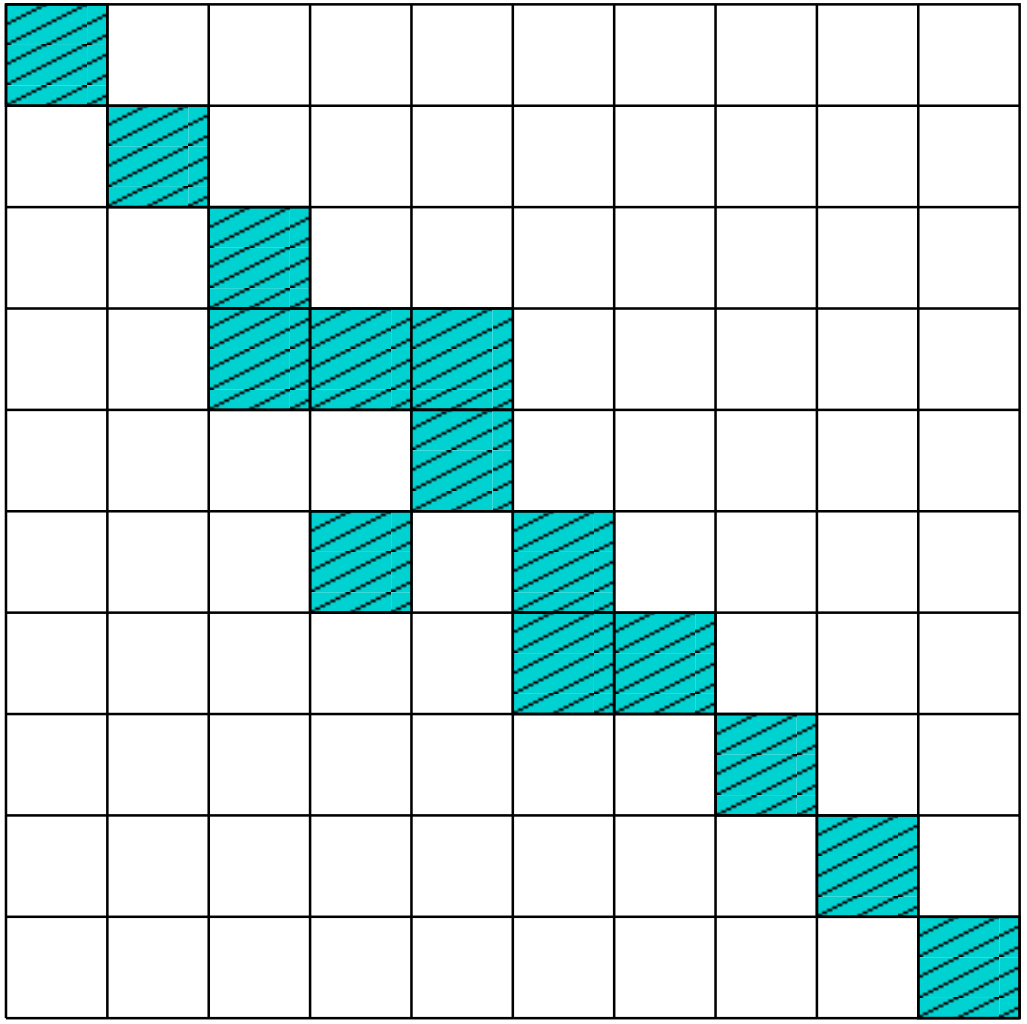
Permuting to SBD form

Continued



Permuting to SBD form

Continued



Cache-aware approach

For $p = \frac{n}{wL}$, the number of cache misses is given by

$$\sum_{i:\ n_i \in \mathcal{N}} (\lambda_i - 1), \quad \text{assuming zig-zag CRS,}$$

where n is the length of the input vector, w is the number of elements which fit into a single cache line, and L is the number of available cache lines.
The $(\lambda - 1)$ *metric* is already used extensively in parallel computing; in particular during *parallel* SpMV multiplication. Partitioners designed to that end, also take into account a *load-imbalance* ϵ.

References:

- Çatalyürek and Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, 1999
- Vastenhouw and Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, 2005

Cache-oblivious approach

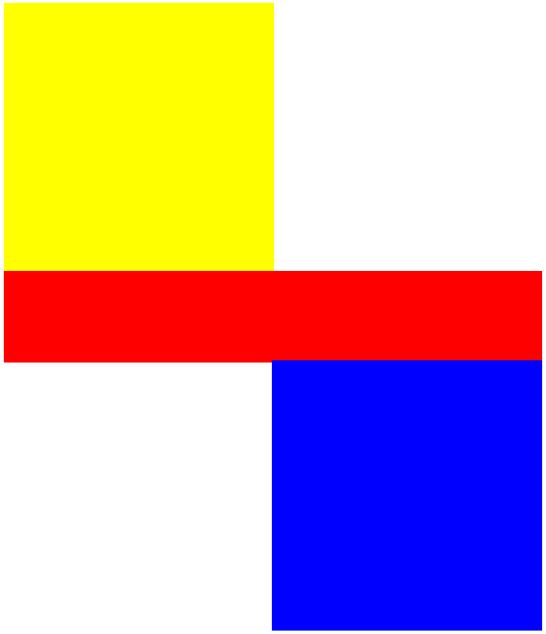
Taking the number of subsets p we partition the columns into as many parts as possible,

$$p \to \infty \quad (p = n).$$

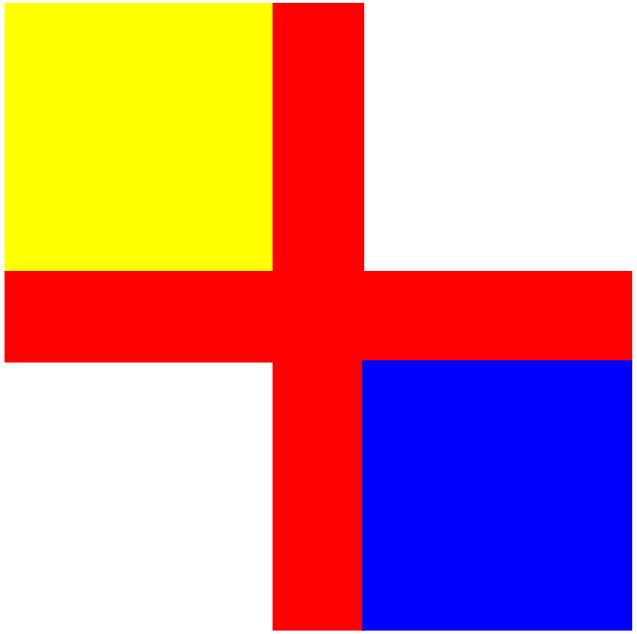
This does not harm efficiency compared to the optimal p, and will inherently optimise for the smaller caches closer to the CPU; that is, *we optimise access for all cache levels irrespective of the cache parameters*.

Moving to two dimensions

Two-dimensional SBD (doubly separated block diagonal)



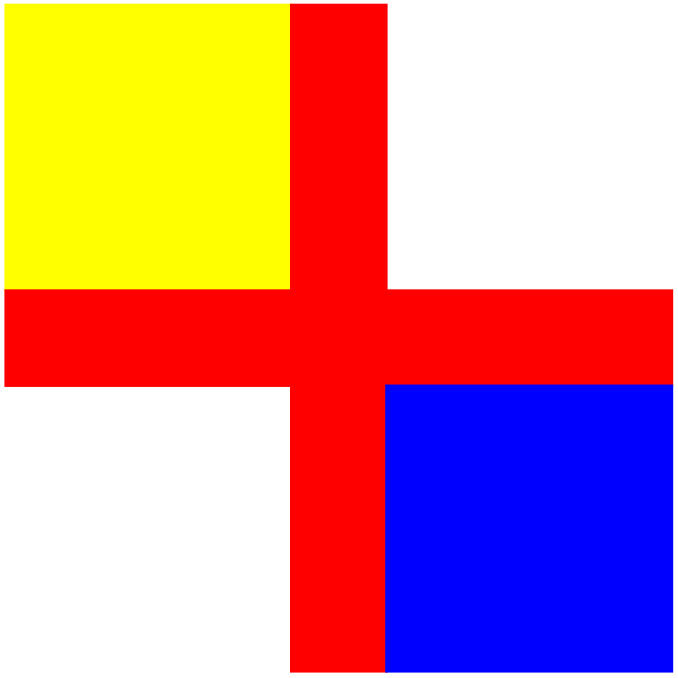
1D



2D

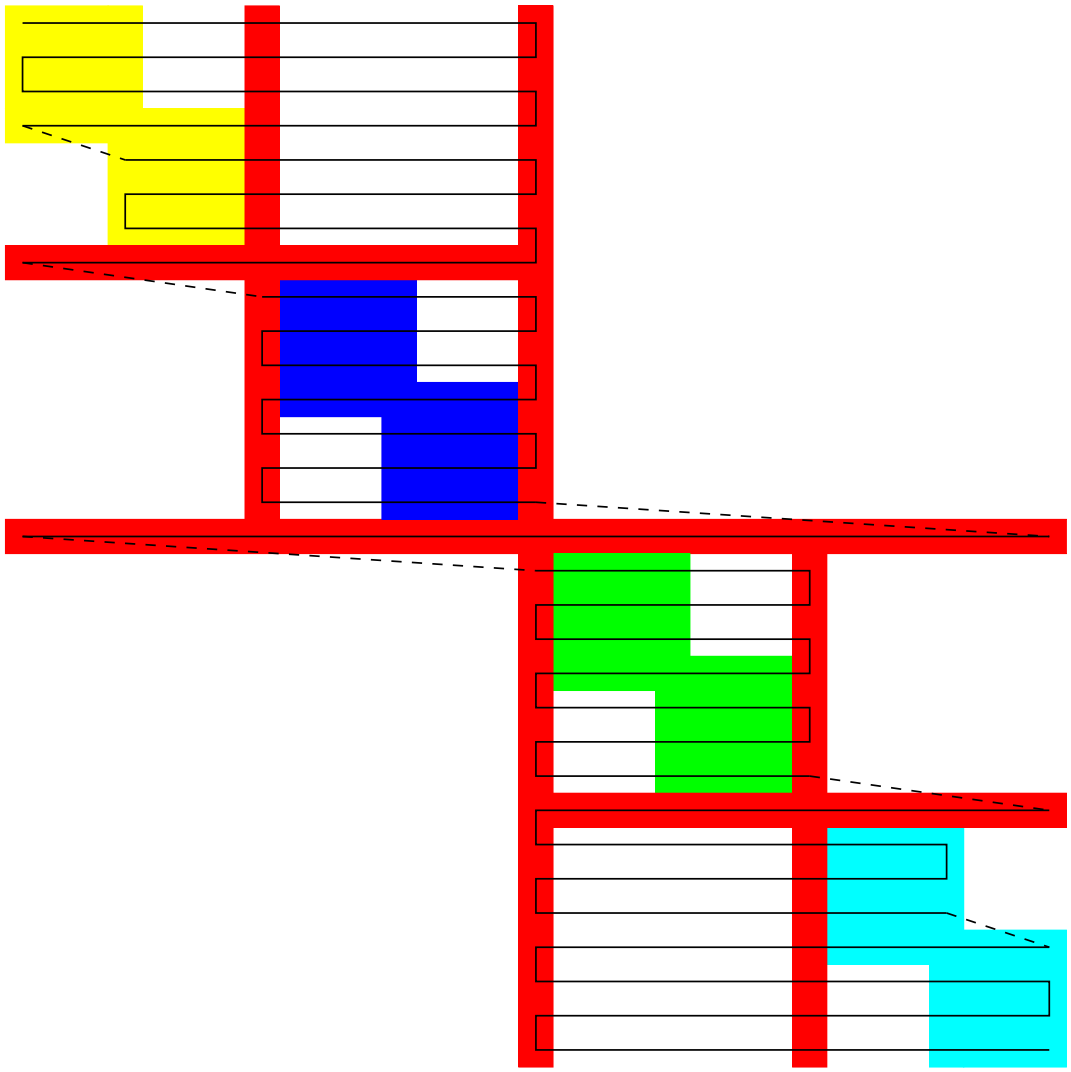
Two-dimensional SBD (doubly separated block diagonal)

Using a fine-grain model of the input sparse matrix, individual nonzeros each correspond to a vertex;
each row and column have a corresponding net.



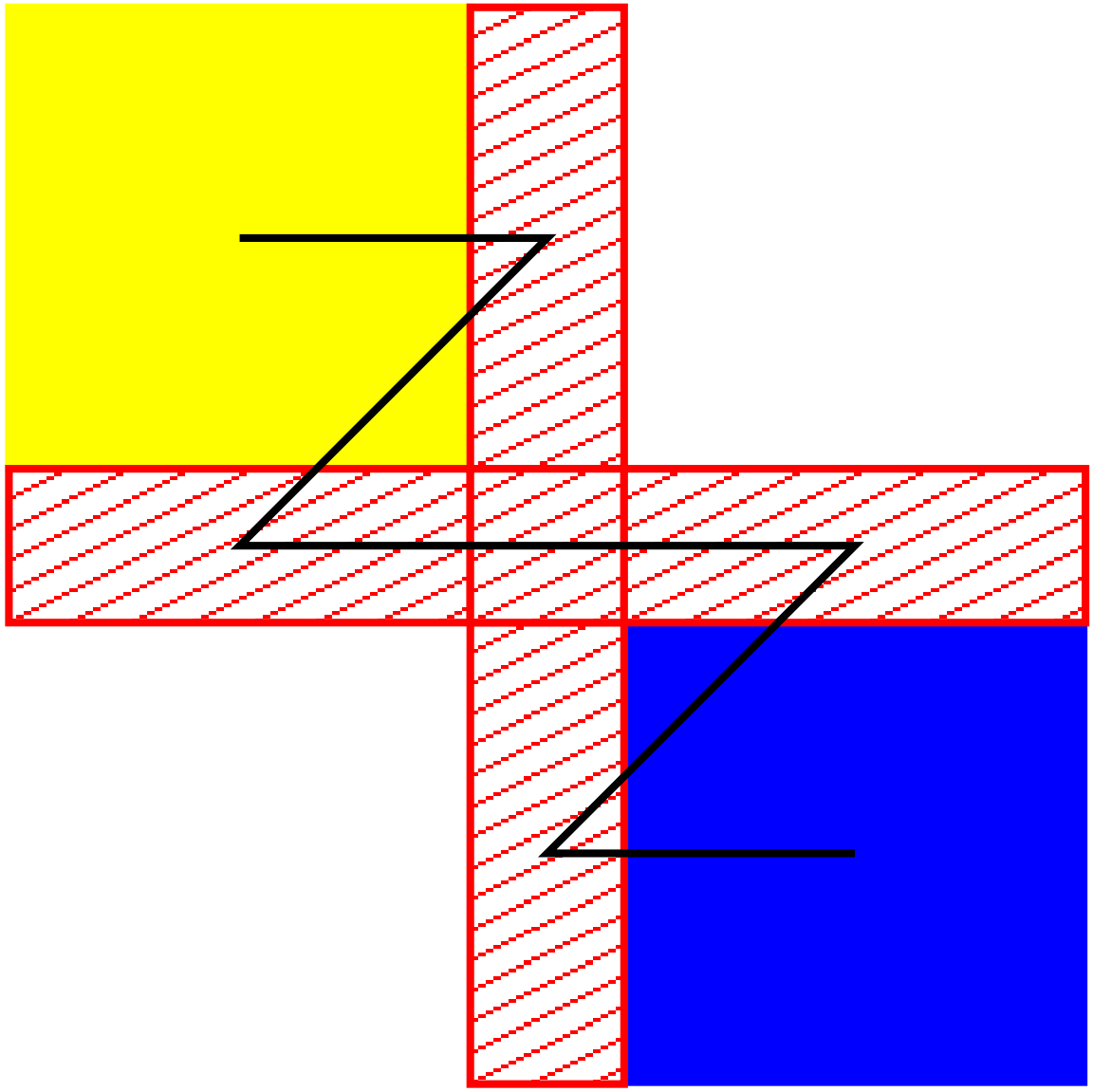
The quantity minimised remains $\sum_i(\lambda_i - 1)$.

Two-dimensional SBD (doubly separated block diagonal)

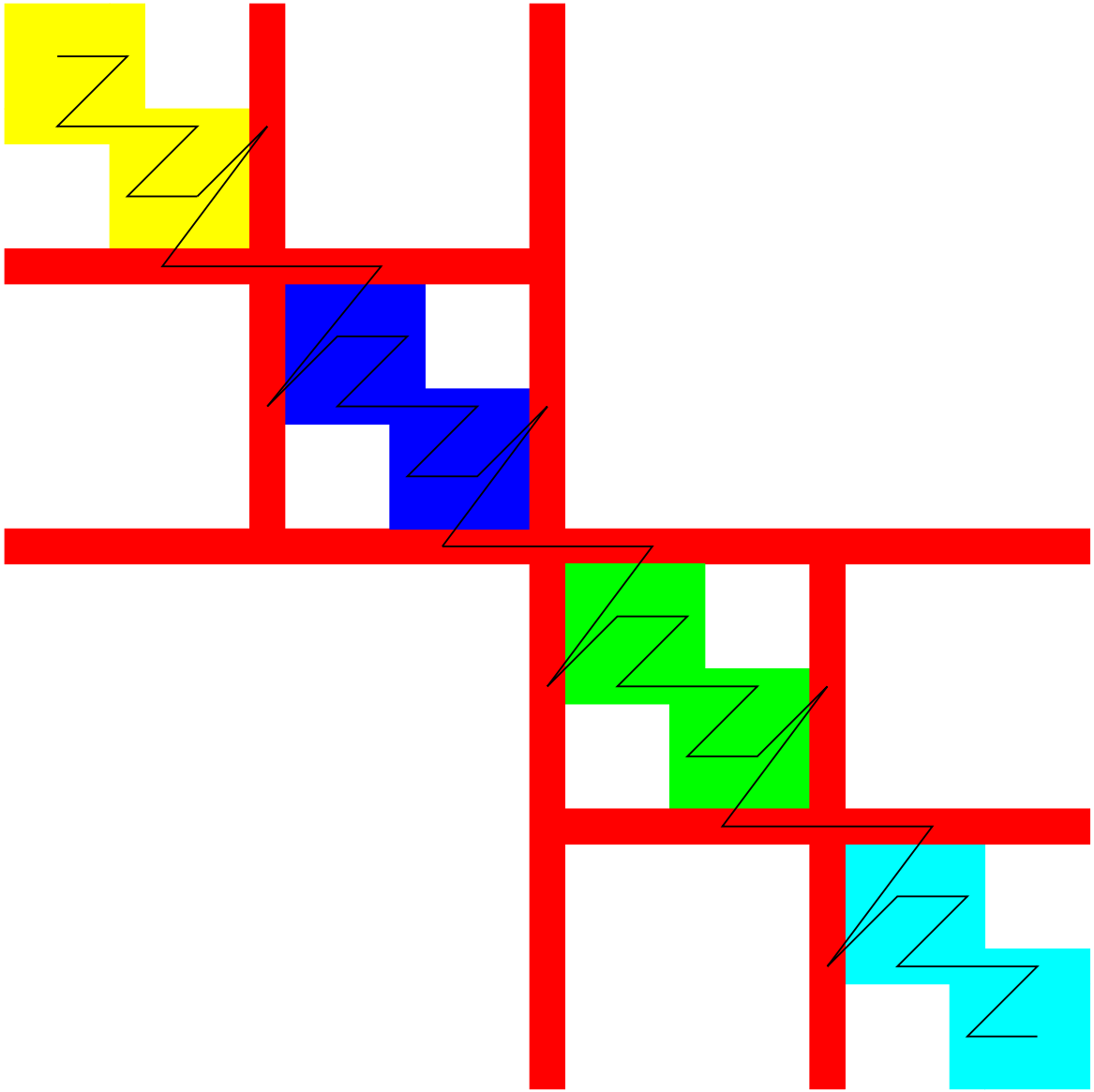


Zig-zag CRS is not suitable for handling 2D SBD!

Two-dimensional SBD; block ordering

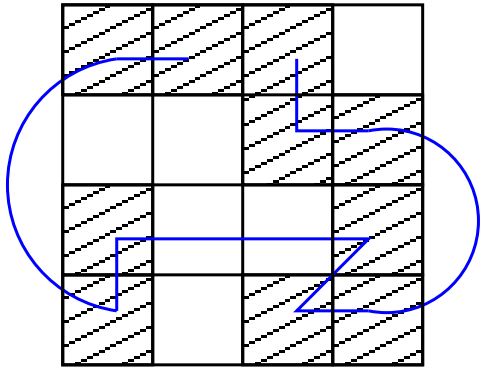


Two-dimensional SBD; block ordering



Bi-directional Incremental CRS (BICRS)

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

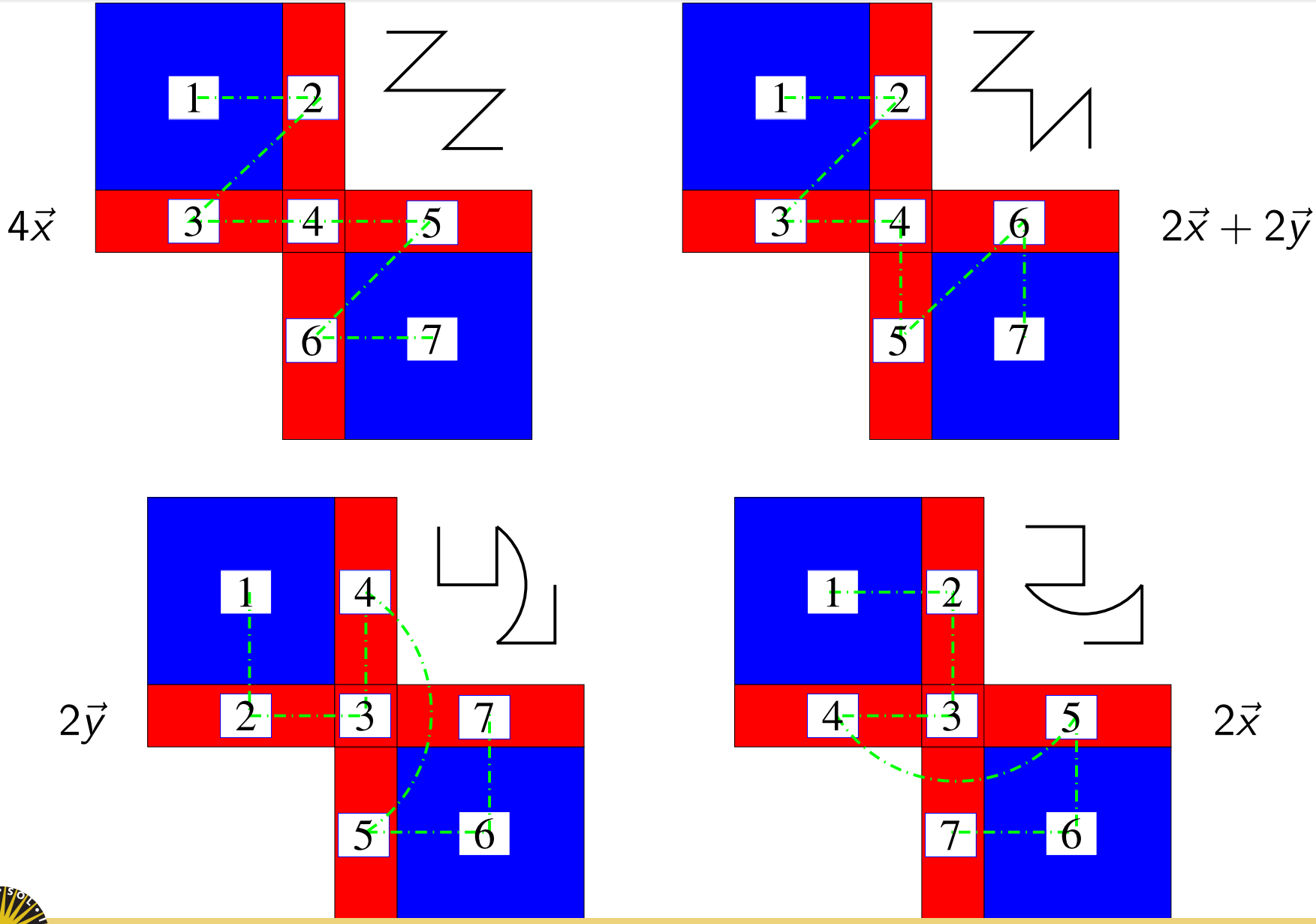


Stored as:

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]

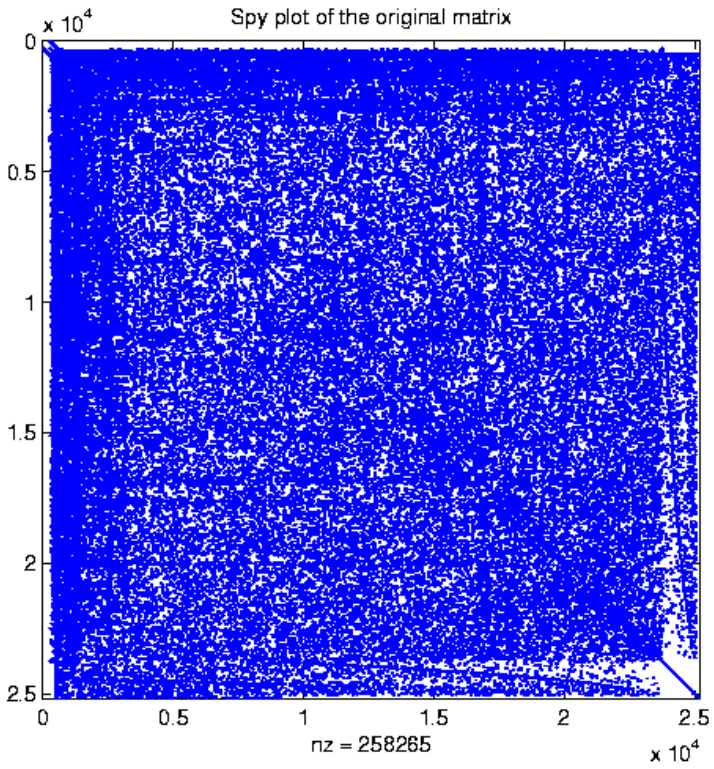
, $2nnz + (row\_jumps + 1)$ accesses

Two-dimensional SBD; block ordering

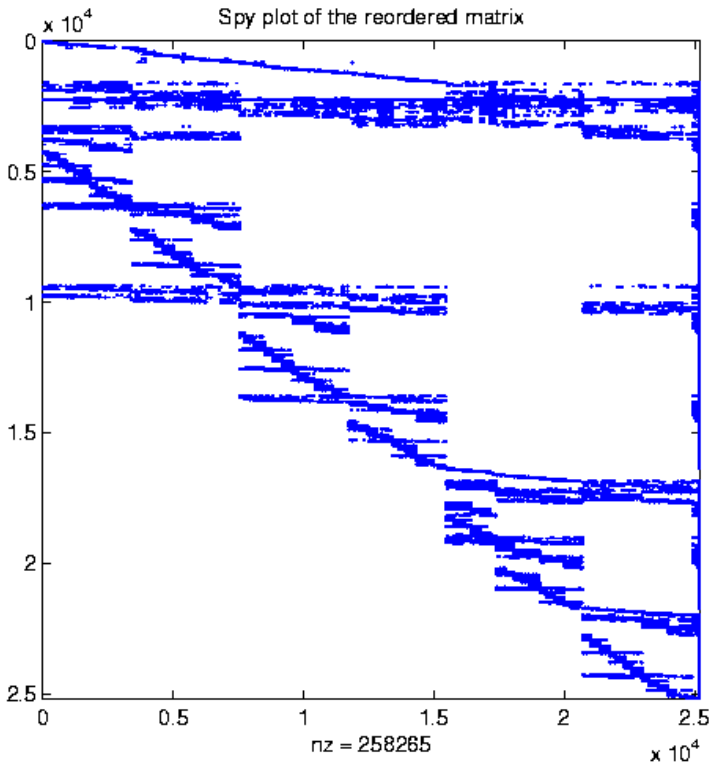


Experimental results

The rhpentium matrix – 1D

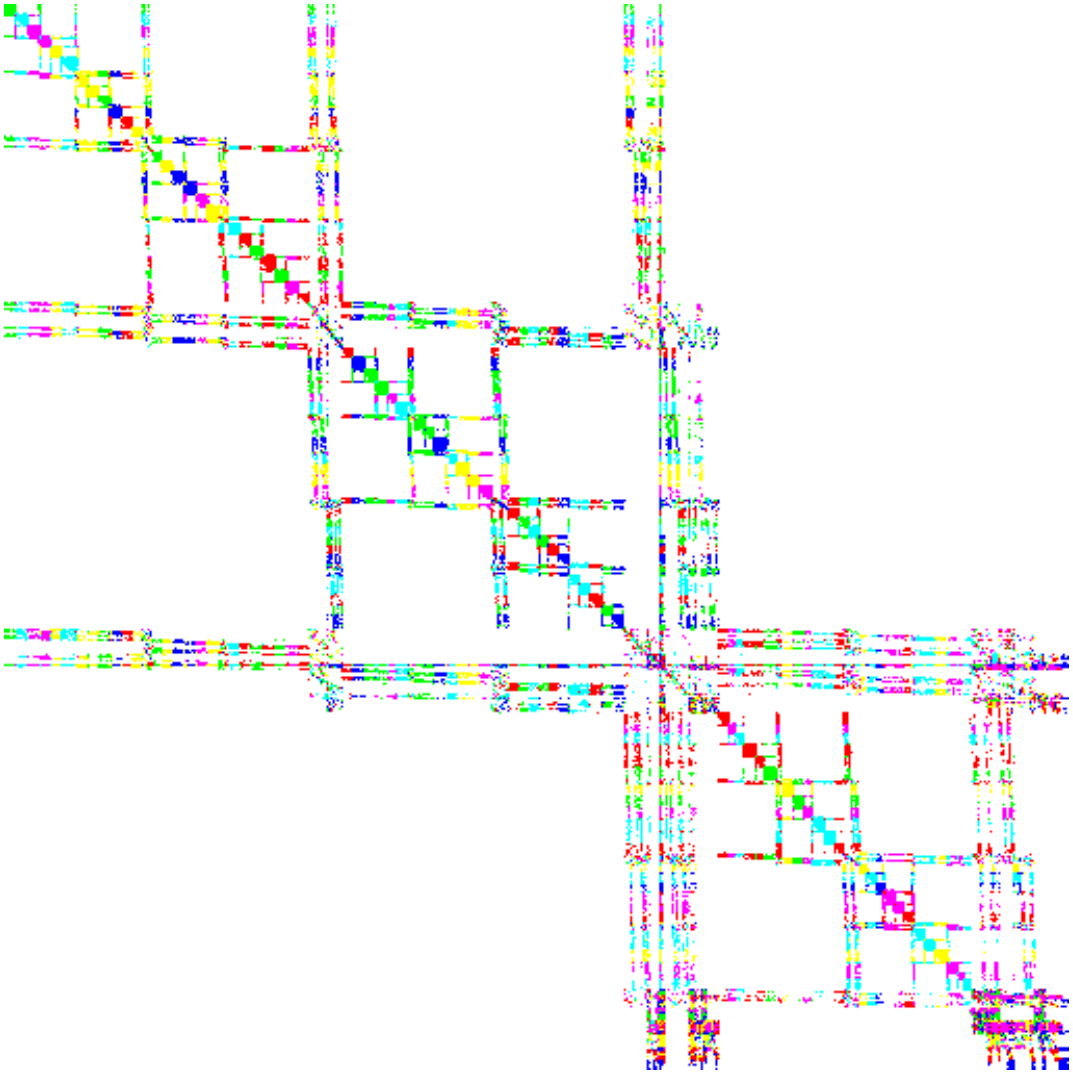


$p = 1$ (original)



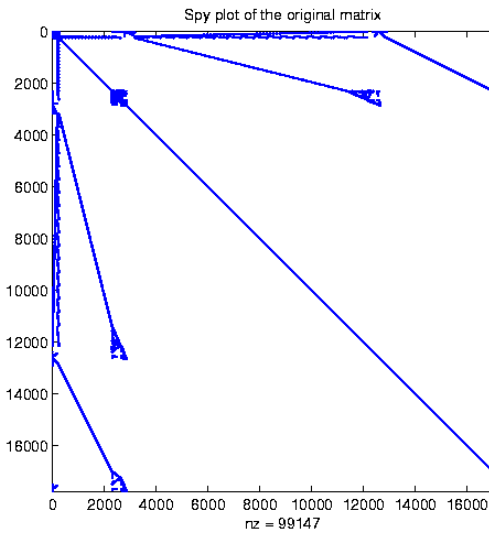
$p = 100, \quad \epsilon = 0.1$

The rhpentium matrix – 2D

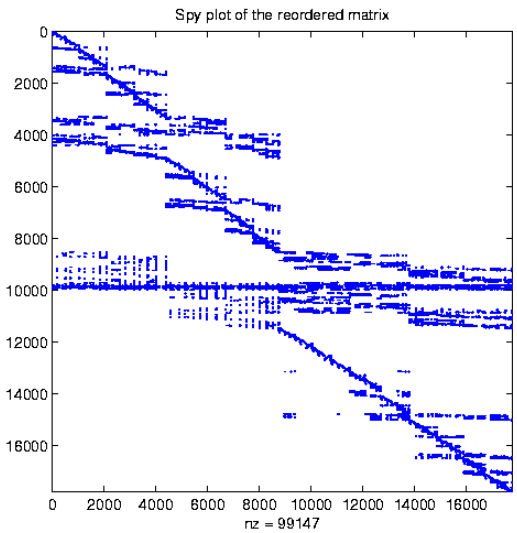


$p = 100, \quad \epsilon = 0.1$

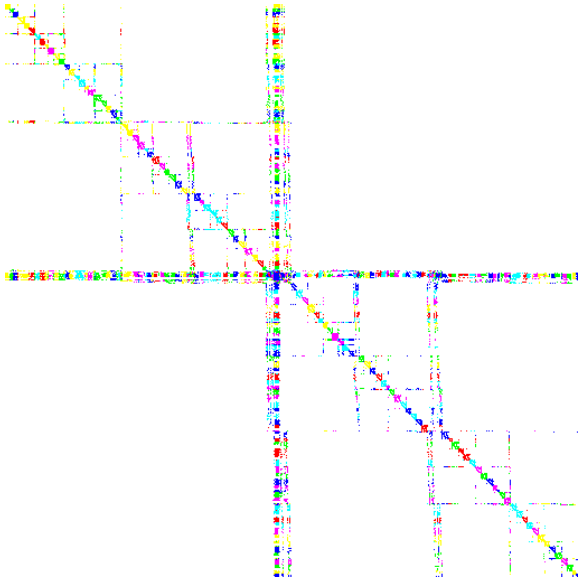
The memplus matrix



Original

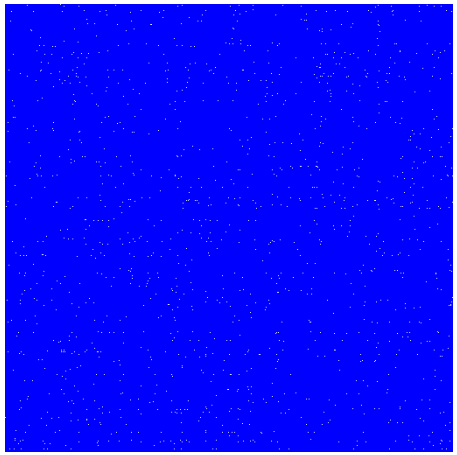


1D, $p = 100$, $\epsilon = 0.1$

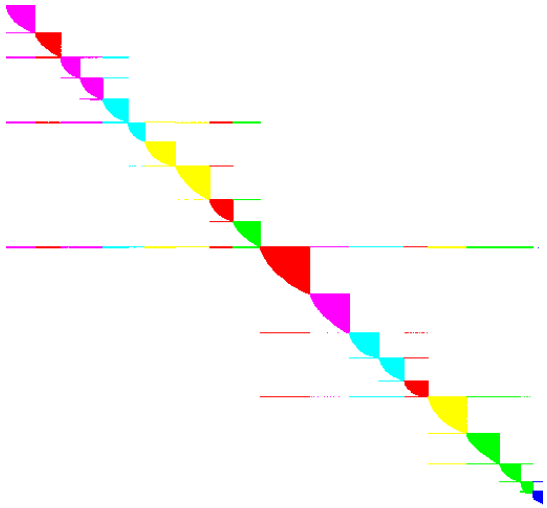


2D, $p = 100$, $\epsilon = 0.1$

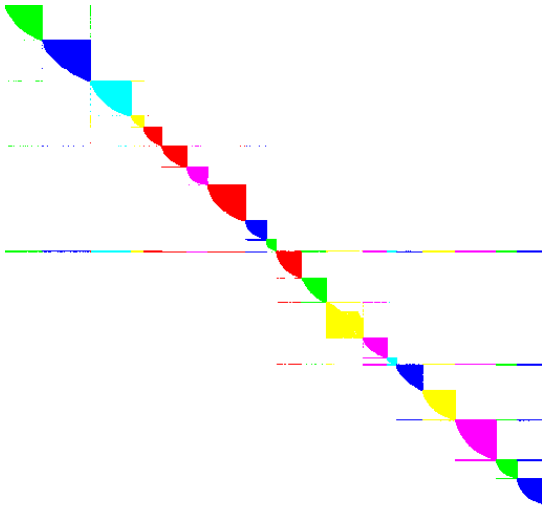
The Stanford link matrix



Original

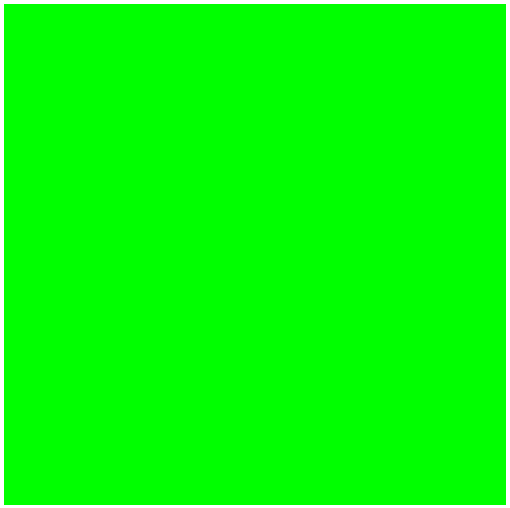


1D ($p = 20$, $\epsilon = 0.1$)

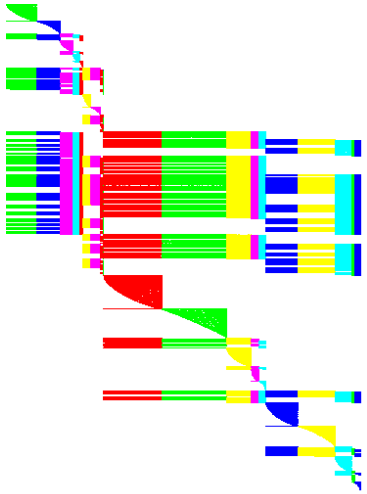


Finegrain ($p = 20$, $\epsilon = 0.1$)

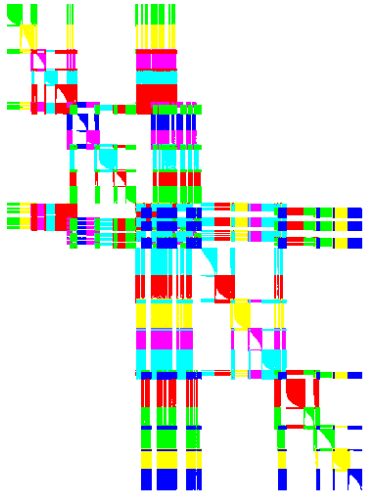
The Wikipedia 2006 link matrix



Original

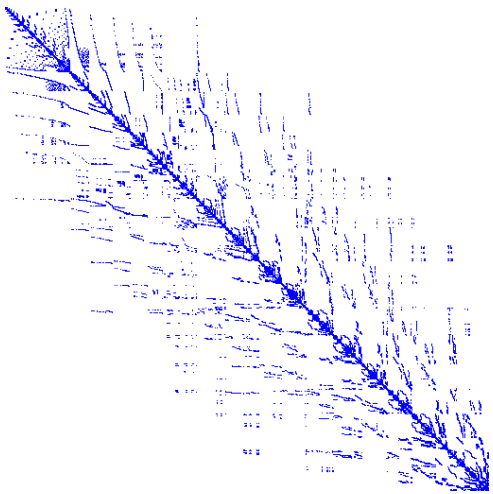


1D ($p = 20$, $\epsilon = 0.1$)

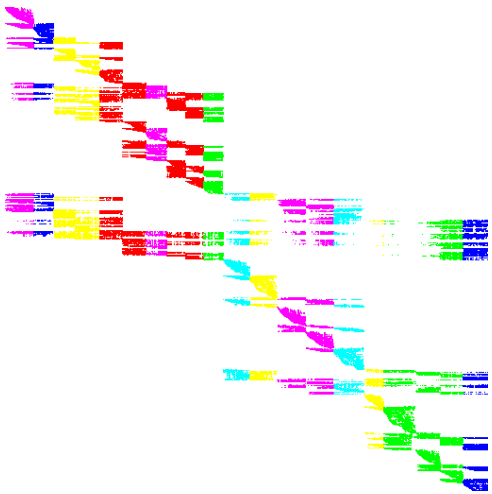


Finegrain ($p = 20$, $\epsilon = 0.1$)

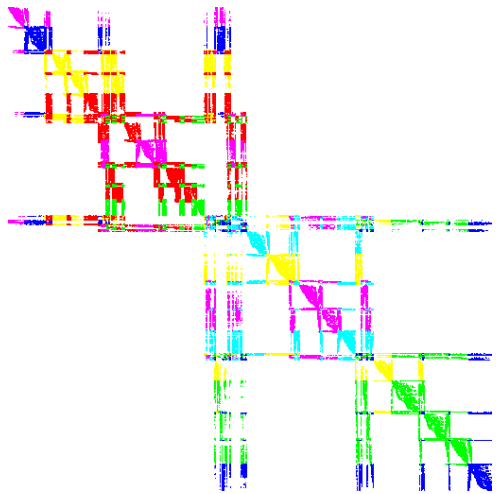
The cage14 matrix



Original



1D ($p = 20$, $\epsilon = 0.1$)



Finegrain ($p = 20$, $\epsilon = 0.1$)

Pre-processing times

1D

Matrix	Reordering time	SpMV time (old/new)
rhpentium, $p = 400$:	1 minute	(0.9 / 0.7 ms.)
memplus, $p = 100$:	4 minutes	(0.4 / 0.4 ms.)
stanford, $p = 20$:	4 minutes	(27 / 14 ms.)
cage14, $p = 20$:	12 minutes	(109 / 123 ms.)
stanford_berkeley, $p = 20$:	13 minutes	(31 / 30 ms.)
wikipedia, $p = 10$:	16 minutes	(353 / 236 ms.)
wikipedia, $p = 20$:	45 minutes	(353 / 237 ms.)

Old: SpMV on the original matrix A
New: SpMV on the reordered matrix PAQ

Results from 2009, run on an Intel Q6600

Pre-processing times

2D

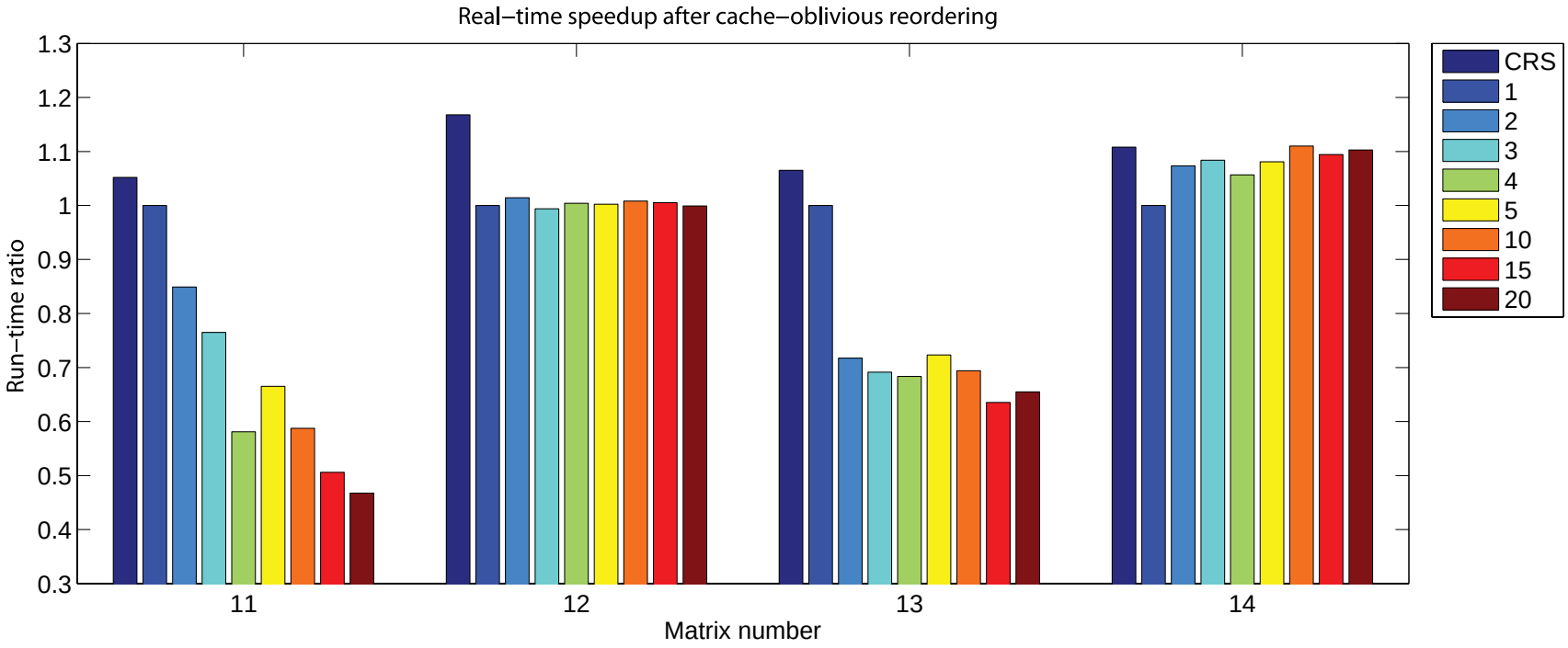
Matrix	Reordering	SpMV time (old/new)
rhpentium, $p = 400$:	37 sec.	0.39/0.41 ms. (ICRS/CRS)
memplus, $p = 100$:	9 sec.	0.88/0.62 ms. (ICRS/ICRS)
stanford, $p = 20$:	3 min.	20.84/8.30 ms. (CRS/BICRS)
cage14, $p = 20$:	29 min.	65.67/79.75 ms. (ICRS/ICRS)
stanford_berkeley, $p = 20$:	15 min.	22.35/21.28 ms. (ICRS/BICRS)
wikipedia05, $p = 20$:	72 min.	280.9/129.7 ms. (CRS/BICRS)
wikipedia06, $p = 20$:	3.5 hours	790.2/273.9 ms. (ICRS/BICRS)

Old: SpMV on the original matrix A

New: SpMV on the reordered matrix PAQ

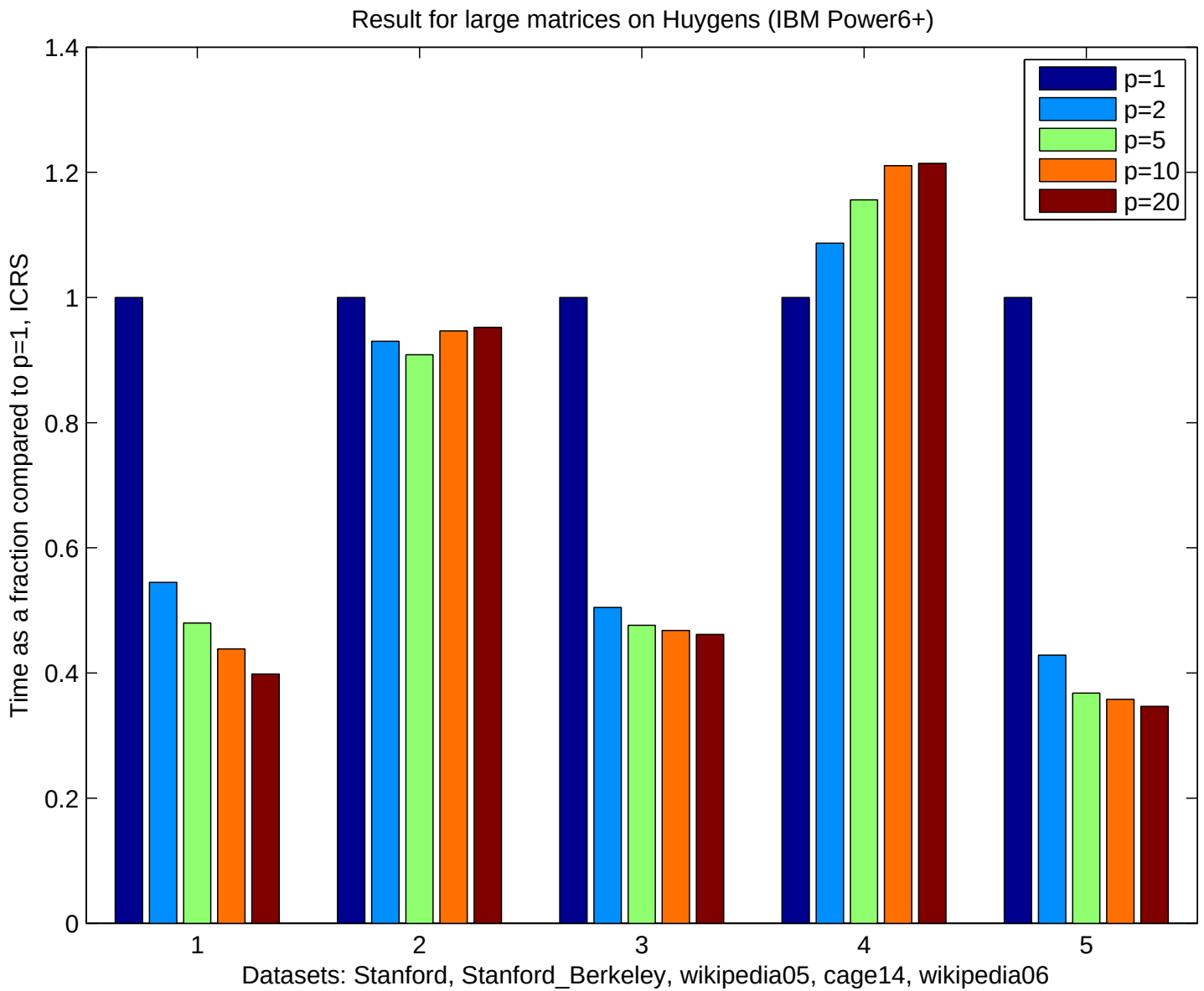
Fresh results. Reordering done on an Opteron 2378, SpMVs on an IBM Power 6 (Huygens).

IBM Power6+ (Huygens), 1D



Matrices used are: 11. Stanford, 12. Stanford_Berkeley, 13. wikipedia20051105, and 14. cage14.

IBM Power6+ (Huygens), 2D



Software

The latest version (3.0) of the sparse matrix partitioner software *Mondriaan* natively supports both the 1D and 2D reordering methods described here. This version has just been released (27th of July), and can be found at:

`http://www.math.uu.nl/people/bisseling/Mondriaan`

Conclusions:

We have introduced a scheme capable of increasing SpMV performance up to a factor two, while:

- remaining cache-oblivious,
- keeping open the option of using specialised external libraries (1D),
- being able to easily introduce (distributed or shared-memory) parallelism into the SpMV.

For already well-structured matrices our approach does not obtain significant speedups, but does not decrease performance much either.

The two-dimensional method promises even greater speedups, but still requires more finetuning and experimenting.