# Cache-oblivious sparse matrix–vector multiplication

Albert-Jan Yzelman

April 3, 2009

Joint work with Rob Bisseling

# Motivations

- Basic implementations can suffer up to 2x slowdown.
- Even worse: dedicated libraries may in some cases still show a similar level of inefficiency.

# Outline



# Memory and multiplication



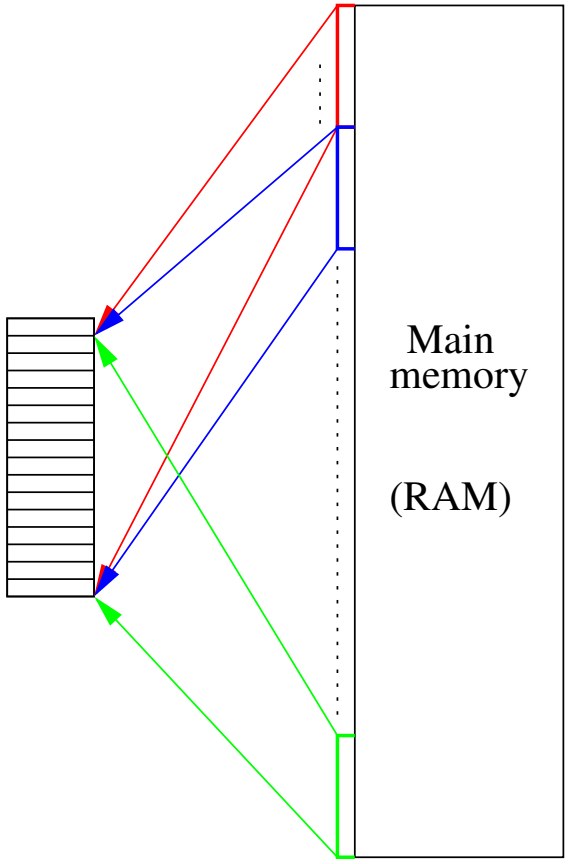
# Cache parameters

- Size $S$ (in bytes)
- Line size $L_S$ (bytes)
- Number of cache lines $L = (S/L_S)$
- Number of subcaches $k$
- Number of levels

# Naive cache

$k = 1$, modulo mapped cache

Memory (of length $L_S$) from RAM with start address $x$ is stored in cache line number $x \mod L$:



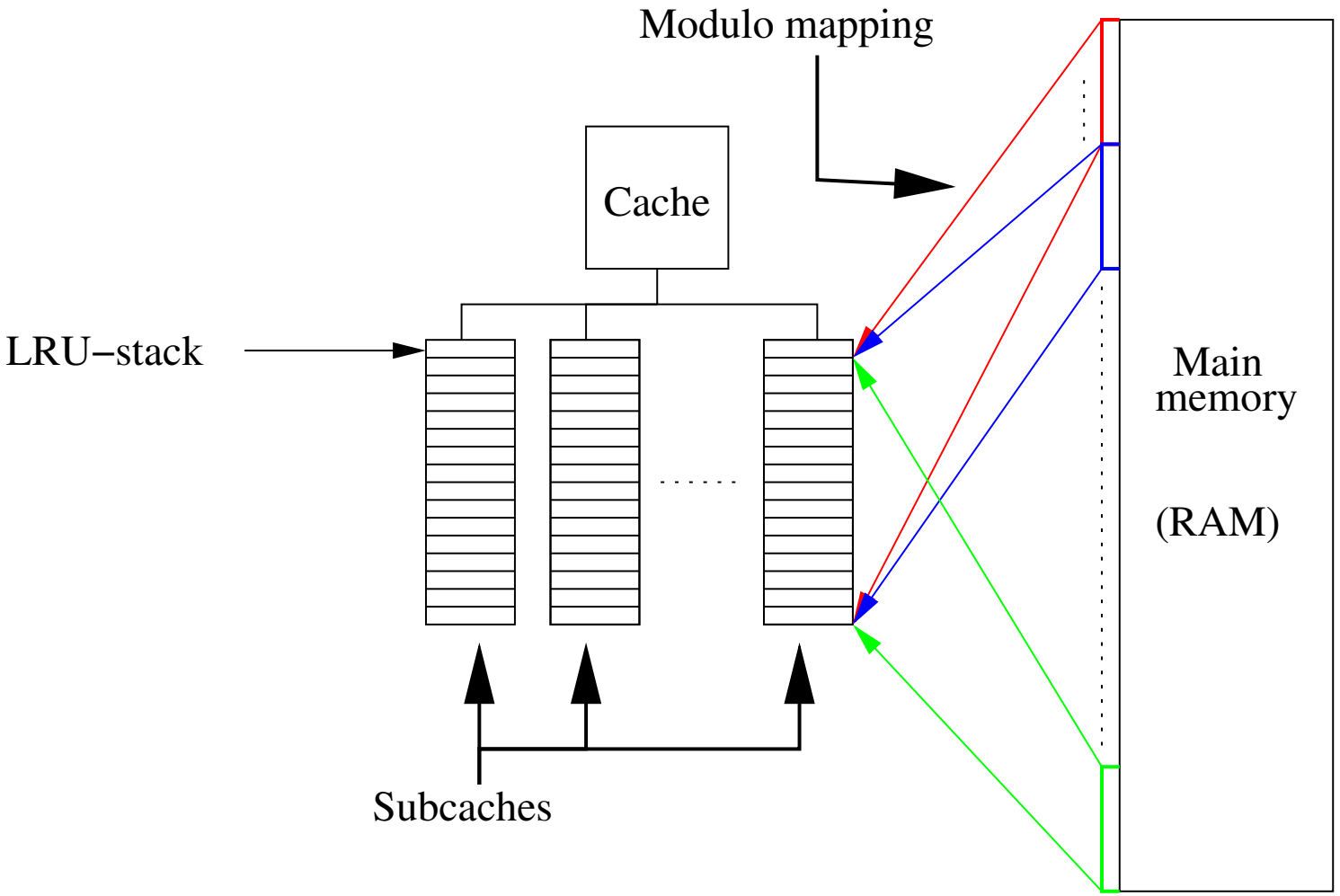
# 'Ideal' cache

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

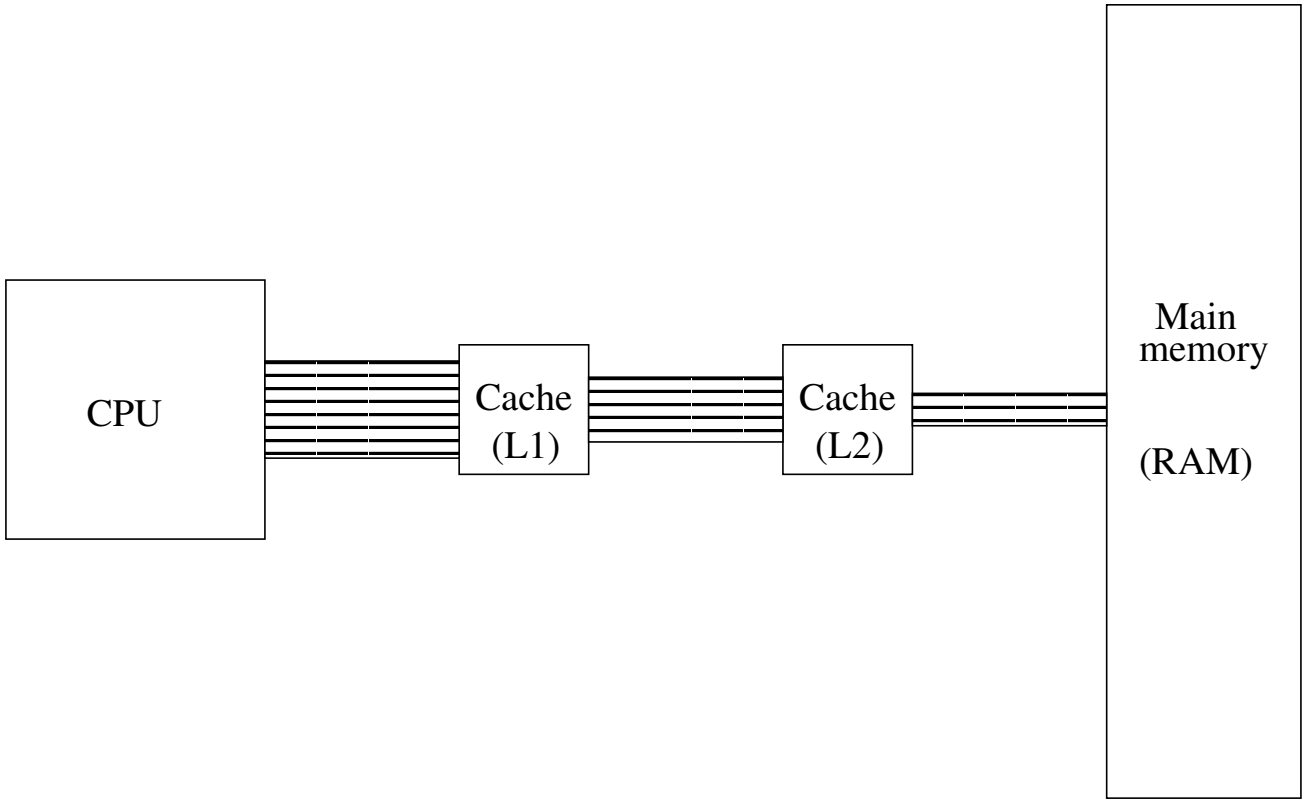
| | | Req. $x_1, \ldots, x_4$ | | Req. $x_2$ | | Req. $x_5$ |
|---|---|---|---|---|---|---|
| | | $x_4$ | | $x_2$ | | $x_5$ |
| | $\Rightarrow$ | $x_3$ | $\Rightarrow$ | $x_4$ | $\Rightarrow$ | $x_2$ |
| | | $x_2$ | | $x_3$ | | $x_4$ |
| | | $x_1$ | | $x_1$ | | $x_3$ |

# Realistic cache

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



# Multilevel caches




| Intel Core2 | | | AMD K8 | | |
|---|---|---|---|---|---|
| L1: | $S = 32$kB | $k = 8$ | L1: | $S = 16$kB | $k = 2$ |
| L2: | $S = 4$MB | $k = 16$ | L2: | $S = 1$MB | $k = 16$ |

# The dense case

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 $k = L = 2$:

$x_0$

———

$\Longrightarrow$

# The dense case

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 $k = L = 2$:

$$\frac{x_0}{\phantom{x}} \Longrightarrow \frac{a_{00}}{x_0} \Longrightarrow$$

# The dense case

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 $k = L = 2$:

$$\begin{array}{c} x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ \hline x_0 \end{array} \Longrightarrow$$

# The dense case

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 $k = L = 2$:

$$\begin{array}{c} x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ \hline x_0 \end{array} \Longrightarrow \begin{array}{c} x_1 \\ y_0 \\ \hline a_{00} \\ x_0 \end{array} \Longrightarrow$$

# The dense case

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 $k = L = 2$:

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

# The dense case

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 $k = L = 2$:

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

When $k, L$ are a bit larger, we can predict the following:

- the lower elements from the vector $x$ (that is, $x_0, x_1, \ldots, x_i$ for some $i < n$) are evicted while processing the entire first row. This causes $\mathcal{O}(n)$ cache misses on the remaining $m - 1$ rows.

Fix:

- stop processing a row before an element from $x$ would be evicted and first continue row-wise: i.e., process $Ax$ by doing MVs on $m \times q$ submatrices: $y = a_0 x + a_1 x + \ldots$

Unwanted side effect:

- now lower elements from the vector $y$ can be prematurely evicted...

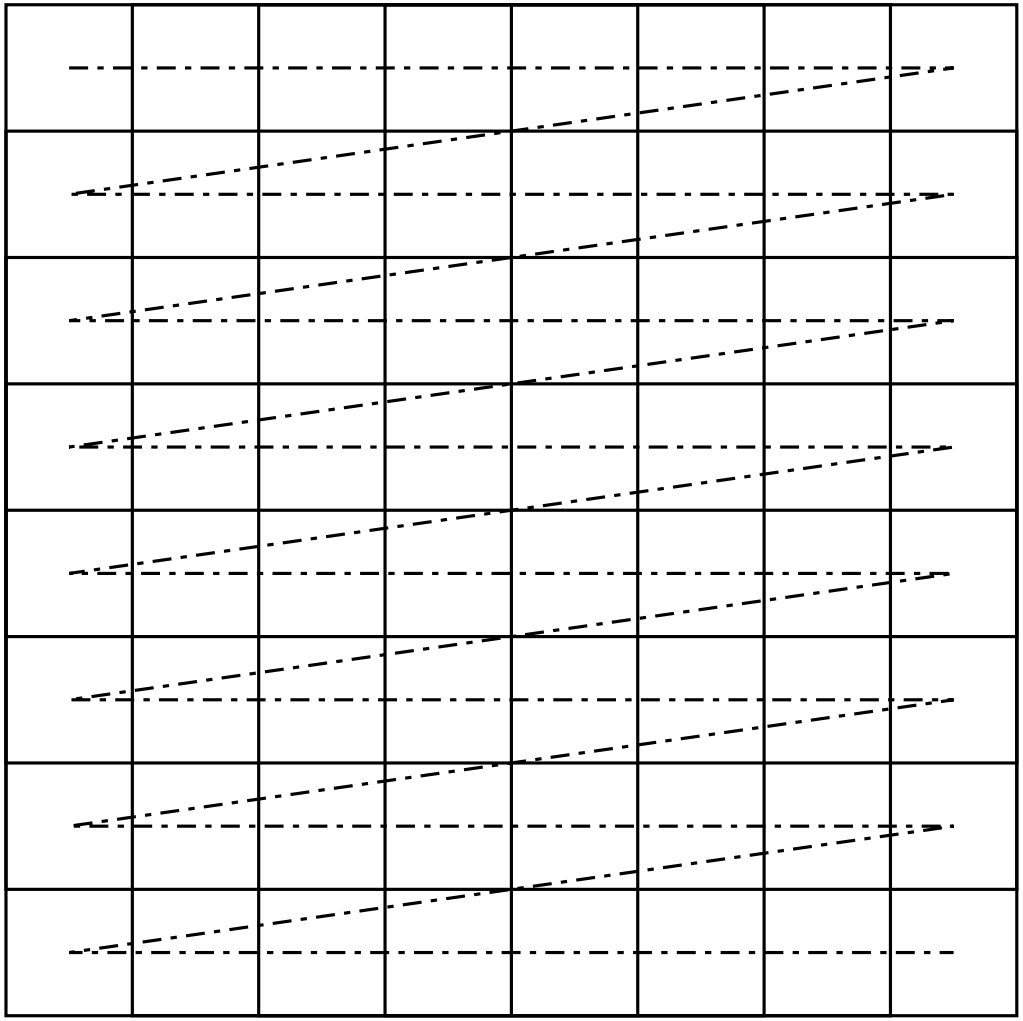
Fix:

- stop processing a submatrix before an element from $y$ would be evicted; the MV routine now is applied on $p \times q$ submatrices.

This approach is *cache-aware*; implemented in, e.g., GotoBLAS.

# The sparse case

Standard datastructure: Compressed Row Storage (CRS)



# The sparse case

Standard datastructure: Compressed Row Storage (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] |
| row: | [0 3 5 7 10] |

# The sparse case

Sparse matrix–vector multiplication (SpMV)

$x$?

$\Longrightarrow$

# The sparse case

Sparse matrix–vector multiplication (SpMV)

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

$\quad\quad\quad\quad$ $x_?$

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

# The sparse case

Sparse matrix–vector multiplication (SpMV)

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

# The sparse case

Sparse matrix–vector multiplication (SpMV)

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

# The sparse case

Sparse matrix–vector multiplication (SpMV)

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

# The sparse case

Sparse matrix–vector multiplication (SpMV)

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

We cannot predict memory accesses in the sparse case!

# Cache-friendly data structures



# Band datastructures

Instead of storing matrices row- or column-wise, store matrix diagonals

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

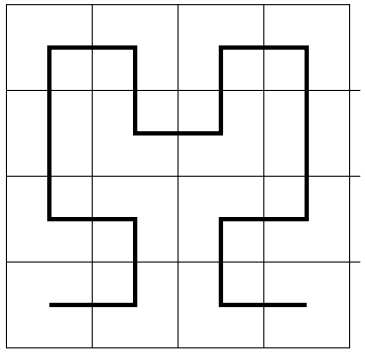
Stored as:

(2) [4 2 0]
(3) [1 3 2]

Reference: Sivan Toledo, *Improving the memory-system performance of sparse-matrix vector multiplication*, 1997

# Fractal datastructures

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



Stored as:

| | |
|---|---|
| nzs: | [7 1 4 1 2 2 3 2 1] |
| i : | [3 2 0 0 1 0 1 2 3] |
| 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

# 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: $4, 1, 3$ - $2, 3$ - $1$ - $2$ - $7$ - $1, 1$

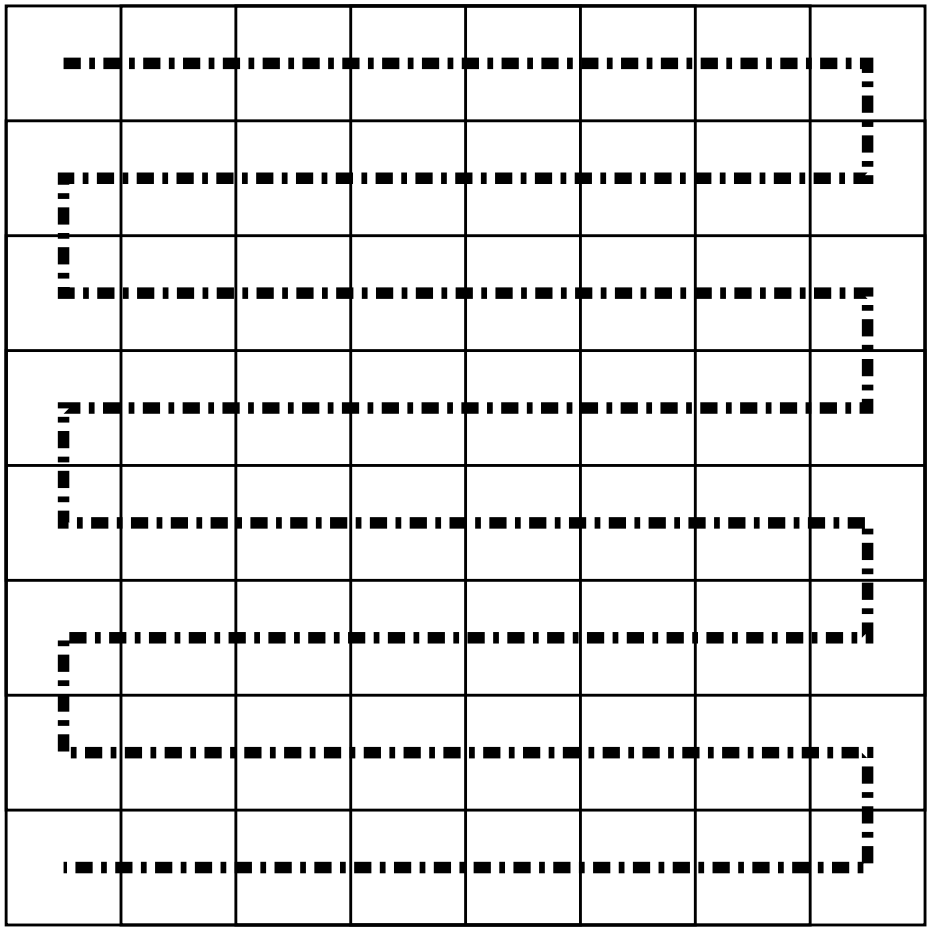
Stored as:

| | |
|---|---|
| nzs: | [4 1 3 2 3 1 2 7 1 1] |
| col: | [0 2 0 3 0 2] |
| row: | [0 1 2 4 6] |
| blk: | [0 3 5 6 7 8] |

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

# Zig-zag CRS

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] |
| row: | [0 3 5 7 10] |

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

# Cache-oblivious sparse matrix structure



# Why not change the structure of the input matrix?

- Assuming zig-zag CRS ordering

# Why not change the structure of the input matrix?

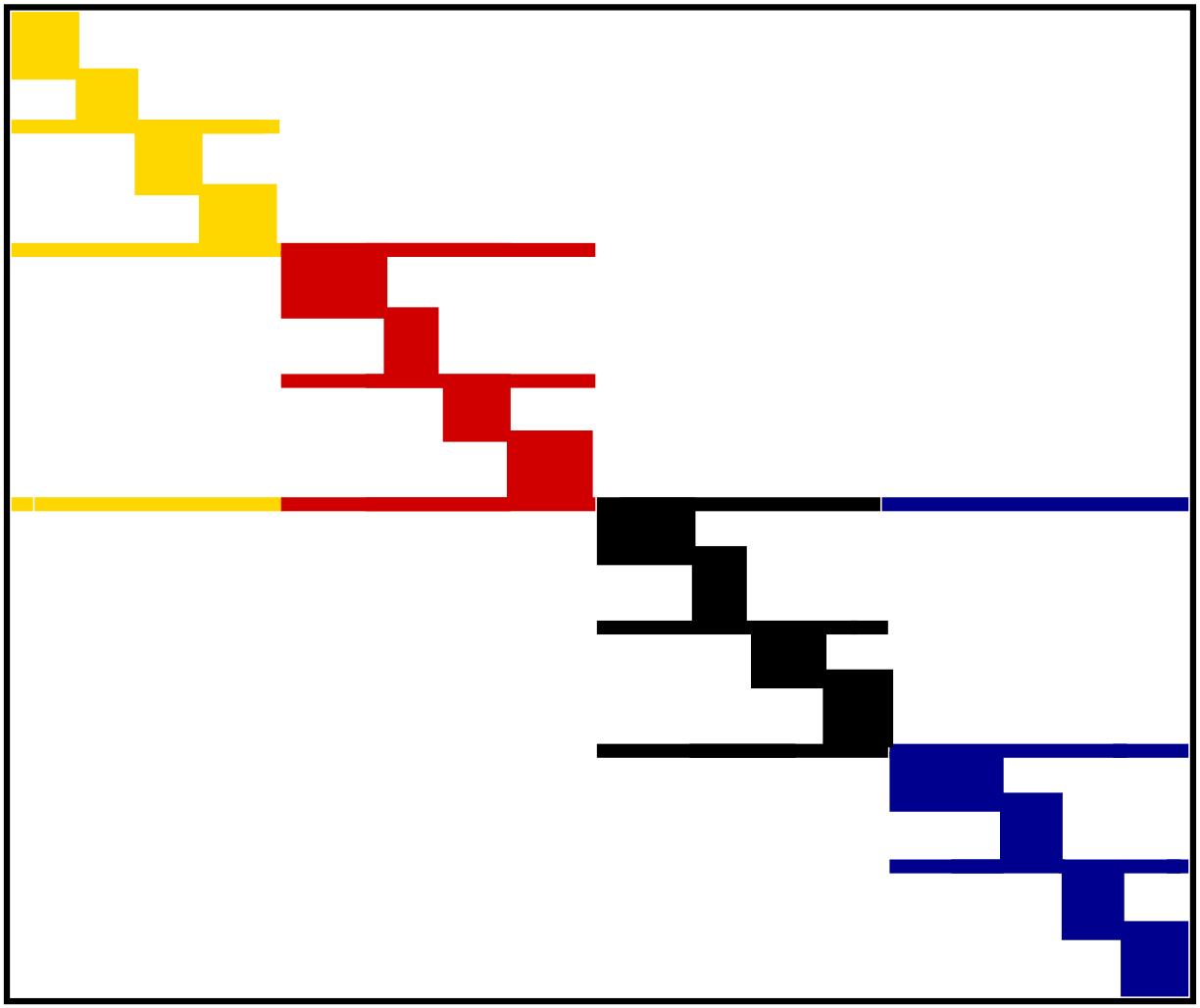
- Assuming zig-zag CRS ordering
- Allowing only row and column permutations

# Why not change the structure of the input matrix?

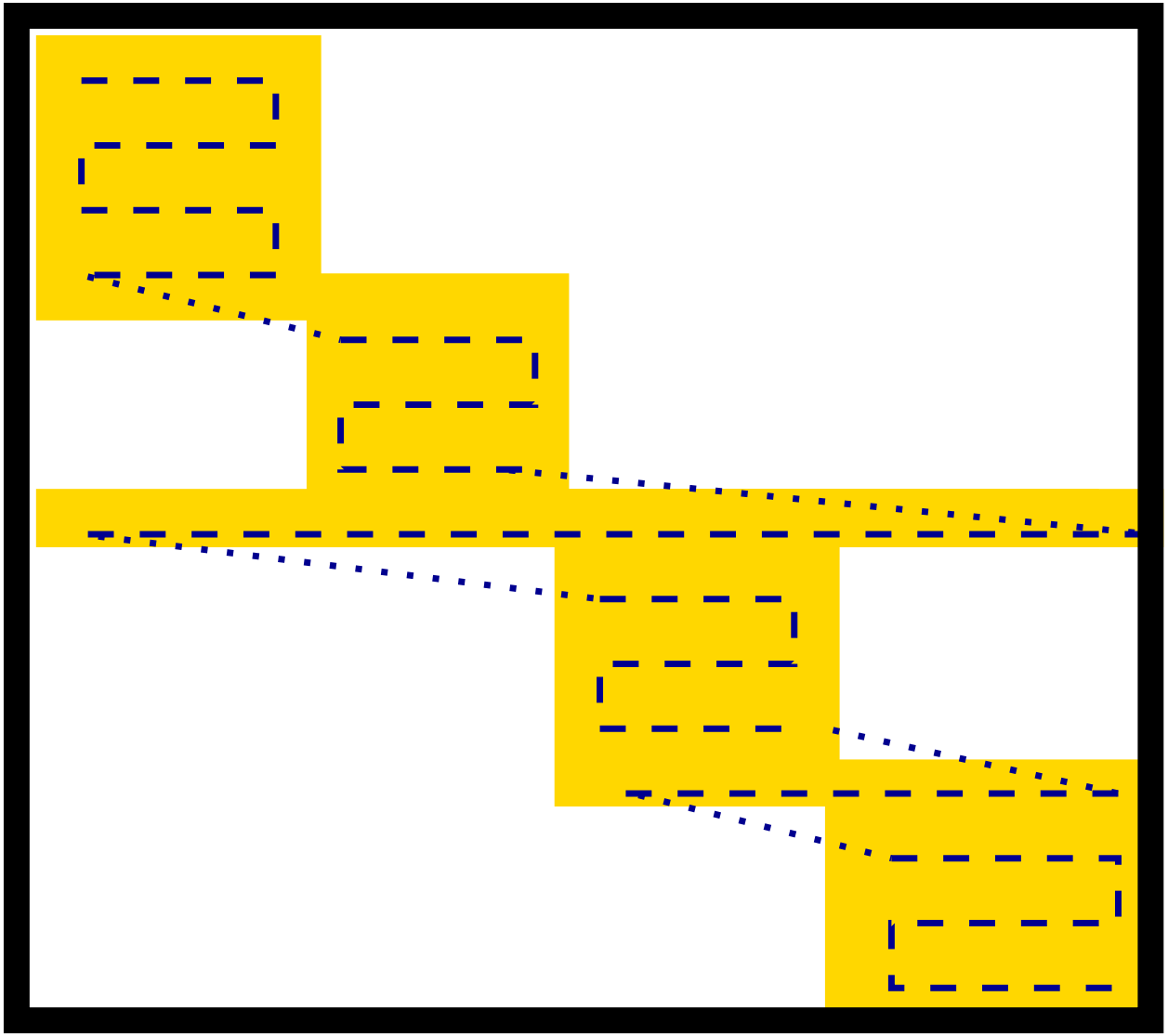
- Assuming zig-zag CRS ordering
- Allowing only row and column permutations

This is, on a certain level, also *matrix-oblivious*. A further advantage is that by knowing that the reordered matrix $A'$ can be written as $PAQ$, we can still apply some strategies from the previous sections (e.g., Blocked CRS), or libraries relying on such strategies (e.g., OSKI).

# Seperated Block Diagonal form



# Seperated Block Diagonal form



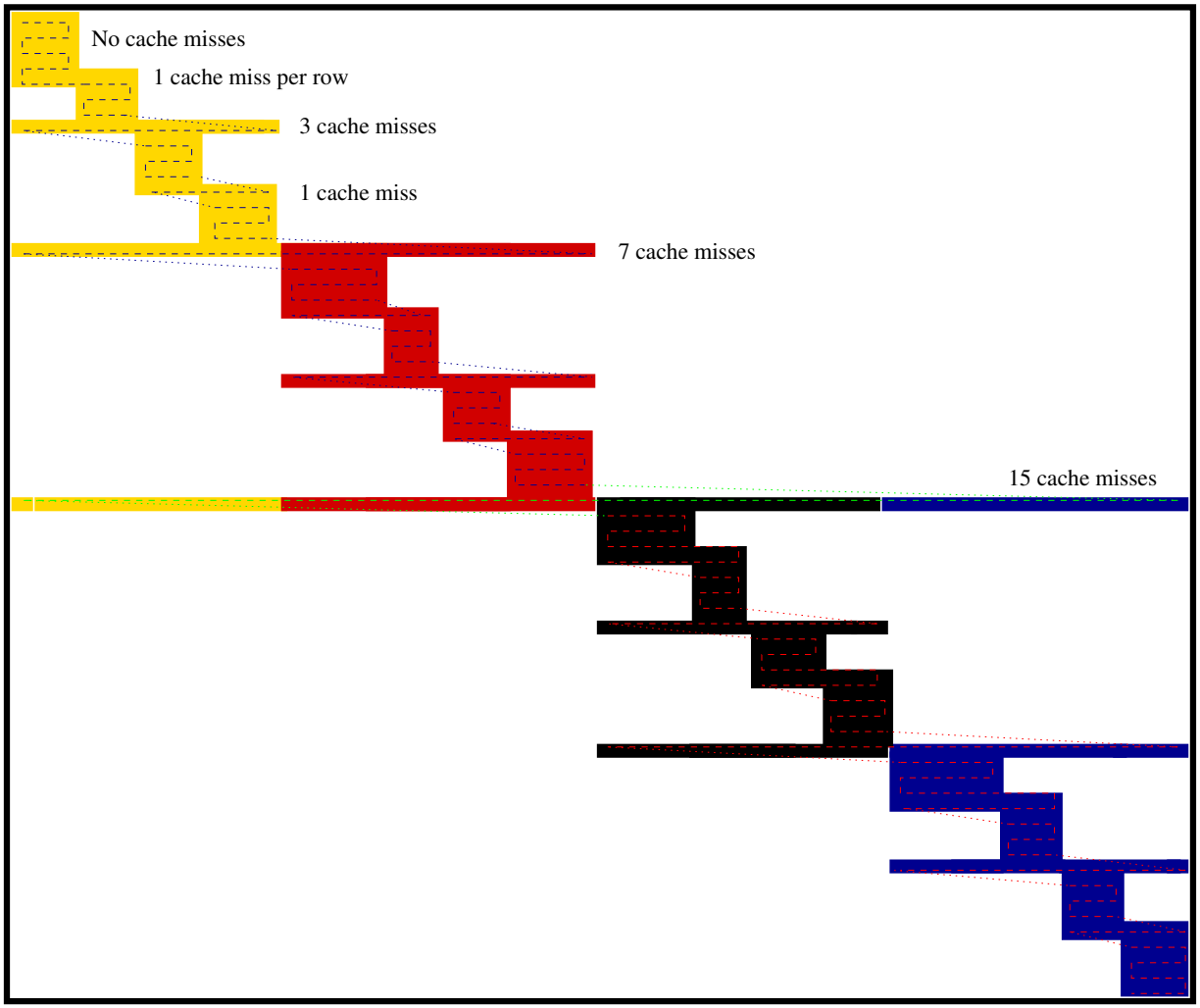
# Seperated Block Diagonal form

Given a specific cache, regarding the number of blocks $p$ to be taken, there is a 'sweet spot' at

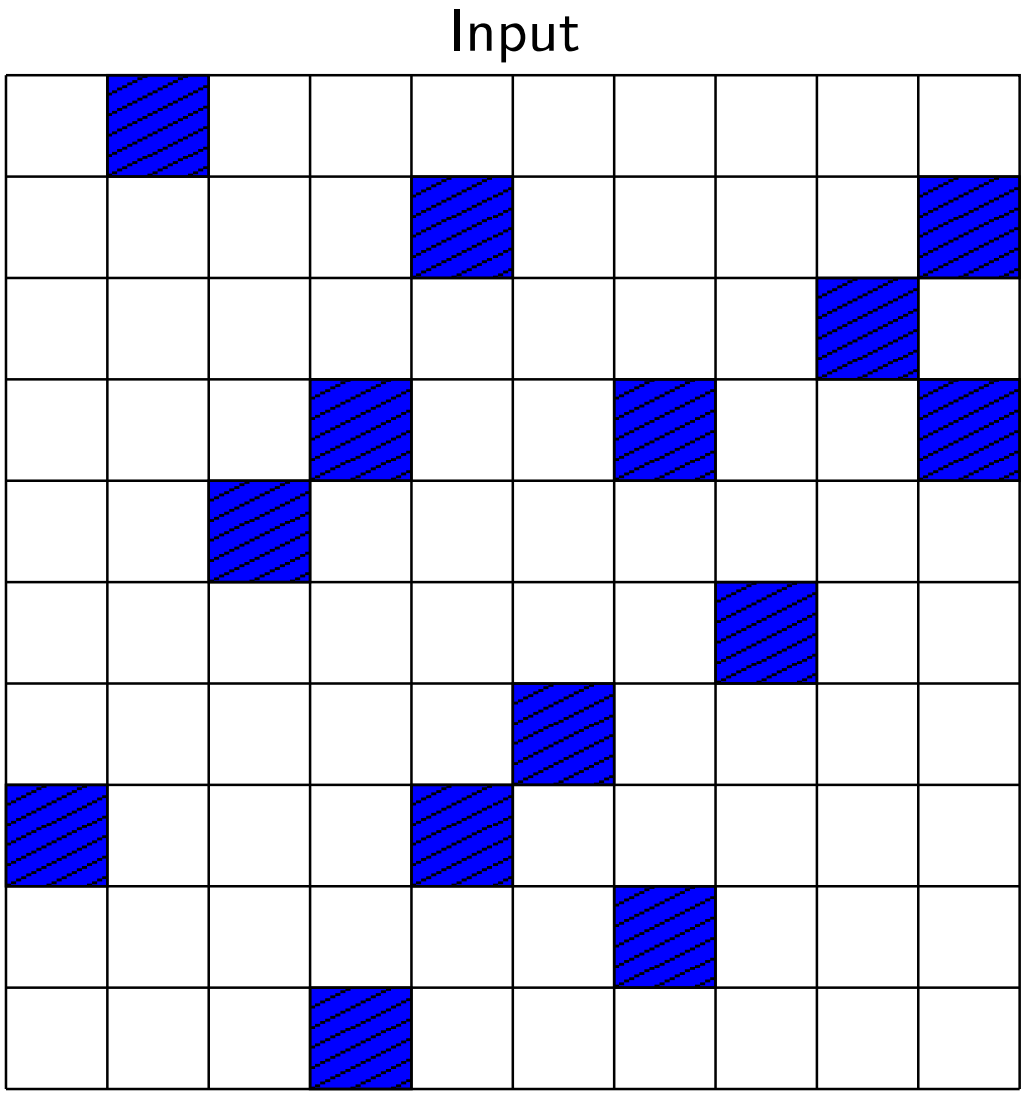
$$p = \frac{n}{wL}.$$

where $w$ is the number of data elements that fit into a single cache line.

# Seperated Block Diagonal form

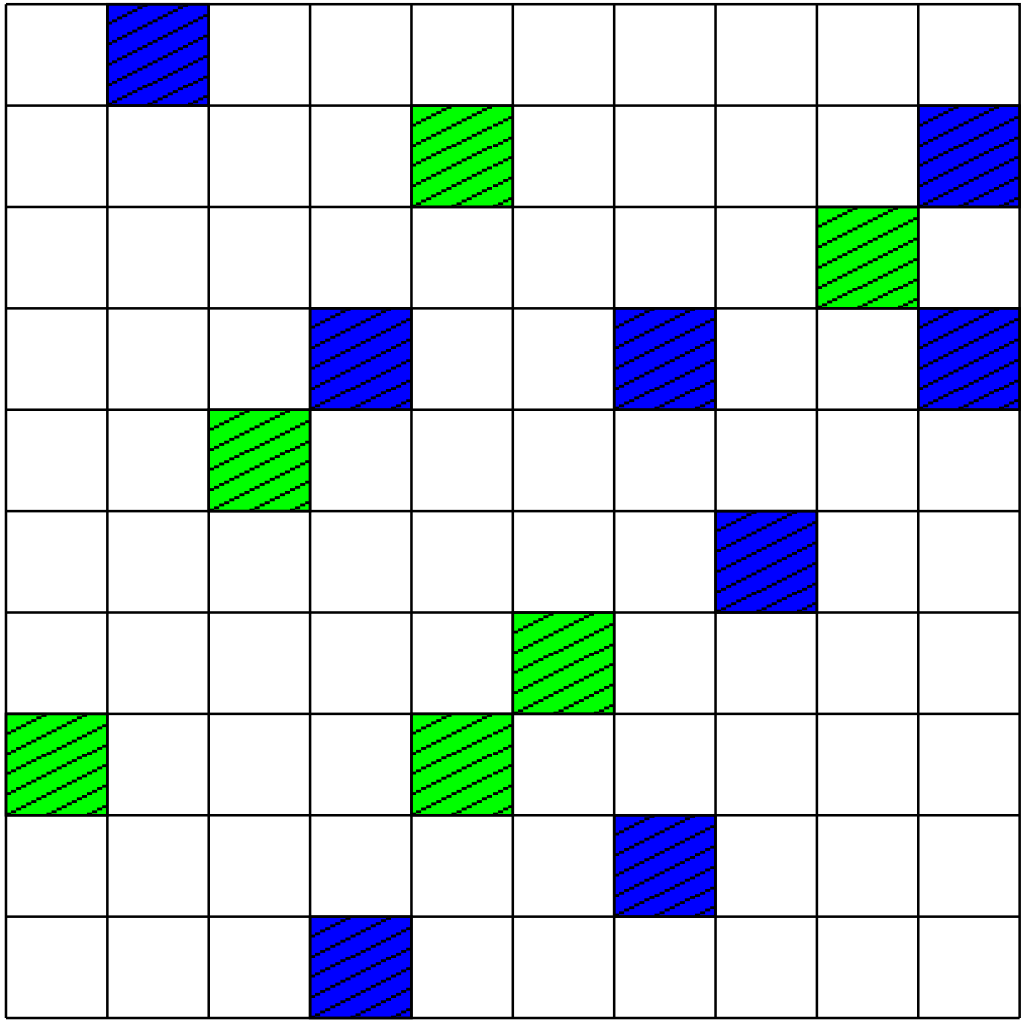


# Permuting to SBD form



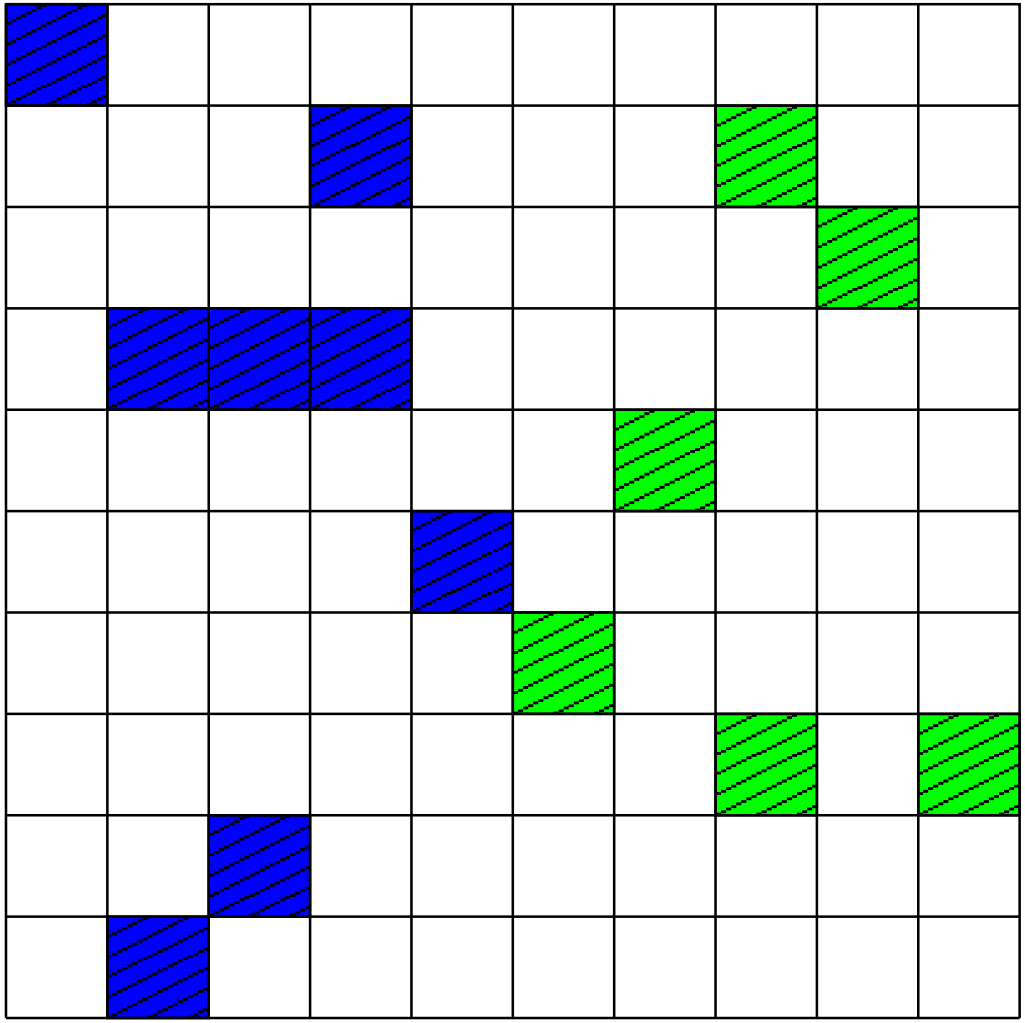
# Permuting to SBD form

Column partitioning

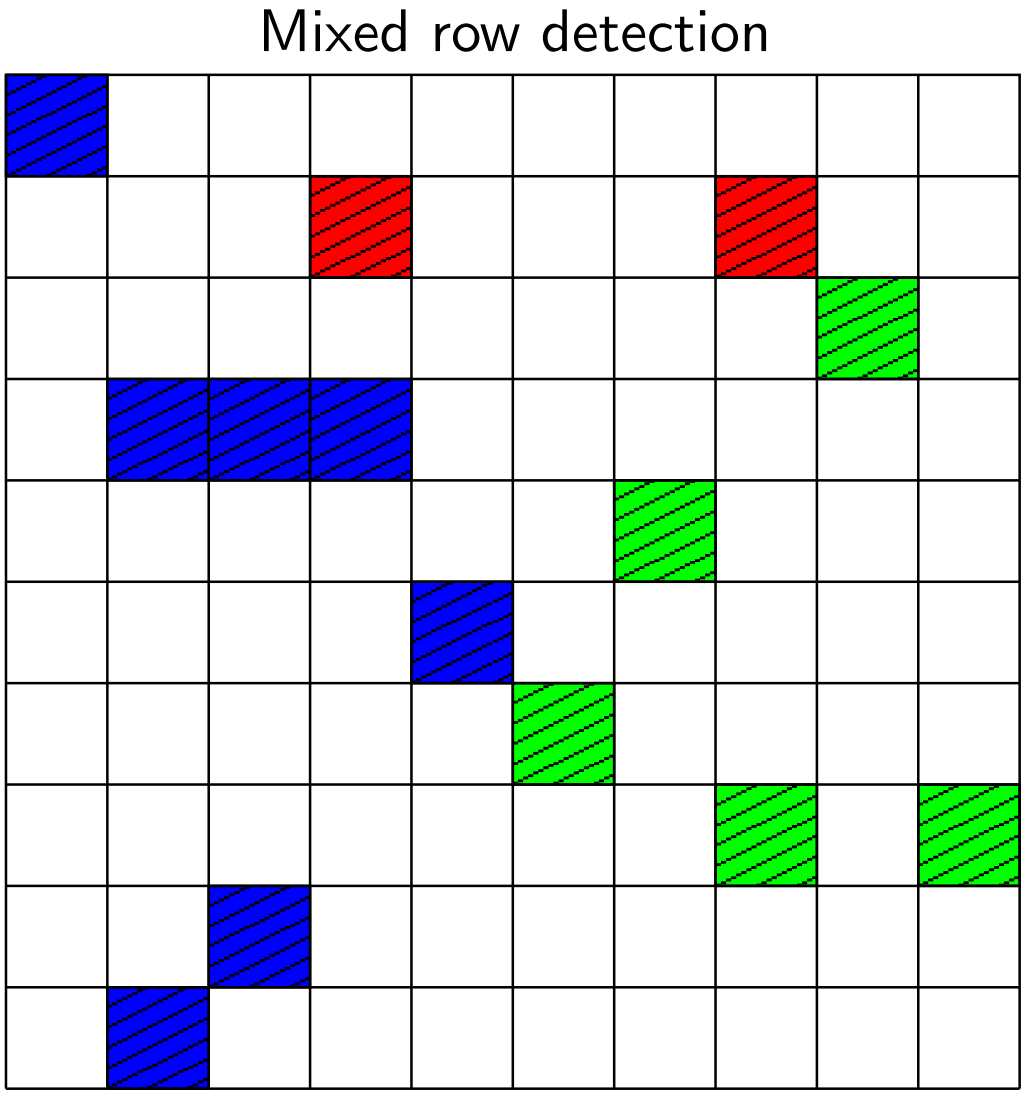


# Permuting to SBD form

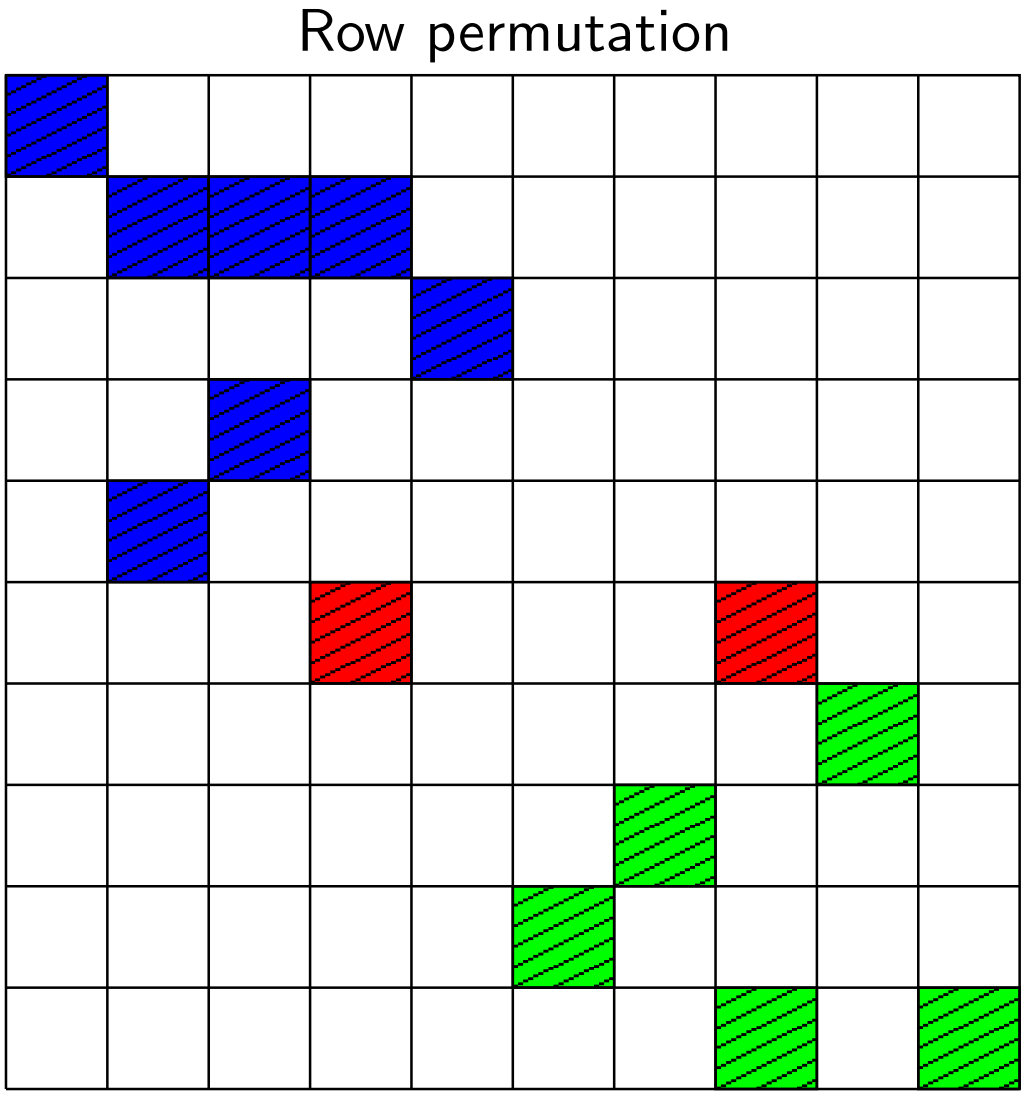
Column permutation



# Permuting to SBD form

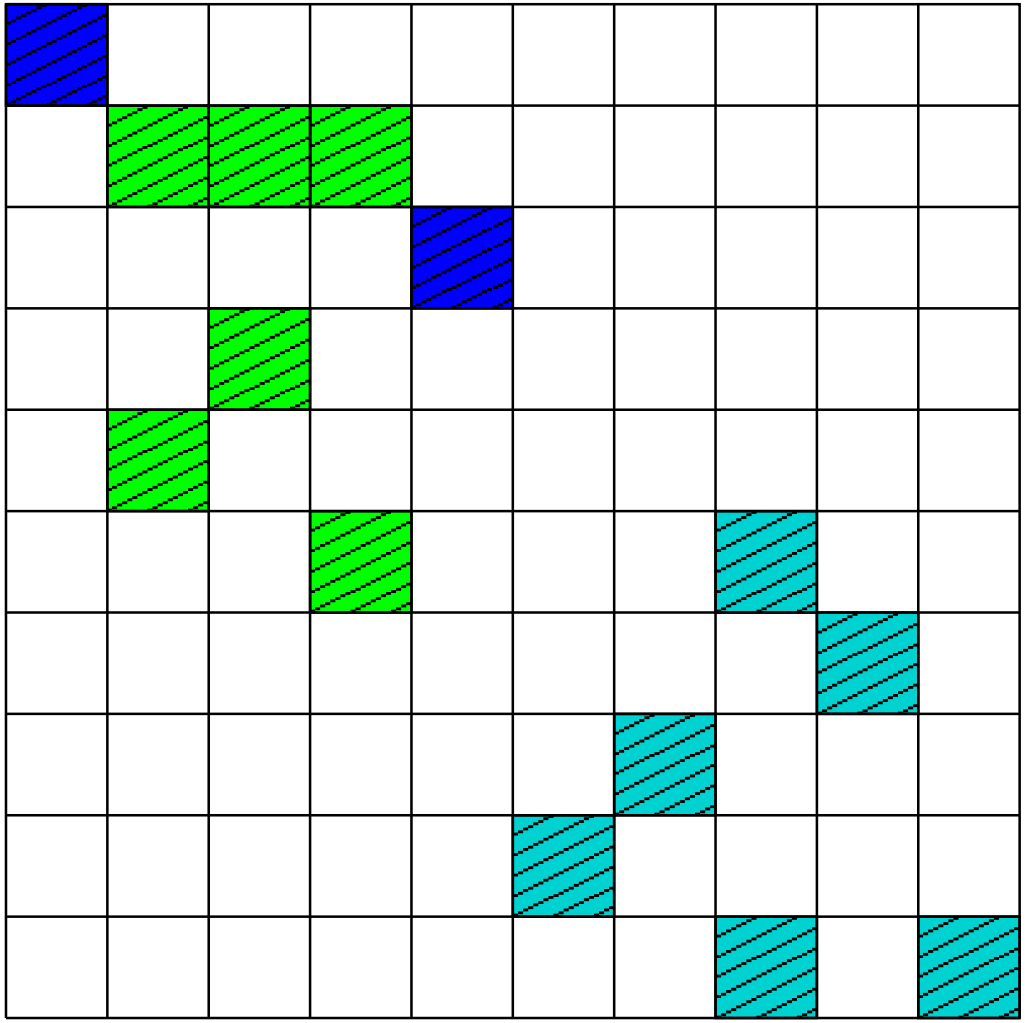


# Permuting to SBD form

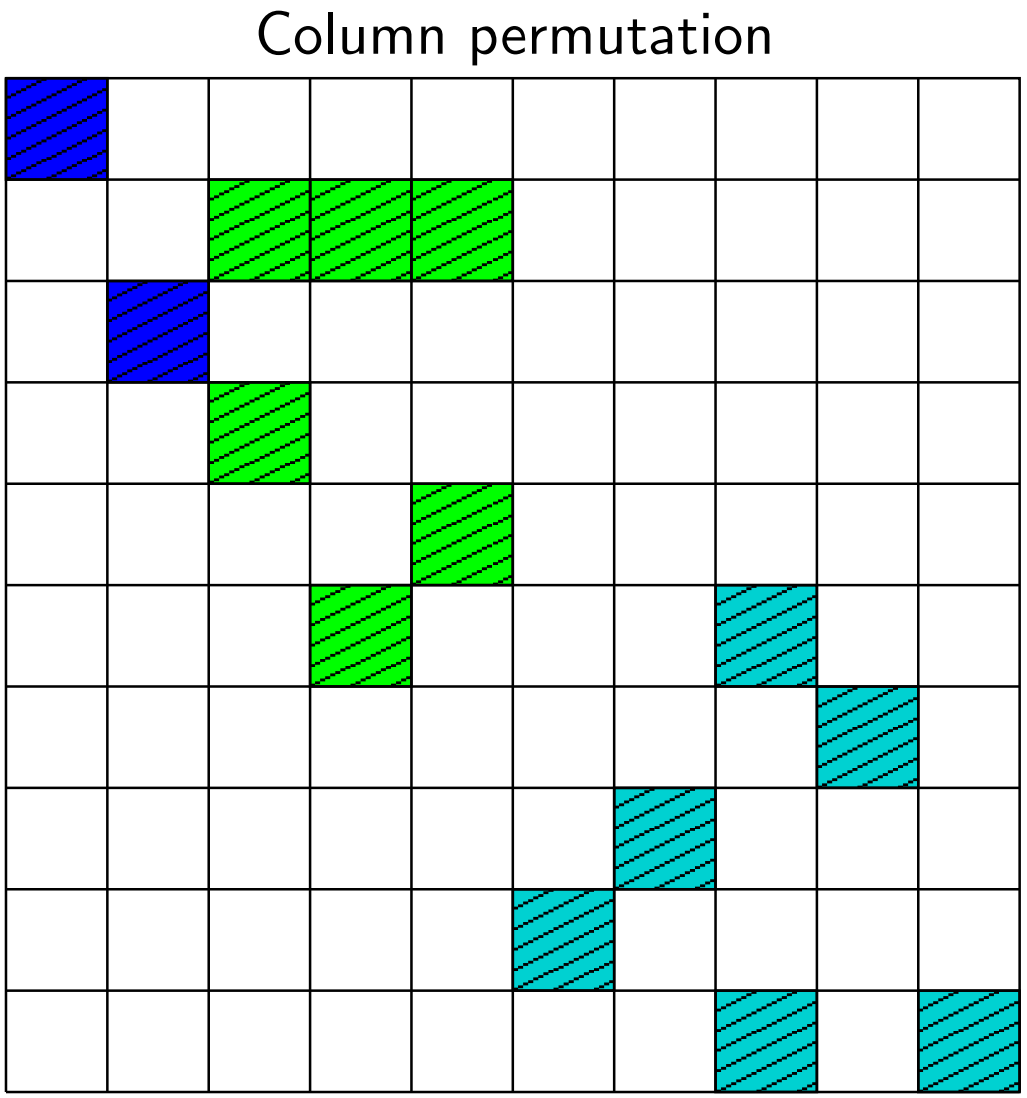


# Permuting to SBD form

Column subpartitioning

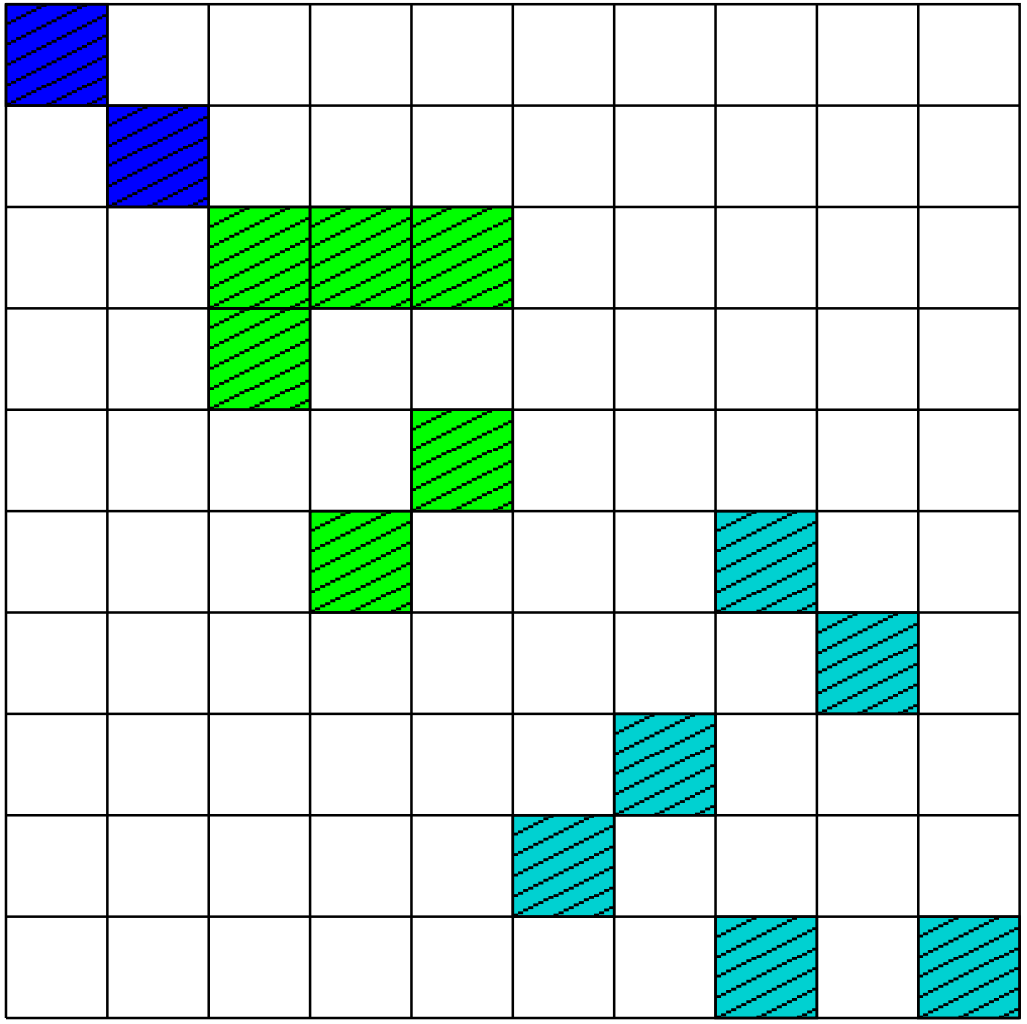


# Permuting to SBD form



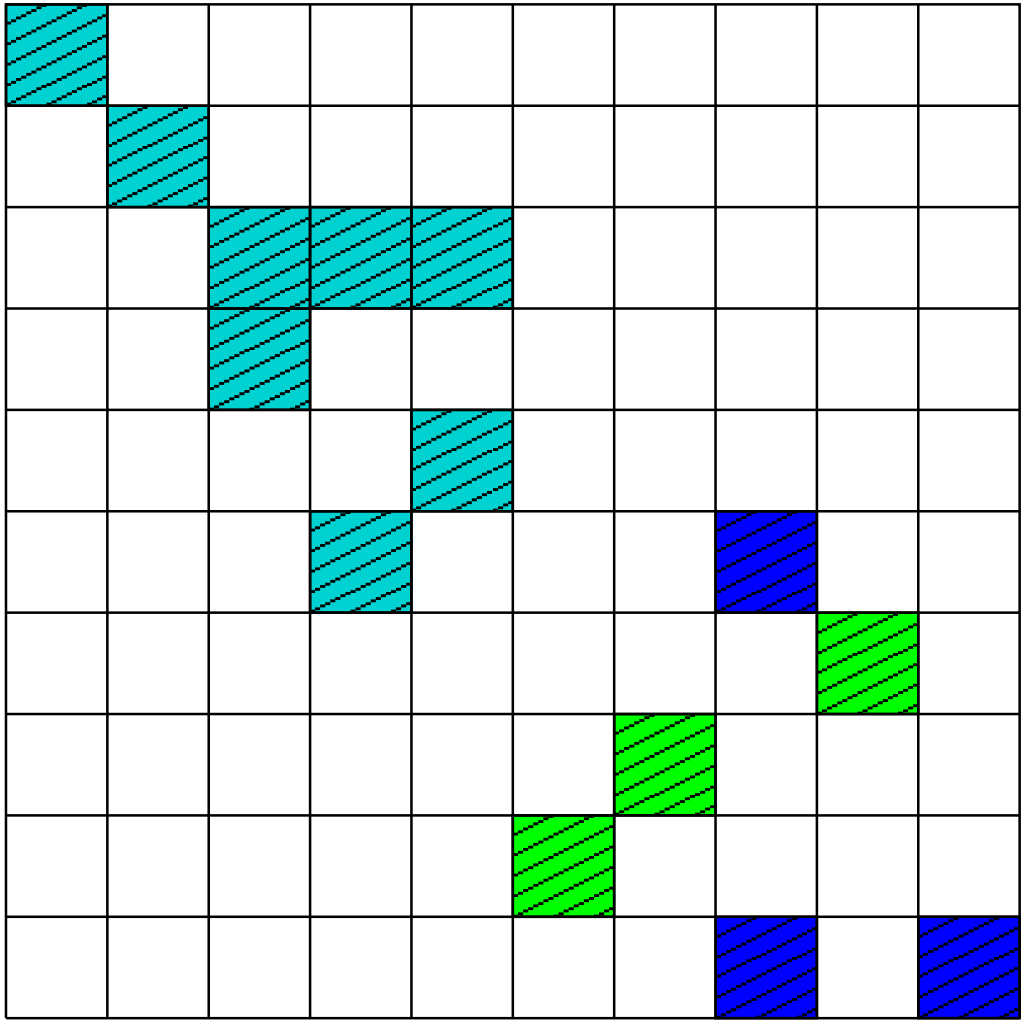
# Permuting to SBD form

No mixed rows - row permutation



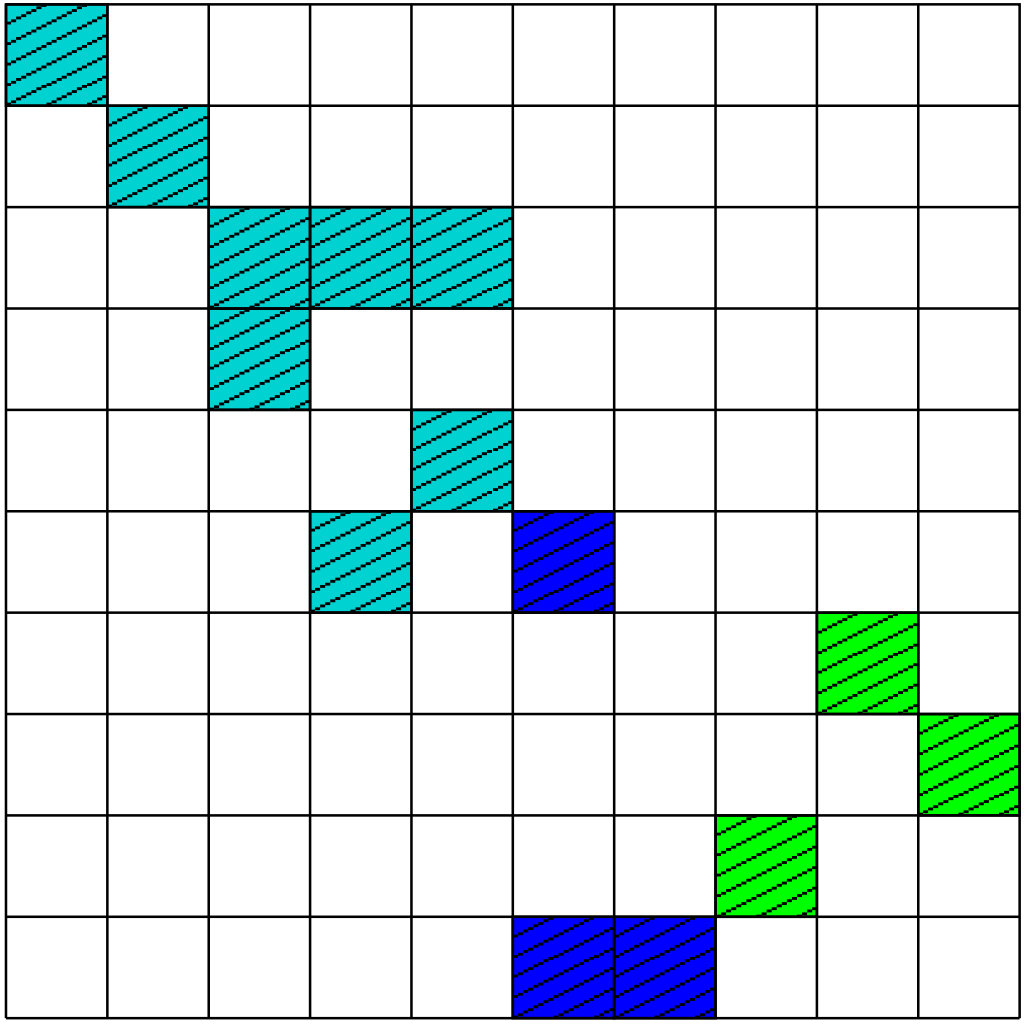
# Permuting to SBD form

Continued



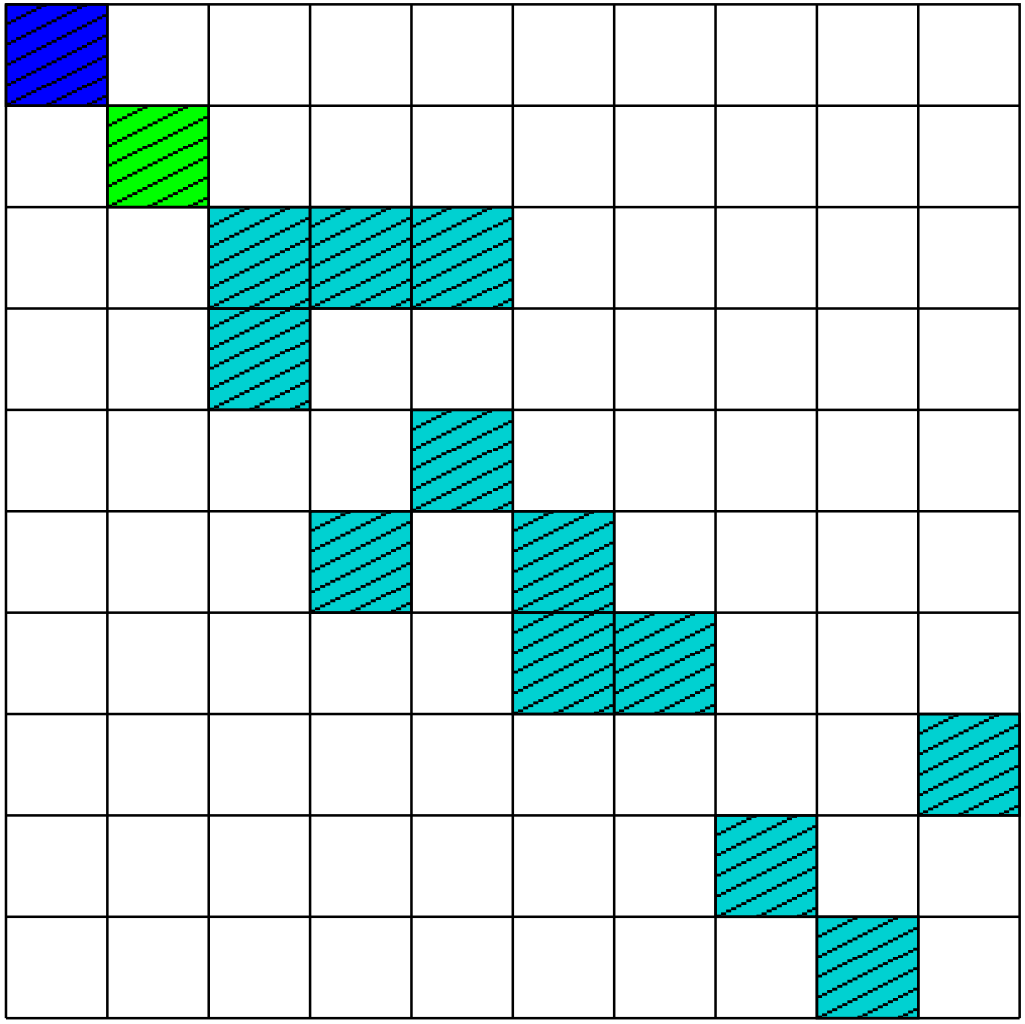
# Permuting to SBD form

Continued



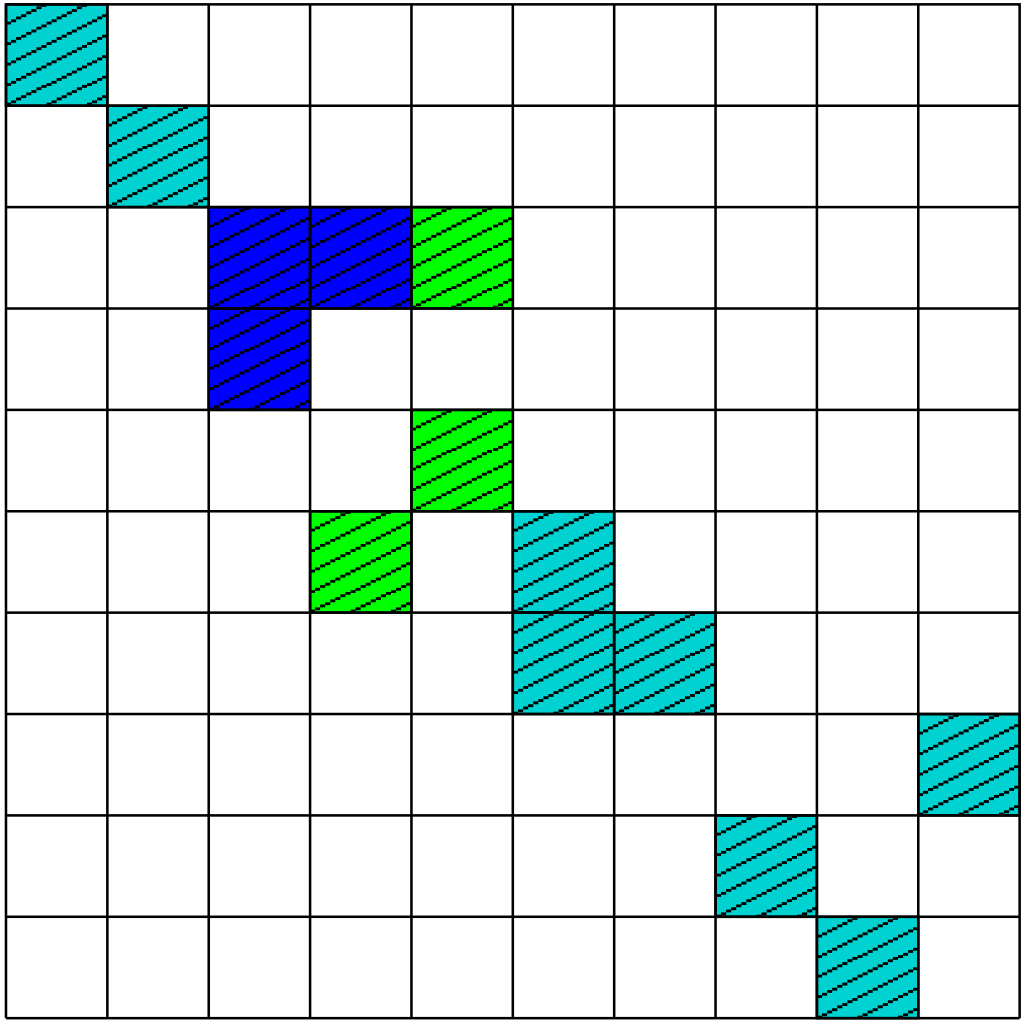
# Permuting to SBD form

Continued



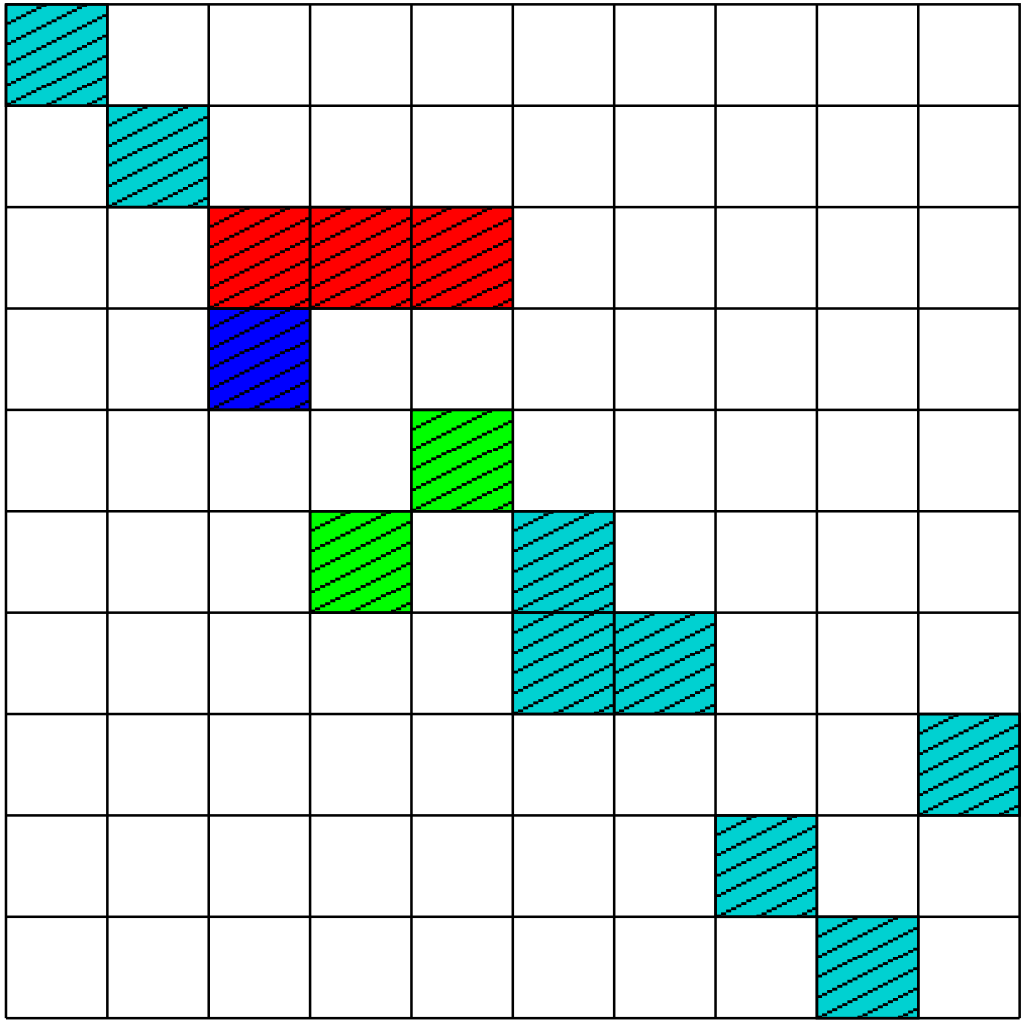
# Permuting to SBD form

Continued



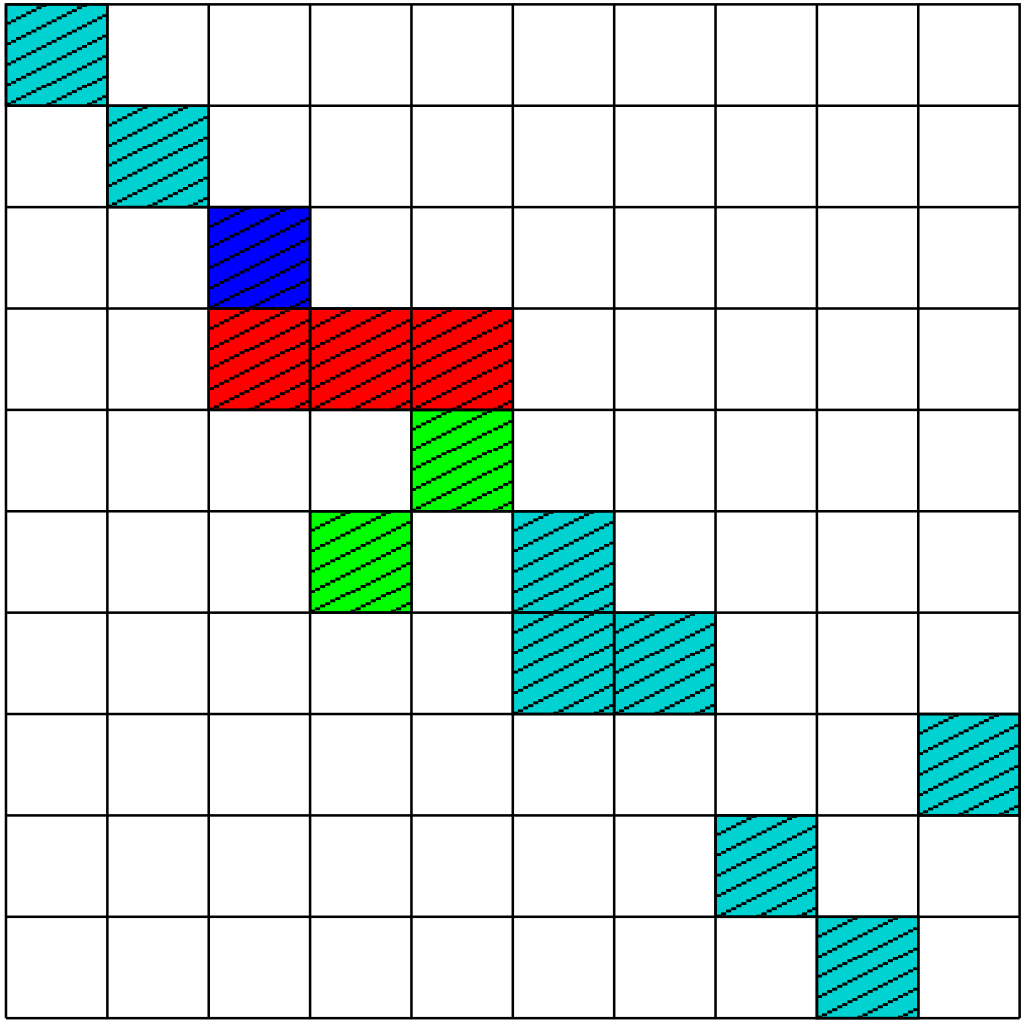
# Permuting to SBD form

Continued



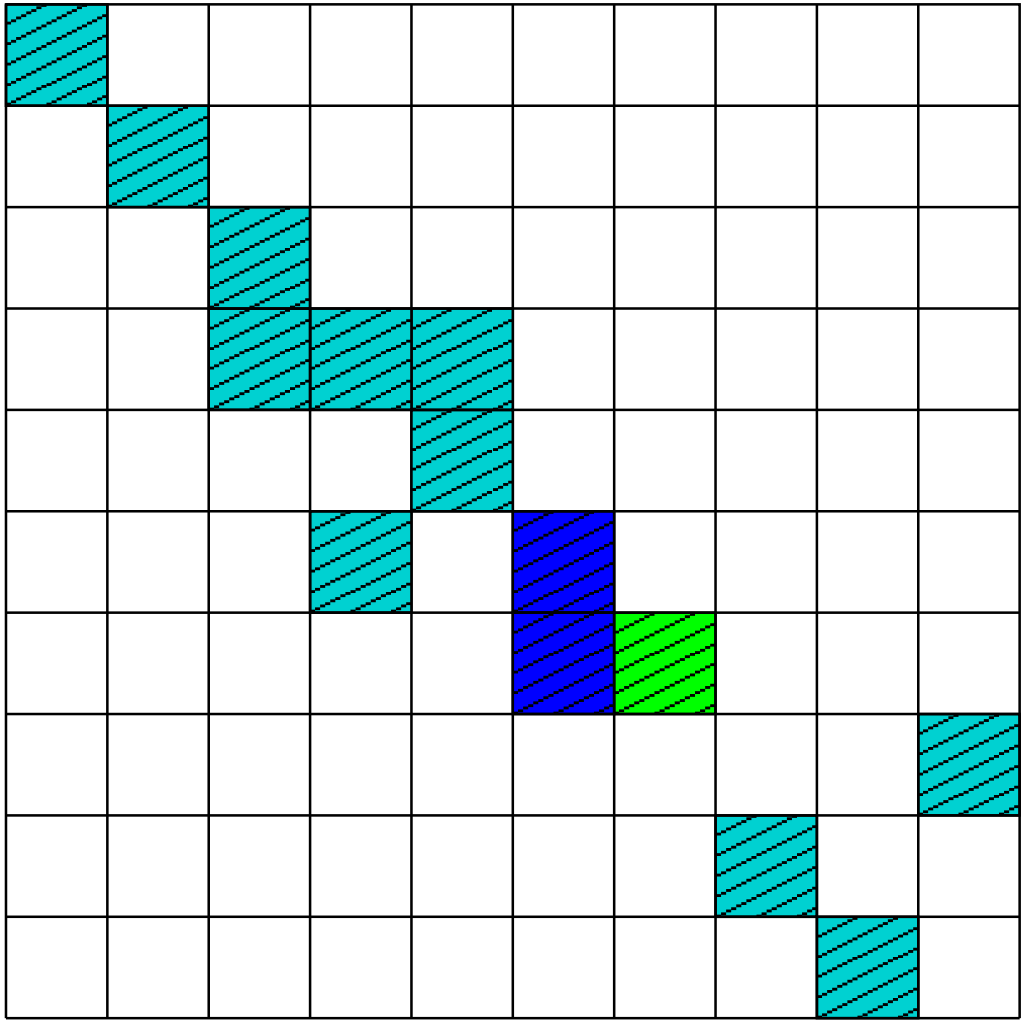
# Permuting to SBD form

Continued



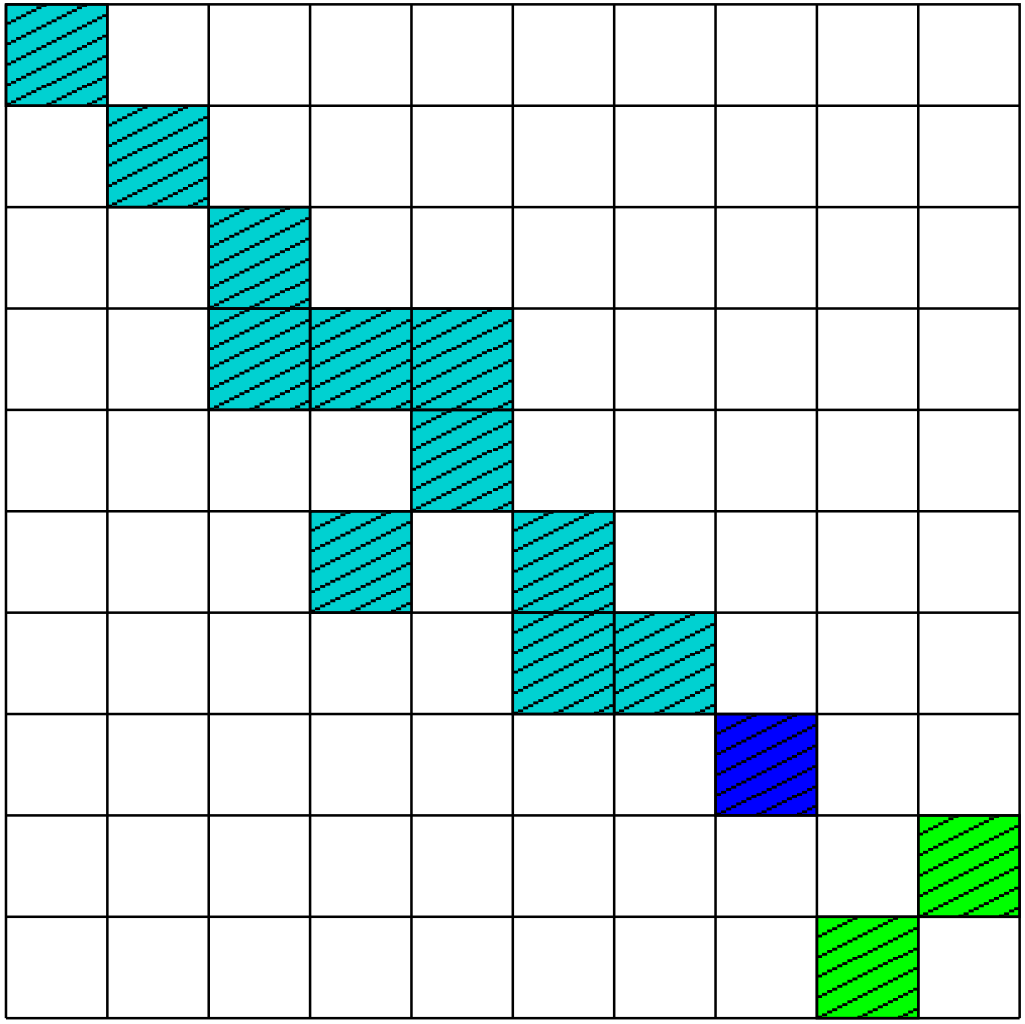
# Permuting to SBD form

Continued



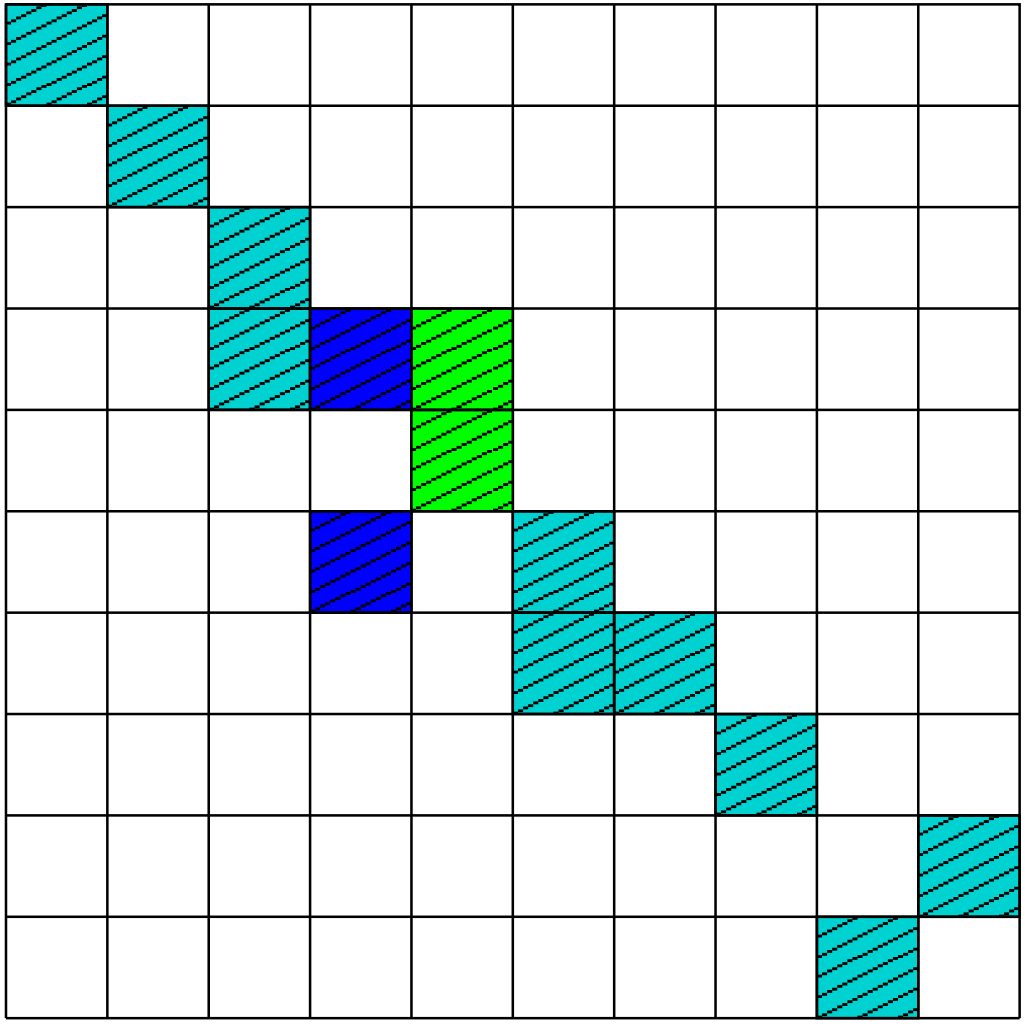
# Permuting to SBD form

Continued



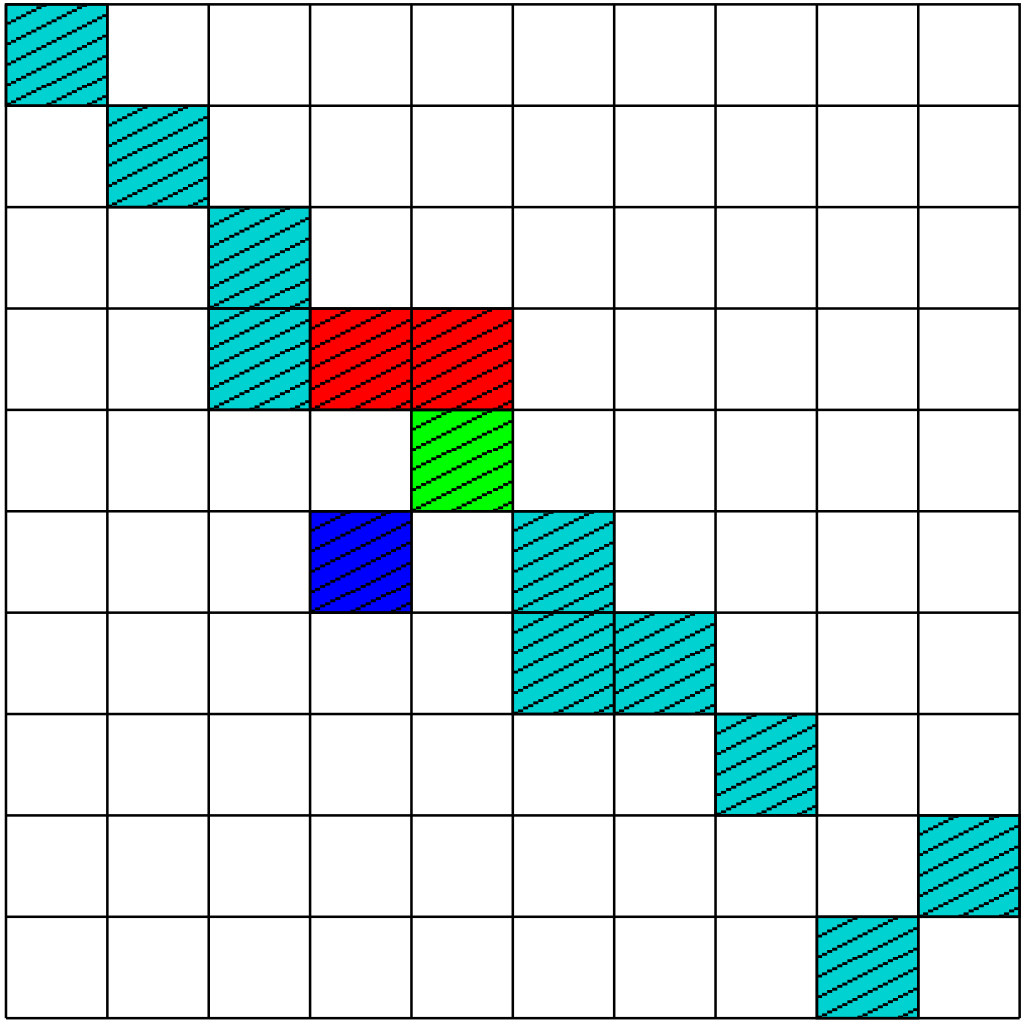
# Permuting to SBD form

Continued



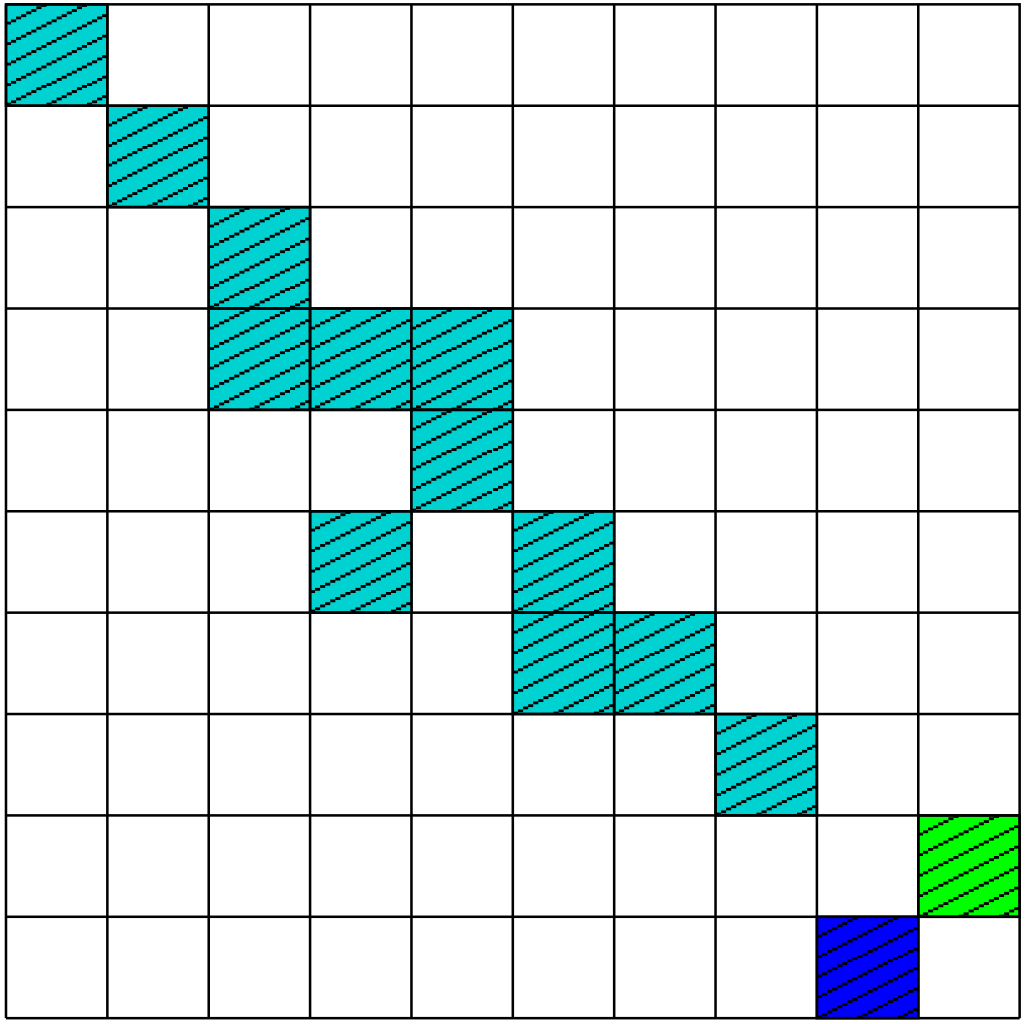
# Permuting to SBD form

Continued



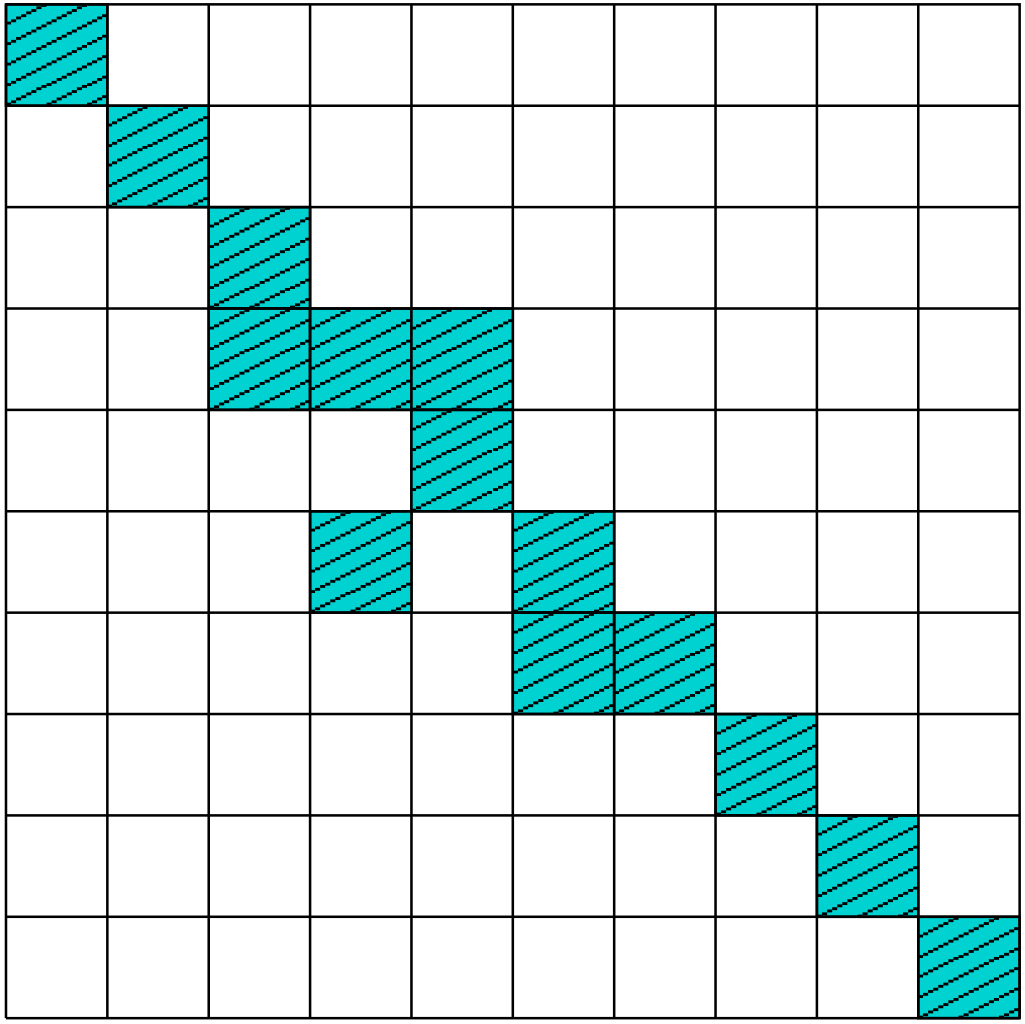
# Permuting to SBD form

Continued



# Permuting to SBD form

Continued

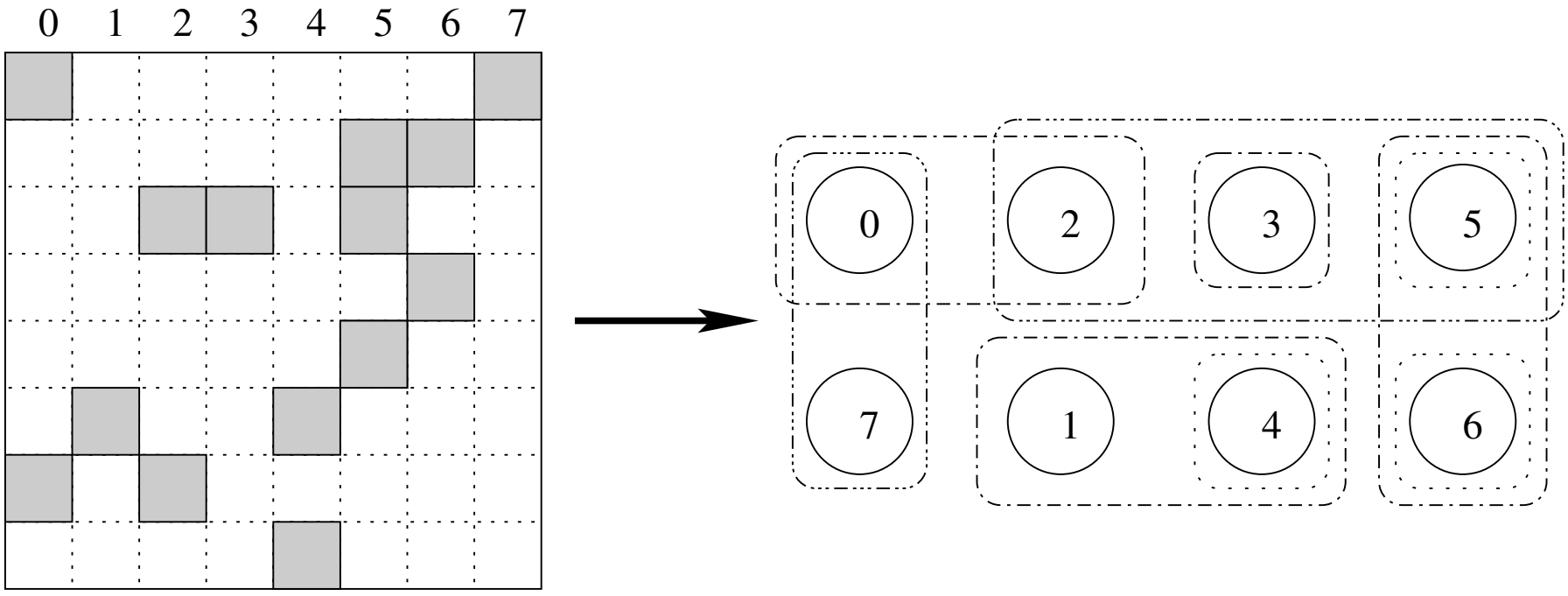


# Obtaining SBD form using partioners



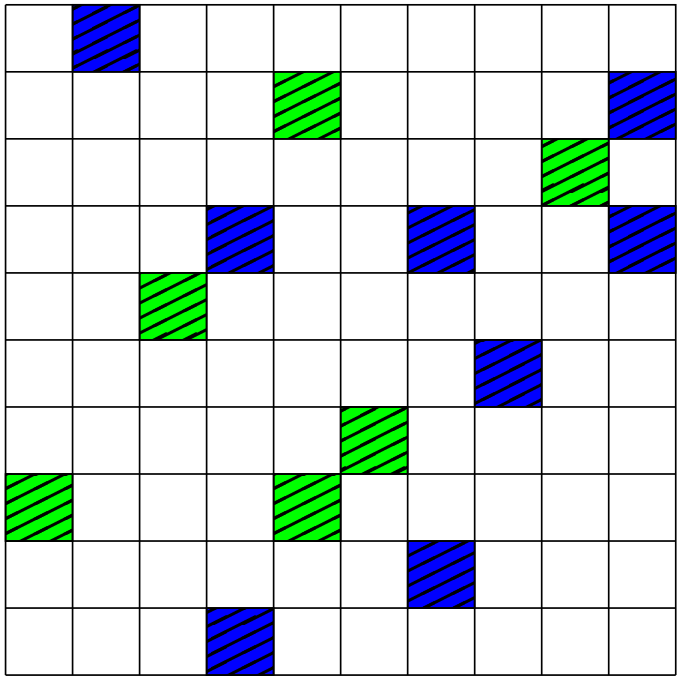
# Sparse matrix to hypergraph conversion

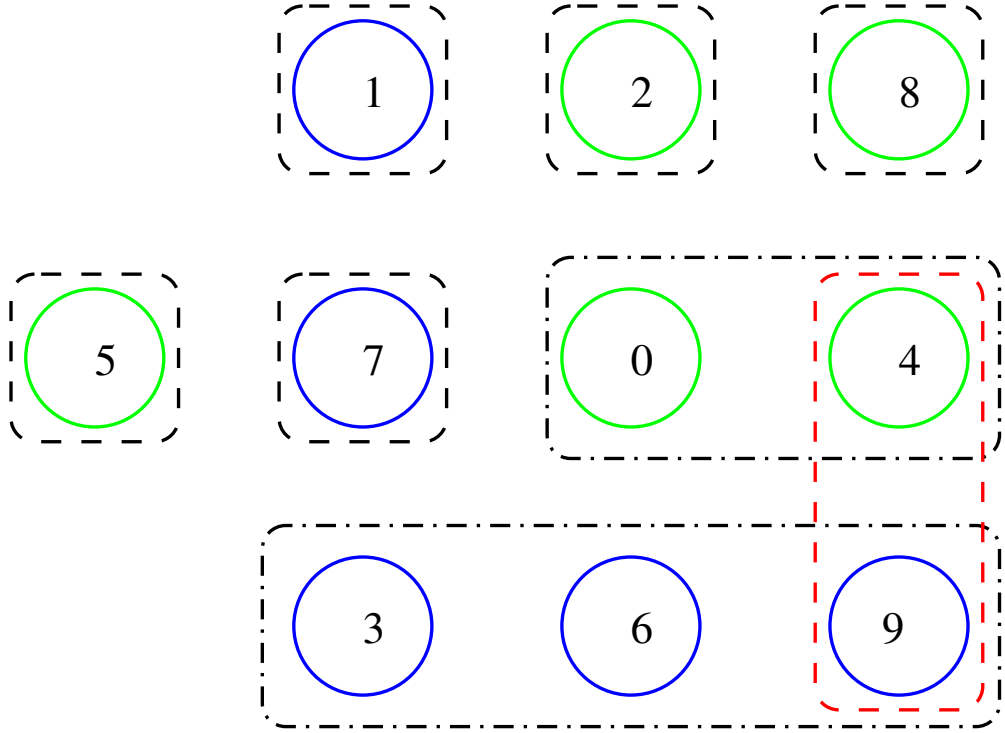
Transform a sparse matrix $A$ to a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ according to the *row-net model*:




Columns correspond to vertices and rows to hyperedges (nets).

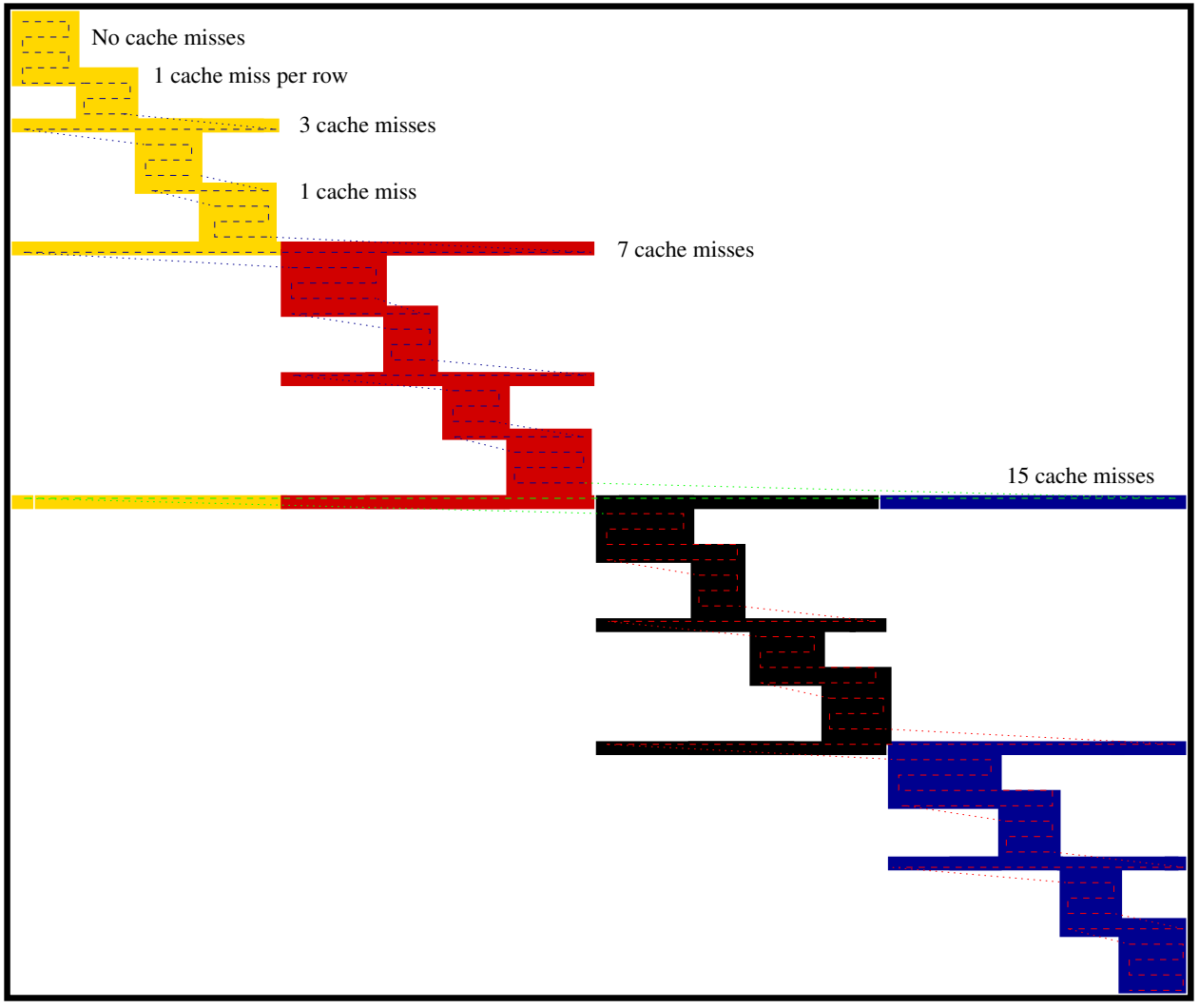
We partition the set of vertices $\mathcal{V}$ in $\mathcal{V}_0$ (blue) and $\mathcal{V}_1$ (green).






Mixed nets are easily detectable as *cut nets* in the hypergraph representation, marked in red.

Eventually we will have partitioned $\mathcal{V}$ in a lot of subsets $\mathcal{V}_0, \ldots, \mathcal{V}_{p-1}$. The nets spread over multiple partitions make up those mixed-row blocks incurring more than 1 cache miss per row.



The number of subsets a net is cast over after partitioning, is called the connectivity $\lambda_i$. For $p = \frac{n}{wL}$, the number of cache misses is given by

$$\sum_{i:\ n_i \in \mathcal{N}} \lambda_i - 1.$$

This is also called the $\lambda - 1$ *metric*, and is already used extensively in parallel computing. Partitioners should also take into account the *load-imbalance*, typically denoted by $\epsilon$.

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

Our method has been implemented by adapting the Mondriaan software.

For a truly cache-oblivious approach, the number of subsets $p$ we partition $\mathcal{V}$ into, must be as large as possible:

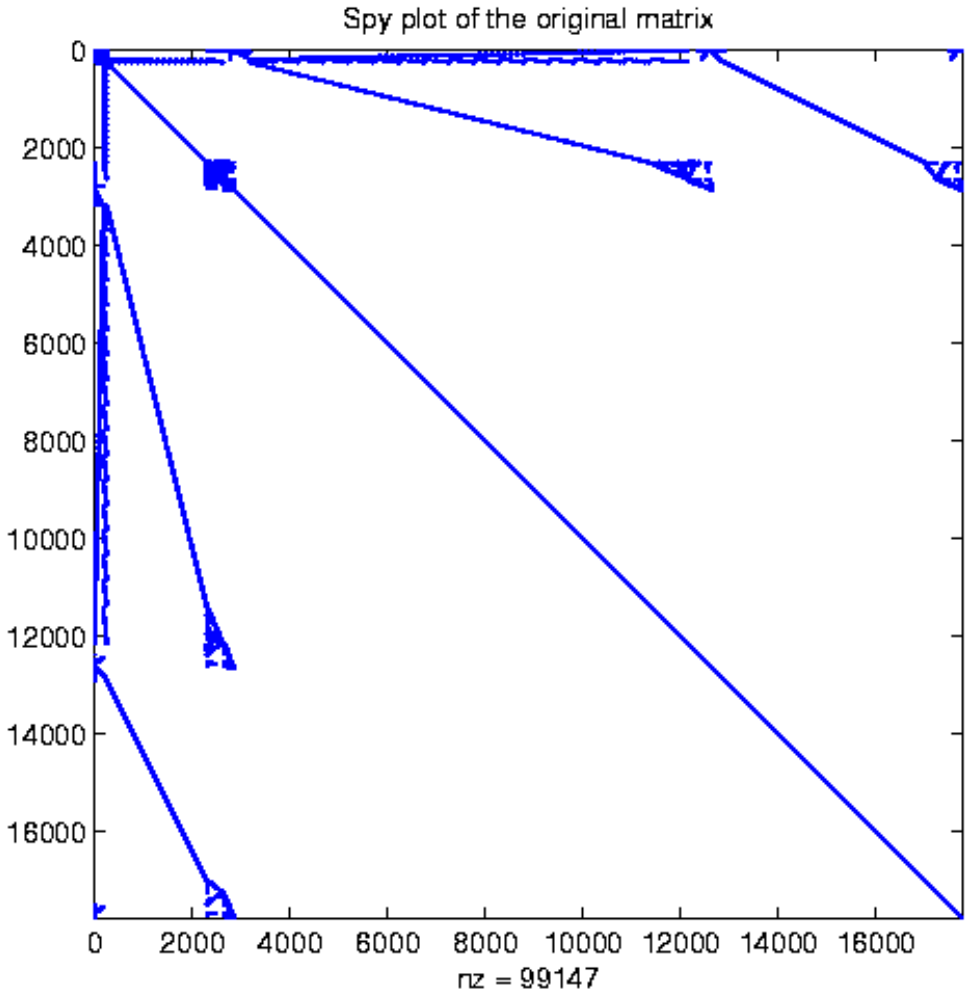
$$p \to \infty \quad (p = n).$$

Taking $p$ much higher does not harm efficiency, and will even optimise for the smaller caches closer to the CPU; *we optimise access for all cache levels irrespective of the cache parameters*.

# Experimental results

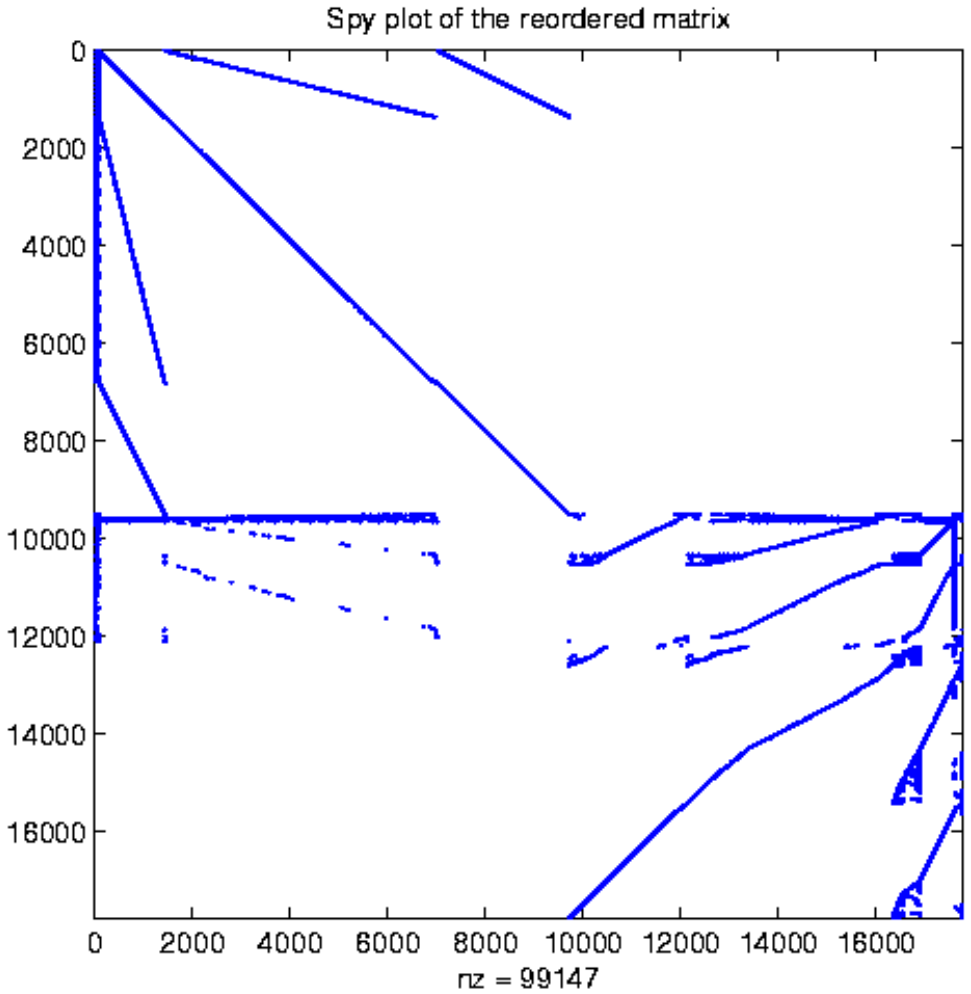


# The memplus matrix



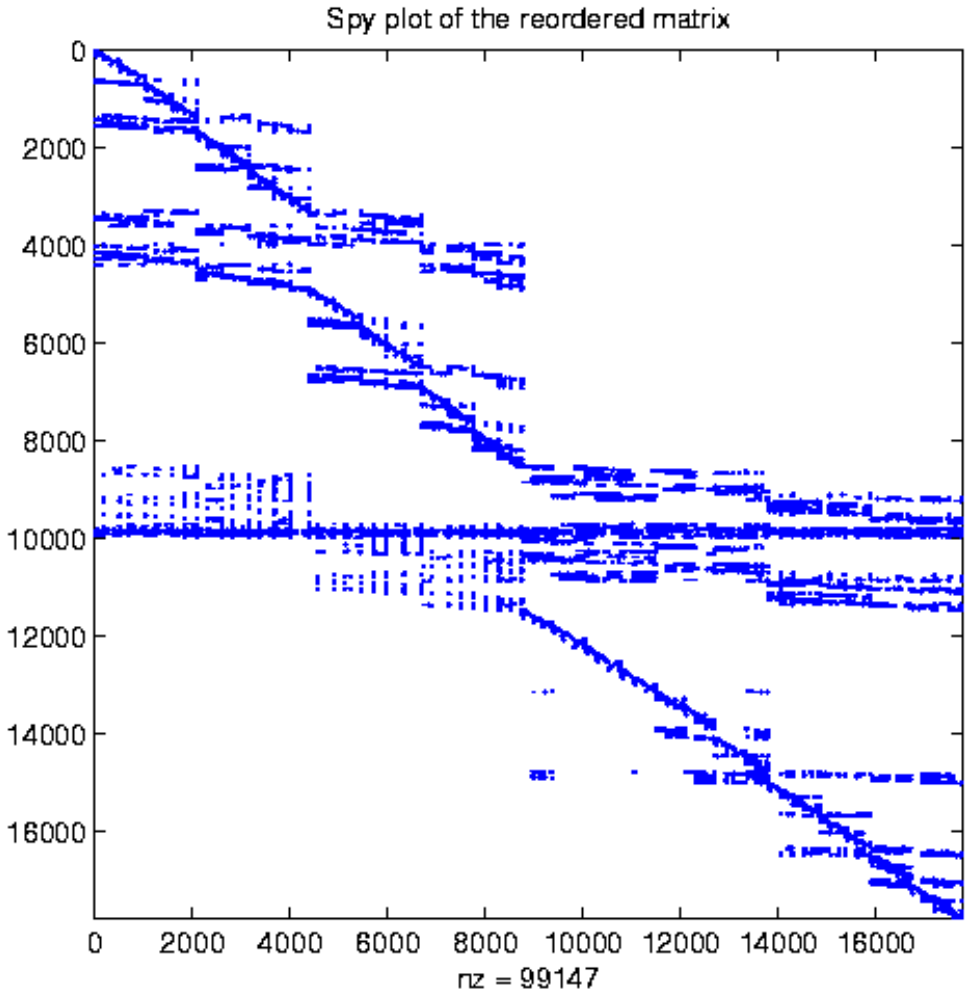

$p = 1$ (original)

# The memplus matrix



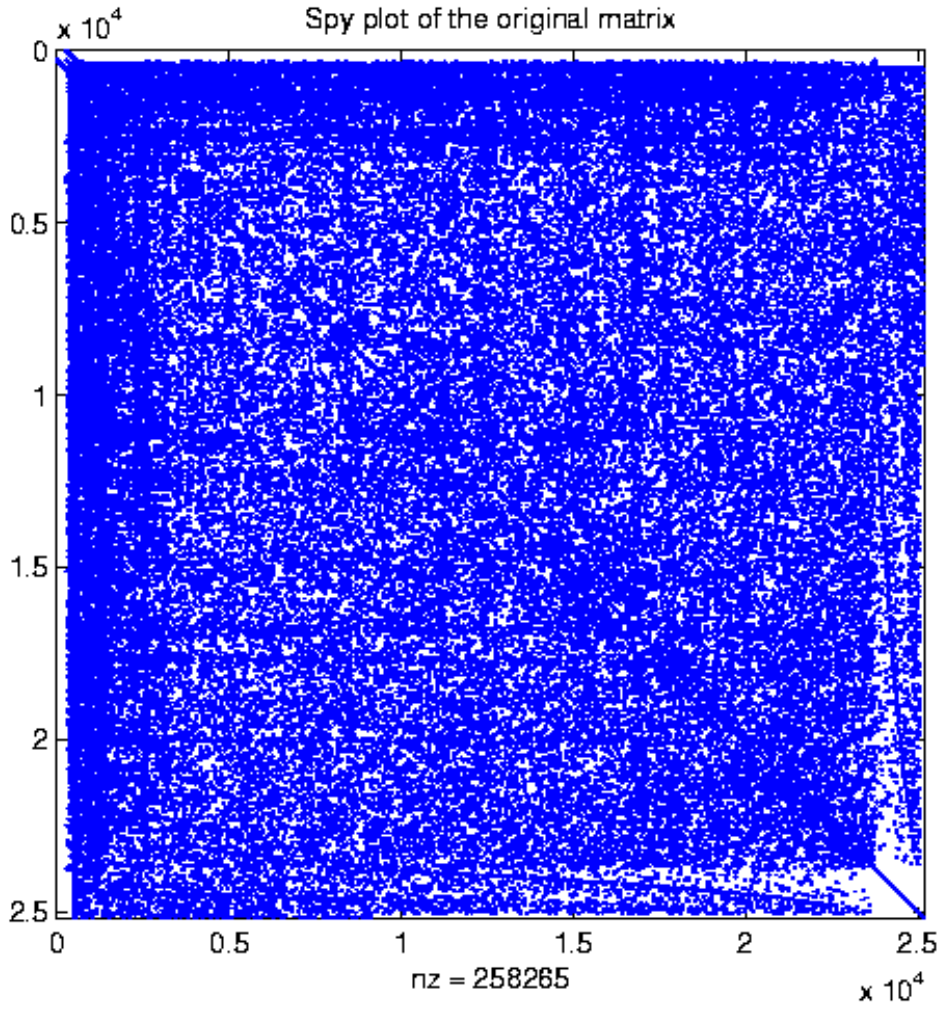

$$p = 2, \quad \epsilon = 0.1$$

# The memplus matrix



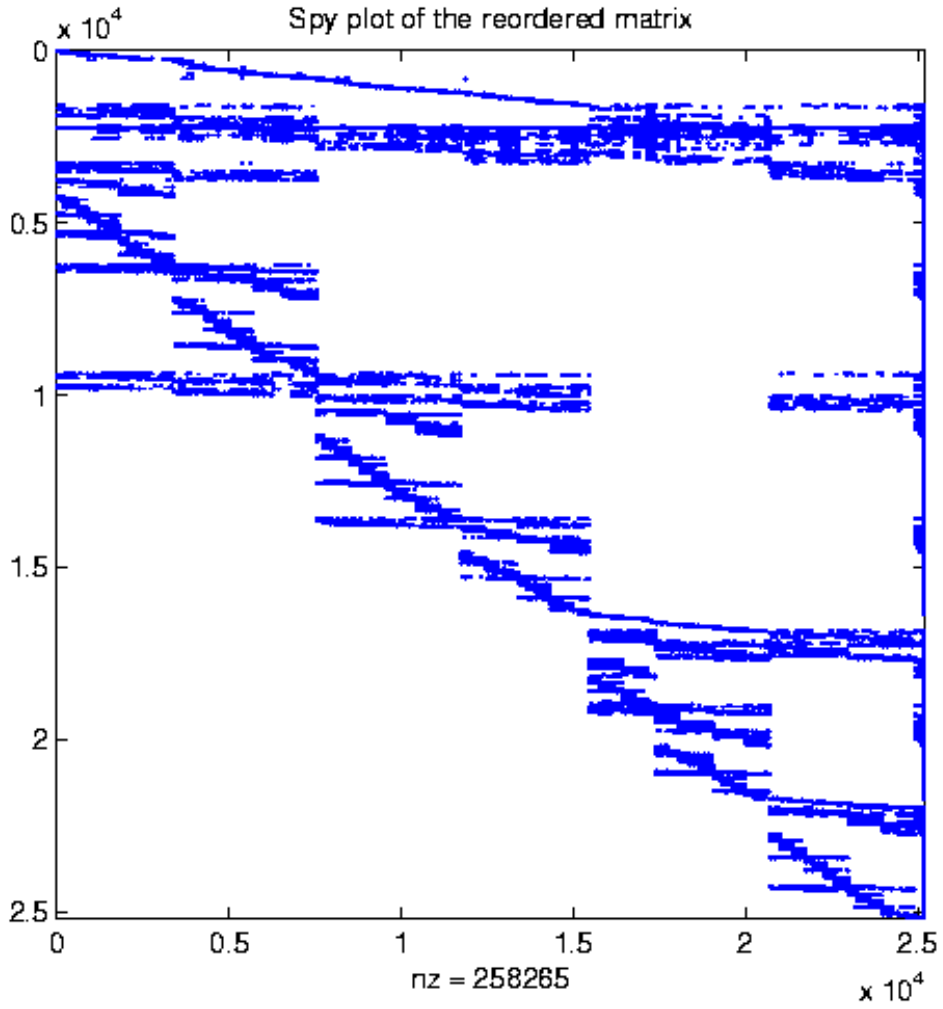

$$p = 100, \quad \epsilon = 0.1$$

# The rhpentium matrix



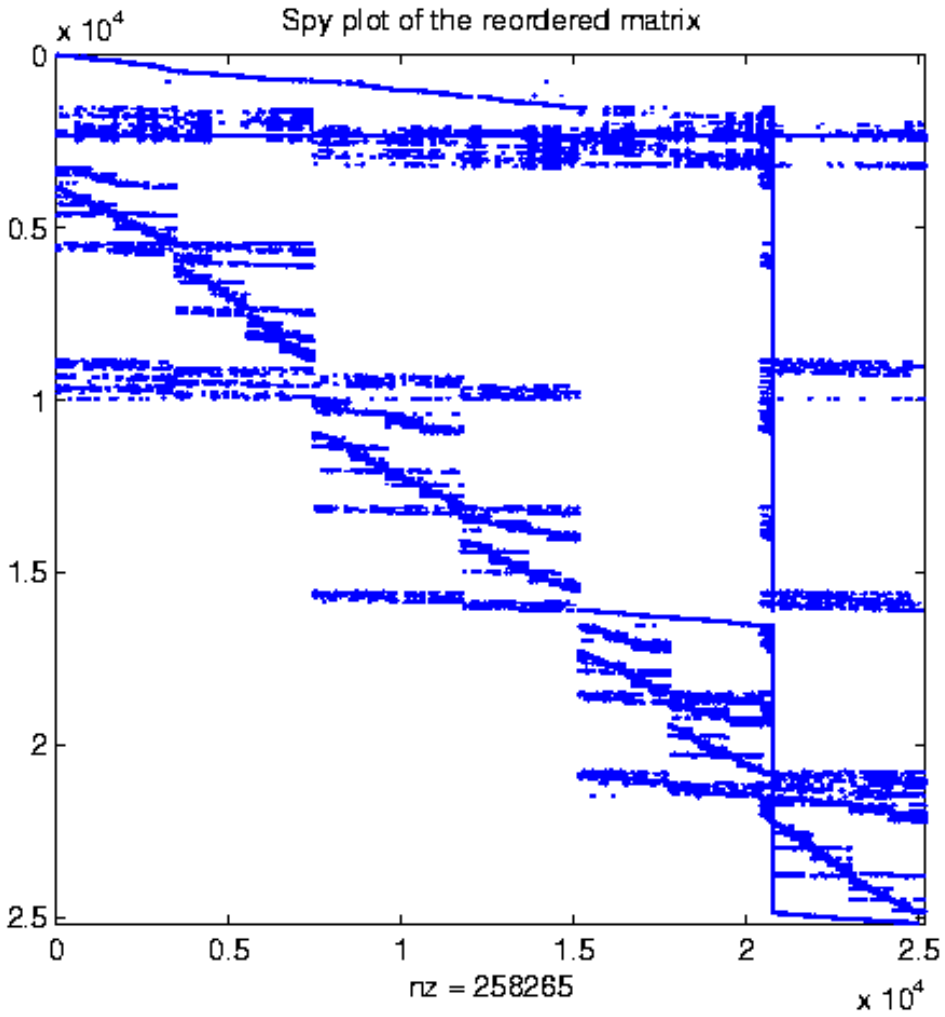

$p = 1$ (original)

# The rhpentium matrix




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

# The rhpentium matrix

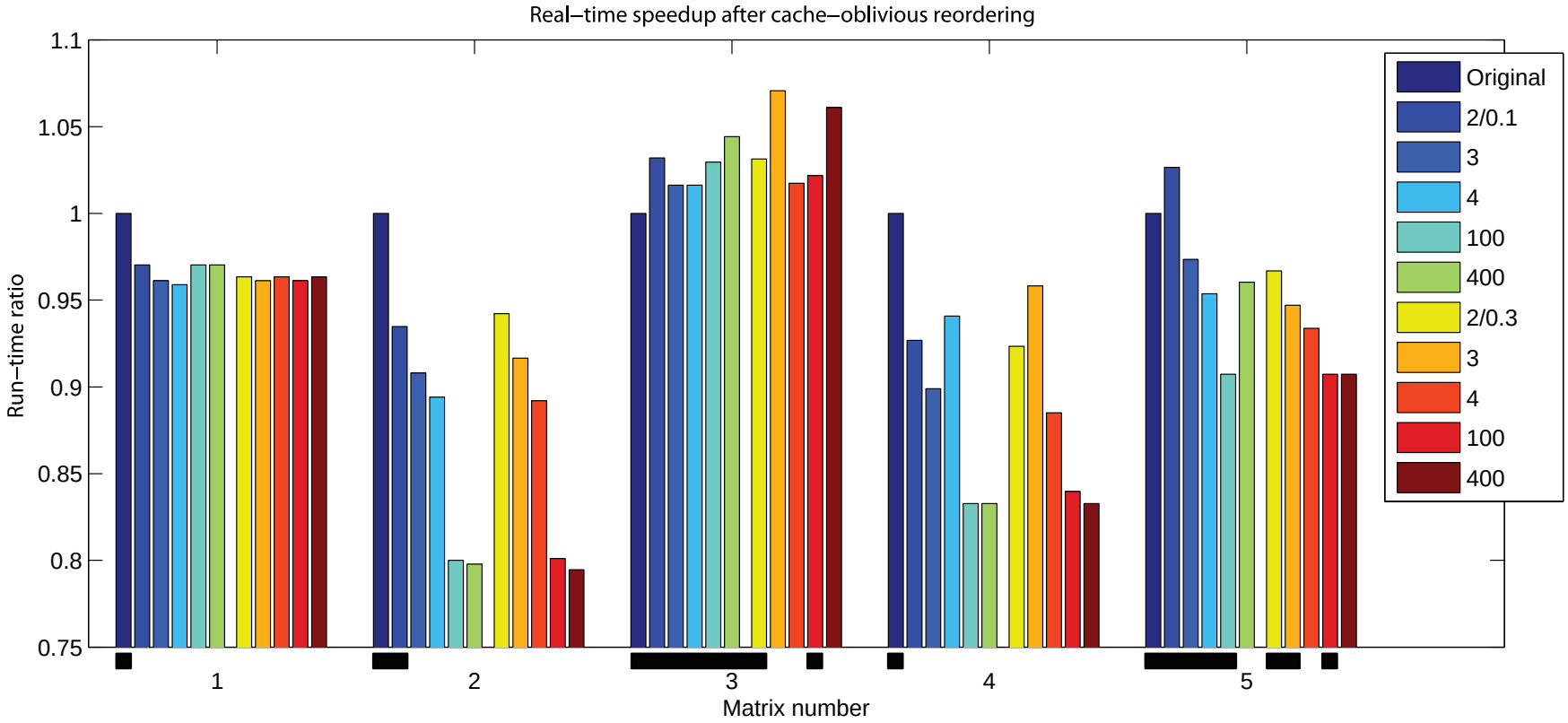



$$p = 400, \quad \epsilon = 0.1$$

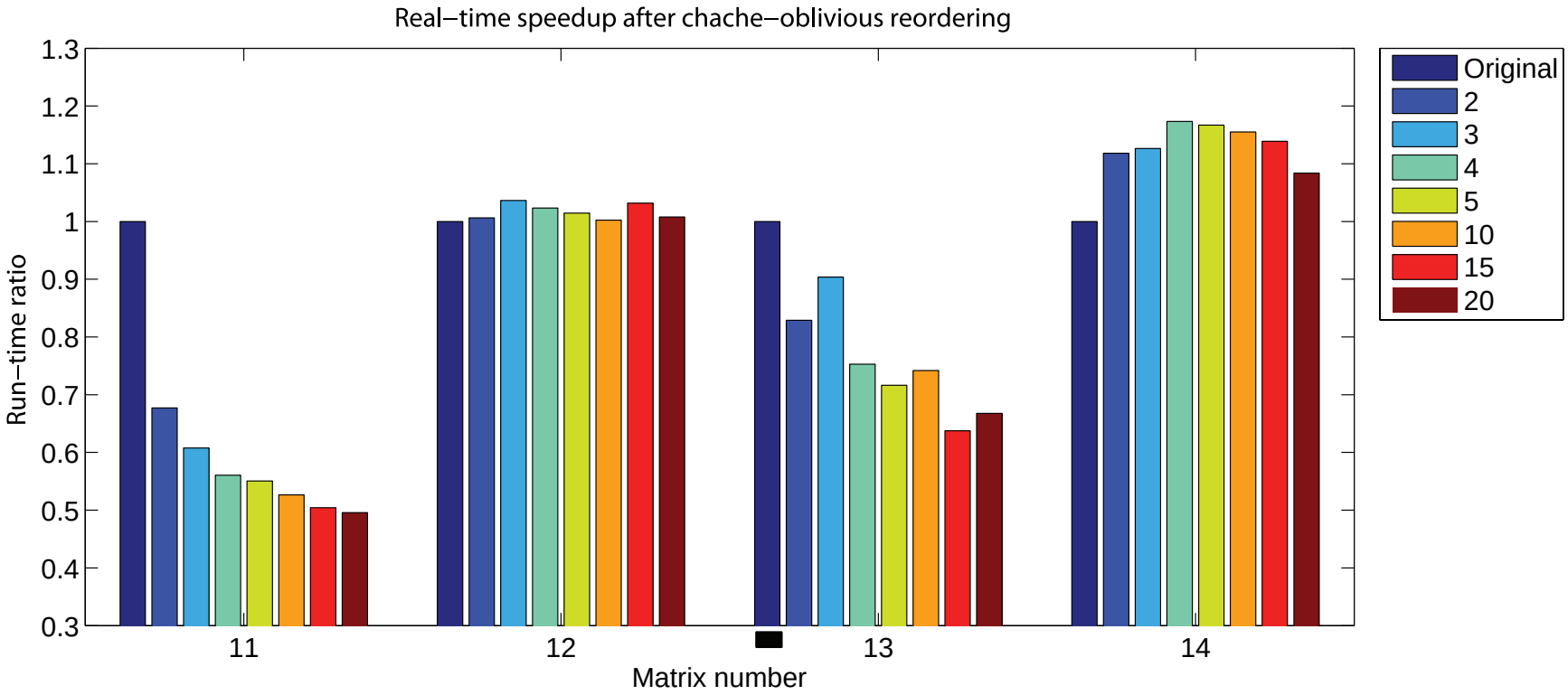
# Cache-simulated results

| Name | $p = 2$ | $p = 100$ | $p = 400$ |
|---|---|---|---|
| memplus | 0.99 (C) | 1.05 (C) | 1.08 (Z) |
| rhpentium | 0.96 (Z) | 0.66 (Z) | 0.63 (I) |
| s3dkt3m2 | 1.00 (I) | 1.00 (C) | 1.00 (C) |
| rand10000 | 0.91 (C) | 0.72 (C) | 0.70 (I) |
| fidap037 | 0.98 (C) | 1.00 (C) | 1.01 (C) |

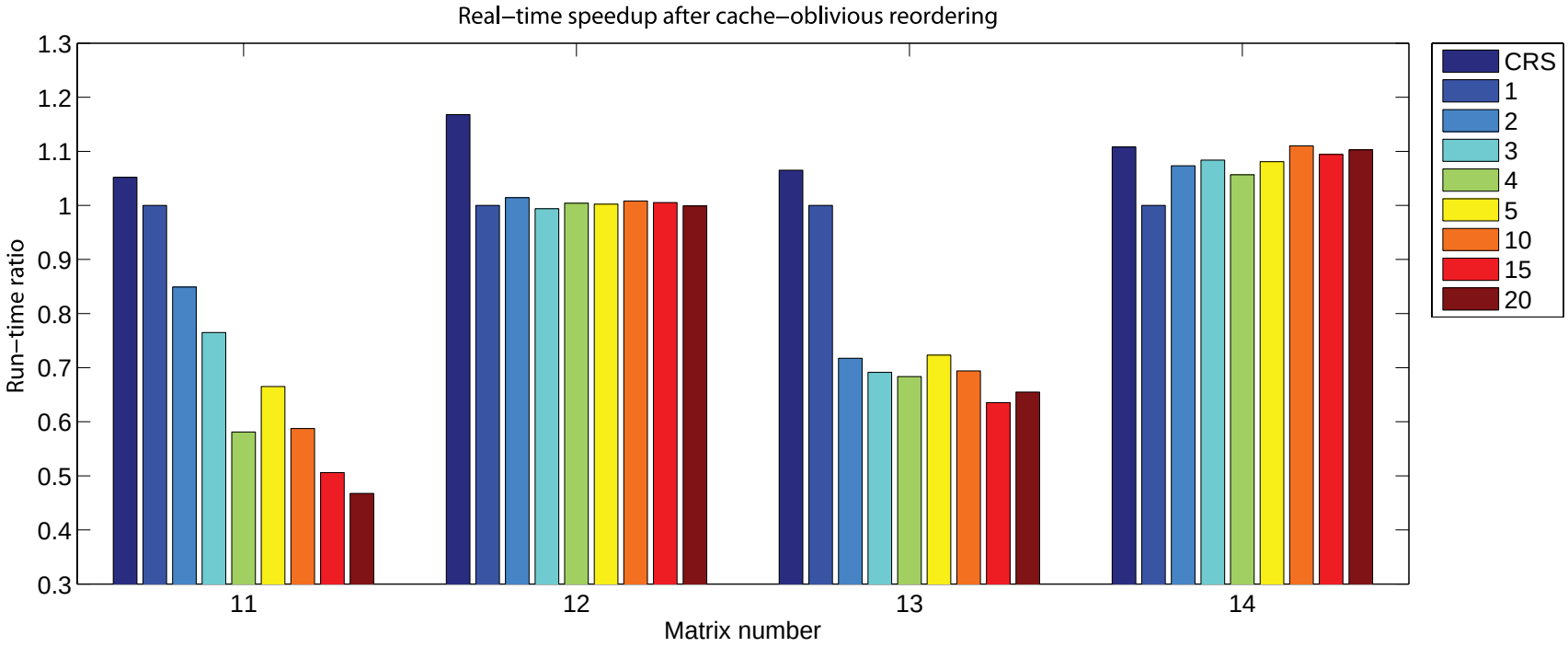
Number of cache misses on reordered matrices ($\epsilon = 0.1$) divided by the number of cache misses on the original matrix during SpMV.
Simulated is an Intel Core2 L1 cache.




Matrices used are: 1. memplus, 2. rhpentium, 3. s3dkt3m2, 4. rand10000, and 5. fidap037.




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




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

# Pre-processing times

| Matrix | Reordering time | SpMV time |
|---|---|---|
| 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.) |

| Name | $p = 2$ | $p = 3$ | $p = 4$ | $p = 100$ | $p = 400$ |
|---|---|---|---|---|---|
| memplus | 1531 | 1914 | 1818 | 12793 | 101744 |
| rhpentium | 5090 | 6752 | 7303 | 17064 | 60251 |
| s3dkt3m2 | 603 | 673 | 740 | 1934 | 5181 |
| rand10000 | 1560 | 1411 | 1820 | 23179 | 103565 |
| fidap037 | 1005 | 1068 | 1131 | 5657 | 12761 |

| Name | $p = 2$ | $p = 3$ | $p = 4$ | $p = 10$ | $p = 20$ |
|---|---|---|---|---|---|
| stanford | 5139 | 7603 | 6828 | 7922 | 8332 |
| stanford_berkeley | 11305 | 13208 | 15093 | 19138 | 21260 |
| wikipedia20051105 | 2152 | 1992 | 2168 | 2570 | 7418 |
| cage14 | 4238 | 3987 | 3635 | 4583 | 5611 |

Table: Cost of reordering in terms of the number matrix multiplications on the original matrix. Here, $\epsilon = 0.1$. Construction times were measured on an Intel Core 2 (Q6600) machine.

# Conclusions & Future Work



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

- remaining cache-oblivious,
- remaining matrix-oblivious, and
- keeping open the option of using specialised sparse BLAS libraries on the background.

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

Future work:

- look into the use of two-dimensional partitioning methods,
- speed up matrix partitioning in Mondriaan.