Sparse Matrix Partitioning, Reordering and Vector Multiplication

Albert-Jan Yzelman, Utrecht University (NL)

May, 2010

Bulk Synchronous Parallel

Supercomputers...

Many different processor types:

- Reduced Instruction Set chips (RISC), (e.g., IBM Power)
- Intel Itanium
- x86-type (your average home PC/laptop)
- Vector (co-)processors
- GPUs
- Stream processors
- ...

Supercomputers...

Many different connectivity;

- Ring
- All-to-all ethernet
- InfiniBand
- Cube
- Hierarchical
- Internet
- ...

Programming Model

One model;

bridging models:

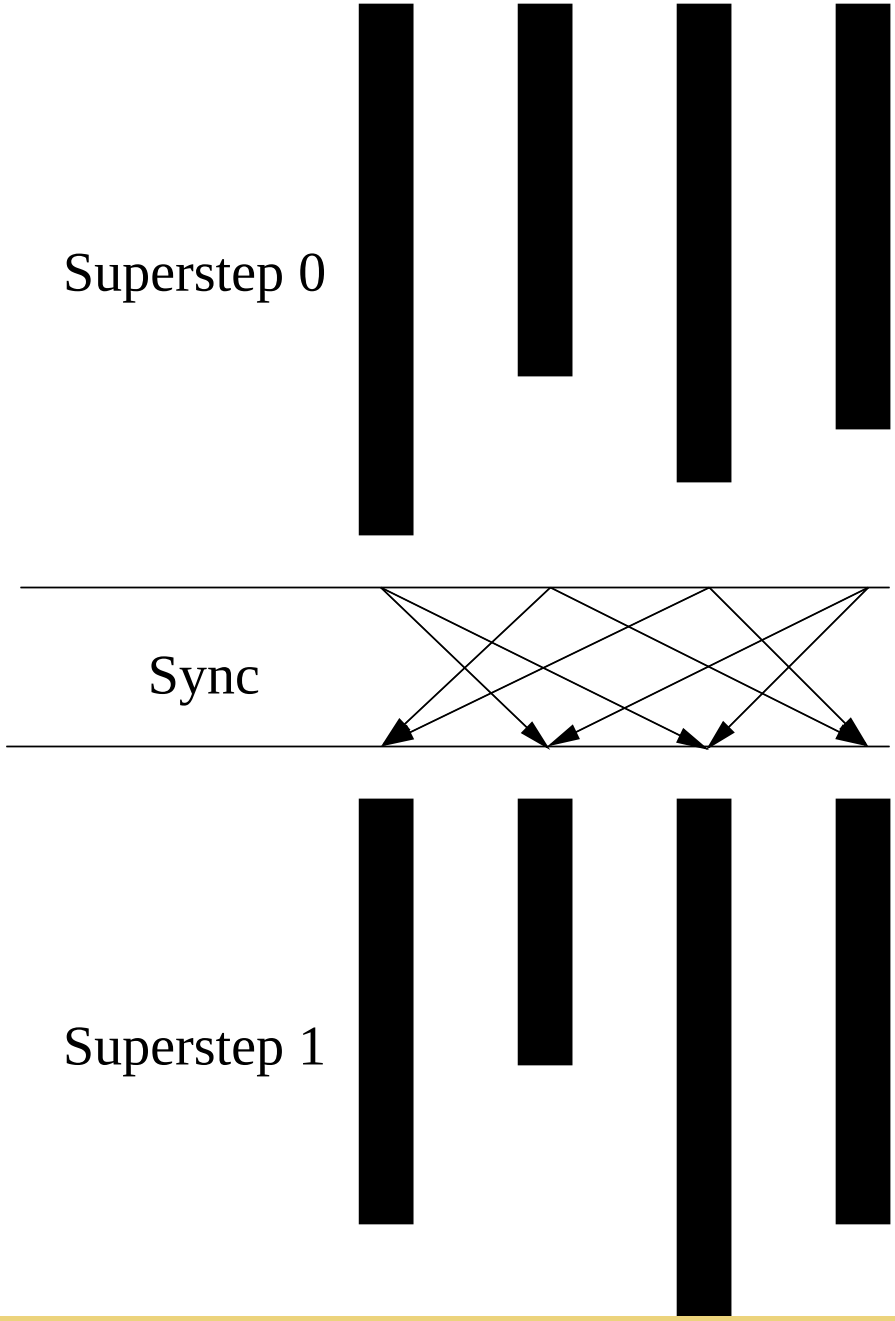
- Message Passing Interface (MPI)
- **Bulk Synchronous Parallel** (BSP)

Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, Volume 33 (1990), pp. 103–111

Bulk Synchronous Parallel

A BSP-computer:

- consists of P processors, each with local memory
- executes a Single Program on Multiple Data (SPMD)
- performs no communication during calculation
- communicates only during *barrier synchronisation*


Superstep 0
Sync
Superstep 1

Bulk Synchronous Parallel

A BSP-computer furthermore:

- has homogenous processors, able to do r flops each second
- takes l time to synchronise
- has a communication speed of g

The model thus only uses four parameters (P, r, l, g).

Bulk Synchronous Parallel

A BSP-*algorithm* can (using BSPlib, BSPonMPI):

- Ask for some environment variables:
  `bsp_nprocs()`
  `bsp_pid()`
- Synchronise:
  `bsp_sync()`
- Perform "direct" remote memory access (DRMA):
  `bsp_put(source, dest, dest_PID)`
  `bsp_get(source, source_PID, dest)`
- Send messages, synchronously (BSMP):
  `bsp_send(data, dest_PID)`
  `bsp_move()`

Exercise

Design a parallel inner-product calculation:

```
double spmd_ip( double *x, double *y, int length ) {
    double sum=...
    ...
    return sum;
}
```

Using:
`bsp_nprocs()`, `bsp_pid()`, `bsp_put(...)`

Exercise

Design a parallel inner-product calculation:

```
double spmd_ip( double *x, double *y, int length ) {
    int i = 0;
    double sum = x[0]*y[0];
    double res[ bsp_nprocs() ];
    for(i=1; i<length; i++)
        sum += x[i]*y[i];
    for(i=0; i<bsp_nprocs(); i++)
        bsp_put(&sum,&res[bsp_pid()],i);
    bsp_sync();
    for(i=0; i<bsp_nprocs(); i++)
        sum += res[i];
    return sum;
}
```

BSP cost model:

- let $w_i^{(s)}$ be the *work* to be done by processor s in superstep i,
- let $r_i^{(s)}$ be the communication *received* by processor s between superstep i and $i+1$,
- let $t_i^{(s)}$ be the communication *transmitted* by processor s.

Furthermore, let T be the total number of supersteps and the *communication bound* of superstep i be given by

$$c_i = \max \left\{ \max_{s \in [0,P-1]} r_i^{(s)}, \max_{s \in [0,P-1]} t_i^{(s)} \right\}.$$

Similarly, the upper bound for the amount of work:

$$w_i = \max_{s \in [0,P-1]} w_i^{(s)}.$$

Then the cost of a BSP algorithm is given by:

$$\sum_{i=0}^{T} w_i + \sum_{i=0}^{T-1} (l + g \cdot c_i)$$

Exercise

Now think on how to do Sparse Matrix–Vector multiplication (SpMV) using BSP.

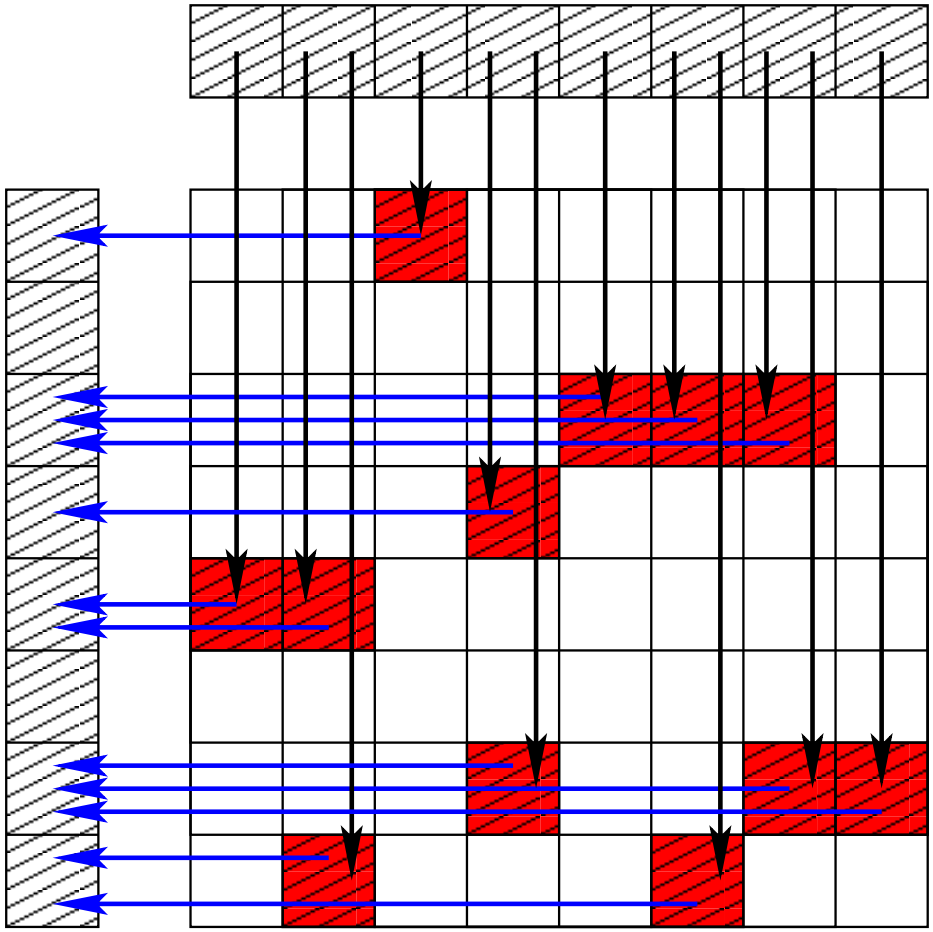
Matrix partitioning for SpMV

Sparse matrix, dense vector multiplication

`y=Ax`:

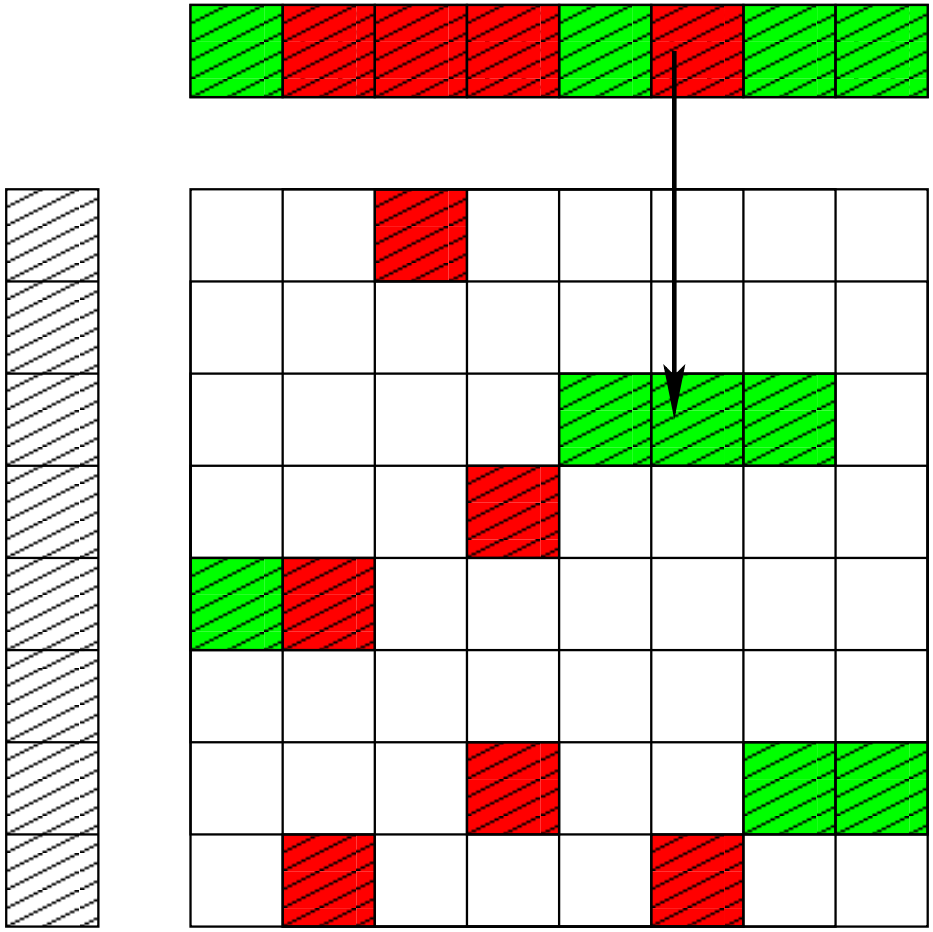
for each nonzero k **from** A

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



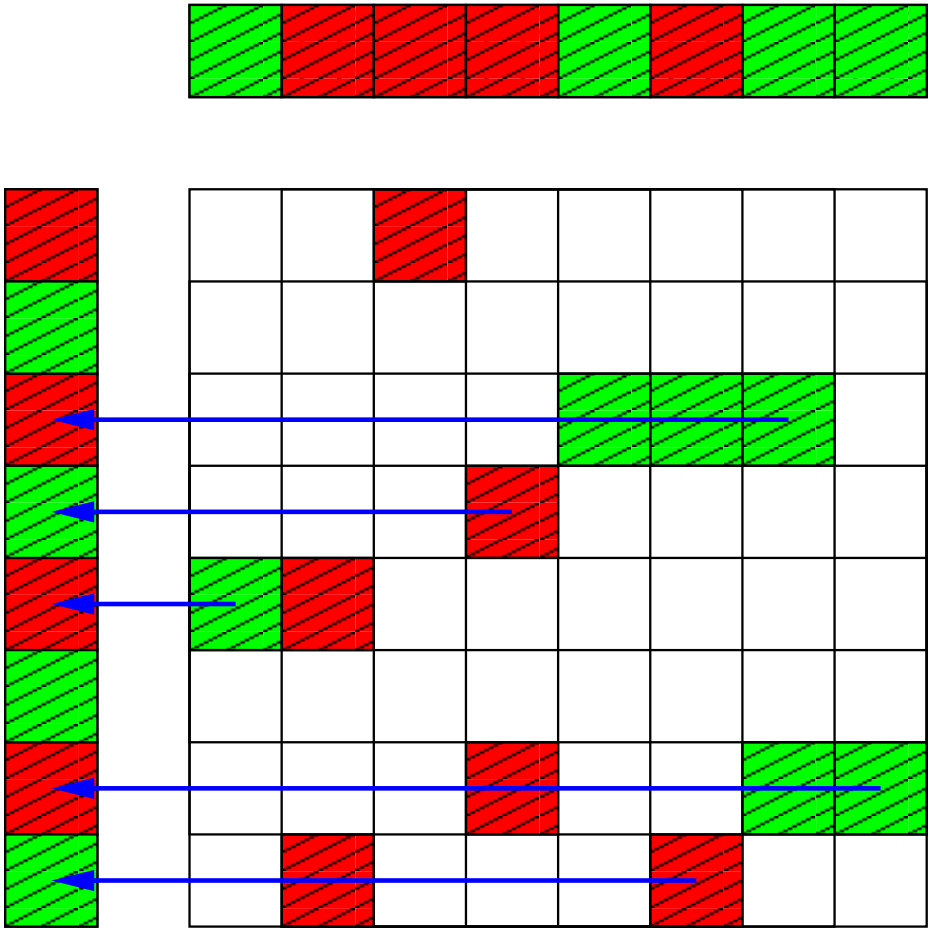
Sparse matrix, dense vector multiplication (parallel)

Step 1 (*fan-out*):
Not all processors have the elements from x they need;
processors need to get the missing items.



Sparse matrix, dense vector multiplication (parallel)

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



Sparse matrix, dense vector multiplication

The algorithm:

for all nonzeroes k **from** A
  if $x[k.j]$ and $x[k.i]$ are local
    add $k.v * x[k.j]$ **to** $y[k.i]$
  if $x[k.j]$ is not local
    get $x[k.j]$ from the responsible processor
  if $y[k.i]$ is not local
    set locally $y[k.i] = 0$
synchronise
for all nonzeroes k **from** A not yet processed
  add $k.v \cdot x[k.j]$ **to** $y[k.i]$
send all local copies $y[i]$ to the responsible processor
synchronise
for all incoming pairs (i, v) (such that $v = y[i]$)
  add v *to* $y[i]$

In the case of sparse matrix–vector multiplication,

what causes the communication?

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

(Assuming the vector distribution of x/y is such that $x[i]$ is distributed to the same processor as at least one entry of the ith column/row of A.)

In the case of sparse matrix–vector multiplication,

what causes the communication?

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

(Assuming the vector distribution of x/y is such that $x[i]$ is distributed to the same processor as at least one entry of the ith column/row of A.)

Easy solution: distribute all nonzeroes to the same processor

Load balancing

For each superstep i, let $\bar{w}_i = \frac{1}{P} \sum_{s \in [0,P-1]} w_i^{(s)}$ be the average workload. We demand that:

$$w_i^{(s)} \leq (1 + \epsilon)\bar{w}_i,$$

where ϵ is the maximum load imbalance parameter.

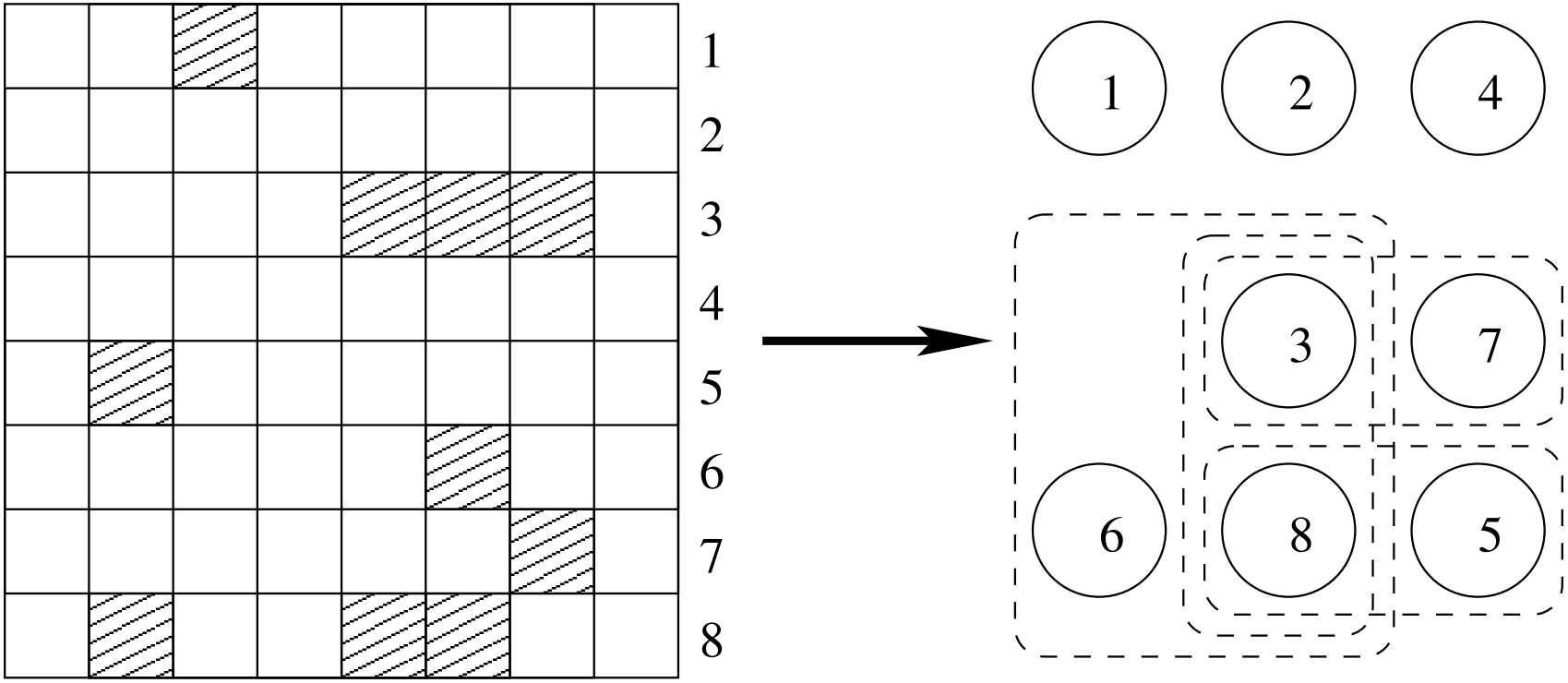
Communication balancing

Recall that the communication-part is assumed to depend on the maximum incoming or outgoing volume over all processors;
for each superstep, we minimise

$$c_i = \max \left\{ \max_{s \in [0,P-1]} r_i^{(s)}, \max_{s \in [0,P-1]} t_i^{(s)} \right\}$$

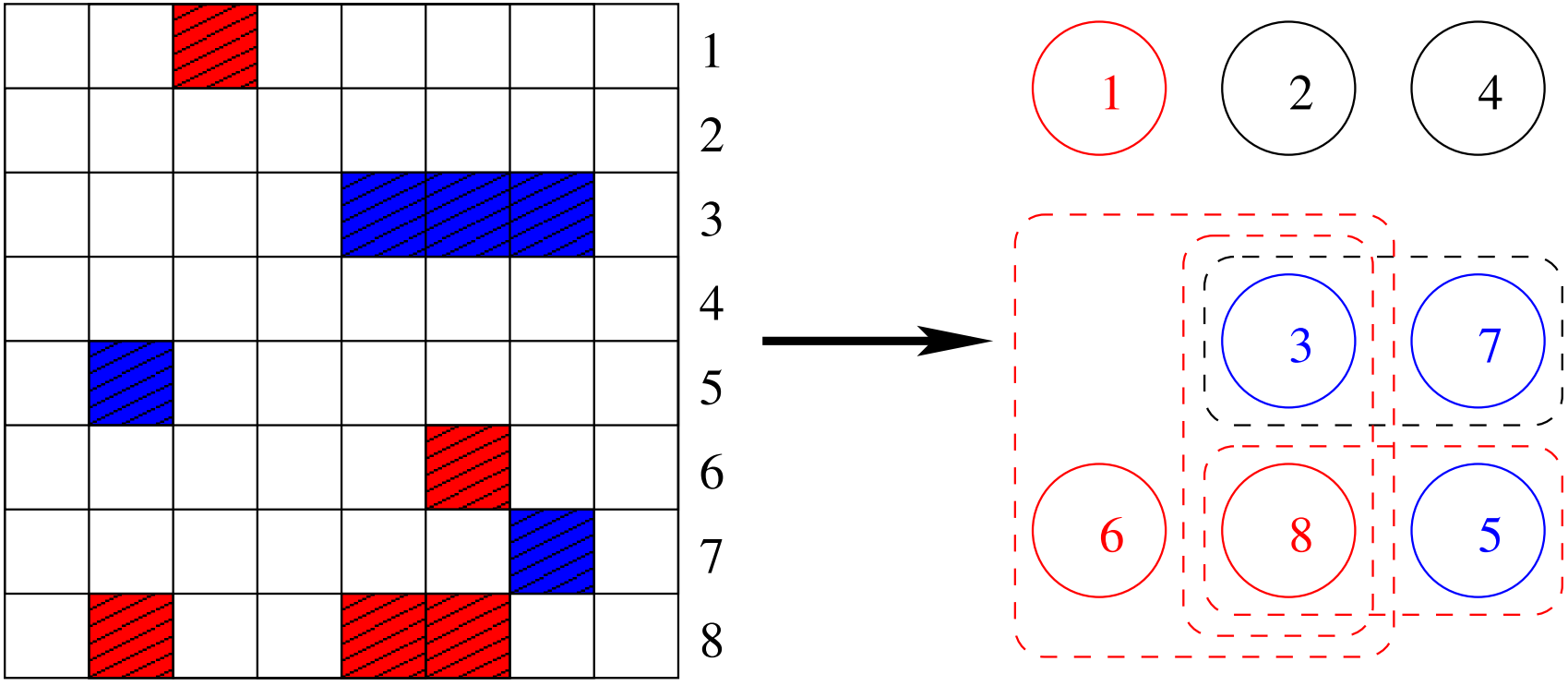
not the total communication volume $\sum_s \left(r_i^{(s)} + t_i^{(s)} \right)$.

"Shared" columns: communication during fan-out



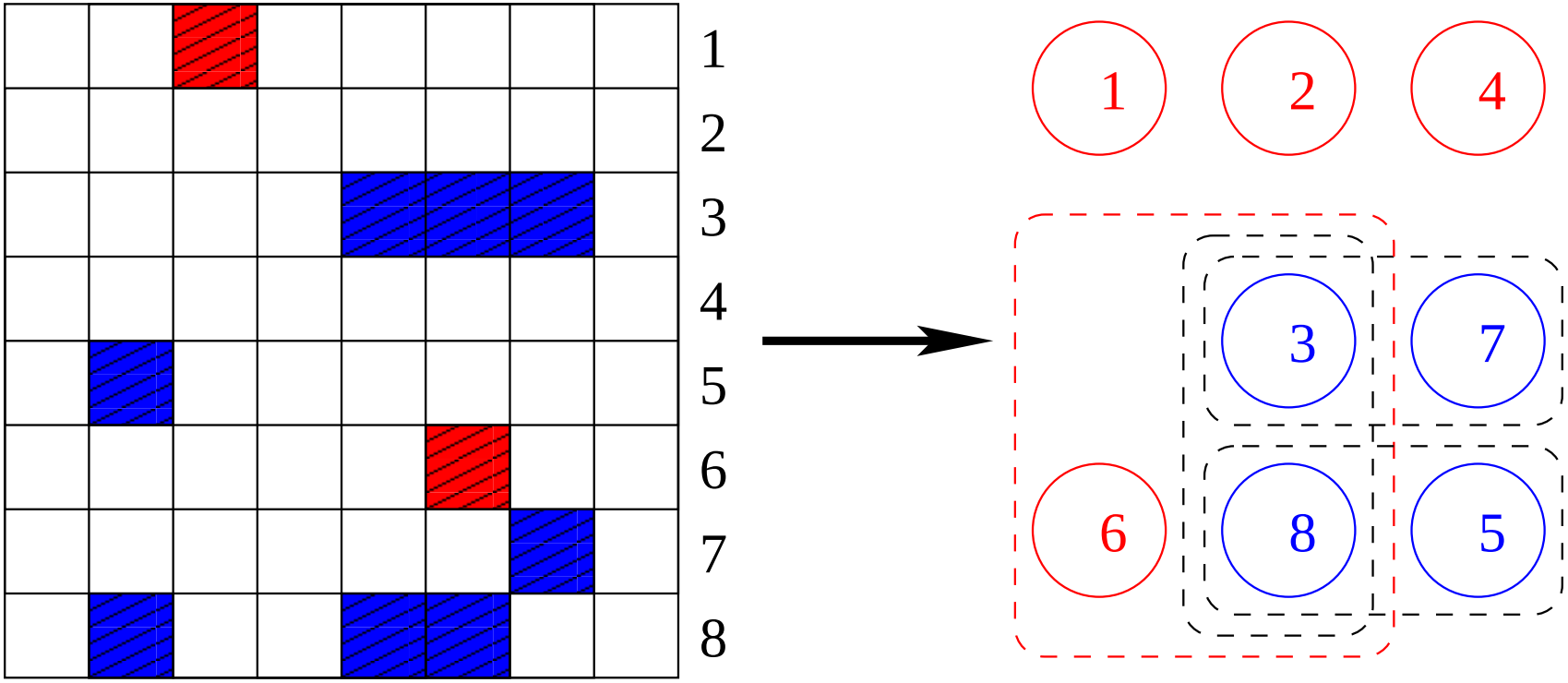
Column-net model; a cut net means a shared column

"Shared" columns: communication during fan-out



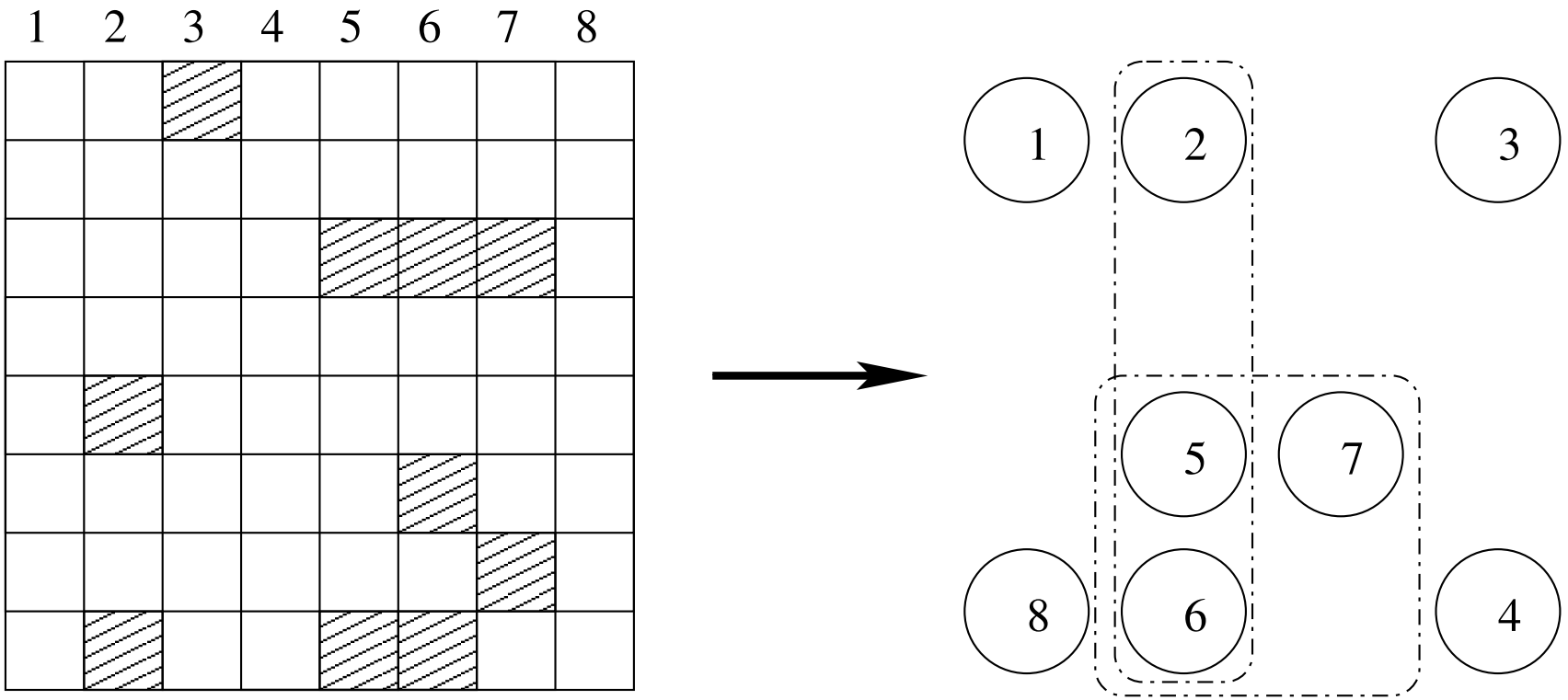
Column-net model; a cut net means a shared column

"Shared" columns: communication during fan-out



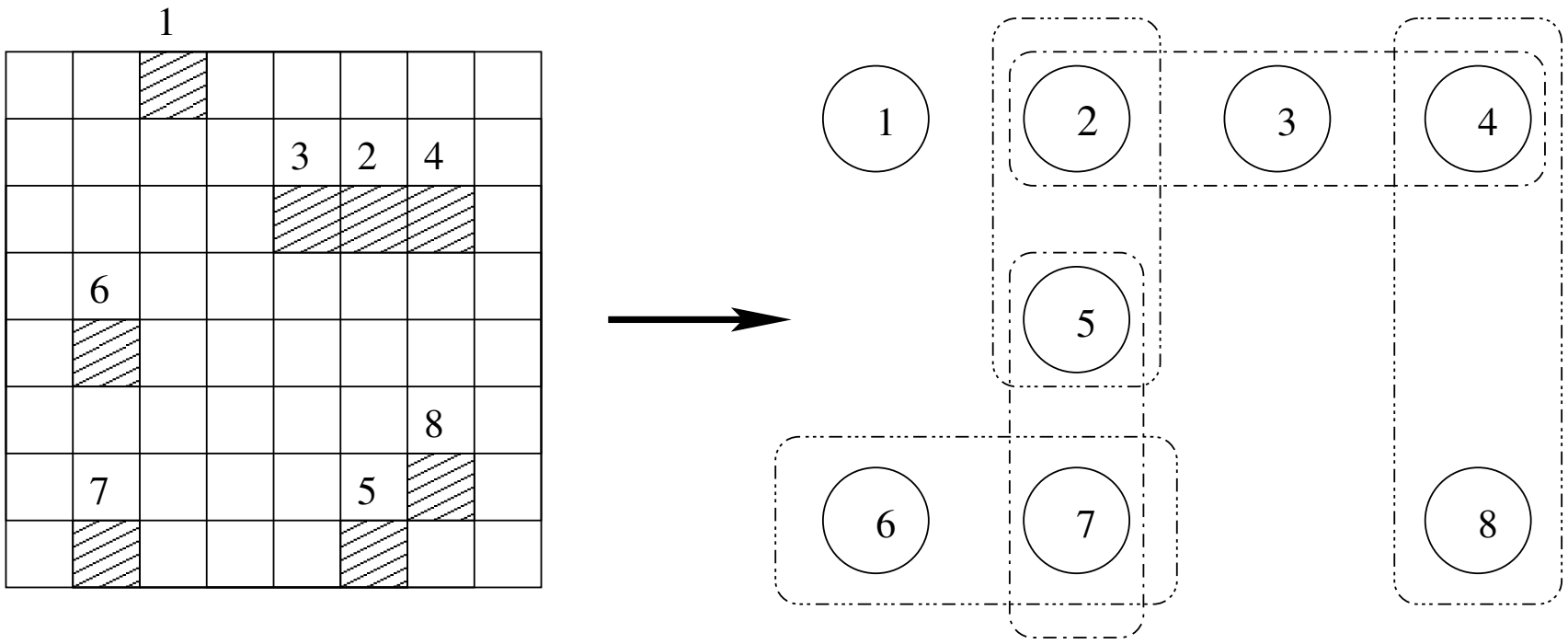
Column-net model; a cut net means a shared column

"Shared" rows: communication during fan-in



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

"Catch" all communication:



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

Emerging partitioning strategy:

- Model the sparse matrix using a hypergraph

Emerging partitioning strategy:

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

Emerging partitioning strategy:

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

Criteria:

load balance (max. ϵ imbalance) + minimising

$$\sum_i (\lambda_i - 1),$$

where λ_i equals the number of partitions within the ith net; the *connectivity* of the ith net. Alternative:

$$\sum_i \begin{cases} 1, & \text{if } \lambda_i > 0, \\ 0, & \text{otherwise} \end{cases} .$$

Emerging partitioning strategy:

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

Kernighan & Lin, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Technical Journal 49 (1970): pp. 291-307

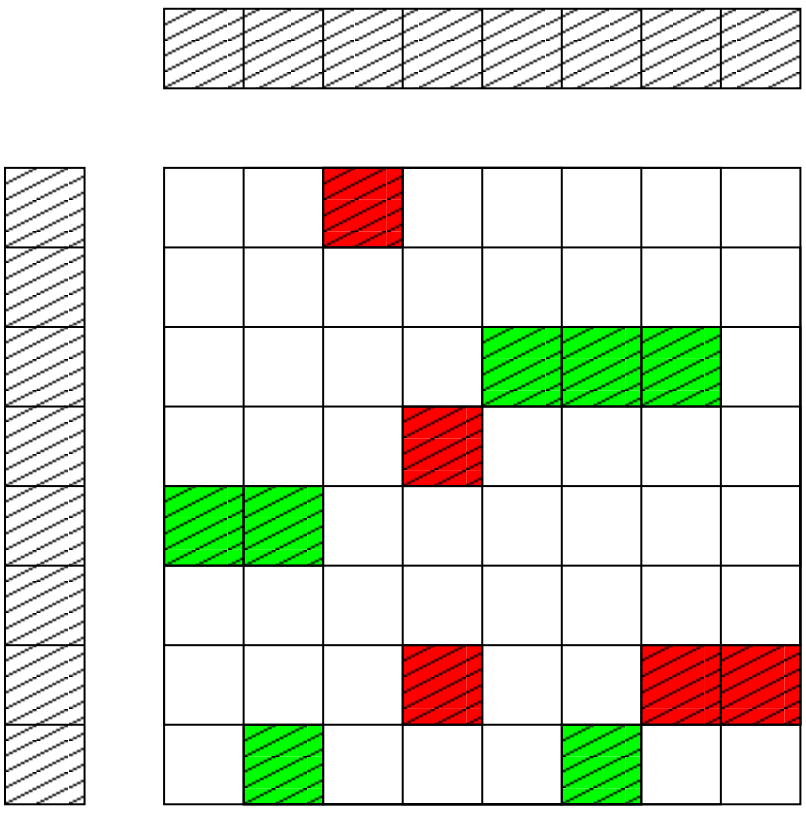
Fiduccia & Mattheyses, *A linear-time heuristic for improving network partitions*, Proceedings of the 19th IEEE Design Automation Conference (1982), pp. 175-181

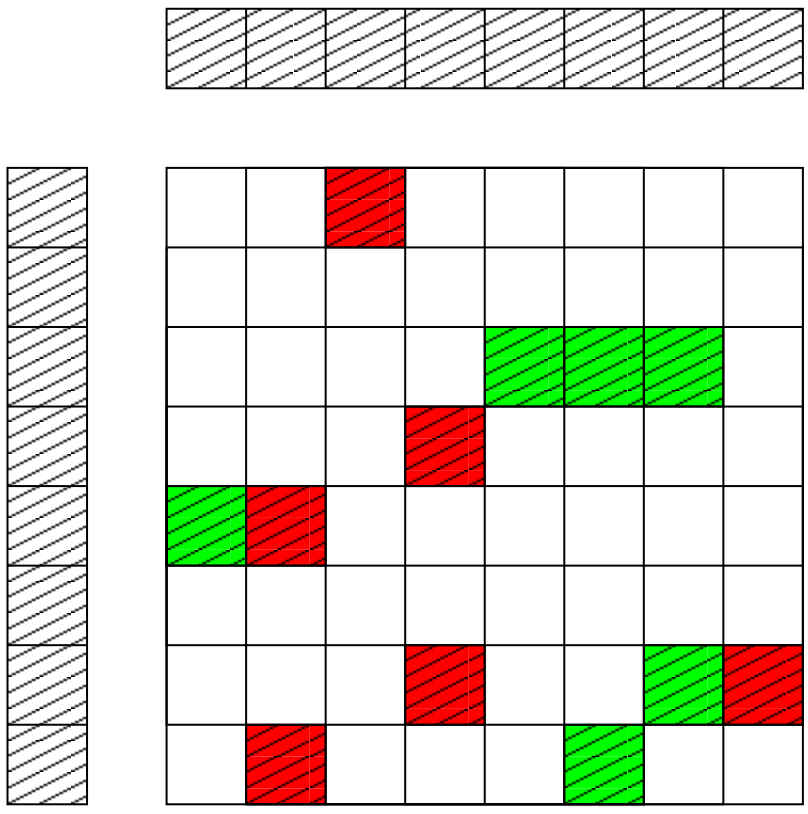
Catalyürek & Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel Distributed Systems 10 (1999), pp. 673-693

Emerging partitioning strategy:

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

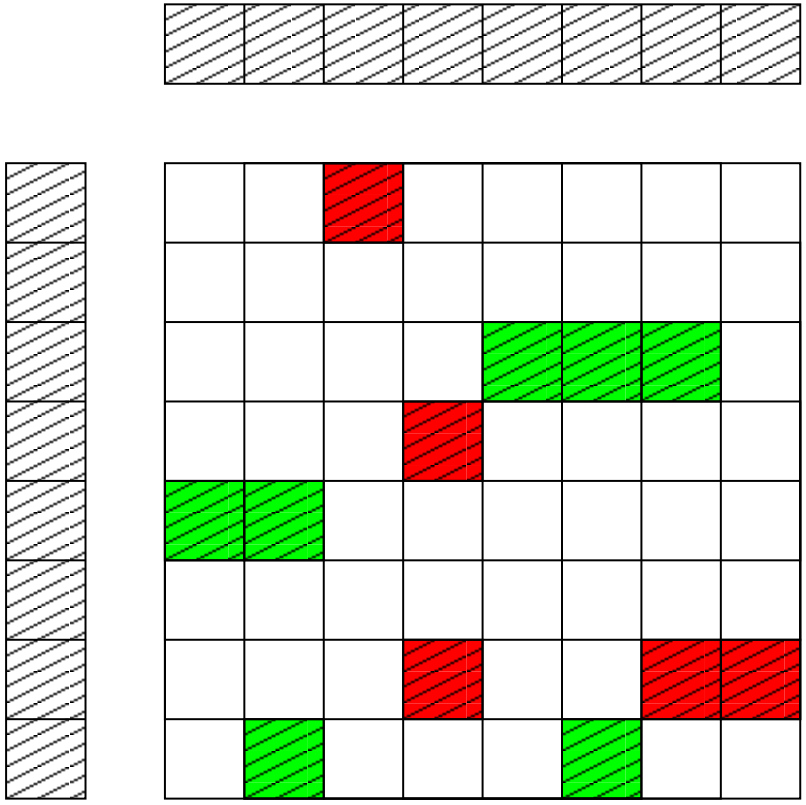
Original Mondriaan: both row- and column-net, and choose best





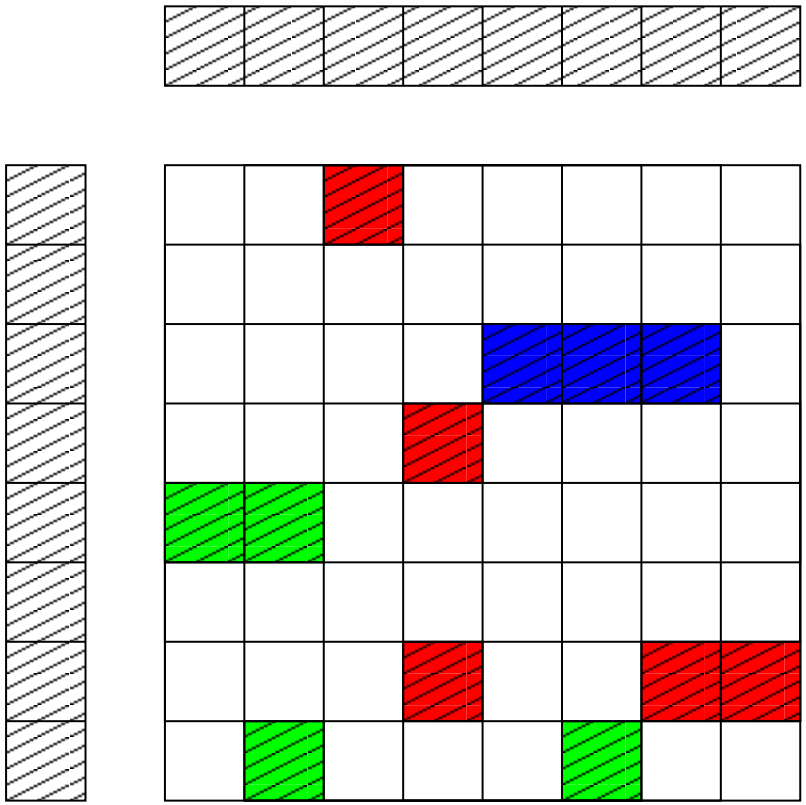
Emerging partitioning strategy:

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



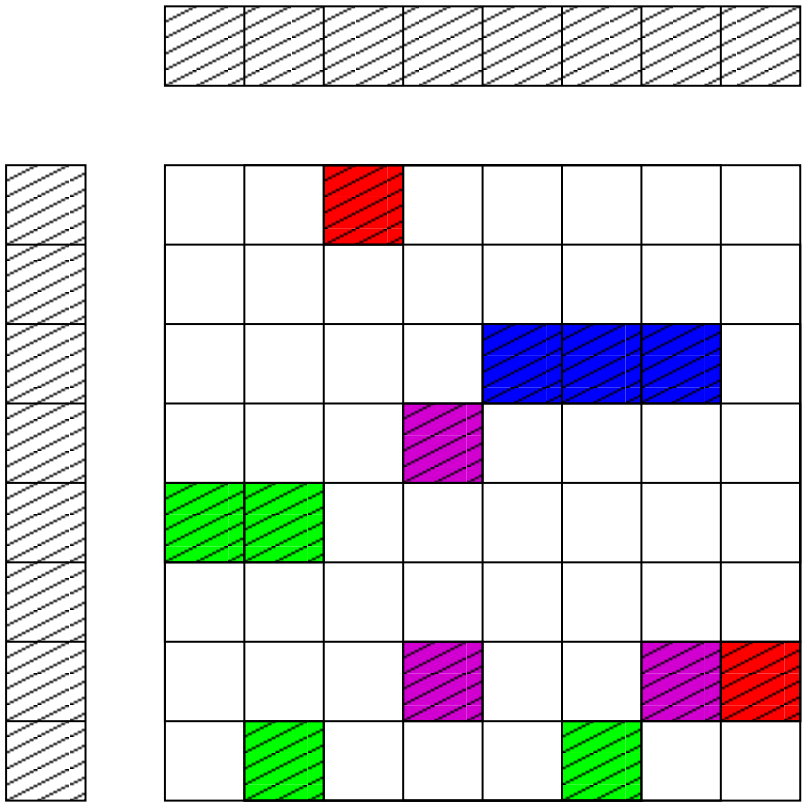
Emerging partitioning strategy:

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



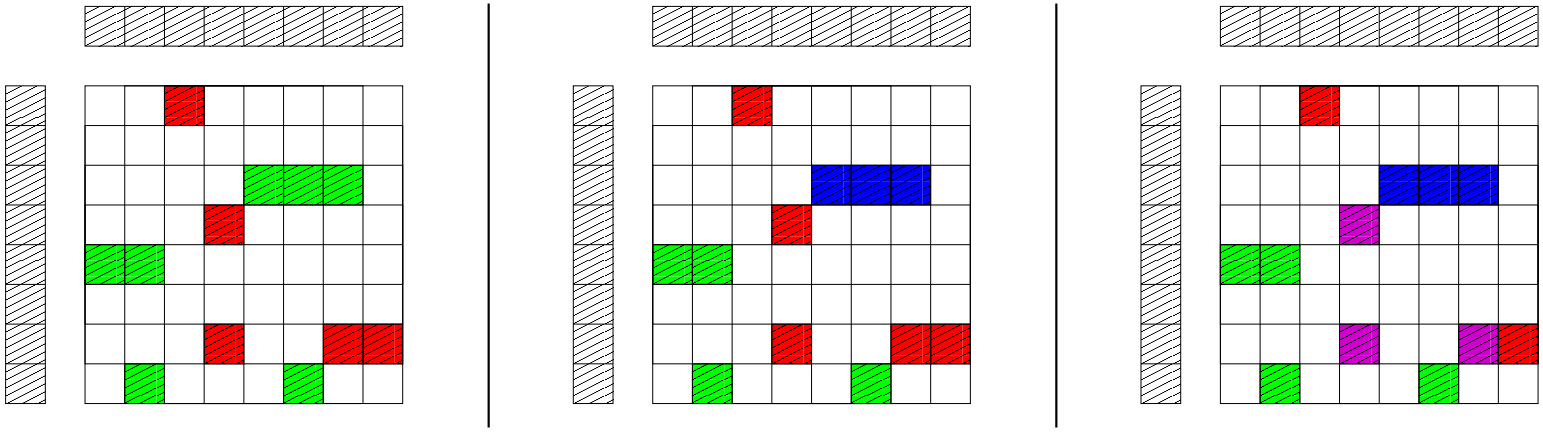
Emerging partitioning strategy:

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



Mondriaan:

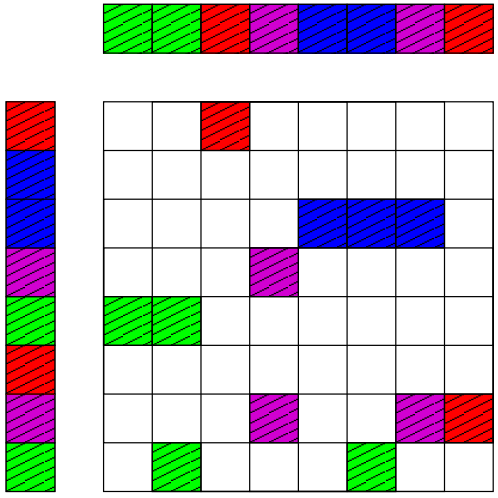
- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



Brendan Vastenhouw and Rob H. Bisseling, *A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication*, SIAM Review, Vol. 47, No. 1 (2005) pp. 67-95

Mondriaan:

- 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**



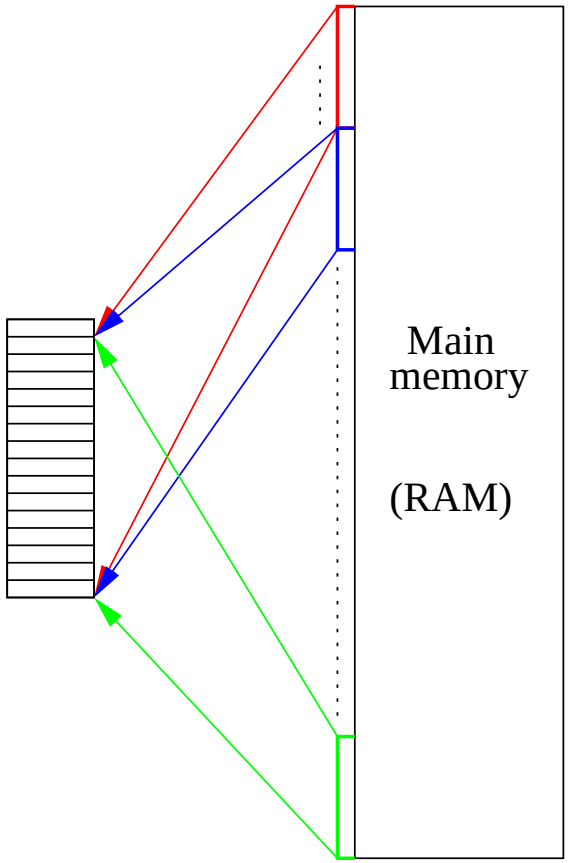
Rob H. Bisseling and Wouter Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis, Vol. 21 (2005) pp. 47-65

Reordering for Sequential SpMV

Naive cache

$k = 1$, modulo mapped cache

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



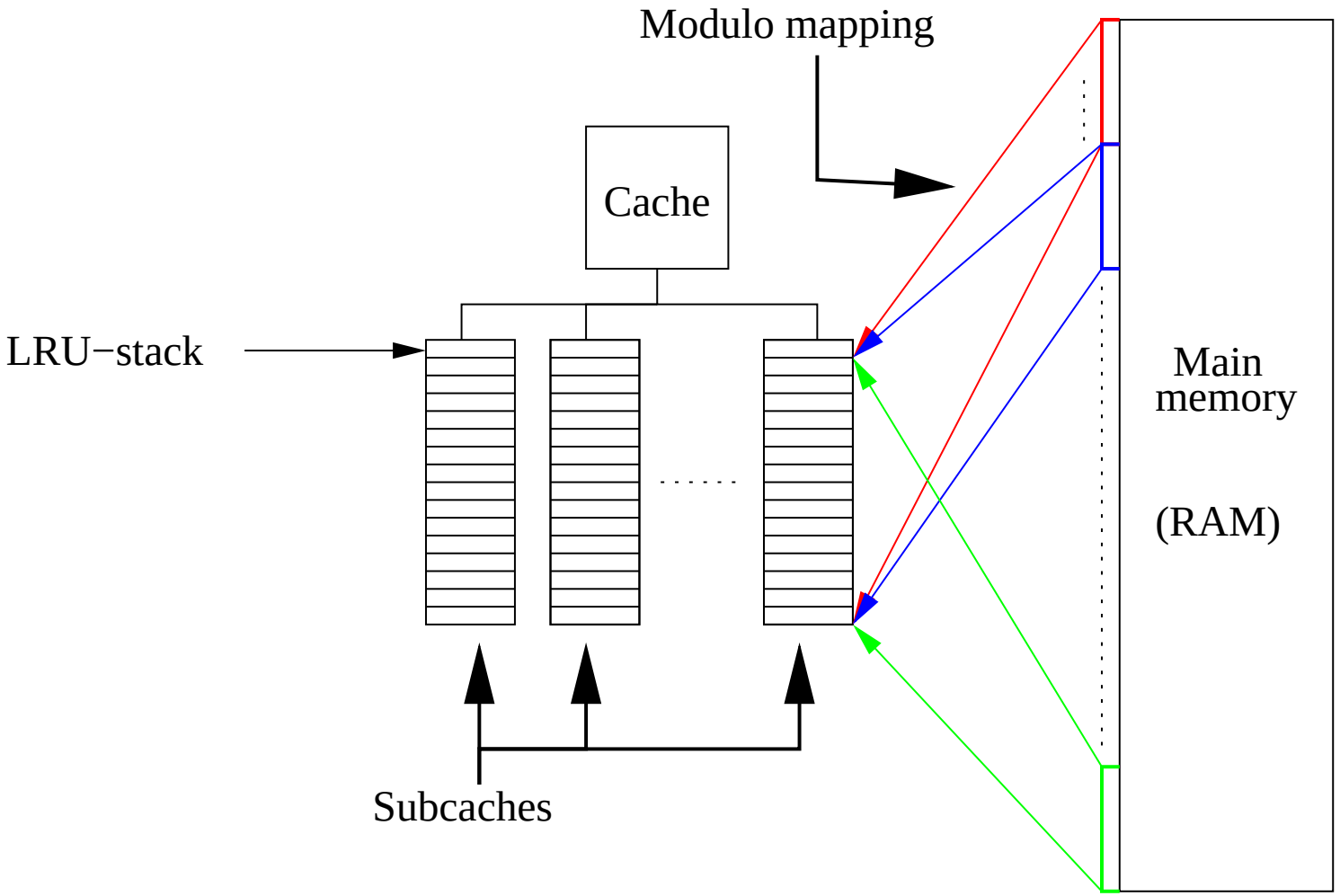
'Ideal' cache

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

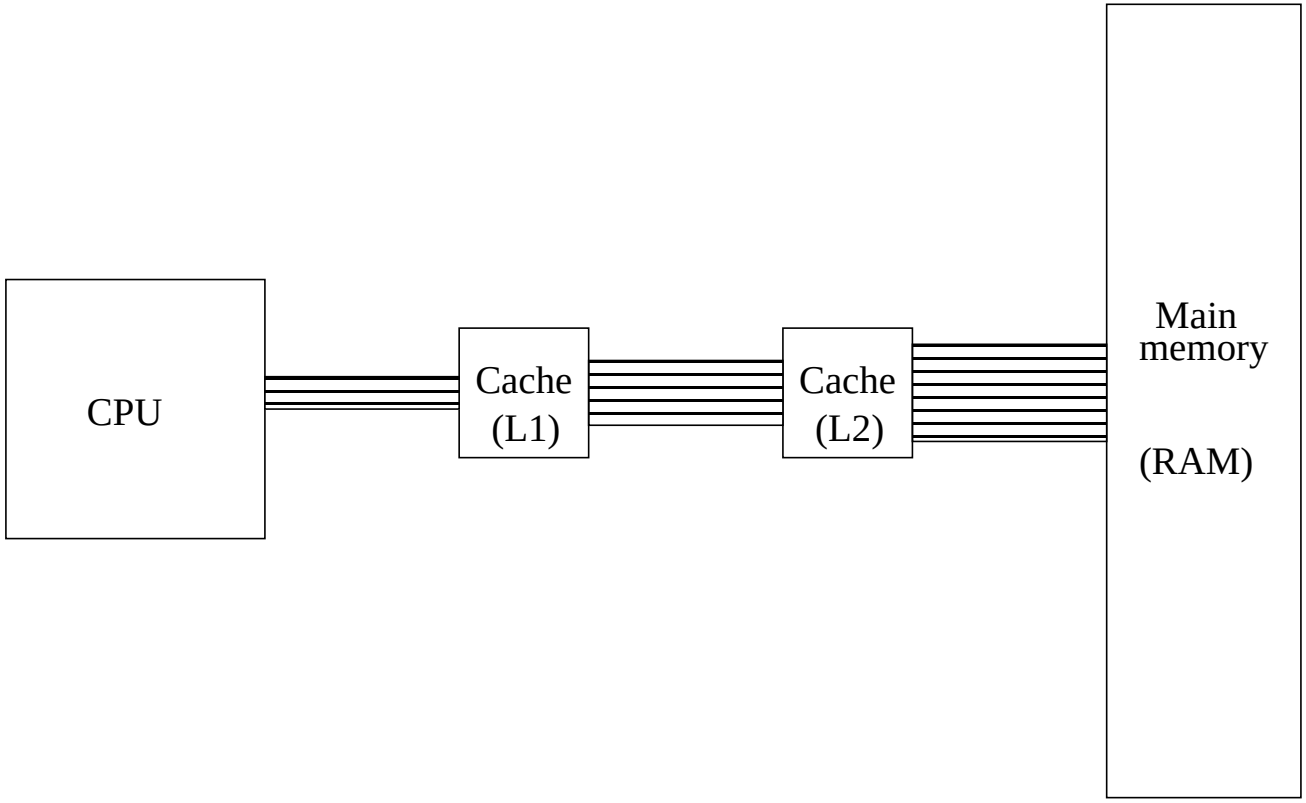
		Req. $x_1, \ldots, x_4$		Req. x_2		Req. x_5
		x_4		x_2		x_5
	$\Rightarrow$	x_3	$\Rightarrow$	x_4	$\Rightarrow$	x_2
		x_2		x_3		x_4
		x_1		x_1		x_3

Realistic cache

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



Multilevel caches



Intel Core2 (Q6600)			AMD Phenom II (945e)	
L1:	32kB	$k = 8$	$S = 64$kB	$k = 2$
L2:	4MB	$k = 16$	$S = 512$kB	$k = 8$
L3:	-	-	$S = 6$MB	$k = 48$

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 $k = L = 2$:

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

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 $k = L = 2$:

$$\frac{x_0}{} \Longrightarrow \frac{a_{00}}{x_0} \Longrightarrow \frac{y_0}{\frac{a_{00}}{x_0}} \Longrightarrow$$

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 $k = L = 2$:

$$\frac{x_0}{} \Longrightarrow \frac{a_{00}}{x_0} \Longrightarrow \frac{y_0}{a_{00}} \Longrightarrow \frac{x_1}{y_0} \Longrightarrow \frac{}{a_{00}} \Longrightarrow \frac{}{x_0}$$

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

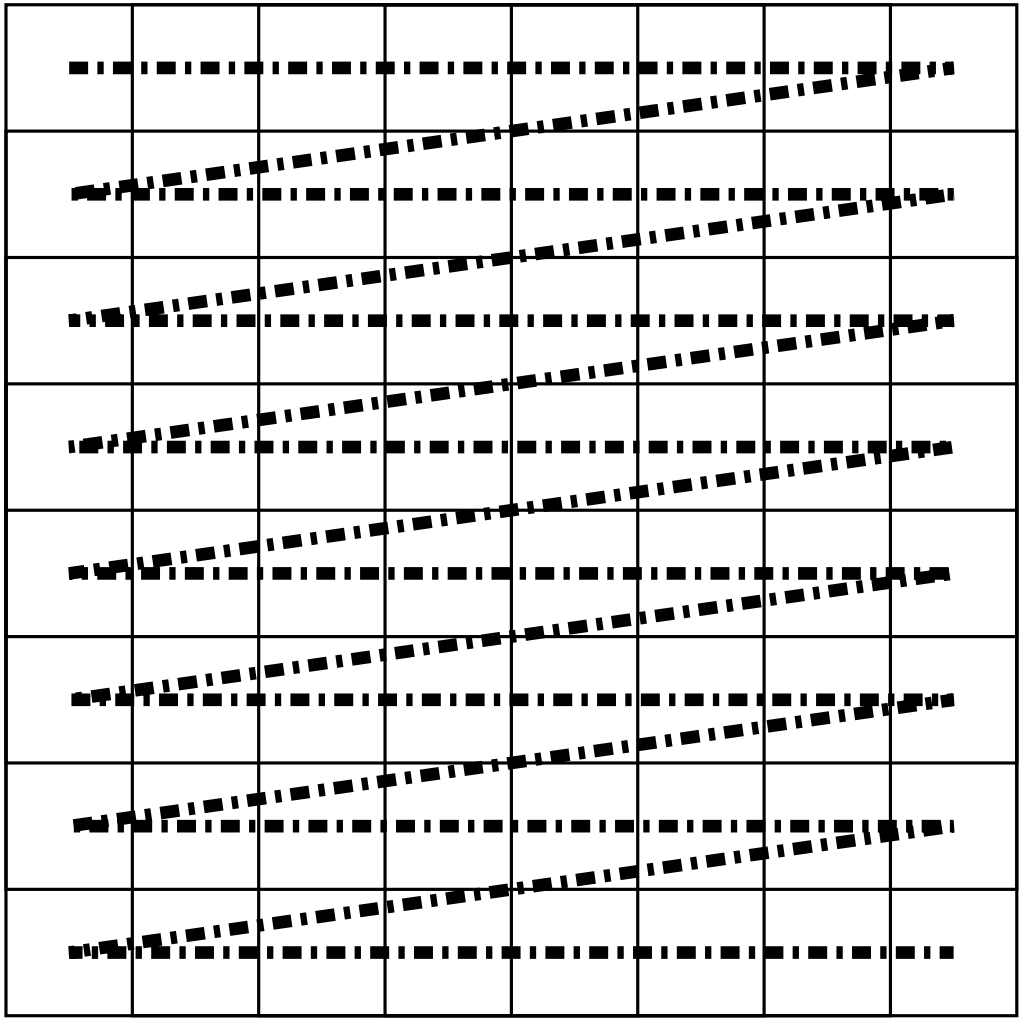
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

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

The sparse case

Standard datastructure: Compressed Row Storage (CRS)



The sparse case

Sparse matrix–vector multiplication (SpMV)

x?

$\Longrightarrow$

The sparse case

Sparse matrix–vector multiplication (SpMV)

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

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)

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

The sparse case

Sparse matrix–vector multiplication (SpMV)

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

We cannot predict memory accesses in the sparse case!

Other storage schemes

Starting point, plain 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, a high-performance computation library* (Master's Thesis), 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:

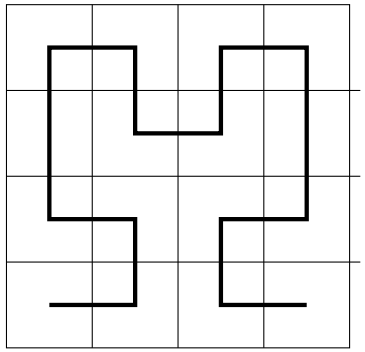
nzs: [4 1 3 2 3 1 2 7 0 1 1]
blk: [0 3 5 6 7 11]
col: [0 2 0 3 0]
row: [0 1 2 4 5]

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

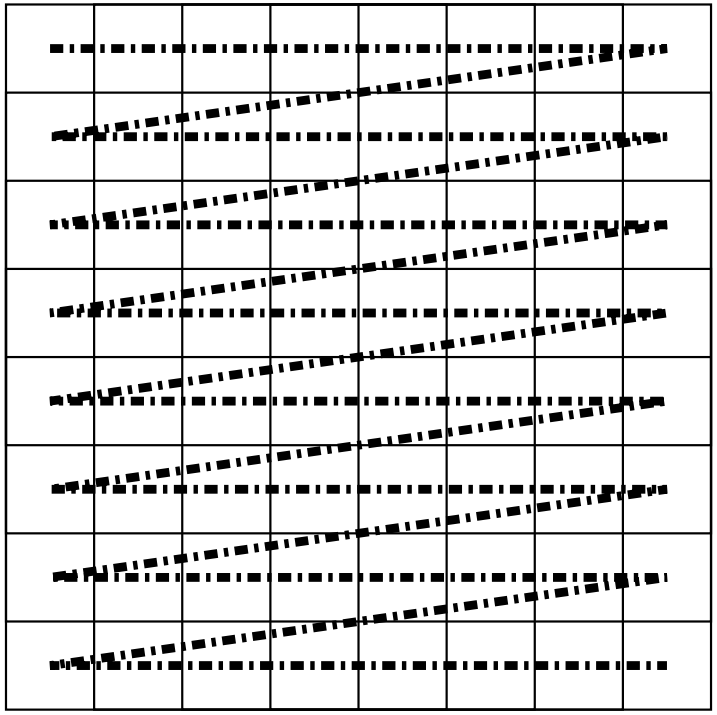
Reference: Haase, Liebmann and Plank, *A Hilbert-Order Multiplication Scheme for Unstructured Sparse Matrices*, 2005

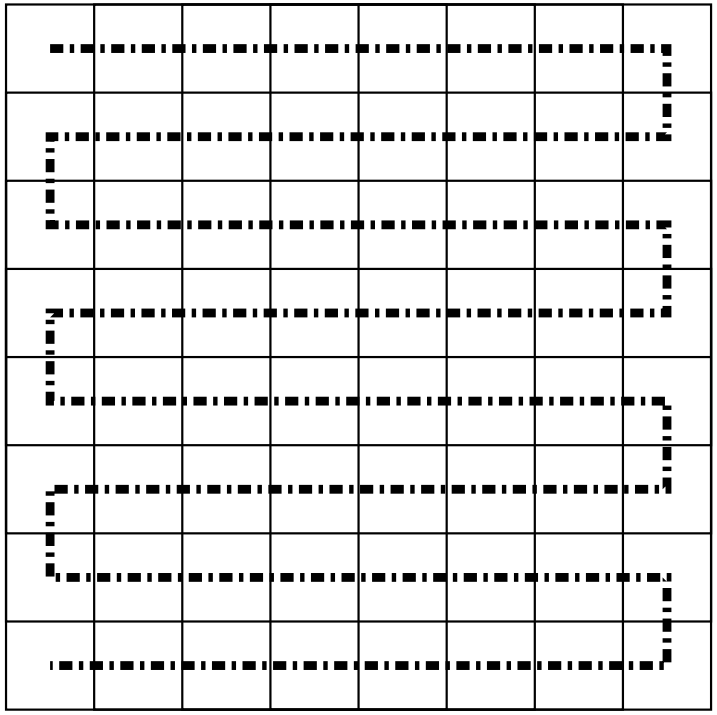
Improving cache–use

We adapt two things;

- **sparse matrix storage**
- sparse matrix structure

CRS to "Zig-zag" 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]

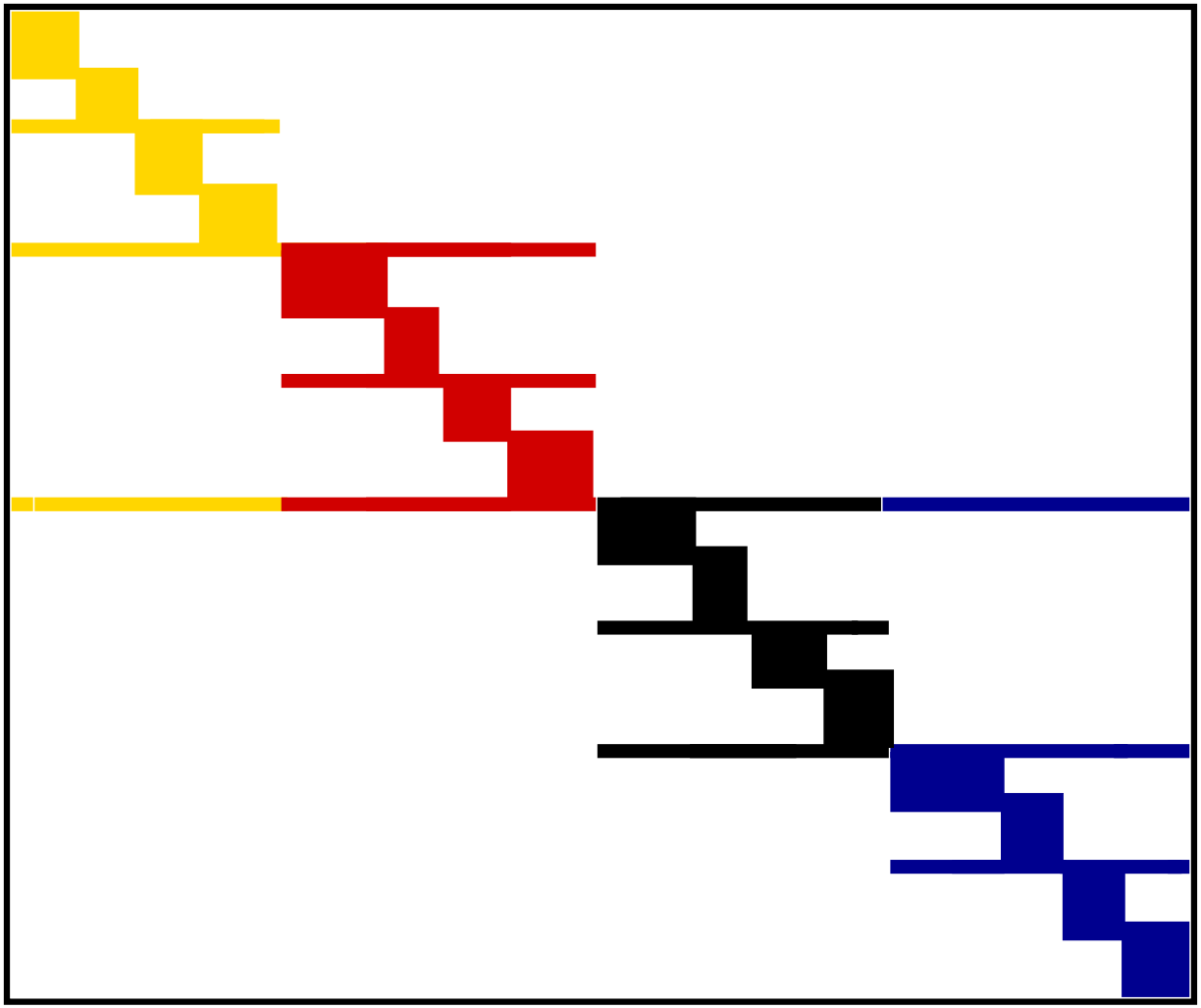
Reference: Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SISC, 2009

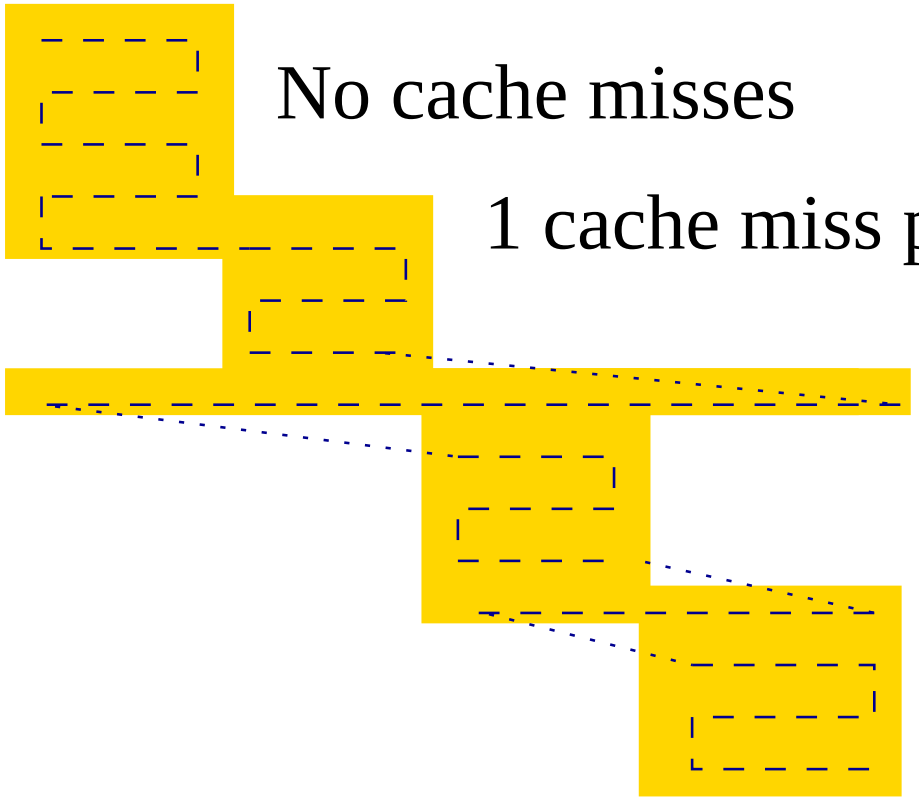
Improving cache–use

We adapt two things;

- sparse matrix storage
- **sparse matrix structure**

Now assume the sparse matrix is in "Separated Block Diagonal" (SBD) form:



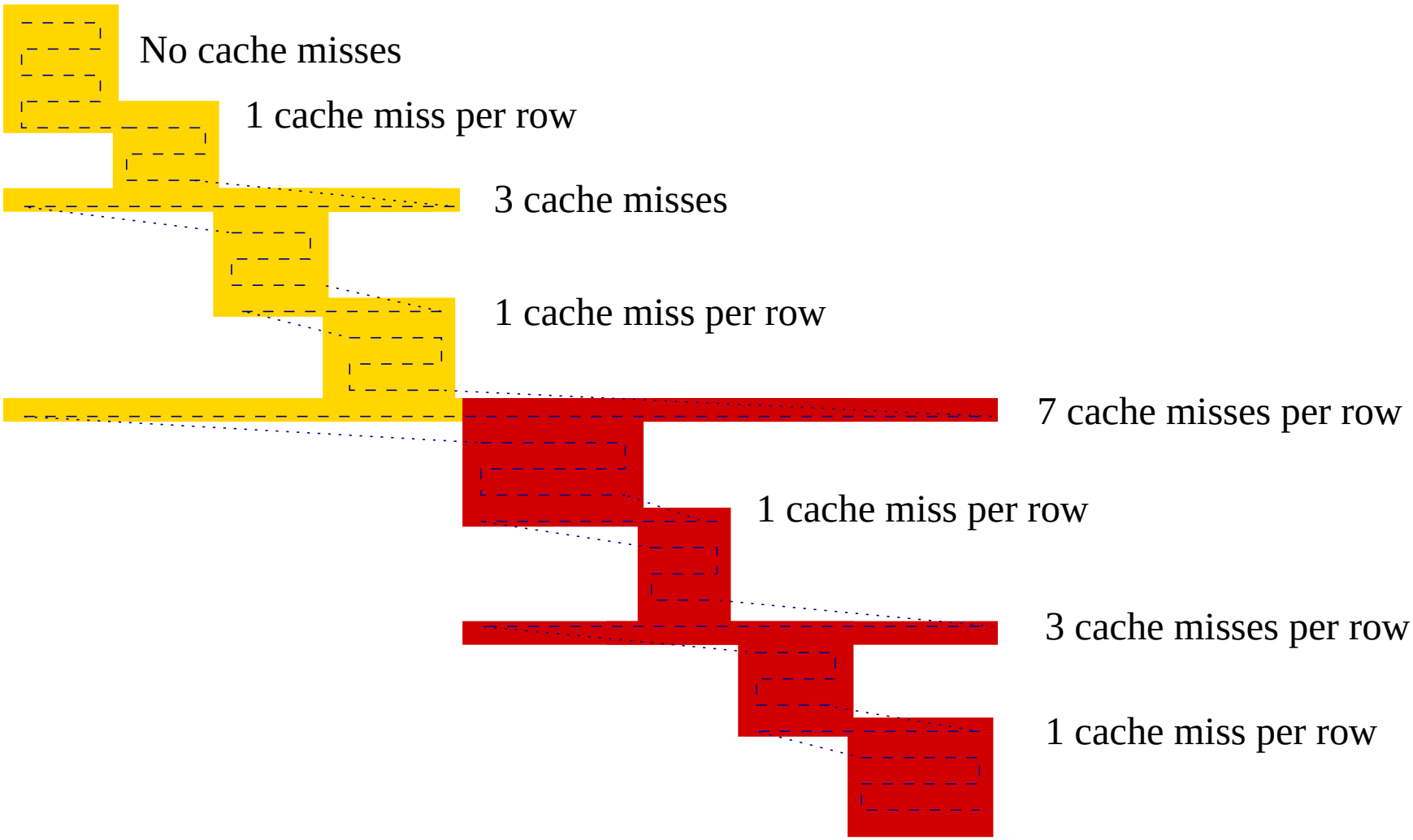


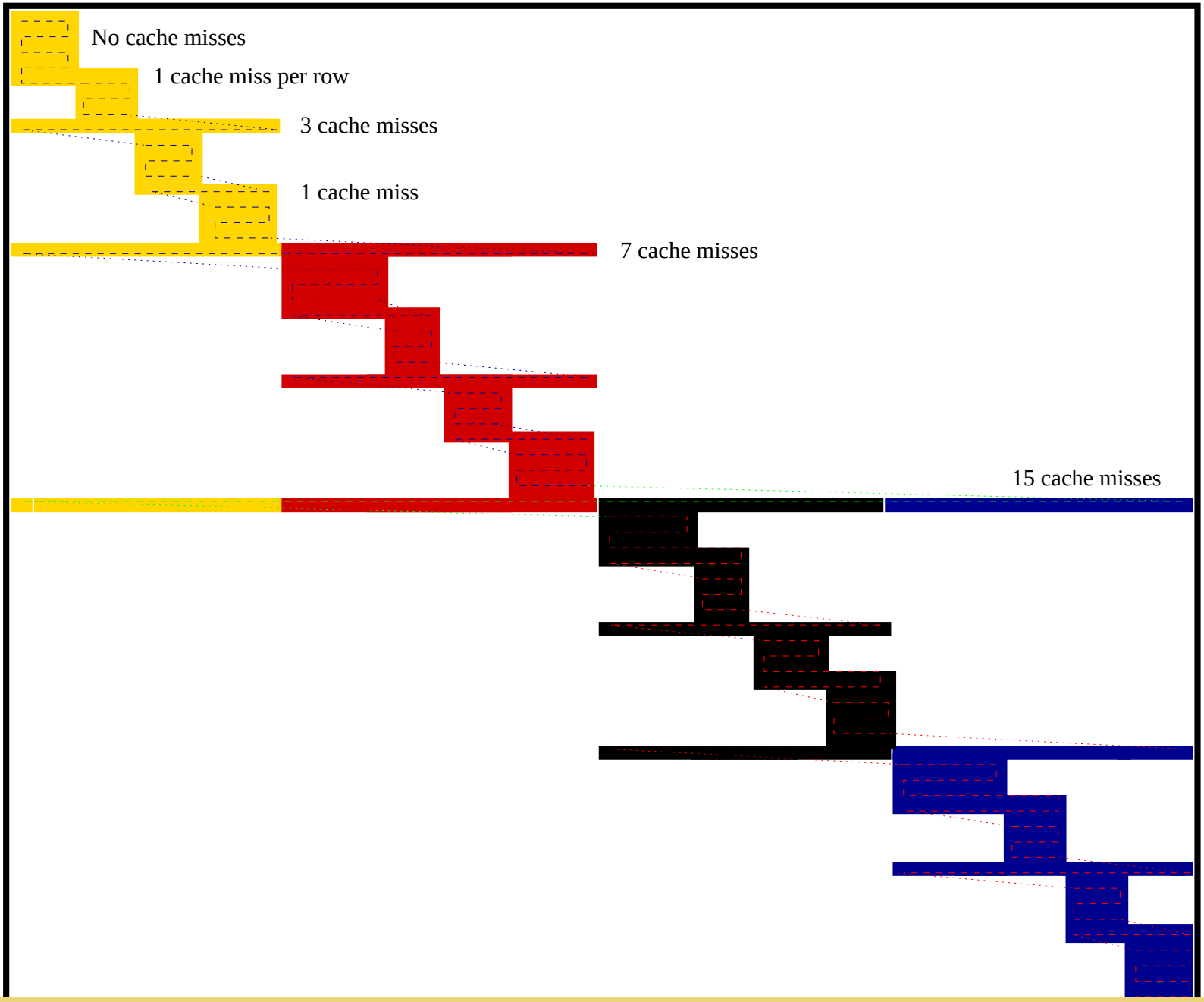
No cache misses

1 cache miss per row

3 cache misses per row

1 cache miss per row


No cache misses
1 cache miss per row
3 cache misses
1 cache miss per row
7 cache misses per row
1 cache miss per row
3 cache misses per row
1 cache miss per row


No cache misses
1 cache miss per row
3 cache misses
1 cache miss
7 cache misses
15 cache misses

Summarising...

If

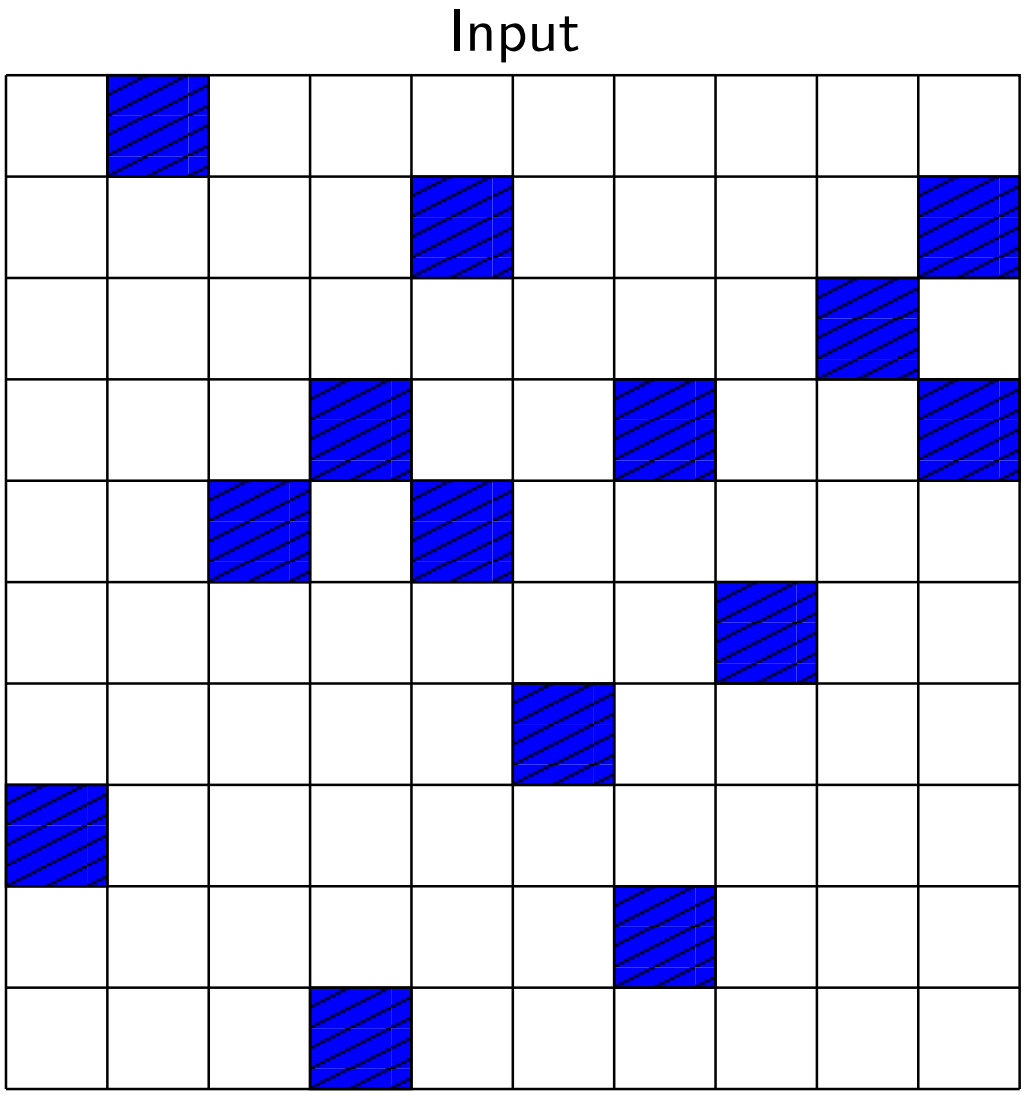
1. the matrix is stored in Zig-zag CRS format,
2. each SBD block corresponds to one vertex,
3. each *row* corresponds to one net (containing a vertex if the corresponding submatrix contains a non-zero);

then, the number of cache-misses is given by:

$$\sum_i (\lambda_i - 1).$$

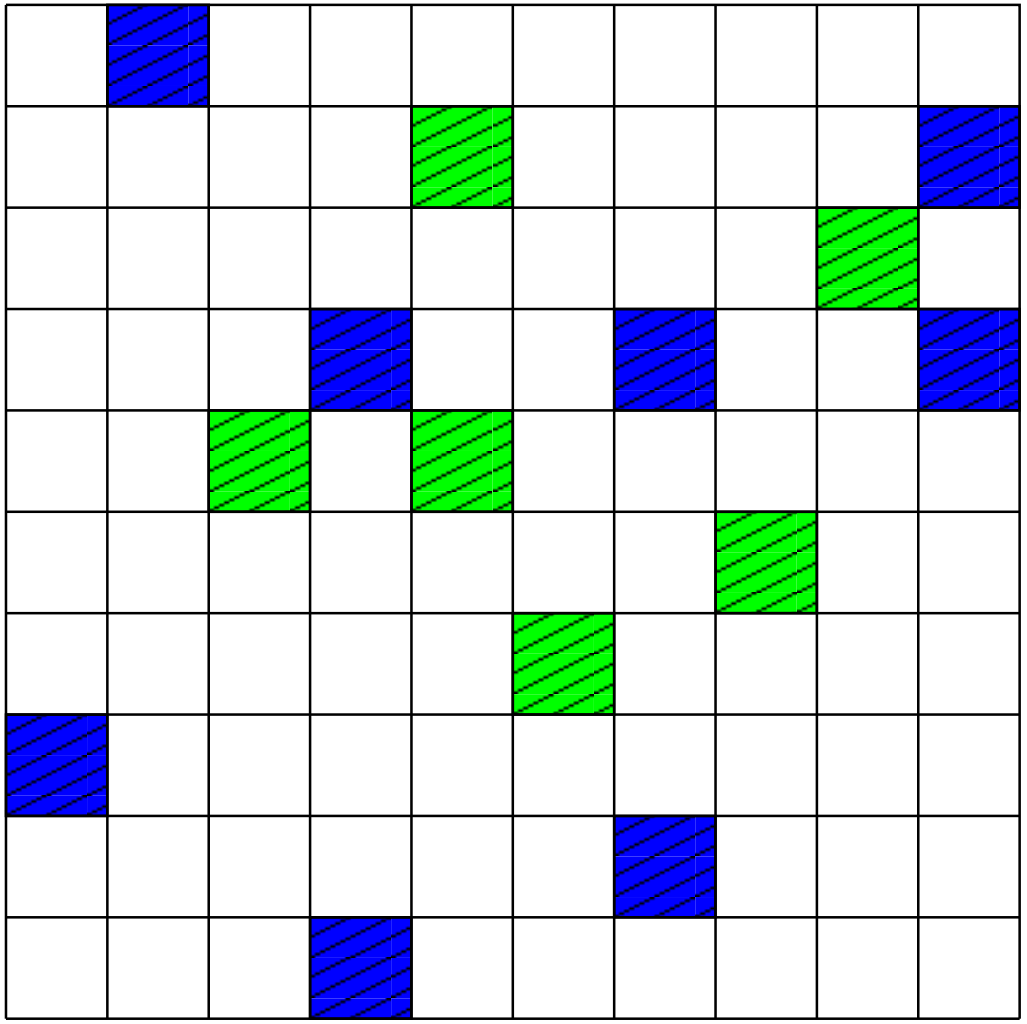
λ_i being the connectivity of the ith row

Using Mondriaan

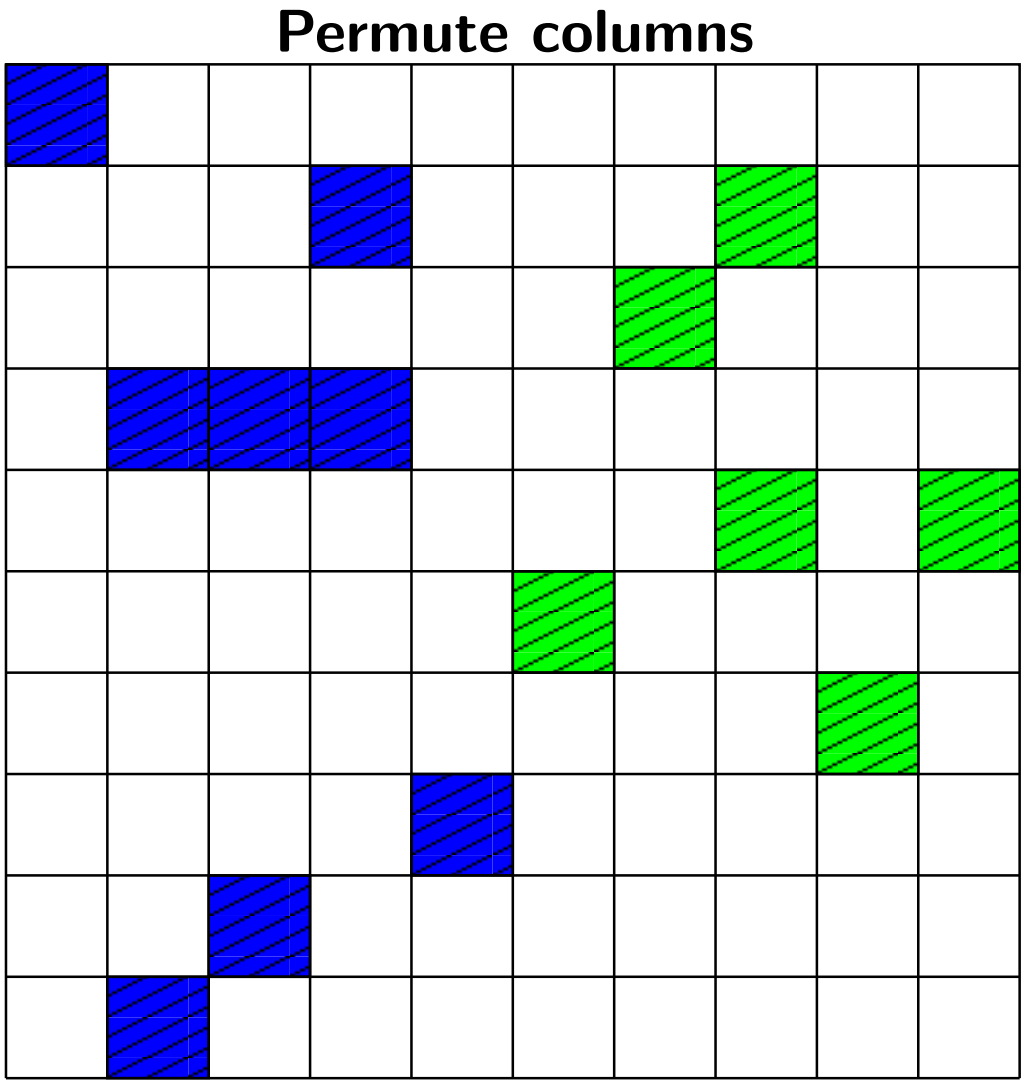


Using Mondriaan

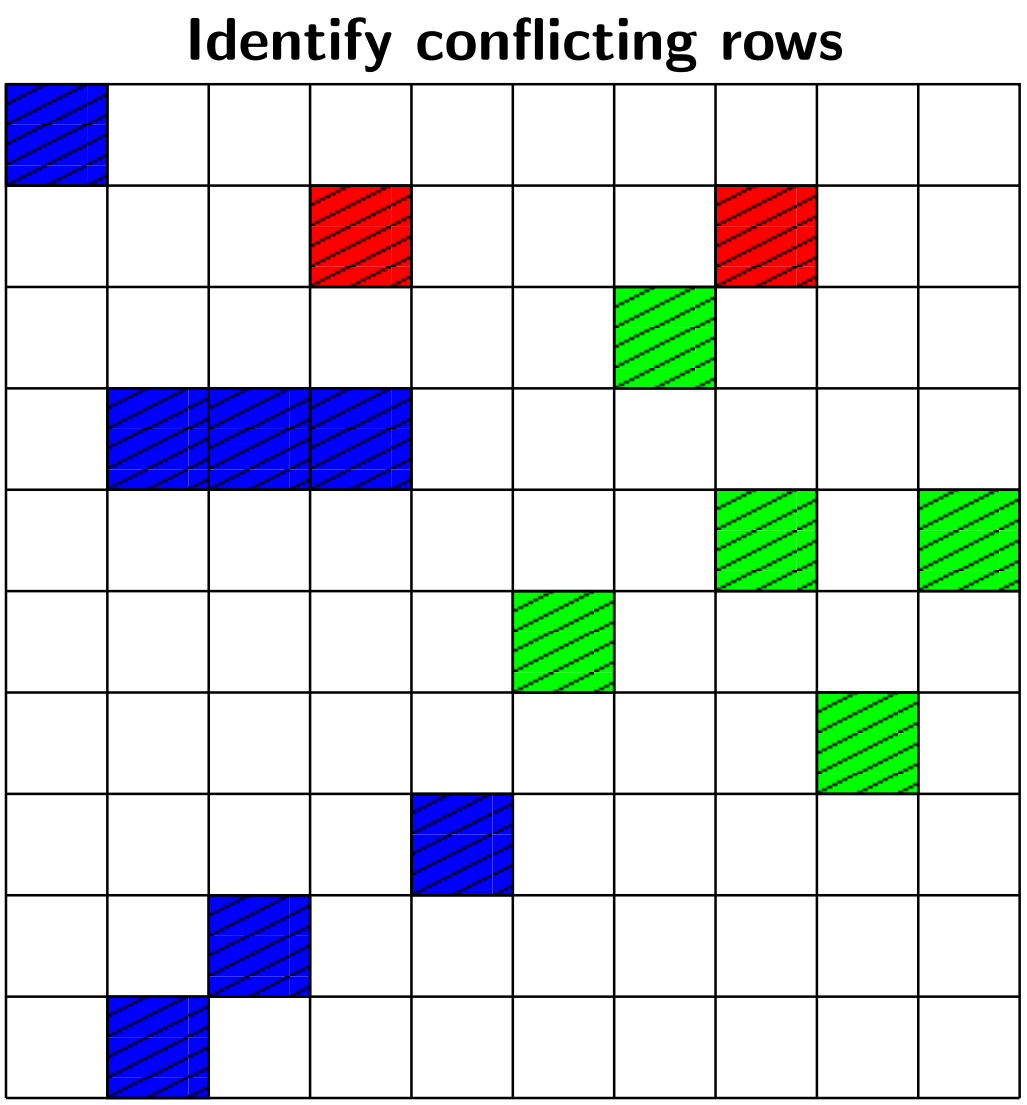
Column partitioning (force row-net model)



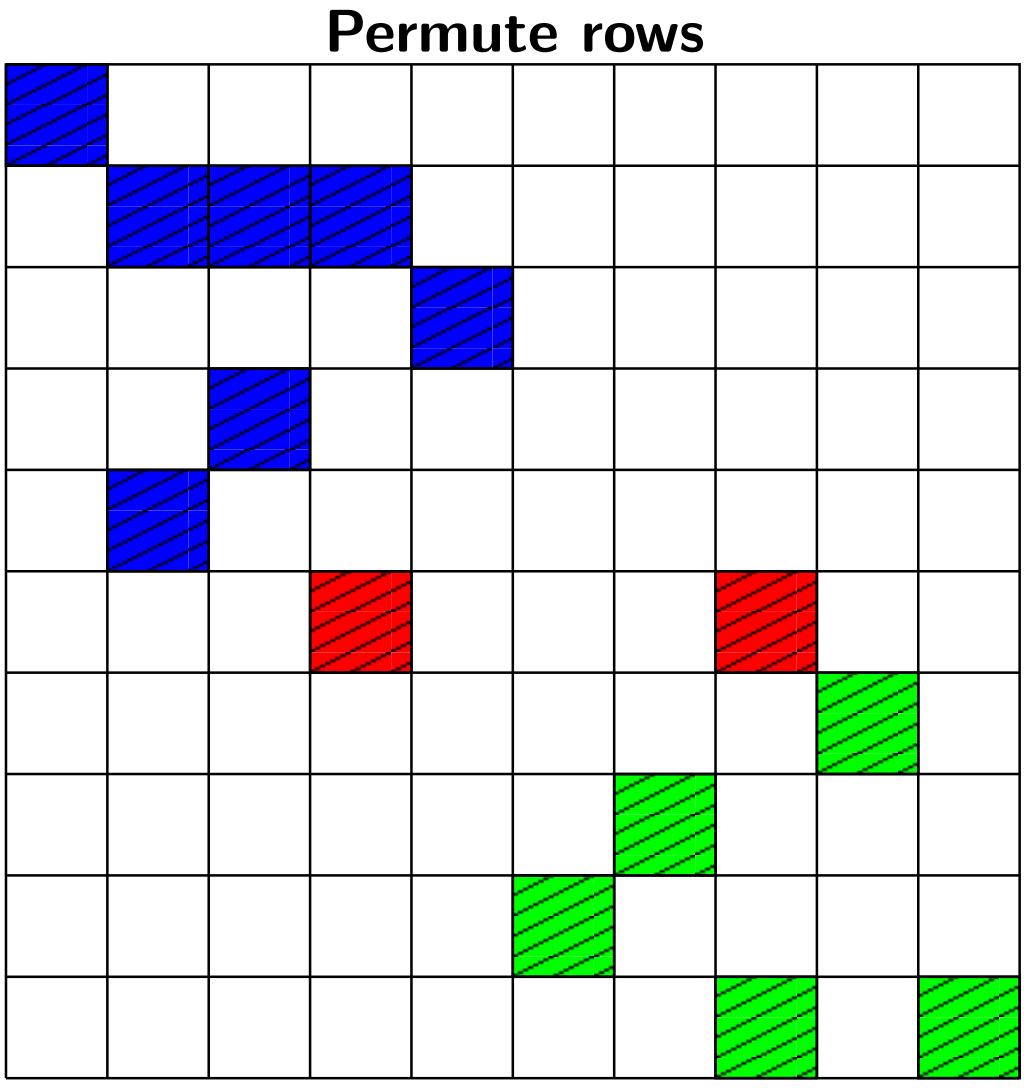
Using Mondriaan



Using Mondriaan

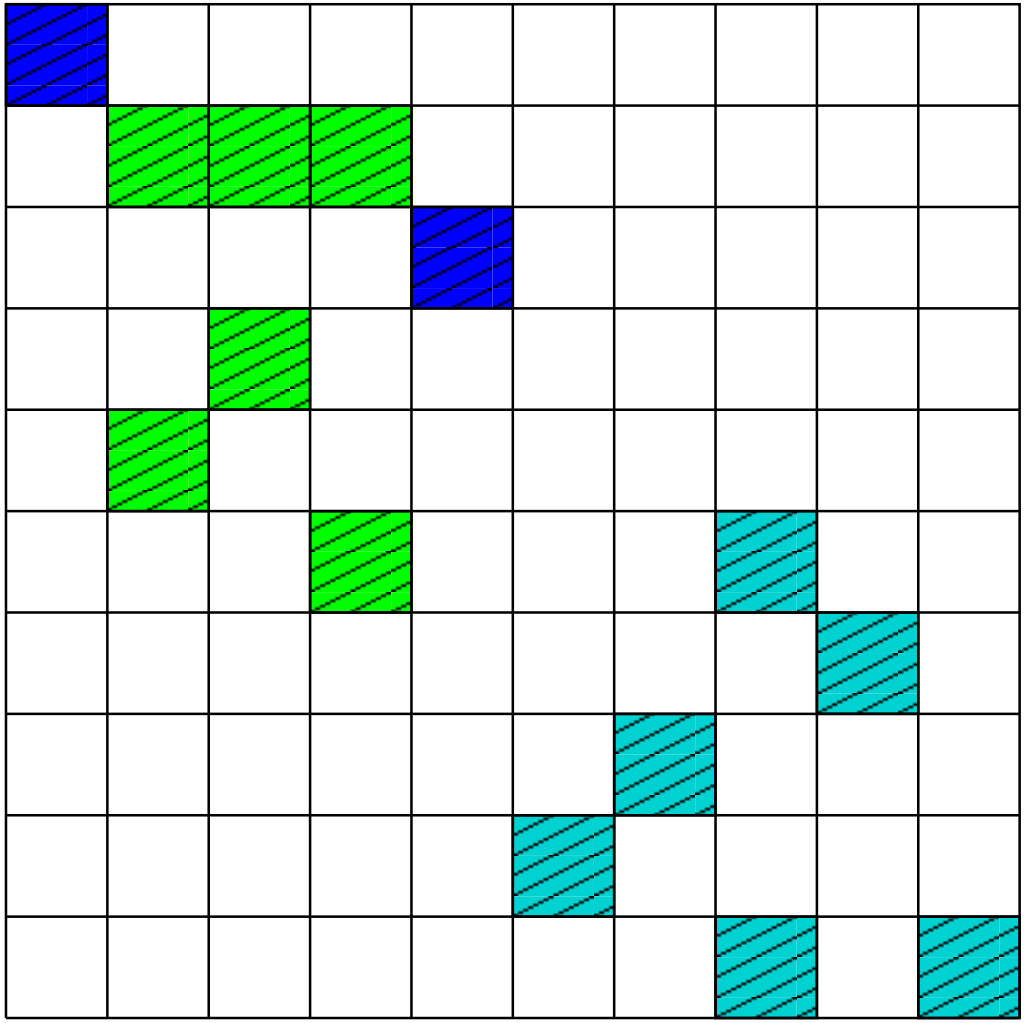


Using Mondriaan



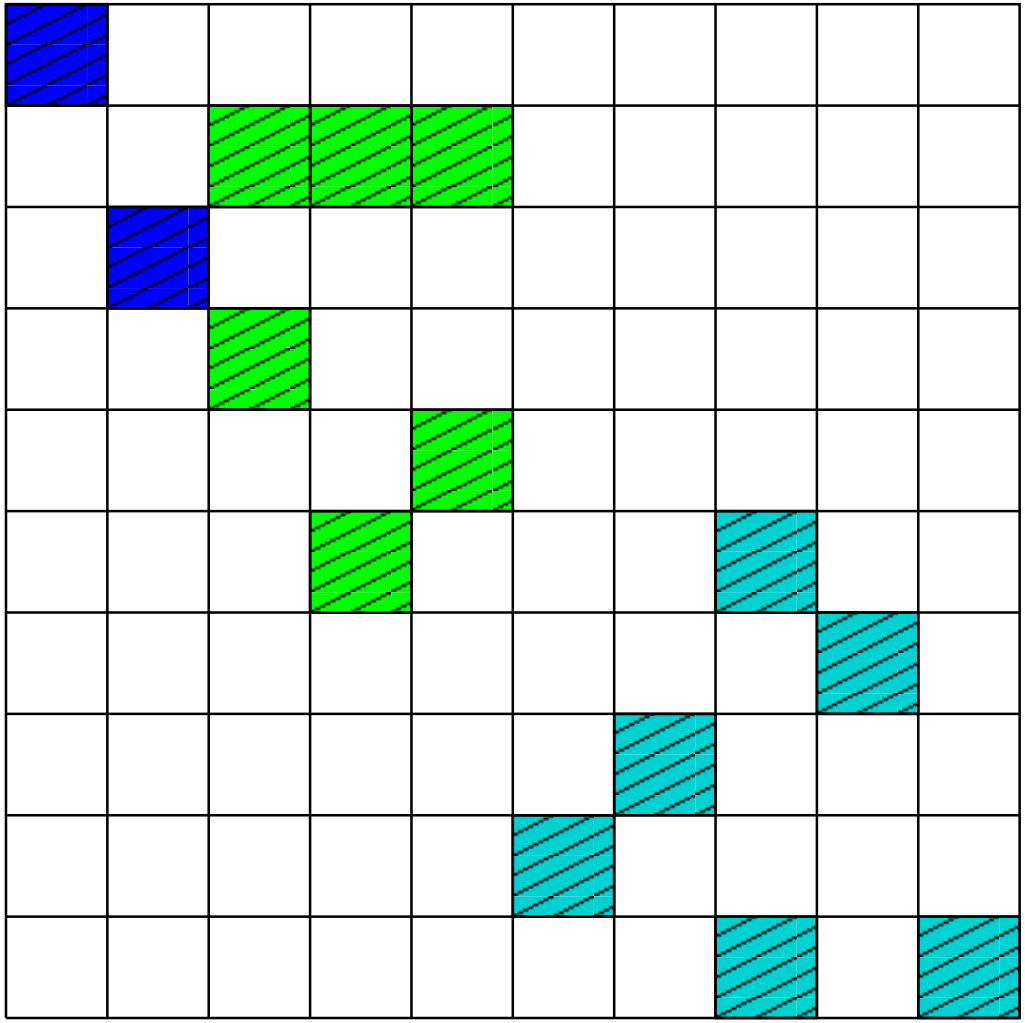
Permuting to SBD form

Column subpartitioning



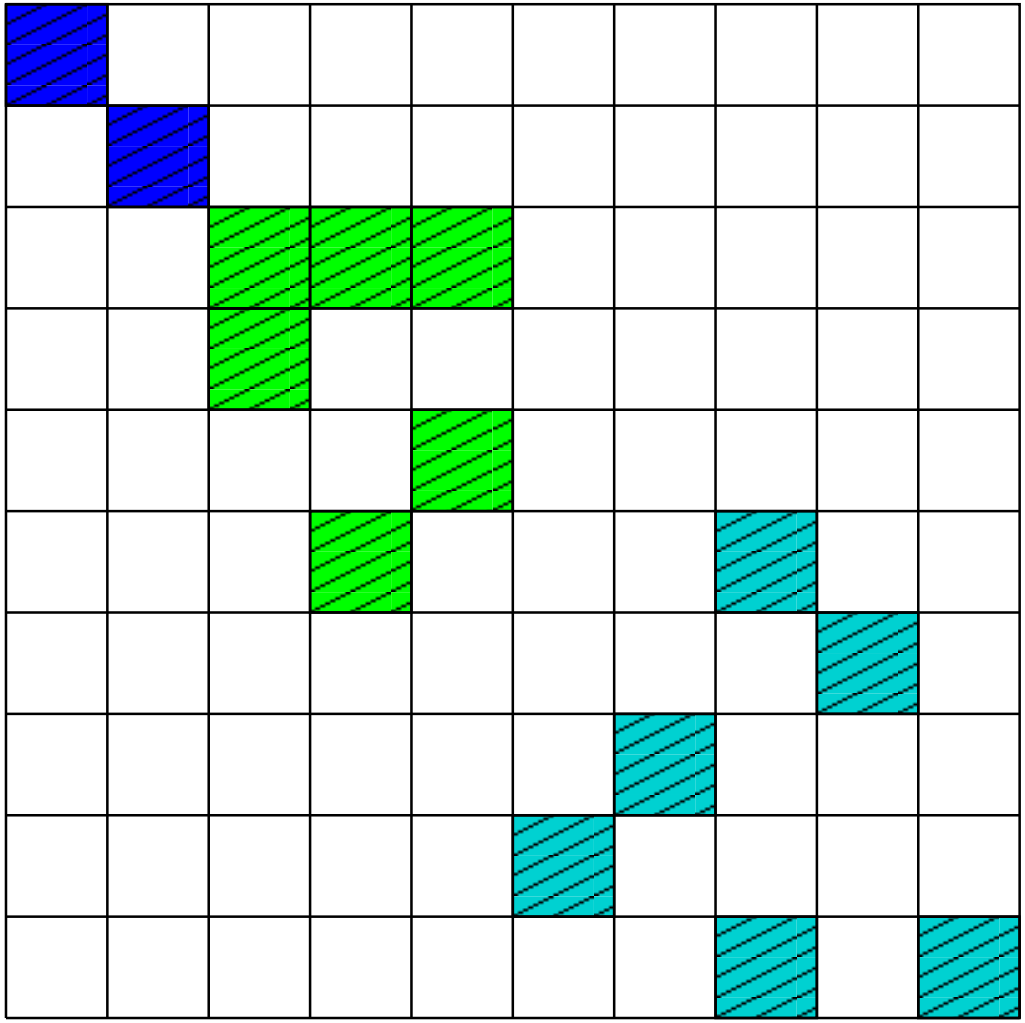
Permuting to SBD form

Column permutation



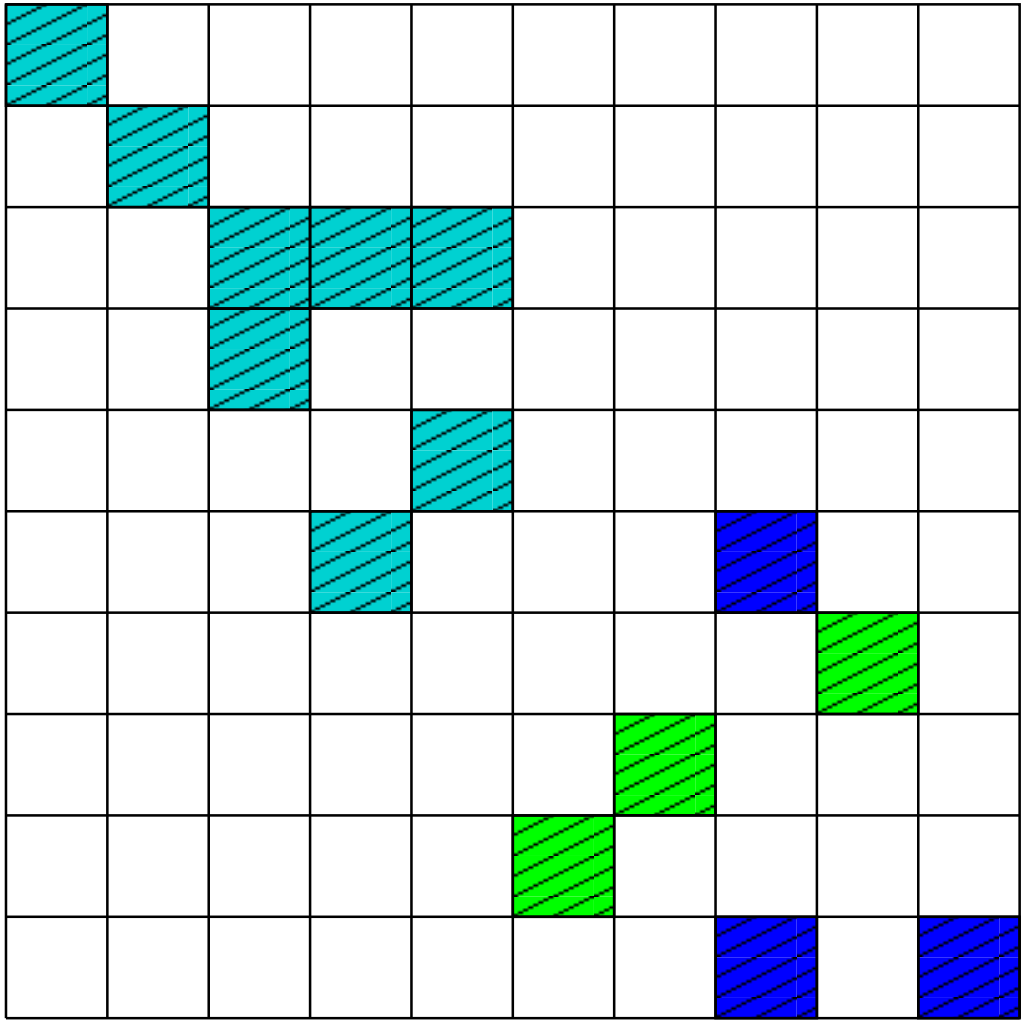
Permuting to SBD form

No mixed rows - row permutation



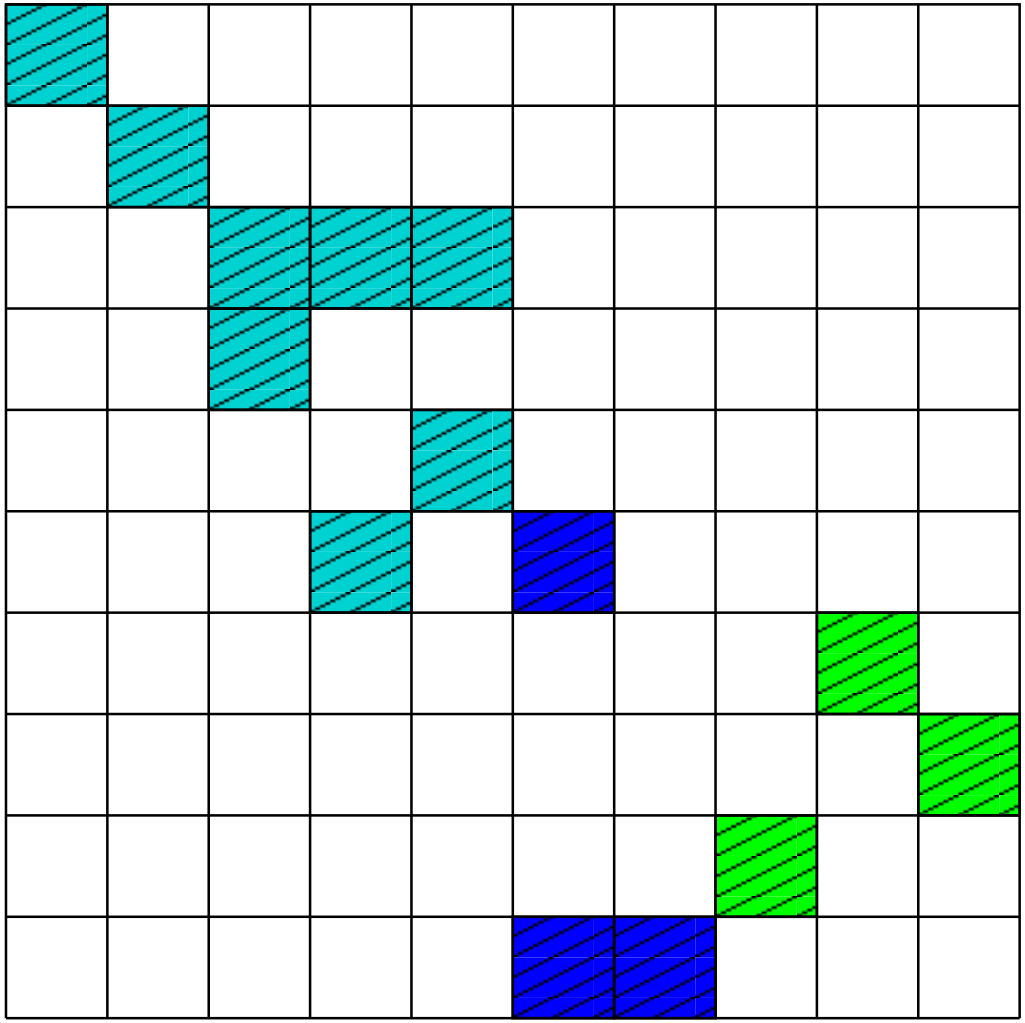
Permuting to SBD form

Continued



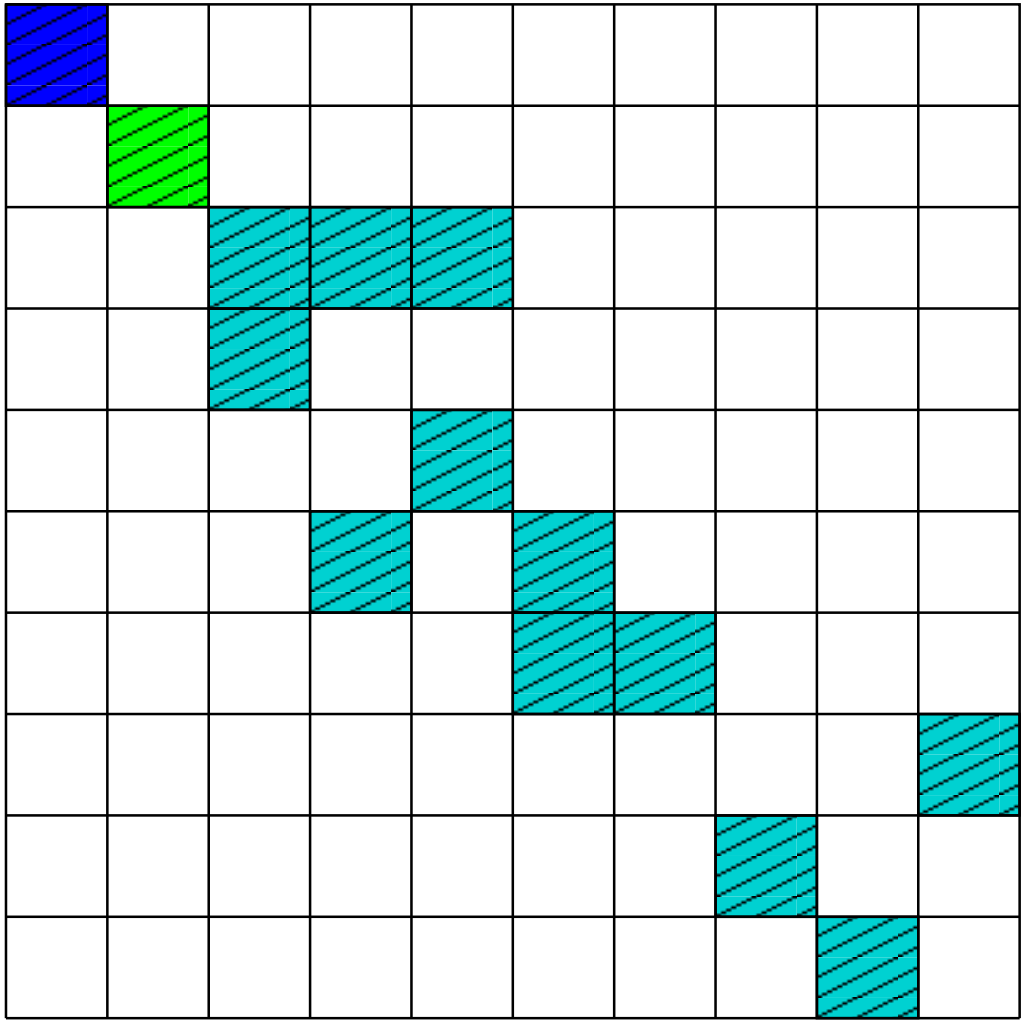
Permuting to SBD form

Continued



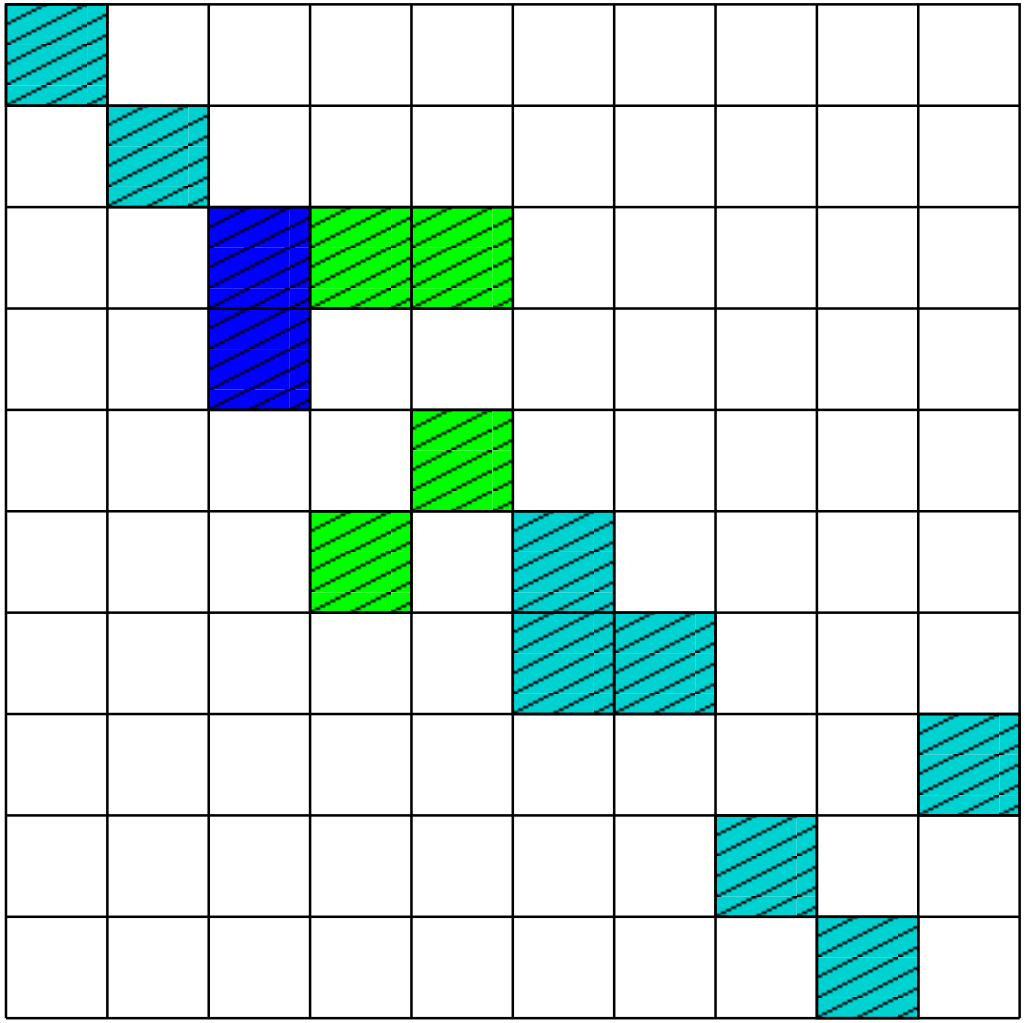
Permuting to SBD form

Continued



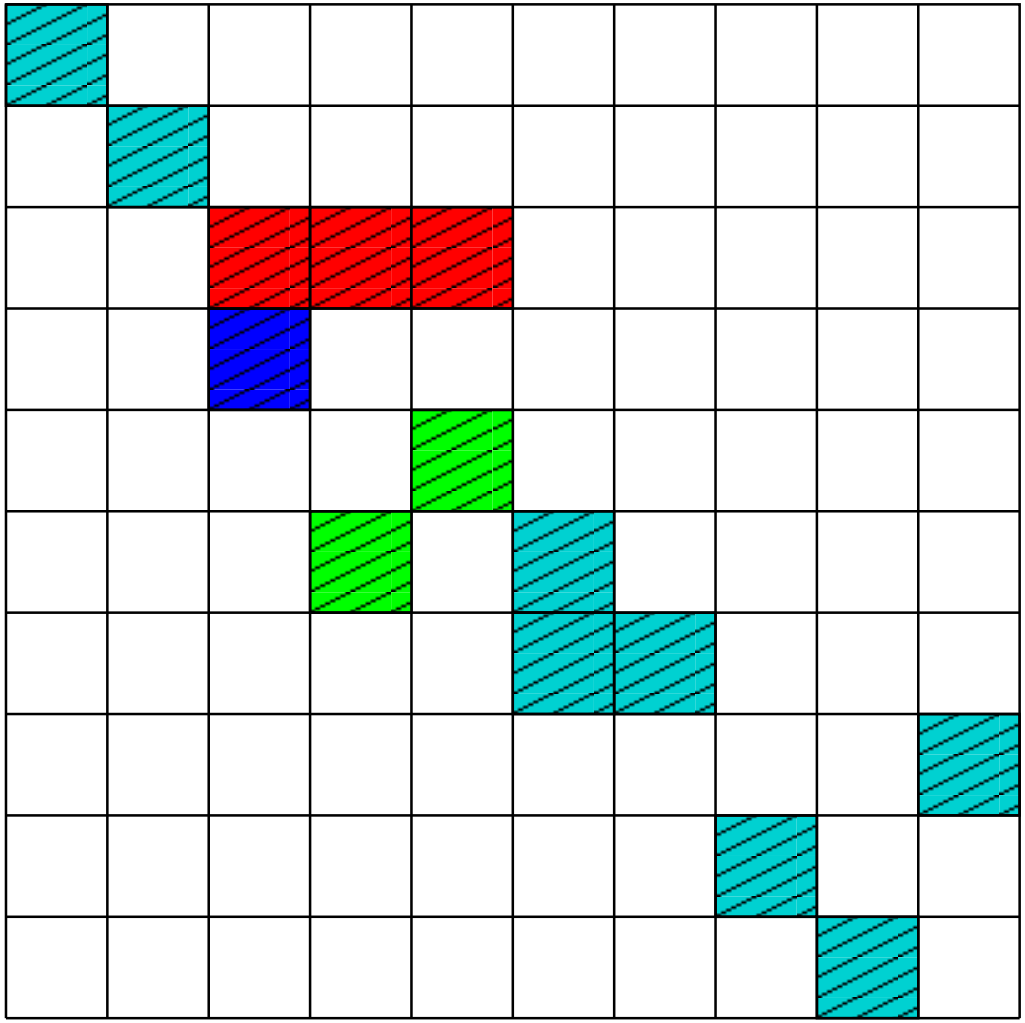
Permuting to SBD form

Continued



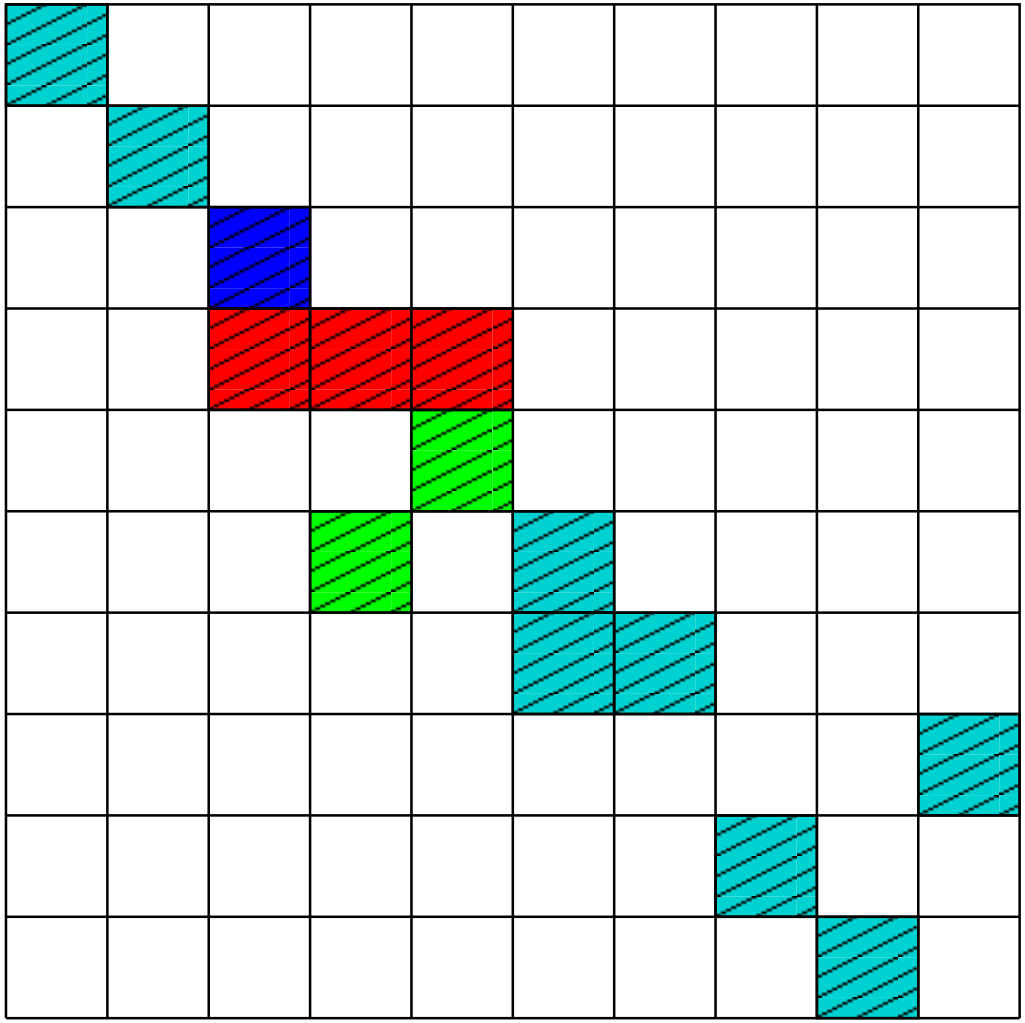
Permuting to SBD form

Continued



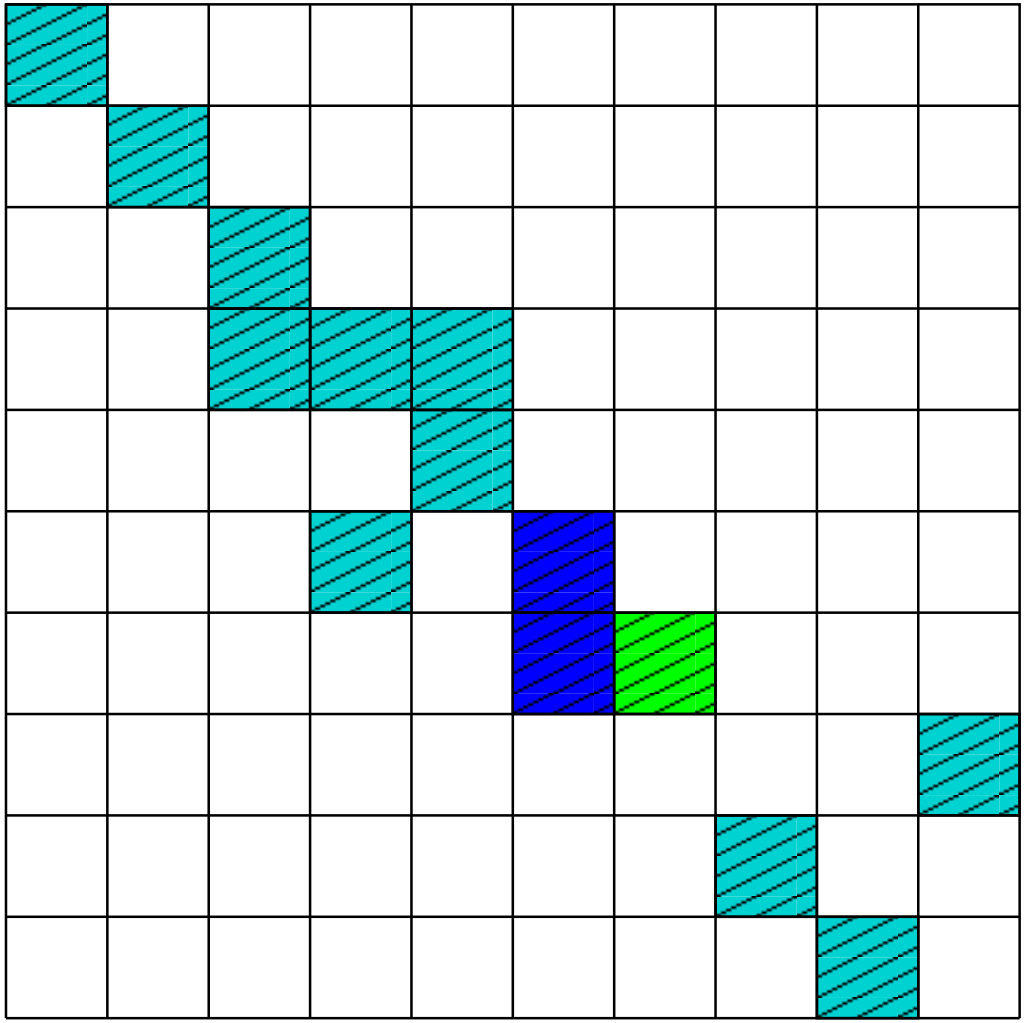
Permuting to SBD form

Continued



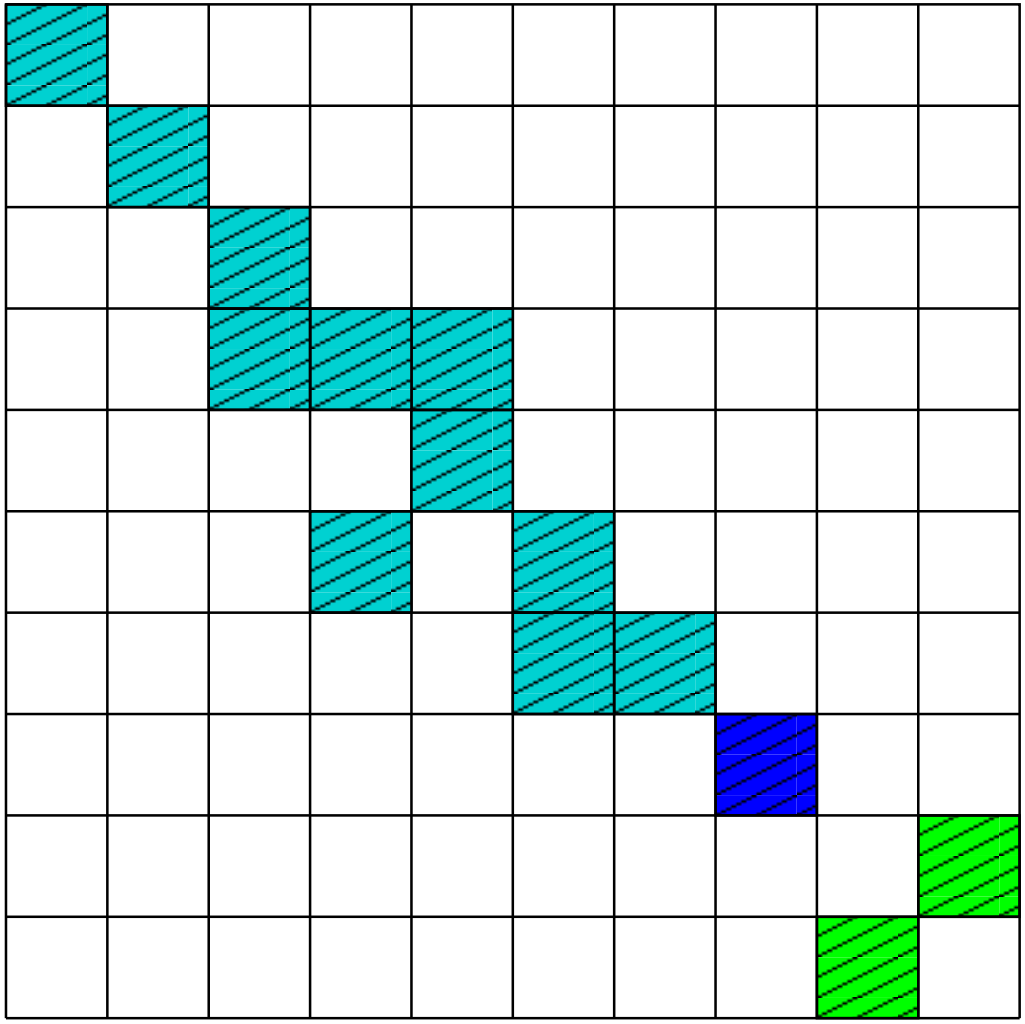
Permuting to SBD form

Continued



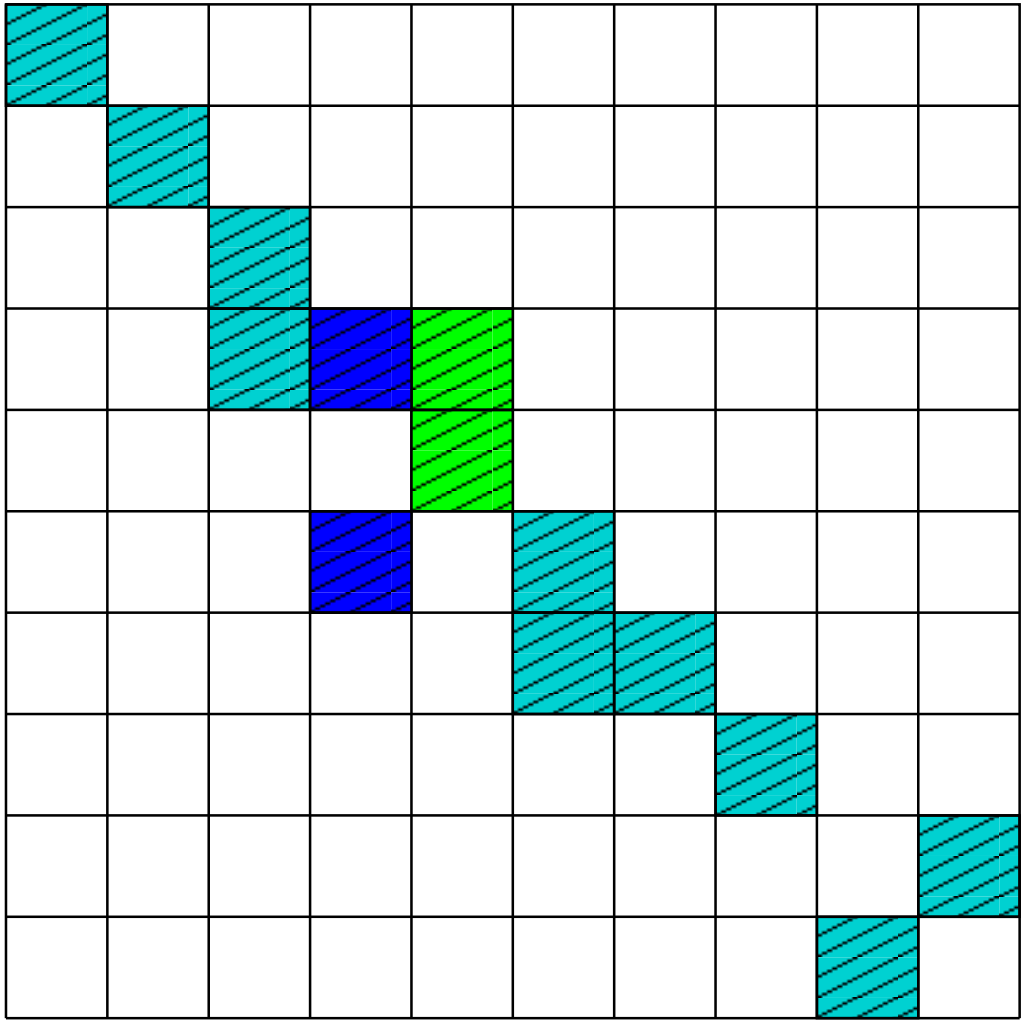
Permuting to SBD form

Continued



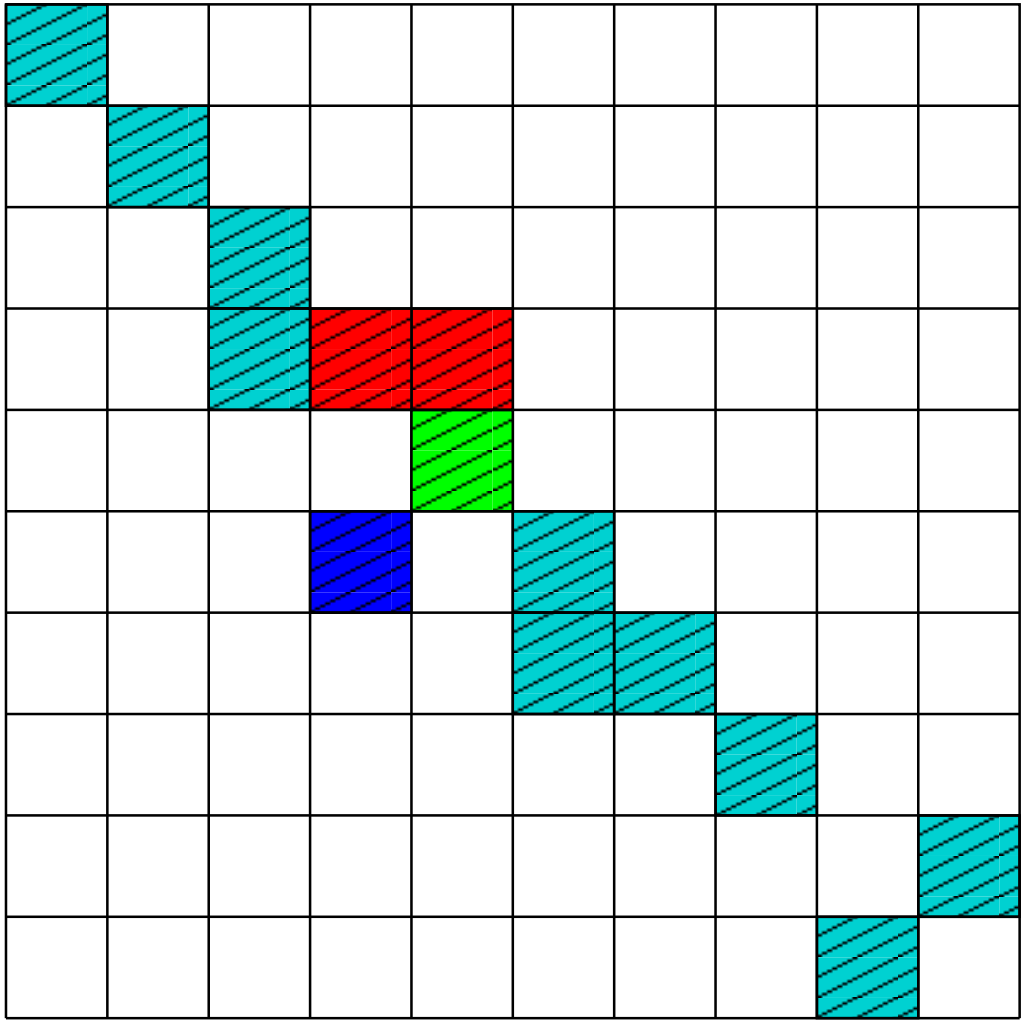
Permuting to SBD form

Continued



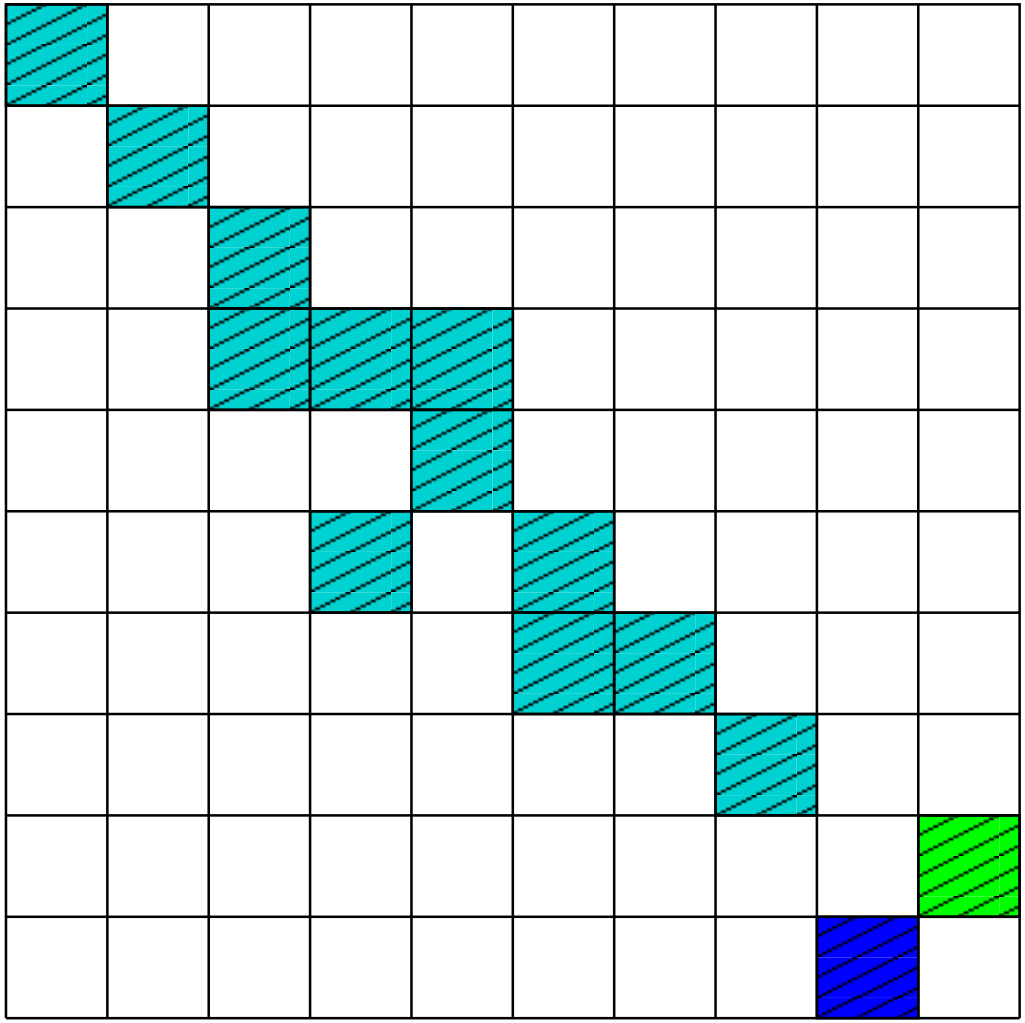
Permuting to SBD form

Continued



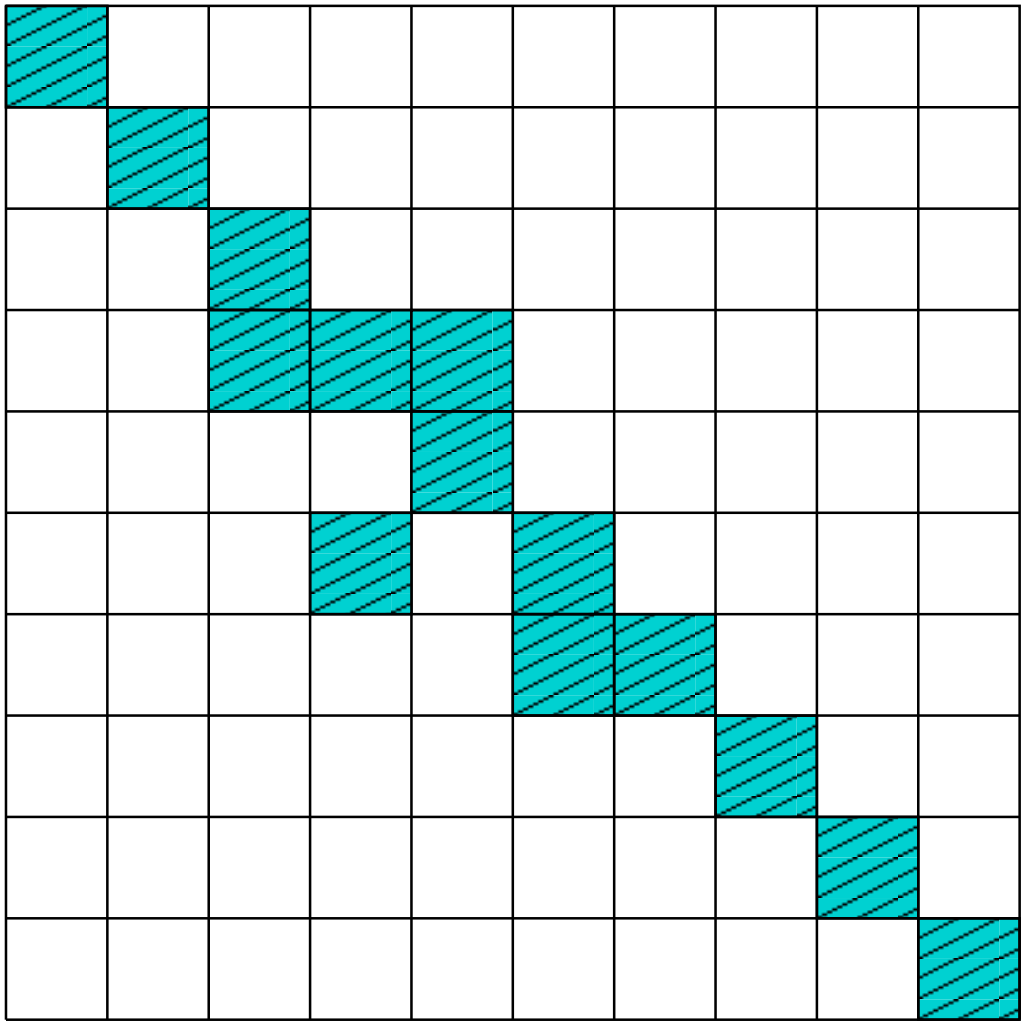
Permuting to SBD form

Continued

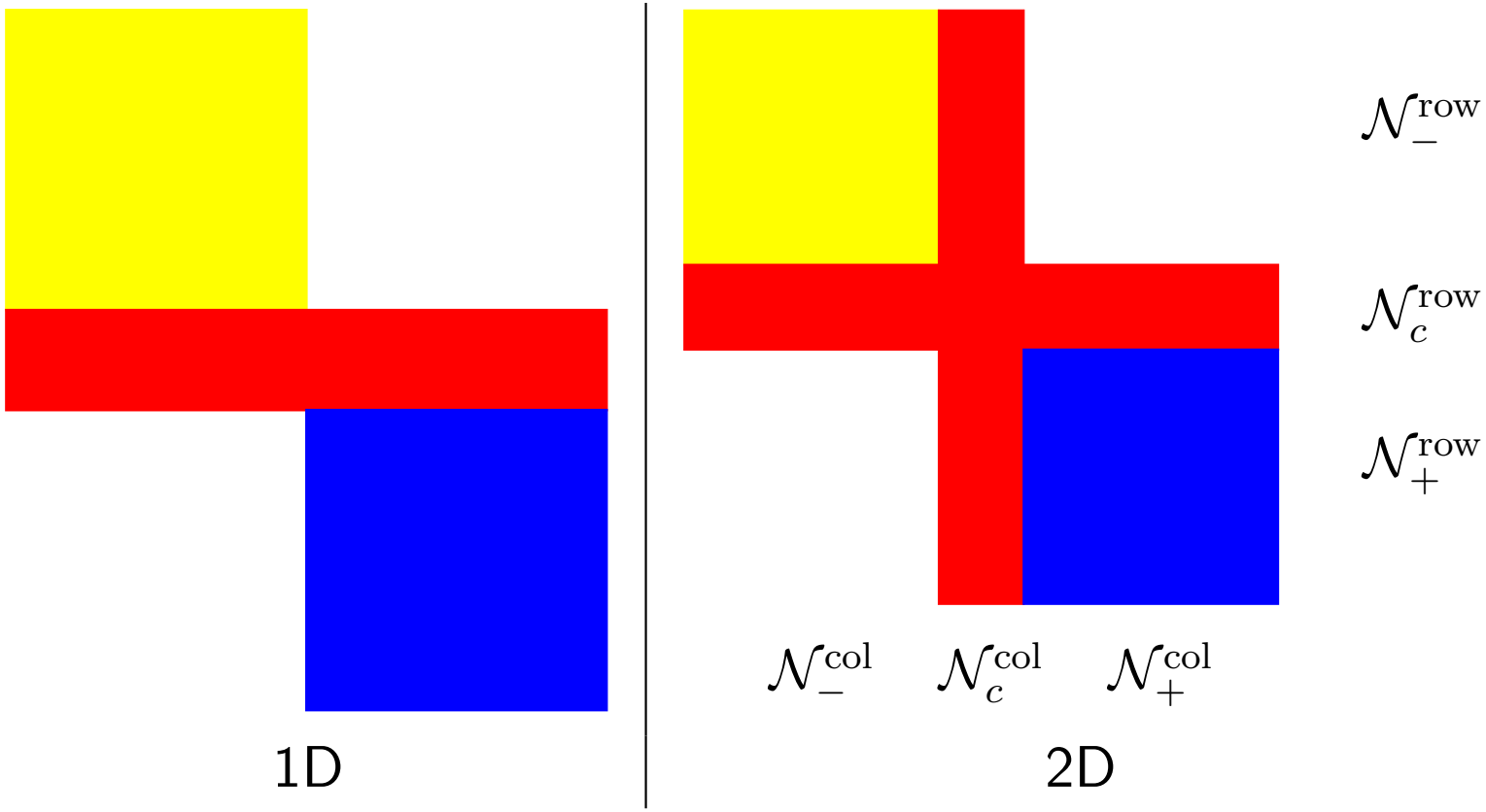


Permuting to SBD form

...and recurse (until n partitions reached)

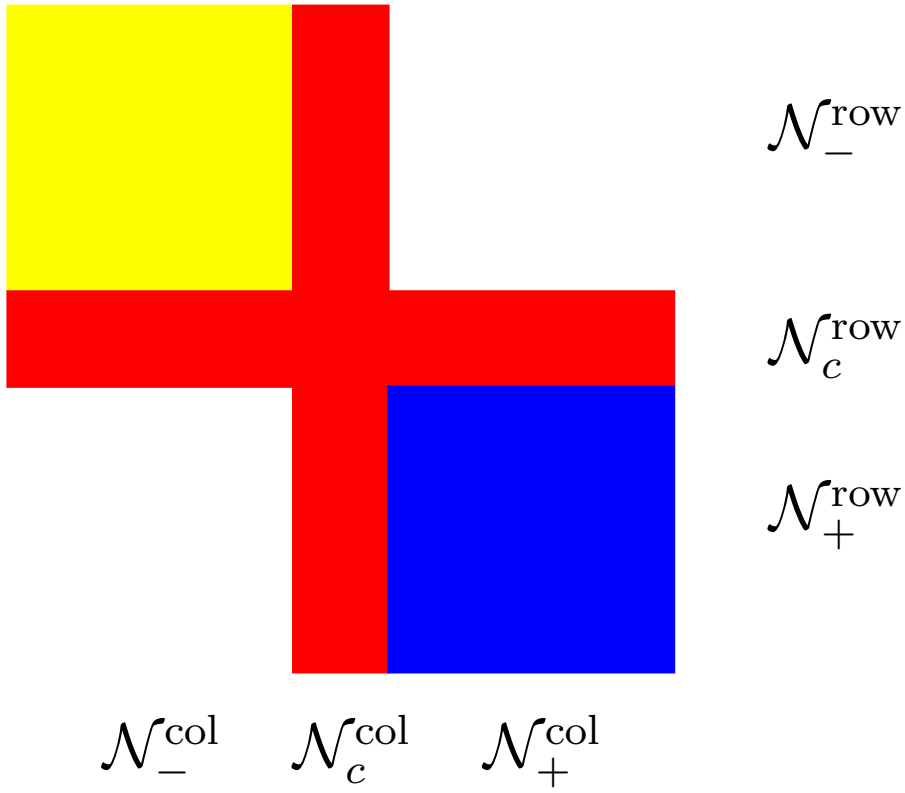


Two-dimensional SBD

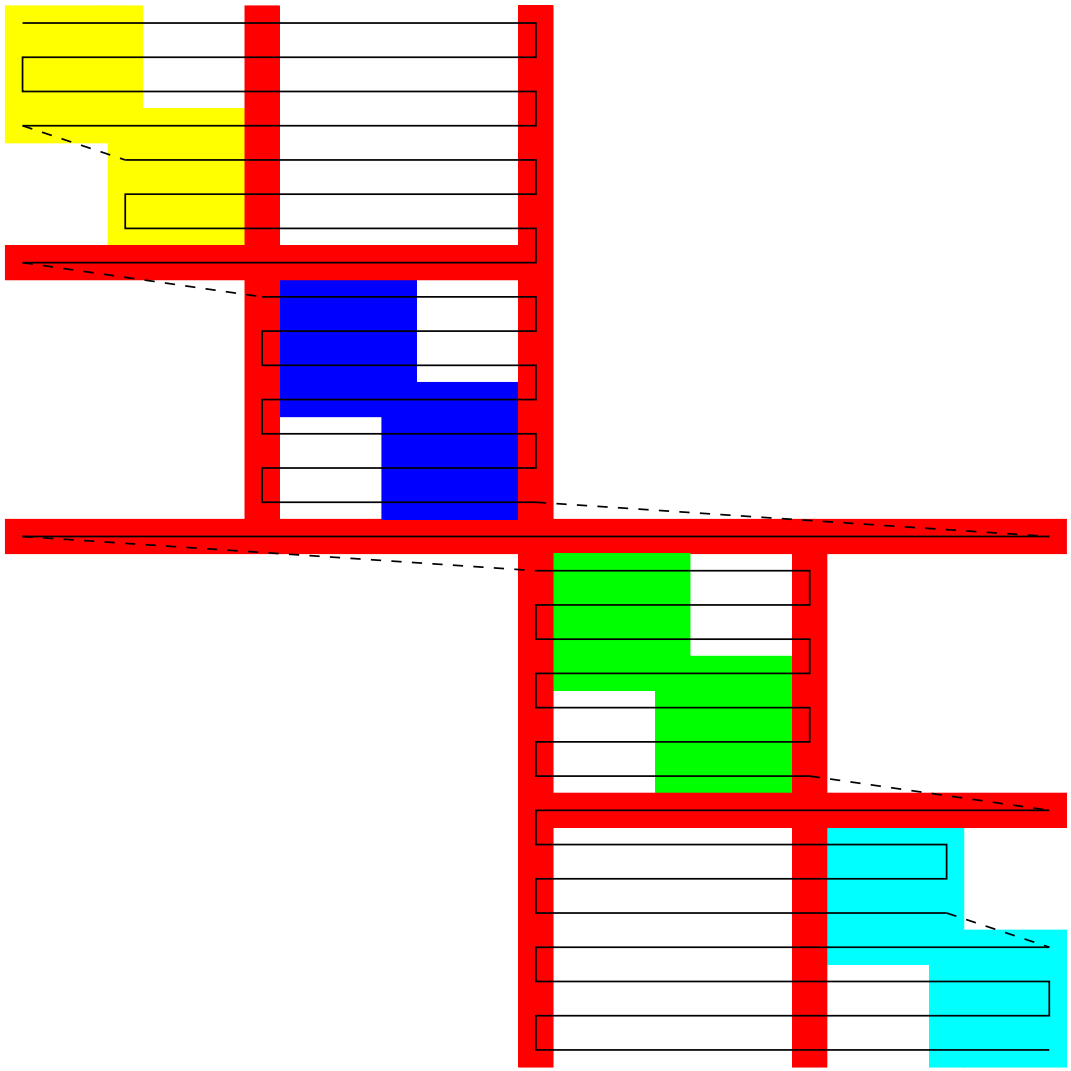


Two-dimensional SBD

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 to minimise remains $\sum_i (\lambda_i - 1)$.

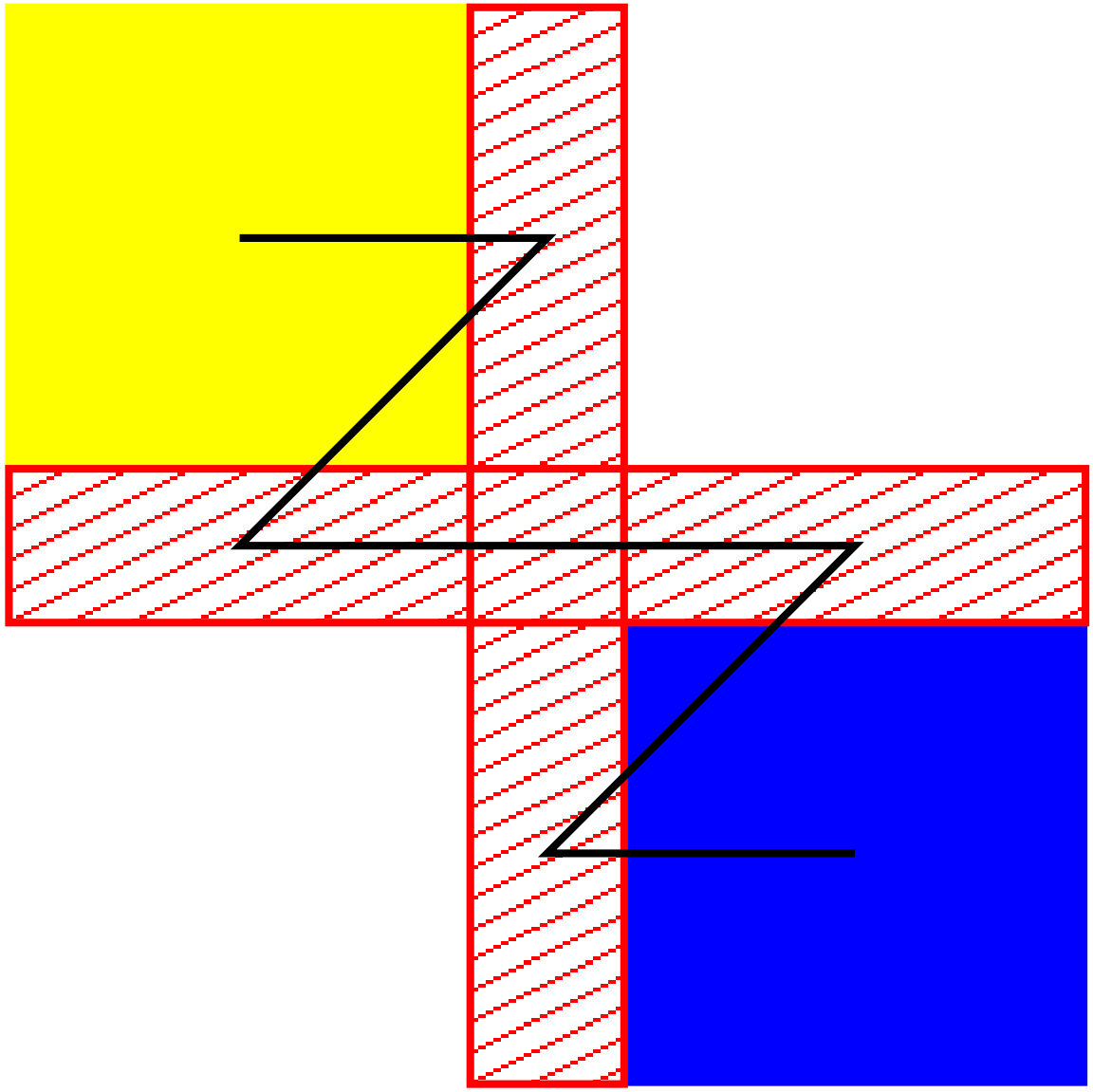


Two-dimensional SBD

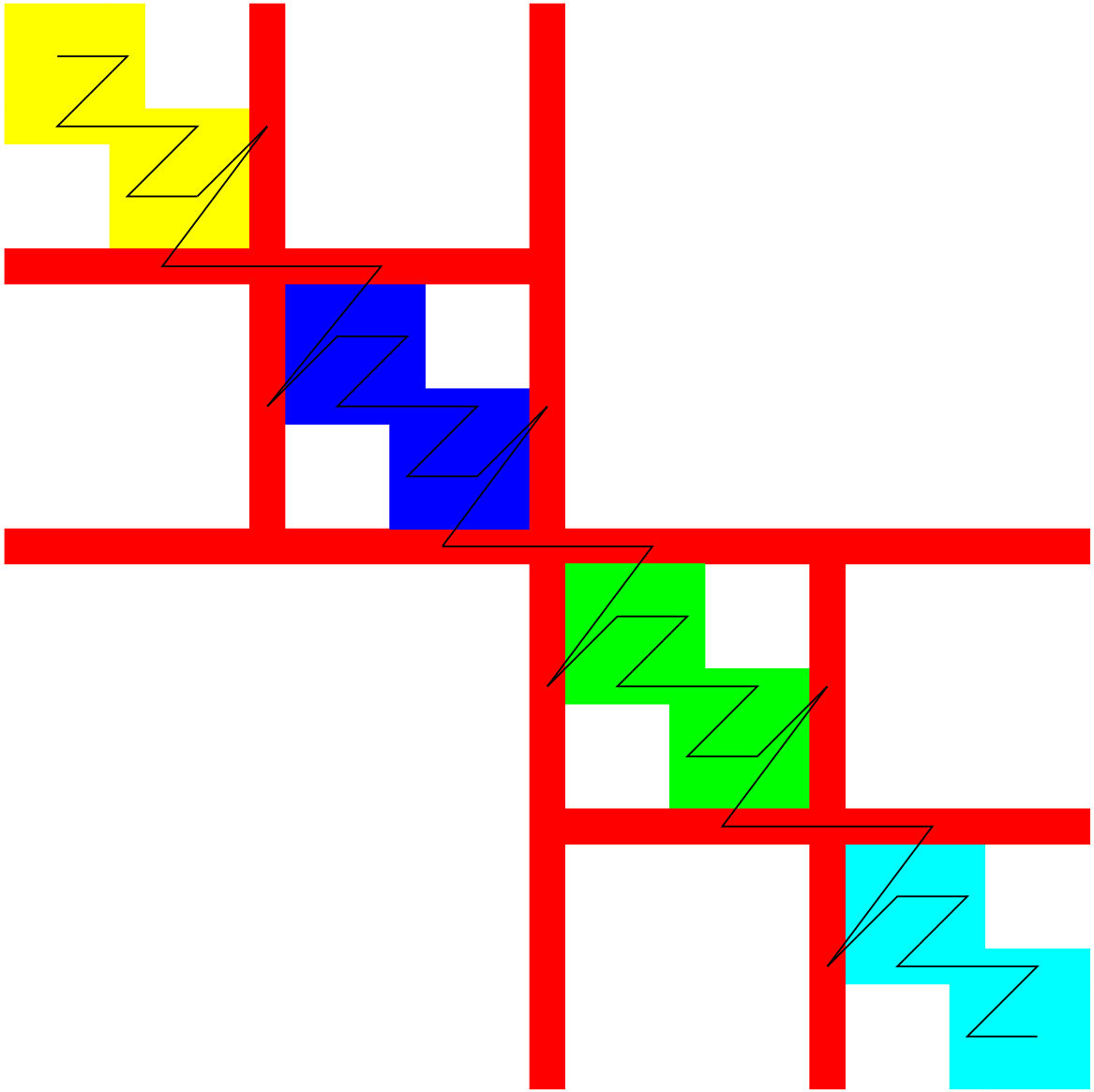


Zig-zag CRS is not suitable for handling 2D SBD

Two-dimensional SBD

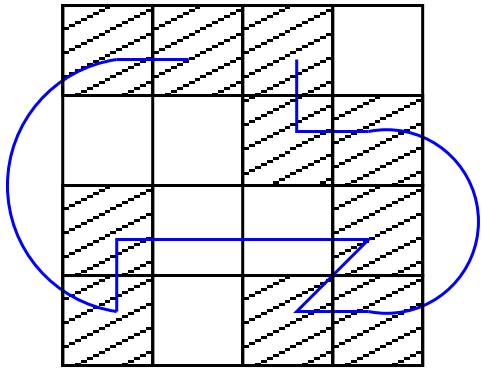


Two-dimensional SBD



Bi-directional Incremental CRS (BICRS)

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

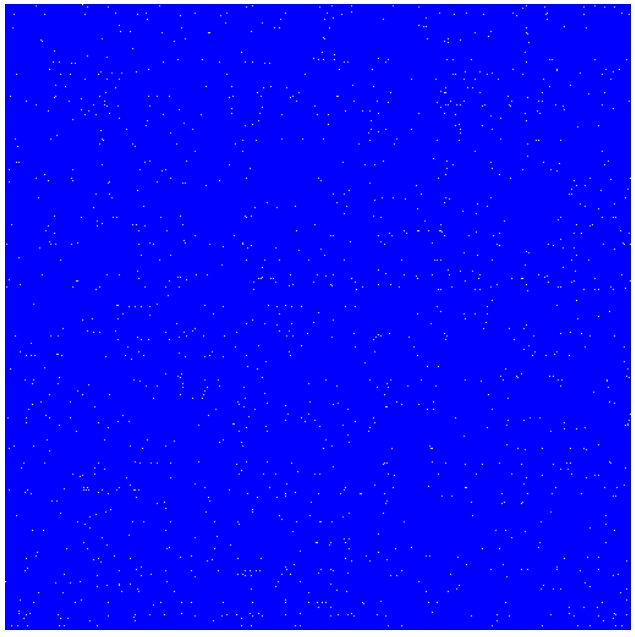


Stored as:

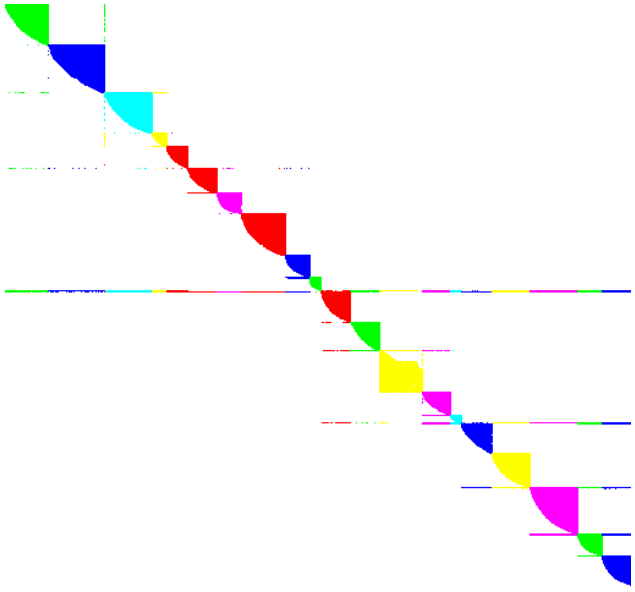
nzs:	[3 2 3 1 1 2 1 7 4 1]
col_increment:	[2 4 1 4 -1 5 -3 4 4 1] ,
row_increment:	[0 1 2 -1 1 -3]

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

Cache-oblivious SpMV – Stanford link matrix



Original



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

$p = 20$, $\epsilon = 0.1$: 8 minutes reordering time, 0.44 gain (BICRS/ICRS)

A.N. Yzelman & Rob H. Bisseling, *Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods*, SIAM Journal of Scientific Computation, Vol. 31, No. 4 (2009), pp 3128–3154

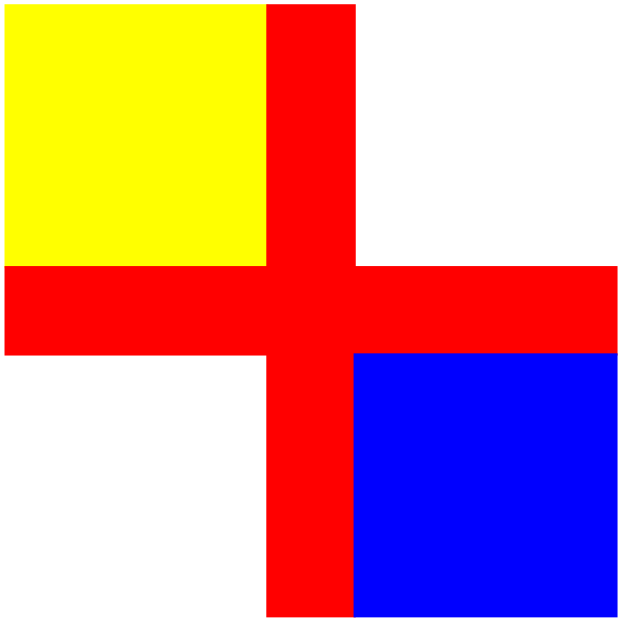
In relation to PSPIKE

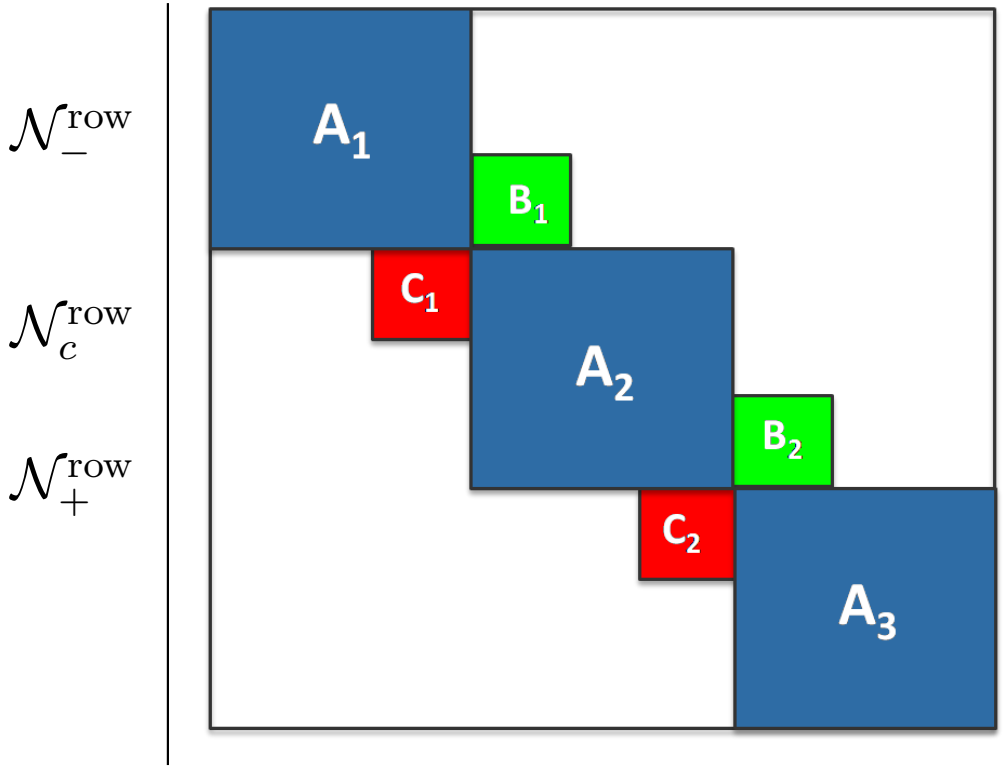
Partitioning

First, make use of Mondriaan as originally intended:

integrate the partitioning scheme, including vector distribution, for use within **parallel Bi-CGStab**. Another use is to increase the SpMV kernel's speed by reordering the partitions themselves.

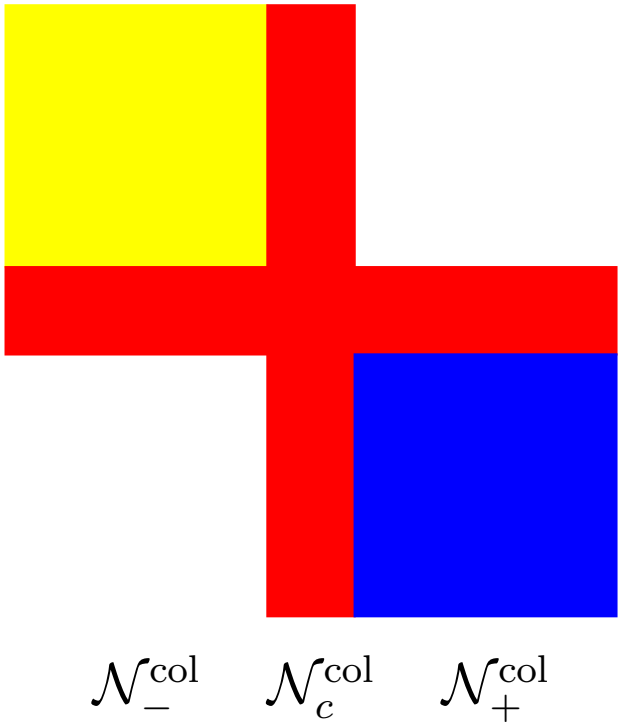
Another reordering scheme

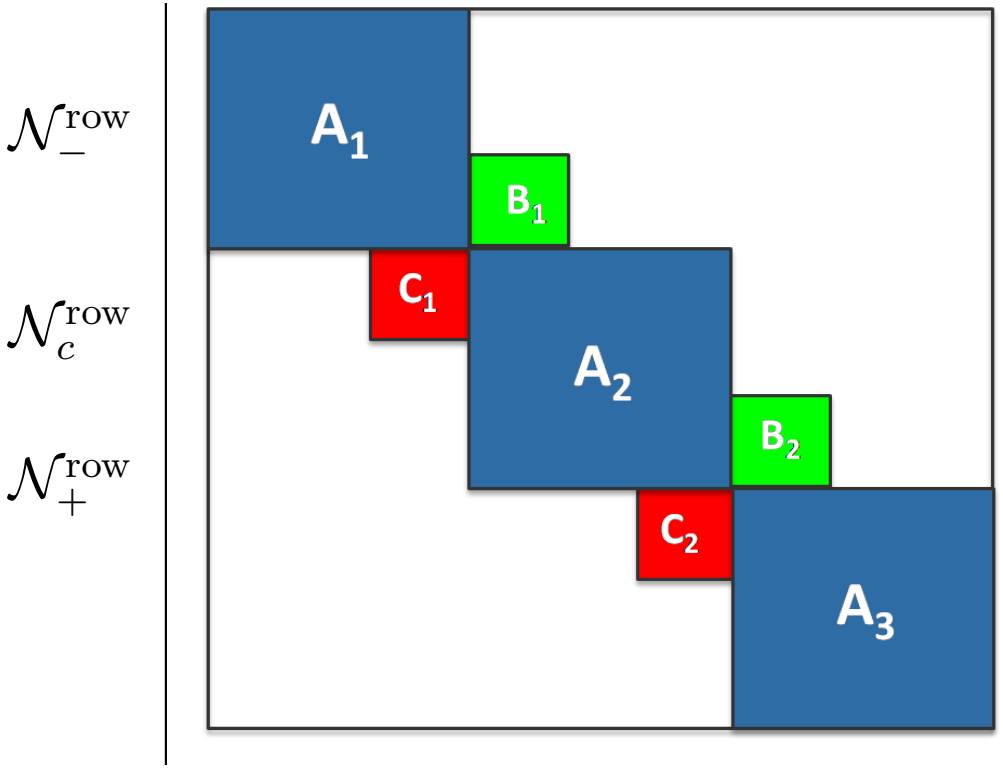




But also, perhaps more interesting, use the reordering facilities to permute (symmetric) input matrices to band matrices, replacing the current Sloan-based algorithm. Then reliably identify the coupling blocks.

Value-dependent reordering





A third item is to integrate moving "heavy" nonzeroes on or towards the diagonal, either as post-processing (could be disadvantageous for matrix structure) or implemented into Mondriaan itself (could be disadvantageous for partitioning).