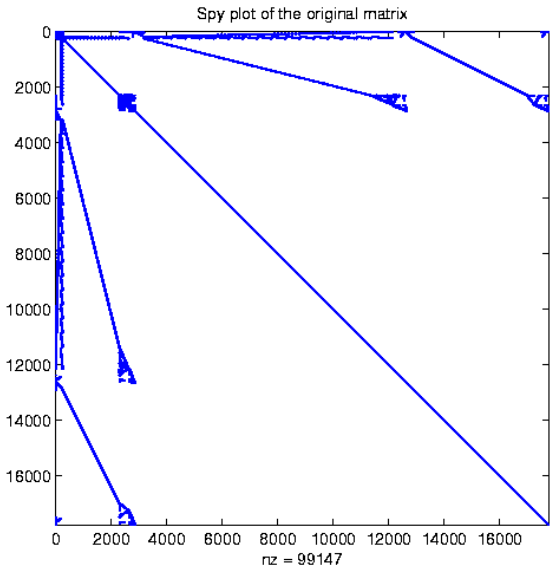
Fast sparse matrix–vector multiplication by partitioning and reordering

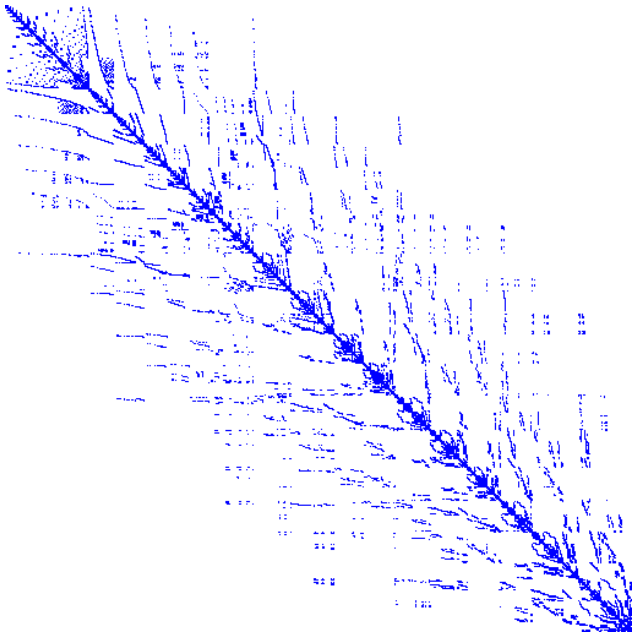
Albert-Jan Yzelman

June, 2011

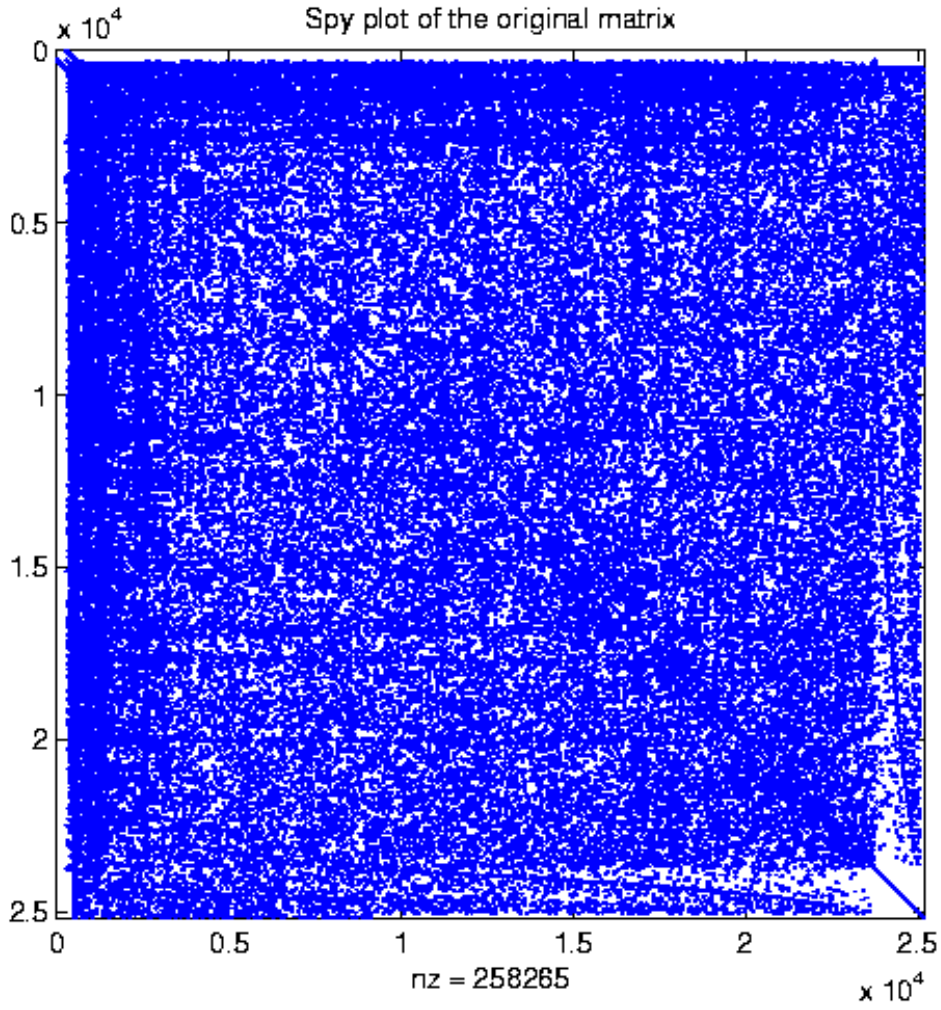
Outline

Chip industry / Markov chain modelling in chemistry

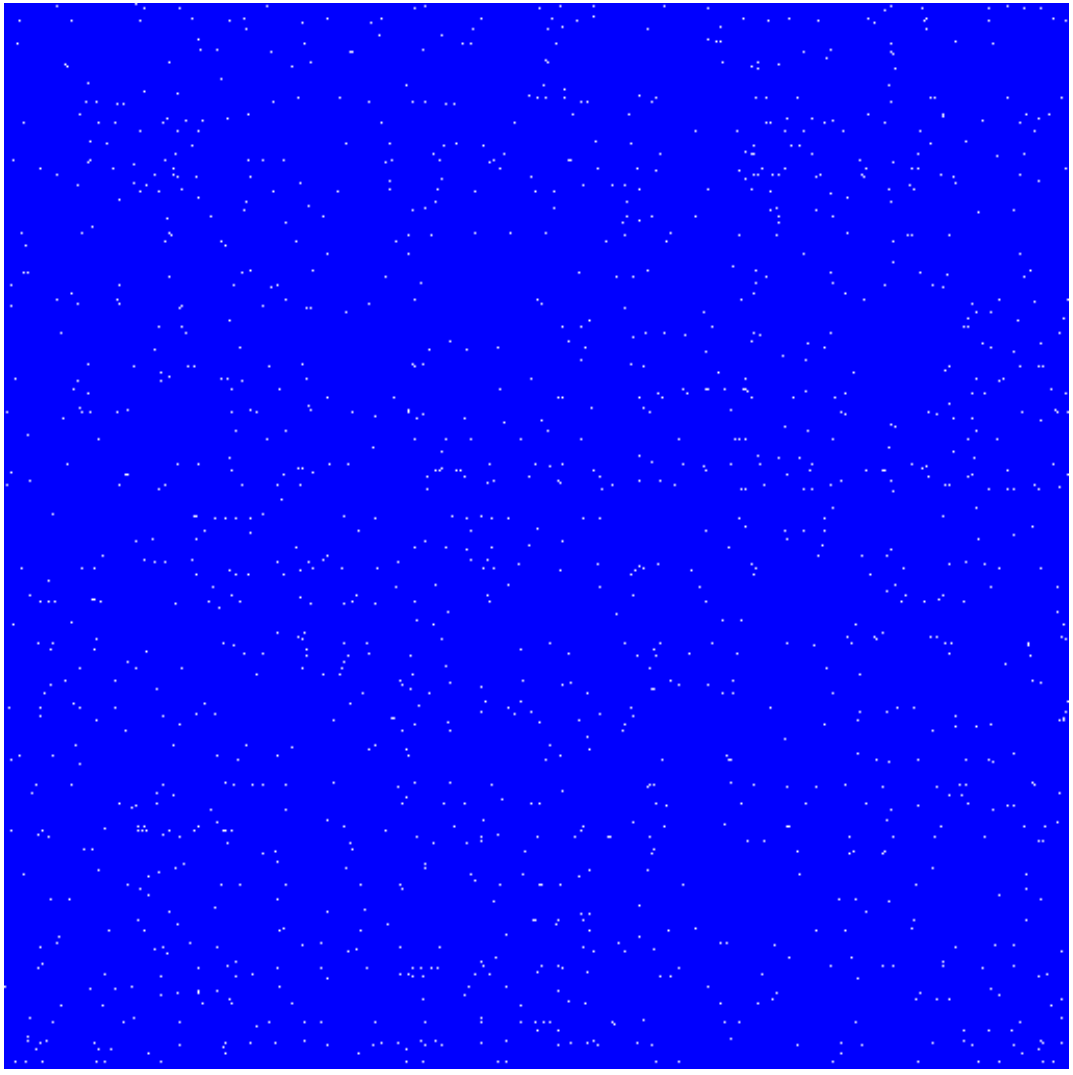




Chip industry



Link matrix



Bulk Synchronous Parallel

Parallel bridging models

- Many different architectures
- One parallel model
- Predictable performance
- Easy implementation

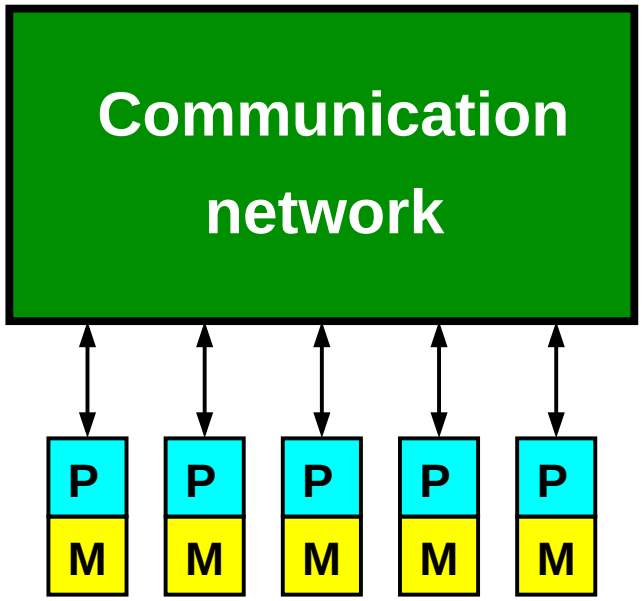
Parallel bridging models

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

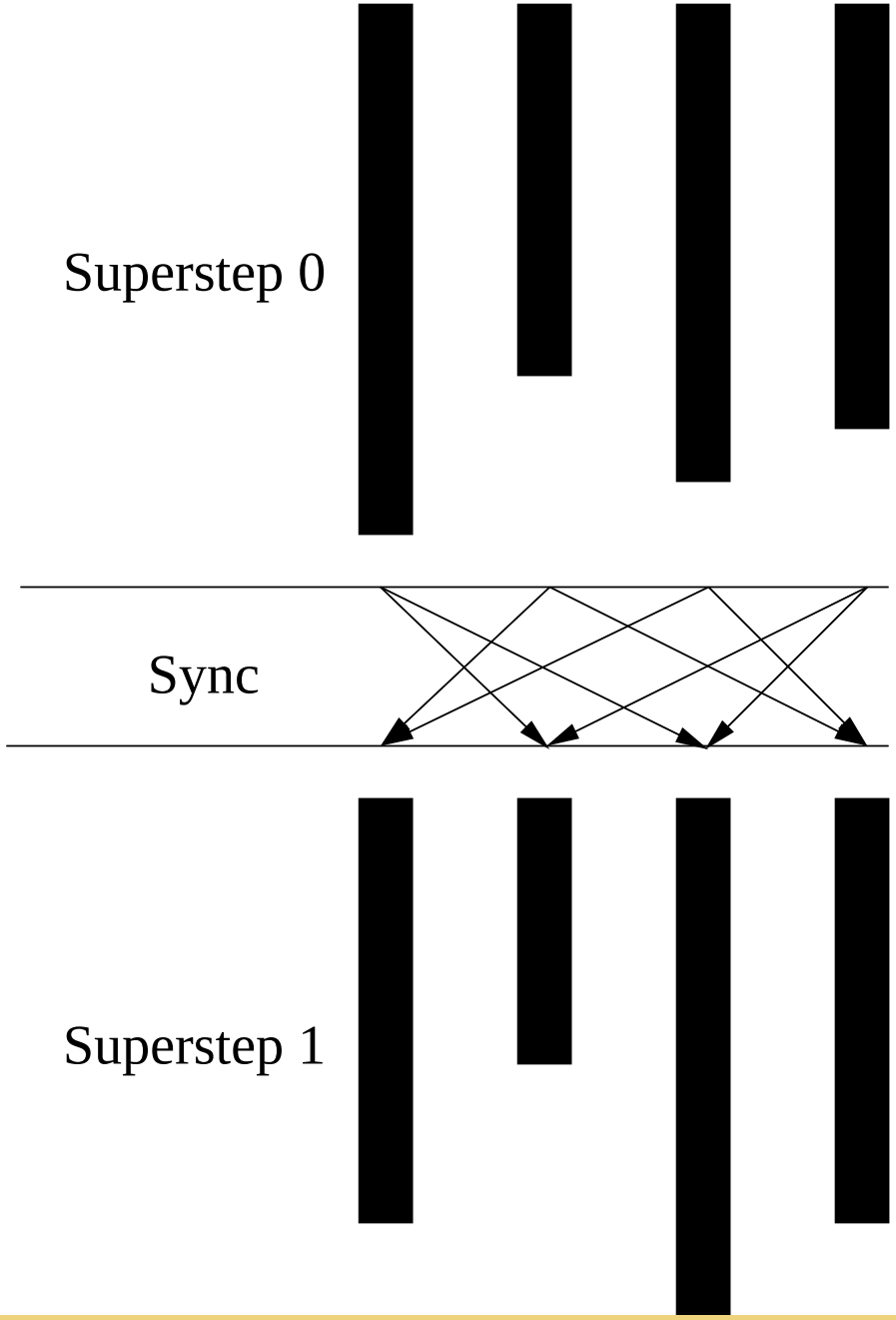
Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, Volume 33 (1990), pp. 103–111

Bulk Synchronous Parallel

A BSP-computer:

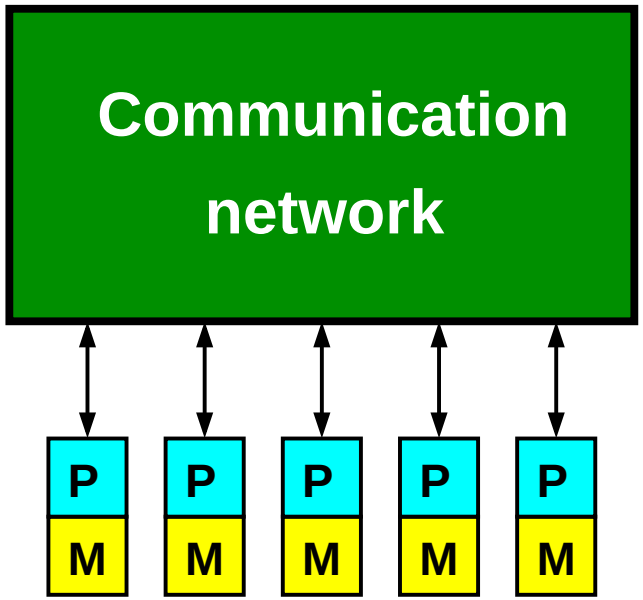


- consists of P heterogenous processors, each with local memory
- executes a Single Program on Multiple Data (SPMD)
- performs no communication during calculation (supersteps)
- communicates only during *barrier synchronisation*


Superstep 0
Sync
Superstep 1

Bulk Synchronous Parallel

A BSP-computer furthermore:



- does r flops each second *per processor*
- 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

Cost model:

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

Define $c_i = \max\left\{\max_s r_i^{(s)}, \max_s t_i^{(s)}\right\}$ and $w_i = \max_s w_i^{(s)}$.

If T is the number of supersteps, the cost of a BSP algorithm is:

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

Bulk Synchronous Parallel

A BSP-*algorithm* can:

- 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_qsize()`
  `bsp_move()`

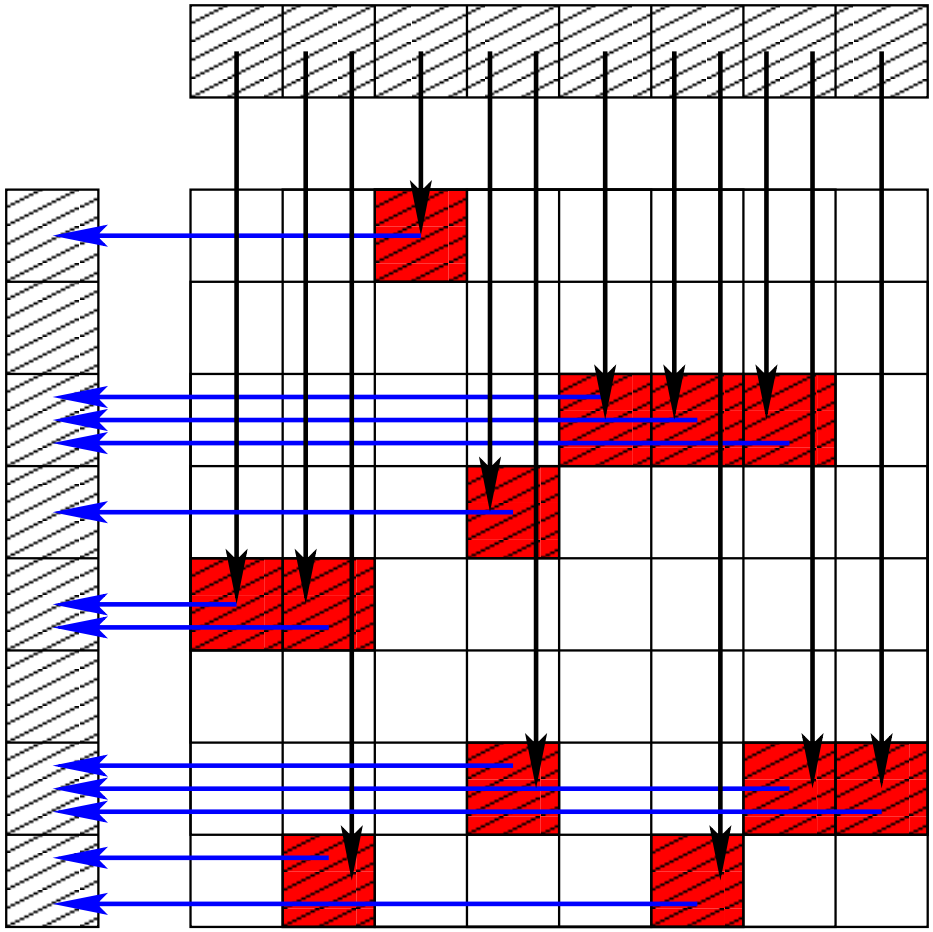
As implemented in, e.g., the Oxford BSP library (Bill McColl et al.)

Example: sparse matrix, dense vector multiplication

`y=Ax`:

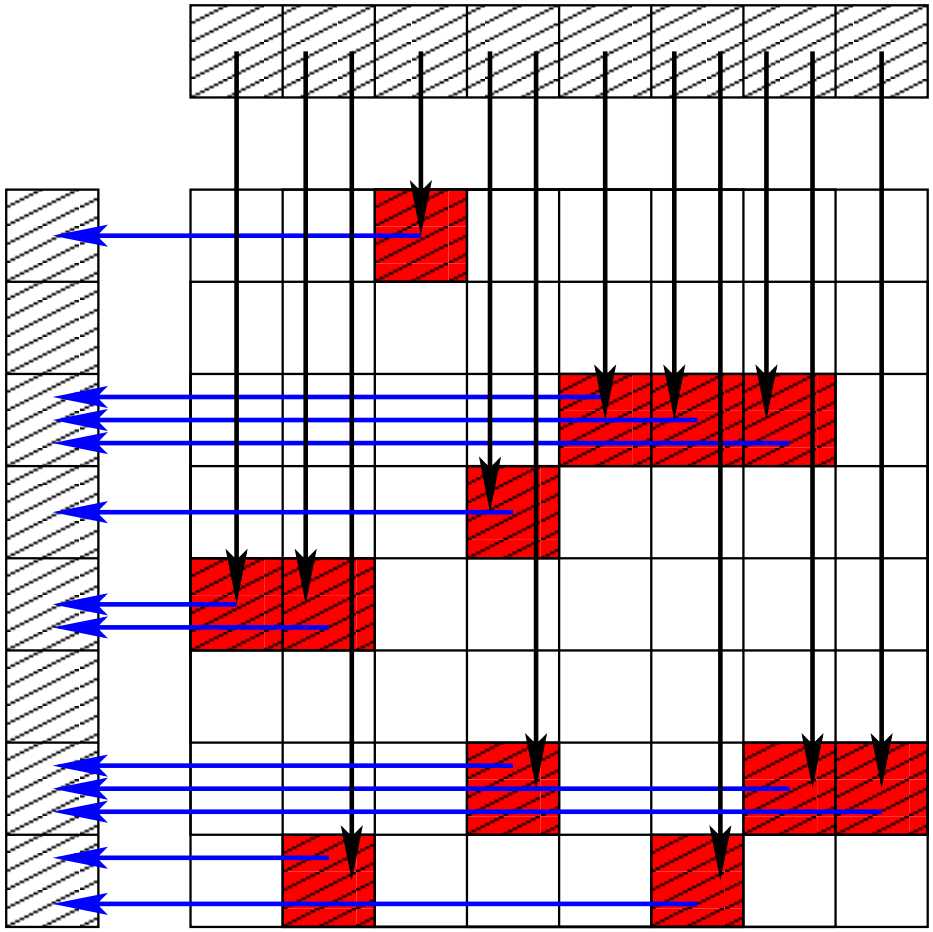
for each nonzero k **from** A

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



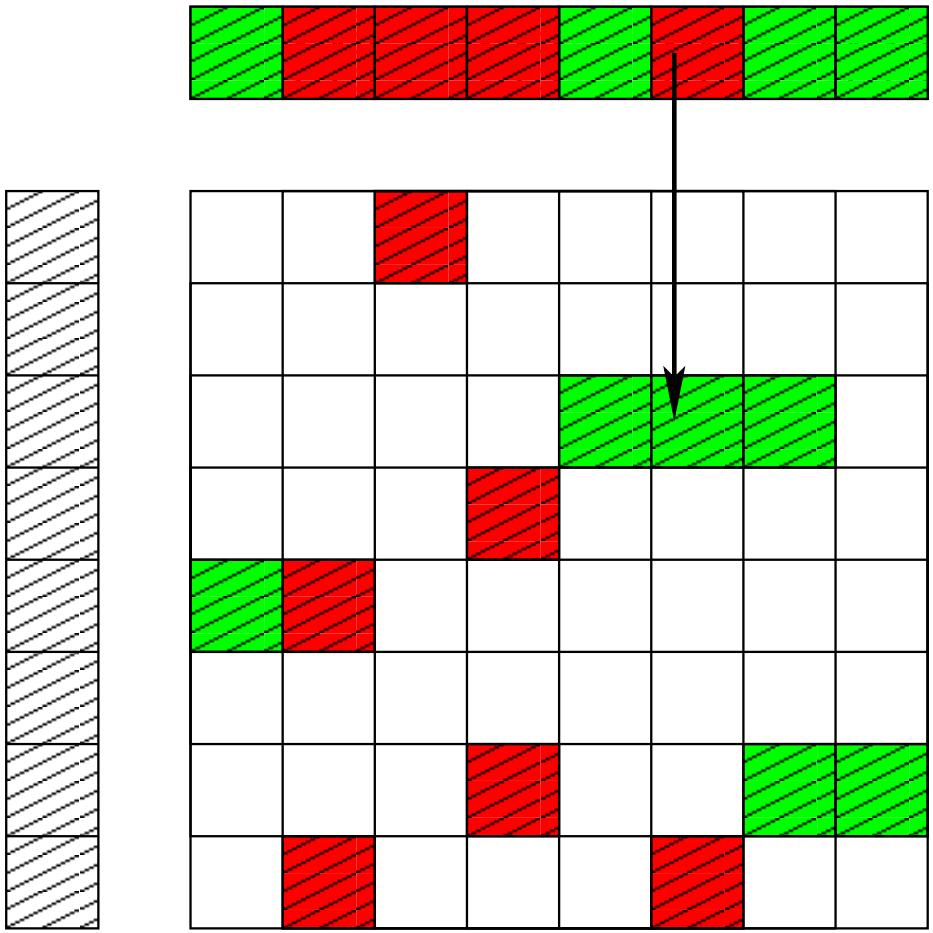
Example: sparse matrix, dense vector multiplication

To do this in parallel:
Distribute the nonzeroes of A, but also distribute x and y;
each processor should have about $1/P$th of the total data (scalability).



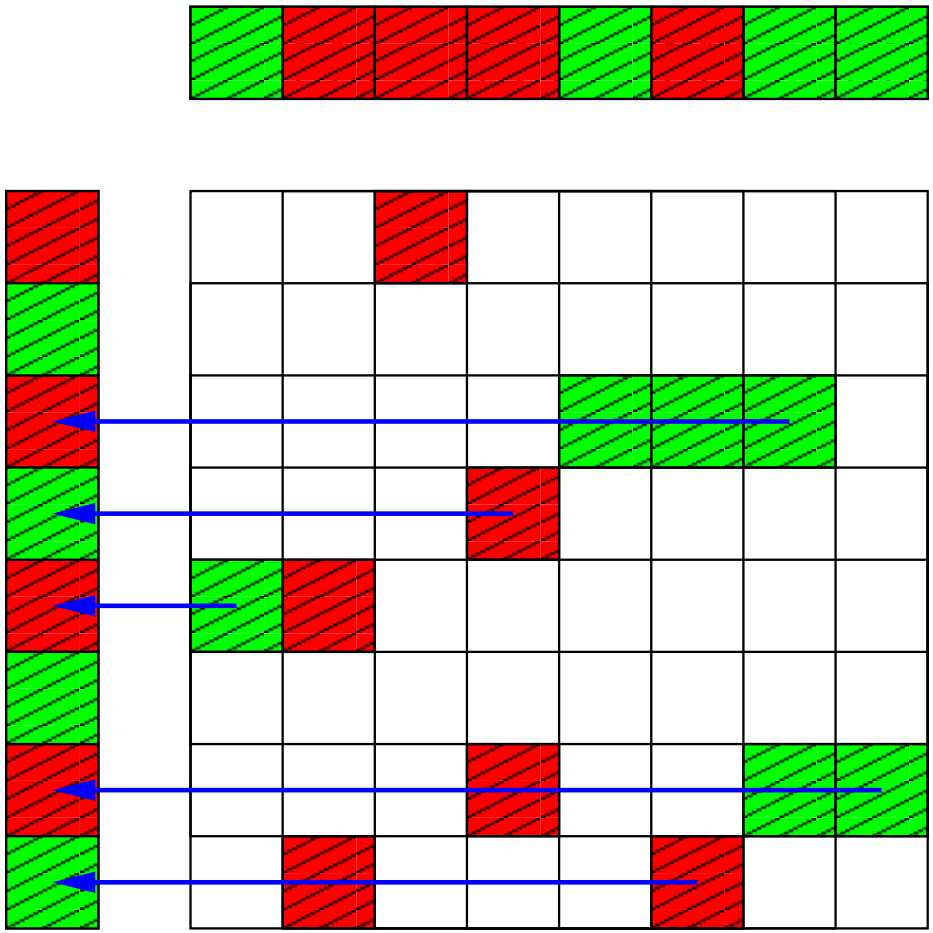
Example: sparse matrix, dense vector multiplication

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



Example: sparse matrix, dense vector 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.



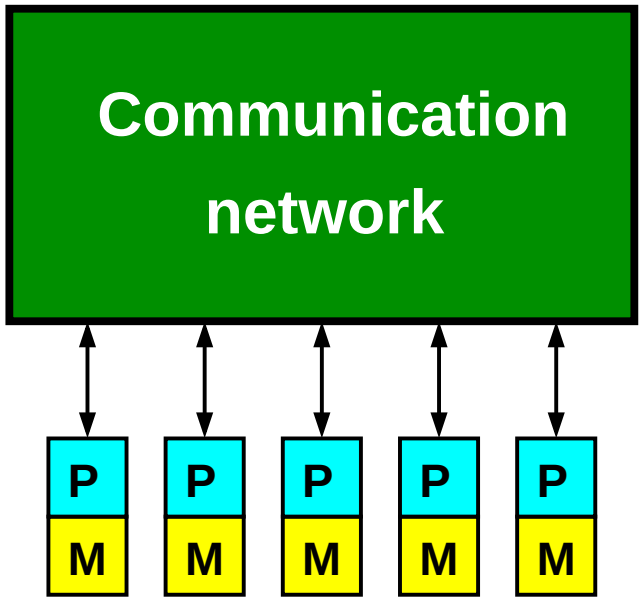
Example: sparse matrix, dense vector multiplication

The algorithm:

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

MulticoreBSP

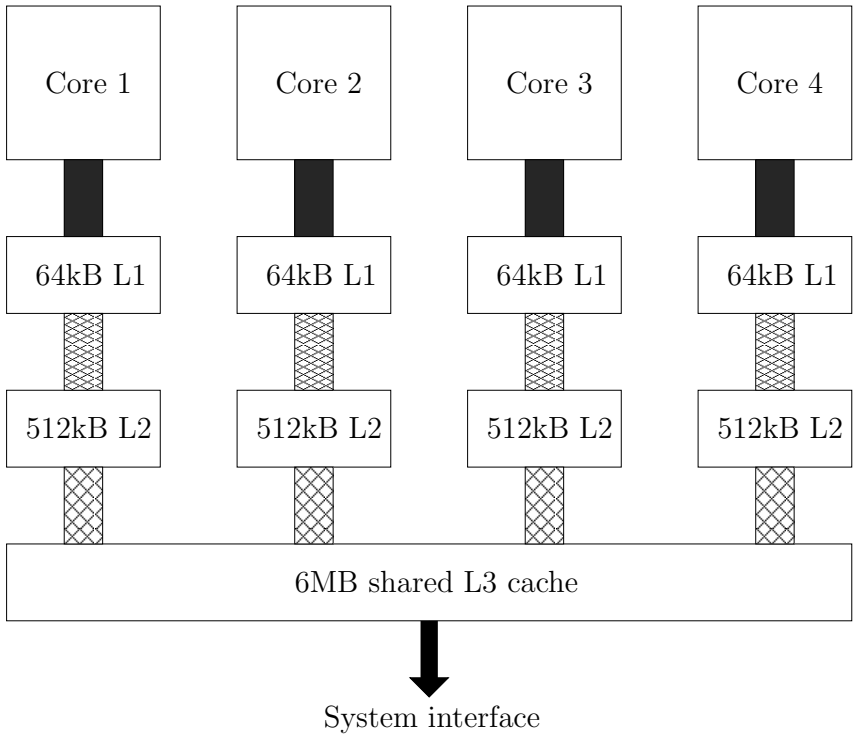
For Multicore, the original model



may no longer apply.

MulticoreBSP

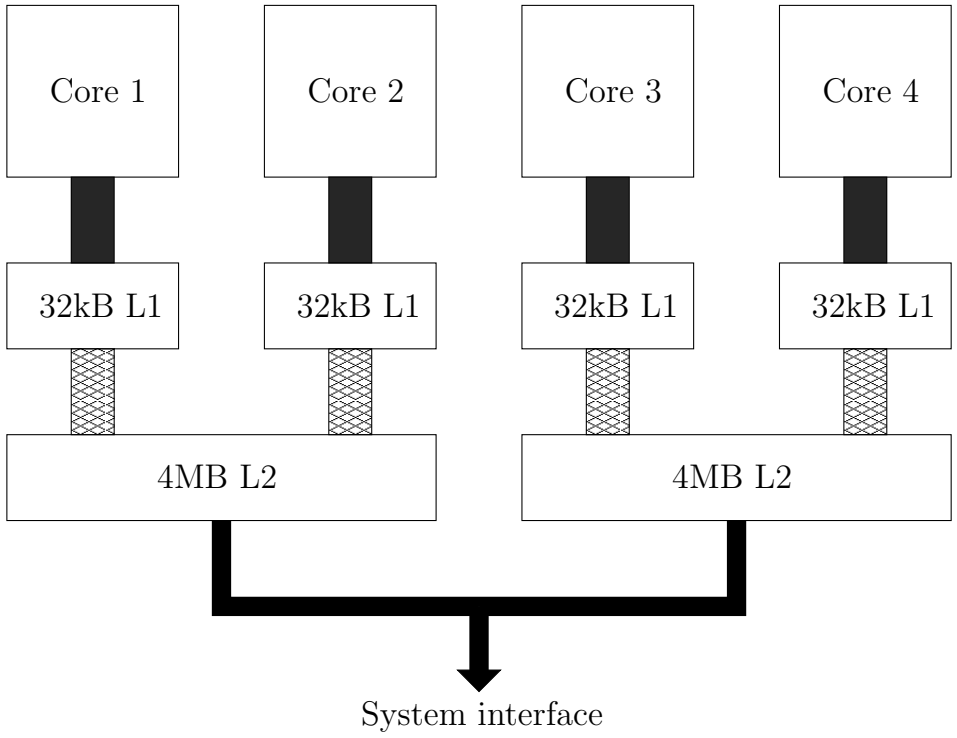
The AMD Phenom II 945e processor has uniform memory access:



is modelled well by BSP;

MulticoreBSP

The Intel Core 2 Q6600 processor has cache-coherent non-uniform memory access (cc-NUMA):



is not modelled well by BSP.

Leslie G. Valiant, *A bridging model for multi-core computing*, *Lecture Notes in Computer Science*, vol. 5193, Springer (2008); pp 13–28.

MulticoreBSP

New primitive:

- 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)
  bsp_direct_get*(source, source_PID, dest)*
- Send messages, synchronously (BSMP):
  bsp_send(data, dest_PID)
  bsp_qsize()
  bsp_move()

MulticoreBSP

MulticoreBSP brings BSP programming to shared-memory architectures.

Programmed in standard Java (5 and up), this is a fully object-oriented library containing only 10 primitives, 2 purely virtual functions (*parallel_part* and *sequential_part*), and 2 interfaces.
Data types which can be communicated with are defined using an interface.

This makes MulticoreBSP

- transparent and easy to learn,
- have predictable performance,
- robust (no data racing, no deadlocks),
- potentially usable for both shared- and distributed-memory systems

MulticoreBSP

Alternative (2-step) SpMV algorithm in MulticoreBSP:

1. **for all** nonzeroes k **from** A
   if both row and column of k are local
   add do the SpMV for k
   if column of k is not local
   direct get element from x, and do SpMV for k
   send all non-local row sums to the correct processor
   synchronise
2. **add** all incoming row sums to the corresponding $y[i]$

MulticoreBSP

Software is available at:

`http://www.multicorebsp.com`

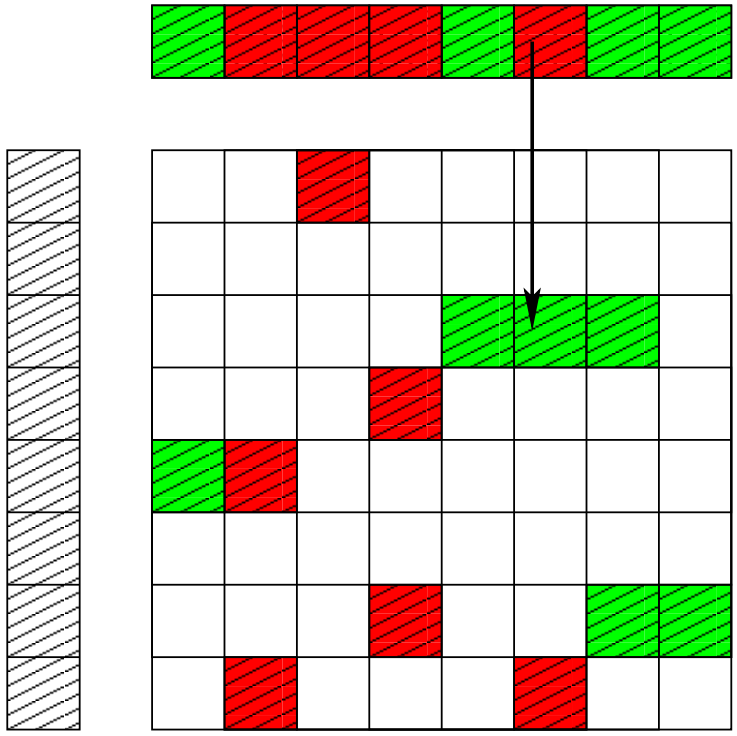
Yzelman and Bisseling, *An Object-Oriented BSP Library for Multicore Programming*, Concurrancy and Computation: Practice and Experience, 2011 (Accepted for publication).

Partitioning

Automatic nonzero partitioning

What causes the communication?

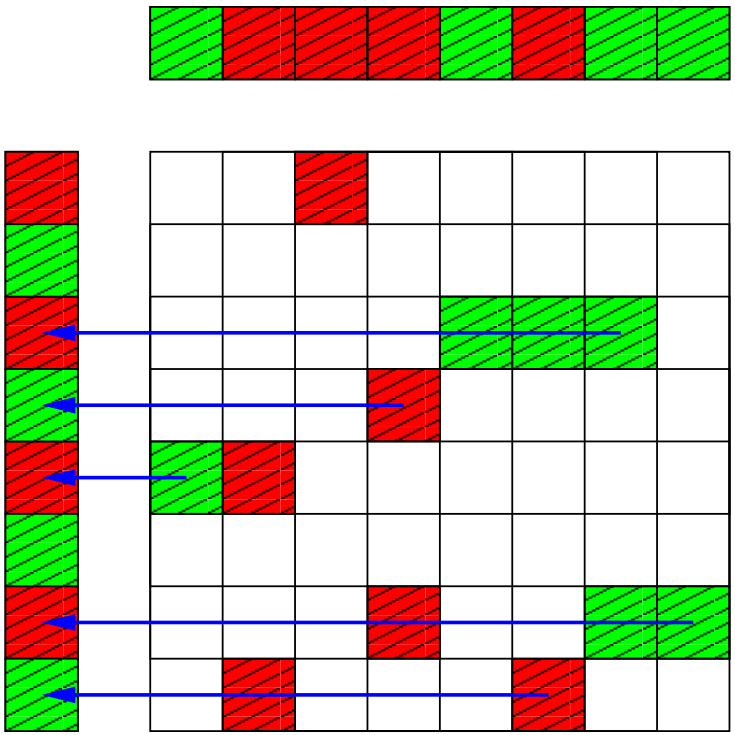
- nonzeroes on the same column distributed to different processors: fan-out communication



Automatic nonzero partitioning

What causes the communication?

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



Automatic nonzero partitioning

Load balancing

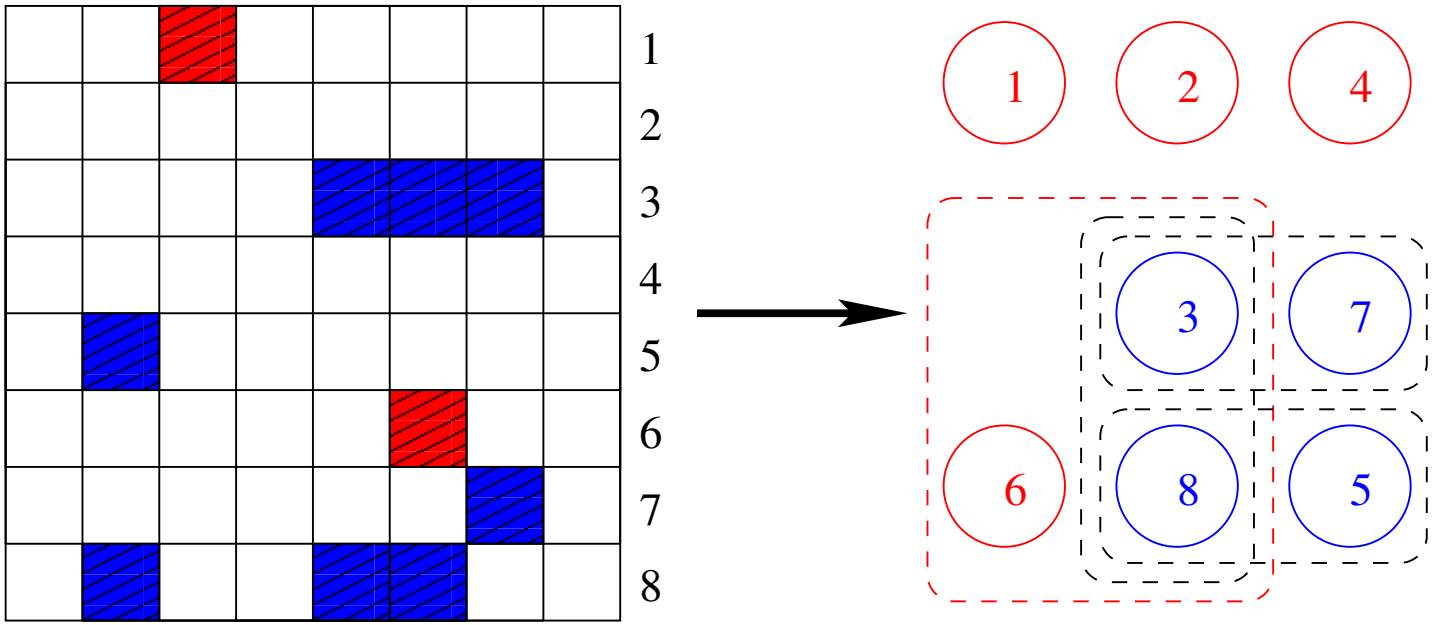
For each superstep i, let $\bar{w}_i = \frac{1}{P} \sum_{s \in [0,P-1]} w_i^{(s)}$ be the average workload. The load-balance constraint is:

$$\max_s |\bar{w}_i - w_i^{(s)}| \leq \epsilon \bar{w}_i,$$

where ϵ is the maximum load imbalance parameter.

Automatic nonzero partitioning

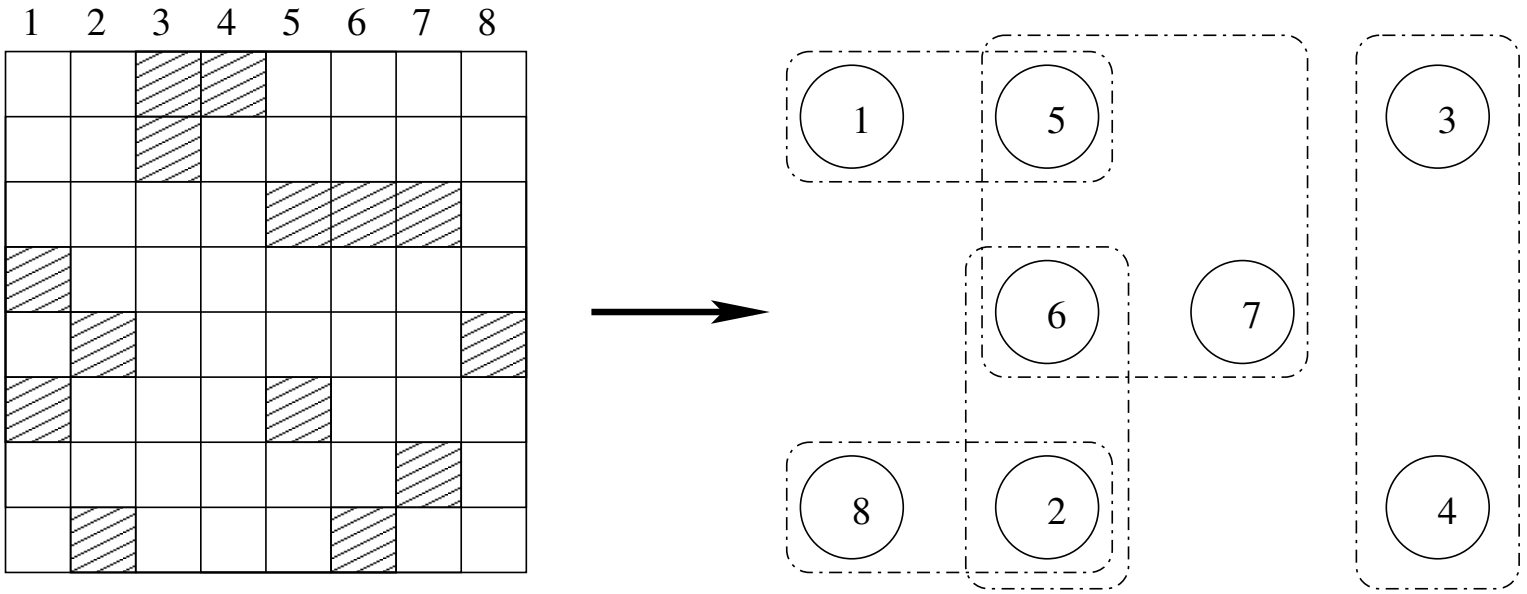
"Shared" columns: communication during fan-out



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

Automatic nonzero partitioning

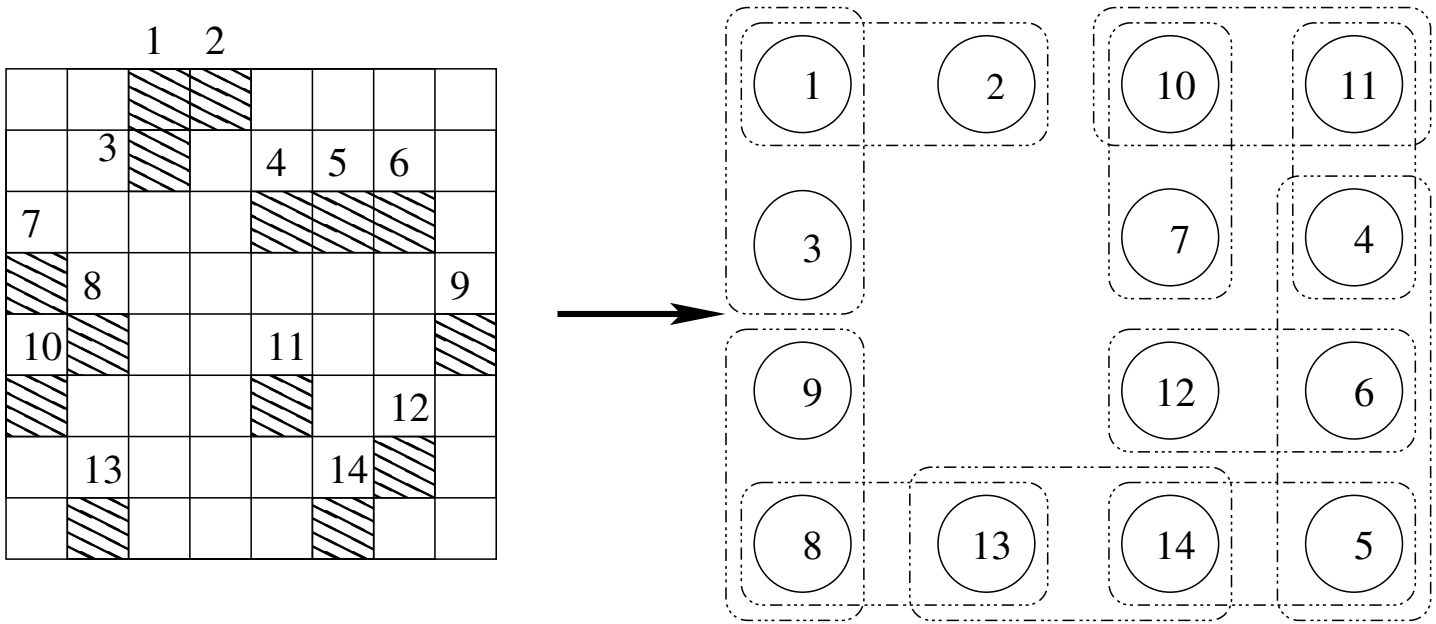
"Shared" rows: communication during fan-in



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

Automatic nonzero partitioning

Catch communication both ways:



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

A cut net n_i means communication. The number of processors involved is:

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

So the quantity to minimise is:

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

Partitioning strategy:

- Model the sparse matrix using a hypergraph
- **Partition the vertices** of that hypergraph in two so that C is minimised under the load-balance constraint.

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).

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

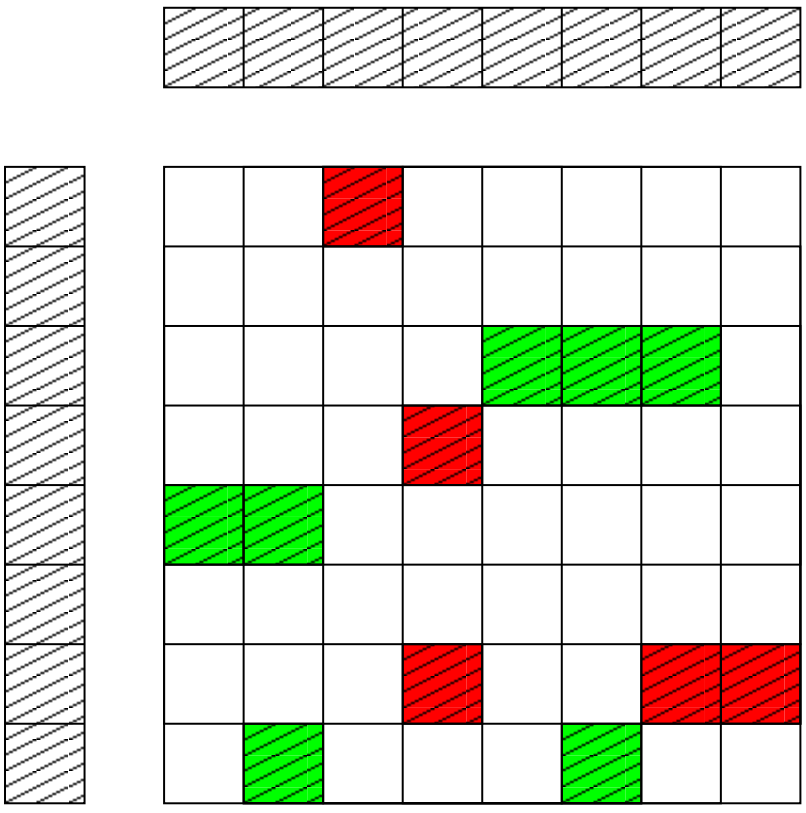
Catalyürek & Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel Distributed Systems 10 (1999).

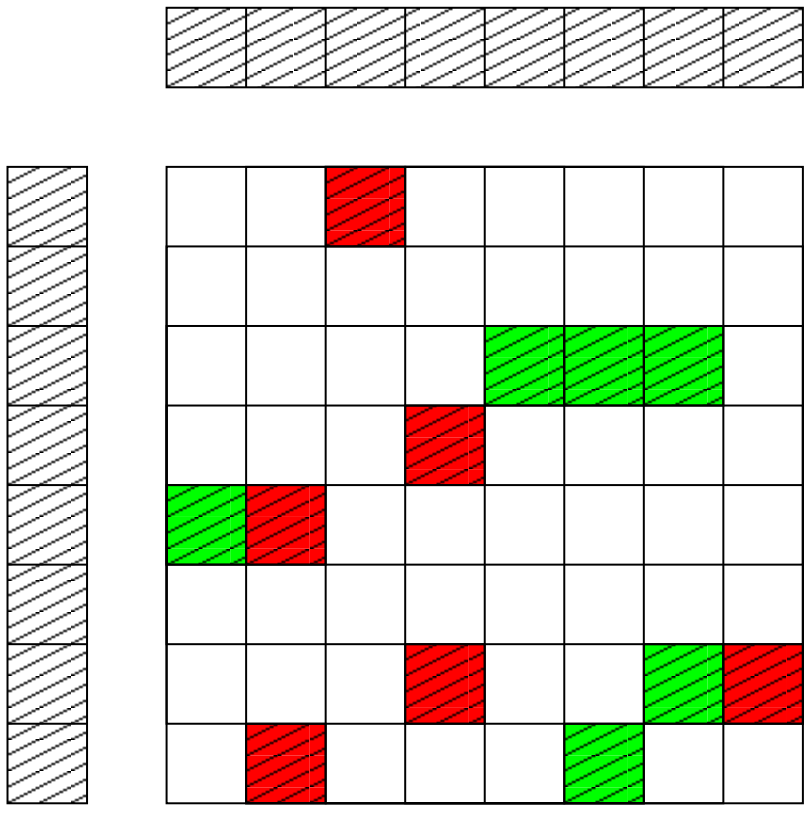
Bisseling & Vastenhouw, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review Vol. 47(1), 2005.

Mondriaan partitioning strategy:

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

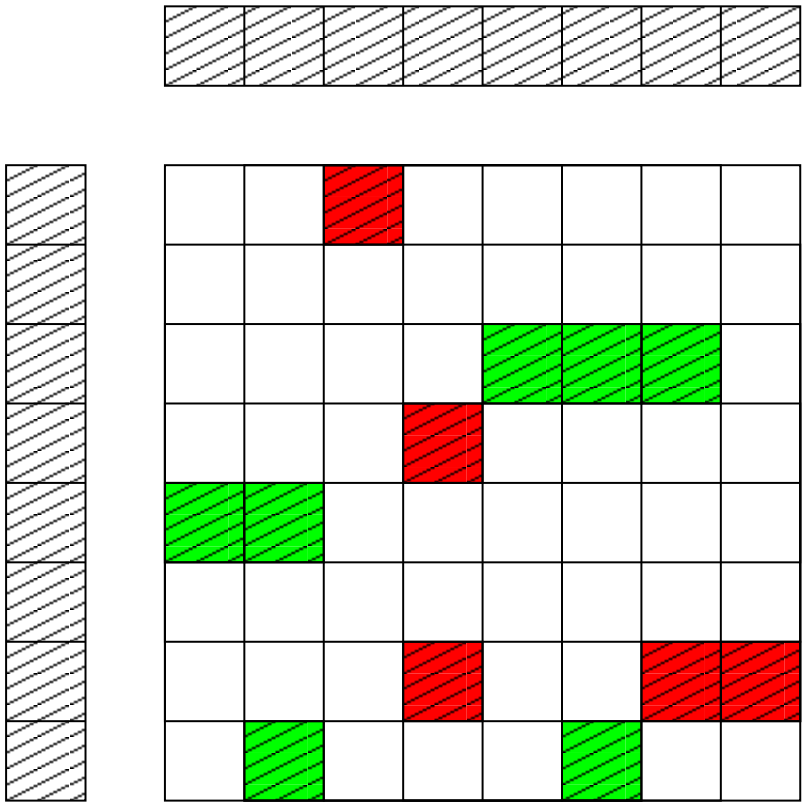
Try both row- and column-net, and choose best





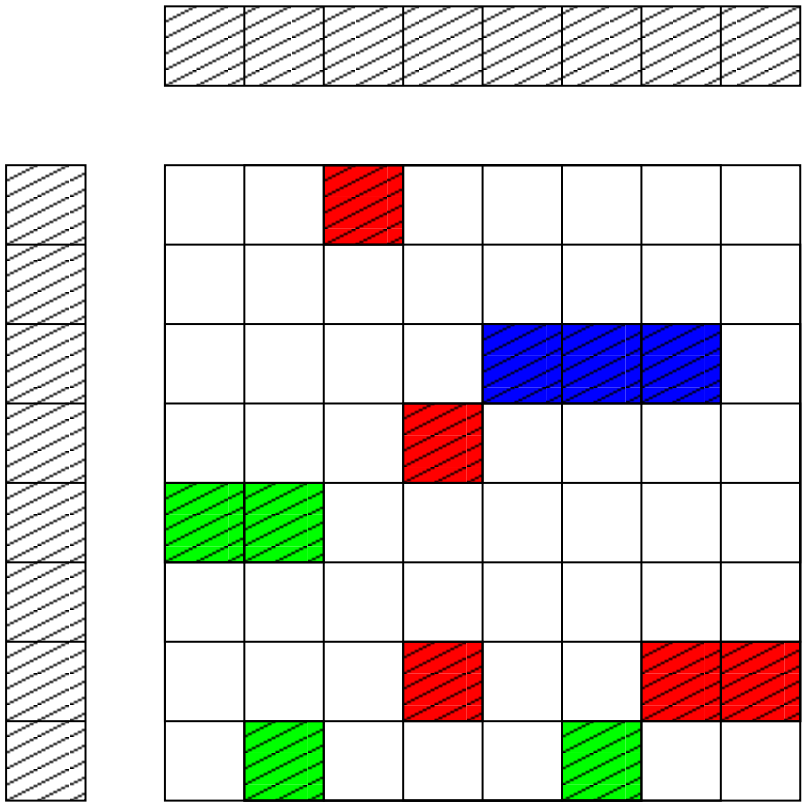
Mondriaan partitioning strategy:

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



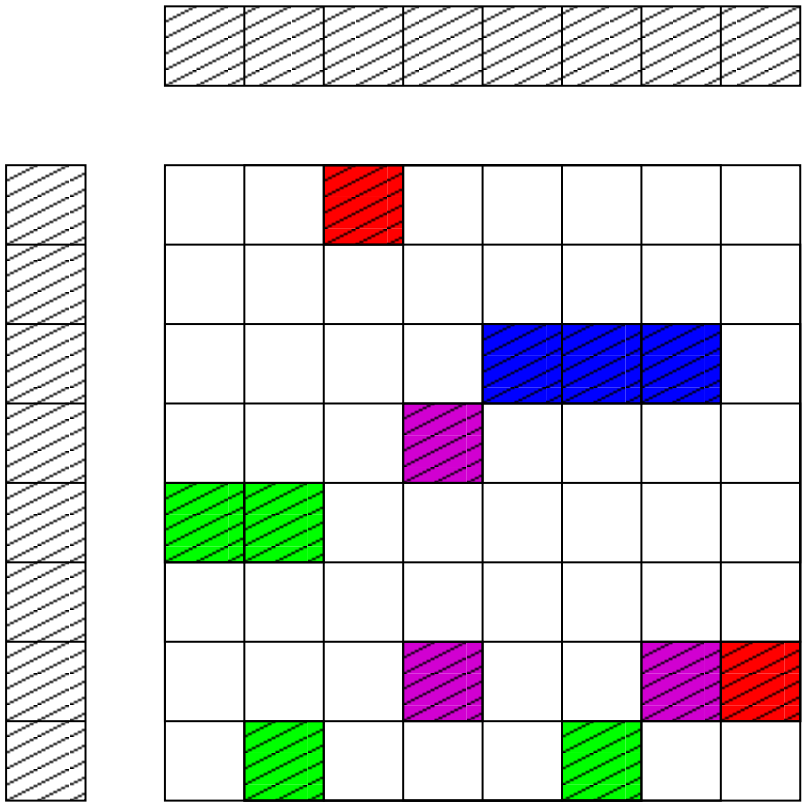
Mondriaan partitioning strategy:

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



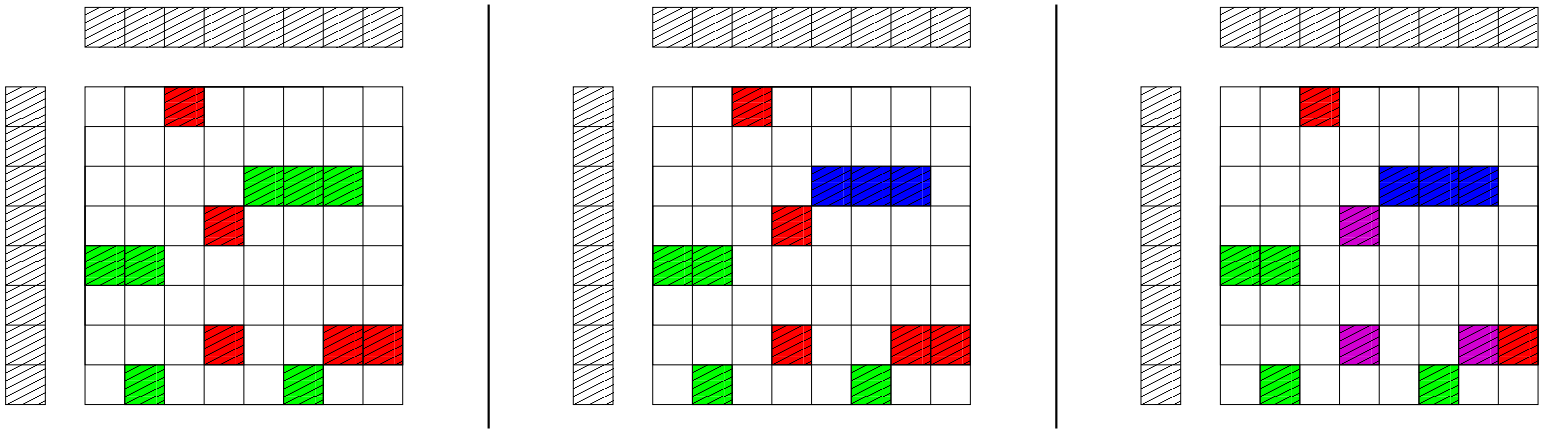
Mondriaan partitioning strategy:

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



Mondriaan:

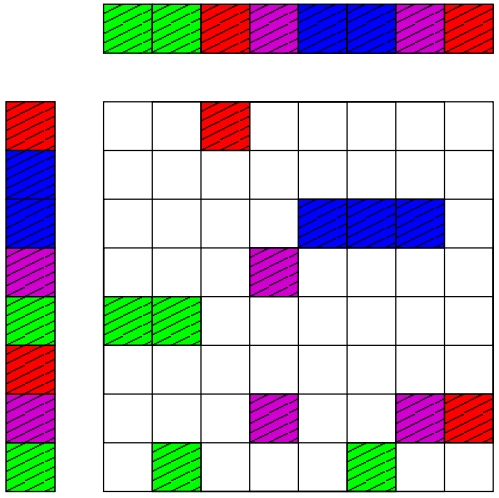
- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



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**

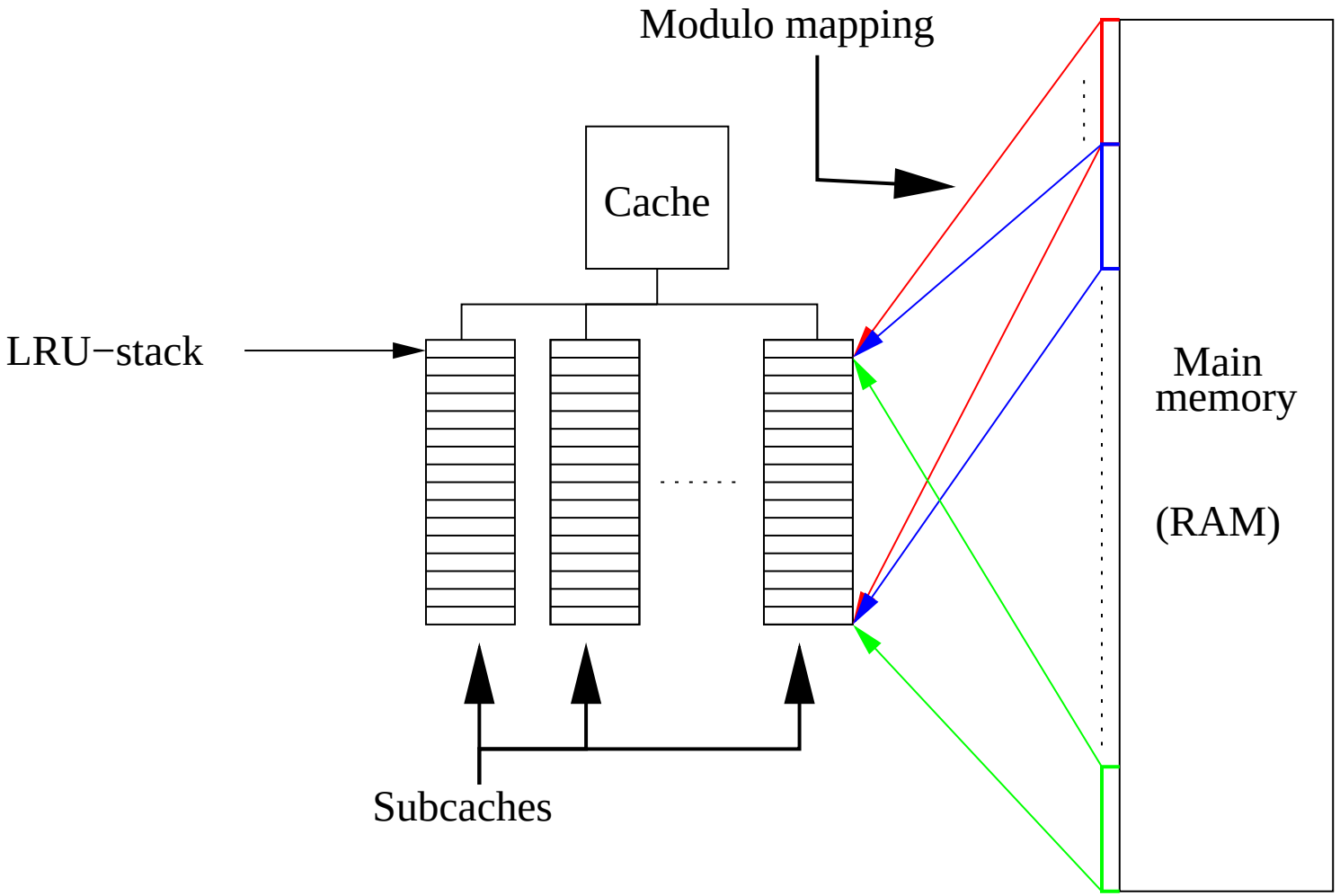


Rob H. Bisseling and Wouter Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis, Vol. 21 (2005) pp. 47-65

Sequential SpMV

Realistic cache

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



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 $k = L = 3$:

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 $k = L = 3$:

$$x_0 \quad \Longrightarrow \quad \begin{matrix} a_{00} \\ x_0 \end{matrix} \quad \Longrightarrow$$

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 $k = L = 3$:

$$\begin{array}{ccccc} x_0 & & a_{00} & & y_0 \\ & & x_0 & & a_{00} \\ \underline{\quad\quad} & \Longrightarrow & \underline{\quad\quad} & \Longrightarrow & \underline{x_0} \end{array} \Longrightarrow$$

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 $k = L = 3$:

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

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 $k = L = 3$:

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

The dense case

Dense matrix–vector multiplication

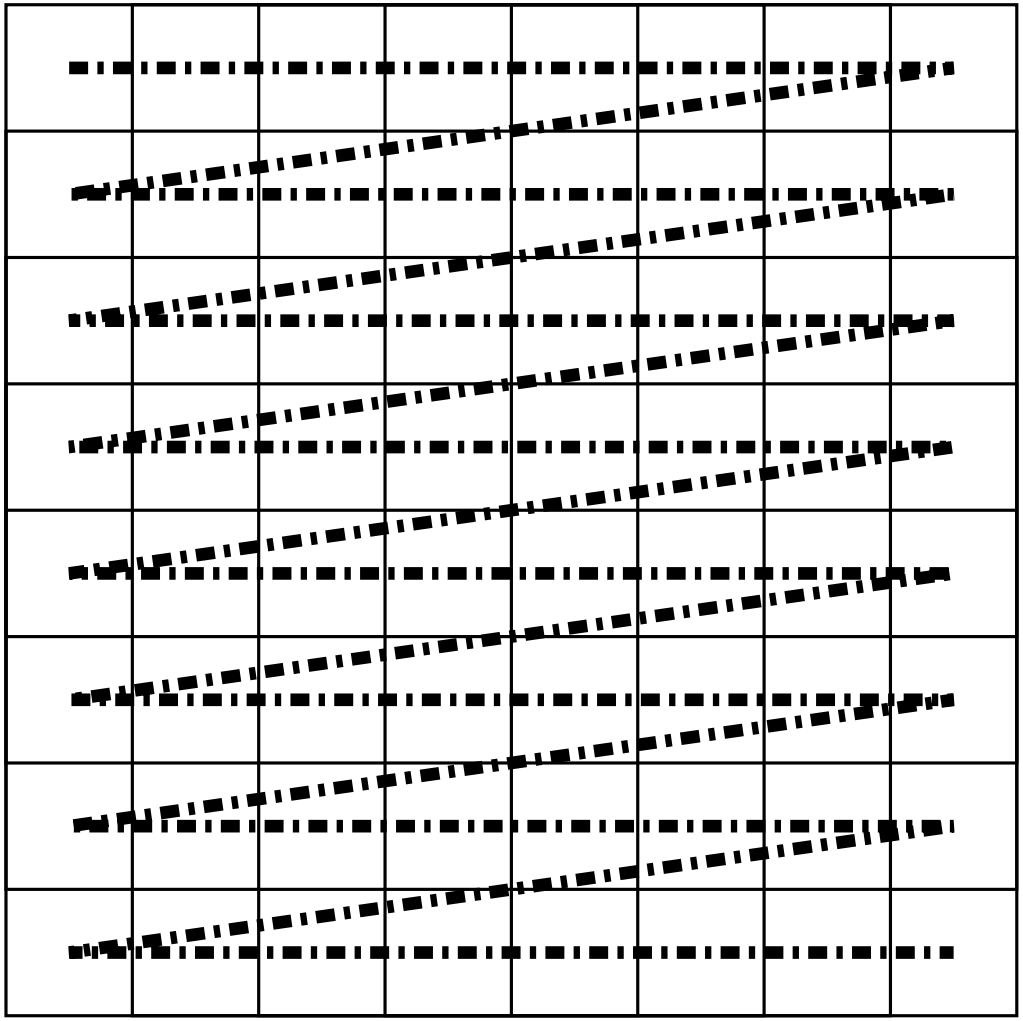
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 3$:

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

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)

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

The sparse case

Sparse matrix–vector multiplication (SpMV)

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

The sparse case

Sparse matrix–vector multiplication (SpMV)

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

We cannot predict memory accesses in the sparse case!

Adapt sparse matrix data structures for locality and lower bandwidth?

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* (Masters 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:

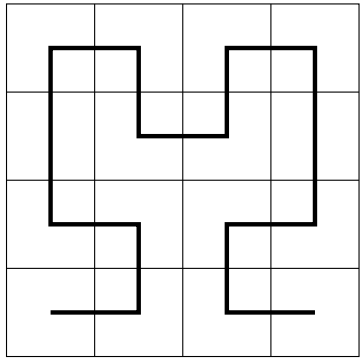
nzs: [4 1 3 2 3 1 2 7 0 1 1]
blk: [0 3 5 6 7 11]
col: [0 2 0 3 0]
row: [0 1 2 4 5]

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

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

Fractal datastructures (triplets)

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



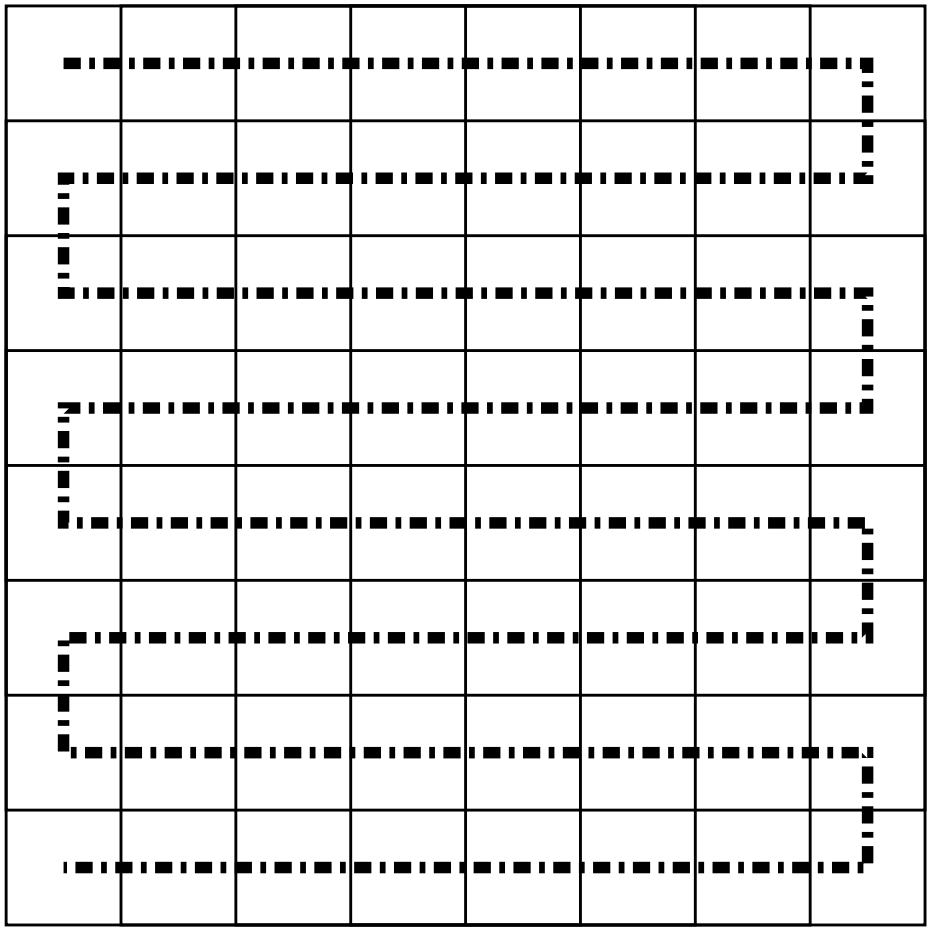
Stored as:

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

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:

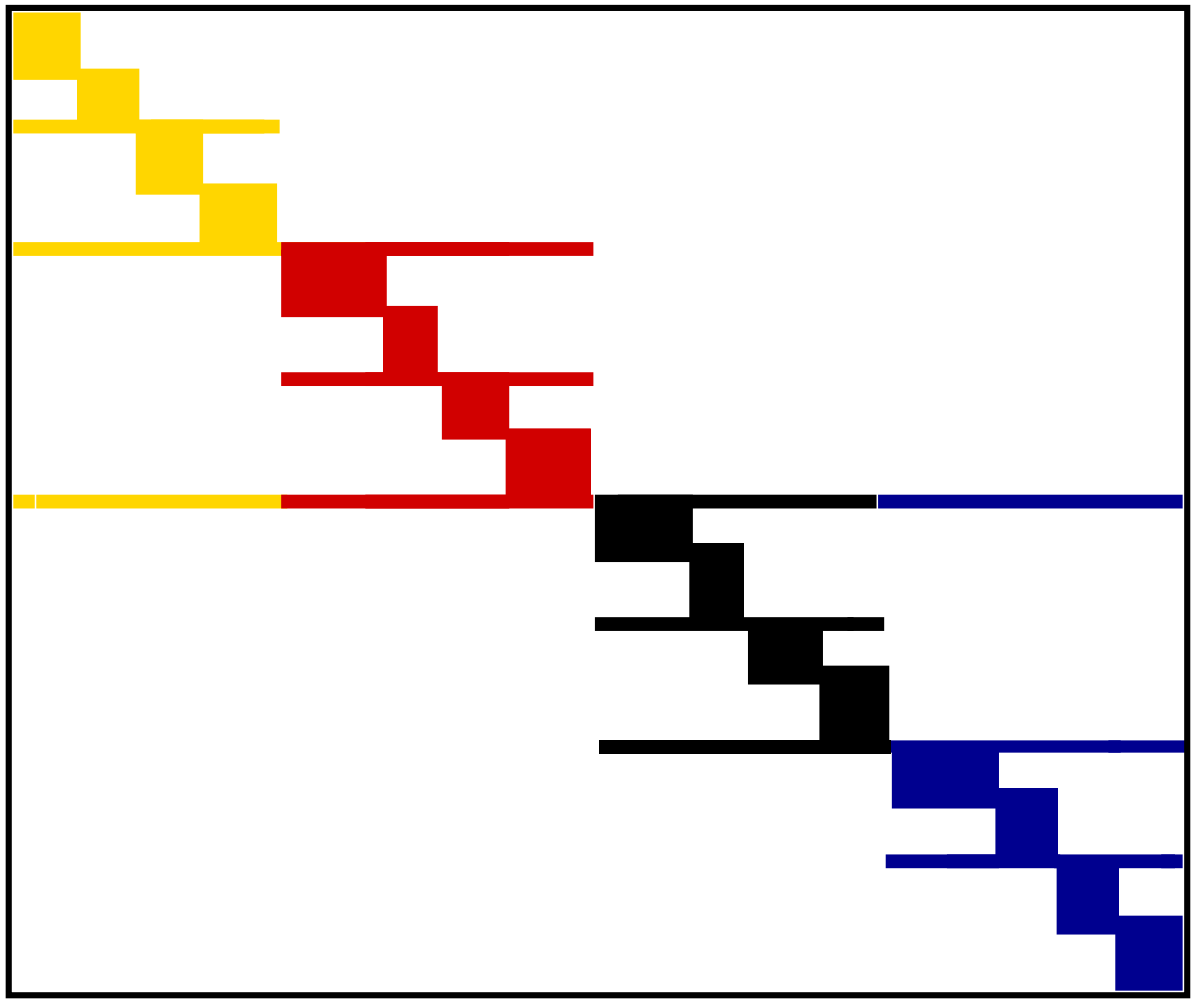
nzs:	[4 1 3 3 2 1 2 1 1 7]
col:	[0 1 2 3 2 0 3 3 2 0] , $2nnz + (m + 1)$ accesses
row:	[0 3 5 7 10]

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

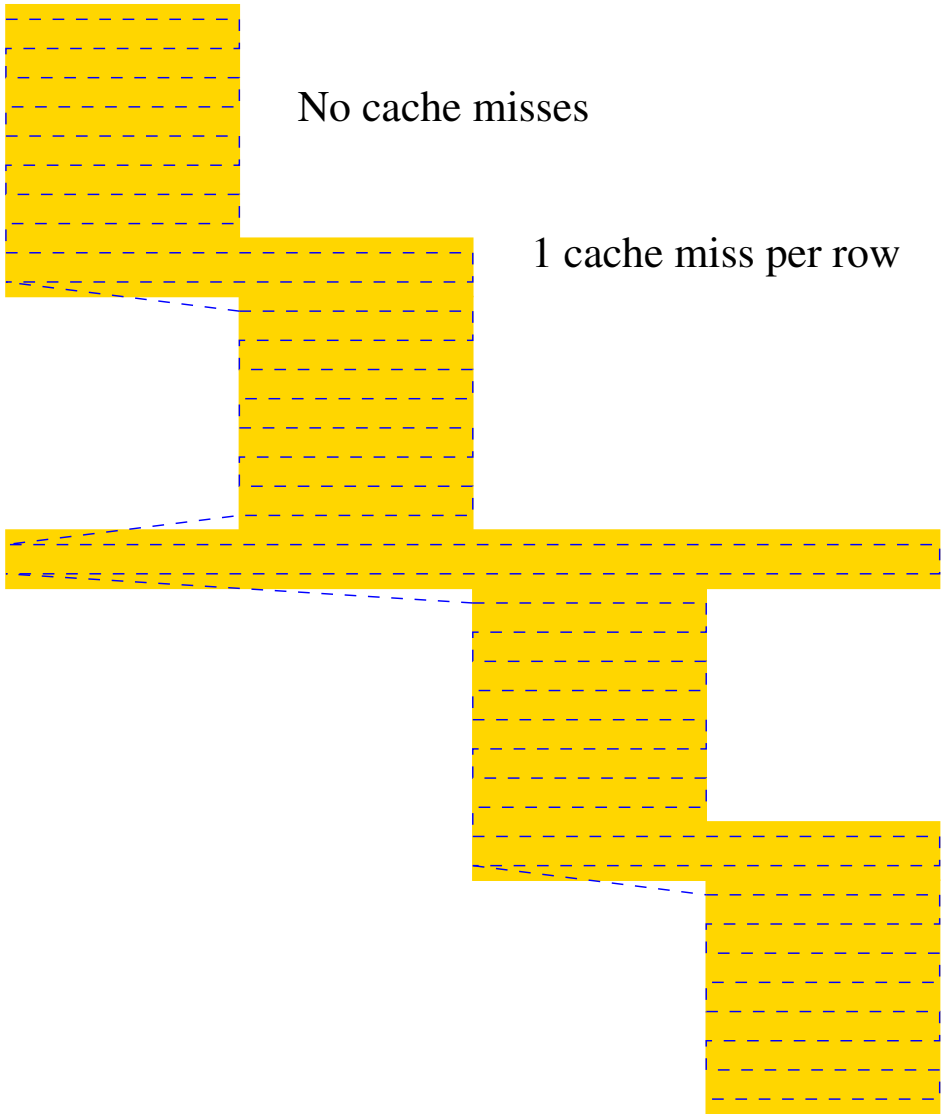
Why not also change the input matrix structure?

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

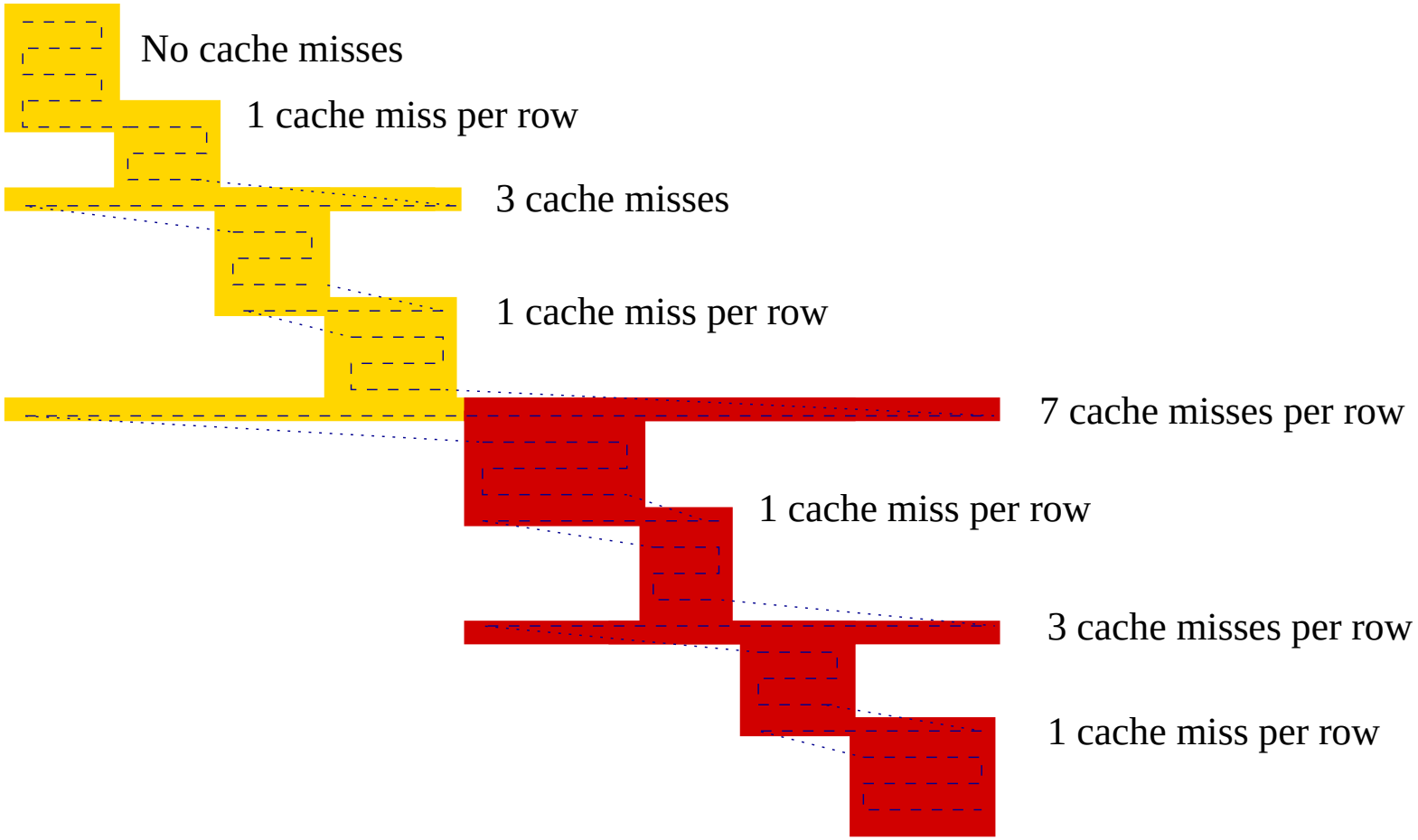
Separated Block Diagonal form



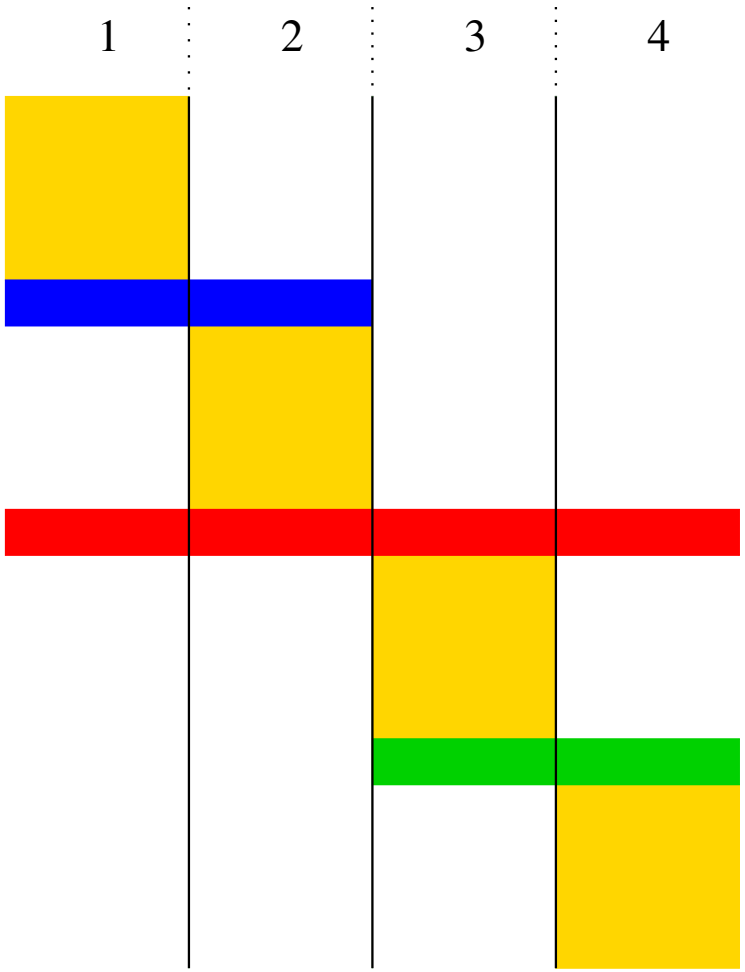
Separated Block Diagonal form

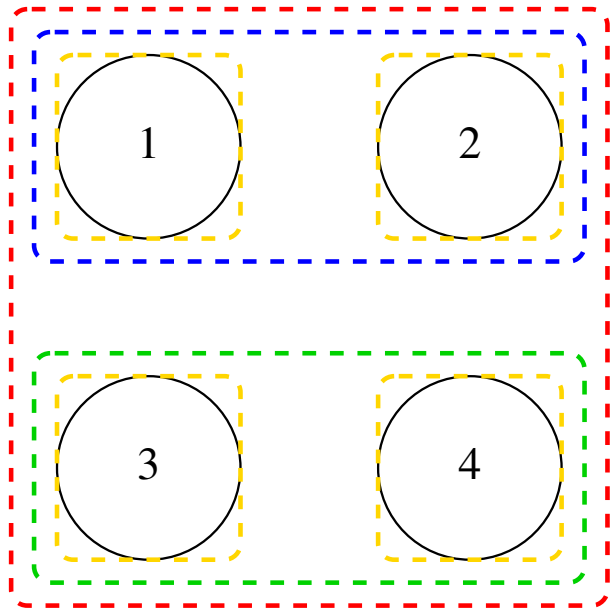


Separated Block Diagonal form



Separated Block Diagonal form





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

Separated Block Diagonal form

In 1D, row and column permutations bring the original matrix A in *Separated Block Diagonal* (SBD) form as follows.

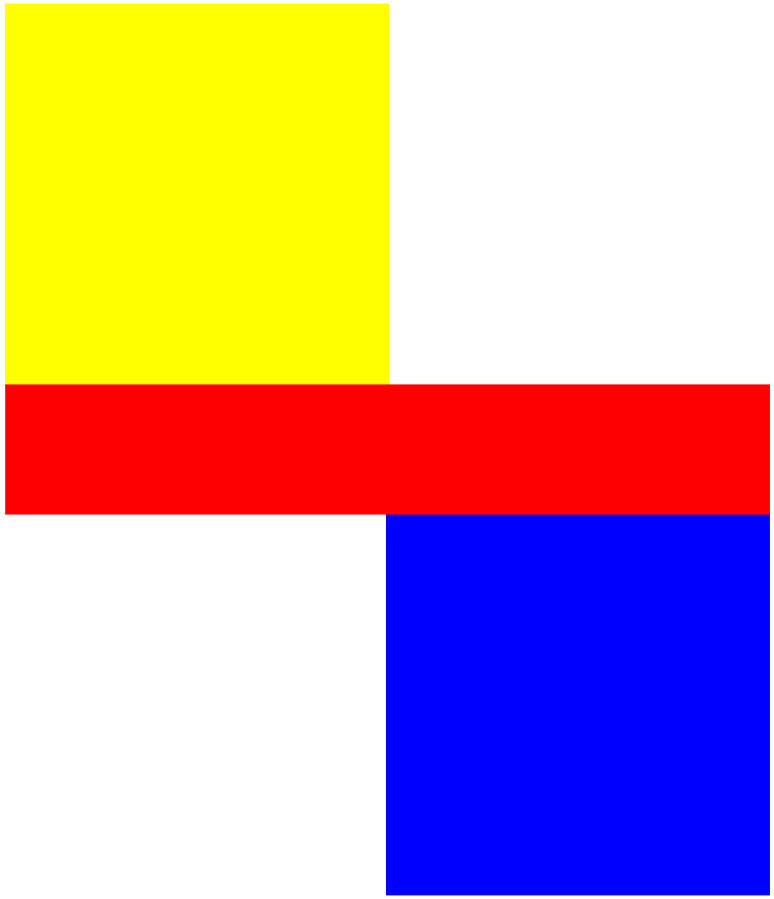
A is modelled as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, with

- $\mathcal{V}$ the set of columns of A,
- $\mathcal{N}$ the set of *hyperedges*, each element is a subset of $\mathcal{V}$ and corresponds to a row of A.

A partitioning $\mathcal{V}_1, \mathcal{V}_2$ of $\mathcal{V}$ can be constructed; and from these, three hyperedge categories can be constructed:

- $\mathcal{N}_-^{\text{row}}$ as the set of hyperedges with vertices only in $\mathcal{V}_1$,
- $\mathcal{N}_c^{\text{row}}$ as the set of hyperedges with vertices both in $\mathcal{V}_1$ and $\mathcal{V}_2$,
- $\mathcal{N}_+^{\text{row}}$ the set of remaining hyperedges.

Separated Block Diagonal form



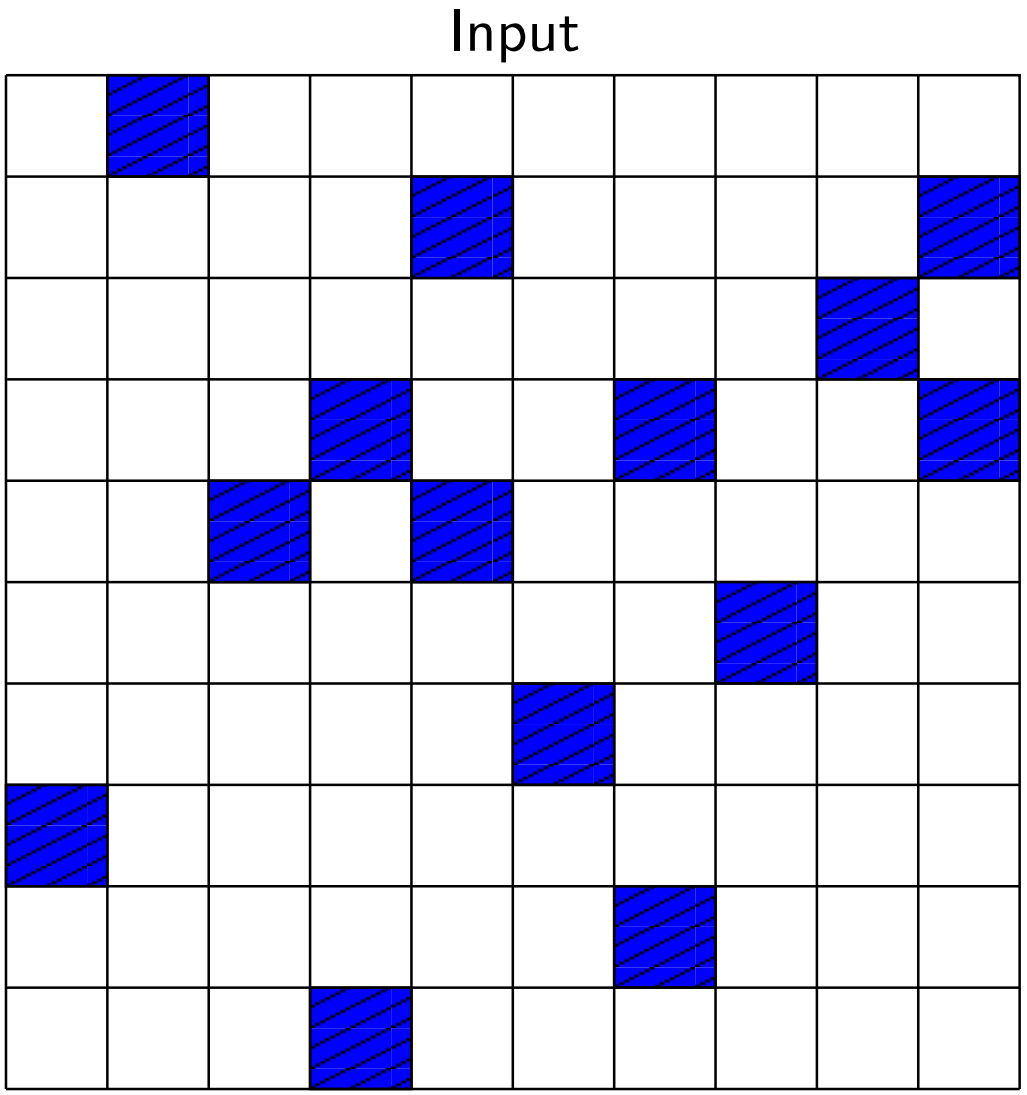
$\mathcal{N}_-^{\text{row}}$

$\mathcal{N}_c^{\text{row}}$

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

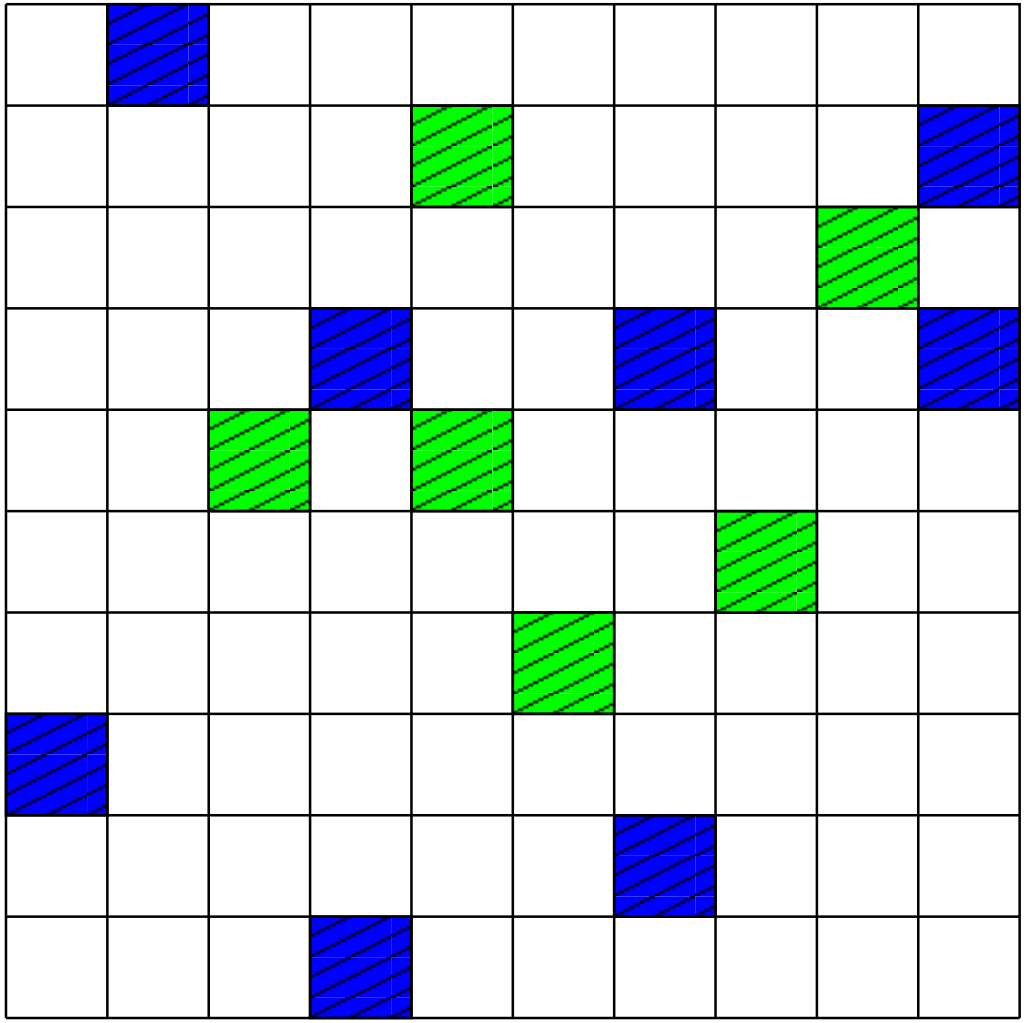
$\mathcal{V}_1$ $\mathcal{V}_2$

Permuting to SBD form



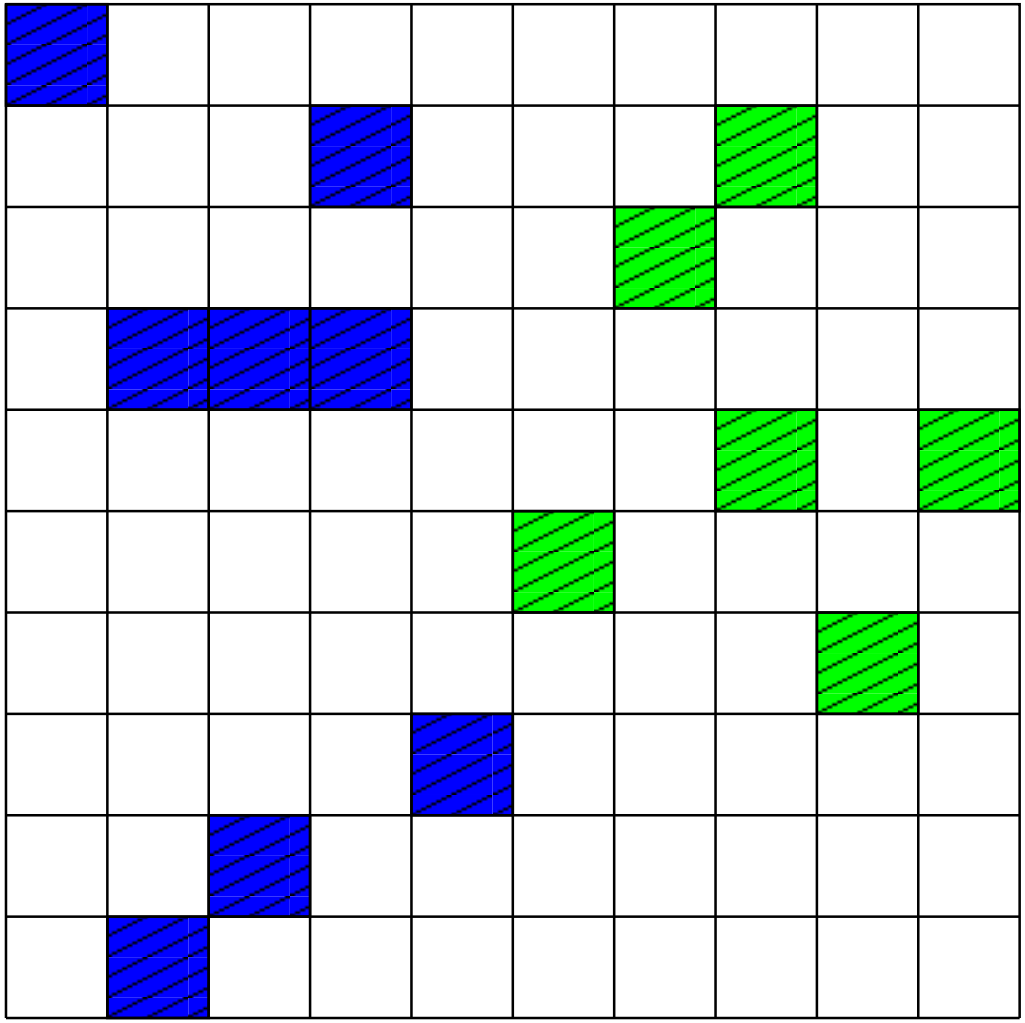
Permuting to SBD form

Column partitioning



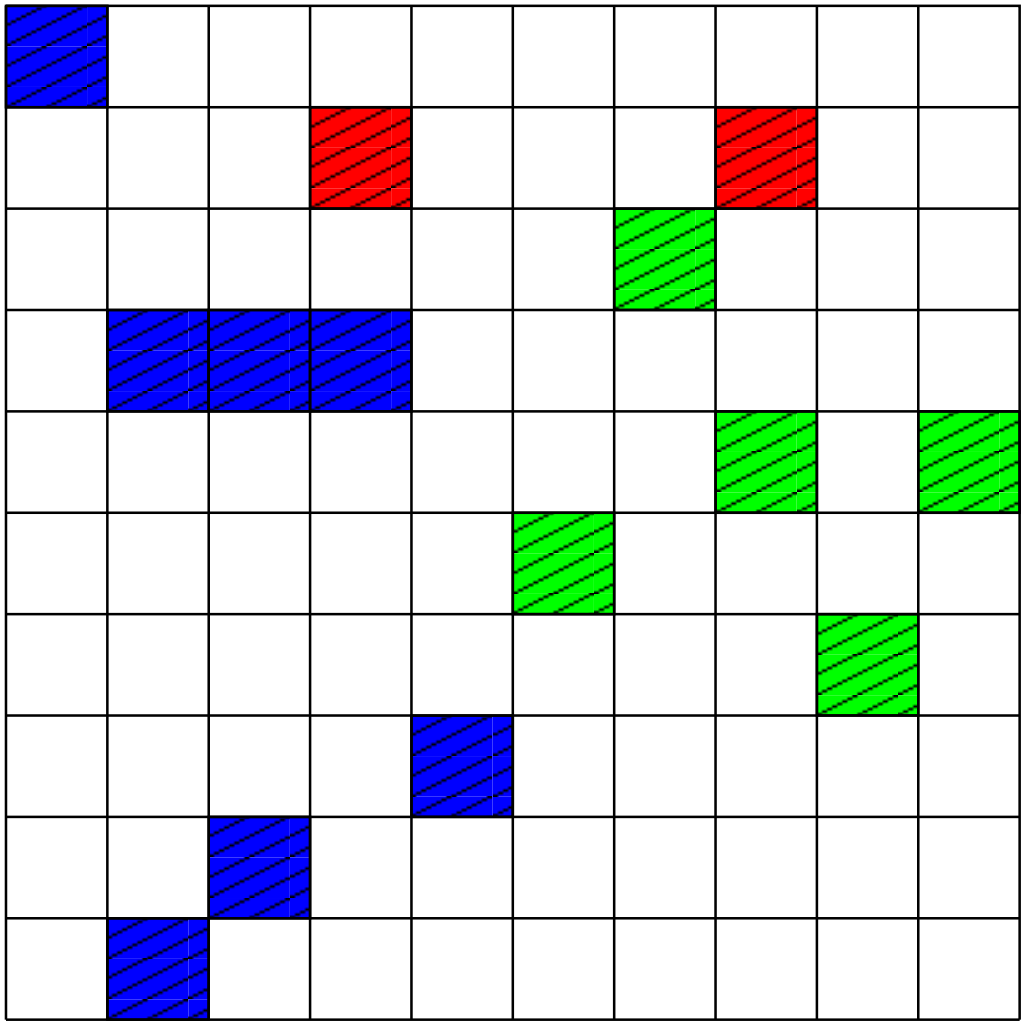
Permuting to SBD form

Column permutation



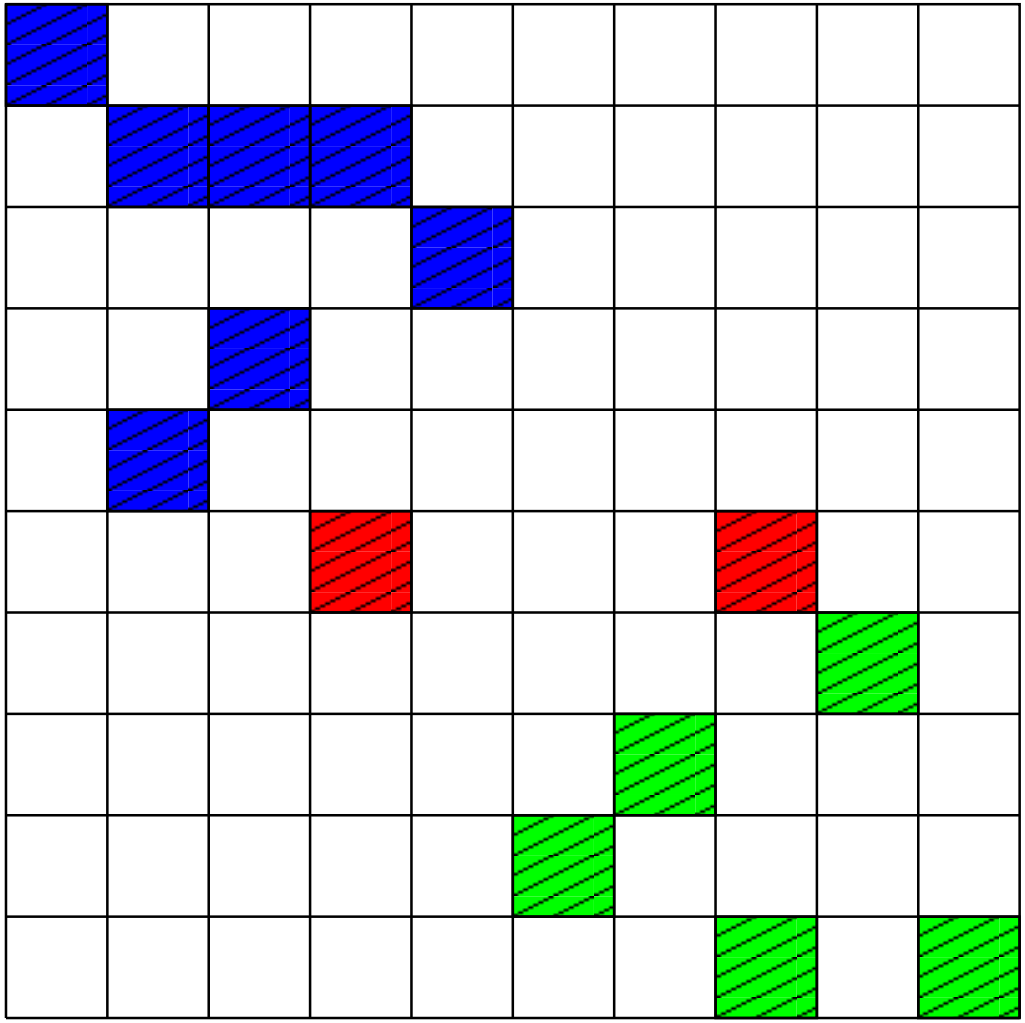
Permuting to SBD form

Mixed row detection

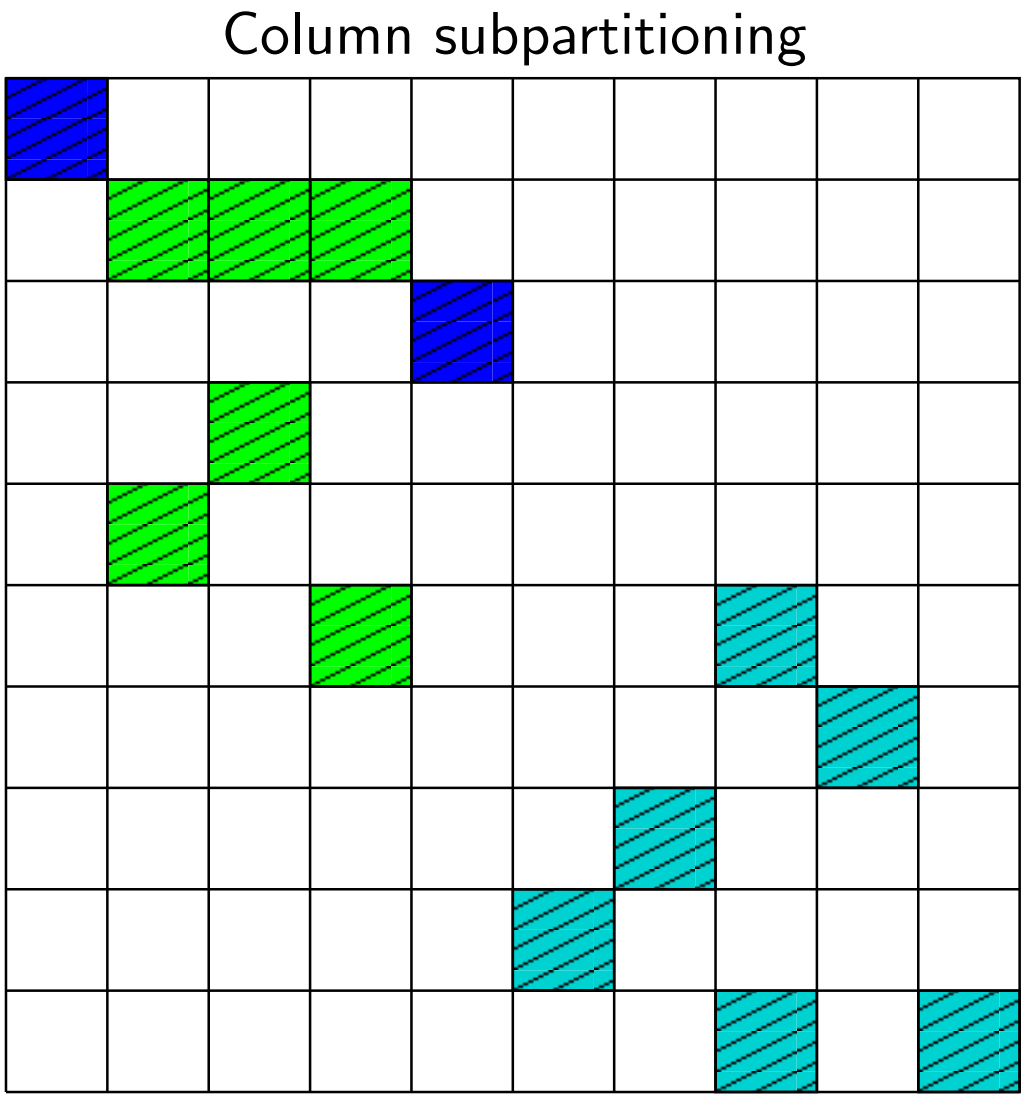


Permuting to SBD form

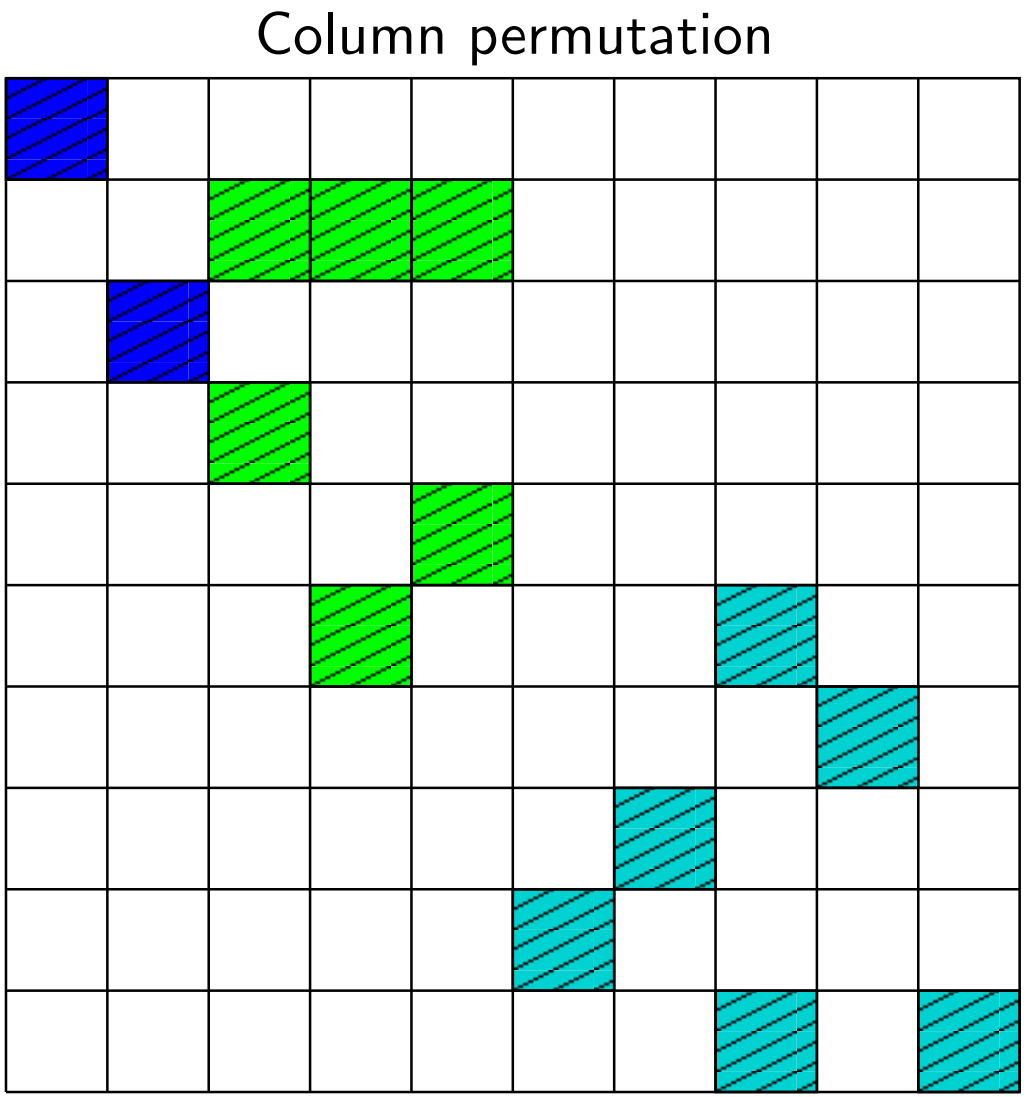
Row permutation



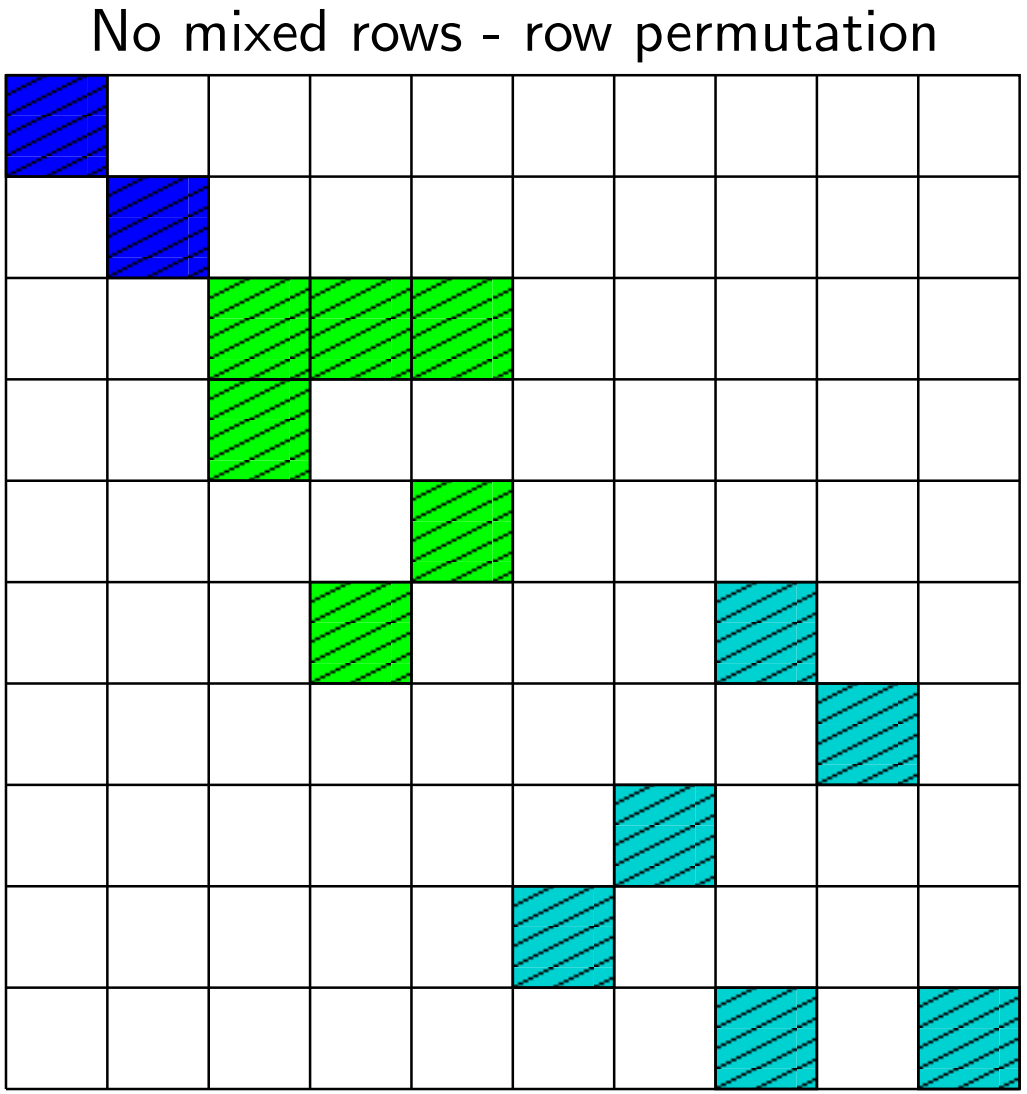
Permuting to SBD form



Permuting to SBD form

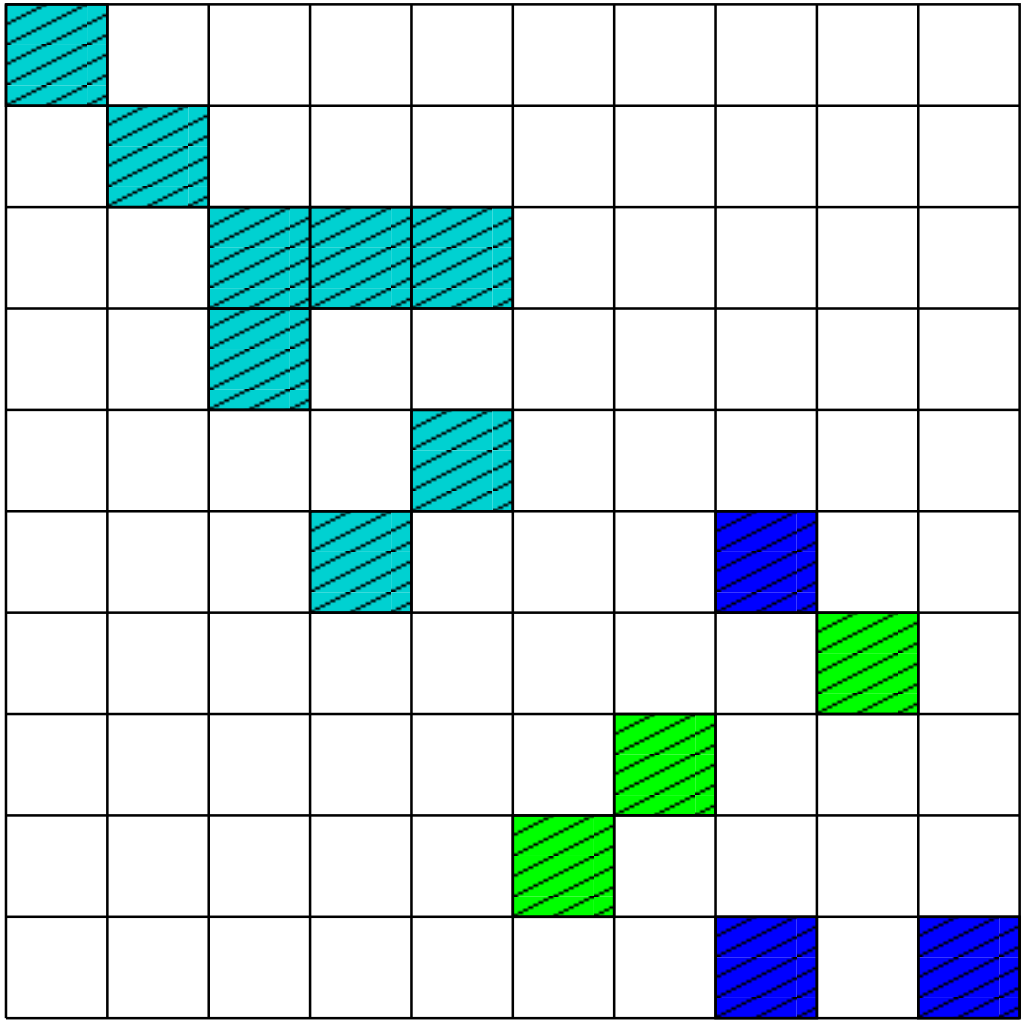


Permuting to SBD form



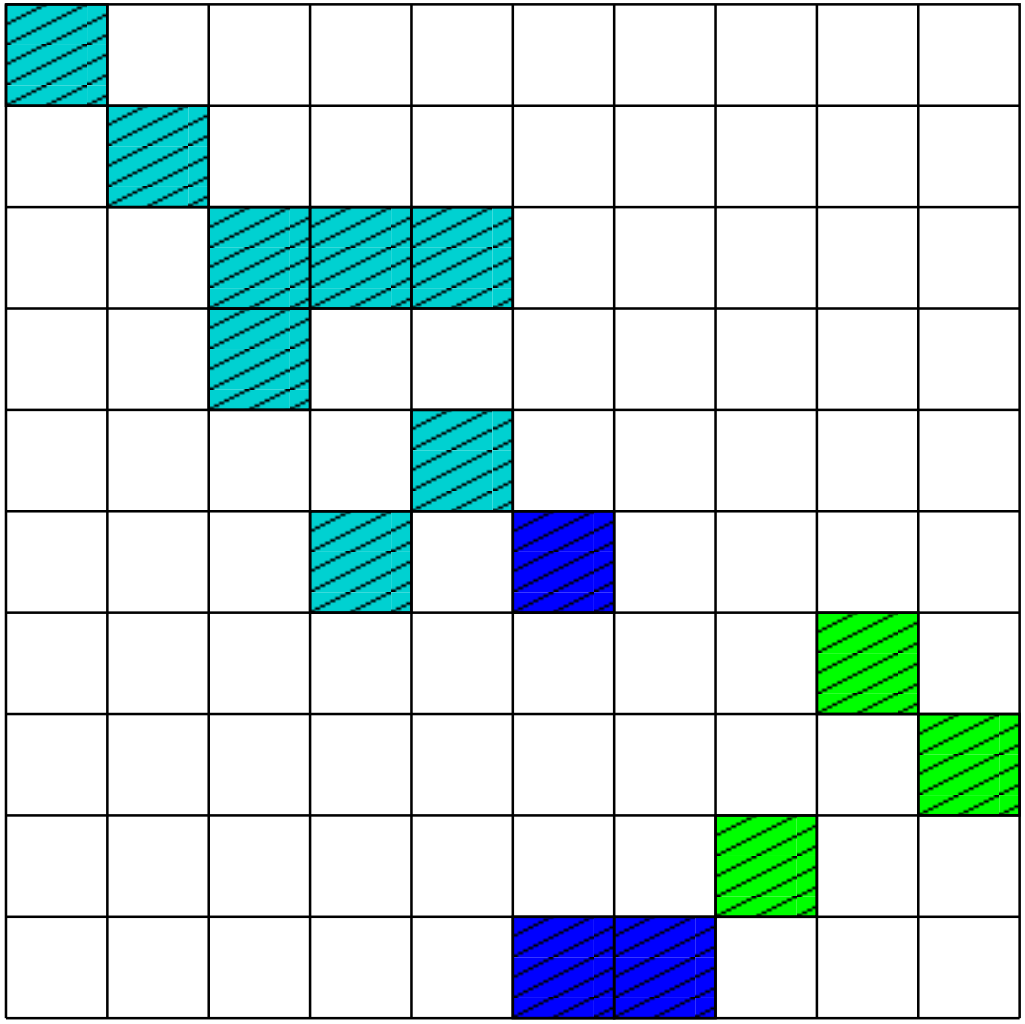
Permuting to SBD form

Continued



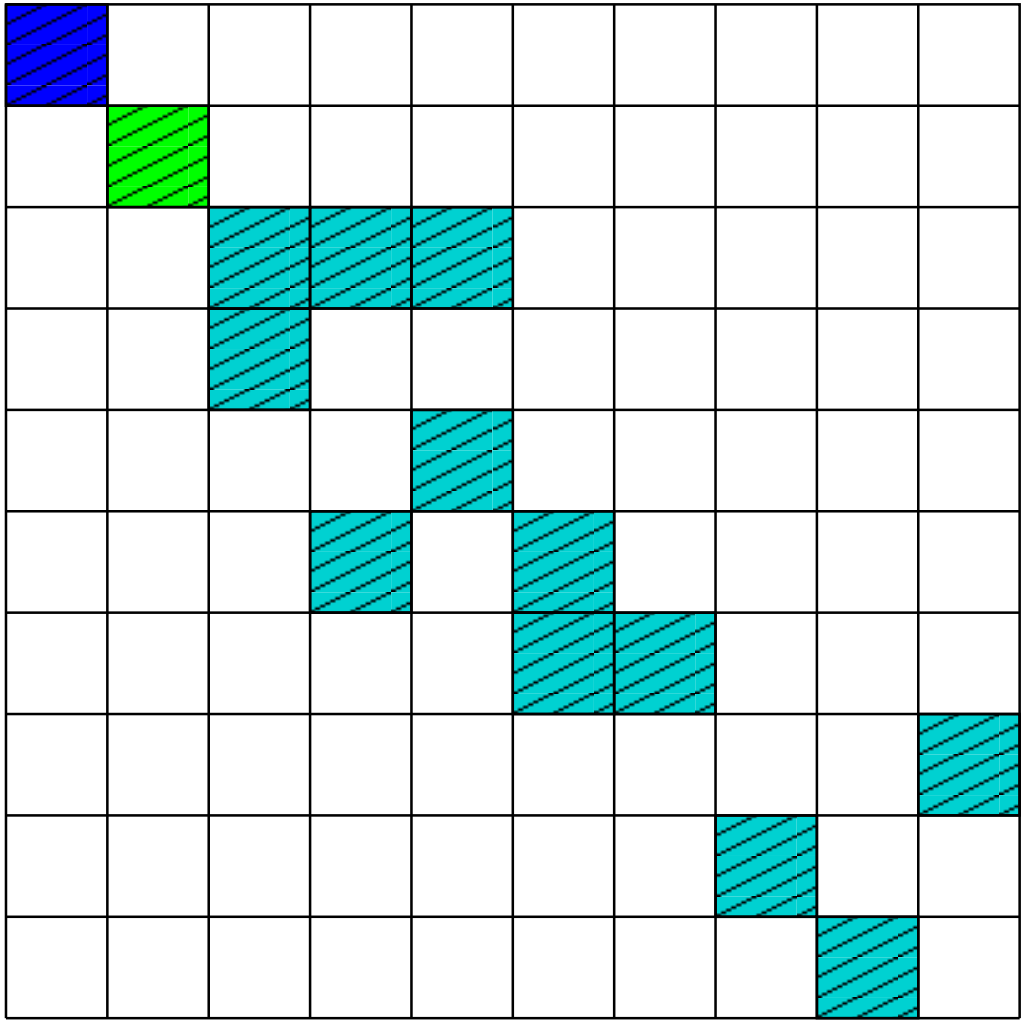
Permuting to SBD form

Continued



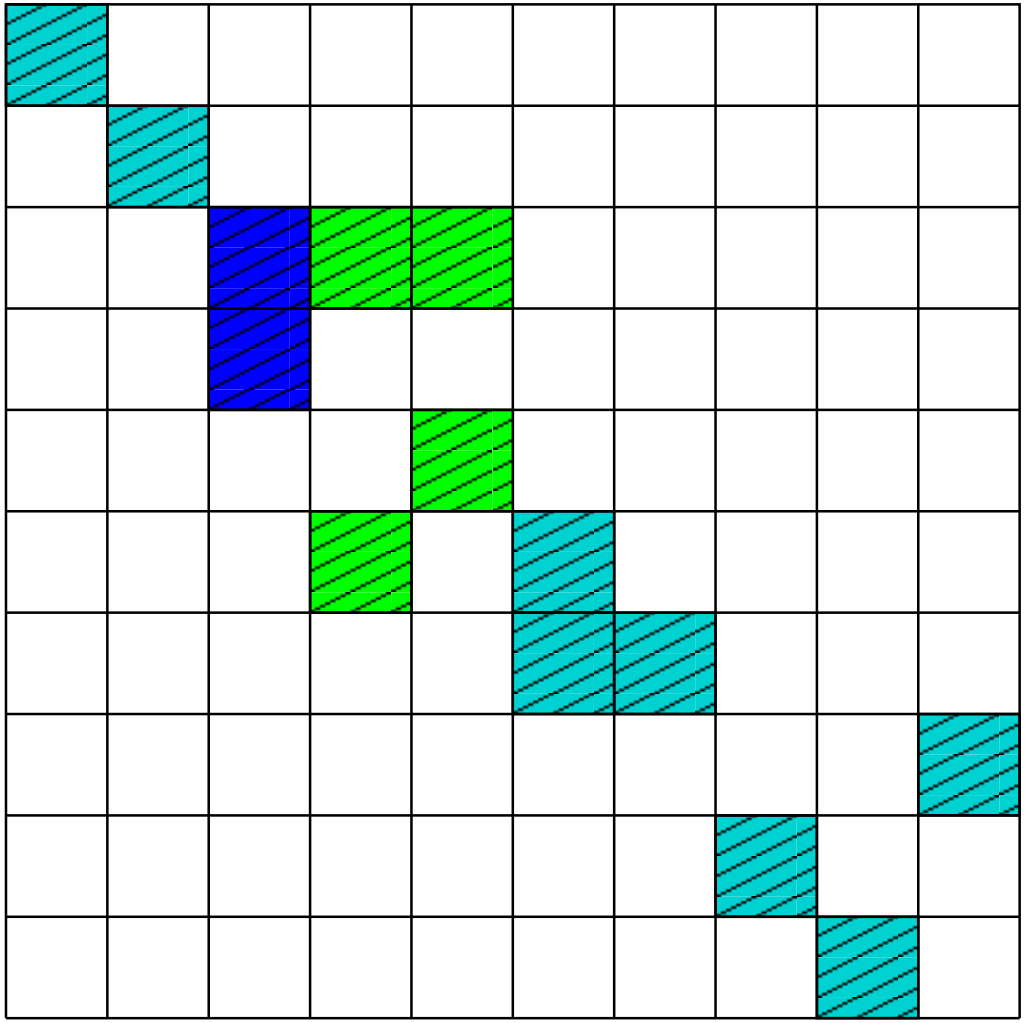
Permuting to SBD form

Continued



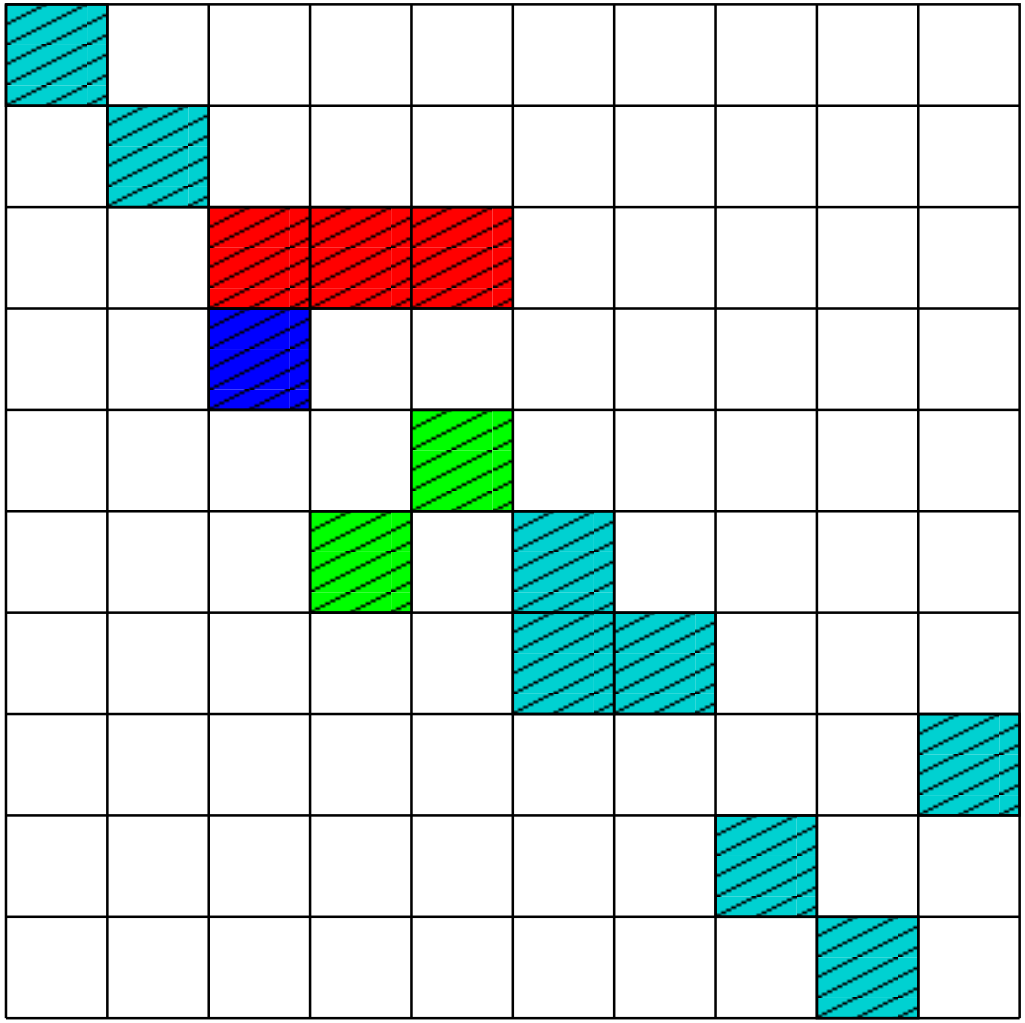
Permuting to SBD form

Continued



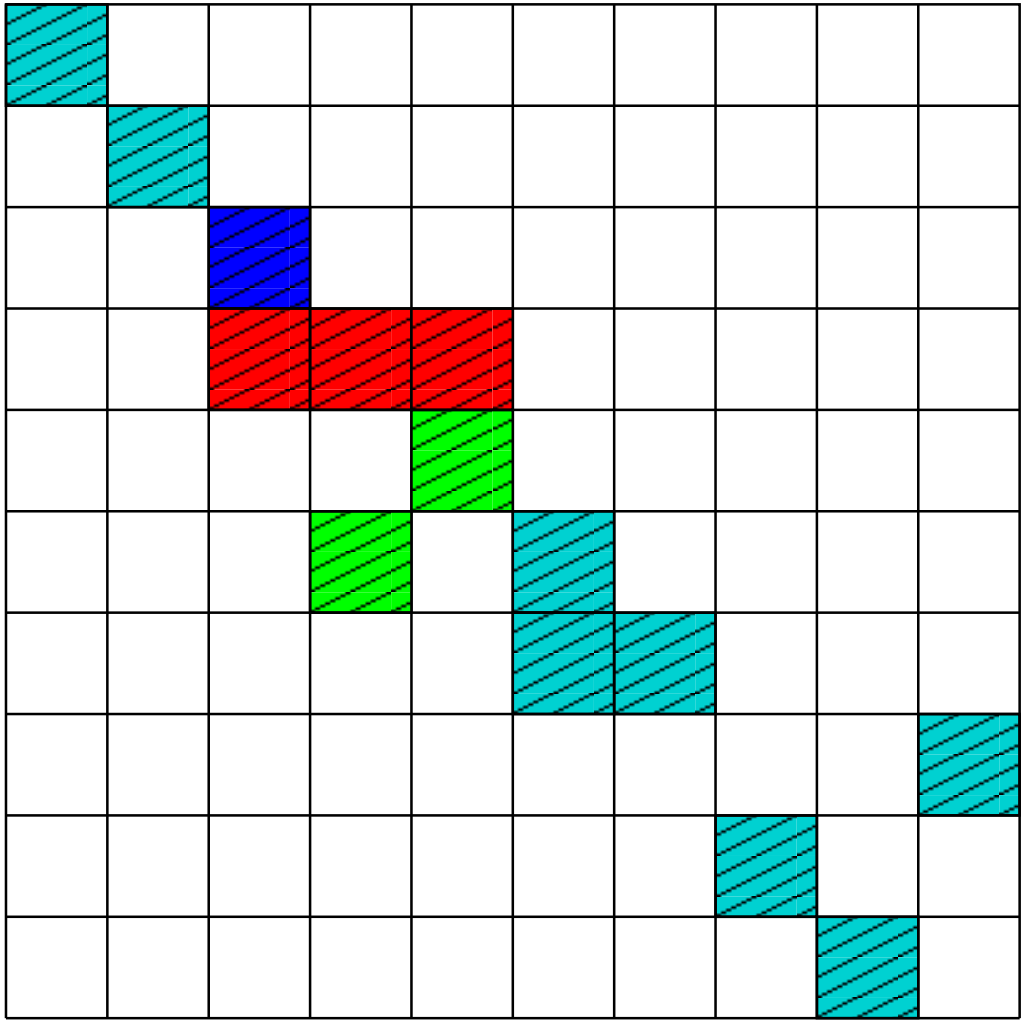
Permuting to SBD form

Continued



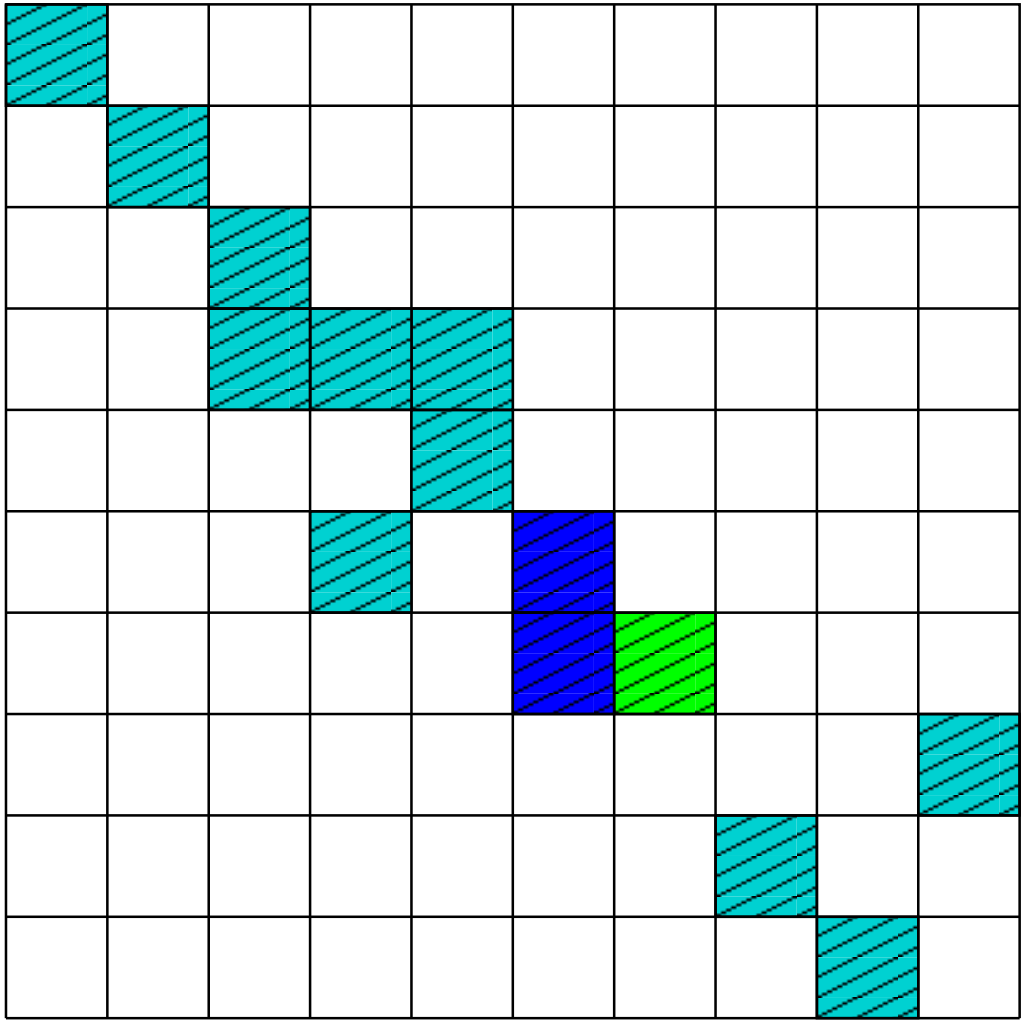
Permuting to SBD form

Continued



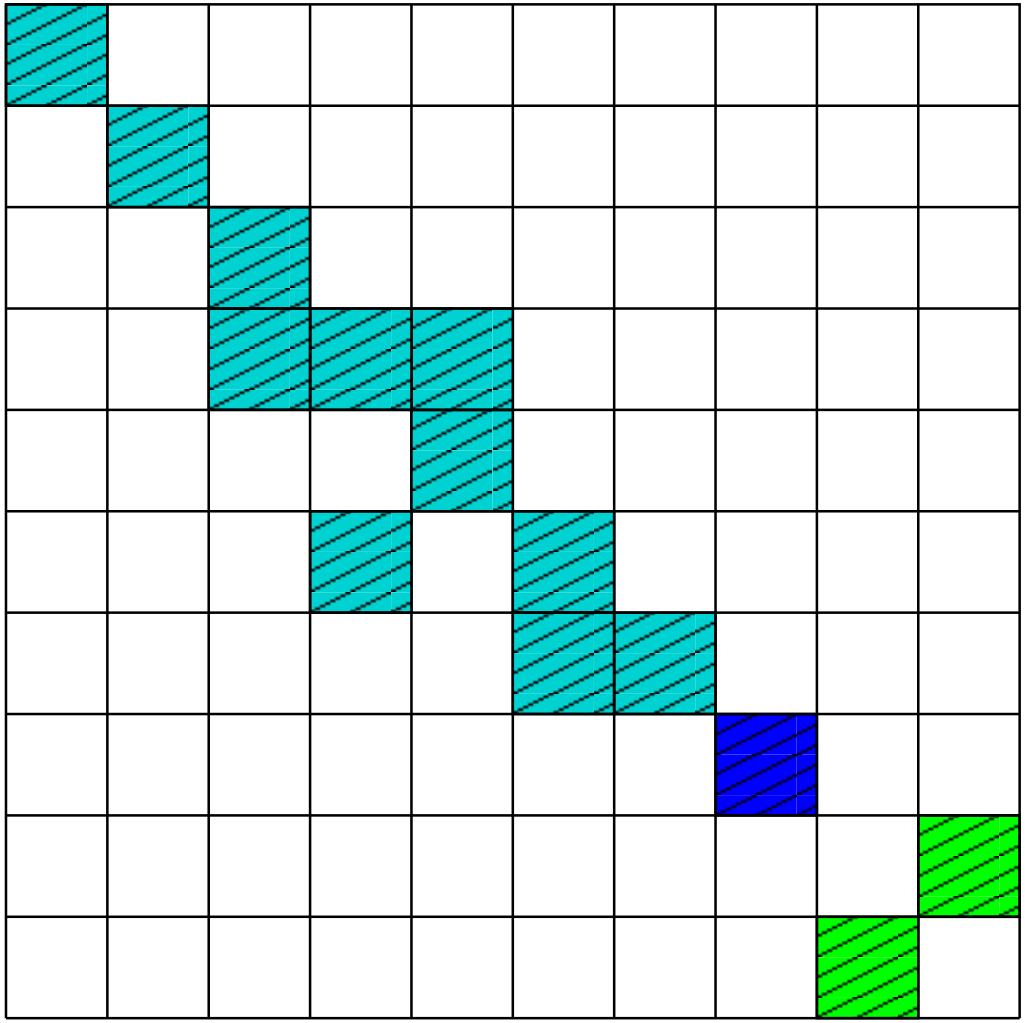
Permuting to SBD form

Continued



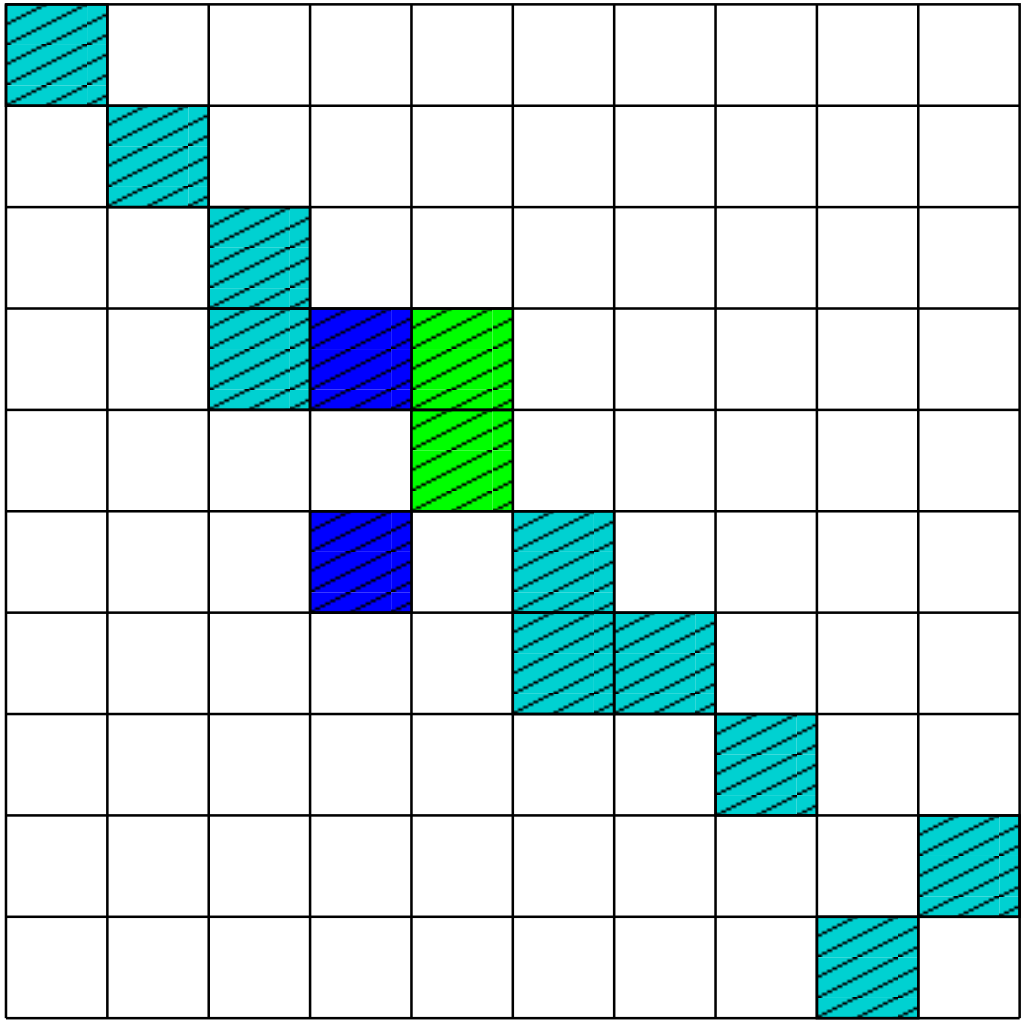
Permuting to SBD form

Continued



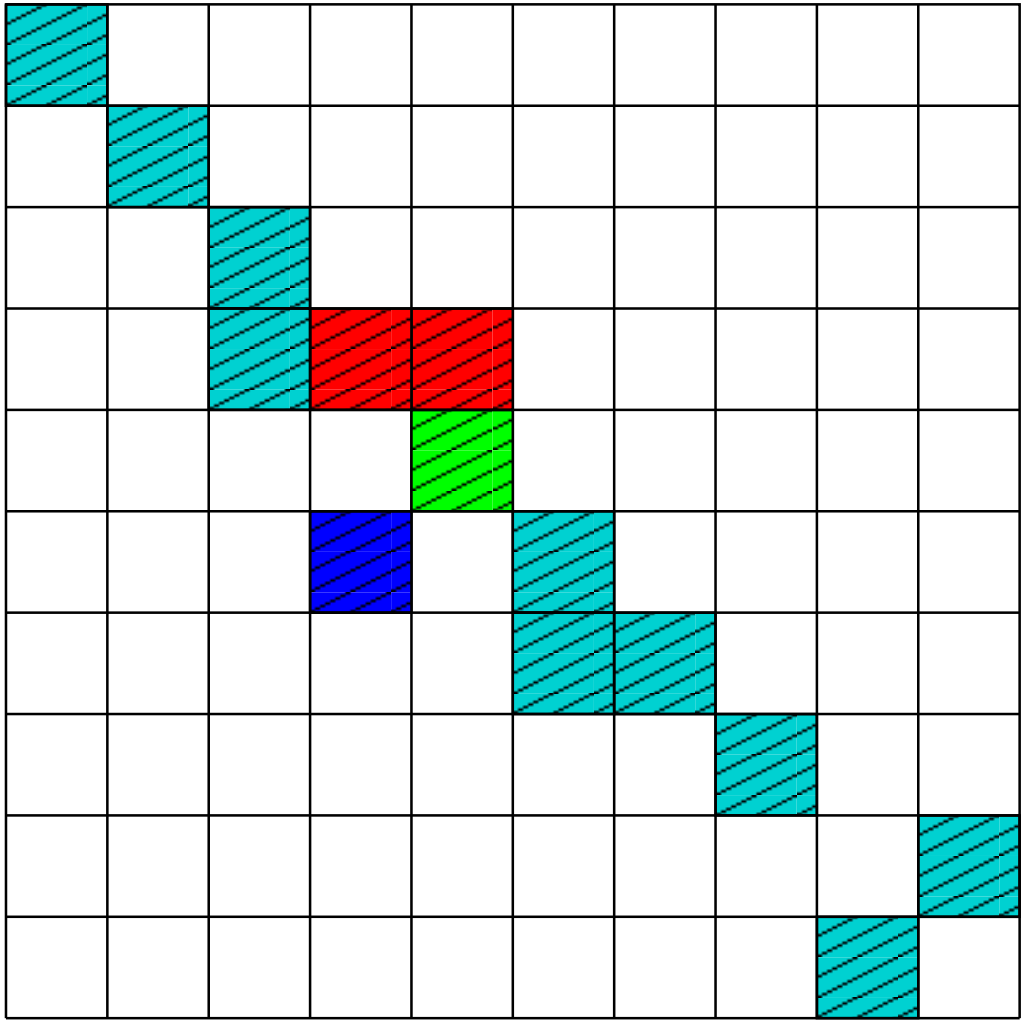
Permuting to SBD form

Continued



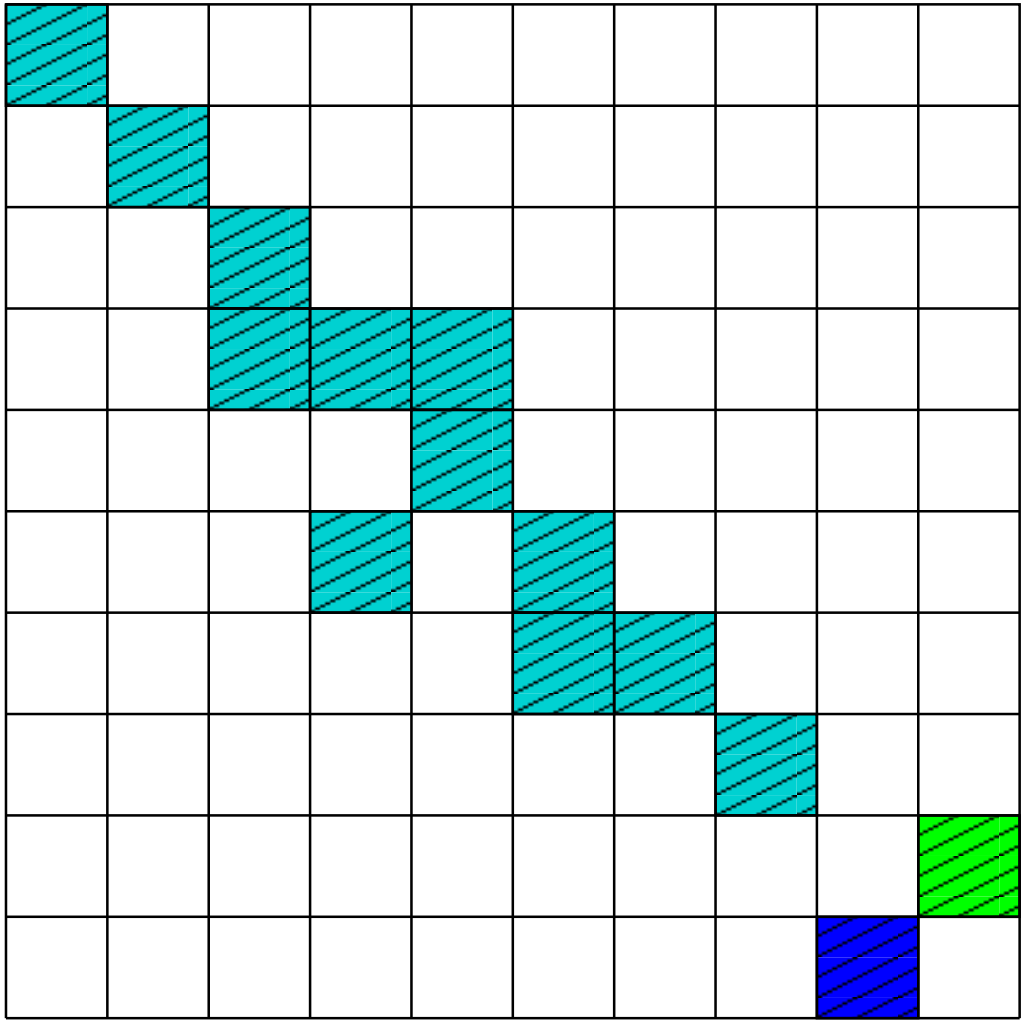
Permuting to SBD form

Continued



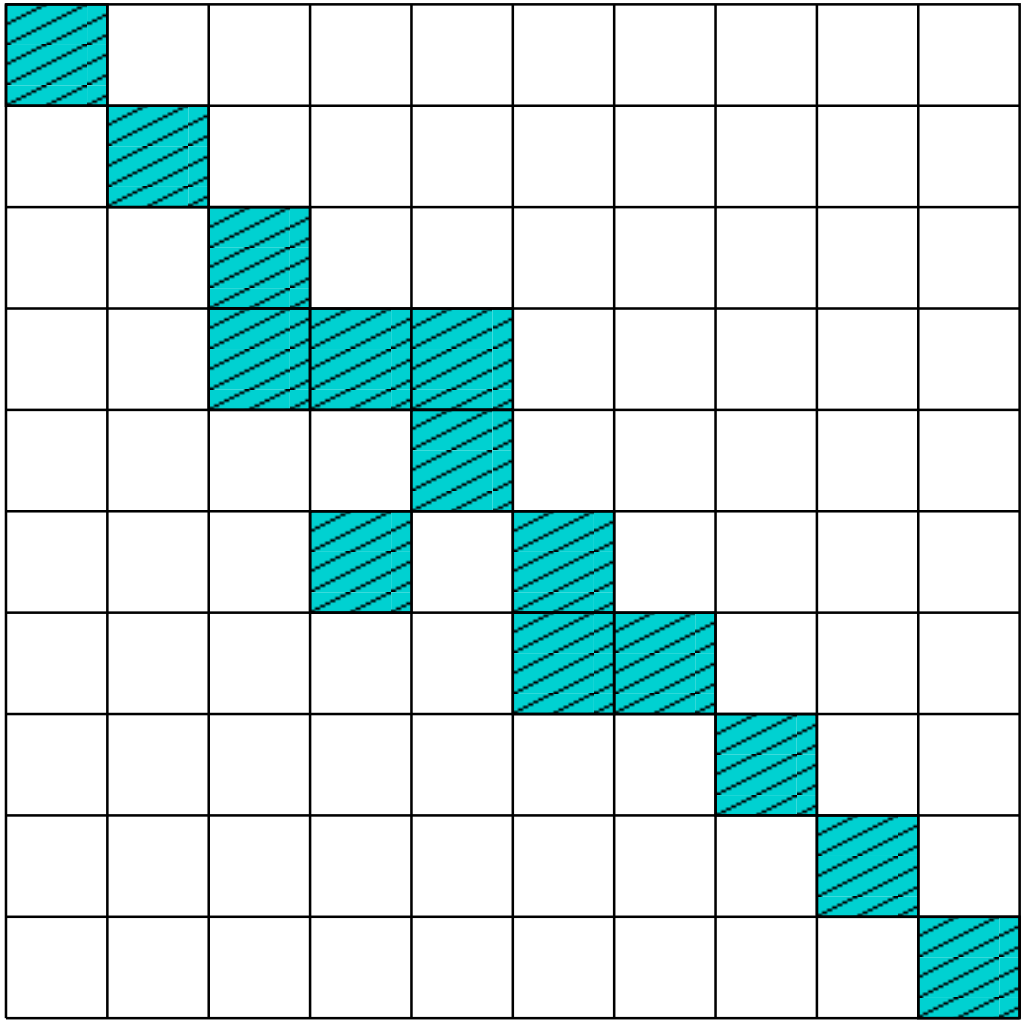
Permuting to SBD form

Continued



Permuting to SBD form

Continued



Reordering parameters

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

the number of cache misses is strictly bounded by

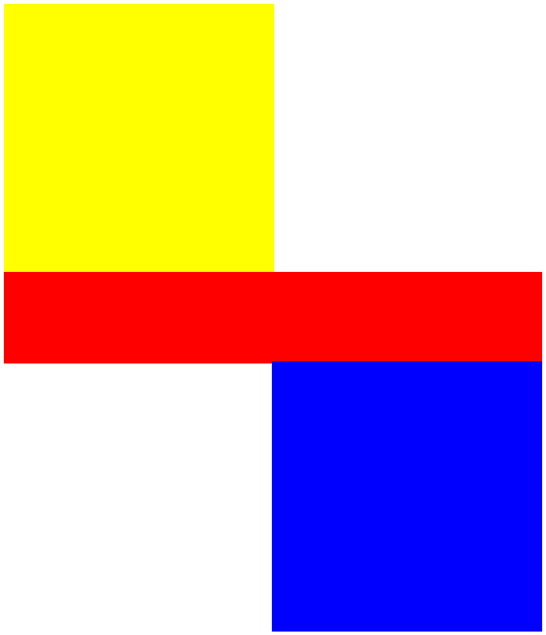
$$\sum_{i:\ n_i \in \mathcal{N}} (\lambda_i - 1);$$

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

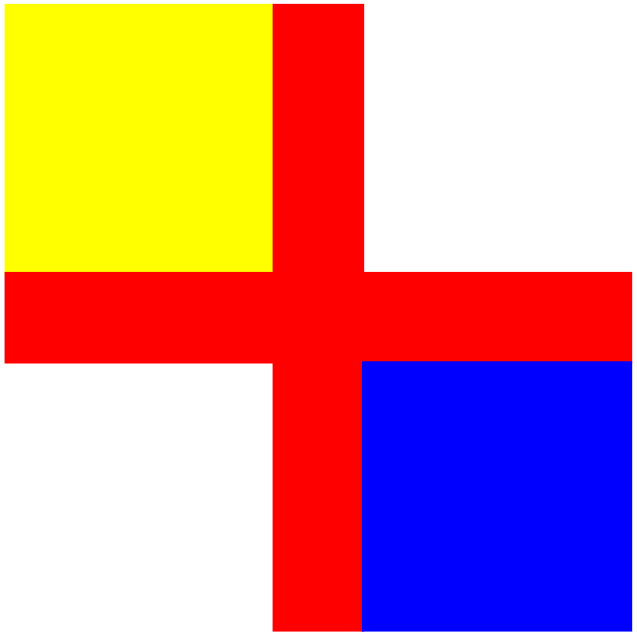
References:

- 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

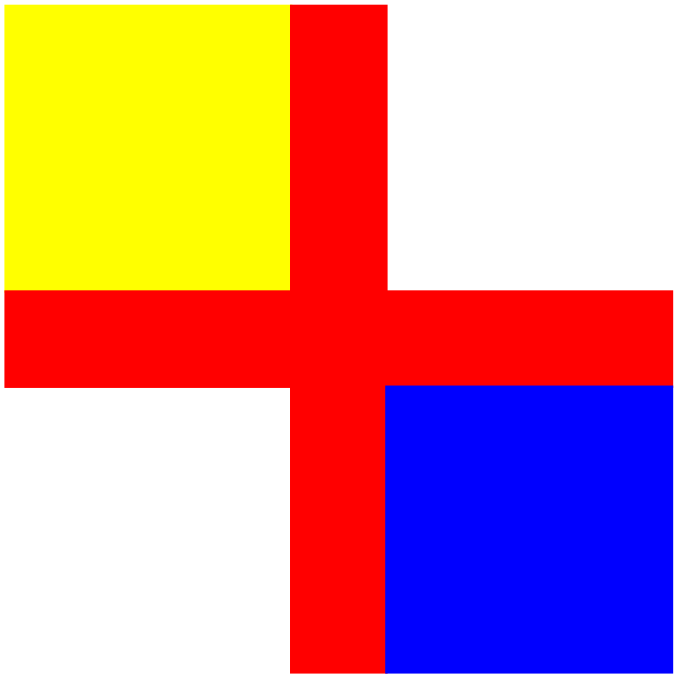


2D

Yzelman and Bisseling, *Two-dimensional cache-oblivious sparse matrix–vector multiplication*, April 2011 (Revised pre-print); `http://www.math.uu.nl/people/yzelman/publications/#pp`

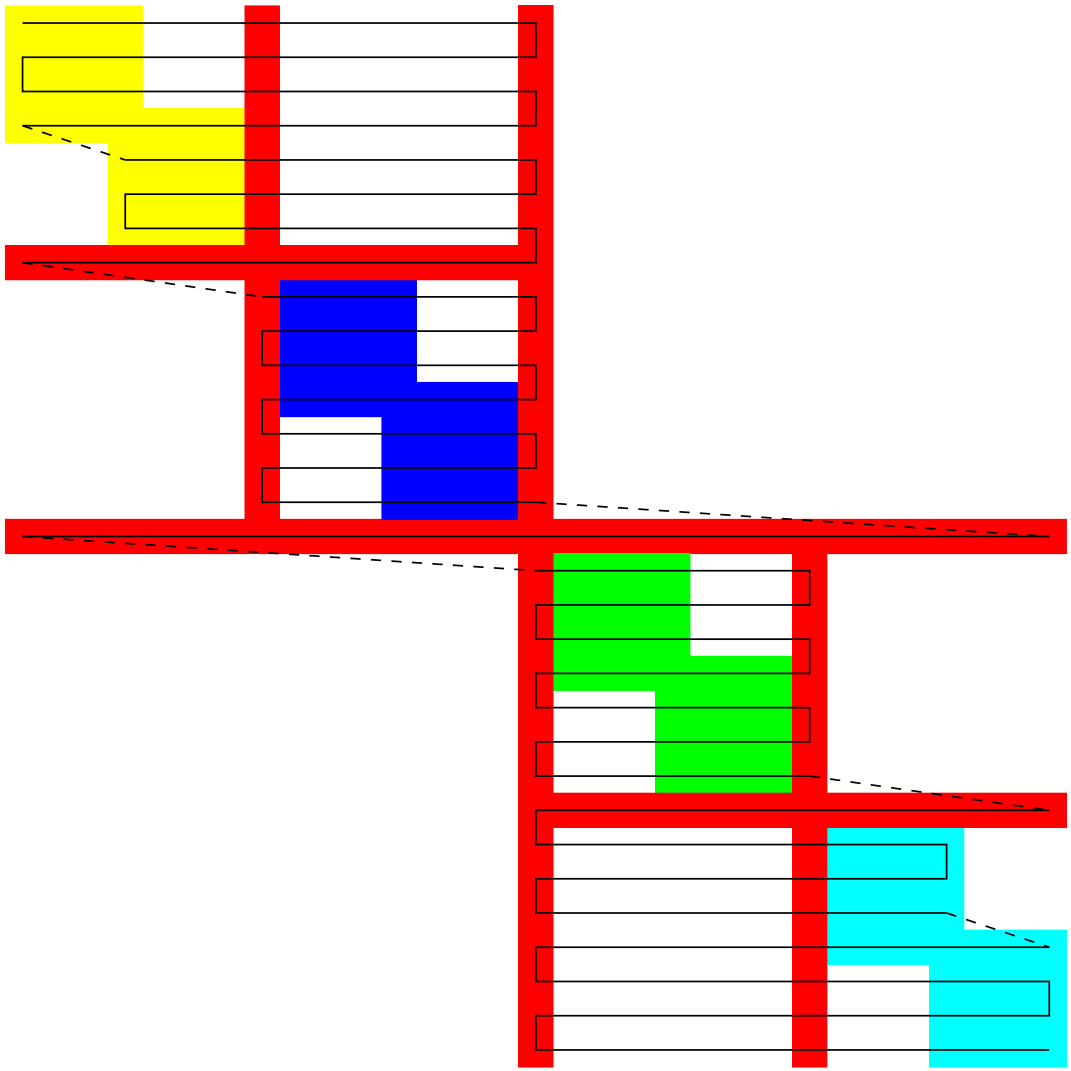
Two-dimensional SBD (doubly separated block diagonal)

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



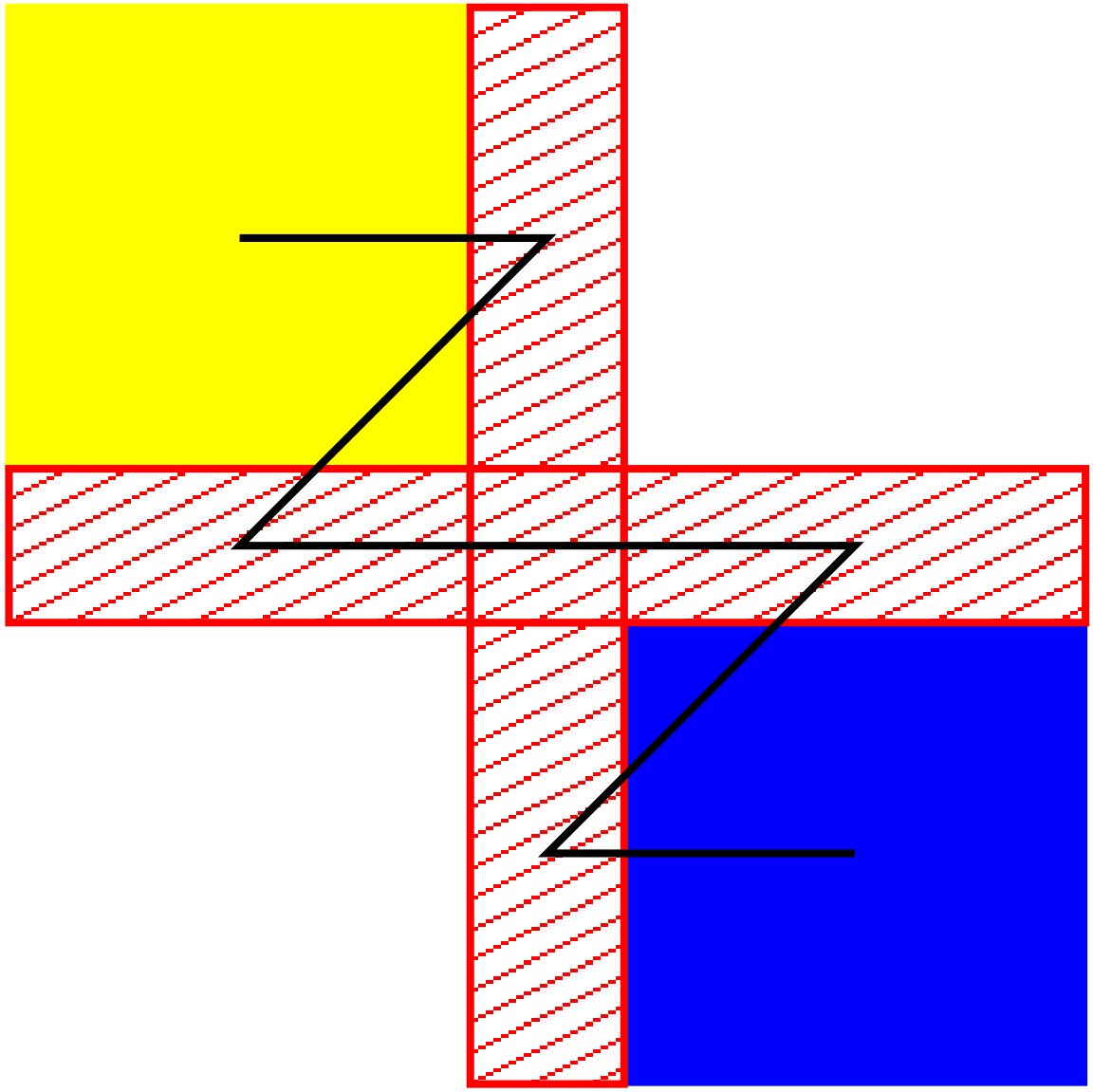
The quantity minimised remains $\sum_i(\lambda_i - 1)$.

Two-dimensional SBD (doubly separated block diagonal)

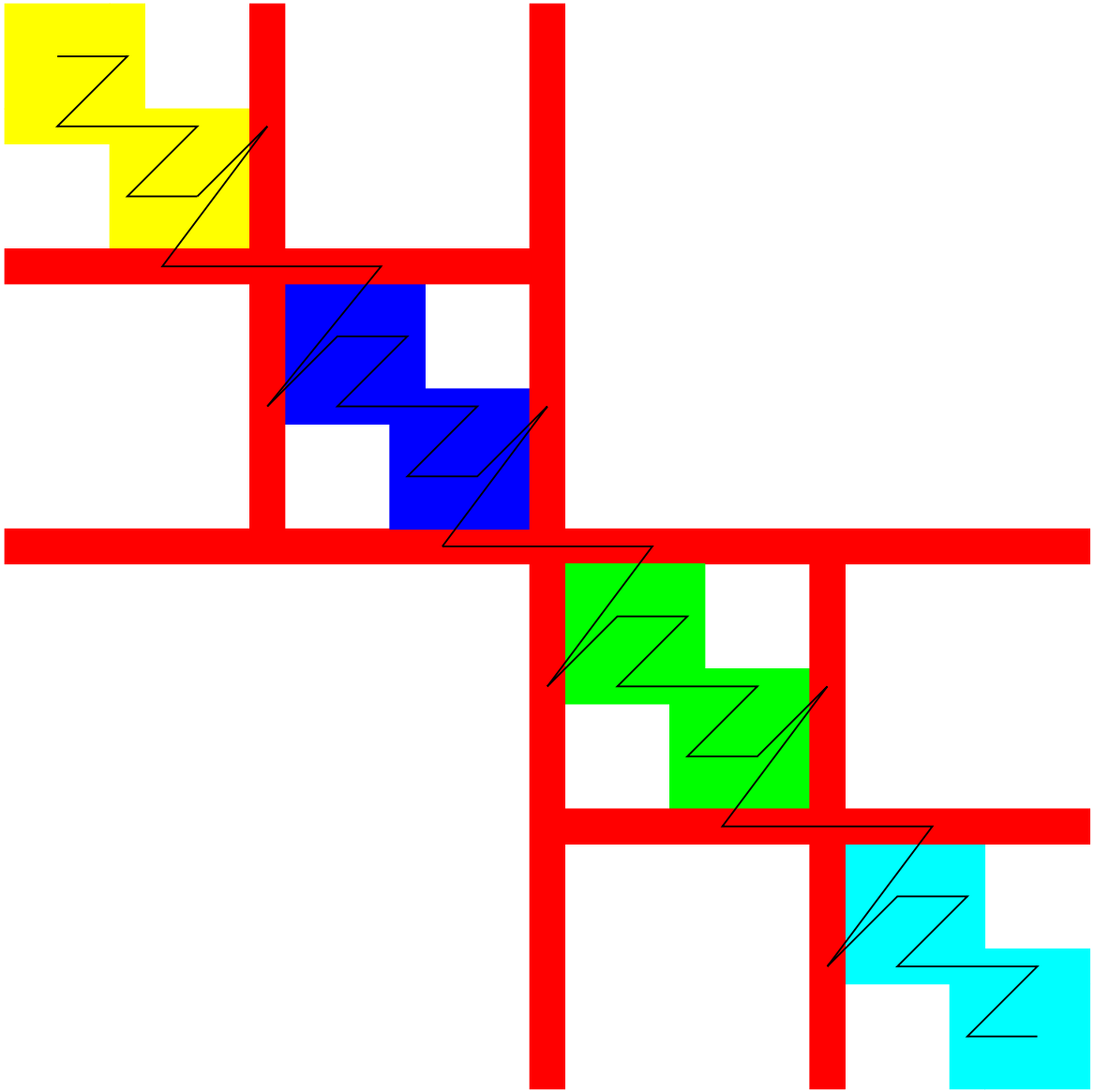


Zig-zag CRS is not suitable for handling 2D SBD!

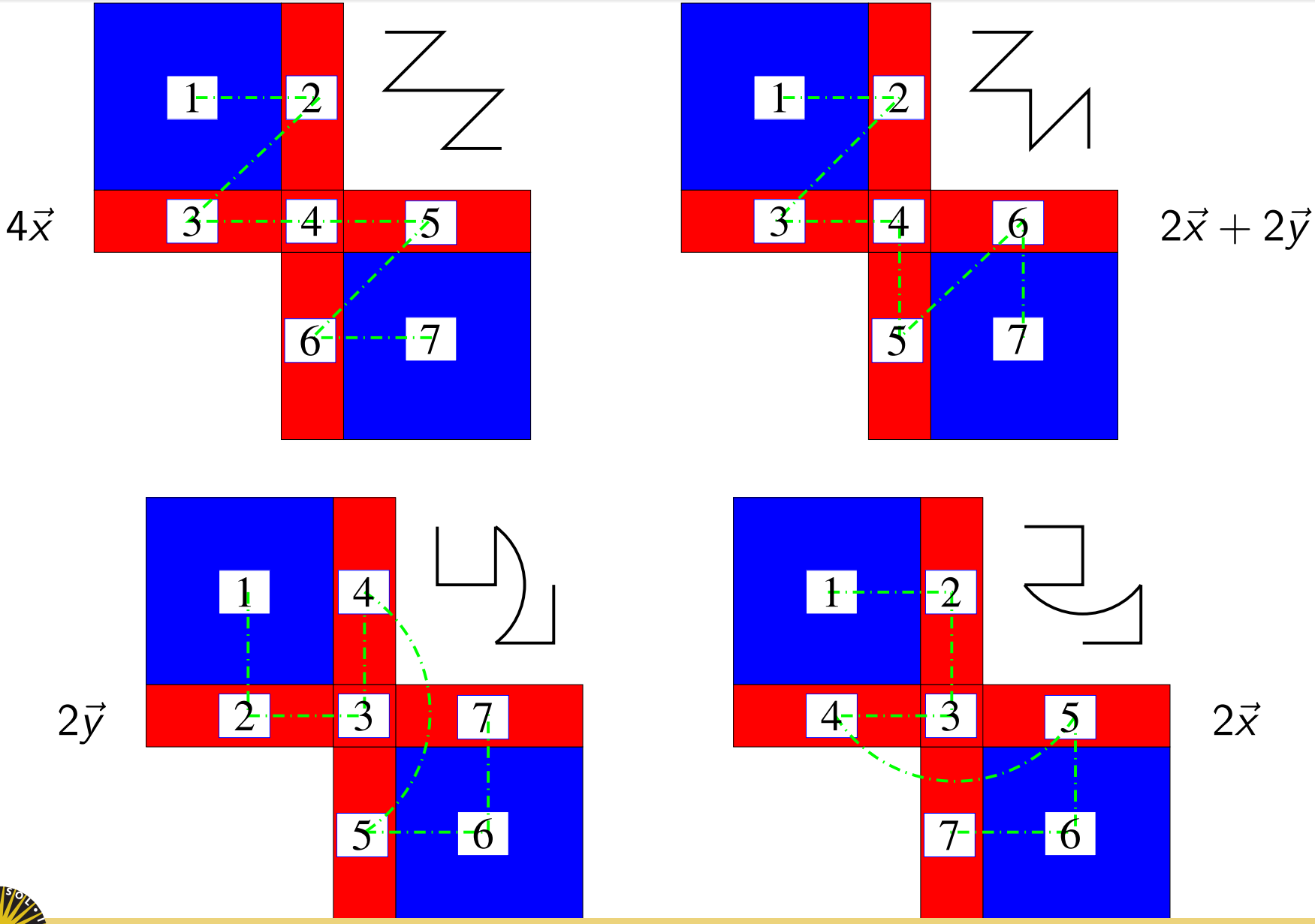
Two-dimensional SBD



Two-dimensional SBD

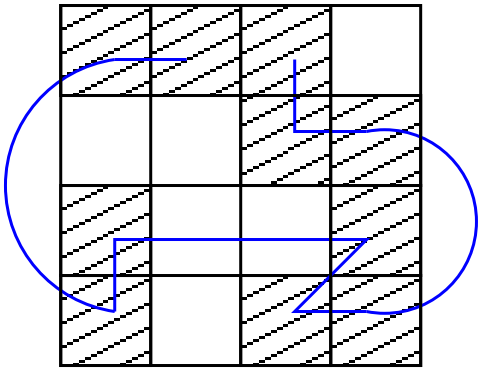


Two-dimensional SBD; block ordering



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:

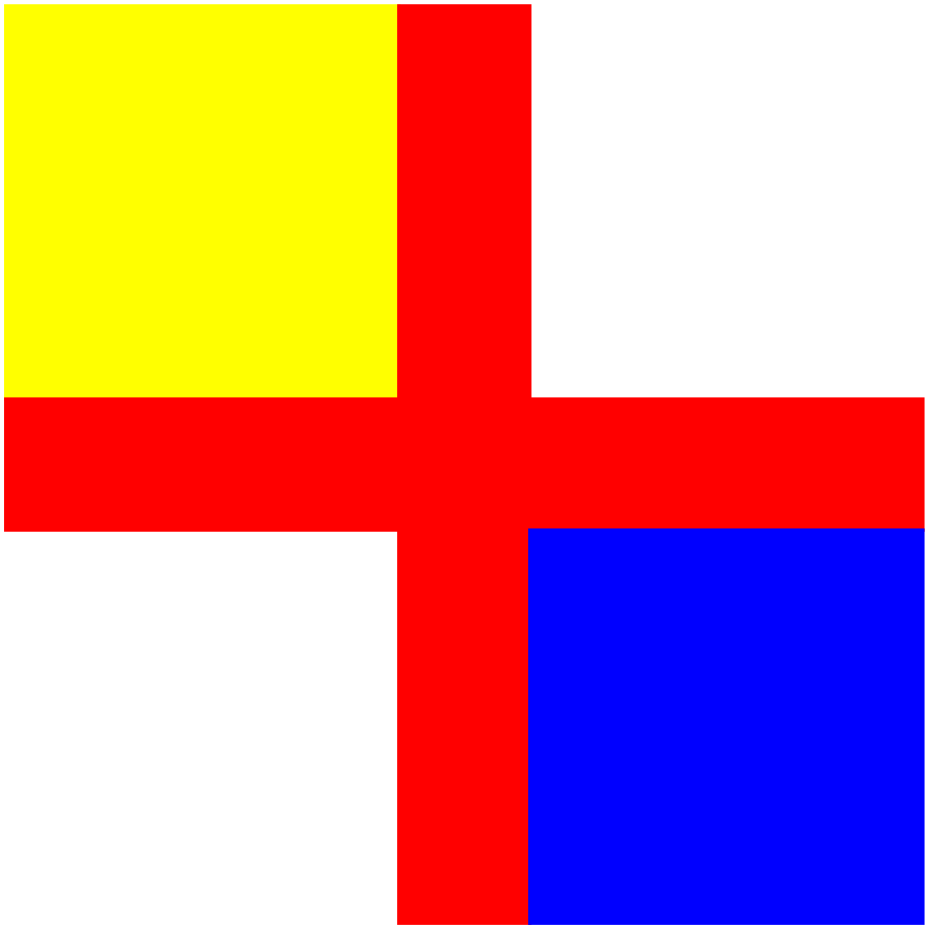
nzs:	[3 2 3 1 1 2 1 7 4 1]
col_increment:	[2 4 1 4 -1 5 -3 4 4 1] ,
row_increment:	[0 1 2 -1 1 -3]

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

Parallel cache-friendly SpMV

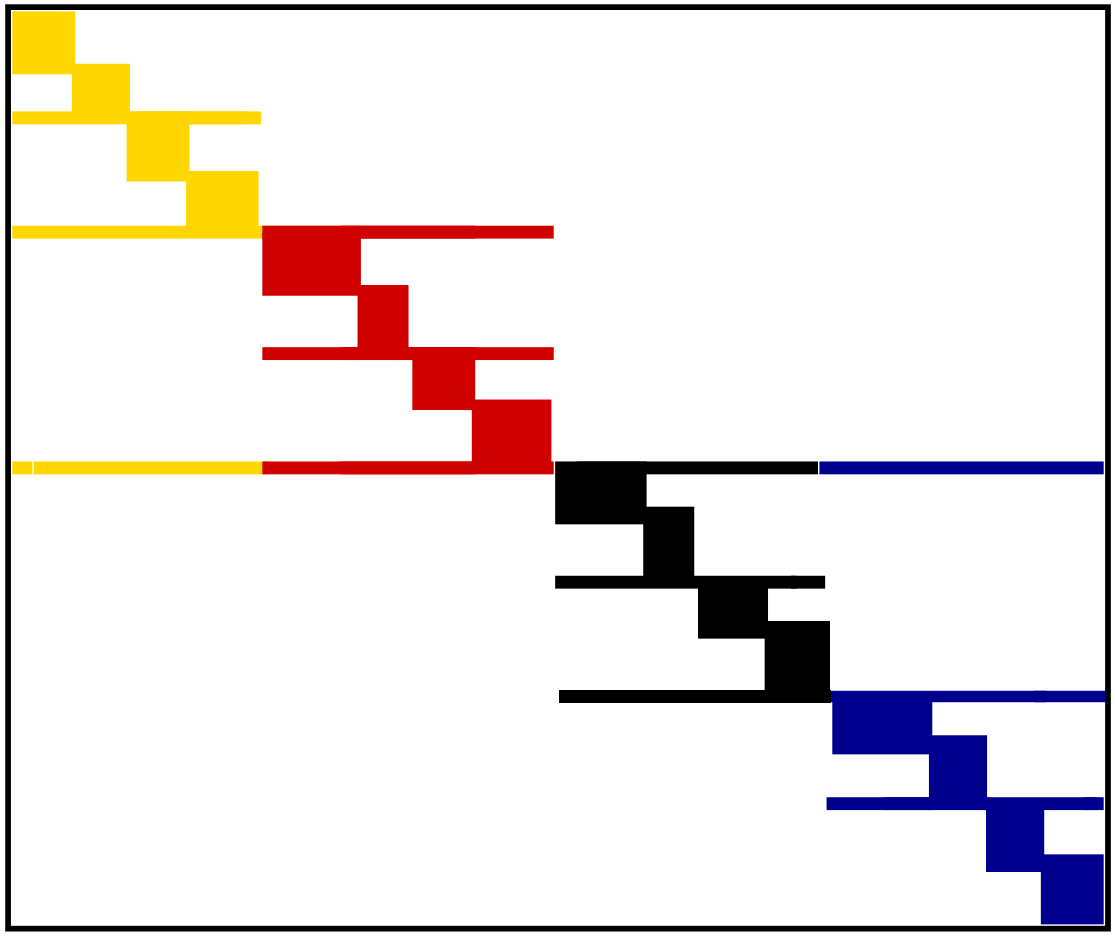
On distributed-memory architectures

Directly use partitioner output:



On distributed-memory architectures

Or: use both partitioner and reordering output: partition for $p \to \infty$, but distribute only over the actual number of processors:



On shared-memory architectures

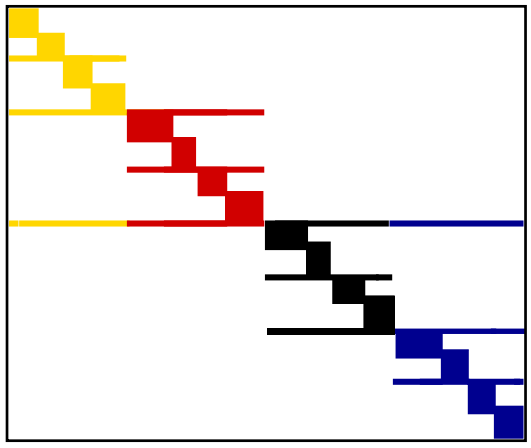
Use:

- global version of the matrix A, stored in BICRS,
- global input vector x,
- global output vector y.

On shared-memory architectures

Use:

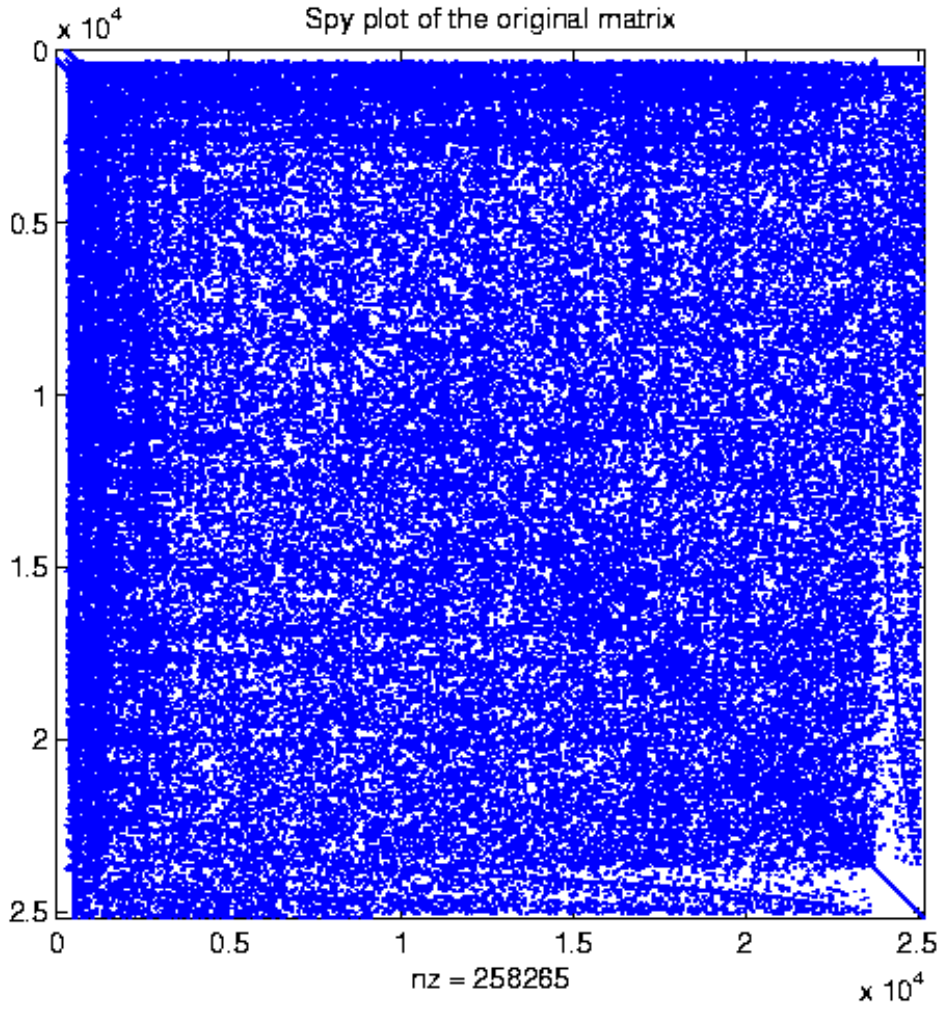
- global version of the matrix A, stored in BICRS,
- global input vector x,
- global output vector y.
- Multiple threads work simultaneously on contiguous blocks in the BICRS data structure;
- conflicts only arise on the row-wise separator areas.



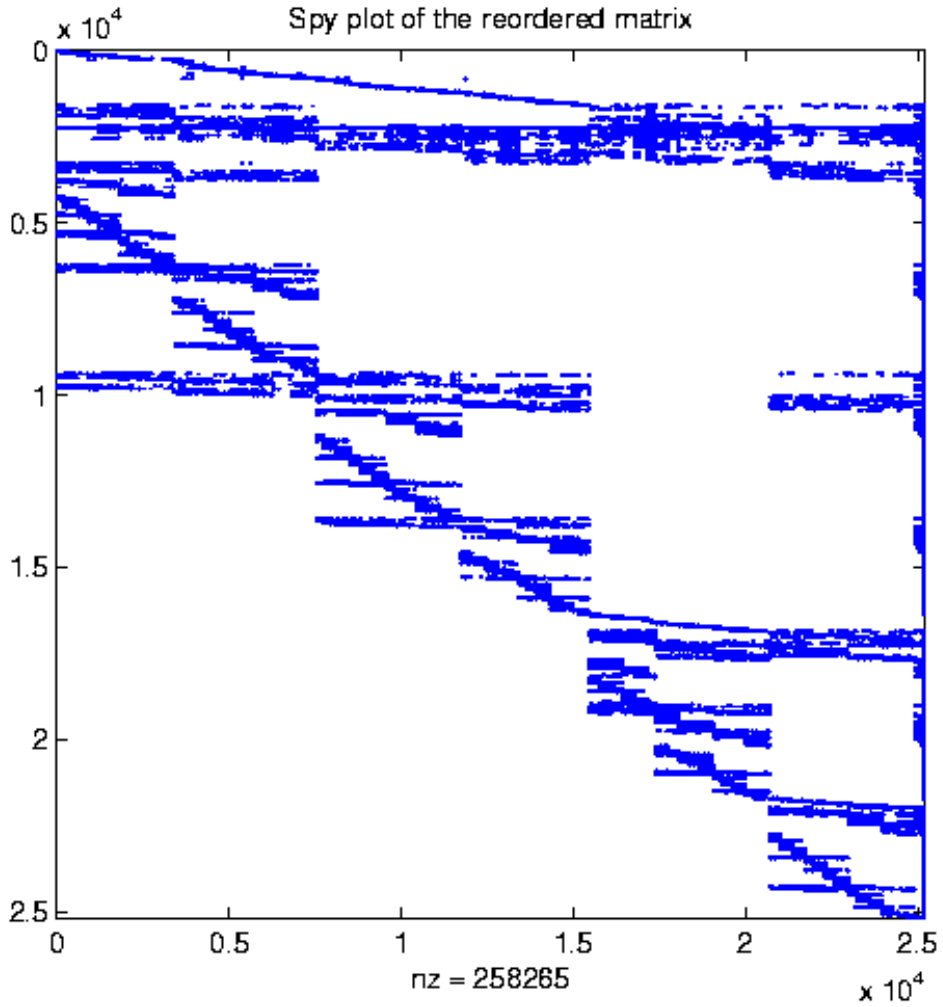
Use $t - 1$ synchronisation steps to prevent concurrent writes.

Experimental results

Chip industry

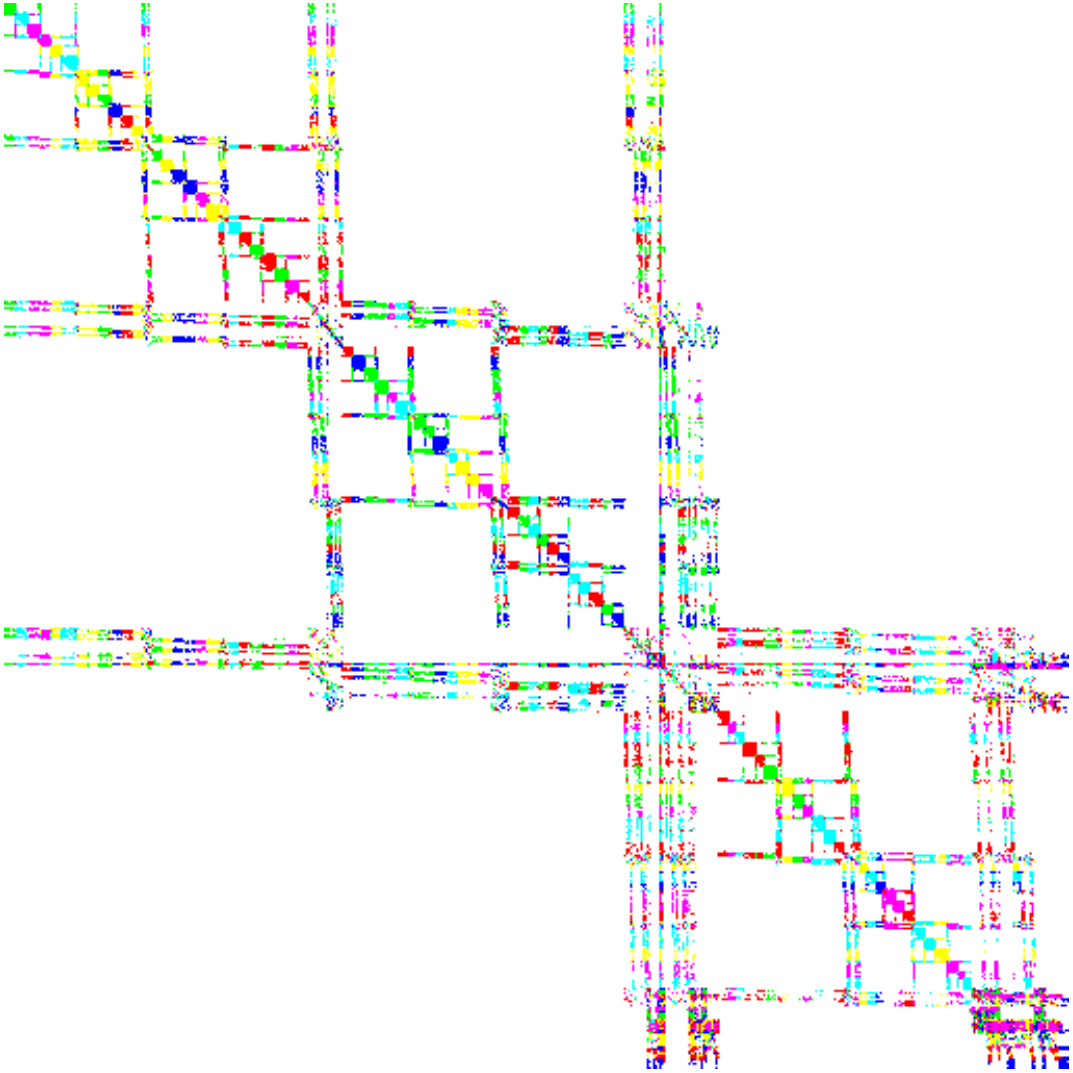


Chip industry – 1D reordering



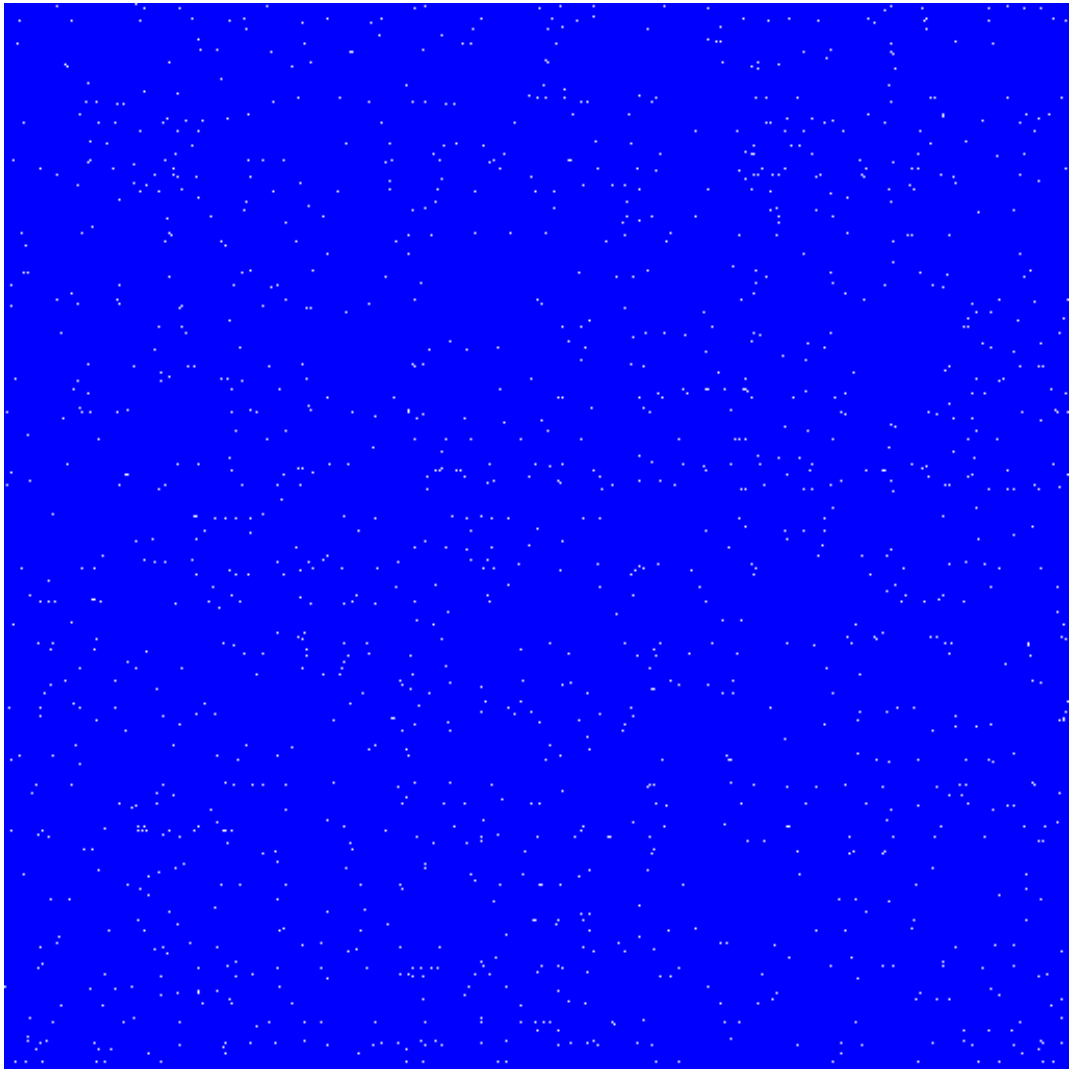
$p = 100, \quad \epsilon = 0.1$

Chip industry – 2D reordering

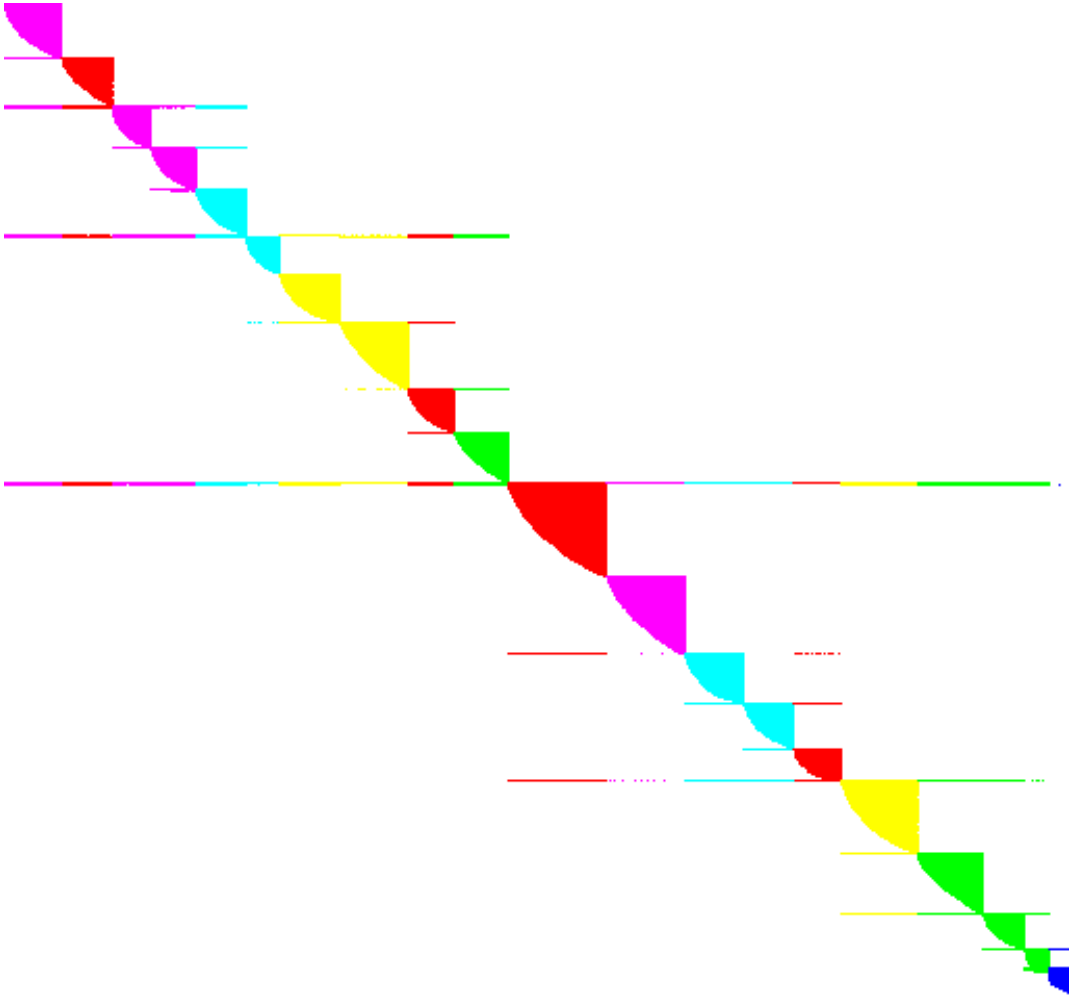


$p = 100, \quad \epsilon = 0.1$

Link matrix

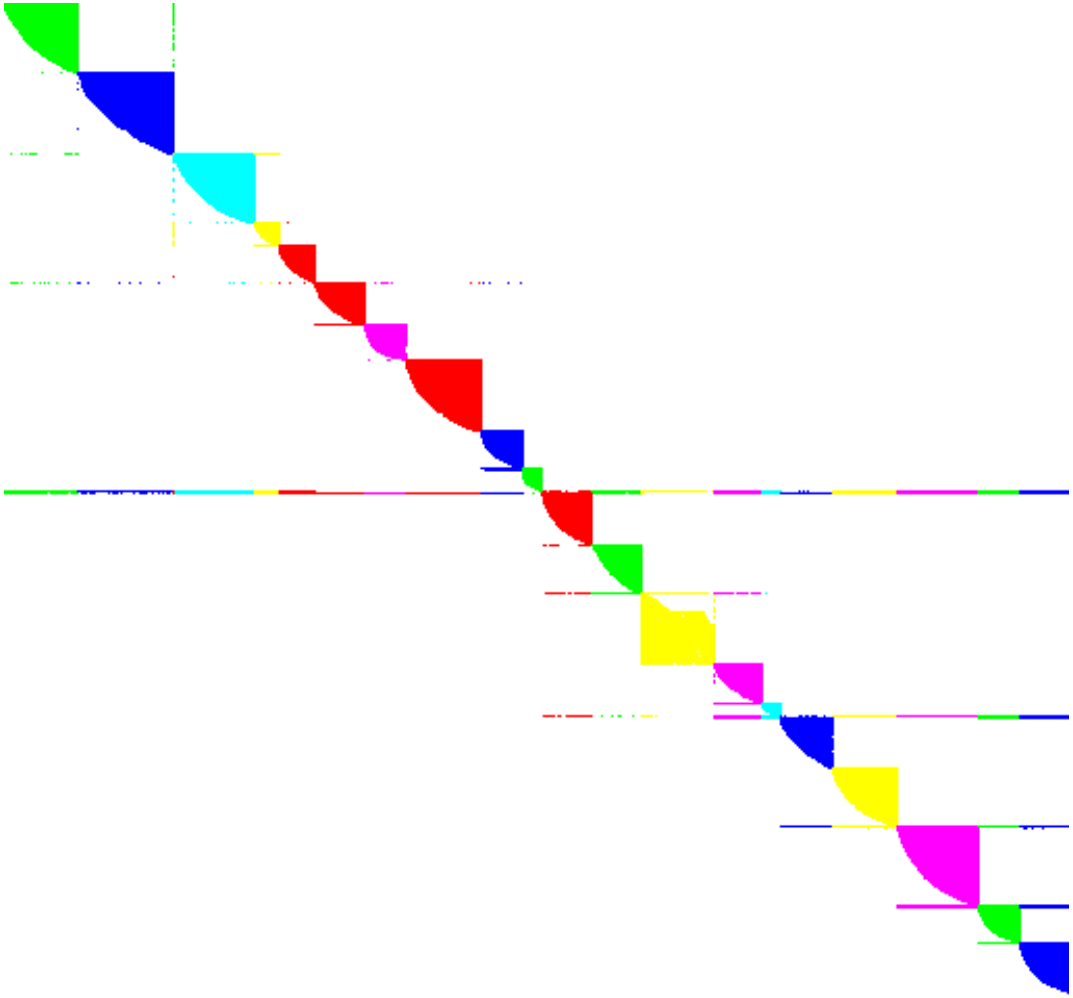


Link matrix – 1D reordering



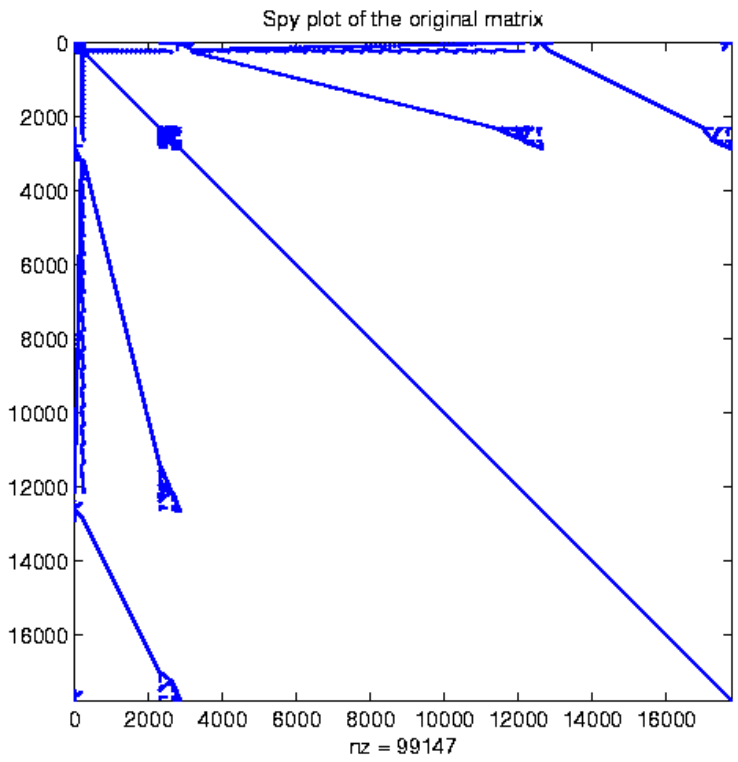
$p = 20, \quad \epsilon = 0.1$

Link matrix – 2D reordering

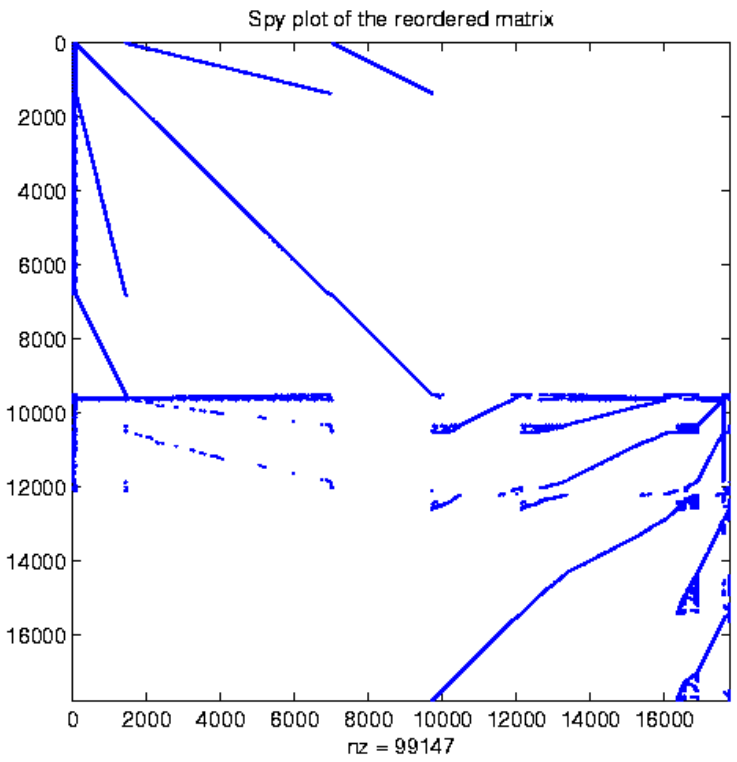


$p = 20, \quad \epsilon = 0.1$

The memplus matrix – 1D reordering

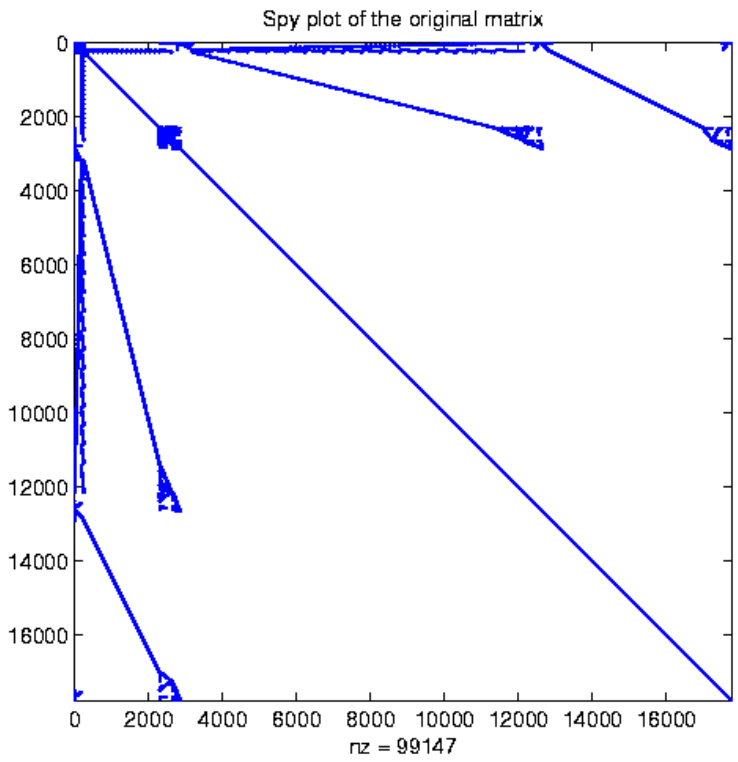


$p = 1$ (original)

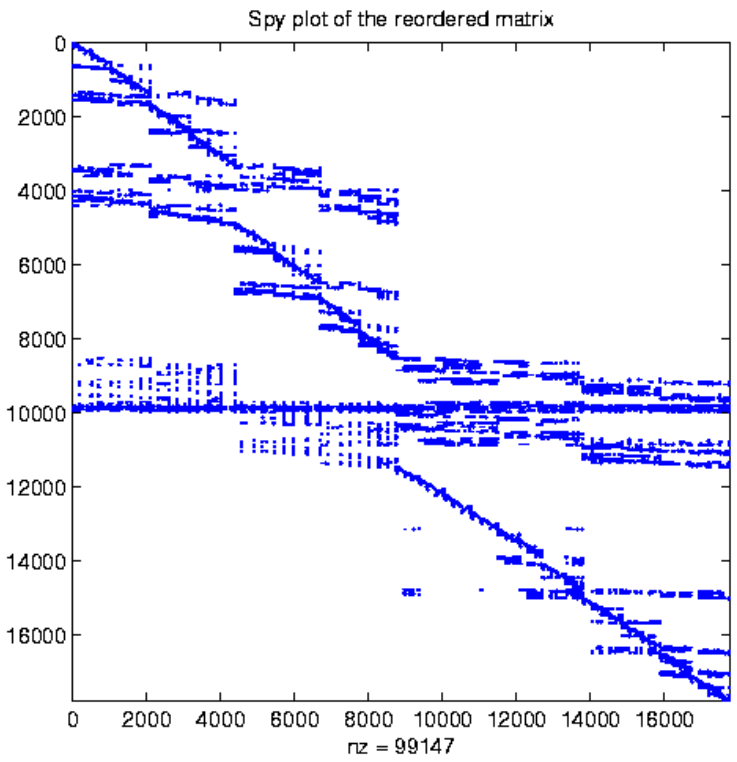


$p = 2, \quad \epsilon = 0.1$

The memplus matrix – 1D reordering

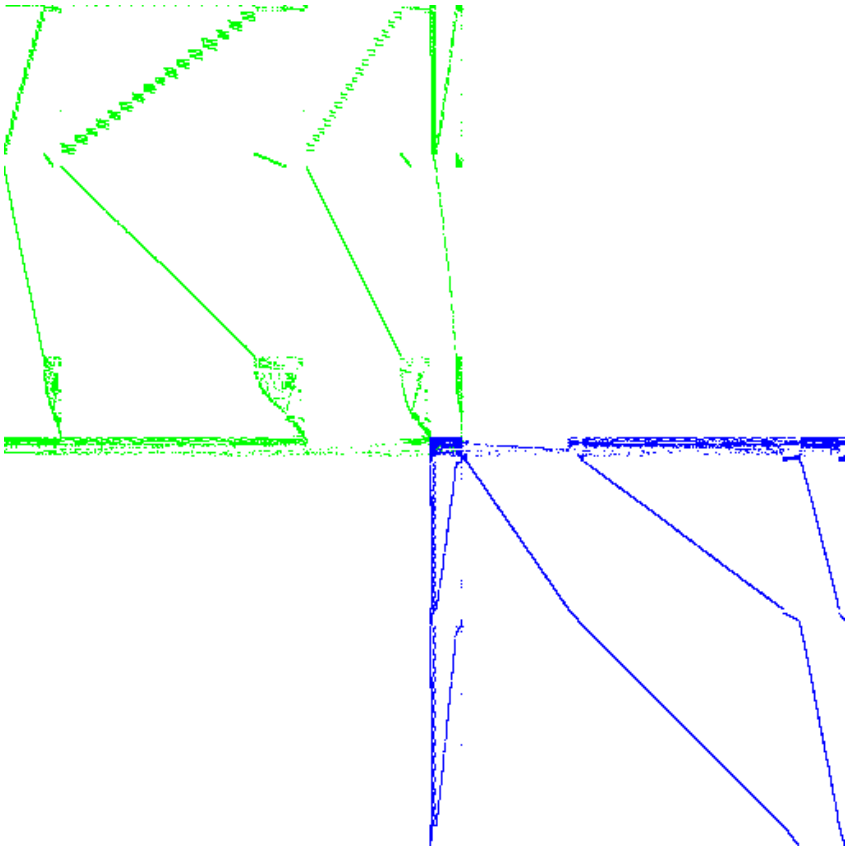


$p = 1$ (original)

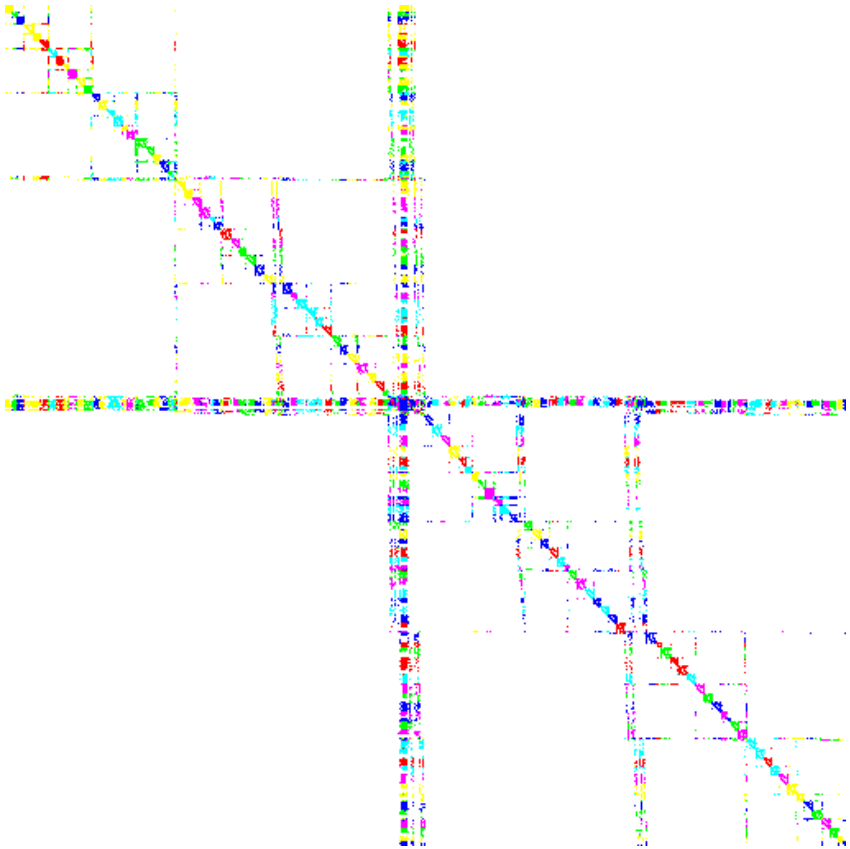


$p = 100, \quad \epsilon = 0.1$

The memplus matrix – 2D reordering

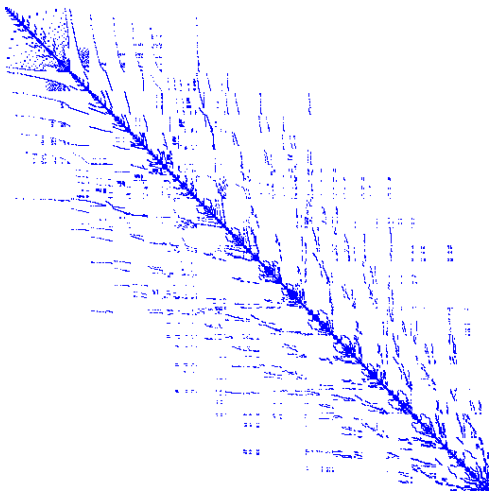


$p = 2$

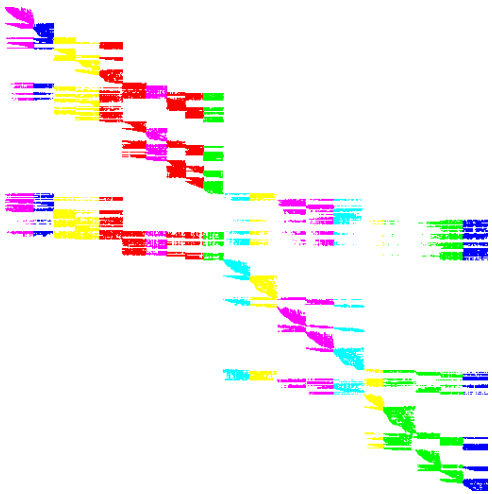


$p = 100$

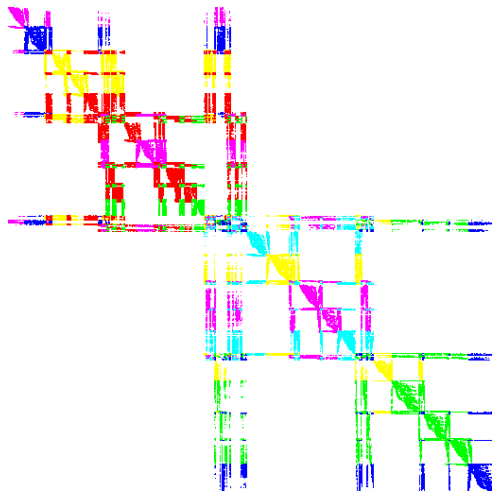
The cage14 matrix



Original



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



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

Pre-processing and SpMV times

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.)

Old: SpMV on the original matrix A
1D: SpMV on the 1D reordered matrix PAQ
2D: SpMV on the 2D reordered matrix PAQ

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

Results from 2011: reordering on an AMD Opteron 2378,
SpMV on an Intel Q6600

Parallel: distributed-memory architectures

Directly use partitioner output:

Matrix	$p = 1$	$p = 4$	$p = 16$	$p = 64$
cage13	372.2	120.7	37.1	*16.1*
stanford_berkeley	552.6	169.3	71.2	*21.4*

Using the BSPOnMPI library with the parallel SpMV kernel from BSPedupack; three superstep algorithm on two nodes of 16 IBM Power6+ processors.

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

Parallel: shared-memory architectures

Directly use partitioner output:

Matrix	sequential unordered	$p = 2$	$p = 3$	$p = 4$
cage14	232.8	272.5	*249.7*	297.1
wiki2005	564.2	285.3	*244.5*	255.0

Using the Java MulticoreBSP library; two superstep algorithm with full synchronisation.

Yzelman and Bisseling, *An Object-Oriented BSP Library for Multicore Programming*, 2011 (Pre-print);
`http://www.math.uu.nl/people/yzelman/publications/#pp`

`http://www.multicorebsp.com`

Combined parallel with reordering

Intel Core 2 Q6600:

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*

(All timing are in milliseconds.)

Combined parallel with reordering

AMD Phenom II 945e:

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

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

(All timing are in milliseconds.)

Conclusions:

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

- remaining cache-oblivious, and
- keeping open the option of using specialised sparse BLAS libraries.
- 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 using the Mondriaan scheme performs best, especially when used with BICRS on x86 architectures. On the Power6+, the block data structures are preferred; see:

Yzelman and Bisseling, *Two-dimensional cache-oblivious sparse matrix–vector multiplication*, April 2011 (Revised pre-print),
`www.math.uu.nl/people/yzelman/publications/#pp`

Software

The latest version (3.11) of the sparse matrix partitioner software *Mondriaan* natively supports both the 1D and 2D reordering methods described here. This version has been released in December 2010, and can be found at:

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

The plain SpMV kernels and block SpMV kernels used are freely available as well, and can be found at:

`http://www.math.uu.nl/people/yzelman/software`

Sequential SpMV times without reordering

	Intel Q6600		AMD 945e	
	s3dkt3m2	GL7d18	s3dkt3m2	GL7d18
Triplet	18	1323	12	466
CRS	14	794	12	437
ICRS	*13*	856	*9*	610
Hilbert	28	*415*	16	*349*