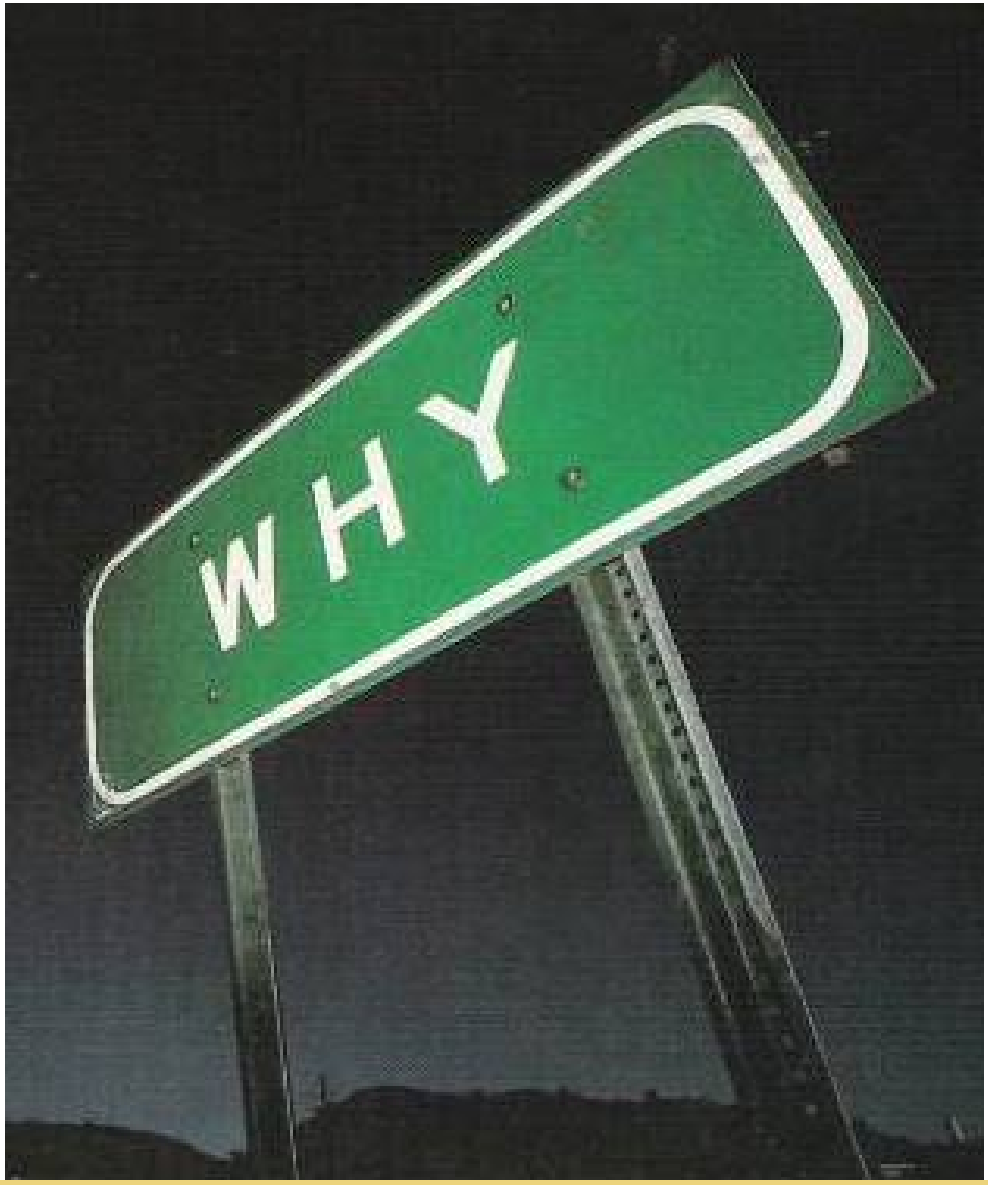
Parallel Computing – the Why and the How

Albert-Jan Yzelman

February, 2010

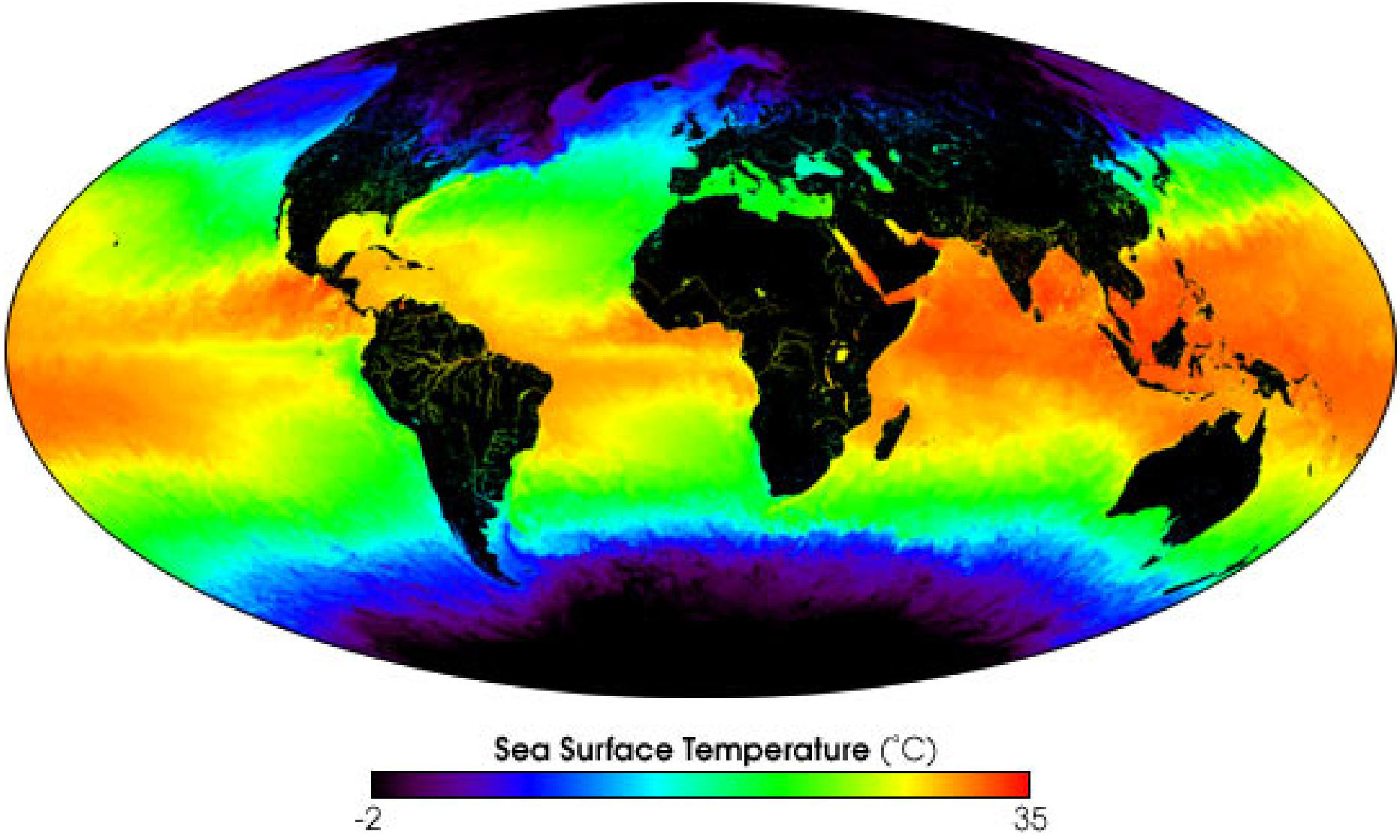
Parallel Computing



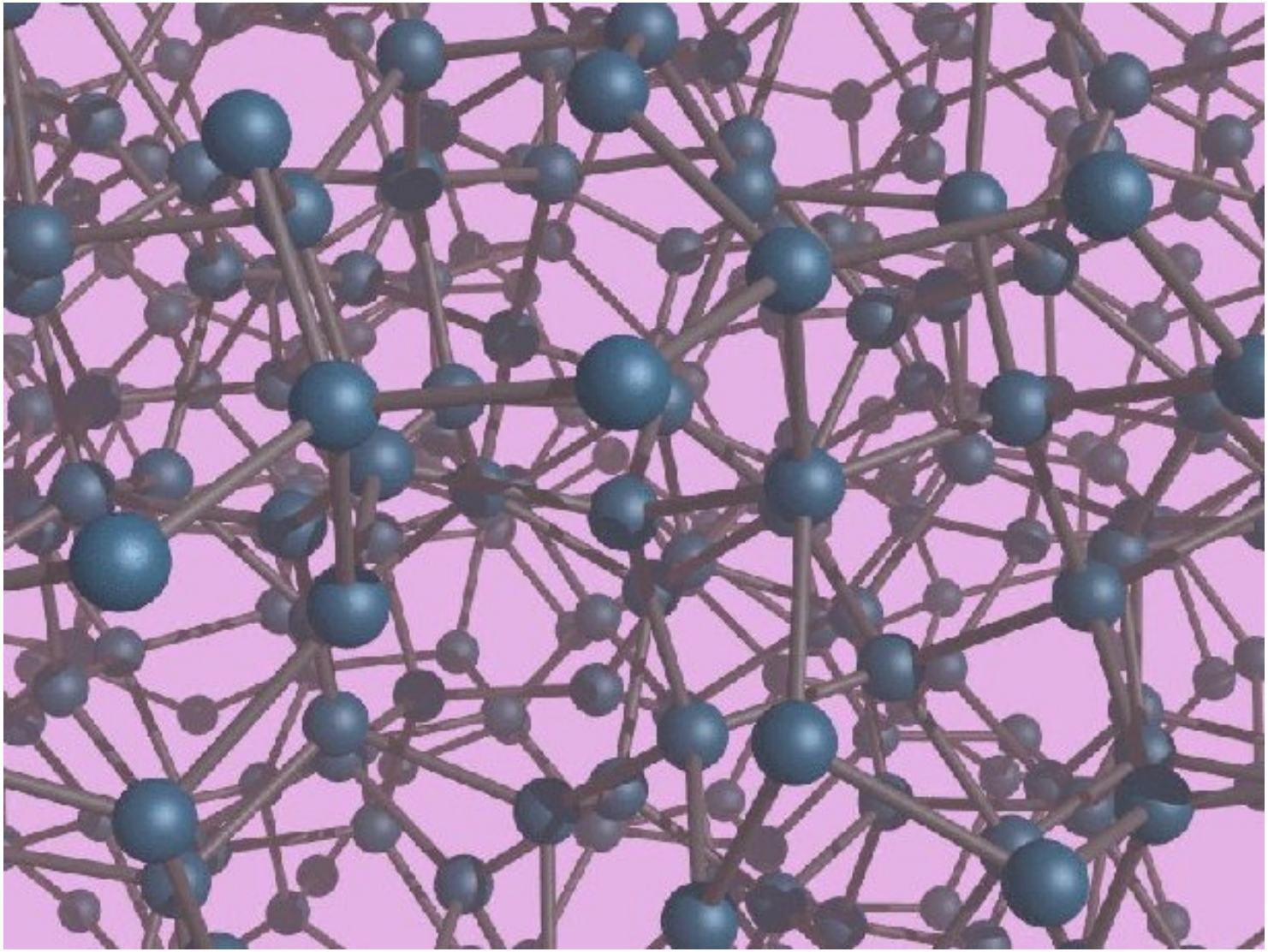
Some problems are too *large* to be solved by one processor

Google

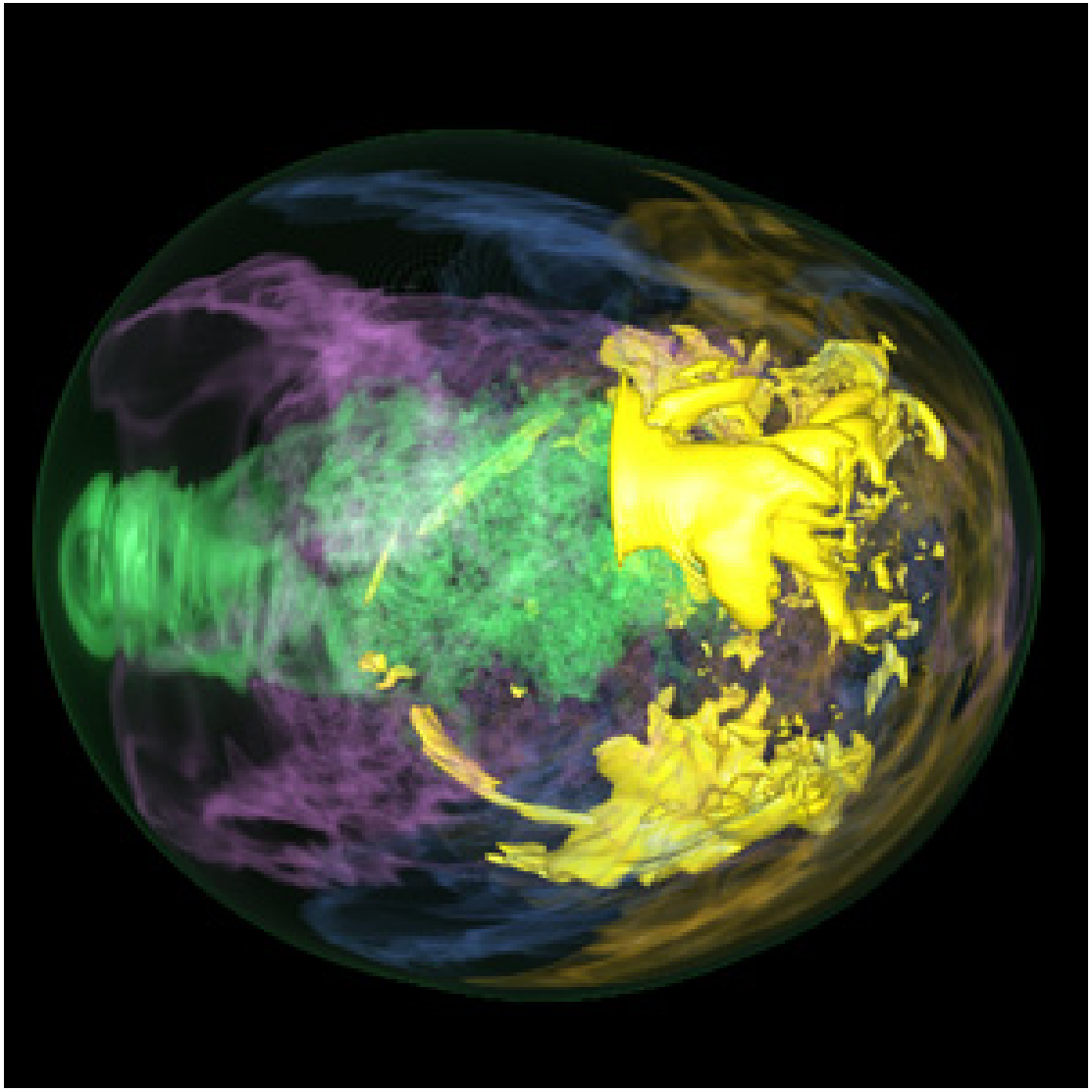
Climate Modeling



Computational Materials Science



N-body simulations



Financial market simulation

Movie rendering

The How

- Many different architectures
- One parallel model
- Load balancing

Architectures

Vector machines (1970s, until early 1990s).
ILLIAC IV (early 1960s-1976):

64 processors, 200Mflop/s, 256KB RAM, 1TB laser recording device

Vector machines (1970s, until early 1990s).
Cray-1 (1972-1976)

12 vector processors, 136 – 250 Mflop/s, 8MB RAM

Vector machines (1970s, until early 1990s).
Cray Y-MP (1988):

max. 8 vector processors, 2.6Gflop/s, 512MB RAM, 4GB SSD
Not only does $a \cdot x + y$ in one instruction, but several (64) of those!

Beowulf clusters (around 1994, named after one at NASA)

Did not find details, but described as a "Giga-flops machine"

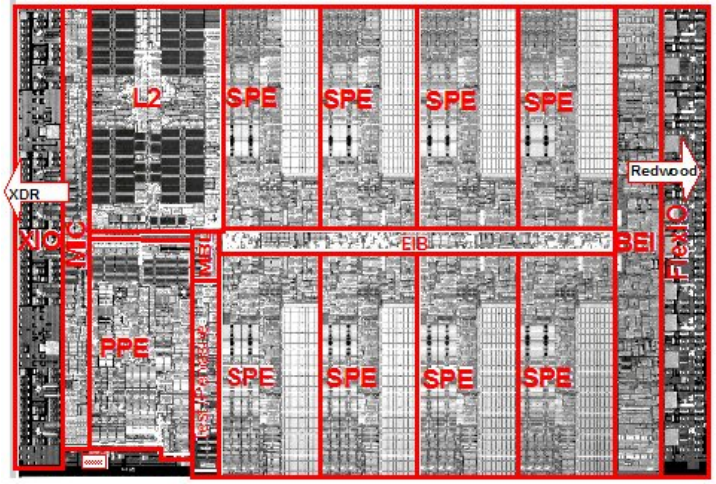
SARA: Teras / Aster, 2001

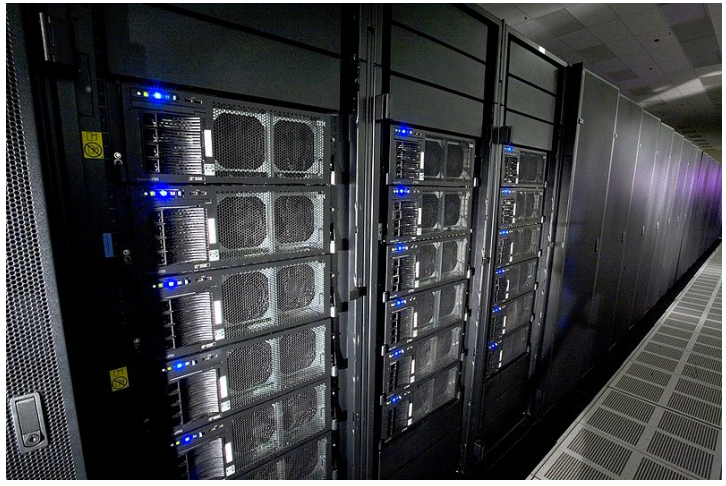
Huygens, 2007

SGI Origin 3800:	1024 MIPS R14000 processors	1 Tflop/s
SGI Altix 3700:	416 Intel Itanium 2 processors	2.2 Tflop/s
IBM System p5-575:	3328 IBM POWER6 processors	62.5 Tflop/s

Grid computing: DAS-3 (2007), 44 Tflop/s(?)

Stream processing: Cell / Roadrunner (Nov. 2008)





Cell: 1 PPE, 8 SPEs; 100 Giga-flop per second

Roadrunner: 6921 Opteron processors,
12960 Cell processors;
1.456 *Peta*-flop per second

Upcoming architectures

- Multicore (OpenMP, PThreads, MPI, OpenCL)
- GPU (OpenCL, CUDA)
- Cloud Computing (Amazon)
- Manycore (hundreds or thousands of cores)

Parallel model

In summary

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
- ...

In summary

Different connectivity;

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

Parallel Models

Solution: *bridging models*

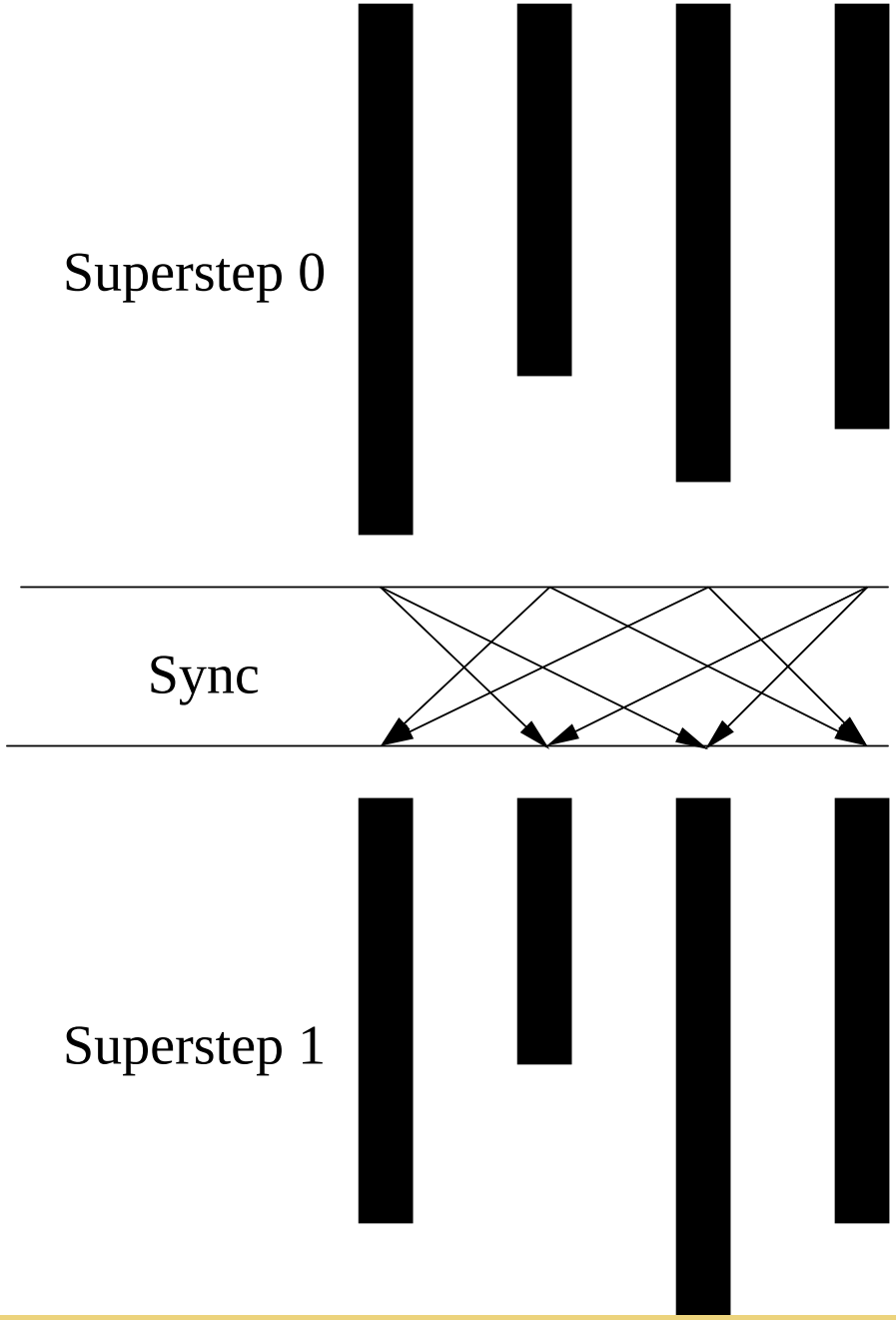
- Message Passing Interface (MPI)
- **Bulk Synchronous Parallel** (BSP)

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 homogeneous 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:

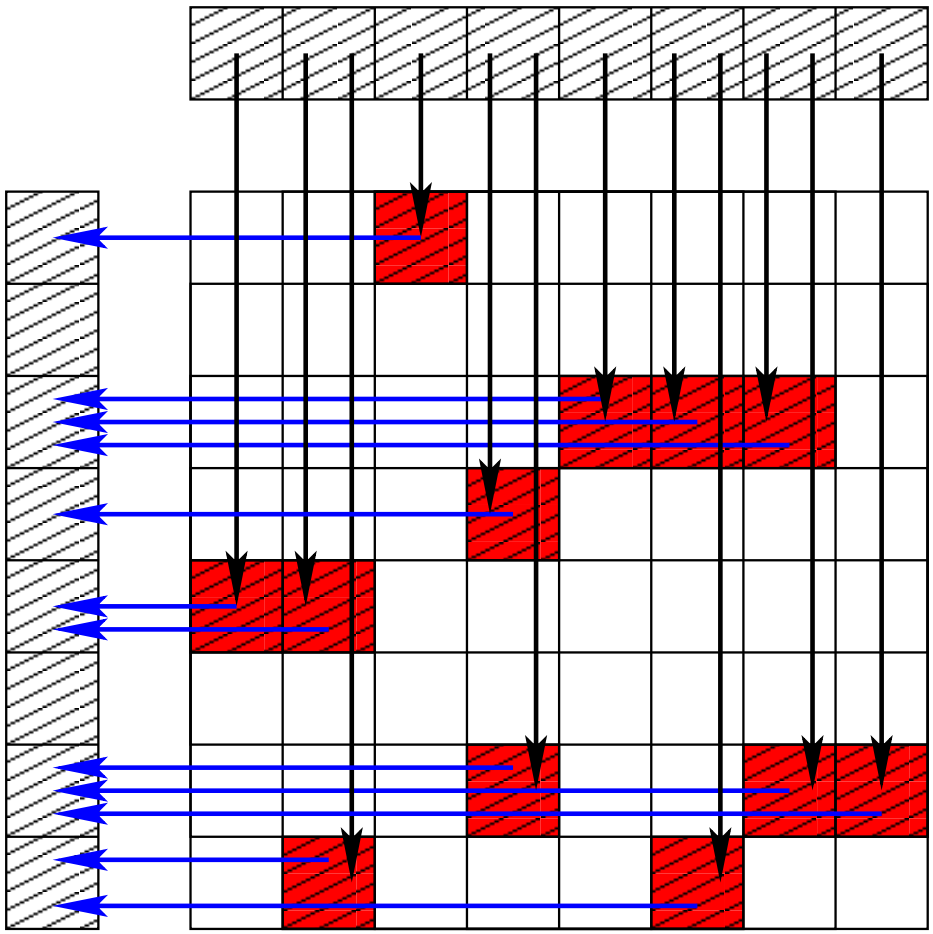
- Ask for some environment variables:
  `bsp_nprocs()`
  `bsp_pid()`
- Synchronise:
  `bsp_sync()`
- Perform "direct" remote memory access (DRMA):
  `bsp_put(source, dest, dest_PID)`
  `bsp_get(source, source_PID, dest)`
- Send messages, synchronously (BSMP):
  `bsp_send(data, dest_PID)`
  `bsp_move()`

Example: sparse matrix, dense vector multiplication

`y=Ax`:

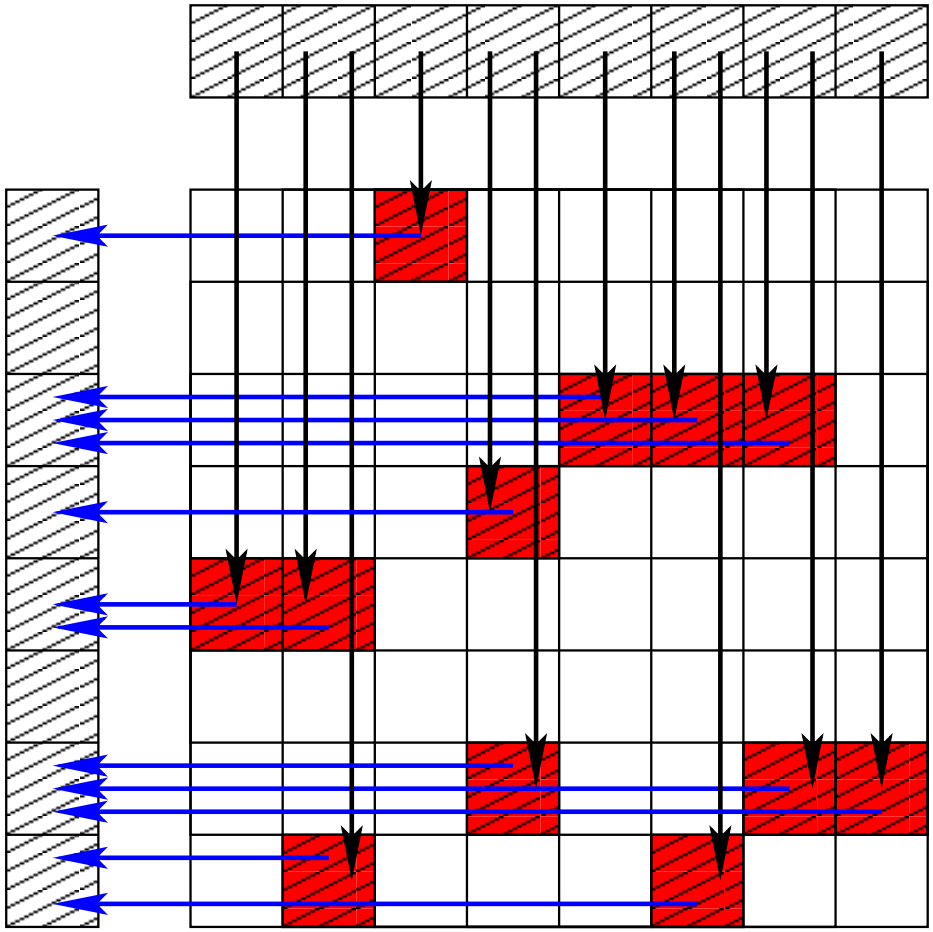
for each nonzero k **from** A

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



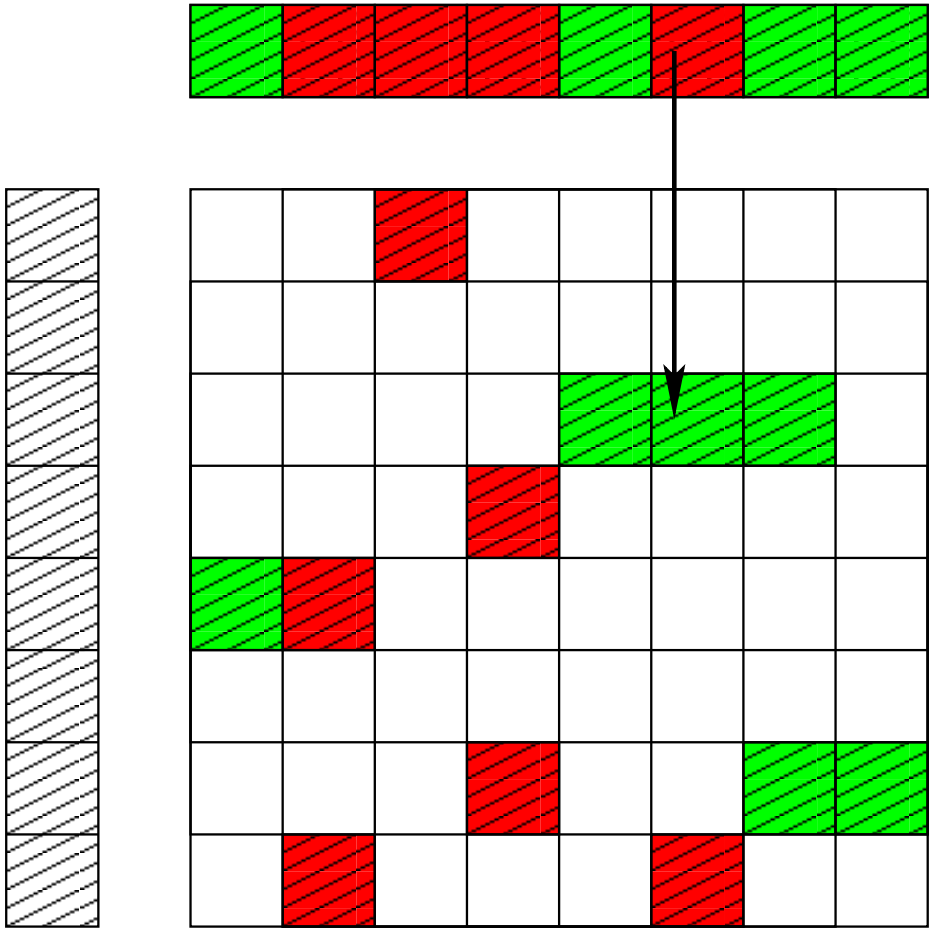
Example: sparse matrix, dense vector multiplication

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.



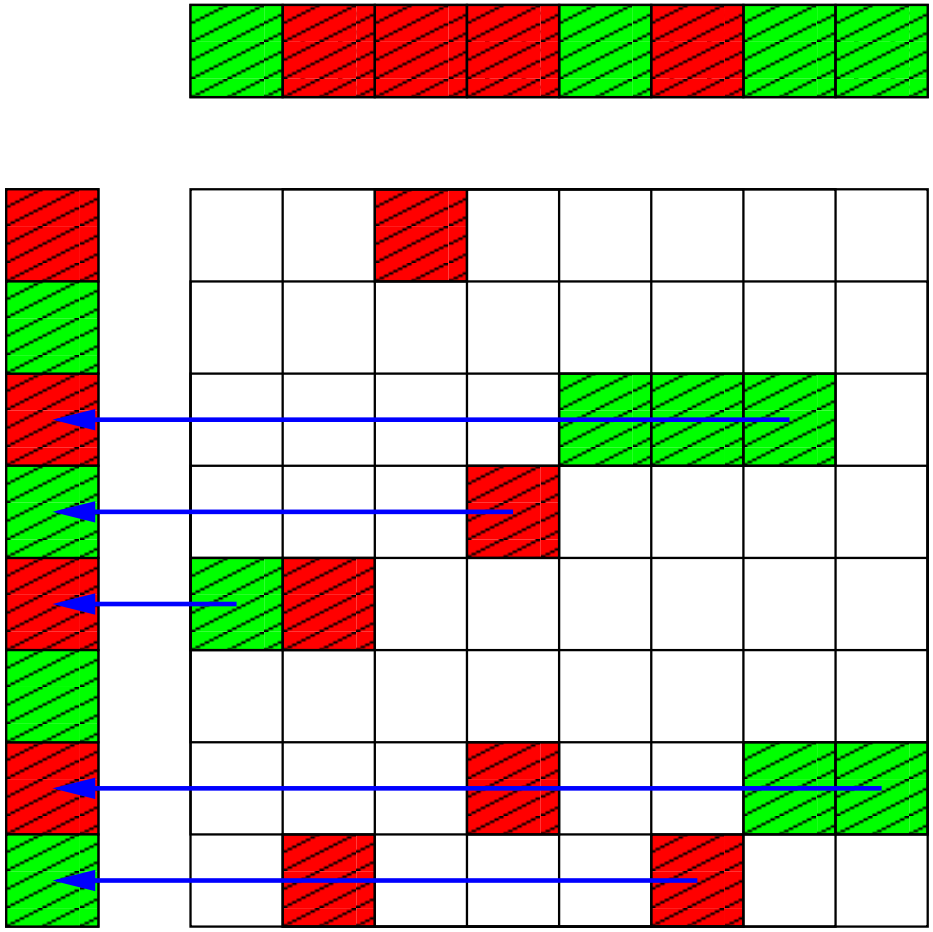
Example: sparse matrix, dense vector multiplication

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



Example: sparse matrix, dense vector multiplication

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



Example: sparse matrix, dense vector multiplication

The algorithm:

for all nonzeroes k **from** A
  if k has a local row and a local column
    add $k.v \cdot x[k.j]$ **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* $y[i]$

Balancing

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 $T+1$ 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:

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

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 $x[i]/y[i]$ 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 $x[i]/y[i]$ 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:

$$\max_{s \in [0,P-1]} |\bar{w}_i - w_i^{(s)}| \leq \epsilon \bar{w}_i,$$

where ϵ is the maximum load imbalance parameter.

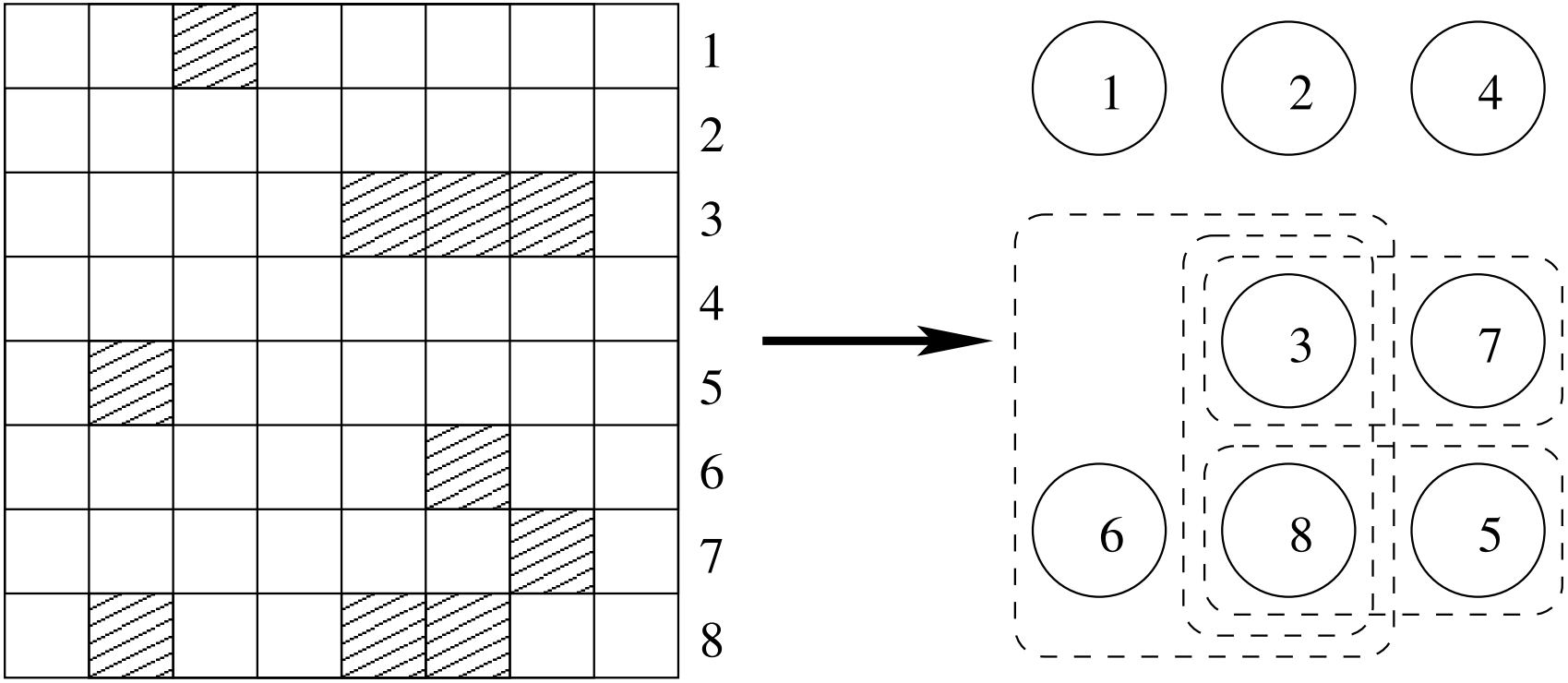
Communication balancing

Recall that the communication-part of the BSP cost depends on the maximum incoming or outgoing volume over all processors;
for each superstep, we minimise

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

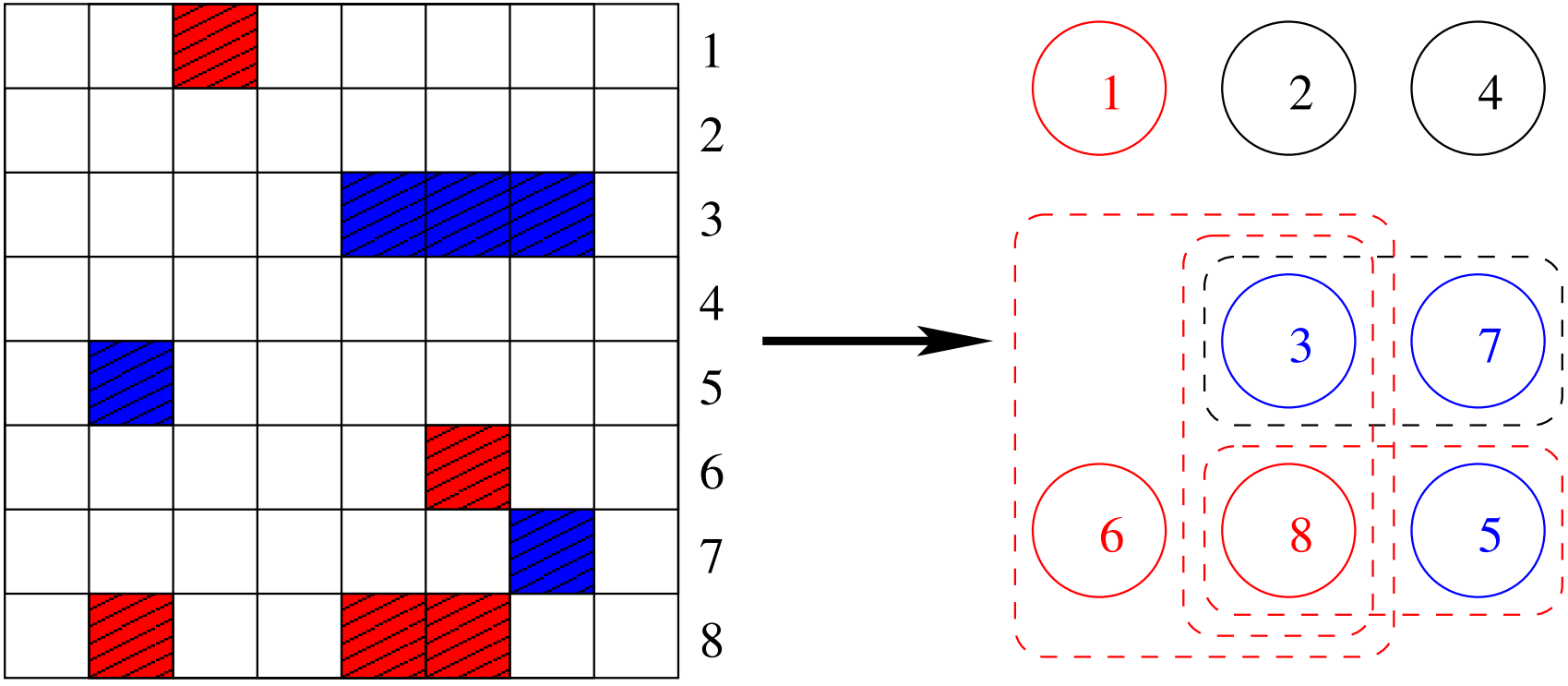
not the total communication volume $\sum_s r_i^{(s)} \left(= \sum_s t_i^{(s)} \right)$.

"Shared" columns: communication during fan-out



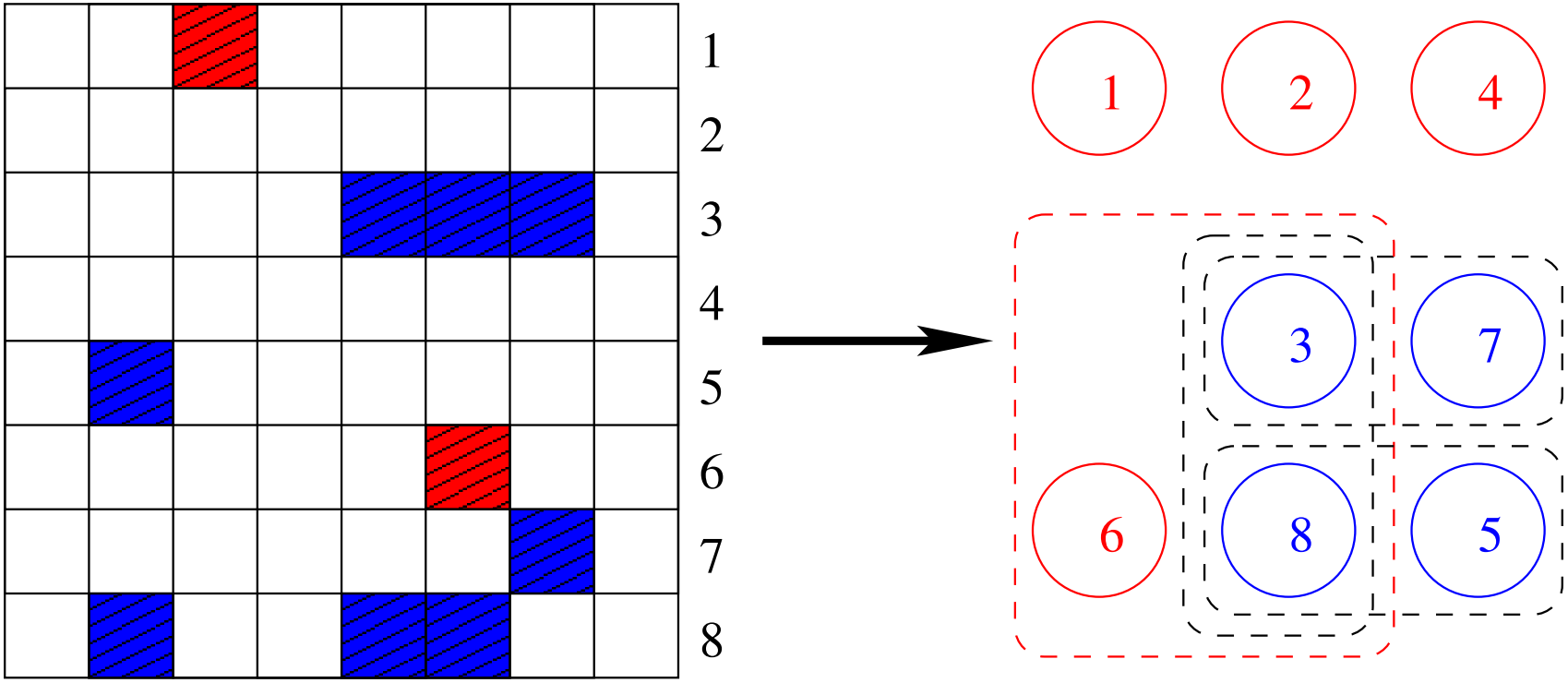
Column-net model; a cut net means a shared column

"Shared" columns: communication during fan-out



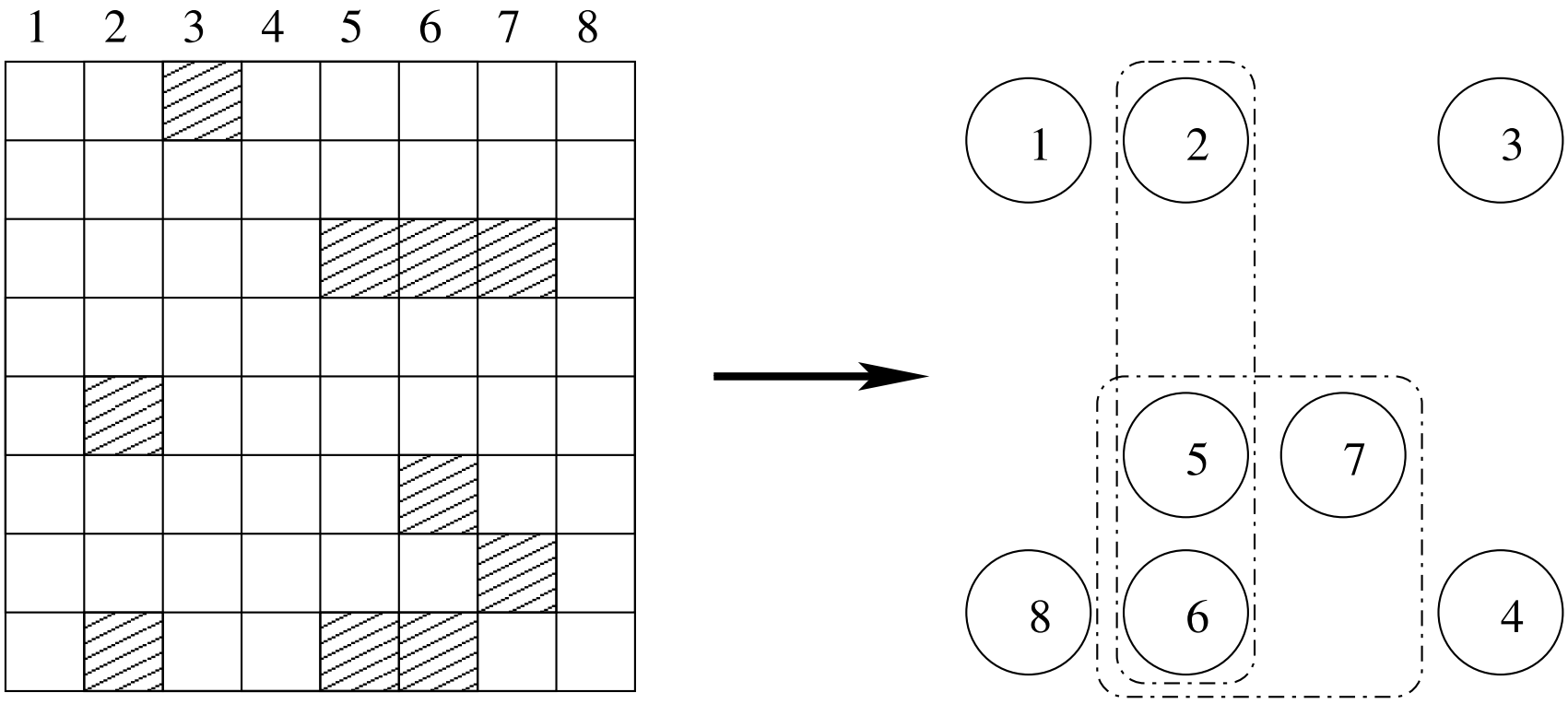
Column-net model; a cut net means a shared column

"Shared" columns: communication during fan-out



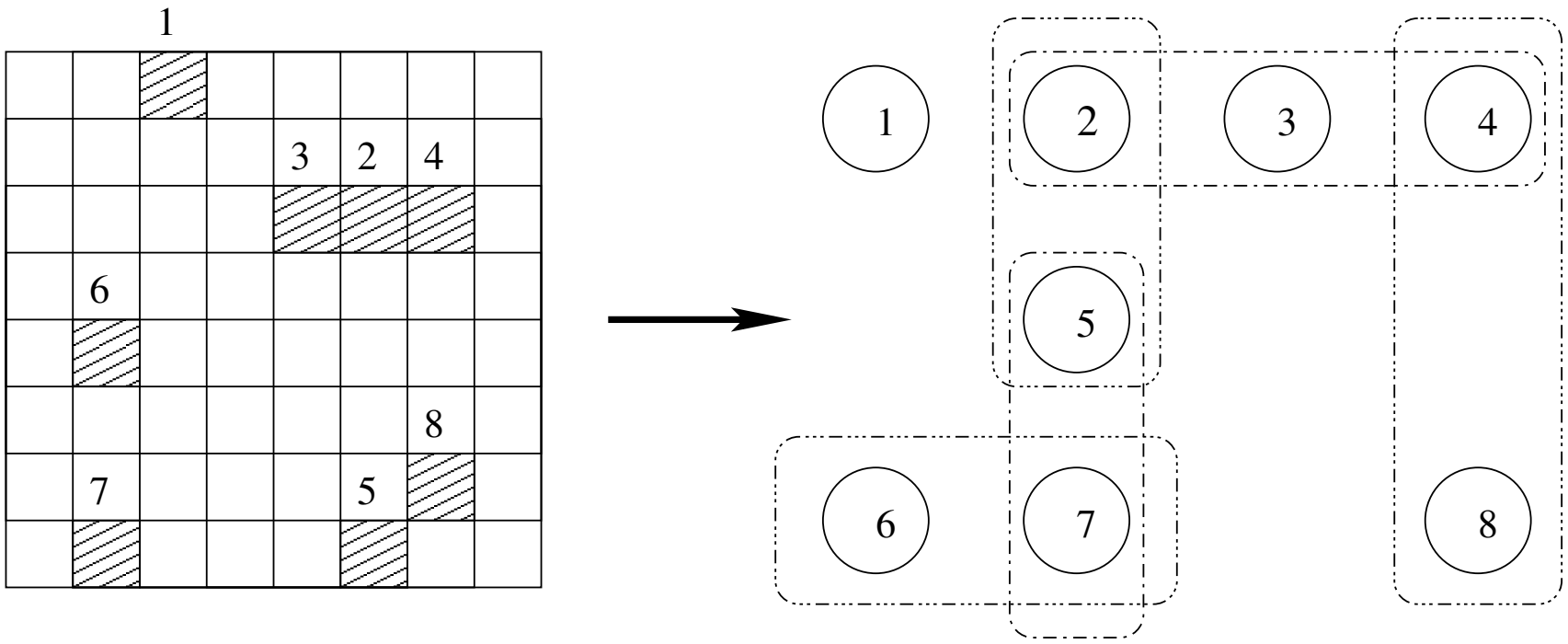
Column-net model; a cut net means a shared column

"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

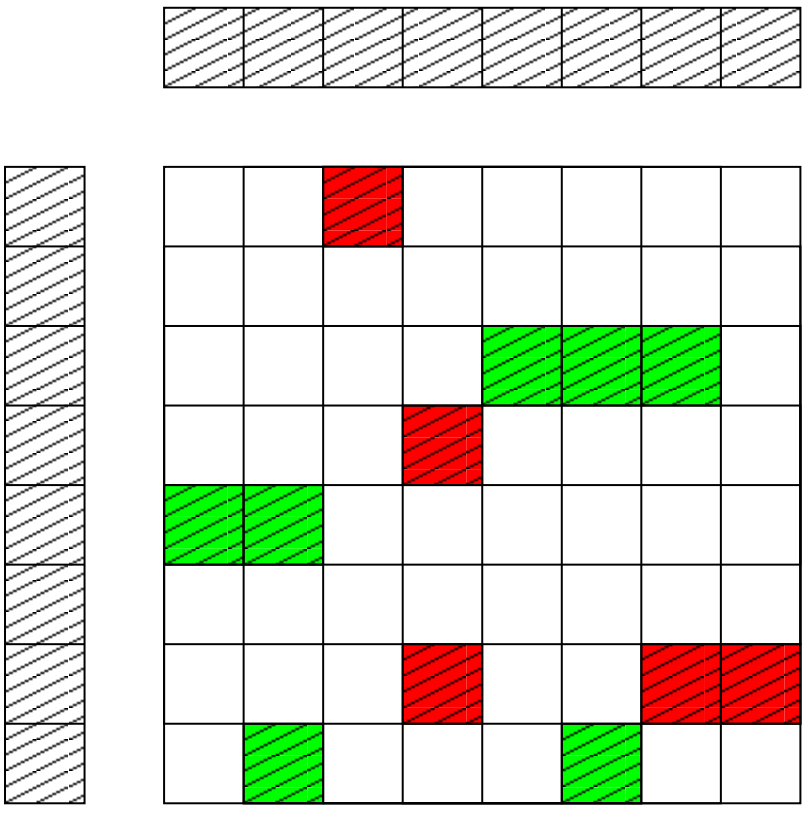
Fiduccia & Mattheyses, *A linear-time heuristic for improving network partitions*, Proceedings of the 19th IEEE Design Automation Conference (1982), pp. 175-181

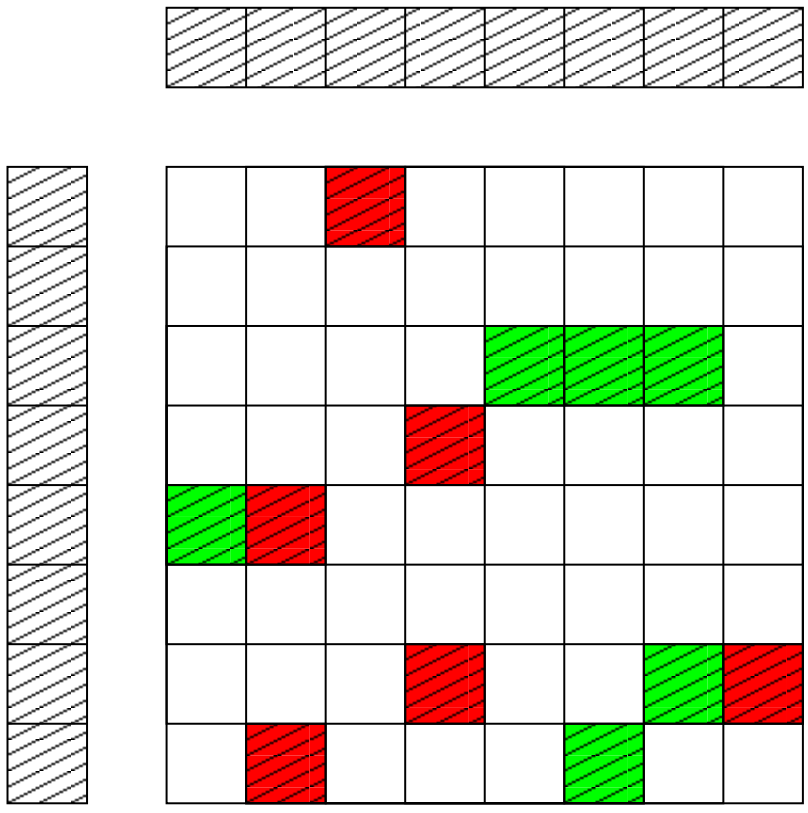
Catalyürek & Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel Distributed Systems 10 (1999), pp. 673-693

Emerging partitioning strategy:

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

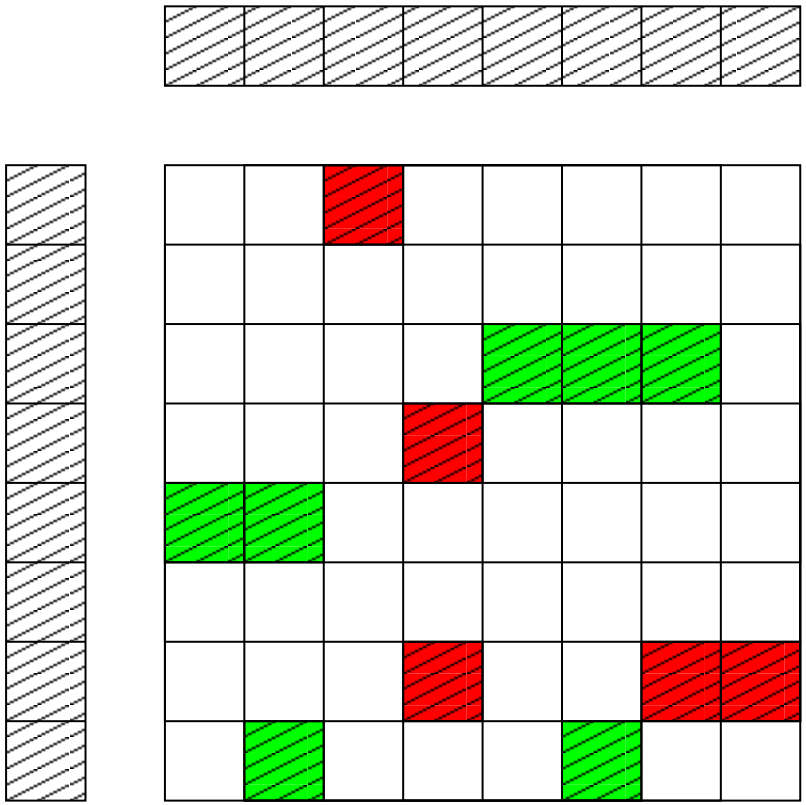
Original Mondriaan: both row- and column-net, and choose best





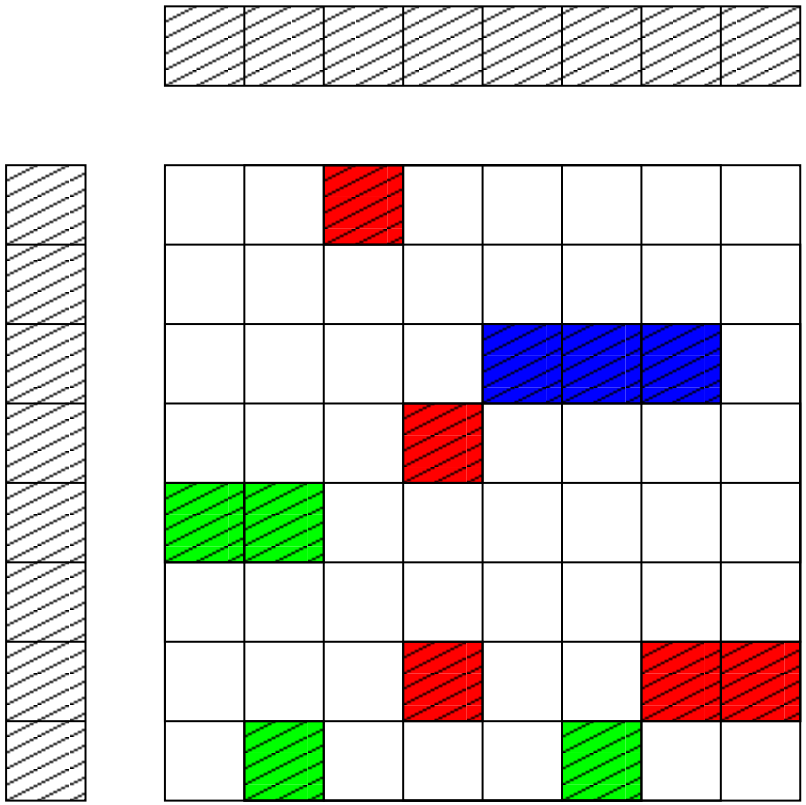
Emerging partitioning strategy:

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



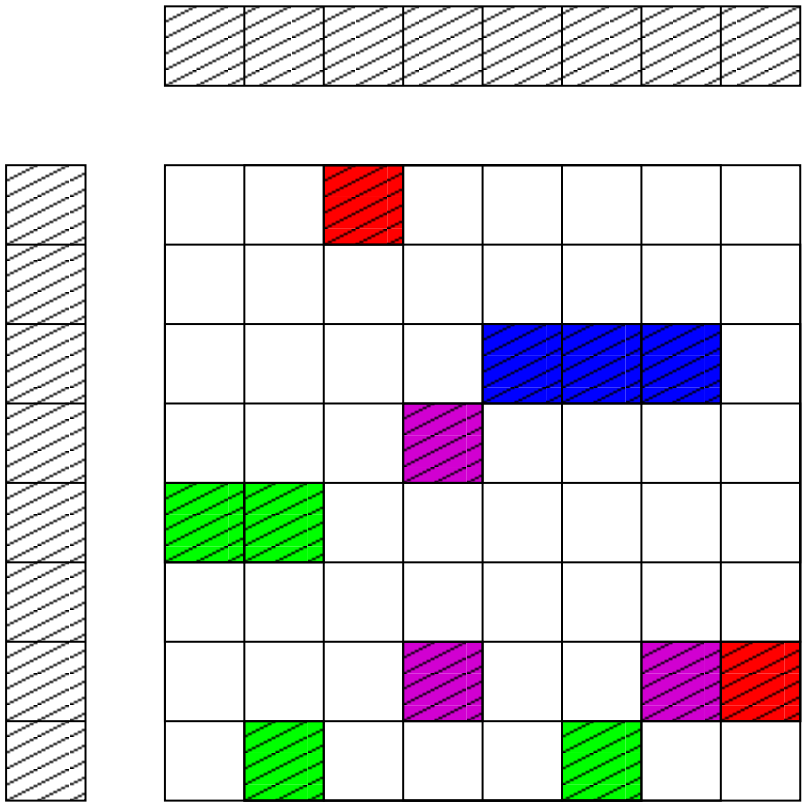
Emerging partitioning strategy:

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



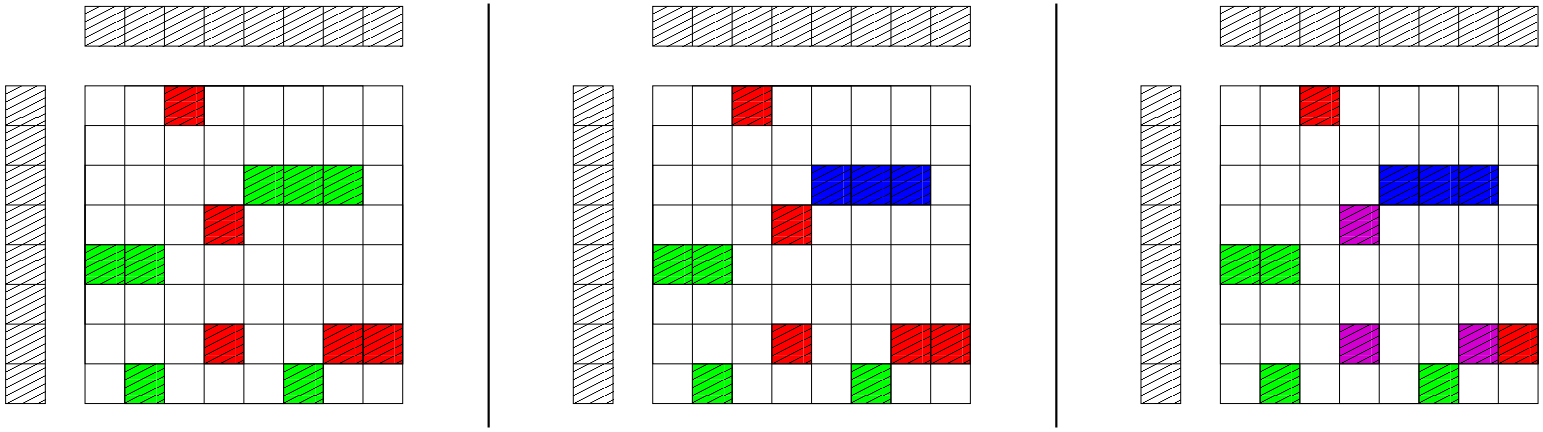
Emerging partitioning strategy:

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



Mondriaan:

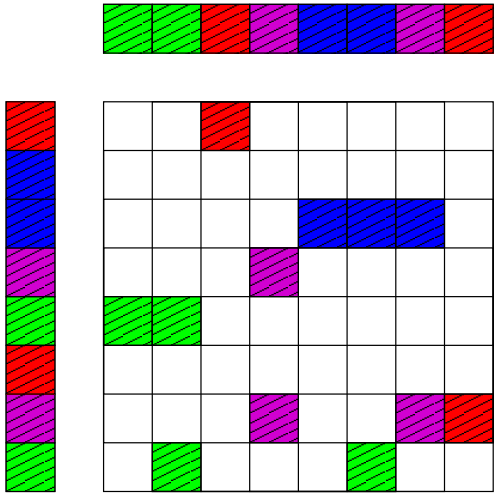
- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



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**



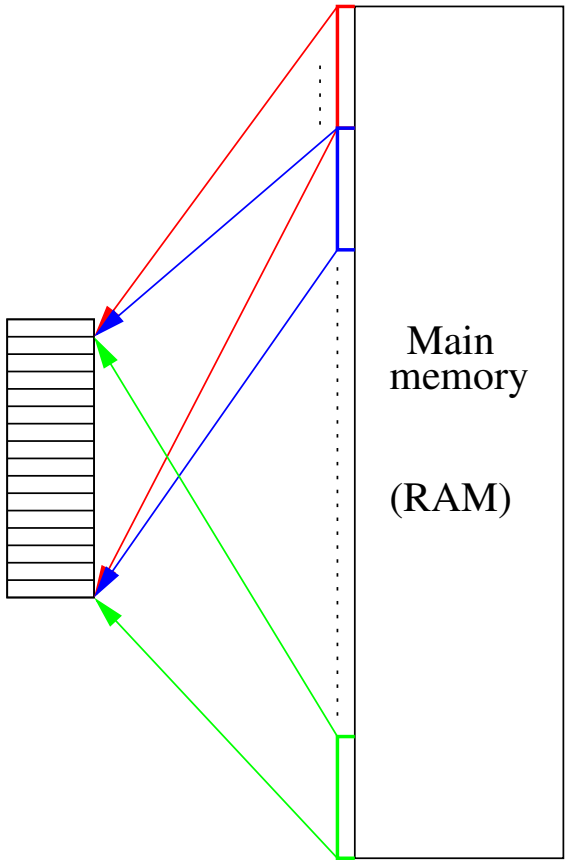
Rob H. Bisseling and Wouter Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis, Vol. 21 (2005) pp. 47-65

Sequential sparse matrix–vector multiplication

Naive cache

$k = 1$, modulo mapped cache

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



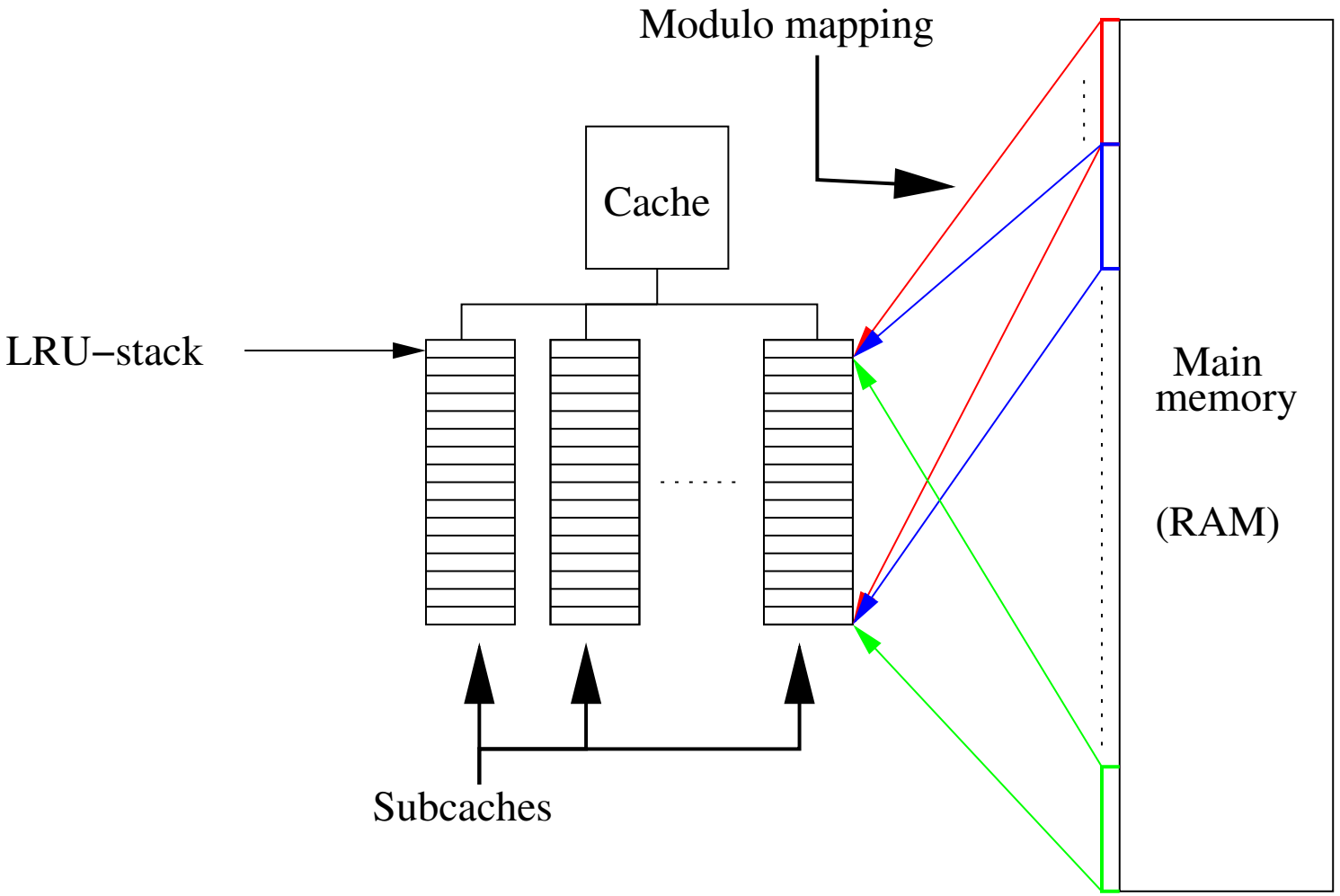
'Ideal' cache

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



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 $k = L = 2$:

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

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

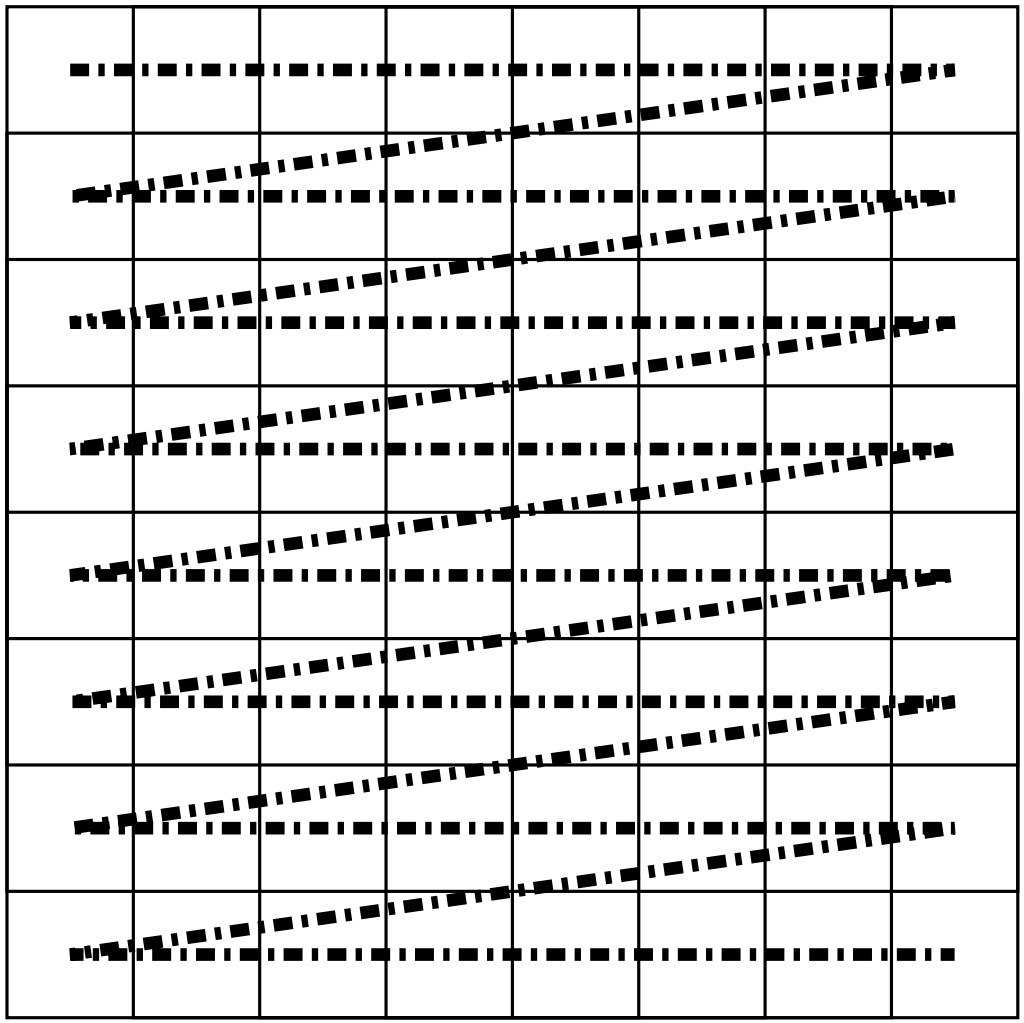
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with $k = L = 2$:

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

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)

$x_?$ $a_{0?}$

$x_?$

$\Longrightarrow$ $\Longrightarrow$

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)

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

The sparse case

Sparse matrix–vector multiplication (SpMV)

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

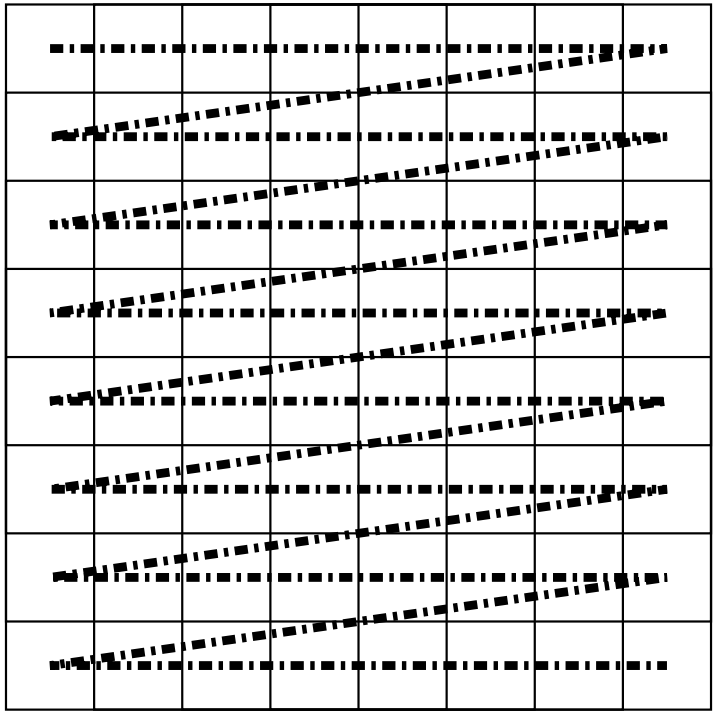
We cannot predict memory accesses in the sparse case!

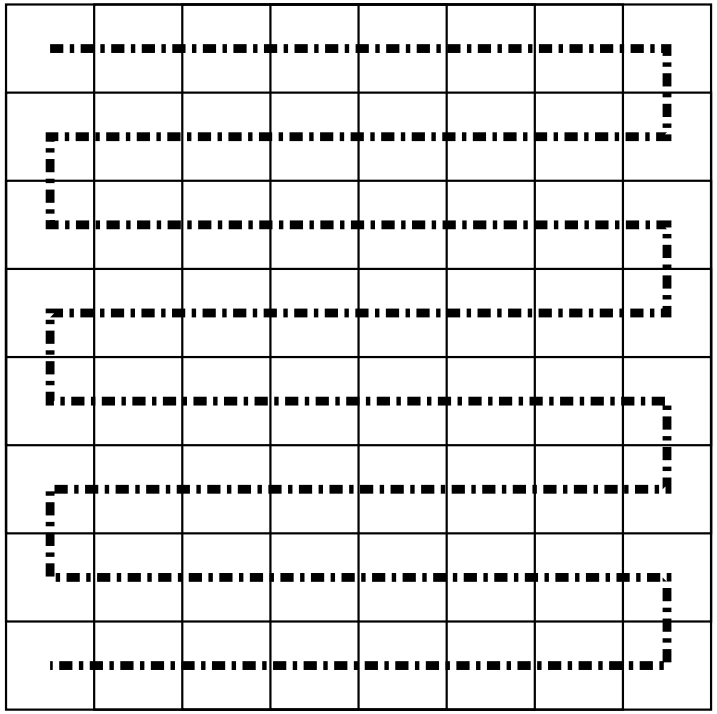
Improving cache–use

We adapt two things;

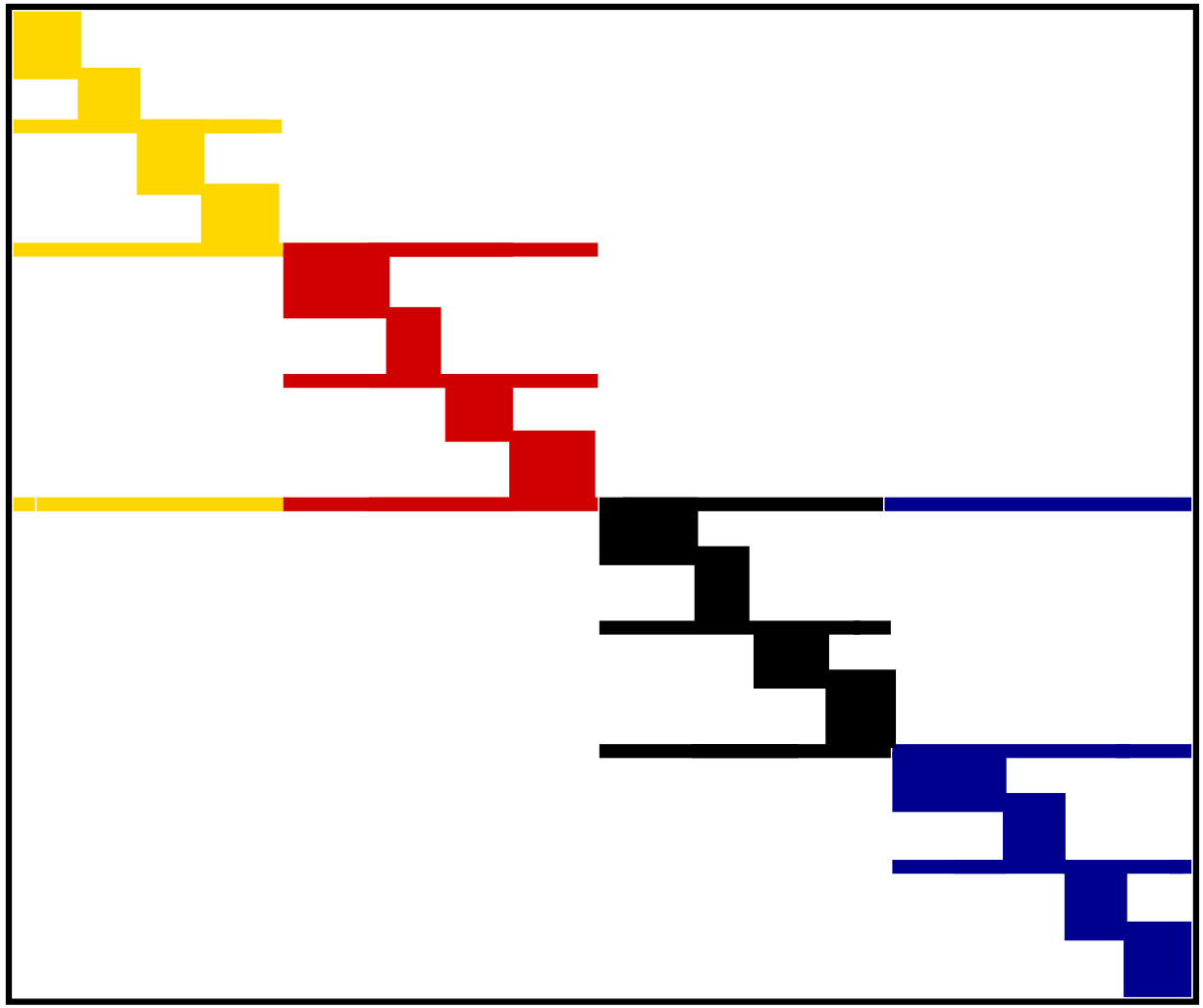
- sparse matrix storage
- sparse matrix structure

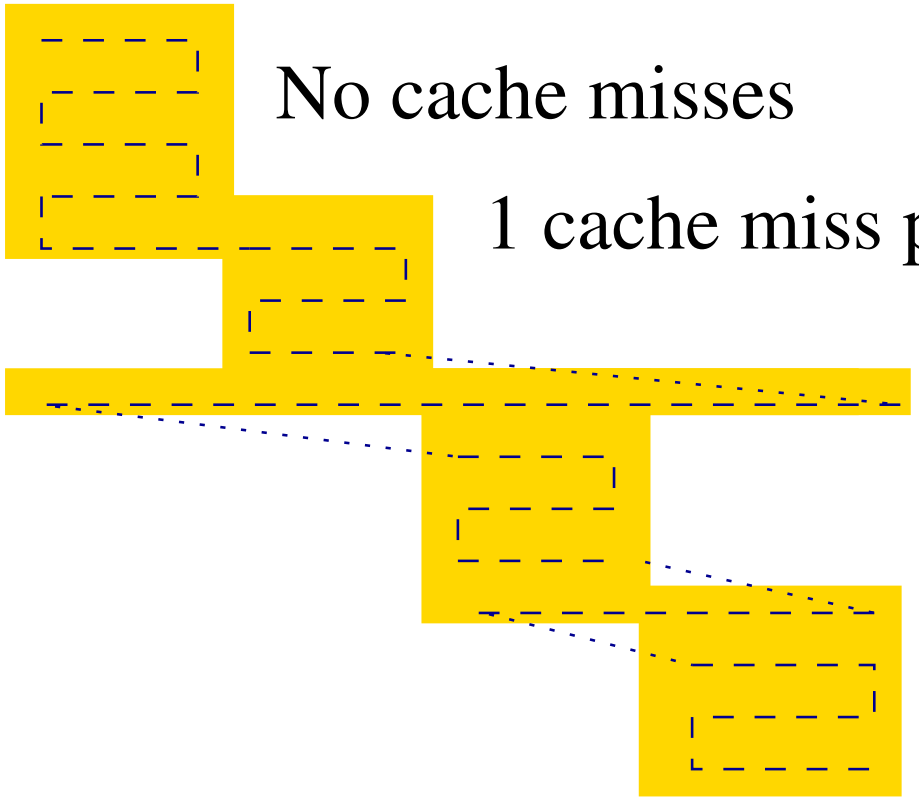
CRS to "Zig-zag" CRS:





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



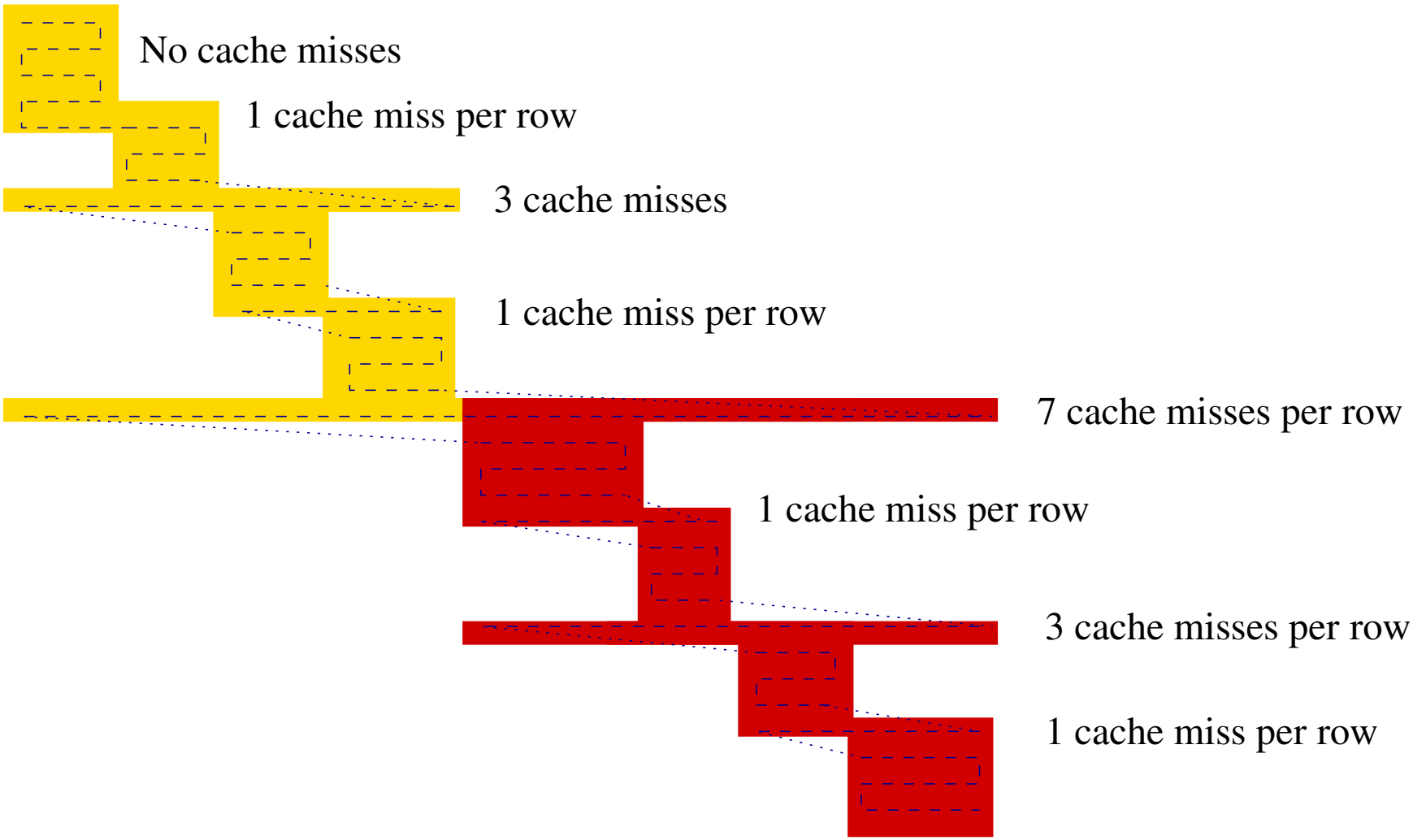


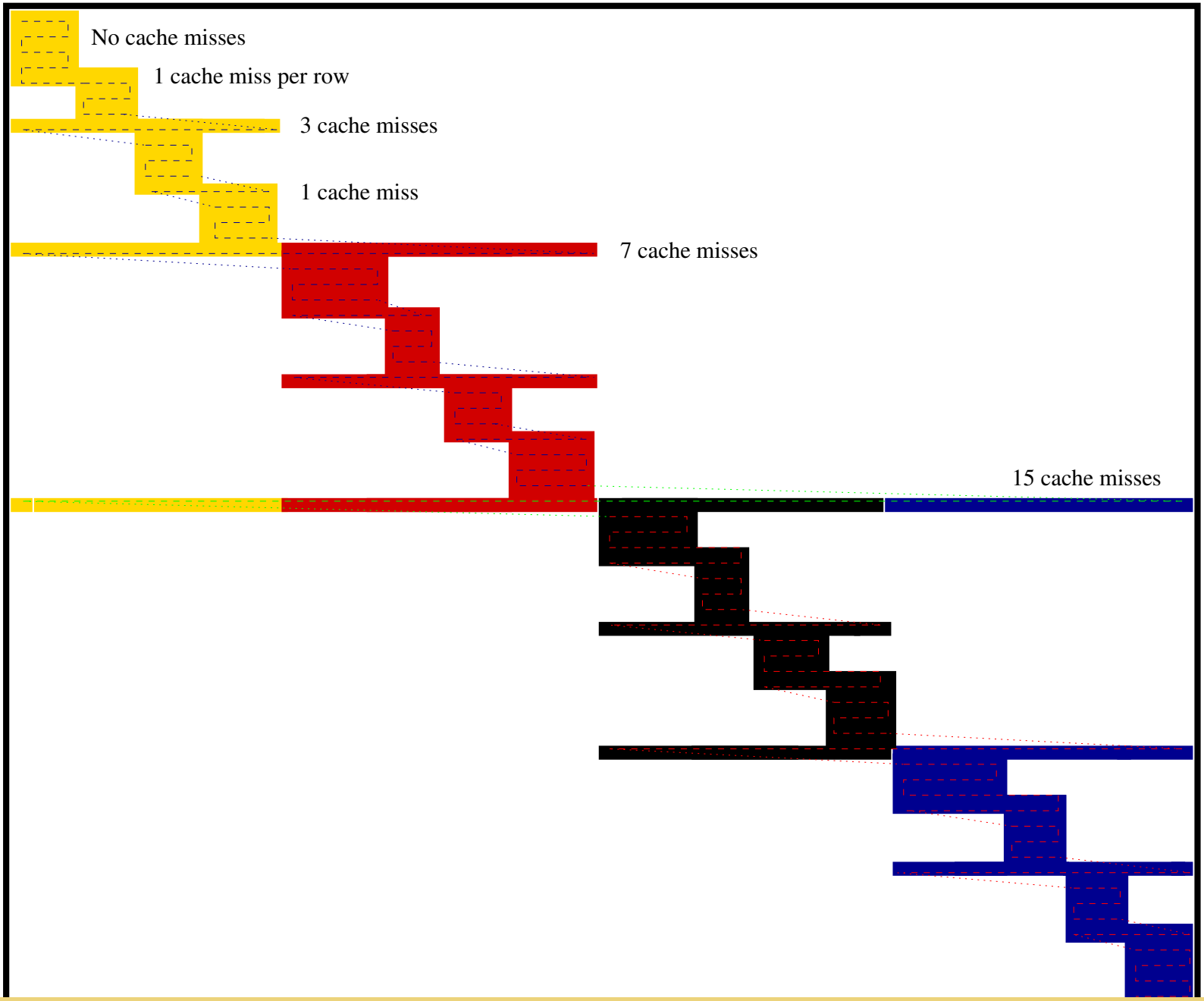
No cache misses

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

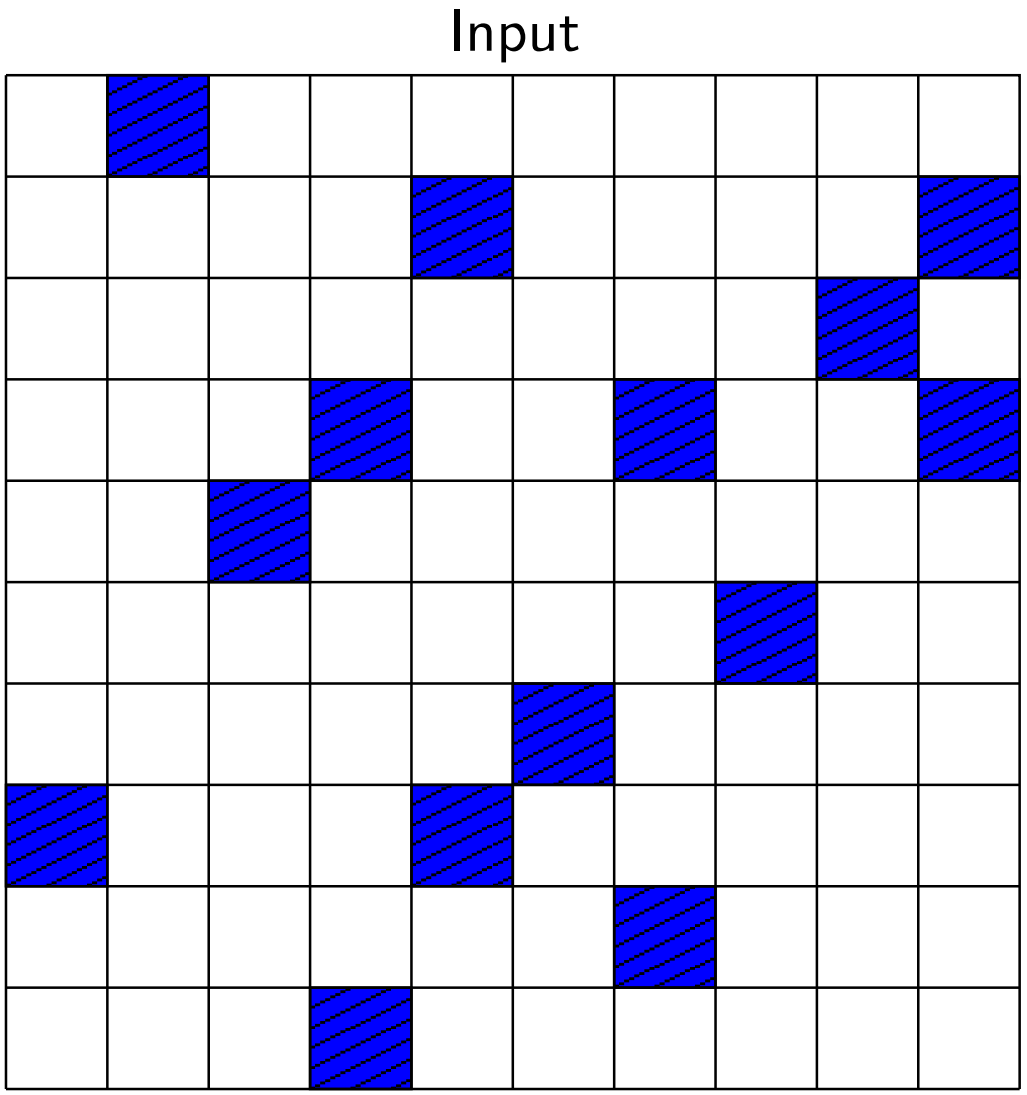
1. the matrix is stored in Zig-zag CRS format,
2. each SBD block corresponds to one vertex,
3. each *row* corresponds to one net (containing a vertex if the corresponding submatrix contains a non-zero);

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

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

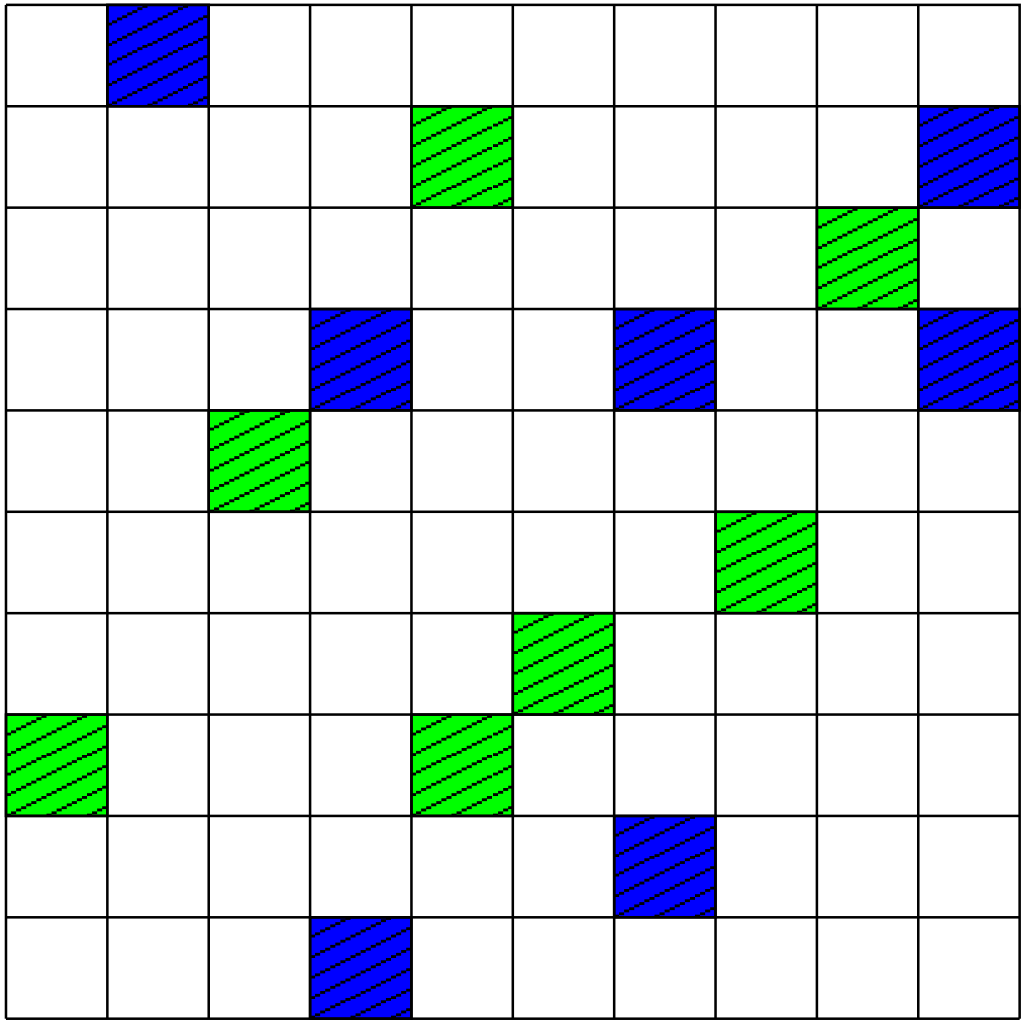
λ_i being the connectivity of the ith row

Using Mondriaan

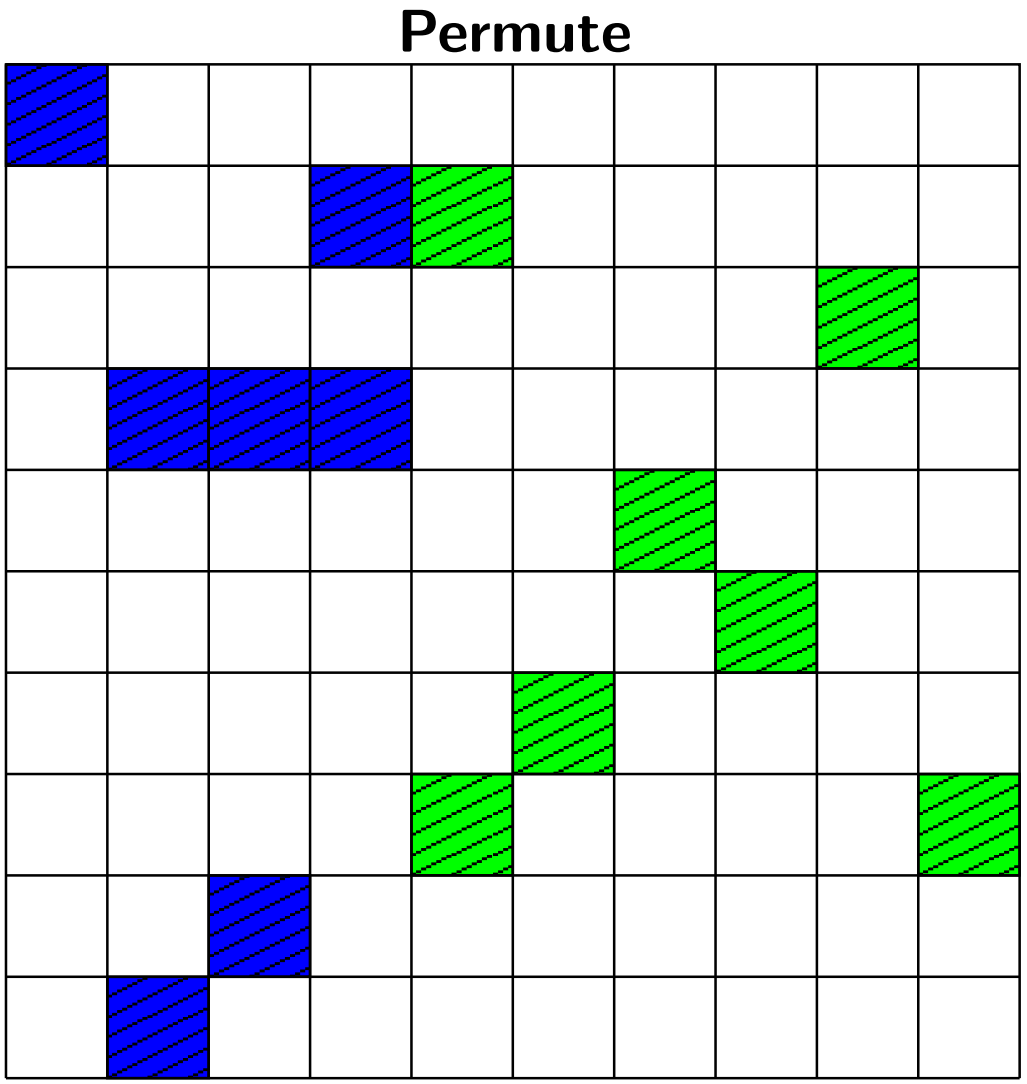


Using Mondriaan

Column partitioning (force row-net model)

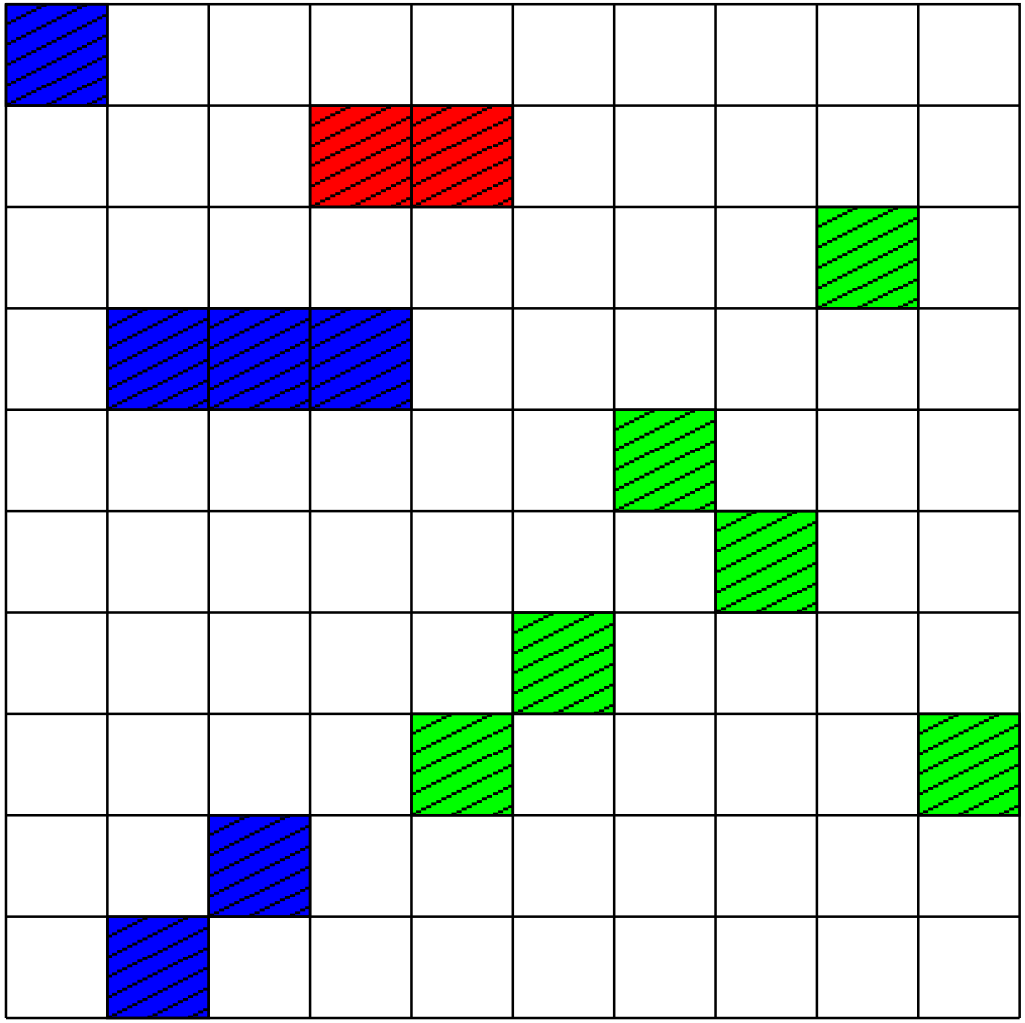


Using Mondriaan

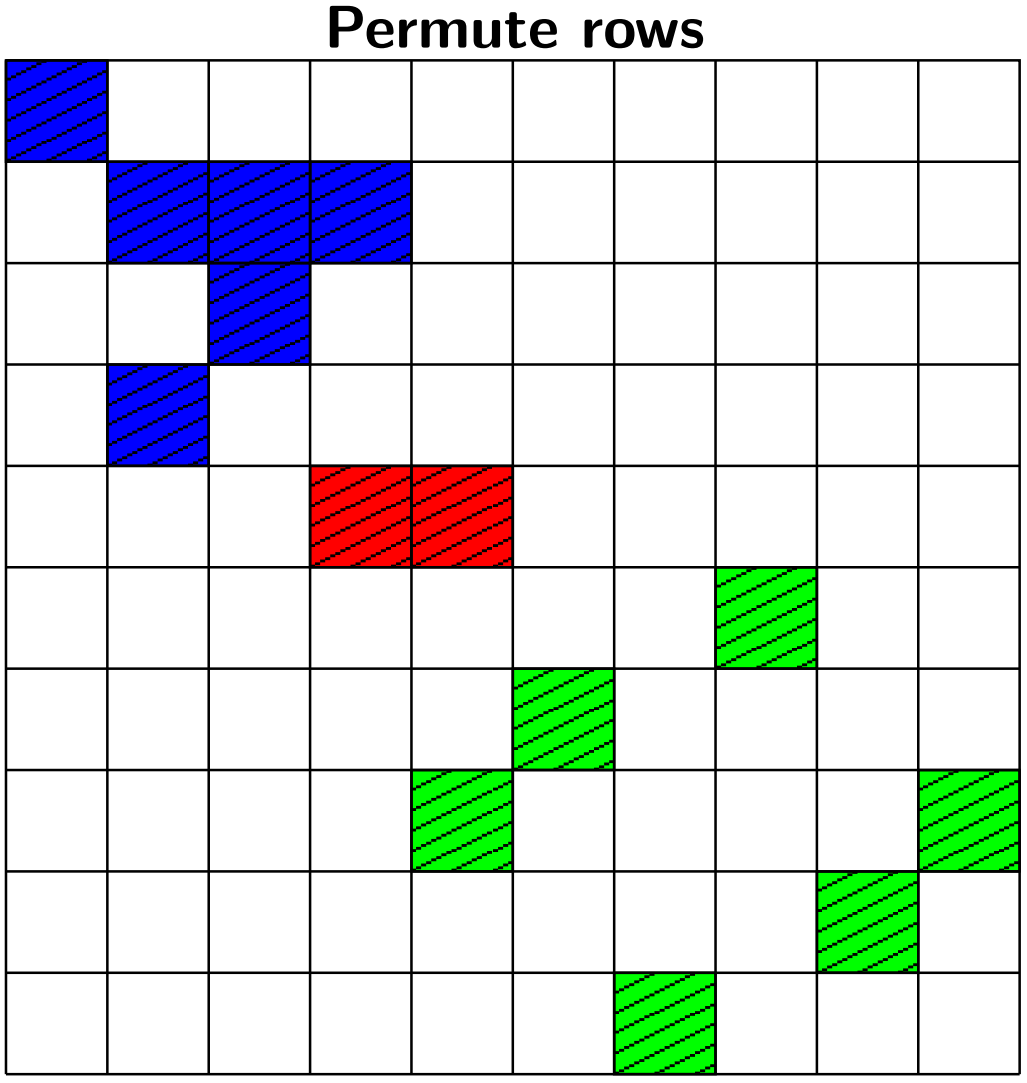


Using Mondriaan

Identify conflicting rows

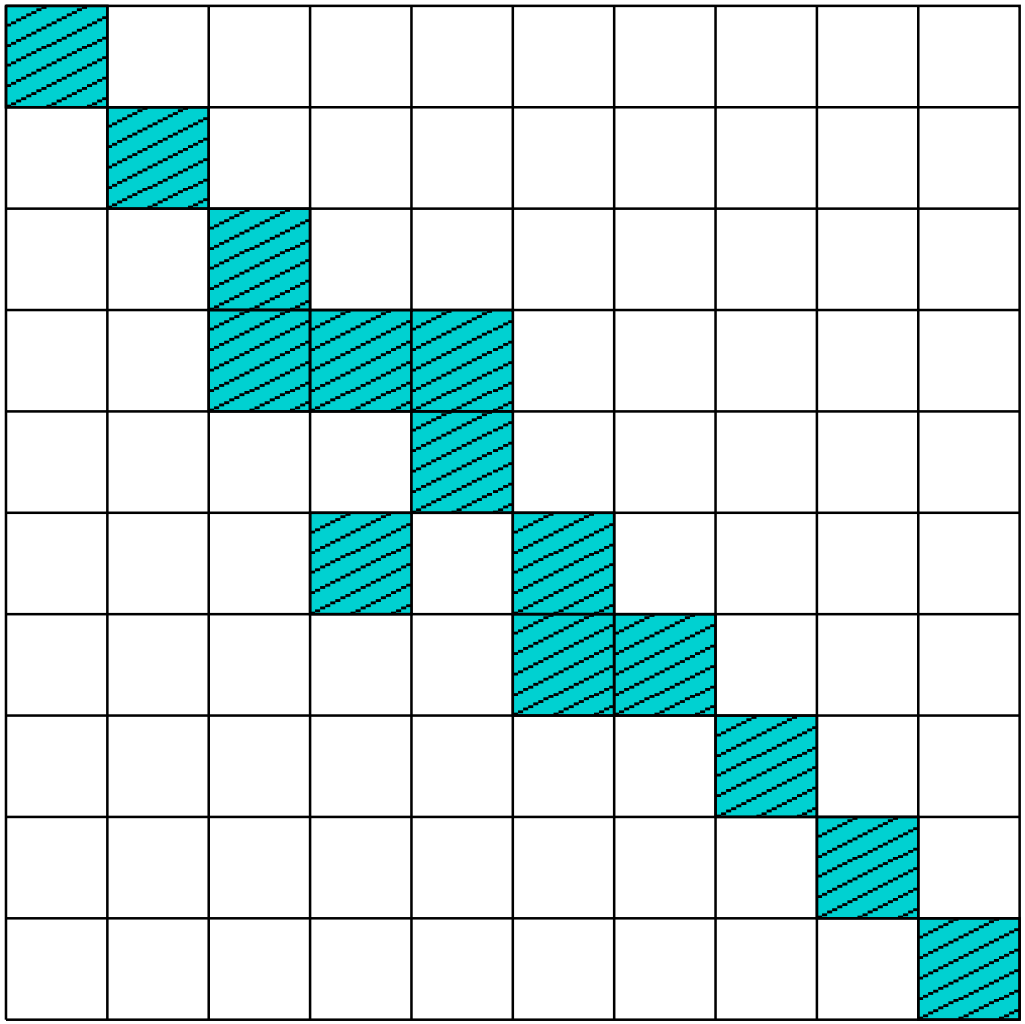


Using Mondriaan

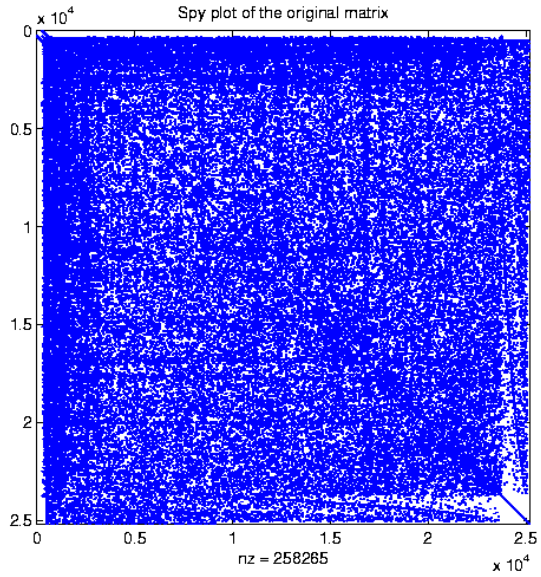


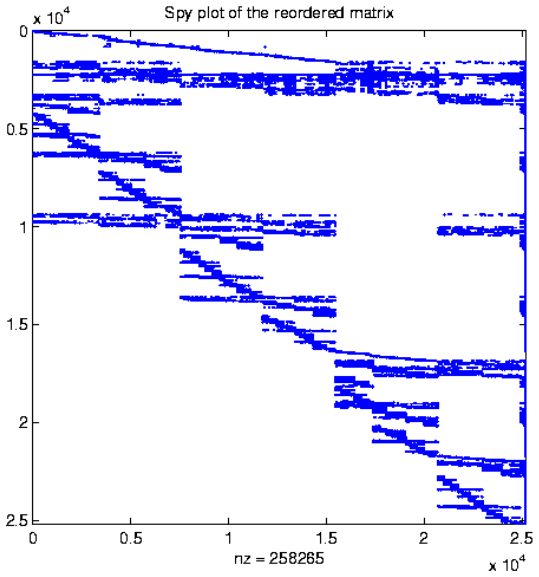
Using Mondriaan

...and recurse (until n partitions reached)



Cache-oblivious SpMV





The "rhpentium_new" matrix, reordered with $p = 100$, $\epsilon = 0.1$;
34 percent faster SpMV

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

Future

- Multicore processing (**MulticoreBSP**, shared caches)
- Further enhancing sequential SpMV (**two-dimensional SBD**)
- Faster/better Mondriaan (**better matching**, *other...*)
- Other uses for Mondriaan (*LU-decomposition*, ...)
- Parallel Mondriaan