Iterative sparse matrix partitioning

Supervisor: Prof. dr. Rob H. Bisseling

Davide Taviani

October 7th, 2013

Universiteit Utrecht

Parallel sparse matrix-vector multiplication

At the core of many iterative solvers (e.g. conjugate gradient method) lies a simple operation: **sparse matrix-vector multiplication**.

Given:

- $m \times n$ sparse matrix A (N nonzeros, $N \ll mn$)
- $n \times 1$ vector $\vec{v}$

we want to compute

$$\vec{u} = A\vec{v}$$

Parallel sparse matrix-vector multiplication

Usually A is fairly large and a lot of computations are required:

- $\mathcal{O}(mn)$ following the definition of matrix-vector multiplication;
- $\mathcal{O}(N)$ only considering the nonzero elements.

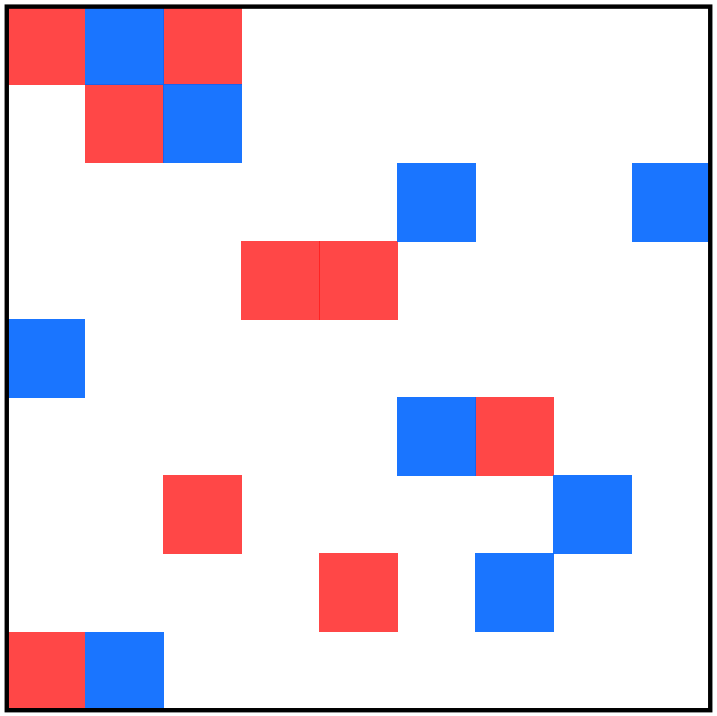
We split the computations among p processors to improve speed.

We make a **partition** of the set of the nonzeros of A, obtaining p disjoint sets $A_0, \ldots, A_{p-1}$.

Furthermore, also the input vector $\vec{v}$ and the final output $\vec{u}$ can be divided among those p processors (their distribution might not necessarily be the same).

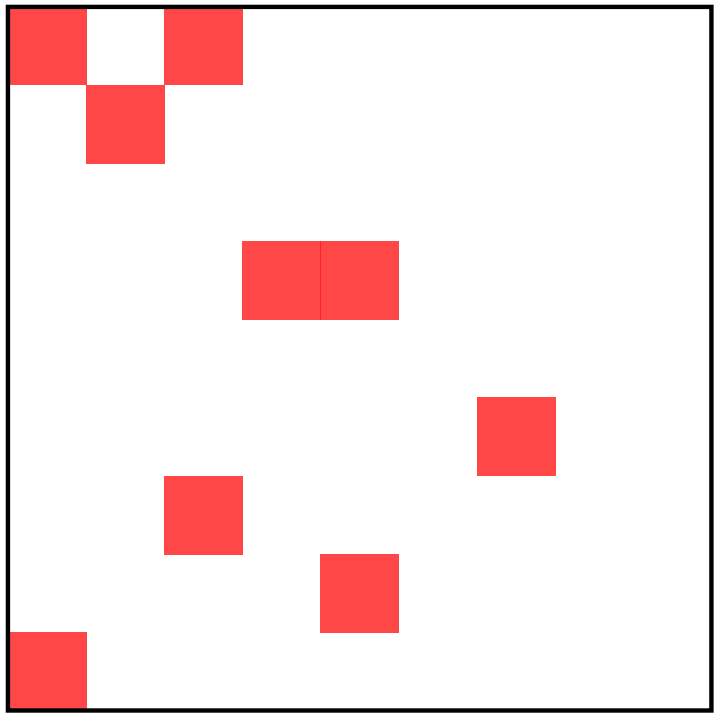
Matrix partitioning

Example of a partition of a 9×9 matrix with 18 nonzeros, with $p = 2$.

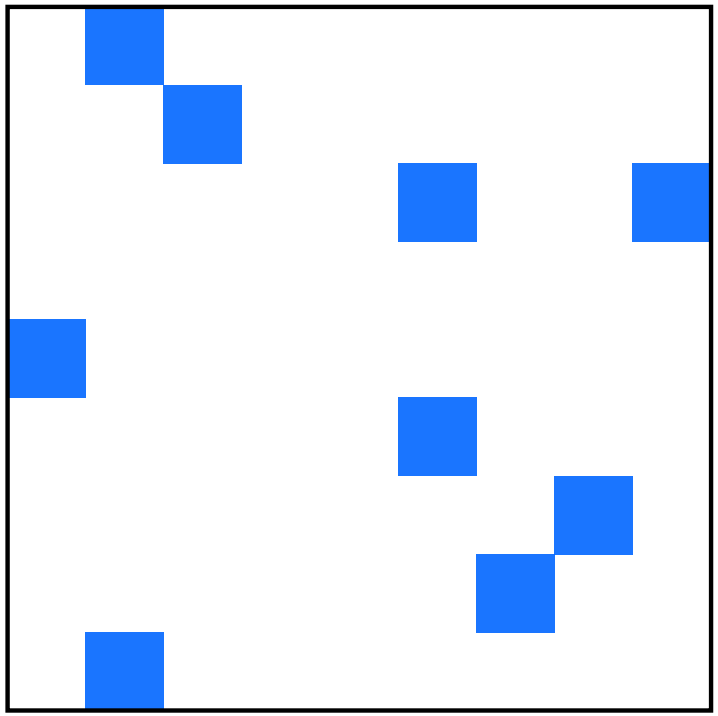


Matrix partitioning

Local view of the matrix for every processor:



$P(0)$



$P(1)$

Parallel matrix-vector multiplication algorithm

Parallel sparse matrix-vector multiplication is made (essentially) by 4 phases:

I) **fan-out**

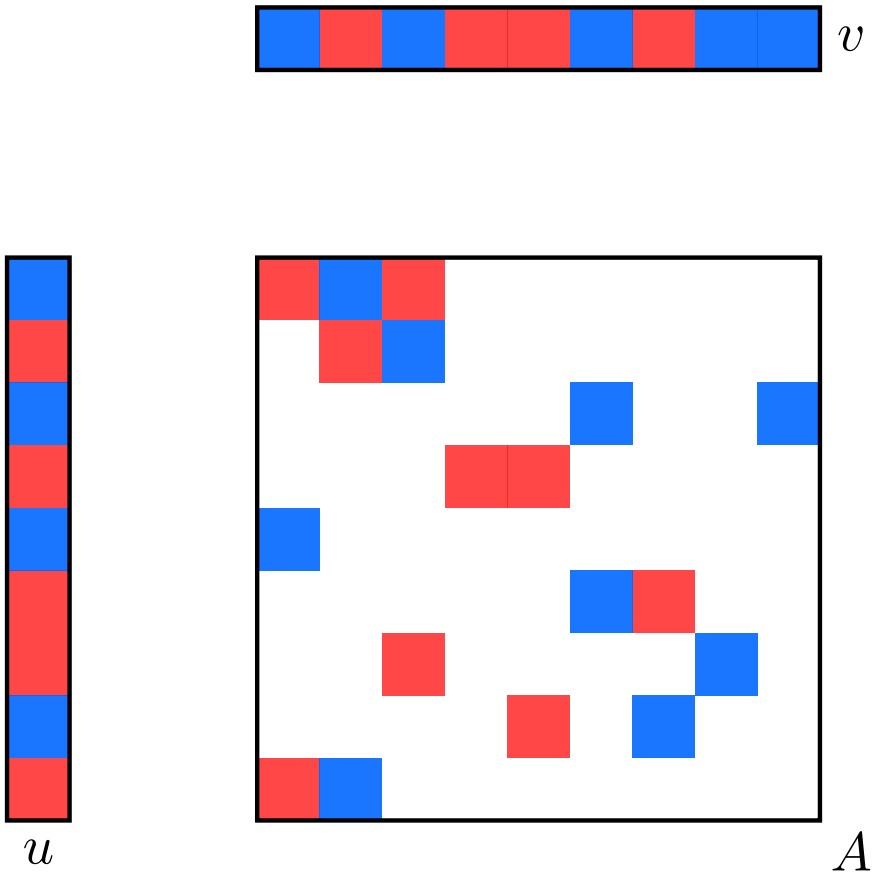
II) **local multiplication**

III) **fan-in**

IV) **sum of all the contributions**

Parallel matrix-vector multiplication algorithm

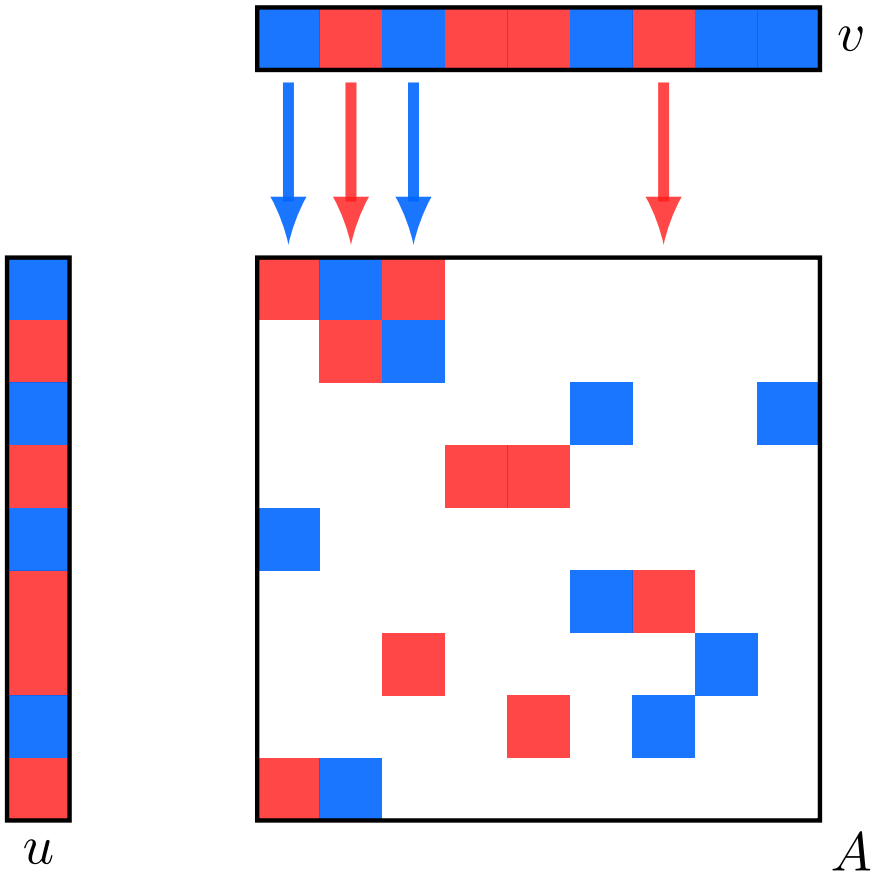
A is partitioned along with u and v



Universiteit Utrecht

Parallel matrix-vector multiplication algorithm

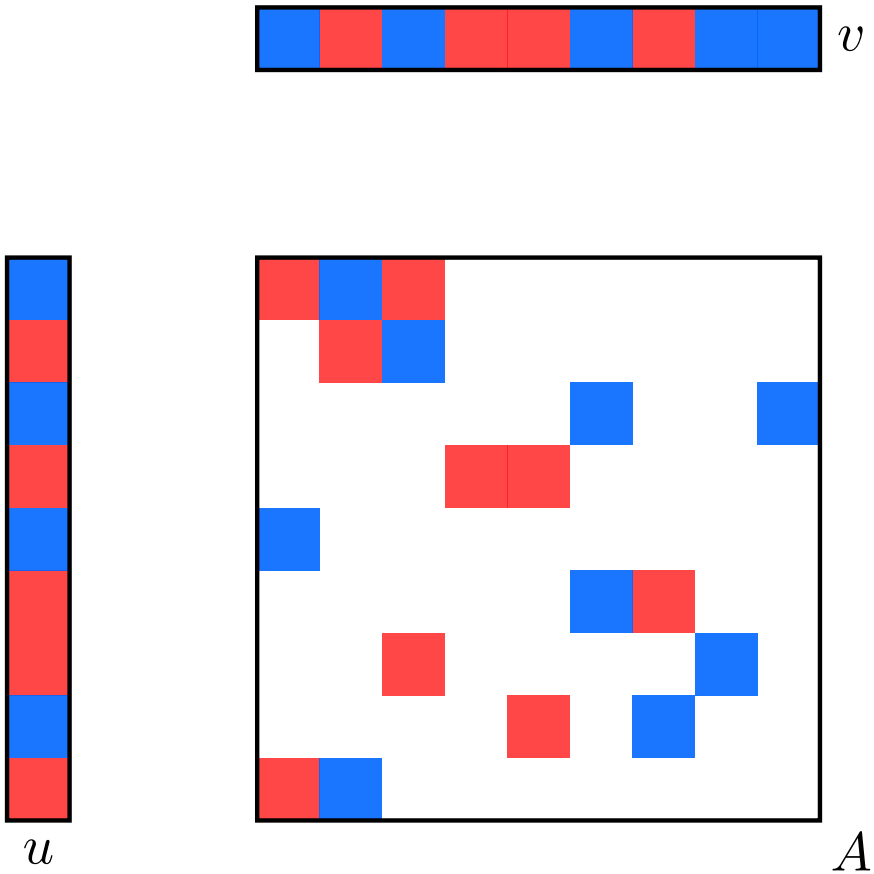
Fan-out: each processor receives the required elements of $\vec{v}$ from the others (according to its distribution)



Universiteit Utrecht

Parallel matrix-vector multiplication algorithm

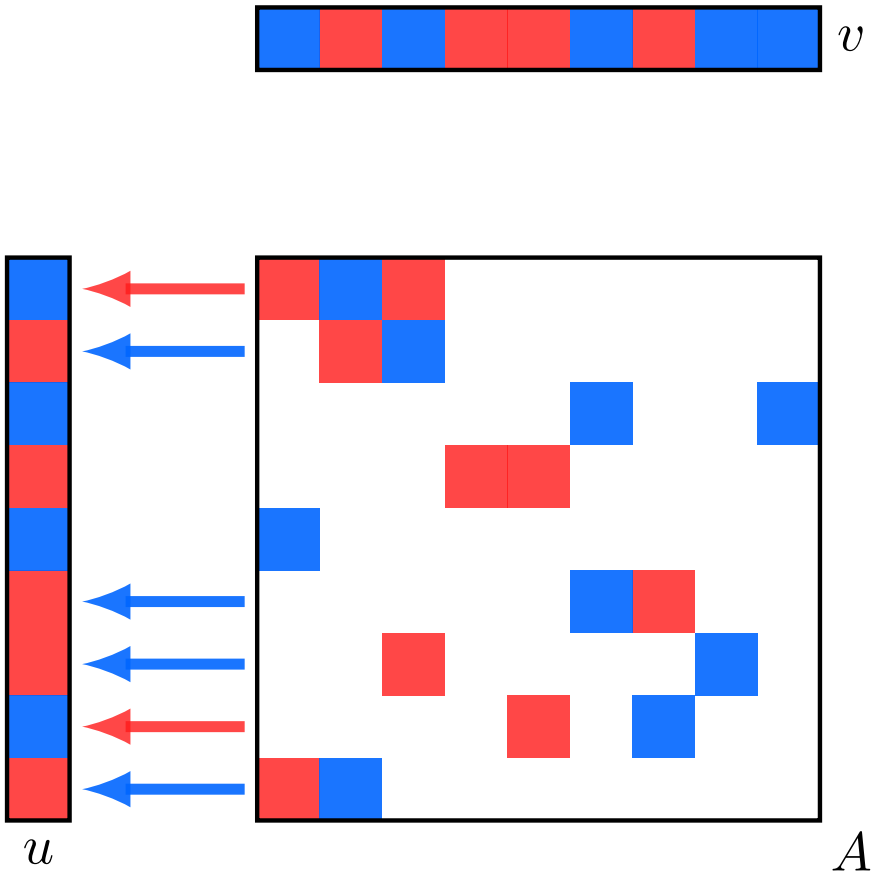
Local multiplication: where the actual computation is performed



Universiteit Utrecht

Parallel matrix-vector multiplication algorithm

Fan-in: where each processor sends his contributions to the other processors according to the distribution of $\vec{u}$



Universiteit Utrecht

Matrix partitioning

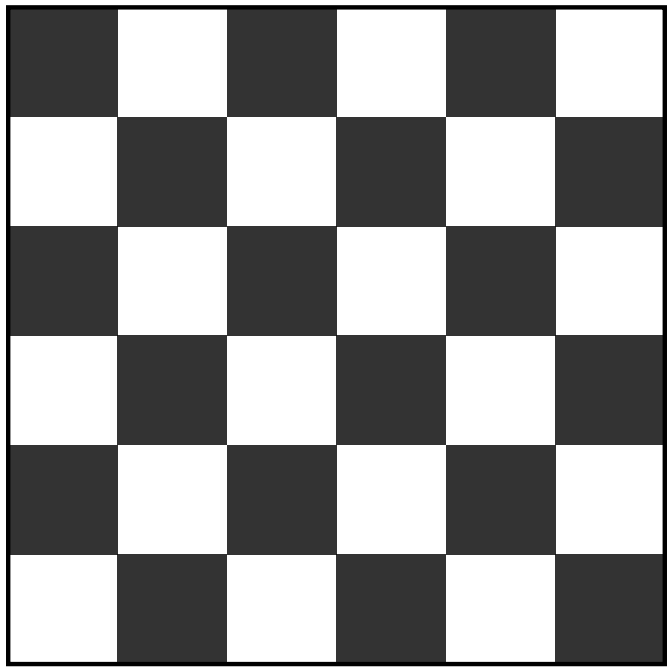
To optimize this process:

- I and III involve communication: it has to be **minimized**
- II is a computation step: we need **balance** in the size of the partitions

Optimization problem: partition the nonzeros such that the balance constraint is satisfied and the communication volume is minimized.

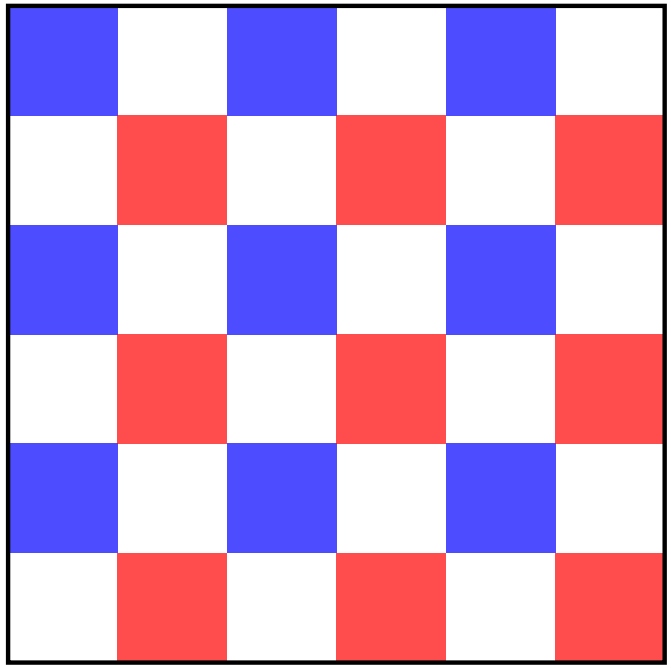
Matrix partitioning

As a last example, a 6×6 "checkerboard" matrix:

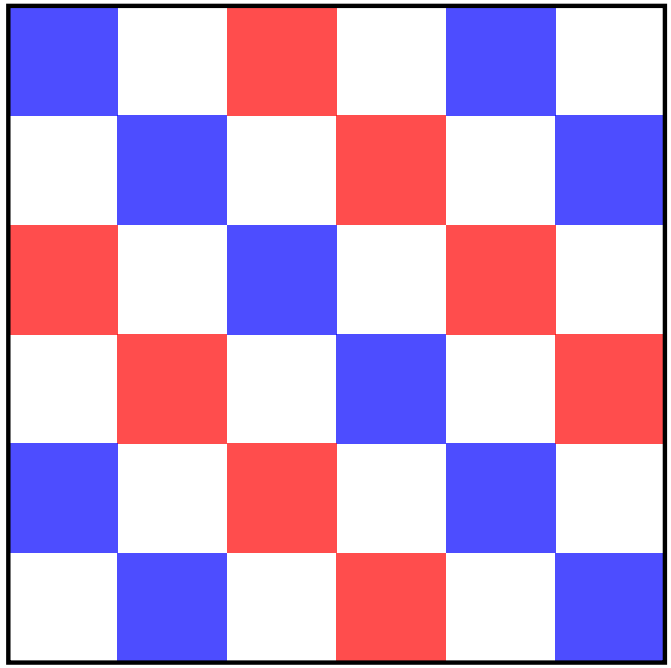


Matrix partitioning

Two different partitionings result in extremely different communication volumes.



(a) Rows and columns are not split, therefore there is no need for communication.



(b) Every row and column is split and causes communication during fan-in and fan-out.

Hypergraph partitioning

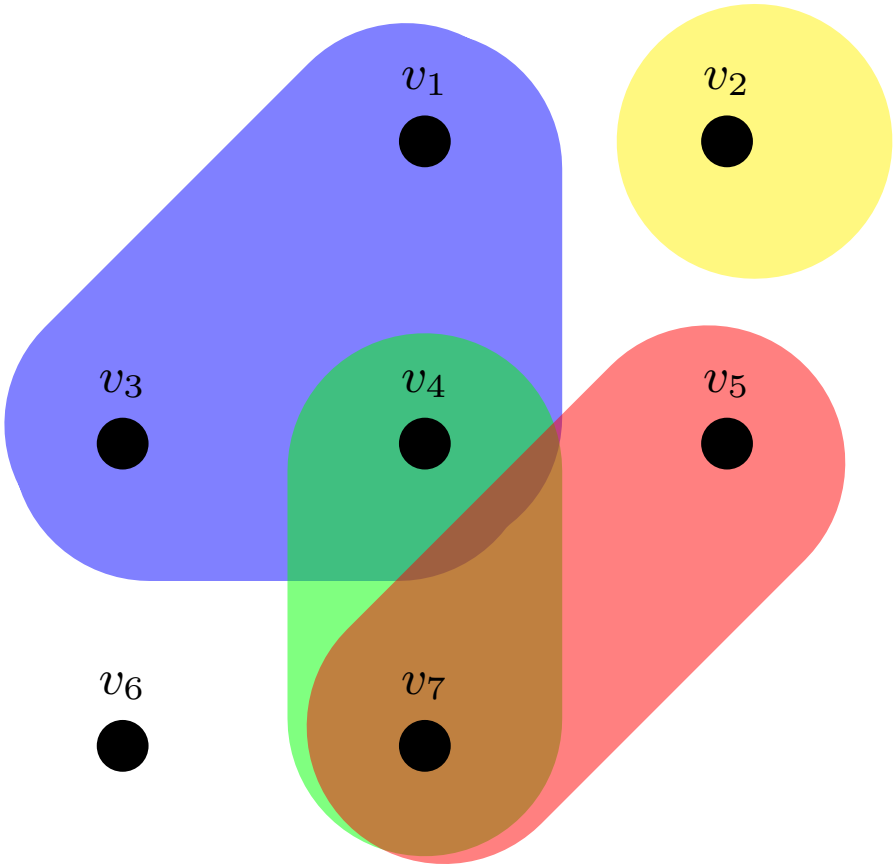
The matrix partitioning can be viewed from the graph theory point of view:

- Data represented as *vertices*
- Connections represented as *edges*

Further iteration: exact modeling of the matrix partitioning problem through **hypergraph partitioning**.

Hypergraph partitioning

Hypergraph: a graph in which a *hyperedge* can connect more than two *vertices* (i.e. a subset of the vertex set V)



$e_1 = \{v_1, v_3, v_4\}$

$e_2 = \{v_4, v_7\}$

$e_3 = \{v_5, v_7\}$

$e_4 = \{v_2\}$

Hypergraph partitioning

A partition of a hypergraph is simply the partition of the set of vertices V into $V_0, \ldots, V_{p-1}$.

A hyperedge $e = \{v_{e_1}, \ldots, v_{e_k}\}$ is **cut** if two of its vertices belong to different sets of the partition.

Hypergraph partitioning

There are several models to translate the matrix partitioning to hypergraph partitioning:

- 1-dimensional
  - **row-net**: each column of A is a vertex in the hypergraph, each row a hyperedge. If $a_{ij} \neq 0$, then column A_i is placed in the hyperedge j.
  - **column-net**: identical to the previous one, with the roles of columns and rows exchanged

As hypergraph partitioning consists in assignment of the vertices, columns/rows are uncut. Advantage of eliminating completely one source of communication, but being 1-dimensional is often a too strong restriction.

Hypergraph partitioning

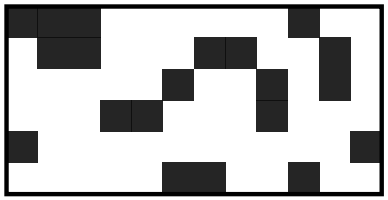
- 2-dimensional (each nonzero independent)
  - **fine grain**: nonzeros of A are vertices, rows and columns are hyperedges. The nonzero a_{ij} is placed in the hyperedges i and j

    A lot of freedom in partitioning (each nonzero can be assigned individually), but the size of the hypergraph (N vertices) is often too large.
  - **medium grain**: middle ground between 1-dimensional models and fine-grain

    Middle ground between 1-dimensional models and fine grain: good compromise between the size of the hypergraph and freedom during the partitioning.

Medium grain

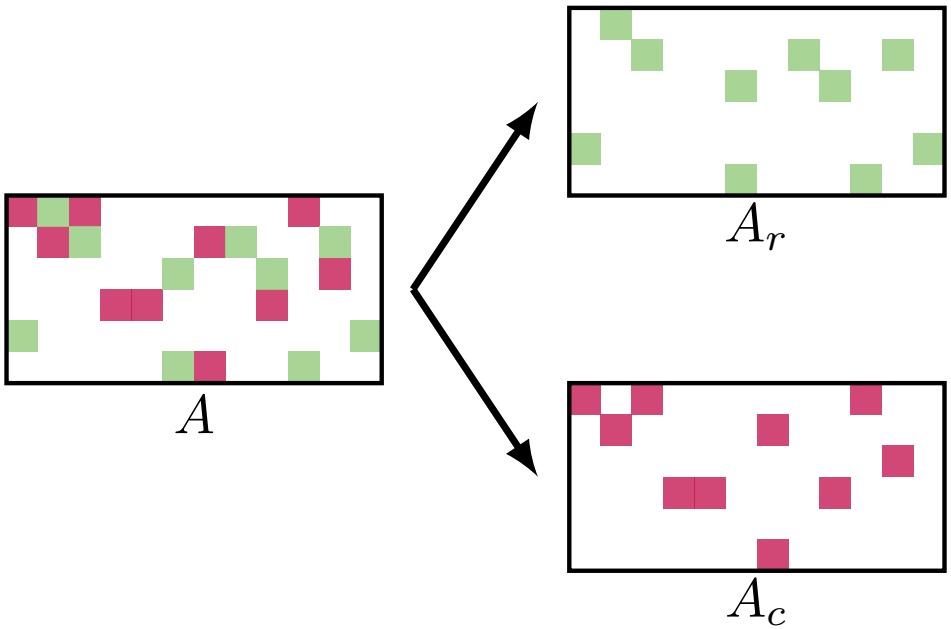
(Daniel M. Pelt and Rob Bisseling, 2013, to appear)



A

Medium grain

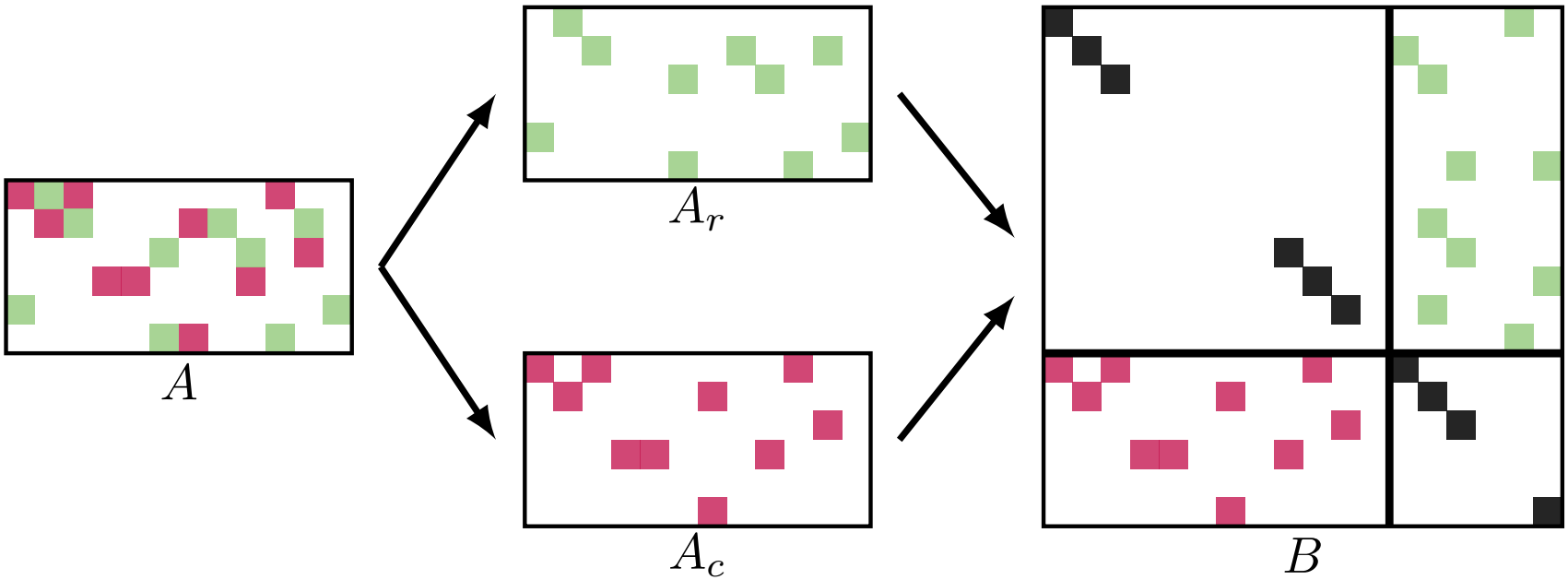
(Daniel M. Pelt and Rob Bisseling, 2013, to appear)



- Initial split of A into A^c and A^r

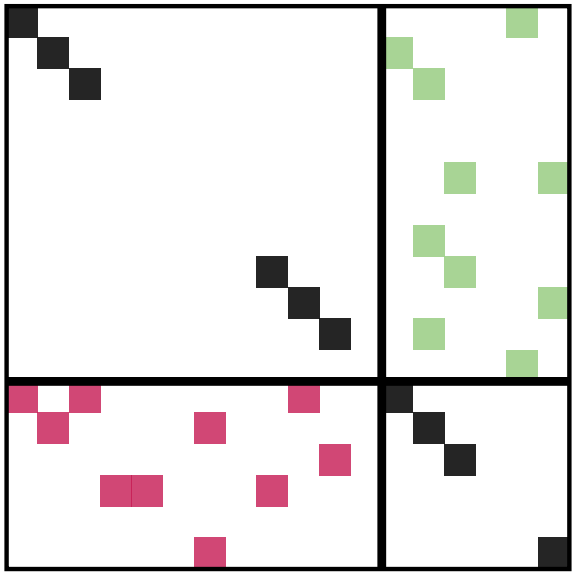
Medium grain

(Daniel M. Pelt and Rob Bisseling, 2013, to appear)

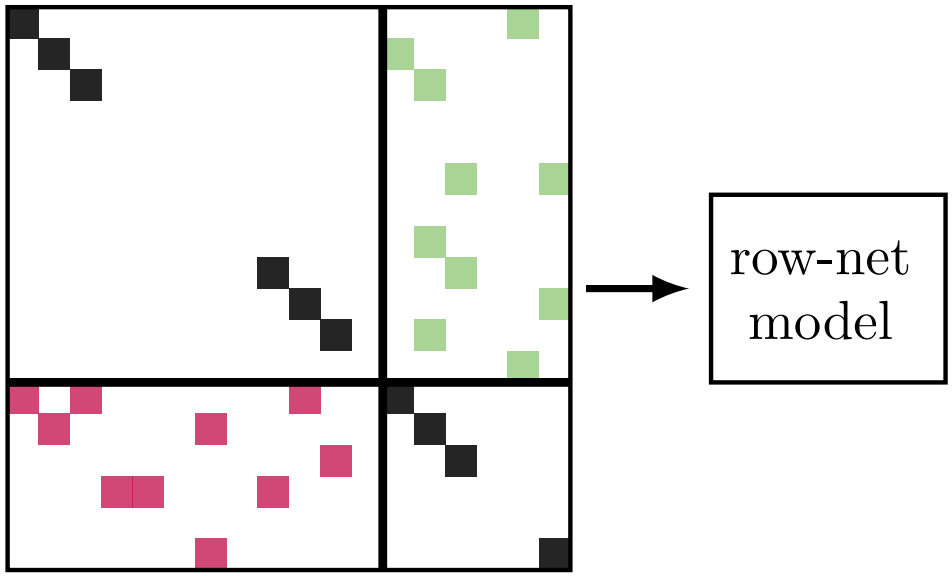


- Initial split of A into A^c and A^r
- Construction of the $(m+n) \times (m+n)$ matrix B (with dummy diagonal elements)

Medium grain

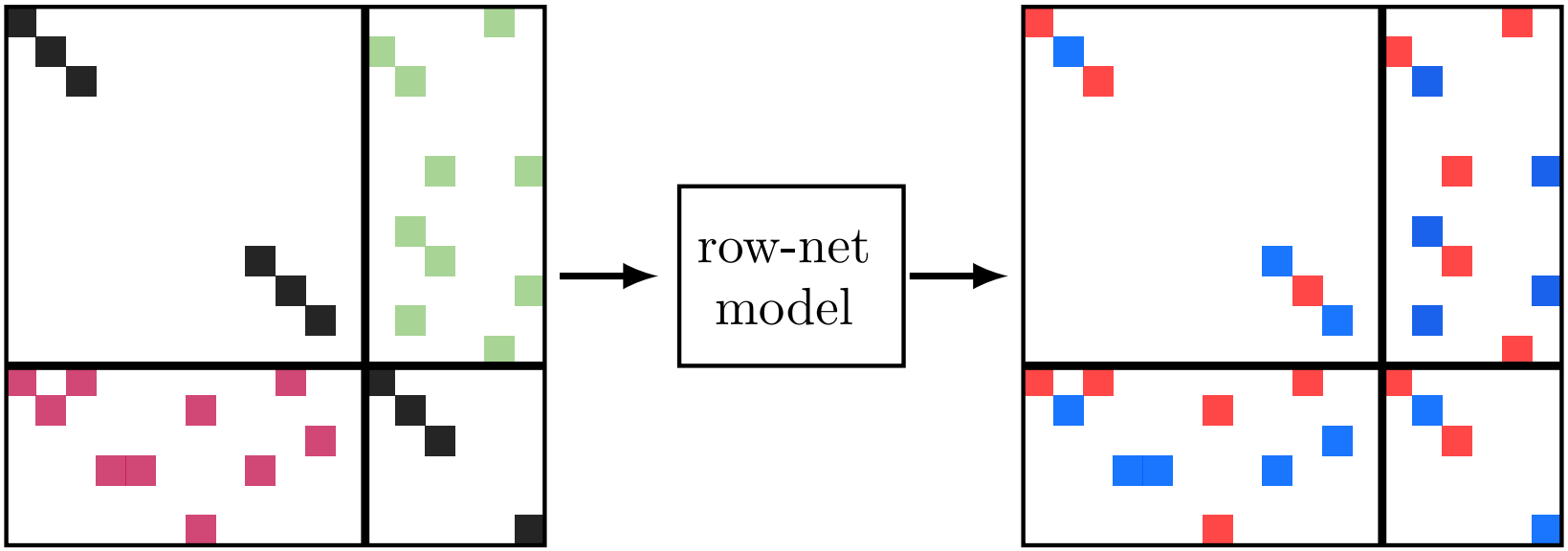


Medium grain



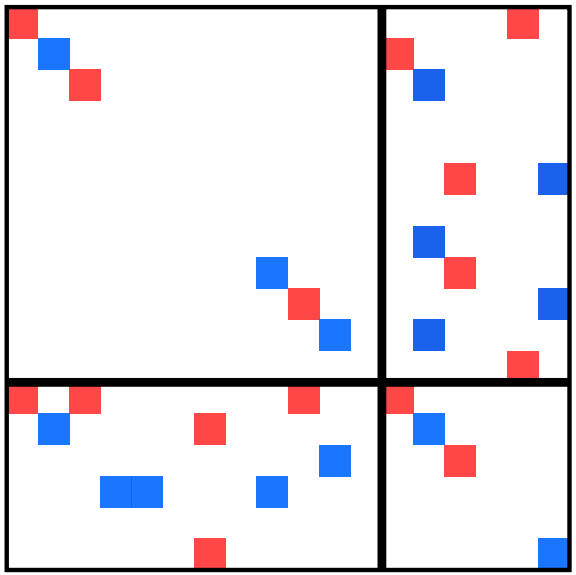
- Partitioning of B with the row-net model (column are kept together)

Medium grain



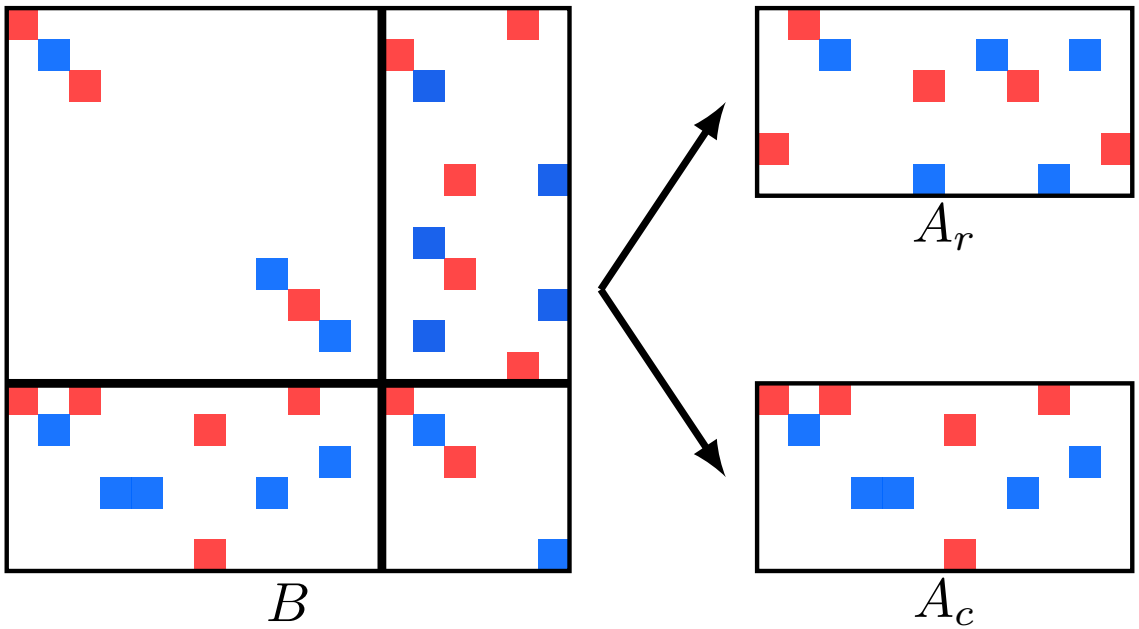
- Partitioning of B with the row-net model (column are kept together)

Medium grain



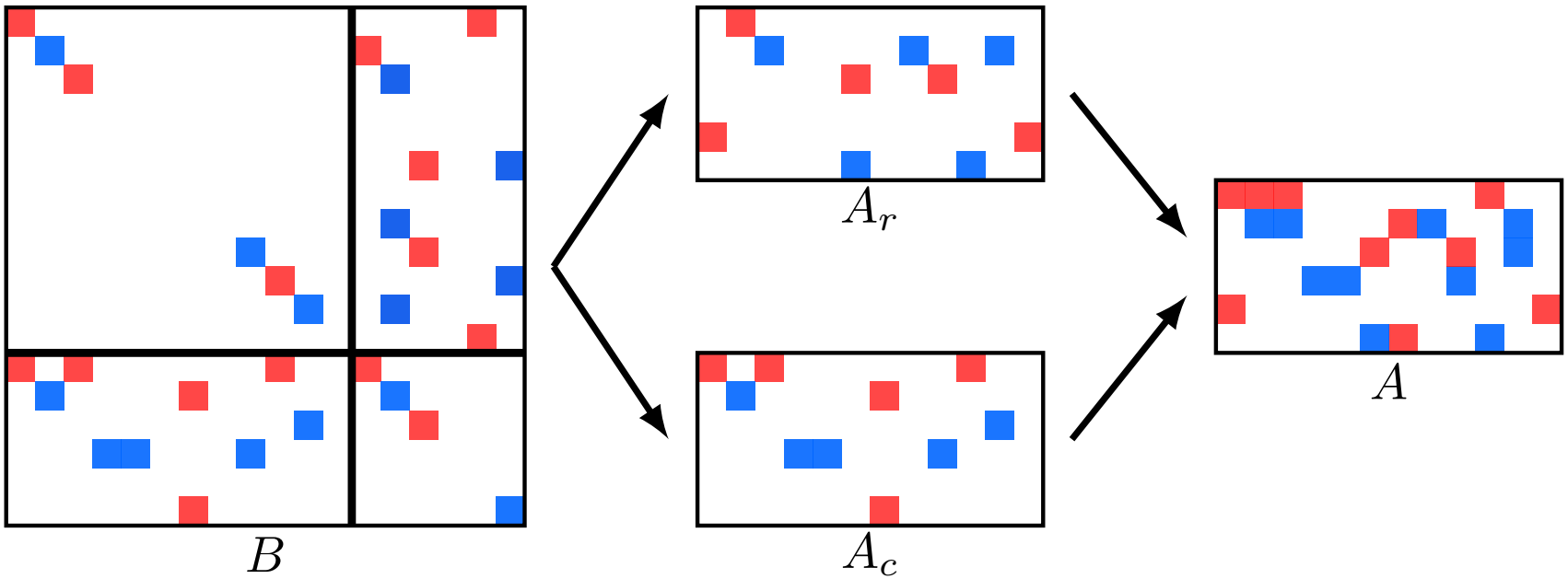
B

Medium grain



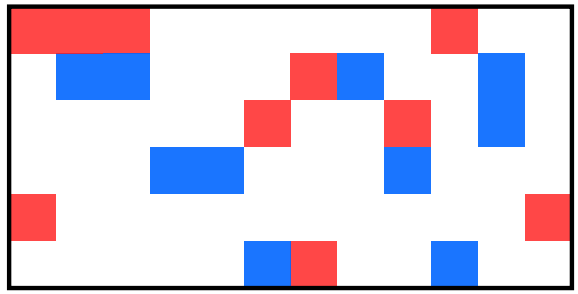
- Retrieval of A_r and A_c with the new partitioning

Medium grain



- Retrieval of A_r and A_c with the new partitioning
- Reassembling of A

Medium grain



A

Clusters of nonzeros are grouped together:

- in A_r we kept together elements of the same row;
- in A_c elements of the same column.

Research directions

Two research directions:

- Improving the initial partitioning of A
- Development of a fully iterative scheme: lowering the communication value by using information on the previous partitioning

These directions can be combined: we can try to find efficient ways of splitting A into A_r and A_c, distinguishing between

- *partition-oblivious* heuristics: no prior information is required
- *partition-aware* heuristics: requirement of A already partitioned

General remarks

A few general principles to guide us in the construction of the heuristics:

- short rows/columns (w.r.t. the number of nonzeros) are more likely to be uncut in a good partitioning
- if a row/column is uncut, the partitioner decided at the previous iteration that it was convenient to do so.
  We shall try to keep, as much as possible, those rows/columns uncut again.

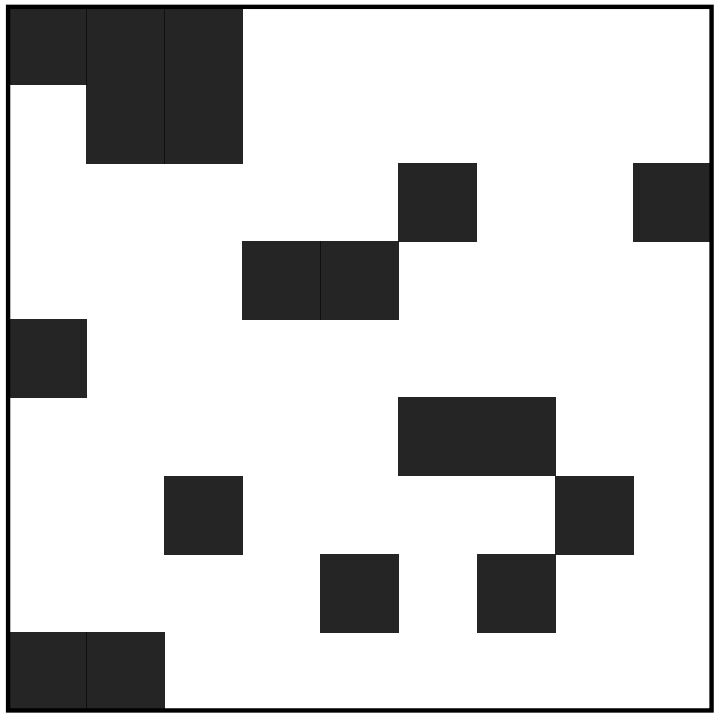
Individual assignment of nonzeros

A simple heuristic is the extension of the original algorithm used in medium-grain.

Partition-oblivious version:

for all $a_{ij} \in A$ **do**
  if $nz_r(i) < nz_c(j)$ **then**
    assign a_{ij} to A_r
  else if $nz_c(j) < nz_r(i)$ **then**
    assign a_{ij} to A_c
  else
    assign a_{ij} to according to tie-breaker
  end if
end for

Individual assignment of nonzeros

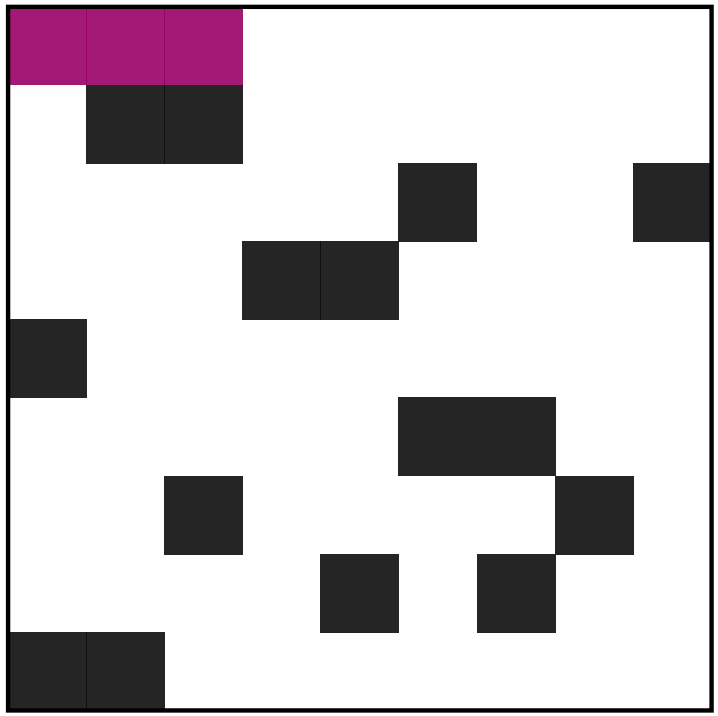


A_c

A_r

tie-breaking: A_c

Individual assignment of nonzeros

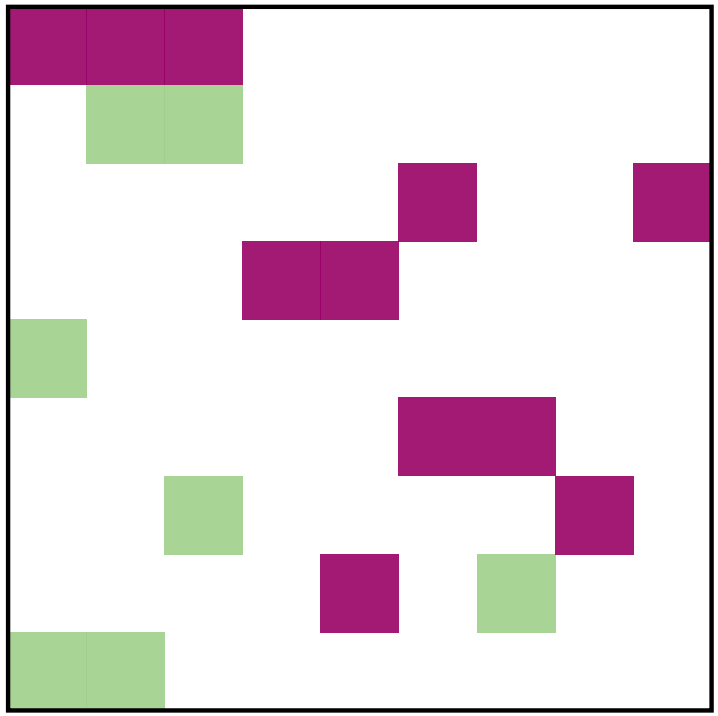


A_c

A_r

tie-breaking: A_c

Individual assignment of nonzeros



A_c

A_r

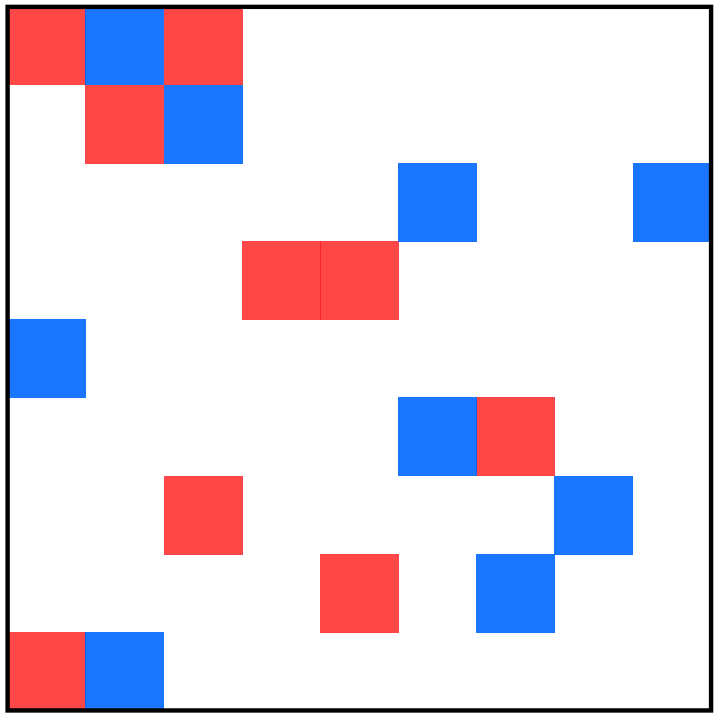
tie-breaking: A_c

Individual assignment of nonzeros

Partition-aware version:

for all $a_{ij} \in A$ **do**
  if row i is uncut and column j is cut **then**
    assign a_{ij} to A_r
  else if row i is cut and column j is uncut **then**
    assign a_{ij} to A_c
  else
    assign a_{ij} as in the partition-oblivious variant
  end if
end for

Individual assignment of nonzeros

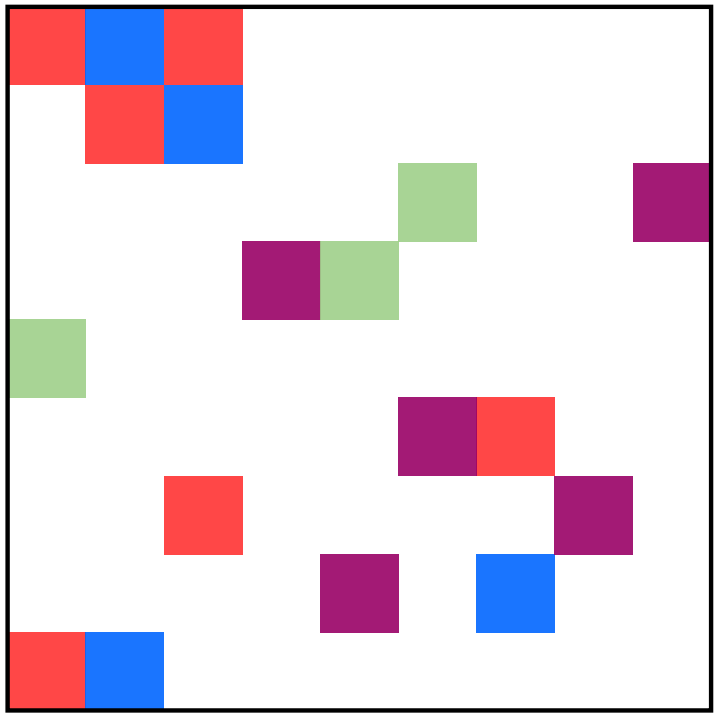


A_c

A_r

tie-breaking: A_r

Individual assignment of nonzeros

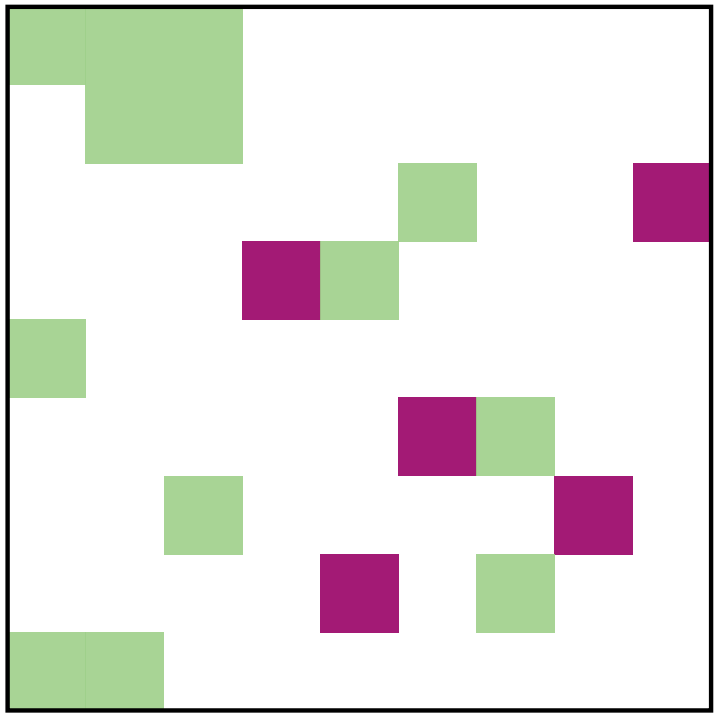


A_c

A_r

tie-breaking: A_r

Individual assignment of nonzeros



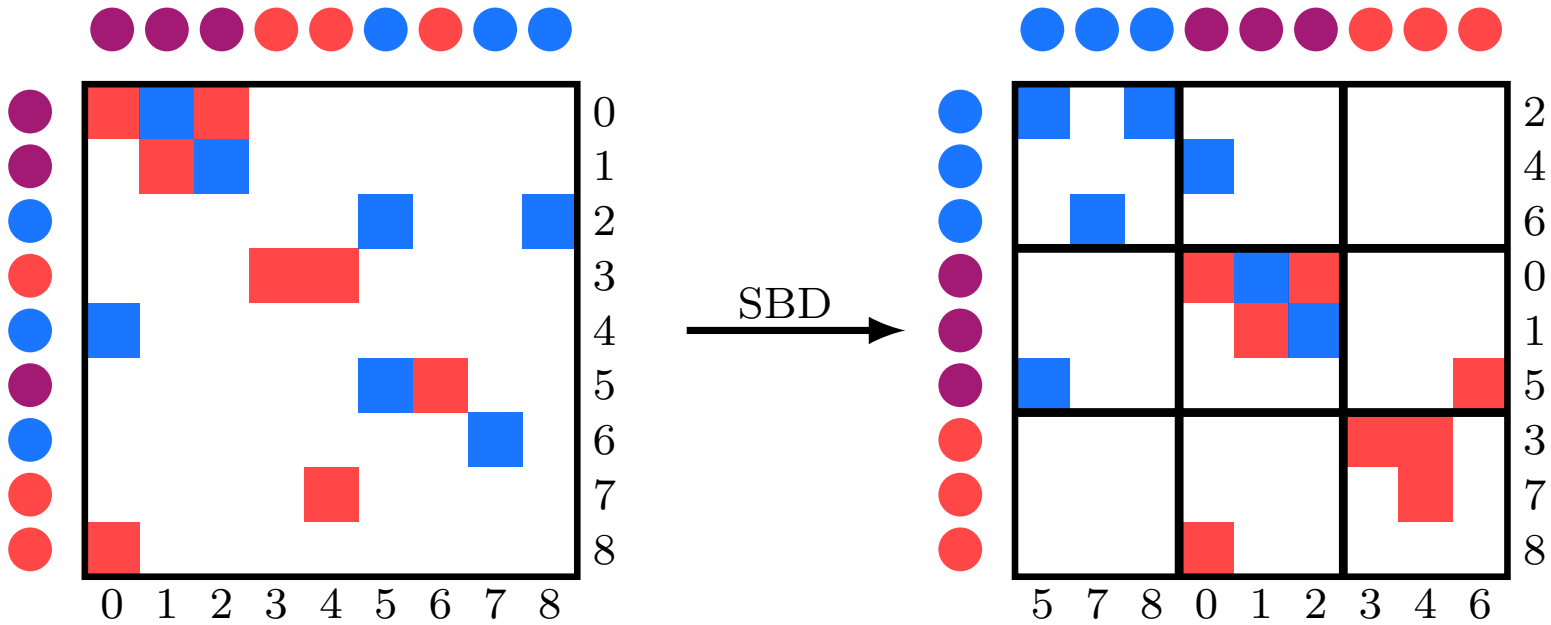
A_c

A_r

tie-breaking: A_r

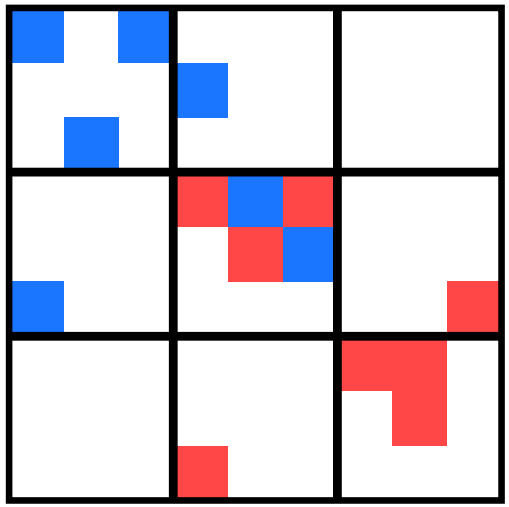
Assignment of blocks of nonzeros

Separated Block Diagonal (SBD) form of a partitioned matrix: we separate uncut and cut rows and columns.



Universiteit Utrecht

Assignment of blocks of nonzeros

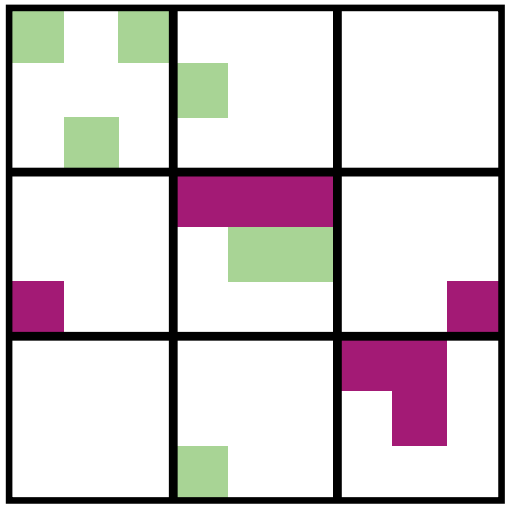


The SBD form is a 3×3 block matrix

$$\begin{bmatrix} \dot{A}_{00} & \dot{A}_{01} & \\ \dot{A}_{10} & \dot{A}_{11} & \dot{A}_{12} \\ & \dot{A}_{21} & \dot{A}_{22} \end{bmatrix}$$

$\dot{A}_{01}$, $\dot{A}_{10}$, $\dot{A}_{12}$, $\dot{A}_{21}$ can be easily assigned in our framework.

Assignment of blocks of nonzeros



$$\begin{bmatrix} A_r/A_c & A_r & \\ A_c & M & A_c \\ & A_r & A_r/A_c \end{bmatrix}$$

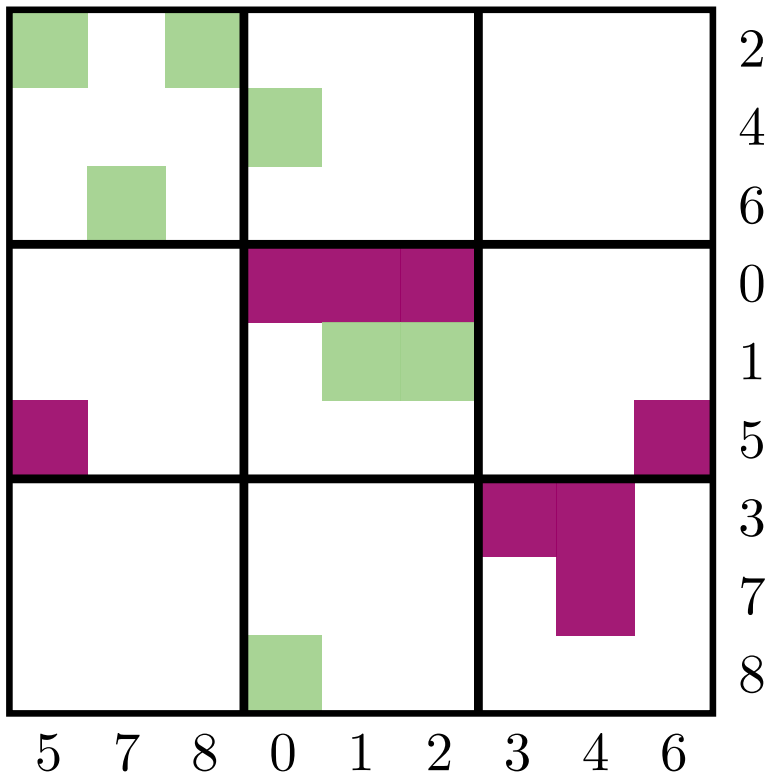
A_c

A_r

$\dot{A}_{01}$, $\dot{A}_{10}$, $\dot{A}_{12}$, $\dot{A}_{21}$ can be easily assigned in our framework.

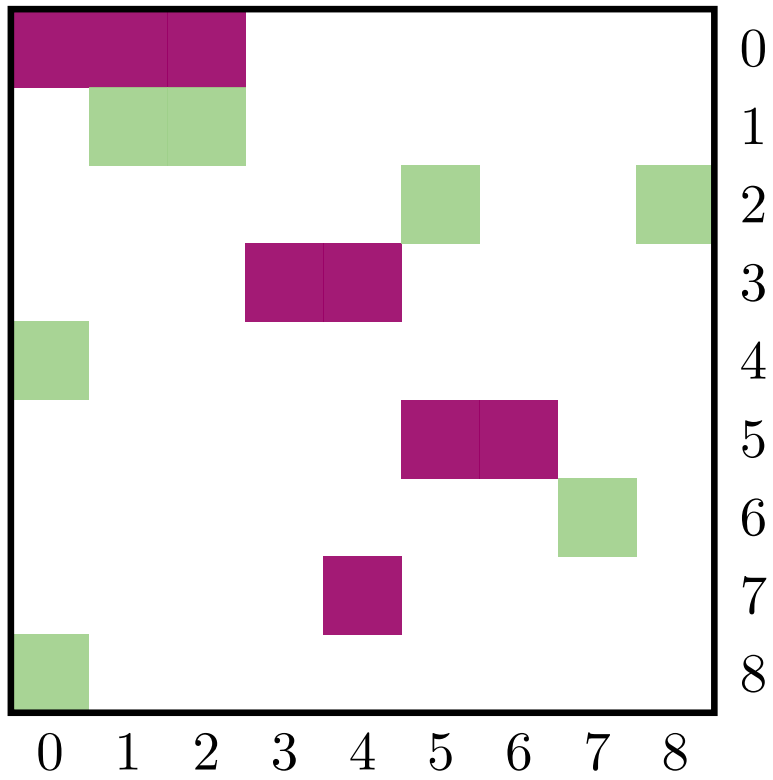
- A_r/A_c means that the size of the block determines whether it is assigned to A_r or A_c;
- the nonzeros in the middle block are assigned individually (M stands for "mixed" assignment)

Assignment of blocks of nonzeros



Universiteit Utrecht

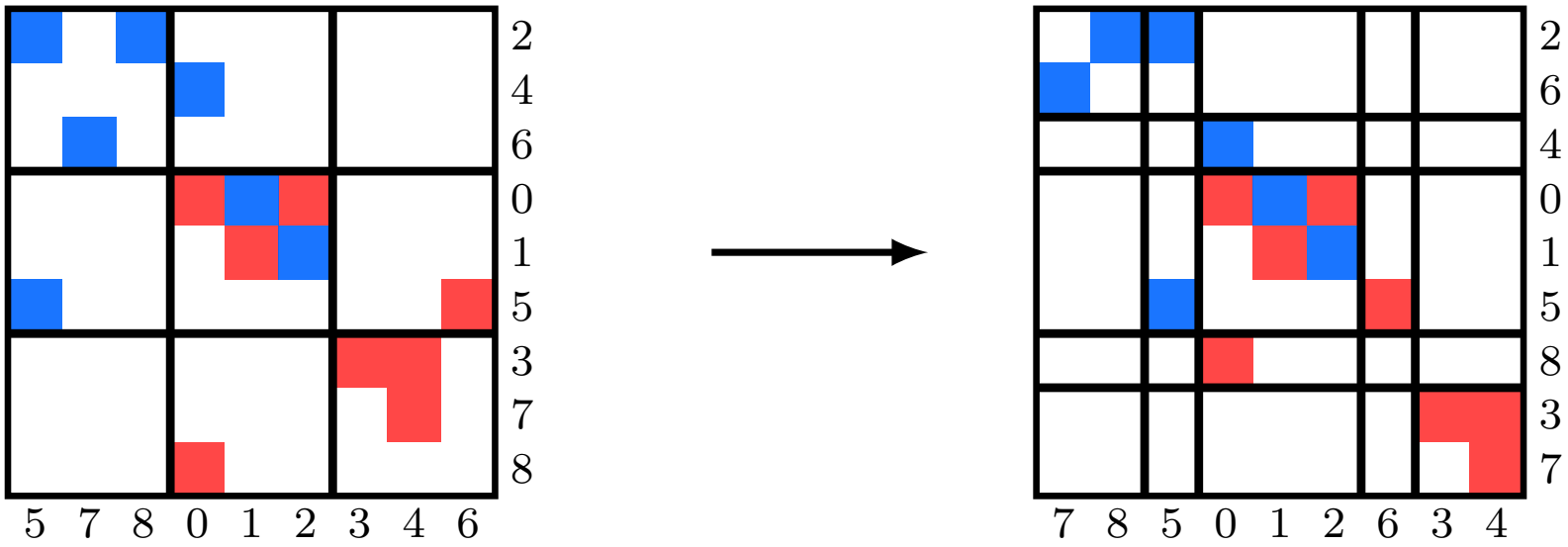
Assignment of blocks of nonzeros



- We reverse the permutations of rows and columns, obtaining A back, with new assignment.

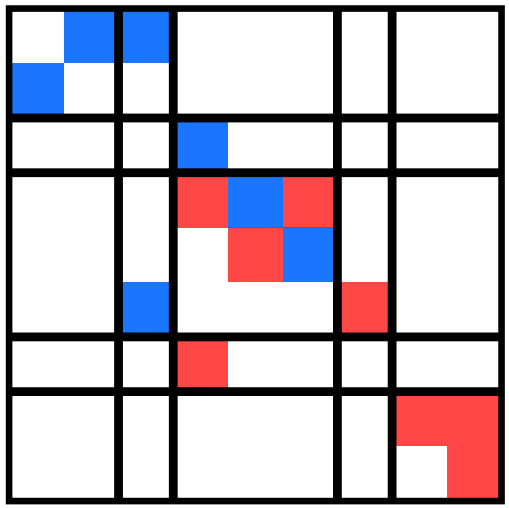
Assignment of blocks of nonzeros

Separated Block Diagonal form of order 2 (SBD2) of a matrix: we split the top, bottom, left and right blocks, separating the empty and nonempty parts.



Universiteit Utrecht

Assignment of blocks of nonzeros

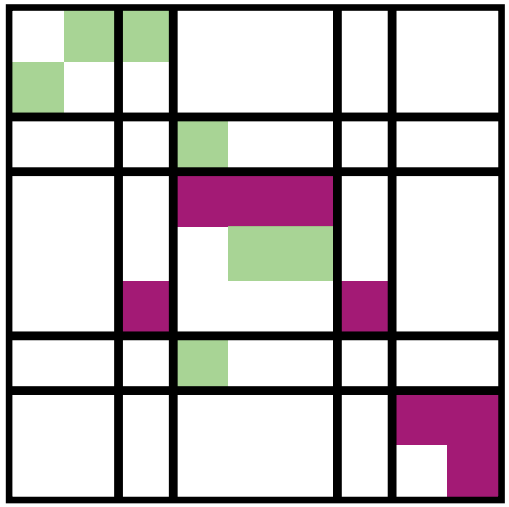


The SBD2 form of a matrix is the following 5×5 block matrix:

$$\begin{bmatrix} \ddot{A}_{00} & \ddot{A}_{01} & & & \\ \ddot{A}_{10} & \ddot{A}_{11} & \ddot{A}_{12} & & \\ & \ddot{A}_{21} & \ddot{A}_{22} & \ddot{A}_{23} & \\ & & \ddot{A}_{32} & \ddot{A}_{33} & \ddot{A}_{34} \\ & & & \ddot{A}_{43} & \ddot{A}_{44} \end{bmatrix}$$

In this form, other than having information on nonzeros (rows/columns cut/uncut), we also have informations on their neighbors (nonzeros in the same row and column).

Assignment of blocks of nonzeros



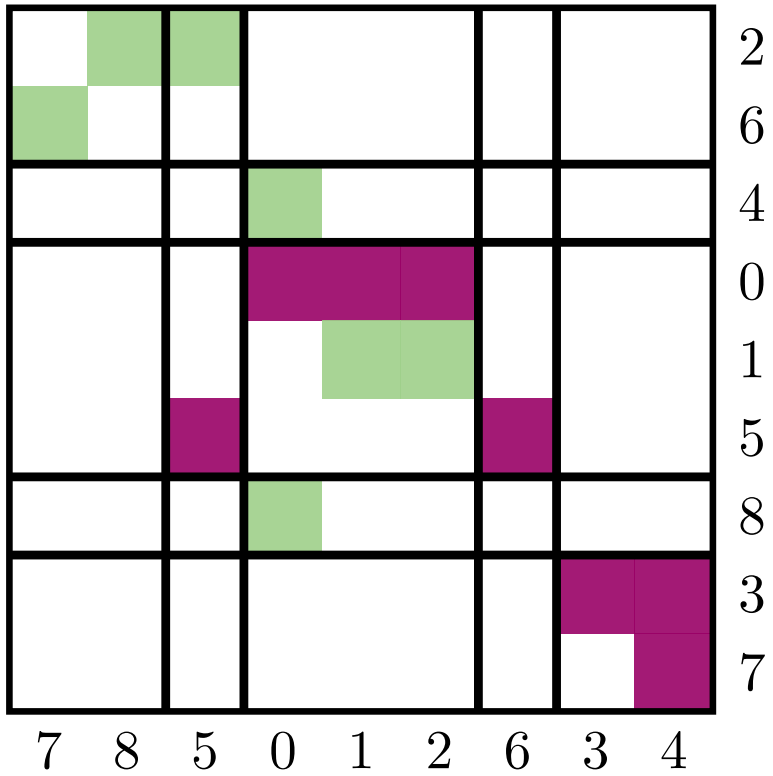
$$\begin{bmatrix} R & R & & & \\ C & R/C & R & & \\ & C & M & C & \\ & & R & R/C & C \\ & & & R & C \end{bmatrix}$$

A_c

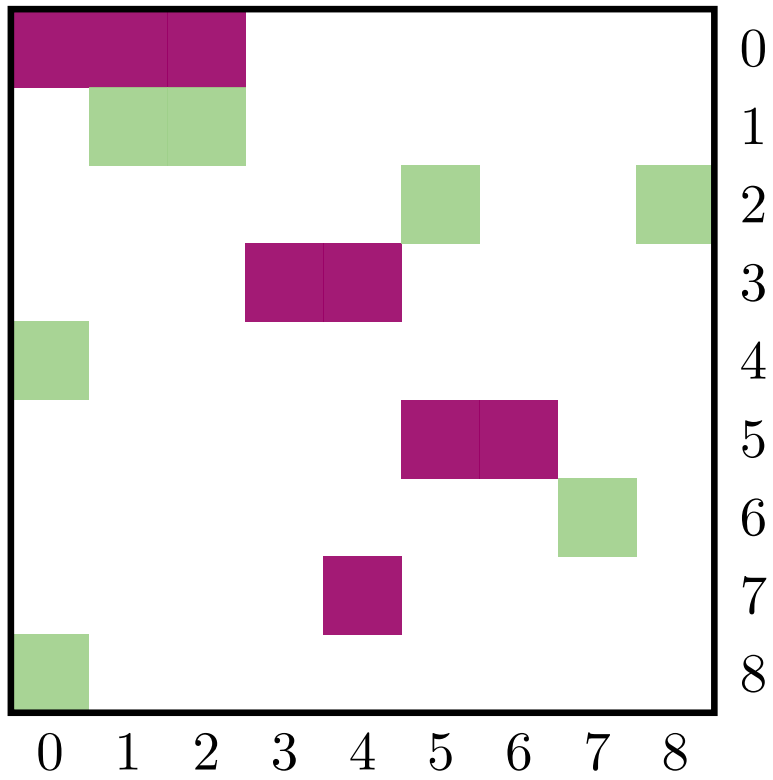
A_r

In this form, other than having information on nonzeros (rows/columns cut/uncut), we also have informations on their neighbors (nonzeros in the same row and column).

Individual assignment of blocks of nonzeros



Individual assignment of blocks of nonzeros



- We reverse the permutations of rows and columns, obtaining A back, with new assignment.

Partial assignment of rows and columns

- Main idea: Every time we assign a nonzero to either A_r or A_c, all the other nonzeros in the same row/column should be assigned to it as well, to prevent communication.
- Main issue: Hard to assign complete rows/column: a nonzero cannot be assigned to both A_r and A_c.

We need to reason in terms of **partial assignment**:

- computation of a **priority vector**: a permutation of the indices $\{0, \ldots, m+n-1\}$ (indices $\{0, \ldots, m-1\}$ correspond to rows, $\{m, \ldots, m+n-1\}$ to columns);
- **overpainting algorithm**.

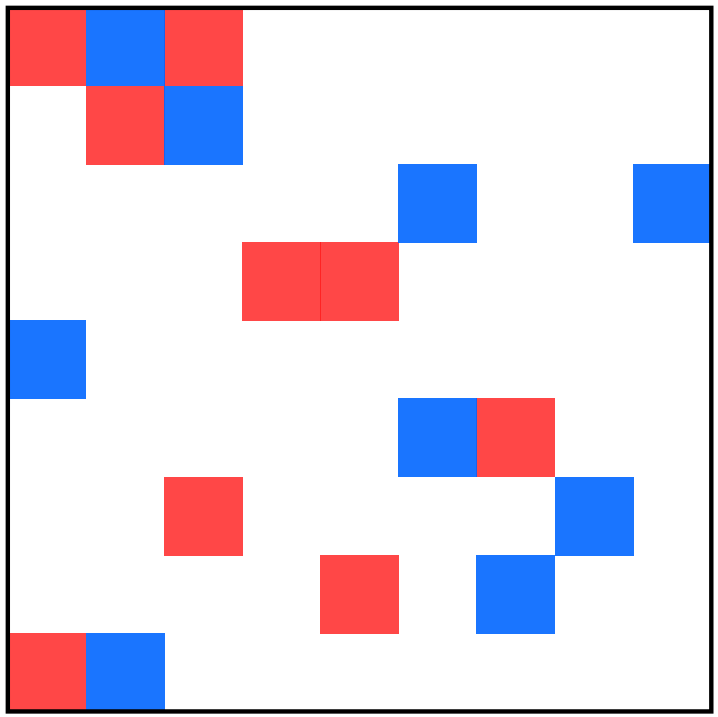
Overpainting algorithm

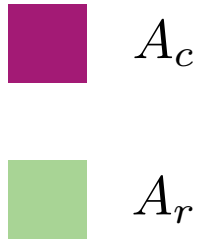
Require: Priority vector v, matrix A
Ensure: A_r, A_c
  $A_r := A_c := \varnothing$
  for $i = m + n - 1, \ldots, 0$ **do**
    if $v_i < m$ **then**
      Add the nonzeros of row i to A_r
    else
      Add the nonzeros of column $i - m$ to A_c
    end if
  end for

- In this formulation of the algorithm, every nonzero is assigned twice;
- the algorithm is **completely deterministic**: A_r and A_c depend entirely on the priority vector v.

Overpainting algorithm

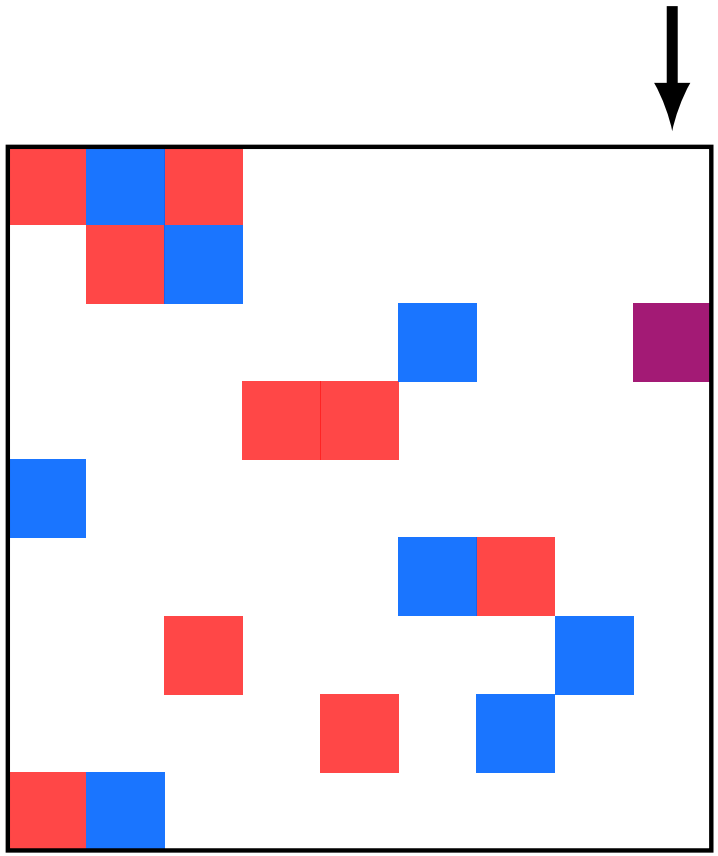
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$





Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



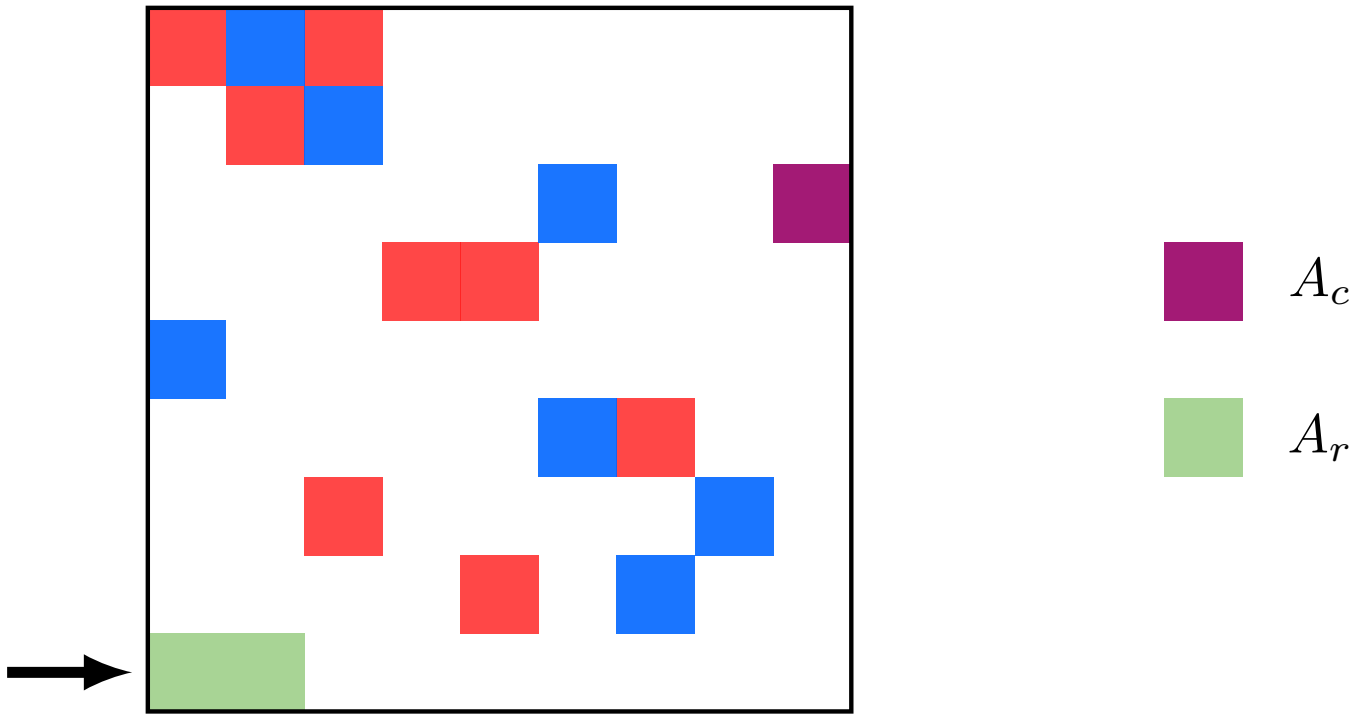
A_c

A_r

Universiteit Utrecht

Overpainting algorithm

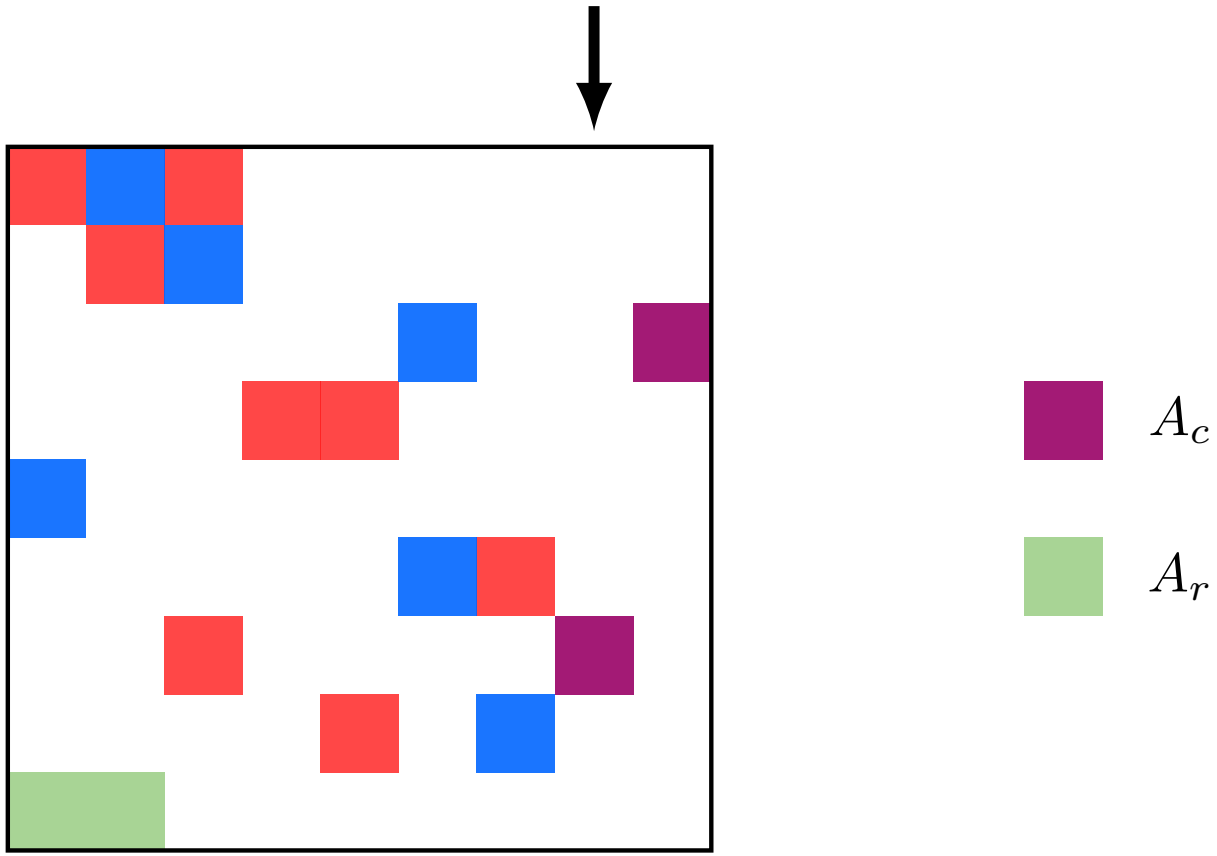
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Universiteit Utrecht

Overpainting algorithm

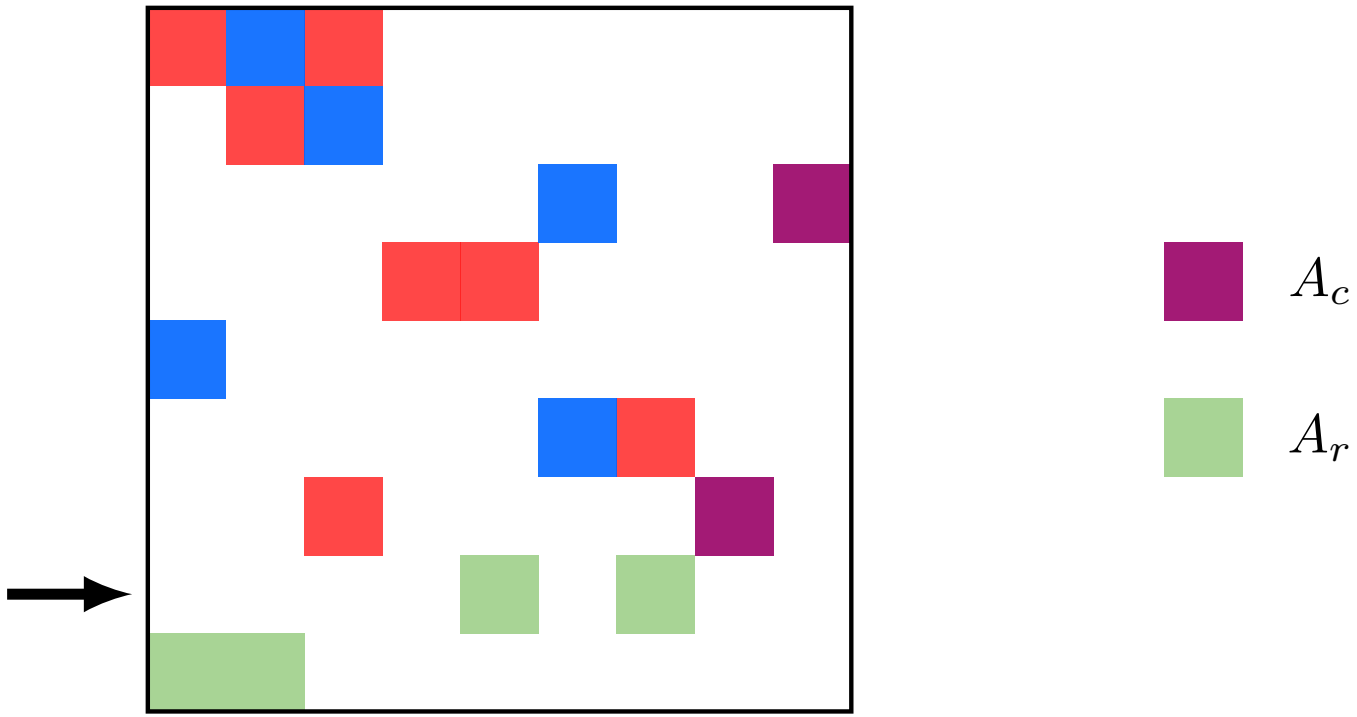
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Universiteit Utrecht

Overpainting algorithm

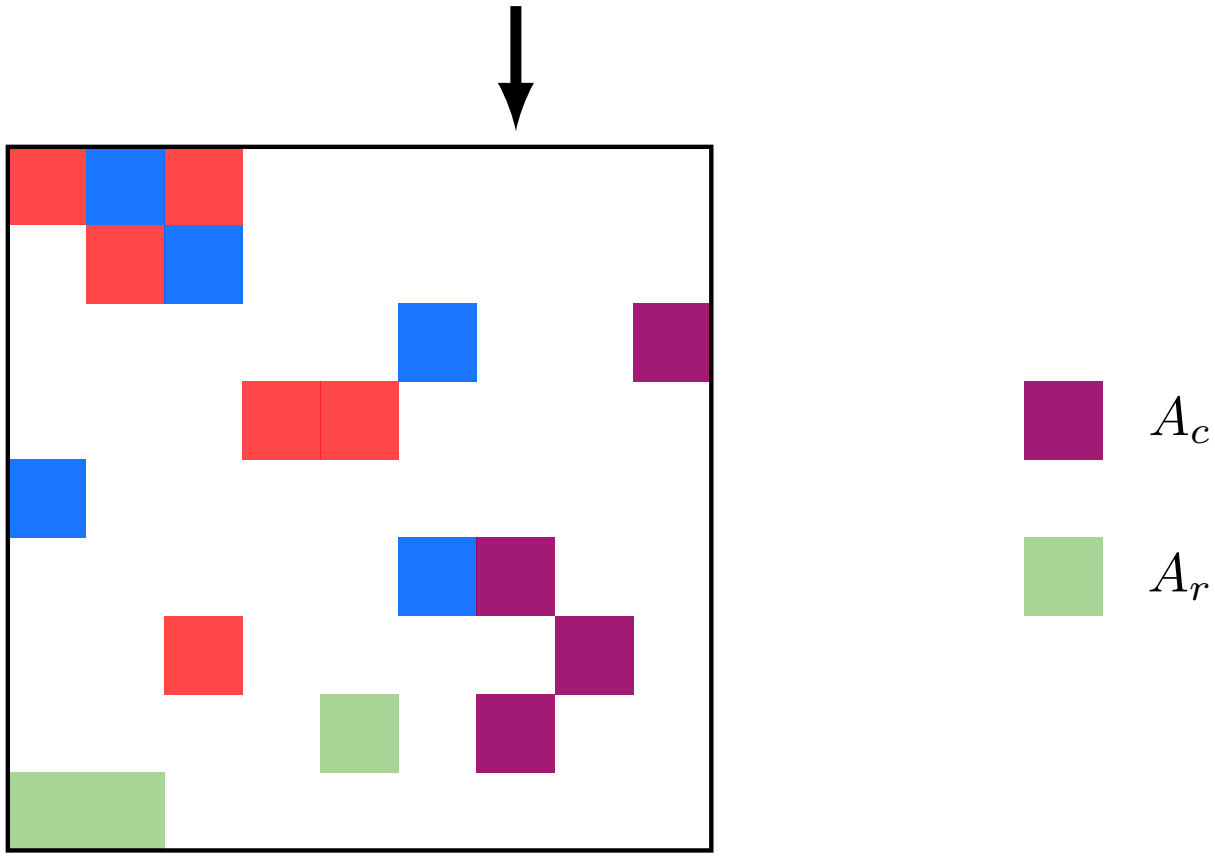
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Universiteit Utrecht

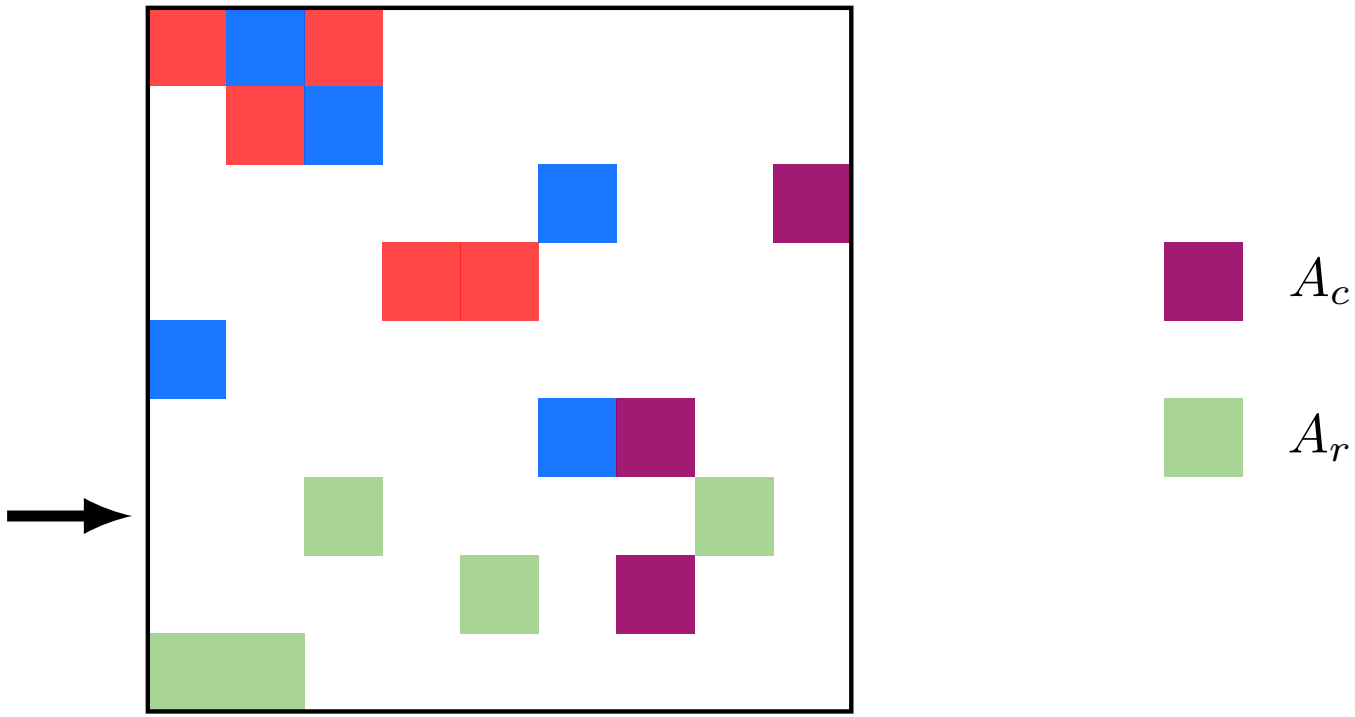
Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Overpainting algorithm

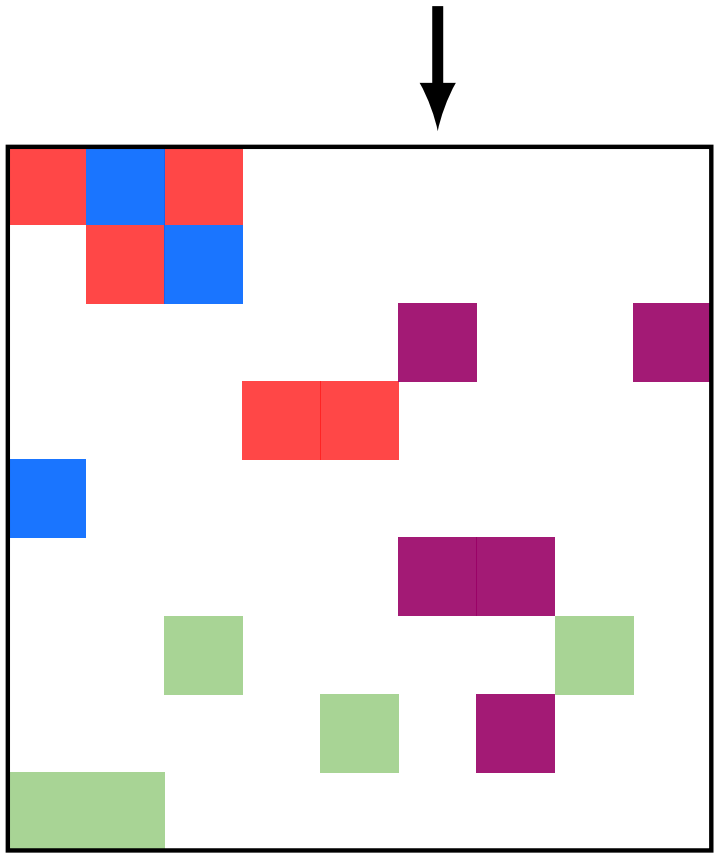
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

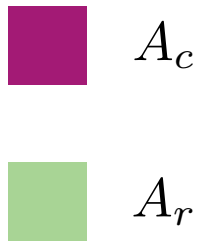


Universiteit Utrecht

Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

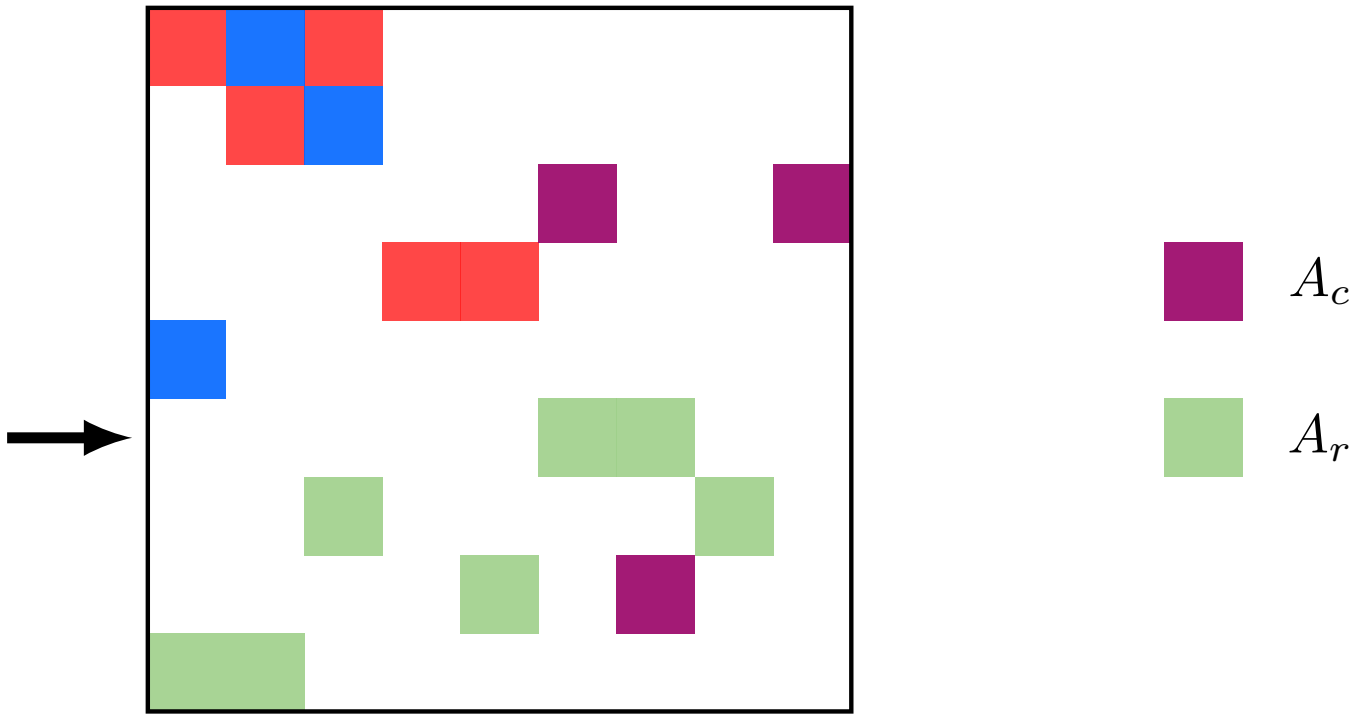




Universiteit Utrecht

Overpainting algorithm

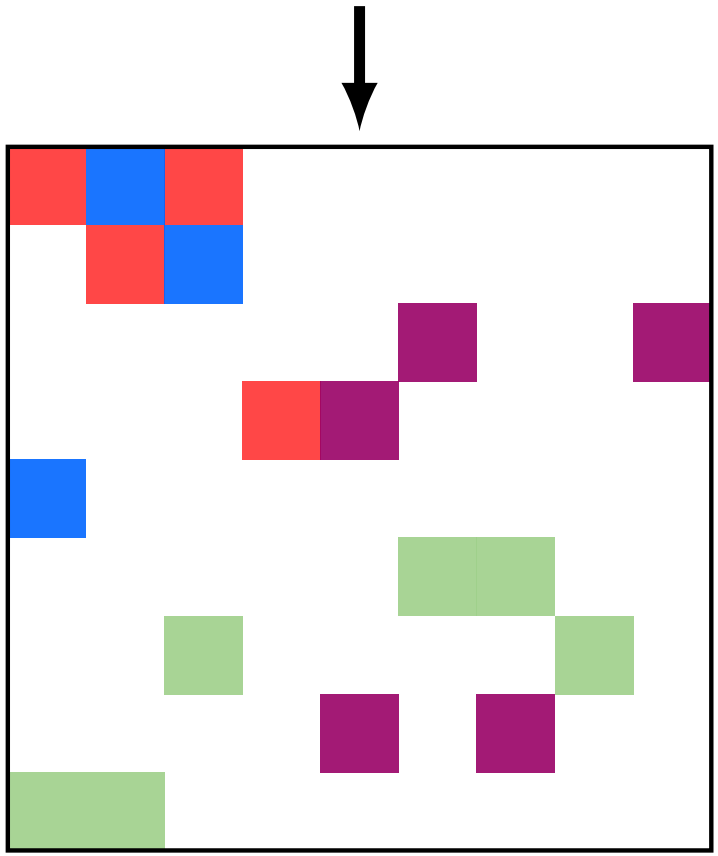
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

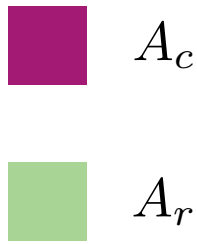


Universiteit Utrecht

Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

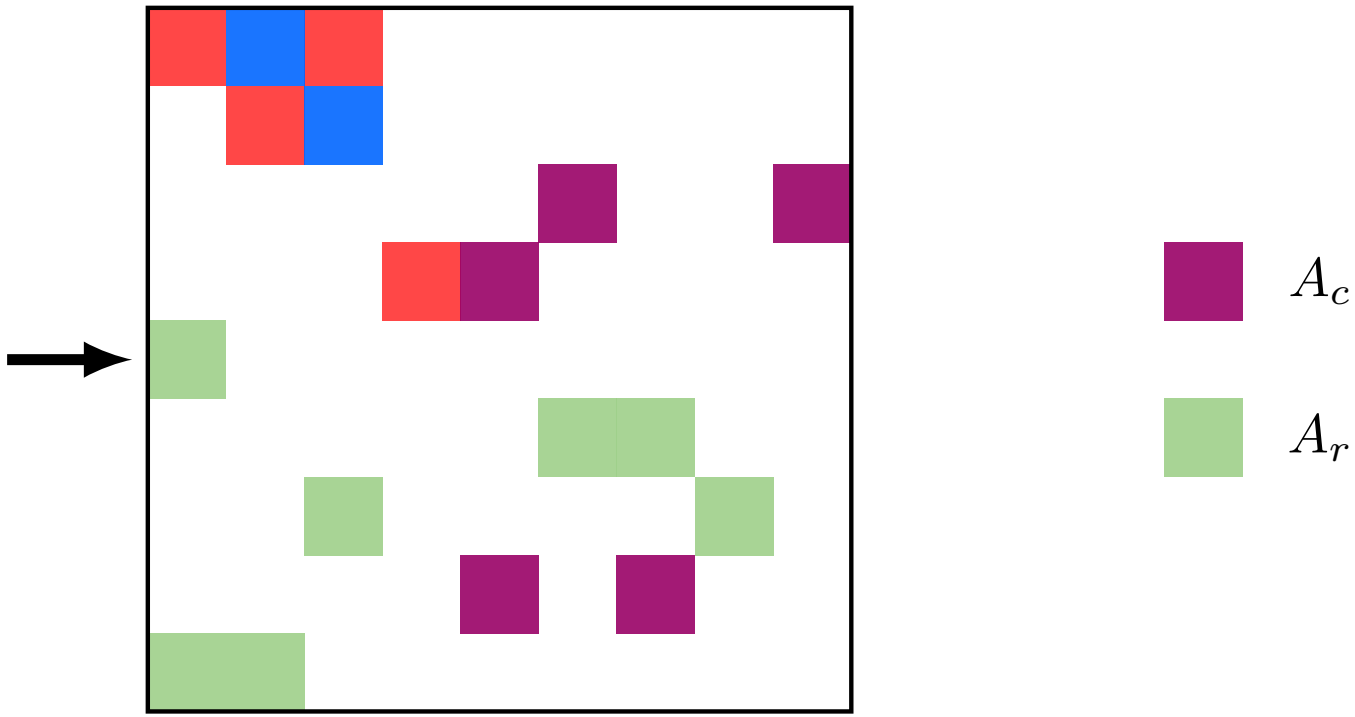




Universiteit Utrecht

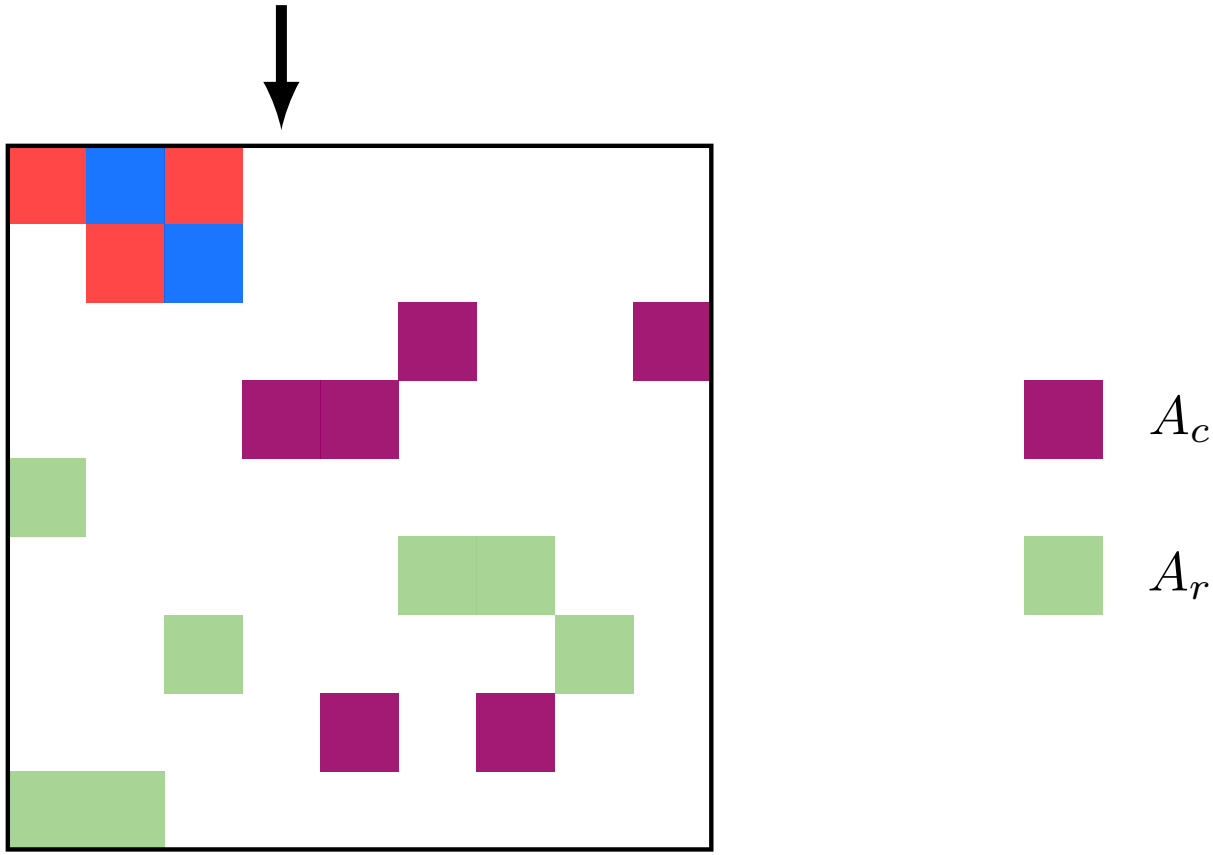
Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Overpainting algorithm

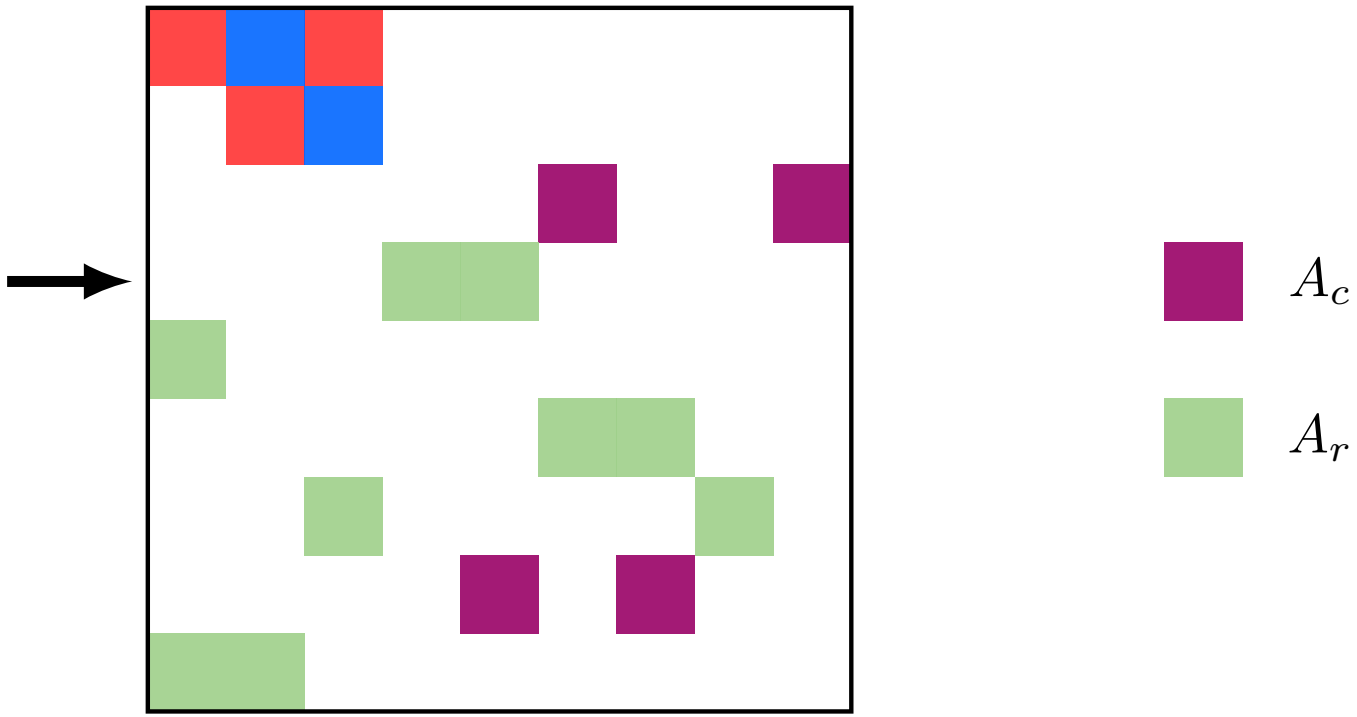
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Universiteit Utrecht

Overpainting algorithm

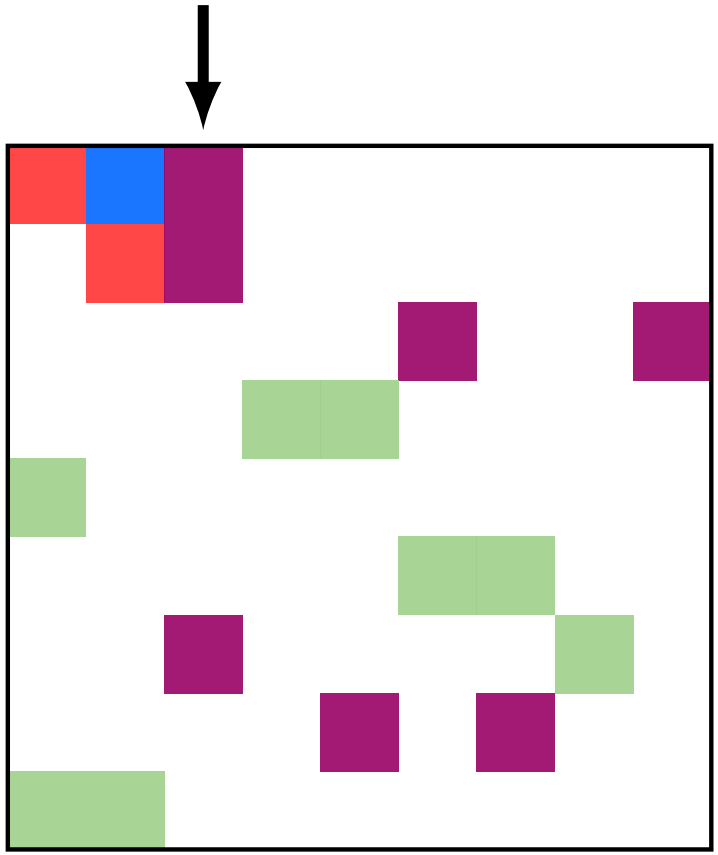
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

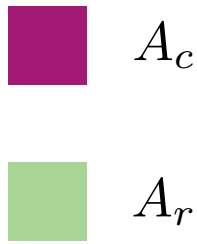


Universiteit Utrecht

Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

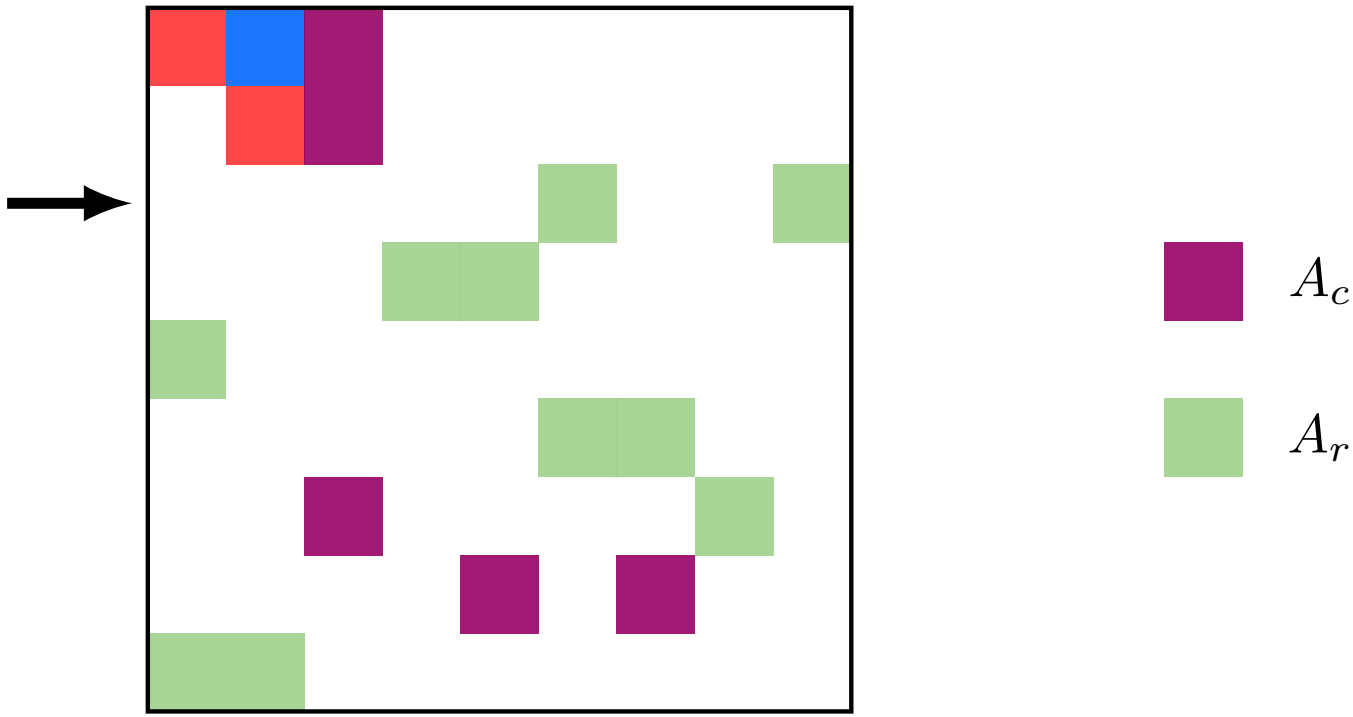




Universiteit Utrecht

Overpainting algorithm

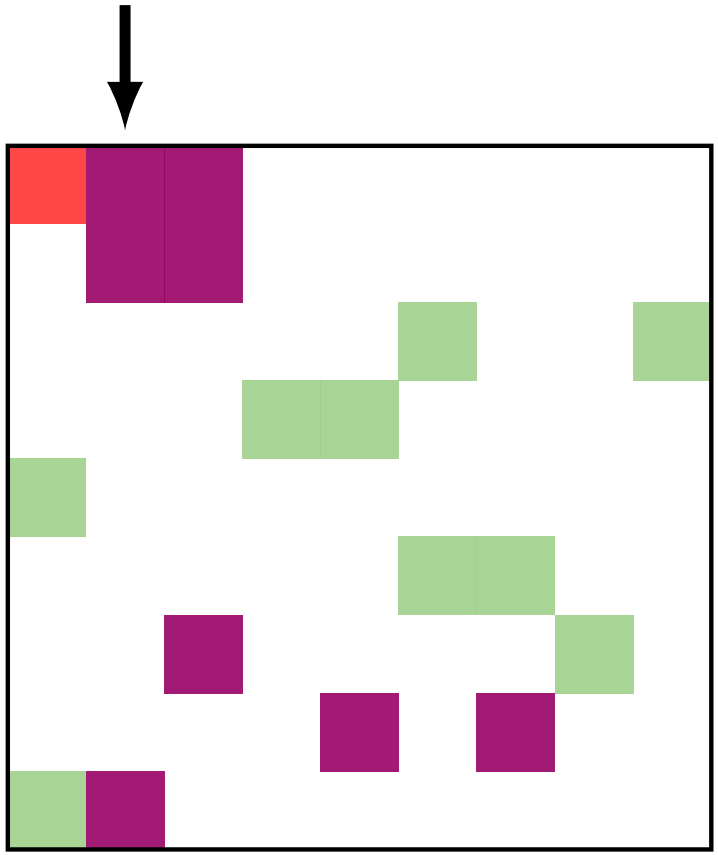
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

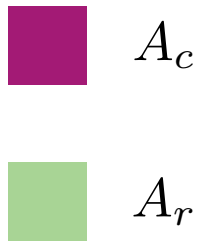


Universiteit Utrecht

Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$

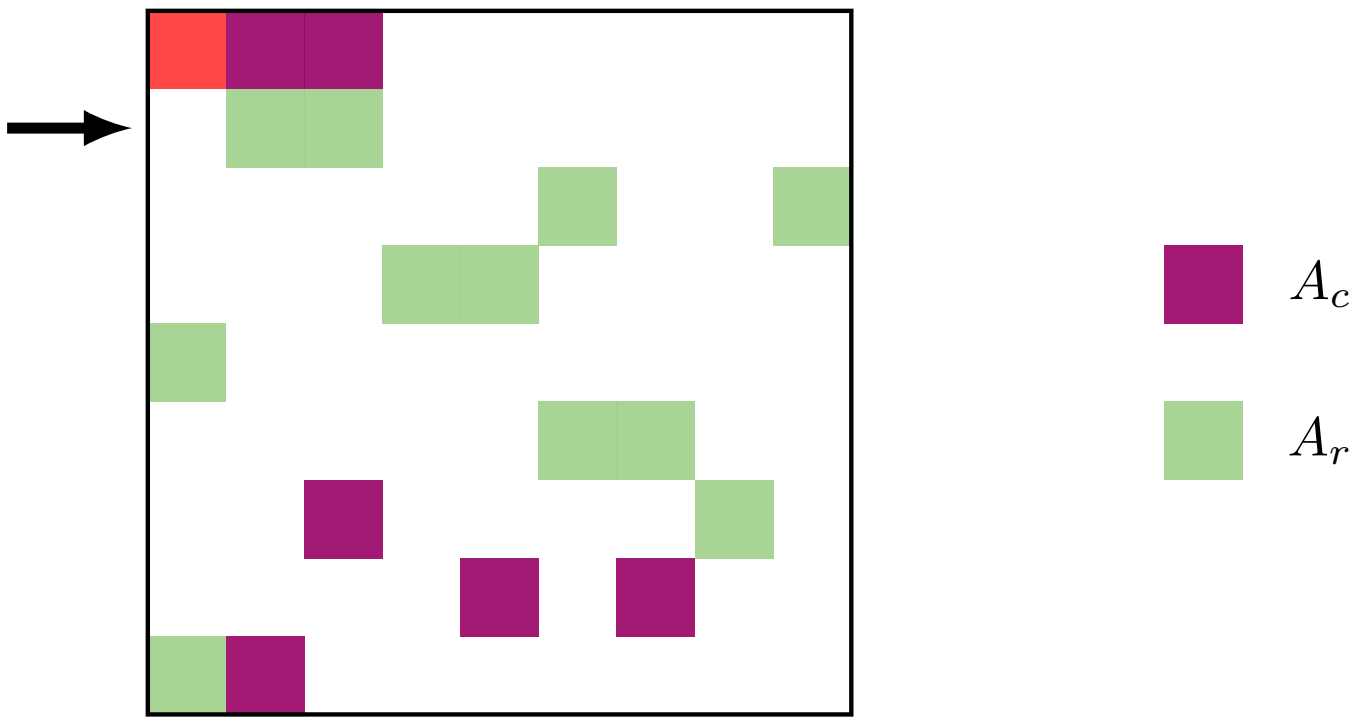




Universiteit Utrecht

Overpainting algorithm

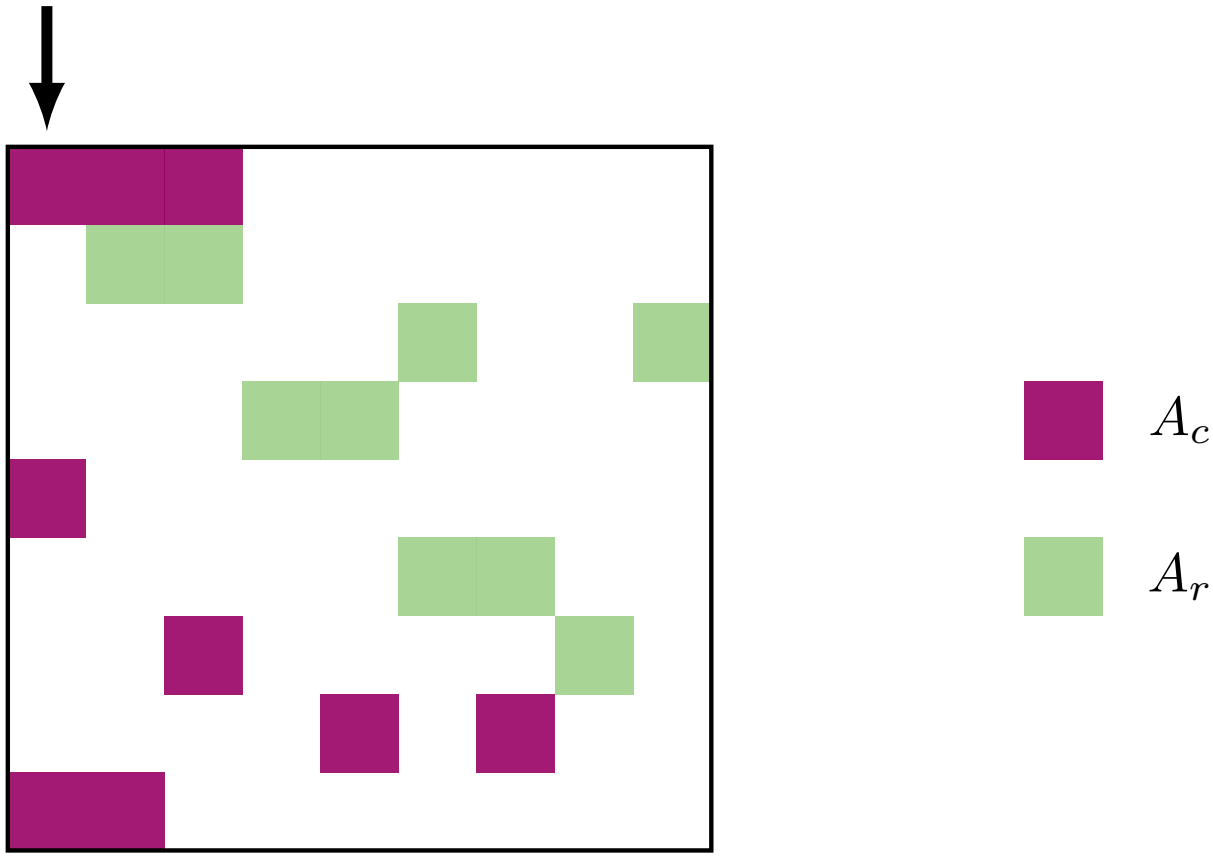
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Universiteit Utrecht

Overpainting algorithm

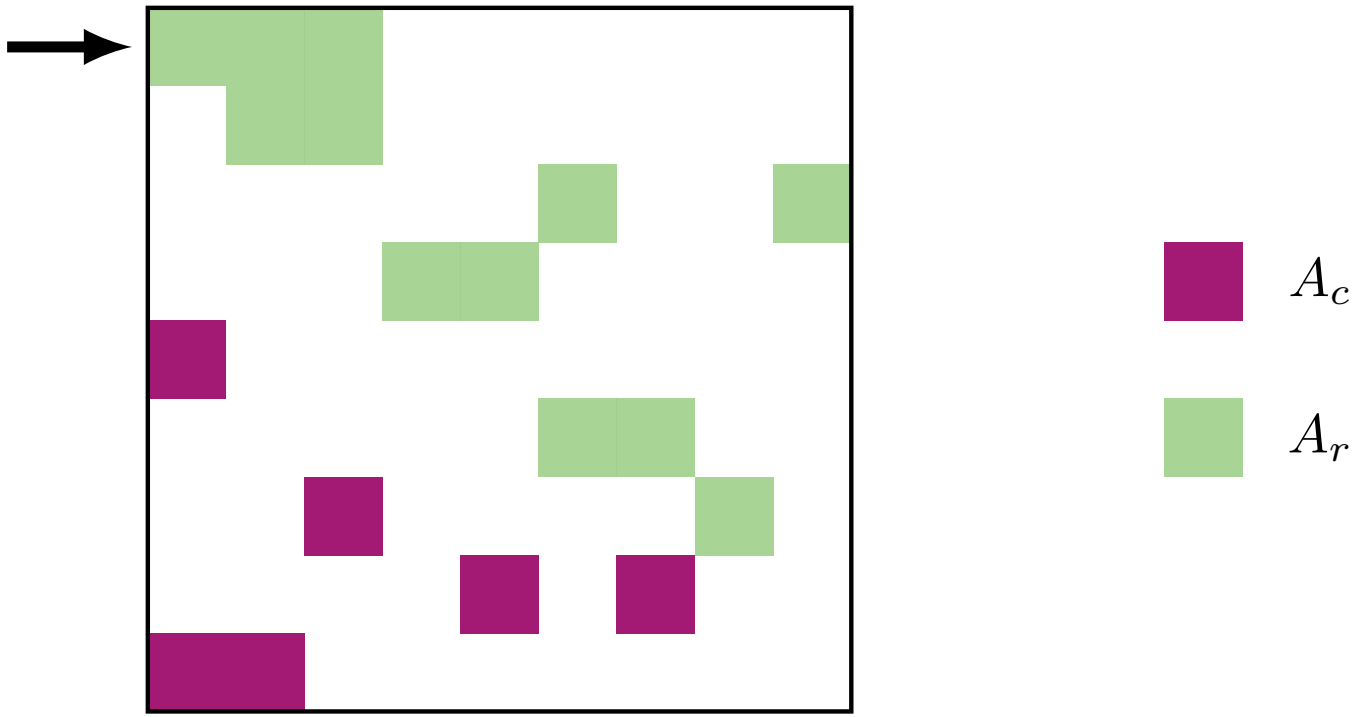
Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Universiteit Utrecht

Overpainting algorithm

Example: let $v := \{0, 9, 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17\}$



Universiteit Utrecht

Computation of the priority vector v

We used a structured approach for the construction of v: 30 different heuristics.

Generating schemes with three steps:

1. Usage of previous partitioning

2. Sorting (w.r.t the number of nonzeros, in ascending order)

3. Internal order of indices

Computation of the priority vector v

We used a structured approach for the construction of v: 30 different heuristics.

Generating schemes with three steps:

1. Usage of previous partitioning
   - partition-oblivious
   - partition-aware
2. Sorting (w.r.t the number of nonzeros, in ascending order)

3. Internal order of indices

Computation of the priority vector v

We used a structured approach for the construction of v: 30 different heuristics.

Generating schemes with three steps:

1. Usage of previous partitioning

2. Sorting (w.r.t the number of nonzeros, in ascending order)
   - sorted (with or without refinement)
   - unsorted
3. Internal order of indices

Computation of the priority vector v

We used a structured approach for the construction of v: 30 different heuristics.

Generating schemes with three steps:

1. Usage of previous partitioning

2. Sorting (w.r.t the number of nonzeros, in ascending order)

3. Internal order of indices
   - ◇ concatenation
   - ◇ mixing (either alternation or spread)
   - ◇ random (only when not sorting)
   - ◇ simple (only when sorting)

Independent set formulation

Partial assignment of rows and columns seems an interesting idea, but we want to reduce, as much as possible, the number of cut rows/columns.

Goal: Find the biggest subset (in terms of nonzeros) of $\{0, \ldots, m+n-1\}$ which can be assigned completely (i.e. full rows and full columns) without causing communication.

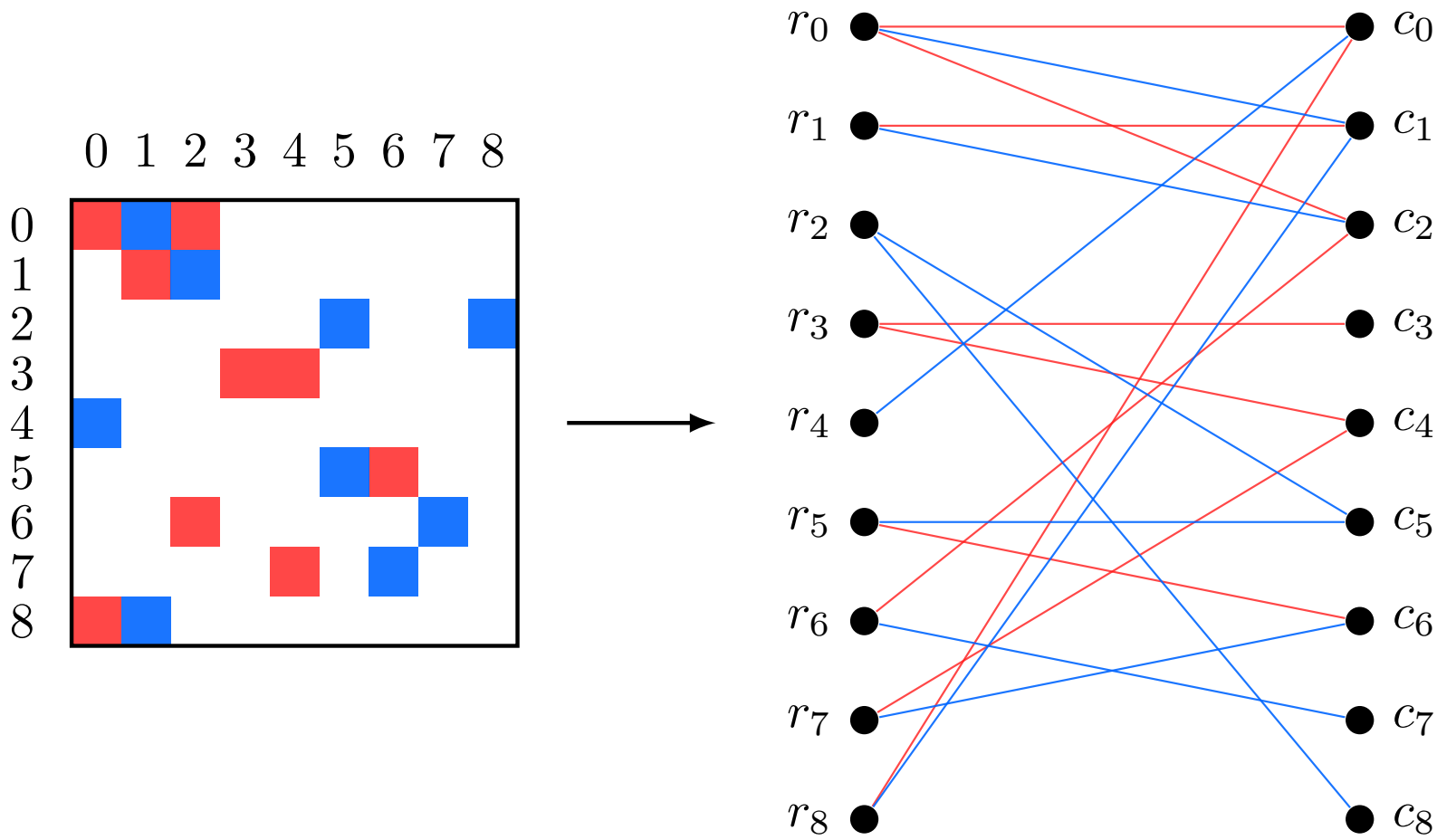
Graph theory approach: translating the sparsity pattern of A in a particular way, we are looking for a **maximum independent set**.

Construction of the graph

We construct the bipartite graph $G = (L \cup R, E)$ as follows:

- Rows and columns are vertices
  - $L = \{r_0, \ldots, r_{m-1}\}$
  - $R = \{c_0, \ldots, c_{n-1}\}$
- Edges correspond to nonzeros: $e = (i, j) \iff a_{ij} \neq 0$

Construction of the graph



Maximum independent set

Definition

An *independent set* is a subset $V' \subseteq V$ such that $\forall u, v \in V'$, $(u, v) \notin E$. A *maximum* independent set is an independent set of G with maximum cardinality.

Definition

A *vertex cover* is a subset $V' \subseteq V$ such that $\forall (u, v) \in E$ we have $u \in V' \vee v \in V'$, i.e. at least one of the endpoints of any edge is in the cover. A *minimum* vertex cover is a vertex cover of G with minimum cardinality.

- A maximum independent set is exactly what we are looking for;
- Immediate to see that independent set and vertex cover are complementary

Maximum independent set

In general, computing a maximum independent set is as hard as partitioning the matrix (both NP-hard problems).

But, luckily, our graph is bipartite:

- Kőnig's Theorem: on bipartite graphs, maximum matchings and minimum vertex covers have the same size;
- Hopcroft-Karp algorithm: $\mathcal{O}\left(N\sqrt{m+n}\right)$ algorithm to compute a maximum matching on a bipartite graph

In our case it is not very demanding to compute a maximum independent set.

Hopcroft-Karp algorithm

Devised by John Hopcroft and Richard Karp in 1973.

Require: Bipartite graph $G = (L \cup R, E)$
Ensure: Maximum matching M
  $M \leftarrow \varnothing$
  repeat
    $l_M \leftarrow$ length of the shortest augmenting path, using the matching M
    $P \leftarrow \{P_1, \ldots, P_k\}$, a maximal set of vertex-disjoint shortest augmenting paths of length l_M
    $M \leftarrow M \oplus (P_1 \cup \cdots \cup P_k)$
  until $P = \varnothing$

In practice, we also progressively compute the minimum vertex cover alongside.

Hopcroft-Karp algorithm

The inner loop is as follows:

1. We construct the directed graph $(L \cup R, D)$ from M:
   - ◇ edges in the matching go from R to L
   - ◇ edges not in the matching go from L to R
2. **breadth-first search** from unmatched vertices in L, which terminates when unmatched vertices in R are reached.
   l_M is the length of these shortest augmenting paths;
3. **depth-first search** from an unmatched vertex in R from the previous step, terminates whenever we reach an unmatched vertex in L, the path found is augmenting.
   We resume with the next depth-first search from another unmatched vertex in R.

Hopcroft-Karp algorithm

- The size of the matching increases in every iteration
- The matching is augmented over several paths simultaneously ($\sqrt{m+n}$ factor in running time)
- The actual running time is usually better than the theoretical one: our matrices are sparse and the graphs is sparse as well, which means fast search phases.

Maximum independent set

Given the set of indices I, let S_I denote the maximum independent set computed on the matrix $A(I)$.

One partition-oblivious heuristic to compute v:

1. let $I = \{0, \ldots, m+n-1\}$, then $v := (S_I, I \setminus S_I)$

For partition-aware heuristics, let U be the set of uncut indices, C be the set of cut indices; we have three possibilities:

1. we compute S_U and have $v := (S_U, U \setminus S_U, C)$;
2. we compute S_U, S_C and have $v := (S_U, U \setminus S_U, S_C, C \setminus S_C)$;
3. we compute S_U, then we define $U' := U \setminus S_U$ and compute $S_{C \cup U'}$, having $v := (S_U, S_{C \cup U'}, (C \cup U') \setminus S_{C \cup U'})$.

General framework for experiments

Require: Sparse matrix A
Ensure: Partitioning for the matrix A
  Partition A with Mondriaan using the default options and the medium-grain method
  for $i = 1, \ldots, iter_{max}$ **do**
    Use any of the heuristics described previously to compute A_r and A_c
    construct B from A_r and A_c
    Partition B with Mondriaan using the default options and the row-net model
    Re-construct A with the new partitioning
  end for

Unique framework for both partition-oblivious and partition-aware types of heuristics.

Implementation

All of the heuristics have been implemented following these steps:

1. MATLAB prototyping
2. Core C implementation (MATLAB compatibility through MEX files)
3. Full C implementation

The Hopcroft-Karp algorithm for the maximum indipendent set computation was implemented in the Python programming language.

Implementation

Randomness involved during the computation of A_r and A_c and during the actual partitioning. To obtain meaningful results:

- 20 independent initial partitionings
- for each, 5 independent runs of the heuristic and subsequent partitioning ($iter_{max} = 1$)

18 matrices used for tests:

- rectangular vs. square
- 10^{th} Dimacs Implementation Challenge

Preliminary selection

Wide selections of heuristics, preliminary selection is necessary.

5 matrices used:

- `dfl001`;
- `tbdlinux`;
- `nug30`;
- `rgg_n_2_18_s0`;
- `bcsstk30`.

Preliminary selection

Partition-oblivious heuristics (17 different algorithms):

- In general, results are much worse than medium-grain method.
- Mixing rows and columns in partial assignment is a bad idea
- Individual assignment of nonzero best strategy (7% worse than medium-grain)
- Maximum independent set computation yields interesting results (16% lower communication volume in one matrix, but in general 12% worse than medium-grain)

2 heuristics (`po_localview` and `po_is`) selected for deeper investigation.

Preliminary selection

Partition-aware heuristics (21 different algorithms):

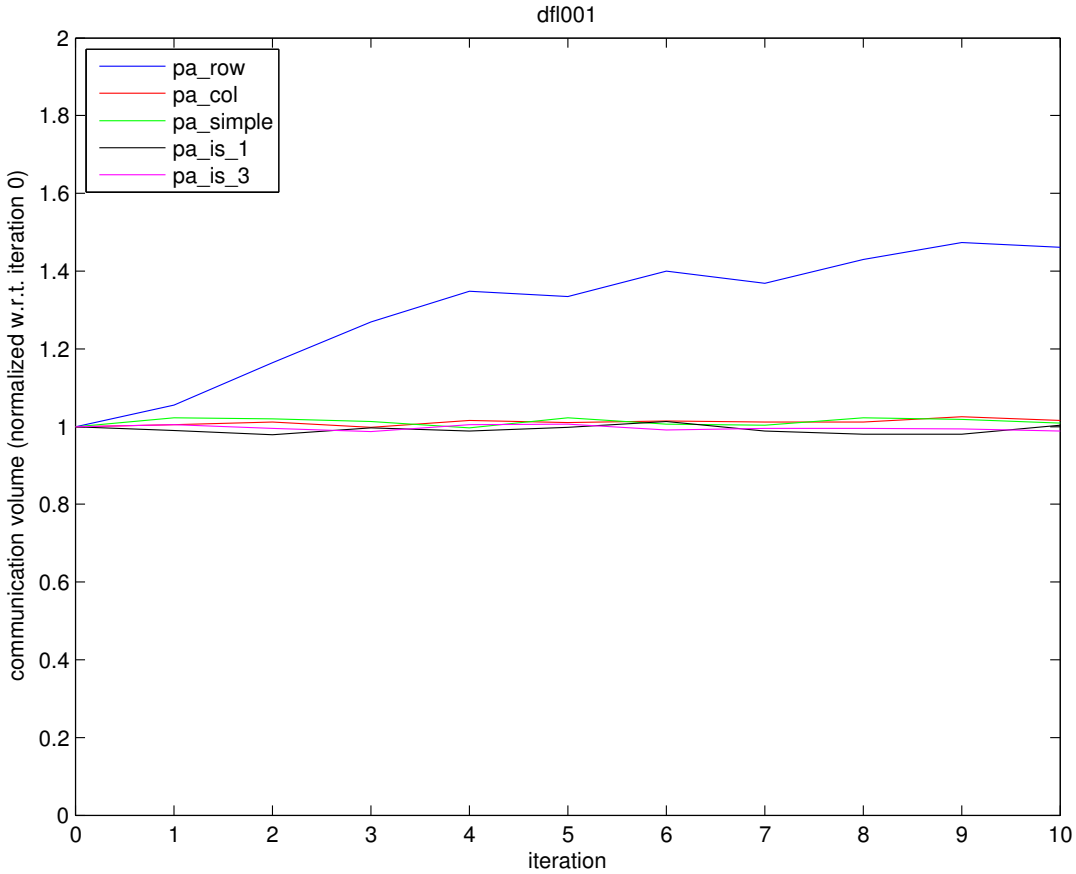
- Results closer to medium-grain efficiency
- SBD and SBD2 forms are not wortwhile, nor individual assignment of nonzeros
- Mixing rows and columns is still not a good idea, but less damaging
- Refinement in sorting does not yield a substantial difference
- Unsorted concatenation of rows and columns produces good results with rectangular matrices: they can be combined into a `localbest` heuristic
- Maximum independent set strategy very close to medium-grain, even better in a few matrices

5 heuristics (`pa_row`, `pa_col`, `pa_simple`, `pa_is_1`, `pa_is_3`) selected for deeper investigation.

Number of iterations

We are developing a fully iterative scheme: how many iterations do we execute?

We run the 5 partition-aware selected heuristics for 10 consecutive iterations for each matrix

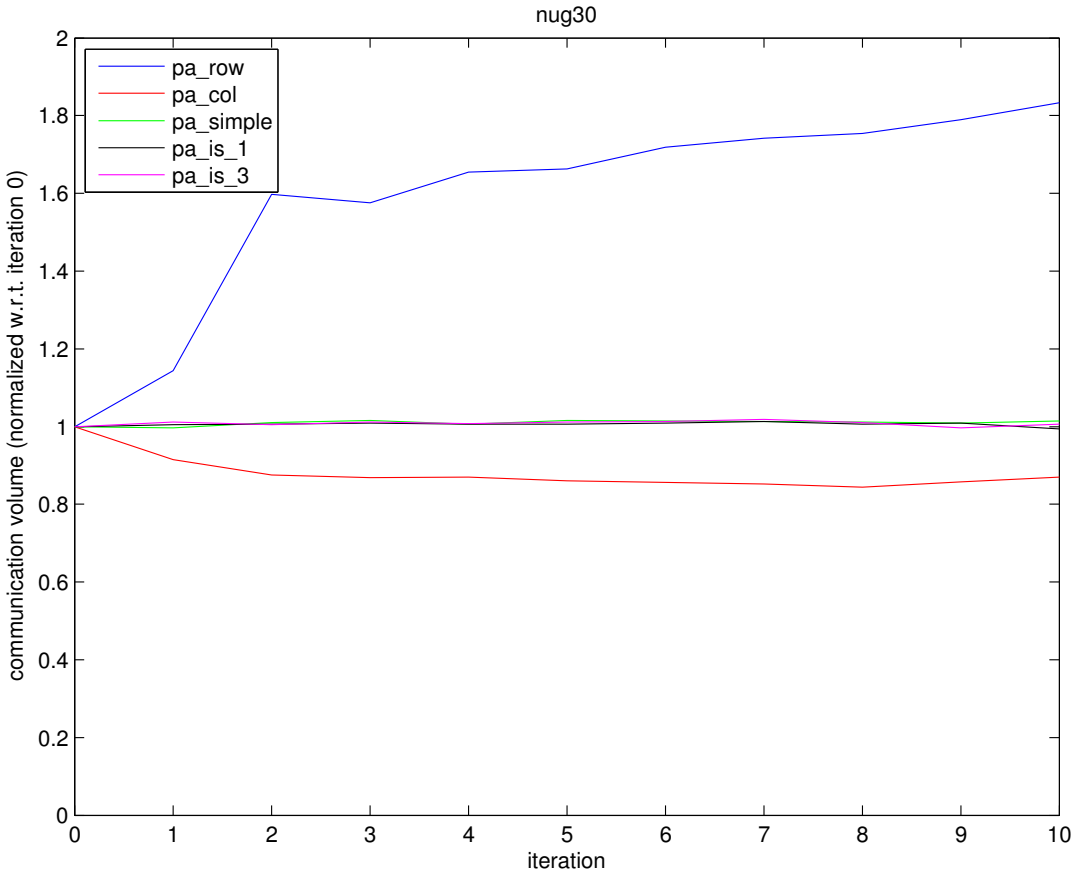


Universiteit Utrecht

Number of iterations

We are developing a fully iterative scheme: how many iterations do we execute?

We run the 5 partition-aware selected heuristics for 10 consecutive iterations for each matrix

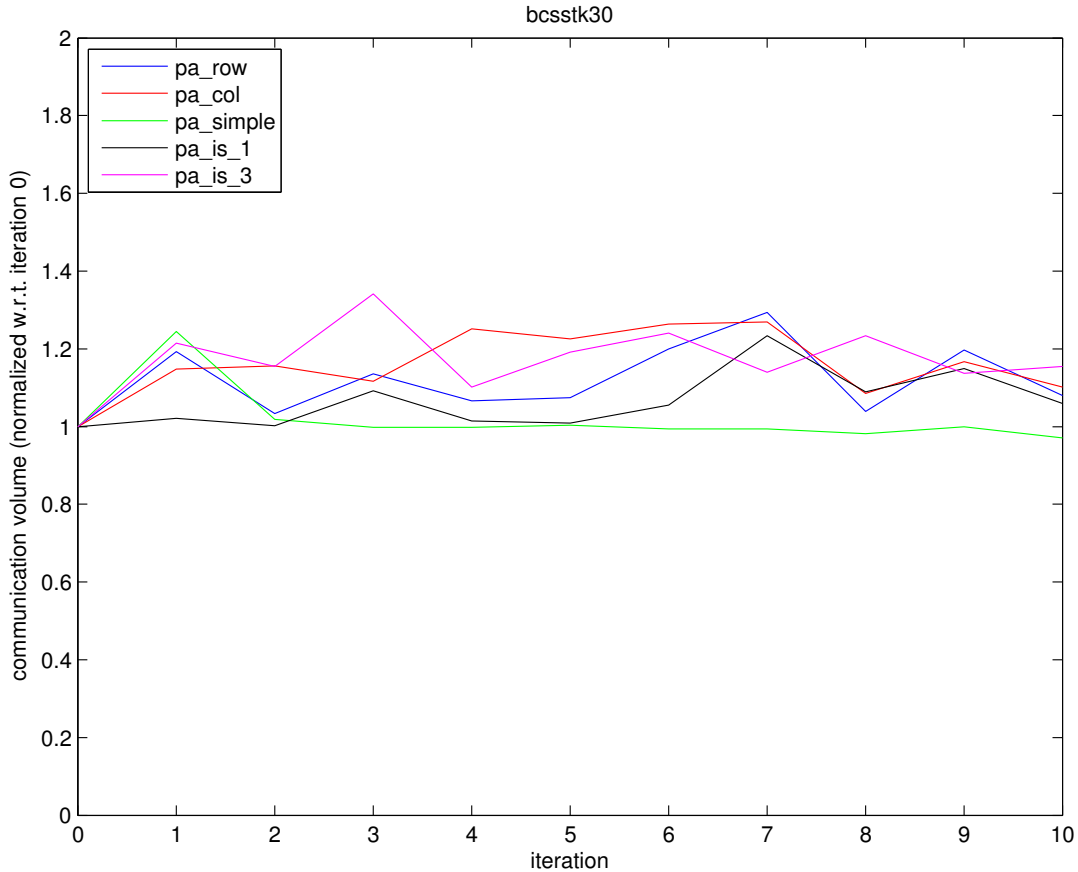


Universiteit Utrecht

Number of iterations

We are developing a fully iterative scheme: how many iterations do we execute?

We run the 5 partition-aware selected heuristics for 10 consecutive iterations for each matrix

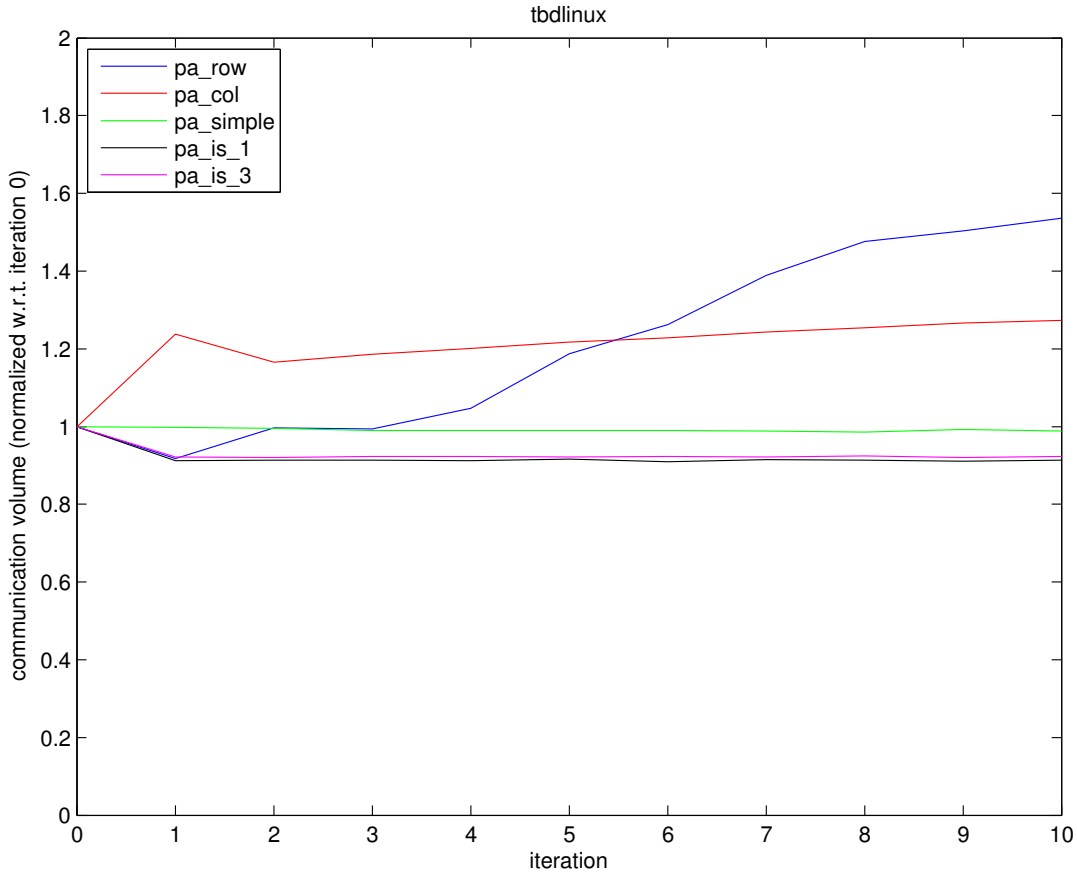


Universiteit Utrecht

Number of iterations

We are developing a fully iterative scheme: how many iterations do we execute?

We run the 5 partition-aware selected heuristics for 10 consecutive iterations for each matrix

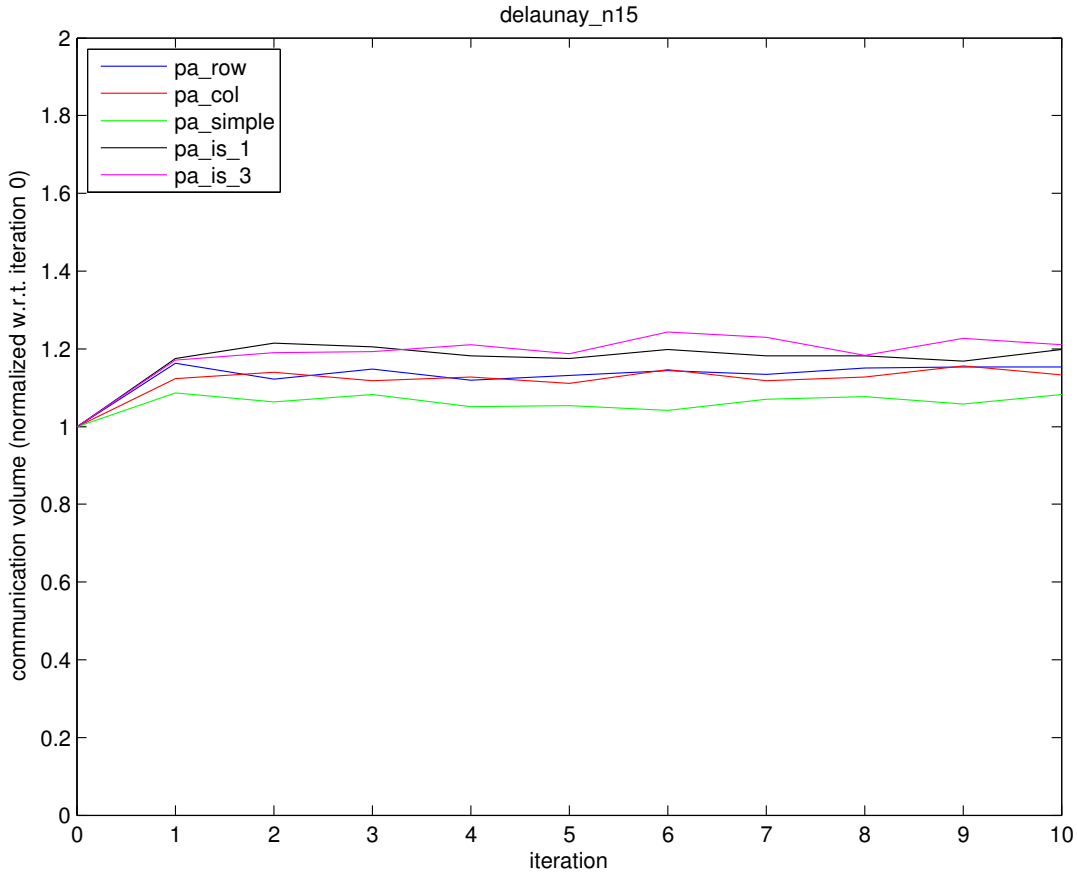


Universiteit Utrecht

Number of iterations

We are developing a fully iterative scheme: how many iterations do we execute?

We run the 5 partition-aware selected heuristics for 10 consecutive iterations for each matrix



Universiteit Utrecht

Number of iterations

We are developing a fully iterative scheme: how many iterations do we execute?

We run the 5 partition-aware selected heuristics for 10 consecutive iterations for each matrix

- Usually 1 iteration is enough to show improvements, if any.
- More iterations can worsen the communication volume.

Analysis of the performance of the best heuristics

Partition-oblivious heuristics:

- No devised heuristic was able to improve medium-grain
- The preliminary results confirmed:
  - Individual assignment of nonzeros 7% worse than medium-grain
  - Computing the maximum independent set 22% worse than medium-grain

Analysis of the performance of the best heuristics

Partition-aware heuristics:

- Results similar to preliminary selections: preliminary matrices were good representatives
- Concatenation interesting strategy:
  - `pa_row` and `pa_col` 8% worse than medium-grain
  - `localbest` method takes best of both: only 4% worse than medium-grain
- Similar good results for the other strategies (between 4% and 8% higher communication volume than medium-grain)
- No algorithm was able to beat medium-grain
- Considering only rectangular matrices, our methods work better: they improve medium-grain, even if only by little (1-2%)

Iterative refinement

Is there something we can do to improve the results?

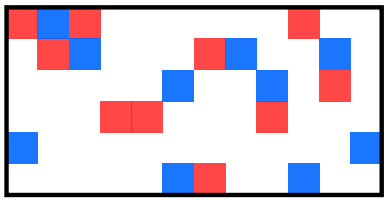
Medium-grain employs a procedure of **iterative refinement**:

1. A is partitioned into two sets (A_0 and A_1)
2. we create again the matrix B of the medium-grain method (example: $A_r = A_0$ and $A_c = A_1$)
3. we retain communication volume: the first n columns of B are assigned to a single processor, and similarly for the other m
4. we create the hypergraph from this B and a single run of Kernighan-Lin is performed
5. we repeat steps 1-4 until no improvement is found, then we swap the roles of A_0 and A_1 for the creation of A_r and A_c
6. we repeat step 5 until no other improvement is found

Iterative refinement

Kernighan-Lin method is **monotonically non-increasing**: during iterative refinement, the communication volume is either lowered or remains at the same value.

Example of the construction of B:

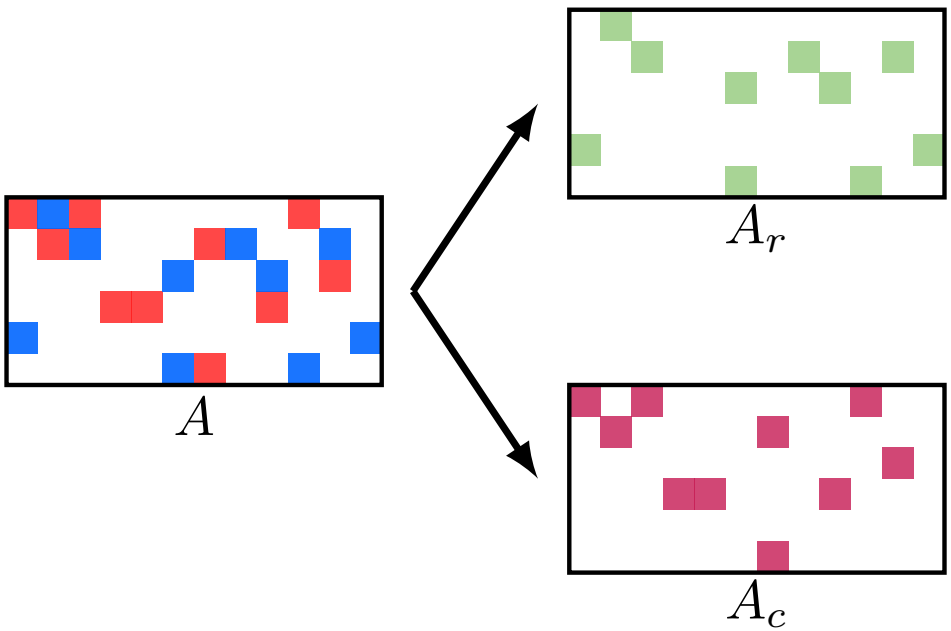


A

Iterative refinement

Kernighan-Lin method is **monotonically non-increasing**: during iterative refinement, the communication volume is either lowered or remains at the same value.

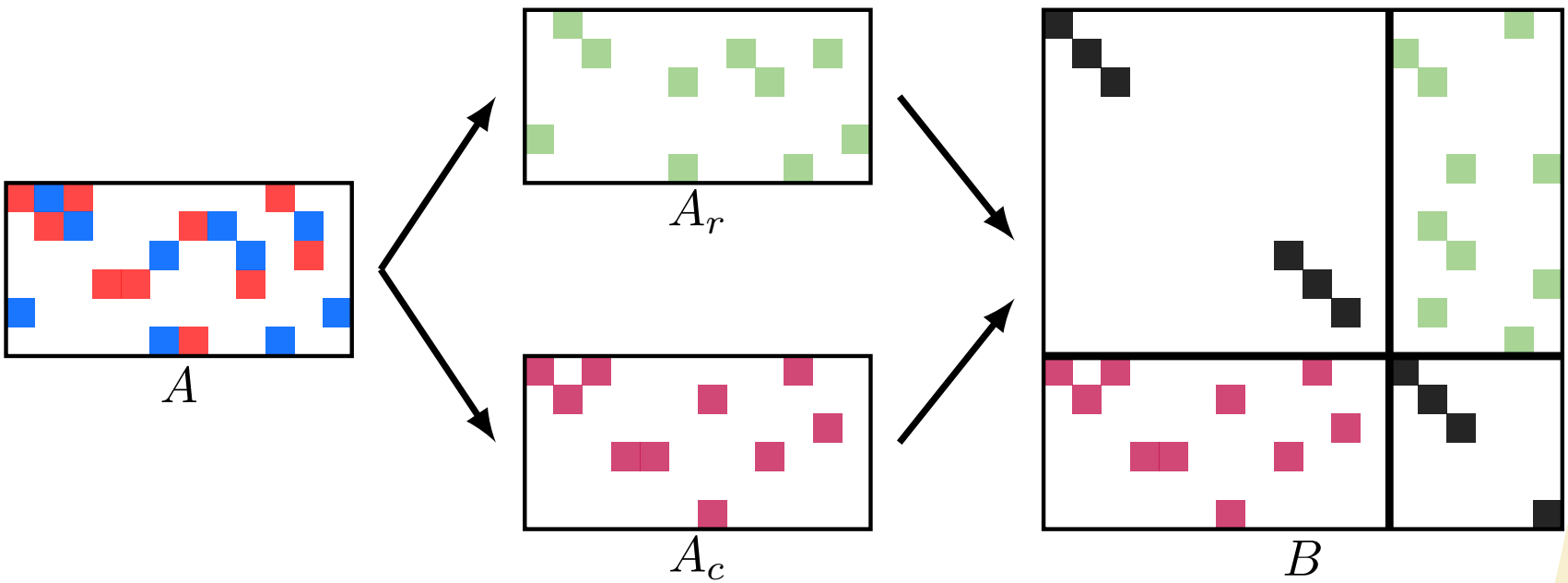
Example of the construction of B:



Iterative refinement

Kernighan-Lin method is **monotonically non-increasing**: during iterative refinement, the communication volume is either lowered or remains at the same value.

Example of the construction of B:

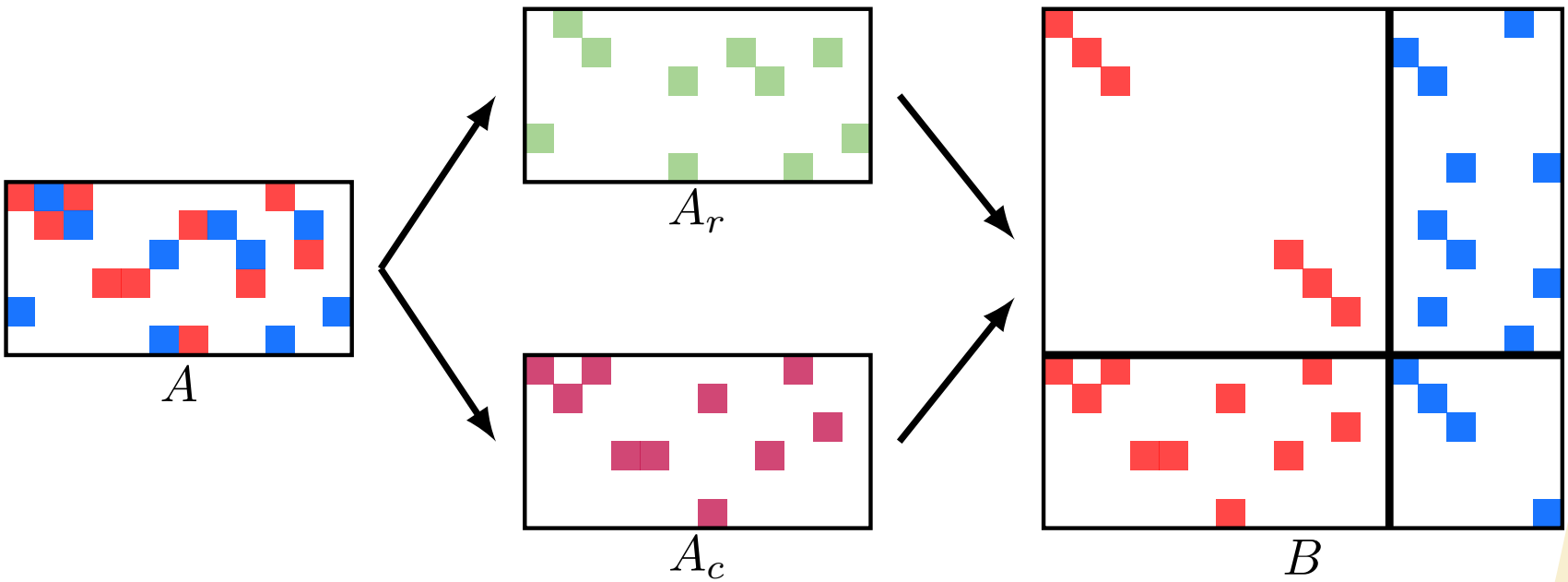


Universiteit Utrecht

Iterative refinement

Kernighan-Lin method is **monotonically non-increasing**: during iterative refinement, the communication volume is either lowered or remains at the same value.

Example of the construction of B:



Universiteit Utrecht

Iterative refinement

With iterative refinements, results are in general better:

- partition-oblivious algorithms:
  - `po_localview` still 7% worse than medium-grain
  - `po_is` now 6% worse than medium-grain (down from 22%)
- partition-aware algorithms:
  - `pa_row` and `pa_col` now 2% and 1% worse than medium-grain (down from 8%)
  - `localbest` now 1% **better** than medium-grain (down from 4% worse)
  - `pa_simple` now 2% worse than medium-grain (down from 4%)
  - `pa_is_1` and `pa_is_3` now 1% worse than medium-grain (down from 5% and 8%)

Now, with rectangular matrices, computing the independent set produces an average communication volume 4% lower than medium-grain.

Conclusions

We originally had two research directions:

- Improving the quality of the initial partitioning
- Developing a fully-iterative scheme

Conclusions

We originally had two research directions:

- Improving the quality of the initial partitioning
  - We were not able to outperform medium-grain
- Developing a fully-iterative scheme

Conclusions

We originally had two research directions:

- Improving the quality of the initial partitioning

- Developing a fully-iterative scheme
  - We were able to outperform medium-grain only by a small margin
  - Computing the independent set is worthwile. Also results about concatenation of rows and columns can be explained with it.
  - Our approach works well with rectangular matrices

Further research

A number of possibilities for further development:

- Keep testing other strategies to gain more confidence on medium-grain
- Hopcroft-Karp algorithm could be implemented in C to tackle bigger problems
- If our approach is confirmed to work consistently well with rectangular matrices, it could be added to Mondriaan:
  1. The program detects that the matrix is strongly rectangular and asks user for input
  2. The user decides whether he wants to sacrifice computation time for a better partitioning
  3. If so, our approach is executed.