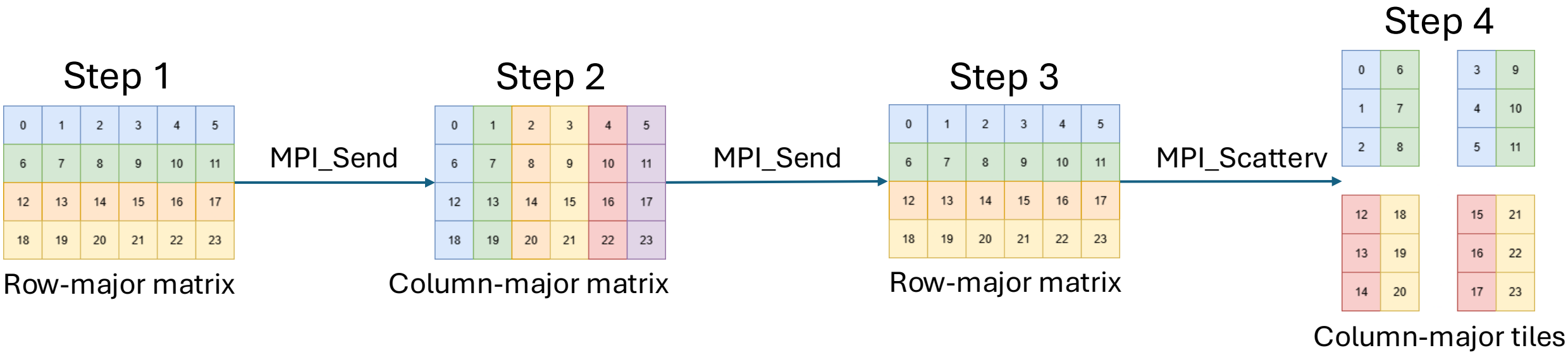
Charles University

Layout-Agnostic MPI Abstraction for Distributed Computing in Modern C++

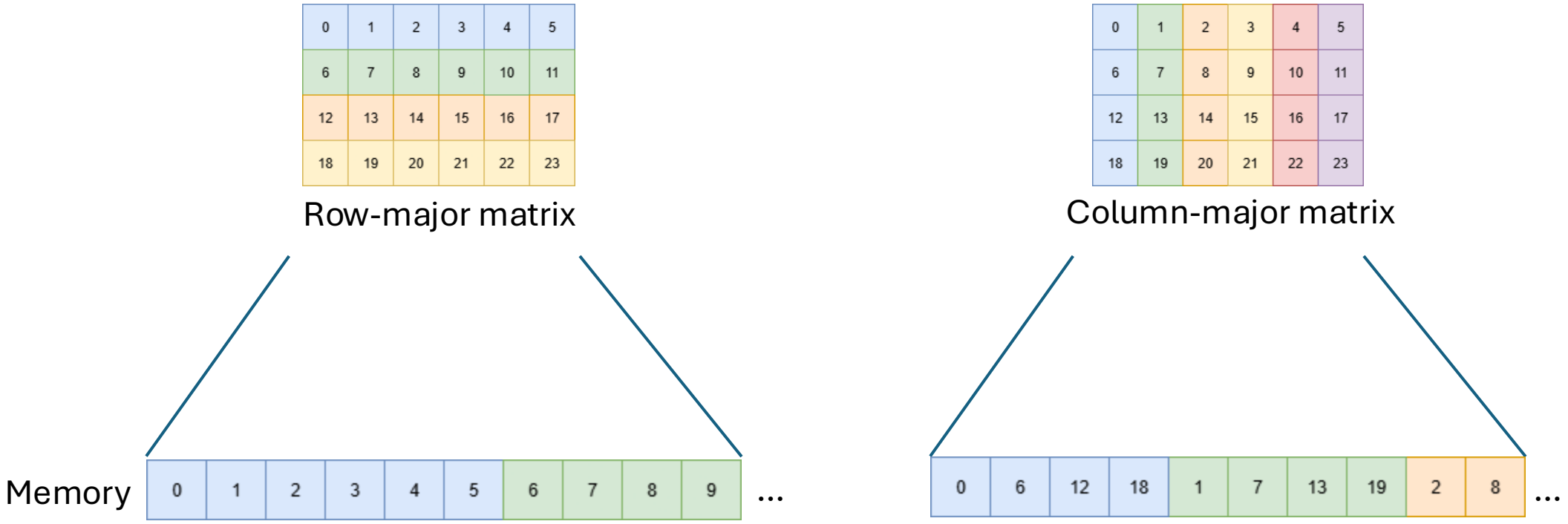
Jiří Klepl, Matyáš Brabec, Martin Kruliš

EUROMPI 25

MPI and layout-agnosticism



Same data in different data layouts



Optimal layouts depend on traversals

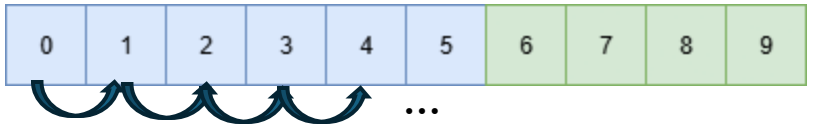
Row-wise traversal

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

Row-major matrix

```
int *row_major = new int[ROWS*COLS];

for (int r = 0; r < ROWS; r++)
  for (int c = 0; c < COLS; c++)
    row_major[r*COLS + c] = init(r, c);
```

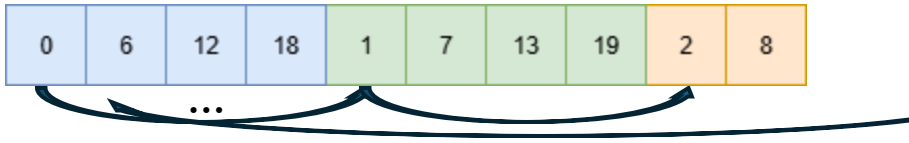


0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

Column-major matrix

```
int *col_major = new int[ROWS*COLS];

for (int r = 0; r < ROWS; r++)
  for (int c = 0; c < COLS; c++)
    col_major[c*ROWS + r] = init(r, c);
```



Optimal layouts depend on traversals

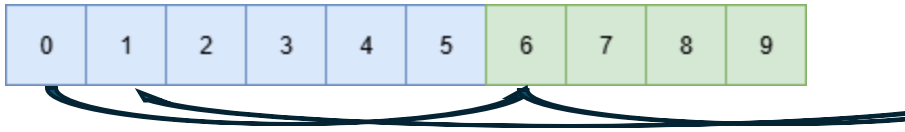
Column-wise traversal

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

Row-major matrix

```
int *row_major = new int[ROWS*COLS];

for (int c = 0; c < COLS; c++)
  for (int r = 0; r < ROWS; r++)
    row_major[r*COLS + c] = init(r, c);
```



0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

Column-major matrix

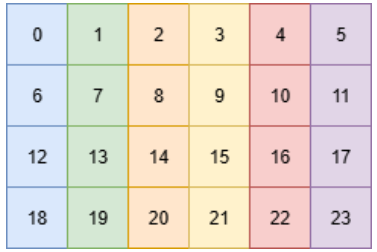
```
int *col_major = new int[ROWS*COLS];

for (int c = 0; c < COLS; c++)
  for (int r = 0; r < ROWS; r++)
    col_major[c*ROWS + r] = init(r, c);
```

Layout agnosticism via the Noarr library

```
int matrix = new int[ROWS*COLS];
```


```
int matrix = new int[ROWS*COLS];
```



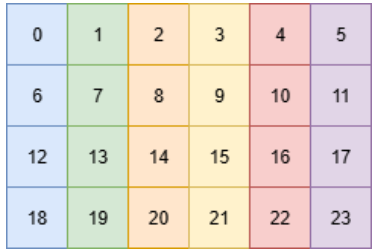
Pure C/C++: accepts only one layout

```
for (int r = 0; r < ROWS; r++)
  for (int c = 0; c < COLS; c++)
    … matrix[r*COLS + c] …
```

Layout agnosticism via the Noarr library

```
int matrix = new int[ROWS*COLS];
auto matrix = bag(scalar<int>()
                ^ vector<'c'>(COLS)
                ^ vector<'r'>(ROWS));
```


```
int matrix = new int[ROWS*COLS];
auto matrix = bag(scalar<int>()
                ^ vector<'r'>(ROWS)
                ^ vector<'c'>(COLS));
```



Pure C/C++: accepts only one layout

```
for (int r = 0; r < ROWS; r++)
  for (int c = 0; c < COLS; c++)
    … matrix[r*COLS + c] …
```

Noarr: accepts either layout (= is layout agnostic)

```
for (int r = 0; r < ROWS; r++)
  for (int c = 0; c < COLS; c++)
    … matrix[ idx<'r', 'c'>(r, c) ] …
```

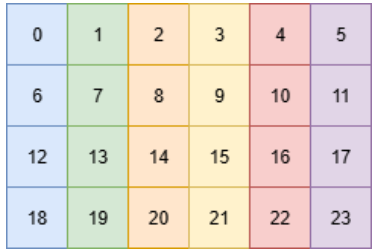
Layout agnosticism via the Noarr library

~~`int matrix = new int[ROWS*COLS];`~~

```
auto matrix = bag(scalar<int>()
                ^ vector<'c'>(COLS)
                ^ vector<'r'>(ROWS));
```

~~`int matrix = new int[ROWS*COLS];`~~

```
auto matrix = bag(scalar<int>()
                ^ vector<'r'>(ROWS)
                ^ vector<'c'>(COLS));
```



Noarr: accepts either layout (= is layout agnostic)

```
for (int r = 0; r < ROWS; r++)
  for (int c = 0; c < COLS; c++)
    … matrix[ idx<'r', 'c'>(r, c) ] …
```

Noarr: abstracting away the traversal itself

```
traverser(matrix).for_each([&](auto idx) {
  auto [r, c] = get_indices<'r', 'c'>(idx);
  … matrix[idx] …
});
```

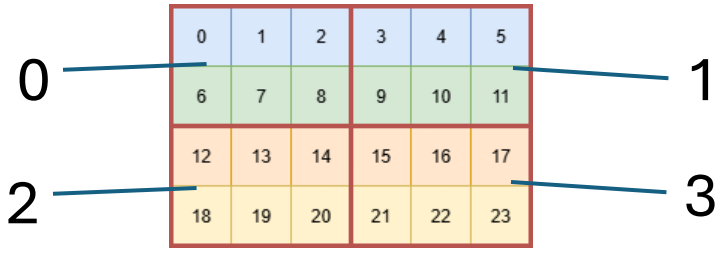
Row-wise: `traverser(matrix) ^ hoist<'r'>()`

Column-wise: `traverser(matrix) ^ hoist<'c'>()`

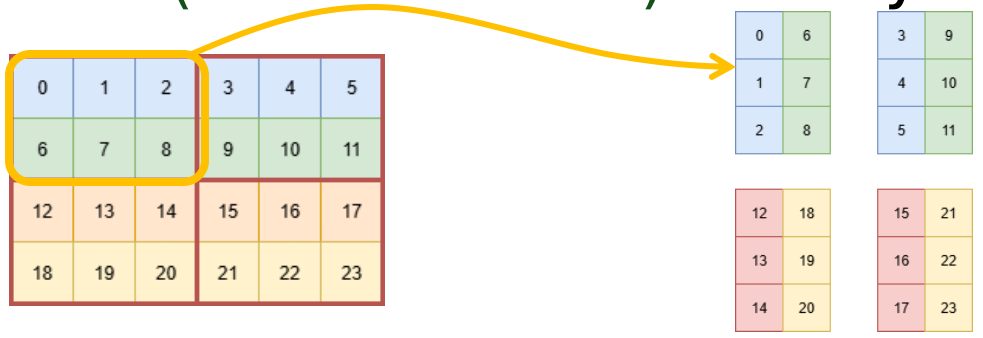
Steps for MPI integration

1. Expressing multiple layouts as matching MPI datatypes

2. Assigning sections of traversal to MPI processes

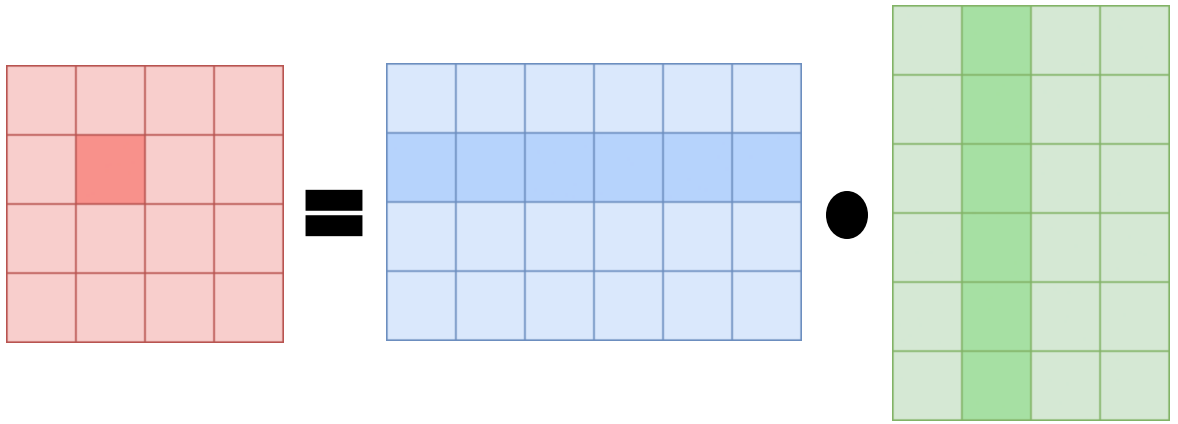


3. Scatter/gather (and transform) sub-layouts to local copies



Running example: Matrix multiplication

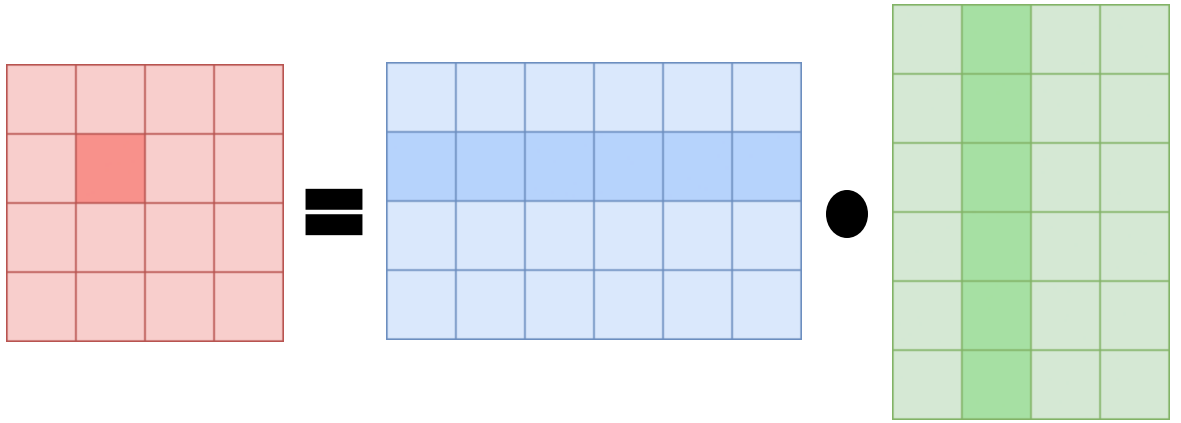
$$C = A \cdot B$$



Each cell = dot product of row and column

Running example: Matrix multiplication

$$C = A \cdot B$$



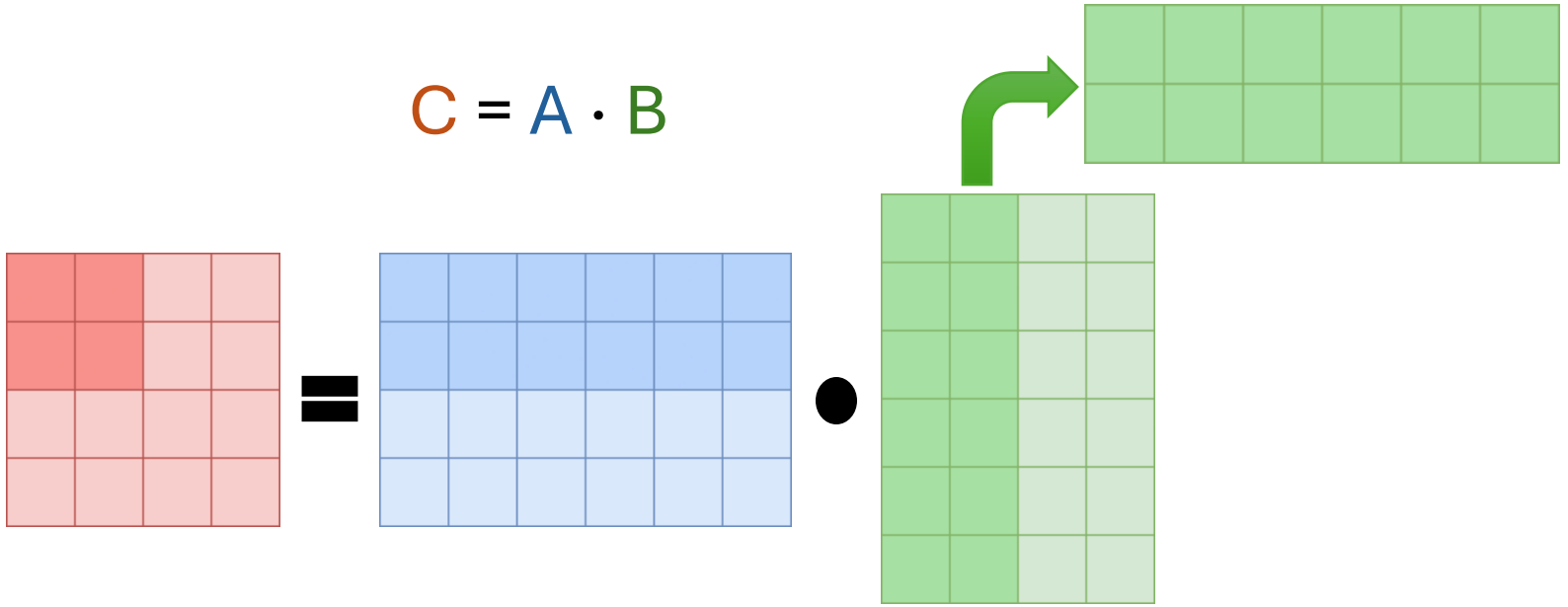
Each cell = dot product of row and column

```
auto C = bag(scalar<float>() ^ vector<'j'>(COLS) ^ vector<'i'>(ROWS);
auto A = bag(scalar<float>() ^ vector<'k'>(K) ^ vector<'i'>(ROWS);
auto B = bag(scalar<float>() ^ vector<'j'>(COLS) ^ vector<'k'>(K);

traverser(C, A, B).for_each([&](auto idx) {
  C[idx] += A[idx] * B[idx];
});
```

Running example: Matrix multiplication

$C = A \cdot B$

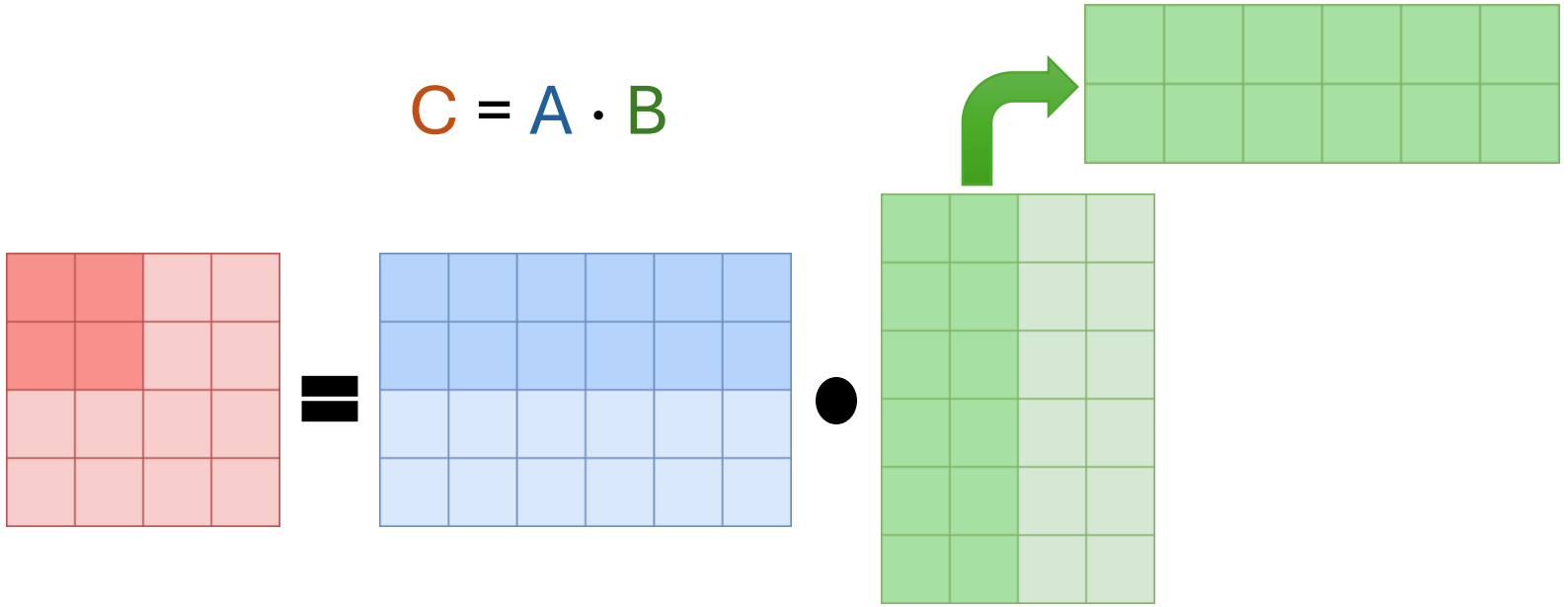


Each cell = dot product of row and column

Optimization → Each tile = dot product of row slice and transposed column slice

Running example: Matrix multiplication

$C = A \cdot B$

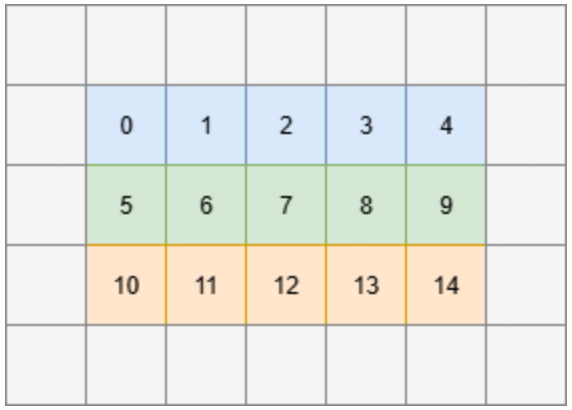


Each cell = dot product of row and column

Optimization → Each tile = dot product of row slice and transposed column slice

Distribution → Each tile computed by a separate process

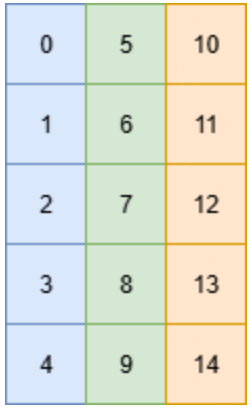
Expressing layouts as MPI datatypes



tile in a row-major matrix

```
send(tile, …)
```

```
1. mpi_scalar = MPI_INT
2. mpi_row = hvector(5, 1, 4, mpi_scalar)
3. mpi_tile = hvector(3, 1, 80, mpi_row)
```



column-major matrix

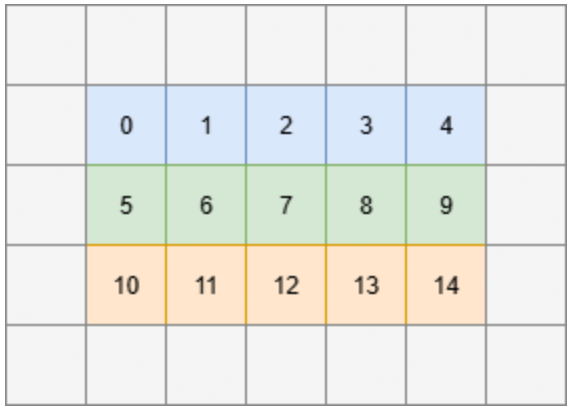
```
recv(matrix, …)
```

```
1. mpi_scalar = MPI_INT
2. mpi_row = hvector(5, 1, 12, mpi_scalar)
3. mpi_matrix = hvector(3, 1, 4, mpi_row)
```

```
auto trav = traverser(tile, matrix);

auto mpi_tile = mpi_transform(tile, trav);
auto mpi_matrix = mpi_transform(matrix, trav);
```

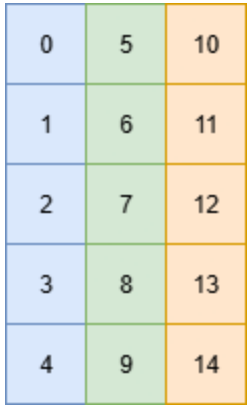
Expressing layouts as MPI datatypes



tile in a row-major matrix

```
send(tile, …)
```

```
1. mpi_scalar = MPI_INT
2. mpi_row = hvector(5, 1, 4, mpi_scalar)
3. mpi_tile = hvector(3, 1, 80, mpi_row)
```



column-major matrix

```
recv(matrix, …)
```

```
1. mpi_scalar = MPI_INT
2. mpi_row = hvector(5, 1, 12, mpi_scalar)
3. mpi_matrix = hvector(3, 1, 4, mpi_row)
```

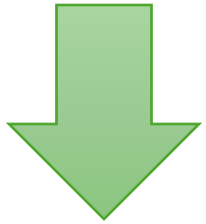
```
auto trav = traverser(tile, matrix);

send(tile, trav, 1);
recv(matrix, trav, 0);
```

Expressing layouts as MPI datatypes

```
1. mpi_scalar = MPI_INT
2. mpi_row = hvector(5, 1, 4, mpi_scalar)
3. mpi_tile = hvector(3, 1, 80, mpi_row)
```

```
1. mpi_scalar = MPI_INT
2. mpi_row = hvector(5, 1, 12, mpi_scalar)
3. mpi_matrix = hvector(3, 1, 4, mpi_row)
```



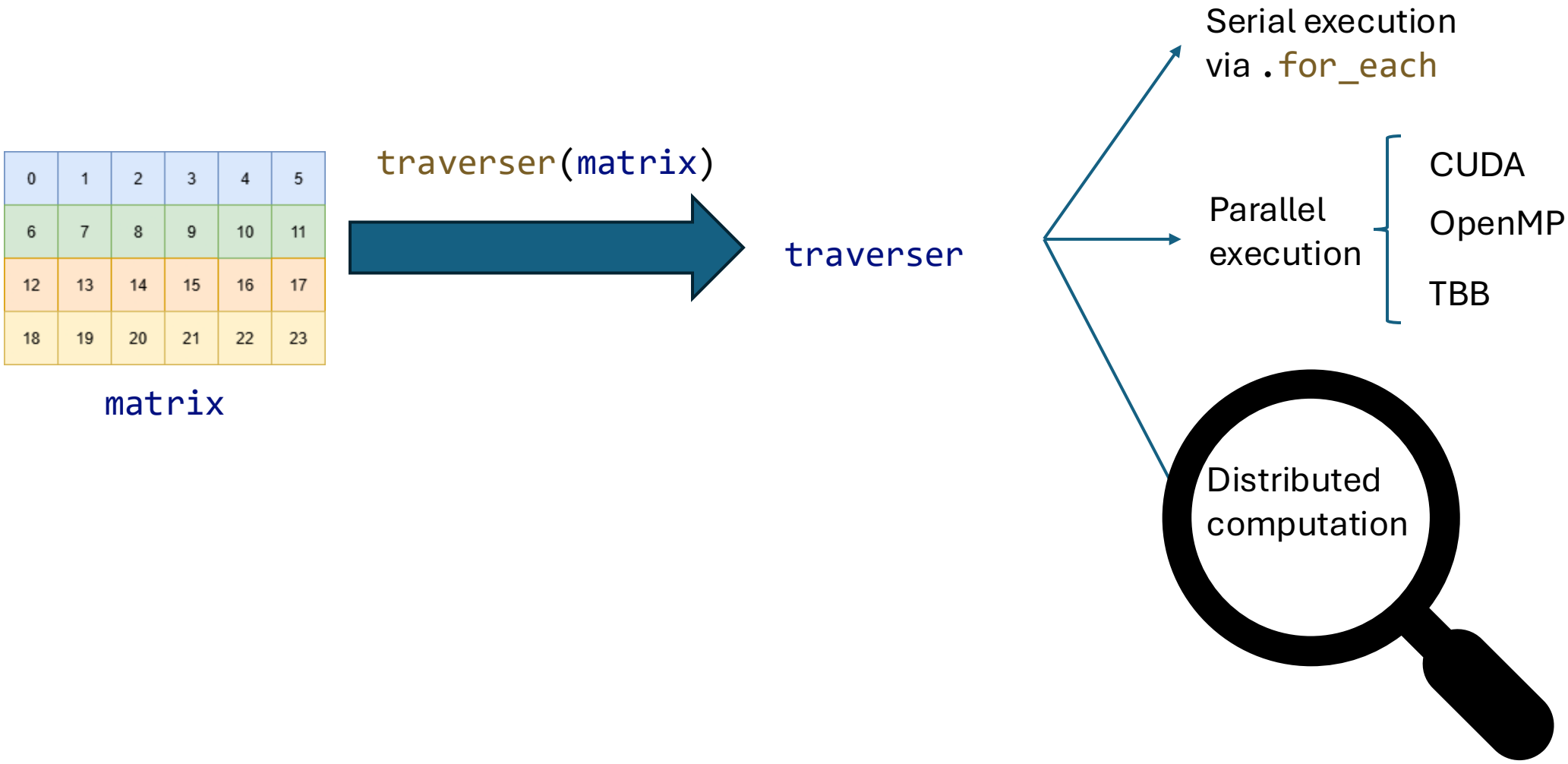
```
1. mpi_scalar = MPI_INT
2. mpi_col = hvector(3, 1, 80, mpi_scalar)
3. mpi_tile = hvector(5, 1, 4, mpi_col)
```

```
1. mpi_scalar = MPI_INT
2. mpi_col = hvector(3, 1, 4, mpi_scalar)
3. mpi_matrix = hvector(5, 1, 12, mpi_col)
```

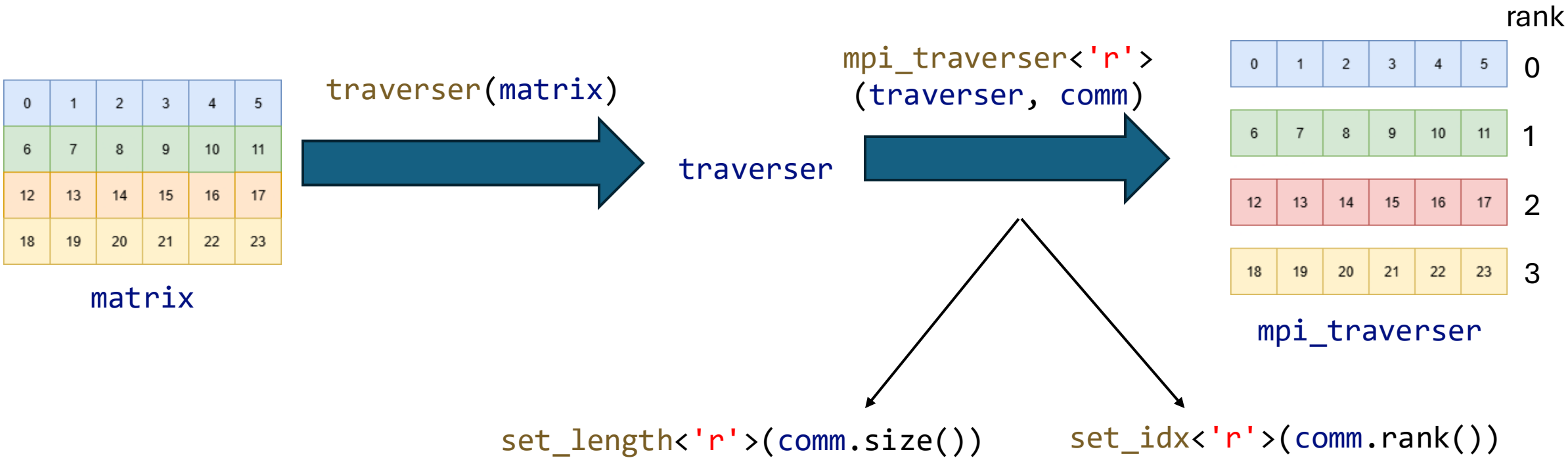
```
auto trav2 = traverser(tile, matrix) ^ hoist<'r'>();

auto mpi_tile2 = mpi_transform(tile, trav2)
auto mpi_matrix2 = mpi_transform(matrix, trav2)
```

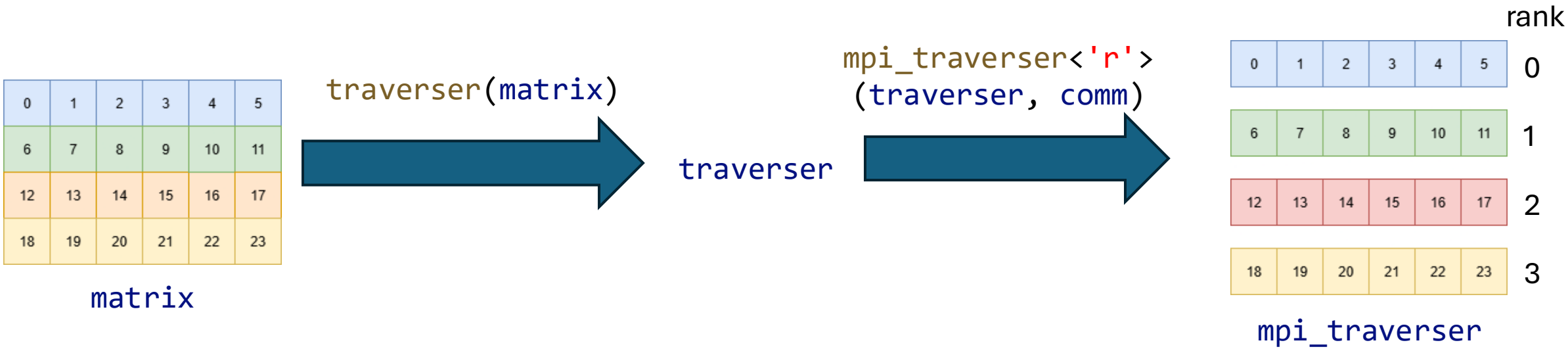
Assigning workers to traversals



Assigning workers to traversals



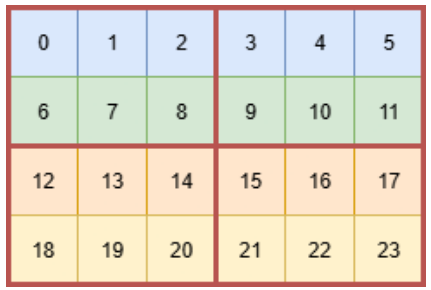
Assigning workers to traversals




```
mpi_traverser<'r'>(traverser(matrix), comm)
  .for_each([&](auto idx) {
    // r == rank(Comm)
    auto [r, c] = get_indices<'r', 'c'>(idx);
    … matrix[idx] …
  });
```

Assigning workers to traversals

'R' ~ Tile row index
'C' ~ Tile column index



grid

traverser(grid)

traverser

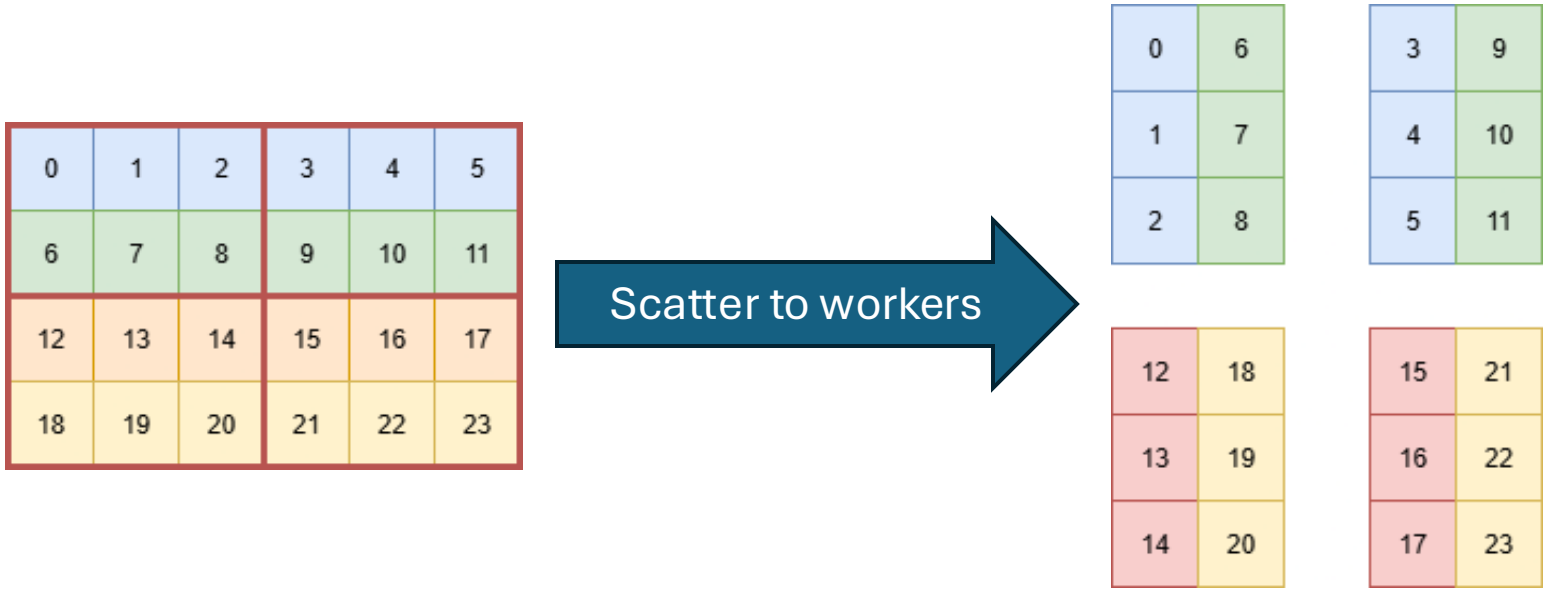
merge_blocks<'R', 'C', 't'>()

mpi_traverser<'t'>
(traverser, comm)


```
mpi_traverser<'t'>(traverser(grid) ^ merge_blocks<'R', 'C', 't'>(), comm)
  .for_each([&](auto idx) {
    auto [r, c, R, C] = get_indices<'r', 'c', 'R', 'C'>(idx);
    … grid[idx] …
  });
```

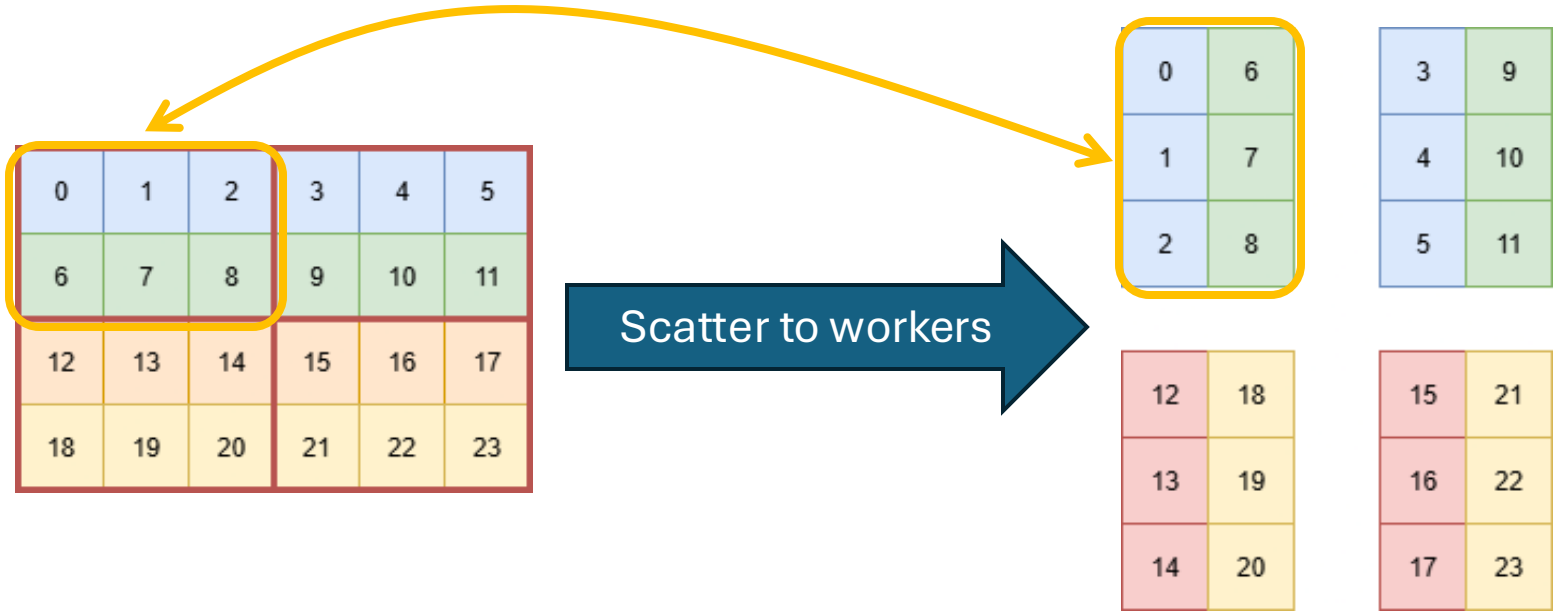
Scatter and transpose

```
MPI_Scatterv(send_buff, 1, offsets, send_type, 1, recv_buff, recv_type, root, comm)
```



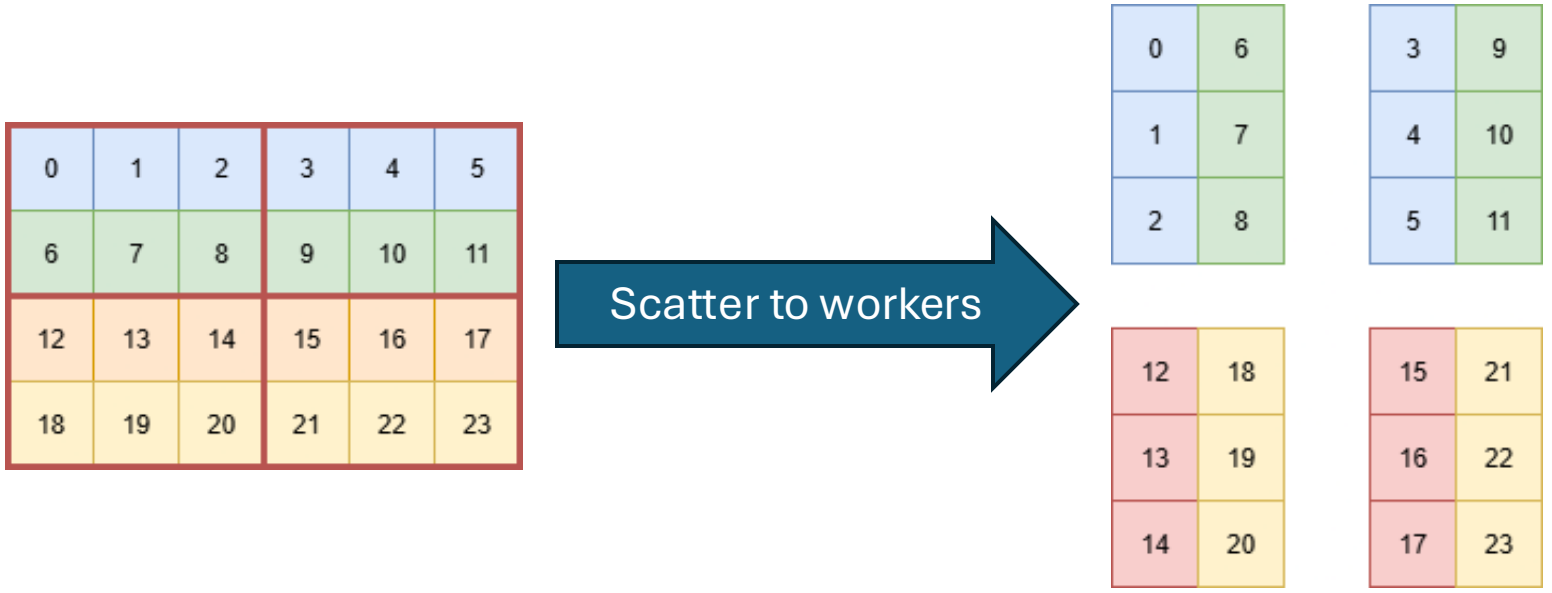
Scatter and transpose

```
MPI_Scatterv(send_buff, 1, offsets, send_type, 1, recv_buff, recv_type, root, comm)
```



Scatter and transpose

```
MPI_Scatterv(send_buff, 1, offsets, send_type, 1, recv_buff, recv_type, root, comm)
```




```
scatter(grid, local_tile, mpi_traverser, root)
```

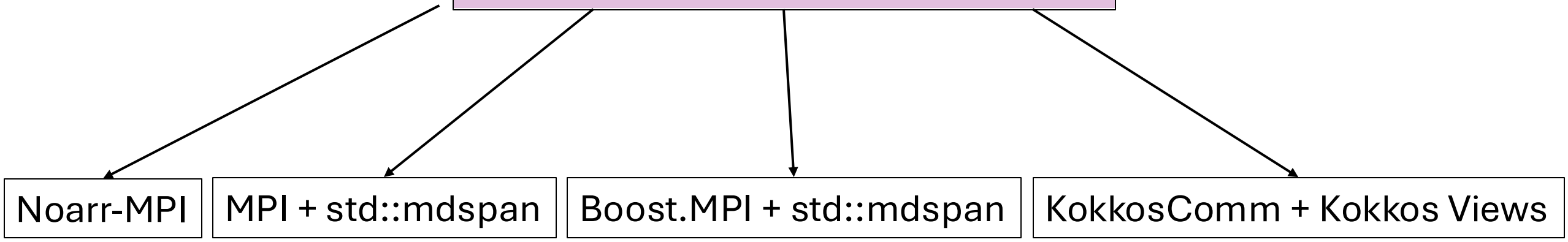
Comparison of abstractions on matrix multiplication

```
auto localC = …;
auto localA = …;
auto localB = …;

scatter(C, localC, …);
scatter(A, localA, …);
scatter(B, localB, …);

computation(localC, localA, localB, …);

gather(localC, C, …);
```



Comparison of abstractions on matrix multiplication

```
auto localC = …;
auto localA = …;
auto localB = …;
```

Noarr-MPI

```
scalar<int>() ^ vector<'i'>()
              ^ vector<'j'>();
```

- Sizes can be bound to the grid tiles

```
lengths_like<'i', 'j'>(grid);
```

- Grid tiles can be bound to ranks

```
mpi_traverser<'r'>(…, comm);
```

Others

```
tileC = submdspan(C, {iStart, iEnd},
                     {jStart, jEnd});
```

- Values computed explicitly

```
iStart = NI / rowRanks * (rank % rowRanks);
```

Comparison of abstractions on matrix multiplication

```
scatter(C, localC, …);
scatter(A, localA, …);
scatter(B, localB, …);
…
gather(localC, C, …);
```

Noarr-MPI

- `scatter(C, localC, mpi_trav, 0)`
  - Automatically constructs MPI datatypes
  - Automatically computes offsets

Others

- Manual expressing of layouts as MPI datatypes
- Manual computation of offsets

Comparison of abstractions on matrix multiplication

```
…
computation(localC, localA, localB, …);
…
```

Noarr-MPI

```
traverser.for_each([&](auto idx) {
  localC[idx] += localA[idx] * localB[idx];
});
```

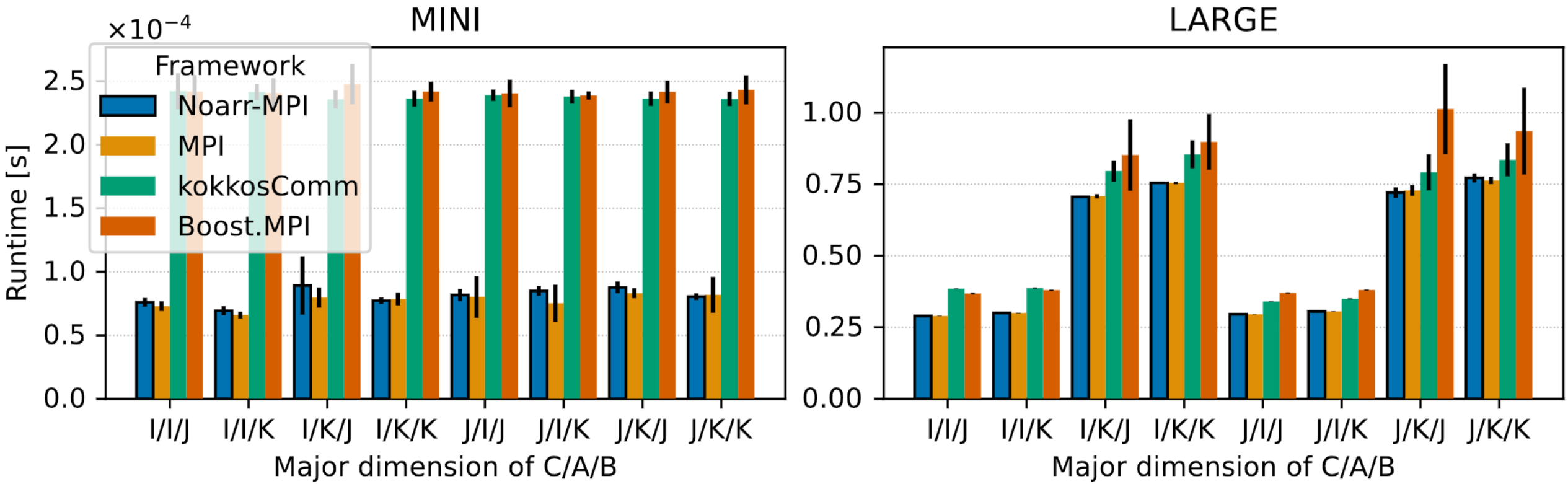
Others

```
for (int i = 0; i < ROWS; i++)
 for (int j = 0; j < COLS; j++)
  for (int k = 0; k < K; k++)
   localC(i, j) += localA(i, k) * localB(k, j);
```

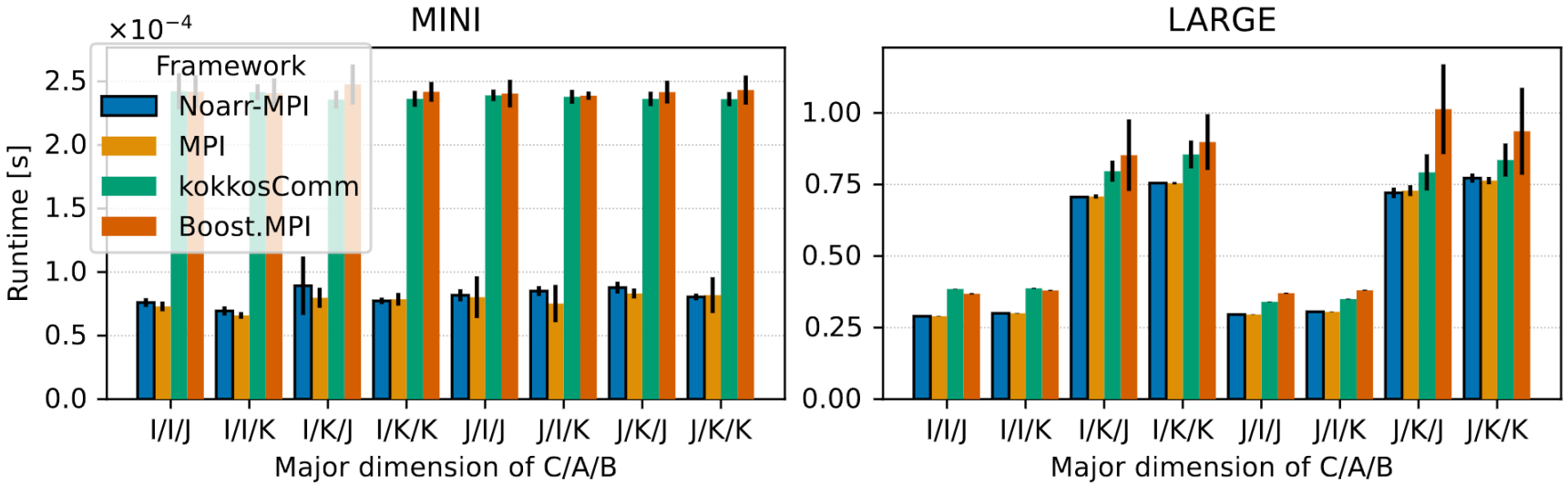
Table 1: Comparison of the MPI abstractions

	Noarr-MPI	native MPI	Boost.MPI	MPP	MPL	Kokkos	KaMPIng
Auto-transforms	✓	✗	✗	✗	✗	✗	✗
Non-contiguous	✓	✓	✓	✓	✓	✓	✗
Mdspan-like	✓	✗	✗	✗	✗	✓	✗
No serialization	✓	✓	✗	✓	✓	✓	✓
Type-safety	✓	✗	✓	✓	✓	✓	✓
Scatter/gather	✓	✓	✗	✗	✓	✗	✗

We don't pay anything extra



Thank you for attention



https://github.com/jiriklepl/noarr-mpi

klepl@d3s.mff.cuni.cz