MPI types, Scatter and Scatterv

MPI types, Scatter and Scatterv

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

Logical and physical layout of a C/C++ array in memory.

A = malloc(6*6*sizeof(int));

35	1	2	3	4	5	6	7	8	9	10	. . .	30	31	32	33	34	35

MPI_Scatter

```
int MPI_Scatter(
    const void *sendbuf,       // data to send
    int sendcount,             // sent to each process
    MPI_Datatype sendtype,     // type of data sent
    void *recvbuf,             // where received
    int recvcount,             // how much to receive
    MPI_Datatype recvtype,     // type of data received
    int root,                  // sending process
    MPI_Comm comm)             // communicator
```

`sendbuf`, `sendcount`, `sendtype` valid only at the sending process

Equal number elements to all processors

A

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

```
int MPI_Scatter(A, 9, MPI_Int, B, 9,
                MPI_Int,0,
                MPI_COMM_WORLD)
```

P_0	0	1	2	3	4	5	6	7	8
P_1	9	10	11	12	13	14	15	16	17
P_2	18	19	20	21	22	23	24	25	26
P_3	27	28	29	30	31	32	33	34	35

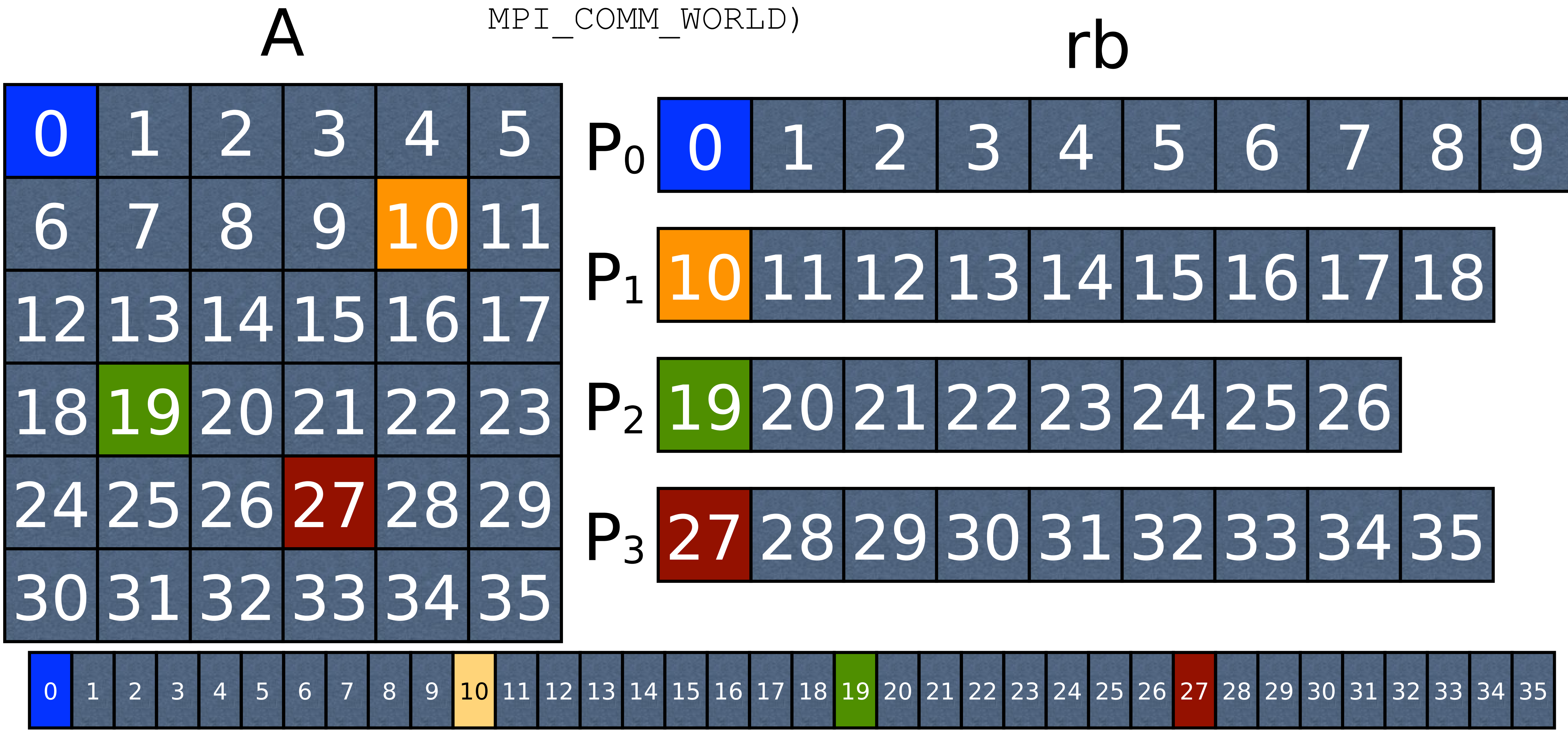
MPI_Scatterv

```
int MPI_Scatter(
     const void *sendbuf,   // data to send
     const int *sendcounts, // sent to each process
     const int* displ       // where in sendbuf
                            // sent data is
     MPI_Datatype sendtype, // type of data sent
     void *recvbuf,         // where received
     int recvcount,         // how much to receive
     MPI_Datatype recvtype, // type of data received
     int root,              // sending process
     MPI_Comm comm)         // communicator
```

`sendbuf`, `sendcount`, `sendtype` valid only at the sending process

Specify the number elements sent to each processor

```
int[] counts = {10, 9, 8, 9};
int[] displ = {0, 10, 19, 27};
int MPI_Scatterv(A, counts, displs, MPI_Int,rb, counts, MPI_Int 0,
                 MPI_COMM_WORLD)
```



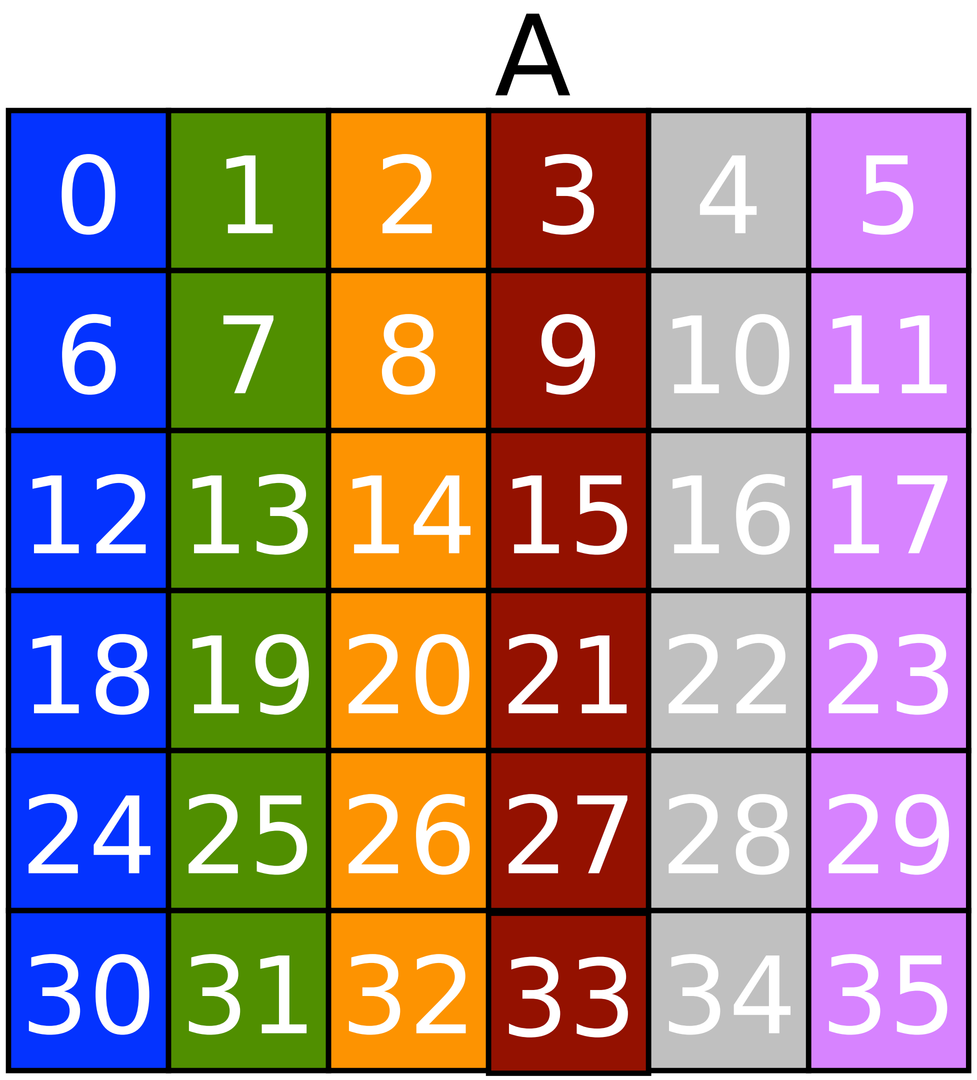
MPI_Type_vector

```
int MPI_Type_vector(
     int count,               // number of blocks
     int blocklength,         // #elts in a block
     int stride,              // #elts between block starts
     MPI_Datatype oldtype,    // type of block elements
     MPI_Datatype *newtype    // handle for new type
)
```

Allows a type to be created that puts together blocks of elements in a vector into another vector.

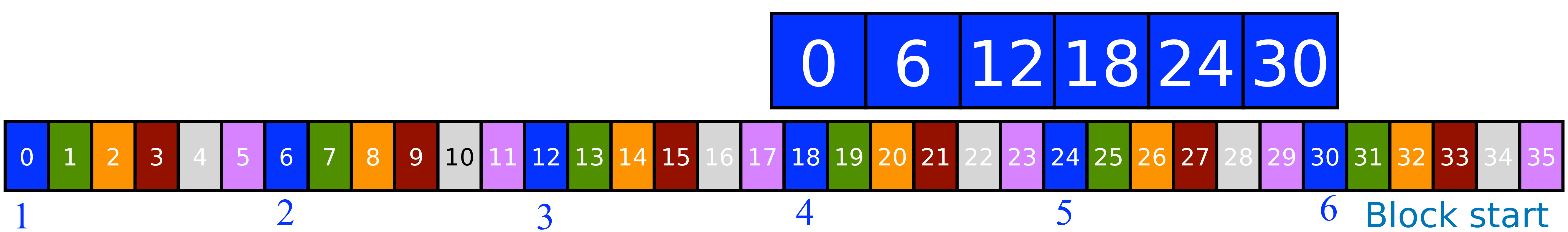
Note that a 2-D array in contiguous memory can be treated as a 1-D vector.

MPI_Type_vector: defining the type

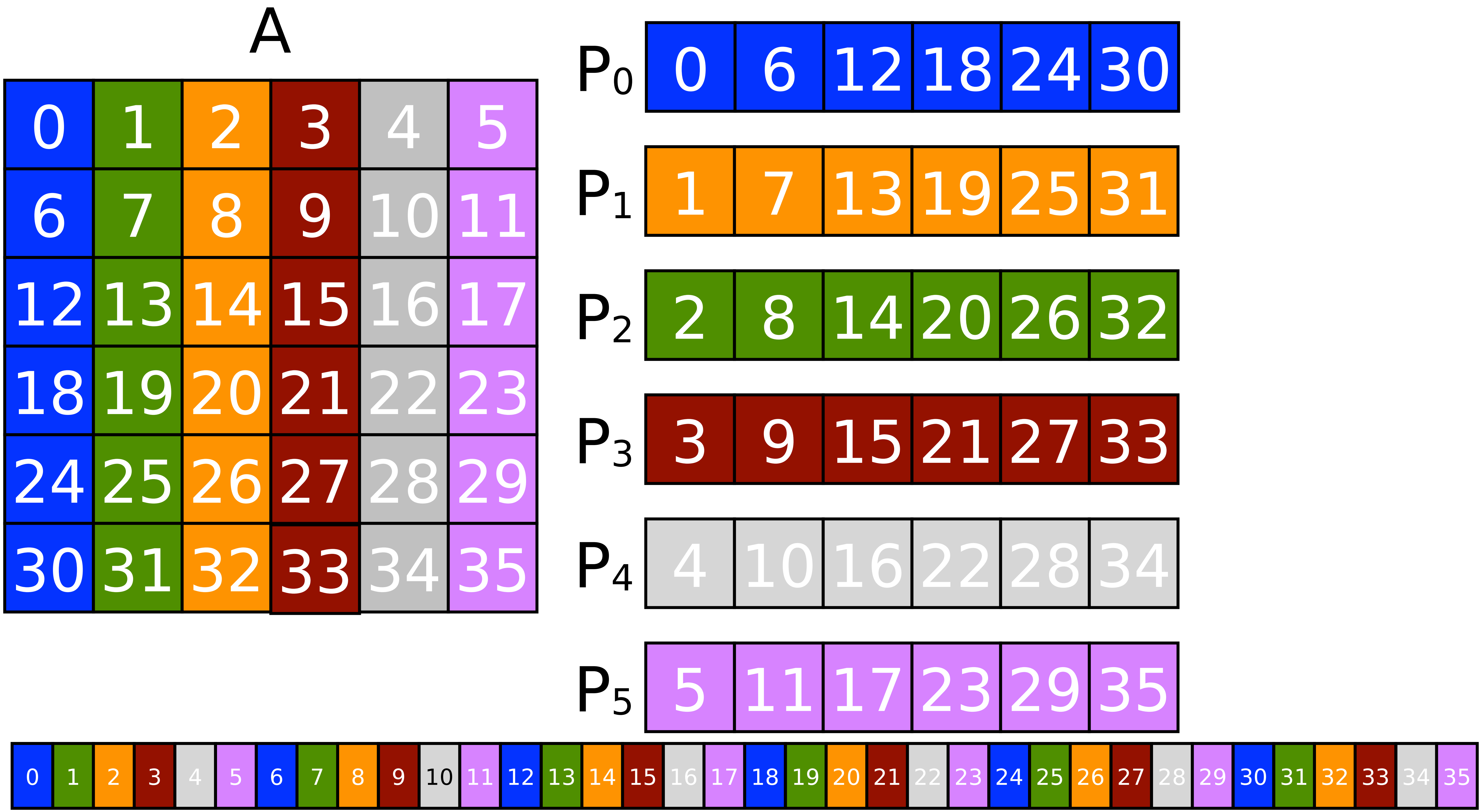



```
MPI_Datatype col, coltype;
MPI_Type_vector(6, 1, 6, MPI_INT,
                &col);
MPI_Type_commit(&col);
MPI_Send(A, 1, col, P-1,
         MPI_ANY_TAG,
         MPI_Comm_World);
```

There are 6 blocks, and each is made of 1 int, and the new block starts 6 positions in the linearized array from the start of the previous block.



What if we want to scatter columns (C array layout)



What if we want to scatter columns?

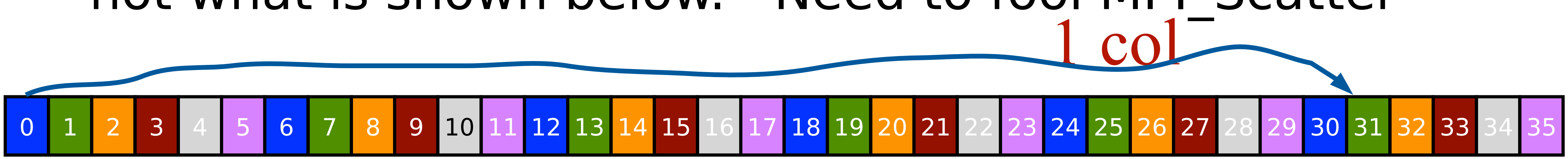
A

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

```
MPI_Datatype col, coltype;
MPI_Type_vector(1, 1, 6, MPI_INT,
                &col);
MPI_Type_commit(&col);
int MPI_Scatter(A, 6, col, AC, 6,
                MPI_Int, 0,
                MPI_Comm_World);
```

The code above won’t work.
Why?
Where does the first col end?

We want the first column to end at 0, the second at 1, etc.
– not what is shown below. Need to fool MPI_Scatter



MPI_Type_create_resized to the rescue

```
int MPI_Type_create_resized(
    MPI_Datatype oldtype,  // type being resized

    MPI_Aint lb, // new lower bound

    MPI_Aint extent, // new extent ("length")
    MPI_Datatype *newtype) // resized type name
)
```

Allows a new size (or *extent*) to be assigned to an existing type.

Allows MPI to determine how far from an object O1 the next adjacent object O2 is. As we will see this is often necessitated because we treat a logically 2-D array as a 1-D vector.

Using MPI_Type_vector

A

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

```
MPI_Datatype col, coltype;

MPI_Type_vector(6, 1, 6, MPI_INT,&col);

MPI_Type_commit(&col);

MPI_Type_create_resized(col, 0,
1*sizeof(int), &coltype);

MPI_Type_commit(&coltype);

MPI_Scatter(A, 1, coltype, rb, 6,
MPI_Int, 0, MPI_COMM_WORLD);
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

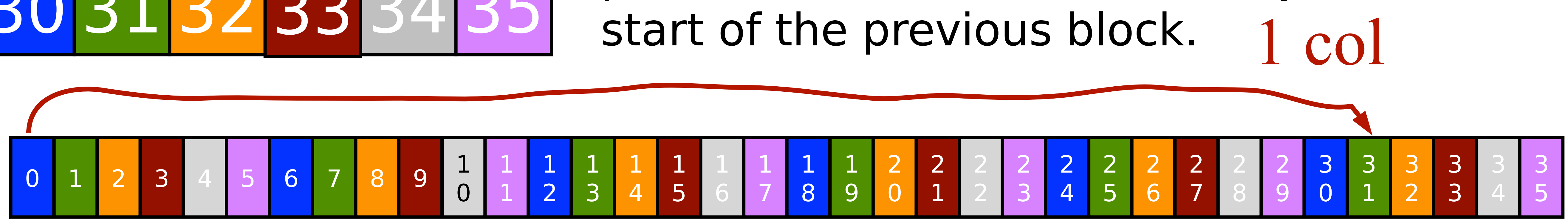
MPI_Type_vector: defining the type

A

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

```
MPI_Datatype col, coltype;
MPI_Type_vector(6, 1, 6, MPI_INT,
                &col);
MPI_Type_commit(&col);
MPI_Type_create_resized(col, 0,
   1*sizeof(int), &coltype);
MPI_Type_commit(&coltype);
MPI_Scatter(A, 1, coltype, rb,
   6, MPI_Int, 0, MPI_COMM_WORLD);
```

Again, there are 6 blocks, and each is made of 1 int, and the new block starts 6 positions in the linearized array from the start of the previous block.



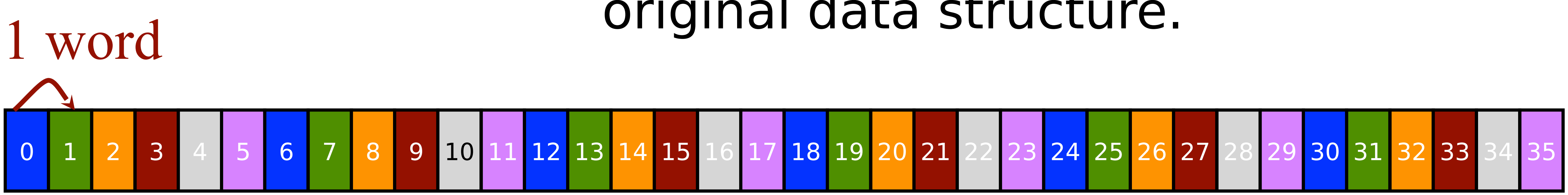
Using MPI_type_create_resized

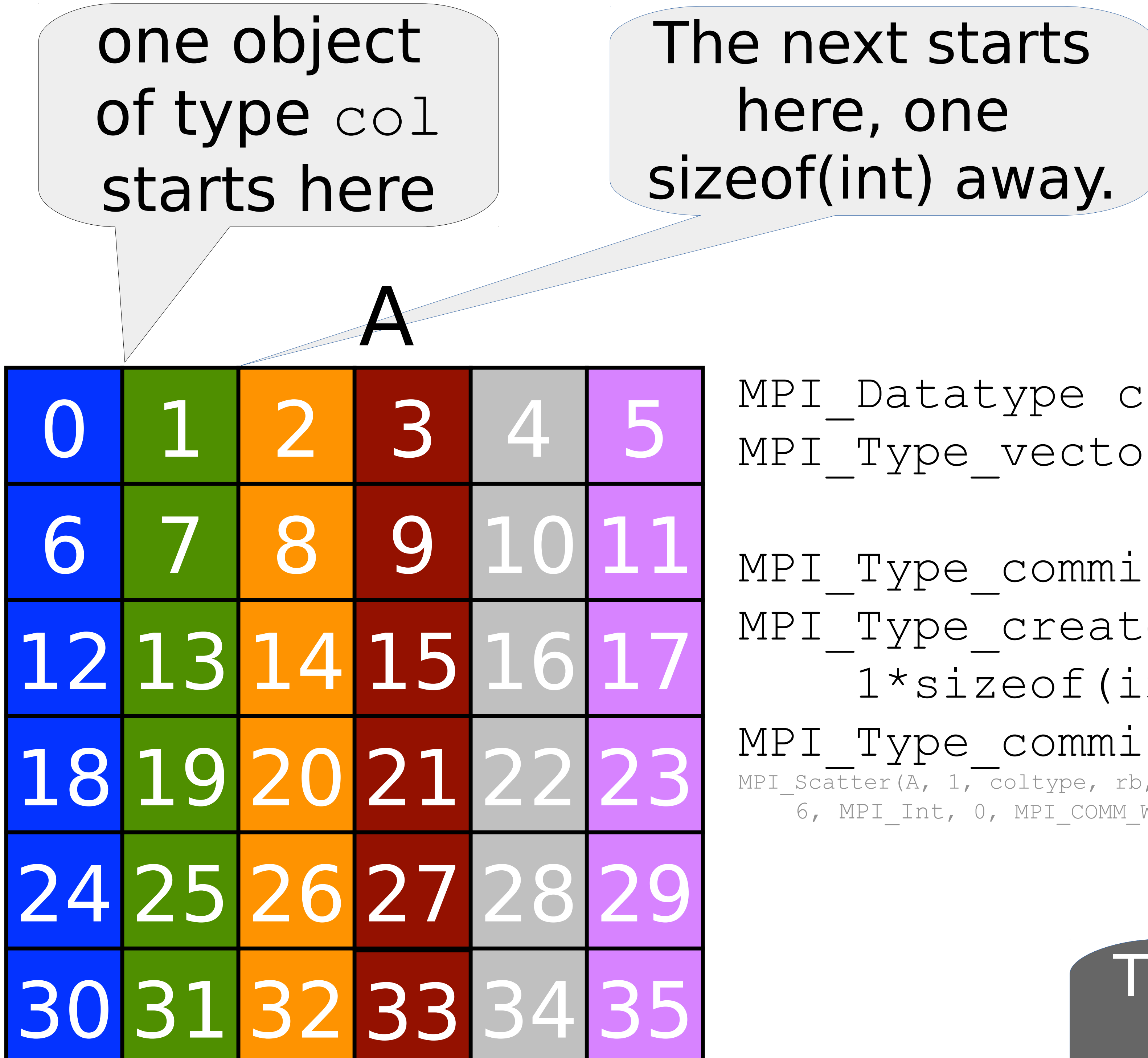
A

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

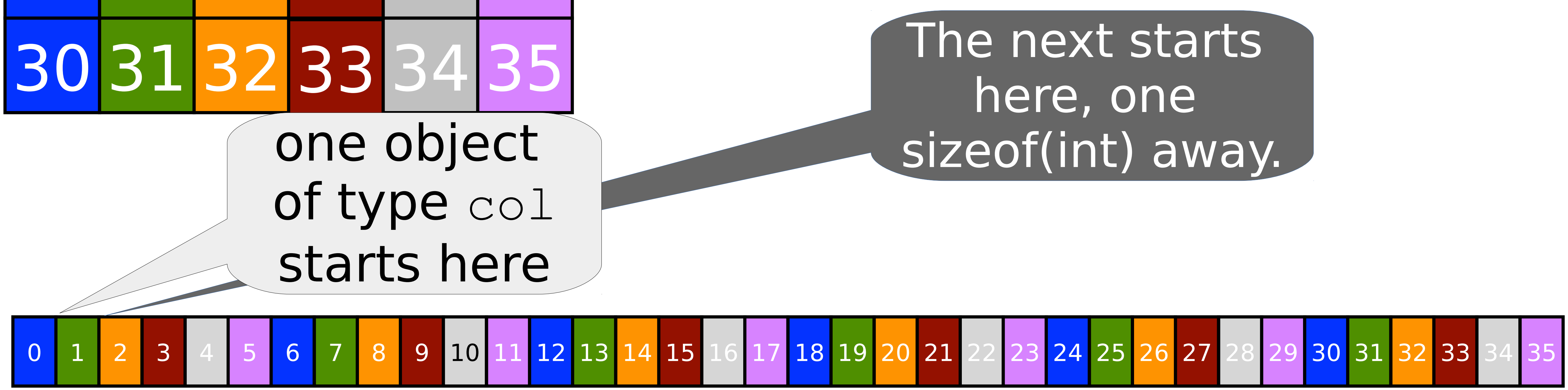
```
MPI_Datatype col, coltype;
MPI_Type_vector(6, 1, 6, MPI_INT,
                &col);
MPI_Type_commit(&col);
MPI_Type_create_resized(col, 0,
    1*sizeof(int), &coltype);
MPI_Type_commit(&coltype);
MPI_Scatter(A, 1, coltype, rb,
    6, MPI_Int, 0, MPI_COMM_WORLD);
```

resize creates a new type from a previous type and changes the *size*. This allows easier computation of the offset from one element of a type to the next element of a type in the original data structure.

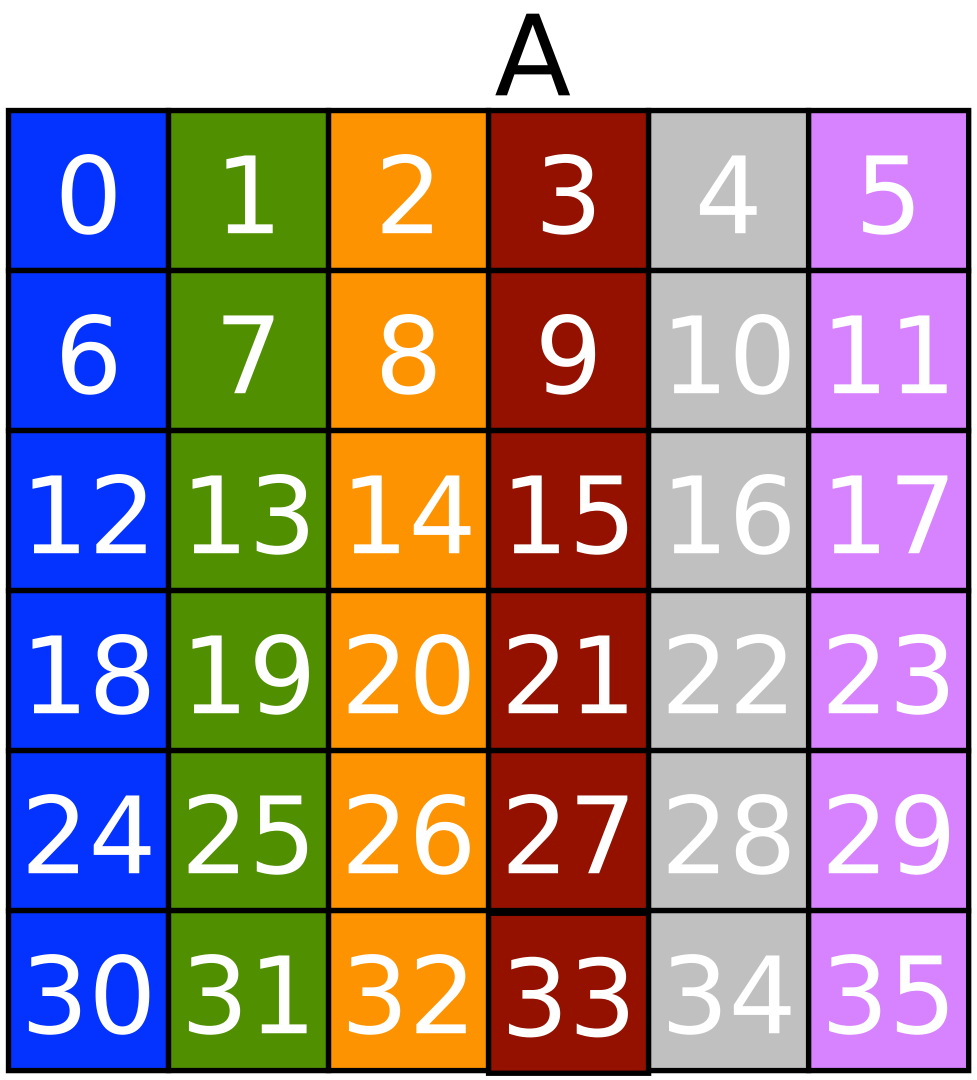




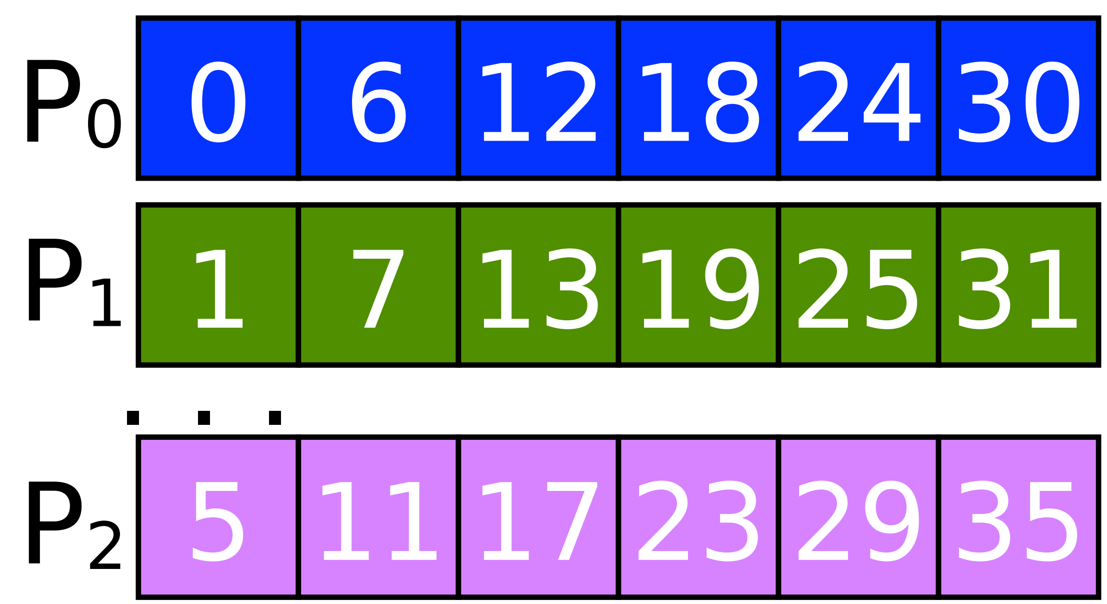

```
MPI_Datatype col, coltype;
MPI_Type_vector(6, 1, 6, MPI_INT,
                &col);
MPI_Type_commit(&col);
MPI_Type_create_resized(col, 0,
    1*sizeof(int), &coltype);
MPI_Type_commit(&coltype);
MPI_Scatter(A, 1, coltype, rb,
    6, MPI_Int, 0, MPI_COMM_WORLD);
```

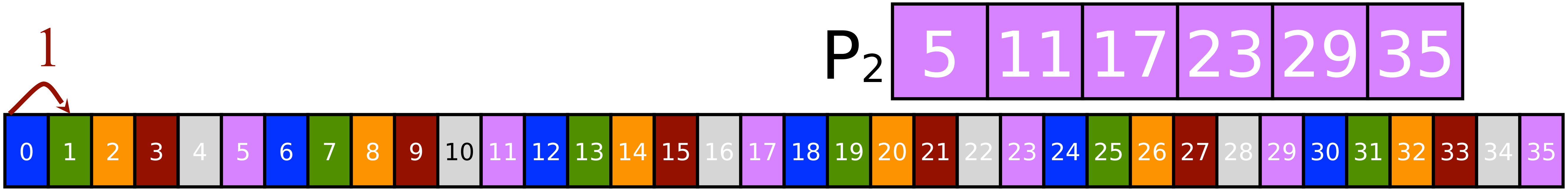


The result of the communication

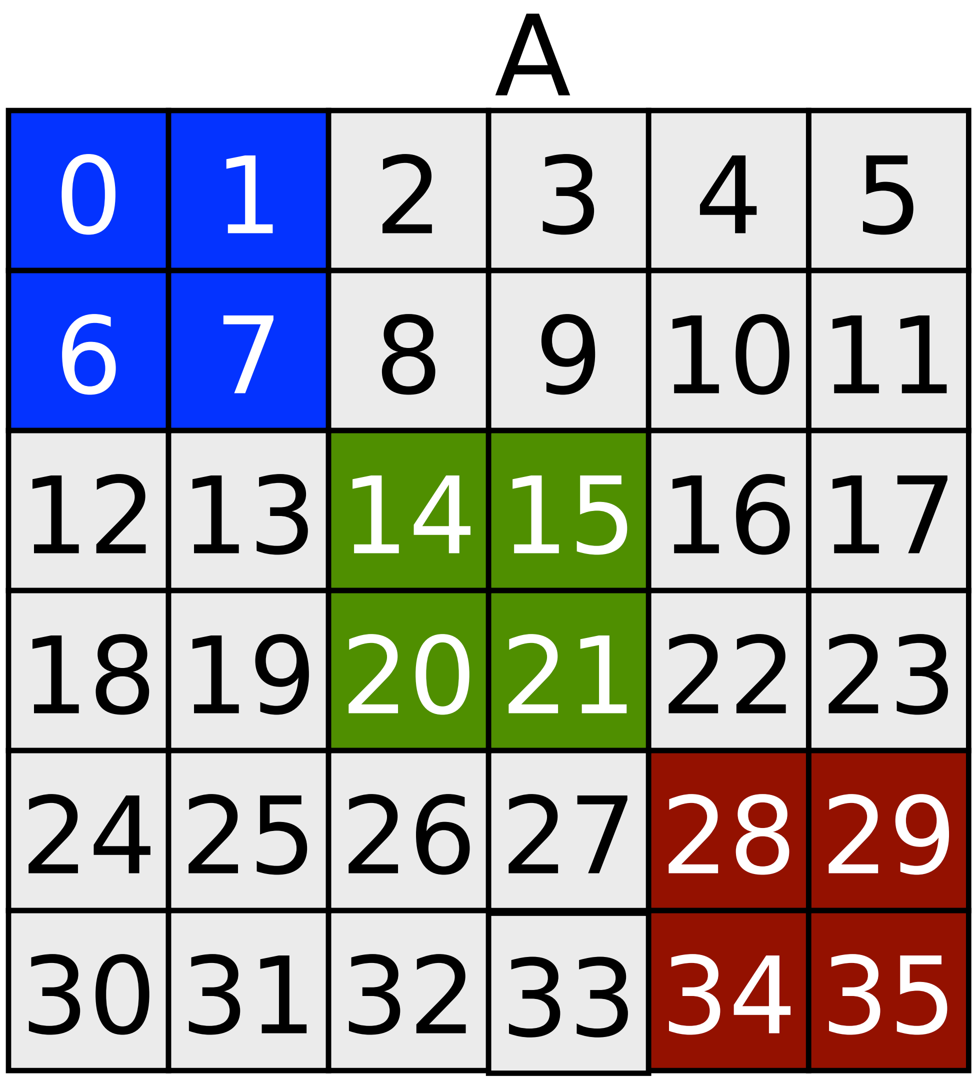



```
MPI_Datatype col, coltype;
MPI_Type_vector(6, 1, 6, MPI_INT,
                &col);
MPI_Type_commit(&col);
MPI_Type_create_resized(col, 0,
    1*sizeof(int), &coltype);
MPI_Type_commit(&coltype);
MPI_Scatter(A, 1, coltype, rb,
    6, MPI_Int, 0, MPI_COMM_WORLD);
```



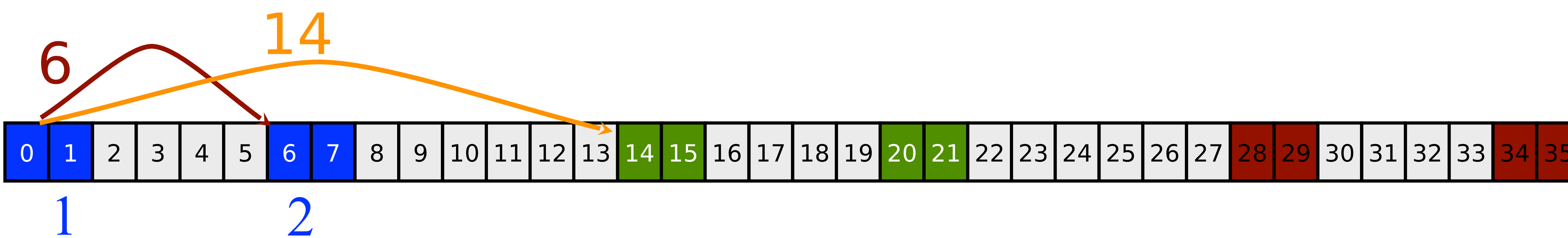


Scattering diagonal blocks



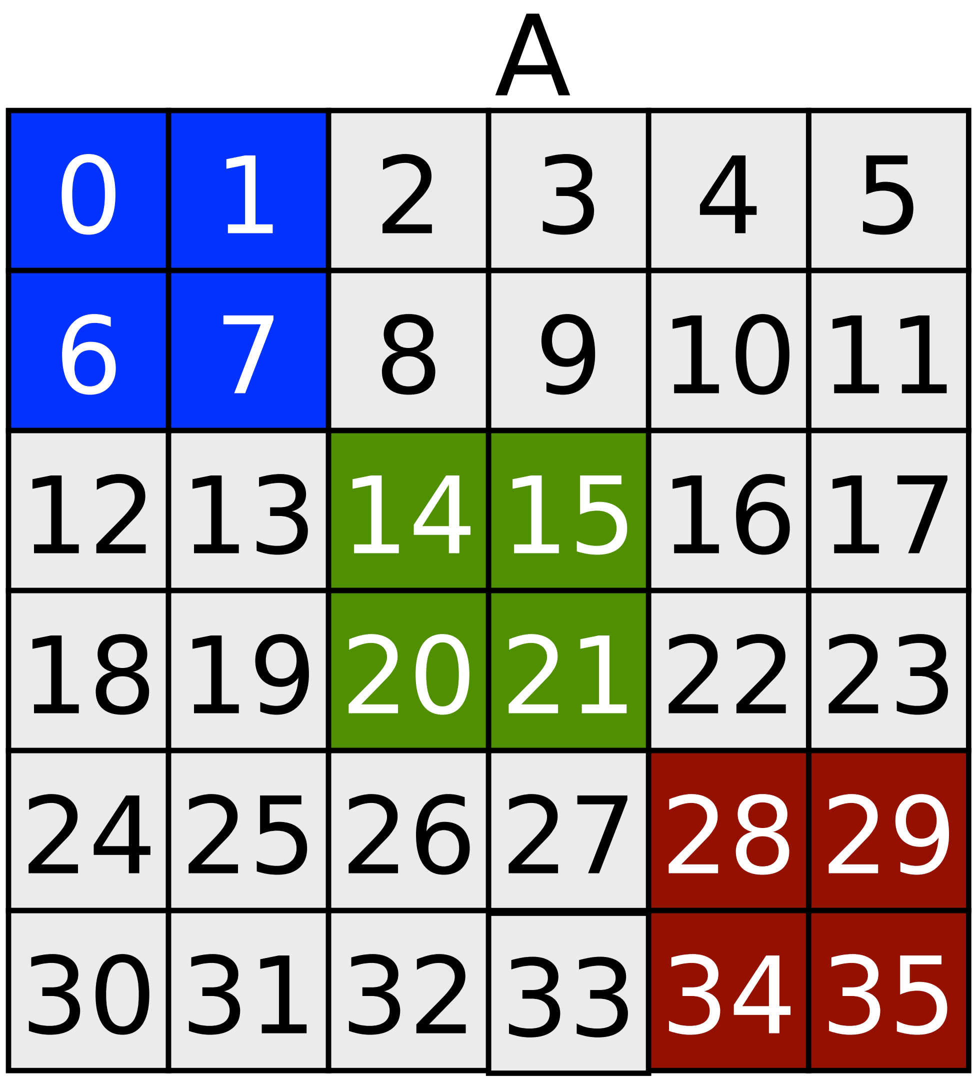

```
MPI_Datatype block, blocktype;
MPI_Type_vector(2, 2, 6, MPI_INT,
                  &block);
MPI_Type_commit(&block);
MPI_Type_create_resized(block, 0,
    14*sizeof(int), &blocktype);
MPI_Type_commit(&blocktype);
int MPI_Scatter(A, 1, blocktype, B, 4,
                 MPI_Int,0,
                 MPI_COMM_WORLD)
```

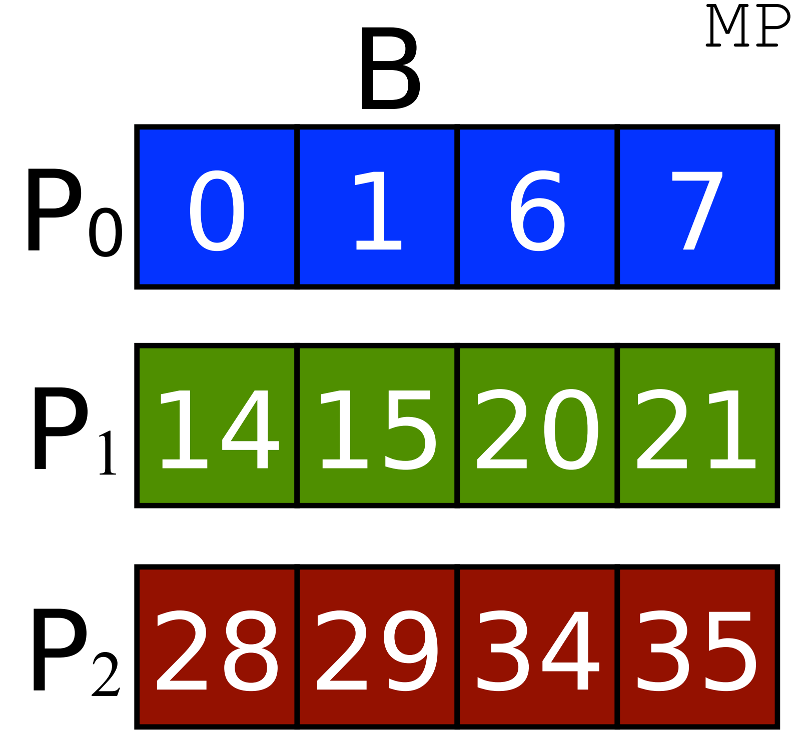
note that 2*numrows + width of block =14



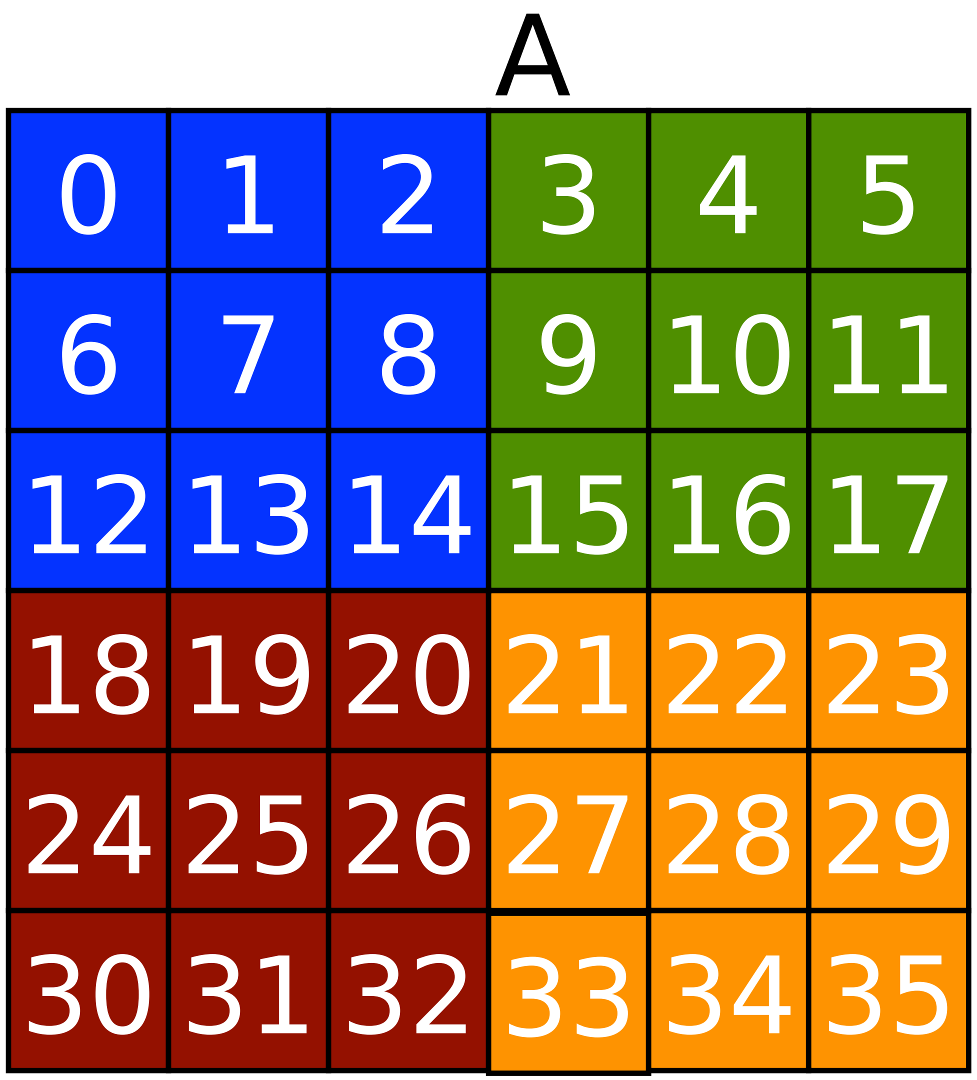
Scattering the blocks

```
MPI_Datatype block, blocktype;
MPI_Type_vector(2, 2, 6, MPI_INT,
                &block);
MPI_Type_commit(&block);
MPI_Type_create_resized(block, 0,
        14*sizeof(int), &blocktype);
MPI_Type_commit(&blocktype);
int MPI_Scatter(A, 1, blocktype, B,
                4,MPI_Int,0,
                MPI_COMM_WORLD)
```

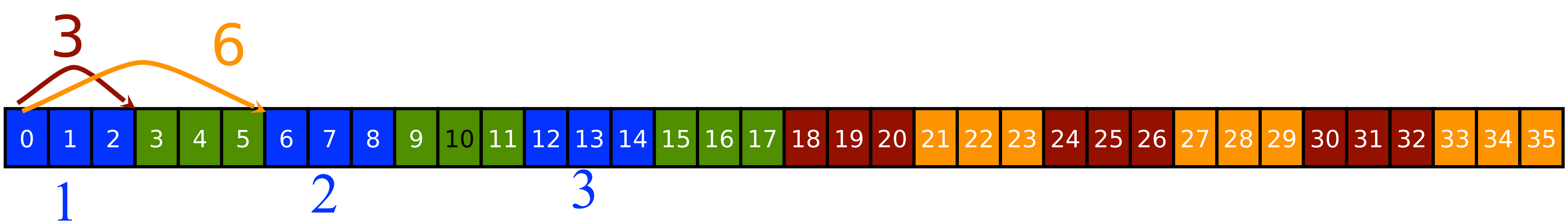




The Type_vector statement describing this




```
MPI_Datatype block, blocktype;
MPI_Type_vector(3, 3, 6, MPI_INT,
                    &block);
MPI_Type_commit(&block);
MPI_Type_create_resized(block, 0,
    3*sizeof(int), &blocktype);
MPI_Type_commit(&blocktype);
```

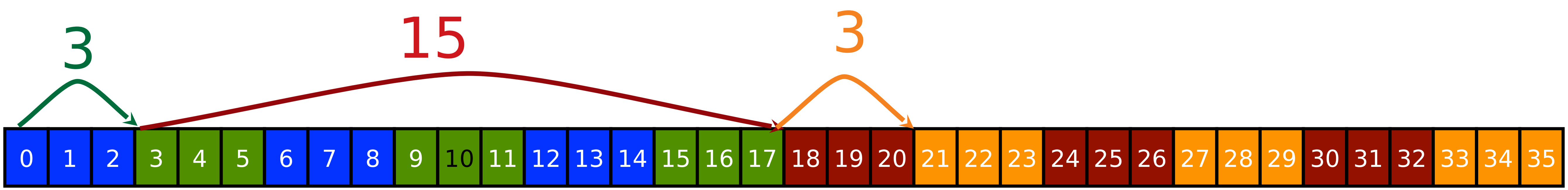


The create_resize statement for this

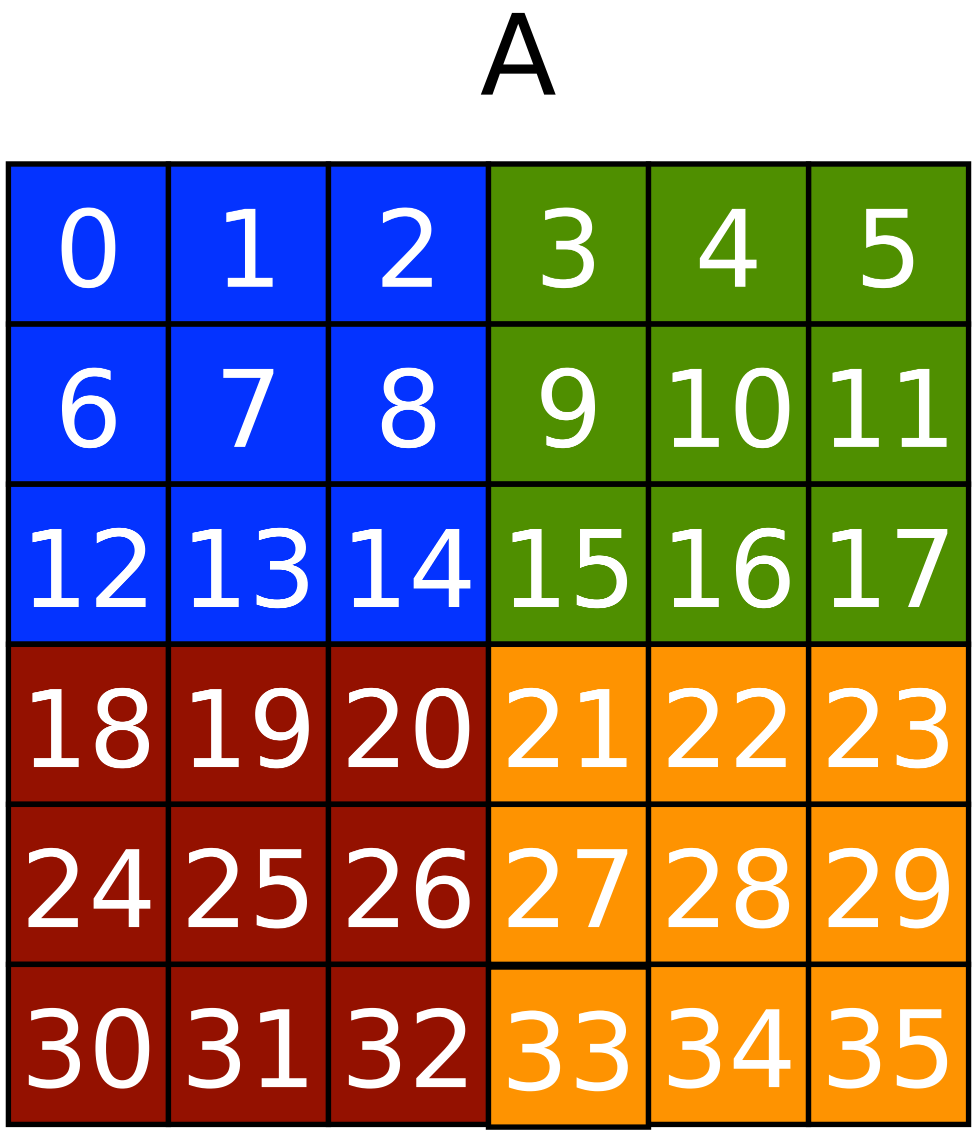
A


```
MPI_Datatype block, blocktype;
MPI_Type_vector(3, 3, 6, MPI_INT,
                &block);
MPI_Type_commit(&block);
MPI_Type_create_resized(block, 0,
     3*sizeof(int), &blocktype);
MPI_Type_commit(&blocktype);
```

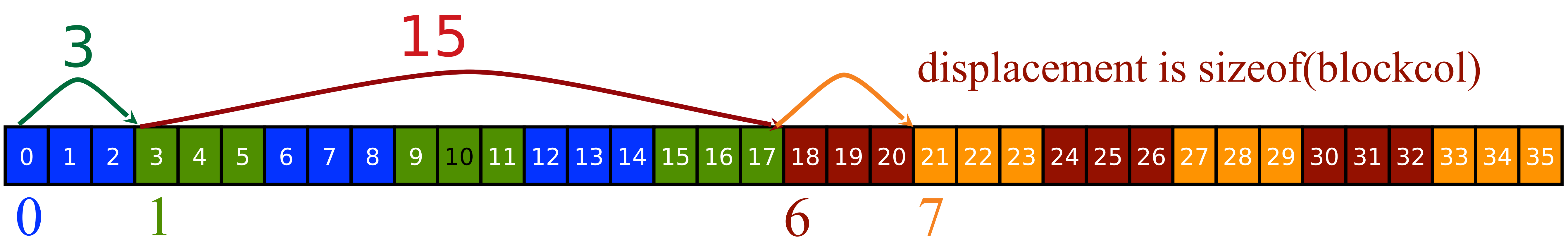
Distance between start of blocks varies, but are multiples of 3. Use MPI_Scatterv



Sending the data



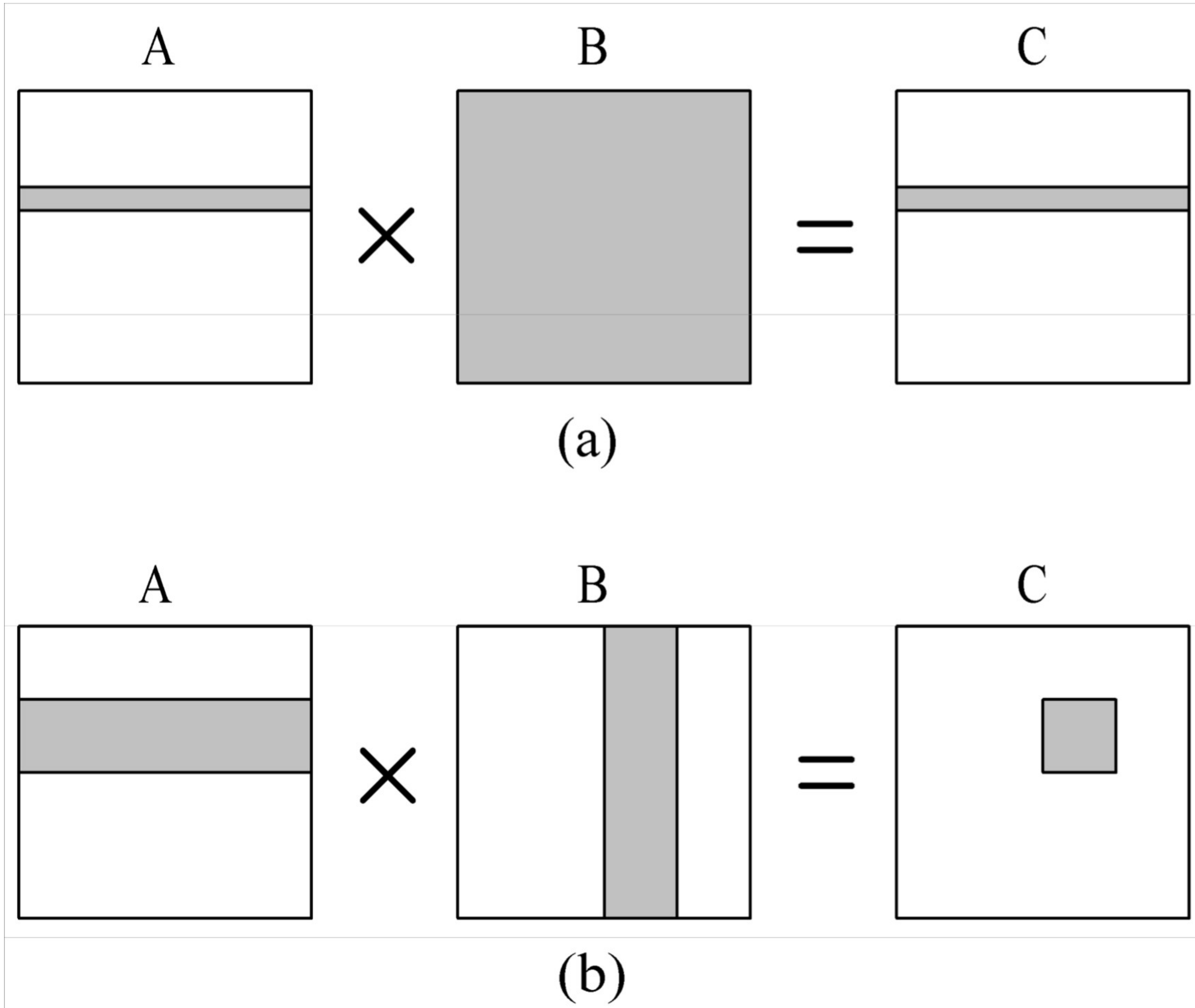

```
MPI_Datatype block, blocktype;
int disp = {0, 1, 6, 7)
int scount = {1, 1, 1, 1}
int rcount = {9, 9, 9, 9}
MPI_Type_vector(3, 3, 6, MPI_INT,
                &block);
MPI_Type_commit(&block);
MPI_Type_create_resized(block, 0,
    3*sizeof(int), &blocktype);
MPI_Type_commit(&blocktype);
int MPI_Scatterv(A, scount, displ,
                 blocktype, rb, rcount,
                 MPI_Int, 0,
                 MPI_COMM_WORLD)
```



Matrix Multiply Cannon's Algorithm

- Useful for the small project
- Algorithm 1 in what follows is the layout we discussed earlier

Elements of A and B Needed to Compute a Process's Portion of C



Algorithm 1

Cannon's Algorithm

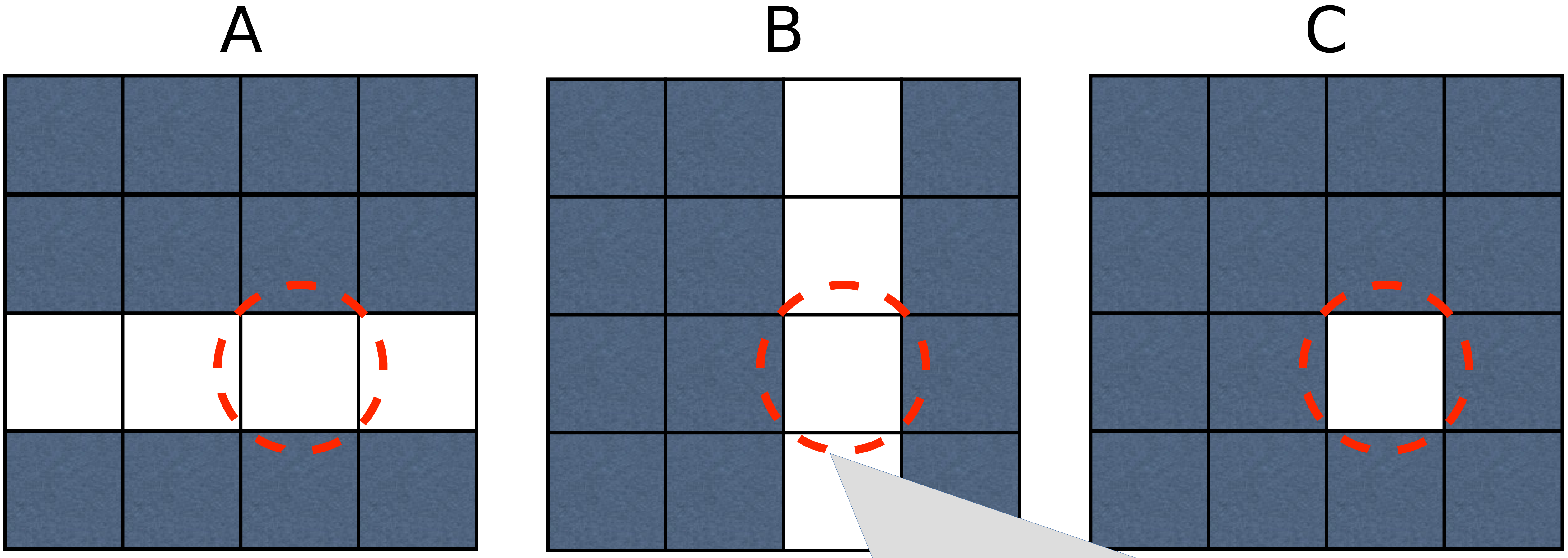
Parallel Algorithm 2 (Cannon's Algorithm)

- Associate a primitive task with each matrix element
- Agglomerate tasks responsible for a square (or nearly square) block of C (the result matrix)
- Computation-to-communication ratio rises to $n / \sqrt{p}$ (same total computation, more computation per communication)
  - $2n / p < n / \sqrt{p}$ when p > 4

Simplifying assumptions

- Assume that
  - *A*, *B* and (consequently) *C* are n x n square matrices
  - $\sqrt{p}$ is an integer, and
  - $n = k \cdot \sqrt{p}$, k an integer (i.e. *n* is a multiple of $\sqrt{p}$

Blocks need to compute a C element

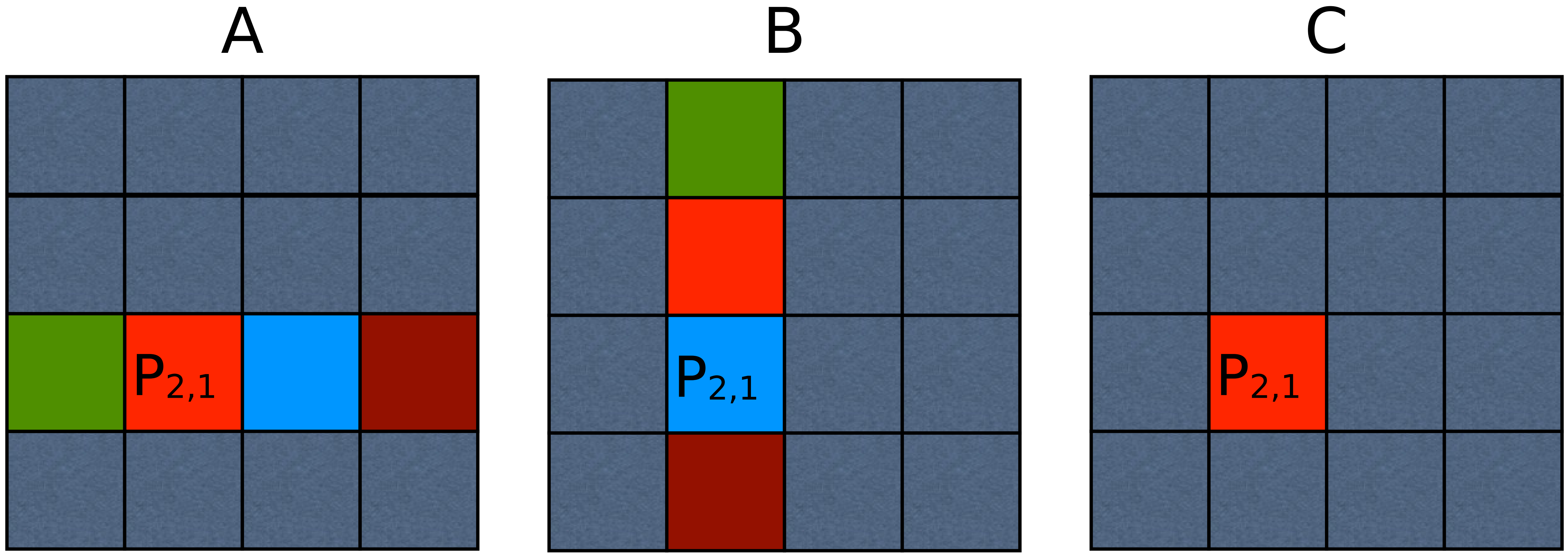


These blocks need to be on the same processor. The processor that owns these blocks fully computes value of the circled C block (but needs more than the circled A and B blocks)

Blocks to compute the C element

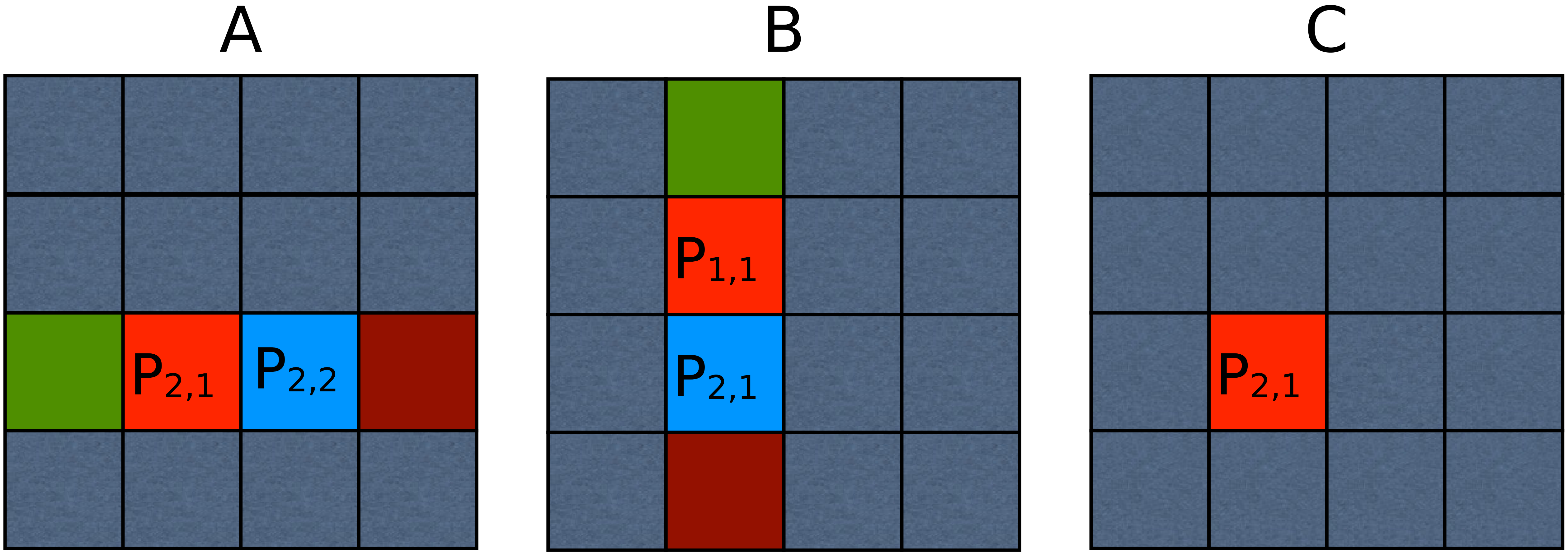
Processor $P_{2,1}$ needs, at some point, to simultaneously hold the green A and B blocks, the red A and B blocks, the blue A and B blocks, and the cayenne A and B blocks.

With the current data layout it cannot do useful work because it does not contain matching A and B blocks (it has a red A and blue B block)



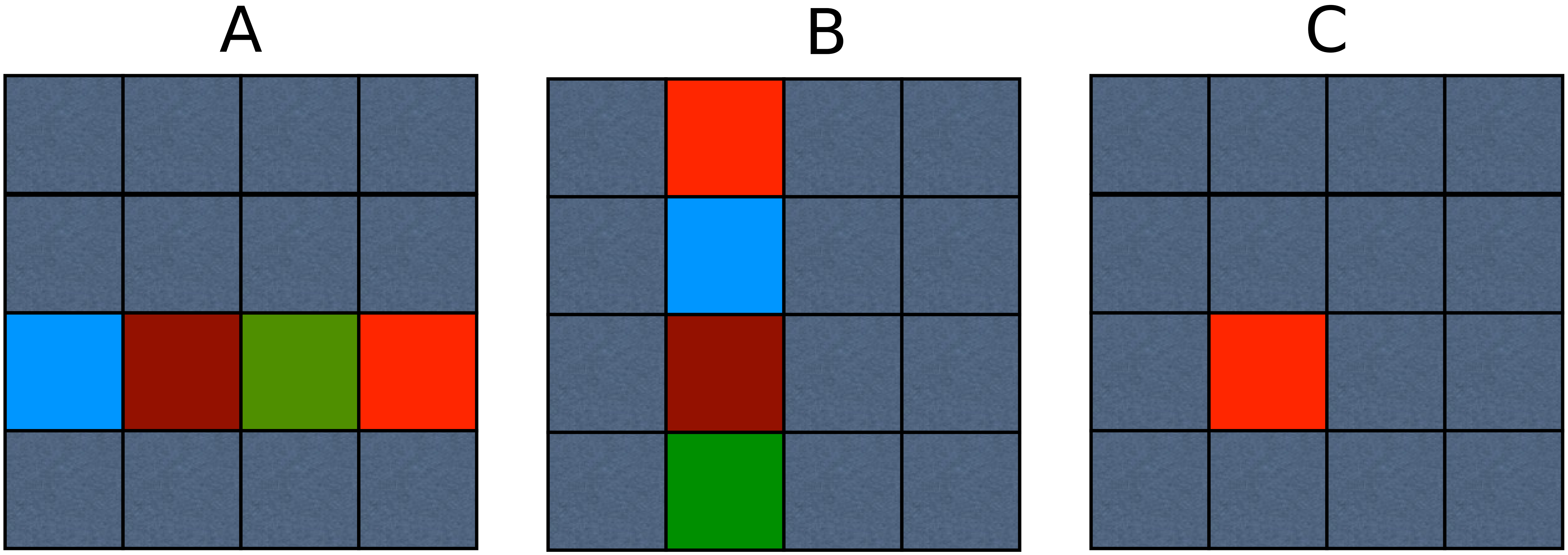
Blocks needed to compute a C element

We need to rearrange the data so that every block has useful work to do



The initial data configuration does not provide for this

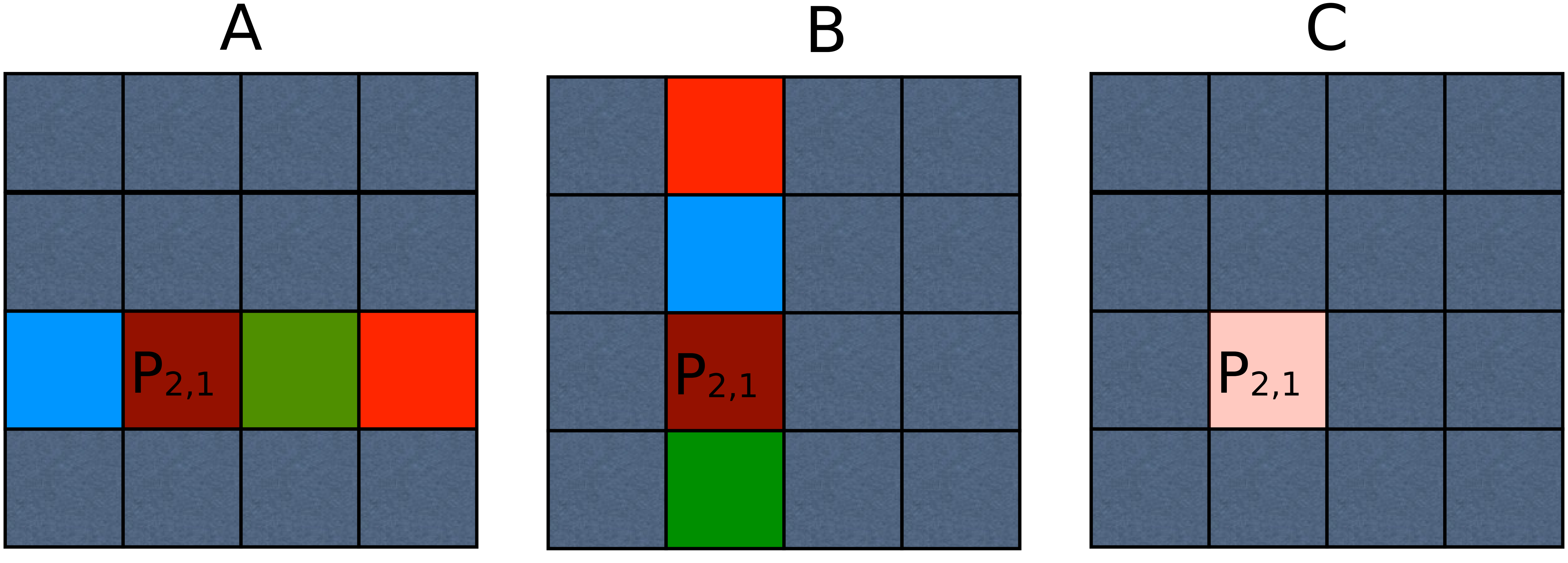
Every processor now has useful work to do



Note -- this only shows the full data layout for one processor

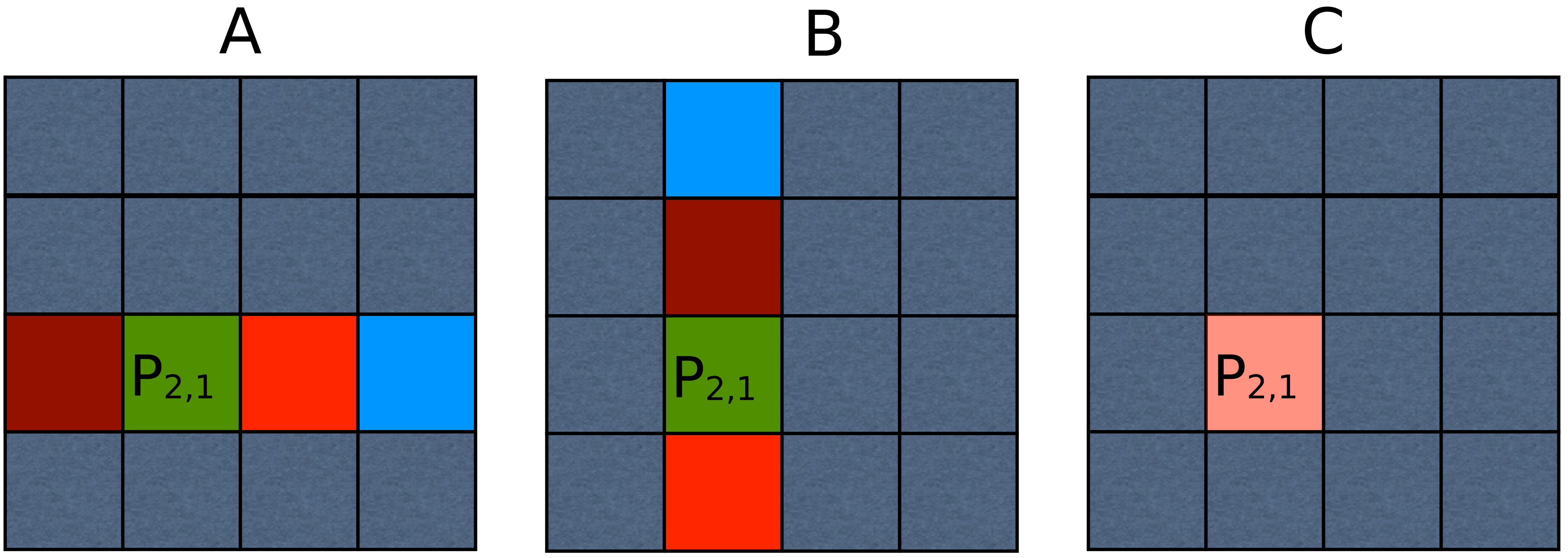
At each step in the multiplication, shift B elements up within their column, and A elements left within their row

First partial sum



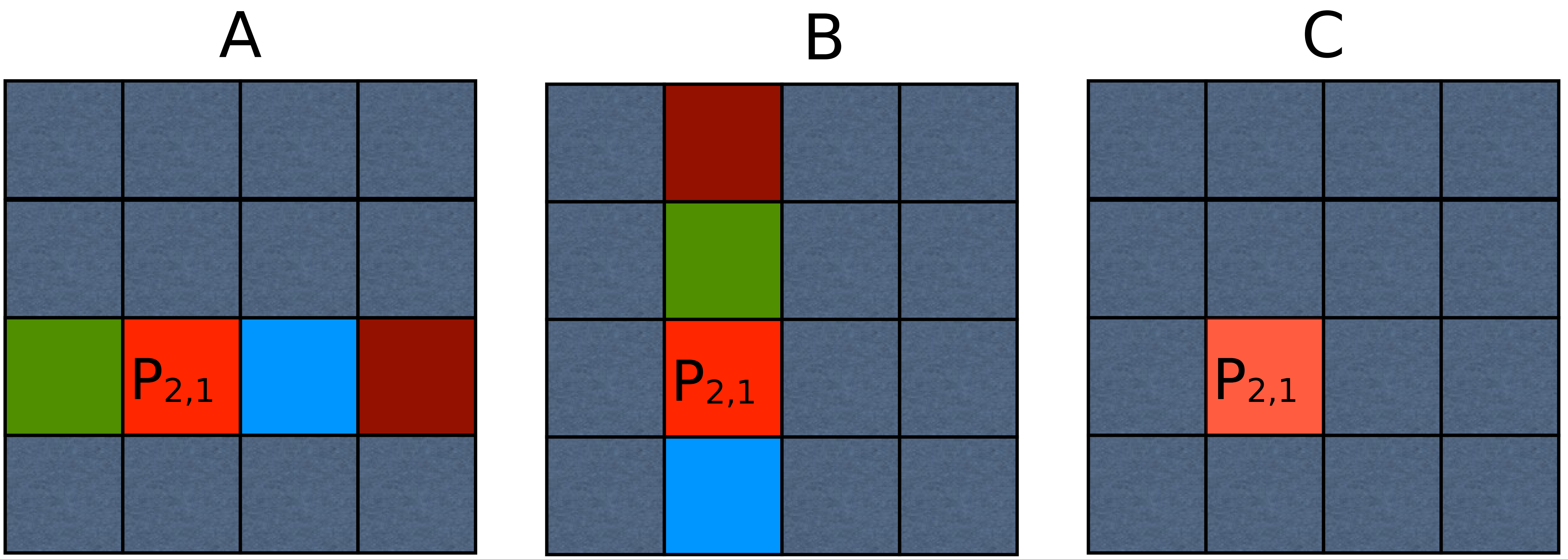
And again . . .

Second partial sum



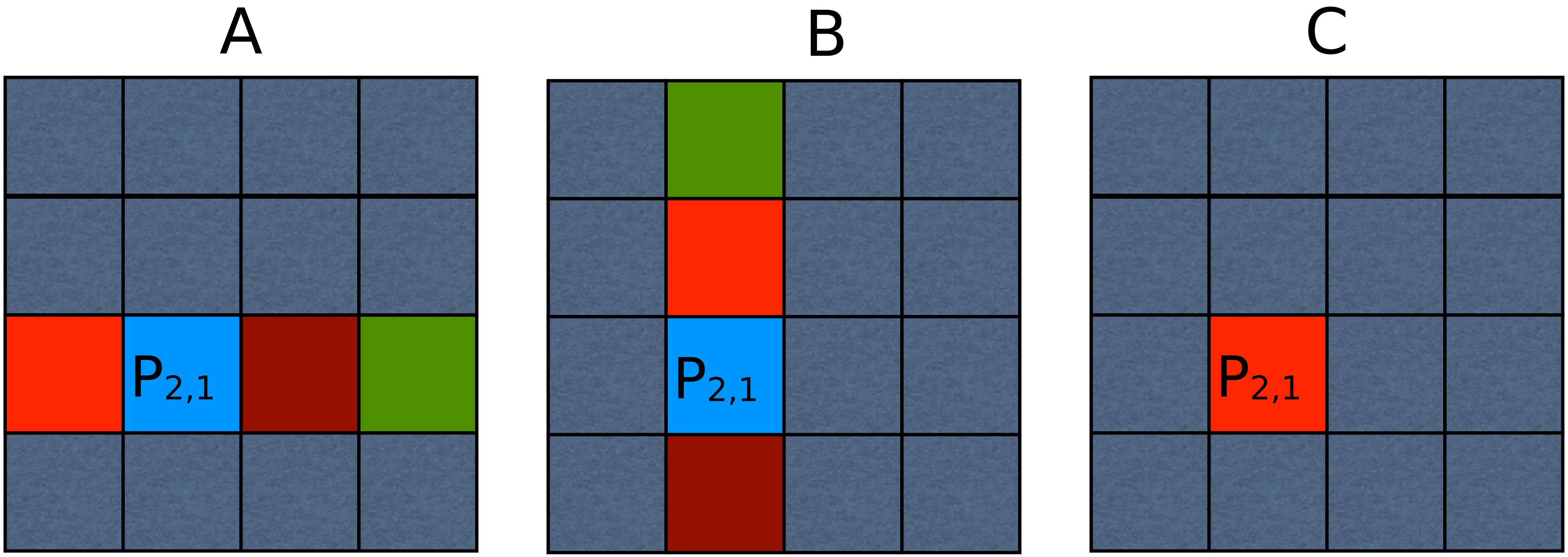
And again . . .

Third partial sum

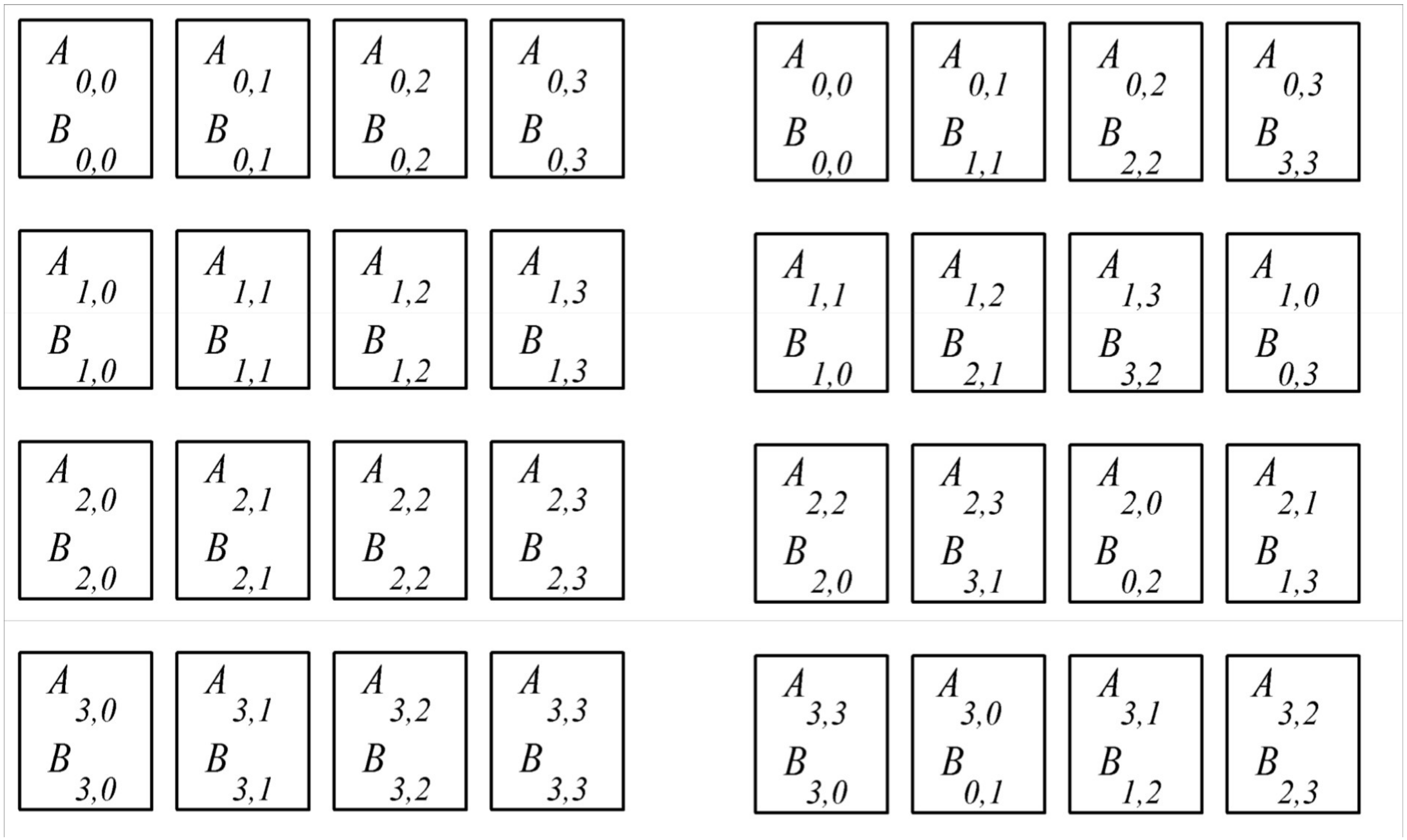


And again

Fourth partial sum



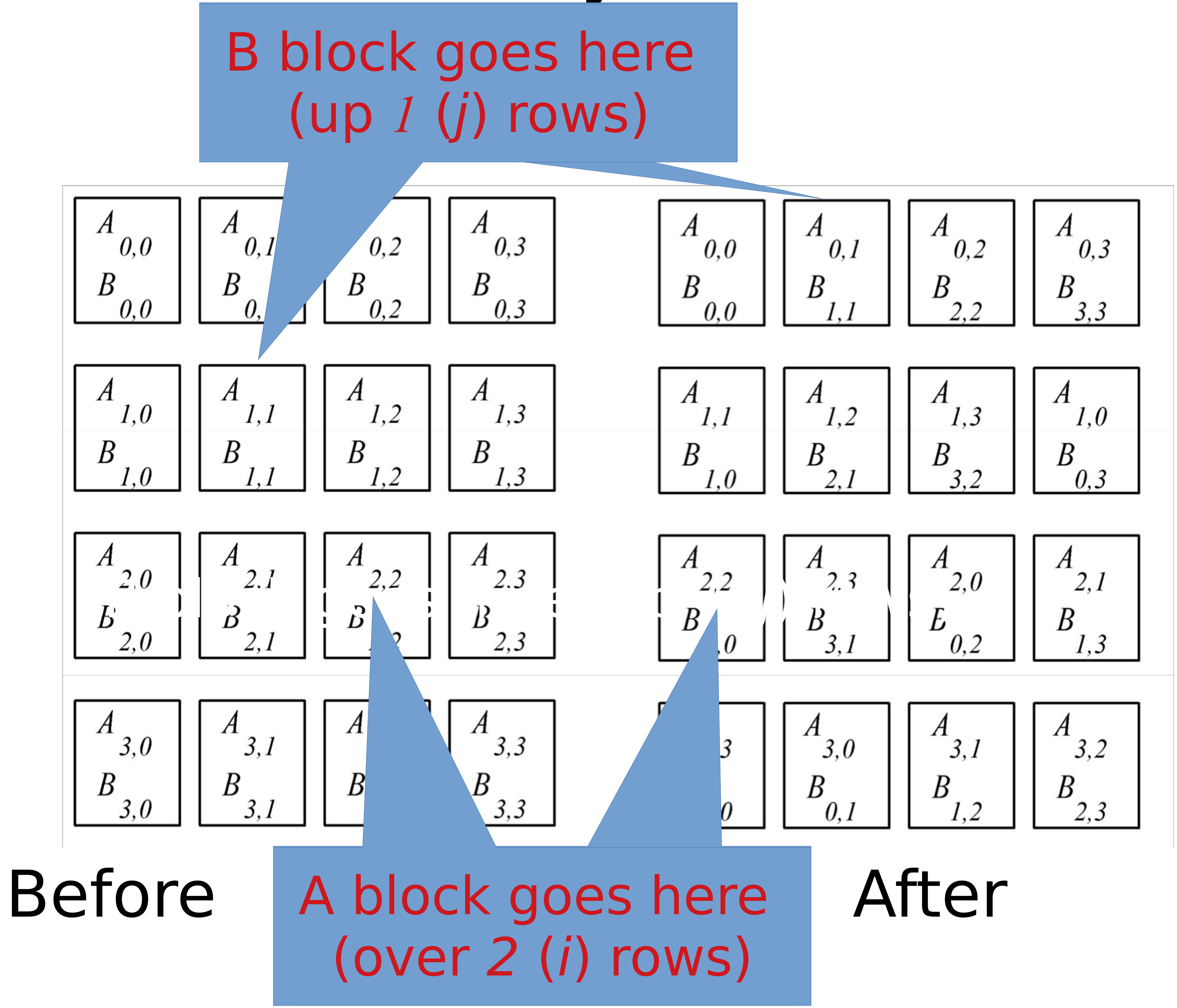
Another way to view this



Before

After

Another way to view this



Yet another way to view this

Each triangle represents a matrix block on a processor

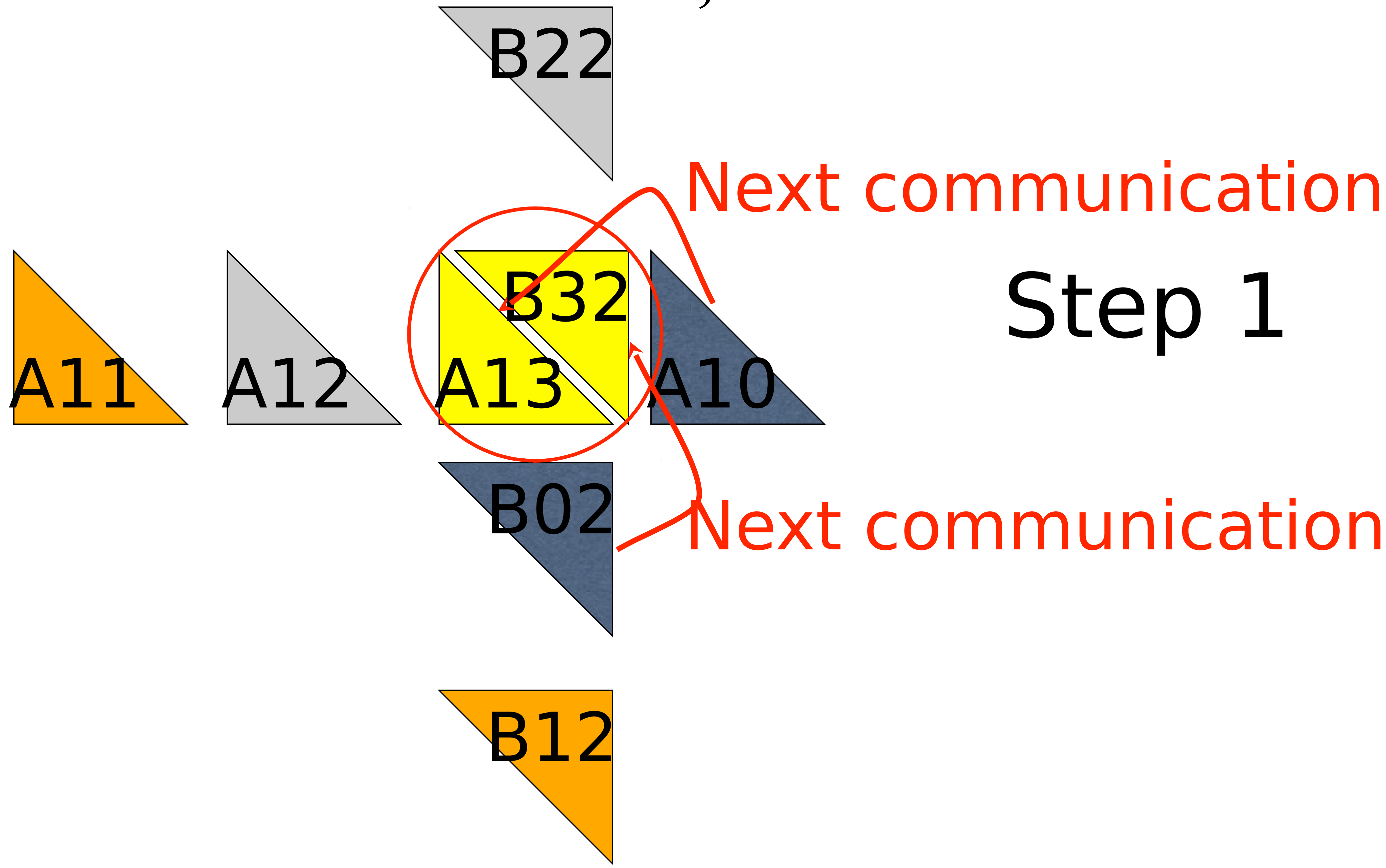
Only same-color triangles should be multiplied

Rearrange Blocks

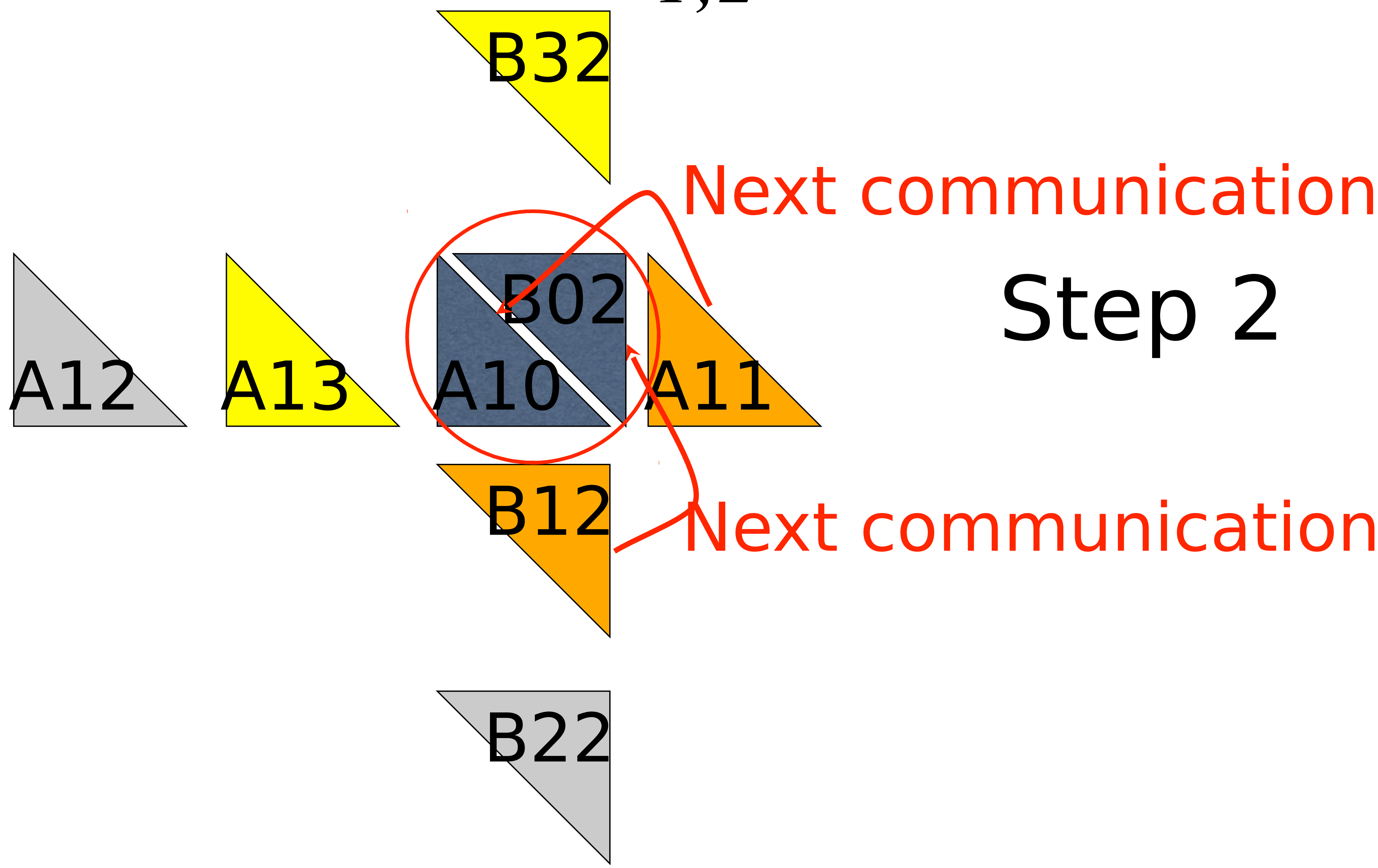
Block A*i*,*j* shifts left *i* positions

Block B*i*,*j* shifts up *j* positions

Consider Process $P_{1,2}$



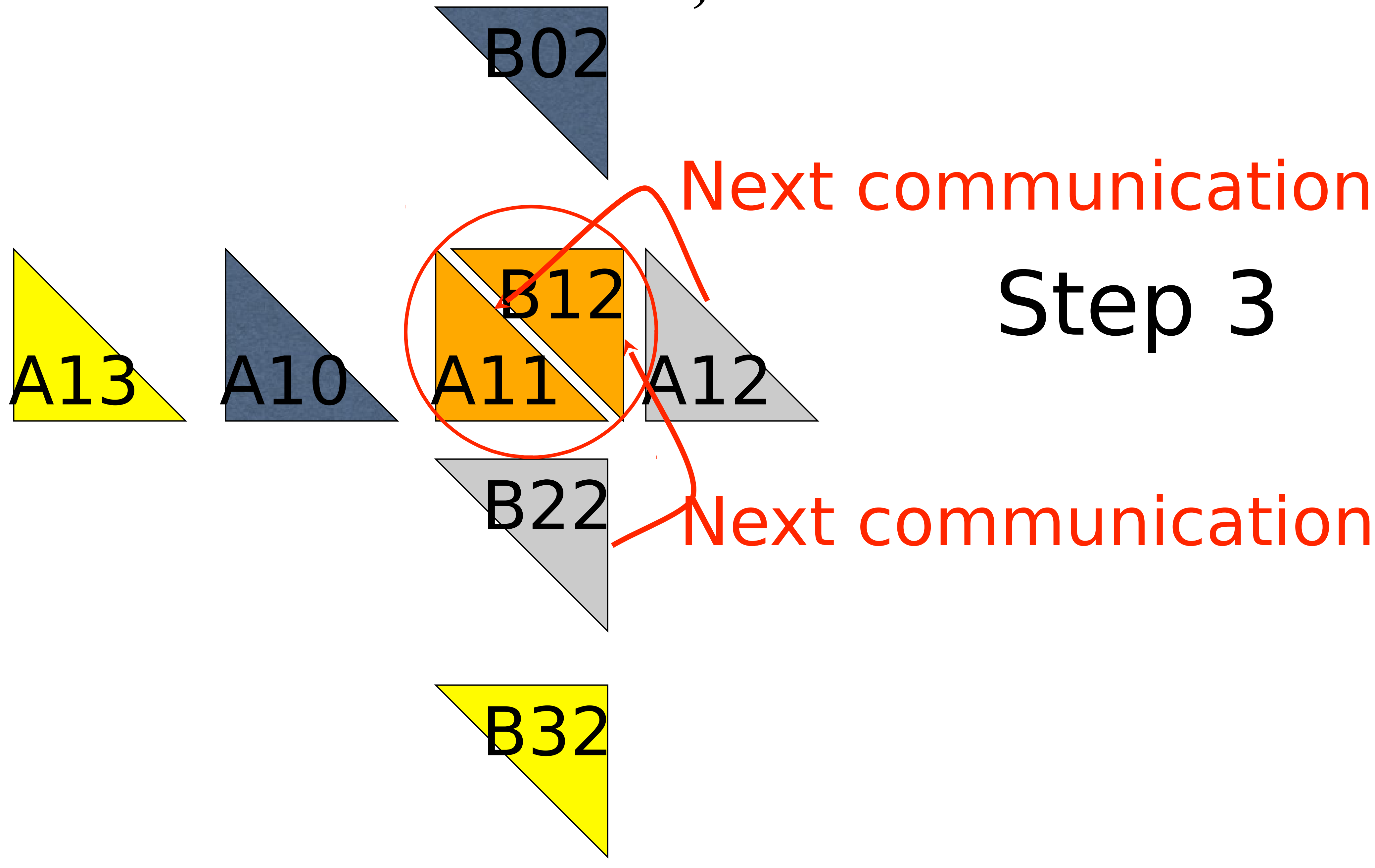
Consider Process $P_{1,2}$



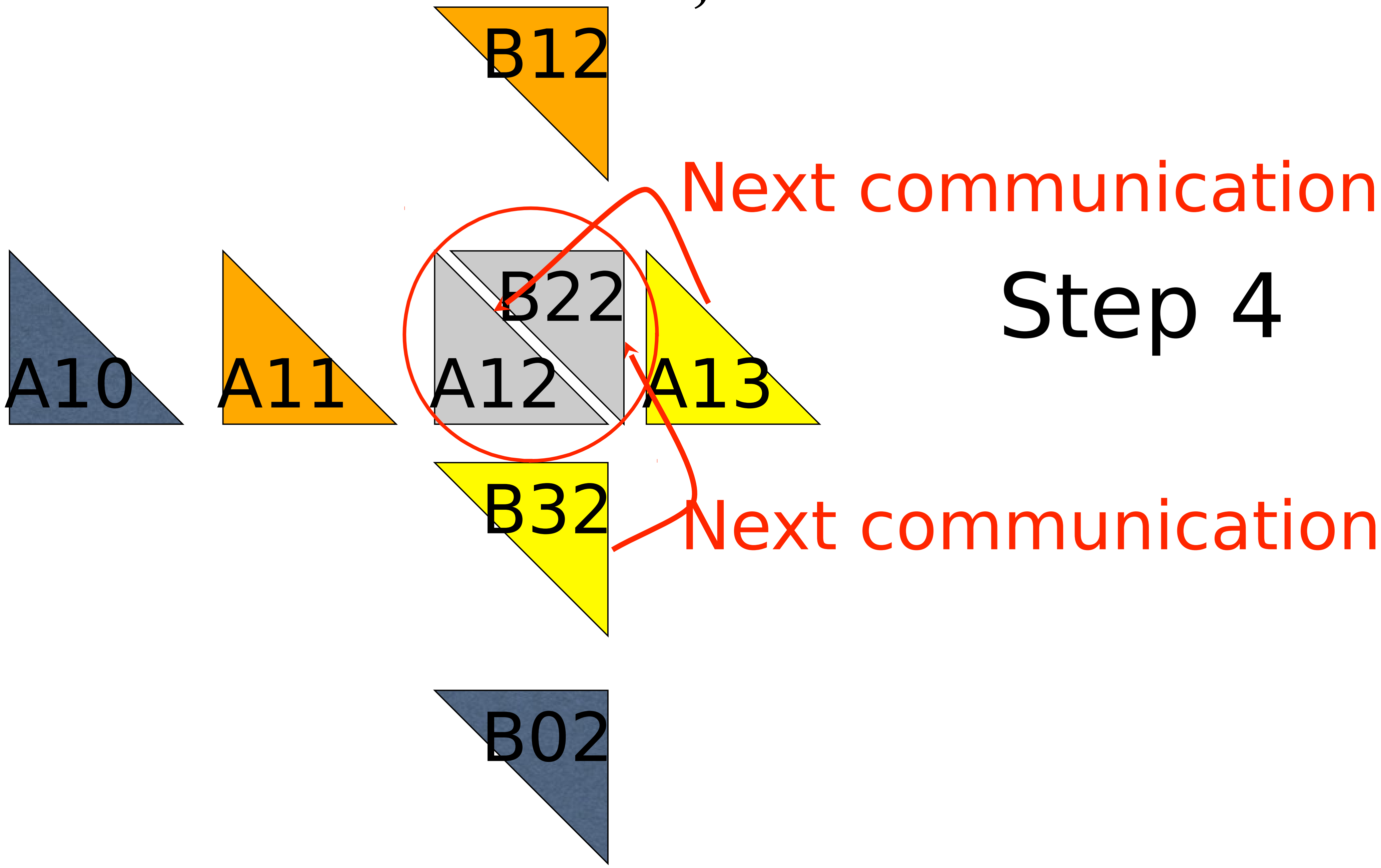
Step 2

Consider Process

$$P_{1,2}$$



Consider Process $P_{1,2}$



Complexity Analysis

- Algorithm has $\sqrt{p}$ iterations
  - During each iteration process multiplies two $(n / \sqrt{p}) \times (n / \sqrt{p})$ matrices: $\Theta(n / \sqrt{p})^3$ or $\Theta(n^3 / p^{3/2})$
- Overall computational complexity: $\sqrt{p}\, n^3/p^{3/2}$ or $\Theta(n^3 / p)$
  - During each $\sqrt{p}$ iterations a process sends and receives two blocks of size $(n / \sqrt{p}) \times (n / \sqrt{p})$
- Overall communication complexity: $\Theta(n^2/ \sqrt{p})$