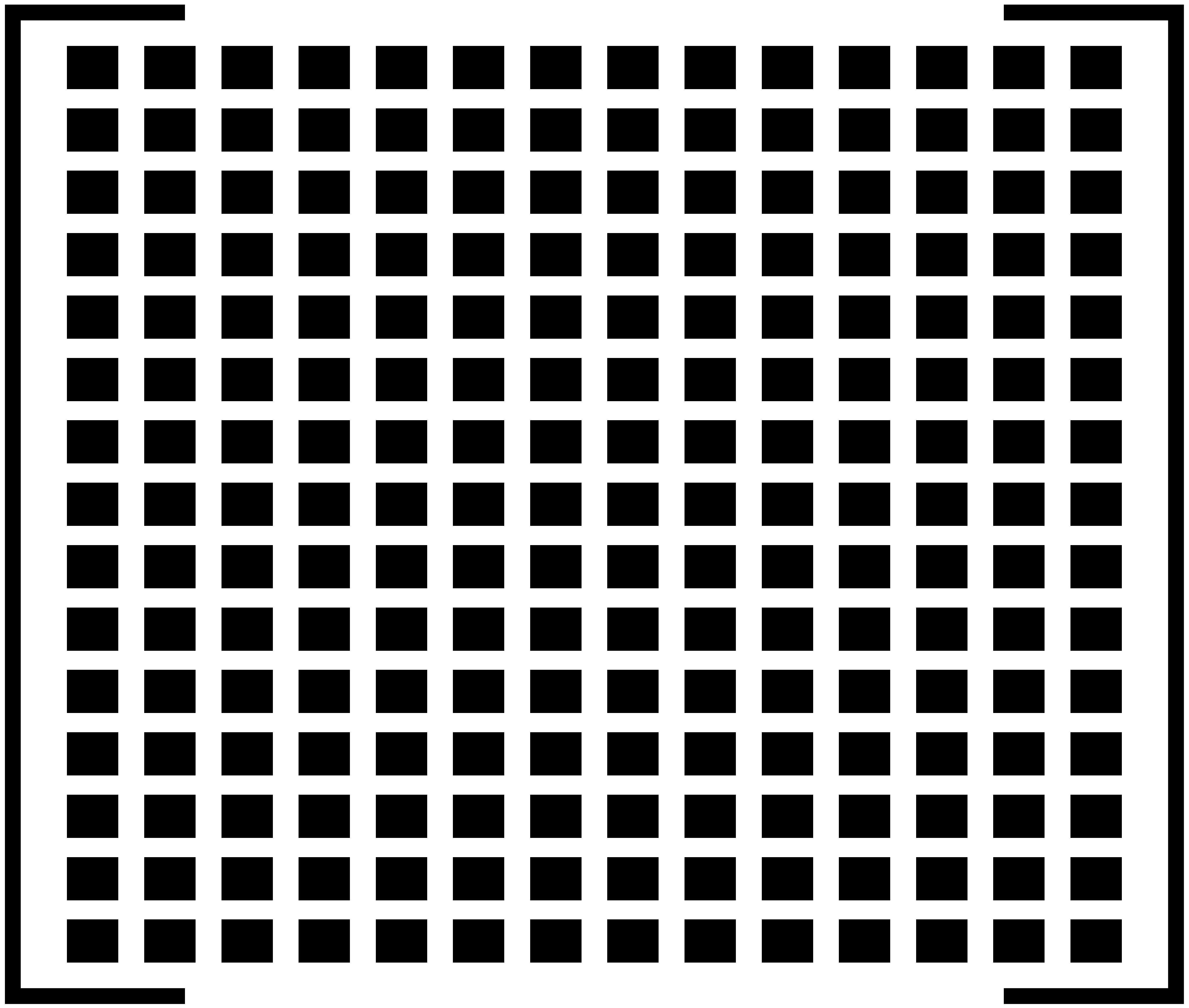
Introduction to Parallel Processing

Lecture 5: MPI Matrix-Matrix Multiplication

Professor Amanda Bienz

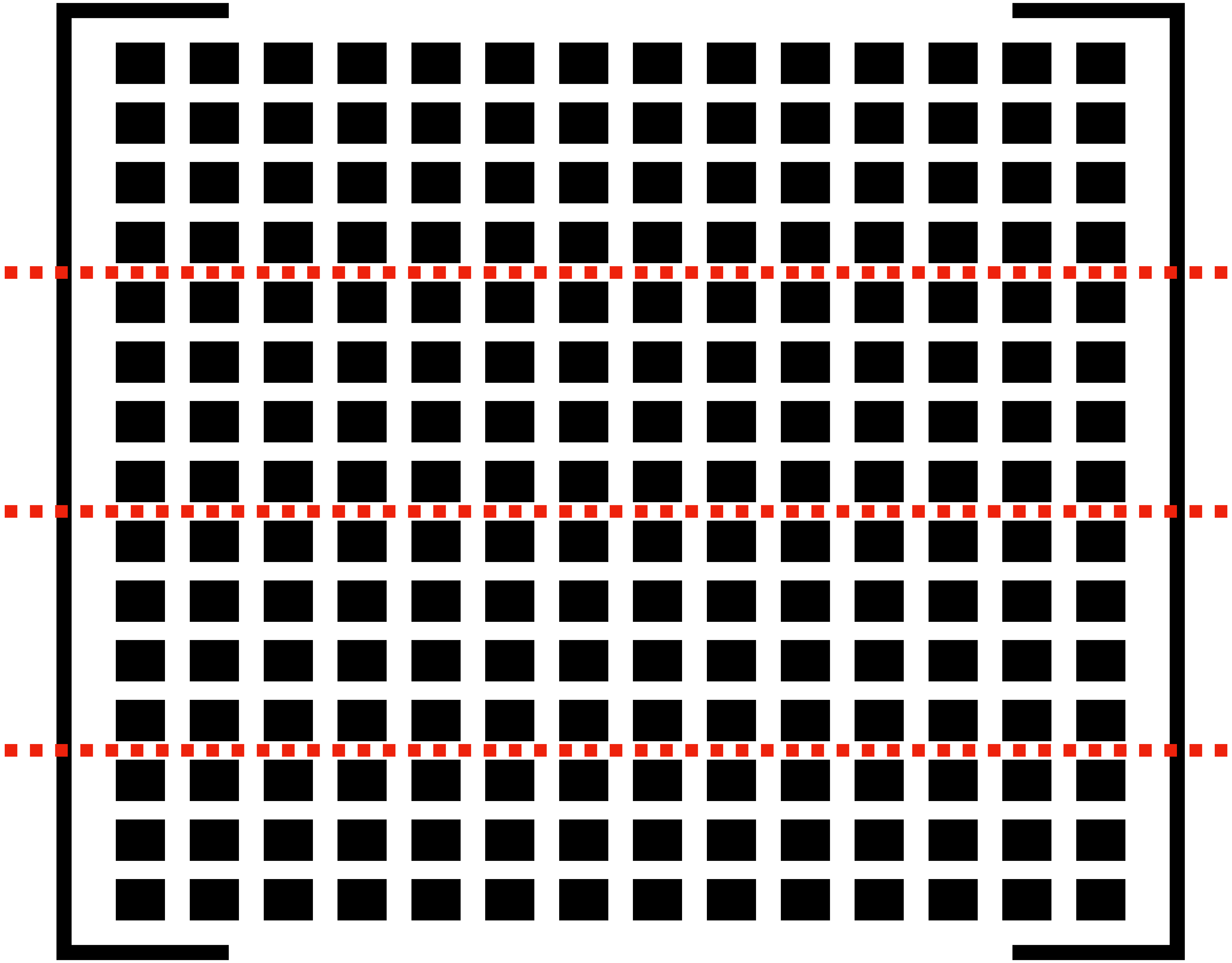
Parallel Matrix Partition

- How do we partition this matrix, so that a part of the matrix is on each process?



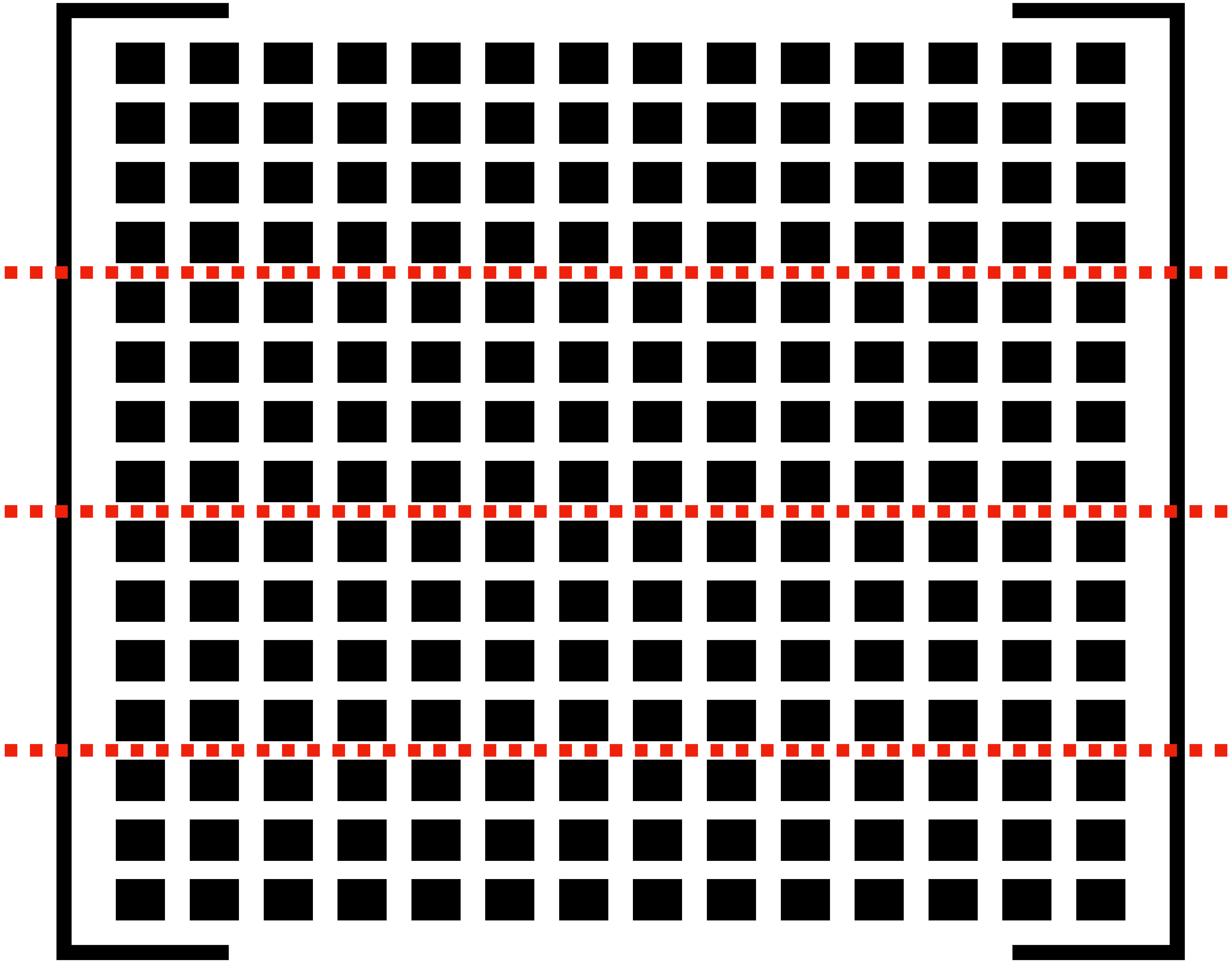
Parallel Matrix Partition

- How do we partition this matrix, so that a part of the matrix is on each process?
- **Row Wise Partition : Each process holds contiguous chunk of rows of the matrix**



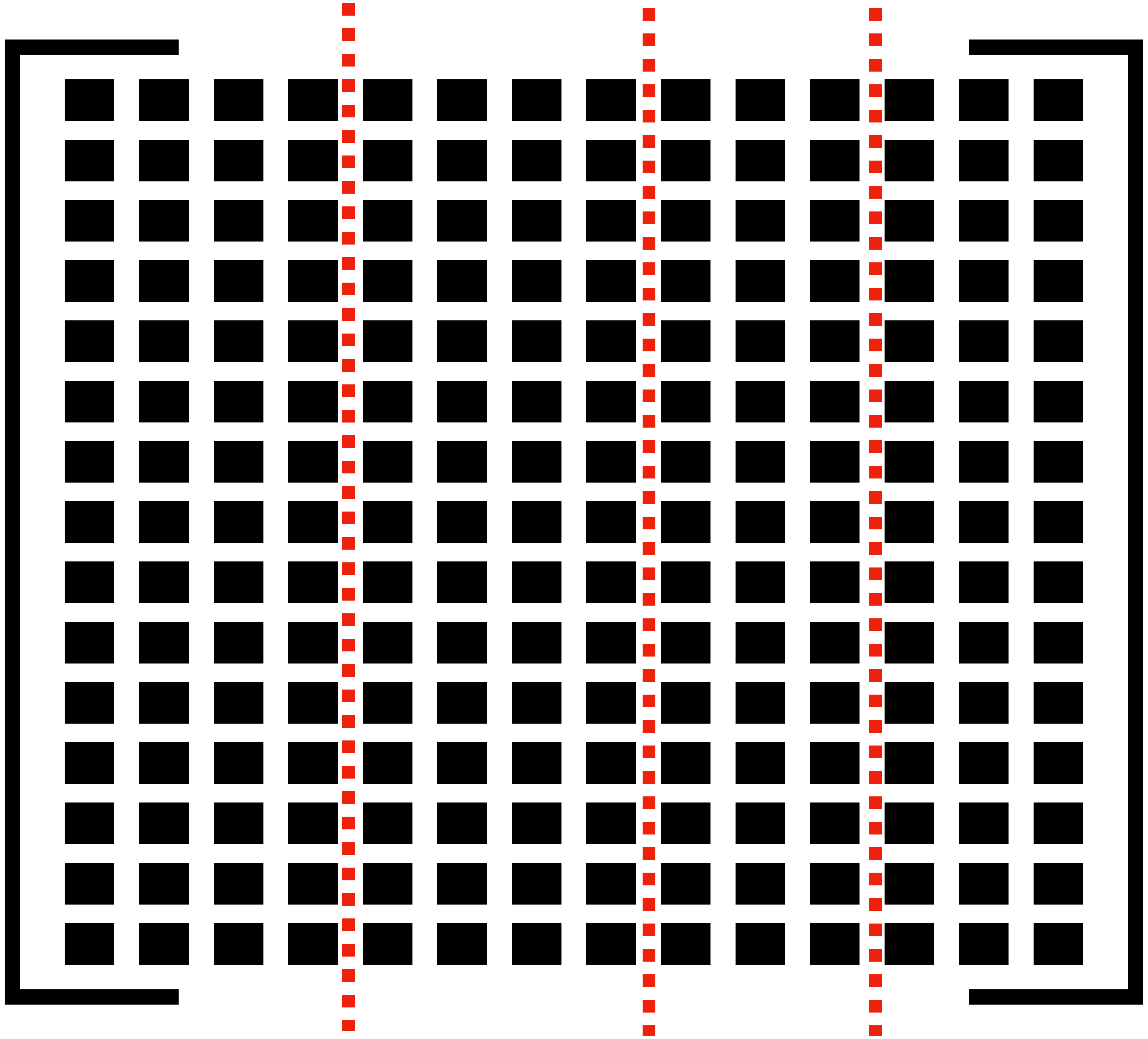
Parallel Matrix Partition

- How do we partition this matrix, so that a part of the matrix is on each process?
- **Problem : What is the smallest partition that we can make?**
- **For an nxn matrix, smallest row-wise partition holds n values**



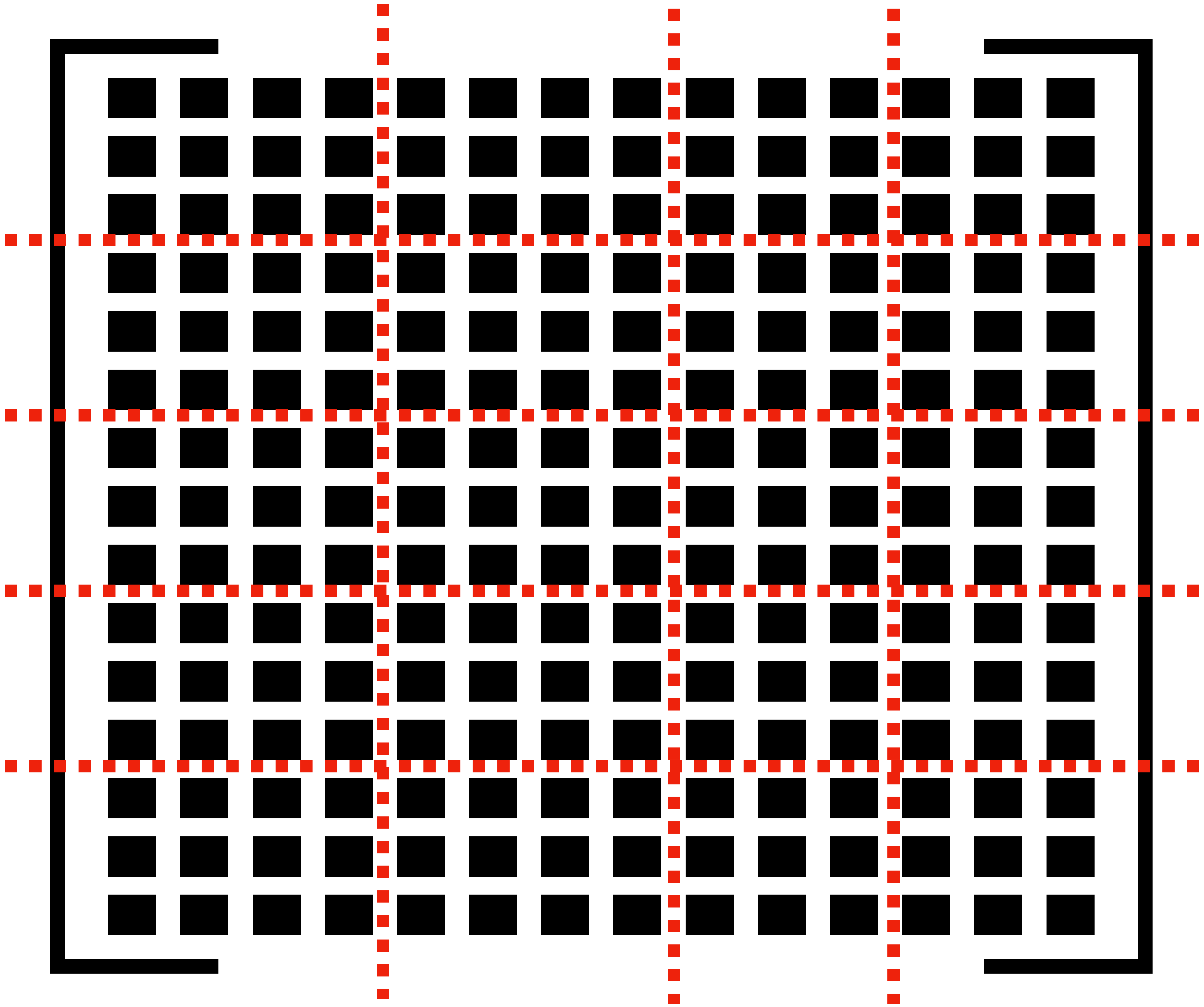
Parallel Matrix Partition

- How do we partition this matrix, so that a part of the matrix is on each process?
- **Similarly, could look at column-wise partition, but we run into the same issue**



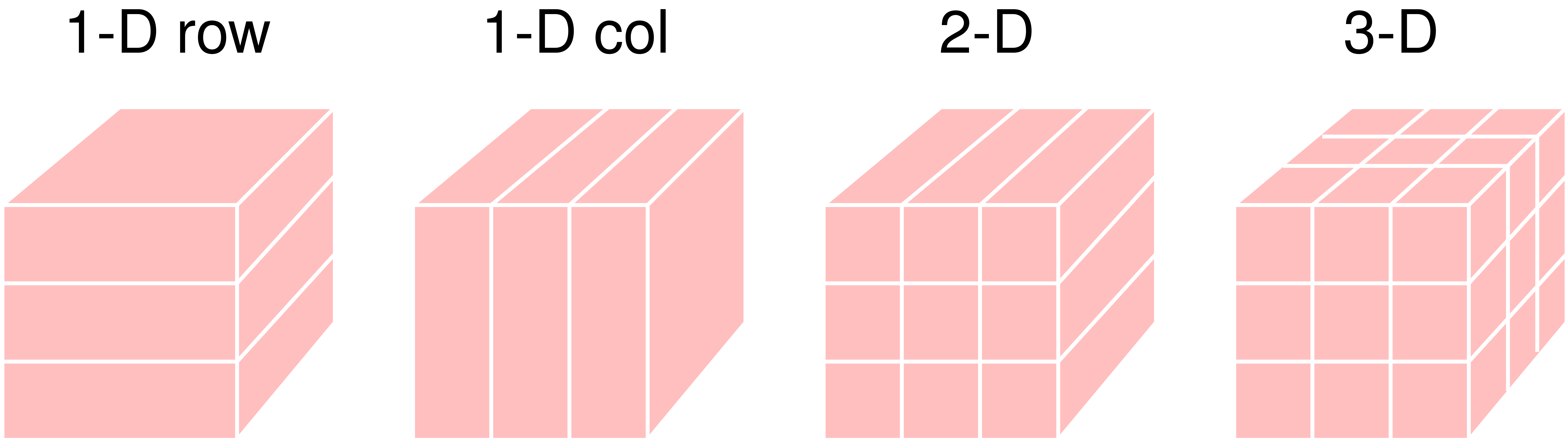
Parallel Matrix Partition

- How do we partition this matrix, so that a part of the matrix is on each process?
- **Solution : 2-dimensional partitions**

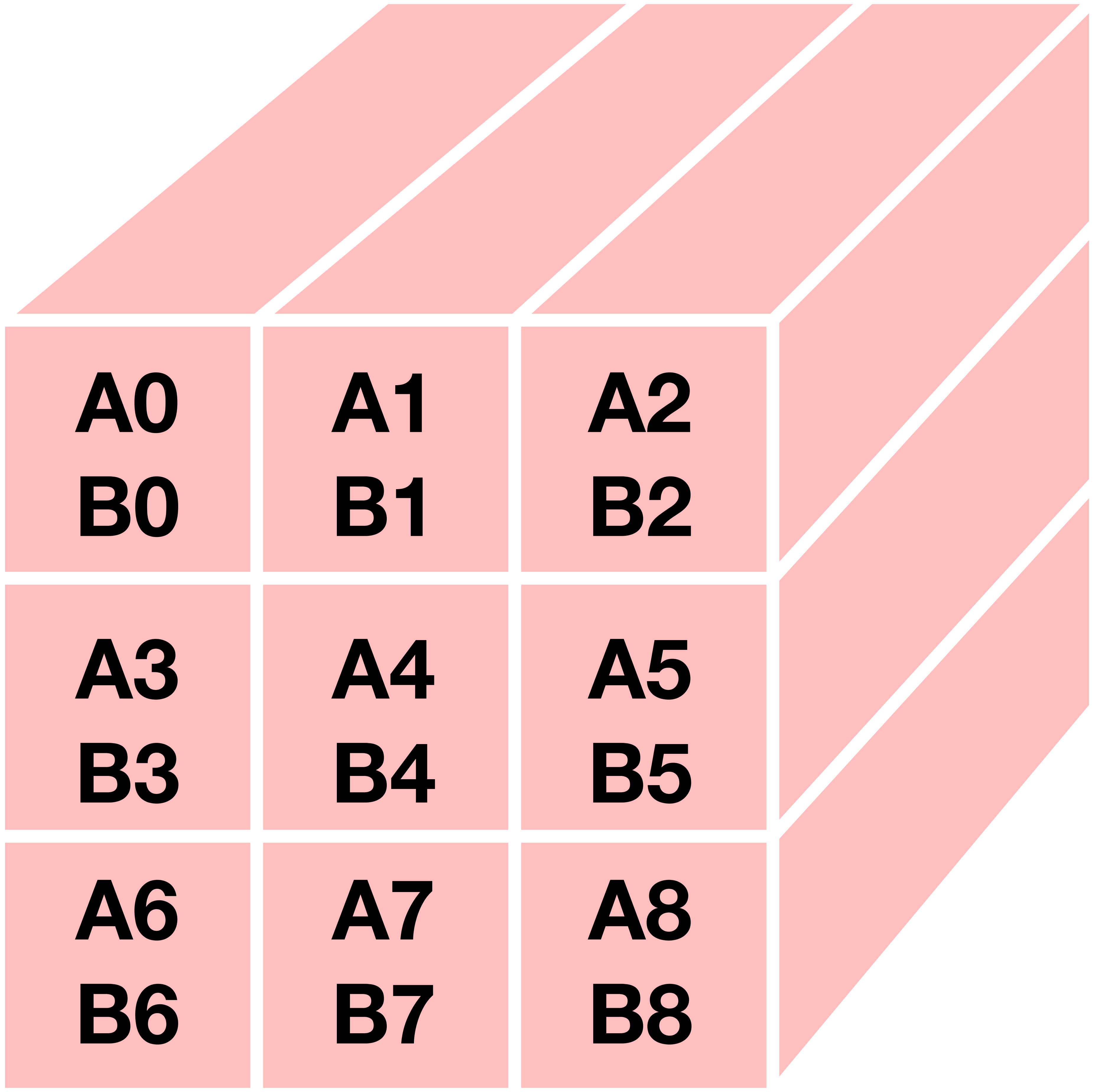


Parallel Matrix Partition

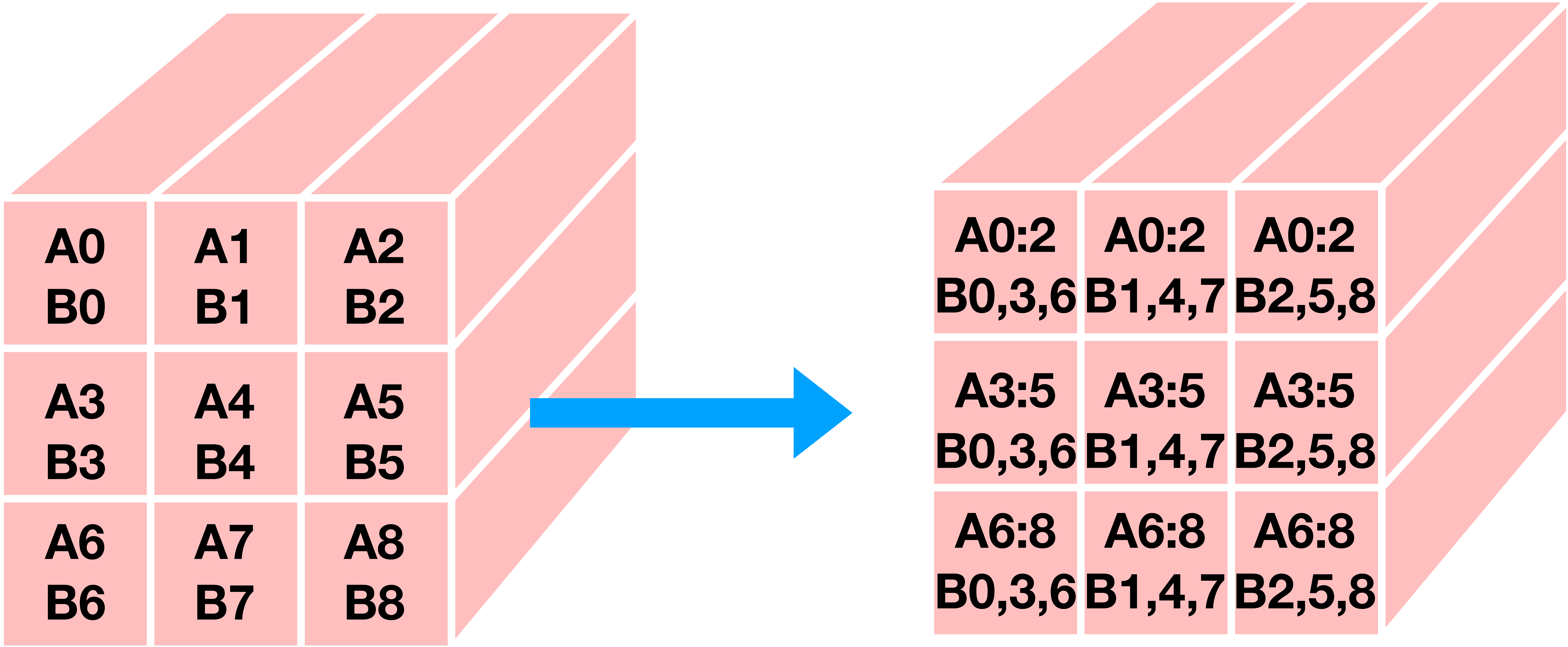
- Could have a 3D partition
- Partition by operations rather than elements of matrix
- Sometimes a good idea for GPUs, but we typically stick with 2D in HPC due to communication overhead



Matrix Multiplication



Simplest Implementation



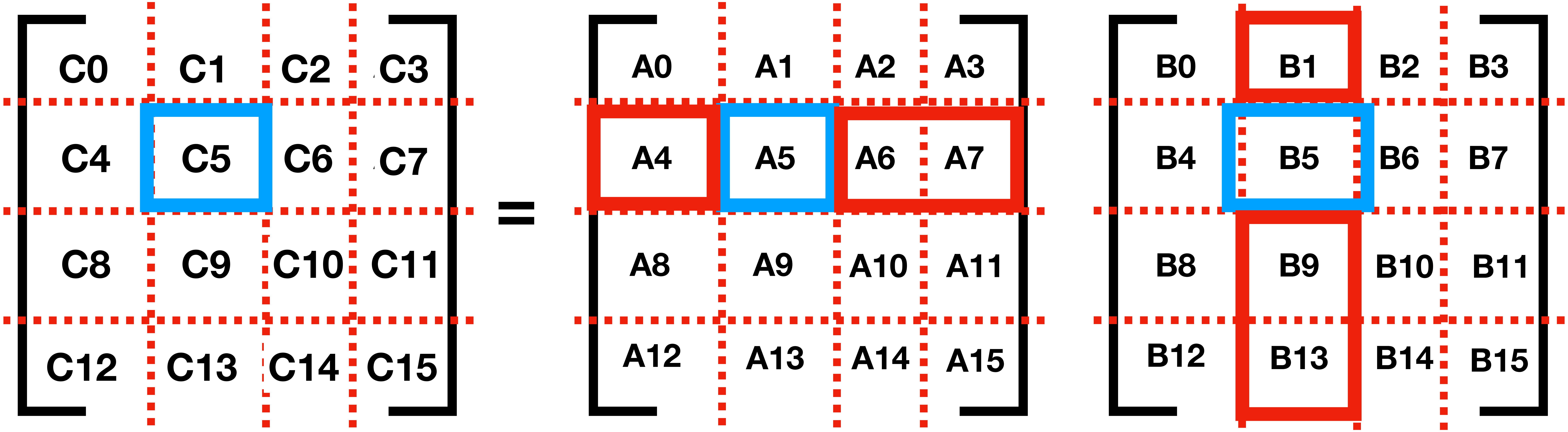
Simplest Implementation

- Every process sends entire rows of A and columns of B to every process that needs them

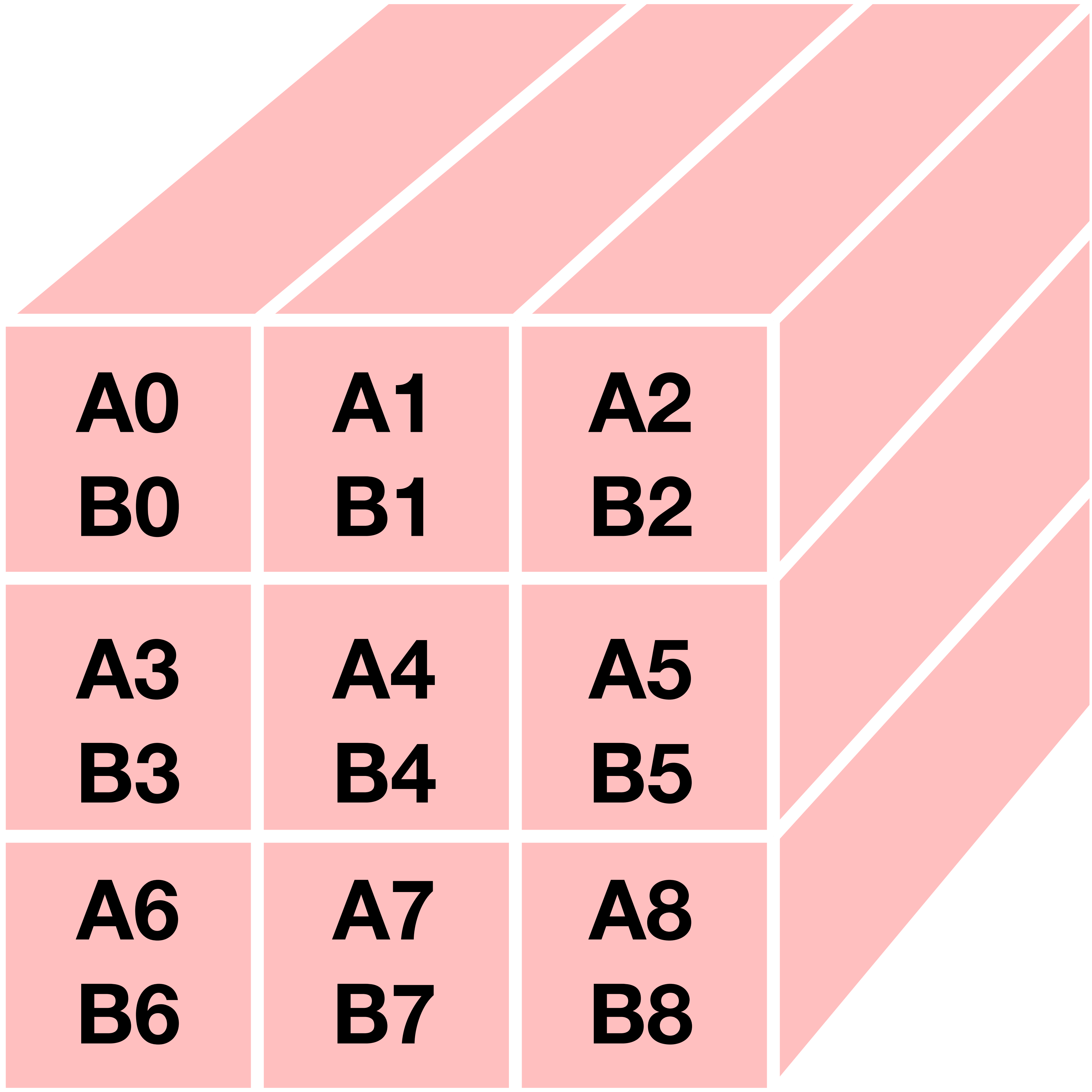
$$\begin{bmatrix} C0 & C1 & C2 & C3 \\ C4 & \boxed{C5} & C6 & C7 \\ C8 & C9 & C10 & C11 \\ C12 & C13 & C14 & C15 \end{bmatrix} = \begin{bmatrix} A0 & A1 & A2 & A3 \\ A4 & \boxed{A5} & A6 & A7 \\ A8 & A9 & A10 & A11 \\ A12 & A13 & A14 & A15 \end{bmatrix} \begin{bmatrix} B0 & B1 & B2 & B3 \\ B4 & \boxed{B5} & B6 & B7 \\ B8 & B9 & B10 & B11 \\ B12 & B13 & B14 & B15 \end{bmatrix}$$

Simplest Implementation

- Every process sends entire rows of A and columns of B to every process that needs them

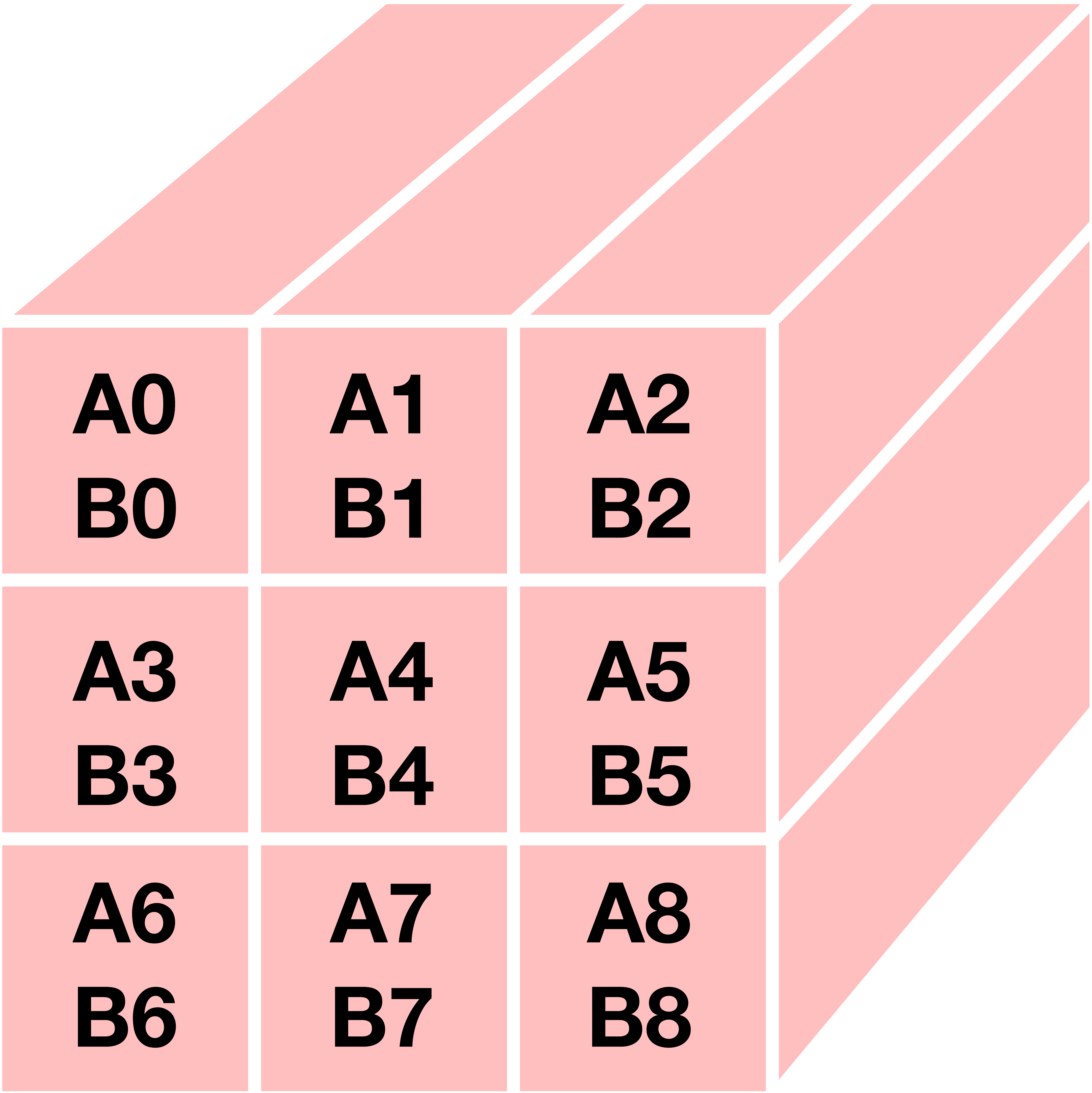


Fox's Algorithm



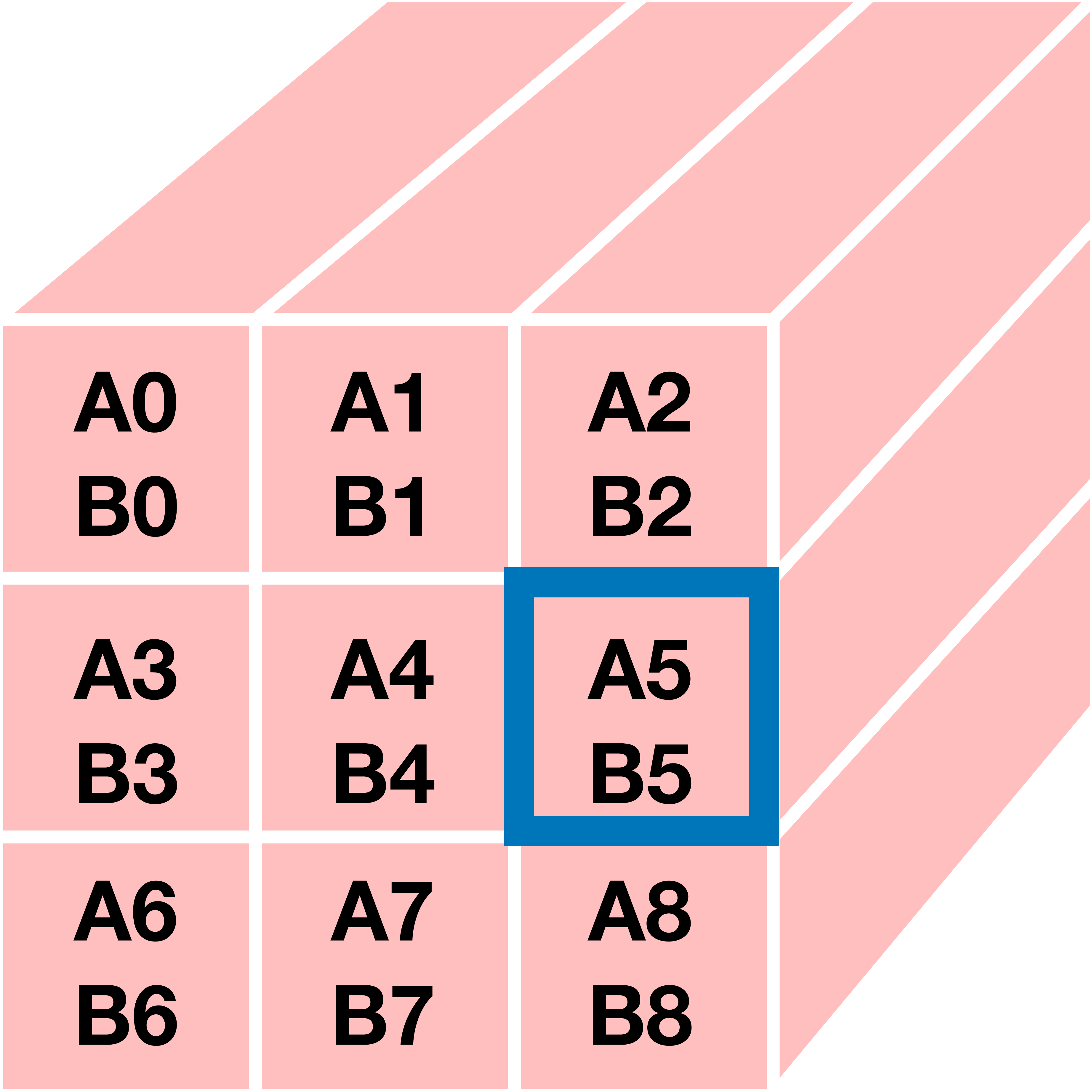
- The processes are laid out in a grid, so each process p has a row I and column J in the process grid
- Gather all of A among all processes in each process row I

Fox's Algorithm



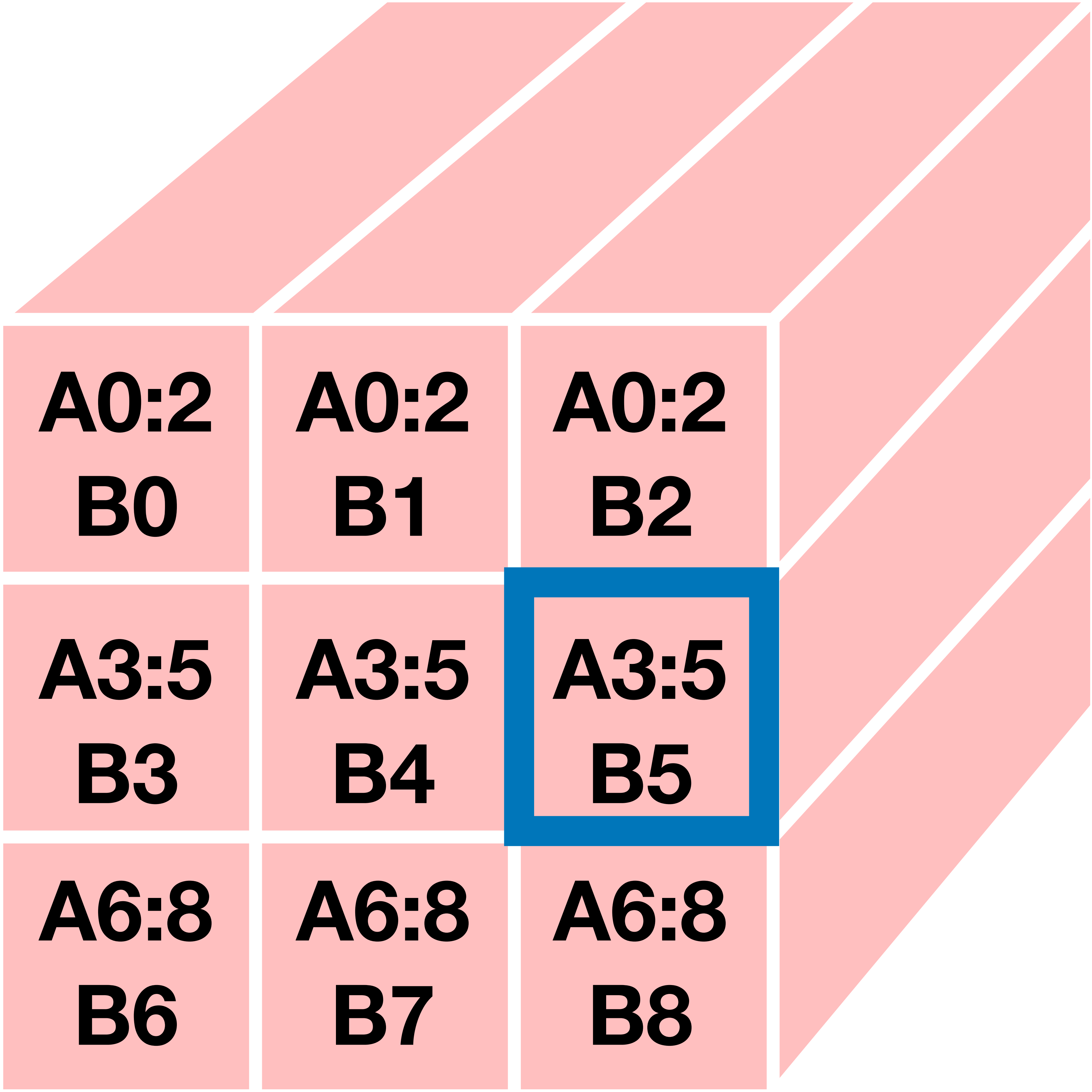
- Multiple A by local portion of B, then rotate B along process column J
  - Process in row I, column J sends local portion of B to process in row I+1, column J
  - Process in row I, column J receives new portion of B from process in row I-1, column J

Fox's Algorithm



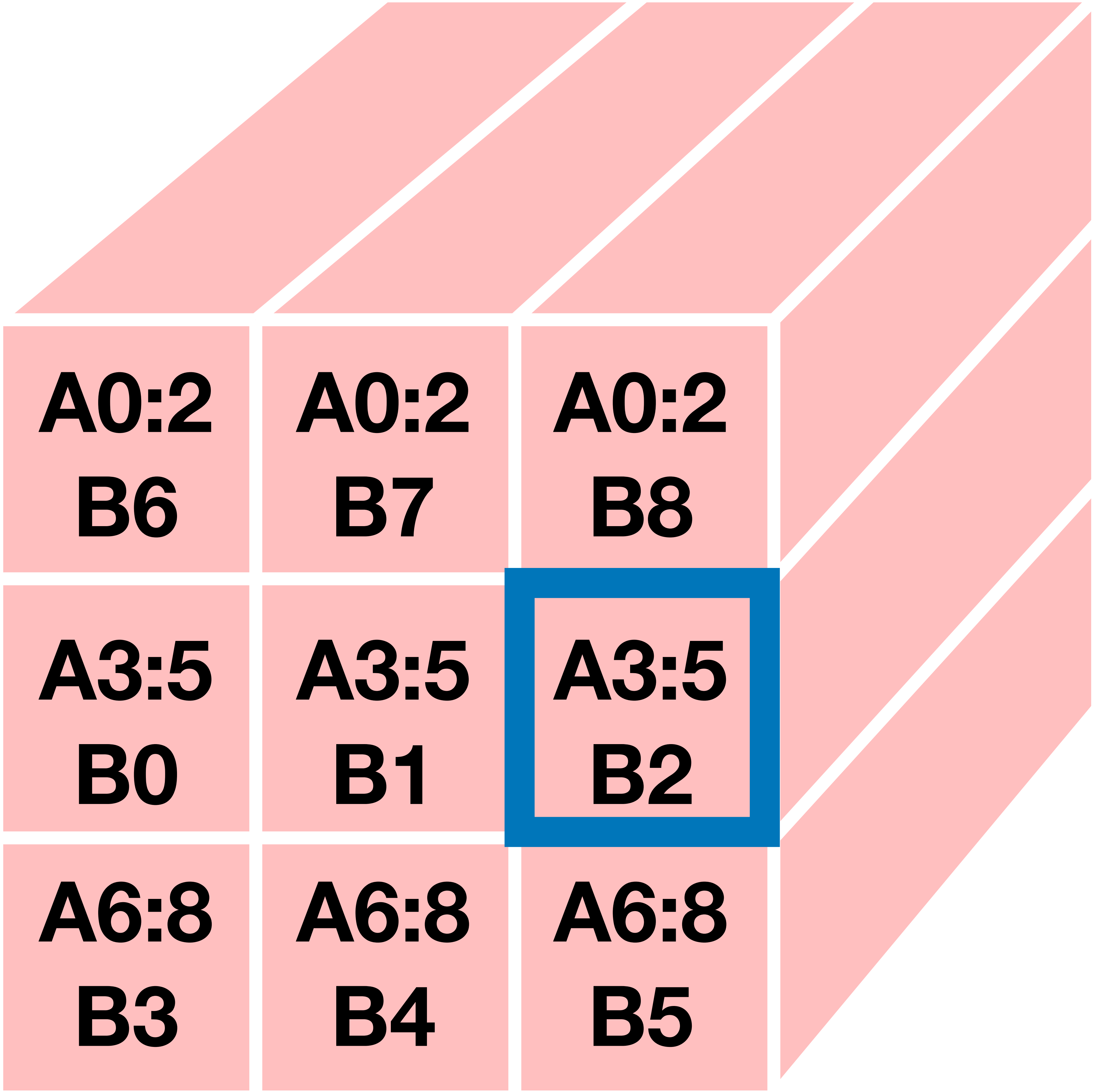
- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

Fox's Algorithm



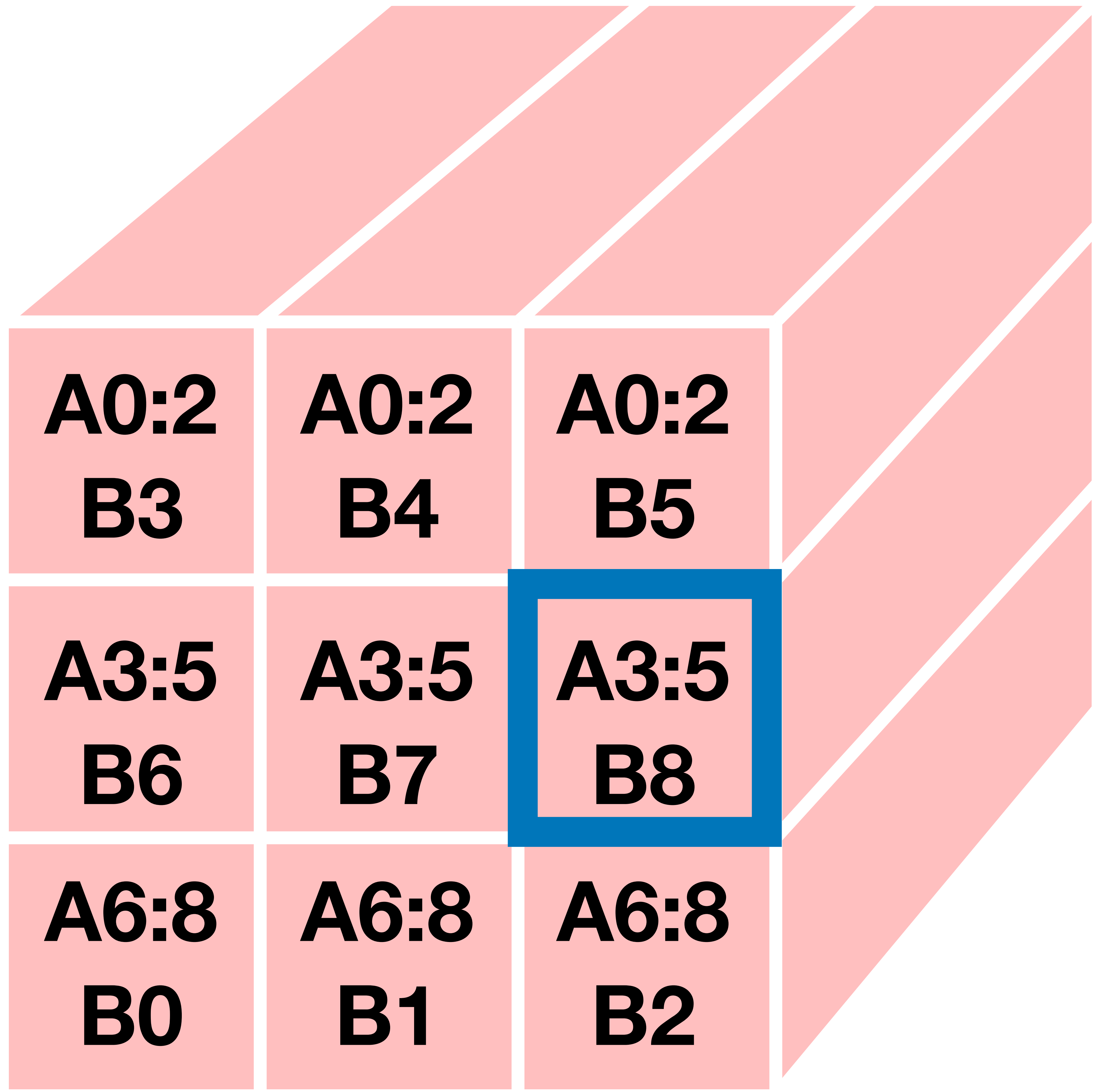
- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

Fox's Algorithm



- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

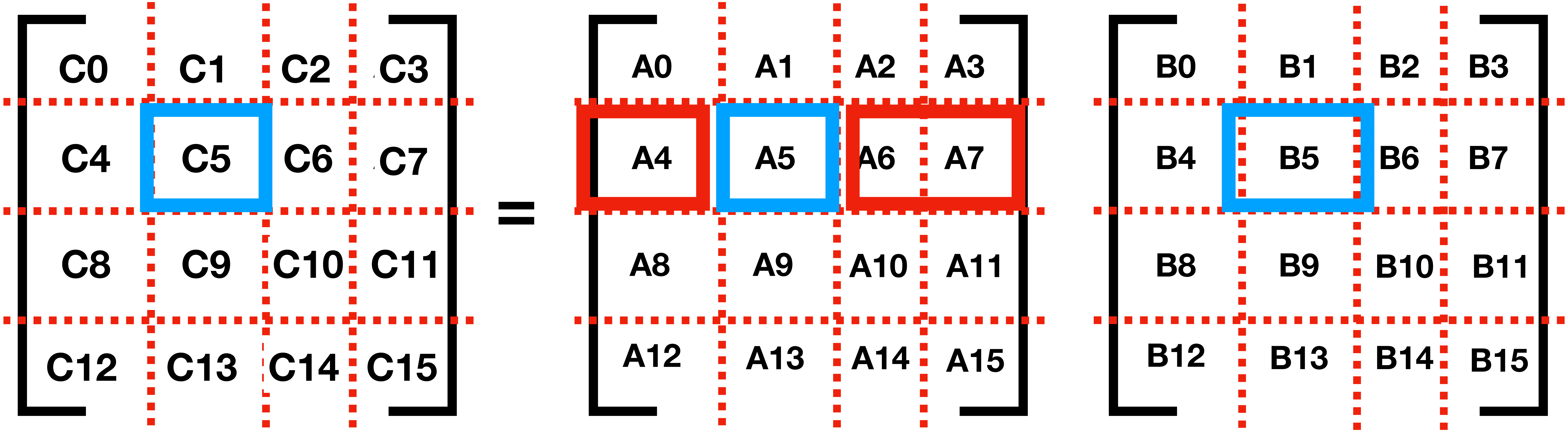
Fox's Algorithm



- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

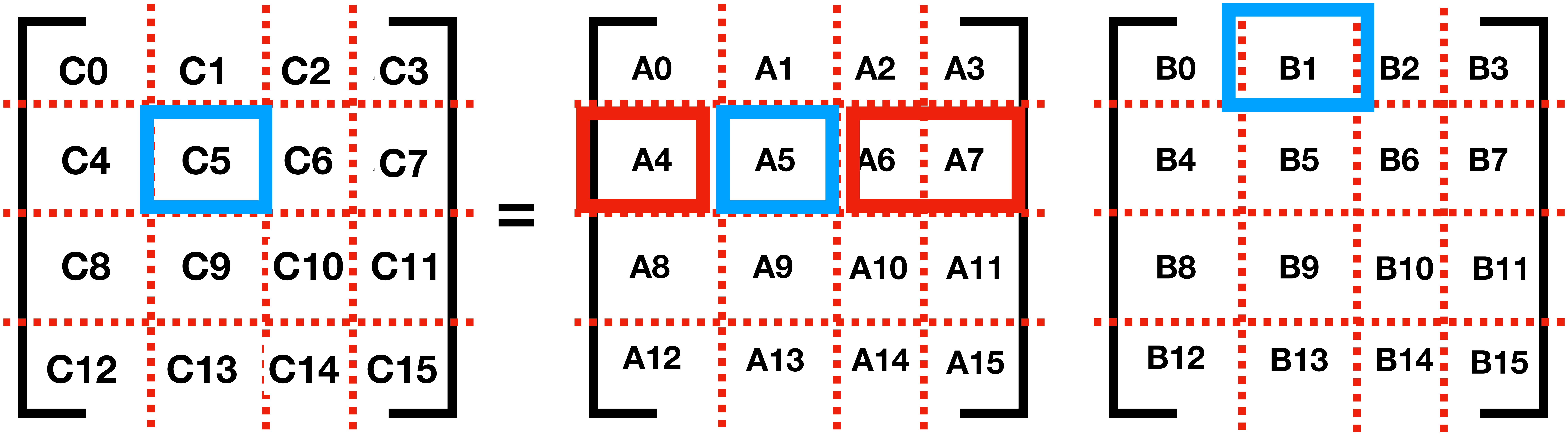
Fox's Algorithm

- Hold all needed values of A, but only one partition of B at a time



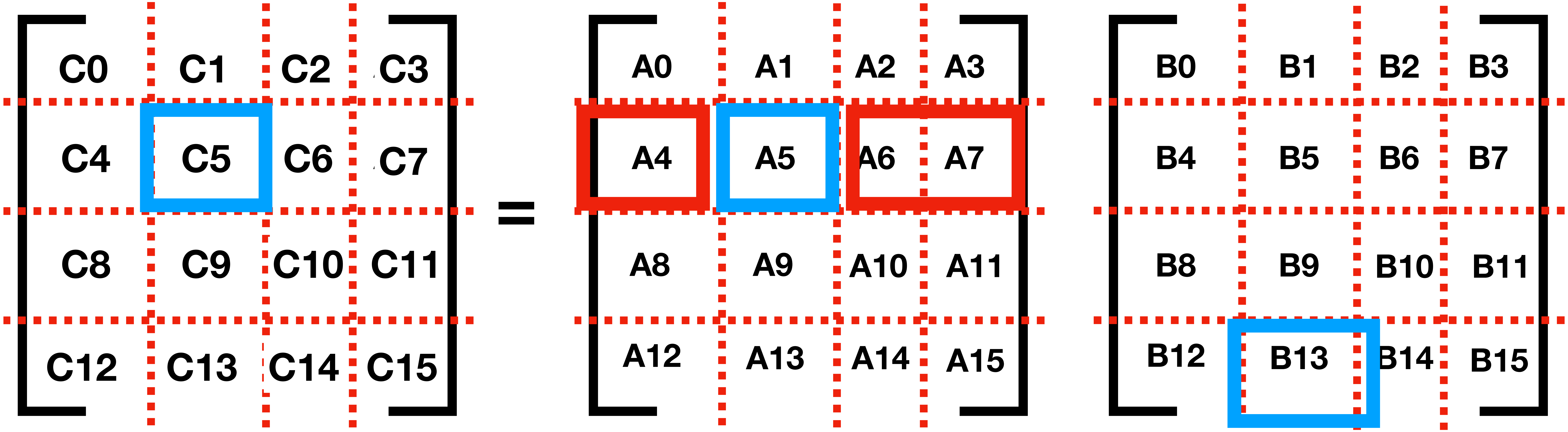
Fox's Algorithm

- Hold all needed values of A, but only one partition of B at a time



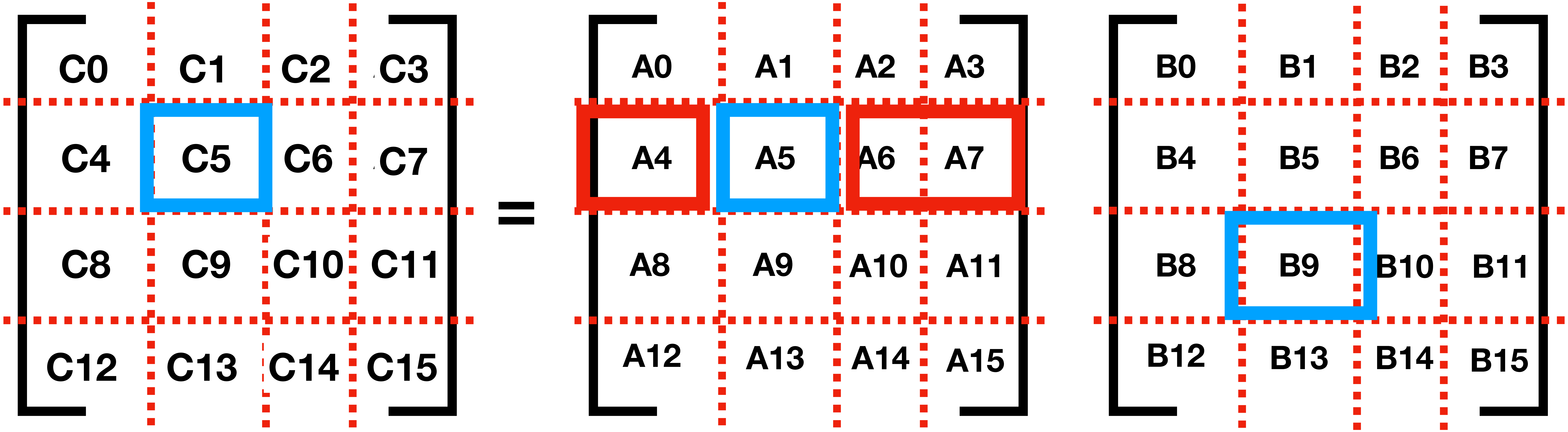
Fox's Algorithm

- Hold all needed values of A, but only one partition of B at a time

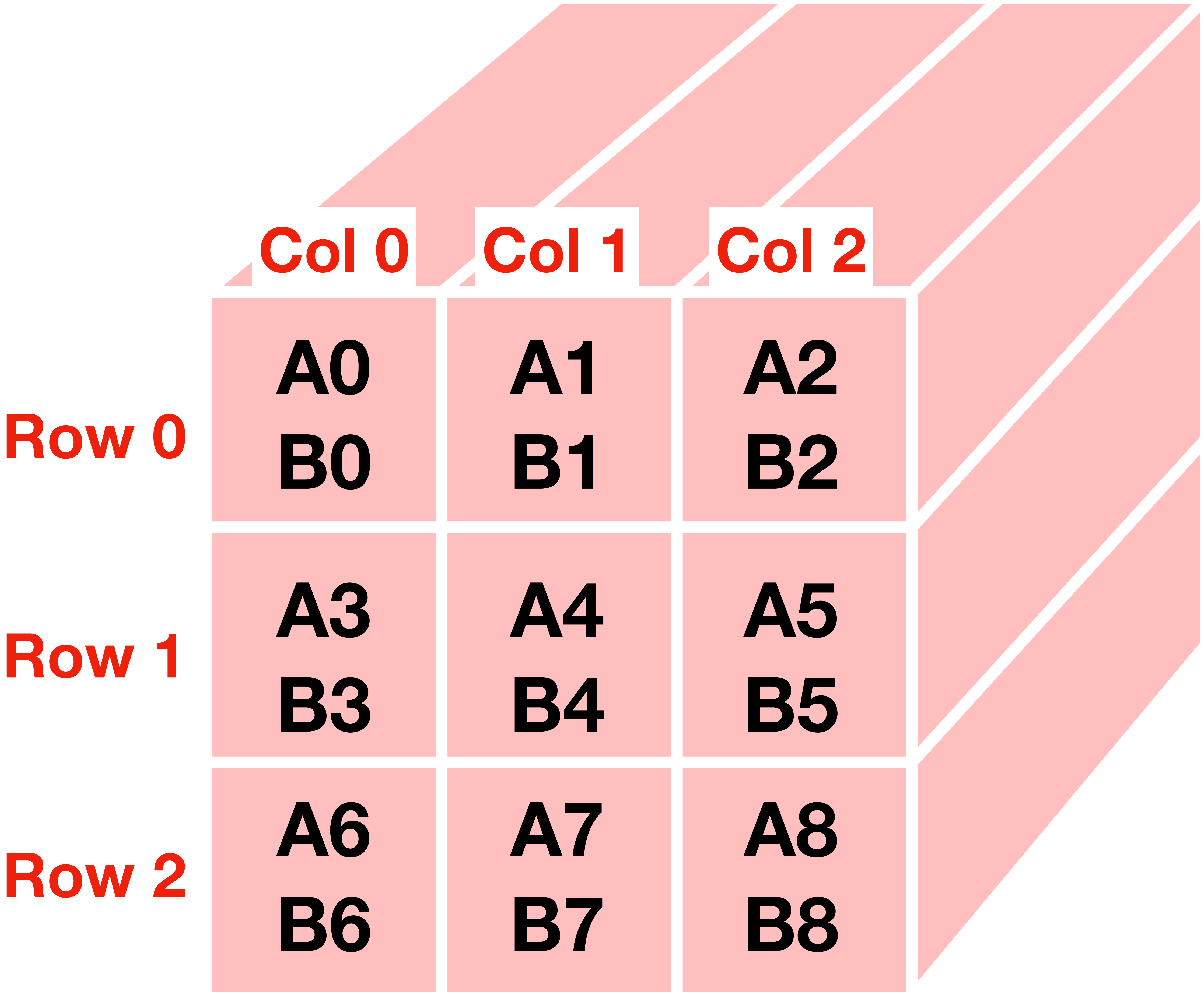


Fox's Algorithm

- Hold all needed values of A, but only one partition of B at a time

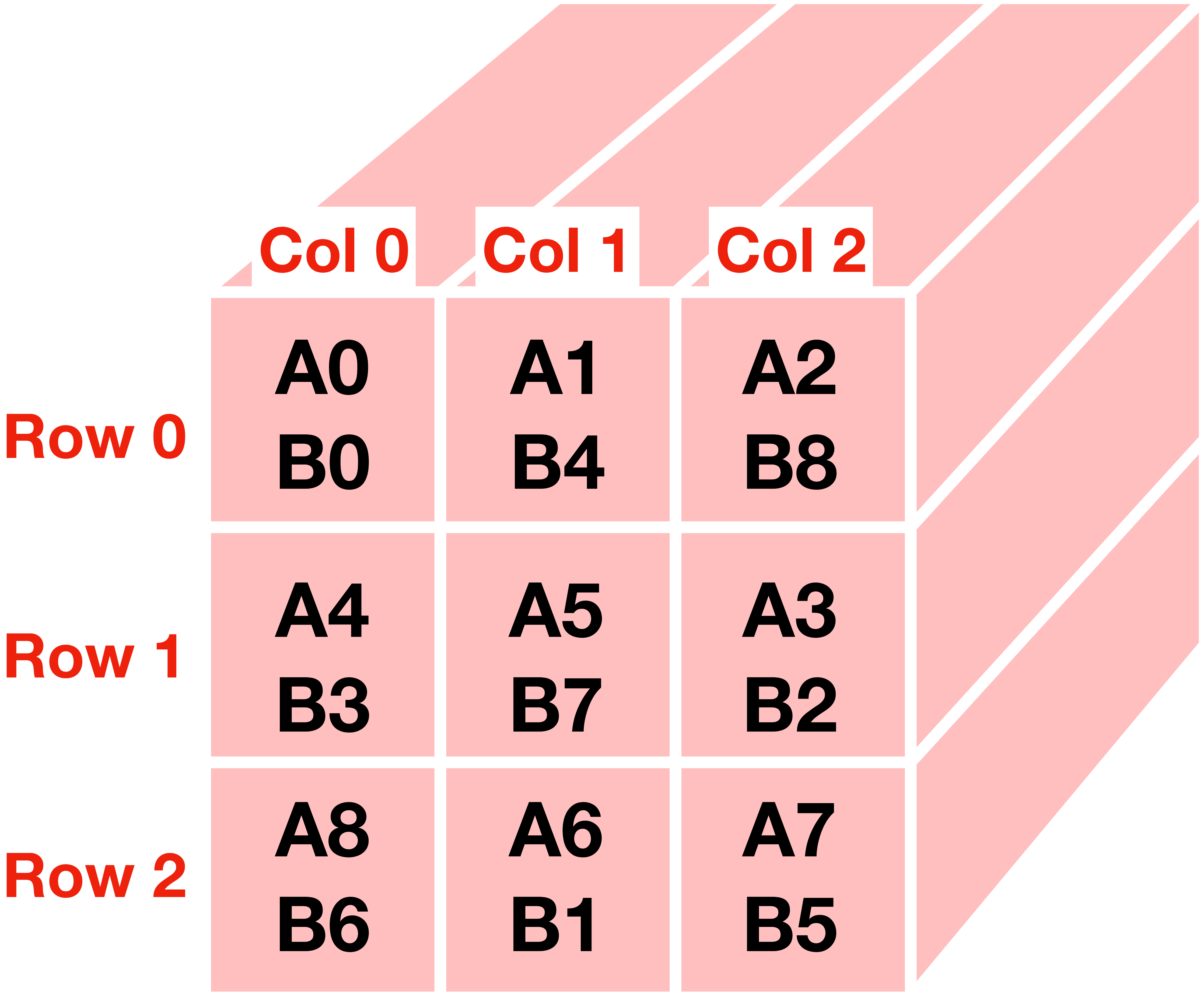


Cannon's Algorithm



- Initial Shift:
  - Process in row I, column J sends local portion of A to process in row I, column J-I
  - Process in row I, column J sends local portion of B to process in row I-J, column J

Cannon's Algorithm

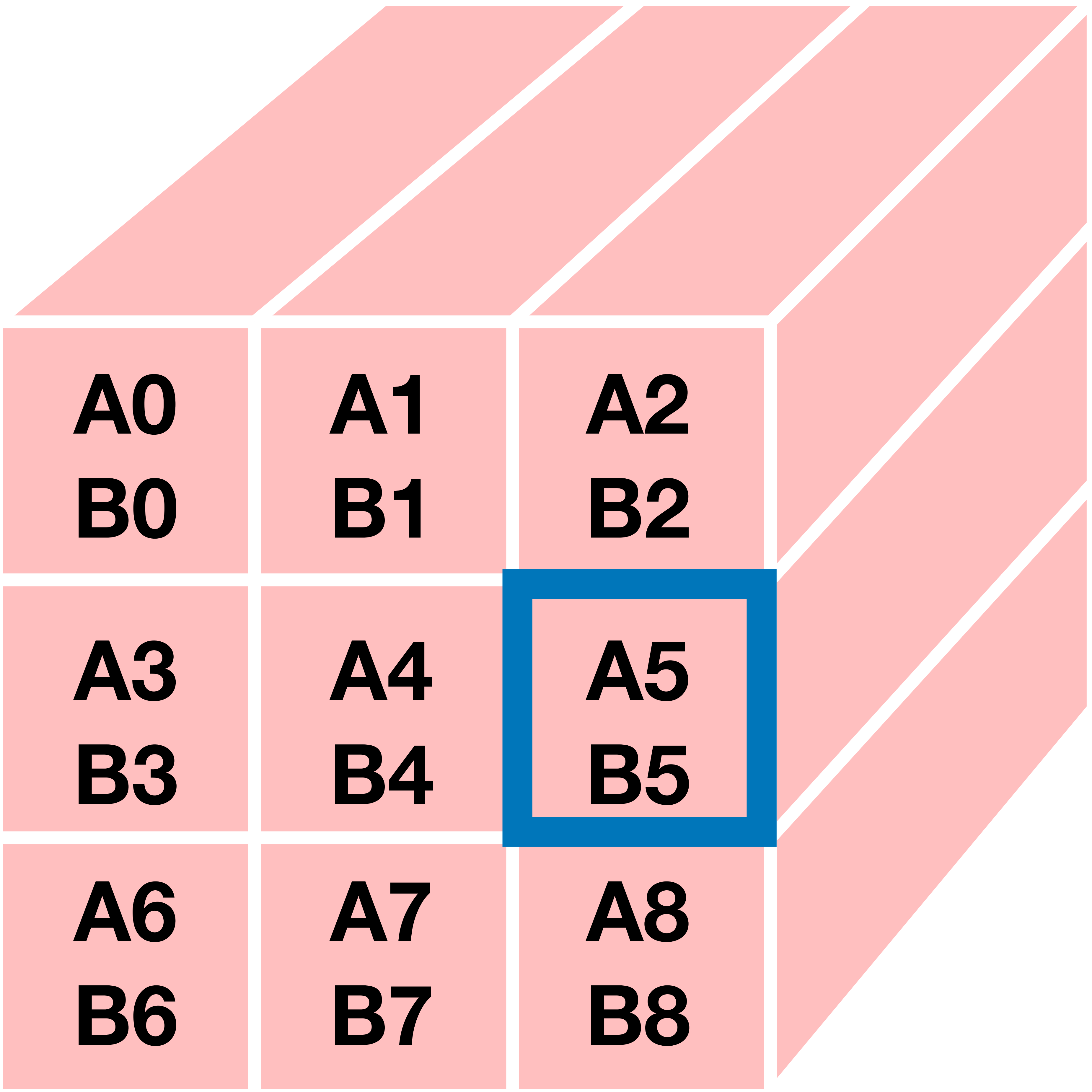


- Initial Shift:
  - Process in row I, column J sends local portion of A to process in row I, column J-I
  - Process in row I, column J sends local portion of B to process in row I-J, column J

Cannon's Algorithm

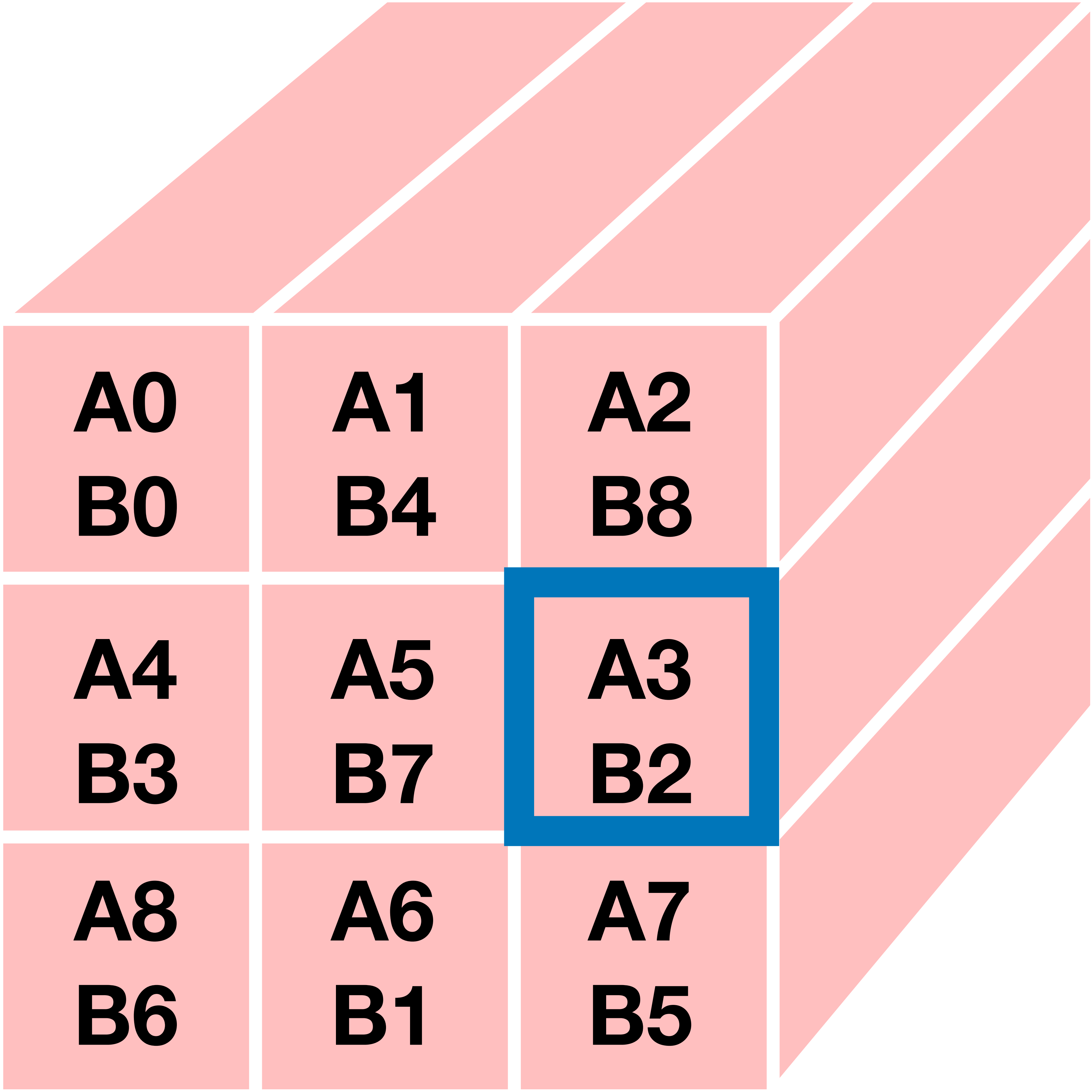
- After initial shift, rotate in opposite direction:
  - Multiply local portion of A with local portion of B
  - Process [I,J] sends local portion of A to Process[I,J+1]
  - Process [I,J] sends local portion of B to process [I+1,J]

Cannon's Algorithm



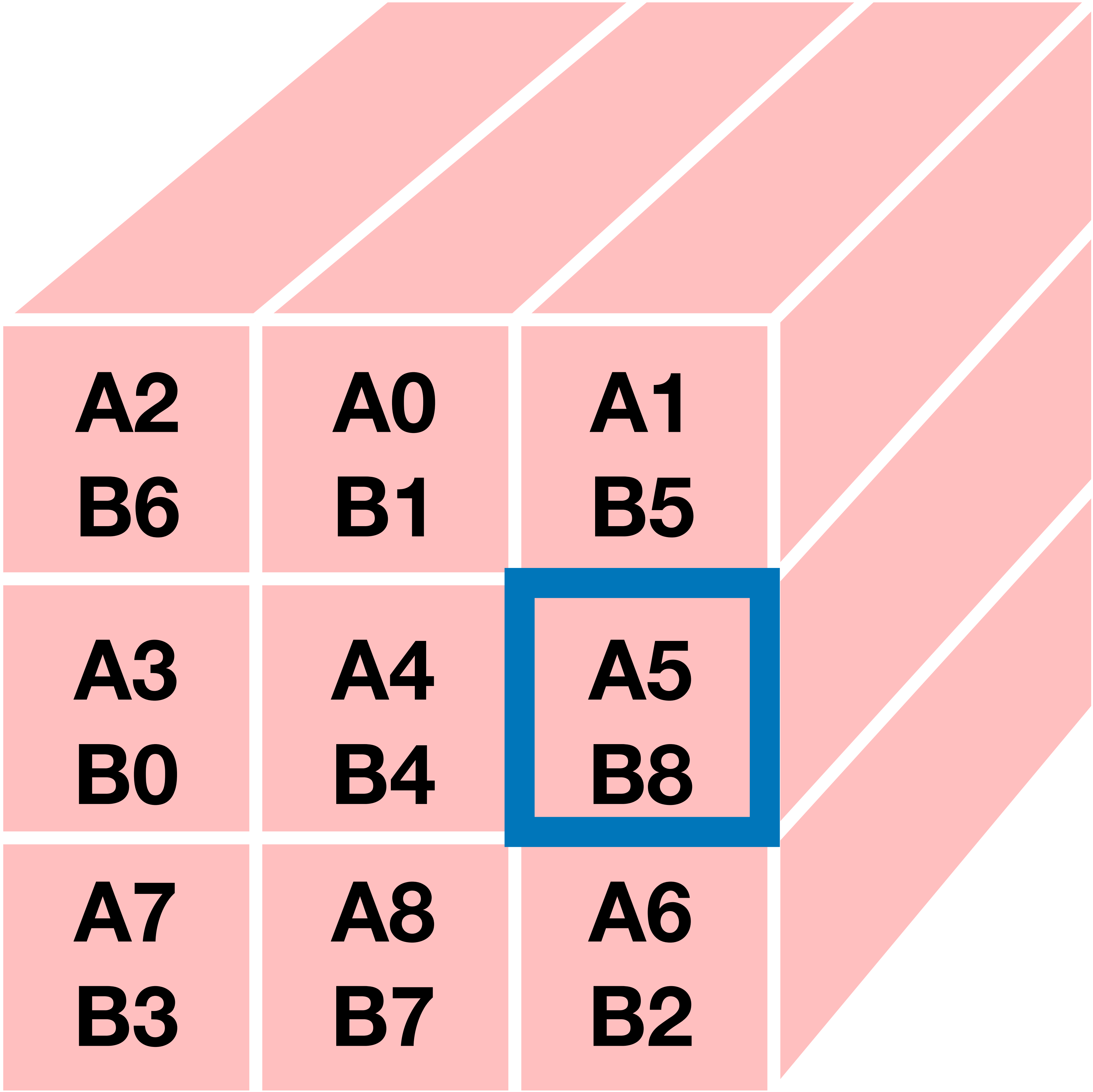
- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

Cannon's Algorithm



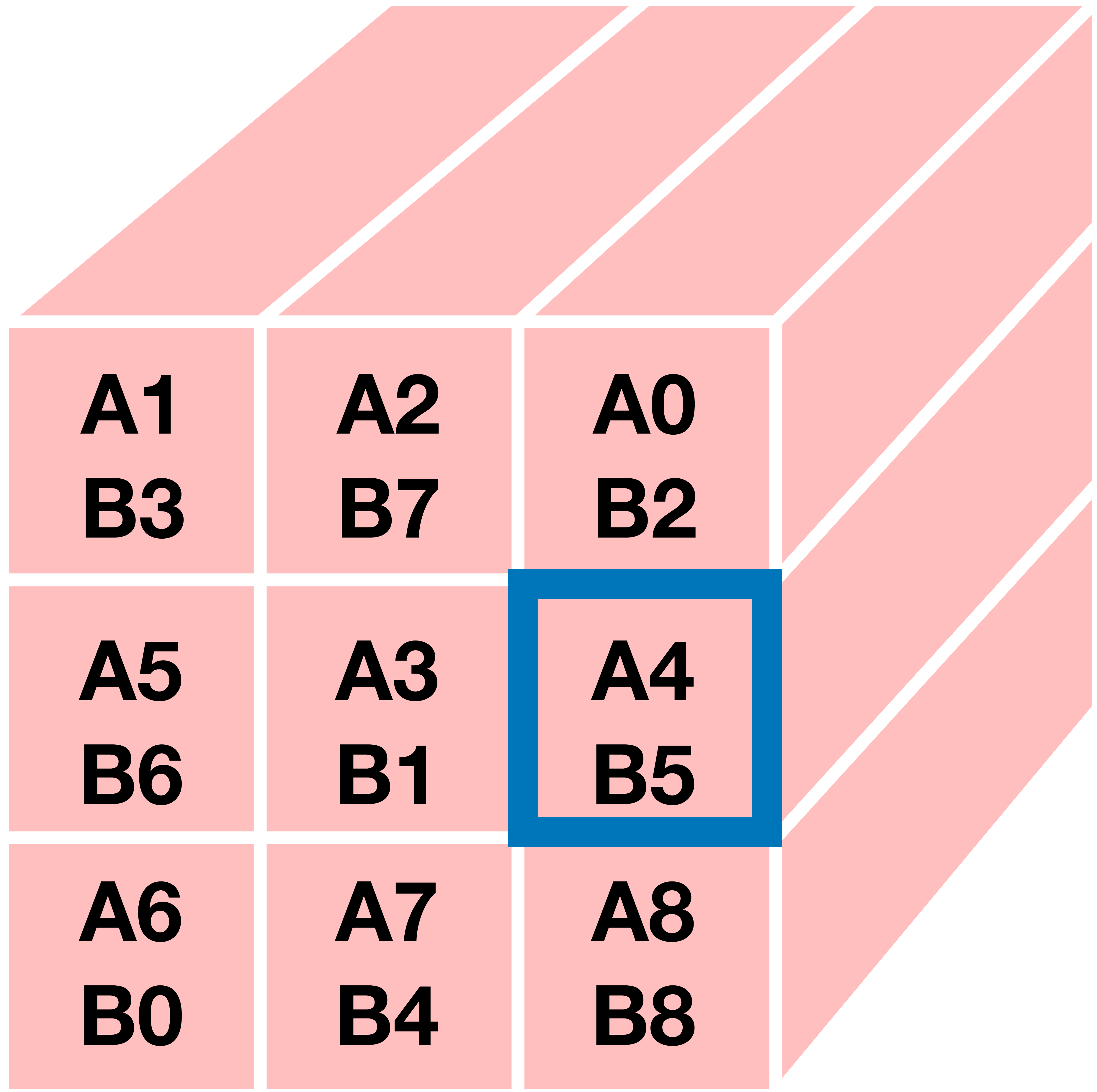
- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

Cannon's Algorithm



- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

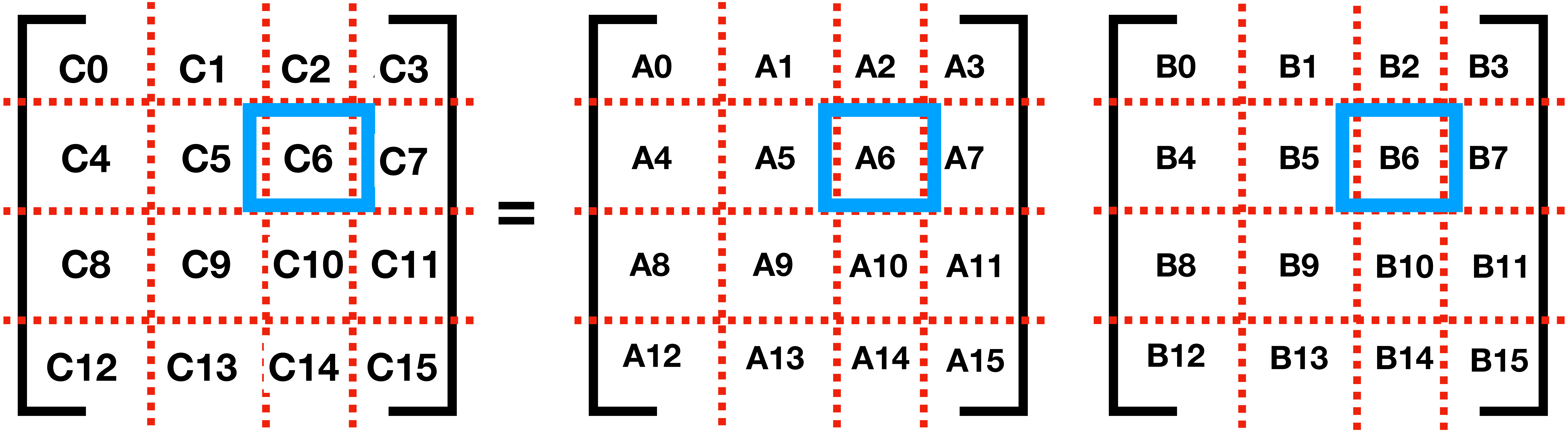
Cannon's Algorithm



- Need to multiply A0 by :
  - B0, B1, B2
- Multiply A1 by :
  - B1, B4, B7
- C5 = [A3*B2 + A4*B5 + A5*B8]

Cannon's Algorithm

- Only hold a single partition of A and a single partition of B at any time



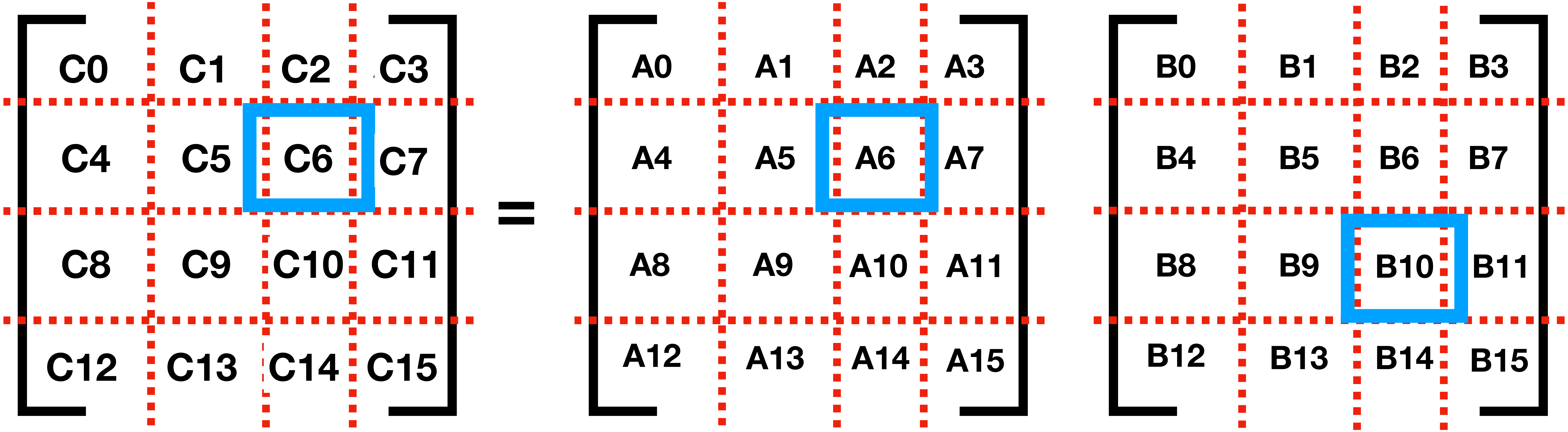
Cannon's Algorithm

- First, rotate data

$$
\begin{bmatrix} C0 & C1 & C2 & C3 \\ C4 & C5 & \boxed{C6} & C7 \\ C8 & C9 & C10 & C11 \\ C12 & C13 & C14 & C15 \end{bmatrix}
=
\begin{bmatrix} A0 & A1 & A2 & A3 \\ A4 & A5 & A6 & \boxed{A7} \\ A8 & A9 & A10 & A11 \\ A12 & A13 & A14 & A15 \end{bmatrix}
\begin{bmatrix} B0 & B1 & B2 & B3 \\ B4 & B5 & B6 & B7 \\ B8 & B9 & B10 & B11 \\ B12 & B13 & \boxed{B14} & B15 \end{bmatrix}
$$

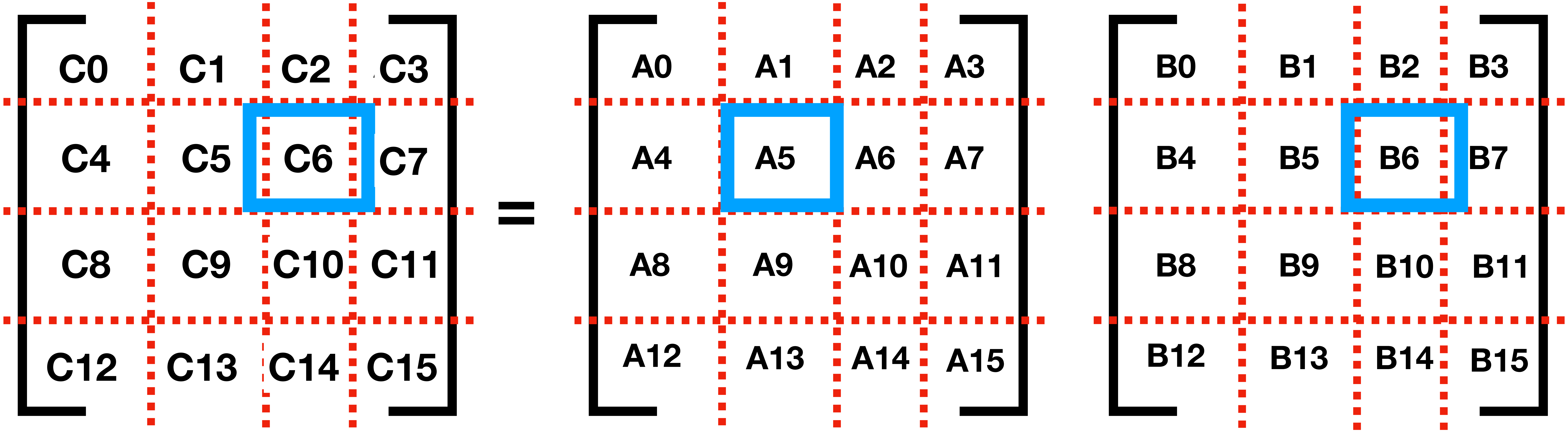
Cannon's Algorithm

- Then multiply local parts and rotate A to the right and B down



Cannon's Algorithm

- Then multiply local parts and rotate A to the right and B down



Cannon's Algorithm

- Then multiply local parts and rotate A to the right and B down