Talk announcement

“Supercomputing scale graph neural network training”
by Dr. Aydın Buluç (Lawrence Berkeley National Laboratory)

Friday 10/18 @ 2-3pm in CODA 114

Abstract: Graph neural networks (GNNs) have demonstrated unprecedented success in numerous challenging scientific problems such as weather prediction, material design, and protein structure prediction. However, training a large-scale GNN is both memory intensive and computationally expensive. Consequently, solving grand challenge scientific problems require supercomputing scale GNN training capabilities. My talk will focus on distributed-memory parallel algorithms for GNN training. We will start by describing how to map full-batch GNN training to communication-avoiding sparse matrix operations. We will then focus on utilizing sparse matrix primitives to parallelize mini-batch GNN training based on node-wise and layer-wise sampling. Finally, we will illustrate techniques that are based on sparsity-aware sparse matrix multiplication algorithms to accelerate both full-graph and mini-batch sampling based GNN training.

CSE 6220/CX 4220
Introduction to HPC

Lecture 15: More on Dense Matrix Algorithms

Helen Xu
hxu615@gatech.edu

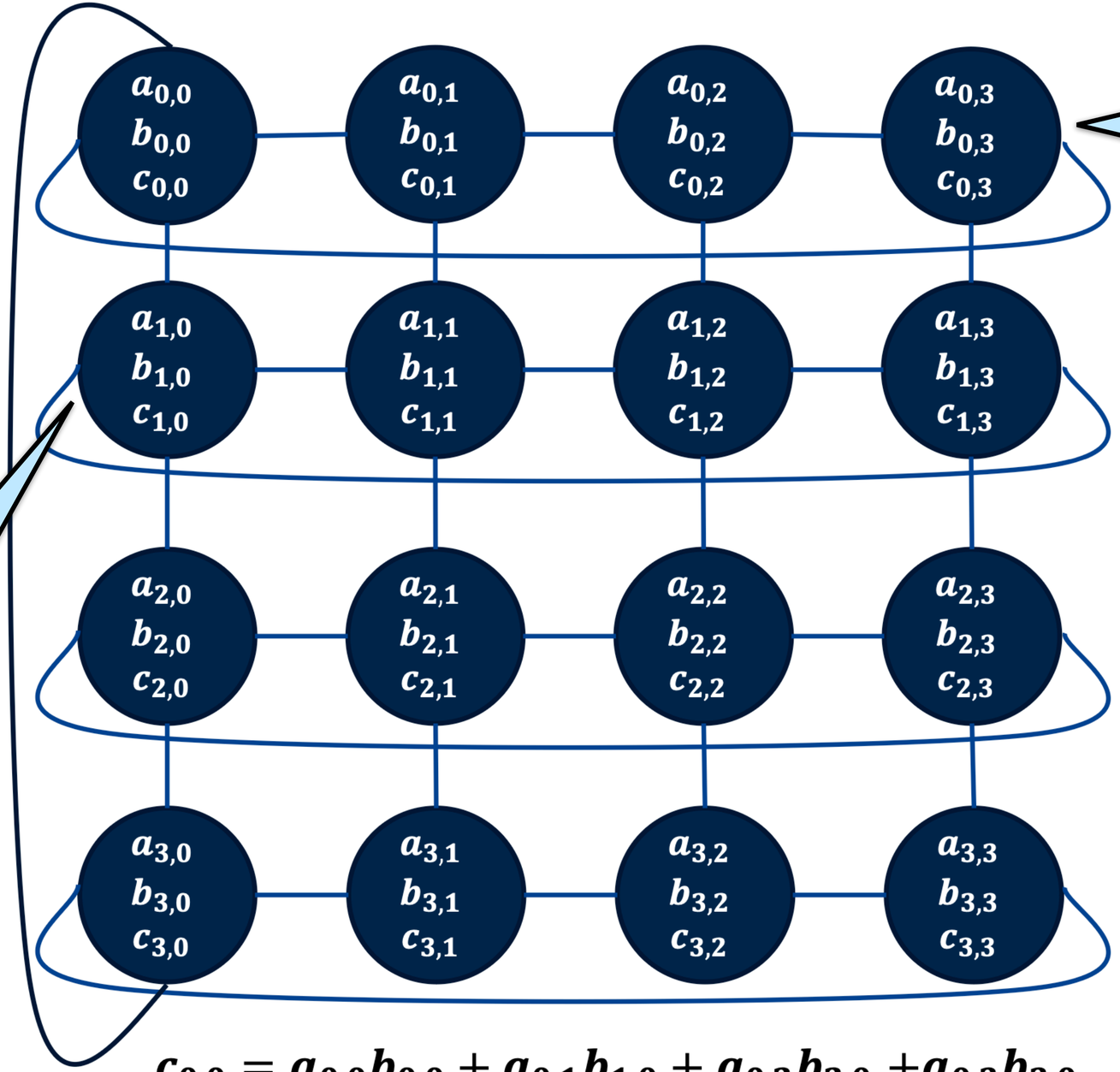
Georgia Tech College of Computing
School of Computational Science and Engineering

Slide credits to Prof. Srinivas Aluru

Restriction - one instance of each element in A, B across all processors

Send a elements to the left one hop at a time

Send b elements to the north one hop at a time

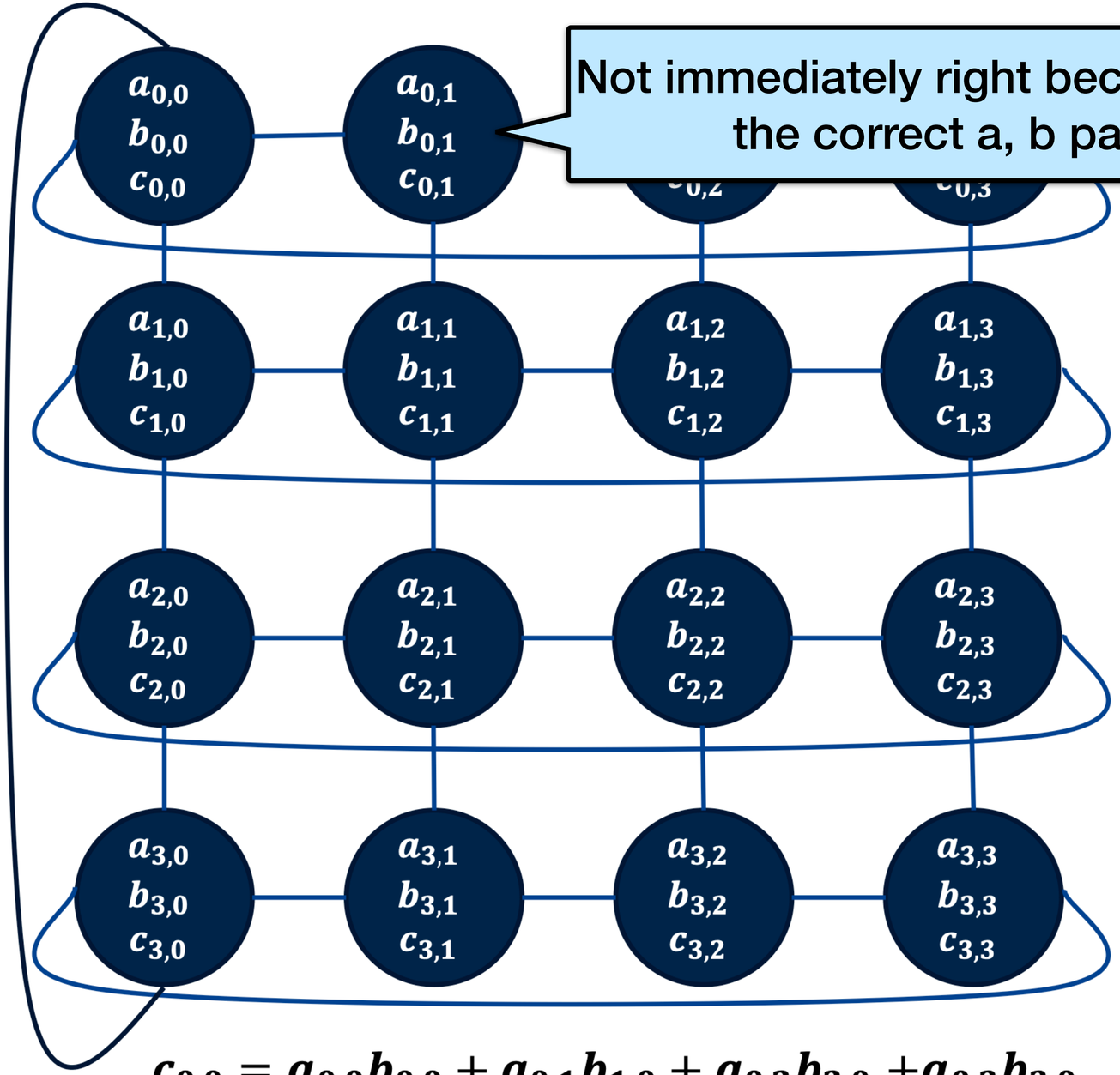


$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0}$$

$$c_{0,1} = a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1}$$

Goal - all processors are computing in each round

We need the corresponding a and b on each processor in each round



Not immediately right because not the correct a, b pair

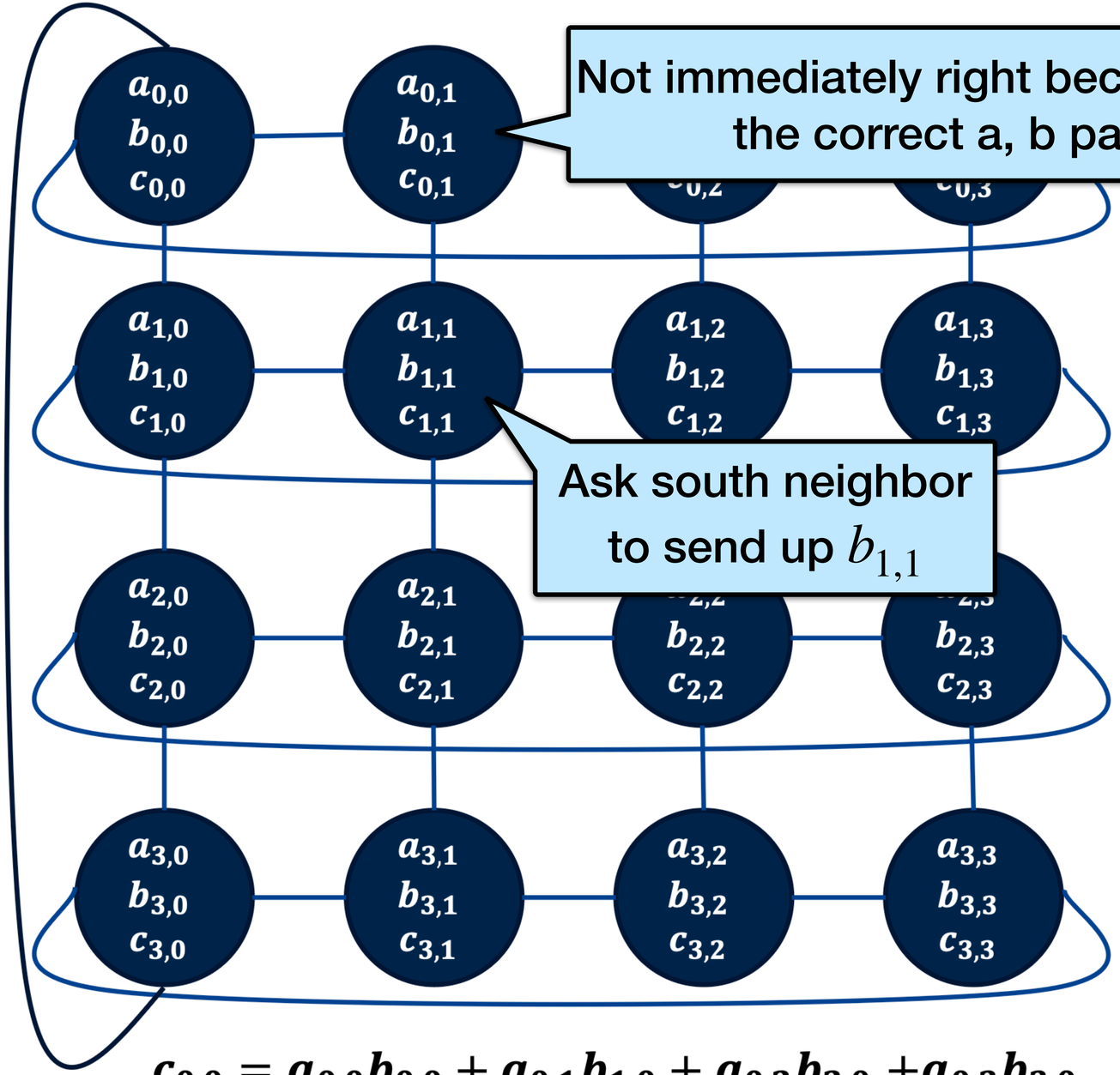
$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0}$$

$$c_{0,1} = a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1}$$

Can compute the terms in any order

Goal - all processors are computing in each round

We need the corresponding a and b on each processor in each round



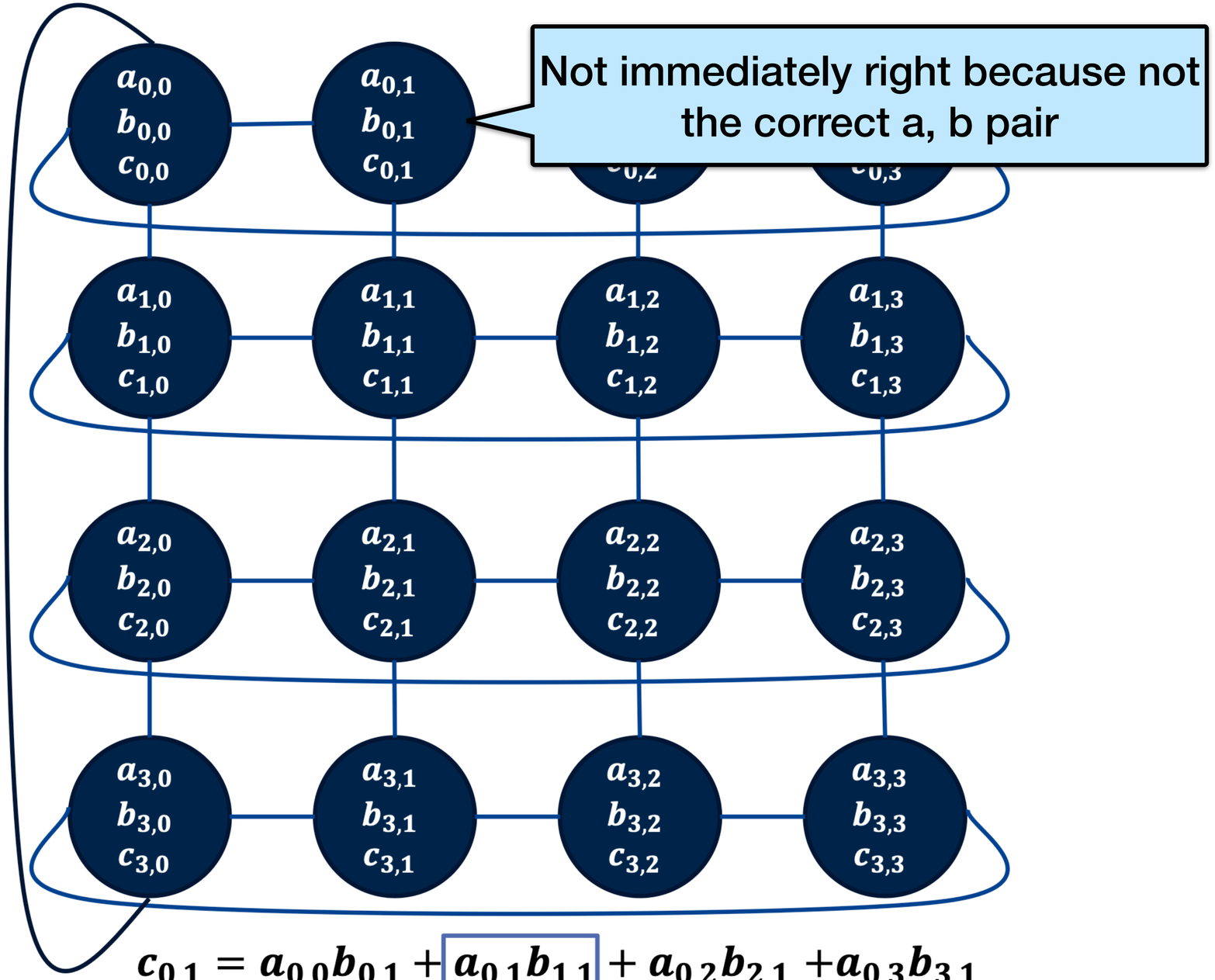
Not immediately right because not the correct a, b pair

Ask south neighbor to send up $b_{1,1}$

$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0}$$

$$c_{0,1} = a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1}$$

Can compute the terms in any order

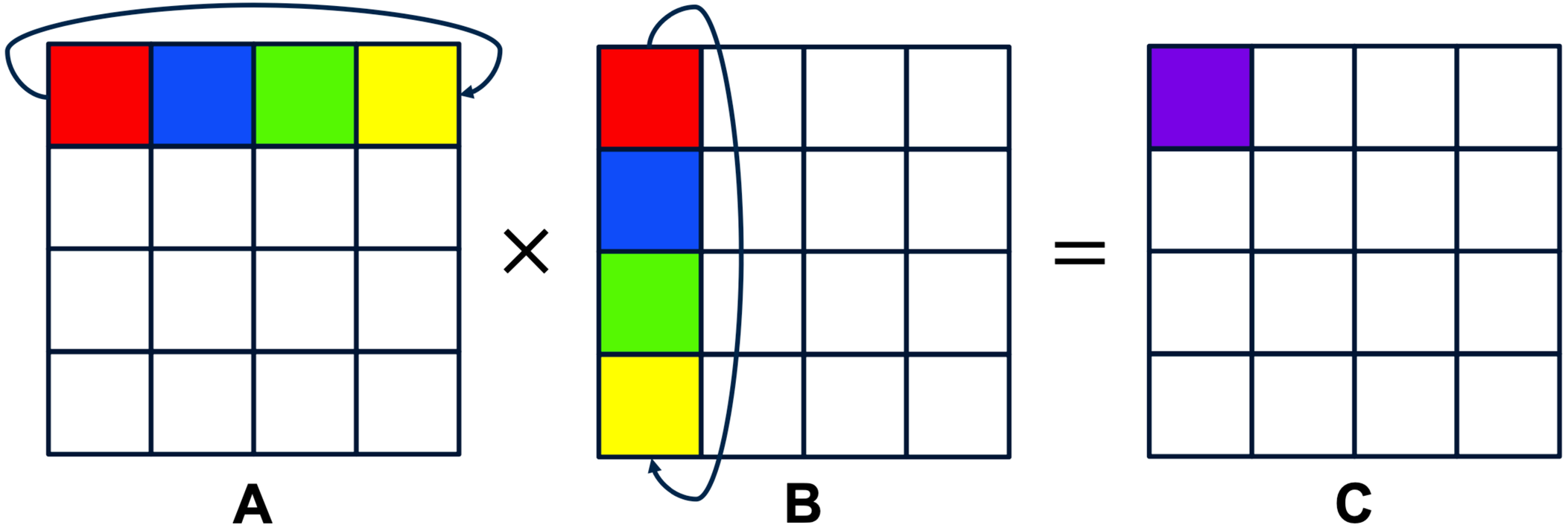


$$c_{0,1} = a_{0,0}b_{0,1} + \boxed{a_{0,1}b_{1,1}} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1}$$

$$c_{0,2} = a_{0,0}b_{0,2} + a_{0,1}b_{1,2} + \boxed{a_{0,2}b_{2,2}} + a_{0,3}b_{3,2}$$

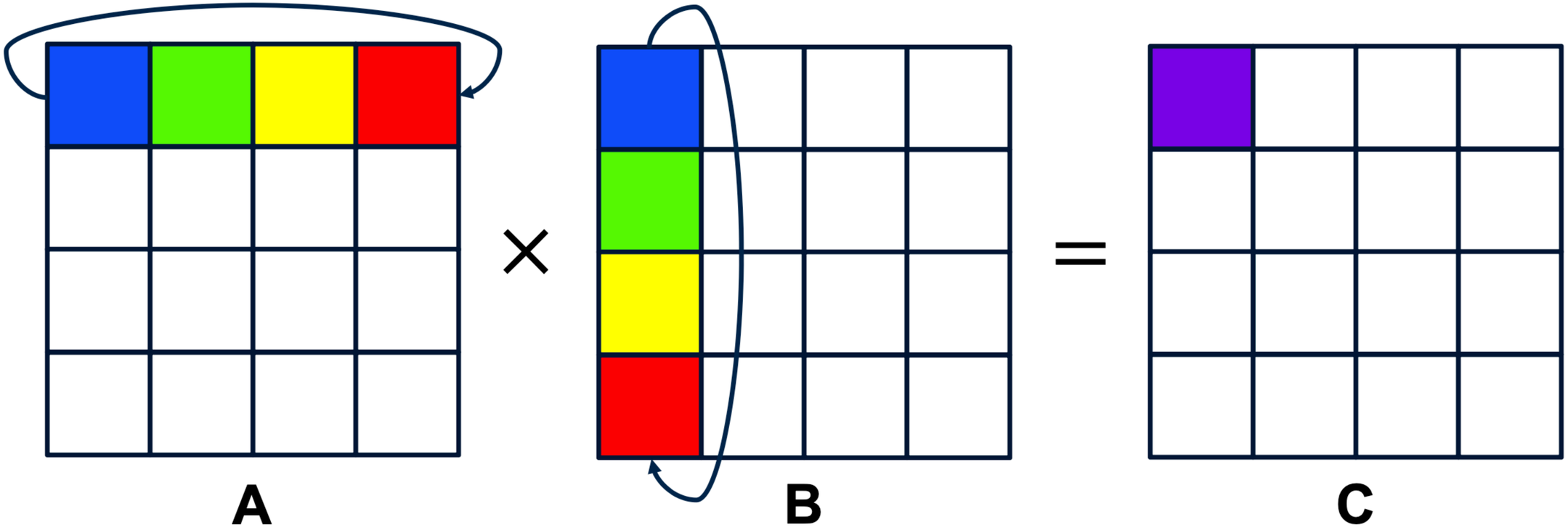
Cannon's Algorithm: Rotations

Consider processor $P_{0,0}$: Step 1 - Multiply the Red blocks



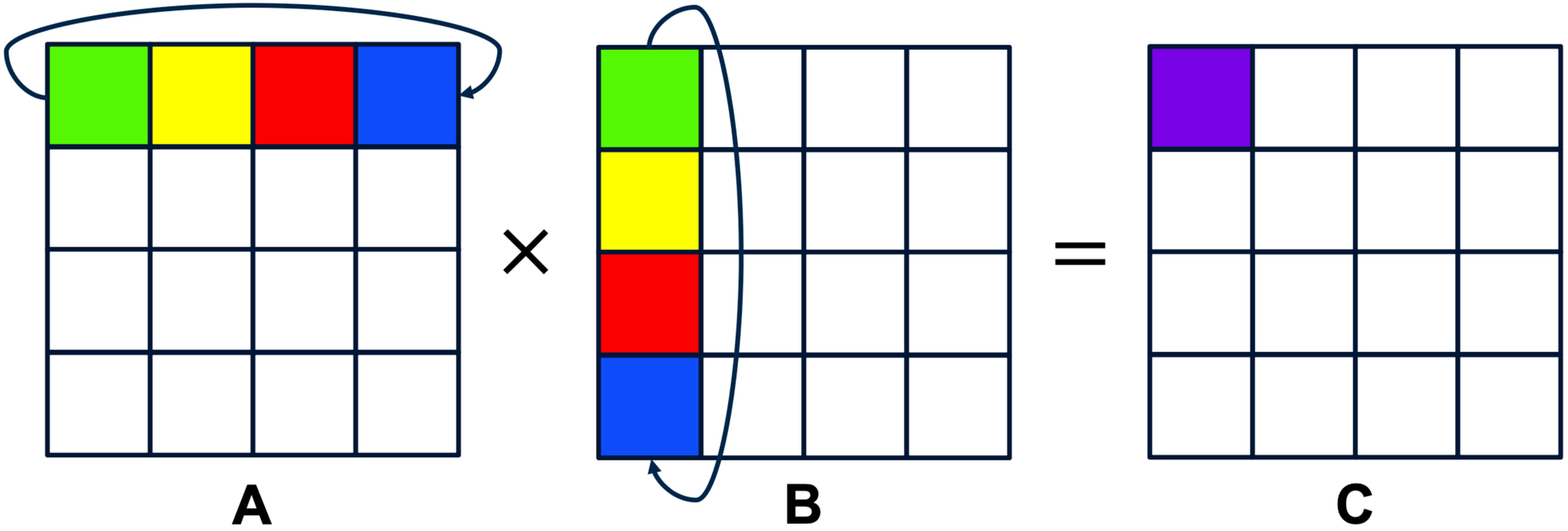
Cannon's Algorithm: Rotations

$P_{0,0}$: Step 2 – Left shift A, Up shift B, Multiply the Blue blocks



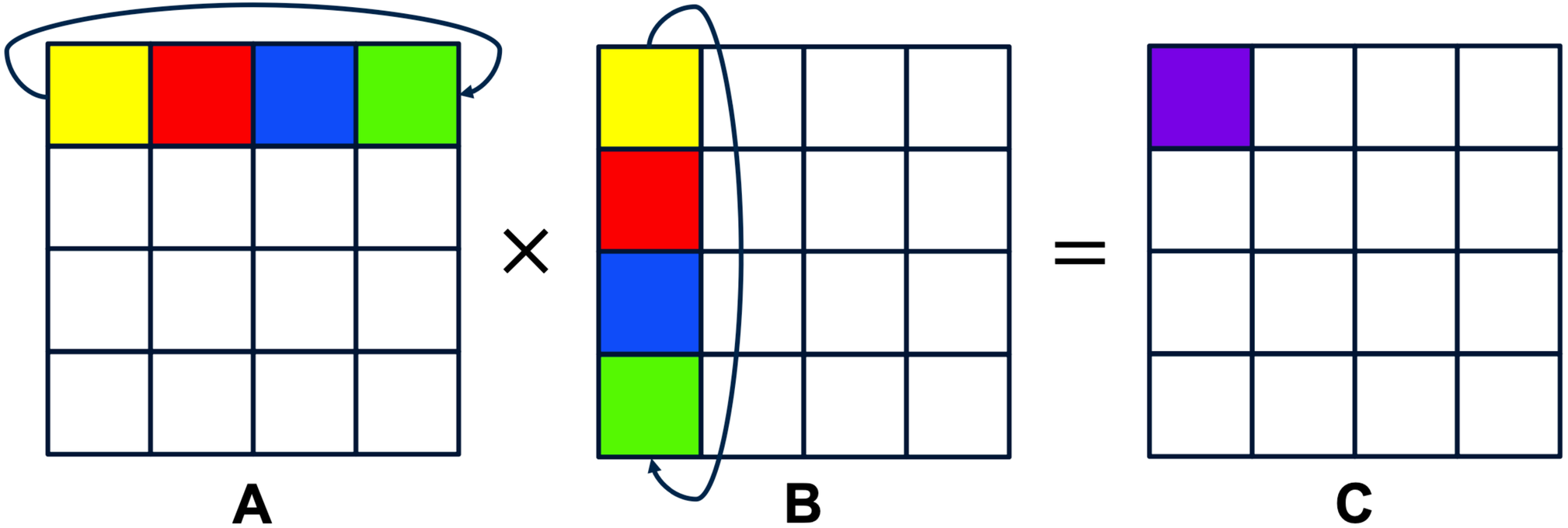
Cannon's Algorithm: Rotations

$P_{0,0}$: Step 3 – Left shift A, Up shift B, Multiply the Green blocks



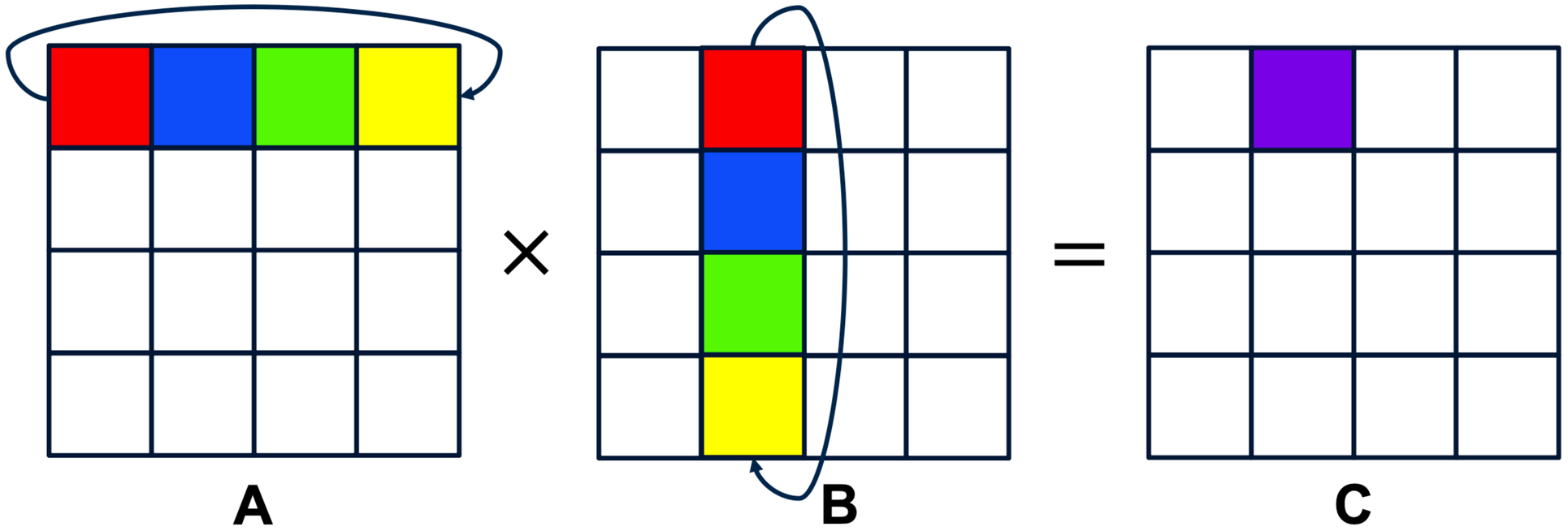
Cannon's Algorithm: Rotations

$P_{0,0}$: Step 4 – Left shift A, Up shift B, Multiply the Yellow blocks



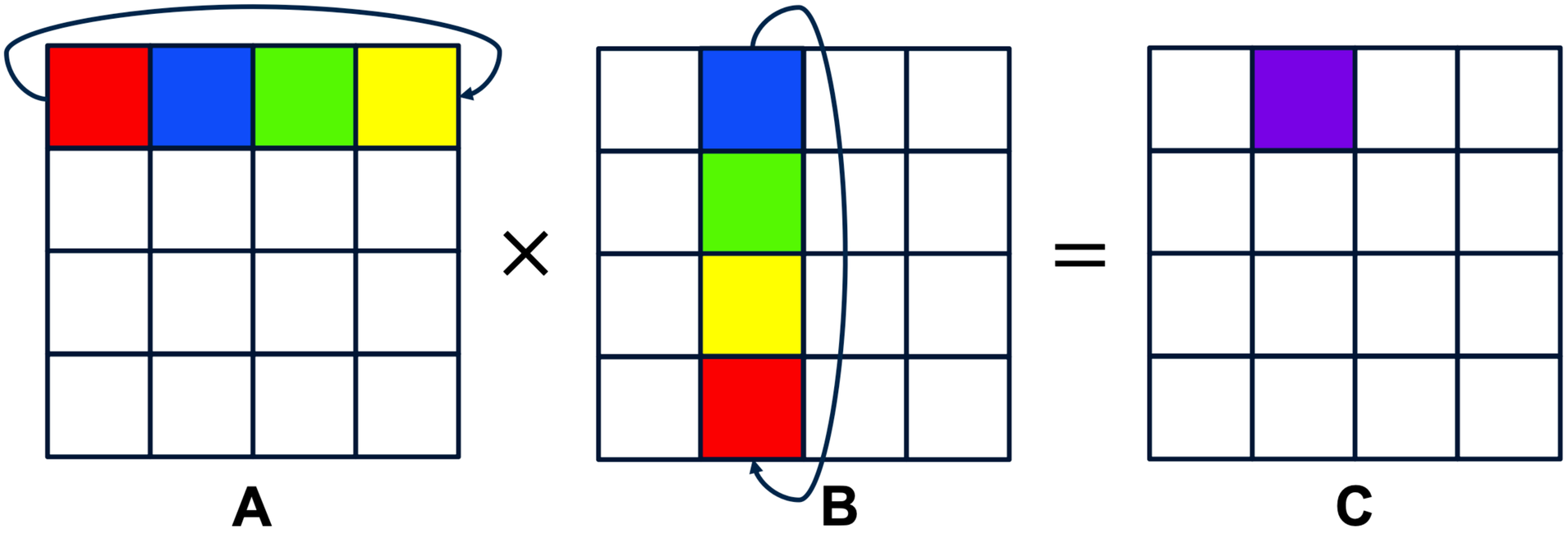
Cannon's Algorithm: Initialization

Consider processor $P_{0,1}$



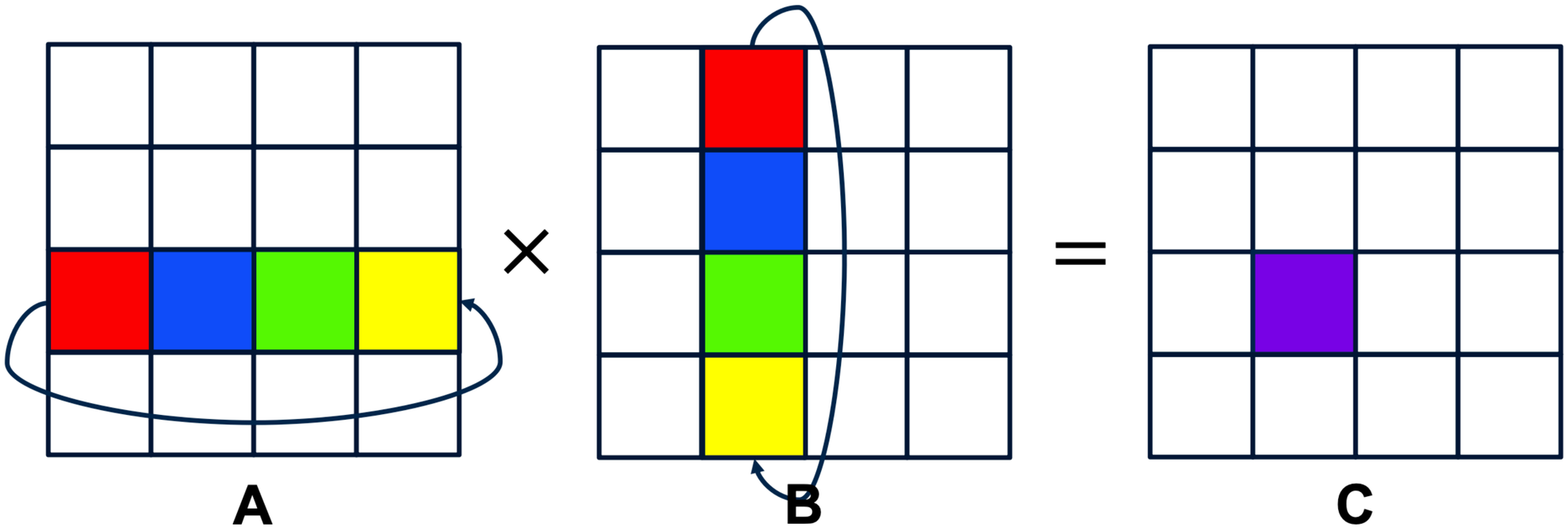
Cannon's Algorithm: Initialization

Consider processor $P_{0,1}$: Align blocks by shifting B up



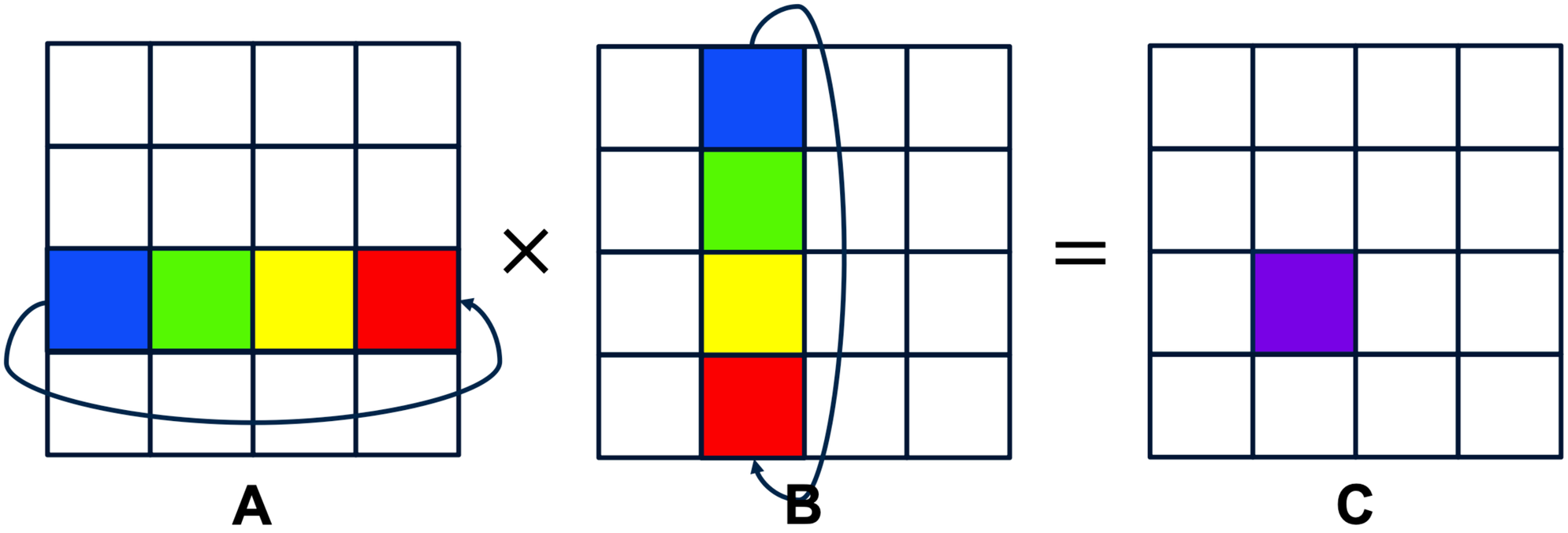
Cannon's Algorithm

- Consider processor $P_{2,1}$
- We need to align the blocks!
  - In row 2 shift A_{ij} left by 2
  - In column 1 shift B_{ij} up by 1



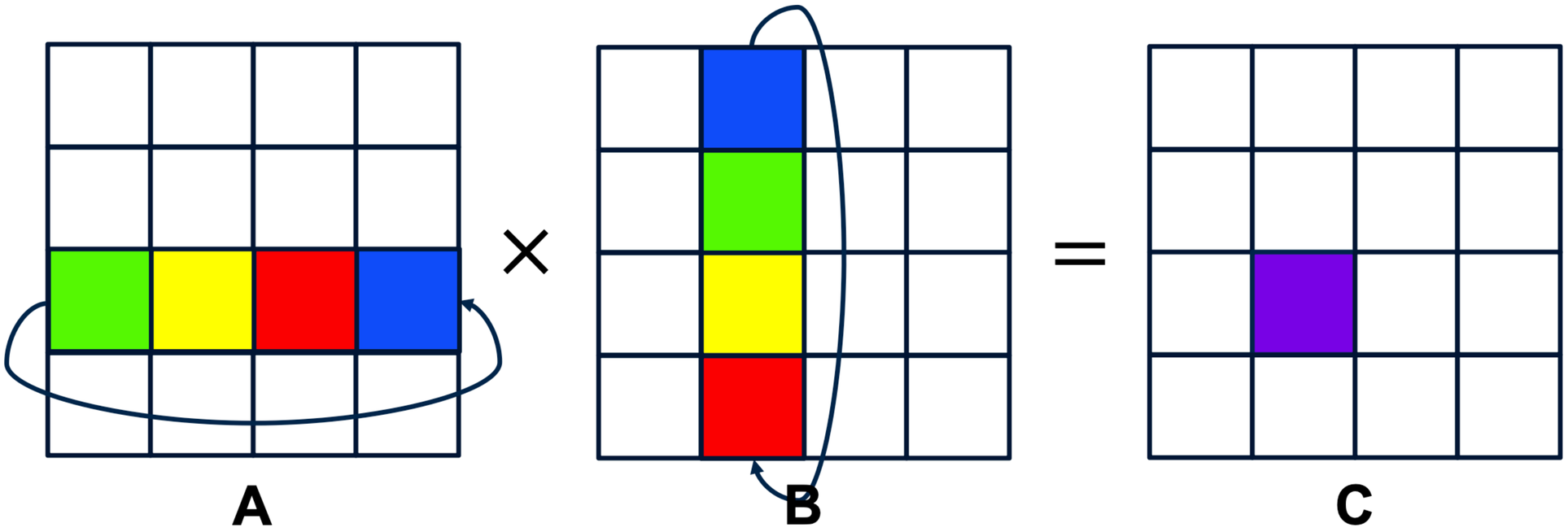
Cannon's Algorithm

- Consider processor $P_{2,1}$
- Alignment Step 1



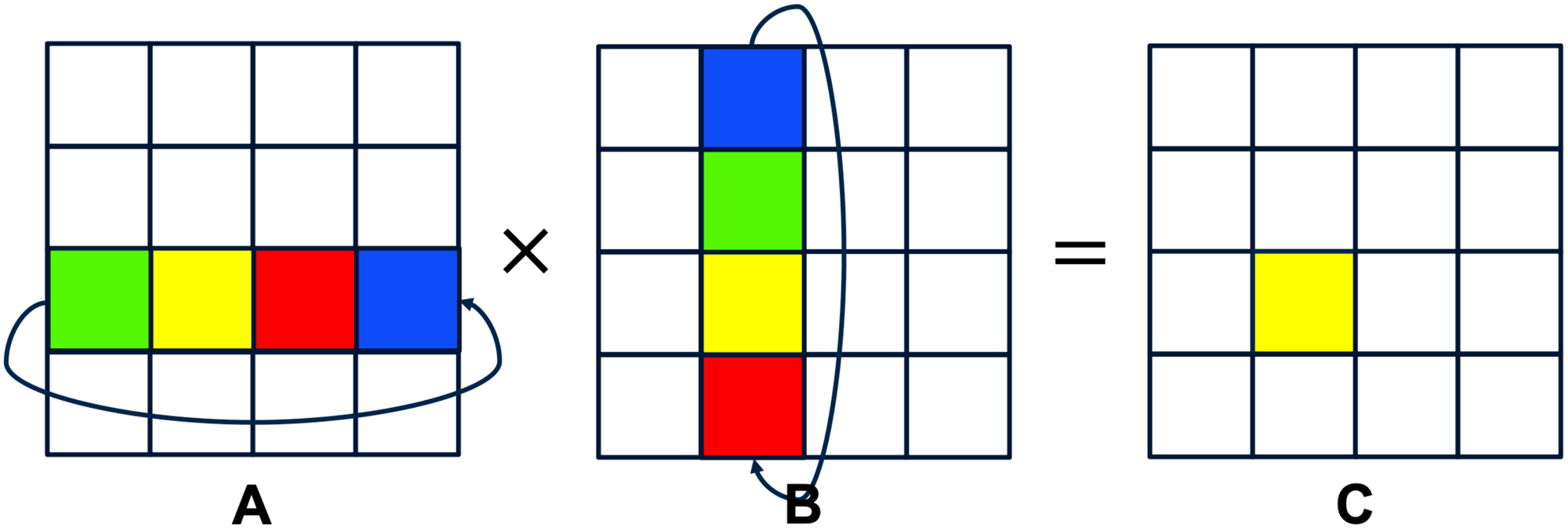
Cannon's Algorithm

- Consider processor $P_{2,1}$
- Alignment Step 2



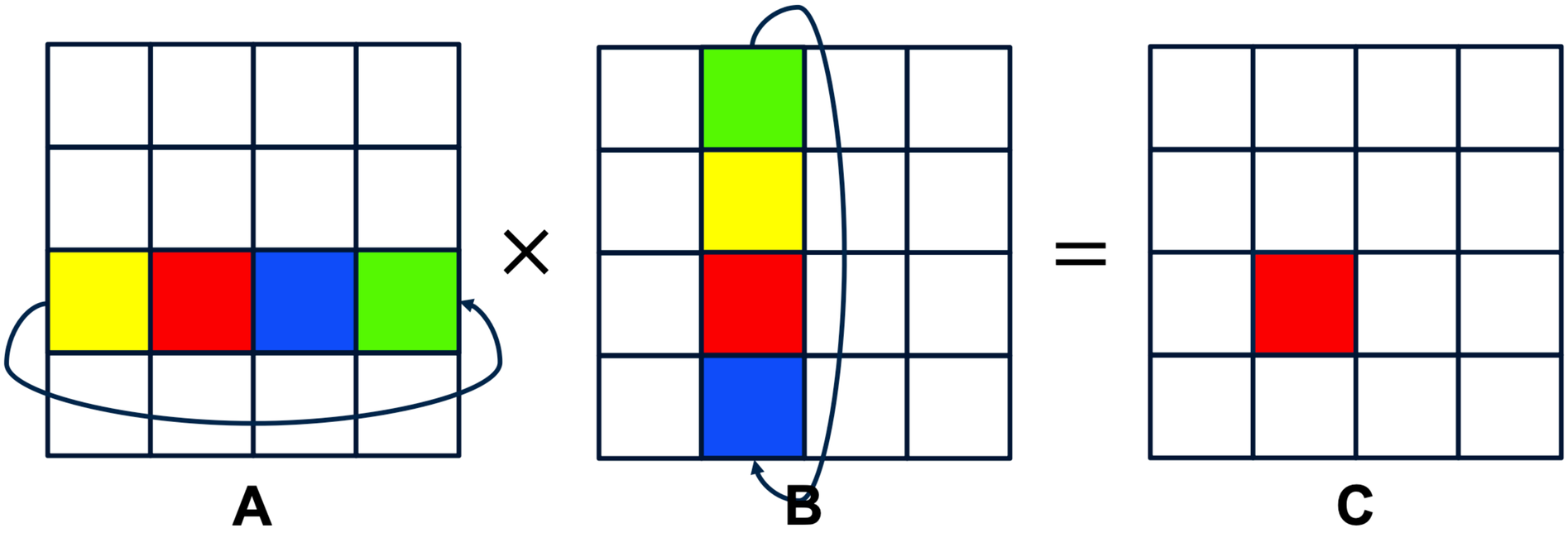
Cannon's Algorithm

- Consider processor $P_{2,1}$
- Multiplication Step 1



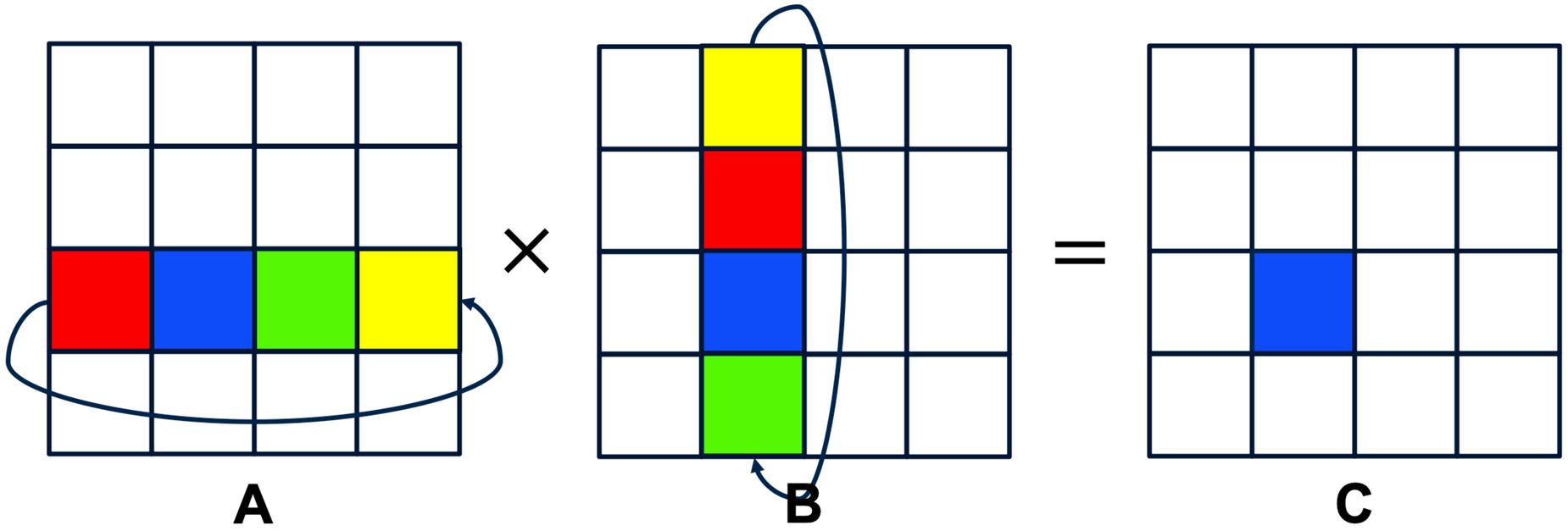
Cannon's Algorithm

- Consider processor $P_{2,1}$
- Multiplication Step 2



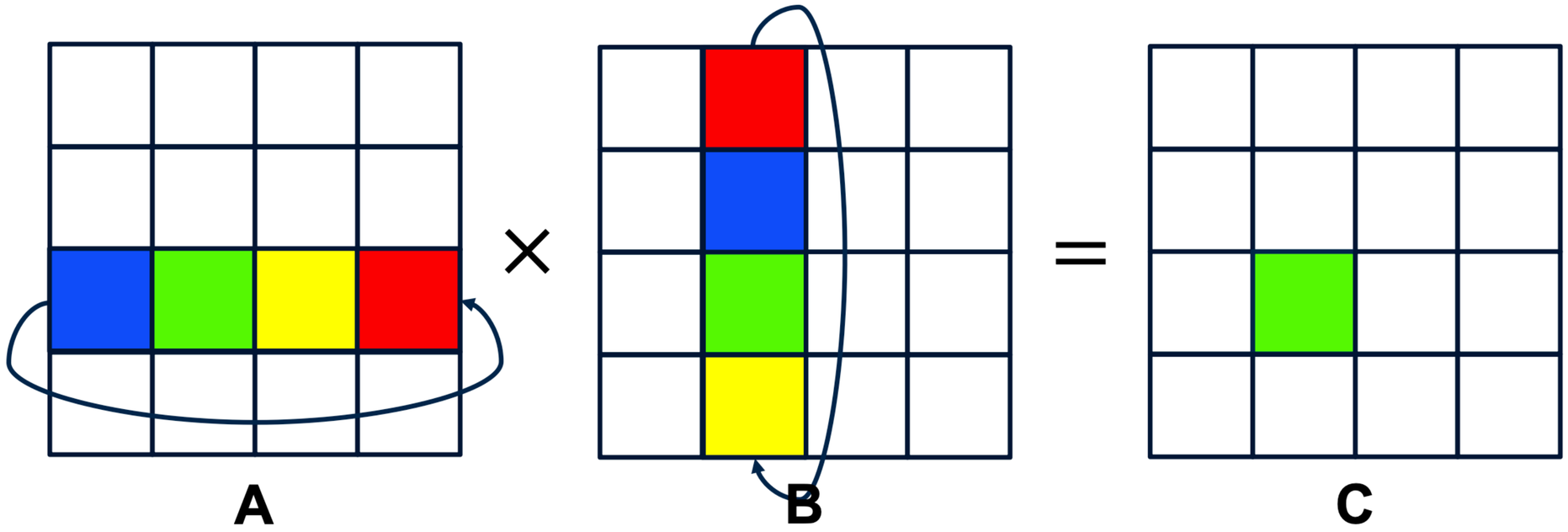
Cannon's Algorithm

- Consider processor $P_{2,1}$
- Multiplication Step 3



Cannon's Algorithm

- Consider processor $P_{2,1}$
- Multiplication Step 4



Cannon's Algorithm: Initial Alignment

Consider processor $P_{i,j}$

- Aligning blocks:
  - Shift $A_{i,j}$ left by i
    - $A_{i,(j+i)}$
  - Shift $B_{i,j}$ up by j
    - $B_{(i+j),j}$
  - $P_{i,j}$ will have $A_{i,(j+i)\ mod\ \sqrt{p}}$ and $B_{(i+j)\ mod\ \sqrt{p},j}$

Line up inner dimension

Analysis of Cannon's Algorithm

- In the alignment step,
  - maximum distance over which a block shifts is $\sqrt{p} - 1$,
  - two shift operations require a total of $O\left(\tau\sqrt{p} + \mu\frac{n^2}{p}\sqrt{p}\right)$ time.

Analysis of Cannon's Algorithm

- In the alignment step,
  - maximum distance over which a block shifts is $\sqrt{p} - 1$,
  - two shift operations require a total of $O\left(\tau\sqrt{p} + \mu\frac{n^2}{p}\sqrt{p}\right)$ time.
- Compute-and-shift phase has $\sqrt{p}$ steps
  - Computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices: $O\left(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\right) = O\left(\frac{n^3}{p}\right)$
  - Communication requires $\sqrt{p}$ shifts $O(\tau\sqrt{p} + \mu\frac{n^2}{p}\sqrt{p})$

Analysis of Cannon's Algorithm

- In the alignment step,
  - maximum distance over which a block shifts is $\sqrt{p} - 1$,
  - two shift operations require a total of $O\left(\tau\sqrt{p} + \mu\frac{n^2}{p}\sqrt{p}\right)$ time.
- Compute-and-shift phase has $\sqrt{p}$ steps
  - Computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices: $O\left(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\right) = O\left(\frac{n^3}{p}\right)$
  - Communication requires $\sqrt{p}$ shifts $O(\tau\sqrt{p} + \mu\frac{n^2}{p}\sqrt{p})$
- Total Time: $O\left(\frac{n^3}{p} + \tau\sqrt{p} + \mu\frac{n^2}{\sqrt{p}}\right)$

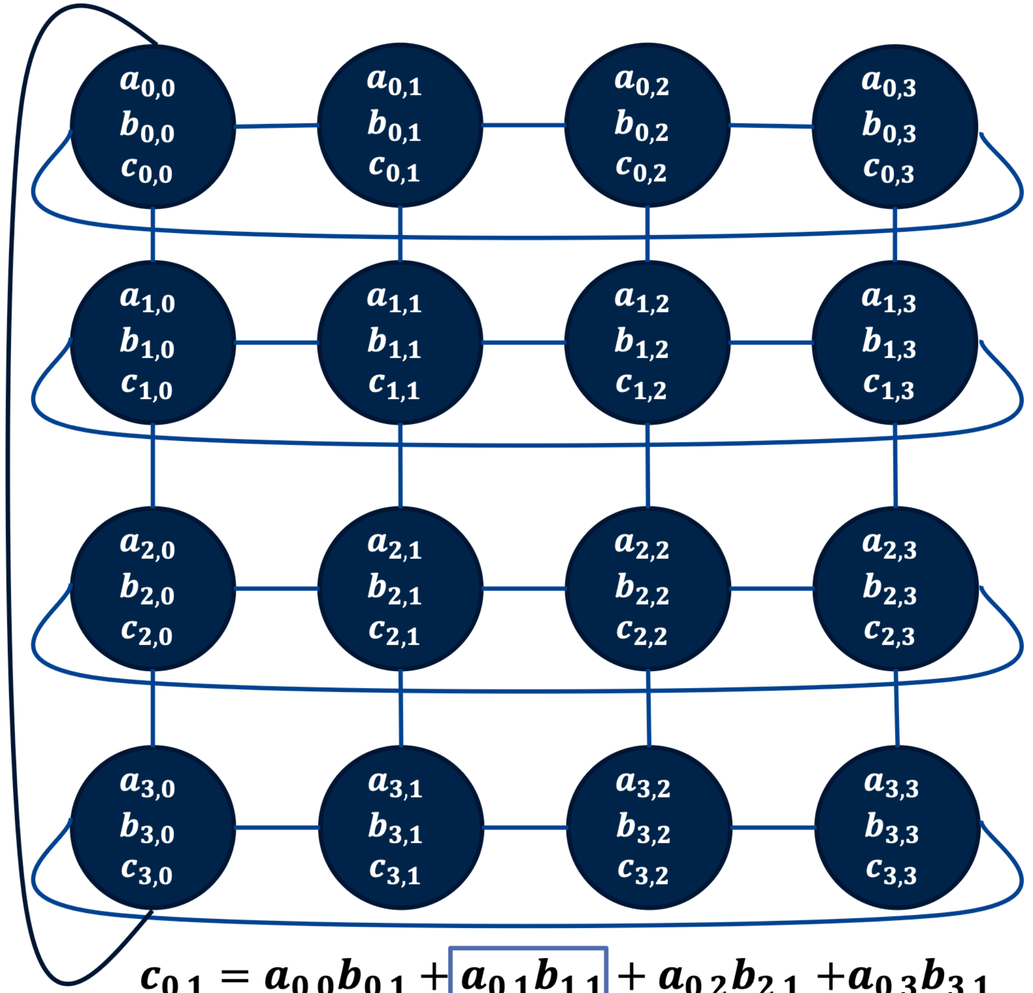
Analysis of Cannon's Algorithm

- In the alignment step,
  - maximum distance over which a block shifts is $\sqrt{p} - 1$,
  - two shift operations require a total of $O\left(\tau\sqrt{p} + \mu\frac{n^2}{p}\sqrt{p}\right)$ time.
- Compute-and-shift phase has $\sqrt{p}$ steps
  - Computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices: $O\left(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\right) = O\left(\frac{n^3}{p}\right)$
  - Communication requires $\sqrt{p}$ shifts $O(\tau\sqrt{p} + \mu\frac{n^2}{p}\sqrt{p})$
- Total Time: $O\left(\frac{n^3}{p} + \tau\sqrt{p} + \mu\frac{n^2}{\sqrt{p}}\right)$
- **More messages, but Cannon's Algorithm is memory optimal.**

```
1  MatrixMatrixMultiply(int n, double *a, double *b, double *c,
2                       MPI_Comm comm)
3  {
4    int i;
5    int nlocal;
6    int npes, dims[2], periods[2];
7    int myrank, my2drank, mycoords[2];
8    int uprank, downrank, leftrank, rightrank, coords[2];
9    int shiftsource, shiftdest;
10   MPI_Status status;
11   MPI_Comm comm_2d;
12
13   /* Get the communicator related information */
14   MPI_Comm_size(comm, &npes);
15   MPI_Comm_rank(comm, &myrank);
16
17   /* Set up the Cartesian topology */
18   dims[0] = dims[1] = sqrt(npes);
19
20   /* Set the periods for wraparound connections */
21   periods[0] = periods[1] = 1;
22
23   /* Create the Cartesian topology, with rank reordering */
24   MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
25
26   /* Get the rank and coordinates with respect to the new topology */
27   MPI_Comm_rank(comm_2d, &my2drank);
28   MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
29
30   /* Compute ranks of the up and left shifts */
31   MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
32   MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
33
34   /* Determine the dimension of the local matrix block */
35   nlocal = n/dims[0];
36
37   /* Perform the initial matrix alignment. First for A and then for B */
38   MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
39   MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest,
40       1, shiftsource, 1, comm_2d, &status);
```

```
41
42    MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
43    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
44        shiftdest, 1, shiftsource, 1, comm_2d, &status);
45
46    /* Get into the main computation loop */
47    for (i=0; i<dims[0]; i++) {
48      MatrixMultiply(nlocal, a, b, c); /* c = c + a*b */
49
50      /* Shift matrix a left by one */
51      MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
52          leftrank, 1, rightrank, 1, comm_2d, &status);
53
54      /* Shift matrix b up by one */
55      MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
56          uprank, 1, downrank, 1, comm_2d, &status);
57    }
58
59    /* Restore the original distribution of a and b */
60    MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
61    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
62        shiftdest, 1, shiftsource, 1, comm_2d, &status);
63
64    MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
65    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
66        shiftdest, 1, shiftsource, 1, comm_2d, &status);
67
68    MPI_Comm_free(&comm_2d);   /* Free up communicator */
69  }
70
71  /* This function performs a serial matrix-matrix multiplication c = a*b */
72  MatrixMultiply(int n, double *a, double *b, double *c)
73  {
74    int i, j, k;
75
76    for (i=0; i<n; i++)
77      for (j=0; j<n; j++)
78        for (k=0; k<n; k++)
79          c[i*n+j] += a[i*n+k]*b[k*n+j];
80  }
```

Limitations of Cannon's algorithm



Requires a square 2D grid of processors

Also, A and B need to be square

$$c_{0,1} = a_{0,0}b_{0,1} + \boxed{a_{0,1}b_{1,1}} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1}$$

$$c_{0,2} = a_{0,0}b_{0,2} + a_{0,1}b_{1,2} + \boxed{a_{0,2}b_{2,2}} + a_{0,3}b_{3,2}$$

Outer Product Matrix Multiplication

Inner-Product Matrix Multiplication

```
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < n; ++j) {
    for(int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

So far, we have been doing inner product matrix multiply, which calculates n^2 dot products (inner products).

$$C[i,j] = A[i,:] \times B[:,j]$$

Outer product definition

Multiplying two vectors u (n x 1) and v (1 x n) gives us a matrix of size n x n where each entry is the product of one element from u and v.

$$\begin{vmatrix} a \\ b \\ c \end{vmatrix} * \begin{bmatrix} e & f & g \end{bmatrix} = \begin{vmatrix} a*e & a*f & a*g \\ b*e & b*f & b*g \\ c*e & c*f & c*g \end{vmatrix}$$

https://cseweb.ucsd.edu/classes/sp11/cse262-a/Lectures/262-pres1-hal.pdf

Outer-Product Matrix Multiplication

```
for(int k = 0; k < n; ++k) {
  for(int i = 0; i < n; ++i) {
    for(int j = 0; j < n; ++j) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

The product of a column vector by a row vector is an n x n matrix, or the "multiplication table" of the two elements.

Running over the outer k loop in matrix multiply, we can express C as the **sum of k of these "multiplication table" matrices**.

$$C = \sum_k A[:, k] \times B[k, :]$$

From John Gilbert

SUMMA algorithm

Scalable Universal Matrix Multiplication Algorithm (SUMMA) [van de Geijn and Watts, 97]

- Used in ScaLAPACK, PLAPACK, and PBLAS libraries
- Based on outer product

Avoids need for square matrices and processor grid

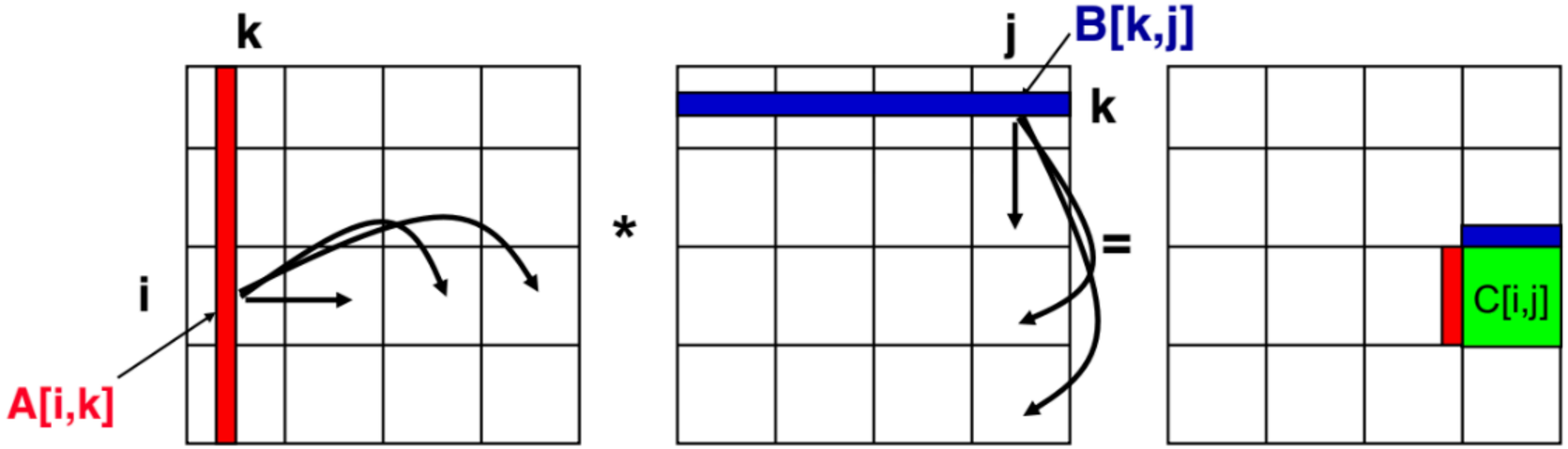
```
for(int k = 0; k < n; ++k) {
  C[:,:] += A[:, k] + B[k, :]
}
```

Parallelize all (i, j)

SUMMA algorithm

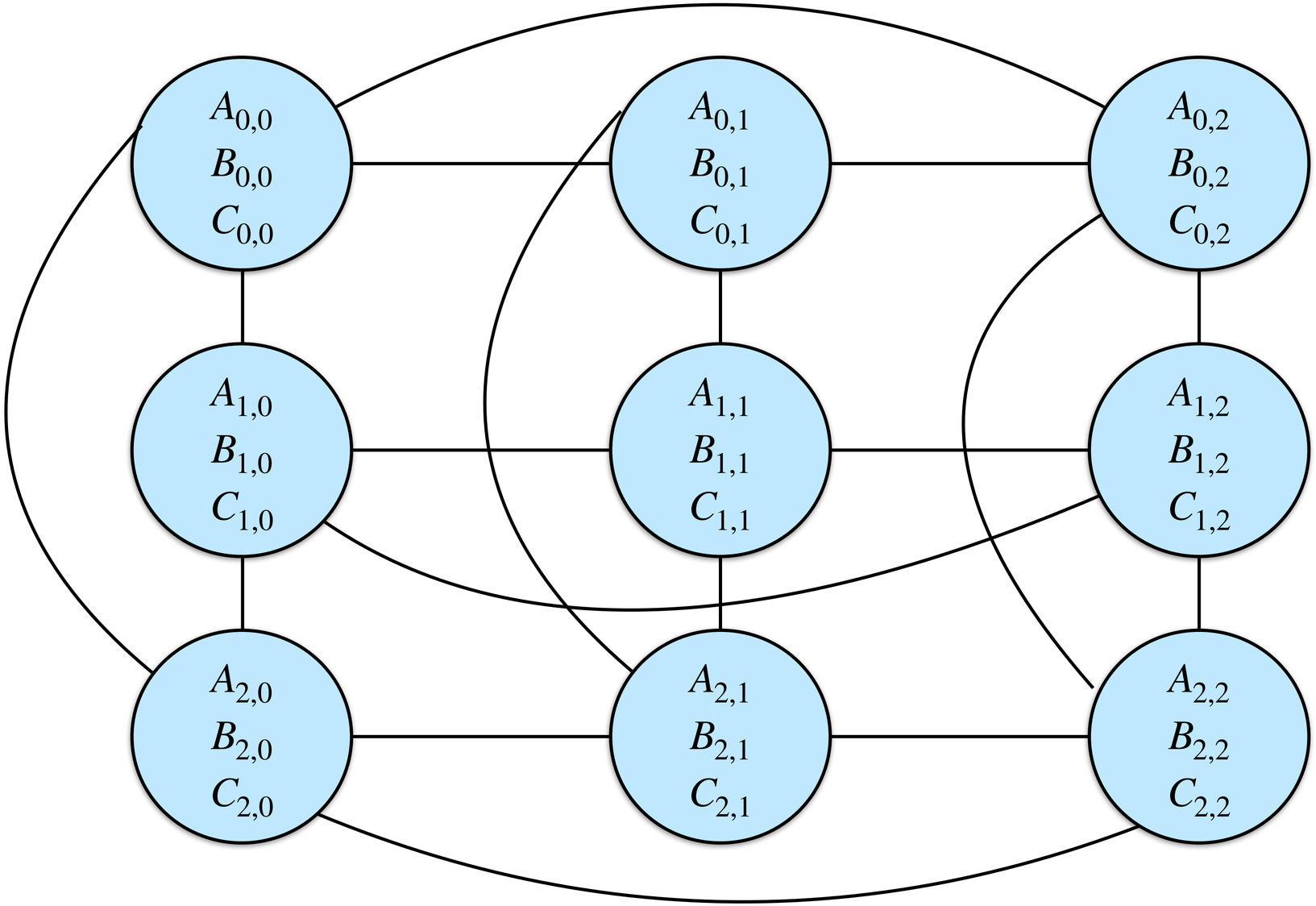
```
On each process P(i, j):

for(int k = 0; k < n; ++k) {
  Bcast column k of A (A[i, k]) within proc row i
  Bcast row k of B (B[k, j]) within proc column j
  Do C += A[i, k] * B[k, j]
}
```

Compute the sum of all outer products



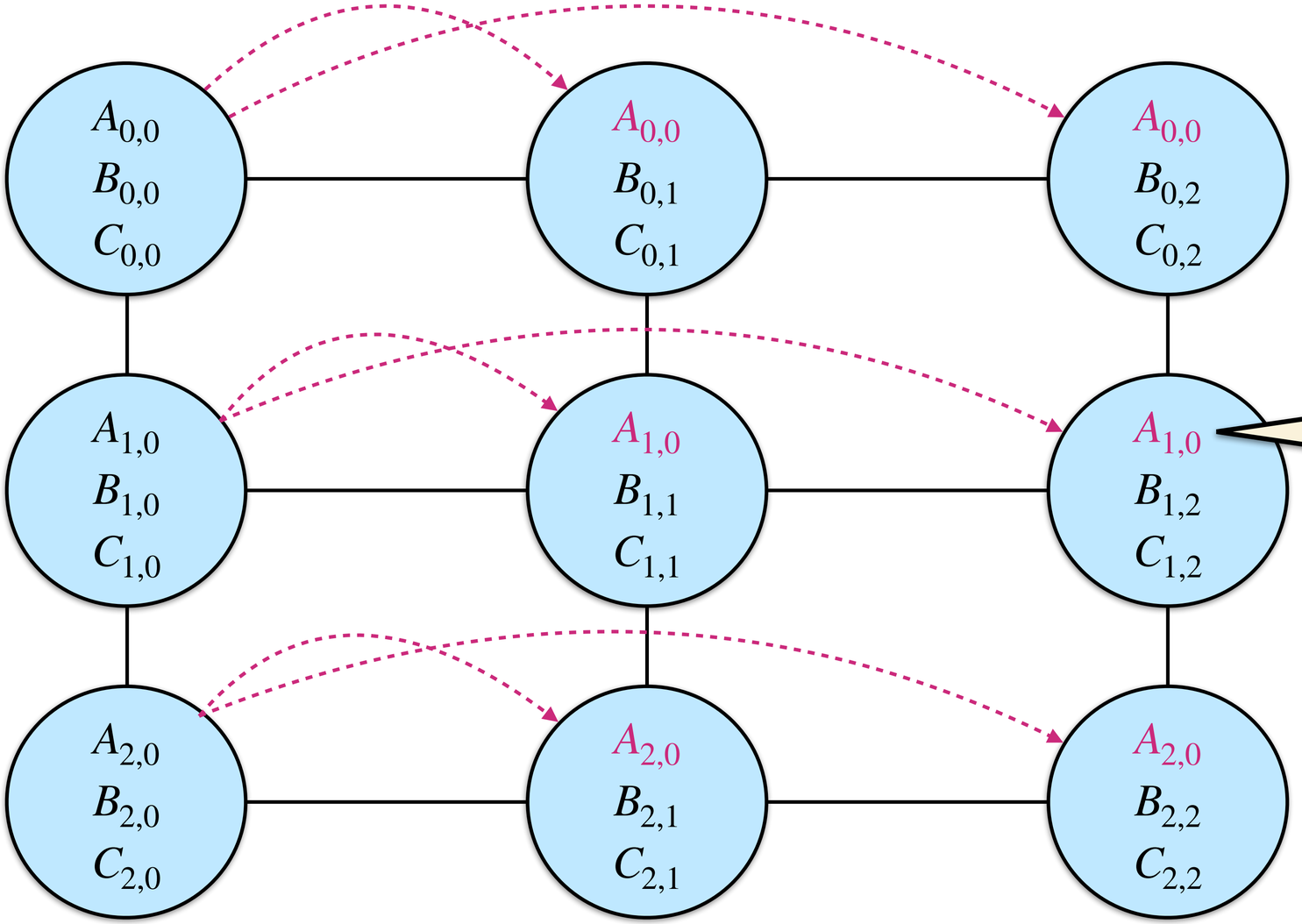
https://cseweb.ucsd.edu/classes/sp11/cse262-a/Lectures/262-pres1-hal.pdf

SUMMA Example: Initial Configuration



SUMMA Example: k = 0

Broadcast $A_{i,0}$ within processor row

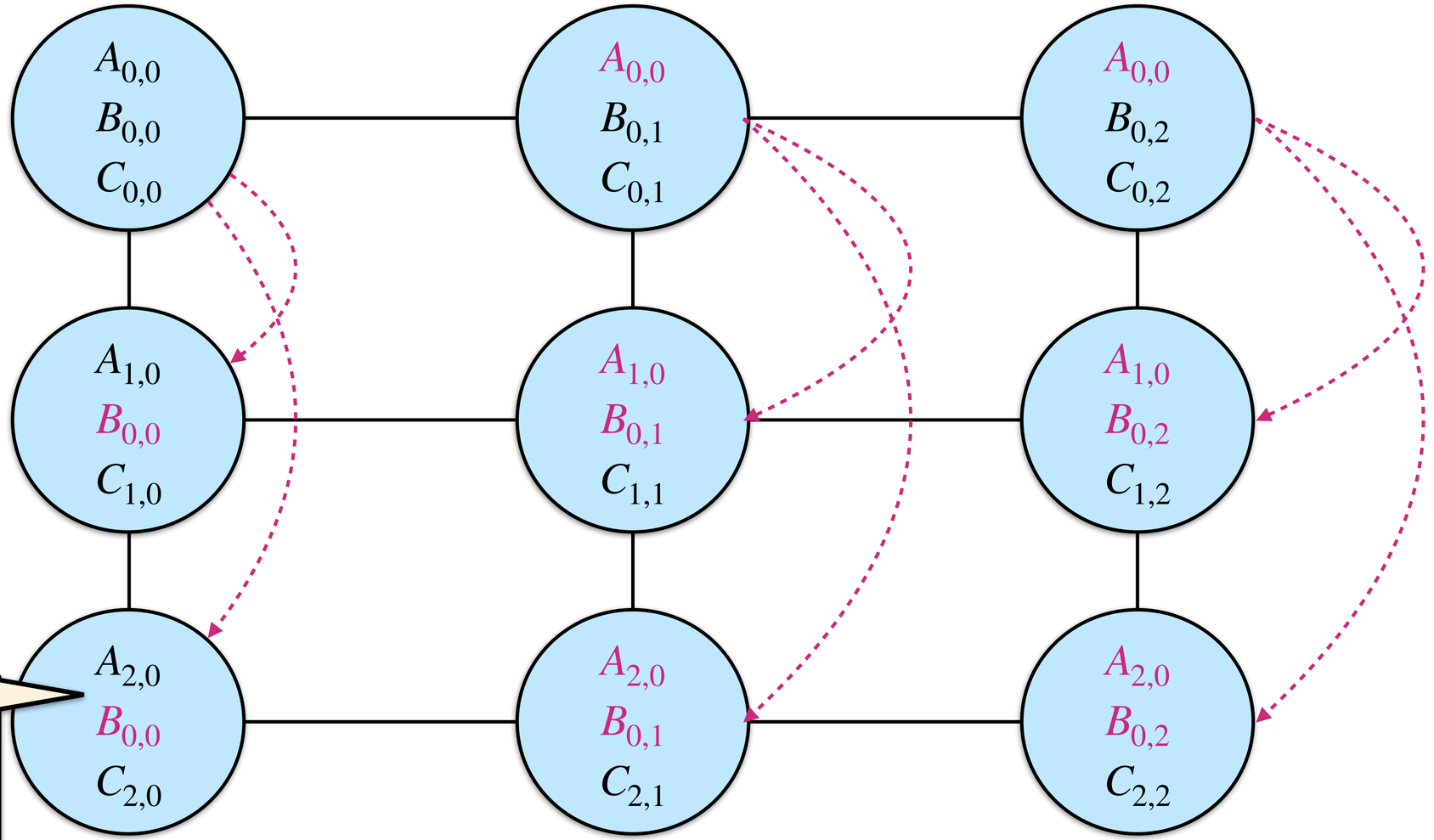


Inner dimension needs to match

keep the original data as well, just not pictured

SUMMA Example: k = 0

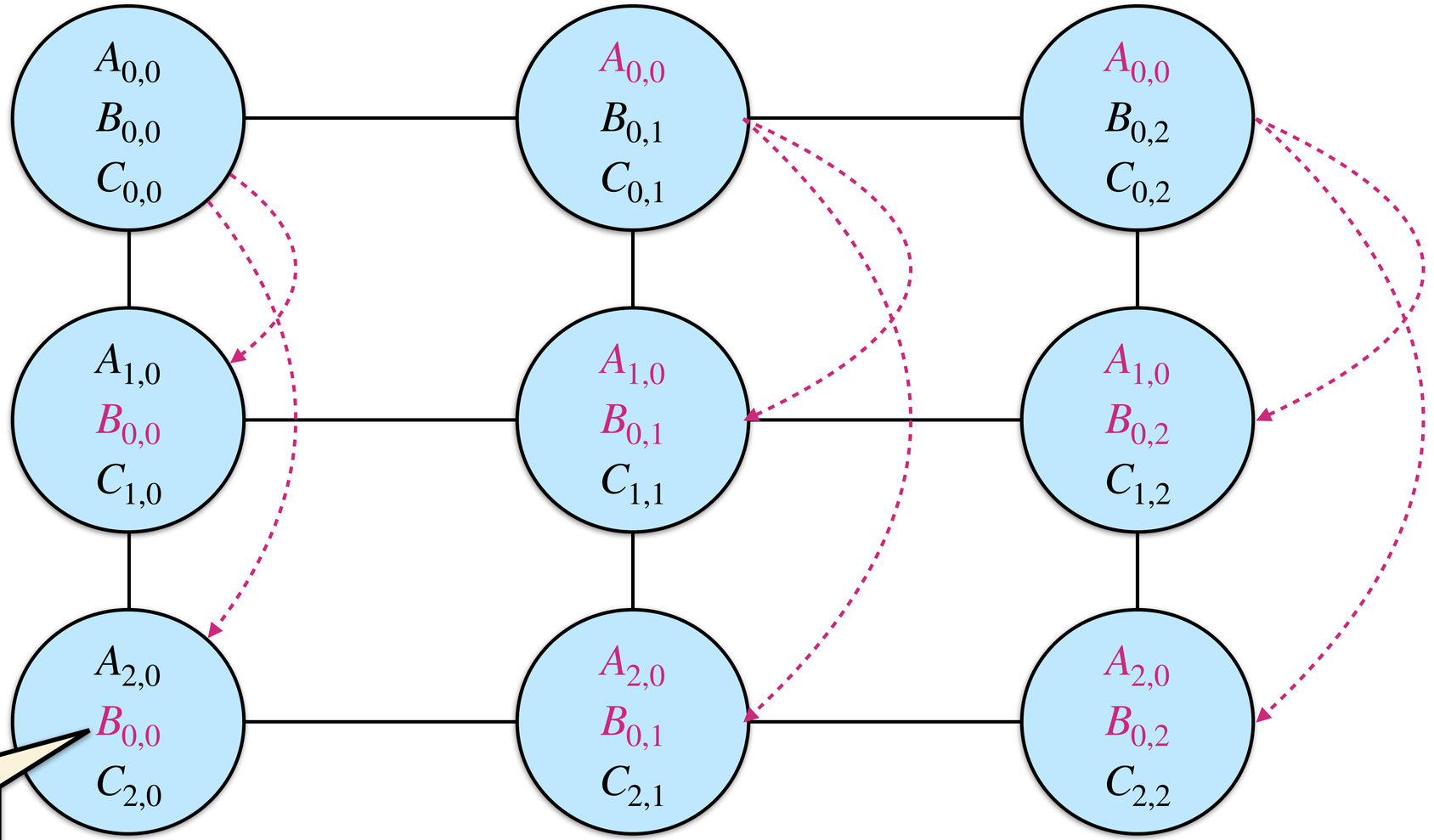
Broadcast $B_{0,j}$ within processor column



Inner dimension matches now: multiply and add to C

SUMMA Example: k = 0

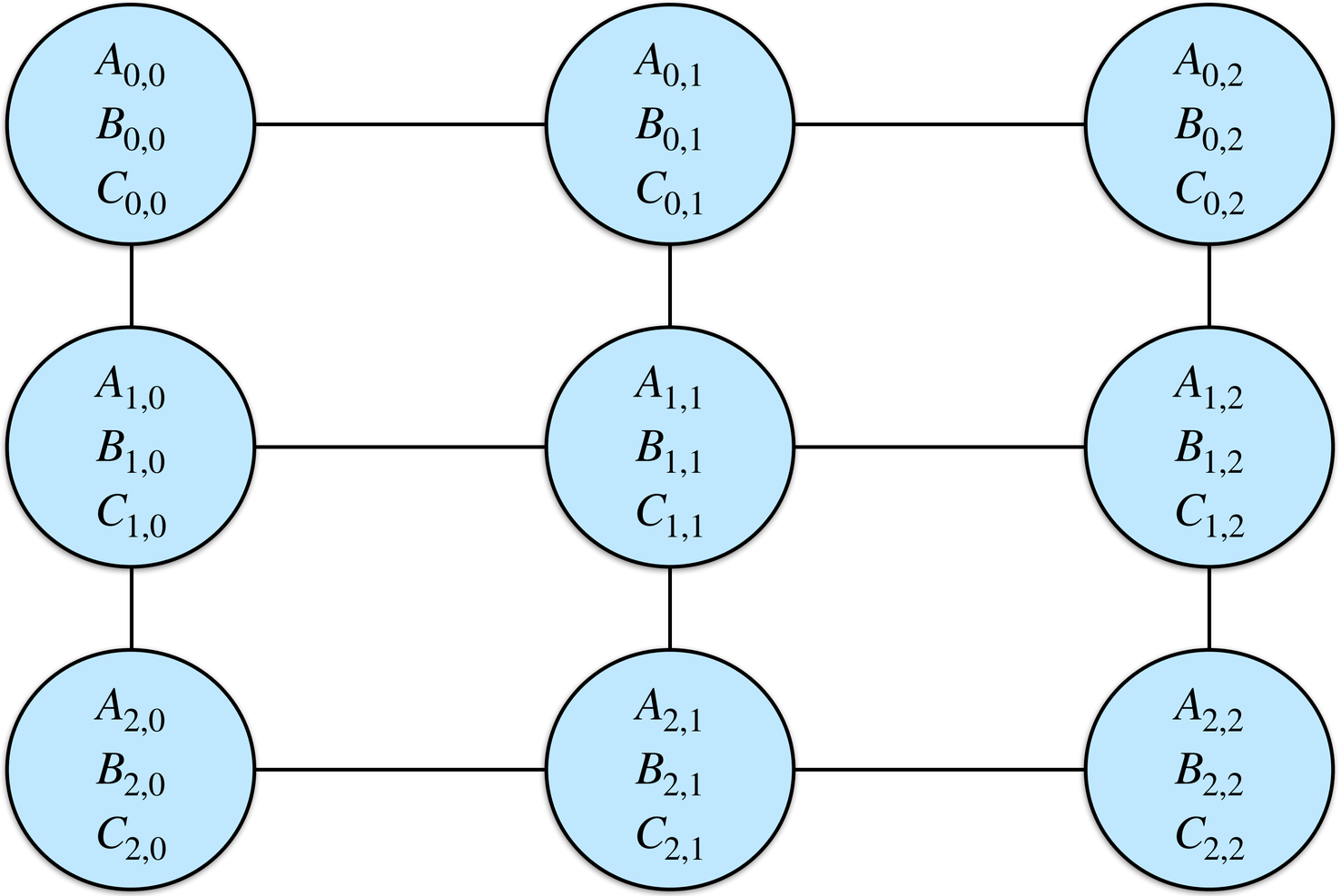
Broadcast $B_{0,j}$ within processor column



Inner dimension matches now

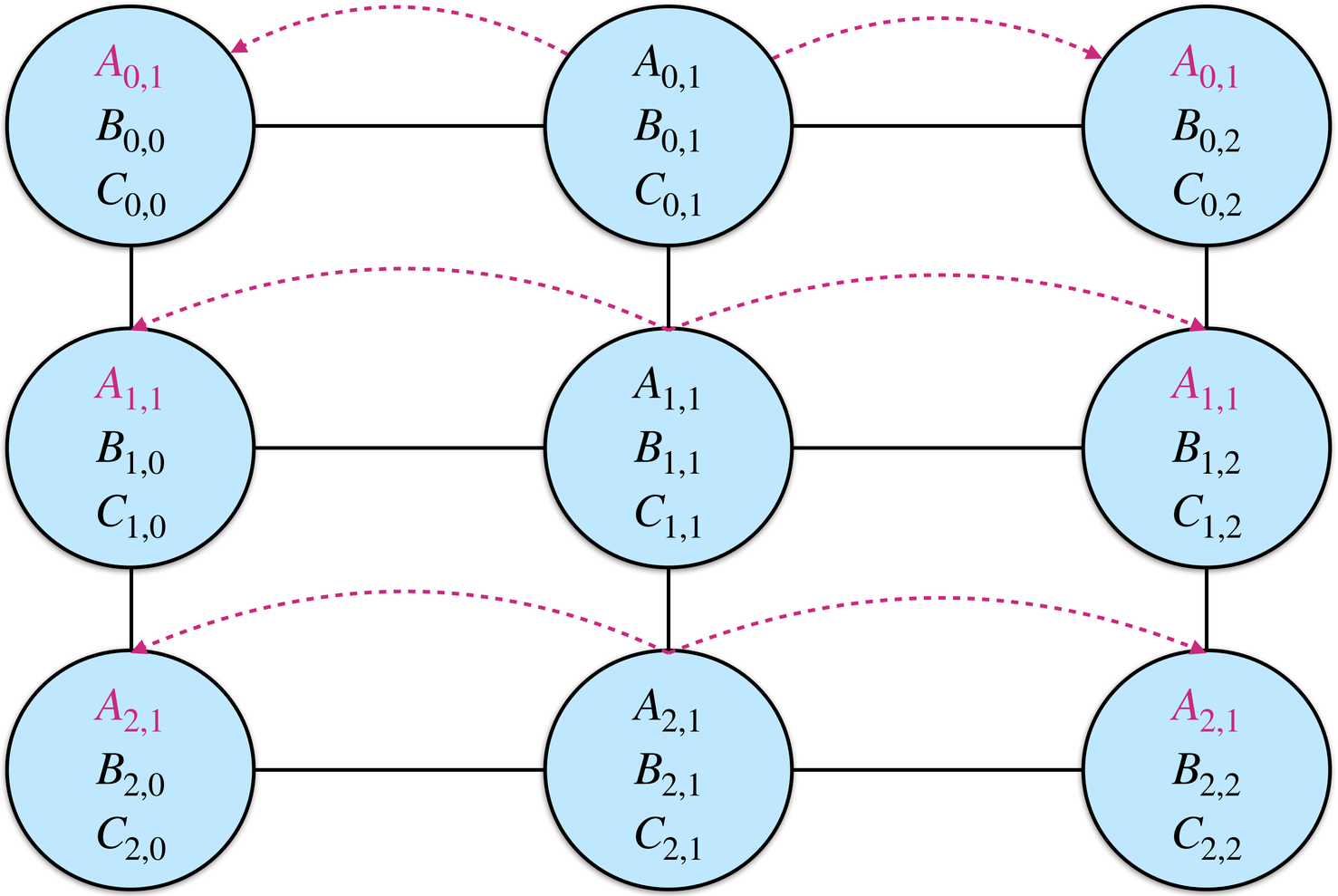
keep the original data as well, just not pictured

Next round: starting with initial configuration again



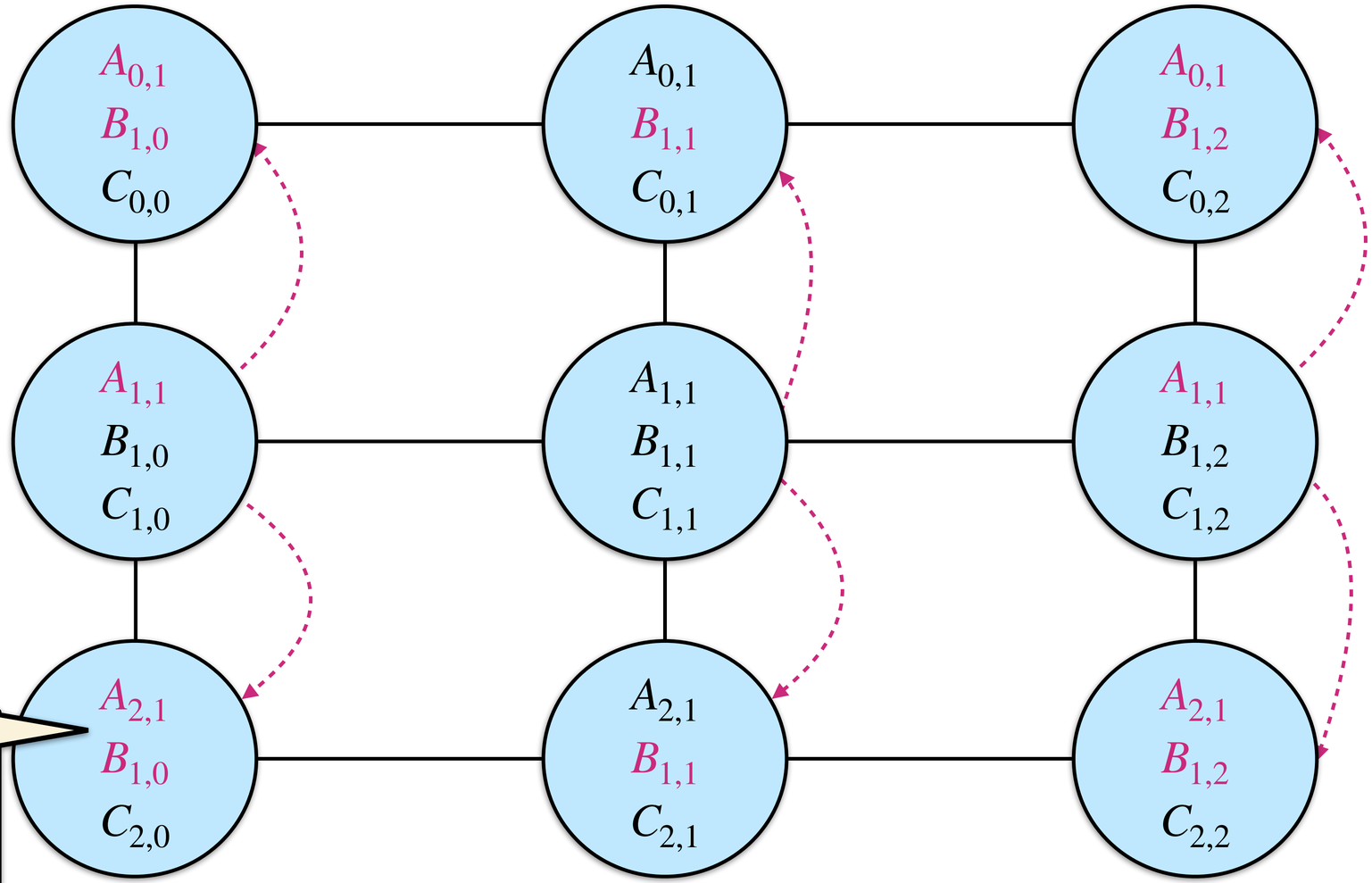
SUMMA Example: k = 1

Broadcast $A_{i,1}$ within processor row



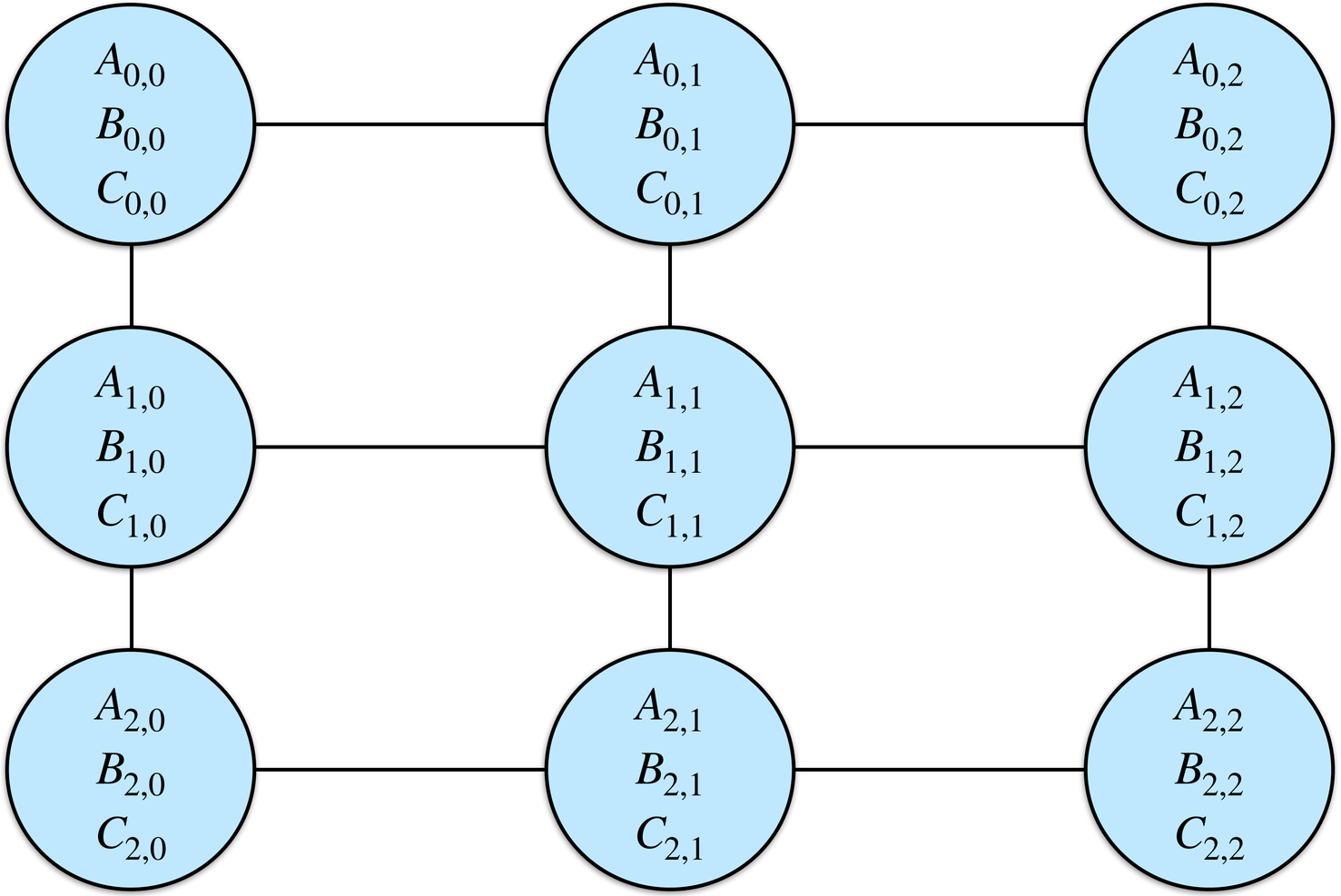
SUMMA Example: k = 1

Broadcast $B_{1,j}$ within processor column



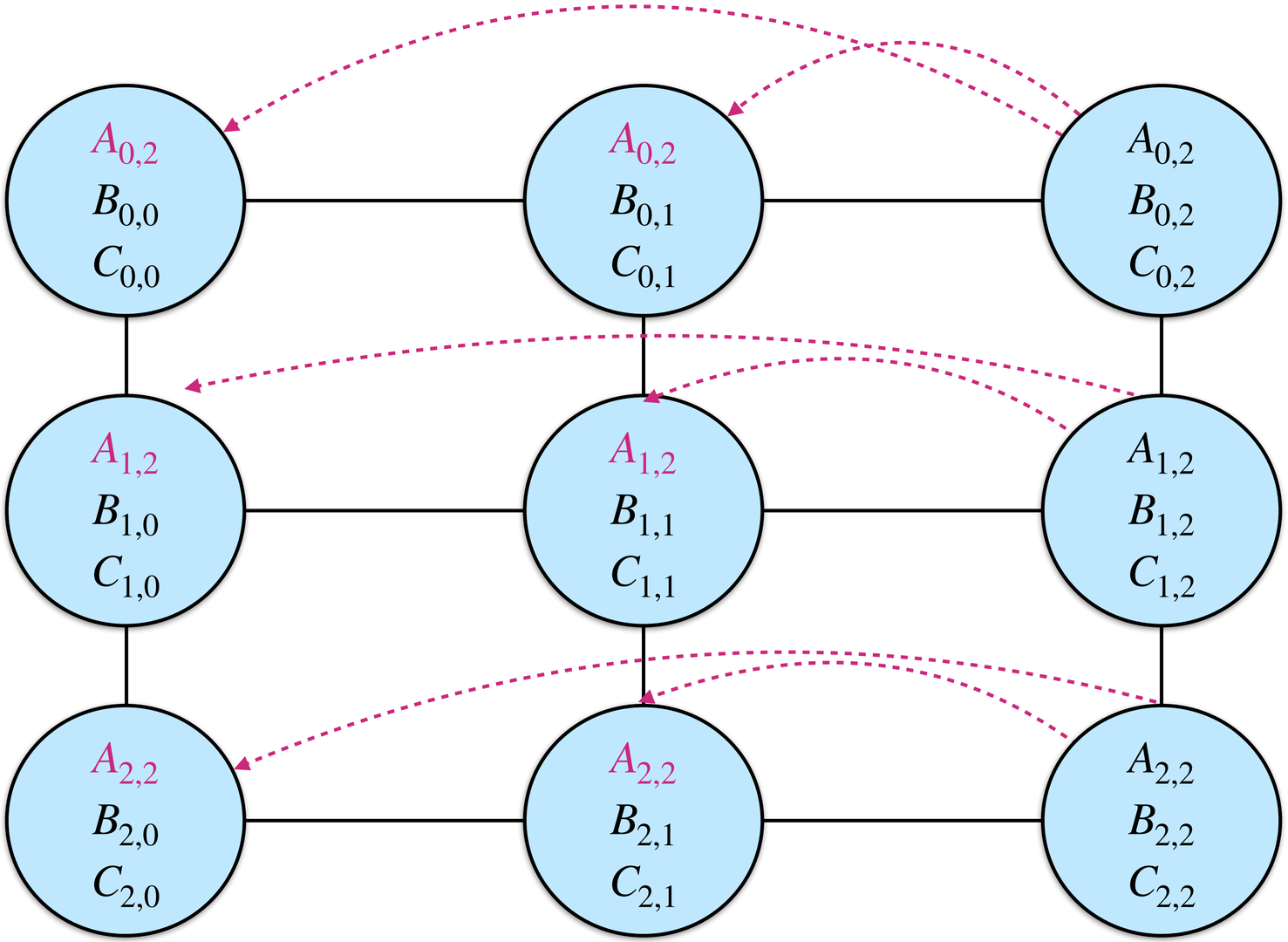
Inner dimension matches now: multiply and add to C

Next round: starting with initial configuration again



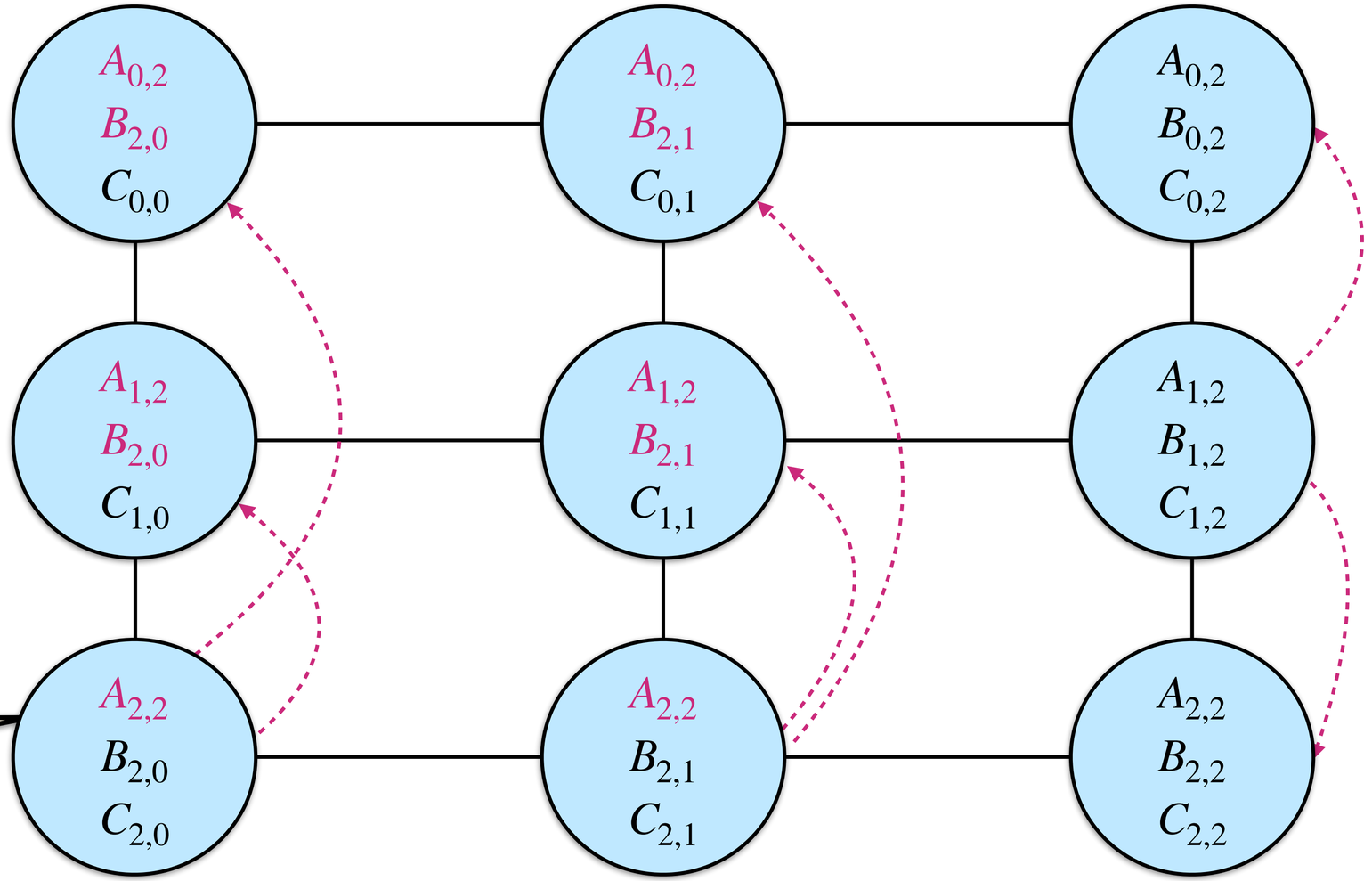
SUMMA Example: k = 2

Broadcast $A_{i,2}$ within processor row



SUMMA Example: k = 2

Broadcast $B_{2,j}$ within processor column



Inner dimension matches now: multiply and add to C

Blocking matrices in SUMMA

Suppose we have n x n square matrices A, B, C (for simplicity).

Rather than doing n iterations of the outer loop, pick some **block size b** (b columns of A and b rows of B) so that we can do n/b iterations of the outer loop.

```
for(int k = 0; k < n; k+=b) {
  C[:,] += A[:, k:k+b-1] + B[k:k+b-1, :]
}
```

Multiply n x b piece of A with b x n piece of B

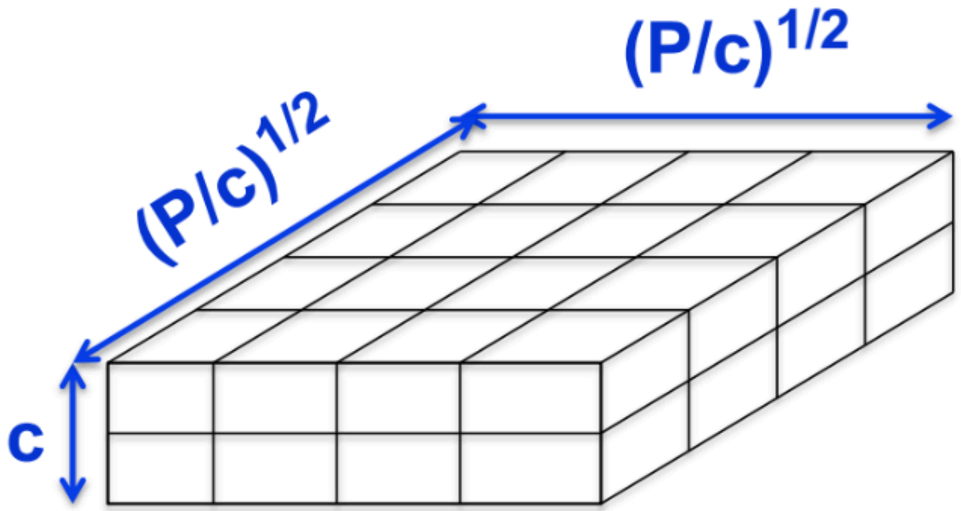
Communication cost

- The simplest choice of block size is $b = O(n/\sqrt{p})$ (assuming p processors in a square grid for simplicity).
- So each n x b submatrix is size $O(n^2/\sqrt{p})$.
- Each broadcast costs $O((\tau + \mu m)\lg(\sqrt{p})) = O((\tau + \mu n^2/\sqrt{p})(\lg p))$
- There are $n/b = O(\sqrt{p})$ rounds
- Total cost is (num rounds) x (cost of broadcast per round)

Substitute $m = n^2/\sqrt{p}$

2.5D Matrix Multiplication

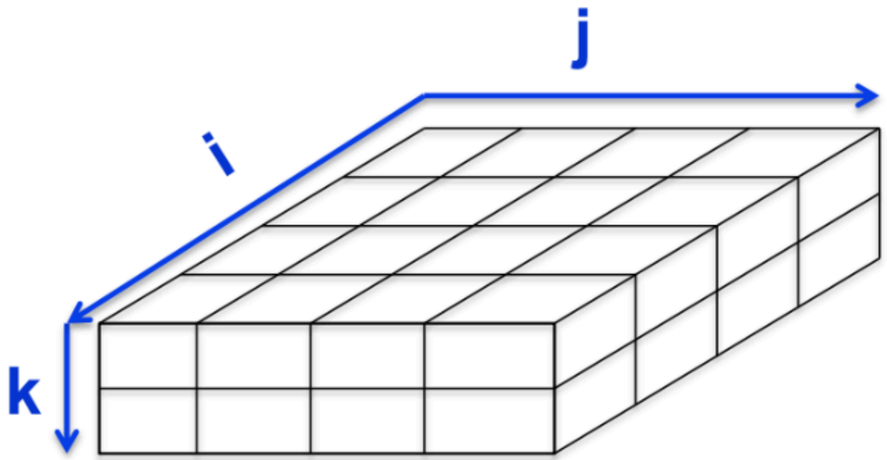
- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2}$ x $(P/c)^{1/2}$ x c grid



Example: P = 32, c = 2

2.5D Matrix Multiplication

- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2}$ x $(P/c)^{1/2}$ x c grid

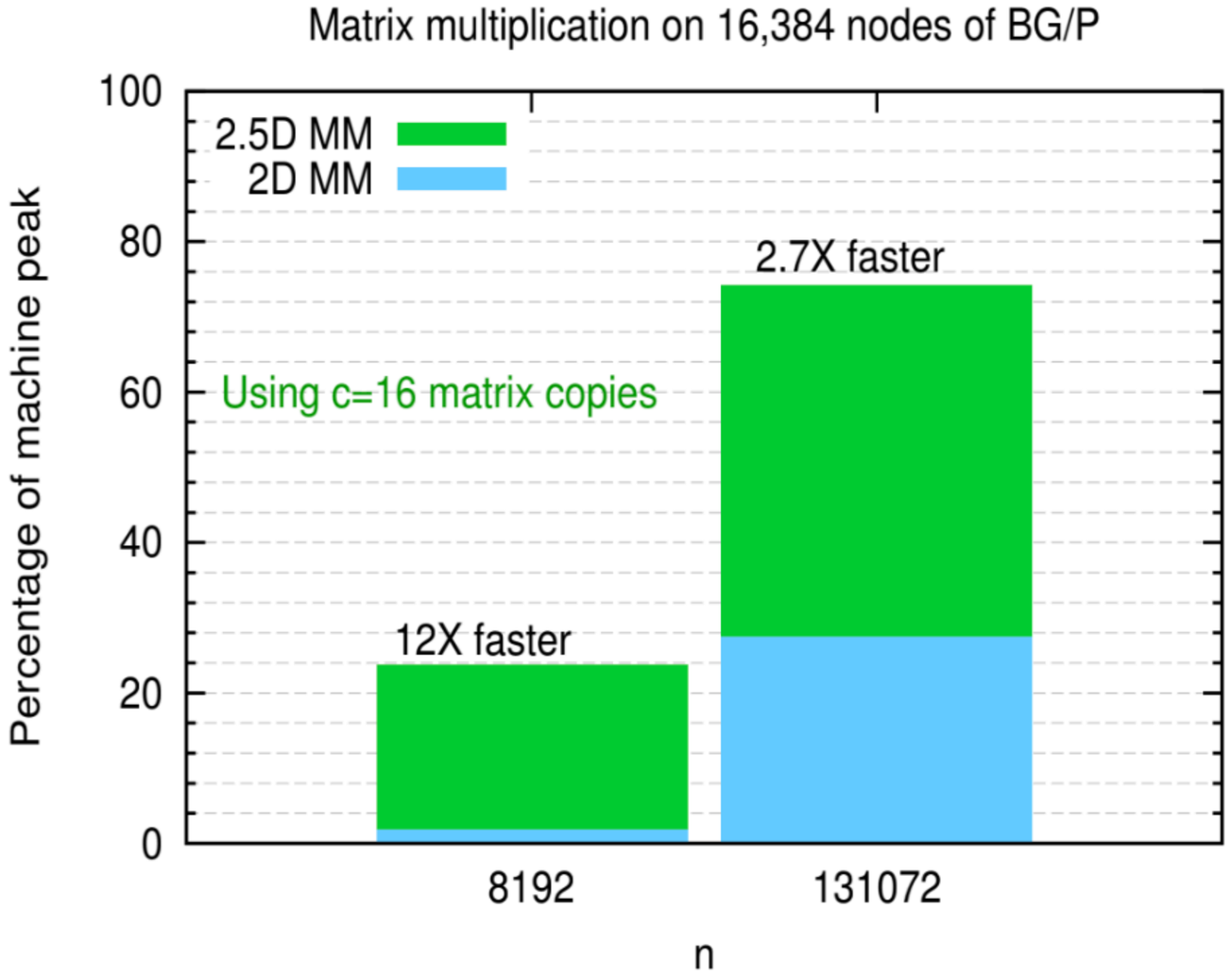


Initially P(i,j,0) owns A(i,j) and B(i,j)
each of size $n(c/P)^{1/2}$ x $n(c/P)^{1/2}$

(1) P(i,j,0) broadcasts A(i,j) and B(i,j) to P(i,j,k)
(2) Processors at level k perform 1/c-th of SUMMA, i.e. 1/c-th of Σ_m A(i,m)*B(m,j)
(3) Sum-reduce partial sums Σ_m A(i,m)*B(m,j) along k-axis so P(i,j,0) owns C(i,j)

From UC Berkeley CS267

2.5D Matrix Multiplication on IBM BG/P, 16k notes / 64k cores



"Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms" - Solomonik and Demmel, '11