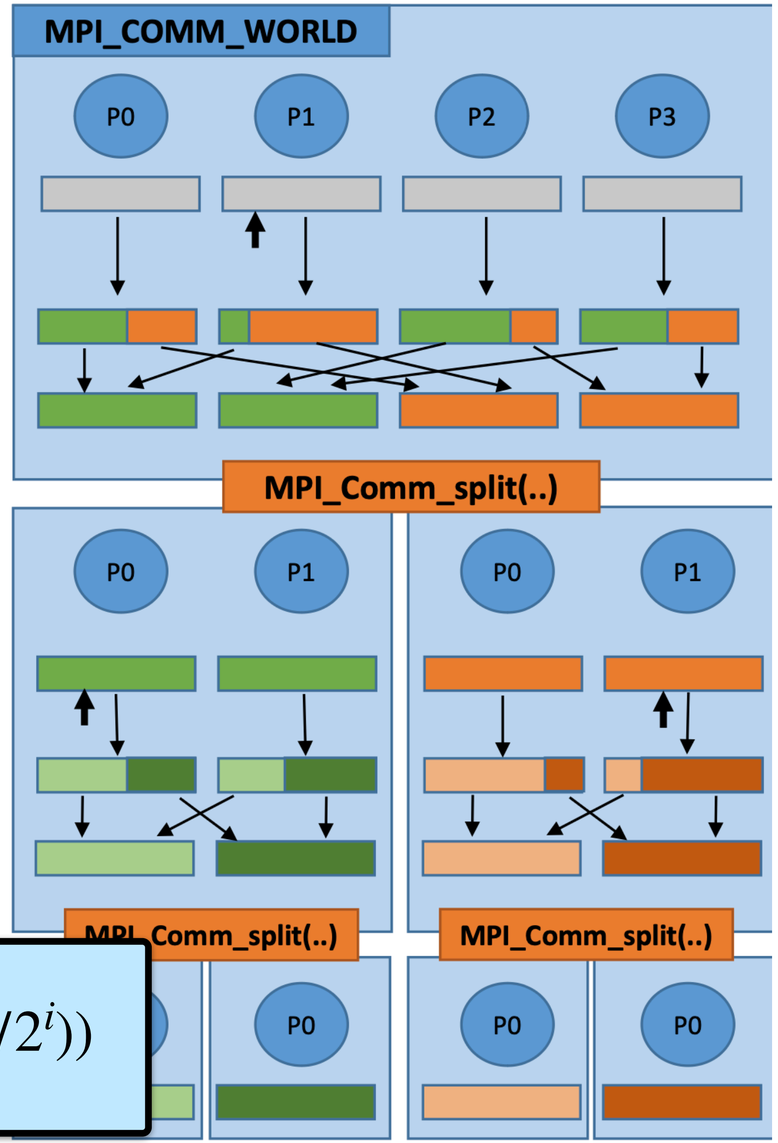
Update: Parallel Quicksort

Parallel Quicksort

1. Pick pivot and broadcast $O(\log p(\tau + \mu))$
2. Partition locally $O\left(\frac{n}{p}\right)$
3. Re-arrange partitions $O\left(\tau + \mu\frac{n}{p}\right)$
4. Split processors and recurse
5. If p = 1, locally sort



Best/expected case (assuming no imbalance):

- $O(\log p)$ iterations before local sort
- Computation: $O\left(\frac{n}{p}\log p + \frac{n}{p}\log\frac{n}{p}\right) = 0\left(\frac{n\log n}{p}\right)$
- Communication: $O\left(\tau \lg^2 p + \mu\frac{n}{p}\log p\right)$

Worst case:

- Much worse!

$$(\tau + \mu)\left(\sum_{i=0}^{\lg p} \lg(p/2^i)\right)$$

CSE 6220/CX 4220
Introduction to HPC

Lecture 14: Dense Matrix Algorithms

Helen Xu
hxu615@gatech.edu

Slide credits to Prof. Srinivas Aluru

Recap: Communication Primitives

- Broadcast
- Reduce
- AllReduce
- Scan

$\Theta(\tau \log p + \mu\, m \log p)$

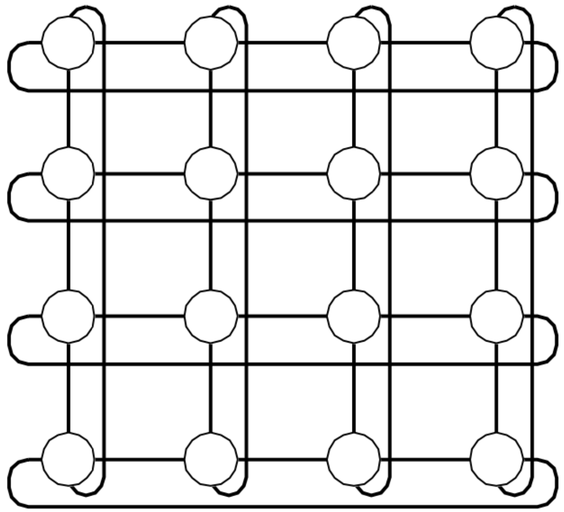
- Gather
- AllGather
- Scatter

$\Theta(\tau \log p + \mu\, m\, p)$

- All_to_All
  - Arbitrary Permutations: $O(\tau \cdot p + \mu \cdot m \cdot p)$
  - Hypercubic Permutations: $O(\tau \cdot \log p + \mu \cdot m \cdot p \cdot \log p)$

Matrix Algorithms

- Regular structure
- Well-suited to **static partitioning** of computation
- Computation typically partitioned based on the data
  - input, output, intermediate
- Typical data partitioning strategies for matrix algorithms
  - 1D and 2D block, cyclic, block-cyclic
- Easier to think logically about the algorithms on a 2D mesh of processors:



Matrix-Vector Multiplication

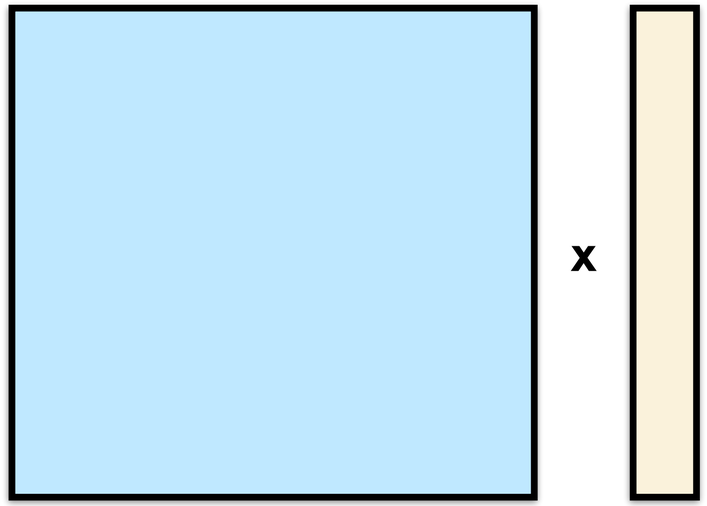
Matrix-Vector Multiplication

Problem:

- multiply a dense $n \times n$ matrix A with an $n \times 1$ vector x
- yield the $n \times 1$ result vector y

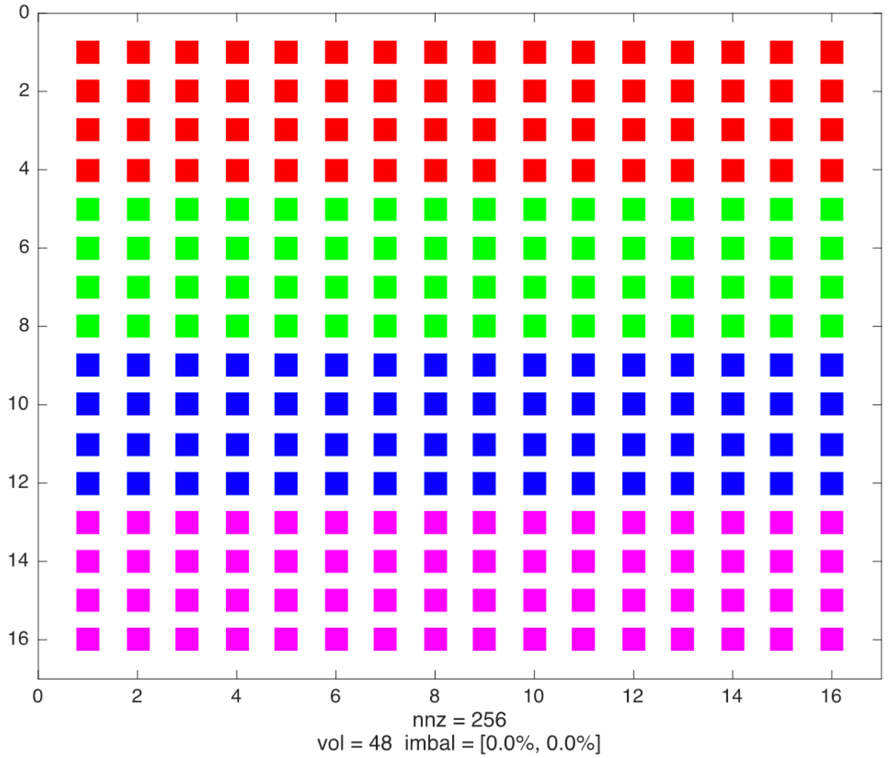
What is the serial time complexity?

- Serial algorithm requires n^2 multiplications and additions



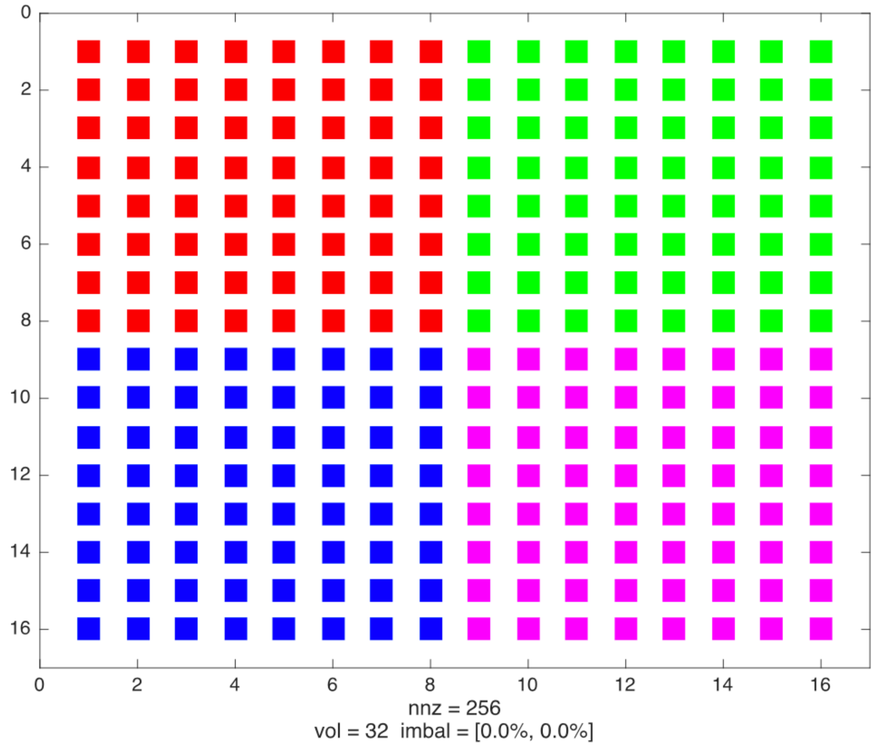
1D Partitioning for Matrix-Vector Multiply

- 1D partitioning splits up the matrix into **block rows** with at least one row on each processor.
- Given n rows, 1D partitioning can use at most n processors.



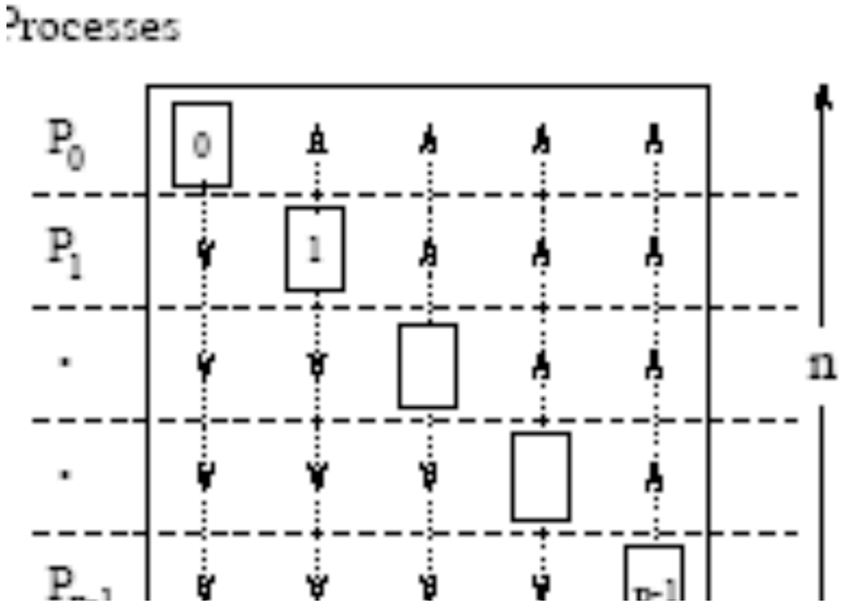
2D Partitioning for Matrix-Vector Multiply

- 2D partitioning splits up the matrix into **square sub-matrices** - at least 1 element per processor
- Theoretically, 2D partitioning can use up to n^2 processors (equal to number of elements)



Analysis of 1D Partitioning

- First, consider simple case: $p = n$
- n x n matrix is partitioned among p processors
  - *Each processor stores a complete row of the matrix*
- n x 1 vector x is also partitioned in a conformal manner
  - Each process owns one element of x



Analysis of 1D Partitioning

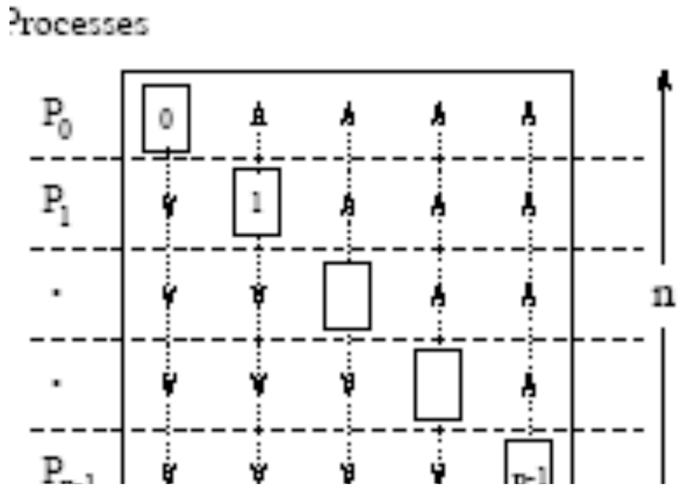
- Step 1: Use AllGather to distribute all of *x* to each processor

$$O(\tau \log n + \mu \cdot n)$$

$\mu n > n$, so time is basically the cost of allgather

- Step 2: Processor P_i computes $y[i] = \sum_{j=0}^{n-1} A[i,j] \cdot x[j]$

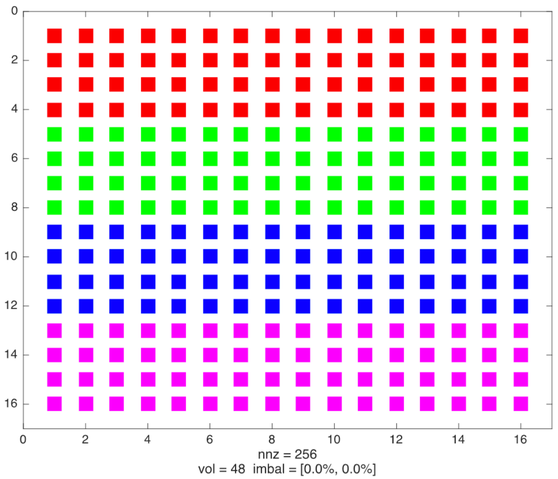
$$O(n)$$



Analysis of 1D Partitioning

For *n>p*

- Initially, each processor stores
  - n/p complete rows of the matrix *A*
  - n/p elements of the vector *x*
- Distribute all of *x* vector to each processor
  - AllGather among p processors, messages of size n/p
- On each processor: n/p local dot products on vectors (rows) of length n

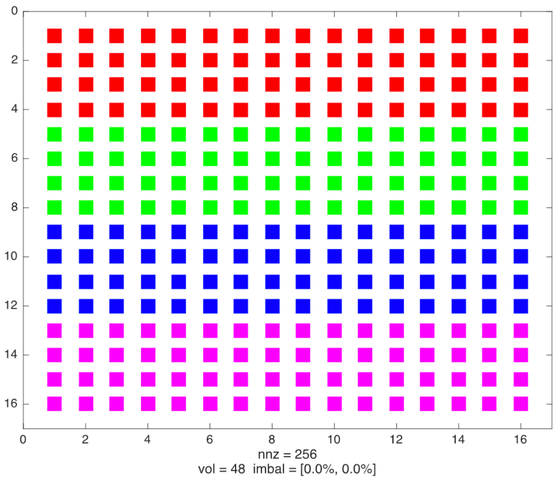


- Step 1: Comm. Time = **What is the communication time?**
- Step 2: Computation time = **What is the computation time?**
- Efficient up to $O(n)$ processors, the maximum the alg. can use.

Analysis of 1D Partitioning

For *n>p*

- Initially, each processor stores
  - n/p complete rows of the matrix *A*
  - n/p elements of the vector *x*
- Distribute all of *x* vector to each processor
  - AllGather among p processors, messages of size n/p
- On each processor: n/p local dot products on vectors (rows) of length n

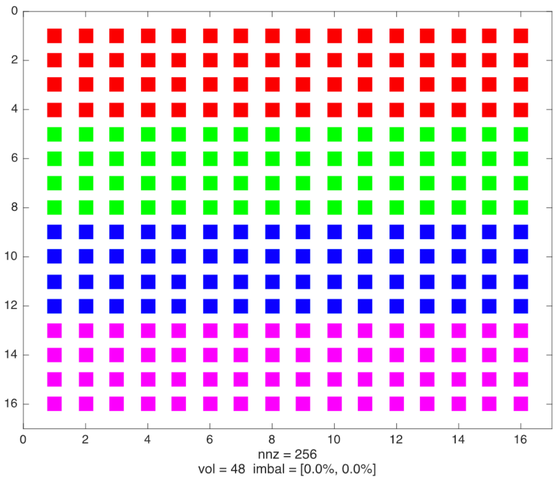


- Step 1: Comm. Time = $O\left(\tau \log p + \mu \cdot \frac{n}{p} \cdot p\right) = O(\tau \log p + \mu \cdot n)$
- Step 2: Computation time = **What is the computation time?**
- Efficient up to $O(n)$ processors, the maximum the alg. can use.

Analysis of 1D Partitioning

For *n>p*

- Initially, each processor stores
  - n/p complete rows of the matrix *A*
  - n/p elements of the vector *x*
- Distribute all of *x* vector to each processor
  - AllGather among p processors, messages of size n/p
- On each processor: n/p local dot products on vectors (rows) of length n



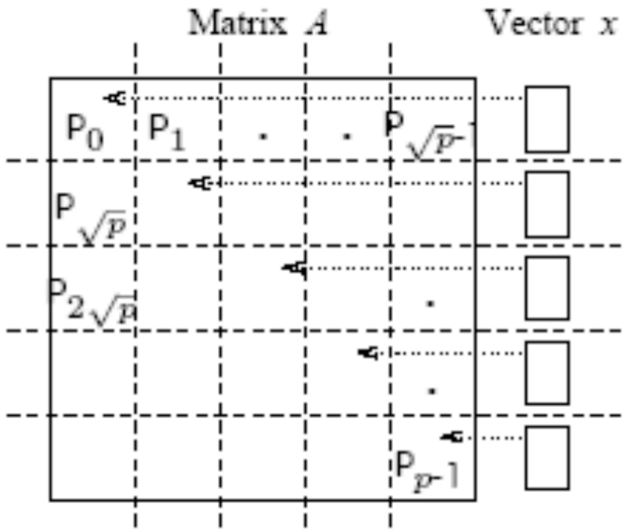
- Step 1: Comm. Time = $O\left(\tau \log p + \mu \cdot \frac{n}{p} \cdot p\right) = O(\tau \log p + \mu \cdot n)$
- Step 2: Computation time = $O\left(\frac{n^2}{p}\right)$
- Efficient up to $O(n)$ processors, the n

Can only use up to n/μ processors because computation time needs to dominate for efficiency

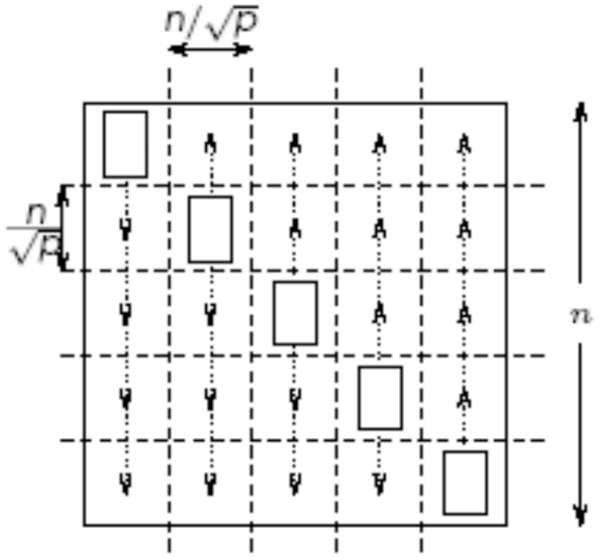
Analysis of 2D Partitioning

- Consider : $p = n^2$
- *n* x *n* matrix is partitioned among $p = n^2$ processors (virtual grid topology)
  - *Each processor owns a single element*
- *n* x 1 vector *x* is distributed in the last column of *n* processors

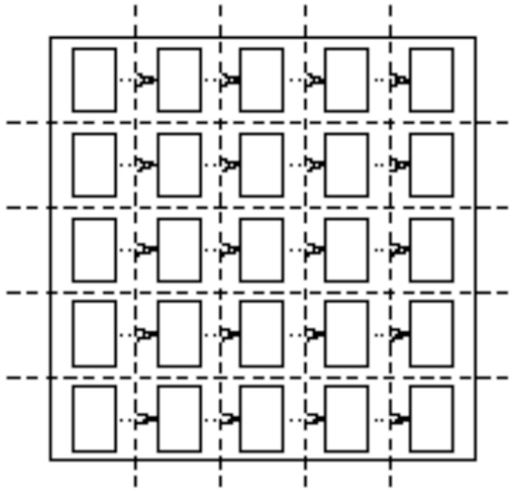
What is a good algorithm given this distribution?



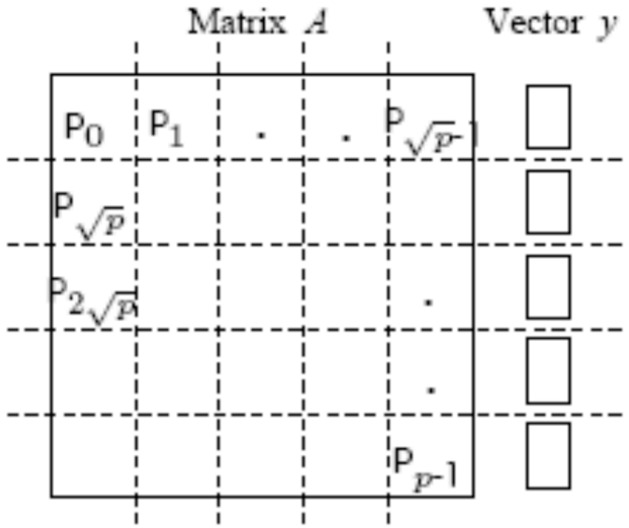
(a) Initial data distribution and communication steps to align the vector along the diagonal



(b) One-to-all broadcast of portions of the vector along process columns



(c) All-to-one reduction of partial results



(d) Final distribution of the result vector

Analysis of 2D Partitioning

$p = n^2$

- Algorithm
  - align vector along the main diagonal: one-to-one communication — $(\tau + \mu)\lg n$
  - Broadcast vector element to n processors in column: one-to-all broadcast — $(\tau + \mu)\lg n$
  - Local multiplication — 1
  - Sum partial y values in each row: all-to-one reduction — $(\tau + \mu)\lg n + \lg n$

Analysis of 2D Partitioning

$p = n^2$

- Algorithm
  - align vector along the main diagonal: one-to-one communication — $(\tau + \mu)\lg n$
  - Broadcast vector element to *n* processors in column: one-to-all broadcast — $(\tau + \mu)\lg n$
  - Local multiplication — 1
  - Sum partial *y* values in each row: all-to-one reduction — $(\tau + \mu)\lg n + \lg n$
- Communication time = $O(\tau \log n + \mu \cdot \log n)$
- Computation time = $O(\log n)$ — From reduction

Is the algorithm efficient?

Analysis of 2D Partitioning

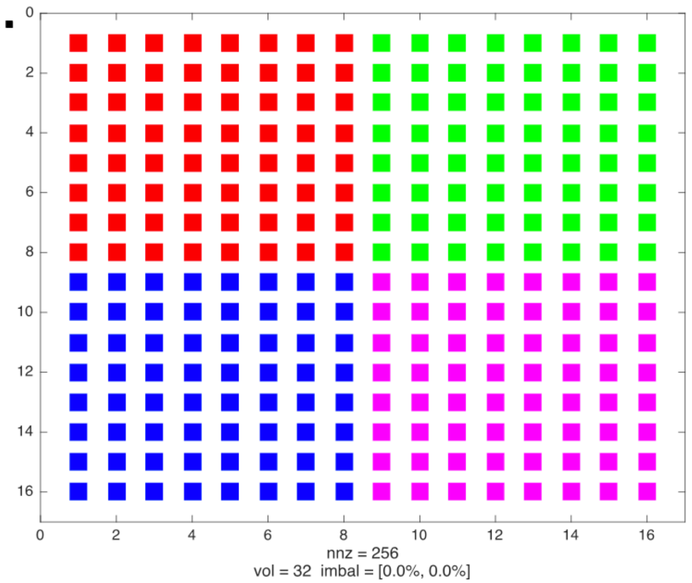
$p = n^2$

- Algorithm
  - align vector along the main diagonal: one-to-one communication — $(\tau + \mu)\lg n$
  - Broadcast vector element to n processors in column: one-to-all broadcast — $(\tau + \mu)\lg n$
  - Local multiplication — 1
  - Sum partial y values in each row: all-to-one reduction — $(\tau + \mu)\lg n + \lg n$
- Communication time = $O(\tau \log n + \mu \cdot \log n)$
- Computation time = $O(\log n)$ — From reduction
- Is the algorithm efficient?
  - No – we need $O(1)$ run-time!

2D Partitioning for Matrix-Vector Multiply

$p < n^2$

- When using fewer than n^2 processors, each process owns an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block of the matrix.
- The vector is distributed in portions of $\frac{n}{\sqrt{p}}$ elements in the last processor-column only.

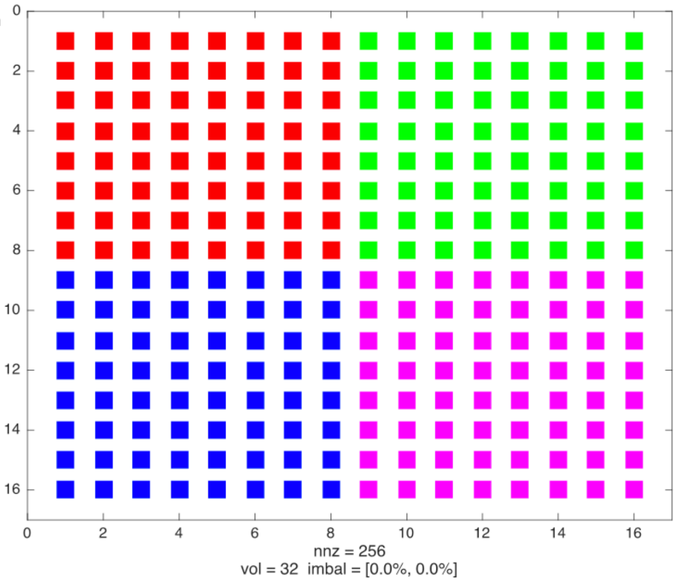


2D Partitioning for Matrix-Vector Multiply

$p < n^2$

- When using fewer than n^2 processors, each process owns an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block of the matrix.
- The vector is distributed in portions of $\frac{n}{\sqrt{p}}$ elements in the last processor-column only.
- In this case, the message sizes for the alignment, broadcast, and reduction are all $\frac{n}{\sqrt{p}}$.

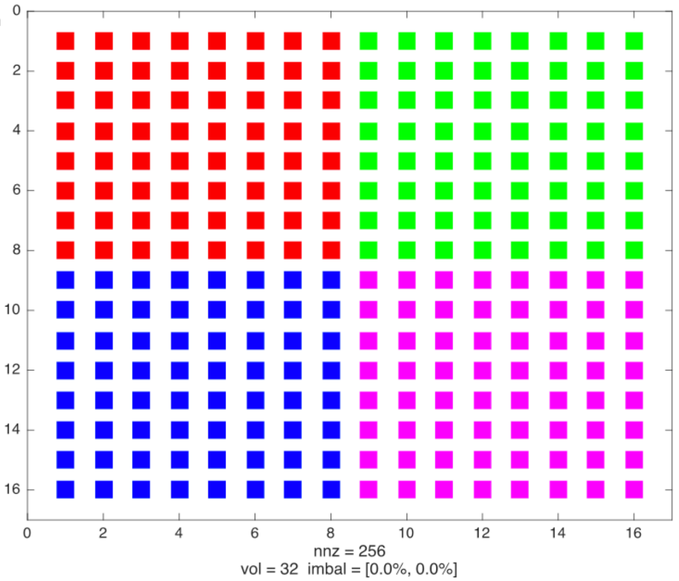
Part of the vector



2D Partitioning for Matrix-Vector Multiply

$p < n^2$

- When using fewer than n^2 processors, each process owns an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block of the matrix.
- The vector is distributed in portions of $\frac{n}{\sqrt{p}}$ elements in the last processor-column only.
- In this case, the message sizes for the alignment, broadcast, and reduction are all $\frac{n}{\sqrt{p}}$.
- The computation is a product of an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ submatrix with a vector of length $\frac{n}{\sqrt{p}}$.



Run serial algorithm on the submatrix on each processor

2D Partitioning for Matrix-Vector Multiply

$p < n^2$

- Algorithm
  - align vector along the main diagonal: $O\left(\tau + \mu \cdot \frac{n}{\sqrt{p}}\right)$
  - Broadcast vector elements to processors in column: $O\left(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}} \log \sqrt{p}\right)$
  - Local multiplication: $O\left(\frac{n^2}{p}\right)$
  - Sum partial y values in each row: $O\left(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}} \log \sqrt{p}\right)$

2D Partitioning for Matrix-Vector Multiply

$p < n^2$

- Algorithm
  - align vector along the main diagonal: $O\left(\tau + \mu \cdot \frac{n}{\sqrt{p}}\right)$
  - Broadcast vector elements to processors in column: $O\left(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}} \log \sqrt{p}\right)$
  - Local multiplication: $O\left(\frac{n^2}{p}\right)$
  - Sum partial y values in each row: $O\left(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}} \log \sqrt{p}\right)$
- Communication time = $O\left(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}} \log \sqrt{p}\right)$
- Computation time = $O\left(\frac{n^2}{p}\right)$
- Efficient up to $O\left(\frac{n^2}{\log^2 n}\right)$ processors.

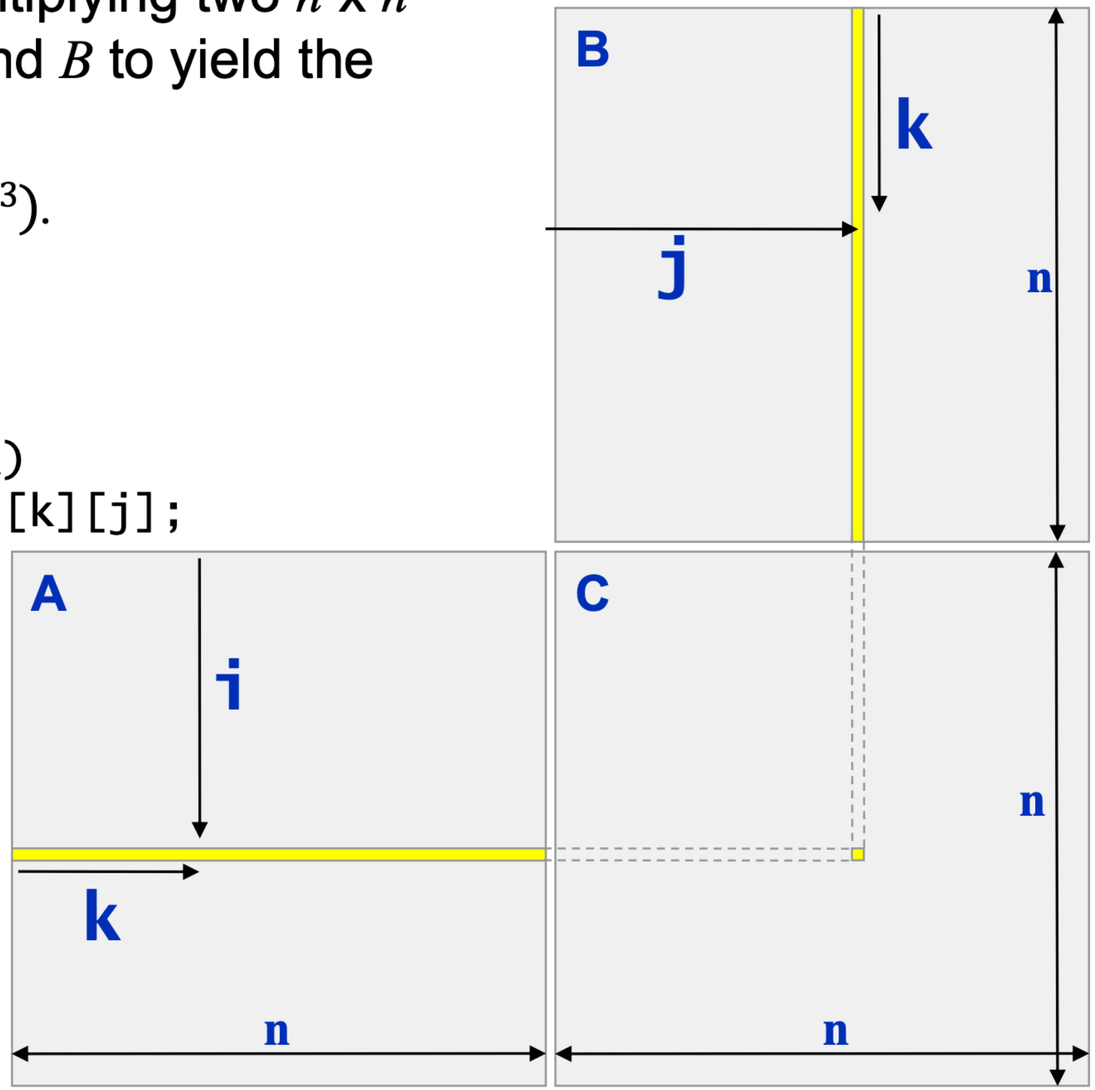
More scalable than 1D on the hypercube

Matrix-Matrix Multiplication

Matrix-Matrix Multiplication

- Consider the problem of multiplying two n x n dense, square matrices A and B to yield the product matrix $C = A$ x B.
- The serial complexity is $O(n^3)$.

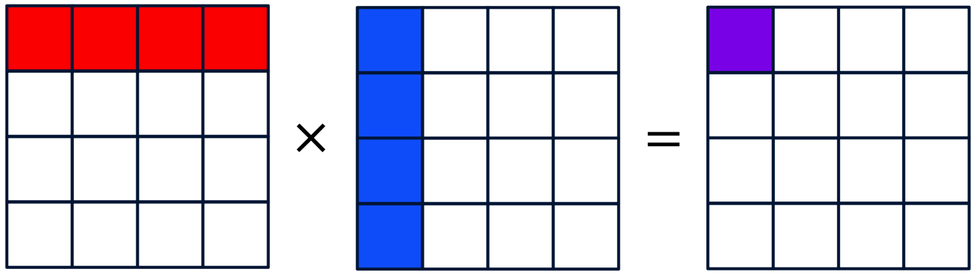
```
for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j) {
        double sum = 0;
        for (k = 0; k < n; ++k)
            sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```



Initial Algorithm for Matrix-Matrix Multiplication

Assume we have n^2 processors, where every processor (i, j) has elements A[i, j] and B[i, j] in the initial distribution.

For processor (i, j) to compute C[i, j], it needs the i-th row of A and j-th col of B.

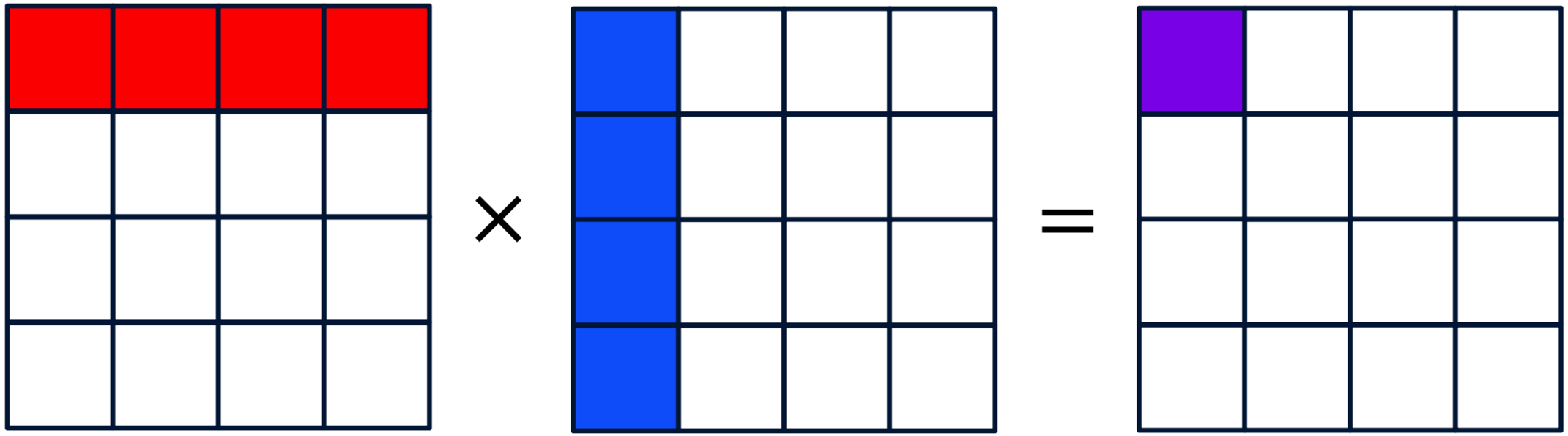


Idea:

- Allgather i-th row of A onto each processor for each row i
- Allgather j-th col of B onto each processor for each col j
- Perform dot product of row and col to get C[i, j]

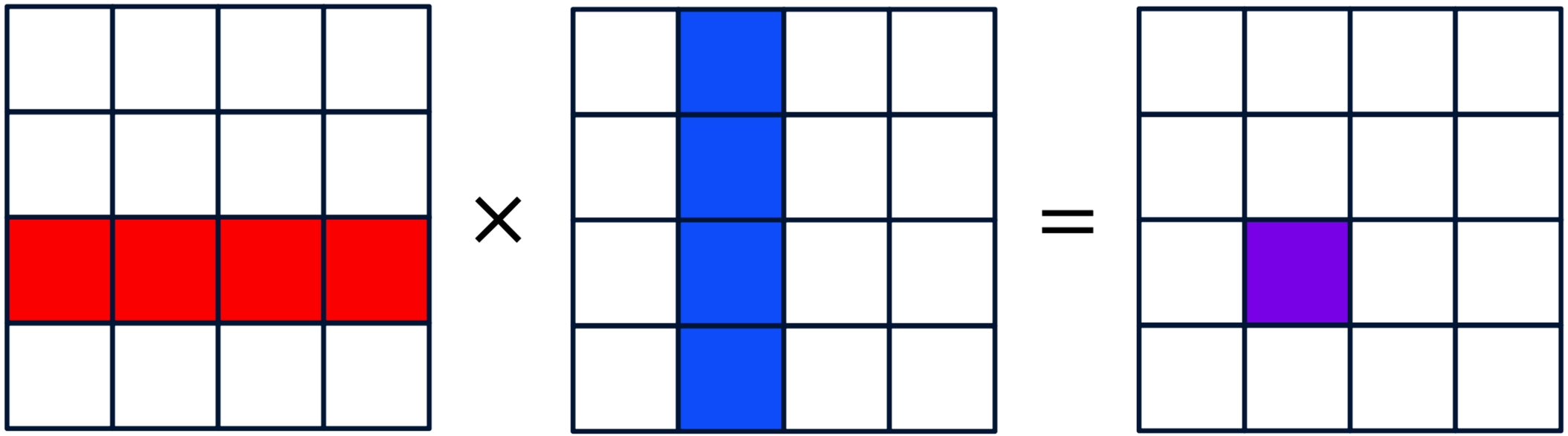
Data Requirements for Matrix-Matrix Multiply

Consider data needed for output matrix block shown in purple



Data Requirements for Matrix-Matrix Multiply

Consider data needed for output matrix block shown in purple



Matrix-Matrix Multiplication

- Consider the problem of multiplying two $n \times n$ dense, square matrices A and B to yield the product matrix $C = A \times B$.
- The serial complexity is $O(n^3)$.
  - We do not consider better serial algorithms (Strassen's method)

    Note: these can still be used as serial kernels in the parallel algorithms.

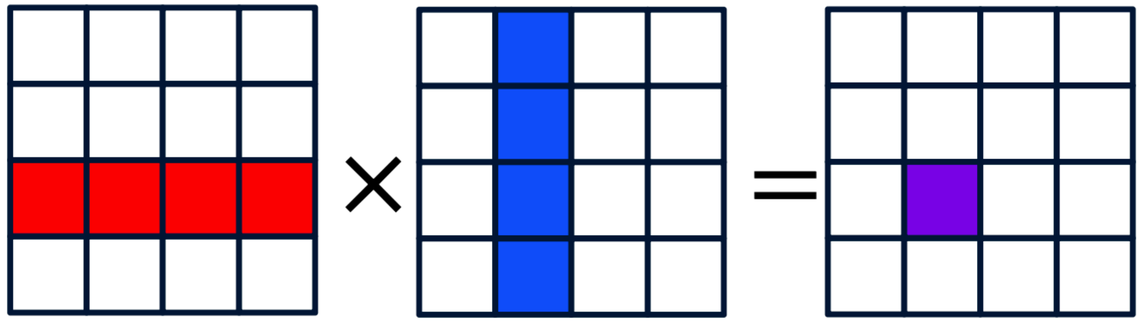
Matrix-Matrix Multiplication

- Consider the problem of multiplying two $n \times n$ dense, square matrices A and B to yield the product matrix $C = A \times B$.
- The serial complexity is $O(n^3)$.
  - We do not consider better serial algorithms (Strassen's method)

    Note: these can still be used as serial kernels in the parallel algorithms.
- A useful concept in this case is called *block* operations
  - an $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks
  - $A_{i,j}$ $(0 \leq i, j < q)$ such that each block is an $(n/q) \times (n/q)$ submatrix.
- In this view, we perform q^3 matrix multiplications, each involving $(n/q) \times (n/q)$ matrices.

Blocked Matrix-Matrix Multiply

- Consider two $n \times n$ matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ $(0 \leq i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix.
- Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$.
  - AllGather blocks of A along rows and B along columns.
- Perform local submatrix multiplication.



Analysis of Blocked Matrix-Matrix Multiply

- The two AllGather operations take time:
  - $O(\tau \log \sqrt{p} + \mu \frac{n^2}{p} \sqrt{p})$

Analysis of Blocked Matrix-Matrix Multiply

- The two AllGather operations take time:
  - $O(\tau \log \sqrt{p} + \mu \frac{n^2}{p} \sqrt{p})$
- The computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices:
  - $O\left(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\right) = O\left(\frac{n^3}{p}\right)$

Analysis of Blocked Matrix-Matrix Multiply

- The two AllGather operations take time:
  - $O(\tau \log \sqrt{p} + \mu \frac{n^2}{p} \sqrt{p})$
- The computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices:
  - $O\left(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\right) = O\left(\frac{n^3}{p}\right)$
- Total Time: $O\left(\frac{n^3}{p} + \tau \log p + \mu \frac{n^2}{\sqrt{p}}\right)$
- Efficient for $p = O(n^2)$

Analysis of Blocked Matrix-Matrix Multiply

- The two AllGather operations take time:
  - $O(\tau \log \sqrt{p} + \mu \frac{n^2}{p} \sqrt{p})$
- The computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices:
  - $O\left(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\right) = O\left(\frac{n^3}{p}\right)$
- Total Time: $O\left(\frac{n^3}{p} + \tau \log p + \mu \frac{n^2}{\sqrt{p}}\right)$
- Efficient for $p = O(n^2)$

Any issues with this algorithm?

Analysis of Blocked Matrix-Matrix Multiply

- The two AllGather operations take time:
  - $O(\tau \log \sqrt{p} + \mu \frac{n^2}{p} \sqrt{p})$
- The computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices:
  - $O\left(\sqrt{p} \times \left(\frac{n}{\sqrt{p}}\right)^3\right) = O\left(\frac{n^3}{p}\right)$
- Total Time: $O\left(\frac{n^3}{p} + \tau \log p + \mu \frac{n^2}{\sqrt{p}}\right)$
- Efficient for $p = O(n^2)$
- Major drawback of the algorithm is that it is not memory optimal.

Extra $\sqrt{p}$ factor of memory consumption!

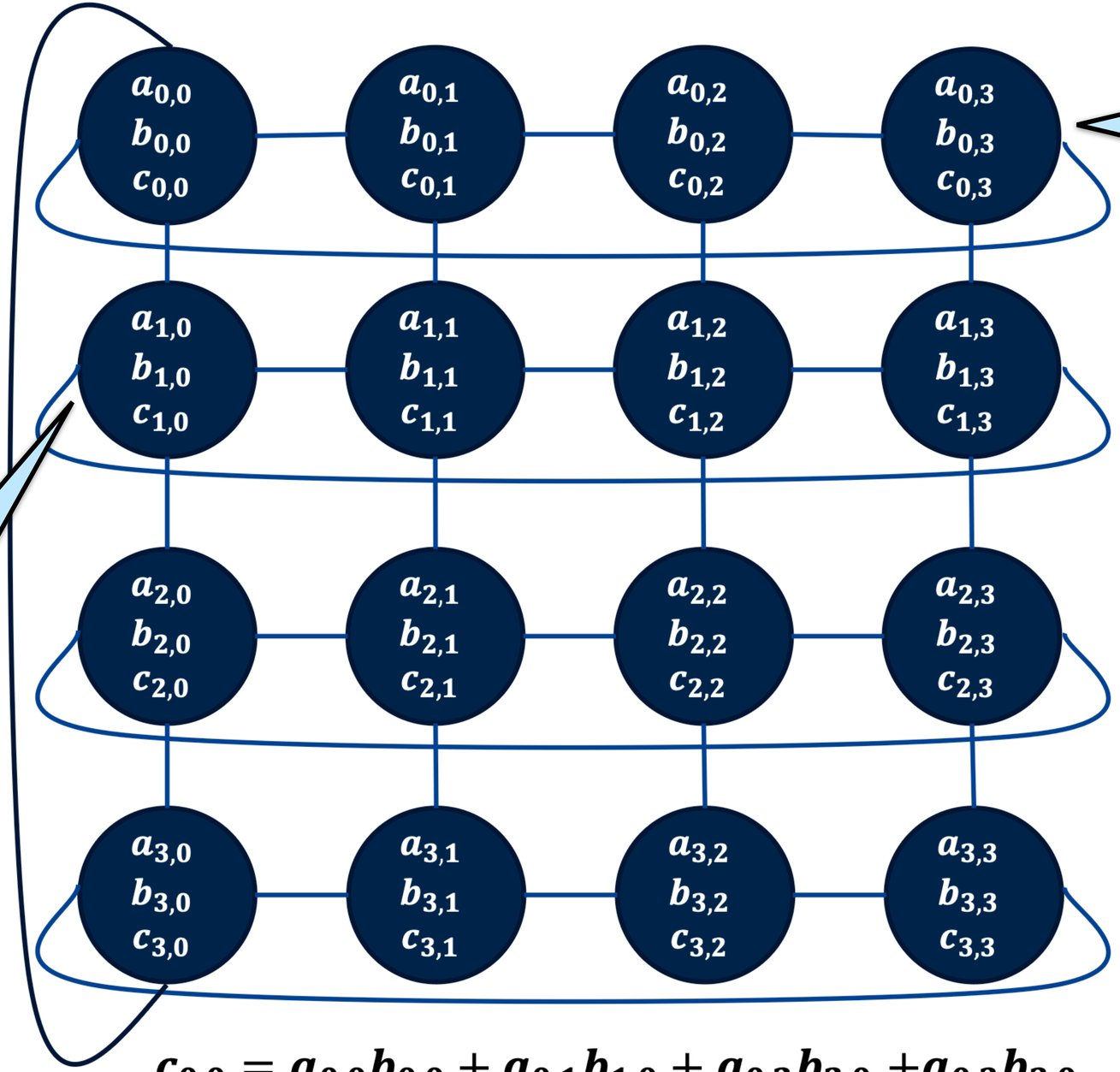
Ideal: Memory is serial_memory / p

START PART 2 HERE

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

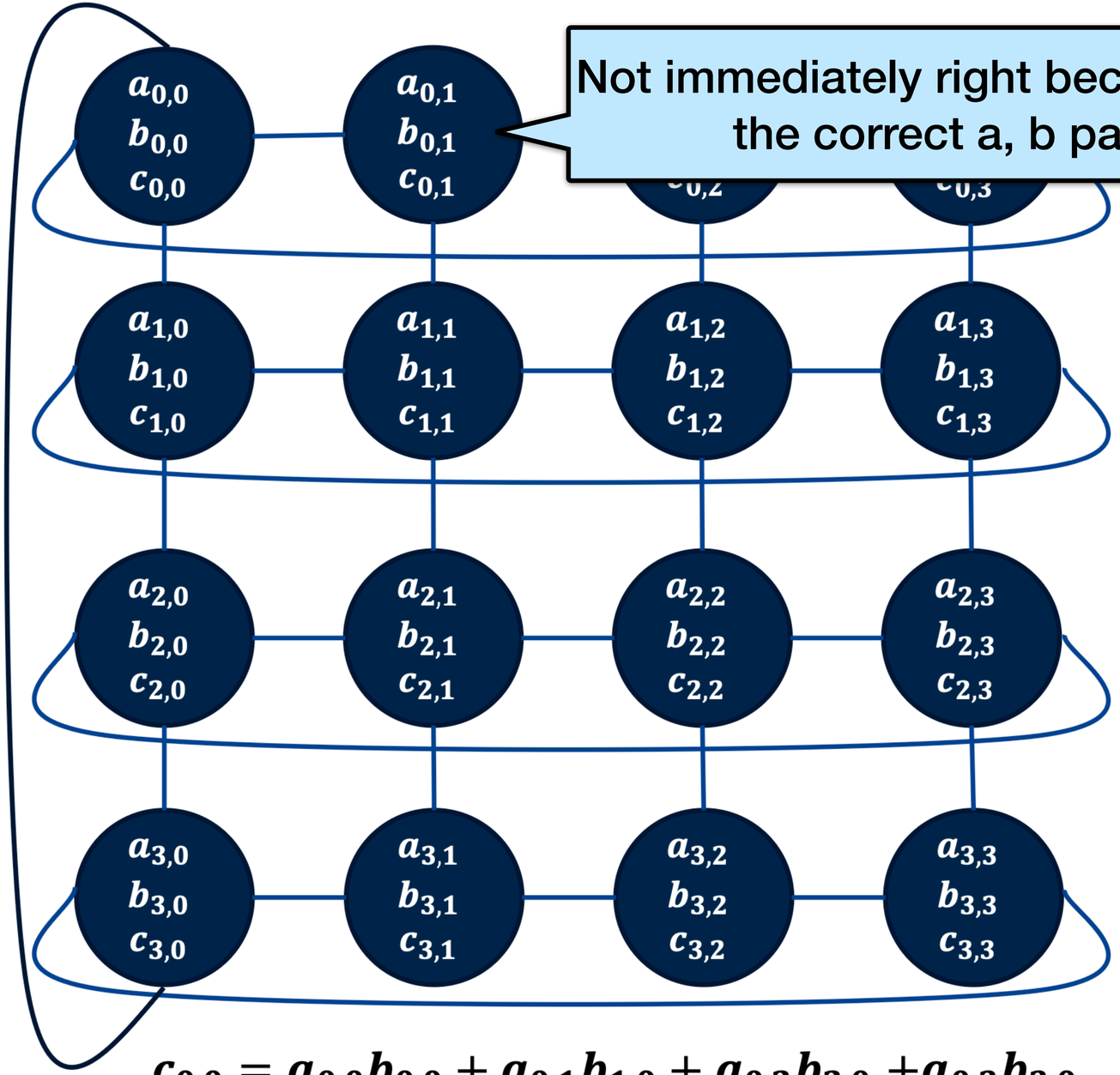


$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0}$$

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

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

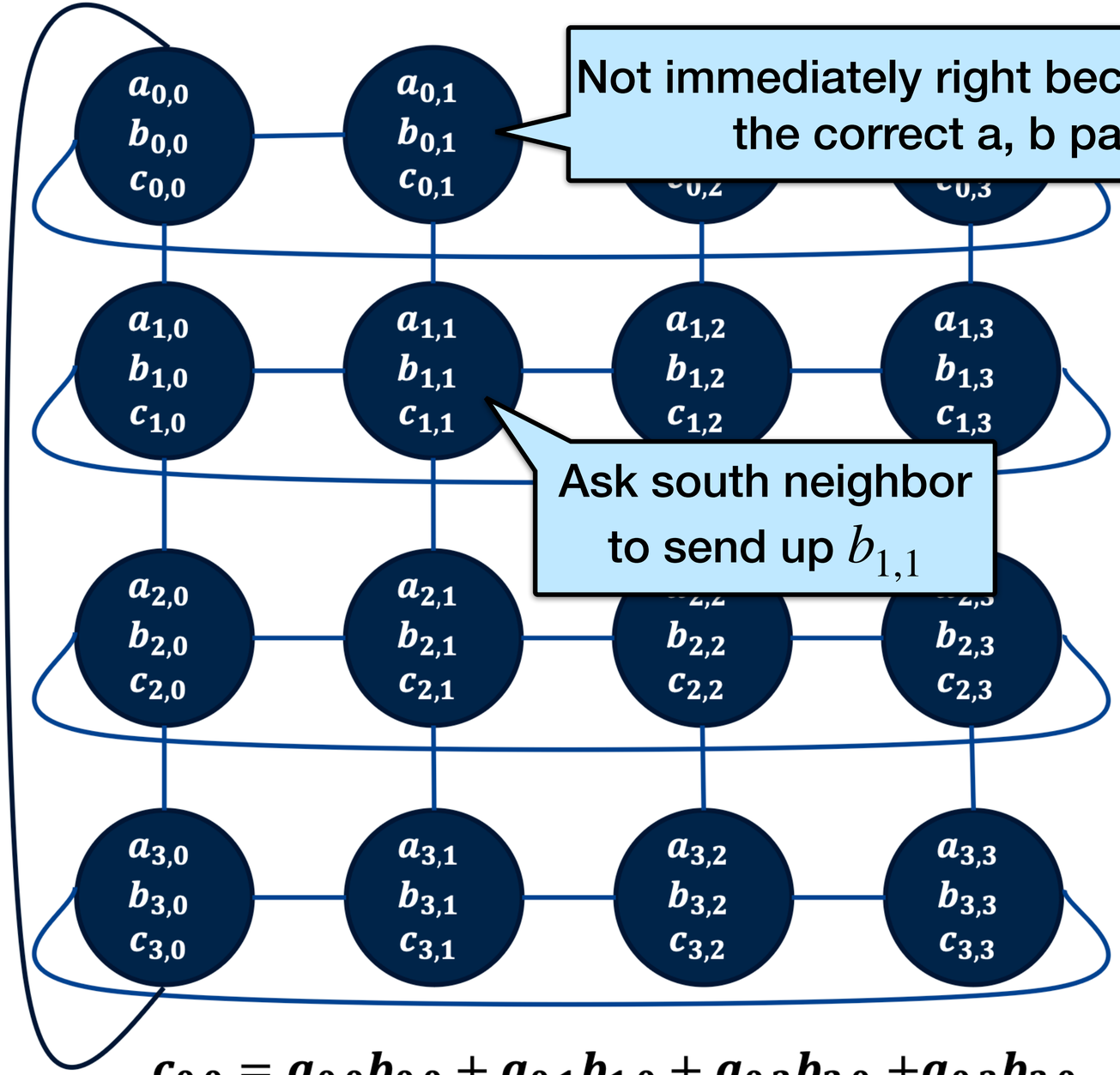
$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} + a_{0,3}b_{3,0}$$

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

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



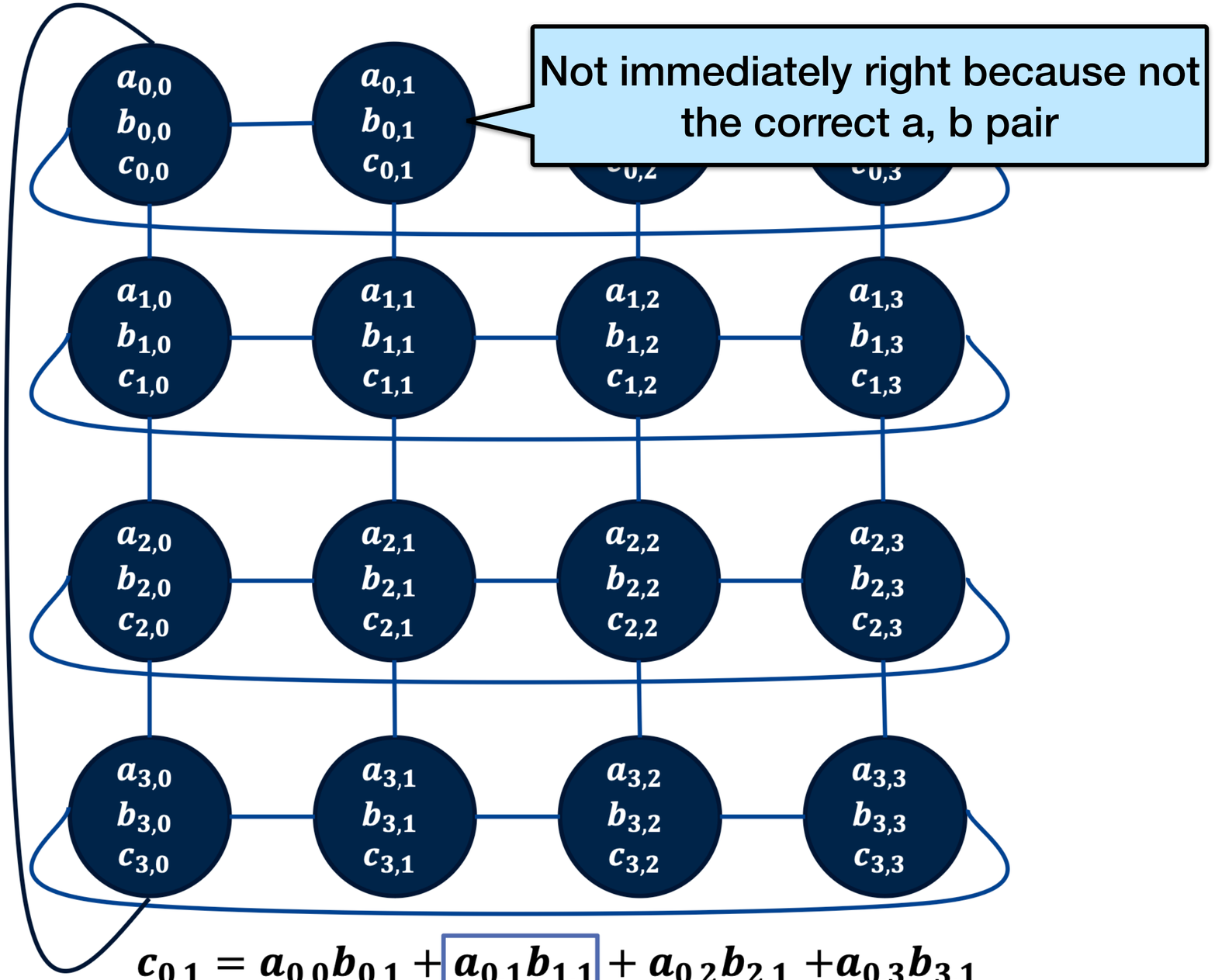
Not immediately right because not the correct a, b pair

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

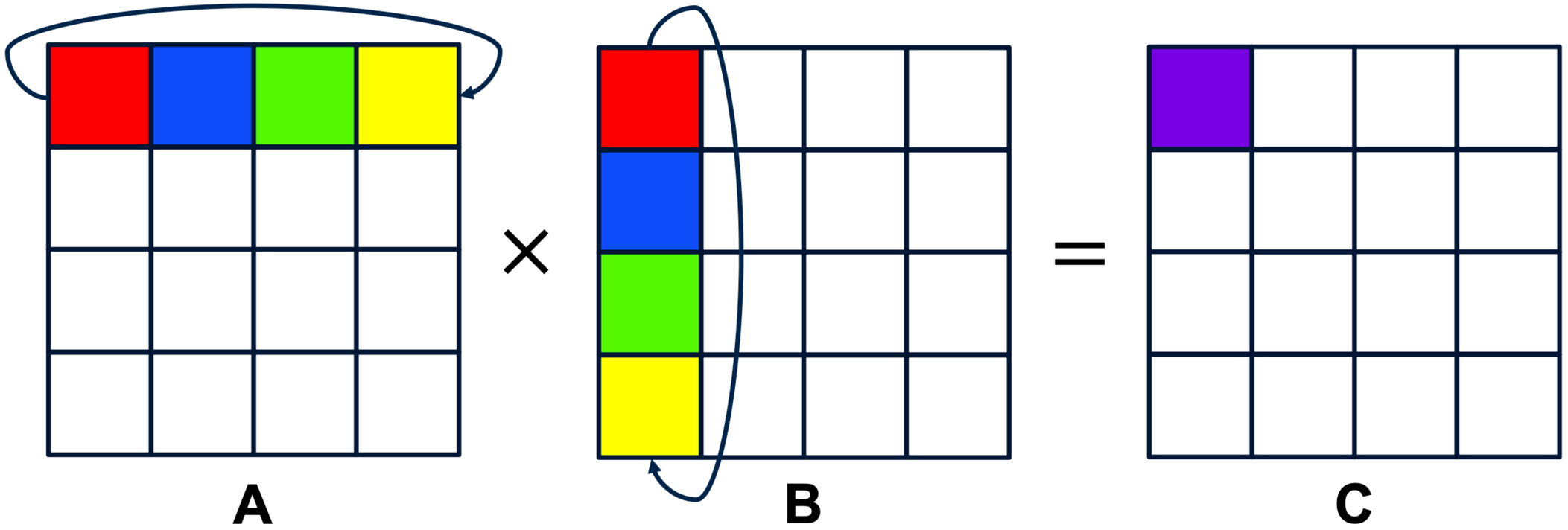


$$c_{0,1} = a_{0,0}b_{0,1} + \boxed{a_{0,1}b_{1,1}} + a_{0,2}b_{2,1} + a_{0,3}b_{3,1}$$

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

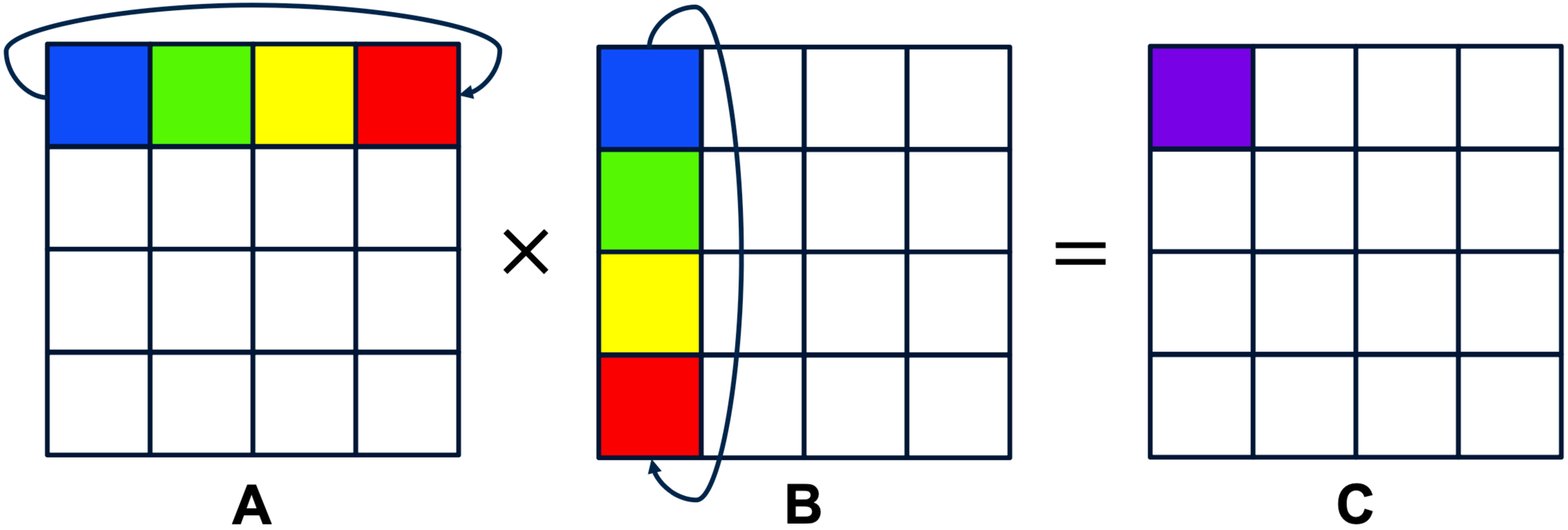
Cannon's Algorithm: Rotations

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



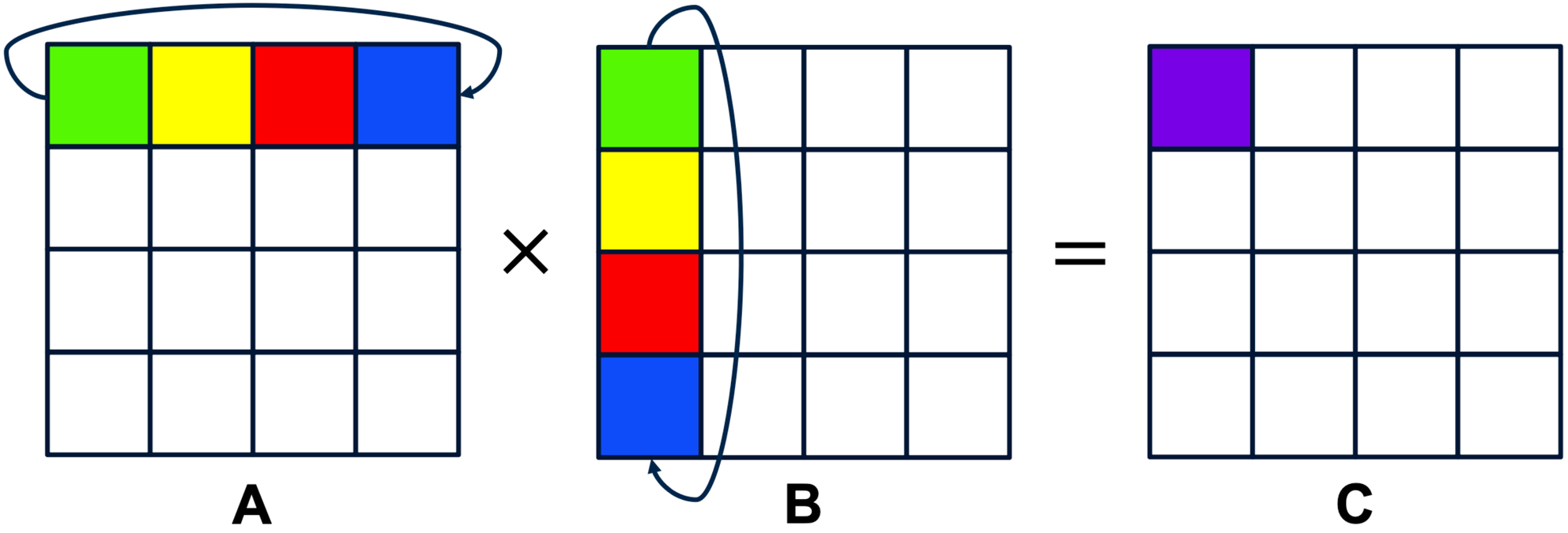
Cannon's Algorithm: Rotations

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



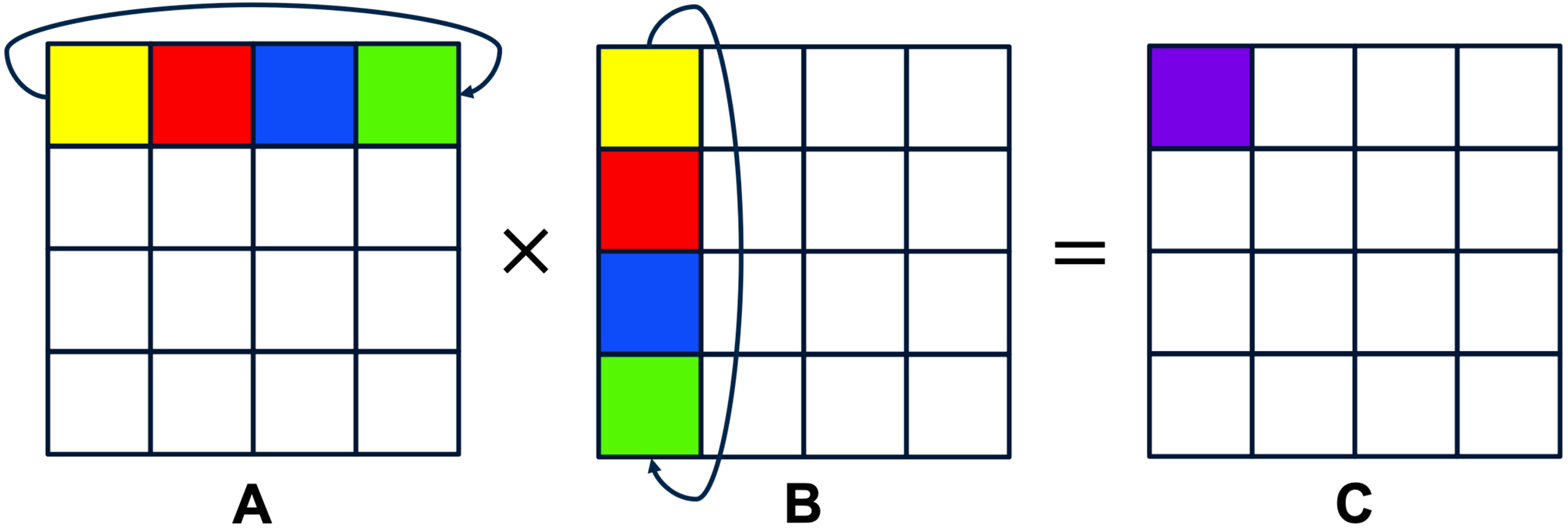
Cannon's Algorithm: Rotations

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



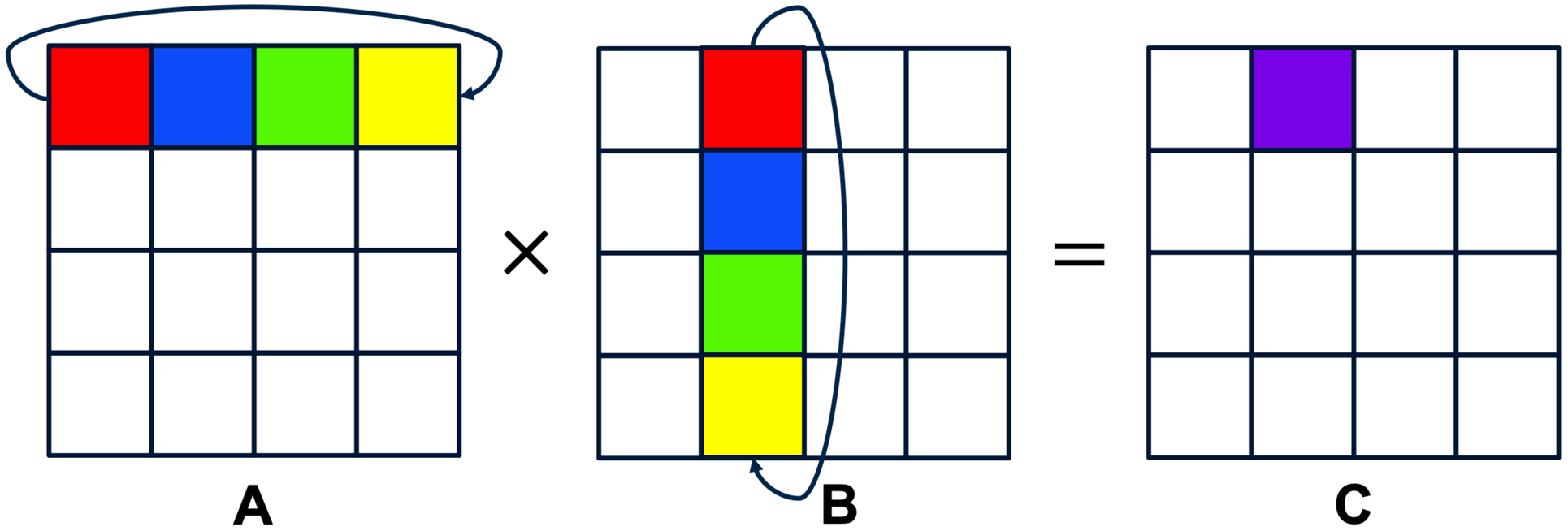
Cannon's Algorithm: Rotations

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



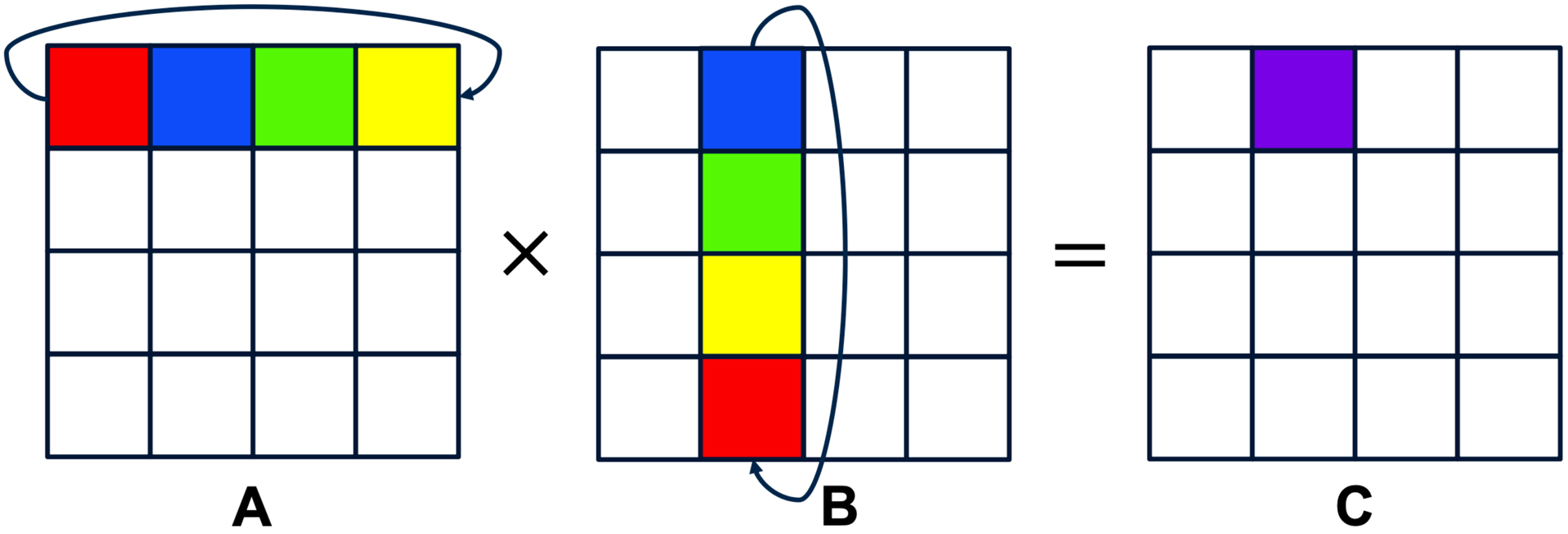
Cannon's Algorithm: Initialization

Consider processor $P_{0,1}$



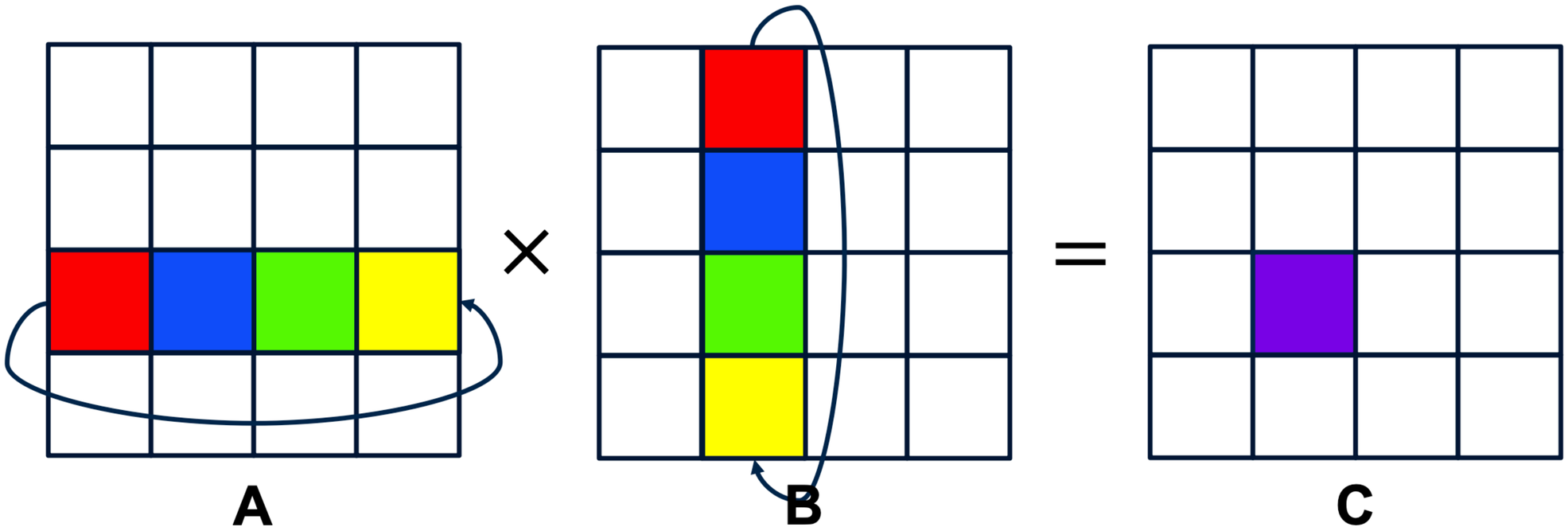
Cannon's Algorithm: Initialization

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



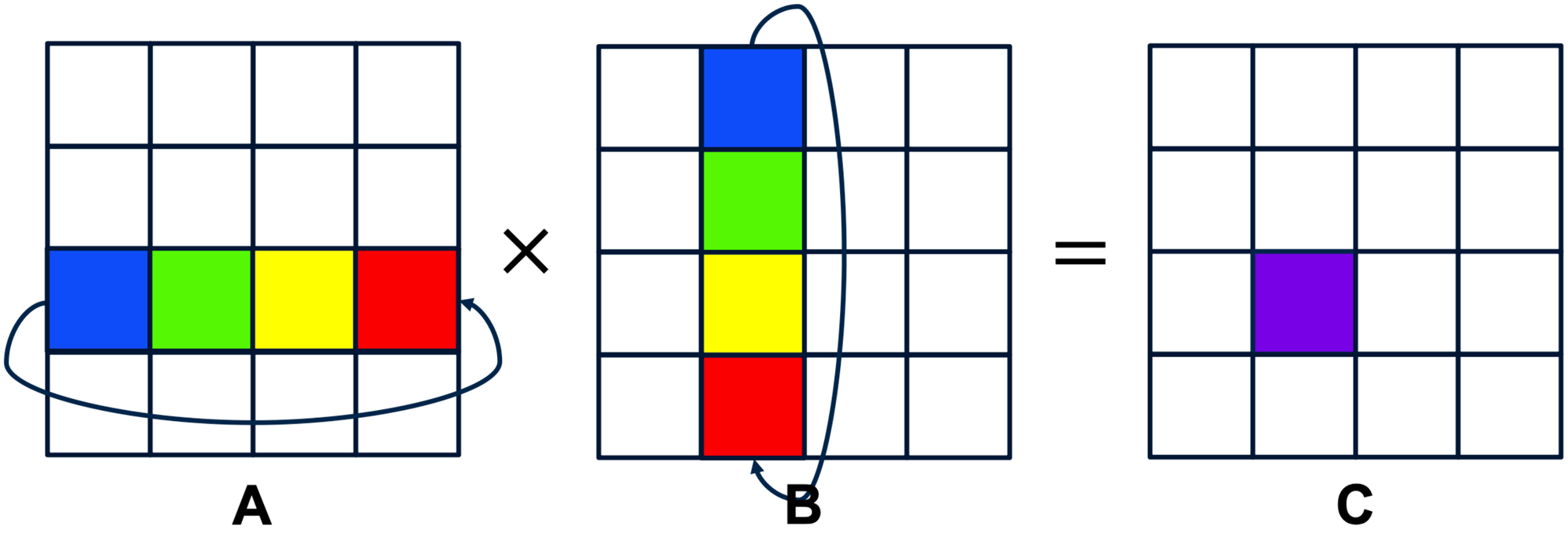
Cannon's Algorithm

- 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



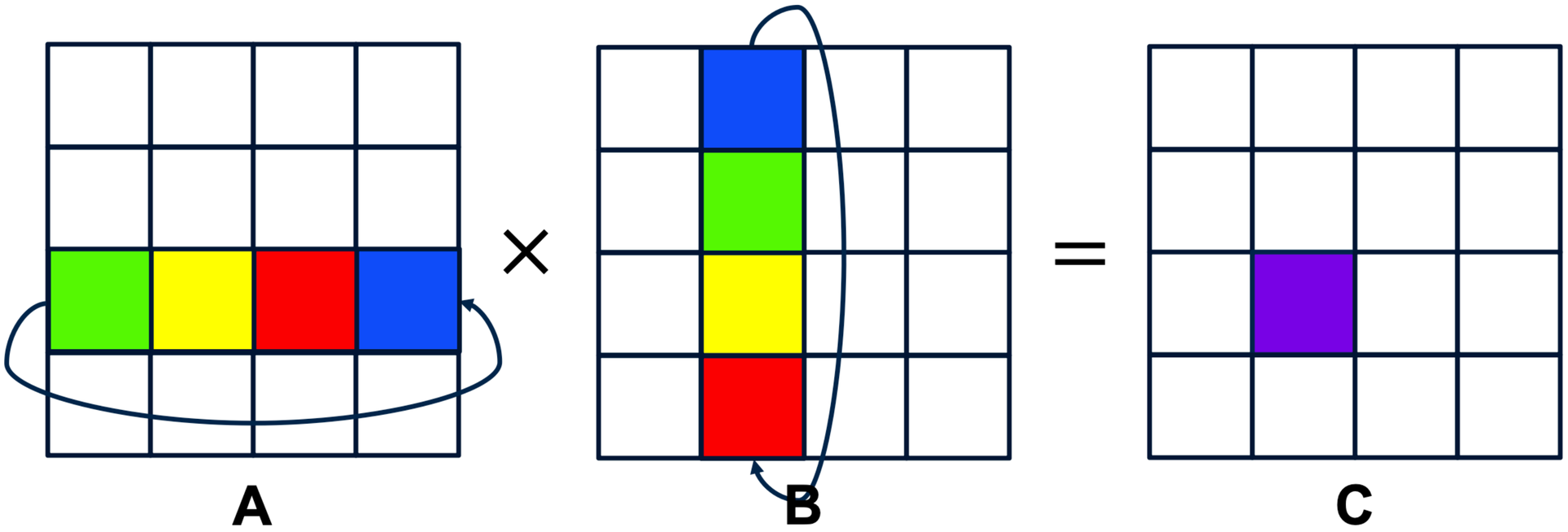
Cannon's Algorithm

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



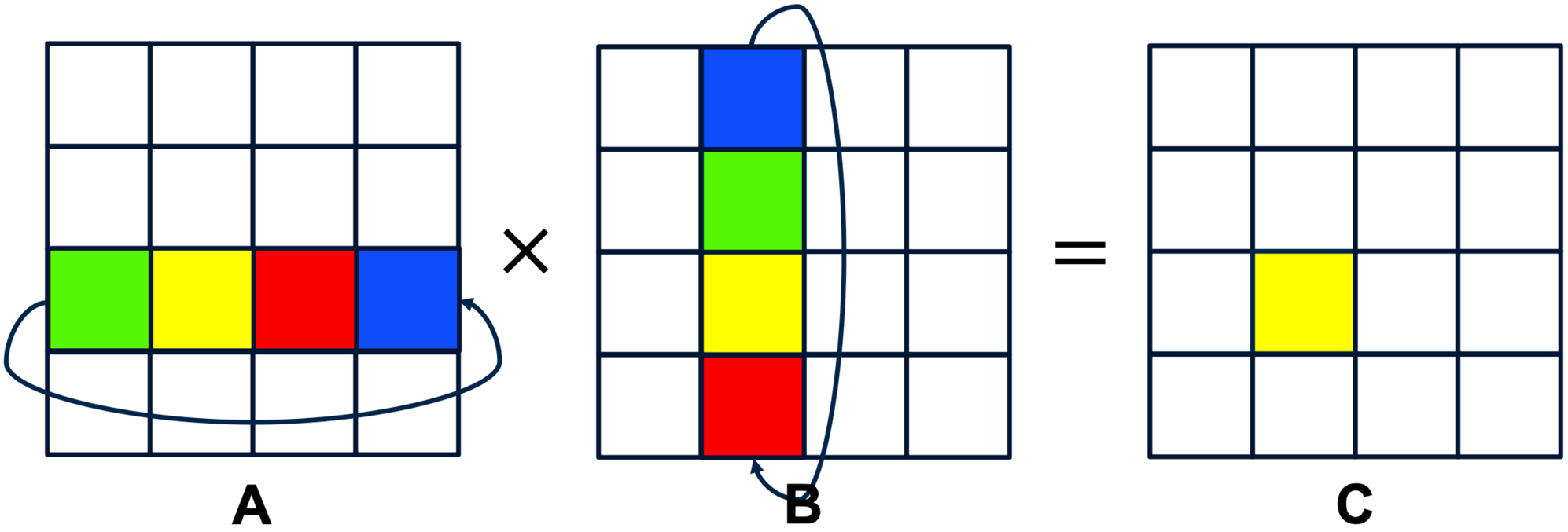
Cannon's Algorithm

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



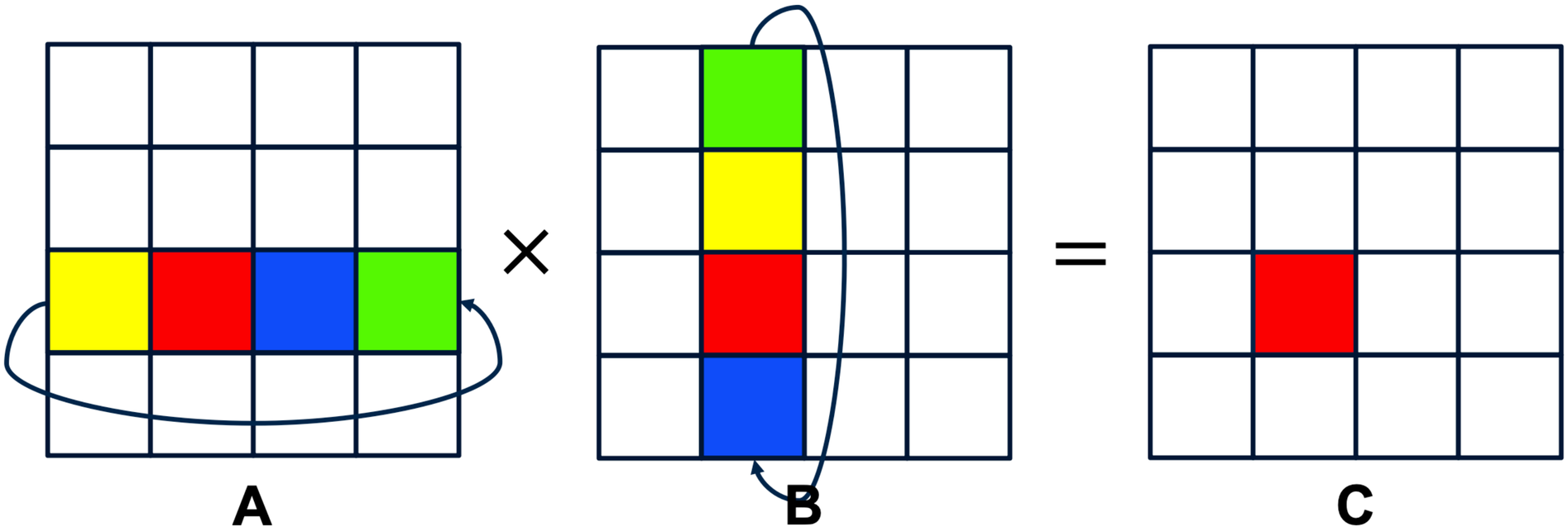
Cannon's Algorithm

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



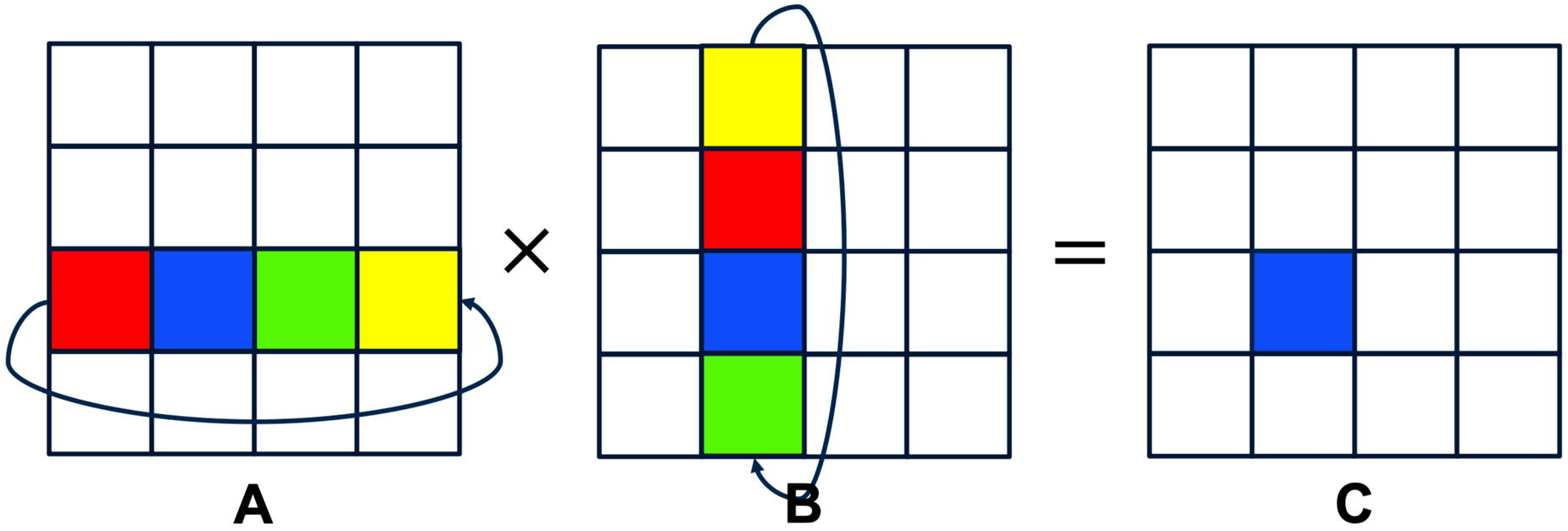
Cannon's Algorithm

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



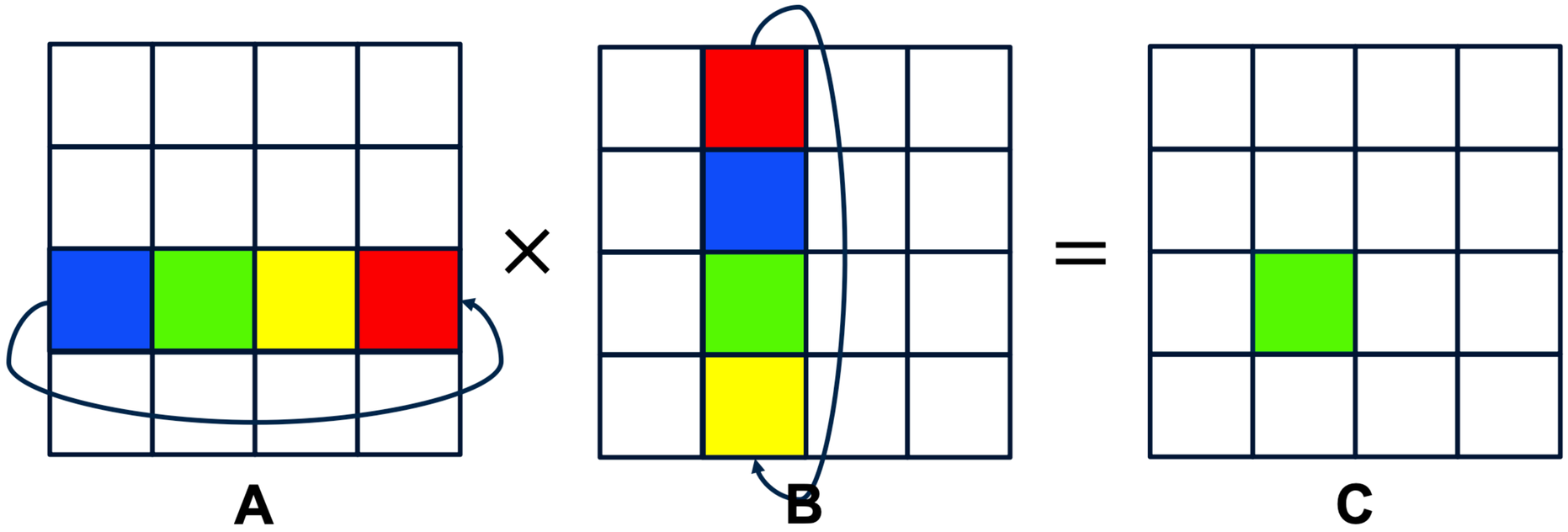
Cannon's Algorithm

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



Cannon's Algorithm

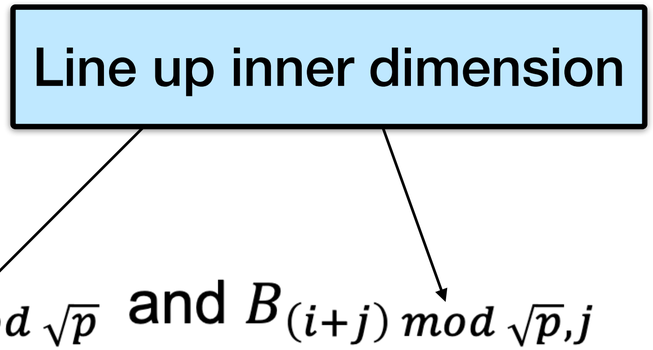
- Consider processor $P_{2,1}$
- Multiplication Step 4



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 $B_{(i+j)\ mod\ \sqrt{p},j}$



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  }
```

Outer Product Matrix Multiplication