ICS 632

Distributed-Memory Computing

1-D Data Distributions on a Ring

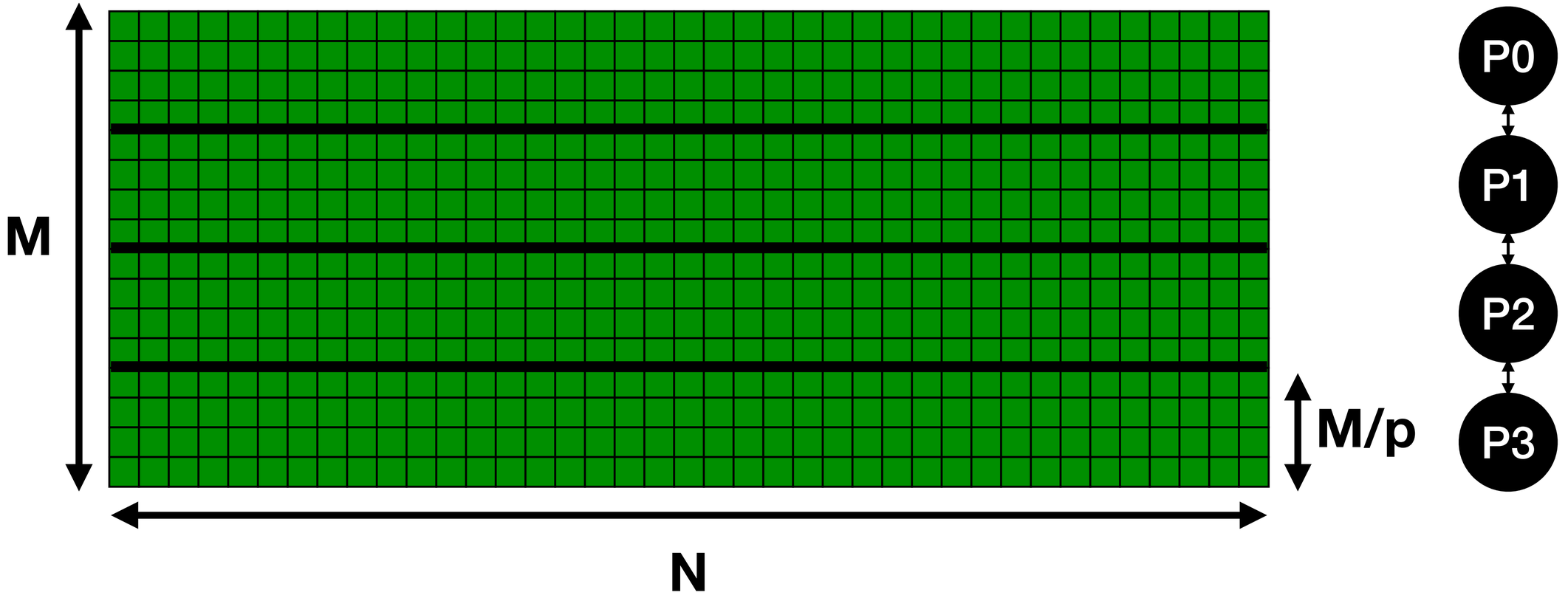
- Two fundamental concepts from previous lectures:
- Data distribution: how we split up data structures across processors
- Virtual topology: which processes will communicate which each other processes
  - MPI gives us linear ranks between 0 and p-1, any process can talk to any other
  - But we can decide to “re-number” the processes and prohibit some communication
  - The goal: make writing the parallel program easy
  - The danger: may have poor performance if physical topology is different
- Let’s do the simplest possible: 1-D Data distributed on a Ring
  - we’ve seen it before...

A Note on Data Distribution

- A common question by students is: how does the data get distributed in the first place???
- Answers:
  - The input data is generated in place by each processor (e.g., based on some input description file / arguments)
  - The input is read from data files by one process, who then sends to everybody else what they need to have
  - The input is read from files by all processes, which requires some coordination, but can benefit from high-performance parallel file systems
- Often, parallel computing libraries assume that, before you call their functions, all the data has been ”distributed”

1-D Data Distribution for a 2-D Array on a Ring

- Say we have p=4 processes running on 4 different machines
- Communication only with rank-1 and rank+1

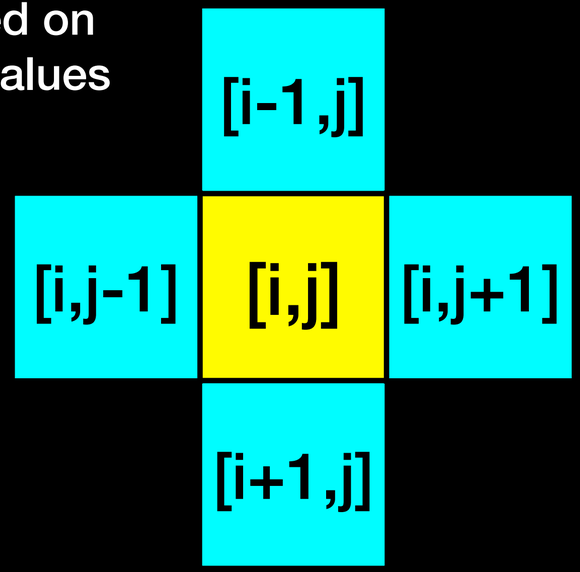


- Each process allocates RAM to store a M/p x N array

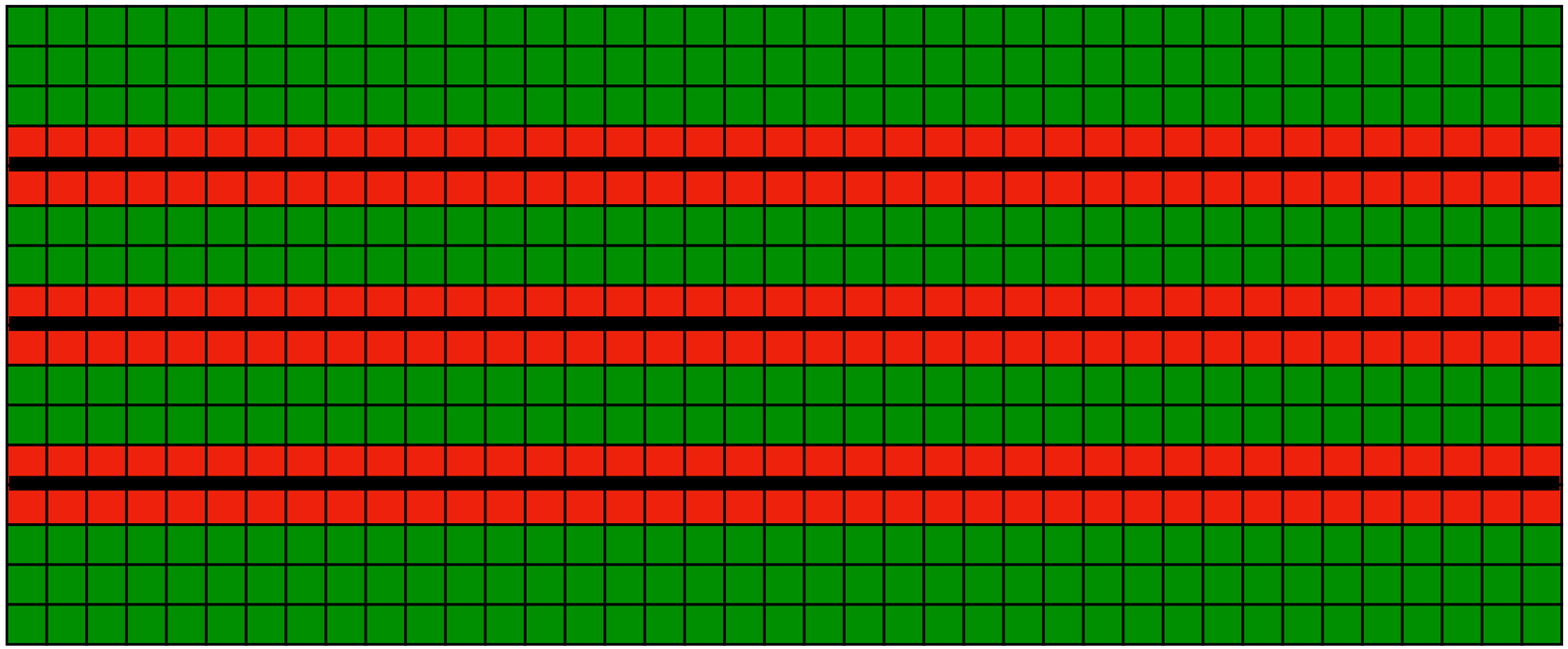
Stencil Computation

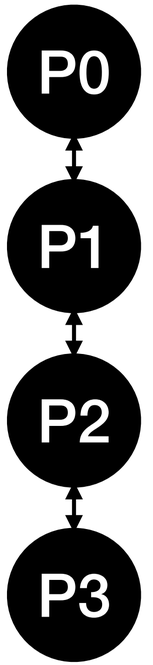
```
double *A, *B;
double *tmp, *cur, *new;
int t, i, j;

// Allocate space for A and B, initialize A
...

cur = A;
new = B;

// 1000 iterations
for (t=0; t < 1000; t++) {
    for (i=1; i < M-1; i++) {
        for (j=1; j < N-1; j++) {
            new[i*N+j] = update(cur[i*N+j], cur[i*N+j-1], cur[i*N+j+1], cur[(i-1)*N+j],cur[(i+1)*N+j]);
        }
    }
    // Swap array pointers
    tmp = cur; cur = new; new = tmp;
}
```

Update value based on
“old” neighboring values



Stencil Computation





- Updating the red cells requires cell values from neighbors

Parallel stencil computation (pseudo-code)

```
for (t=0; t < 1000; i++) {
  [ send my row 0 to rank - 1]
  [ recv row M/p-1 from rank -1]
  [ send my row N/p-1 to rank + 1]
  [ recv row 0 from rank + 1]
  < update my green cells >
  < update my red cells >
  < swap pointers as in the sequential version>
}
```

- Real code would be more complex because some processes don't have two neighbors
- We assume a bi-directional ring, so if links are not full-duplex, then there will be some contention...

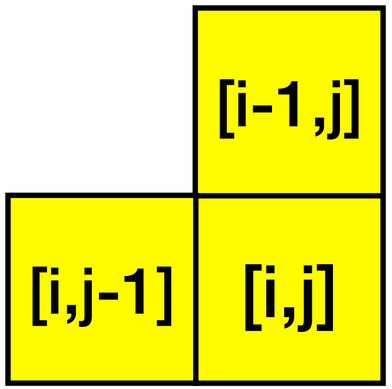
With non-blocking communication

```
for (t=0; t < 1000; i++) {
  [ send row 0 to rank - 1, asynchronously]
  [ send row N/p-1 to rank + 1, asynchronously]

  < update my green cells >

  [ recv row M/p-1 from rank -1]
  [ recv row 0 from rank + 1]

   < update my red cells >
  < swap pointers as in the sequential version>
}
```

- Should be implemented using MPI_Isend
- If the time to send/ receive rows of cells is shorter than the time to update green cells, then we have fully hidden the communication overhead
  - Depends on network speed, data sizes, number of processes, etc.
- If fully hidden, we could get ~100% parallel efficiency

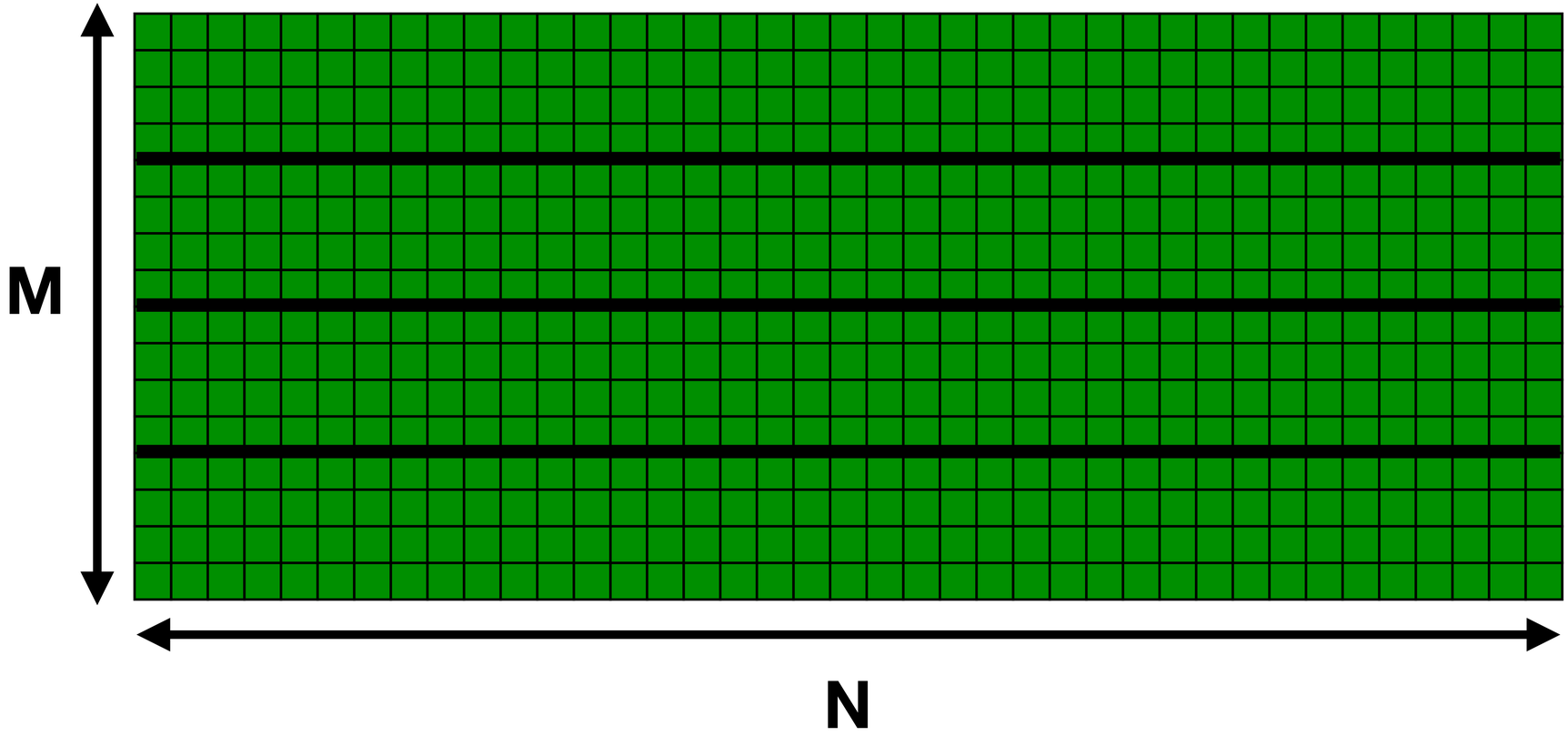
Less simple stencil?

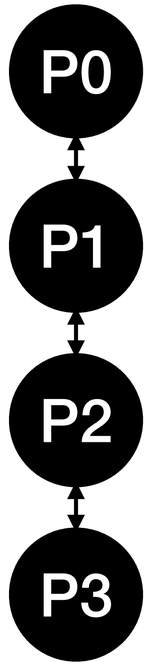
- This was easy enough, and you should feel that you could write the MPI version without too much trouble
- Recall from Homework #3 that we looked at a less simple case:



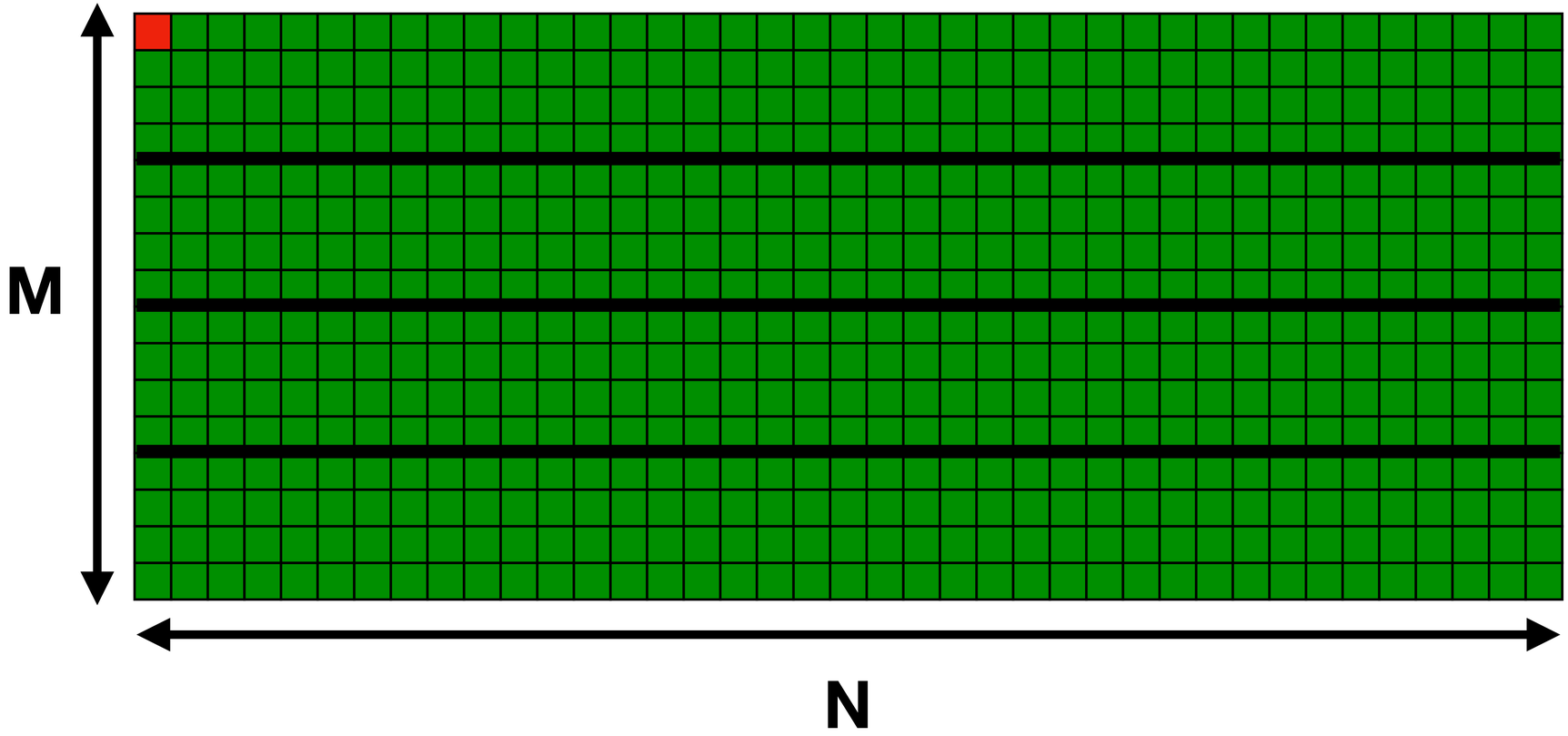
- We have a single array, and we update a cell based on the updated West and North neighbors
- In Homework #3 you did some re-arranging to expose parallelism
- But what about in distributed memory?

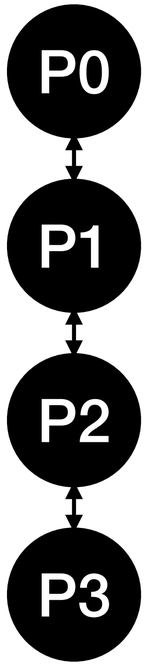
Less easy stencil: basic execution



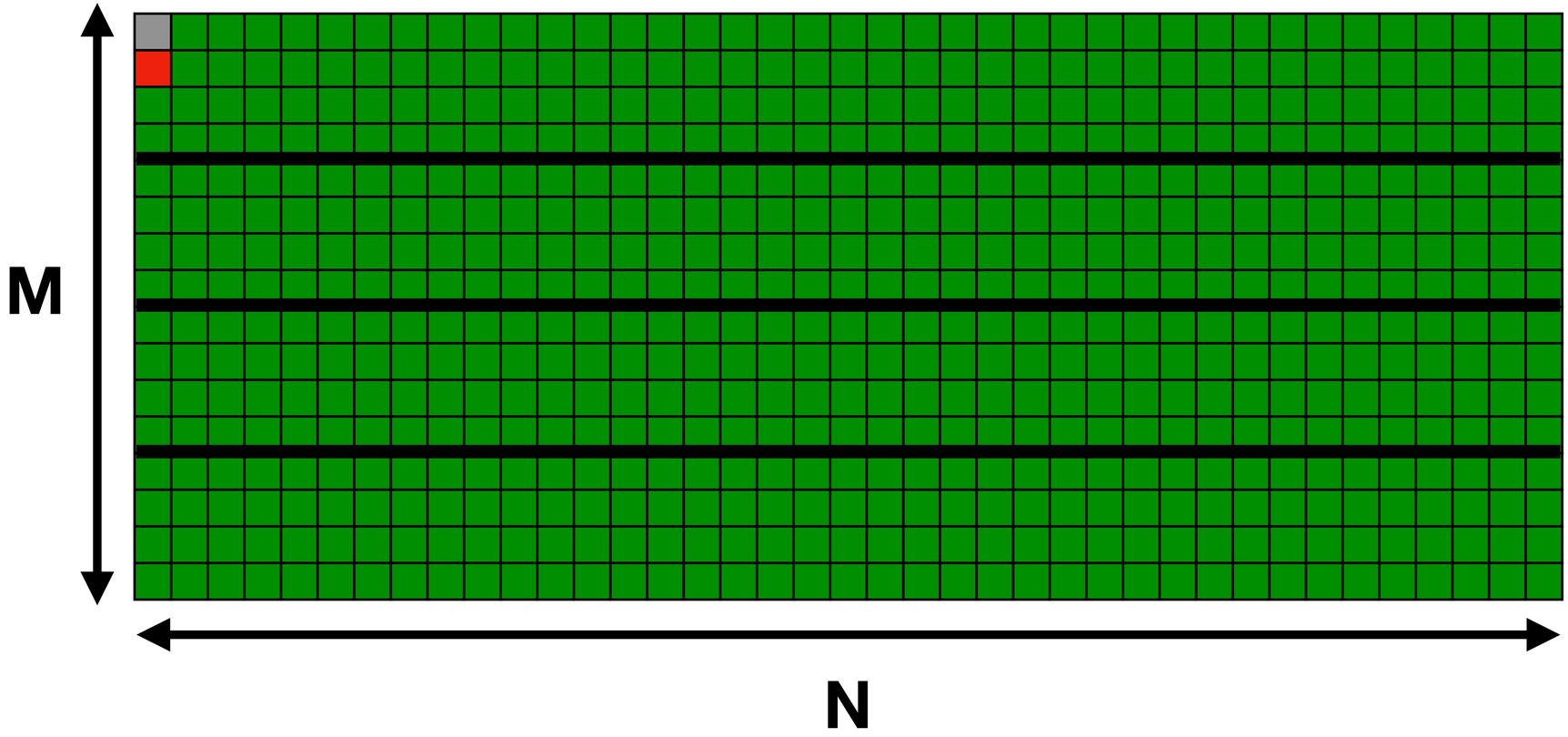


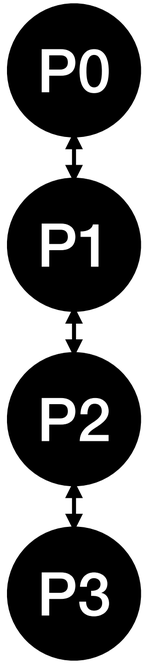
Less easy stencil: basic execution



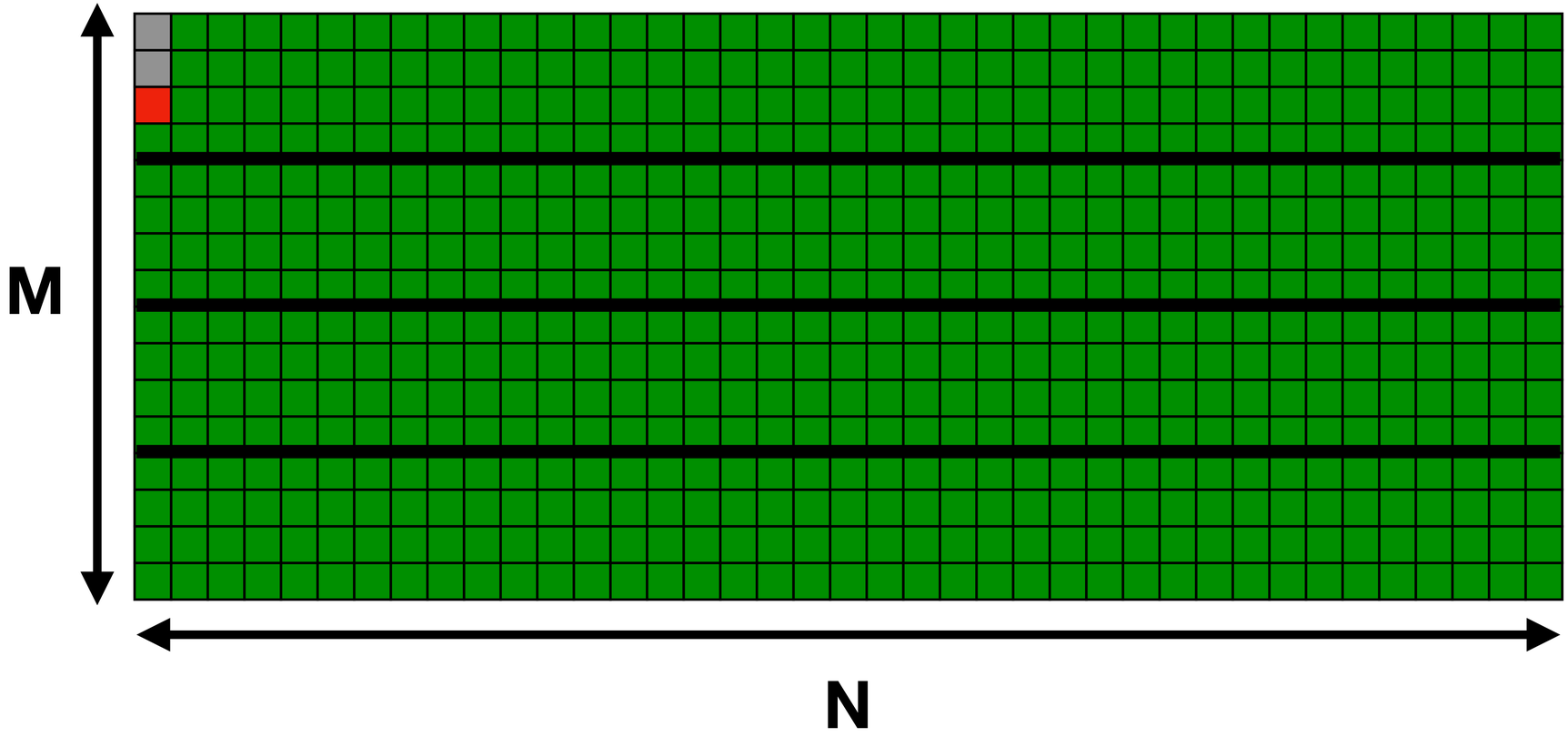


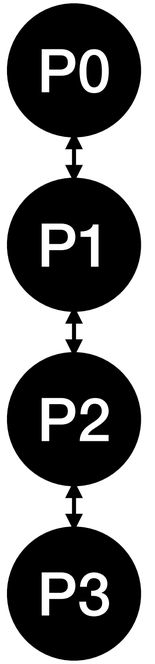
Less easy stencil: basic execution



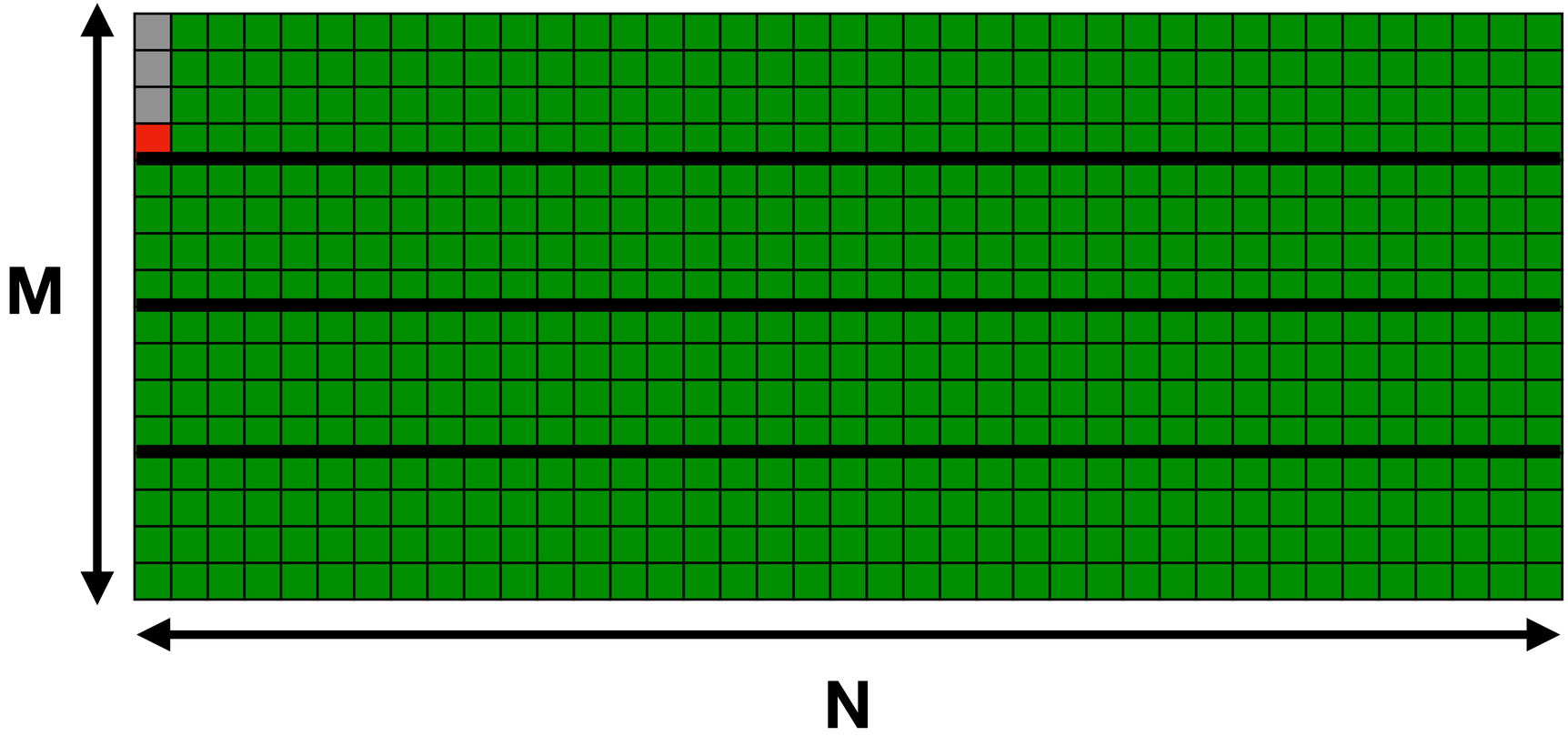


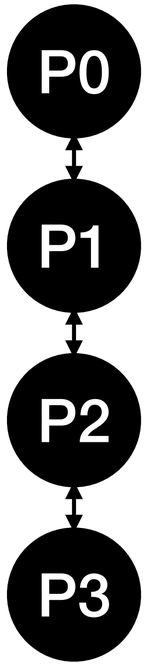
Less easy stencil: basic execution



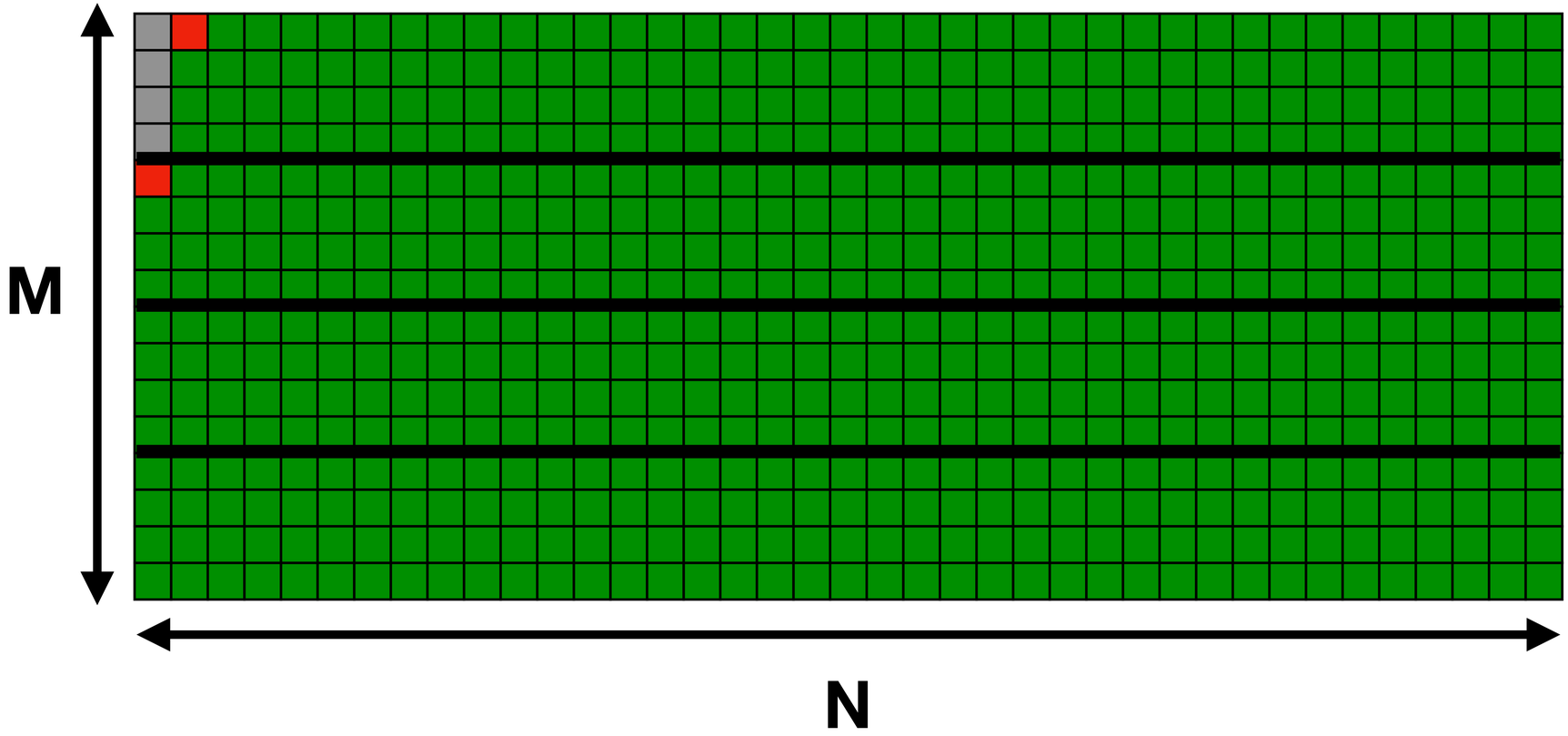


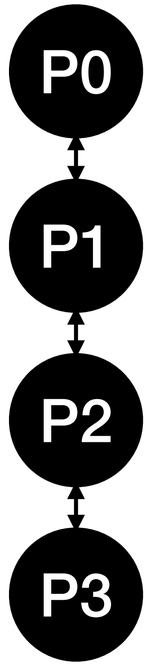
Less easy stencil: basic execution



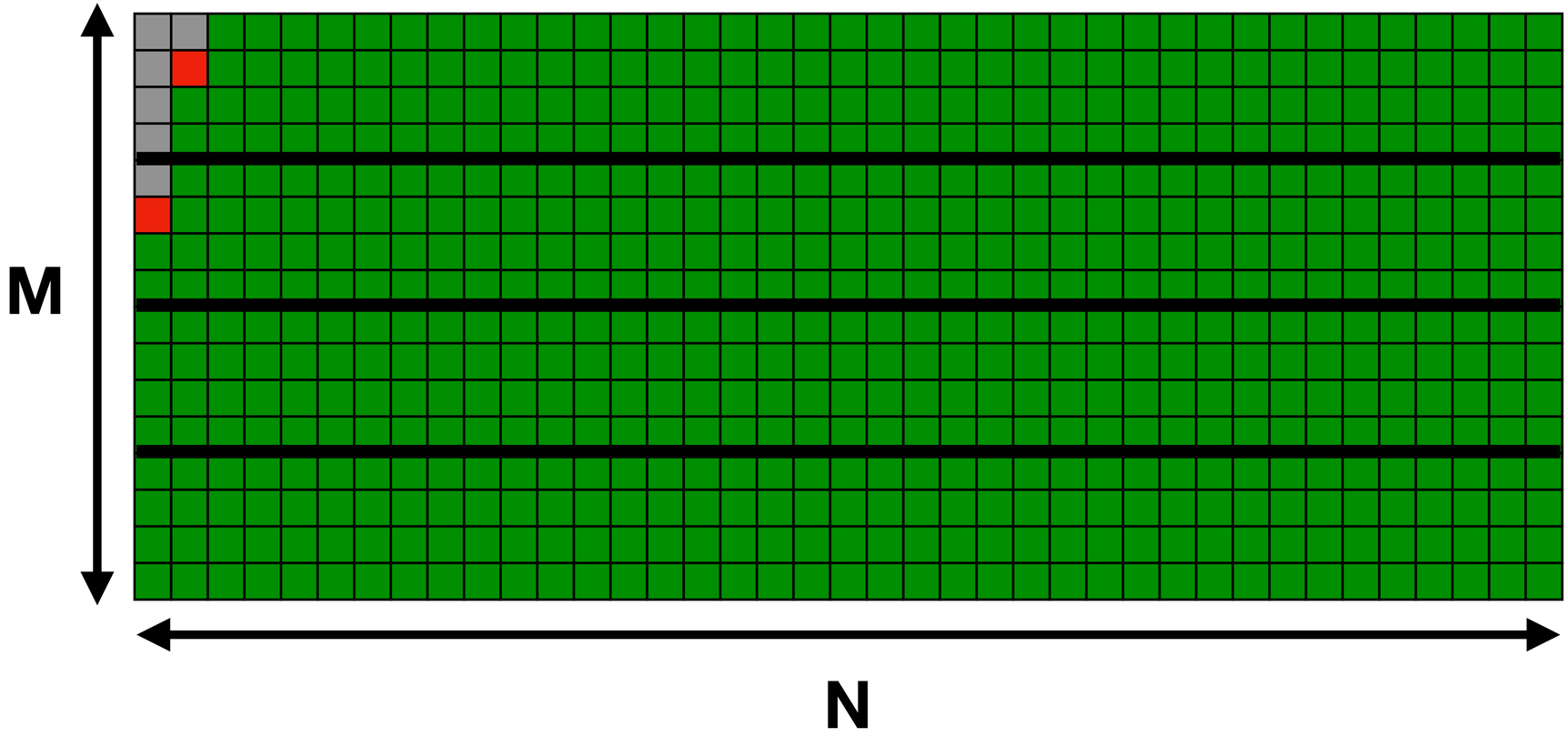


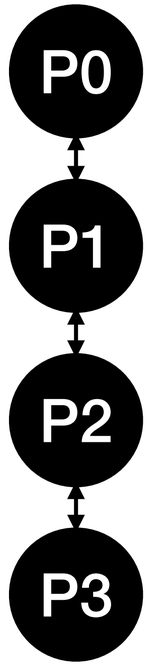
Less easy stencil: basic execution





Less easy stencil: basic execution





- And so on.....
- Question: What's he parallel speedup?

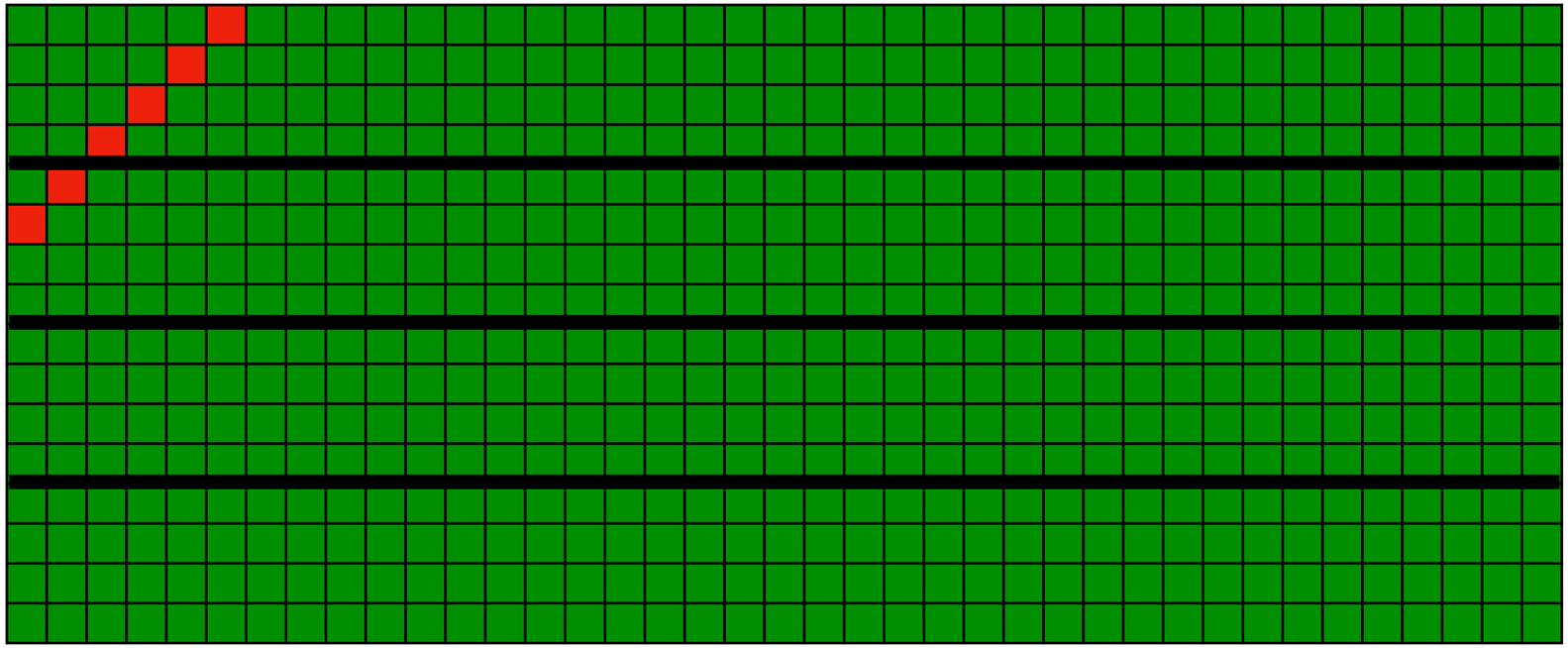
Parallel Speedup

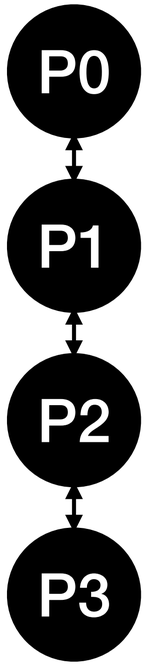
- Say the communication is infinitely fast (or 100% hidden)
- Say the time to update a cell is c
- What’s the execution time? Any idea.....

Parallel Speedup

- Say the communication is infinitely fast (or 100% hidden)
- Say the time to update a cell is c
- What's the execution time? Any idea.....
- The last process begins computing at time $(p-1) \times (N/p) \times c$
- It then computes for $(M/p) \times N \times c$ units of time
- Parallel execution time: $(p-1) \times (M/p) \times c + (M/p) \times N \times c$
- Sequential execution time: $M \times N \times c$
- Parallel speedup: $pM/(p-1+M)$
- If M is large, then speedup is close to p: asymptotically optimal

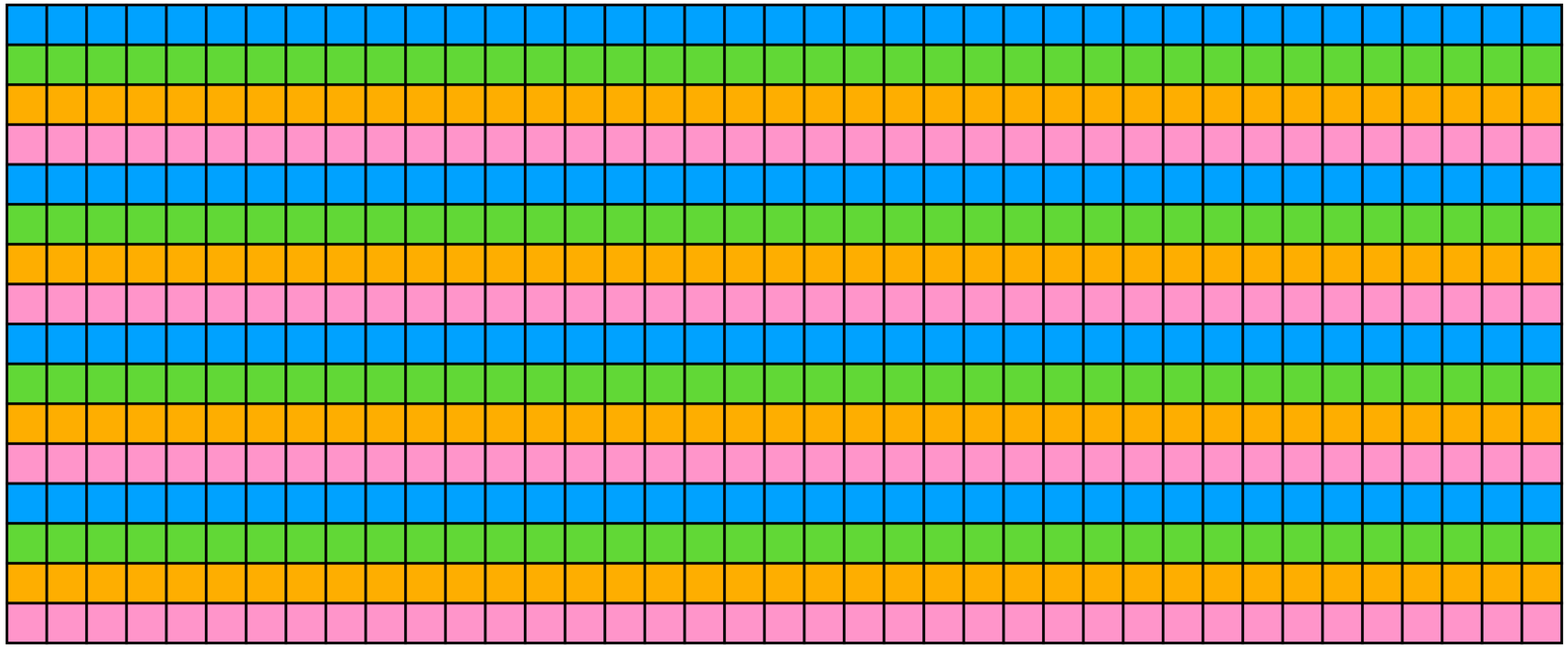
Better execution: along the anti-diagonal

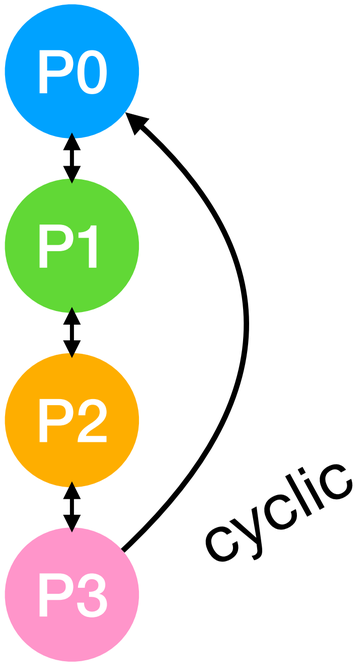




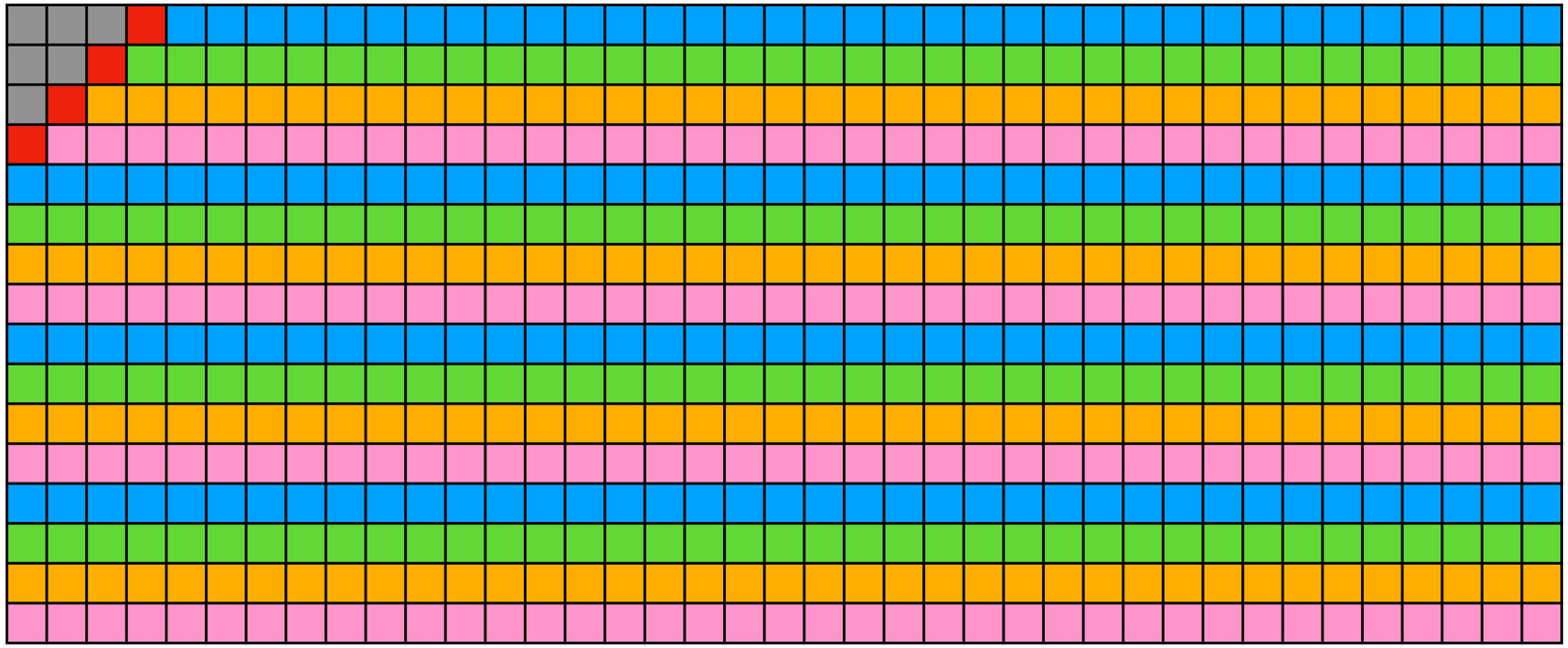
- This is the same idea you used in Homework #2
- But here, we don’t get great parallelism because of the 1-D data distribution
  - In the above, we can compute 6 values in parallel but only 2 processes can work
- What could we do? ...

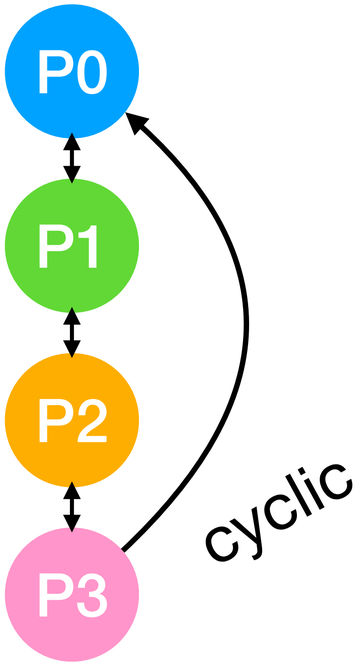
Cyclic Data Distribution





Cyclic Data Distribution





- Every processor can do work
- Of course, this complexifies the code quite a bit (the distribution is no longer as simple)
- And we haven’t even talked about memory locality
- And of course we should combine this with OpenMP, perhaps using tiling!!

Stencil Computation

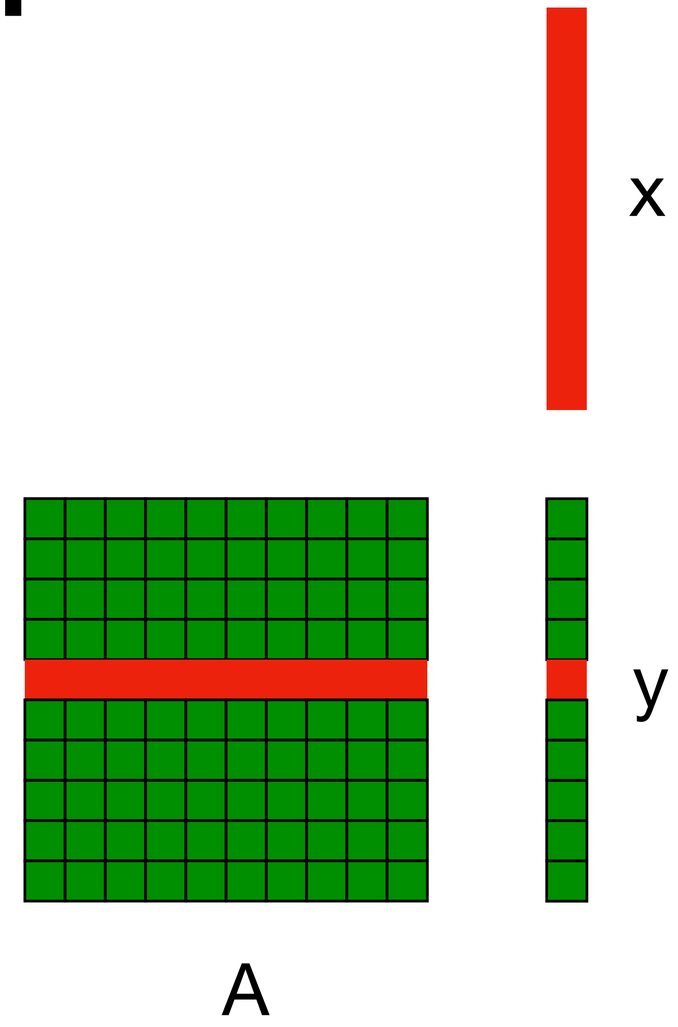
- It turns out we could keep going and analyze this implementation and come up with other implementations
- This seemingly simple computation can become quite a handful if you're looking at getting high performance on a parallel machine
  - Good end-of-the-semester project idea
- All these possible implementations have asymptotically optimal speedup, but the goal is to get constant factors lower in low-order term

- Let's leave all this at that for now, and turn to a perhaps simpler computation: Matrix-Vector Multiplication...

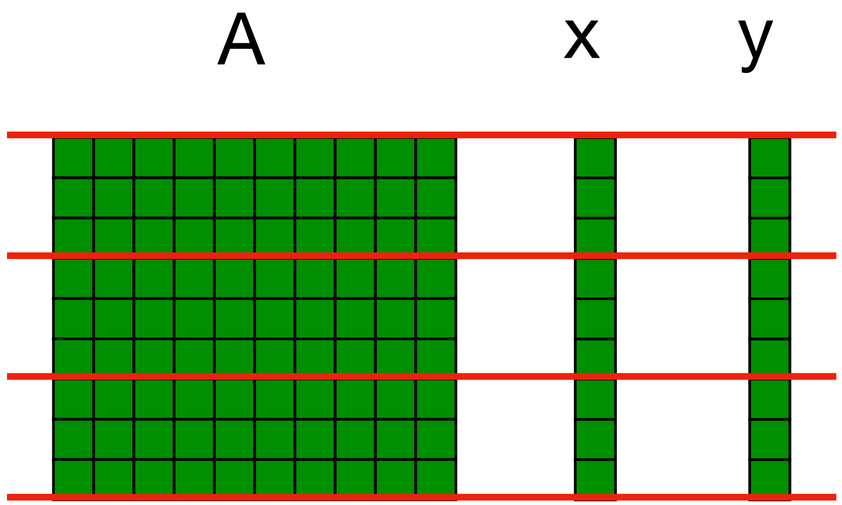
Matrix-Vector Multiplication

- Classic $O(N^2)$ algorithm

```
int A[N][N], x[N], y[N];

// y = A * x

for (int i=0; i < N; i++) {
    y[i] = 0;
    for (int j=0; j < N; j++)
        y[i] += A[i][j] * x[j];
}
```



1-D Data Distribution

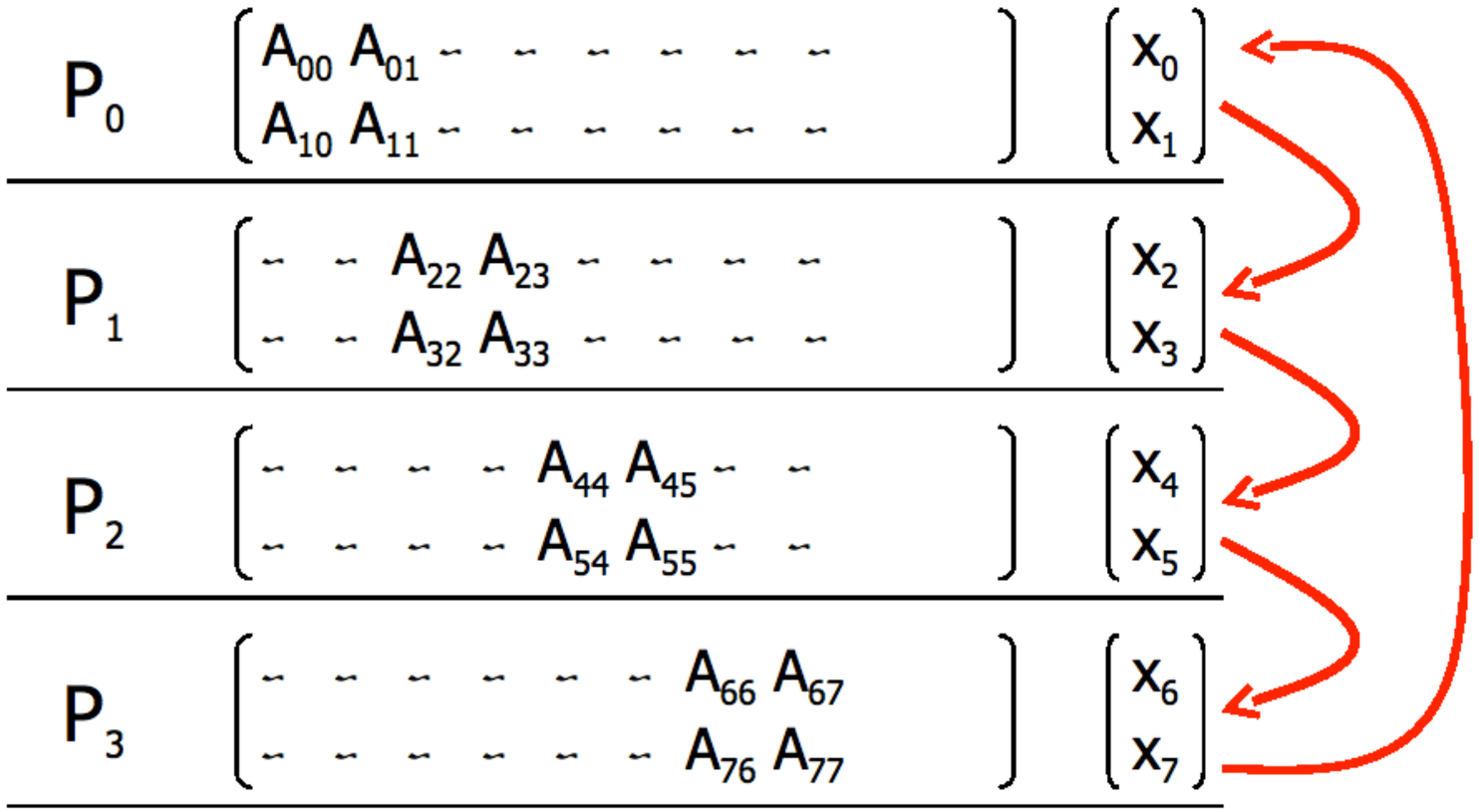


- Each processor holds a slice of A, a slice of x, and a slice of y
- We thus go fully distributed memory
  - We could have said: “let’s replicate x across all processors”
  - That’s easier, but very non-standard
- Let’s look at an example for 4 processors and a 8x8 matrix

Initial State (N=8, p=4)

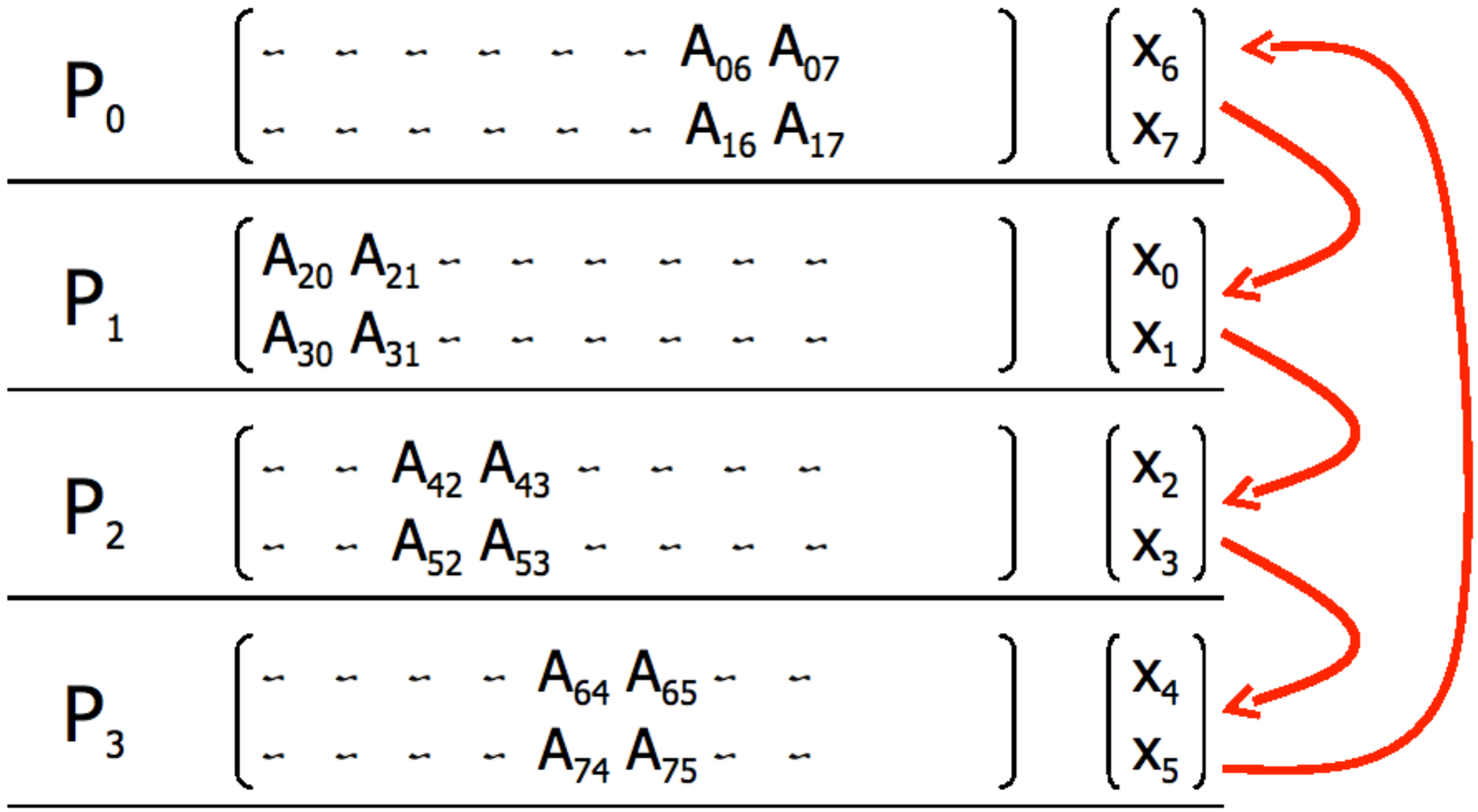
$$
\begin{array}{l|l|l}
P_0 & \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\
\hline
\end{array}
$$

Step 1: Compute + Send



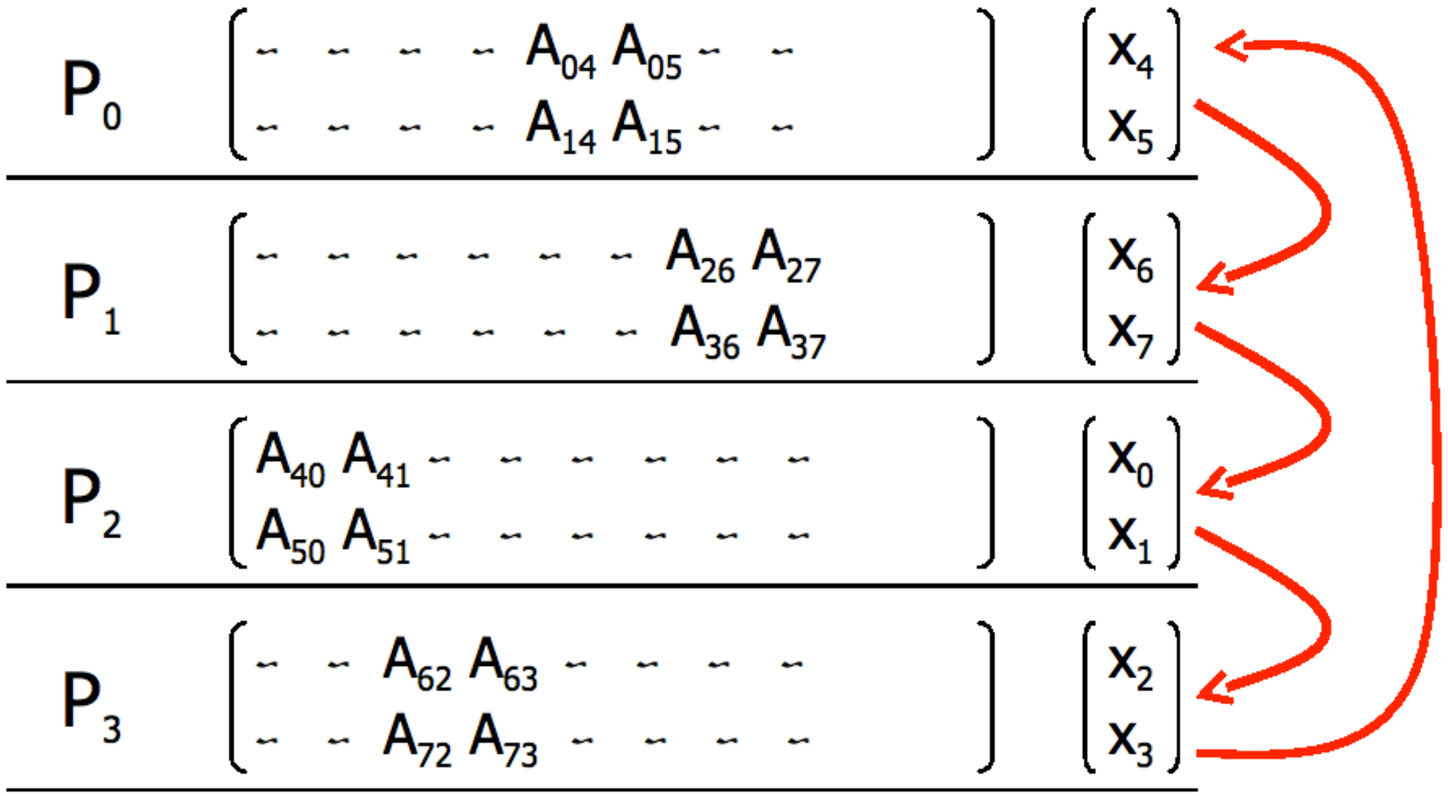
- Each process computes a 2x2 mat-vec multiply to update its elements of y
  - Highlighted in the figure above
- Each process sends its slice of x to its successor

Step 2: Compute + Send



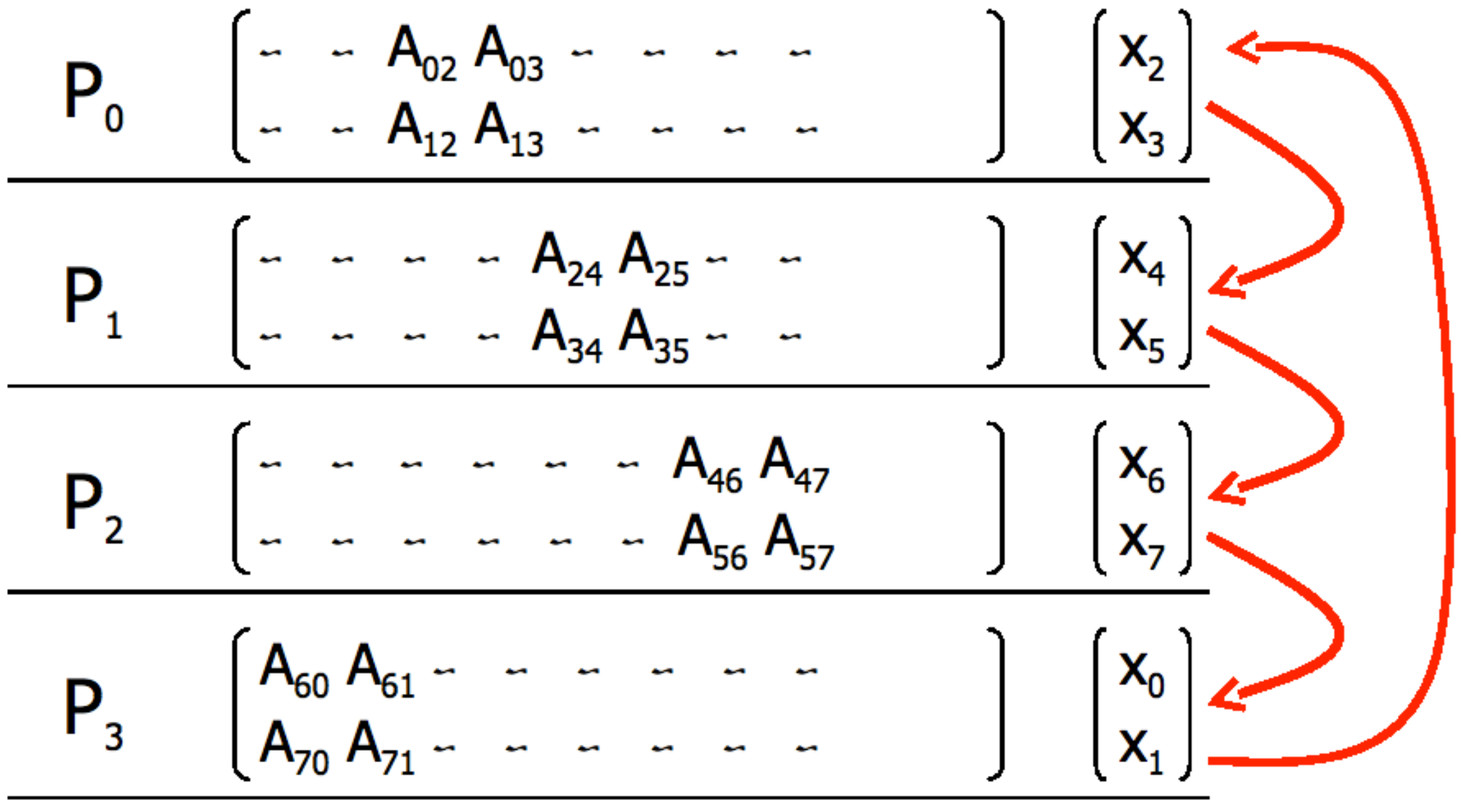
- Each process computes a 2x2 mat-vec multiply to update its elements of y
  - Highlighted in the figure above
- Each process sends its slice of x to its successor

Step 3: Compute + Send



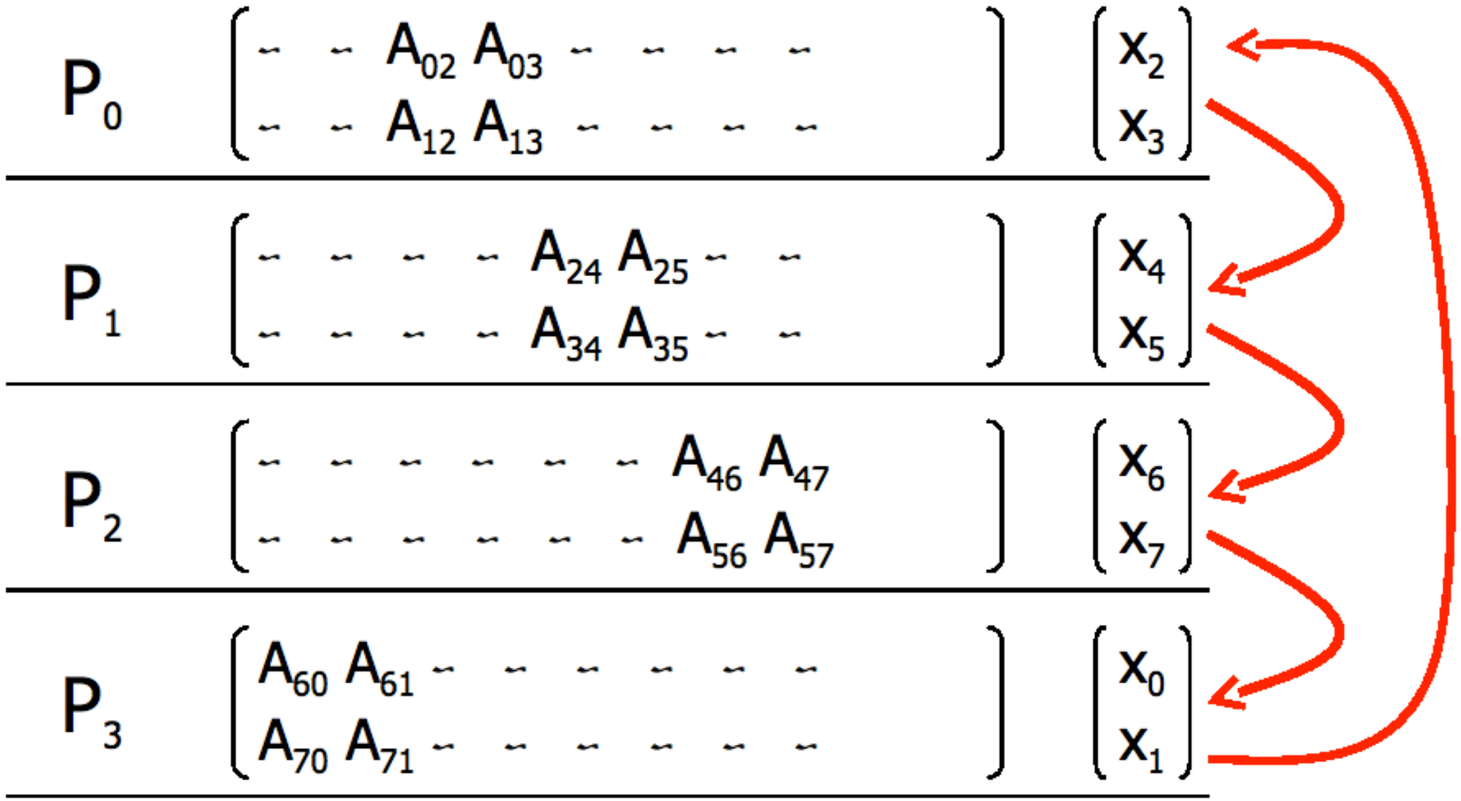
- Each process computes a 2x2 mat-vec multiply to update its elements of y
  - Highlighted in the figure above
- Each process sends its slice of x to its successor

Step 4: Compute + Send



- Each process computes a 2x2 mat-vec multiply to update its elements of y
  - Highlighted in the figure above
- Each process sends its slice of x to its successor

Step 5: Send



- One last send so that vector x is as it was at the beginning of the computation
- Each elements of y is a sum of 8 terms, there were 4 steps, and at each step 2 terms were added to each element of y

Pseudo-Code

```
int rank = my_rank();
int p = num_procs();
int A[N/p][N], x[N/p], y[N/p];
int buffer[N/p];
int *tempR = buffer;  // receive buffer
int *tempS = x;       // to send

for (step = 0; step < p; step++) {
    send(tempS, N/p);  // my slice of x to my successor
    recv(tempR, N/P);   // my predecessor's slice of x to me

    for (i=0; i < N/p; i++)
        for (j=0; j < N/p; j++)
            y[i] += a[ i ] [ (rank - step mod p) * N/p + j ] * tmpS[ j ];

   tmpR <-> tmpS;   // swap pointers (overwriting my slice of x with my predecessor's)
}
```

Performance Analysis

- Each processor goes through p steps
- Each step involves:
  - sending N/p elements: takes time $\alpha + \beta N/p$
  - receiving N/p elements: takes time $\alpha + \beta N/p$
  - Computing (N/p)^2 elements: takes time $(N/p)^2 c$
- Parallel time: $p((N/p)^2 c + 2\alpha + 2\beta N/p)$
- Sequential time: $N^2 c$
- Speedup when $N \to \infty$: p
- This algorithm is asymptotically optimal again
- Can we do better? Sure, with non-blocking communications!
  - To hide some of the communication cost

1-D Algorithms galore

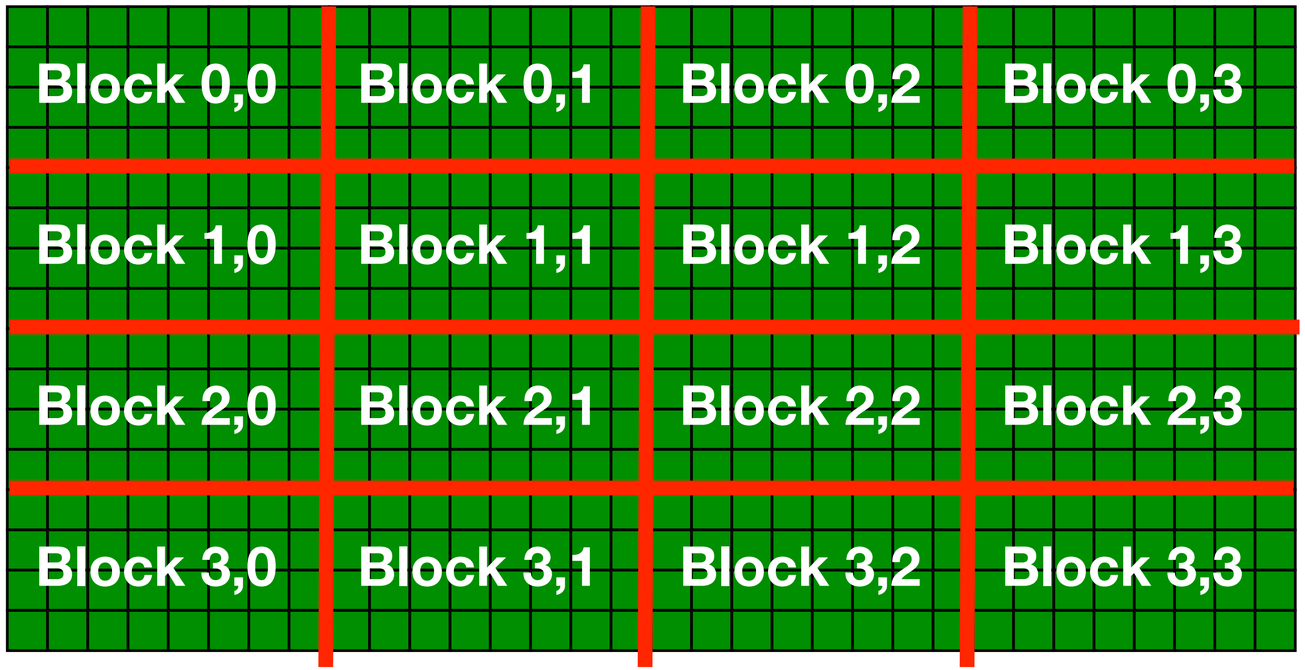
- There are many, many known good parallel algorithms and implementations for 1-D distributions
  - Everything linear-algebra of course, but much more
- We could almost spend a whole semester on this
- And although we haven't done so here, we have to throw everything into the mix to extract the best performance:
  - Dealing with memory locality is a must (and complicates matters)
  - Combining MPI and OpenMP is typically a good idea
- These are things some of you can explore in projects later this semester
- But let's now look at 2-D distributions...

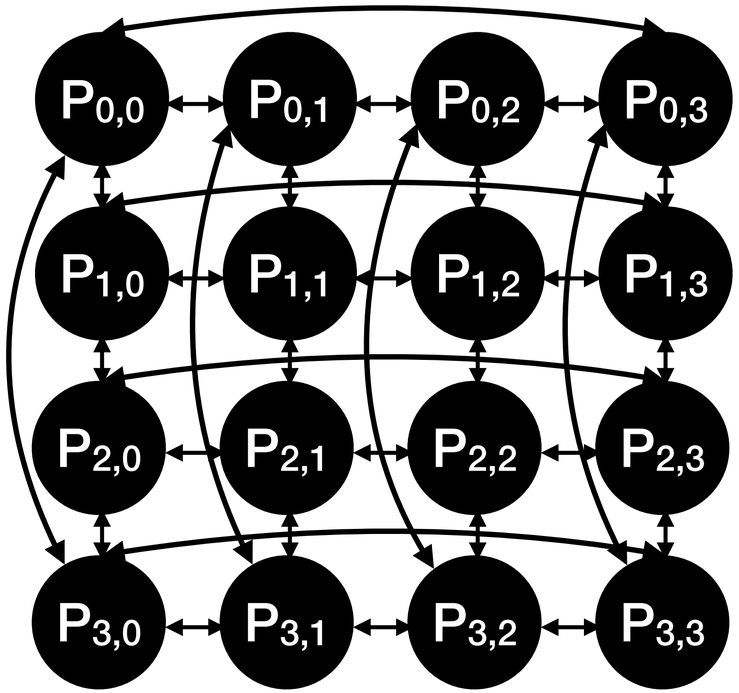
2-D Data Distribution and Tori

- For many classical algorithms, people have used 2-D data distributions
- Let’s again consider simple 2-D arrays
- Instead of assigning processes slices of the array, we assign them blocks of the array, and consider that they communicate in a Torus

2-D Data Distribution and Tori

- For many classical algorithms, people have used 2-D data distributions
- Let's again consider simple 2-D arrays
- Instead of assigning processes slices of the array, we assign them blocks of the array, and consider that they communicate in a Torus





Process Numbering

- MPI gives us 1-D (linear) ranks: 0 to p-1
- But for convenience, we really need to number the processes using a 2-D scheme
- Each process will be in a “process column” and a “process row”
- Thanks to the magic of discrete math, assuming that p is a perfect square, we have the following 1-D to 2-D mapping:
  - process row $= \lfloor \text{rank}/\sqrt{p} \rfloor$
  - process column $=$ rank $\mod \sqrt{p}$
- In code, right after calling MPI_Rank(), you would then compute the above to know which process you are in the grid/torus

Matrix Multiply

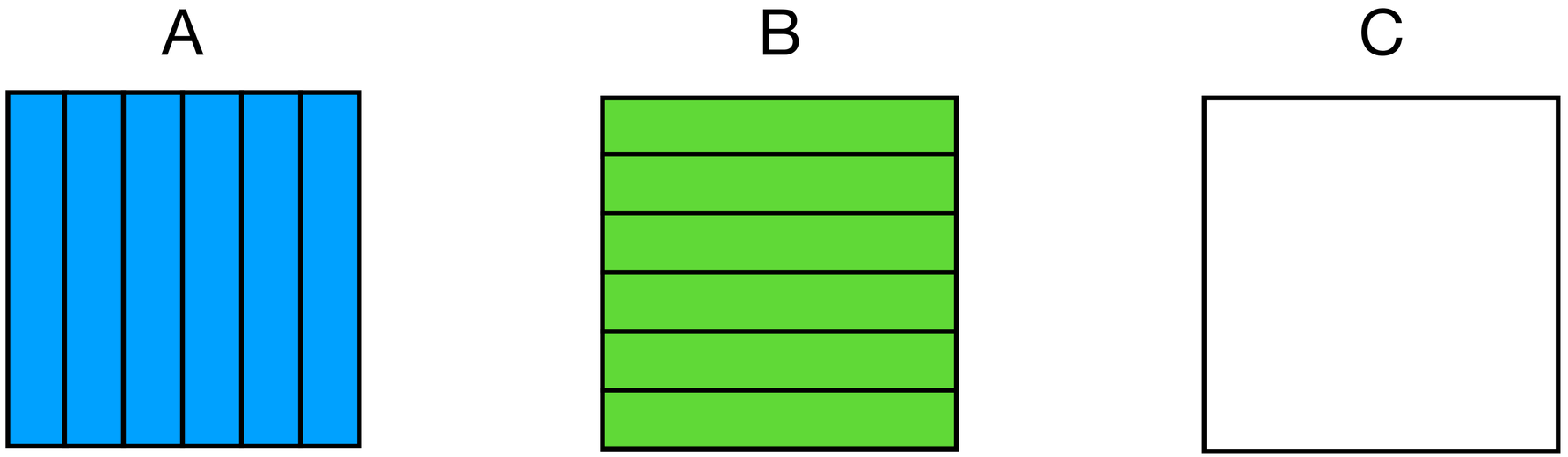
- Let's pick the simples possible computation: square matrix multiply
- And let's write it in parallel using a 2-D distribution
- Let us consider the k-i-j order:

```
for (k = 0; k < N; k ++)
    for (i = 0; i < N; i ++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

- One way to look at this algorithm is a sequence of outer-products!
  - Outer-product: product of a column vector by a row vector

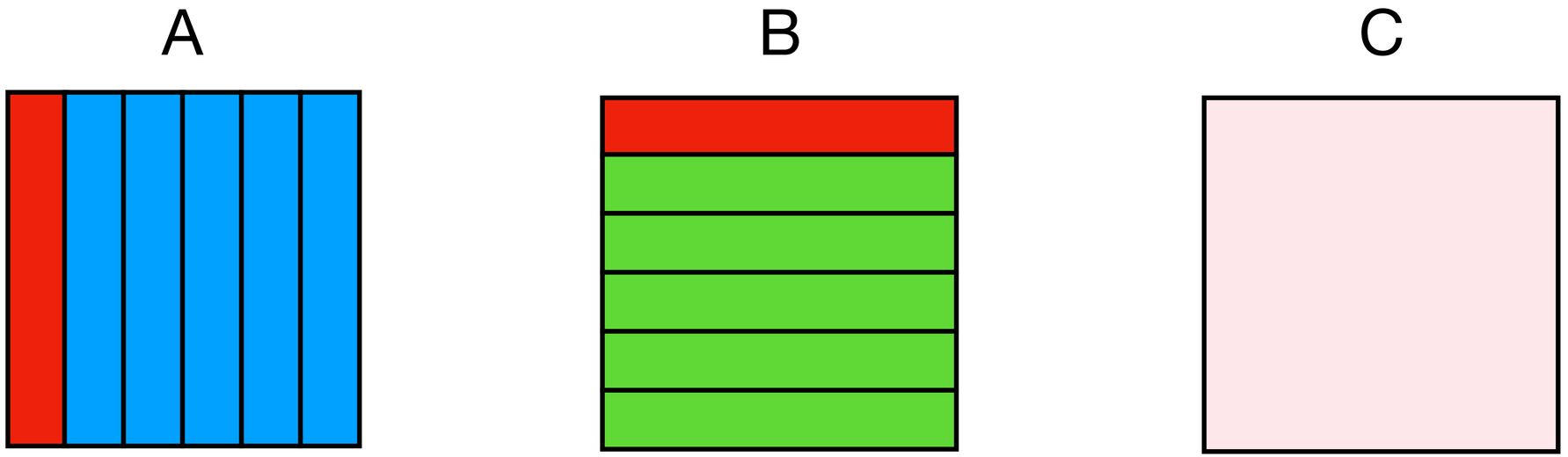
Matrix Multiply: Sequence of Outer-Products

```
for (k = 0; k < N; k ++)
    // Multiply a column of A by a row of B
    for (i = 0; i < N; i ++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```



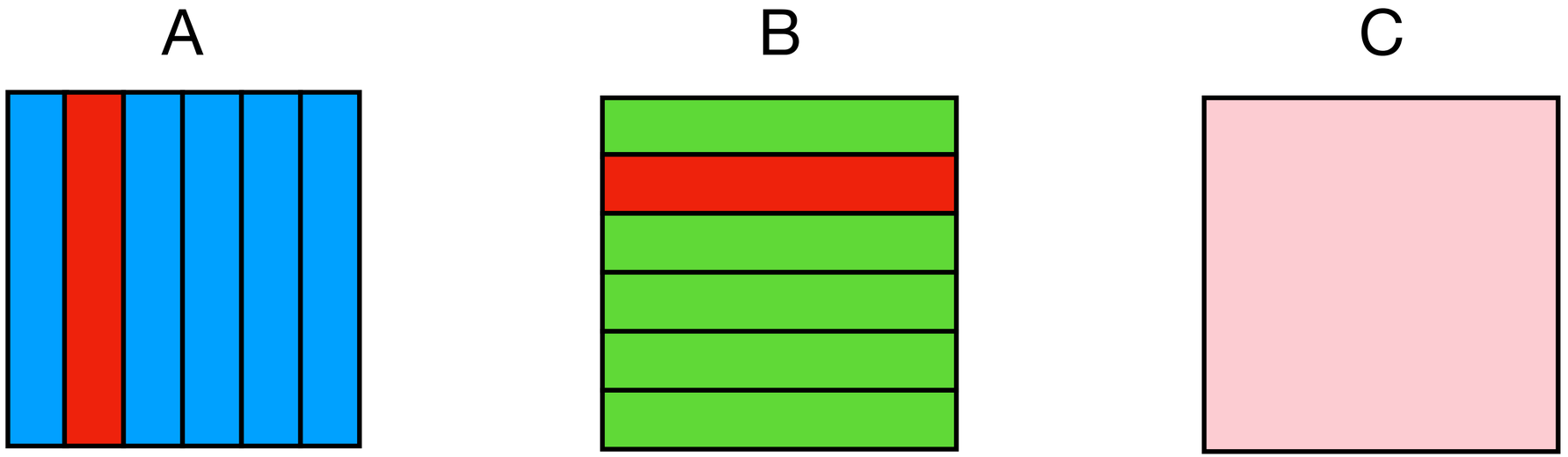
Matrix Multiply: Sequence of Outer-Products

```
for (k = 0; k < N; k ++)
    // Multiply a column of A by a row of B
    for (i = 0; i < N; i ++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```



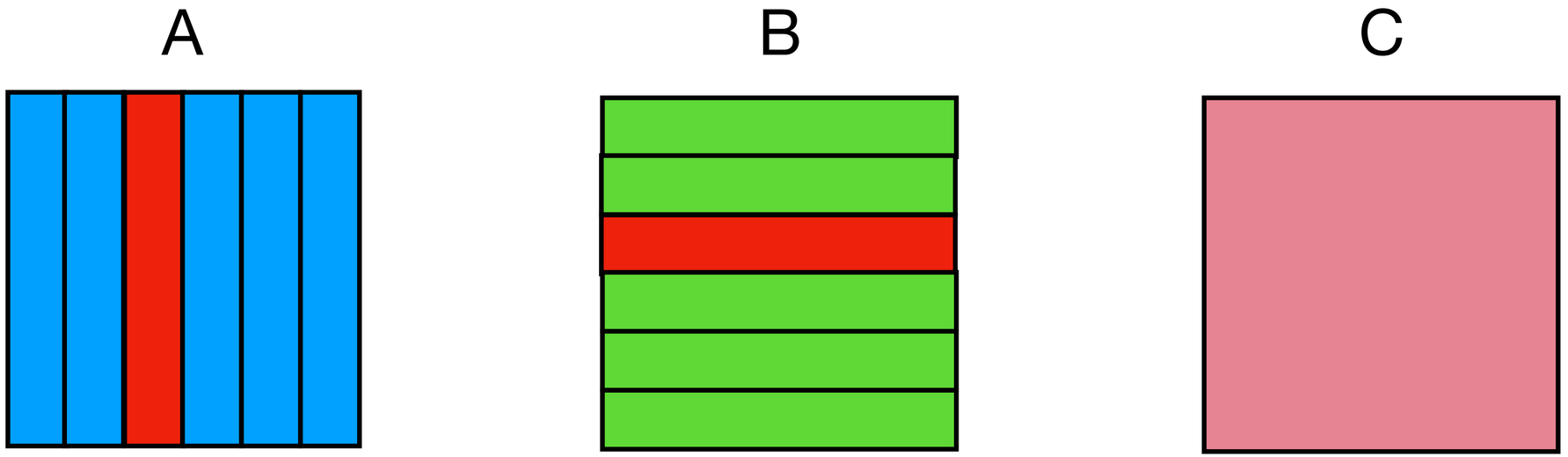
Matrix Multiply: Sequence of Outer-Products

```
for (k = 0; k < N; k ++)
    // Multiply a column of A by a row of B
    for (i = 0; i < N; i ++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```



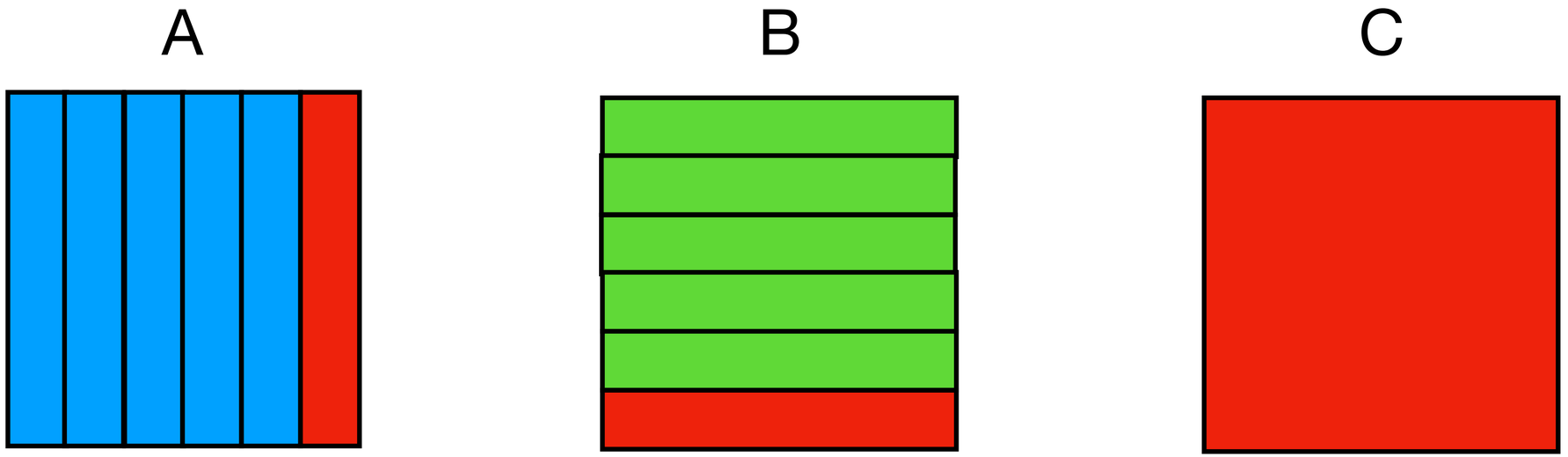
Matrix Multiply: Sequence of Outer-Products

```
for (k = 0; k < N; k ++)
    // Multiply a column of A by a row of B
    for (i = 0; i < N; i ++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```



Matrix Multiply: Sequence of Outer-Products

```
for (k = 0; k < N; k ++)
    // Multiply a column of A by a row of B
    for (i = 0; i < N; i ++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```



So What?

- Why would we care about thinking of matrix multiplication in this way?
- Because it turns out it leads to a very elegant parallel algorithm
- Let's see an example on a 5x5 torus of processors

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{0,4}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{1,4}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{2,4}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$	$B_{3,4}$
$B_{4,0}$	$B_{4,1}$	$B_{4,2}$	$B_{4,3}$	$B_{4,4}$

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{0,3}$	$C_{0,4}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$	$C_{1,4}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$	$C_{2,4}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$	$C_{3,4}$
$C_{4,0}$	$C_{4,1}$	$C_{4,2}$	$C_{4,3}$	$C_{4,4}$

- Process i,j holds $A_{i,j}$, $B_{i,j}$ and $C_{i,j}$

Algorithm steps

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{0,4}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{1,4}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{2,4}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$	$B_{3,4}$
$B_{4,0}$	$B_{4,1}$	$B_{4,2}$	$B_{4,3}$	$B_{4,4}$

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{0,3}$	$C_{0,4}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$	$C_{1,4}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$	$C_{2,4}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$	$C_{3,4}$
$C_{4,0}$	$C_{4,1}$	$C_{4,2}$	$C_{4,3}$	$C_{4,4}$

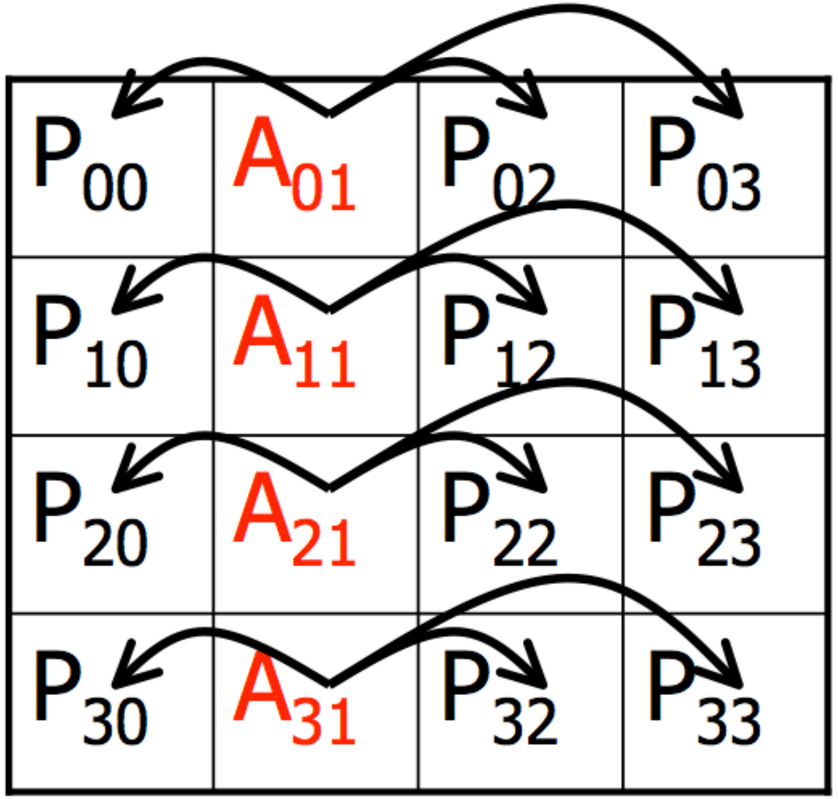
```
for (k = 0; k < N; k ++)
    // Multiply a column of A by a row of B
    for (i = 0; i < N; i ++)
        for (j = 0; j < N; j++)
            // matrix block operation
            Ci,j += Ai,k * Bk,j ;
```

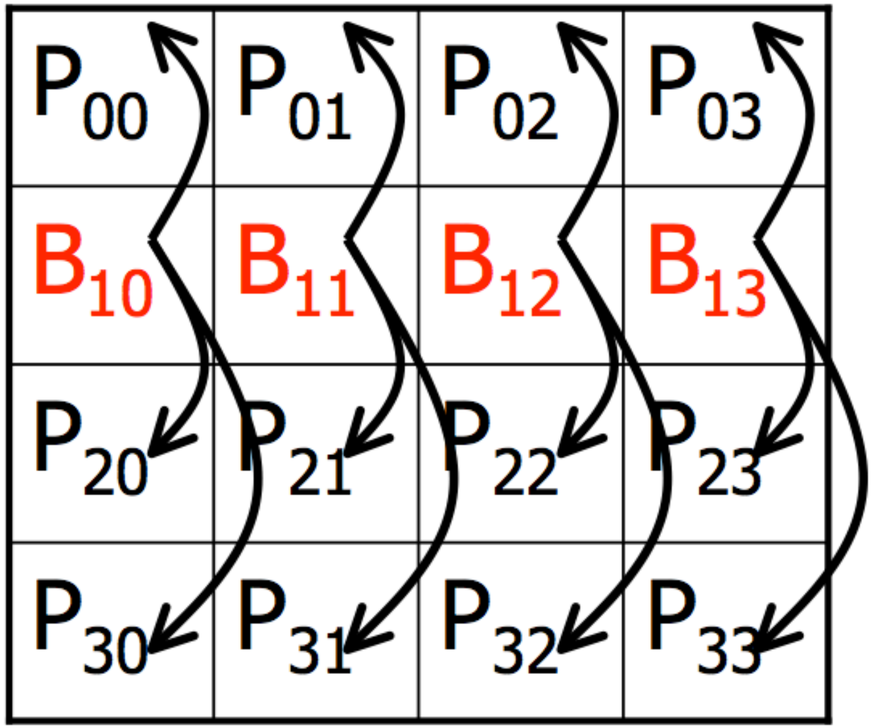
- At step k, process i,j needs $A_{i,k}$ and $B_{k,j}$ to update $C_{i,j}$
- If j=k, process i,j already has $A_{i,k}$ and it needs to send it to others
- If i=k, process i,j already has $B_{k,j}$ and it needs to send it to others

The Algorithm

- From the previous slide we had at step k:
  - If j=k, process i,j already has $A_{i,k}$ and it needs to send it to others
  - If i=k, process i,j already has $B_{k,j}$ and it needs to send it to others
- Therefore, at step k:
  - $\forall i$, process i,k must broadcast its block of A to all processes *,i
  - $\forall j$, process k,j must broadcast its block of A to all processes j,*
- Confused yet?
- Let’s see this on a picture....

Communications at step k = 1





Pseudo-Code

```
p = sqrt(num_procs());                  // assume a perfect square
(myrow, mycol) = my_2D_rank();     // get my 2-D rank based on the 1-D rank

int A[N/p][N/p], B[N/p][N/p], C[N/p][N/p];  // my blocks of A, B, and C
int bufferA[N/p][N/p], bufferB[N/p][N/p];  // for receiving blocks from others

// Go through the p steps
for (int k=0; k < p; k++) {
  // Broadcast A blocks along rows
  BroadcastRow((myrow,k), A, bufferA, N/p * N/p);  // first argument: broadcast root
  // Broadcast B blocks along columns
  BroadcastColumn((k,mycol), B, bufferB, N/p * N/p);  // first argument: broadcast root

  // Multiply Matrix blocks (assuming a convenient MatrixMultiplyAdd() function)
  if ((myrow == k) && (mycol == k))
    MatrixMultiplyAdd(C, A, B, N/p, N/p);              // I had both blocks!
  else if (myrow == k)
    MatrixMultiplyAdd(C, bufferA, B, N/p, N/p);      // I was missing the A block!
  else if (mycol == k)
    MatrixMultiplyAdd(C, A, bufferB, N/p, N/p);      // I was missing the B block!
  else
    MatrixMultiplyAdd(C, bufferA, bufferB, N/p, N/p);  // I was missing both blocks!
}
```

Torus vs. Ring?

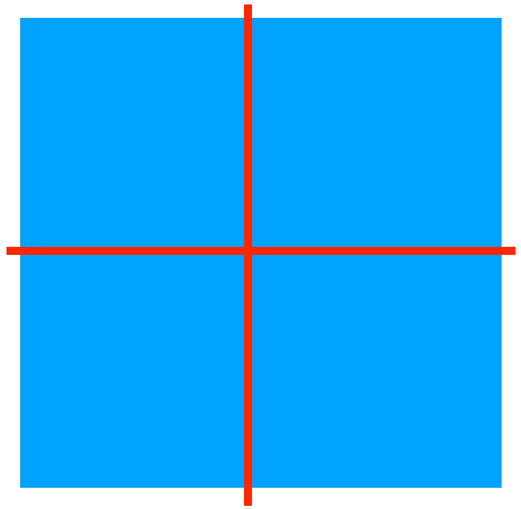
- It's very easy to show that the speedup of our algorithm is asymptotically optimal
- But we already had an asymptotically optimal algorithm on a Ring
  - For a matrix-vector multiply, but easy to augment to deal with matrix-matrix multiply, which we didn't show
- So who cares??
- If N is huge, we don't care, because asymptotic is enough
- But in fact, using the 2-D distribution reduces communication costs
  - The algorithm sends less data overall
- Also, many platforms have a torus network
- And even if not, it can be proven that the 2-D algorithm is better than the 1-D algorithm on many physical topologies, even if they are not tori!

Other Matrix Multiplication Algorithms

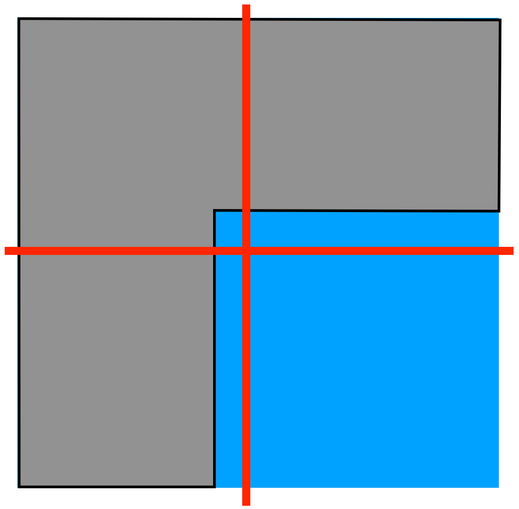
- People have come up with many different matrix multiply algorithm for 2-D data distribution
  - Cannon (1969)
  - Fox (1987)
  - Snyder (1992)
  - etc.
- They all correspond to re-organization the operations in different (possibly really confusing) orders
  - Some of these algorithms start by shuffling things around in each matrix
- You could spend months reviewing existing parallel matrix multiplication algorithms...

Problem with 1-D and 2-D distribution

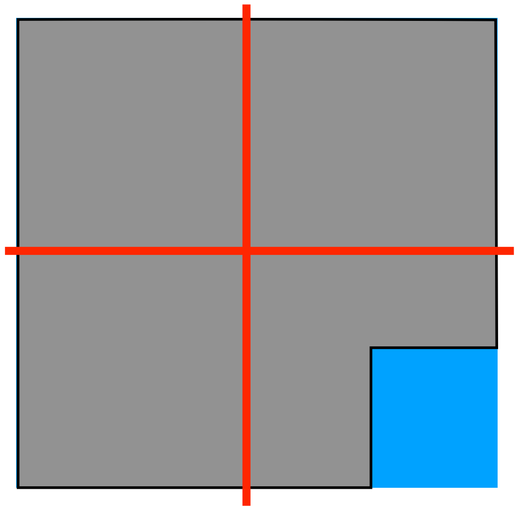
- If N >> p, then blocks are large
- Not really a problem for matrix multiplication for instance
- But some algorithms have different patterns of computation
- Some, for instance, stop working on parts of the data after a while
- The famous example: the LU factorization algorithm proceeds so that it it no longer updates top/left elements



all 4 processors work



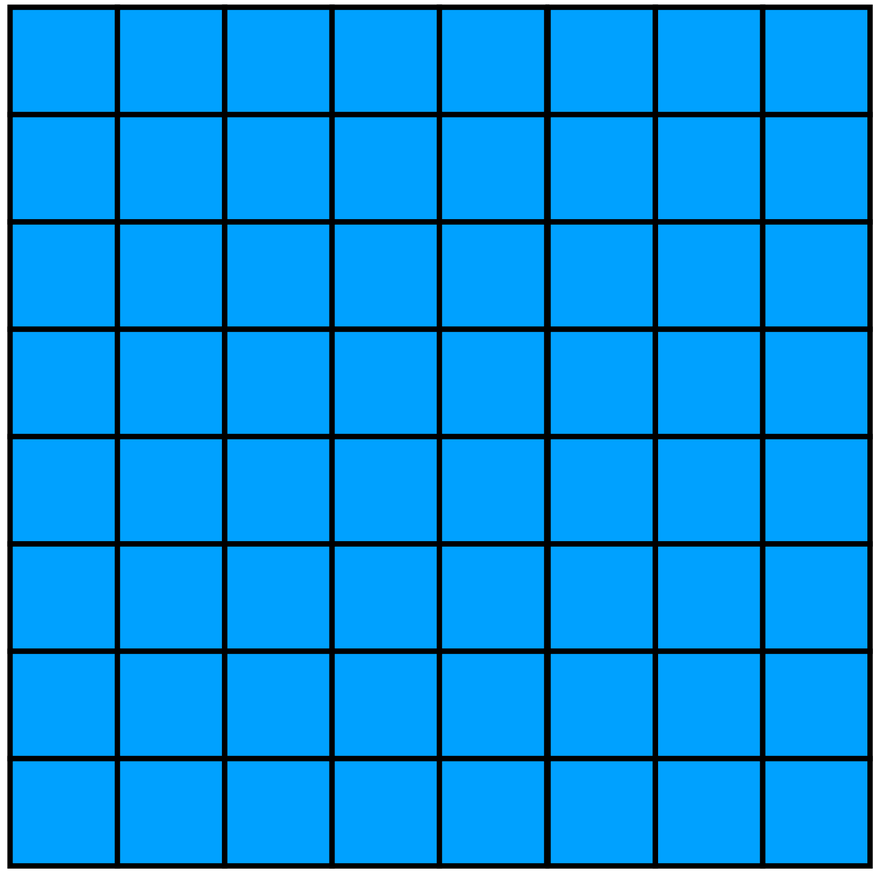
3 processes have little to do



only 1 process works!

2-D Block Cyclic Distribution

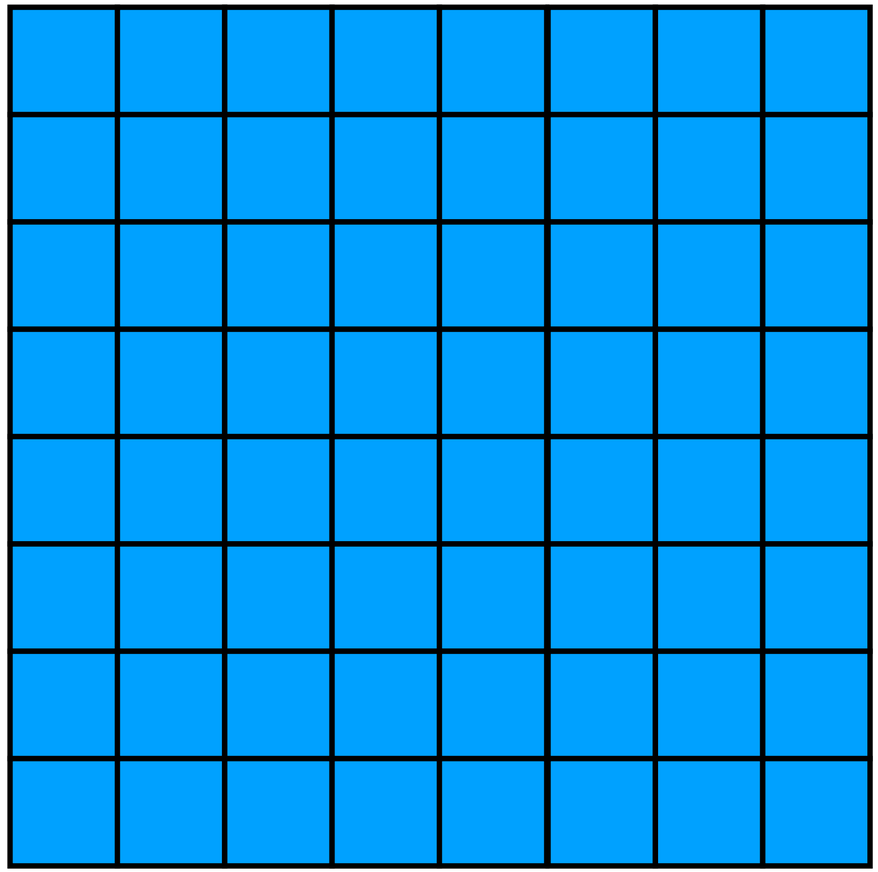
- This is a bit of a “Swiss army knife”, as it should work well regardless of the pattern of the algorithm



- Array is structured as many logical contiguous blocks, regardless of the number of processes that will be used

2-D Block Cyclic Distribution

- This is a bit of a “Swiss army knife”, as it should work well regardless of the pattern of the algorithm

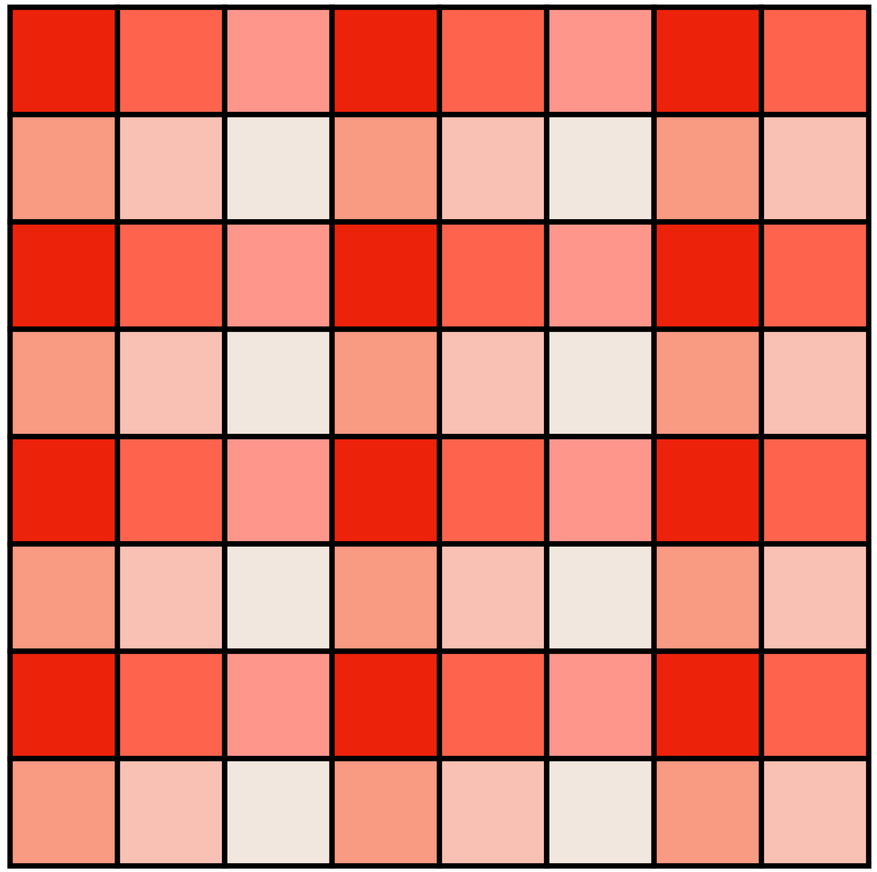


P1	P2	P3
P4	P5	P6

- Processes are organized as some 2-D grid, in this example 2x3

2-D Block Cyclic Distribution

- This is a bit of a “Swiss army knife”, as it should work well regardless of the pattern of the algorithm



P1	P2	P3
P4	P5	P6

- The “process grid” is cyclically “stamped” onto the array
- In this example P3 and P6 have fewer blocks because 3 does not divide 8
- But the point is: each process holds blocks located all over the array

2-D Block Cyclic Distribution

- All algorithms we've seen so far can be implemented using the 2-D Block Cyclic distribution
  - As done is, e.g., the ScaLAPACK library

- The local-global index translations can be quite intricate
  - It's "just" discrete math of course, but it can get ugly

- Researchers have explored many other options, which I won't be discussing here as we're trying to keep things "easy"

Conclusion

- We have two programming assignment in which you implement the outer-product algorithm using a (non-cyclic) 2-D distribution
  - It’s really a single assignment, but it’s split in to
  - First one is about correctness
  - Second one is about performance

- Let’s look at them now...