Xi'an Jiaotong-Liverpool University
西交利物浦大學

DTS205TC High Performance Computing

Lecture 9 Dense Matrix Algorithms

Di Zhang, Spring 2024

*cheatsheet for system design

- Event-driven, or batch processing?
  - Former. -》Distributed systems, stream computing, etc
- Can you predetermine the communication relationship between tasks?
  - No. Multi-agent system, peer-to-peer system
- Can the required speedup ratio be achieved in theory? (Is there enough concurrency?)
  - No. uses other algorithms, or approximate algorithms, or hardware acceleration.
- Do I have to use > 30 hardware threads (Maximum in experience) to meet the speedup ratio?
  - No. -》openMP。
- Task decomposition, or data decomposition?
  - Task decomposition. You can simply use multi-process, Pthread, etc.
- Under data decomposition, the amount of computation is unpredictable?
  - Dynamic scheduling.
- Can be predicted. Use static mapping:
  - Grid, Graph decomposition

Section

- Matrix multiplication

linear algebra

Xi'an Jiaotong-Liverpool University
西交利物浦大學

Linear algebra is the branch of mathematics concerning linear equations such as:

$$a_1 x_1 + \cdots + a_n x_n = b,$$

linear maps such as:

$$(x_1, \ldots, x_n) \mapsto a_1 x_1 + \cdots + a_n x_n,$$

and their representations in vector spaces and through matrices.[1][2][3]

- Linear algebra is central to almost all areas of mathematics.

A system of linear equations

$$\begin{cases} u = x + y + 2z \\ v = 2x + 4y - 3z \\ w = 3x + 6y - 5z \end{cases} \quad \text{with matrix} \quad A = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix}$$

- Abstraction:
  - First level: we ignore the context of a specific problem and go straight to the system of equations
    Second level: we ignore variable naming and look directly at linear mappings. We found that the map itself can also do operations, for example: multiplication!

Matrix multiplication

$$\begin{cases} u &= x &+y &+2z \\ v &= 2x &+4y &-3z \\ w &= 3x &+6y &-5z \end{cases} \qquad \text{with matrix} \quad A = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix}$$

- we substitute in turn linear expressions for each of x, y, and z

$$\begin{cases} x &= s &+2t \\ y &= 3s &-t \\ z &= -s &+t \end{cases} \qquad \text{with matrix} \quad B = \begin{bmatrix} 1 & 2 \\ 3 & -1 \\ -1 & 1 \end{bmatrix}$$

- We see that u, v, and w then become linear expressions in s and t

$$\begin{cases} u &= x+y+2z &= s+2t+3s-t-2s+2t &= 2s+3t \\ v &= 2x+4y-3z &= 2s+4t+12s-4t+3s-3t &= 17s-3t \\ w &= 3x+6y-5z &= 3s+6t+18s-6t+5s-5t &= 26s-5t \end{cases}$$

Matrix multiplication

Xi'an Jiaotong-Liverpool University 西交利物浦大學

$$AB = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & -1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 17 & -3 \\ 26 & -5 \end{bmatrix}$$

- From this example, we can define matrix multiplication

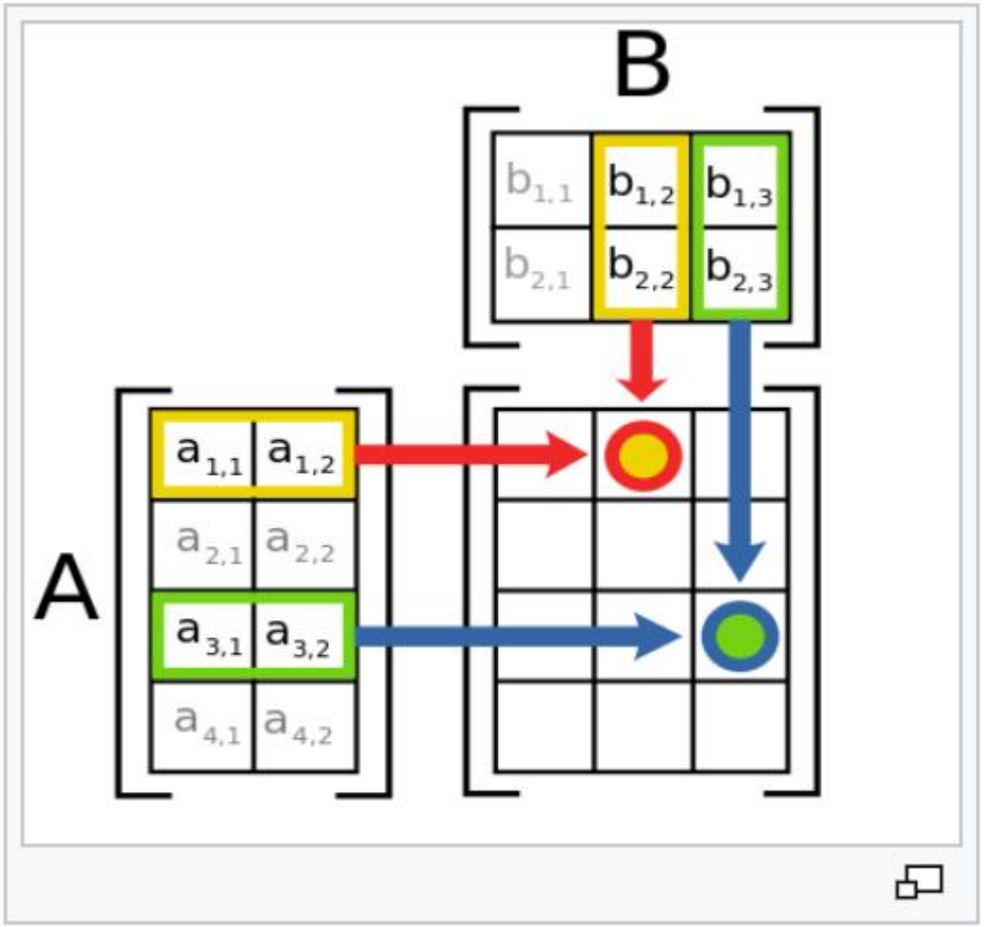
$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Matrix multiplication

Xi'an Jiaotong-Liverpool University
西交利物浦大學

$$\underset{4\times 2 \text{ matrix}}{\begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix}} \underset{2\times 3 \text{ matrix}}{\begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix}} = \underset{4\times 3 \text{ matrix}}{\begin{bmatrix} \cdot & c_{12} & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & c_{33} \\ \cdot & \cdot & \cdot \end{bmatrix}}$$



- Features of matrix multiplication:
  - For multiplying matrices, their dimensions must match
    It satisfies the law of associativity and distribution, but not the commutative law

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

$$\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC} \qquad (\mathbf{B}+\mathbf{C})\mathbf{D} = \mathbf{BD}+\mathbf{CD}$$

$$\mathbf{AB} \neq \mathbf{BA}$$

Block matrix

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$\left[\begin{array}{cc|ccc} 0 & 2 & 3 & 3 & 3 \\ 2 & 0 & 3 & 3 & 3 \\ \hline 4 & 4 & 5 & 0 & 5 \\ 4 & 4 & 0 & 5 & 0 \\ 4 & 4 & 5 & 0 & 5 \end{array}\right].$$

$$A=\begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix}$$

$$B=\begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}$$

$$C=\begin{bmatrix} 4 & 4 \\ 4 & 4 \\ 4 & 4 \end{bmatrix}$$

$$D=\begin{bmatrix} 5 & 0 & 5 \\ 0 & 5 & 0 \\ 5 & 0 & 5 \end{bmatrix};$$

Corresponding substitution relationship:

- Y1 = A1 + B1
- Y2 = A2 + B2
- Y3 = C1 + D1
- …
- A1 = 0 X1 + 2 X2

Block matrix multiplication

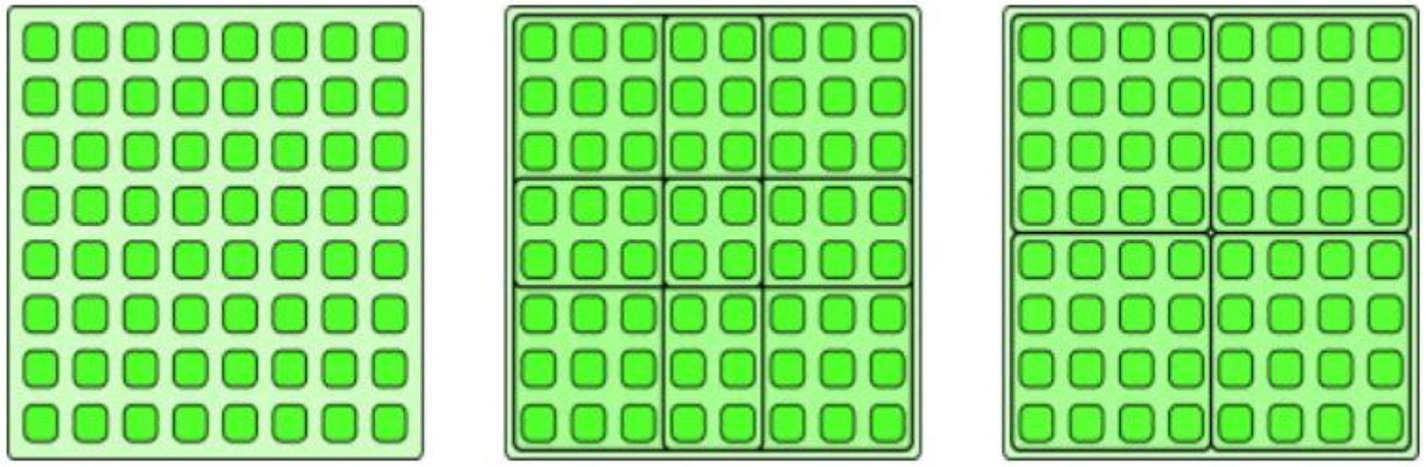
$$\begin{bmatrix} A_1 & B_1 \\ C_1 & D_1 \end{bmatrix}\begin{bmatrix} A_2 & B_2 \\ C_2 & D_2 \end{bmatrix}$$

$$= \begin{bmatrix} A_1 A_2 + B_1 C_2 & A_1 B_2 + B_1 D_2 \\ C_1 A_2 + D_1 C_2 & C_1 B_2 + D_1 D_2 \end{bmatrix}.$$

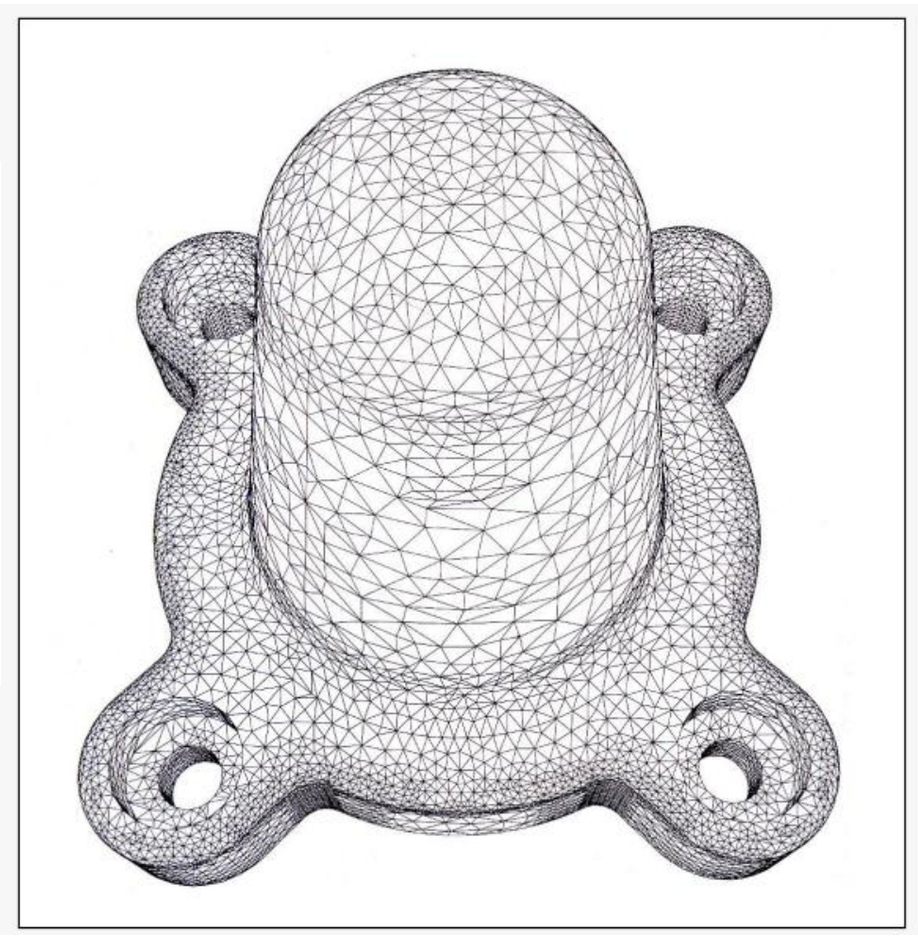
- In short, if we treat the matrix (linear map) itself as an object and as an element of another matrix, then the properties of the latter with respect to the matrix will be preserved entirely!
  * It has now been verified that putting real numbers, complex numbers, and matrices into the matrix as objects does not destroy its properties. But not any object is OK, such as natural numbers, positive real numbers. So how to check it? This problem needs to be studied in a more advanced mathematics course (abstract algebra). Matrix ring - Wikipedia

Parallelization Models

- geometric-decomposition



2-D Grid

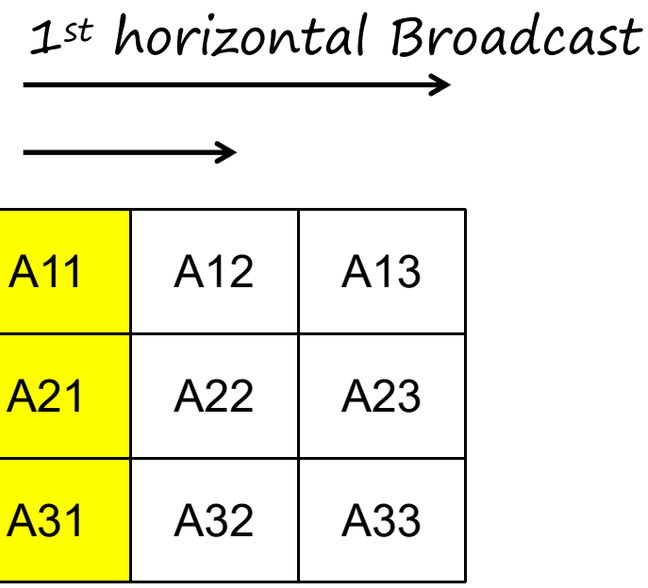


Irregular Mesh

- Features:
  - Based on the intrinsic topology of the data, it is segmented and assigned to different compute nodes
    Each node performs all calculations involving the data it owns
  - The communication process is predictable and is not affected by the calculation process

Simple Parallel Algorithm

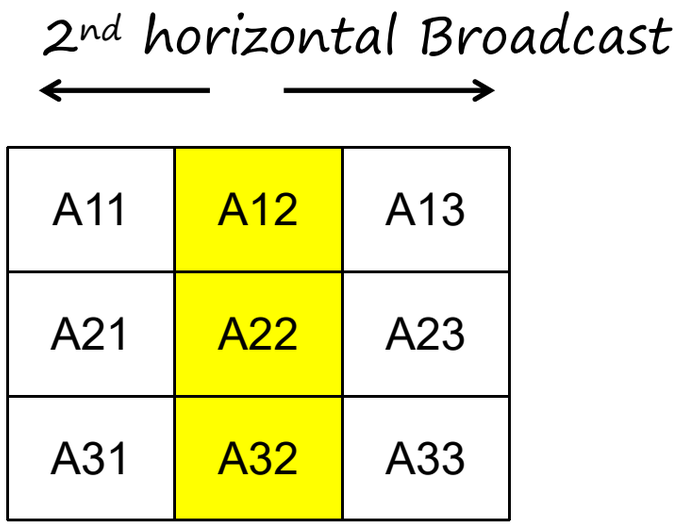
Xi'an Jiaotong-Liverpool University
西交利物浦大學



- Stages:
  - Store blocks A _{i,j}, B_{i,j} on the corresponding nodes
    Each node broadcasts its own A_{i,j}, B_{i,j} along the horizontal and vertical axes; After the broadcast, each node has all blocks of data in the same column and row
    - Step 1

Simple Parallel Algorithm

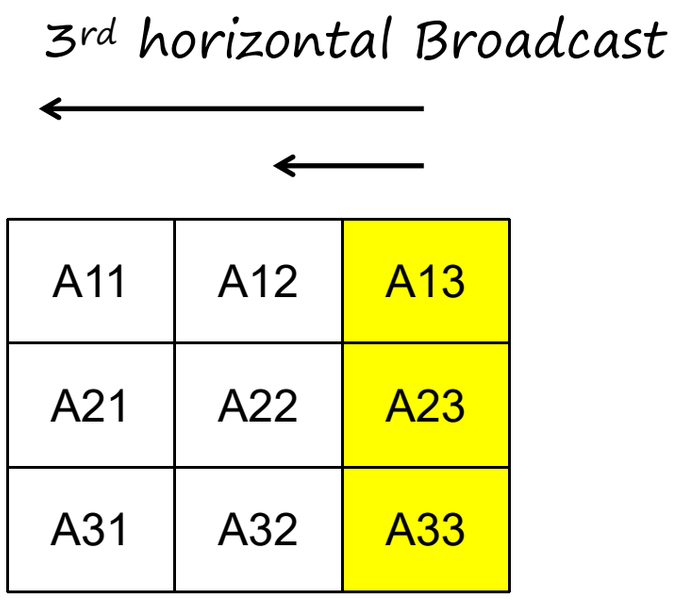
Xi'an Jiaotong-Liverpool University 西交利物浦大學



- Stages:
  - Each node broadcasts its own A_{i,j}, B_{i,j} along the horizontal and vertical axes; After the broadcast, each node has all blocks of data in the same column and row
    - Step 2

Simple Parallel Algorithm

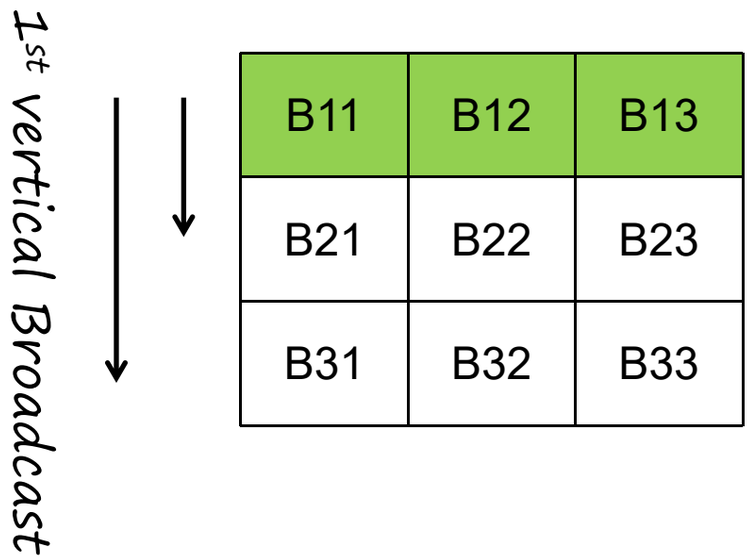
Xi'an Jiaotong-Liverpool University 西交利物浦大學



- Stages:
  - Each node broadcasts its own A_{i,j}, B_{i,j} along the horizontal and vertical axes; After the broadcast, each node has all blocks of data in the same column and row
    - Step 3

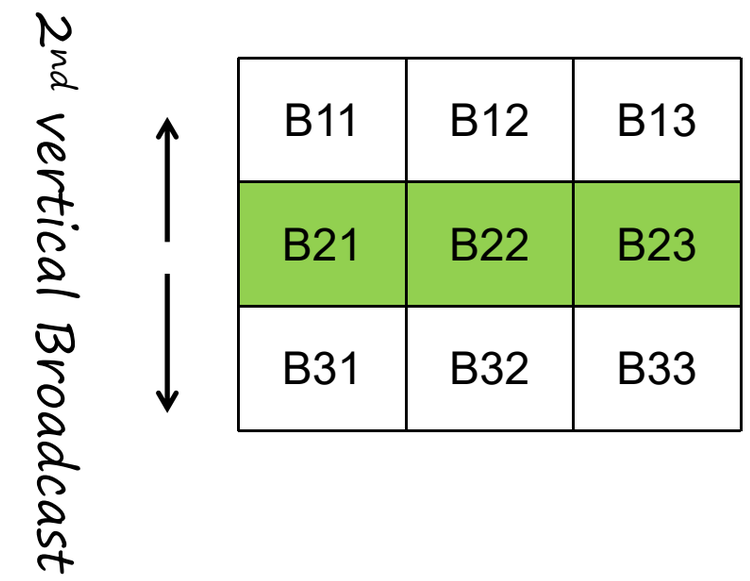
Simple Parallel Algorithm

Xi'an Jiaotong-Liverpool University
西交利物浦大學



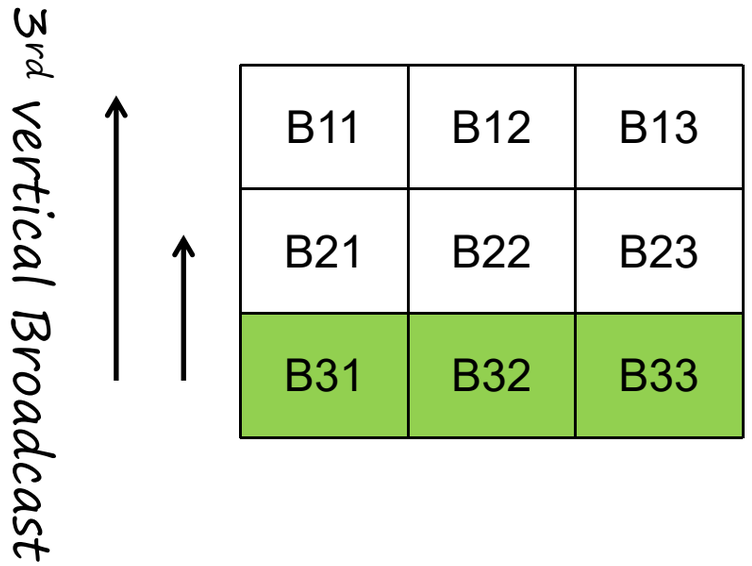
- Stages:
  - Each node broadcasts its own A_{i,j}, B_{i,j} along the horizontal and vertical axes; After the broadcast, each node has all blocks of data in the same column and row
    - Step 4

Simple Parallel Algorithm



- Stages:
  - Each node broadcasts its own A_{i,j}, B_{i,j} along the horizontal and vertical axes; After the broadcast, each node has all blocks of data in the same column and row
    - Step 5

Simple Parallel Algorithm



- Stages:
  - Each node broadcasts its own A_{i,j}, B_{i,j} along the horizontal and vertical axes; After the broadcast, each node has all blocks of data in the same column and row
    - Step 6

Simple Parallel Algorithm

Xi'an Jiaotong-Liverpool University 西交利物浦大學

C11 needs these...

A11	A12	A13
A21	A22	A23
A31	A32	A33

C11 needs these...

B11	B12	B13
B21	B22	B23
B31	B32	B33

- Stages:
  - Each node calculates C_ {i,j}, for example
    - C11 = A11 B11 + A12 B21 + A13 B31

code


```
# environment info
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nproc = comm.Get_size()
assert nproc == int(nproc ** (1 / 2)) ** 2  # should be a squared number
```

- Get the MPI environment
  - Note that the number of processes must be squared

code


```
20    # construct matrices
21    X = natmat(L)
22    Y = natmat(L)
23    Z_ = np.matmul(X, Y)
24
25    # initial distribution of sub-matrix on processes
26    X = split(X, B)[rank // B][rank % B]
27    Y = split(Y, B)[rank // B][rank % B]
28    Z_ = split(Z_, B)[rank // B][rank % B]
```

- Generate data
  - Initially, all nodes generate data, but only intercept what they need to keep
    Note: In practice, input data is distributed from node 0 at first, but here it is simplified for practice

Process topology

Xi'an Jiaotong-Liverpool University 西交利物浦大學

```
30  # topology among processes
31  cart_comm = comm.Create_cart(dims=[B, B], periods=[False, False]) # global
32  row_comm = cart_comm.Sub(remain_dims=[False, True]) # row-wise
33  col_comm = cart_comm.Sub(remain_dims=[True, False]) # column-wise
```

- The above code defines a Cartesian process topology that does the following:
  - Two-dimensional coordinates are provided to facilitate positioning processes

```
proc= 2 ,topo= ([3, 3], [0, 0], [0, 2])
proc= 1 ,topo= ([3, 3], [0, 0], [0, 1])
proc= 8 ,topo= ([3, 3], [0, 0], [2, 2])
proc= 0 ,topo= ([3, 3], [0, 0], [0, 0])
proc= 3 ,topo= ([3, 3], [0, 0], [1, 0])
proc= 7 ,topo= ([3, 3], [0, 0], [2, 1])
proc= 6 ,topo= ([3, 3], [0, 0], [2, 0])
proc= 5 ,topo= ([3, 3], [0, 0], [1, 2])
proc= 4 ,topo= ([3, 3], [0, 0], [1, 1])
```

0 (0,0)	1 (0,1)	2 (0,2)
3 (1,0)	4 (1,1)	5 (1,2)
6 (2,0)	7 (2,1)	8 (2,2)

topo = (size of grid, periodic, coordinates)

Explain later

Process topology

Xi'an Jiaotong-Liverpool University 西交利物浦大學

```
30    # topology among processes
31    cart_comm = comm.Create_cart(dims=[B, B], periods=[False, False]) # global
32    row_comm = cart_comm.Sub(remain_dims=[False, True]) # row-wise
33    col_comm = cart_comm.Sub(remain_dims=[True, False]) # column-wise
```

- Turn the grid into lower-dimensional subgrids

```
a@a:~/PycharmProjects/try_mpi$ mpirun -np 9 python task42.py
proc= 2 ,row topo= ([3], [0], [2])
proc= 4 ,row topo= ([3], [0], [1])
proc= 5 ,row topo= ([3], [0], [2])
proc= 0 ,row topo= ([3], [0], [0])
proc= 6 ,row topo= ([3], [0], [0])
proc= 3 ,row topo= ([3], [0], [0])
proc= 1 ,row topo= ([3], [0], [1])
proc= 7 ,row topo= ([3], [0], [1])
proc= 8 ,row topo= ([3], [0], [2])
a@a:~/PycharmProjects/try_mpi$ mpirun -np 9 python task42.py
proc= 1 ,col topo= ([3], [0], [0])
proc= 2 ,col topo= ([3], [0], [0])
proc= 5 ,col topo= ([3], [0], [1])
proc= 0 ,col topo= ([3], [0], [0])
proc= 8 ,col topo= ([3], [0], [2])
proc= 7 ,col topo= ([3], [0], [2])
proc= 6 ,col topo= ([3], [0], [2])
proc= 3 ,col topo= ([3], [0], [1])
proc= 4 ,col topo= ([3], [0], [1])
```

0 (0)	1 (1)	2 (2)
3 (0)	4 (1)	5 (2)
6 (0)	7 (1)	8 (2)

row_comm: Only the second coordinate is retained

0 (0)	1 (0)	2 (0)
3 (1)	4 (1)	5 (1)
6 (2)	7 (2)	8 (2)

col_comm: Only the first coordinate is retained

topo = (size of grid, periodic, coordinates)

Explain later

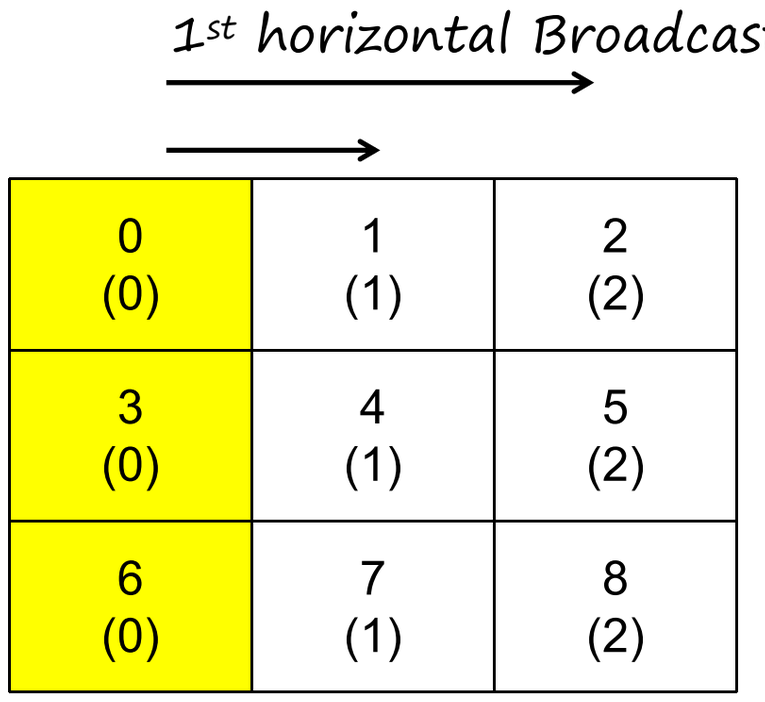
Process topology


```
30  # topology among processes
31  cart_comm = comm.Create_cart(dims=[B, B], periods=[False, False]) # global
32  row_comm = cart_comm.Sub(remain_dims=[False, True]) # row-wise
33  col_comm = cart_comm.Sub(remain_dims=[True, False]) # column-wise
```

- Each grid has a communicator object associated with it, which can be used for local broadcasting. For example

```
row_comm.bcast(X, root=i)
```

- It sends X horizontally from all nodes with coordinates i to other nodes



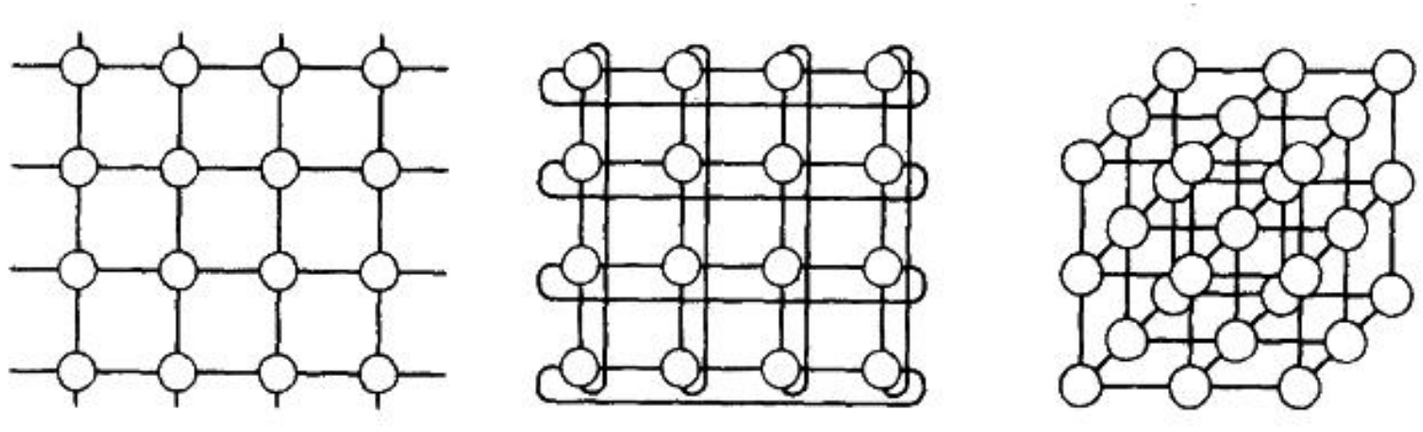
Process topology

Xi'an Jiaotong-Liverpool University 西交利物浦大學

- Finally, if I don't use *bcast*, I'll use point-to-point sending, okay?

0 (0)	1 (1)	2 (2)
3 (0)	4 (1)	5 (2)
6 (0)	7 (1)	8 (2)

- Of course. For example, in the figure above, we can call send/recv six times to complete the data transfer. (If you write this in Lab 4, you can get part of points)
  But the point is, if you define topology explicitly, the MPI library has the potential to automatically match your processes to the most suitable location in the hardware network! Otherwise, the MPI library can only be mapped randomly, which affects performance!



These network topologies are especially suitable for matrix operations!

Cannon’s algorithm

Xi’an Jiaotong-Liverpool University 西交利物浦大學

- For the simple algorithm introduced earlier, one of its drawbacks is that it takes up too much storage space. That is, each node stores all the blocks of data in the same row and column. To improve this, consider alternating transmission and computation. This is Cannon's algorithm.
  Steps：
  - 1. Distribute, this is the same as before

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

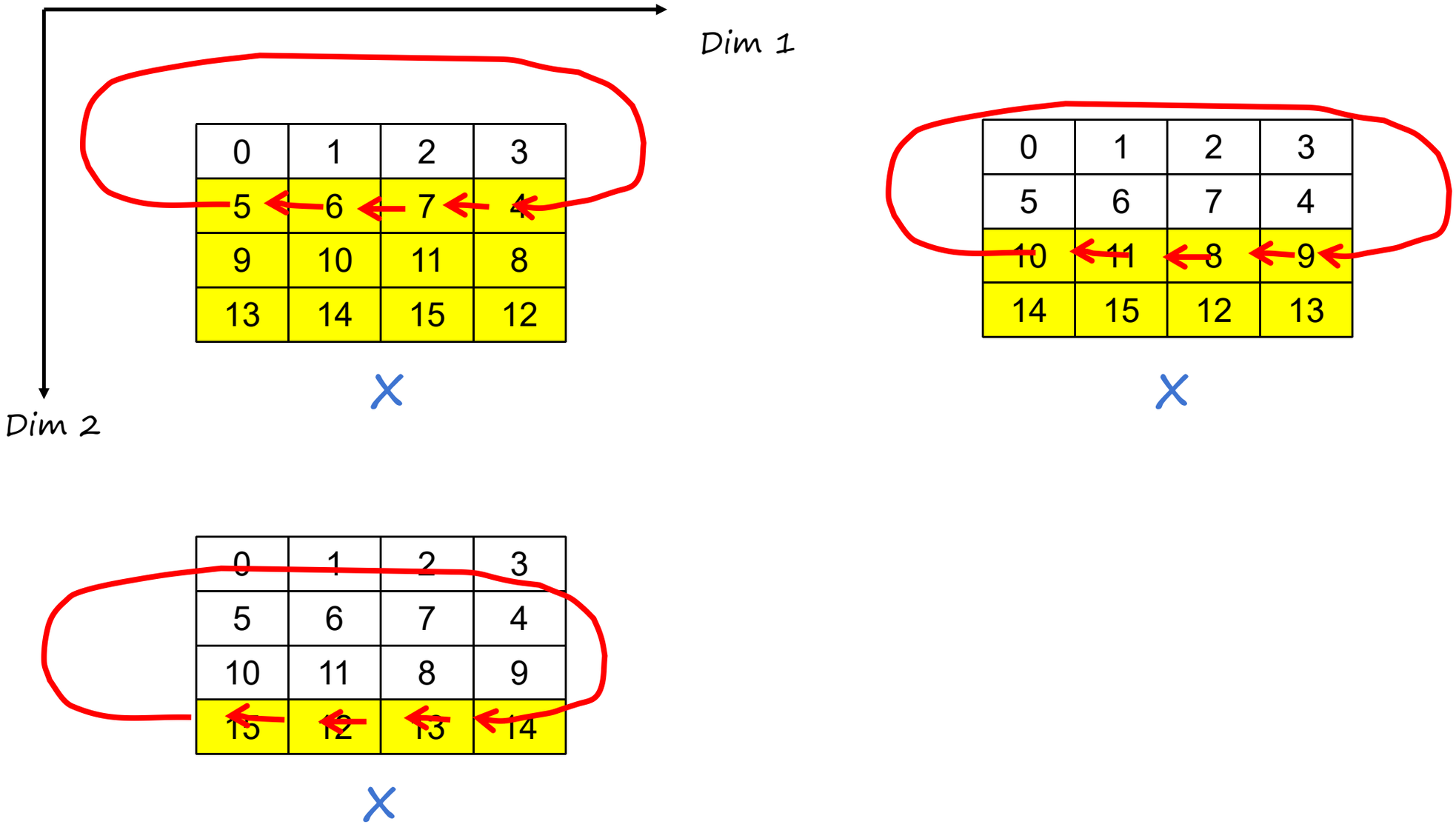
X

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Y

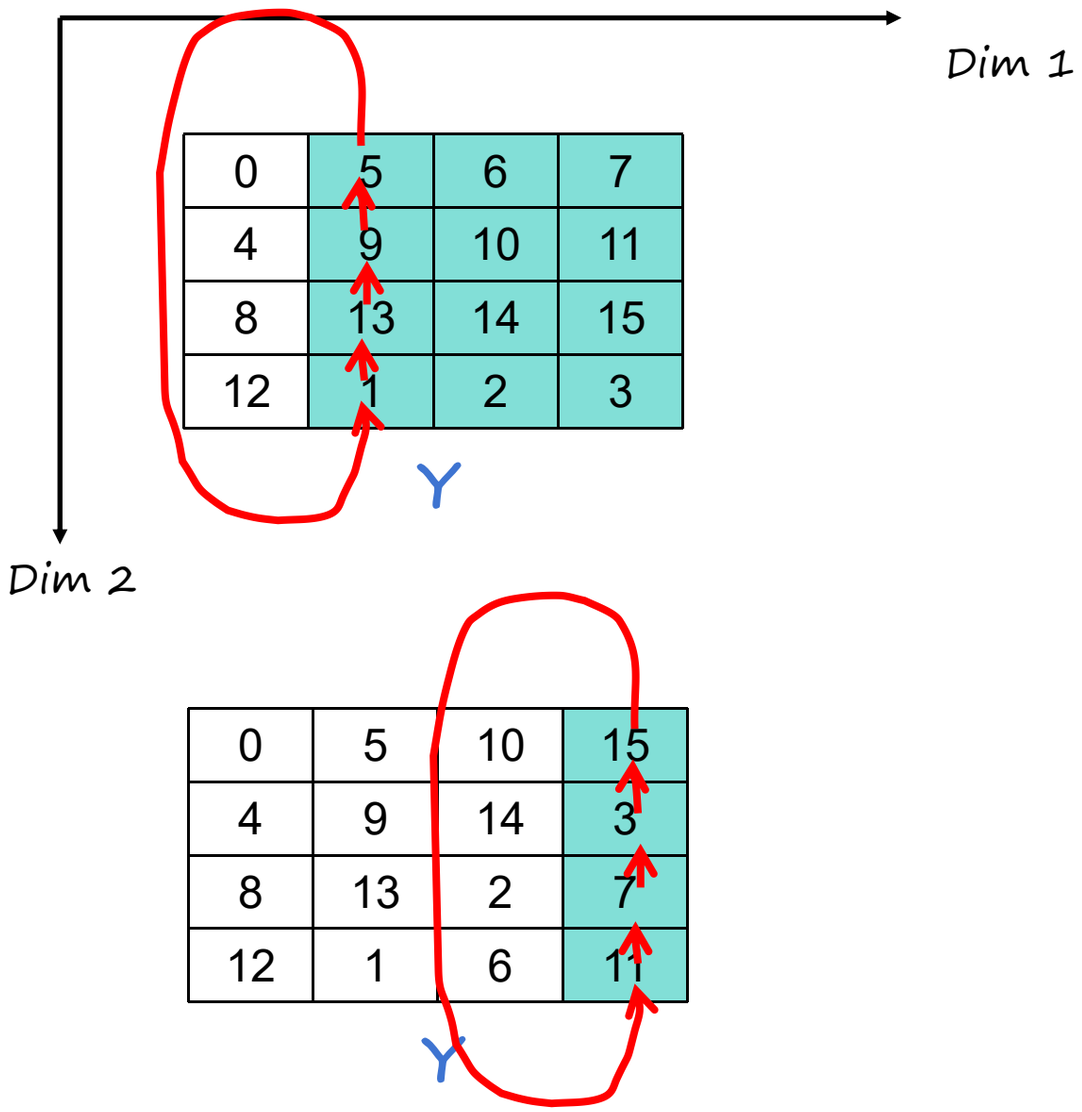
  - 2. Rearrange

Rearrange



- First rearrange X horizontally: cyclically rotate to left by times equaling to dim 2

Rearrange



- then rearrange X vertically: cyclically rotate to up by times equaling to dim 1

compute

0	1	2	3
5	6	7	4
10	11	8	9
15	12	13	14

X

0	5	10	15
4	9	14	3
8	13	2	7
12	1	6	11

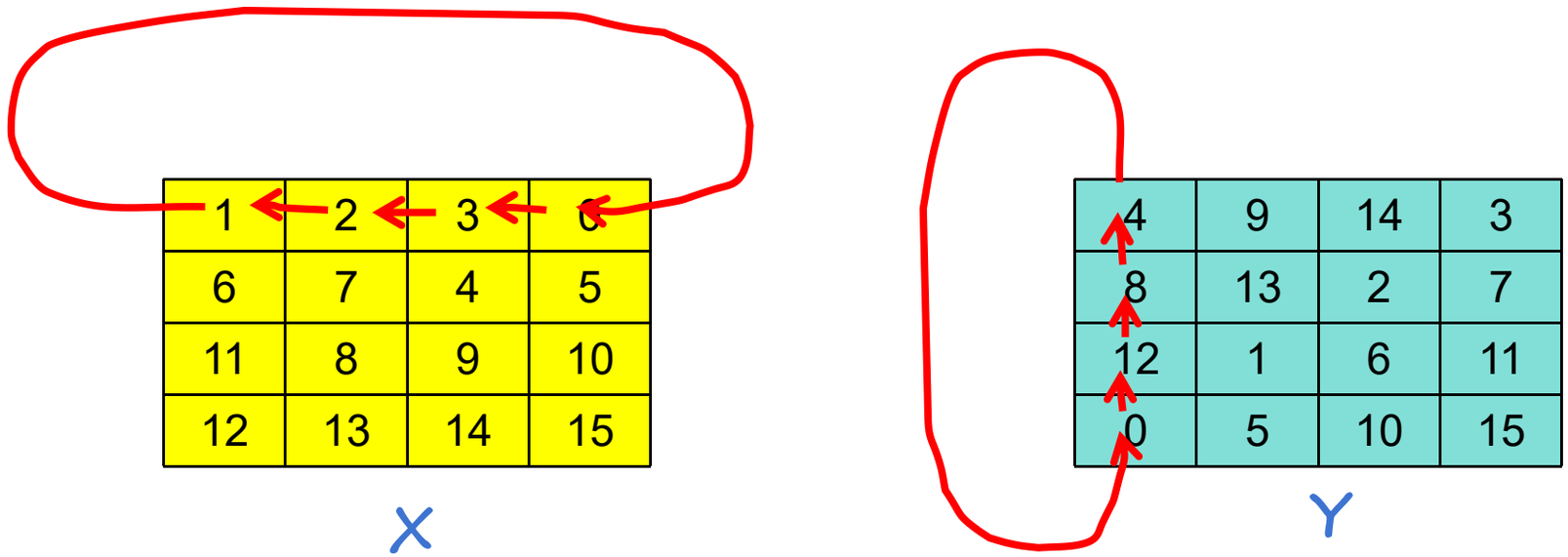
Y

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Z

- Multiply the data X and Y owned by this node
  They happen to be part of Z=XY, e.g.
  - Z0 = X0 Y0 + X1 Y4 + X2 Y8 + X3 Y12
  - Z5 = X4 Y1 + X5 Y5 + X6 Y9 + X7 Y13
  - Z14 = X12 Y2 + X13 Y6 + X14 Y10 + X15 Y14

Rotate as a whole



- Rotate to the left for X as a whole and upward for Y as a whole

compute

1	2	3	0
6	7	4	5
11	8	9	10
12	13	14	15

X

4	9	14	3
8	13	2	7
12	1	6	11
0	5	10	15

Y

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Z

- They happen to be **another** part of Z=XY, e.g.
  Z0 = X0 Y0 + X1 Y4 + X2 Y8 + X3 Y12
  - Z5 = X4 Y1 + X5 Y5 + X6 Y9 + X7 Y13
  - Z14 = X12 Y2 + X13 Y6 + X14 Y10 + X15 Y14

iterate

- Repeat the calculation + rotate process B times, and the data retained by each node at the end is what we want

code

- The rotation operations used here are directly supported in MPI. It requires: 1) define a looping topolopy,

```
# topology among processes
cart_comm = comm.Create_cart(dims=[B,B],periods=[True,True])
row_comm = cart_comm.Sub(remain_dims=[False, True])
col_comm = cart_comm.Sub(remain_dims=[True, False])
```

- 2) Use the shift method to query the source and destination of the data stored in the rotation of this node

```
sd_row = cart_comm.Shift(1, -1)
sd_col = cart_comm.Shift(0, -1)
```

- For example, for node 5, for X (horizontal), it needs to send to 4 and receive 6; For Y (vertical), it needs to go to 1 and receive 9

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Sample output

Xi'an Jiaotong-Liverpool University
西交利物浦大學

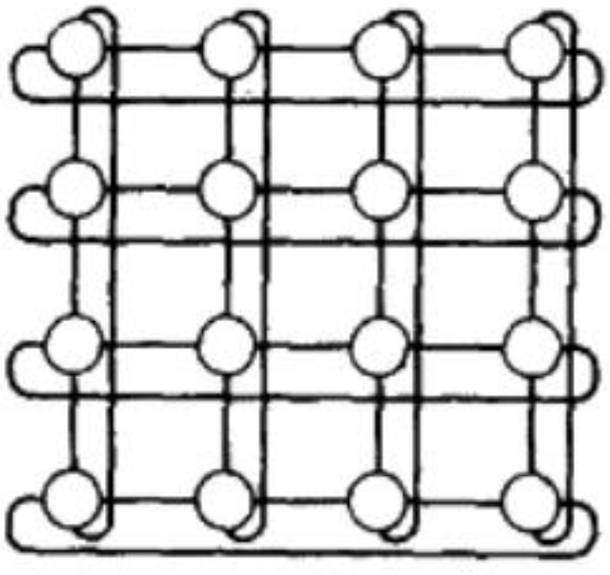
```
a@a:~/PycharmProjects/try_mpi$ mpirun -np 16 python task43.py
proc= 3 ,source= 0 ,dest= 2 along row
proc= 1 ,source= 2 ,dest= 0 along row
proc= 0 ,source= 1 ,dest= 3 along row
proc= 2 ,source= 3 ,dest= 1 along row
proc= 8 ,source= 9 ,dest= 11 along row
proc= 9 ,source= 10 ,dest= 8 along row
proc= 7 ,source= 4 ,dest= 6 along row
proc= 11 ,source= 8 ,dest= 10 along row
proc= 13 ,source= 14 ,dest= 12 along row
proc= 5 ,source= 6 ,dest= 4 along row
proc= 4 ,source= 5 ,dest= 7 along row
proc= 6 ,source= 7 ,dest= 5 along row
proc= 10 ,source= 11 ,dest= 9 along row
proc= 15 ,source= 12 ,dest= 14 along row
proc= 14 ,source= 15 ,dest= 13 along row
proc= 12 ,source= 13 ,dest= 15 along row
```

```
a@a:~/PycharmProjects/try_mpi$ mpirun -np 16 python task43.py
proc= 1 ,source= 5 ,dest= 13 along col
proc= 5 ,source= 9 ,dest= 1 along col
proc= 13 ,source= 1 ,dest= 9 along col
proc= 3 ,source= 7 ,dest= 15 along col
proc= 0 ,source= 4 ,dest= 12 along col
proc= 7 ,source= 11 ,dest= 3 along col
proc= 4 ,source= 8 ,dest= 0 along col
proc= 9 ,source= 13 ,dest= 5 along col
proc= 11 ,source= 15 ,dest= 7 along col
proc= 8 ,source= 12 ,dest= 4 along col
proc= 12 ,source= 0 ,dest= 8 along col
proc= 15 ,source= 3 ,dest= 11 along col
proc= 6 ,source= 10 ,dest= 2 along col
proc= 14 ,source= 2 ,dest= 10 along col
proc= 10 ,source= 14 ,dest= 6 along col
proc= 2 ,source= 6 ,dest= 14 along col
```

code

- 3) MPI relay communication: Trigger multiple transmit and receive processes simultaneously. This function not only makes the rotation operations simple, but also has the possibility of optimizing them with specific hardware.

```
X = row_comm.sendrecv(X, source=sd_row_init[0], dest=sd_row_init[1])
```



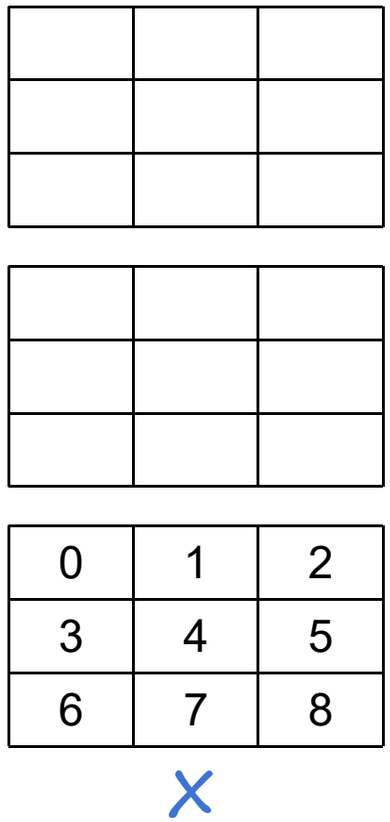
2-D wraparound grid

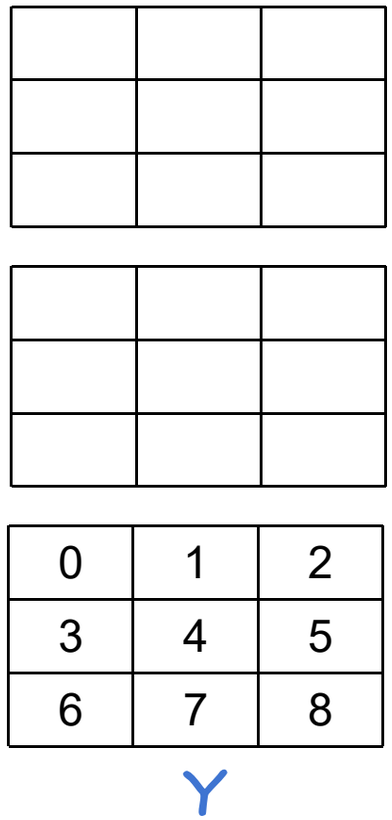
- *If the software-defined topology and hardware network match, the resources will be fully utilized!

DNS algorithm

Xi'an Jiaotong-Liverpool University 西交利物浦大學

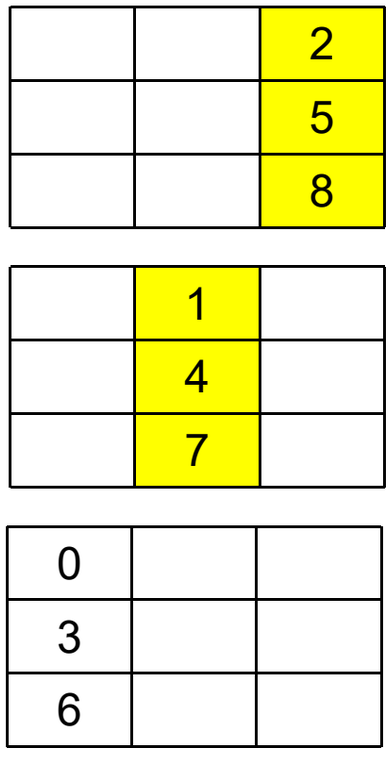
- Simple and Connon's algorithms, using O(N^2) processors - can it be faster?
  Yes! With O(N^3) processors, The program can be further accelerated.
  The basic idea is to use nodes to process intermediate data (has N^3), not inputs or outputs (only N^2).
- Steps:
  - First of all, put the data to the 0~N^2-1 node, which is the same as before

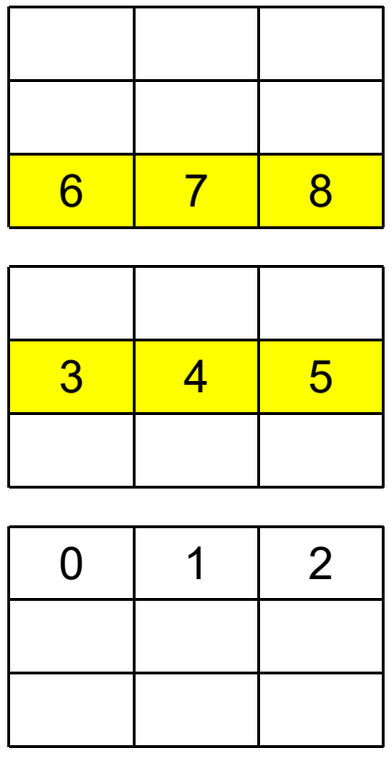




Propagation

- 1) Send different columns of A and different rows of B to different planes





X Y

Propagation

- 2) On every planes, broadcast existing data

2	2	2
5	5	5
8	8	8

6	7	8
6	7	8
6	7	8

1	1	1
4	4	4
7	7	7

3	4	5
3	4	5
3	4	5

0	0	0
3	3	3
6	6	6

0	1	2
0	1	2
0	1	2

X

Y

```
a@a:~/PycharmProjects/try_mpi$ mpirun -np 27 python task44.py
proc= 0 ,block of X= [[0.]]
proc= 3 ,block of X= [[3.]]
proc= 5 ,block of X= [[3.]]
proc= 1 ,block of X= [[0.]]
proc= 2 ,block of X= [[0.]]
proc= 4 ,block of X= [[3.]]
proc= 18 ,block of X= [[2.]]
proc= 20 ,block of X= [[2.]]
proc= 6 ,block of X= [[6.]]
proc= 7 ,block of X= [[6.]]
proc= 12 ,block of X= [[4.]]
proc= 13 ,block of X= [[4.]]
proc= 21 ,block of X= [[5.]]
proc= 22 ,block of X= [[5.]]
proc= 14 ,block of X= [[4.]]
proc= 10 ,block of X= [[1.]]
proc= 9 ,block of X= [[1.]]
proc= 19 ,block of X= [[2.]]
proc= 15 ,block of X= [[7.]]
proc= 23 ,block of X= [[5.]]
proc= 17 ,block of X= [[7.]]
proc= 11 ,block of X= [[1.]]
proc= 16 ,block of X= [[7.]]
proc= 8 ,block of X= [[6.]]
proc= 24 ,block of X= [[8.]]
proc= 26 ,block of X= [[8.]]
proc= 25 ,block of X= [[8.]]
```

multiplication

Xi'an Jiaotong-Liverpool University 西交利物浦大學

2	2	2
5	5	5
8	8	8

1	1	1
4	4	4
7	7	7

0	0	0
3	3	3
6	6	6

X

×

6	7	8
6	7	8
6	7	8

3	4	5
3	4	5
3	4	5

0	1	2
0	1	2
0	1	2

Y

=

0	1	2
3	4	5
6	7	8

Z

- At this time, in each node, there are exactly the data blocks required for each multiplication, a total of which is N^3. Multiply it!
  - Z1 = X0 Y1 + X1 Y4 + X2 Y7
  - Z5 = X3 Y2 + X4 Y5 + X5 Y8
  - Z6 = X6 Y0 + X7 Y3 + X8 Y6

Reduction

- 3）For results on different planes, perform reduction (sum) operations. In plane 0, that is, what we want.

```
Z = dim1_comm.reduce(Z, op=MPI.SUM, root=0)
```

- *Understand this algorithm from a coder's perspective: 1) Loop unfolding. If we rewrite

```
# call self-defined multiplication
Z = np.zeros((L, L), dtype=int)
for i in range(L):
    for j in range(L):
        for k in range(L):
            Z[i][j] += X[i][k] * Y[k][j]
print('Z=\n',Z)
```

- as
  - for i in range(N**3): ….(omit)…
- 2）Parallel execution. For unrelated loops, a parallel step can be done.
  3）Reduction. A O(logN) step is required (explained later).

Solving a System of Linear Equations

- Consider the problem of solving linear equations of the kind:

$$
\begin{aligned}
a_{0,0}x_0 &+ a_{0,1}x_1 &+ \cdots &+ a_{0,n-1}x_{n-1} &= b_0,\\
a_{1,0}x_0 &+ a_{1,1}x_1 &+ \cdots &+ a_{1,n-1}x_{n-1} &= b_1,\\
&\vdots &&\vdots &\vdots\\
a_{n-1,0}x_0 &+ a_{n-1,1}x_1 &+ \cdots &+ a_{n-1,n-1}x_{n-1} &= b_{n-1}.
\end{aligned}
$$

- Thi

$a_{i,j}$, b is an n x 1 vector [b_0, b_1, ... , b_n]^T, and x is the solution.

Solving a System of Linear Equations

Two steps in solution are: reduction to triangular form, and back-substitution. The triangular form is as:

$$\begin{aligned}
x_0 + u_{0,1}x_1 + u_{0,2}x_2 + \cdots + u_{0,n-1}x_{n-1} &= y_0, \\
x_1 + u_{1,2}x_2 + \cdots + u_{1,n-1}x_{n-1} &= y_1, \\
&\vdots \qquad\qquad \vdots \\
x_{n-1} &= y_{n-1}.
\end{aligned}$$

We write this as: $Ux = y$.

A commonly used method for transforming a given matrix into an upper-triangular matrix is Gaussian Elimination.

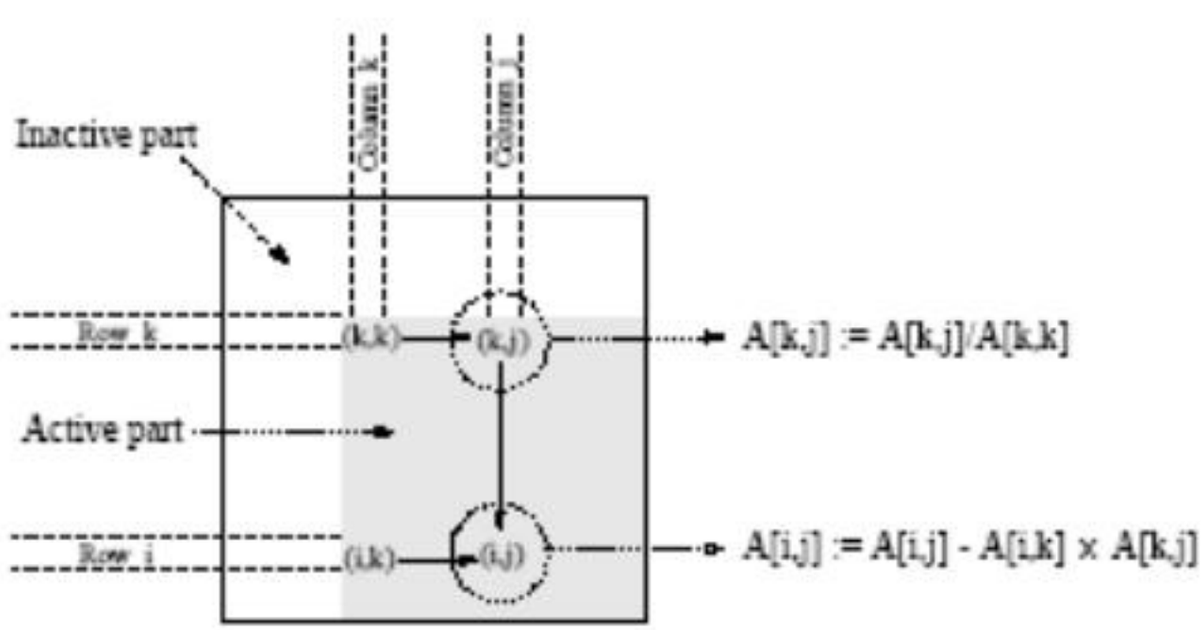
Gaussian Elimination


```
1.      procedure GAUSSIAN_ELIMINATION (A, b, y)
2.      begin
3.          for k := 0 to n − 1 do                    /* Outer loop */
4.          begin
5.              for j := k + 1 to n − 1 do
6.                  A[k, j] := A[k, j]/A[k, k];   /* Division step */
7.              y[k] := b[k]/A[k, k];
8.              A[k, k] := 1;
9.              for i := k + 1 to n − 1 do
10.             begin
11.                 for j := k + 1 to n − 1 do
12.                     A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.                 b[i] := b[i] − A[i, k] × y[k];
14.                 A[i, k] := 0;
15.             endfor;              /* Line 9 */
16.         endfor;                  /* Line 3 */
17.     end GAUSSIAN_ELIMINATION
```

Serial Gaussian Elimination

Gaussian Elimination

- The computation has three nested loops - in the kth iteration of the outer loop, the algorithm performs $(n-k)^2$ computations. Summing from $k = 1..n$, we have roughly $(n^3/3)$ multiplications-subtractions.



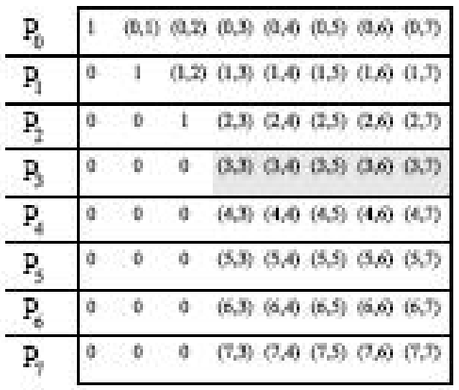
A typical computation in Gaussian elimination.

Parallel Gaussian Elimination

- Assume $p = n$ with each row assigned to a processor.
- The first step of the algorithm normalizes the row. This is a serial operation and takes time $(n\text{-}k)$ in the kth iteration.
- In the second step, the normalized row is broadcast to all the processors. This takes time $(t_s + t_w(n-k-1))\log n$.
- Each processor can independently eliminate this row from its own. This requires $(n\text{-}k\text{-}1)$ multiplications and subtractions.
- The total parallel time can be computed by summing from $k = 1$ ... $n\text{-}1$ as
- The formulation is not cost optimal because of the t_w term.

$$T_p = \frac{3}{2}n(n-1) + t_s n \log n + \frac{1}{2}t_w n(n-1)\log n.$$

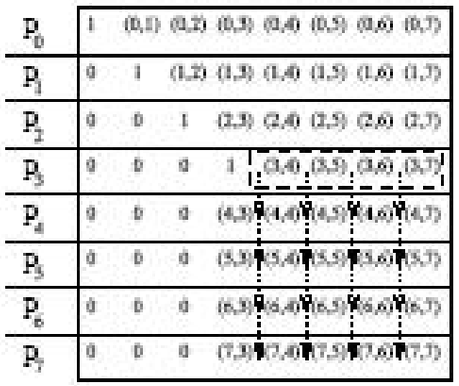
Parallel Gaussian Elimination



(a) Computation:

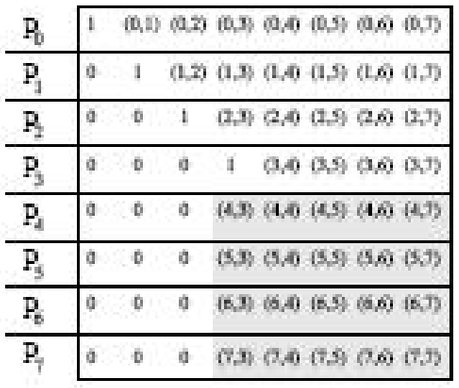
(i) A[k,j] := A[k,j]/A[k,k] for $k < j < n$

(ii) A[k,k] := 1



(b) Communication:

One-to-all brodcast of row A[k,*]



(c) Computation:

(i) A[i,j] = A[i,j] - A[i,k] × A[k,j]
for $k < i < n$ and $k < j < n$

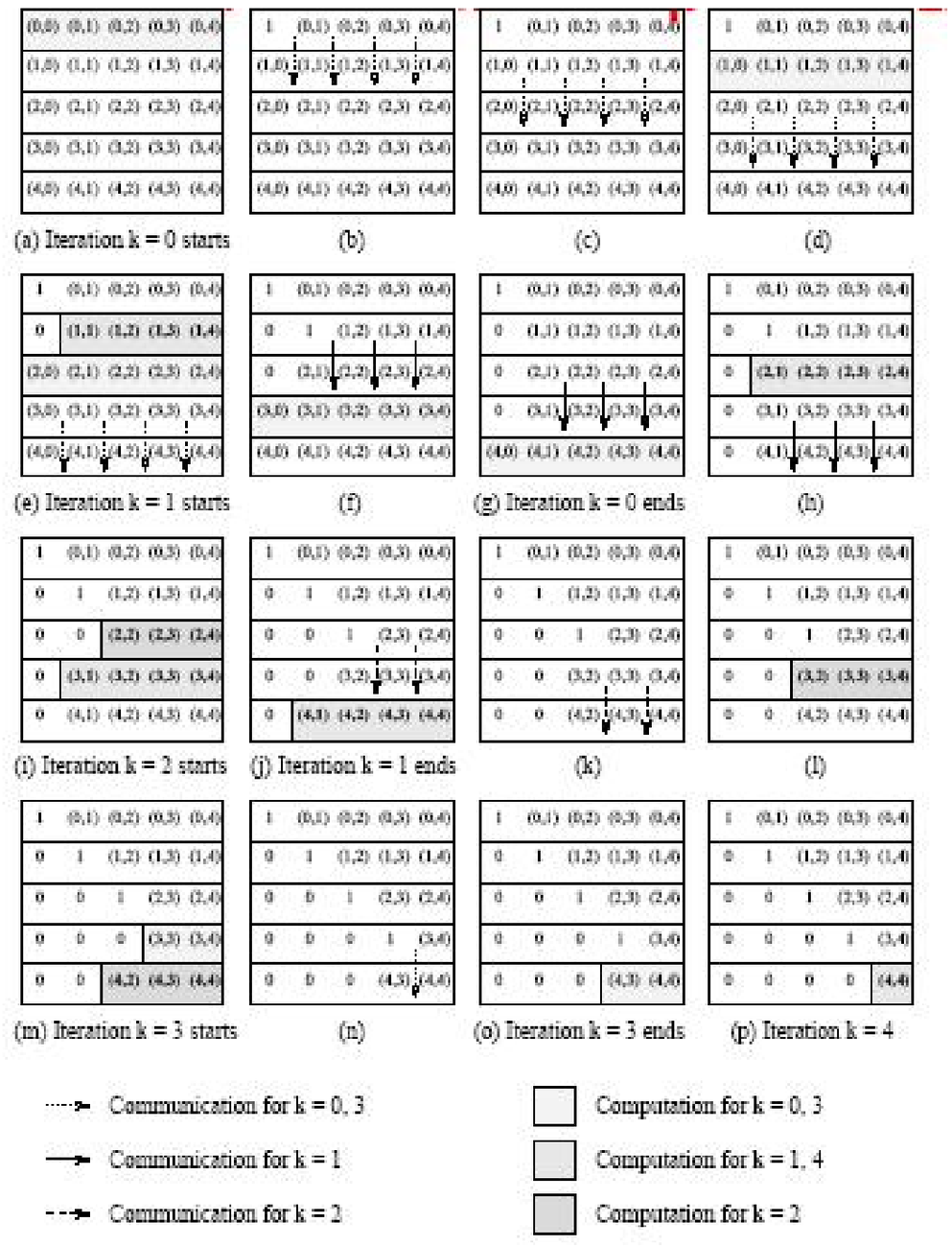
(ii) A[i,k] := 0 for $k < i < n$

Gaussian elimination steps during the iteration corresponding k = 3 for an 8 x 8 matrix partitioned rowwise among eight processes.

Parallel Gaussian Elimination: Pipelined Execution

- In the previous formulation, the (k+1)st iteration starts only after all the computation and communication for the kth iteration is complete.
- In the pipelined version, there are three steps - normalization of a row, communication, and elimination. These steps are performed in an asynchronous fashion.
- A processor P_k waits to receive and eliminate all rows prior to k.
- Once it has done this, it forwards its own row to processor Pk+1.

Parallel Gaussian Elimination: Pipelined Execution



Pipelined Gaussian elimination on a 5 x 5 matrix partitioned withone row per process.

Parallel Gaussian Elimination: Pipelined Execution

- The total number of steps in the entire pipelined procedure is $\Theta(n)$.
- In any step, either $O(n)$ elements are communicated between directly-connected processes, or a division step is performed on $O(n)$ elements of a row, or an elimination step is performed on $O(n)$ elements of a row.
- The parallel time is therefore $O(n^2)$.
- This is cost optimal.

Parallel Gaussian Elimination: Pipelined Execution

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_1	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_2	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_3	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

The communication in the Gaussian elimination iteration corresponding to k = 3 for an 8 x 8 matrix distributed among four processes using block 1-D partitioning.

Parallel Gaussian Elimination: Block 1D with $p < n$

- The above algorithm can be easily adapted to the case when $p < n$.
- In the kth iteration, a processor with all rows belonging to the active part of the matrix performs $(n - k$ -1) / np multiplications and subtractions.
- In the pipelined version, for $n > p$, computation dominates communication.
- The parallel time is given by:

  $$2(n/p)\Sigma_{k=0}^{n-1}(n-k-1)$$

  or approximately, n^3/p.
- While the algorithm is cost optimal, the cost of the parallel algorithm is higher than the sequential run time by a factor of 3/2.

Parallel Gaussian Elimination: Block 1D with $p < n$

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_1	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_2	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_3	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Block 1-D mapping

	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_0	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_1	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_2	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_3	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(b) Cyclic 1-D mapping

Computation load on different processes in block and cyclic
1-D partitioning of an 8 x 8 matrix on four processes during the
Gaussian elimination iteration corresponding to k = 3.

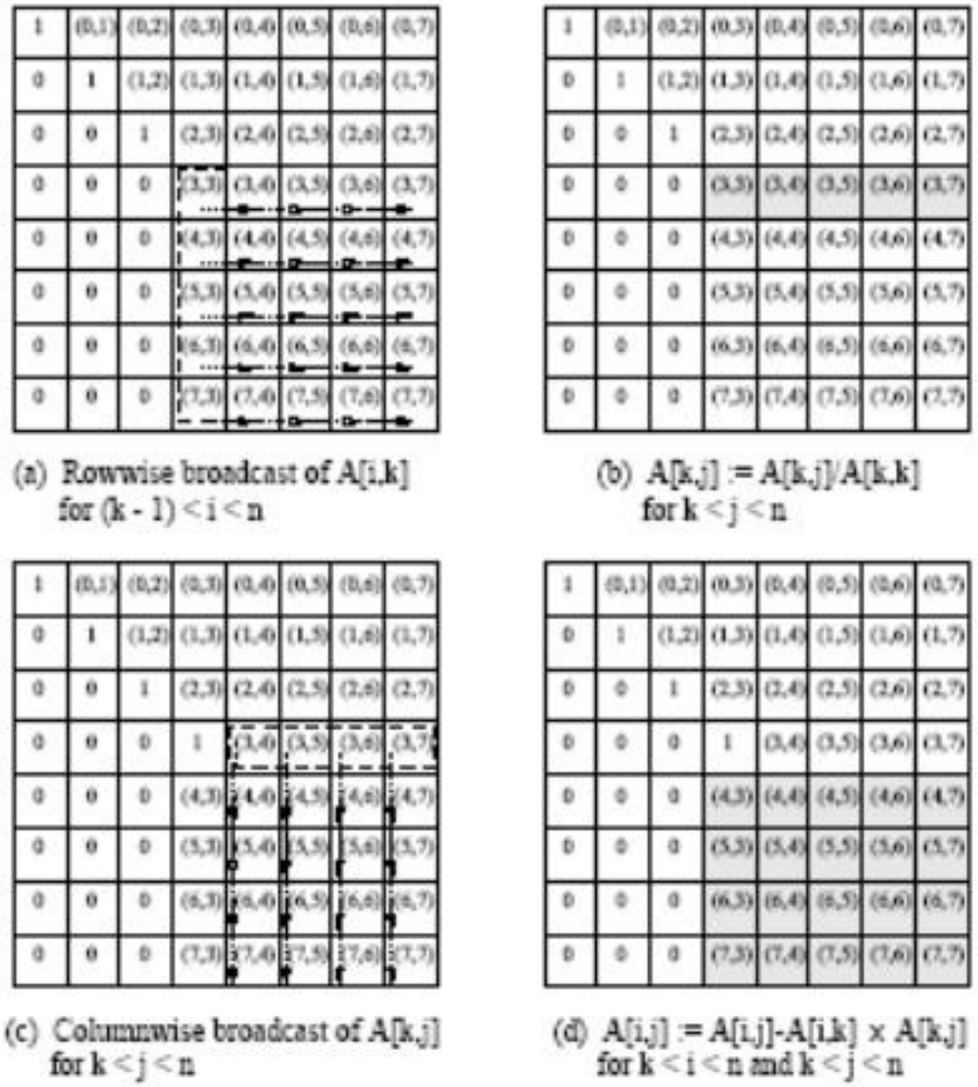
Parallel Gaussian Elimination: Block 1D with $p < n$

- The load imbalance problem can be alleviated by using a cyclic mapping.
- In this case, other than processing of the last p rows, there is no load imbalance.
- This corresponds to a cumulative load imbalance overhead of $O(n^2p)$ (instead of $O(n^3)$ in the previous case).

Parallel Gaussian Elimination: 2-D Mapping

- Assume an n X n matrix A mapped onto an n X n mesh of processors.
- Each update of the partial matrix can be thought of as a scaled rank-one update (scaling by the pivot element).
- In the first step, the pivot is broadcast to the row of processors.
- In the second step, each processor locally updates its value. For this it needs the corresponding value from the pivot row, and the scaling value from its own row.
- This requires two broadcasts, each of which takes $\log n$ time.
- This results in a non-cost-optimal algorithm.

Parallel Gaussian Elimination: 2-D Mapping



(a) Rowwise broadcast of A[i,k] for $(k - 1) < i < n$

(b) $A[k,j] := A[k,j]/A[k,k]$ for $k < j < n$

(c) Columnwise broadcast of A[k,j] for $k < j < n$

(d) $A[i,j] := A[i,j]-A[i,k] \times A[k,j]$ for $k < i < n$ and $k < j < n$

Various steps in the Gaussian elimination iteration corresponding to k = 3 for an 8 x 8 matrix on 64 processes arranged in a logical two-dimensional mesh.

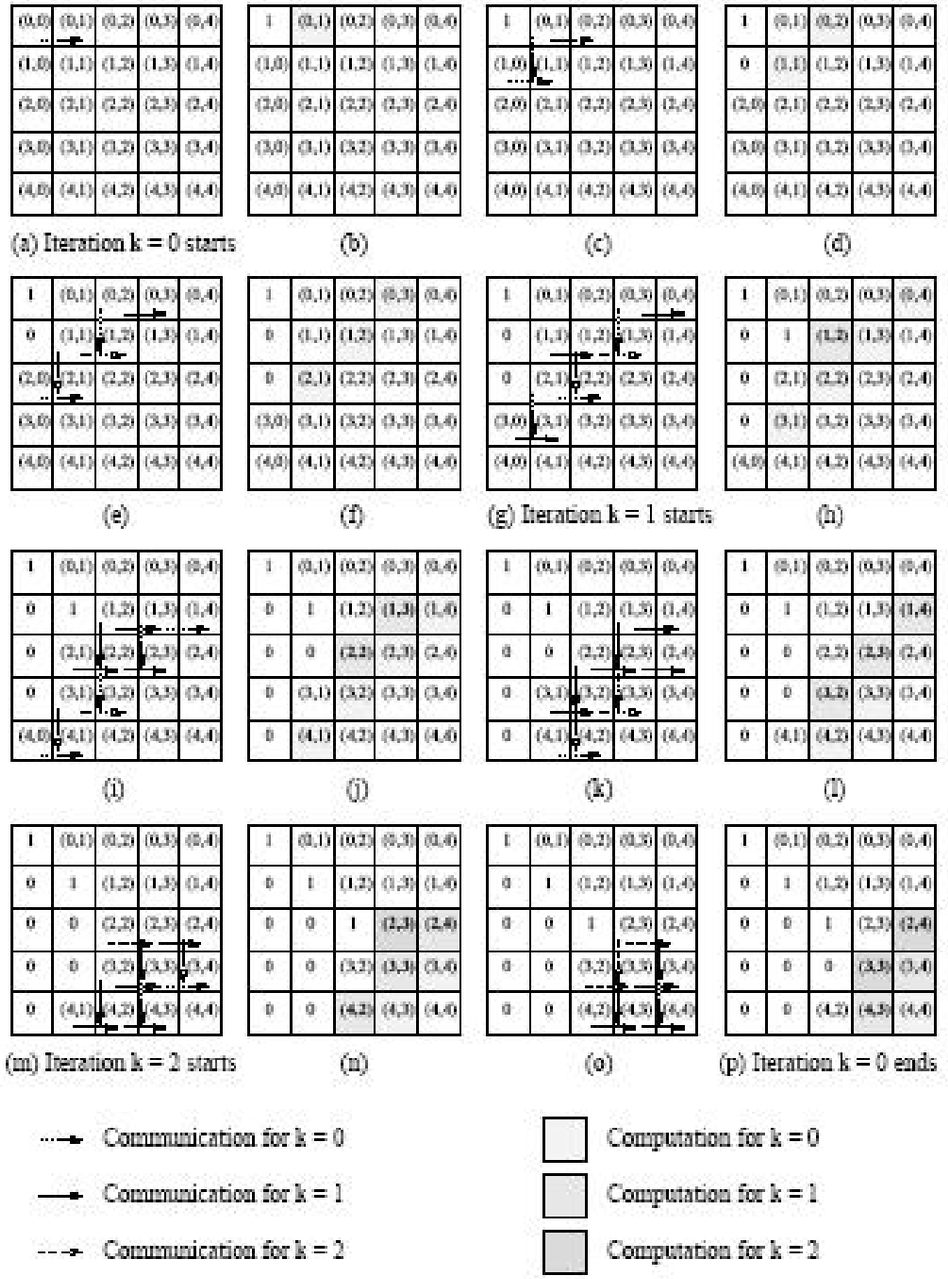
Parallel Gaussian Elimination: 2-D Mapping with Pipelining

- We pipeline along two dimensions. First, the pivot value is pipelined along the row. Then the scaled pivot row is pipelined down.
- Processor $P_{i,j}$ (not on the pivot row) performs the elimination step $A[i, j] := A[i, j]] - A[i, k]\ A[k, j]$ as soon as $A[i, k]$ and $A[k, j]$ are available.
- The computation and communication for each iteration moves through the mesh from top-left to bottom-right as a ``front."
- After the front corresponding to a certain iteration passes through a process, the process is free to perform subsequent iterations.
- Multiple fronts that correspond to different iterations are active simultaneously.

Parallel Gaussian Elimination: 2-D Mapping with Pipelining

- If each step (division, elimination, or communication) is assumed to take constant time, the front moves a single step in this time. The front takes $\Theta(n)$ time to reach $P_{n-1,n-1}$.
- Once the front has progressed past a diagonal processor, the next front can be initiated. In this way, the last front passes the bottom-right corner of the matrix $\Theta(n)$ steps after the first one.
- The parallel time is therefore $O(n)$, which is cost-optimal.

2-D Mapping with Pipelining

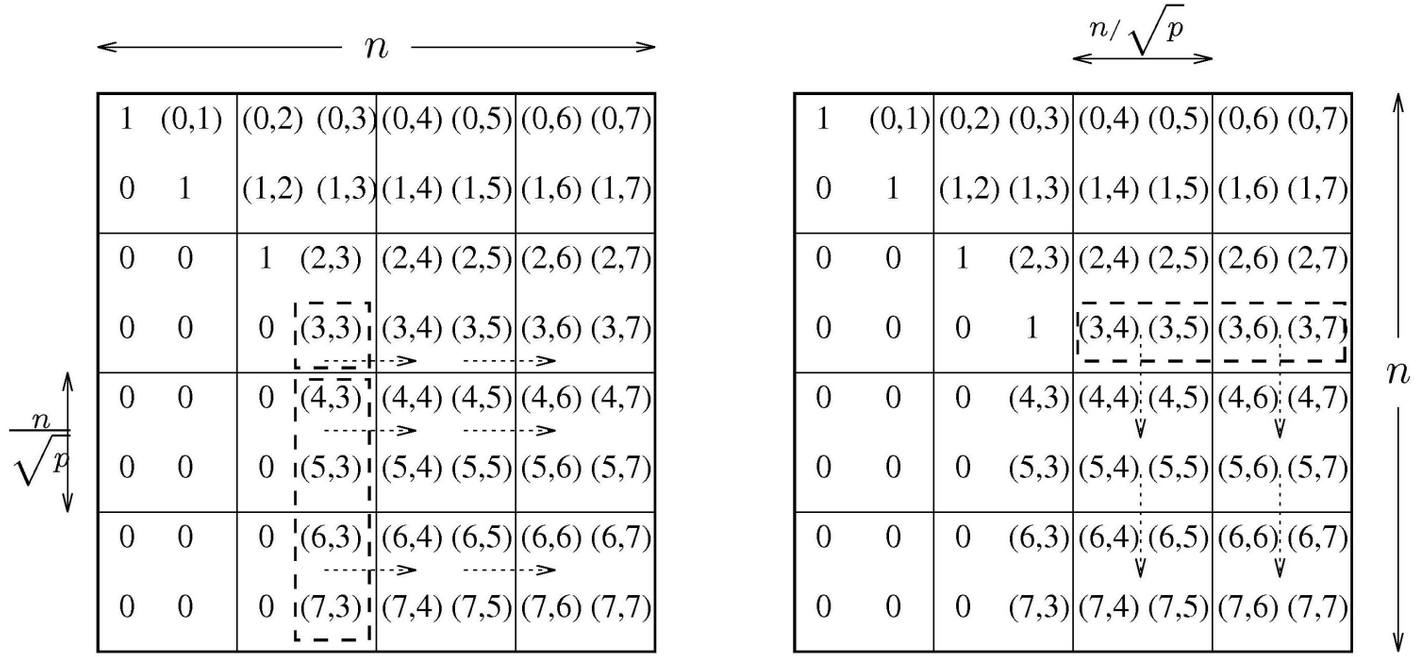


Pipelined Gaussian elimination for a 5 x 5 matrix with 25 processors

Parallel Gaussian Elimination: 2-D Mapping with Pipelining and $p < n$

- In this case, a processor containing a completely active part of the matrix performs n^2/p multiplications and subtractions, and communicates $n/\sqrt{p}$ words along its row and its column.
- The computation dominates communication for $n >> p$.
- The total parallel run time of this algorithm is $(2n^2/p) \times n$, since there are n iterations. This is equal to $2n^3/p$.
- This is three times the serial operation count!

2-D Mapping with Pipelining and $p < n$



(a) Rowwise broadcast of A[i,k]
for i = k to (n - 1)

(b) Columnwise broadcast of A[k,j]
for j = (k + 1) to (n - 1)

The communication steps in the Gaussian elimination iteration corresponding to k = 3 for an 8 x 8 matrix on 16 processes of a two-dimensional mesh.

2-D Mapping with Pipelining and $p < n$

1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Block-checkerboard mapping

1	(0,4)	(0,1)	(0,5)	(0,2)	(0,6)	(0,3)	(0,7)
0	(4,4)	0	(4,5)	0	(4,6)	(4,3)	(4,7)
0	(1,4)	1	(1,5)	(1,2)	(1,6)	(1,3)	(1,7)
0	(5,4)	0	(5,5)	0	(5,6)	(5,3)	(5,7)
0	(2,4)	0	(2,5)	1	(2,6)	(2,3)	(2,7)
0	(6,4)	0	(6,5)	0	(6,6)	(6,3)	(6,7)
0	(3,4)	0	(3,5)	0	(3,6)	(3,3)	(3,7)
0	(7,4)	0	(7,5)	0	(7,6)	(7,3)	(7,7)

(b) Cyclic-checkerboard mapping

Computational load on different processes in block and cyclic 2-D mappings of an 8 x 8 matrix onto 16 processes during the Gaussian elimination iteration corresponding to k = 3.

Parallel Gaussian Elimination: 2-D Cyclic Mapping

- The idling in the block mapping can be alleviated using a cyclic mapping.
- The maximum difference in computational load between any two processes in any iteration is that of one row and one column update.
- This contributes $\Theta(n\sqrt{p})$ to the overhead function. Since there are n iterations, the total overhead is $\Theta(n^2\sqrt{p})$.

Gaussian Elimination with Partial Pivoting

- For numerical stability, one generally uses partial pivoting.
- In the k th iteration, we select a column i (called the *pivot* column) such that $A[k, i]$ is the largest in magnitude among all $A[k, i]$ such that $k \leq j < n$.
- The k th and the i th columns are interchanged.
- Simple to implement with row-partitioning and does not add overhead since the division step takes the same time as computing the max.
- Column-partitioning, however, requires a global reduction, adding a $\log p$ term to the overhead.
- Pivoting precludes the use of pipelining.

Gaussian Elimination with Partial Pivoting: 2-D Partitioning

- Partial pivoting restricts use of pipelining, resulting in performance loss.
- This loss can be alleviated by restricting pivoting to specific columns.
- Alternately, we can use faster algorithms for broadcast.

Solving a Triangular System: Back-Substitution

- The upper triangular matrix U undergoes back-substitution to determine the vector x.

```
1.    procedure BACK_SUBSTITUTION (U, x, y)
2.    begin
3.        for k := n − 1 downto 0 do   /* Main loop */
4.        begin
5.            x[k] := y[k];
6.            for i := k − 1 downto 0 do
7.                y[i] := y[i] − x[k] × U[i, k];
8.        endfor;
9.    end BACK_SUBSTITUTION
```

A serial algorithm for back-substitution.

Solving a Triangular System: Back-Substitution

- The algorithm performs approximately $n^2/2$ multiplications and subtractions.
- Since complexity of this part is asymptotically lower, we should optimize the data distribution for the factorization part.
- Consider a rowwise block 1-D mapping of the n x n matrix U with vector y distributed uniformly.
- The value of the variable solved at a step can be pipelined back.
- Each step of a pipelined implementation requires a constant amount of time for communication and $\Theta(n/p)$ time for computation.
- The parallel run time of the entire algorithm is $\Theta(n^2/p)$.

Solving a Triangular System: Back-Substitution

- If the matrix is partitioned by using 2-D partitioning on a $\sqrt{p} \times \sqrt{p}$ logical mesh of processes, and the elements of the vector are distributed along one of the columns of the process mesh, then only the $\sqrt{p}$ processes containing the vector perform any computation.
- Using pipelining to communicate the appropriate elements of U to the process containing the corresponding elements of y for the substitution step (line 7), the algorithm can be executed in time.
- While this is not cost optimal, since this does not dominate the overall computation, the cost optimality is determined by the factorization.

$$\Theta(n^2/\sqrt{p})$$