ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY

SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Parallel Computation Problems

References

- Michael J. Quinn. **Parallel Computing. Theory and Practice**. McGraw-Hill
- Albert Y. Zomaya. **Parallel and Distributed Computing Handbook**. McGraw-Hill
- Ian Foster. **Designing and Building Parallel Programs**. Addison-Wesley.
- Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar **. Introduction to Parallel Computing, Second Edition.** Addison Wesley.
- Joseph Jaja**. An Introduction to Parallel Algorithm.** Addison Wesley.
- Nguyễn Đức Nghĩa**. Tính toán song song.** Hà Nội 2003.

9.1 Numerical approach for dense matrix

Review

Summary of communication times of various operations discussed in Sections 4.1–4.7 on a hypercube interconnection network. The message size for each operation is m and the number of nodes is p.

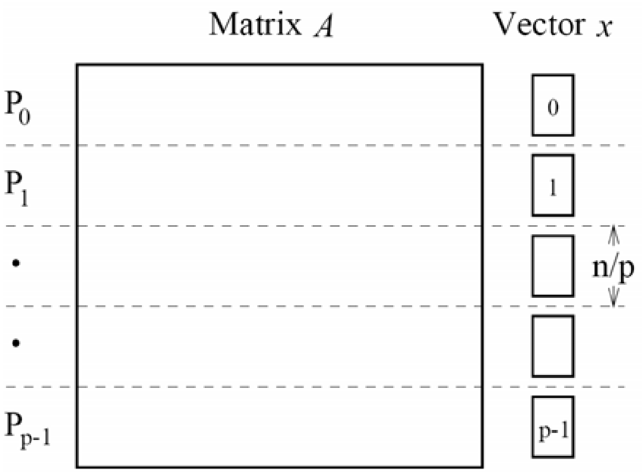
Operation	Hypercube Time	B/W Requirement
One-to-all broadcast, All-to-one reduction	$\min((t_s + t_w m)\log p, 2(t_s \log p + t_w m))$	$\Theta(1)$
All-to-all broadcast, All-to-all reduction	$t_s \log p + t_w m(p-1)$	$\Theta(1)$
All-reduce	$\min((t_s + t_w m)\log p, 2(t_s \log p + t_w m))$	$\Theta(1)$
Scatter, Gather	$t_s \log p + t_w m(p-1)$	$\Theta(1)$
All-to-all personalized	$(t_s + t_w m)(p-1)$	$\Theta(p)$
Circular shift	$t_s + t_w m$	$\Theta(p)$

Matrix-Vector Multiplication

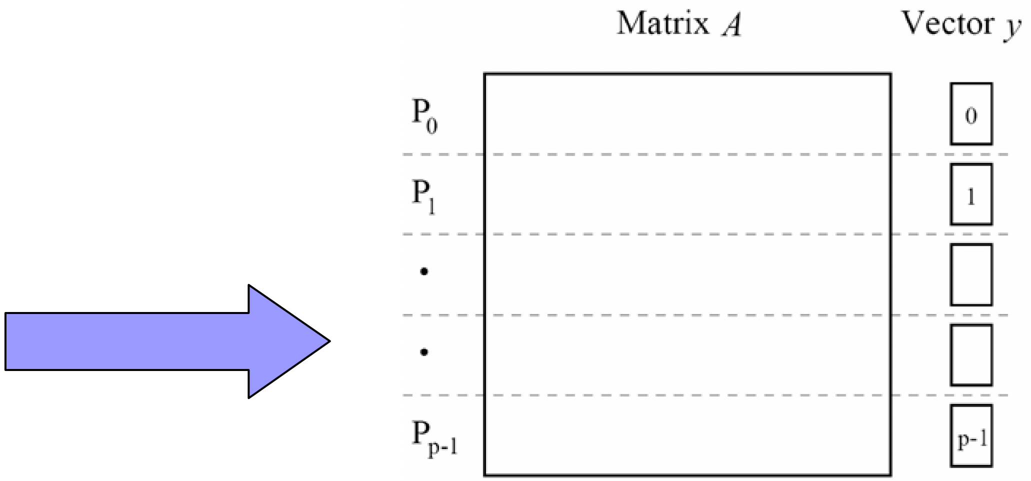
- Compute: $y = Ax$
  - y, x are nx1 vectors
  - A is an *nxn dense matrix*
- Serial complexity: $W = O(n^2)$.
- We will consider:
  - 1D & 2D partitioning.

```
procedure MAT_VECT (A, x, y)
begin
    for i := 0 to n − 1 do
    begin
        y[i] := 0;
        for j := 0 to n − 1 do
            y[i] := y[i] + A[i, j] × x[j];
    endfor;
end MAT_VECT
```

Row-wise 1D Partitioning



(a) Initial partitioning of the matrix and the starting vector x



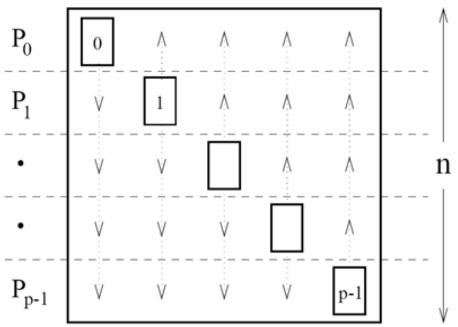
(d) Final distribution of the matrix and the result vector y

How do we perform the operation?

Row-wise 1D Partitioning

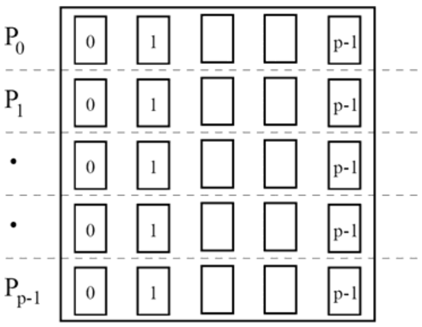
Each processor needs to have the entire *x* vector.

All-to-all broadcast



(b) Distribution of the full vector among all the processes by all-to-all broadcast

Local computations



(c) Entire vector distributed to each process after the broadcast

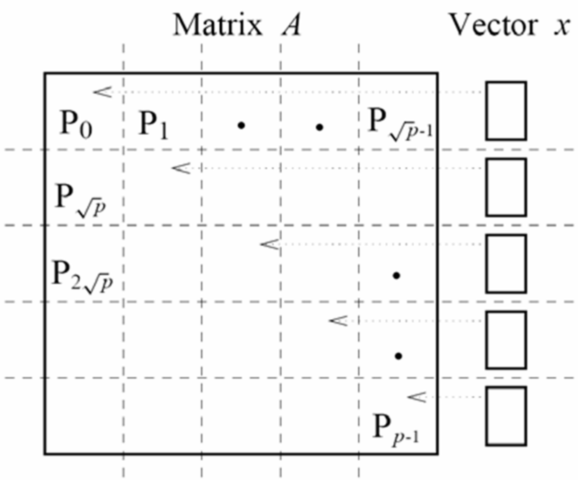
Analysis?

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$$

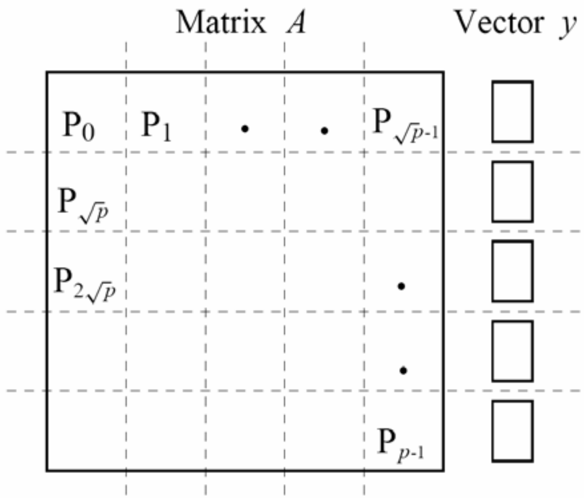
$$T_o = t_s p \log p + t_w n p.$$

$$W = \Theta(p^2)$$

Block 2D Partitioning



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

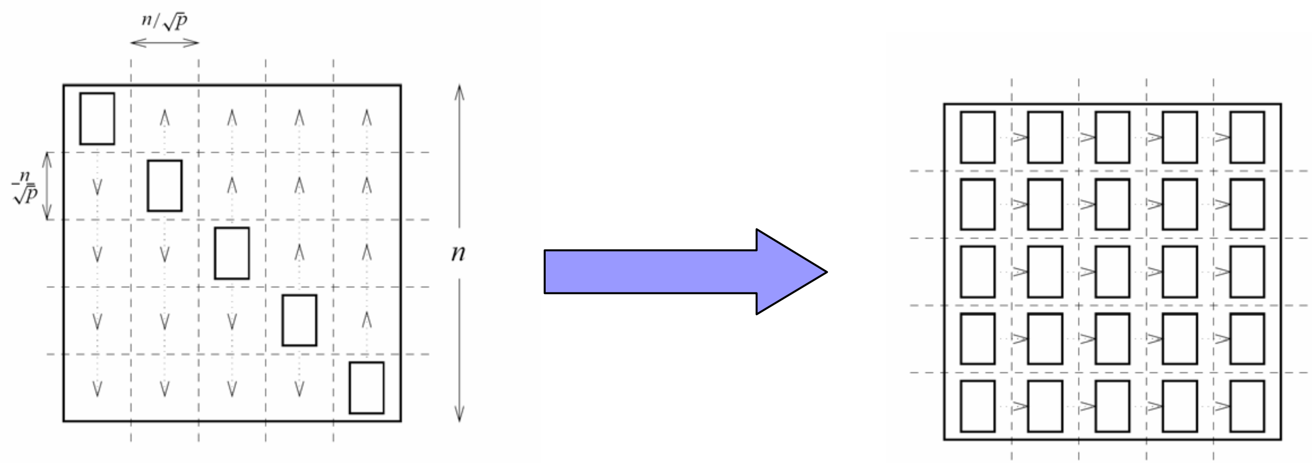


(d) Final distribution of the result vector

How do we perform the operation?

Block 2D Partitioning

Each processor needs to have the portion of the *x* vector that corresponds to the set of columns that it stores.



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

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

Analysis?

$$
\begin{aligned}
T_P &= \overbrace{n^2/p}^{\text{computation}} + \overbrace{t_s + t_w n/\sqrt{p}}^{\text{aligning the vector}} + \\
&\quad \overbrace{(t_s + t_w n/\sqrt{p})\log(\sqrt{p})}^{\text{columnwise one-to-all broadcast}} + \overbrace{(t_s + t_w n/\sqrt{p})\log(\sqrt{p})}^{\text{all-to-one reduction}} \\
&\approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p
\end{aligned}
$$

$$T_o = t_s p \log p + t_w n \sqrt{p} \log p.$$

$$W = \Theta(p \log^2 p).$$

1D vs 2D Formulation

- Which one is better?

Matrix-Matrix Multiplication

- Compute: $C = AB$
  - A, B, & C are nxn dense matrices.
- Serial complexity: $W = O(n^3)$.
- We will consider:
  - 2D & 3D partitioning.

```
procedure MAT_MULT (A, B, C)
begin
   for i := 0 to n − 1 do
      for j := 0 to n − 1 do
         begin
            C[i, j] := 0;
            for k := 0 to n − 1 do
               C[i, j] := C[i, j] + A[i, k] × B[k, j];
         endfor;
end MAT_MULT
```

Simple 2D Algorithm

- Processors are arranged in a logical sqrt(p)*sqrt(p) 2D topology.
- Each processor gets a block of (n/sqrt(p))*(n/sqrt(p)) block of A, B, & C.
- It is responsible for computing the entries of C that it has been assigned to.
- Analysis?

$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}. \qquad W = \Theta(p^{3/2})$$

How about the memory complexity?

Cannon's Algorithm

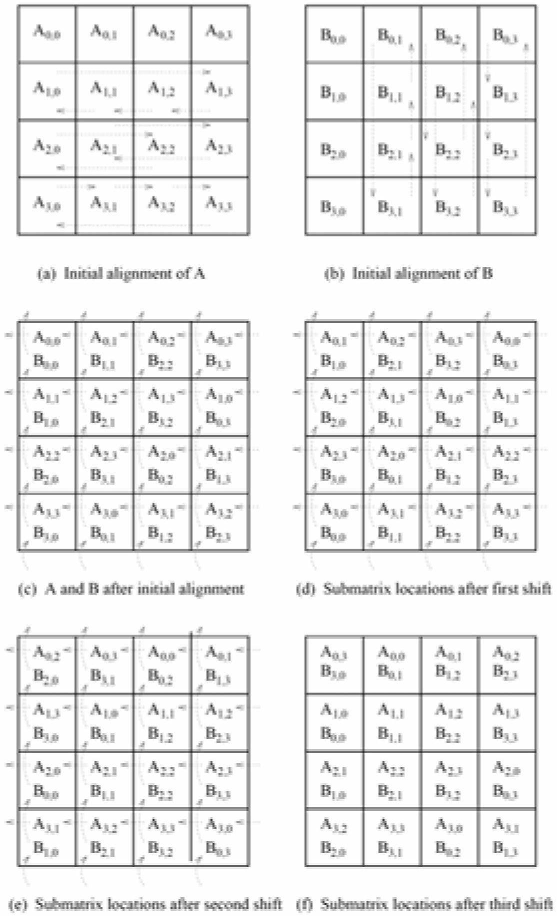
- Memory efficient variant of the simple algorithm.
- Key idea:
  - Replace traditional loop:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j}$$

  - With the following loop:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j}$$

- During each step, processors operate on different blocks of *A* and *B*.



The communication steps in Cannon's algorithm on 16 processes.

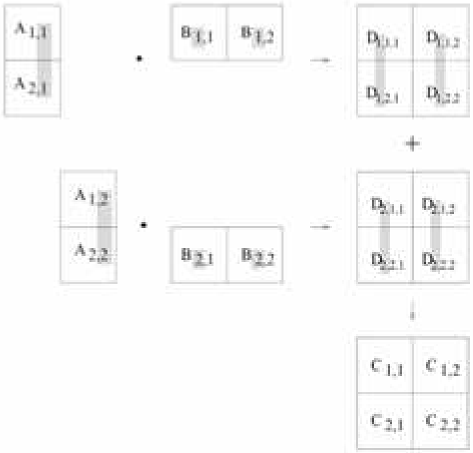
$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}.$$

Can we do better?

- Can we use more than $O(n^2)$ processors?
- So far the task corresponded to the dot-product of two vectors
  - i.e., $C_{i,j} = A_{i,*} . B_{*,j}$
- How about performing this dot-product in parallel?
- What is the maximum concurrency that we can extract?

3D Algorithm—DNS Algorithm

- Partitioning the intermediate data



Multiplication of matrices A and B with partitioning of the three-dimensional intermediate matrix D.

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \left(\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \atop \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \right)$$

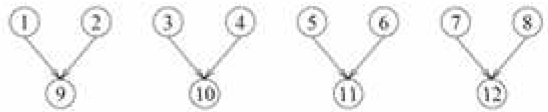
Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

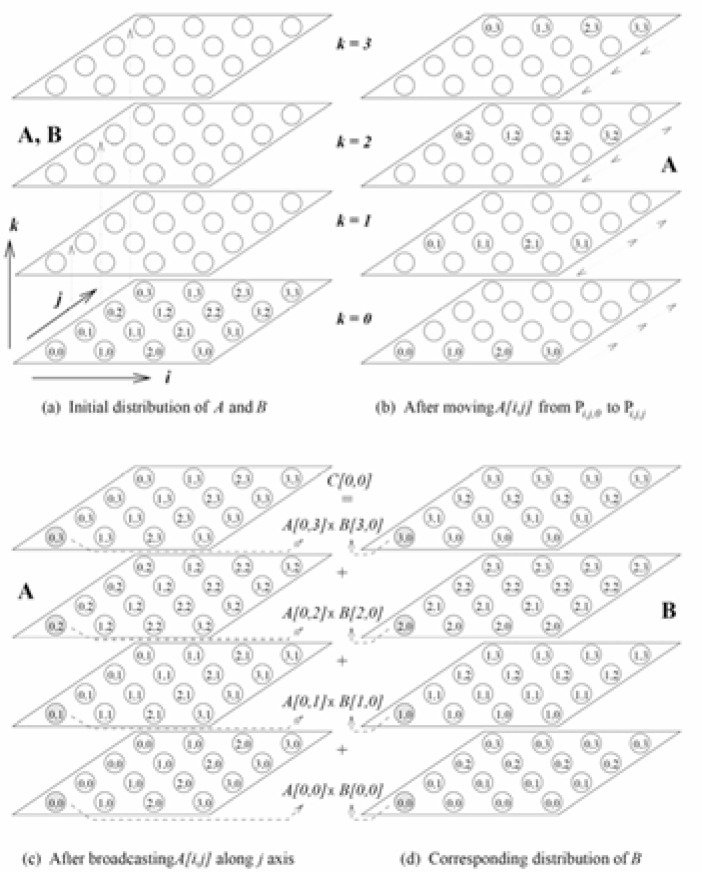
Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$
Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$
Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$
Task 04: $D_{2,1,2} = A_{1,2}B_{2,2}$
Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$
Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$
Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$
Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$
Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$
Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.



The task-dependency graph of the decomposition shown in Figure 3.15.

3D Algorithm—DNS Algorithm



The communication steps in the DNS algorithm while multiplying 4 × 4 matrices A and B on 64 processes. The shaded processes in part (c) store elements of the first row of A and the shaded processes in part (d) store elements of the first column of B.

$$q = p^{1/3}$$

$$T_P \approx \left(\frac{n}{q}\right)^3 + 3t_s \log q + 3t_w \left(\frac{n}{q}\right)^2 \log q$$

$$T_P = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p.$$

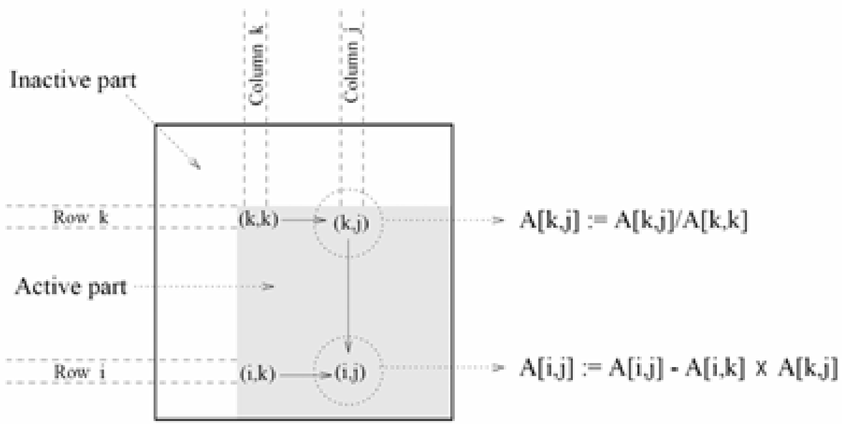
$$W = \Theta(p(\log p)^3)$$

Gaussian Elimination

- Solve $Ax=b$
  - A is an nxn dense matrix.
  - x and b are dense vectors
- Serial complexity: $W = O(n^3)$.
- There are two key steps in each iteration:
  - Division step
  - Rank-1 update
- We will consider:
  - 1D & 2D partitioning, and introduce the notion of pipelining.

```
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
```

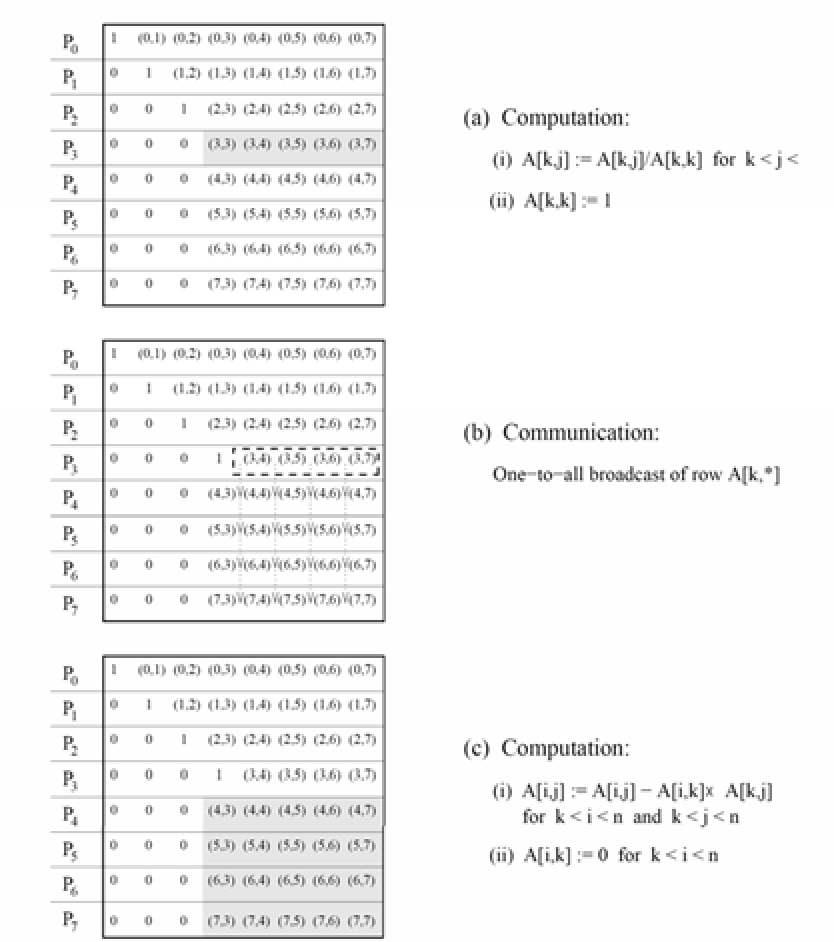
A serial Gaussian elimination algorithm that converts the system of linear equations $Ax = b$ to a unit upper-triangular system $Ux = y$. The matrix U occupies the upper-triangular locations of A. This algorithm assumes that $A[k, k] \neq 0$ when it is used as a divisor on lines 6 and



A typical computation in Gaussian elimination.

1D Partitioning

- Assign *n/p* rows of *A* to each processor.
- During the *i*th iteration:
  - Divide operation is performed by the processor who stores row *i.*
  - Result is broadcasted to the rest of the processors.
  - Each processor performs the rank-1 update for its local rows.
- Analysis?



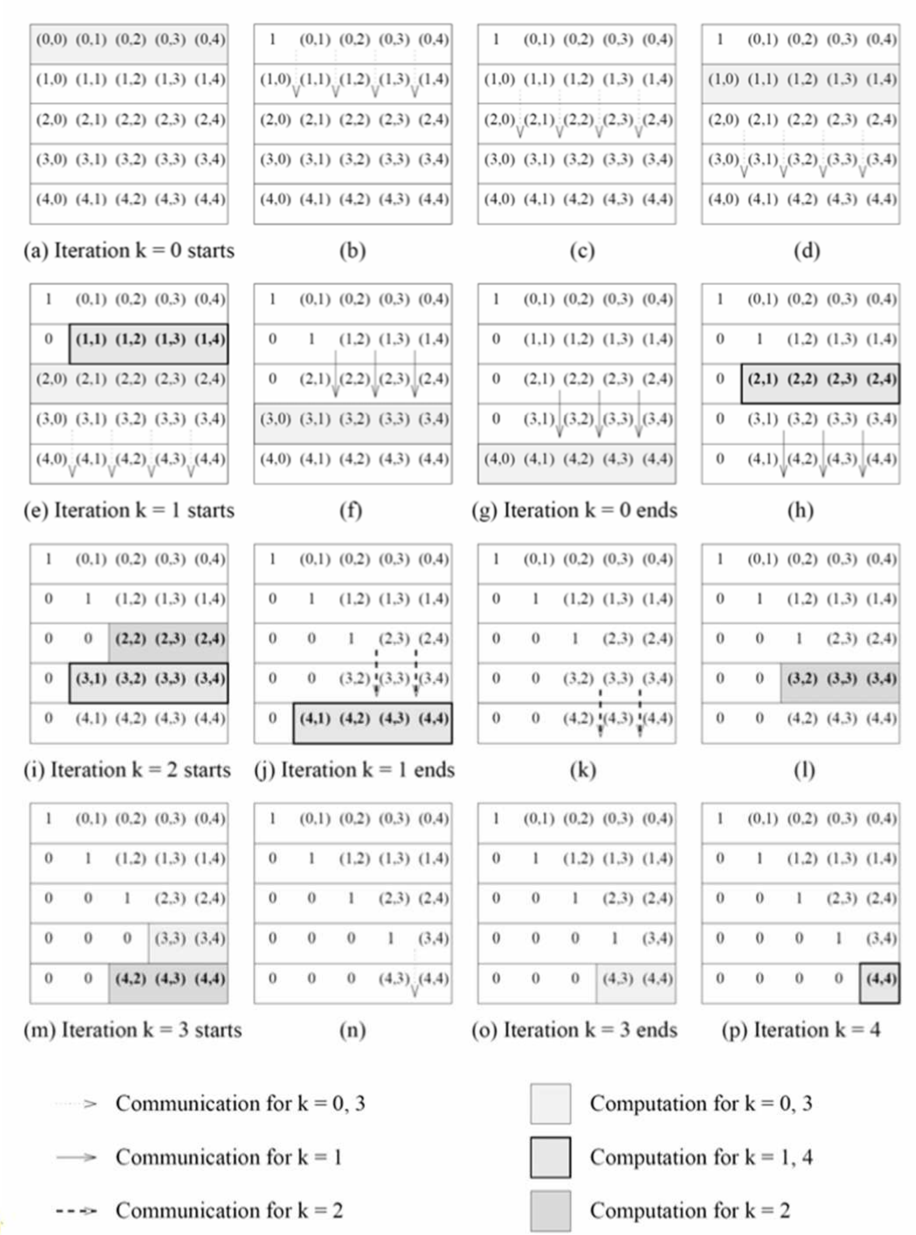
Gaussian elimination steps during the iteration corresponding to $k = 3$ for an 8×8 matrix partitioned rowwise among eight processes.

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

(one element per processor)

1D Pipelined Formulation

- Existing Algorithm:
  Next iteration starts only when the previous iteration has finished.
- Key Idea:
  The next iteration can start as soon as the rank-1 update involving the next row has finished.
  - Essentially multiple iterations are perform simultaneously!



Cost-optimal with
n processors

1D Partitioning

- Is the block mapping a good idea?

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

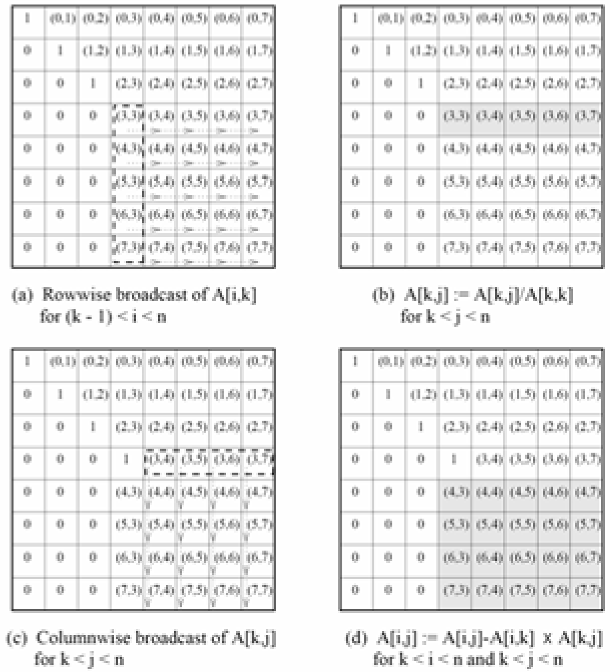
1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	P_0
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	P_1
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	P_2
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	
0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	P_3
0	0	0	(7,3)	(7,4)	(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 × 8 matrix on four processes during the Gaussian elimination iteration corresponding to $k = 3$.

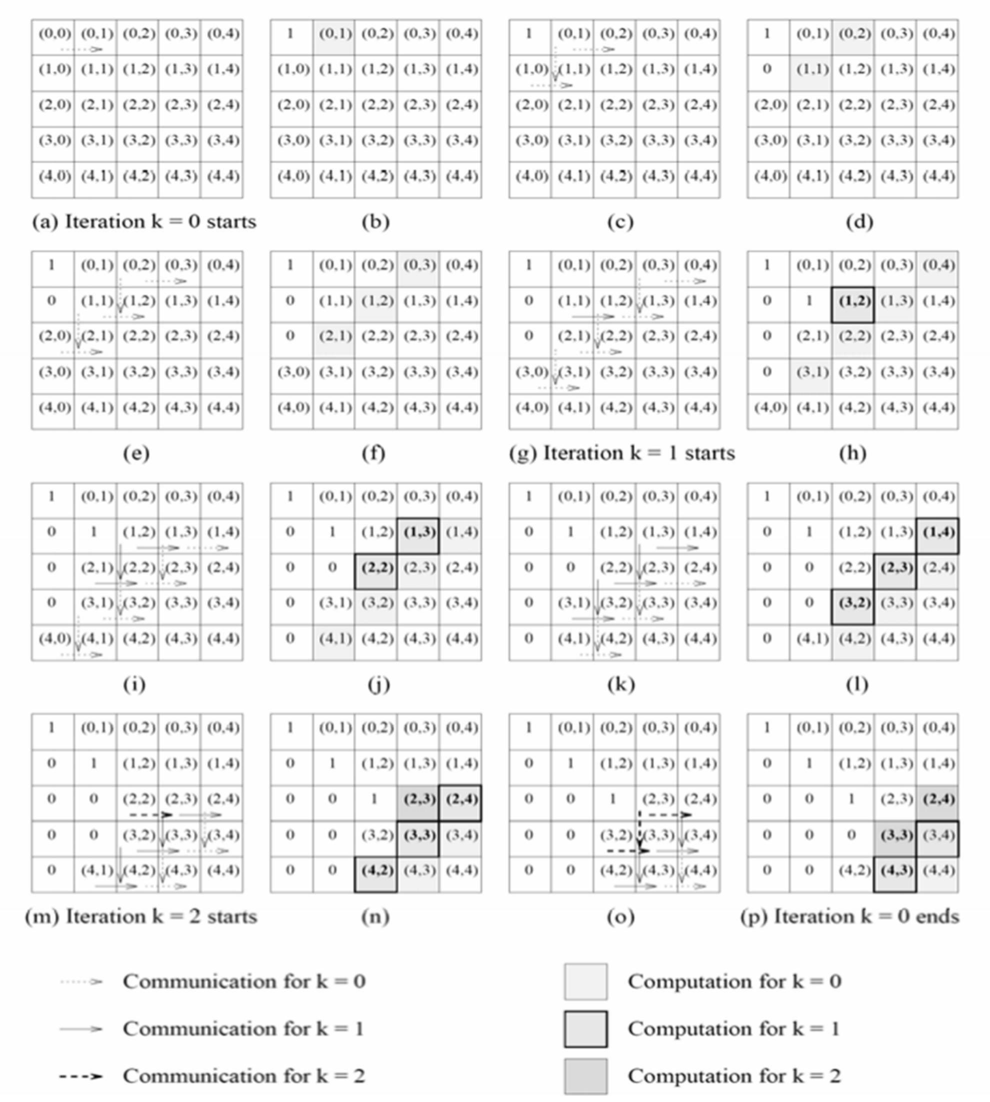
2D Mapping

- Each processor gets a 2D block of the matrix.
- Steps:
  - Broadcast of the “active” column along the rows.
  - Divide step in parallel by the processors who own portions of the row.
  - Broadcast along the columns.
  - Rank-1 update.
- Analysis?



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

2D Pipelined



Cost-optimal with n^2 processors

9.2 Numerical approach for PDE problem

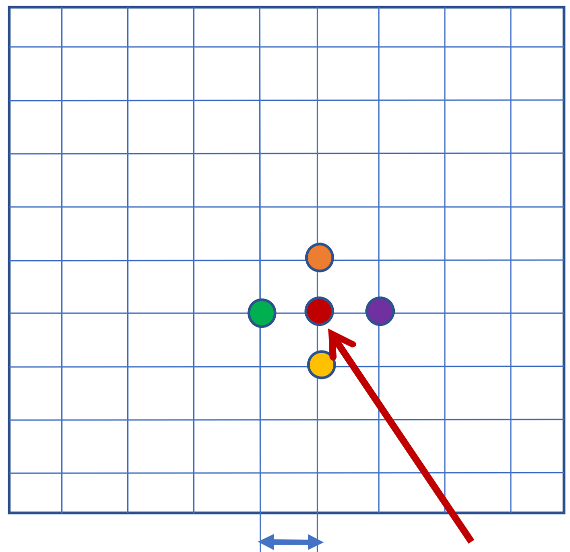
PARALLEL SOLUTION TO PDEs
CASE STUDY: HEAT EQUATIONS

Mathematic model and algorithm

- Heat equations (PDE):

$$\frac{\partial C}{\partial t} = D\nabla^2 C$$

$$\nabla^2 C = \frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2}$$



- Algorithm
  - Initial input value: $C^0_{i,j}$
  - At step n+1:

$$\nabla^2 C^{tn}_{\boxed{i,j}} = FD^{tn}_{\boxed{i,j}} = \frac{\left(C^{tn}_{\boxed{i+1,j}} + C^{tn}_{\boxed{i-1,j}} + C^{tn}_{\boxed{i,j+1}} + C^{tn}_{\boxed{i,j-1}} - 4C^{tn}_{\boxed{i,j}}\right)}{dx^2}$$

$$C^{tn+1}_{\boxed{i,j}} = C^{tn}_{\boxed{i,j}} + dt * D * FD^{tn}_{\boxed{i,j}}$$

- Data dependdncy?

Data dependency

- Calculation at point (i,j) needs data from neighboring points: (i-1,j), (i+1,j) , (i,j-1) , (i,j+1)
- This is data dependency
- Solution
  - Shared memory system: Synchronization
  - Distributed memory system: Communication and Synchronization (Difficult, Optimization)
- Exercise:
  - Write a OpenMP program to implement Heat Equations problem
  - Write a MPI program to implement Heat Equations problem

Mathematic model and algorithm

- Notation:

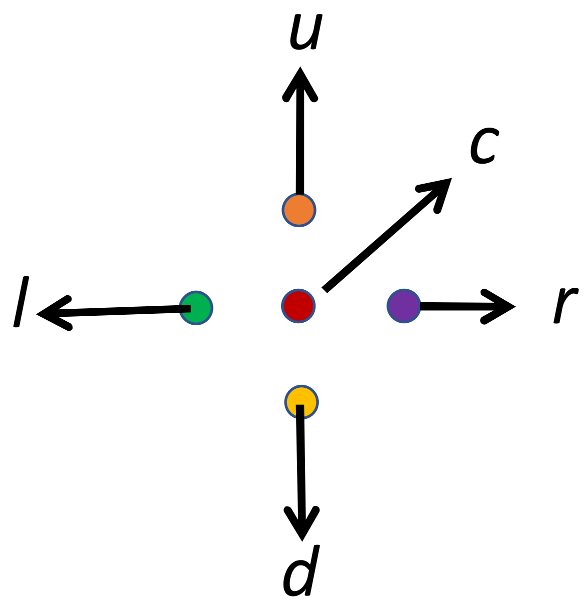
$c: \quad C_{i,j},$

$u: \quad C_{i-1,j},$

$d: \quad C_{i+1,j},$

$l: \quad C_{i,j-1},$

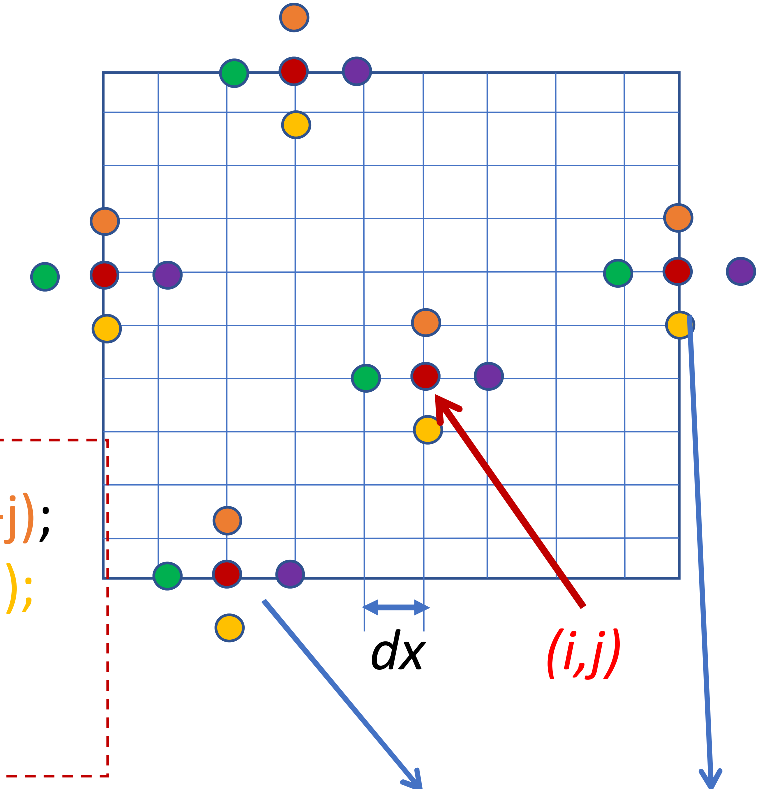
$r: \quad C_{i,j+1}.$



Implementation: Spatial Discretization (FD)

$$\nabla^2 C_{\boxed{i,j}}^{tn} = FD_{\boxed{i,j}}^{tn} = \frac{\left(C_{\boxed{i+1,j}}^{tn} + C_{\boxed{i-1,j}}^{tn} + C_{\boxed{i,j+1}}^{tn} + C_{\boxed{i,j-1}}^{tn} - 4C_{\boxed{i,j}}^{tn}\right)}{dx^2}$$

```
void FD(float *C, float *dC) {
int i, j;
float c,u,d,l,r;
for (  i = 0 ; i < m ; i++ )
  for ( j = 0 ; j < n ; j++ )
   {
     c = *(C+i*n+j);
     u = (i==0)        ? *(C+i*n+j) : *(C+(i-1)*n+j);
     d = (i==m-1) ? *(C+i*n+j) : *(C+(i+1)*n+j);
     l = (j==0)       ? *(C+i*n+j) : *(C+i*n+j-1);
     r = (j==n-1)  ? *(C+i*n+j) : *(C+i*n+j+1);
     *(dC+i*n+j) = (1/(dx*dx))*(u+d+l+r-4*c);
   }
}
```



Boundary condition

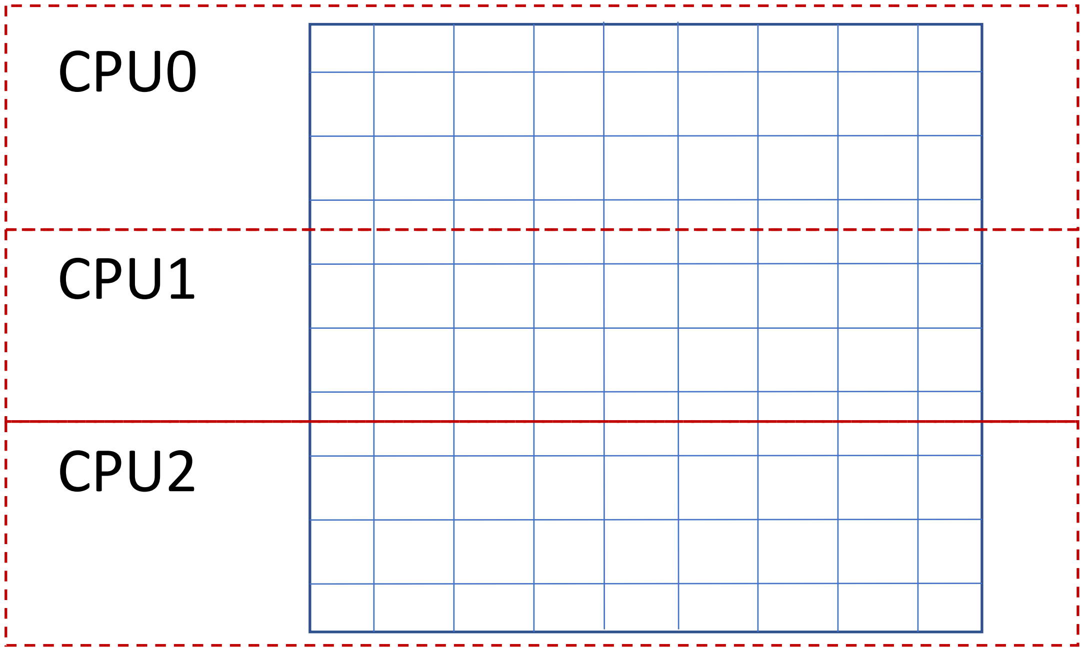
Implementation: Time Integration

$$C_{\boxed{i,j}}^{tn+1} = C_{\boxed{i,j}}^{tn} + dt * D * FD_{\boxed{i,j}}^{tn}$$

```
while (t<=T)
{
    FD(C, dC);
    for (  i = 0 ; i < m ; i++ )
       for ( j = 0 ; j < n ; j++ )
          *(C+i*n+j) = *(C+i*n+j) + dt*(*(dC+i*n+j));
     t=t+dt;
}
```

SPMD Parallel Algorithm (1)

- SPMD: Single Program Multiple Data



Domain Decomposition

SPMD Parallel Algorithm SPMD (2)

- B1: Input data
  - Usually, initial input data at CPU 0 (Root)
- B2: Domain decomposition
- B3: Distribute Input data from Root to all other CPUs
- B4: Computation (Each CPU calculate on its subdomain)
- B5: Gather Output from all other CPUs to Root

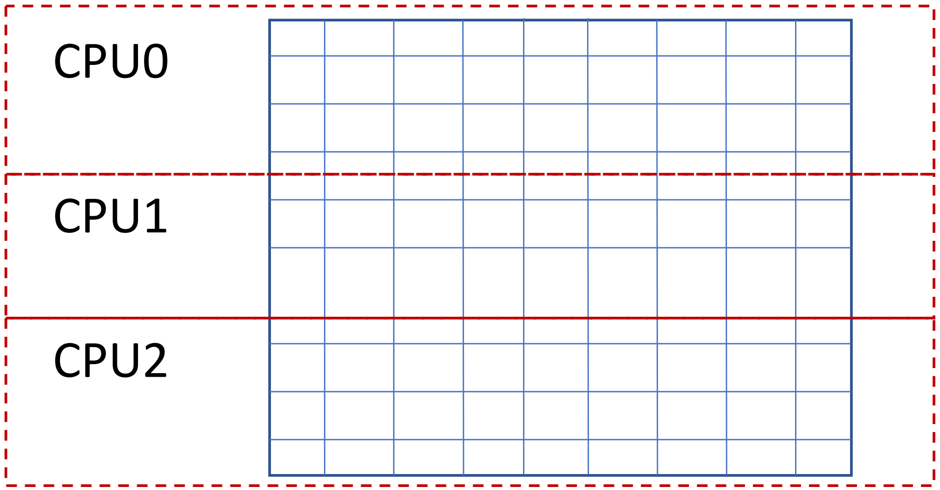
B3 and B5: Communication (Input và Output)

SPMD Parallel Algorithm SPMD (3)

- B1: Input data
  - Depending on requirement of each problem
  - Different input results in different output

SPMD Parallel Algorithm SPMD (4)

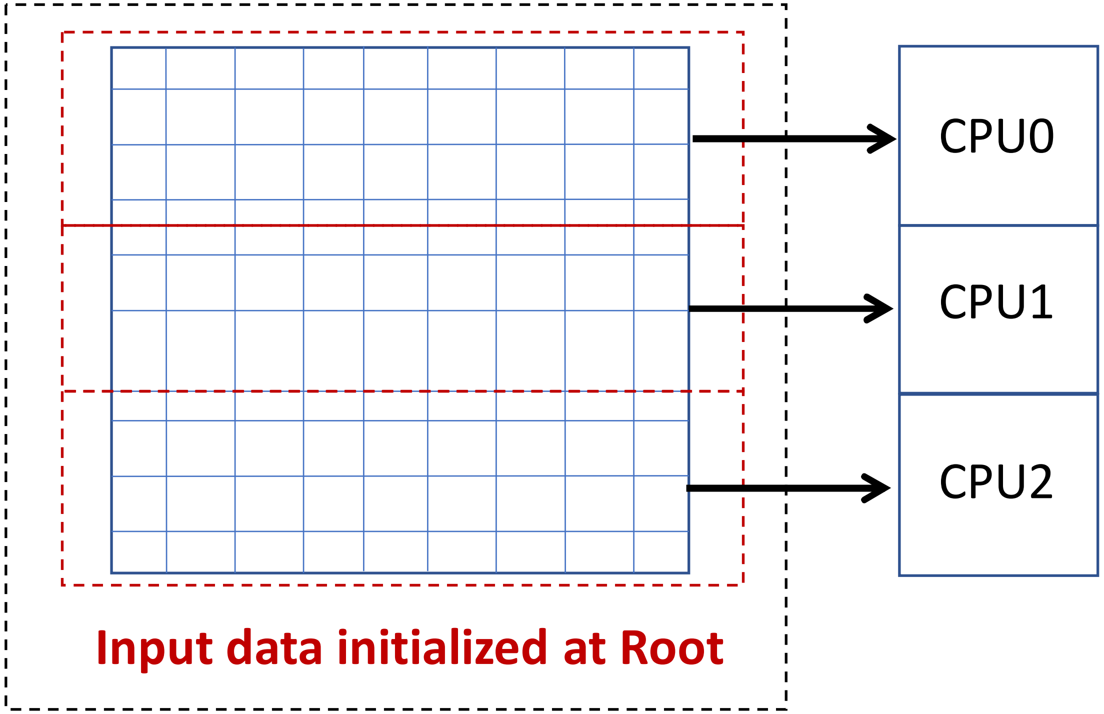
- B2: Domain decomposition
  - Many approaches
  - Different approach has different efficiency
  - Following is row-wise domain decomposition
    - Given that the size of domain is: mxn
    - Subdomain for each CPU: mcxn, where: **mc=m/NP** with **NP** is the number of CPUs



SPMD Parallel Algorithm SPMD (5)

- B3: Distribute Input data from Root to all other CPUs

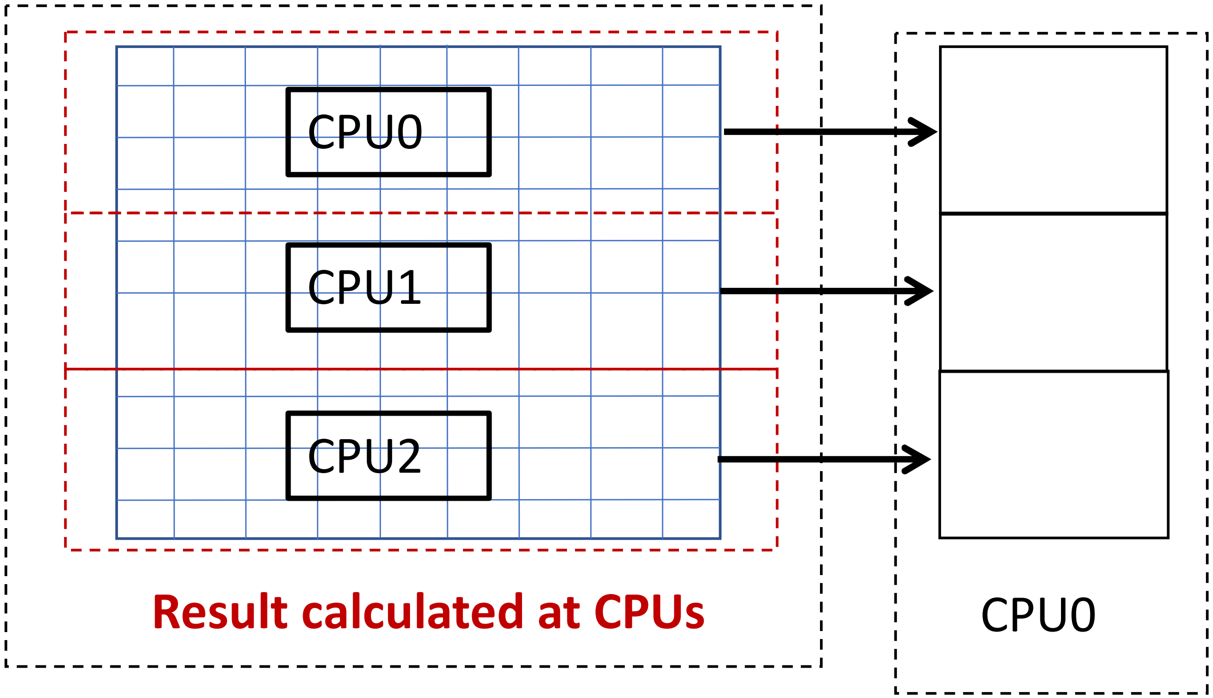
```
MPI_Scatter (C, mc*n, MPI_FLOAT,
             Cs, mc*n, MPI_FLOAT, 0,
             MPI_COMM_WORLD);
```



SPMD Parallel Algorithm SPMD (6)

- B5: Gather Output from all other CPU to Root

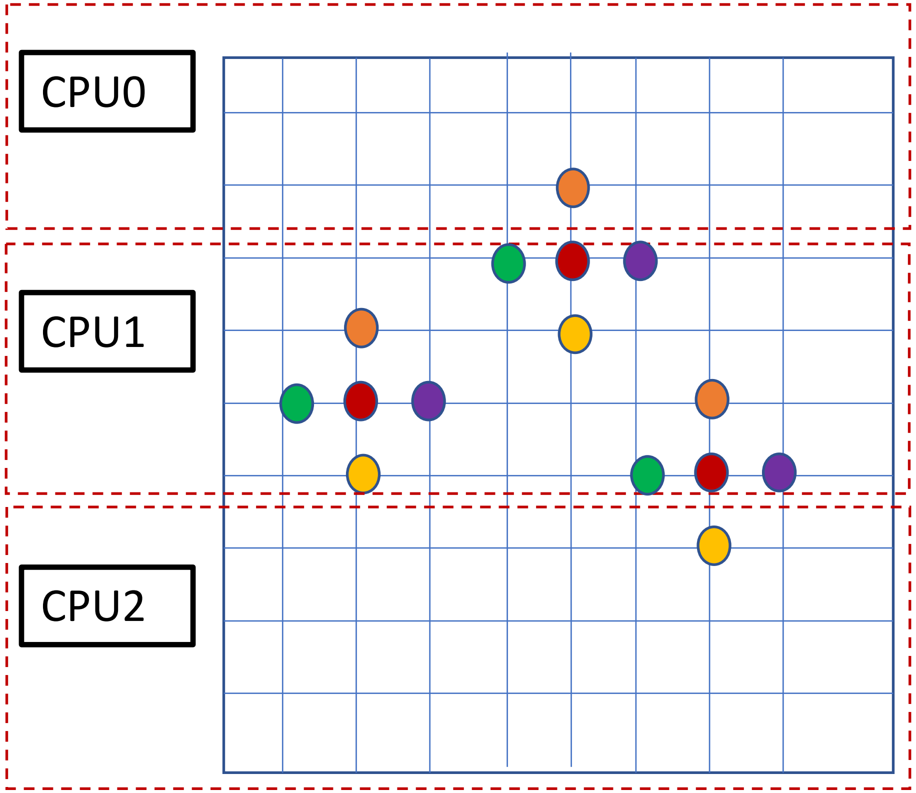
```
MPI_Gather ( Cs, mc*n, MPI_FLOAT,
             C, mc*n, MPI_FLOAT, 0,
             MPI_COMM_WORLD);
```



SPMD Parallel Algorithm SPMD (7)

- B4: Computation

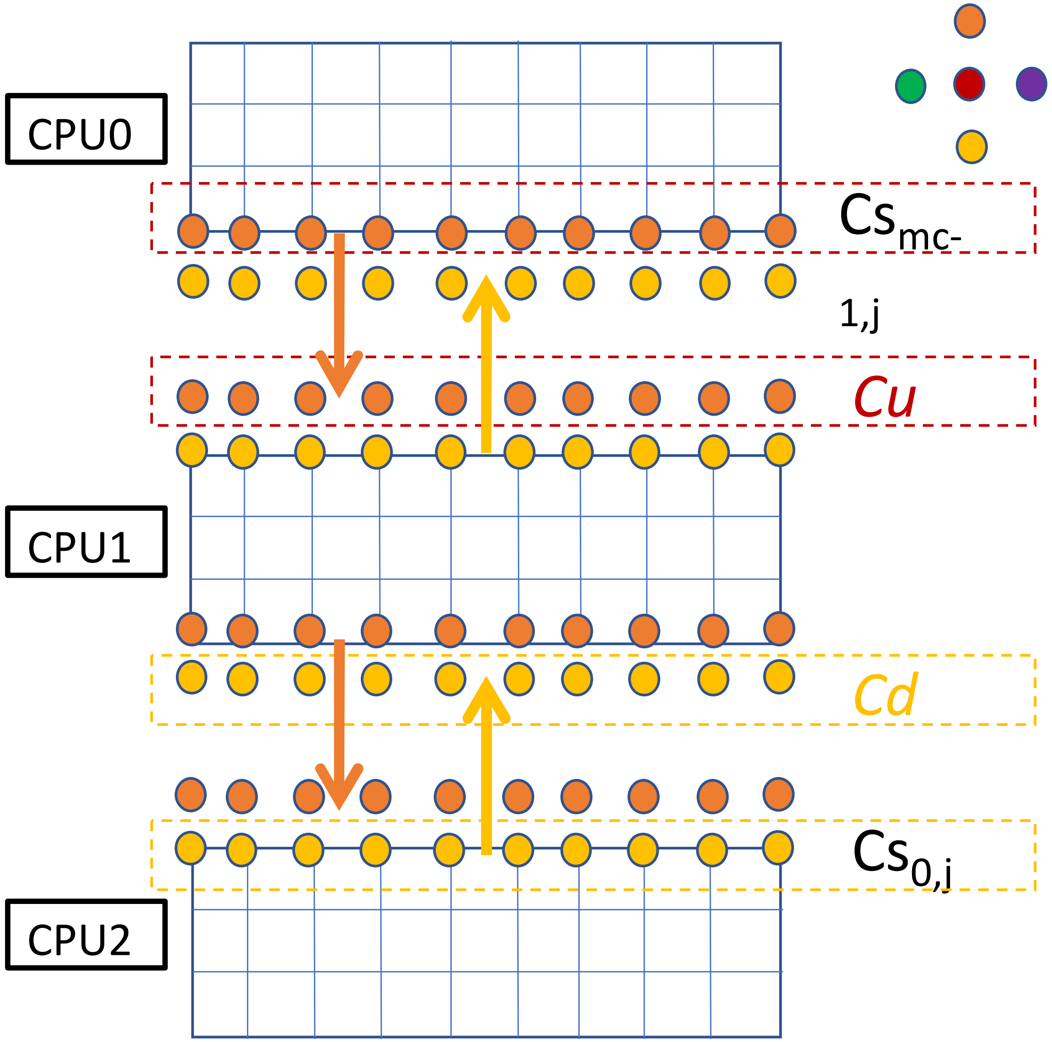
- *B4.1: Communication*
- *B4.2: Calculation*



SPMD Parallel Algorithm SPMD (8)

- B4.1: Communication

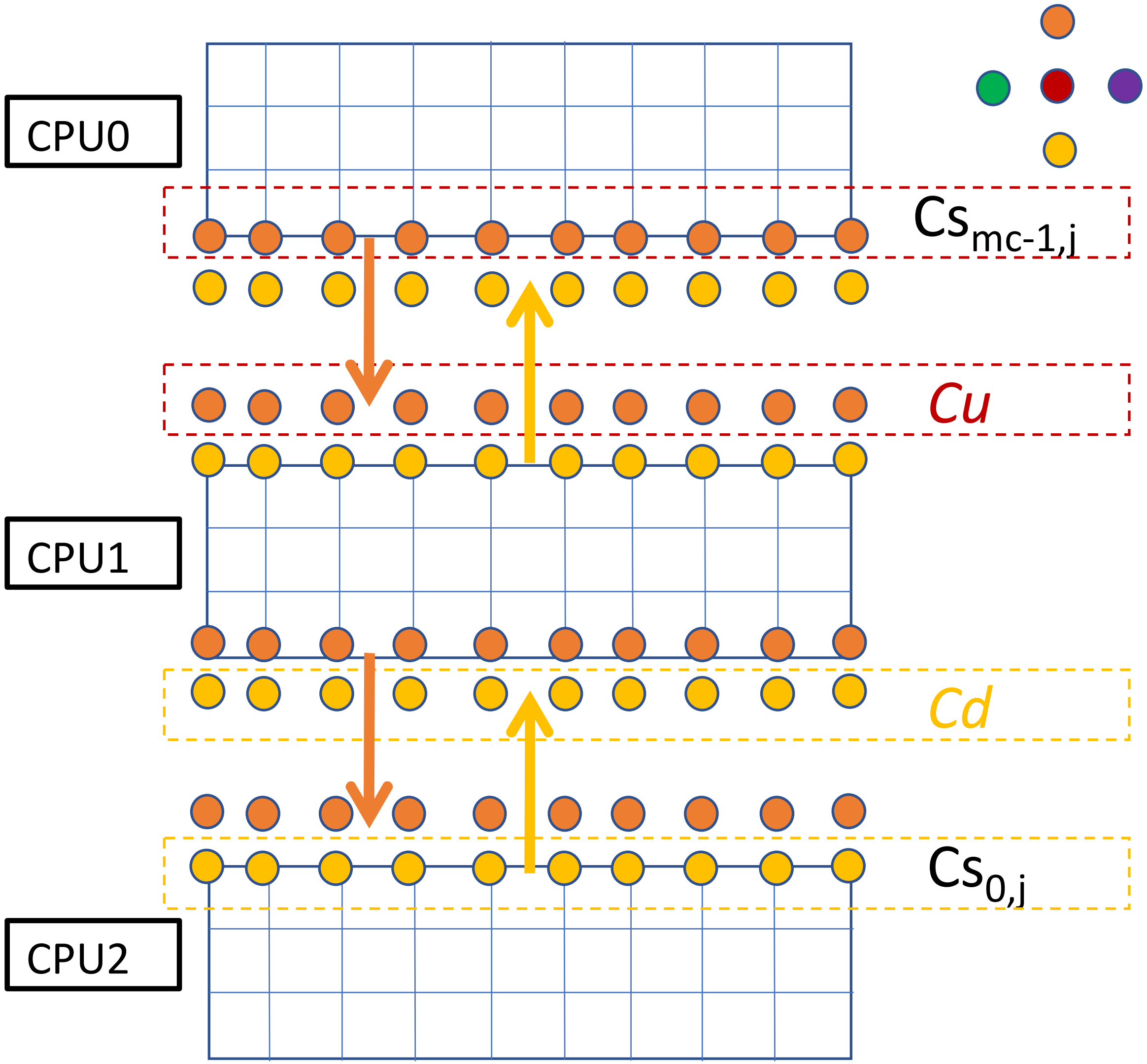
- B4.1a): Communicate array ***Cu***
- B4.1b): Communicate array ***Cd***



SPMD Parallel Algorithm SPMD (9)

- B4.1a): Communicate array *Cu*

```
if (rank==0){
    for (j=0; j<n; j++)  *(Cu+j) = *(Cs+0*n+j);
    MPI_Send (Cs+(mc- )*n, n, MPI_FLOAT, rank+1, rank, …);
  } else if (rank==NP-1) {
    MPI_Recv (Cu, n, MPI_FLOAT, rank-1, rank-1, …);
  } else {
    MPI_Send (Cs+(mc-1)*n, n, MPI_FLOAT, rank+1, rank,…);
    MPI_Recv(Cu, n, MPI_FLOAT, rank-1, rank-1, …);
  }
```



CPU0
$Cs_{mc-1,j}$
Cu
CPU1
Cd
$Cs_{0,j}$
CPU2

SPMD Parallel Algorithm SPMD (10)

- B4.1b): Communicate array ***Cd***

```
if (rank==NP-1){
  for (j=0; j<n; j++)  *(Cd+j) = *(Cs+(mc-1)*n+j);
  MPI_Send (Cs, n, MPI_FLOAT, rank-1, rank, …);
 } else if (rank==0) {
  MPI_Recv (Cd, n, MPI_FLOAT, rank+1, rank+1, …);
 } else {
  MPI_Send (Cs, n, MPI_FLOAT, rank-1, rank, …);
  MPI_Recv (Cd, n, MPI_FLOAT, rank+1, rank+1, …);
 }
```

SPMD Parallel Algorithm SPMD (11)

- B4.2: Calculation

```
void FD(float *Cs, float *Cu, float *Cd, float *dCs ,int ms) {
int i, j;
float c,u,d,l,r;
for (  i = 0 ; i < ms ; i++ )
  for ( j = 0 ; j < n ; j++ )
    {
      c = *(Cs+i*n+j);
      u = (i==0)        ?  *(Cu+j)        : *(Cs+(i-1)*n+j);
      d = (i==ms-1)     ?  *(Cd+j)        : *(Cs+(i+1)*n+j);
      l = (j==0)     ?  *(Cs+i*n+j)  : *(Cs+i*n+j-1);
      r = (j==n-1)      ?  *(Cs+i*n+j) : *(Cs+i*n+j+1);
      *(dCs+i*n+j) = (D/(dx*dx))*(u+d+l+r-4*c);
    }
}
```

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict