矩阵向量乘积并行算法

(基于 OpenMP / 共享内存)

潘建瑜

华东师范大学

目录页

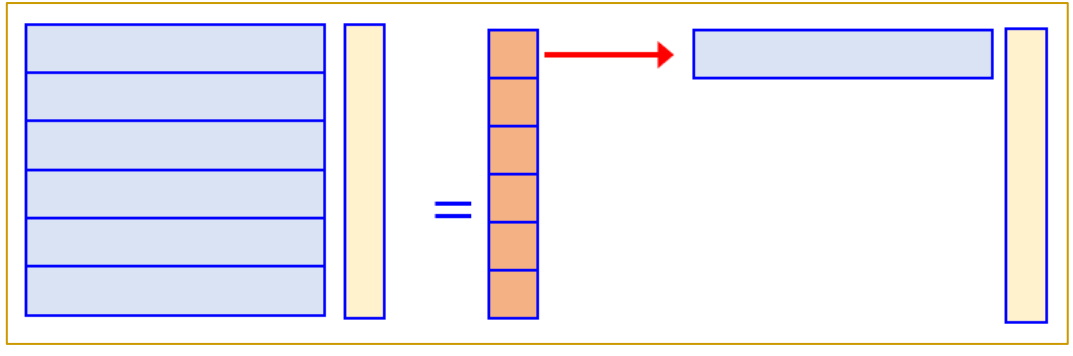
Contents

1 矩阵向量乘积串行算法

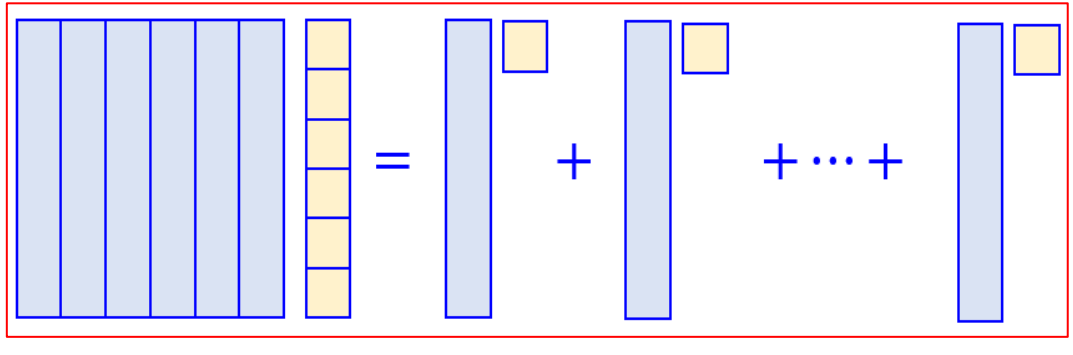
串行算法

—— 两种不同计算顺序

$$y = Ax \quad \left(A \in \mathbb{R}^{m \times n},\ x \in \mathbb{R}^{n}\right)$$



```
for i=1 to m  // i-j 循环，行存储友好
    for j=1 to n
        y(i)=y(i)+A(i,j)*x(j)
    end for
end for
```



```
for j=1 to n // j-i 循环，列存储友好
    for i=1 to m
        y(i)=y(i)+A(i,j)*x(j)
    end for
end for
```

2 矩阵向量乘积 OMP 并行算法

并行算法

—— 自动并行：循环结构共享

自动并行：for 循环结构共享

OMP_MatVec_for.c

```
double A[m][n], x[n], y[m];
... ...

#pragma omp parallel for shared(A,x,y2) private(j)
for(i = 0; i < m; i++)
{
    y2[i] = 0.0;
    for(j = 0; j < n; j++)
        y2[i] += A[i][j] * x[j];
}
```

2 矩阵向量乘积 OMP 并行算法

并行算法

—— 手工并行：按 行 划分数据

手工划分：按 **行** 划分数据

$$Ax = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix} x = \begin{bmatrix} A_0 x \\ A_1 x \\ \vdots \\ A_{p-1} x \end{bmatrix}$$

每个线程计算 $A_i x$ 。

注：划分时需考虑负载平衡和缓存行对齐。

手工划分：按 **行** 划分数据

```
#pragma omp parallel shared(A,x,y2) private(i,j,tid,nthreads)
{
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    for(i=tid; i<m; i+=nthreads)
    {
        y2[i] = 0.0;
        for(j=0; j<n; j++)
           y2[i] += A[i][j] * x[j];
    }
}
```

OMP_MatVec_row.c

注：这里用的是卷帘方式划分数据，建议用分块方式。

2 矩阵向量乘积 OMP 并行算法

并行算法

—— 手工并行：按 **列** 划分数据

手工划分：按 **列** 划分数据

将矩阵 A 按列划分，并对 x 也做相应的划分，即

$$Ax = \left[A_0, A_1, \ldots, A_{q-1} \right] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{bmatrix} = A_0 x_0 + A_1 x_1 + \cdots + A_{p-1} x_{p-1}$$

每个线程计算 $A_i x_i$，然后求和。

矩阵矩阵乘积并行算法

(基于 OpenMP / 共享内存)

目录页

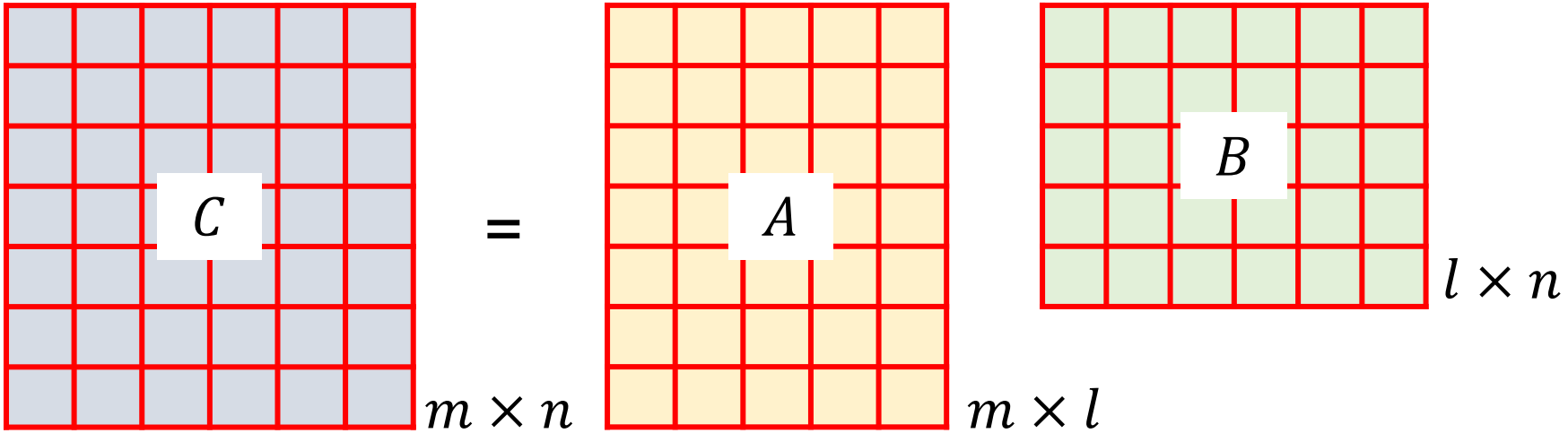
Contents

1 矩阵乘积串行算法

串行算法

—— 六种不同计算顺序

$C = AB$



六种不同顺序的循环：

IKJ、KIJ、IJK、JIK、KJI、JKI，

详见课程主页“矩阵乘积的快速算法”

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

2 矩阵乘积 OMP 并行算法

并行算法

—— 自动并行：循环结构共享

自动并行：for 循环结构共享

```
#pragma omp parallel for shared(A,B,C) ......
for(i=0; i<M; i++)
    for(j=0; j<N; j++)
    {
        C[i][j]=0;
        for(k=0; k<L; k++)
            C[i][j]=C[i][j] + A[i][k]*B[k][j];
    }
```

OMP_matmul_for.c

2 矩阵乘积 OMP 并行算法

并行算法

—— 手工并行：行、列、二维

手工并行：数据（任务）划分

$$AB = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix} \begin{bmatrix} B_0 & B_1 & \cdots & B_{p-1} \end{bmatrix} = \begin{bmatrix} A_0B_0 & A_0B_1 & \cdots & A_0B_{p-1} \\ A_1B_0 & A_1B_1 & \cdots & A_1B_{p-1} \\ \vdots & & \ddots & A_0B_{p-1} \\ A_{p-1}B_0 & A_{p-1}B_1 & \cdots & A_{p-1}B_{p-1} \end{bmatrix}$$

手工并行 ⇒ 任务分配：由用户分配计算任务

（即每个线程负责计算 C 的哪些部分）

⇓

C：按行分块、按列分块、二维分块

假定：M, L, N 均能能 p 整除，其中 p 为线程个数。（行和列的块数可以不一样）

手工并行：按 **行** 分配任务

记 $A=\begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix}$ ⇒ $C=AB=\begin{bmatrix} A_0B \\ A_1B \\ \vdots \\ A_{p-1}B \end{bmatrix}$

按行分配任务

- 第 i 号线程负责计算 C_i，其中 $C_i = A_iB$

按 **行** 分配任务举例

例：按行分配任务，并行计算矩阵乘积，其中

$$A=\left[a_{ij}\right]\in\mathbb{R}^{n\times n},\quad a_{ij}=\frac{1}{i+j-1}$$

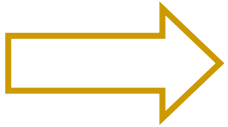
$$B=\left[b_{ij}\right]\in\mathbb{R}^{n\times n},\quad b_{ij}=i+j-1$$

（取 n=1024, p=4）

OMP_MatMul_rc.c

手工并行：按 **列** 分配任务

$$B = \left[B_0, B_1, \ldots, B_{p-1} \right]$$



$$C = \left[AB_0, AB_1, \ldots, AB_{p-1} \right]$$

按列分配任务

- 第 i 号线程负责计算 C_i，其中 $C_i = AB_i$

手工并行：**二维** 划分任务

$$A=\begin{bmatrix}A_0\\A_1\\\vdots\\A_{p-1}\end{bmatrix},\quad B=\begin{bmatrix}B_0,B_1,\ldots,B_{q-1}\end{bmatrix}$$

$$C=\begin{bmatrix}A_0B_0 & A_0B_1 & \cdots & A_0B_{q-1}\\A_1B_0 & A_1B_1 & \cdots & A_1B_{q-1}\\\vdots & \vdots & \ddots & \vdots\\A_{p-1}B_0 & A_{p-1}B_1 & \cdots & A_{p-1}B_{q-1}\end{bmatrix}$$

二维方式（行列）分配任务

- 线程总数为 **nthreads** = $p\times q$，其中 p 和 q 为两个正整数
- 第 (i, j) 号线程负责计算 C_{ij}，其中 $C_{ij}=A_iB_j$

矩阵向量乘积并行算法

(基于 MPI / 分布式并行计算)

矩阵向量乘积 MPI 并行算法

并行算法

—— 按 行 划分数据

注：P_i 表示第 i 个处理器或第 i 个进程

并行算法：按行划分

将矩阵 A 按行划分成行块子矩阵

$$Ax = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix} x = \begin{bmatrix} A_0 x \\ A_1 x \\ \vdots \\ A_{p-1} x \end{bmatrix}$$

数据存储方案

- 矩阵：按行划分，存储在各个处理器中
- 向量 x：每个处理器都存储 x
- 向量 y：每个处理器计算一部分，在 0 号进程中合并，并广播给其他进程

- 将 A_i 存放在处理器 P_i 中，每个处理器计算 $A_i x$
- 最后调用 **MPI_Gather** 或 **MPI_Gatherv** 即可

举例

例：按 行 划分，用 p 个进程并行计算矩阵向量乘积，其中

$$A=\left[a_{ij}\right]\in\mathbb{R}^{n\times n},\quad a_{ij}=\frac{1}{i+j-1},\qquad x=[1,\ 2,\ \ldots,\ n]^T\in\mathbb{R}^n$$

取 n=1024, p=4, 矩阵划分：按顺序连续划分，即：

0 号进程存储 A[0:255, :]，

1 号进程存储 A[256:511, :]，依次类推。

MPI_matvec.c　　MPI_matvec_v.c

思考：x 也分块存储在不同处理器中

数据存储方案：矩阵 A 按行分块，x 也做相应的分块

$$Ax=\begin{bmatrix}A_0\\A_1\\\vdots\\A_{p-1}\end{bmatrix}x=\begin{bmatrix}A_{0,0} & A_{0,1} & \cdots & A_{0,p-1}\\A_{1,0} & A_{1,1} & \cdots & A_{1,p-1}\\ & & & \\A_{p-1,0} & A_{p-1,1} & \cdots & A_{p-1,p-1}\end{bmatrix}\begin{bmatrix}x_0\\x_1\\\vdots\\x_{p-1}\end{bmatrix}$$

处理器 P_0

处理器 P_1

处理器 P_{p-1}

计算方案

- A 保持不变，x 在各个处理器中轮转（向上）

矩阵向量乘积 MPI 并行算法

并行算法

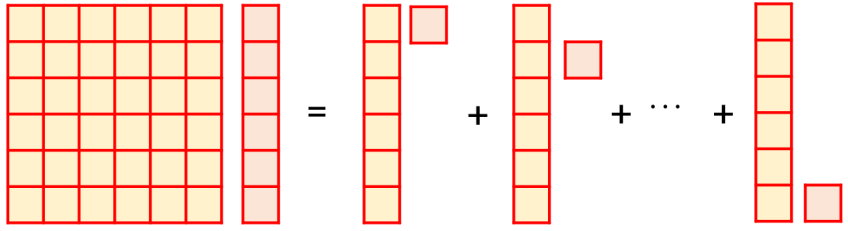
—— 按 列 划分数据

注：P_i 表示第 i 个处理器或第 i 个进程

并行算法：按列划分

将矩阵 A 按列划分，并对 x 也做相应的划分

$$Ax = \left[A_0, A_1, \ldots, A_{q-1}\right] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{bmatrix} = A_0 x_0 + A_1 x_1 + \cdots + A_{p-1} x_{p-1}$$



- 将 A_i 和 x_i 存放在 P_i 中，每个处理器计算 $A_i x_i$
- 最后调用 **MPI_Reduce** 或 **MPI_Allreduce** 即可。

矩阵矩阵乘积并行算法

(基于 MPI / 分布式并行计算)

矩阵乘积 MPI 并行算法

—— 行列划分、行行划分、列列划分、列行划分

—— Cannon 算法、Fox 算法

一些记号和设定

- 假设有 p 个处理器/进程/结点，每个结点运行一个进程
- P_j 表示第 j 个结点，P_{myid} 表示当前结点或进程
- **send(x; j)** 表示在 P_{myid} 中把数据 x 发送给 P_j
- **recv(x; j)** 表示 P_{myid} 从 P_j 接收数据块 x
- i **mod** p 表示 i 对 p 做模运算

并行算法 $C = A \times B$

并行计算 → 由用户分配 **数据** 与 **计算任务**

↓

对 A、B 进行分块

↓

行列划分、行行划分、列列划分、列行划分

† 为了描述方便，假定 m, l, n 均能能 p 整除，其中 p 为 **进程** 个数。

矩阵向量乘积 MPI 并行算法

并行算法

—— 行列 划分

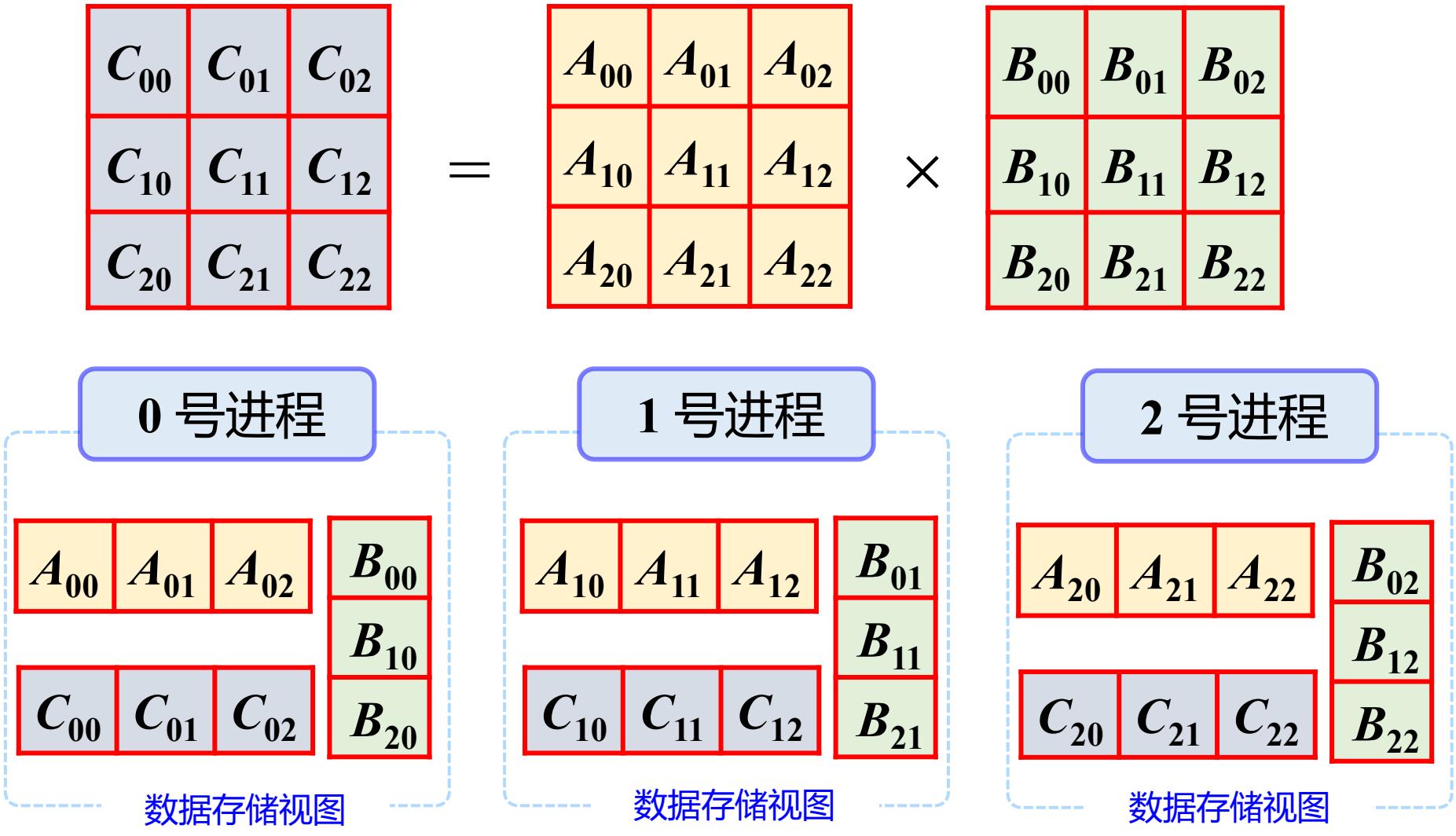
并行算法：行列划分

行列划分

$$AB=\begin{bmatrix}A_0\\A_1\\\vdots\\A_{p-1}\end{bmatrix}\begin{bmatrix}B_0 & B_1 & \cdots & B_{p-1}\end{bmatrix}=\begin{bmatrix}A_0B_0 & A_0B_1 & & A_0B_{p-1}\\A_1B_0 & A_1B_1 & & A_1B_{p-1}\\\vdots & & \ddots & \vdots\\A_{p-1}B_0 & A_{p-1}B_1 & & A_{p-1}B_{p-1}\end{bmatrix}=\begin{bmatrix}C_{ij}\end{bmatrix}$$

数据存储与计算方案

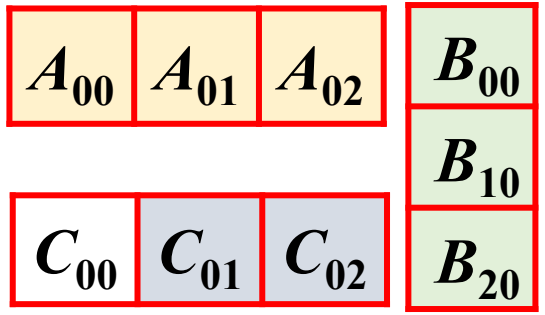
- 存储方案：A_i, B_i 和 C_{ij} ($j=0, 1, ..., p-1$) 存放在第 i 个处理器中，按循环方式交换数据 B_i
- P_i 负责计算 C_{ij} ($j=0, 1, ..., p-1$)
- 由于使用 p 个处理器，每次每个处理器只计算一个 C_{ij}，故计算出整个 C 需要 p 次完成
- C_{ij} 的计算是按对角线进行的


C_{00} C_{01} C_{02}
C_{10} C_{11} C_{12}
C_{20} C_{21} C_{22}
=
A_{00} A_{01} A_{02}
A_{10} A_{11} A_{12}
A_{20} A_{21} A_{22}
×
B_{00} B_{01} B_{02}
B_{10} B_{11} B_{12}
B_{20} B_{21} B_{22}
0 号进程
A_{00} A_{01} A_{02}
B_{00}
B_{10}
B_{20}
C_{00} C_{01} C_{02}
数据存储视图
1 号进程
A_{10} A_{11} A_{12}
B_{01}
B_{11}
B_{21}
C_{10} C_{11} C_{12}
数据存储视图
2 号进程
A_{20} A_{21} A_{22}
B_{02}
B_{12}
B_{22}
C_{20} C_{21} C_{22}
数据存储视图

第 1 步：计算

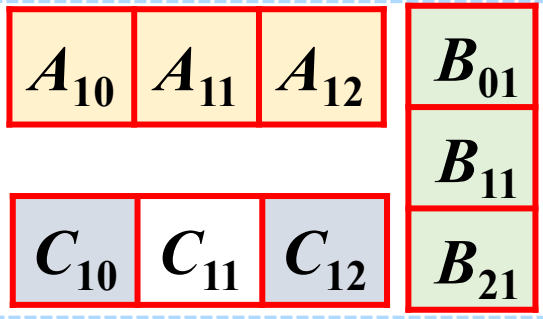
0号进程

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$



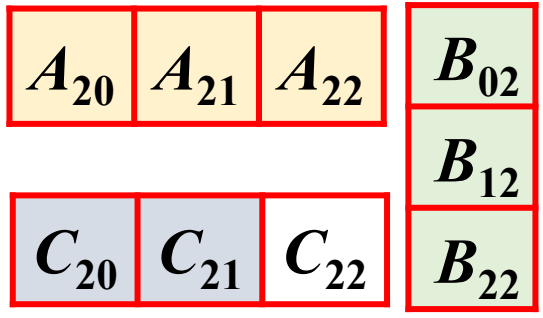
1号进程

$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

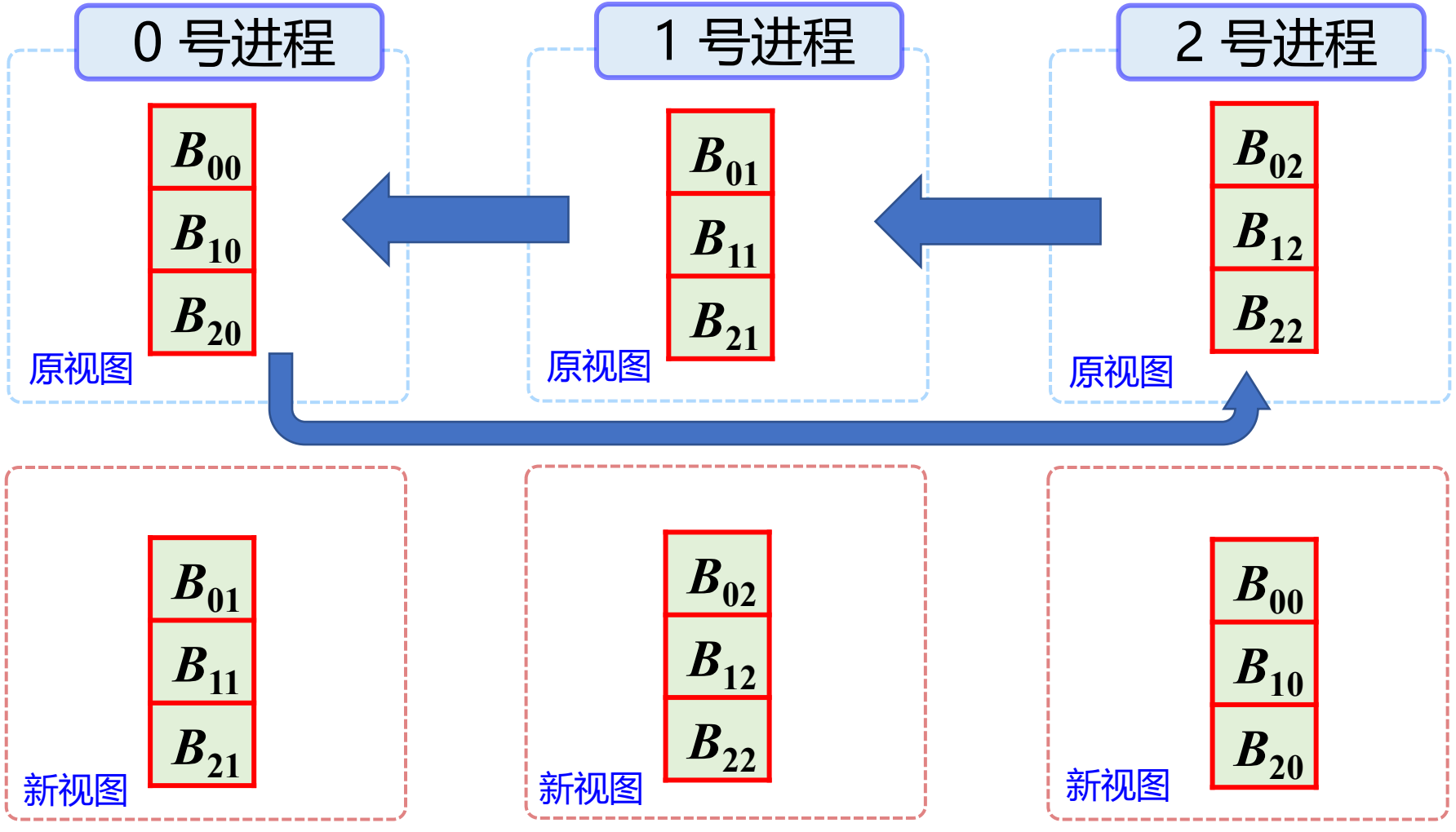


2号进程

$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$



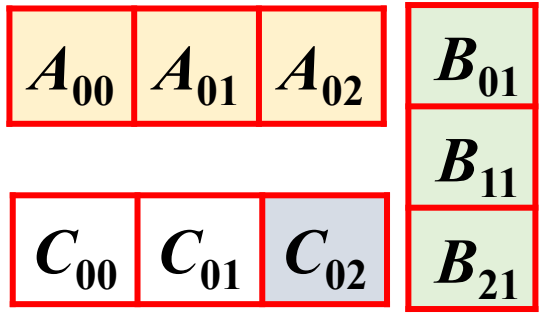
第 2 步：数据传递


0 号进程
1 号进程
2 号进程
B_{00}
B_{10}
B_{20}
B_{01}
B_{11}
B_{21}
B_{02}
B_{12}
B_{22}
原视图
原视图
原视图
B_{01}
B_{11}
B_{21}
B_{02}
B_{12}
B_{22}
B_{00}
B_{10}
B_{20}
新视图
新视图
新视图

第 3 步：计算

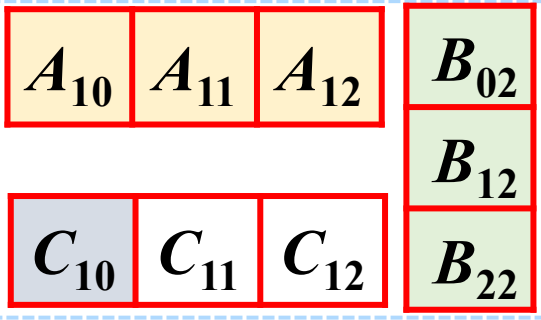
0号进程

$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$



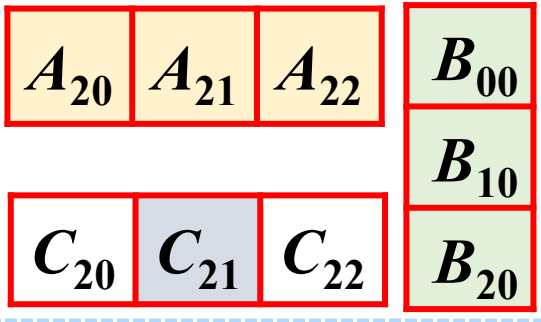
1号进程

$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

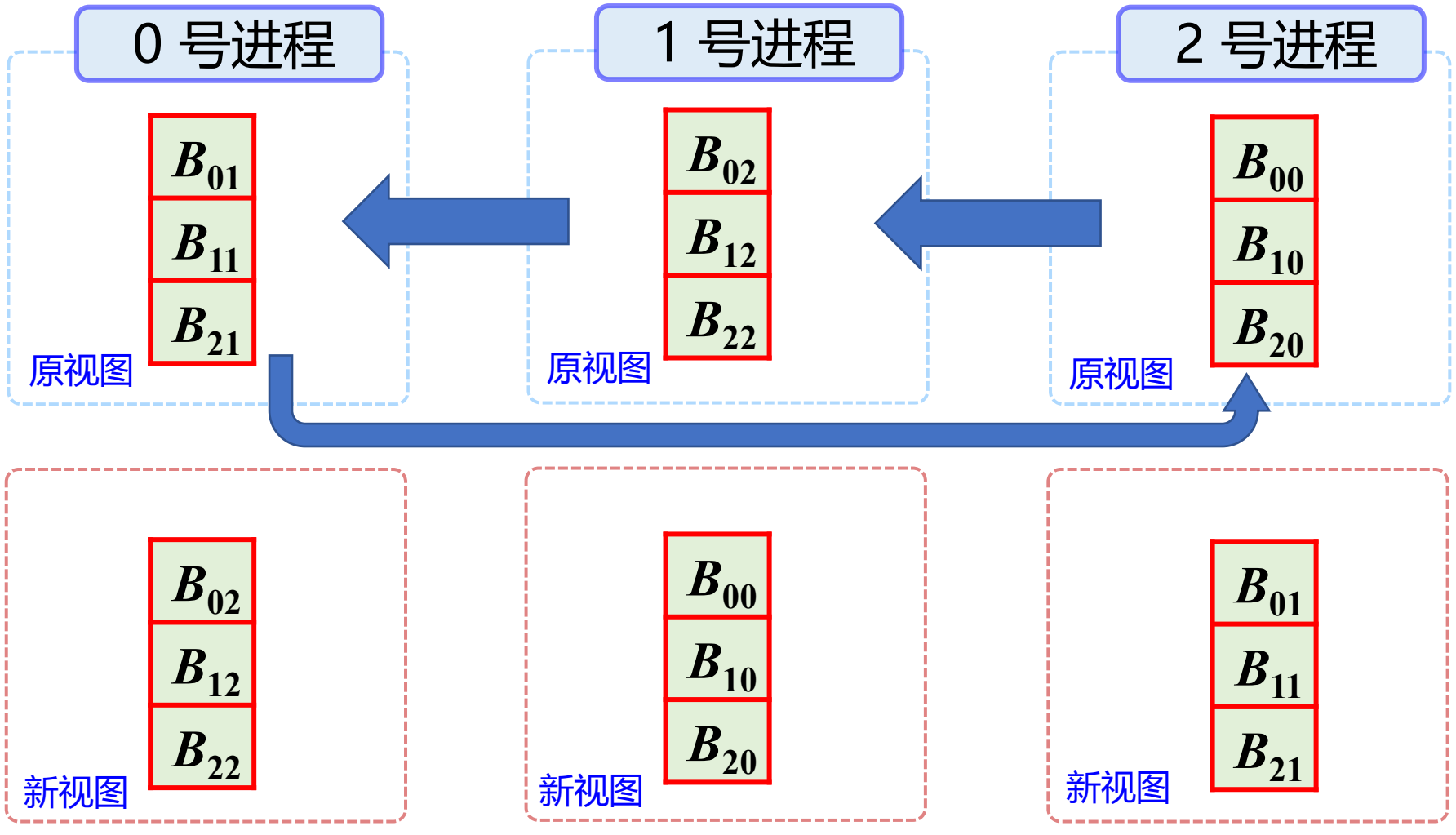


2号进程

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$



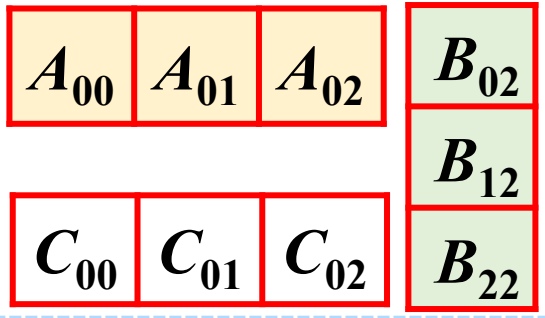
第 4 步：数据传递


0 号进程
1 号进程
2 号进程
B_{01}
B_{11}
B_{21}
B_{02}
B_{12}
B_{22}
B_{00}
B_{10}
B_{20}
原视图
原视图
原视图
B_{02}
B_{12}
B_{22}
B_{00}
B_{10}
B_{20}
B_{01}
B_{11}
B_{21}
新视图
新视图
新视图

第 5 步：计算

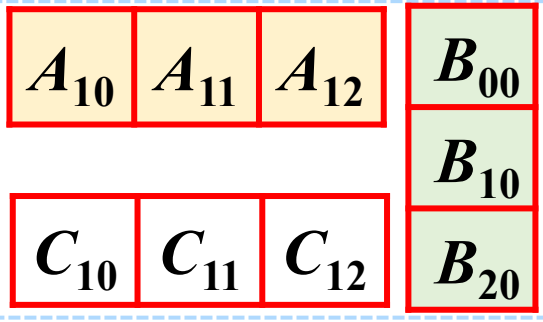
0号进程

$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$



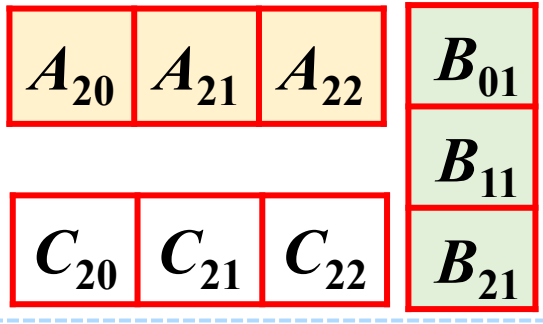
1号进程

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$



2号进程

$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$



并行算法：行列划分

```
for i=0 to p-1
   j=(i+myid) mod p
   Cj=A*B
   src = (myid+1) mod p
   dest = (myid-1+p) mod p
   if (i!=p-1)
      send(B,dest)
      recv(B,src)
   end if
end for
```

- 本算法中，$C_j = C_{myid,j}$，$A = A_{myid}$，B 在处理器中每次循环向前移动一个处理器，即每次交换一个子矩阵数据块，共交换 p-1 次。

举例

例：按 行列 划分，用 p 个进程并行计算矩阵矩阵乘积，其中

$$A=\left[a_{ij}\right]\in\mathbb{R}^{n\times n},\quad a_{ij}=\frac{1}{i+j-1},\qquad B=\left[b_{ij}\right]\in\mathbb{R}^{n\times n},\quad b_{ij}=i+j-1$$

取 n=1024, p=4, 矩阵划分：按顺序连续划分，并假定 n 能被 p 整除。

MPI_matmul.c

矩阵向量乘积 MPI 并行算法

并行算法

—— 行行 划分

并行算法：行行划分

行行划分

$$A=\begin{bmatrix}A_0\\A_1\\\vdots\\A_{p-1}\end{bmatrix},\ B=\begin{bmatrix}B_0\\B_1\\\vdots\\B_{p-1}\end{bmatrix},\ C=\begin{bmatrix}C_0\\C_1\\\vdots\\C_{p-1}\end{bmatrix}$$

数据存储与计算方案

- 存储方案：A_i，B_i和 C_i 存放在第 i 个处理器中，P_i 负责计算 C_i
- 实际计算时需要对 A_i 做进一步划分（按列，与 B 的行分块相对应）

$$A_i=\left[A_{i,0},A_{i,1},\ldots,A_{i,p-1}\right] \Rightarrow C_i=A_{i,0}B_0+A_{i,1}B_1+\cdots+A_{i,p-1}B_{p-1}$$

- 每次每个处理器计算其中一个乘积，然后对 B_i 进行数据交换，整个过程需 p 次完成

并行算法：行行划分

```
for i=0 to p-1
   j = (i+myid) mod p
   C = C + Aj*B
   src = (myid+1) mod p
   dest = (myid-1+p) mod p
   if (i!=p-1)
      send(B,dest)
      recv(B,src)
   end if
end for
```

- 本算法中，$C = C_{myid}$，$A_j = A_{myid,j}$，B 在处理器中每次循环向前移动一个处理器
- 本算法中的数据交换量和计算量均与行列划分相同，不同的只是计算 C 的方式。

并行算法

—— 列列 划分

并行算法：列列划分

列列划分

$$A=\left[A_0,A_1,\ldots,A_{p-1}\right],\quad B=\left[B_0,B_1,\ldots,B_{p-1}\right],\quad C=\left[C_0,C_1,\ldots,C_{p-1}\right]$$

数据存储与计算方案

- 存储方案：A_i，B_i和 C_i 存放在第 i 个处理器中，P_i 负责计算 C_i
- 实际计算时需要对 B_j 做进一步划分（按列，与 A 的列分块相对应）

$$B_j=\begin{bmatrix}B_{0,j}\\B_{1,j}\\\vdots\\B_{p-1,j}\end{bmatrix}\Rightarrow C_j=A_0B_{0,j}+A_1B_{1,j}+\cdots+A_{p-1}B_{p-1,j}$$

- 与行行划分类似，但进行数据传递的是 A_i

并行算法：列列划分

```
for i=0 to p-1
   k = (i+myid) mod p
   C = C + A*Bk
   src = (myid+1) mod p
   dest = (myid-1+p) mod p
   if (i!=p-1)
      send(A,dest)
      recv(A,src)
   end if
end for
```

- 本算法中，$C = C_{myid}$，$B_k = B_{k,myid}$，A 在处理器中每次循环向前移动一个处理器
- 本算法计算量与前面相同，当 $m \neq n$ 时，通信量有所不同，在具体应用时可以根据实际情况选择合适的算法。

并行算法

—— 列行 划分

并行算法：列行划分

列行划分

$$A=\left[A_0,A_1,\ldots,A_{p-1}\right],\quad B=\begin{bmatrix}B_0\\B_1\\\vdots\\B_{p-1}\end{bmatrix},\quad C=A_0B_0+A_1B_1+\cdots+A_{p-1}B_{p-1}$$

数据存储与计算方案

- 存储方案：A_i，B_i和 C_i 存放在第 i 个处理器中，P_i 负责计算 C_i
- 实际计算时需要对 B_j 做进一步划分（按列，与 A 的列分块相对应）

$$B_i=\left[B_{i,0},B_{i,1},\ldots,B_{i,p-1}\right]\Rightarrow C_i=A_0B_{0,i}+A_1B_{1,i}+\cdots+A_{p-1}B_{p-1,i}$$

- 每次计算后更新 C，需要指出的是，进行数据传递的是 C

矩阵向量乘积 MPI 并行算法

并行算法

—— Cannon 算法与 Fox 算法

并行算法：Cannon 算法

为了讨论方便，我们做以下假定

- 矩阵 A，B，C 维数相等且都可以写成 $m \times m$ 分块矩阵：

$$A=\left[A_{ij}\right]_{m\times m}，\ B=\left[B_{ij}\right]_{m\times m}，\ C=\left[C_{ij}\right]_{m\times m}$$

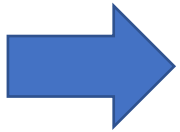
  其中 A_{ij}，B_{ij}，C_{ij} 都是 $n \times n$ 的。

- 使用 m^2 个处理器来计算，分别标号为 P_{ij}

并行算法：Cannon 算法

- 定义分块置换矩阵 Q：

$$Q=\begin{bmatrix} 0 & I & 0 & \cdots & 0 \\ 0 & 0 & I & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & I \\ I & 0 & 0 & \cdots & 0 \end{bmatrix}$$



- 矩阵 A 左乘 Q，即 QA：

  将 A 的所有行（分块意义下）向 **上** 移一位

- 矩阵 A 右乘 Q，即 AQ：

  将 A 的所有列（分块意义下）向 **右** 移一位

并行算法：Cannon 算法

- 定义分块对角矩阵

$$D_A^{(l)} = \begin{bmatrix} D_0^{(l)} & & \\ & \ddots & \\ & & D_{m-1}^{(l)} \end{bmatrix}, \quad \text{其中 } D_i^{(l)} = A_{i,i+l \bmod m}, \quad l = 0,1,\ldots,m-1$$

$$\Rightarrow \quad A = \sum_{l=0}^{m-1} D_A^{(l)} Q^l$$

以 3 × 3 分块矩阵为例：

$$D_A^{(0)} = \begin{bmatrix} A_{00} & & \\ & A_{11} & \\ & & A_{22} \end{bmatrix}, \quad D_A^{(1)} = \begin{bmatrix} A_{01} & & \\ & A_{12} & \\ & & A_{20} \end{bmatrix}, \quad D_A^{(2)} = \begin{bmatrix} A_{02} & & \\ & A_{10} & \\ & & A_{21} \end{bmatrix}$$

并行算法：Cannon 算法

- 计算矩阵 C

$$C = AB = \sum_{l=0}^{m-1} D_A^{(l)} Q^l B = D_A^{(0)} B^{(0)} + D_A^{(1)} B^{(1)} + \cdots + D_A^{(m-1)} B^{(m-1)}$$

其中 $B^{(l)} = Q^l B = QB^{(l-1)}$

数据存储与计算方案

根据上面的关系式，我们可以通过依次计算 C 表达式中的右端项来计算矩阵乘积

- 把处理器编号从一维映射到二维，即 $P_{myid} = P_{ij}$
- 将数据 A_{ij}，B_{ij}，C_{ij} 存放在 P_{ij} 中

Cannon 算法：以 $m \times m$ 为例，9 个进程

- A、B 的起始存放位置

A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}

- 第一轮：计算 $D_A^{(0)} B^{(0)}$

横向广播

A_{00}	A_{00}	A_{00}
A_{11}	A_{11}	A_{11}
A_{22}	A_{22}	A_{22}

B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}

- 第二轮：计算 $D_A^{(1)} B^{(1)}$

横向广播

A_{01}	A_{01}	A_{01}
A_{12}	A_{12}	A_{12}
A_{20}	A_{20}	A_{20}

纵向移位

B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}
B_{00}	B_{01}	B_{02}

- 第三轮：计算 $D_A^{(2)} B^{(2)}$

横向广播

A_{02}	A_{02}	A_{02}
A_{10}	A_{10}	A_{10}
A_{21}	A_{21}	A_{21}

纵向移位

B_{20}	B_{21}	B_{22}
B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}

处理器 P_{ij} 上的计算过程

算法 2.1. 矩阵乘积 Cannon 并行算法

1: $C = 0$
2: **for** $i = 0$ to $m - 1$ **do**　% 第 i 步计算右端的第 $i + 1$ 项
3: 　$k \equiv (myrow + i) \mod m$
4: 　$mp1 = (mycol + 1) \mod m$, 　$mm1 = (mycol - 1) \mod m$
5: 　**if** $mycol = k$ **then**　% 将 $A_{myrow,k}$ 广播给 $P_{myrow,j}$, $j = 0, \ldots, m-1, j \neq k$
6: 　　send$(A, (myrow, mp1))$; 　copy$(A, tmpA)$
7: 　**else**
8: 　　recv$(tmpA, (myrow, mm1))$
9: 　　**if** $k \neq mp1$ **then** send$(tmpA, (myrow, mp1))$
10: 　**end if**
11: 　$C = C + tmpA * B$; $mp1 = myrow + 1 \mod m$; $mm1 = myrow - 1 \mod m$
12: 　**if** $i \neq m - 1$ **then**　% 在同列中滚动 B
13: 　　send$(B, (mm1, mycol))$; recv$(B, (mp1, mycol))$
14: 　**end if**
15: **end for**

几点注记

- 该算法具有很好的负载平衡
- 数据传递特点：在同一行中广播 A（因此需要存放两个子矩阵），在同列中滚动 B（也可以是滚动 A + 广播 B）
- 与前面介绍的四种并行算法相比，计算量一样，但当进程个数大于4时，Cannon 算法的通信量较少。

Cannon 算法：第二种数据传递方式

Cannon 算法也可以通过水平方向（向左）滚动 A，垂直方向（向上）滚动 B 来实现

- 初始位置：A 的第 i 行向 **左移** 动 i-1 格，B 的第 j 列向上移动 j-1 格

A_{00}	A_{01}	A_{02}
A_{11}	A_{12}	A_{10}
A_{22}	A_{20}	A_{21}

B_{00}	B_{11}	B_{22}
B_{10}	B_{21}	B_{02}
B_{20}	B_{01}	B_{12}

- 每计算完一轮后，沿水平方向**向左滚动** A，同时沿垂直方向**向上滚动** B
- 好处：不需要 **tmpA**

Cannon 算法：第二种数据传递方式

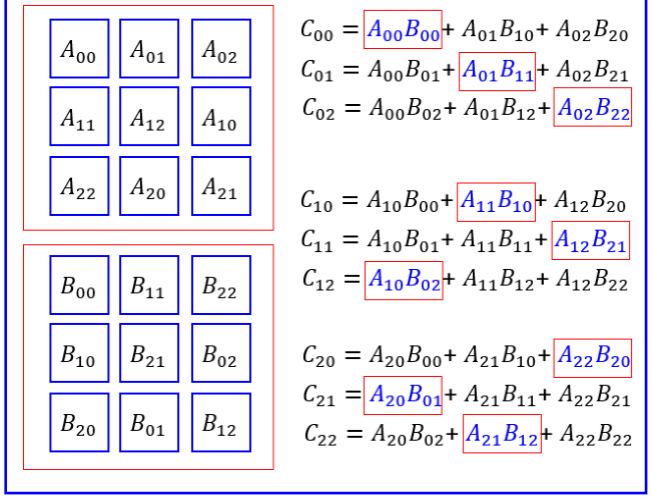
数据存储方案

- 初始的数据存储方式：

  A 按行移动，第一行不变，第二行向左移一格，第三行向左移两格，依此类推；

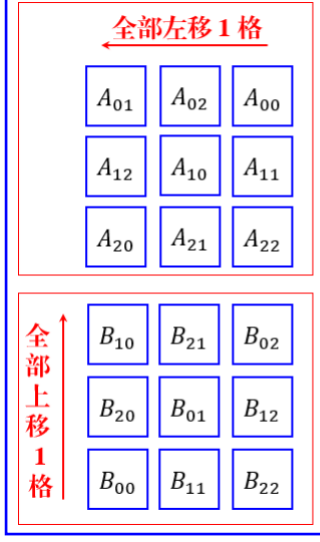
  B 则按列移动，第一列不变，第二列向上移一格，第三列向上移两格，依此类推。
- C_{ij} 仍然存放在 P_{ij} 中
- 在计算过程中，A 不断向左滚动，B 不断向上滚动

初始状态



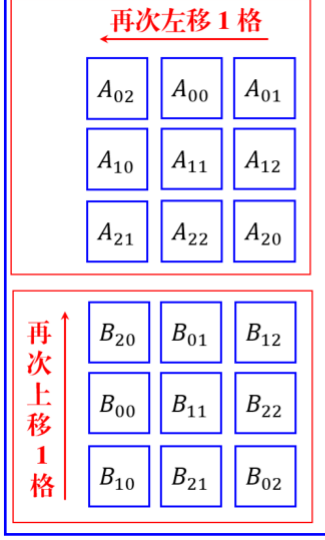
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$
$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$
$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

第一轮数据移动



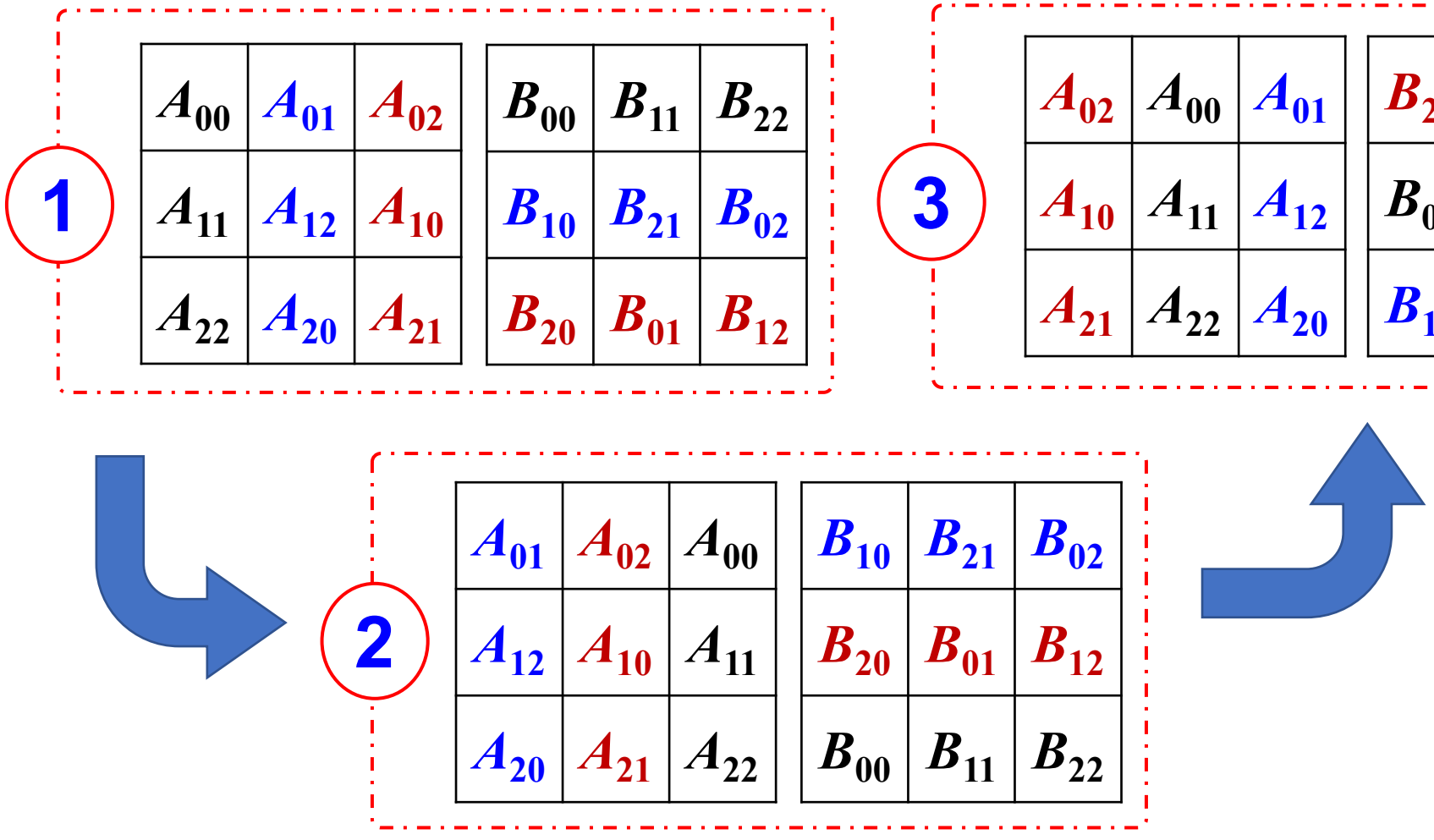
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$
$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$
$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

第二轮数据移动



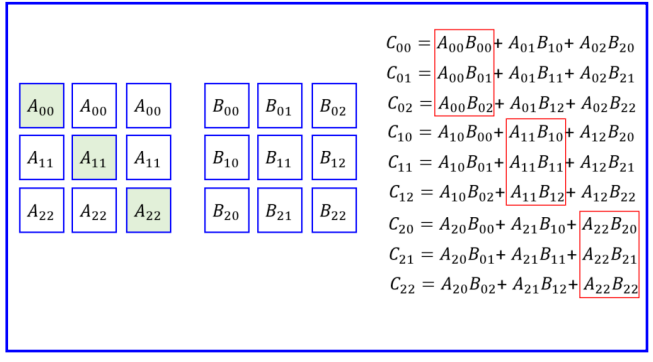
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$
$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$
$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

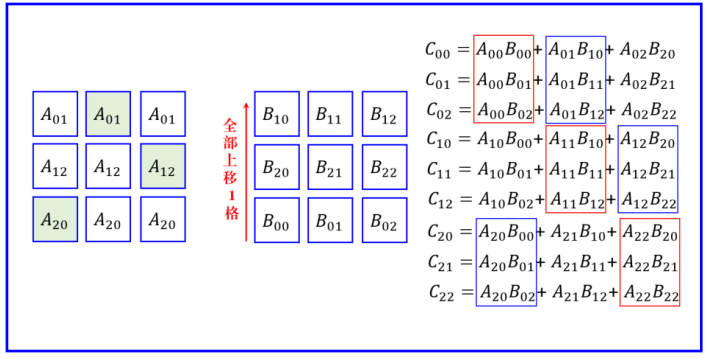
Cannon 算法：第二种数据传递方式

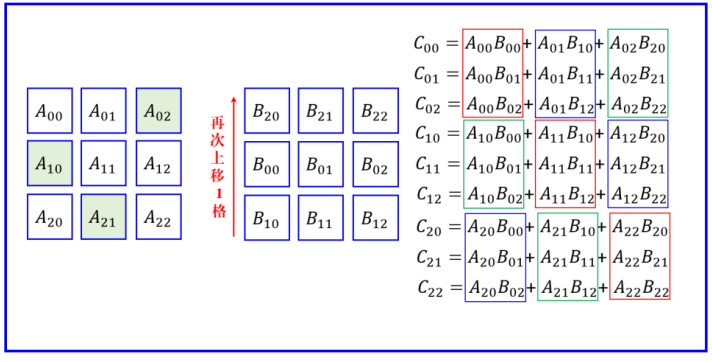


Fox 算法

- Fox 算法是计算大规模矩阵乘积的另外一种常用并行算法







- 与 Cannon 算法的区别：数据移动方式不同

谢 谢

THANK YOU