

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

# Chapter 8

## 矩阵-向量乘

# 章节目标

- 复习矩阵-向量乘法
- 分发数据以满足计算
- 基于不同数据分解的并行算法

# 大纲

- 串行算法及其复杂性
- 三个并行程序的设计、分析和实现
  - 按行分块
  - 按列分块
  - 棋盘式分块

# 串行算法

2	1	0	4
3	2	1	1
4	3	1	2
3	0	2	0

 $\times$ 

1
3
4
1

 $=$ 

9
14
19
11

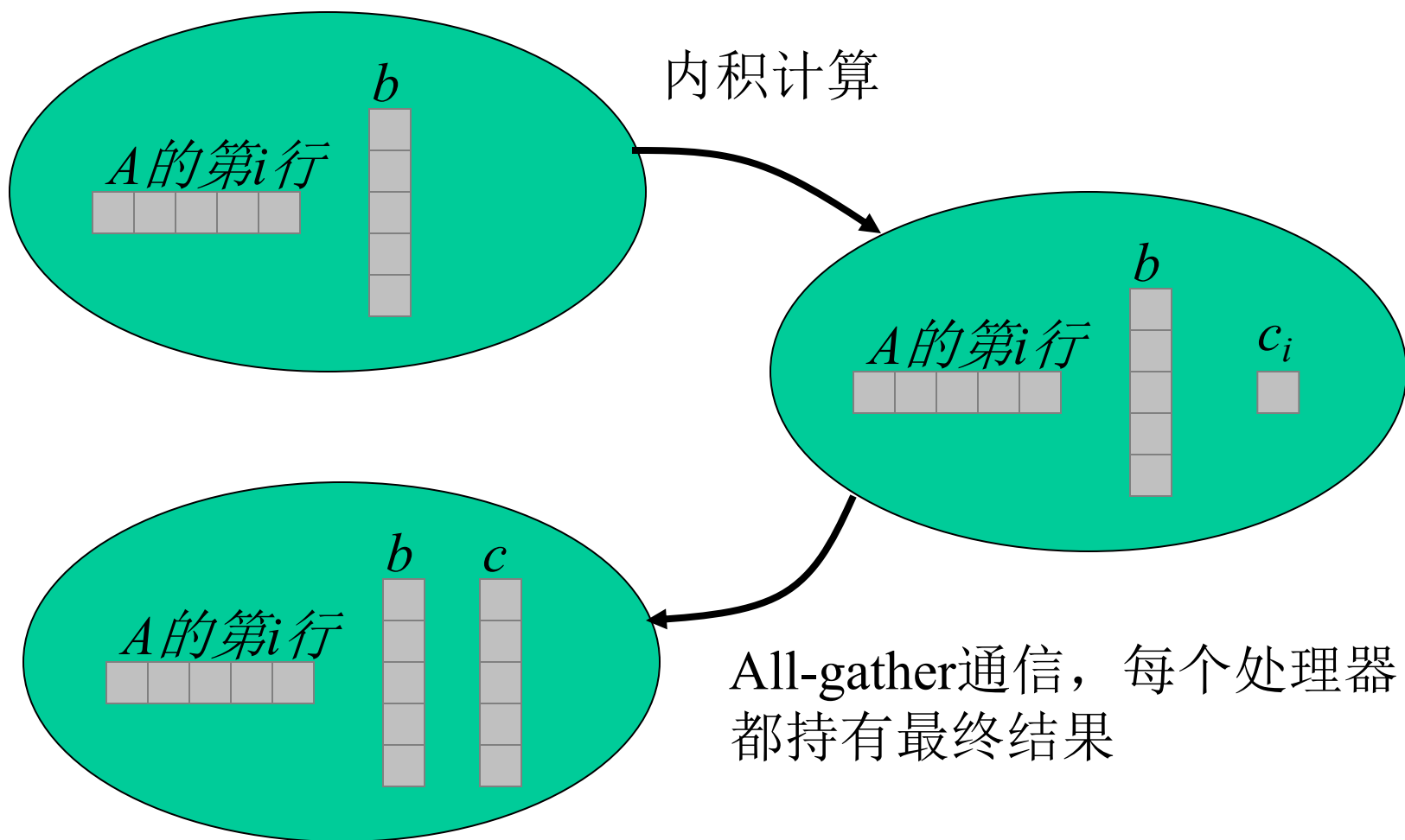
# 存储向量

- 在进程间划分向量元素
- 复制向量元素
- 向量复制是可接受的：相比矩阵的 $n^2$ 个元素，向量只有 $n$ 个元素。

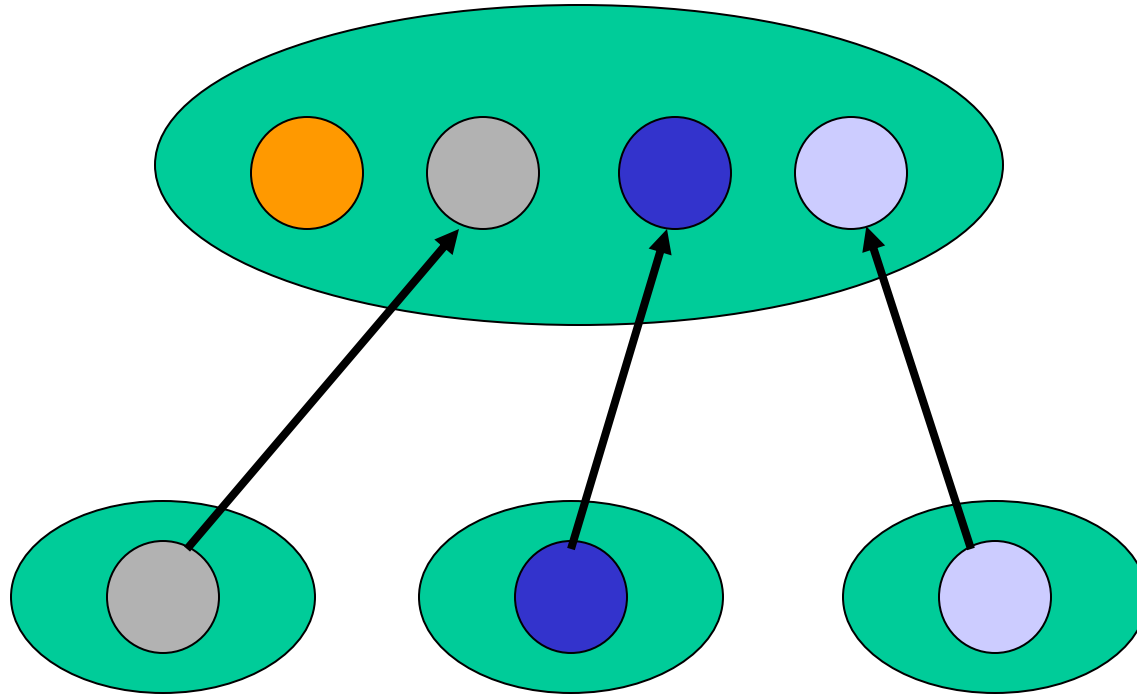
# 算法一：按行分块

- 通过域分解划分
- 每个任务存储
  - 对应矩阵的行
  - 整个向量

# 算法示意

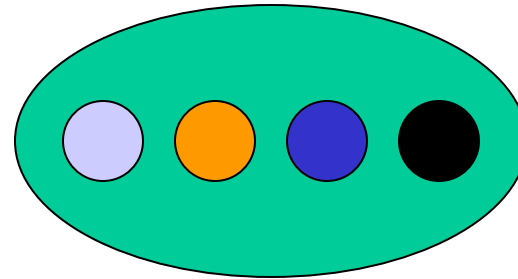
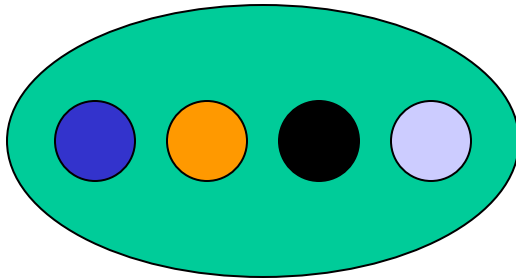
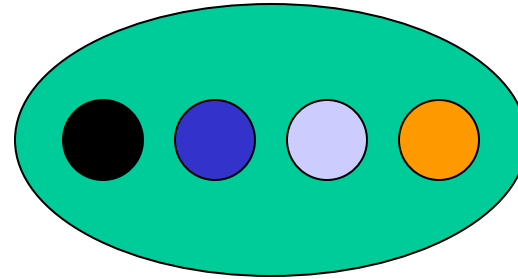
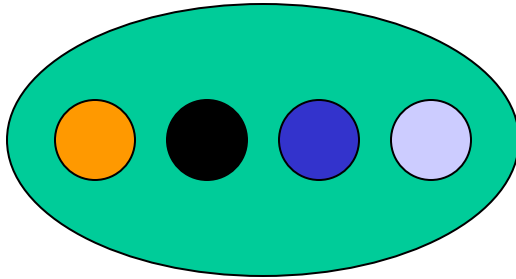


# Gather





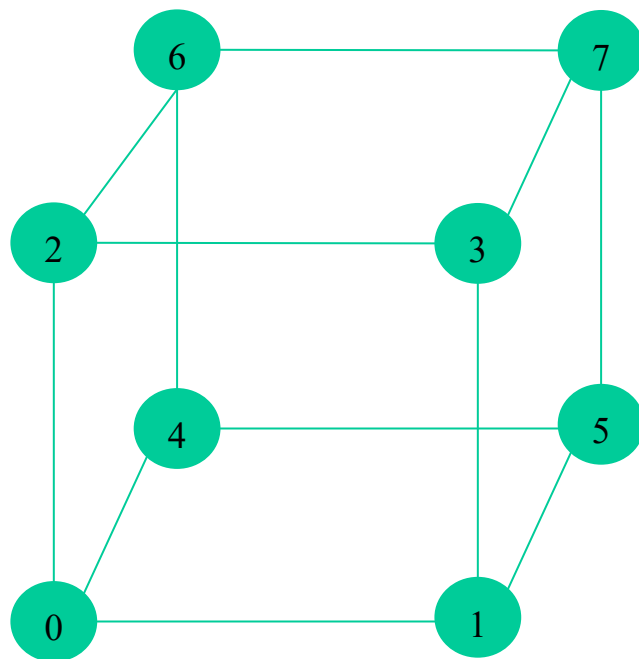
# All-gather



# 聚合和映射

- 静态的任务数量
- 有规律的通信模式 ( all-gather )
- 每个任务的计算时间是恒定的
- 策略:
  - 行聚合
  - 每个行组对应一个任务

# All gather通信时间复杂度分析



$$\sum_{i=1}^{\log p} \left( \lambda + \frac{2^{i-1} n}{\beta p} \right) = \lambda \log p + \frac{n(p-1)}{\beta p}$$

# 算法时间复杂度分析

- 串行算法复杂度:  $\Theta(n^2)$
- 并行算法的计算复杂度:  $\Theta(n^2/p)$
- all-gather的通信复杂度:  $\Theta(\log p + n)$
- 总体复杂度:  $\Theta(n^2/p + n + \log p)$

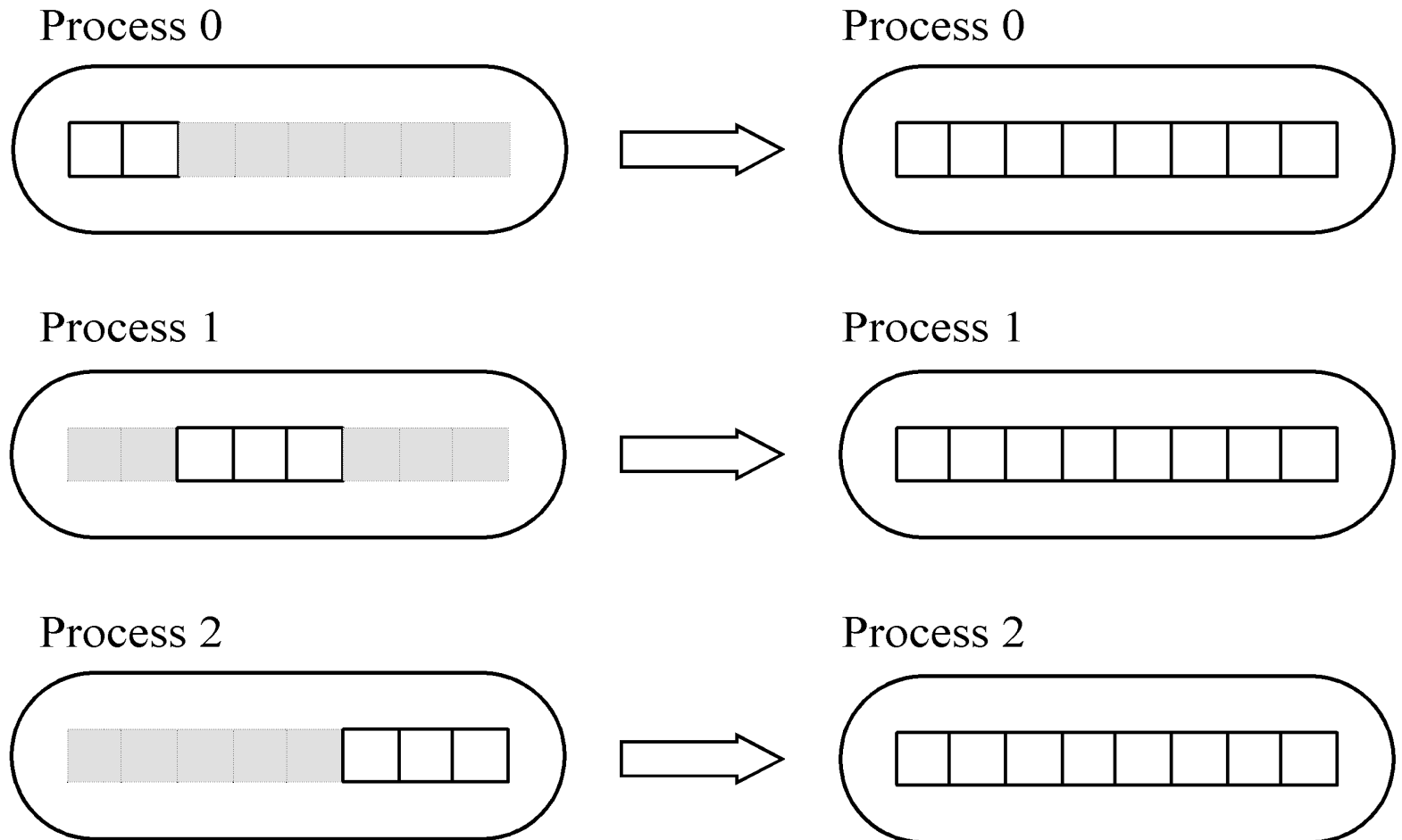
# 等效率分析

- 顺序时间复杂度:  $\Theta(n^2)$
- 唯一的并行开销是all-gather
  - 当n很大时, 消息传输时间支配着消息延迟
  - 并行通信时间:  $\Theta(n)$
- $n^2 \geq Cpn \Rightarrow n \geq Cp$  and  $M(n) = n^2$

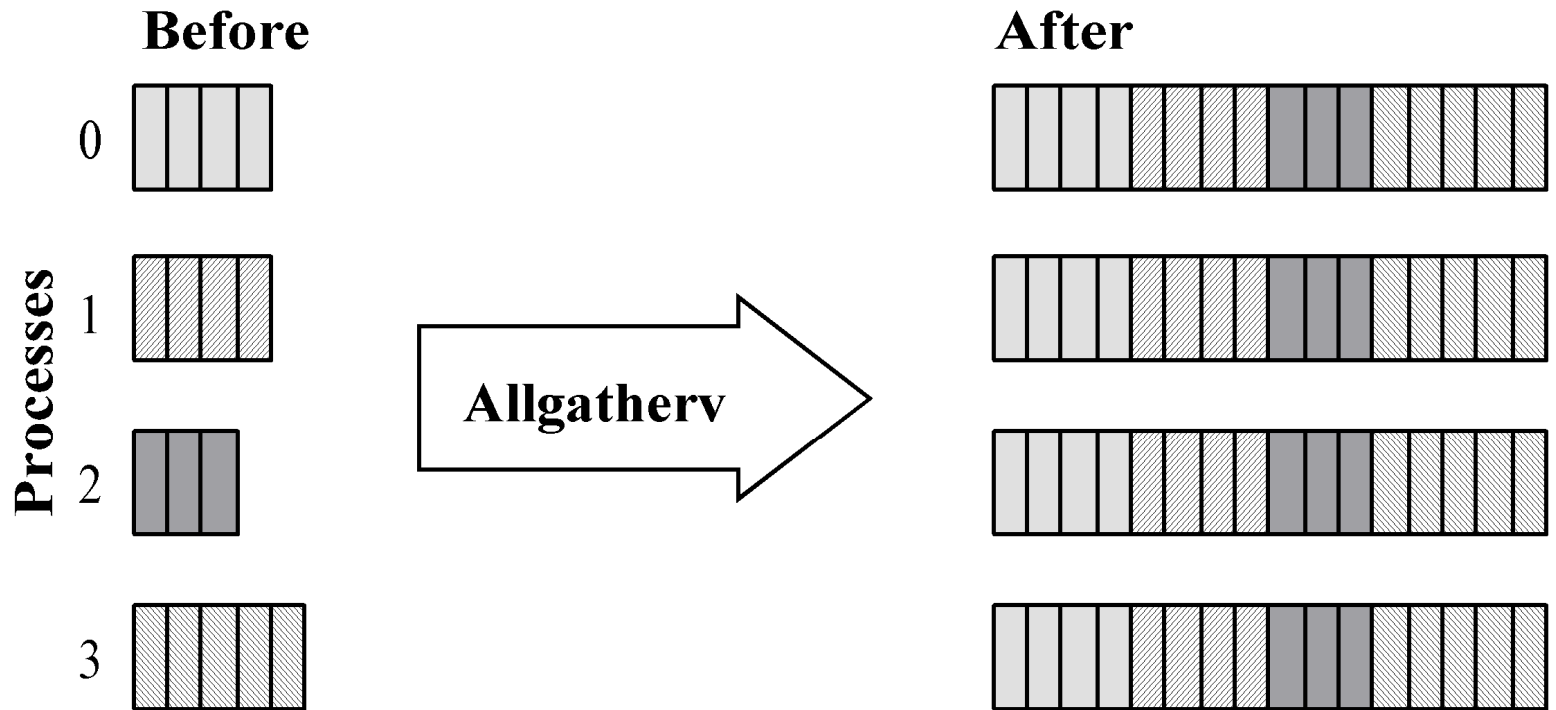
$$M(Cp) / p = C^2 p^2 / p = C^2 p$$

- 系统的可扩展性不高

# 结果向量拼接



# MPI\_Allgatherv

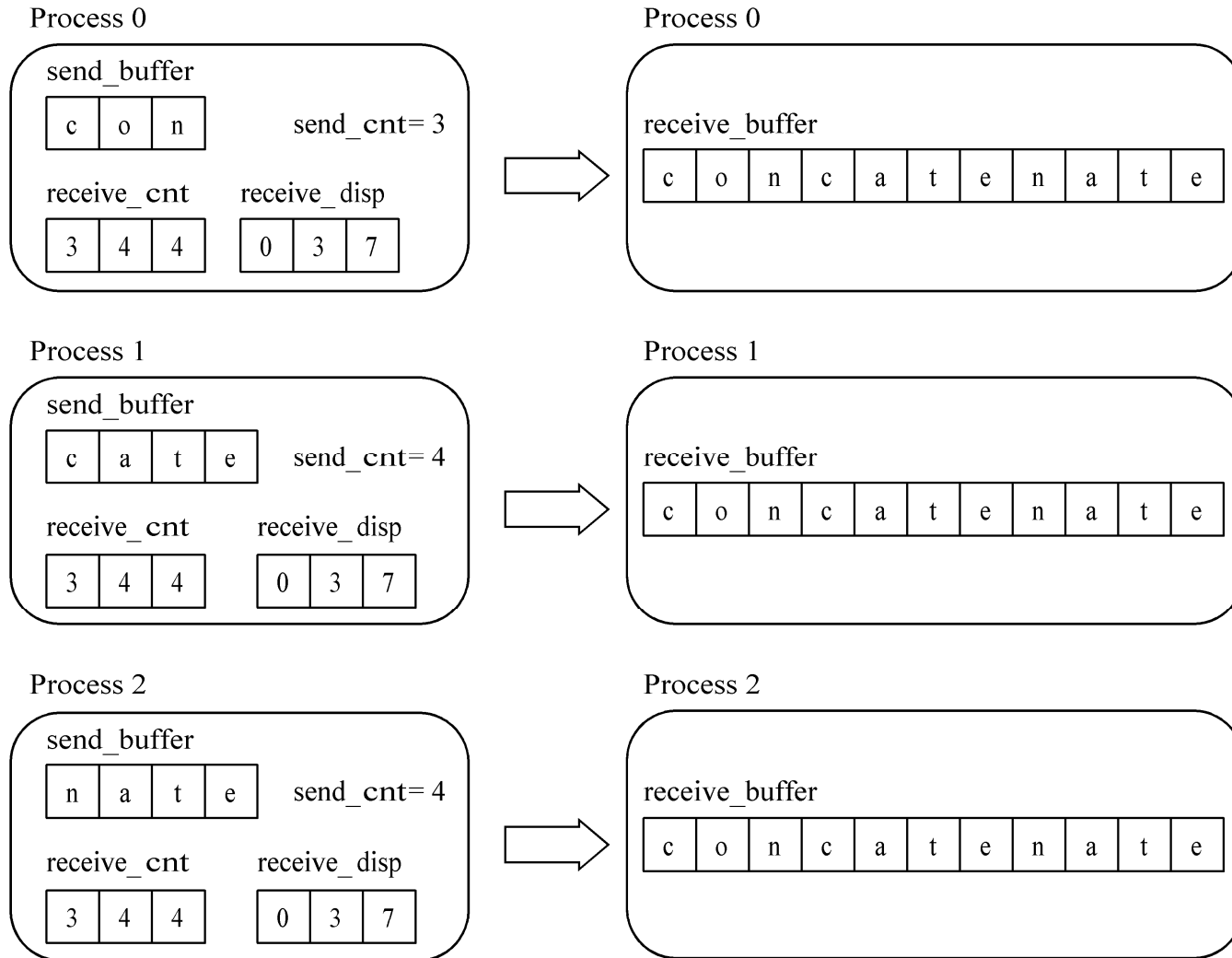


# MPI\_Allgatherv

```
int MPI_Allgatherv (  
    void                *send_buffer,  
    int                 send_cnt,  
    MPI_Datatype         send_type,  
    void                *receive_buffer,  
    int                 *receive_cnt,  
    int                 *receive_disp,  
    MPI_Datatype         receive_type,  
    MPI_Comm             communicator)
```



# MPI\_Allgatherv示意



# 示例代码

## **create\_mixed\_xfer\_arrays**

- 第一个数组 **count**
  - 每个进程贡献多少个元素
  - 使用**BLOCK\_SIZE**宏确定每个进程负责的矩阵行数
- 第二个数组 **disp**
  - 每个进程块的起始位置
  - 假设块按进程号(rank)排列

# create\_mixed\_xfer\_arrays

```
void create_mixed_xfer_arrays (  
    int id,          /* IN - Process rank */  
    int p,           /* IN - Number of processes */  
    int n,           /* IN - Total number of elements */  
    int **count,     /* OUT - Array of counts */  
    int **disp)      /* OUT - Array of displacements */  
{  
  
    int i;  
  
    *count = my_malloc (id, p * sizeof(int));  
    *disp = my_malloc (id, p * sizeof(int));  
    (*count)[0] = BLOCK_SIZE(0,p,n);  
    (*disp)[0] = 0;  
    for (i = 1; i < p; i++) {  
        (*disp)[i] = (*disp)[i-1] + (*count)[i-1];  
        (*count)[i] = BLOCK_SIZE(i,p,n);  
    }  
}
```

# 运行时间

- $\chi$ : 内积循环迭代时间
- 计算时间:  $\chi \lceil n/p \rceil$
- **All-gather**需要 $\lceil \log p \rceil$  消息, 延迟 $\lambda$
- 传送的向量元素总数:  
 $n(2^{\lceil \log p \rceil} - 1) / 2^{\lceil \log p \rceil}$
- 总执行时间:  $\chi \lceil n/p \rceil + \lambda \lceil \log p \rceil + n(2^{\lceil \log p \rceil} - 1) / (2^{\lceil \log p \rceil} \beta)$

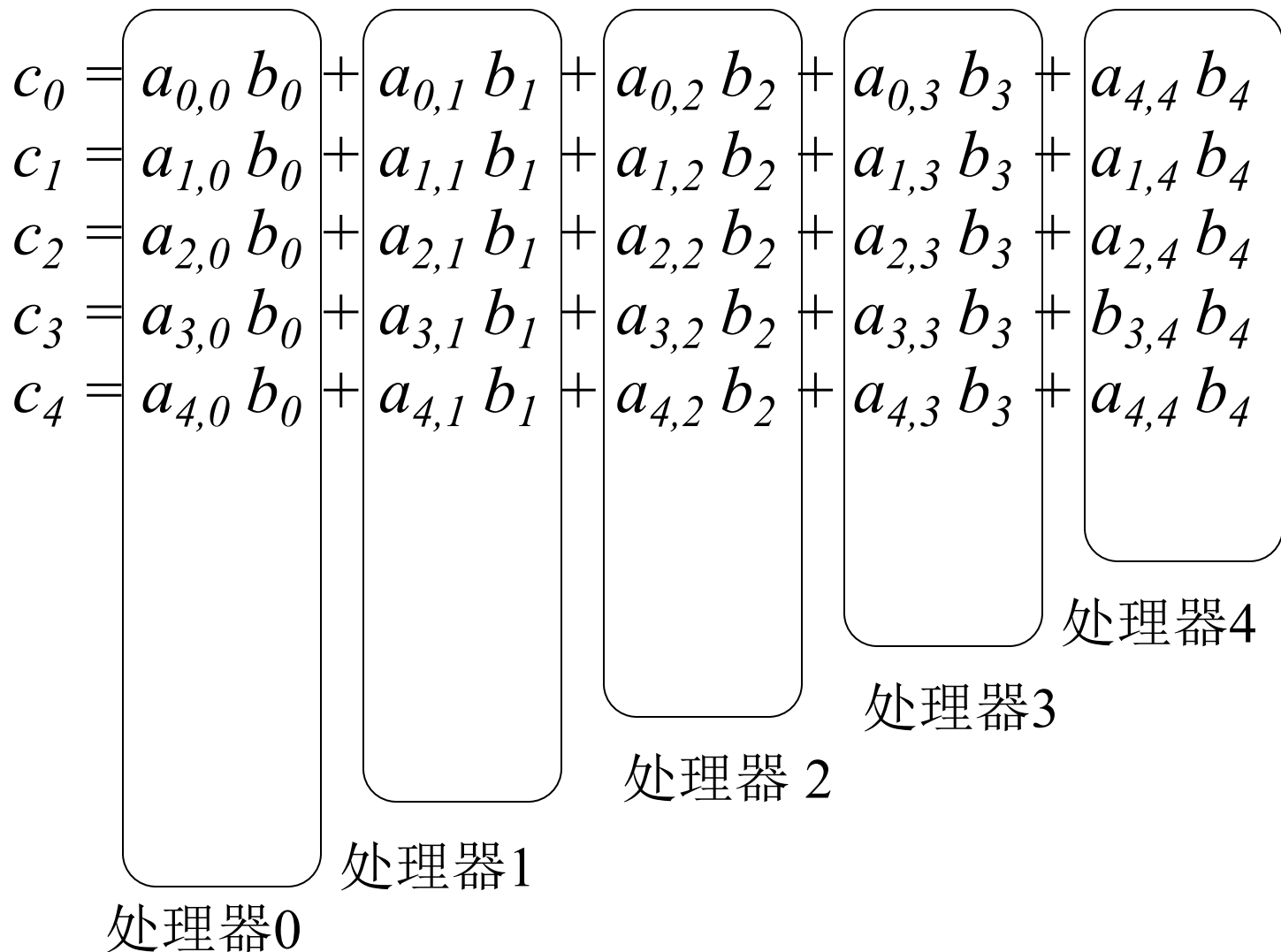
# 基准测试结果

	<i>Execution Time (msec)</i>			
<i>p</i>	<i>Predicted</i>	<i>Actual</i>	<i>Speedup</i>	<i>Mflops</i>
1	63.4	63.4	1.00	31.6
2	32.4	32.7	1.94	61.2
3	22.3	22.7	2.79	88.1
4	17.0	17.8	3.56	112.4
5	14.1	15.2	4.16	131.6
6	12.0	13.3	4.76	150.4
7	10.5	12.2	5.19	163.9
8	9.4	11.1	5.70	180.2
16	5.7	7.2	8.79	277.8

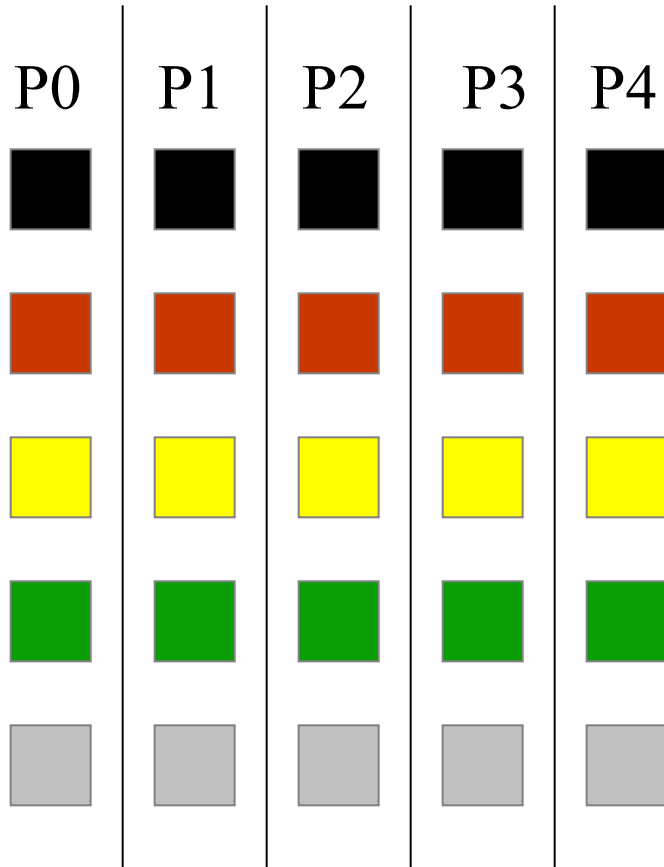
# 算法二：按列分块

- 基于数据域的分解划分
- 每个任务存储
  - 矩阵的一列
  - 向量元素

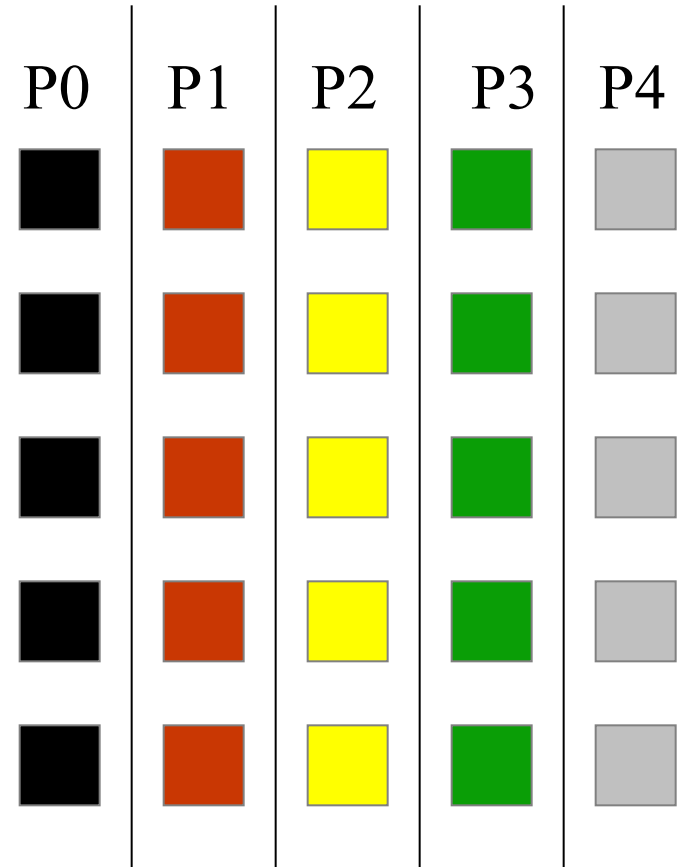
# 矩阵-向量乘法



# All-to-all Exchange



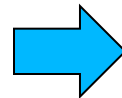
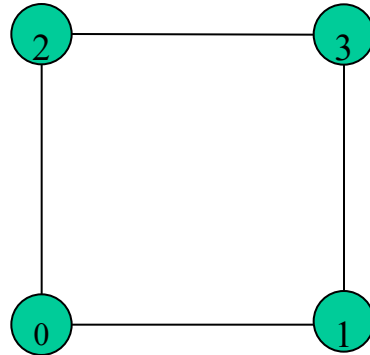
(Before)



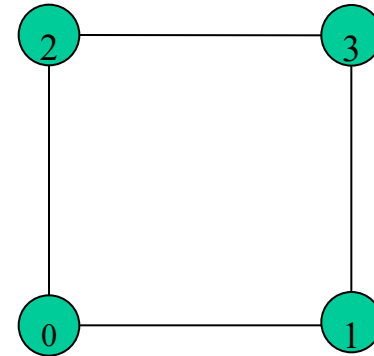
(After)



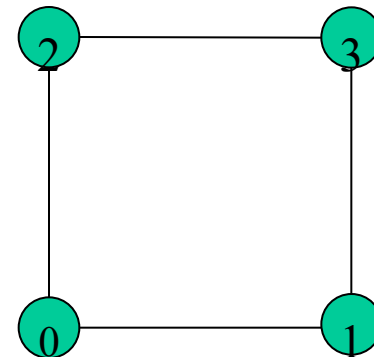
(a4 b4 c4 d4) (a3 b3 c3 d3)



(a1 b1 c1 d1) (a2 b2 c2 d2)



(d1 d2 d3 d4) (c1 c2 c3 c4)



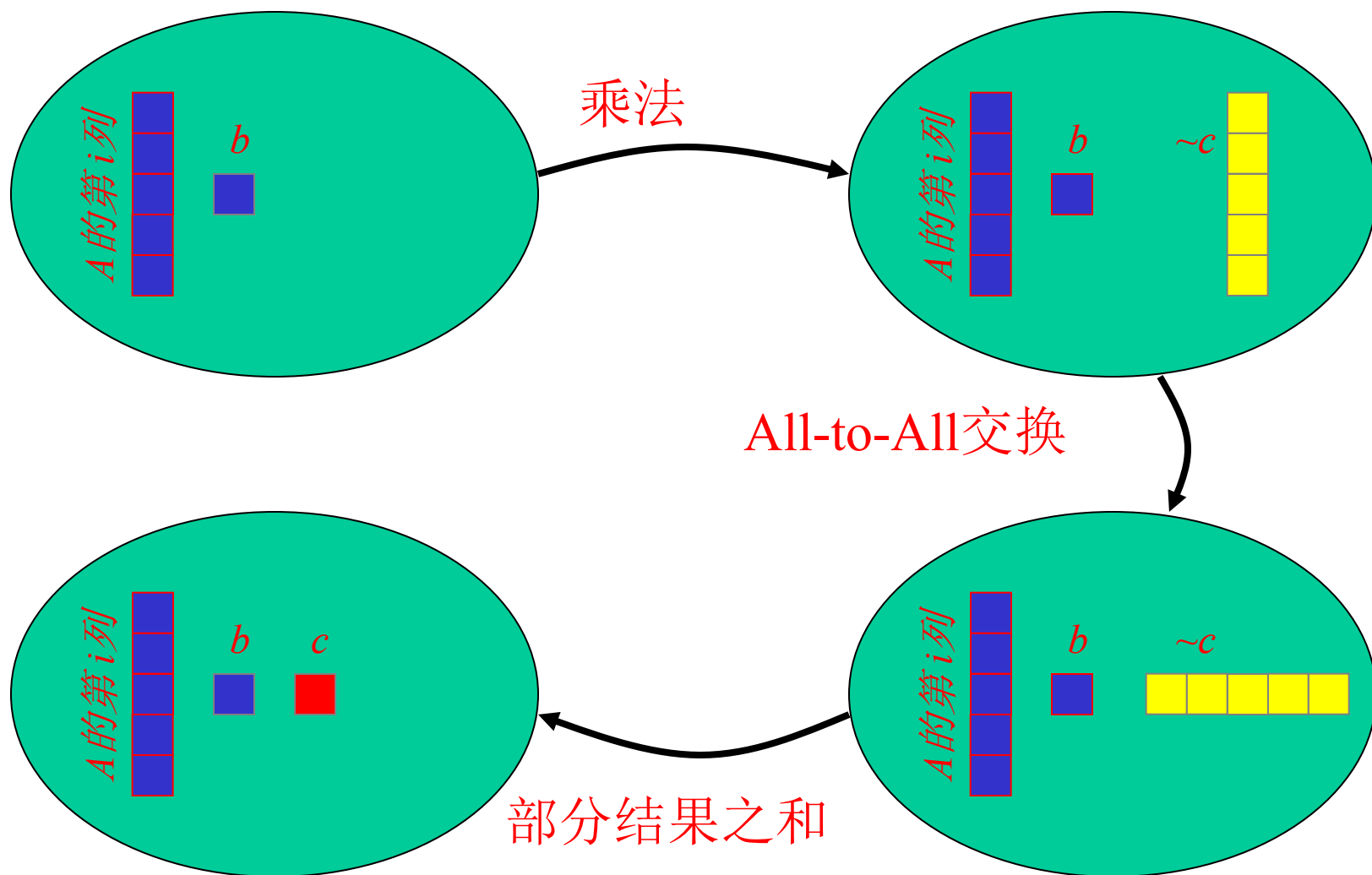
(a1 a2 a3 a4) (b1 b2 b3 b4)

D = 1  
 while (D < p)  
   Exchange & accumulate data with rank  $\otimes$  D  
   Left shift D by 1  
end while

# All-to-all Exchange通信复杂度

- 在 $\lceil \log p \rceil$ 的每一阶段，所有节点都与其他节点交换一半的数据；
- 在任何时候都只发送 $P/2$ 条信息；
- 发送和接收的元素总数为 $n \lceil \log p \rceil$ ；
- 通信复杂度： $O(n \lceil \log p \rceil)$

# 算法示意



# 聚合和映射

- 静态的任务数量
- 正则通信模式(all-to-all)
- 每个任务的计算时间是恒定的
- 策略:
  - 聚合列的群组
  - 每个MPI进程创建一个任务

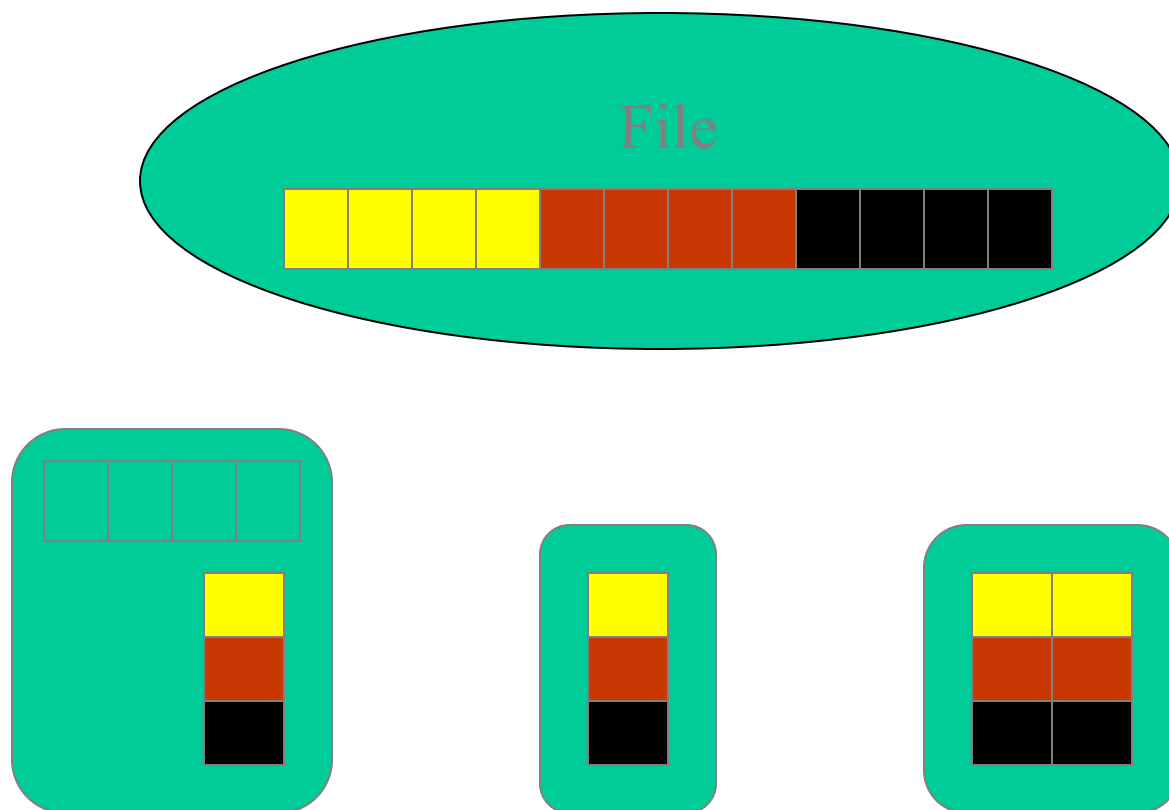
# 复杂度分析

- 串行算法复杂度:  $\Theta(n^2)$
- 并行算法的计算复杂度:  $\Theta(n^2/p)$
- 全对全的通信复杂度:  $\Theta(n \log p)$
- 总体复杂度:  $\Theta(n^2/p + n \log p)$

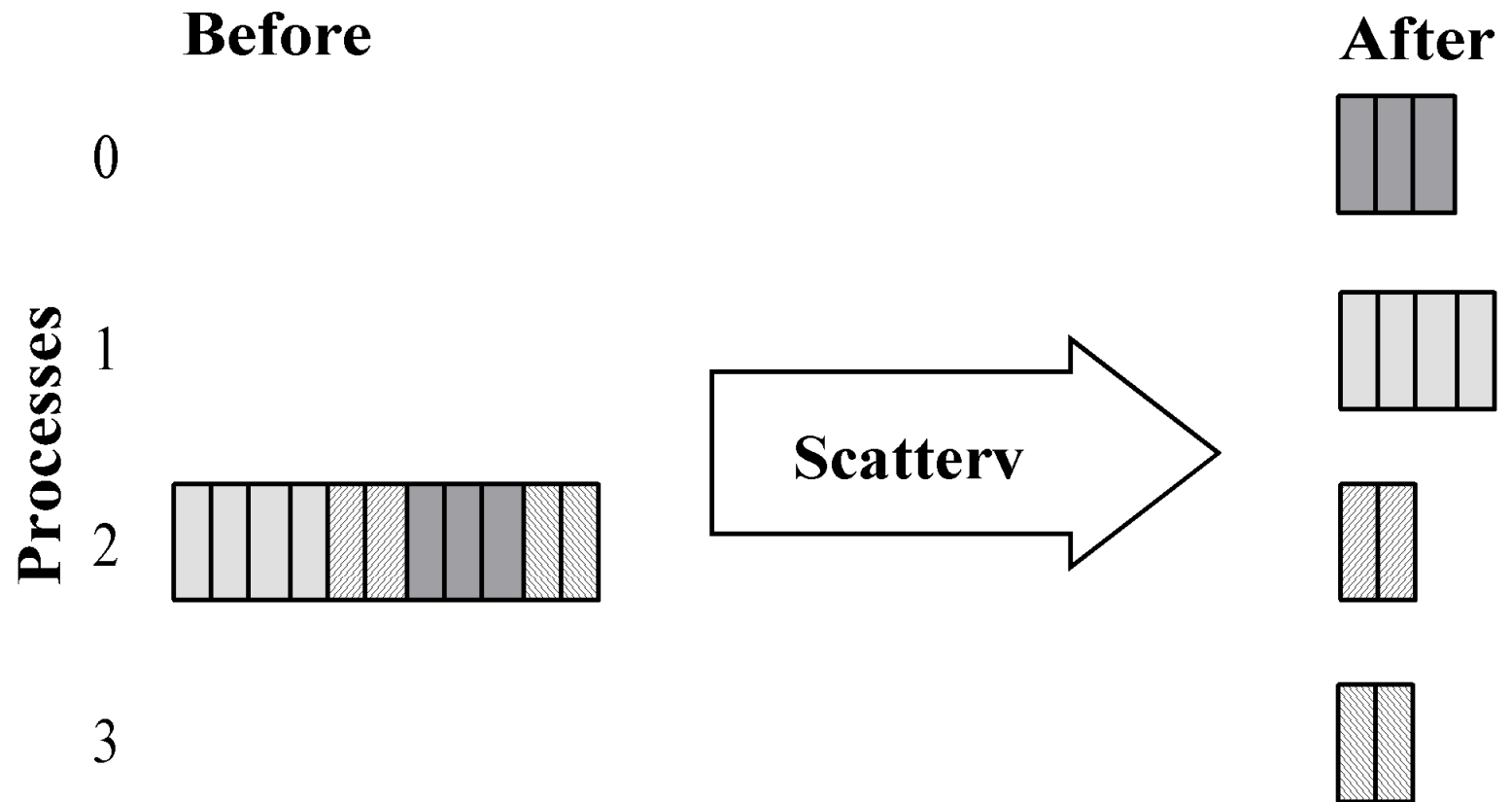
# 等效性分析

- 串行时间复杂度:  $\Theta(n^2)$
- 只有all-to-all消息的并行开销
  - 当n很大时, 消息传输时间支配着消息延迟
  - 并行通信时间:  $\Theta(n \log(p))$
- $n^2 \geq Cpn \log(p) \Rightarrow n \geq Cp \log(p)$
- 可扩展性函数高于rowwise算法:  $C^2p$

# 数据分发-按列存储



# MPI\_Scatterv





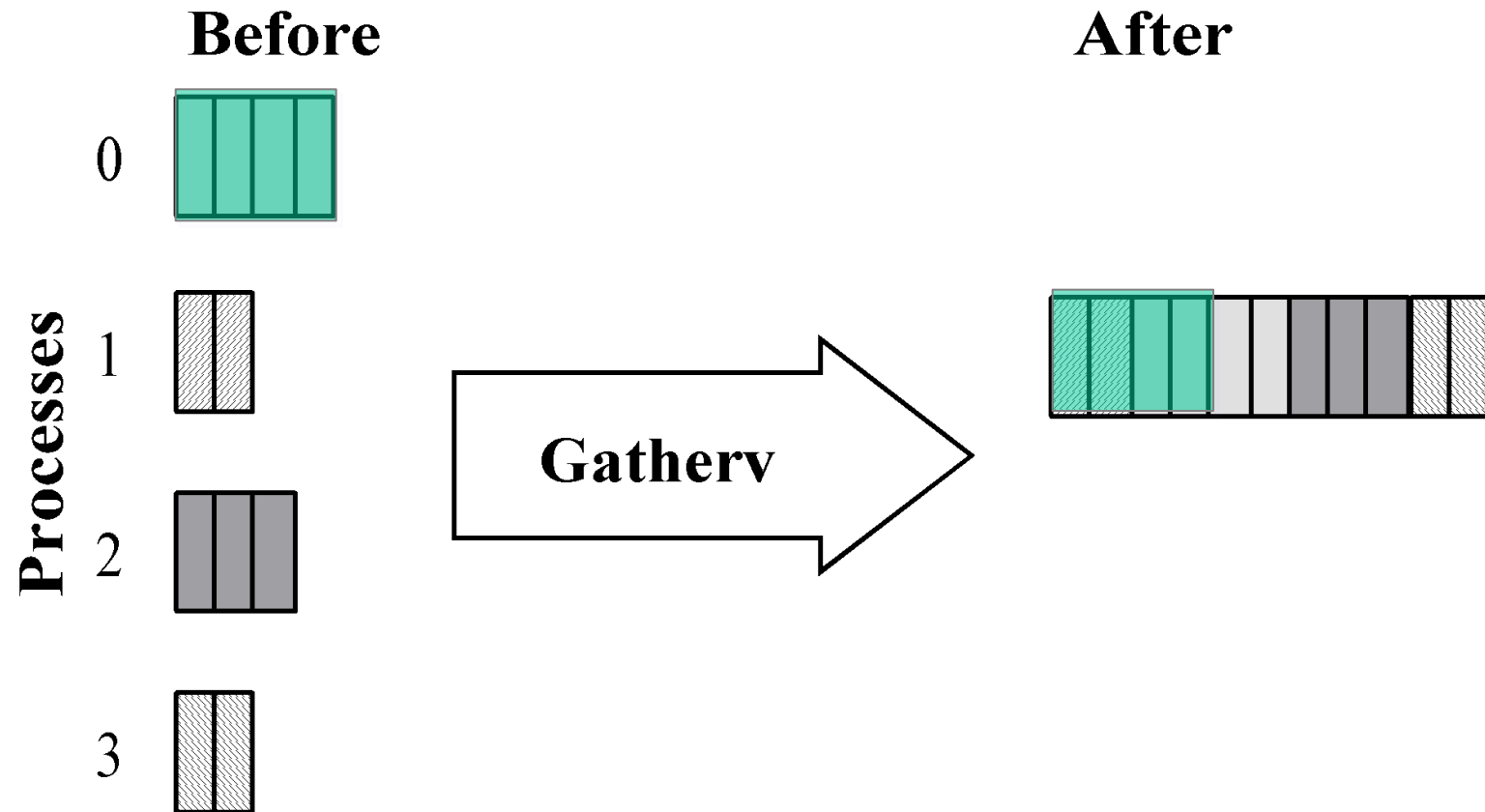
# MPI\_Scatterv接口

```
int MPI_Scatterv (  
    void                *send_buffer,  
    int                 *send_cnt,  
    int                 *send_disp,  
    MPI_Datatype        send_type,  
    void                *receive_buffer,  
    int                 receive_cnt,  
    MPI_Datatype        receive_type,  
    int                 root,  
    MPI_Comm            communicator)
```

# 打印分布于各个处理器的按列分布矩阵

- 数据运动与我们读取矩阵时相反
- 用 “聚集” 取代 “分散”
- 使用 “v” 变体，因为不同的进程负责了不同数量的列

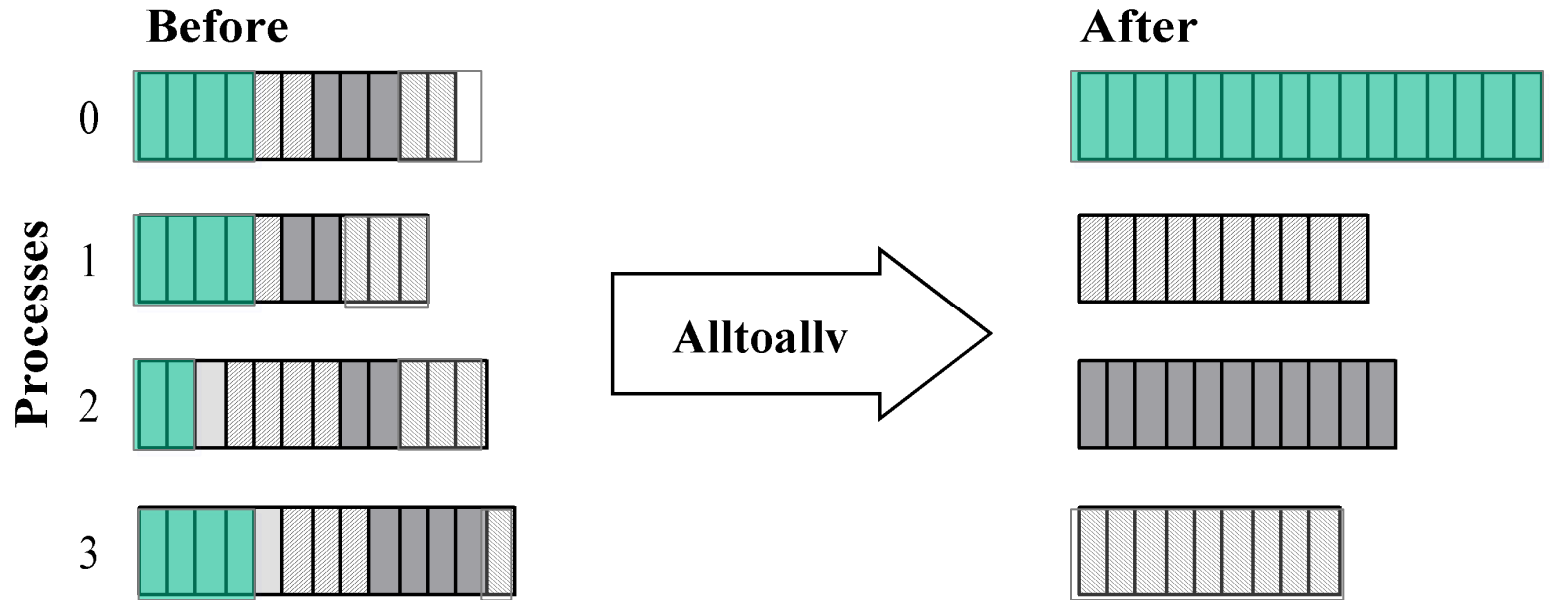
# 函数 MPI\_Gatherv



# MPI\_Gatherv的接口

```
int MPI_Gatherv (  
    void *send_buffer,  
    int send_cnt,  
    MPI_Datatype send_type,  
    void *receive_buffer,  
    int *receive_cnt,  
    int *receive_disp,  
    MPI_Datatype receive_type,  
    int root,  
    MPI_Comm communicator)
```

# 函数 MPI\_Alltoallv



# MPI\_Alltoallv的接口

```
int MPI_Alltoallv (  
    void                *send_buffer,  
    int                 *send_cnt,  
    int                 *send_disp,  
    MPI_Datatype        send_type,  
    void                *receive_buffer,  
    int                 *receive_cnt,  
    int                 *receive_disp,  
    MPI_Datatype        receive_type,  
    MPI_Comm            communicator)
```

# 运行时间

- $\chi$ : 内积循环迭代时间
- 计算时间:  $\chi \lceil n/p \rceil$
- All-gather 需要  $p-1$  消息, 每个消息长度为  $n/p$
- 每个元素为8字节
- 总执行时间:  
 $\chi \lceil n/p \rceil + (p-1)(\lambda + (8n/p)/\beta)$

# Benchmarking 结果

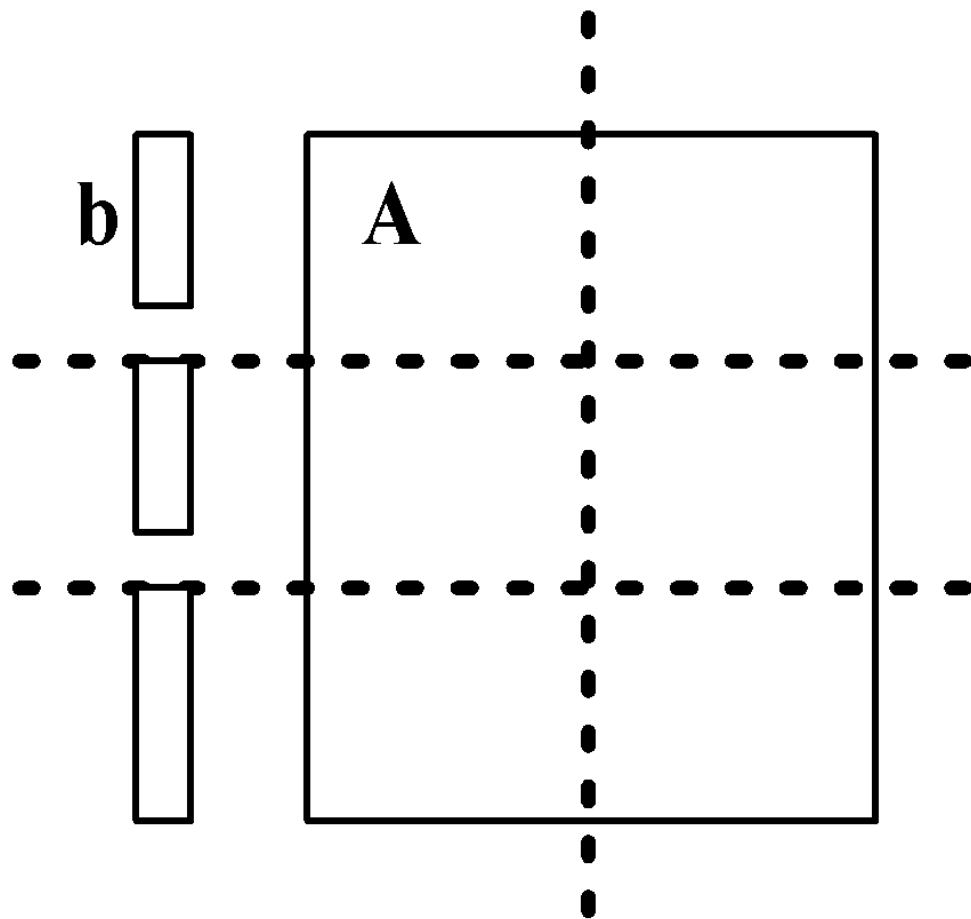
	<i>Execution Time (msec)</i>			
<i>p</i>	<i>Predicted</i>	<i>Actual</i>	<i>Speedup</i>	<i>Mflops</i>
1	63.4	63.8	1.00	31.4
2	32.4	32.9	1.92	60.8
3	22.2	22.6	2.80	88.5
4	17.2	17.5	3.62	114.3
5	14.3	14.5	4.37	137.9
6	12.5	12.6	5.02	158.7
7	11.3	11.2	5.65	178.6
8	10.4	10.0	6.33	200.0
16	8.5	7.6	8.33	263.2



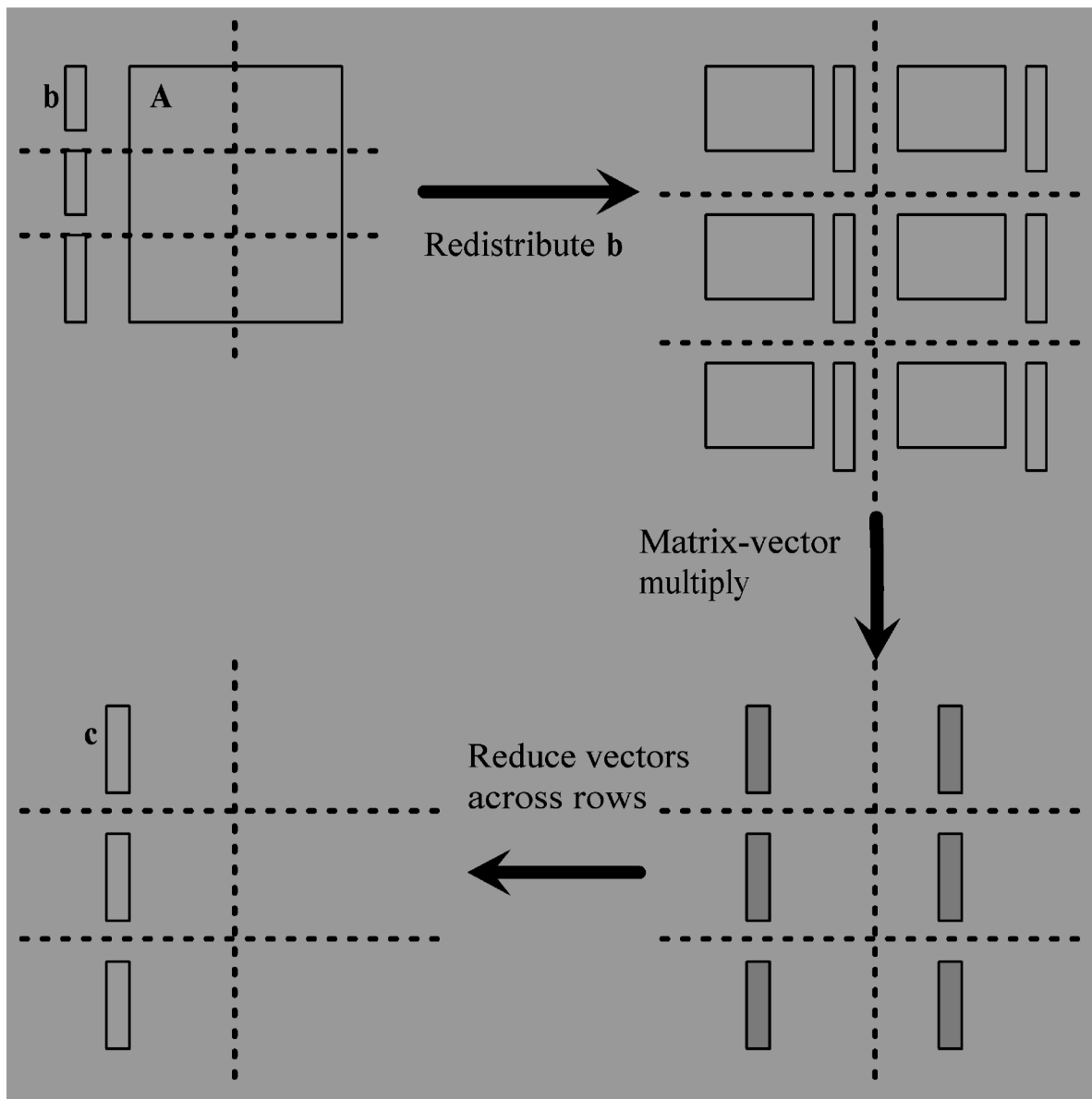
# 算法三：棋盘块分解

- 将矩阵  $A$  的每个元素与一个原始任务关联起来，
- 每个原始任务执行一次乘法运算。
- 将原始任务聚合成矩形块，
- 进程形成 2-D 网格，
- 向量  $b$  在网格的第一列进程中按块分布

# 聚合后的任务



# 算法的各个阶段

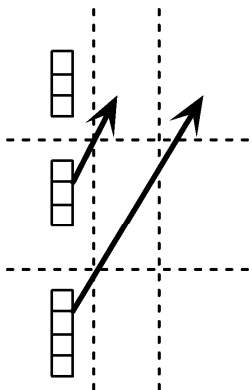


# 重分配向量**b**

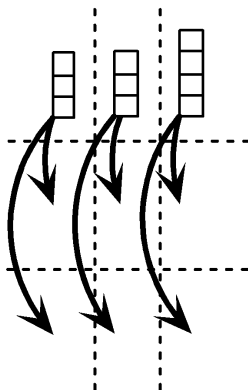
- 第一步：将 **b** 从第一行进程移动到第一列进程
  - 如果  $p$  是平方数
    - 第一列/第一行进程发送/接收部分 **b**
  - 如果  $p$  不是平方数
    - 在进程 0,0 上 gather **b**
    - 进程 0,0 将其 scatter 到第一行进程
- 第二步：第一行进程在列内广播 **b**

# 重分配向量**b**

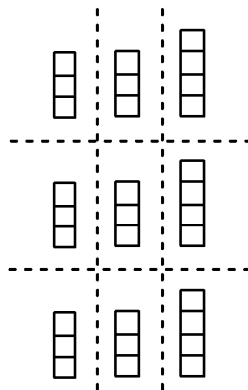
Send/Recv  
blocks of **b**



Broadcast  
blocks of **b**



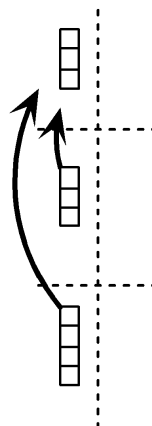
When  $p$  is a square number



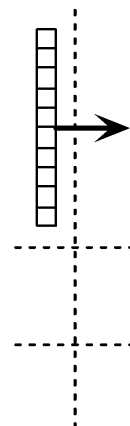
(a)

When  $p$  is not a square number

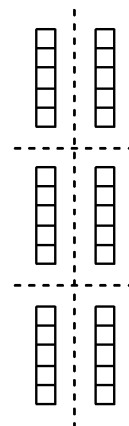
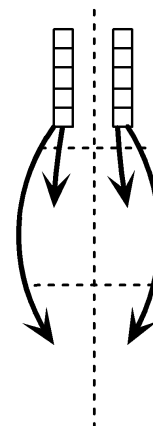
Gather **b**



Scatter **b**



Broadcast  
blocks of **b**



(b)

# 复杂度分析

- 假设 $p$ 是平方数
  - 如果网格是  $1 \times p$ ，则演变为按列块条带。
  - 如果网格是  $p \times 1$ ，则演变为按行块条带。

# 复杂度分析

- 每个进程执行的计算:  $\Theta(n^2/p)$
- 重分配 **b**:  $\Theta(n / \sqrt{p} + \log p(n / \sqrt{p})) = \Theta(n \log p / \sqrt{p})$
- 规约:  
 $\Theta(n \log p / \sqrt{p})$
- 总体并行复杂度:  
 $\Theta(n^2/p + n \log p / \sqrt{p})$

# 等效性分析

- 串行复杂度:  $\Theta(n^2)$
- 并行通信复杂度:  
 $\Theta(n \log p / \sqrt{p})$
- 等效性函数:  
 $n^2 \geq C n \sqrt{p} \log p \Rightarrow n \geq C \sqrt{p} \log p$

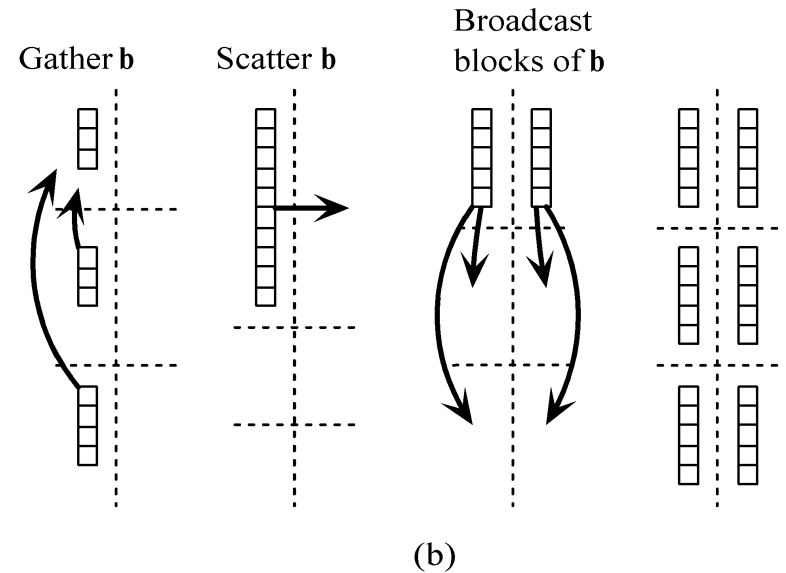
$$M(C \sqrt{p} \log p) / p = C^2 p \log^2 p / p = C^2 \log^2 p$$

- 这个系统比前两个实现更具可扩展性。



# 创建通信域

- 进程子集的四中集合通信：
  - 当  $p$  不是平方数时，虚拟进程网格的第一列进程参与聚集向量  $b$  的通信。
  - 当  $p$  不是平方数时，虚拟进程网格的第一行进程参与切分向量  $b$  的通信。
  - 每个第一行进程将其  $b$  块广播给虚拟进程网格中同一列的其他进程
  - 网格中的每一行进程执行独立的求和规约，生成在第一列进程中的向量  $c$ 。



# 创建通信域

- 在虚拟的 2-D 网格中使用进程
- 创建一个自定义通信域来实现
- 集合通信涉及通信域中的所有进程
- 需要在进程子集之间进行广播、规约
- 为同一行或同一列中的进程创建通信域

# 通信域包含什么？

- 进程组
- 上下文
- 属性
  - 拓扑结构（让我们以另一种方式寻址进程）
  - 其他我们不考虑的内容。

# 通信域的拓扑结构

- 拓扑结构可以将寻址模式与进程关联起来
- 拓扑结构在本质上是虚拟的，因为它们不受处理器的实际组织方式限制
- 拓扑结构的两种类型
  - 笛卡尔（网格）
  - 图

# 创建虚拟2-D网格进程

- MPI\_Dims\_create
  - 输入参数
    - 网格中的总进程数
    - 网格维数
  - 返回每个维度的进程数量
- MPI\_Cart\_create
  - 创建具有笛卡尔拓扑结构的通信域

# MPI\_Dims\_create

```
int MPI_Dims_create (  
    int nodes,  
        /* Input - Procs in grid */  
  
    int dims,  
        /* Input - Number of dims */  
  
    int *size)  
    /* Input/Output - Size of  
        each grid dimension */
```

# MPI\_Cart\_create

```
int MPI_Cart_create (  
    MPI_Comm old_comm, /* Input - old communicator */  
  
    int dims, /* Input - grid dimensions */  
  
    int *size, /* Input - # procs in each dim */  
  
    int *periodic,  
        /* Input - periodic[j] is 1 if dimension j  
           wraps around; 0 otherwise */  
  
    int reorder,  
        /* 1 if process ranks can be reordered */  
  
    MPI_Comm *cart_comm)  
    /* Output - new communicator */
```

# Using MPI\_Dims\_create and MPI\_Cart\_create

```
MPI_Comm cart_comm;
int p;
int periodic[2];
int size[2];
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, size,
                1, &cart_comm);
```



# 与网格相关的实用函数

- MPI\_Cart\_rank
  - 给定笛卡尔通信域中进程的坐标，返回进程序号
- MPI\_Cart\_coords
  - 给定笛卡尔通信域中进程的排名，返回进程坐标

# Header for MPI\_Cart\_rank

```
int MPI_Cart_rank (  
    MPI_Comm comm,  
        /* In - Communicator */  
    int *coords,  
        /* In - Array containing process'  
            grid location */  
    int *rank)  
    /* Out - Rank of process at  
        specified coords */
```

# Header for MPI\_Cart\_coords

```
int MPI_Cart_coords (  
    MPI_Comm comm,  
        /* In - Communicator */  
    int rank,  
        /* In - Rank of process */  
    int dims,  
        /* In - Dimensions in virtual grid */  
    int *coords)  
    /* Out - Coordinates of specified  
        process in virtual grid */
```

# read\_checkerboard\_matrix

```
void read_checkerboard_matrix (char *s, void ***subs, void  
**storage, MPI_Datatype dtype, int *m, int *n, MPI_Comm  
grid_comm) {
```

```
.....
```

```
int dest_id; /* Rank of receiving proc */
```

```
int dest_coord[2]; /* Process coords */
```

```
int grid_id; /* Process rank */
```

```
for (i=0; i<m; i++){
```

```
    dest_coord[0]=BLOCK_OWNER(i,r,m);
```

```
    dest_coord[1]=1;
```

```
    MPI_Cart_rank(grid_comm, dest_coord, &dest_id);
```

```
    if (grid_id==0){
```

```
        /* read matrix row 'i'*/
```

```
        .....
```

```
        /* send matrix row I to process dest_id */
```

```
    }else if (grid_id==dest_id){
```

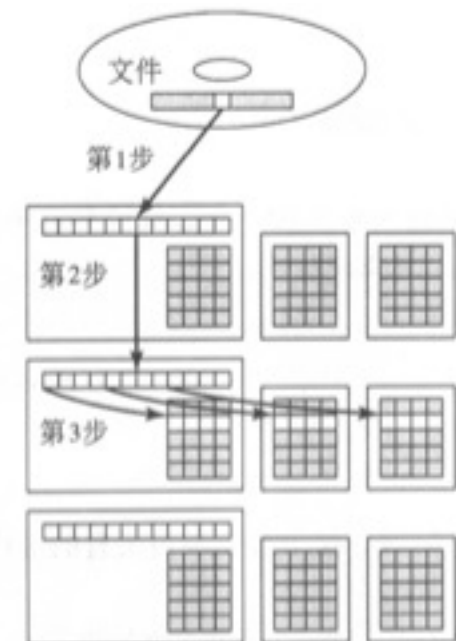
```
        /*receive matrix row I form process 0*/
```

```
        .....
```

```
    }
```

```
}
```

```
}
```



# MPI\_Comm\_split

- 将通信域中的进程划分为一个或多个子组
- 为每个子组构建一个通信域
- 允许每个子组中的进程执行自己的集合通信
- 对于按列 scatter 和按行 reduce 是必需的

# Header for MPI\_Comm\_split

```
int MPI_Comm_split (  
    MPI_Comm old_comm,  
        /* In - Existing communicator */  
  
    int partition, /* In - Partition number */  
  
    int new_rank,  
        /* In - Ranking order of processes  
           in new communicator */  
  
    MPI_Comm *new_comm)  
    /* Out - New communicator shared by  
       processes in same partition */
```

# Example: Create Communicators for Process Rows

```
MPI_Comm grid_comm; /* 2-D process grid */

MPI_Comm grid_coords[2];
                /* Location of process in grid */

MPI_Comm row_comm;
                /* Processes in same row */

MPI_Comm_split (grid_comm, grid_coords[0],
                grid_coords[1], &row_comm);
```

# 运行时间

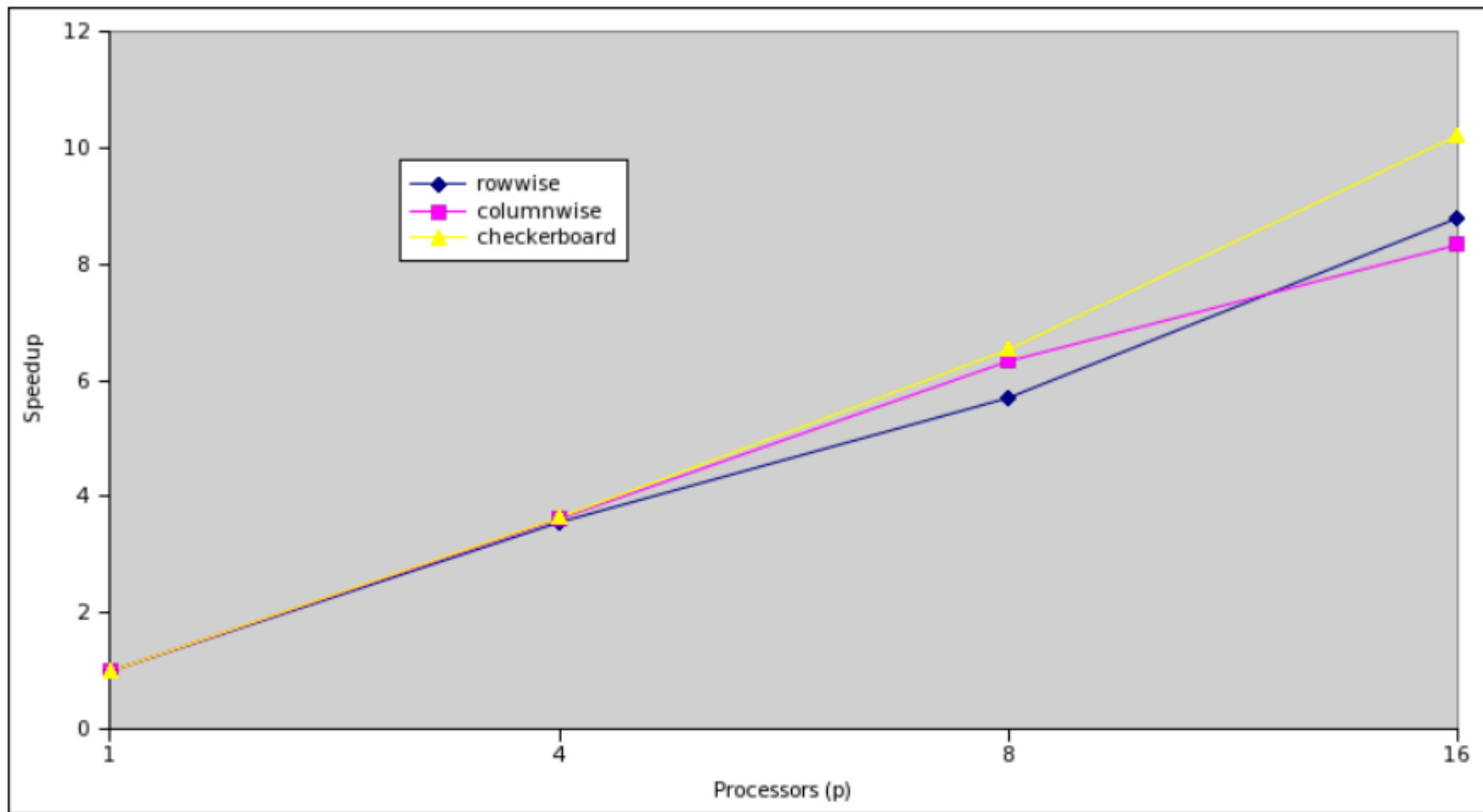
- 计算时间:  $\chi \lceil n/\sqrt{p} \rceil \lceil n/\sqrt{p} \rceil$
- 假定  $p$  是一个平方数
- 重分配 **b**
  - 发送/接收:  $\lambda + 8 \lceil n/\sqrt{p} \rceil / \beta$
  - 广播:  $\log \sqrt{p} (\lambda + 8 \lceil n/\sqrt{p} \rceil / \beta)$
- 规约:  
 $\log \sqrt{p} (\lambda + 8 \lceil n/\sqrt{p} \rceil / \beta)$



# Benchmarking

Procs	Predicted (msec)	Actual (msec)	Speedup	Megaflops
1	63.4	63.4	1.00	31.6
4	17.8	17.4	3.64	114.9
9	9.7	9.7	6.53	206.2
16	6.2	6.2	10.21	322.6

# 三种算法对比



# 总结 (1/3)

- 矩阵分解  $\Rightarrow$  通信
  - 按行块: all-gather
  - 按列块: all-to-all exchange
  - 棋盘块: gather, scatter, broadcast, reduce
- 三种算法: 消息数量大致相同
- 每个进程传输的元素数量不同
  - 前两种算法:  $\Theta(n)$  元素 / 进程
  - 棋盘算法:  $\Theta(n/\sqrt{p})$  元素/进程
- 棋盘块算法具有更好的可扩展性

# 总结 (2/3)

- 使用笛卡尔拓扑结构的通信域
  - 创建
  - 通过进程序号或坐标识别进程
- 对通信域进行细分
  - 允许在进程子集之间进行集合操作

## 总结 (3/3)

- 并行程序及其支持函数比C语言对应部分要复杂得多
- 需要编写额外的代码来读取、分发和打印矩阵和向量
- 开发和调试这些函数很繁琐，也很困难
- 因此将这些函数进行泛化并放入库中以供重用是合理的。