



中科院计算所  
INSTITUTE OF COMPUTING TECHNOLOGY, CAS

# 高级MPI编程技术

谭光明

[tgm@ncic.ac.cn](mailto:tgm@ncic.ac.cn)

中国科学院计算技术研究所  
国家智能计算机研究开发中心  
计算机体系结构国家重点实验室

# 最基本的MPI

MPI调用借口的总数虽然庞大，但根据实际编写MPI的经验，常用的MPI调用的个数确什么有限。下面是6个最基本的MPI函数。

1. MPI\_Init(...);
2. MPI\_Comm\_size(...);
3. MPI\_Comm\_rank(...);
4. MPI\_Send(...);
5. MPI\_Recv(...);
6. MPI\_Finalize();

MPI\_Init(...);  
...  
并行代码;  
...  
MPI\_Finalize();  
只能有串行代码;



# 标准阻塞发送

## MPI\_Send()

```
int MPI_Send( void *buff, int count,  
              MPI_Datatype datatype, int dest,  
              int tag, MPI_Comm comm)
```

- tag 的取值范围为0 - MPI\_TAG\_UB.
- dest 的取值范围为0 - np-1 (np 为通信器comm中的进程数) 或MPI\_PROC\_NULL.
- count 是指定数据类型的个数, 而不是字节数.



# 标准阻塞接收MPI\_Recv

()

```
int MPI_Recv( void *buff, int count,  
             MPI_Datatype datatype, int source,  
             int tag, MPI_Comm comm,  
             MPI_Status *status)
```

- count 给出接收缓冲区的大小(指定数据类型的个数),它是接收数据长度的上界. 具体接收到的数据长度可通过调用**MPI\_Get\_count** 函数得到



# 非阻塞发送 MPI\_Isend()

- ```
int MPI_Isend (  
    void*          message /* in */,  
    int            count   /* in */,  
    MPI_Datatype    datatype /* in */,  
    int            dest    /* in */,  
    int            tag     /* in */,  
    MPI_Comm        comm   /* in */,  
    MPI_Request*    request /* out */)
```

The “I” stands for “Immediate”





# 非阻塞接收 MPI\_Irecv()

- ```
int MPI_Irecv (  
    void*          message /*out*/,  
    int            count   /* in */,  
    MPI_Datatype    datatype /* in */,  
    int            source  /* in */,  
    int            tag     /* in */,  
    MPI_Comm        comm   /* in */,  
    MPI_Request*    request /*out*/)
```




# MPI\_Sendrecv()

- int MPI\_Sendrecv (

send	void*	send_buf	/* in */,
	int	send_count	/* in */,
	MPI_Datatype	send_type	/* in */,
	int	dest	/* in */,
	int	send_tag	/* in */,
recv	void*	recv_buf	/* out */,
	int	recv_count	/* in */,
	MPI_Datatype	recv_type	/* in */,
	int	source	/* in */,
	int	recv_tag	/* in */,
	MPI_Comm	comm	/* in */,
	MPI_Status*	status	/* out */)



# MPI内容

- 基本概念
- 点到点通信
- 自定义数据类型 
- 集合通信





# MPI数据类型分类

- 预定义数据类型
- 自定义数据类型



# MPI预定义数据类型

MPI预定义数据类型	相应的C数据类型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	无对应类型
MPI_PACKED	无对应类型



# 为什么自定义数据类型

- MPI 的消息收发函数只能处理连续存储的同一类型的数据.
- 不同系统有不同的数据表示格式。MPI预先定义一些基本数据类型，在实现过程中在这些基本数据类型为桥梁进行转换。
- 派生数据类型:允许消息来自不连续或类型不一致的存储区域，如数组散元与结构类型等的传送.
- 它的使用可有效地减少消息传递的次数, 增大通信粒度, 并且在收/发消息时避免或减少数据在内存中的拷贝、复制.



# 数据类型的定义

- MPI 数据类型由两个 $n$  元序列构成,  $n$  为正整数.
  - 第一个序列包含一组数据类型, 称为类型序列 (type signature):

$\text{Typesig} = \{\text{type}_0, \text{type}_1, \dots, \text{type}_{n-1}\}.$

- 第二个序列包含一组整数位移, 称为位移序列 (type displacements):

$\text{Typedisp} = \{\text{disp}_0, \text{disp}_1, \dots, \text{disp}_{n-1}\}.$

位移序列中位移总是以字节为单位计算的.



# 数据类型的定义(续)

- 构成类型序列的数据类型称为基本数据类型, 它们可以是原始数据类型, 也可以是任何已定义的数据类型.
- 因此MPI 的数据类型是嵌套定义的. 为了以后叙述方便, 我们称非原始数据类型为复合数据类型.





# 类型图(type map)

- 类型序列刻划了数据的类型特征，位移序列则刻划了数据的位置特征。类型序列和位移序列元素的一一配对构成序列的类型图。

$$\text{Typemap} = \{(\text{type}_0, \text{disp}_0), (\text{type}_1, \text{disp}_1), \dots, (\text{type}_{n-1}, \text{disp}_{n-1})\}.$$

- 假设数据缓冲区的起始地址为 $\text{buff}_0$ ，则由上述类型图所定义的数据类型包含 $n$ 块数据，第 $i$ 块数据的地址为

$$\text{buff}_0 + \text{disp}_i, \text{类型为} \text{type}_i, i = 0, 1, \dots, n-1.$$



# 类型图(type map)(续)

- MPI 的原始数据类型的类型图可以写成  $\{(\text{类型}, 0)\}$ . 如MPI\_INTEGER 的类型图为  $\{(\text{INTEGER}, 0)\}$ .
- 位移序列中的位移不必是单调上升的, 表明数据类型中的数据块不要求按顺序排放. 位移也可以是负的, 即数据类型中的数据可以位于缓冲区起始地址之前.



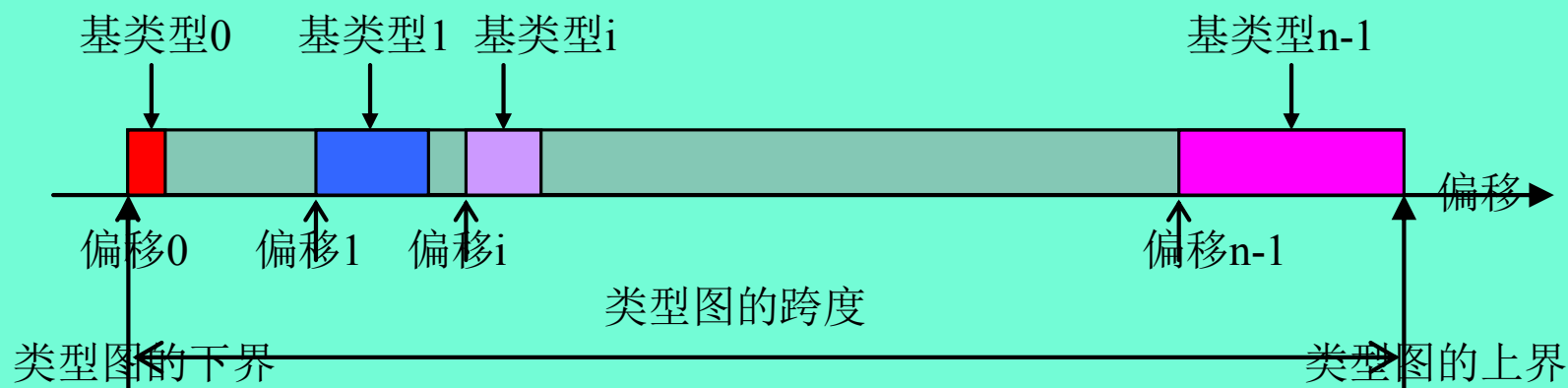
# 类型图的表示

类型图 = {  
< 基类型, 偏移 >  
< 基类型, 偏移 >  
< 基类型, 偏移 >  
...  
< 基类型, 偏移 >  
}



# 类型图的图示

类型图 = {<基类型0, 偏移0>, <基类型1, 偏移1>, ..., <基类型n-1, 偏移n-1>}



# 例1

- 假设数据类型**TYPE** 类型图为:

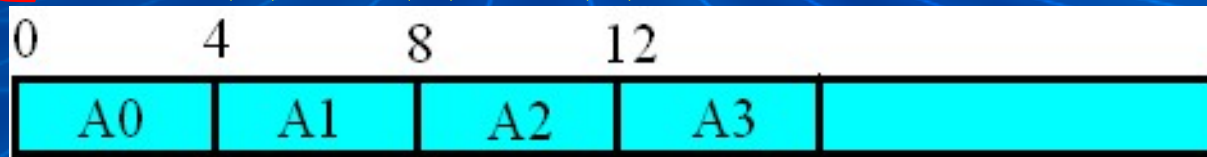
$\{(\text{integer}, 4), (\text{integer}, 12), (\text{integer}, 0)\}$

则语句:

`int A(100)`

`MPI_Send(A, 1, TYPE, ...)`

将发送?  $A(1), A(3), A(0)$





# 数据类型的大小

- 指该数据类型中包含的数据长度(字节数), 它等于类型序列中所有基本数据类型的大小之和。 数据类型的大小就是消息传递时需要发送或接收的数据长度。
- 假设数据类型type的类型图为:

$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}$

则该数据类型的大小为:

$$Sizeof(type) = \sum_{i=0}^{n-1} sizeof(type_i)$$



# 下界、上界与域

- 下界(lower bound): 数据的最小位移
- 上界(upper bound): 数据的最大位移加1, 再加上一个使得数据类型满足操作系统地址对齐要求(alignment) 的修正量  $\varepsilon$  .
- 域(extent): 上界与下界之差



# 数据类型的对界量

- 原始数据类型的对界量由编译系统决定
- 复合数据类型的对界量则定义为它的所有基本数据类型对界量的最大值
- 地址对界要求一个数据类型在内存中的(字节)地址必须是它的对界量的整数倍.



# C语言中的对界 (例2)

```
typedef struct main() {  
    {  
        char a;  
        double b;  
        int c;  
    } T;  
    T m;  
    printf( "sizeof(T)=%d,sizeof(m)=%d\n",  
           sizeof(T), sizeof(m) );  
    printf( "m.a=%d, m.b=%d, m.c=%d\n",  
           (char *)&m.a - (char *)&m,  
           (char *)&m.b - (char *)&m,  
           (char *)&m.c - (char *)&m );  
}
```



# C语言中的对界

zf@loginNode:~

```
zf@gnode1:~> ./a.out  
sizeof(T)=24, sizeof(m)=24  
m.a=0, m.b=8, m.c=16  
zf@gnode1:~> █
```

struct( char, double, int )



char

double

int





# MPI\_LB 和 MPI\_UB

- MPI提供了两个特殊数据类型MPI\_LB 和 MPI\_UB, 称为伪数据类型(pseudo datatype). 它们的大小是0, 作用是让用户人工指定一个数据类型的上下界.
- MPI 规定: 如果一个数据类型的基本类型中含有 MPI\_LB, 
$$lb(type) = \min_i \{disp_i \mid type_i = MPI\_LB\}$$
- 如果一个数据类型的基本类型中含有MPI\_UB, 则它的上界为 
$$ub(type) = \max_i \{disp_i \mid type_i = MPI\_UB\}$$



# 例 3

- 类型图  $\{(\text{MPI\_LB}, -4),$   
 $(\text{MPI\_UB}, 20),$   
 $(\text{MPI\_DOUBLE}, 0),$   
 $(\text{MPI\_INTEGER}, 8),$   
 $(\text{MPI\_BYTE}, 12)\}$

问题：下界为 -4，上界为 20，  
域为 24。



# 数据类型查询函数

- 查询指定数据类型的大小:

```
int MPI_Type_size (
    MPI_Datatype Datatype /* in */,
    int* size /*out*/)

```

- 查询指定数据类型的域:

```
int MPI_Type_extent (
    MPI_Datatype Datatype /* in */,
    MPI_Aint* extent /*out*/)

```



# 数据类型查询函数

- 查询指定数据类型的上界:

```
int MPI_Type_ub (  
    MPI_Datatype  Datatype    /* in */,  
    MPI_Aint*     displacement /*out*/)
```

- 查询指定数据类型的下界:

```
int MPI_Type_lb (  
    MPI_Datatype  Datatype    /* in */,  
    MPI_Aint*     displacement /*out*/)
```



# MPI自定义数据类型

- 连续数据类型
- 向量数据类型
- 索引数据类型
- 结构数据类型

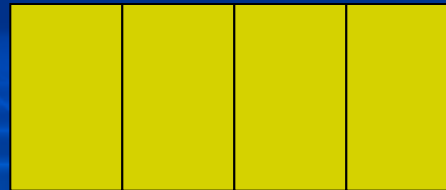




# 连续数据类型的创建

```
int MPI_Type_contiguous(  
    int          count      /* in */,  
    MPI_Datatype oldtype    /* in */,  
    MPI_Datatype* newtype   /* out */) 
```

同一类型的多次重复



作用:将连续的基类型重复作为一个整体看待



# 新类型的递交和释放

- 提交:

```
int MPI_Type_commit(  
    MPI_Datatype*    datatype)
```

- 将数据类型映射进行转换或“编译”
- 一种数据类型变量可反复定义，连续提交

- 释放:

```
int MPI_Type_free(  
    MPI_Datatype*    datatype)
```

- 将数据类型设为MPI\_DATATYPE\_NULL



# 定义矩阵的一行(例4)

- 在C中定义矩阵的一行

```
float a[4][4]; MPI_Datatype C_R;
```

```
MPI_Comm_dup (MPI_COMM_WORLD, &comm);
```

```
MPI_Type_contiguous (4,MPI_FLOAT,&C_R);
```

```
MPI_Type_commit(&C_R);
```

```
MPI_Send(&(a[2][0]),1,C_R, right, tag, comm);
```

```
MPI_Recv(&b[i][j],4,MPI_FLOAT,.....)
```



# 定义矩阵的一行图示

```
count = 4;  
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0
-----	------	------	------

1 element of  
rowtype

?

如何  
发送  
一行?



# MPI自定义数据类型

- 连续数据类型
- 向量数据类型
- 索引数据类型
- 结构数据类型



# 向量数据类型的生成

```
int MPI_Type_vector(  
    int          count      /* in */,  
    int          blocklen   /* in */,  
    int          stride     /* in */,  
    MPI_Datatype oldtype    /* in */,  
    MPI_Datatype* newtype   /* out */) 
```

- count 块数（非负整数）
- blocklength 每块中的元素个数（非负整数）
- stride 每块开始间隔的 **元素个数** (integer)





# 向量数据类型生成

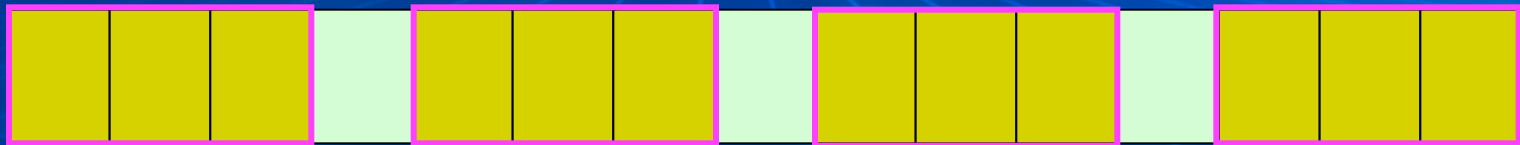
`MPI_Type_vector(count, blocklength, stride, oldtype, newtype)`

- 类型重复形成块，多个块按一定的间隔排列  
间隔可以是以本类型为单位，也可以是以字节为单位



4,3,4

块内部是没有间隔的



# 向量数据类型生成

```
int MPI_Type_hvector(  
    int          count      /* in */,  
    int          blocklen   /* in */,  
    MPI_Aint      stride     /* in */,  
    MPI_Datatype  oldtype    /* in */,  
    MPI_Datatype* newtype    /* out */) 
```

- MPI\_Type\_vector 中 stride 以 oldtype 的域为单位
- MPI\_Type\_hvector 中 stride 以字节为单位



# 定义矩阵的一列(例5)

- 在C中定义矩阵的一列

```
float a[4][4]; MPI_Datatype C_C;
```

```
MPI_Type_vector (4,1,4,MPI_FLOAT,&C_C);
```

```
MPI_Type_commit(&C_C);
```

```
MPI_SEND(&(a[0][1]),1,C_C, right, tag, comm)
```



# 定义矩阵的一系列图示

```
count = 4; blocklength = 1; stride = 4;  
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,  
                &column_type);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[0][1], 1, column_type, dest, tag, comm);
```

2.0	6.0	10.0	14.0
-----	-----	------	------

1 element of  
column\_type



# 思考

$U(0, 0)$	$U(0, 1)$	.....	$U(0, M-1)$	$U(0, M)$
$U(1, 0)$	$U(1, 1)$	.....	$U(1, M-1)$	$U(1, M)$
.....	.....	.....	.....	.....
$U(N-1, 0)$	$U(N-1, 1)$	.....	$U(N-1, M-1)$	$U(N-1, M)$
$U(N, 0)$	$U(N, 1)$	.....	$U(N, M-1)$	$U(N, M)$



# 答案

```
int a[n+1][m+1]; MPI_Datatype N_T;
```

```
MPI_Type_vector (n-1,m-1,m+1,MPI_INT,&N_T);
```

```
MPI_Type_commit(&N_T);
```

```
MPI_SEND(&(a[1][1]),1,N_T, right, tag, comm);
```





# MPI自定义数据类型

- 连续数据类型
- 向量数据类型
- 索引数据类型
- 结构数据类型



# 索引数据类型

```
int MPI_Type_indexed(  
    int          count      /* in */,  
    int          blocklens[] /* in */,  
    int          indices[]  /* in */,  
    MPI_Datatype oldtype    /* in */,  
    MPI_Datatype* newtype   /* out */) 
```

- **count** number of blocks -- also number of entries in indices and blocklens
- **blocklens** number of elements in each block
- **indices** displacement of each block in multiples of old\_type

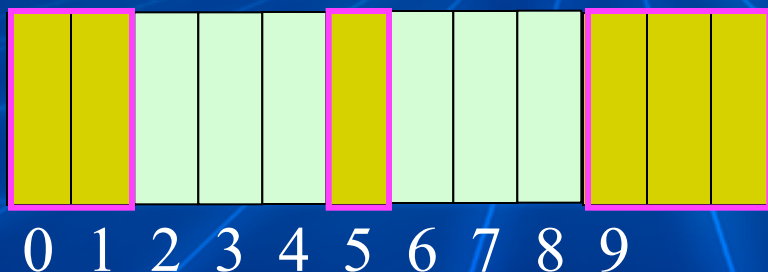


# 索引数据类型

`MPI_TYPE_INDEXED(count,array_of_blocklengths,array_of_displacements,  
oldtype,newtype)`

重复形成块，不同的块放到不同的位置，位置的指定可以是以旧数据类型为单位

`3,{2,1,3},{0,5,9}`



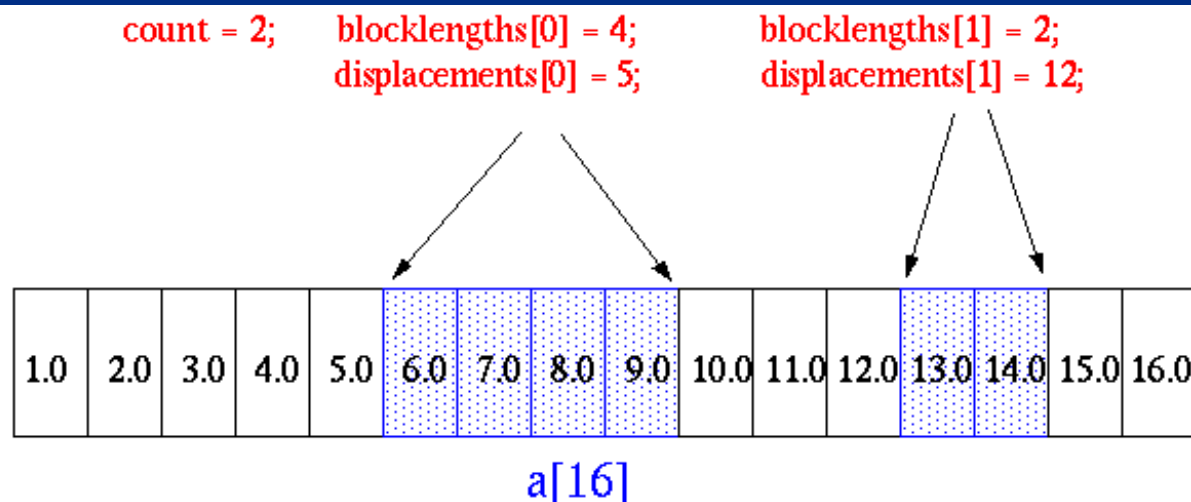
# 索引数据类型

```
int MPI_Type_hindexed(  
    int          count          /* in */,  
    int          blocklens[]    /* in */,  
    int          indices[]      /* in */,  
    MPI_Datatype oldtype        /* in */,  
    MPI_Datatype* newtype       /* out */);
```

- **count** number of blocks -- also number of entries in indices and blocklens
- **blocklens** number of elements in each block
- **indices** displacement of each block in *bytes*

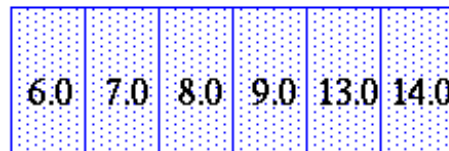


# 例 6



```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



1 element of  
indextype



# 问题

- MPI\_type\_vector()与MPI\_type\_indexed()的区别?
- MPI\_Type\_indexed 与MPI\_Type\_vector 的区别在于每个数据块的长度可以不同, 数据块间也可以不等距.





# MPI自定义数据类型

- 连续数据类型
- 向量数据类型
- 索引数据类型
- 结构数据类型



# 结构数据类型

```
int MPI_Type_struct(  
    int          count      /* in */,  
    int          blocklens[] /* in */,  
    MPI_Aint      indices[]  /* in */,  
    MPI_Datatype types[]     /* in */,  
    MPI_Datatype* newtype    /* out */) 
```

- **count** number of blocks (integer)
- **blocklens** number of elements in each block
- **indices** byte displacement of each block (array)
- **types** of elements in each block



# 结构数据类型

`MPI_Type_struct(count,array_of_blocklens,array_of_displacements,  
array_of_types,newtype)`

将多个不同的旧数据类型进行组合，而前面的数据类型生成方法都是对一个旧数据类型进行重复。

`3,{2,1,3},{0,6,18}`

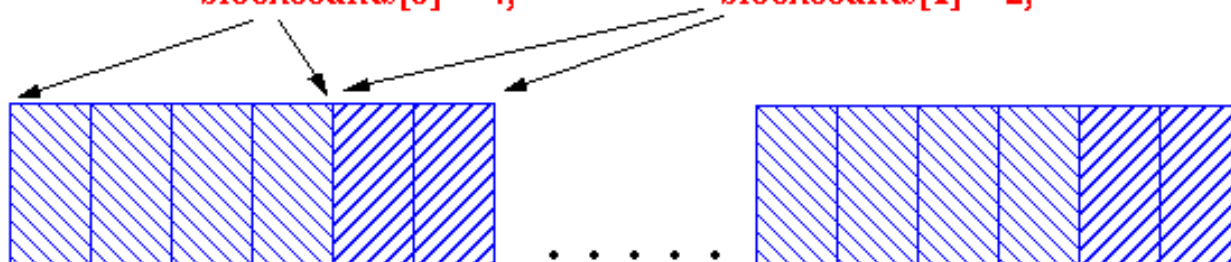


# 例7

```
typedef struct { float x,y,z,velocity; int n,type; } Particle;  
Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);
```

```
count = 2; oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT;  
offsets[0] = 0; offsets[1] = 4 * extent;  
blockcounts[0] = 4; blockcounts[1] = 2;
```



particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);
```

```
MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.



# 数据的打包与拆包

- 在MPI 中, 通过使用特殊数据类型 `MPI_PACKED`, 用户可以将不同的数据进行打包后再一次发送出去, 接收方在收到消息后再进行拆包.
- 为了与早期其它并行库兼容
- **MPI不建议用户进行显式的数据打包**



# 数据的打包

```
int MPI_Pack (  
    void*          inbuf      /* in */,  
    int            incount    /* in */,  
    MPI_Datatype    datatype  /* in */,  
    void*          outbuf     /* in */,  
    int            outsize    /* in */,  
    int*           position   /*in/out*/,  
    MPI_Comm        comm      /* in */) 
```

- 该函数将缓冲区inbuf 中的incount 个类型为 datatype 的数据进行打包. 打包后的数据放在缓冲区outbuf 中. outsize 给出的是outbuf 的总长度 (字节数, 供函数检查打包缓冲区是否越界用).





# 数据的打包

- position 是打包缓冲区中的位移，第一次调用MPI\_Pack 前用户程序将position 设为0
- 随后MPI\_Pack 将自动修改它, 使得它总是指向打包缓冲区中尚未使用部分的起始位置
- 每次调用MPI\_Pack 后的position 实际上就是已打包的数据的总长度



## 例 8

```
MPI_Comm_dup(MPI_COMM_WORLD, &com);  
if(my_rank == 0){  
    position = 0;  
    MPI_Pack(n,1,MPI_FLOAT,buff,64,position,com);  
    MPI_Pack(m,1,MPI_INT,buff,64,position,com);  
    MPI_Pack(A,5,MPI_FLOAT,buff,64,position,com);  
    MPI_Send(buff,64,MPI_PACK,1,111,com);  
}
```



# 数据的拆包

```
int MPI_Unpack (  
    void*          packbuf      /* in */,  
    int            insize       /* in */,  
    int*           position     /*in/out*/,  
    void*          outbuf       /*out*/,  
    int            outcount     /* in */,  
    MPI_Datatype   datatype     /* in */,  
    MPI_Comm       comm        /* in */)
```

- 从packbuf 中拆包outcount 个类型为datatype 的数据到 outbuf 中. 函数中各参数的含义与MPI Pack 类似, 只不过这里的packbuf和insize 对应于MPI Pack中的outbuf 和outsize, 而outbuf 和outcount 则对应于MPI\_Pack 中的 inbuf 和incount.



## 例 9

```
MPI_Comm_dup(MPI_COMM_WORLD, &com);  
if(my_rank == 1){  
    MPI_Recv(buff1,64,MPI_PACK,0,111,com,&status);  
    position = 0;  
    MPI_Unpack(buff1,64,position,n1,1,MPI_FLOAT,com);  
    MPI_Unpack(buff1,64,position,m1,1,1,MPI_INT,com);  
    MPI_Unpack(buff1,64,position,A1,5,MPI_FLOAT,com);  
}
```



# 数据类型函数汇总

函数类型	函数表达
连续数据	<b>MPI_Type_contiguous</b>
向量数据	<b>MPI_Type_vector</b>
	<b>MPI_Type_hvector</b>
索引数据	<b>MPI_Type_indexed</b>
	<b>MPI_Type_hindexed</b>
结构数据	<b>MPI_Type_struct</b>
类型查询	<b>MPI_Type_size</b>
	<b>MPI_Type_extent</b>
	<b>MPI_Type_lb</b>
	<b>MPI_Type_ub</b>
类型递交	<b>MPI_Type_commit</b>
类型释放	<b>MPI_Type_free</b>



# 上机实验

- 将矩阵A 的转置拷贝到矩阵B 中

Type1

$a(0,0)$	$a(0,1)$	.....	$a(0,n)$
$a(1,0)$	$a(1,1)$	.....	$a(1,n)$
.....			
$a(m,0)$	$a(m,1)$	.....	$a(m,n)$





# 矩阵转置

多种实现方法

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16



# 矩阵转置

```
#include <stdio.h>
#include "mpi.h"
#define N 10
int main(int argc, char* argv[]) {
    int    my_rank, tag = 0, a[N][N], b[N][N], i, j;
    MPI_Status    status;
    MPI_Comm      comm;
    MPI_Datatype   column_type;
    MPI_Datatype   row_type;

    MPI_Init(&argc, &argv);                /*初始化MPI*/
    MPI_Comm_dup (MPI_COMM_WORLD, &comm);
    MPI_Comm_rank (comm, &my_rank);         /*获得进程的ID*/

    for (i = 0; i < N; i++) {                /*Initial the matrix a & b*/;
        for (j = 0; j < N; j++) {
            a[i][j] = tag++;    b[i][j] = 0;
        }
    }
}
```



# 矩阵转置

```
MPI_Type_vector(N,1,N,MPI_INT,&column_type); /*定义新数据类型 */
```

```
MPI_Type_commit(&column_type); /*递交新数据类型 */
```

```
if(my_rank == 0){ /* P0 send to P1*/
```

```
    for (i = 0; i<N; i++)
```

```
        MPI_Send(&(a[0][i]),1,new_type,1,tag+i,comm);
```

```
    打印矩阵A;
```

```
}
```

```
if(my_rank == 1){ /* P1 recv from P0 */
```

```
    for(i = 0; i < N; i++)
```

```
        MPI_Recv(&b[i][0],N,MPI_INT,0,tag+i,comm,&status);
```

```
    打印矩阵B;
```

```
}
```

```
MPI_Finalize();
```

```
}
```



中国科学院计算技术研究所

INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

# 矩阵转置

```
MPI_Type_vector(N,1,N,MPI_INT,&column_type); /*定义新数据类型 */
MPI_Type_contiguous(N,MPI_INT,&raw_type);
MPI_Type_commit(&column_type); /*递交新数据类型 */
MPI_Type_commit(&raw_type); /*递交新数据类型 */

if(my_rank == 0){ /* P0 send to P1*/
    for (i = 0; i<N; i++)
        MPI_Send(&a[0][i],1,column_type,1,tag+i,comm);
    打印矩阵A;
}

if(my_rank == 1){ /* P1 recv from P0 */
    for(i = 0; i < N; i++)
        MPI_Recv(&b[i][0],1,raw_type,0,tag+i,comm,&status);
        /* MPI_Recv(&b[i][0],N,MPI_INT,0,tag+i,comm,&status); */
    打印矩阵B;
}

MPI_Finalize();
}
```



# MPI内容

- 基本概念
- 点到点通信
- 自定义数据类型
- 集合通信



# 组通信概述

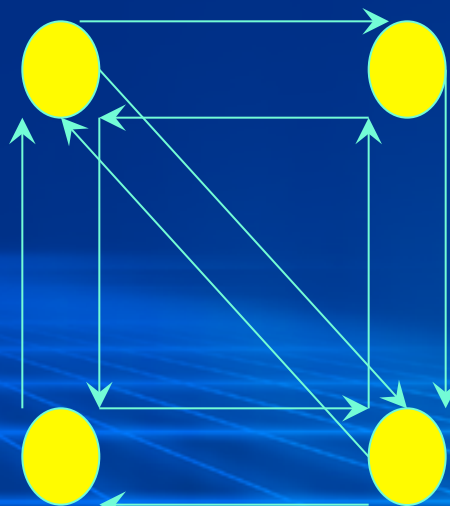
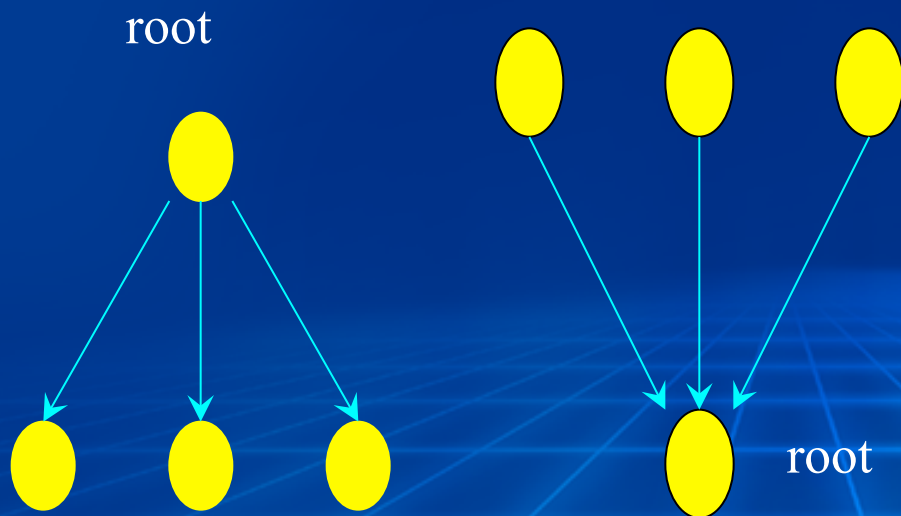
- 通信域限定哪些进程参加以及组通信的上下文
- 组通信调用可以和点对点通信共用一个通信域
  - MPI保证由组通信调用产生的消息不会和点对点调用产生的消息相混淆
- 在组通信中不需要通信消息标志参数
- 组通信一般实现三个功能通信、同步和计算
  - 通信功能主要完成组内数据的传输
  - 同步功能实现组内所有进程在特定地点在执行进度上取得一致
  - 计算功能要对给定的数据完成一定的操作





# 三种通信方式

- 一对多、多对一、多对多 (按通信方向)



# 组通信中的同步

- 点到点通信的完成，重新使用缓冲区
- 一个进程组通信的完成，并不表示其他所有进程的组通信都已经完成。
- 同步操作，完成各个进程之间的同步，协调各个进程的进度和步伐。



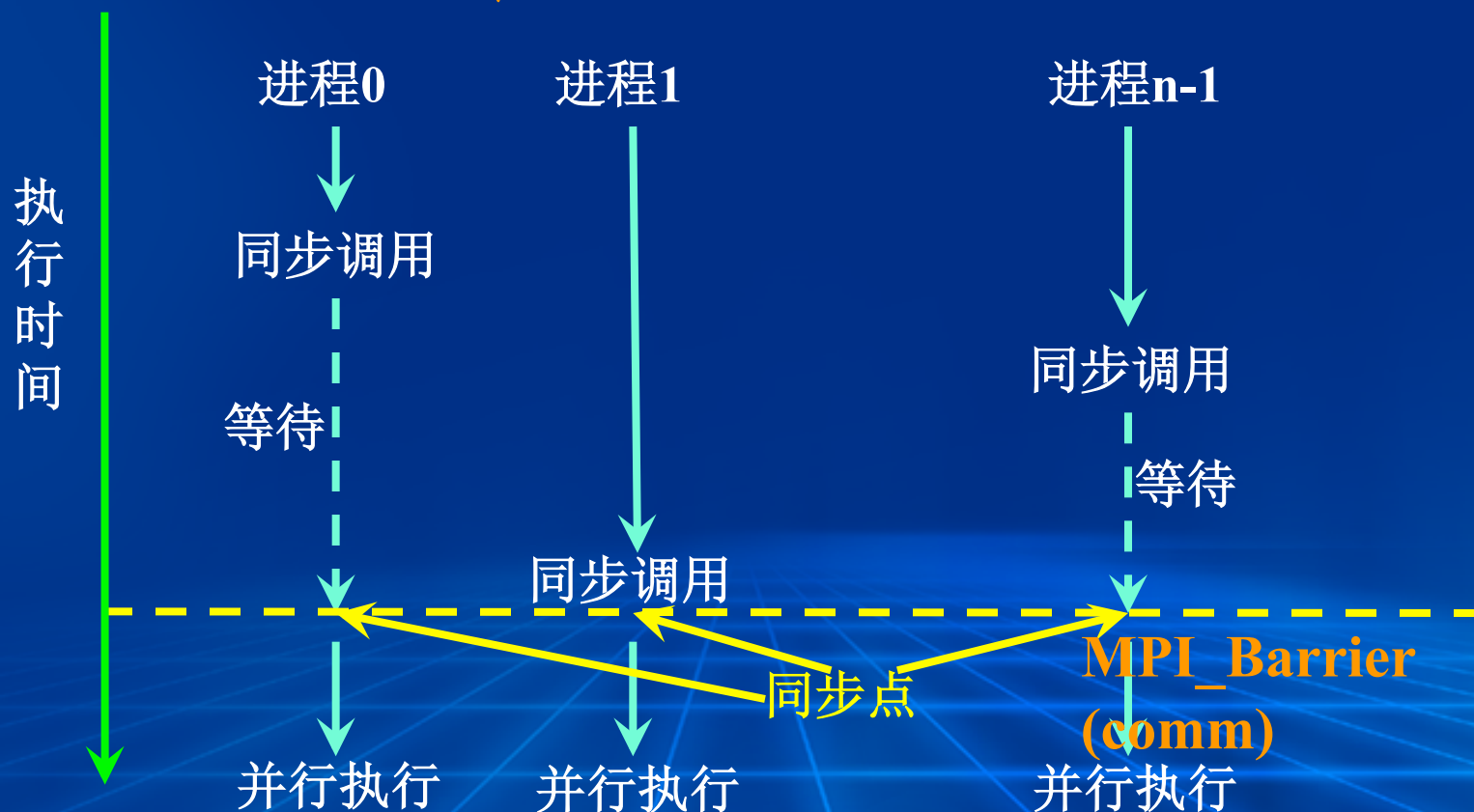
# 同步函数 MPI\_Barrier

```
int MPI_Barrier (  
    MPI_Comm comm /* in */)
```

- *MPI* 唯一的一个同步函数, 当comm中的所有进程都执行这个函数后才返回。
- 如果有一个进程没有执行此函数, 其余进程将处于等待状态。在执行完这个函数之后, 所有进程将同时执行其后的任务。



# 组通信中的同步



# 实例：MPI\_Barrier

```
MPI_Init( &argc, &argv );  
MPI_Comm_rank( MPI_COMM_WORLD, &rank );  
MPI_Comm_dup ( MPI_COMM_WORLD, &comm);  
if (rank == 0) value = 100;  
MPI_Bcast( &value, 1, MPI_INT, 0, comm);  
    /*将该数据广播出去*/  
printf( "Process %d got %d\n", rank, value );  
MPI_Barrier (MPI_COMM_WORLD);  
    /* 同步 */  
MPI_Finalize();
```

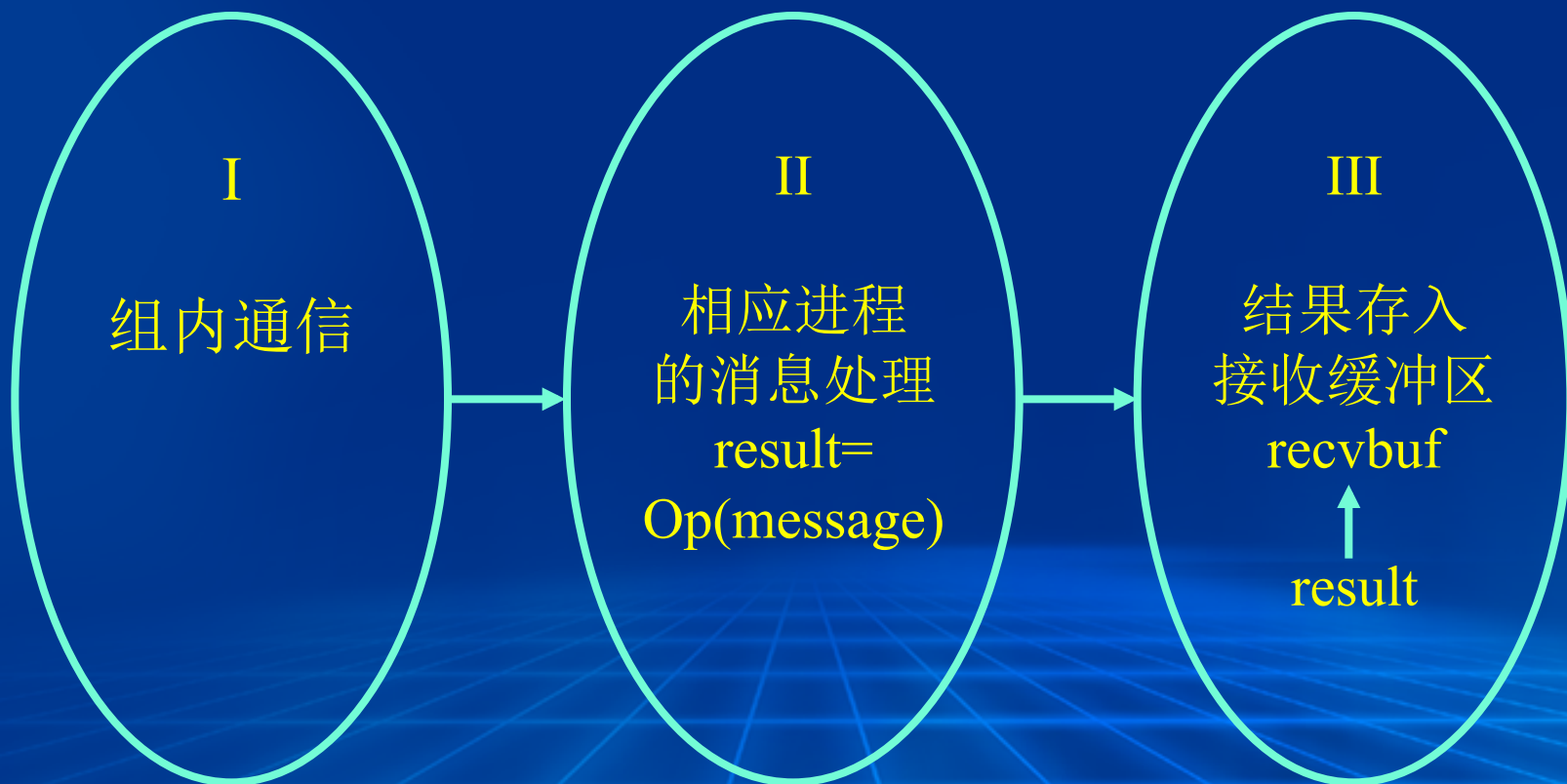


# 组通信中的计算

- 组通信除了通信和同步之外，还可进行计算。  
MPI组通信的计算功能是分三步实现的
  - 首先是通信的功能，即消息根据要求发送到目的进程，目的进程也已经接收到了各自所需要的消息
  - 然后是对消息的处理即计算部分。MPI组通信有计算功能的调用都指定了计算操作，用给定的计算操作对接收到的数据进行处理
  - 最后一步是将处理结果放入指定的接收缓冲区



# 组通信中的计算图示



# 全局数据运算Reduce

- MPI\_Reduce将组内每个进程输入缓冲区中的数据按给定的操作op进行运算，并将结果返回到根进程的输出缓冲区中。
- 输入缓冲区由参数sendbuf、count和datatype定义。输出缓冲区由参数recvbuf、count和datatype定义
- 要求两者的元素数目和类型都必须相同。所有组成员都用同样的count、datatype、op、root和comm来调用此例程，故所有进程都提供长度相同、元素类型相同的输入和输出缓冲区
- 每个进程可能提供一个元素或一系列元素，组合操作依次针对每个元素进行



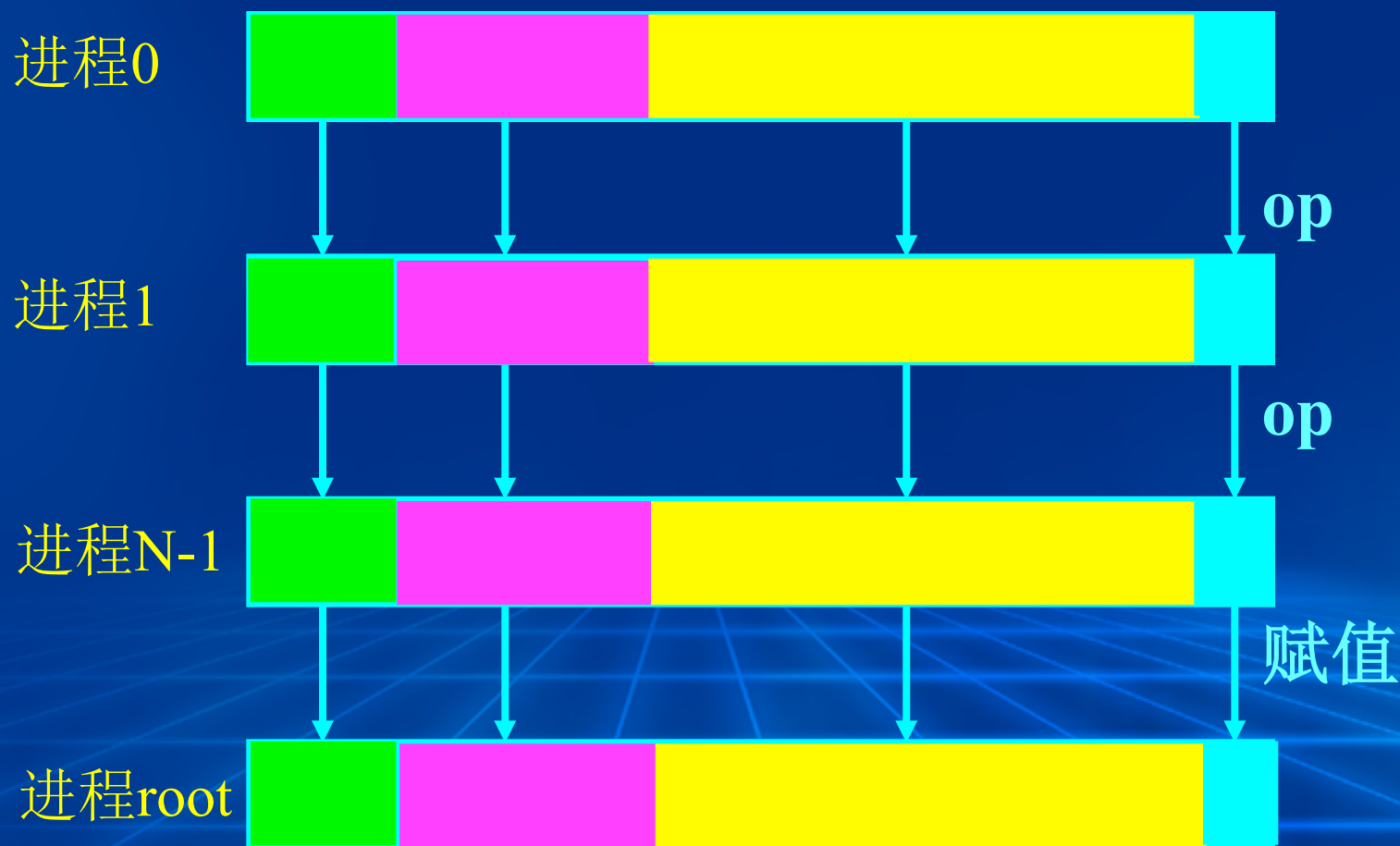
# 全局归约 MPI\_Reduce

```
int MPI_Reduce (  
    void* openand /* in */,  
    void* result /* in */,  
    int count /* in */,  
    MPI_Datatype datatype /* in */,  
    MPI_Op operator /* out */,  
    int root /* in */,  
    MPI_Comm comm /* in */) 
```

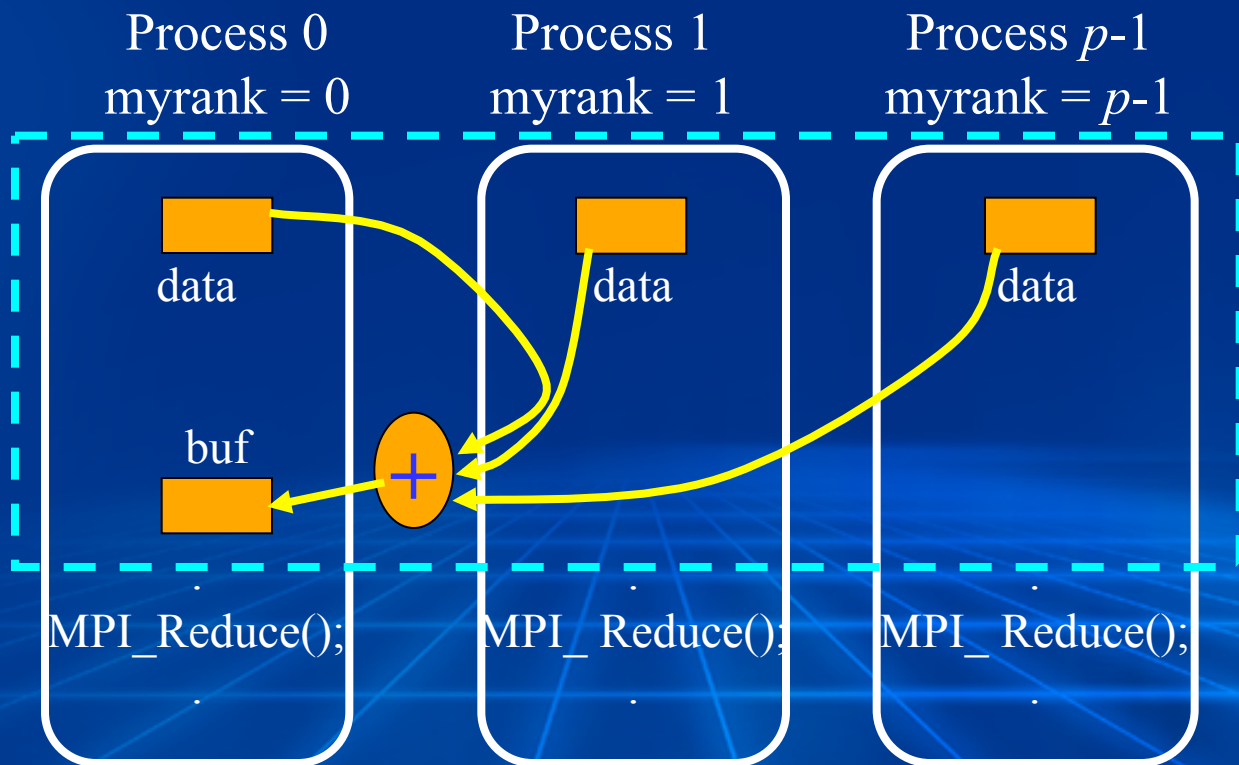
openand	操作数(发送缓冲区)起始地址
result	接收缓冲区(结果)的地址
count	发送缓冲区数据个数
operator	归约操作符

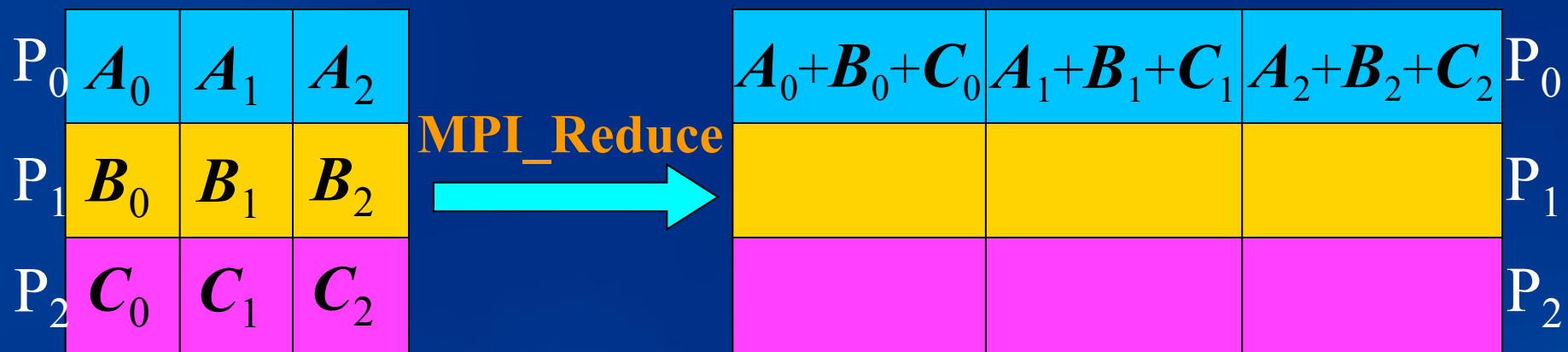


# MPI\_Reduce 图示



# 归约求和图示







# MPI预定义操作

名字	含义	名字	含义
<b>MPI_MAX</b>	最大值	<b>MPI_LOR</b>	逻辑或
<b>MPI_MIN</b>	最小值	<b>MPI_BOR</b>	按位或
<b>MPI_SUM</b>	求和	<b>MPI_LXOR</b>	逻辑异或
<b>MPI_PROD</b>	求积	<b>MPI_BXOR</b>	按位异或
<b>MPI LAND</b>	逻辑与	<b>MPI_MAXLOC</b>	求最大值位置
<b>MPI_BAND</b>	按位与	<b>MPI_MINLOC</b>	求最小值位置



# 数据广播 MPI\_Bcast

- MPI\_Bcast完成从root进程将一条消息广播发送到组内的所有进程，包括它本身在内
  - 其执行结果是将根进程通信消息缓冲区中的消息拷贝到其他所有进程中去
  - 组内所有进程不管是root进程本身还是其它的进程都使用**同一个通信域comm和根标识root**
  - 数据类型datatype可以是预定义或派生数据类型
  - 其它进程指定的**通信元素个数count、数据类型datatype**必须和根进程指定的count和datatype**保持一致**



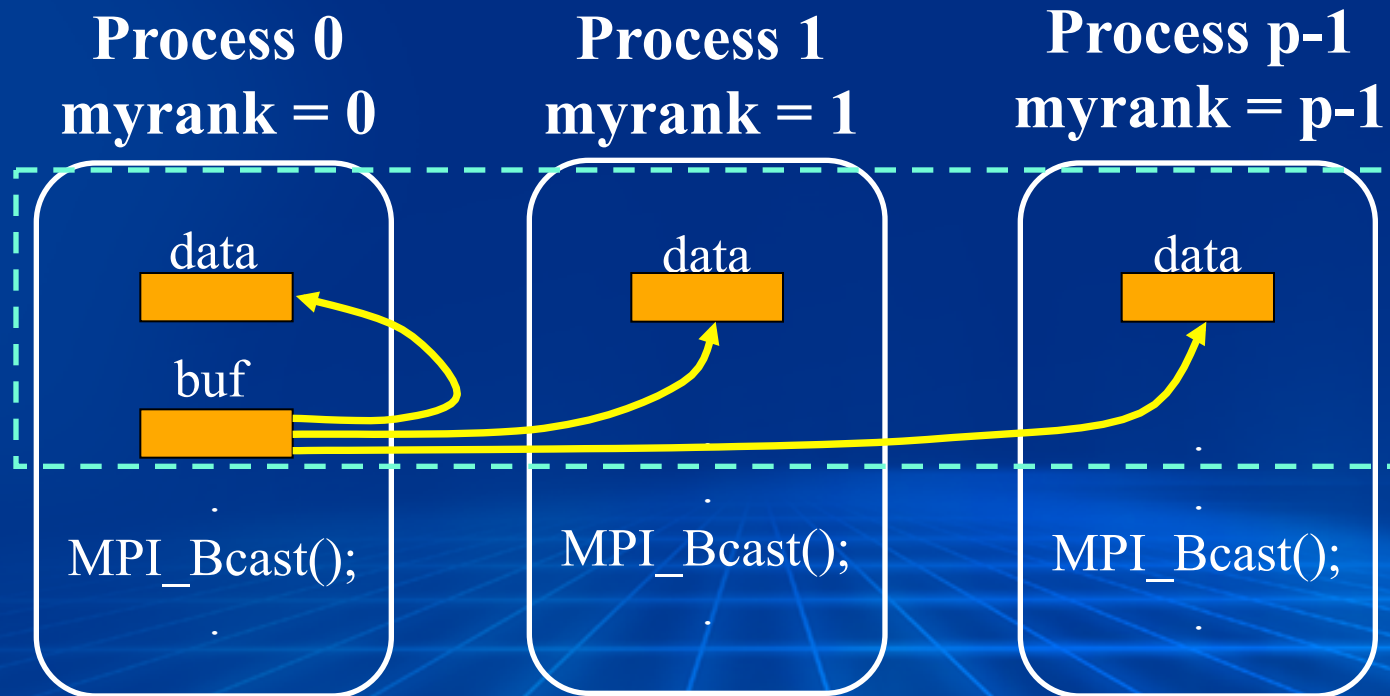
# MPI\_Bcast

```
int MPI_Bcast (  
    void*          buffer      /*in/out*/  
    int            count       /* in */  
    MPI_Datatype    datatype   /* in */  
    int            root        /* in */  
    MPI_Comm        comm       /* in */)
```

buffer	通信消息缓冲区的起始地址
count	将广播出去/或接收的数据个数
datatype	广播/接收数据的数据类型
root	广播数据的根进程的标识号
comm	通信域



# MPI\_Bcast 图示





# 实例：MPI\_Bcast

```
MPI_Init( &argc, &argv );  
MPI_Comm_rank( MPI_COMM_WORLD, &rank );  
MPI_Comm_dup ( MPI_COMM_WORLD, &comm);  
if (rank == 0)    /*进程0读入需要广播的数据*/  
    scanf( "%d", &value );  
MPI_Bcast( &value, 1, MPI_INT, 0, comm);  
    /*将该数据广播出去*/  
printf( "Process %d got %d\n", rank, value );  
    /*各进程打印收到的数据*/  
MPI_Finalize( );
```

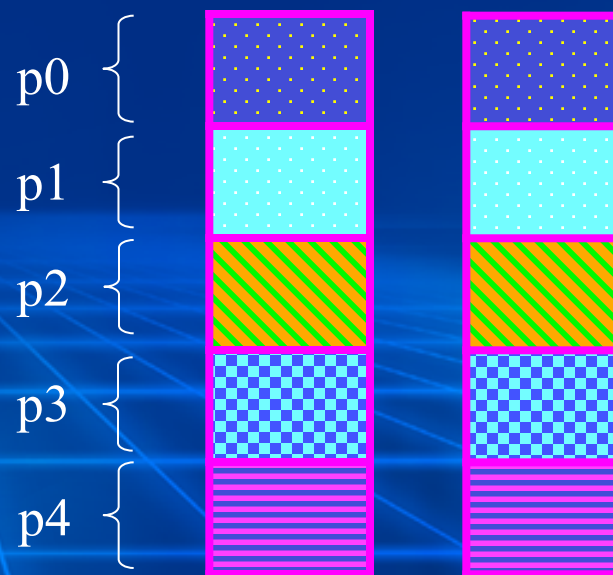
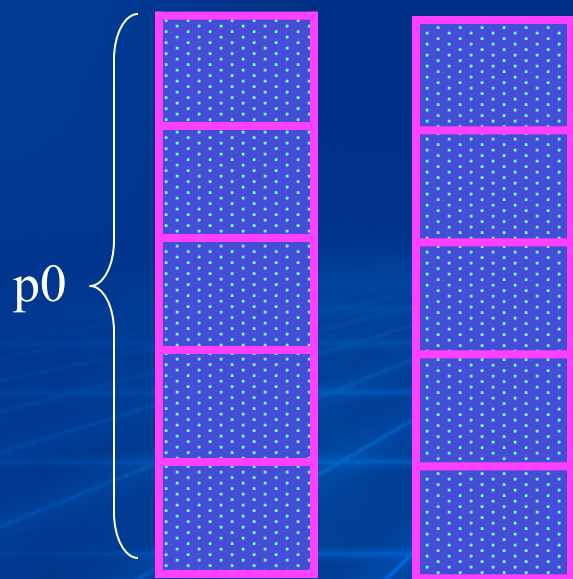




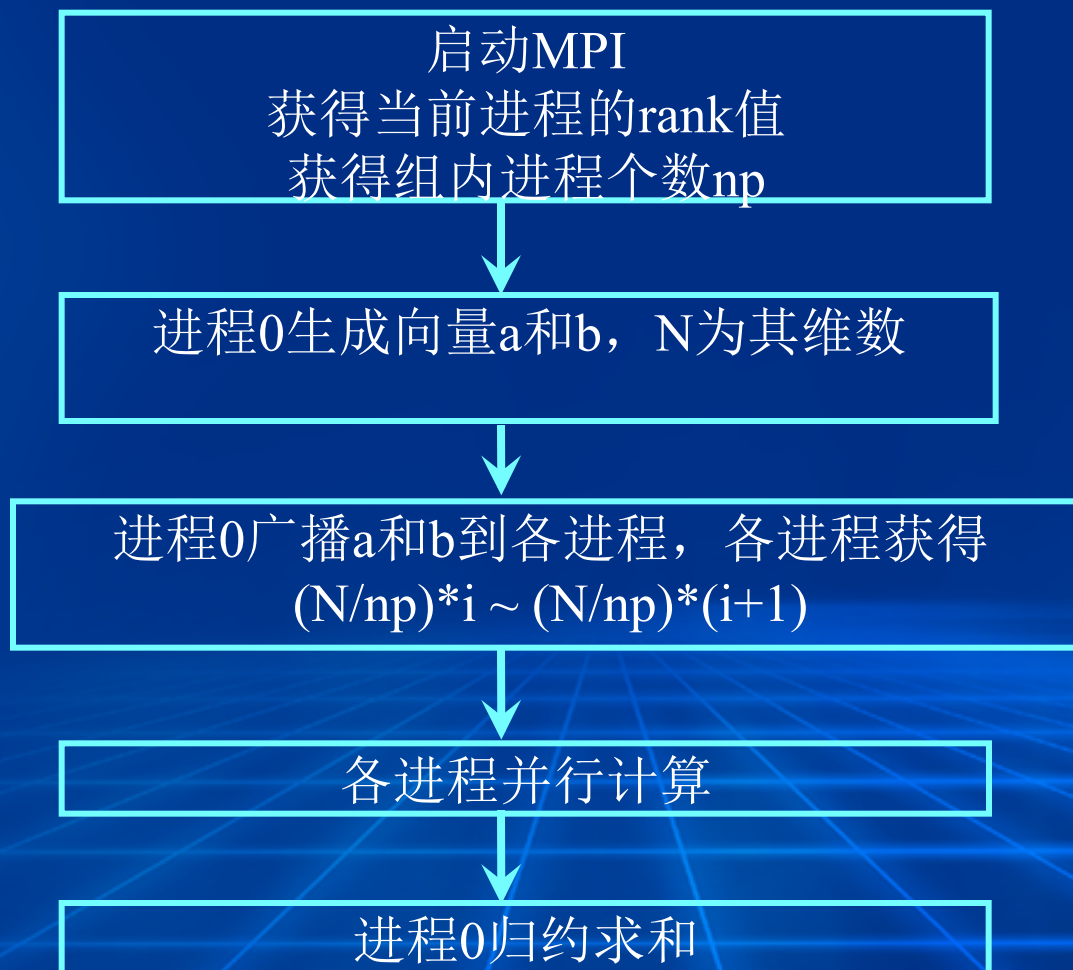
# 实例：求向量点积

$$c = \sum_{i=0}^{n-1} a_i \cdot b_i$$

$$c = \sum_{j=0}^{n/p} \sum_{i=0}^{n_j-1} a_i \cdot b_i$$



# 求向量点积计算流程



# 向量点积代码1

```
#define N 20000
```

```
main(int argc, char** argv){  
    int *x, *y, gsize, size, myrank, i;  
    float local_sum=0.0, sum;  
    MPI_Status status; MPI_Comm comm;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_dup(MPI_COMM_WORLD, &comm);  
    MPI_Comm_rank(comm, &myrank);  
    MPI_Comm_size(comm, &gsize);
```

```
    x=(int*)malloc(N * sizeof(int));  
    y=(int*)malloc(N * sizeof(int));
```



# 向量点积代码2

```
if(myrank == 0)    /*给两个向量x, y 赋值*/  
    for(i=0;i<N;i++) { x[i] = i + 1;  y[i] = i + 1; }  
  
/* 进程0广播向量x和y到各个进程*/  
MPI_Bcast(x,N,MPI_INT,0,comm);  
MPI_Bcast(y,N,MPI_INT, 0,comm);  
  
size = N / gsize;  
for(i=0; i<size; i++) { /*各进程并行计算局部向量点积*/  
    local_sum = local_sum + x[myrank*size+i] * y[myrank*size  
+i];}  
  
MPI_Reduce(&local_sum,&sum,1, /*进程0归约求和*/  
    MPI_FLOAT,MPI_SUM,0,comm);  
  
if(myrank==0) printf("the sum of dot produce is:%f\n",sum);  
free(x);  
free(y);
```



# 数据收集 MPI\_Gather

- 把所有进程(包括root) 的数据聚集到root 进程中, 并且按顺序存放在接收缓冲区中.
- 其结果就象一个进程组中的N个进程(包括root)都执行了一个发送调用, 同时根进程执行了N次接收调用.



# MPI\_Gather 函数

```
int MPI_Gather (  
    void* sendbuf /* in */  
    int sendcount /* in */  
    MPI_Datatype sendtype /* in */  
    void* recvbuf /* out */  
    int recvcount /* in */  
    MPI_Datatype recvtype /* in */  
    int root /* in */  
    MPI_Comm comm /* in */) 
```

sendbuf 发送缓冲区起始地址  
sendcount 发送缓冲区数据个数  
sendtype 发送缓冲区数据类型

recvbuf 接收缓冲区起始地址  
recvcount 接收缓冲区数据个数  
recvtype 接收缓冲区数据类型

**Root ONLY**

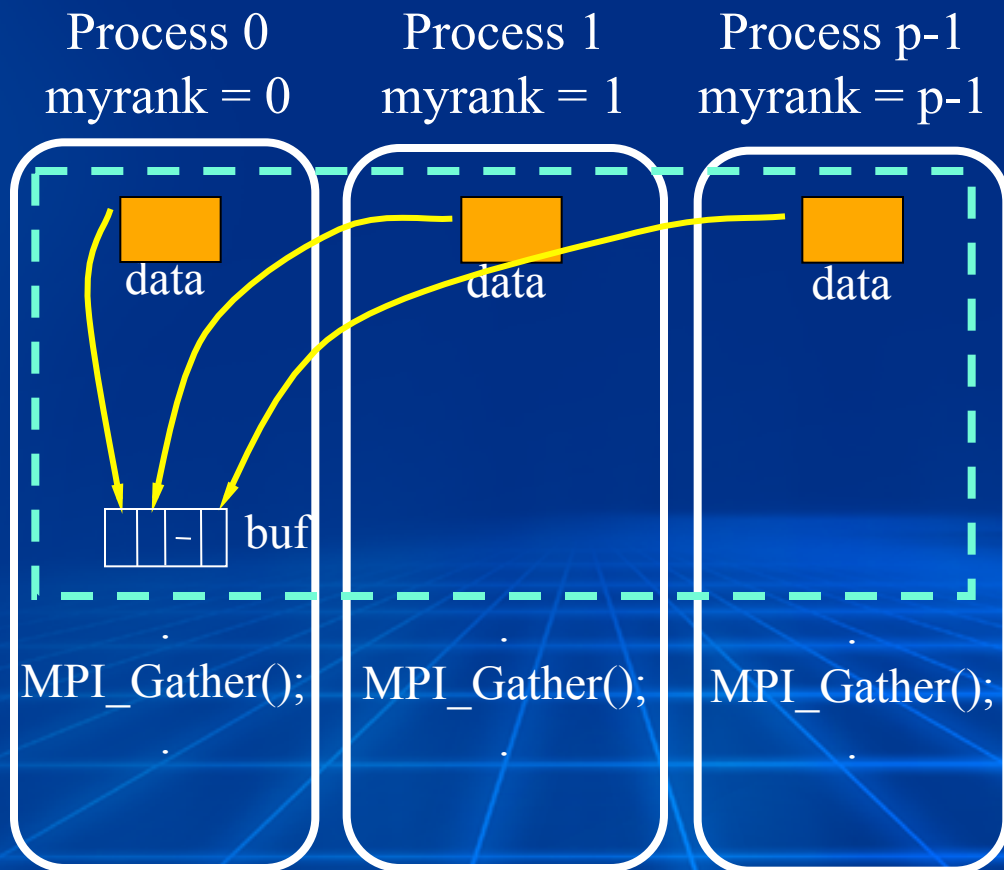




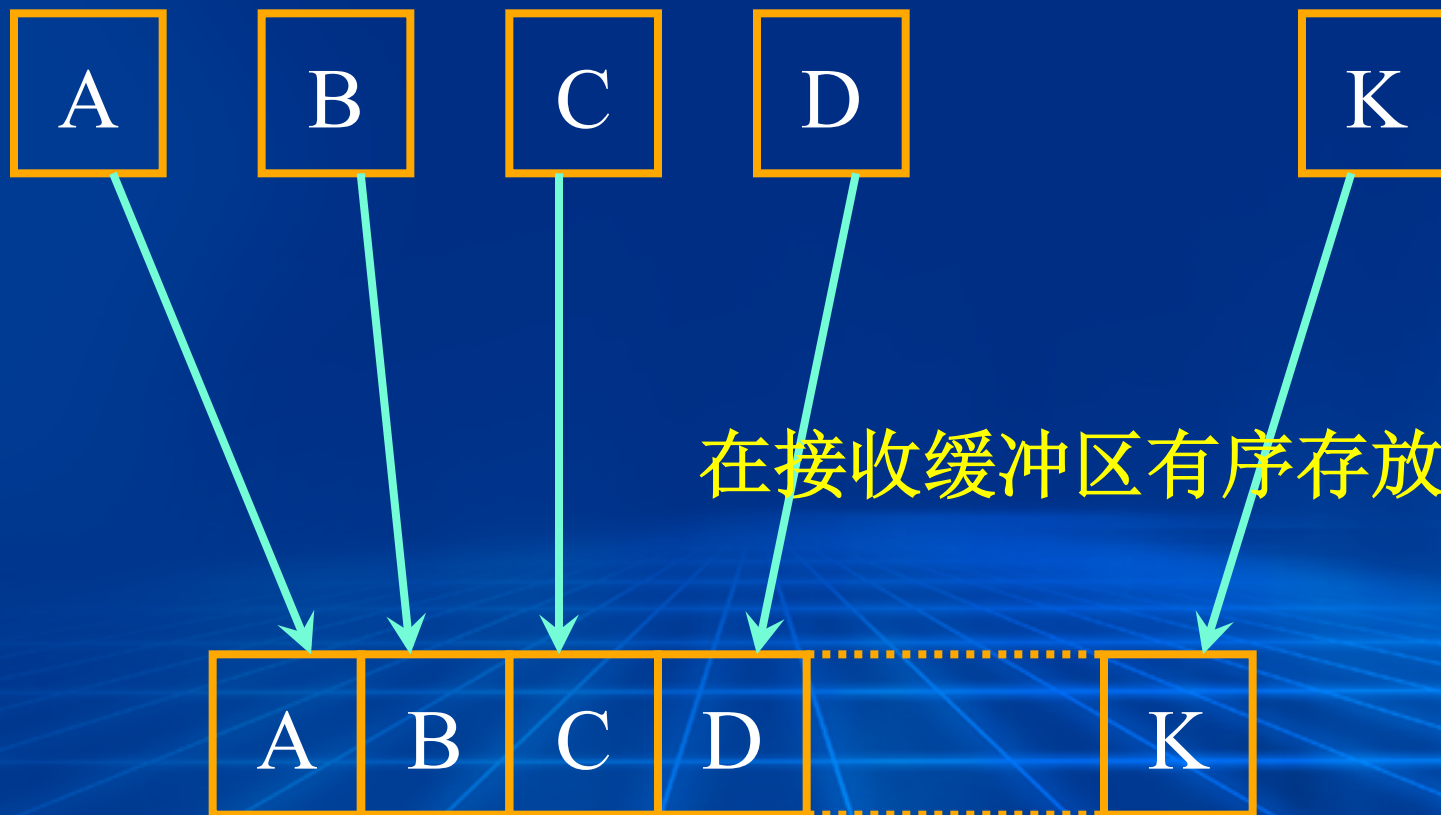
# MPI\_Gather函数详解

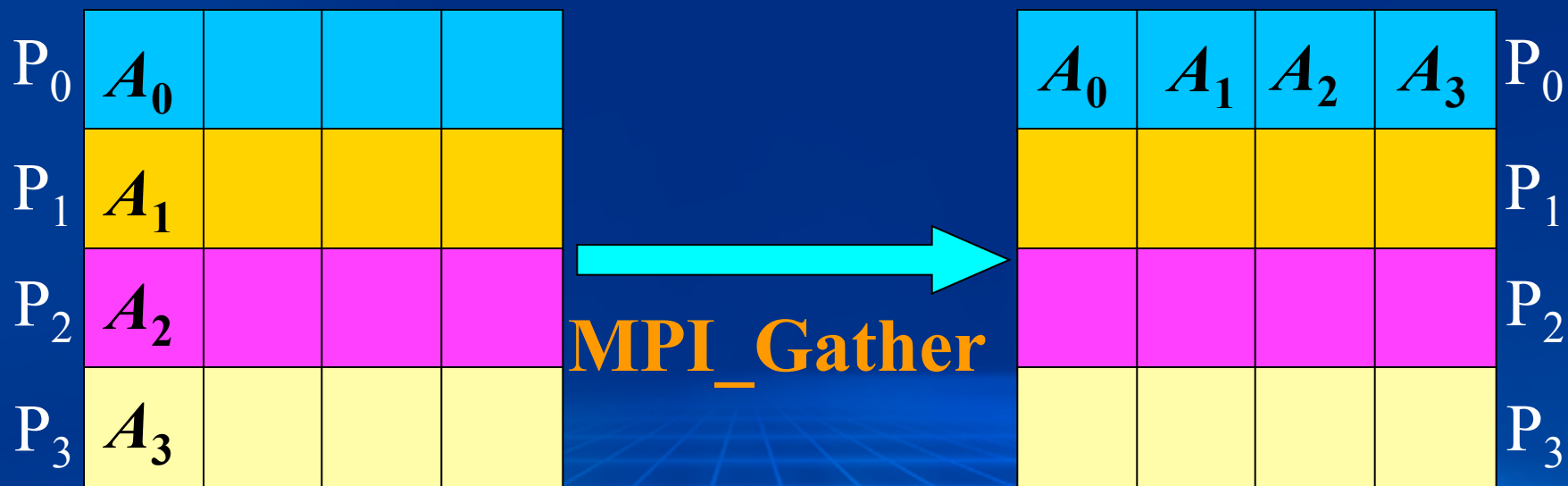
- 从各个进程收集到的数据一般是互不相同的
- 收集调用每个进程的发送数据个数sendcount和发送数据类型sendtype都是相同的，都和根进程中接收数据个数recvcount和接收数据类型recvtype相同。
- root和comm在所有进程中都必须是一致的
- 根进程中指定的接收数据个数是指从每一个进程接收到的数据的个数而不是总的接收个数
- 对于所有非根进程接收消息缓冲区被忽略但是各个进程必须提供这一参数
- 所有参数对根进程都是有意义的，而对于其它进程只有sendbuf、sendcount、sendtype、root和comm有意义，其它的参数虽没有意义但却**不能省略**

# MPI\_Gather 图示



# MPI\_Gather图示





# 实例：MPI\_Gather

```
MPI_Comm comm;
```

```
int size, root, s_data[100], *rbuf;
```

```
...
```

```
MPI_Comm_size( comm, &size );
```

```
rbuf = (int *) malloc( size * 100*sizeof(int));
```

```
/* 申请接收缓冲区 */
```

```
MPI_Gather( s_data, 100, MPI_INT, rbuf, 100,  
            MPI_INT, root, comm);
```

```
...
```

```
MPI_Finalize( );
```



# 数据散发 MPI\_Scatter

- MPI\_Scatter是一对多的组通信调用
- 但是和广播不同，Root向各个进程发送的数据可以是不同的
- MPI\_Scatter和MPI\_Gather的效果正好相反两者互为逆操作





# MPI\_Scatter 函数

```
int MPI_Scatter (  
    void*                sendbuf    /* in */  
    int                  sendcount  /* in */  
    MPI_Datatype          sendtype   /* in */  
    void*                recvbuf    /* out */  
    int                  recvcnt    /* in */  
    MPI_Datatype          recvtype   /* in */  
    int                  root        /* in */  
    MPI_Comm              comm       /* in */) ROOT ONLY
```

sendbuf	发送缓冲区起始地址	recvbuf	接收缓冲区起始地址
sendcount	发送缓冲区数据个数	recvcnt	接收缓冲区数据个数
sendtype	发送缓冲区数据类型	recvtype	接收缓冲区数据类型

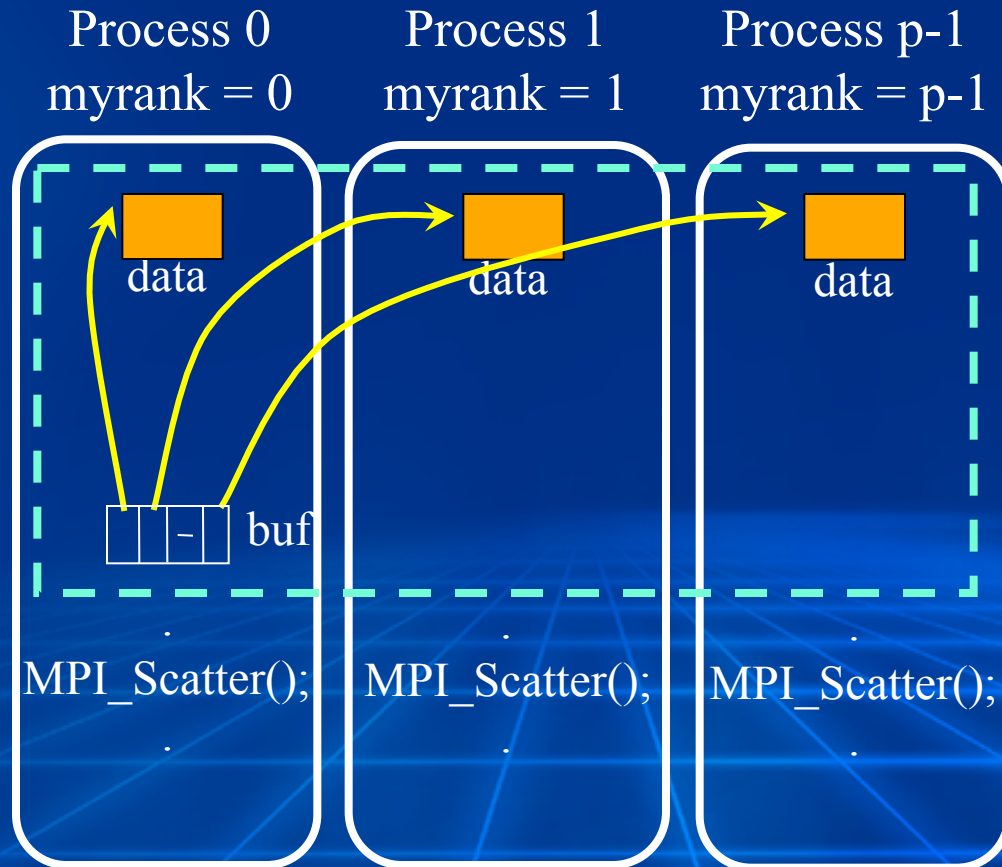


# MPI\_Scatter函数详解

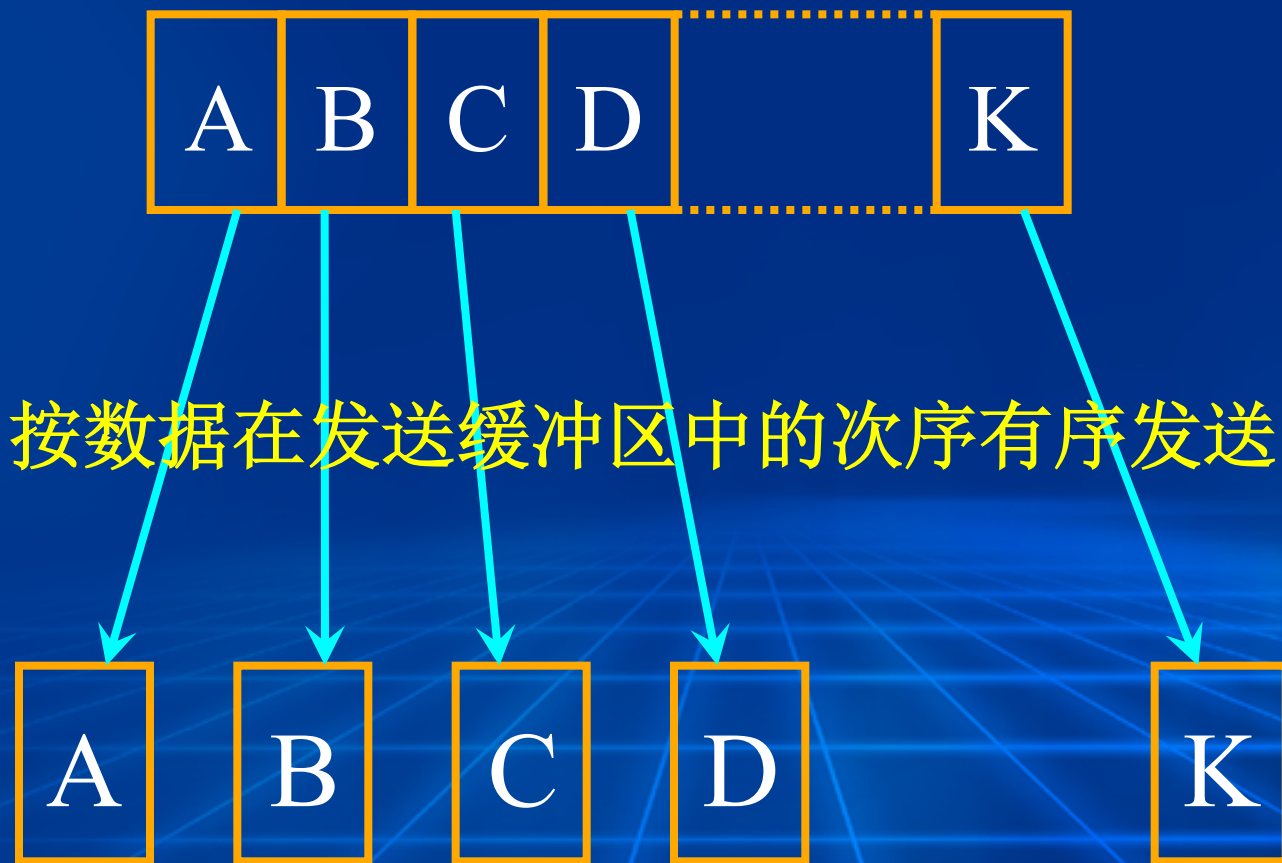
- 根进程中的发送数据元素个数sendcount和发送数据类型sendtype必须和所有进程的接收数据元素个数recvcount和接收数据类型recvtype相同
- 对于所有非根进程，发送消息缓冲区被忽略
- 根进程发送元素个数指的是发送给每一个进程的数据元素的个数而不是总的数据个数
- 此调用中的所有参数对根进程来说都是有意义的而对于其他进程来说只有recvbuf、recvcount、recvtype、root和comm是有意义的参数
- root和comm在所有进程中都必须是一致的

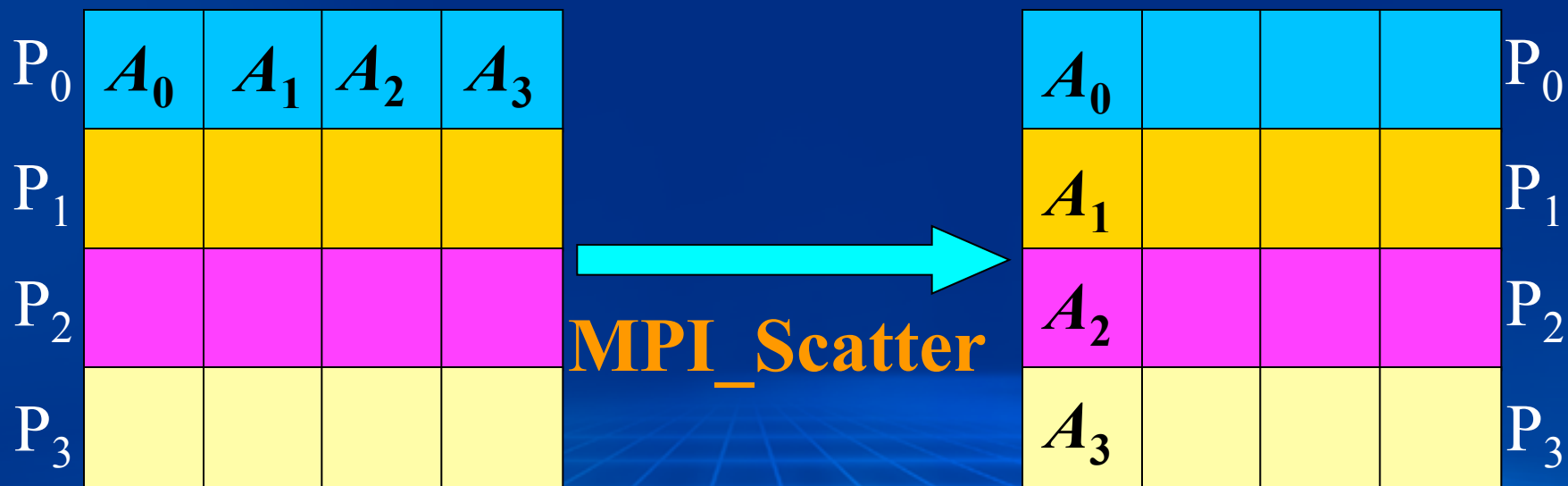


# MPI\_Scatter 图示



# MPI\_Scatter 图示





# 实例：MPI\_Scatter

```
MPI_Comm comm;
```

```
int size, *sendbuf;
```

```
int root, rbuf[100];
```

```
.....
```

```
MPI_Comm_size(comm, &size);
```

```
sendbuf = (int *)malloc(size * 100 * sizeof(int));
```

```
.....
```

```
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf,  
            100, MPI_INT, root, comm);
```





# MPI\_Gatherv 函数

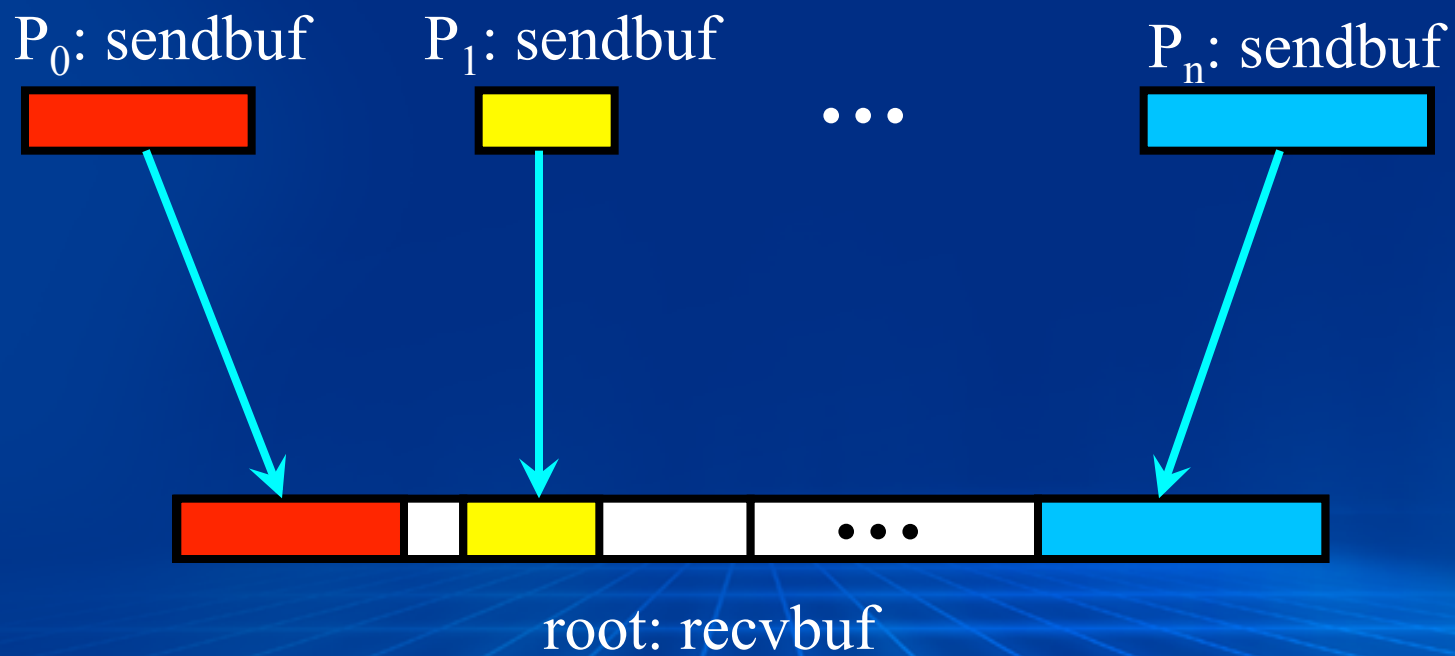
```
int MPI_Gatherv (  
    void*          sendbuf          /* in */  
    int            sendcount        /* in */  
    MPI_Datatype   sendtype         /* in */  
    void*          recvbuf          /* out */  
    int            recvcnts[]       /* in */  
    int            displs[]         /* in */  
    MPI_Datatype   recvtype         /* in */  
    int            root              /* in */  
    MPI_Comm       comm             /* in */)
```



# MPI\_Gatherv函数详解

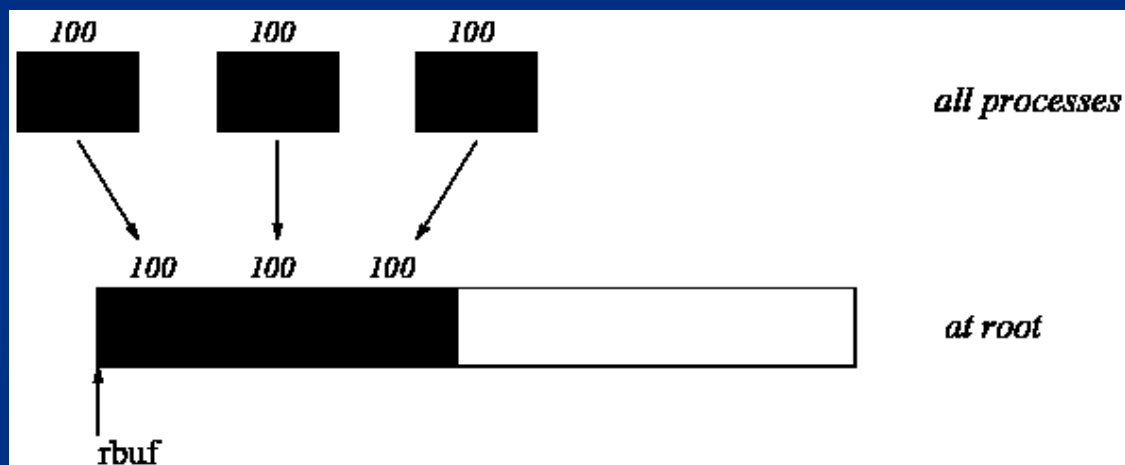
- 收集不同长度数据块。从不同进程接收不同数量的数据，为此接收数据元素的个数`recvcunts`是一个数组，用于指明从不同的进程接收的数据元素的个数
- 根从每一个进程接收的数据元素的个数可以不同，但是发送和接收的个数必须一致
- 除此之外它还为每一个接收消息在接收缓冲区的位置提供了一个位置偏移`displs`数组用户可以将接收的数据存放到根进程消息缓冲区的任意位置
- `MPI_Gatherv`明确指出了从不同的进程接收数据元素的个数以及这些数据在Root接收缓冲区存放的起始位置



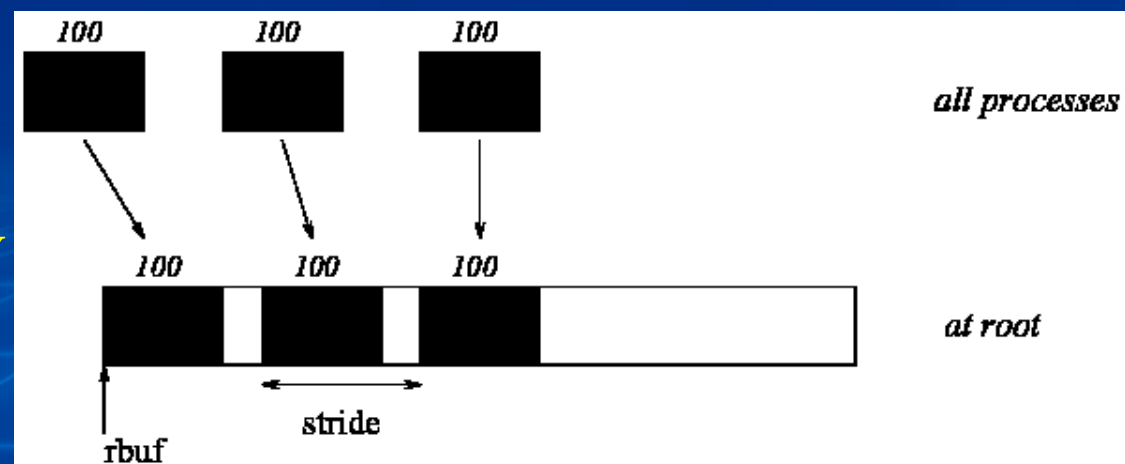


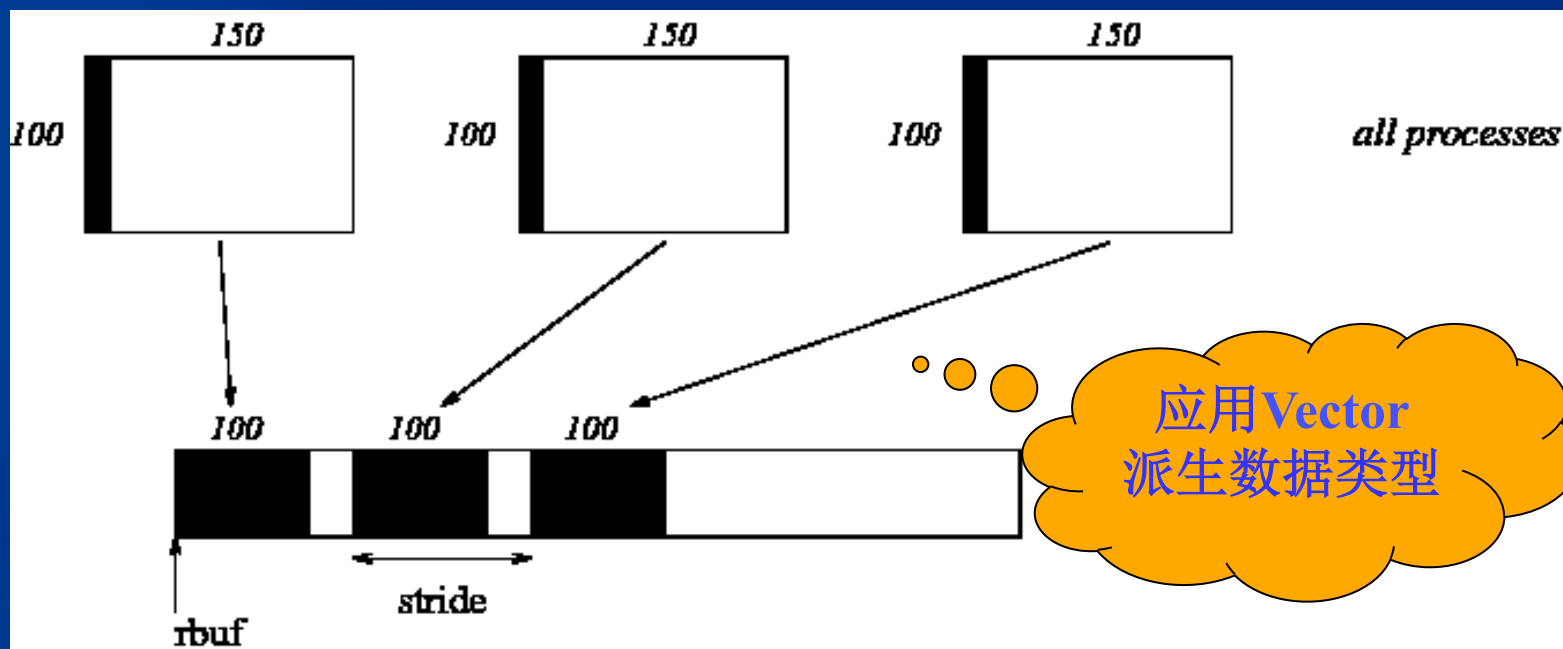
# MPI\_Gather vs MPI\_Gatherv

Gather



Gatherv





# 实例: MPI\_Gatherv

```
int    size, root, s_data[100], *rbuf, *displs, *rcount;
int    stride = 120;
...
MPI_Comm_size( comm, &size );

rbuf = (int *) malloc( size * stride * sizeof(int));
displs = (int *) malloc( size * sizeof(int)); /* 申请缓冲区
rcounts = (int *) malloc( size * sizeof(int)); */
for (i = 0; i < size; i ++){
    displs[i] = i * stride; rcounts[i] = 100;}

MPI_Gatherv( s_data, 100, MPI_INT, rbuf,
rcounts, displs, MPI_INT, root, comm);
```

...





# MPI\_Scatterv 函数

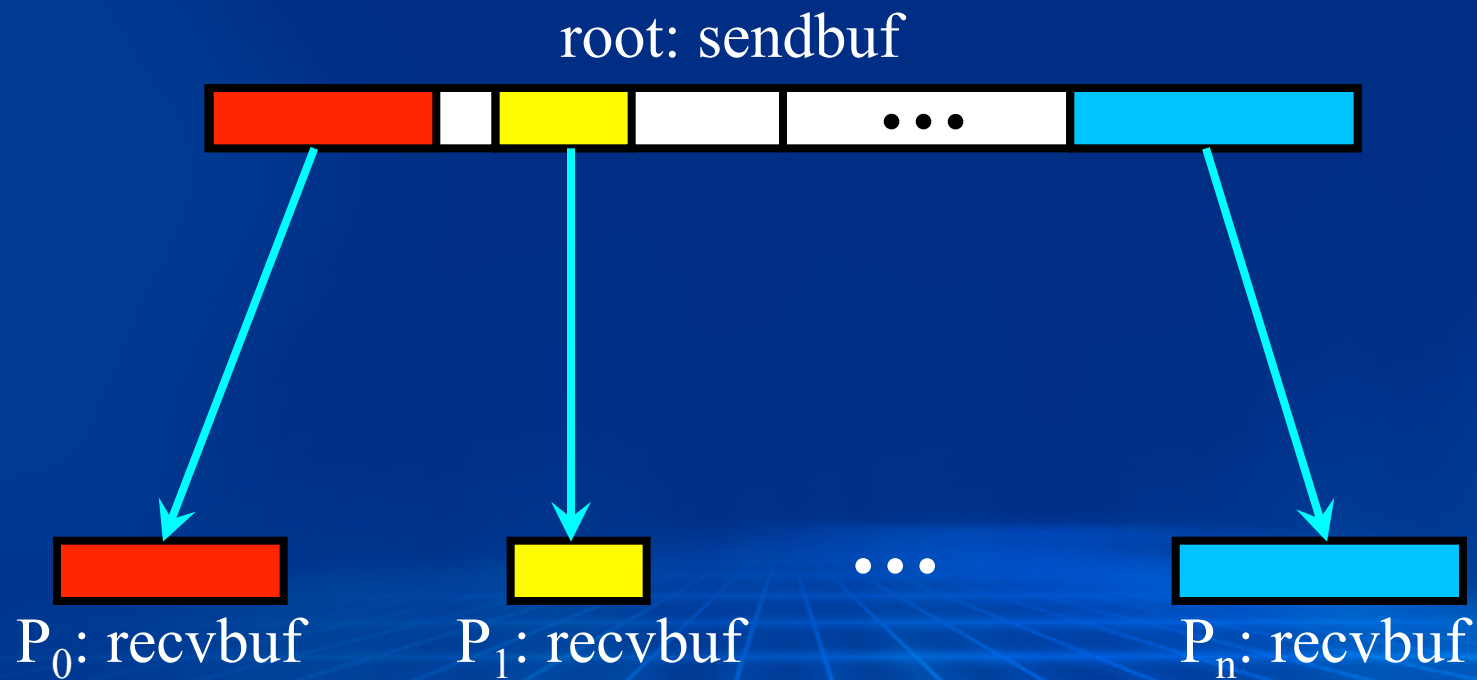
```
int MPI_Scatterv (  
    void*          sendbuf      /* in */  
    int            sendcounts[] /* in */  
    int            displs[]     /* in */  
  
    MPI_Datatype   sendtype     /* in */  
    void*          recvbuf      /* out */  
    int            recvcnt      /* in */  
    MPI_Datatype   recvtype     /* in */  
    int            root         /* in */  
    MPI_Comm       comm         /* in */) 
```

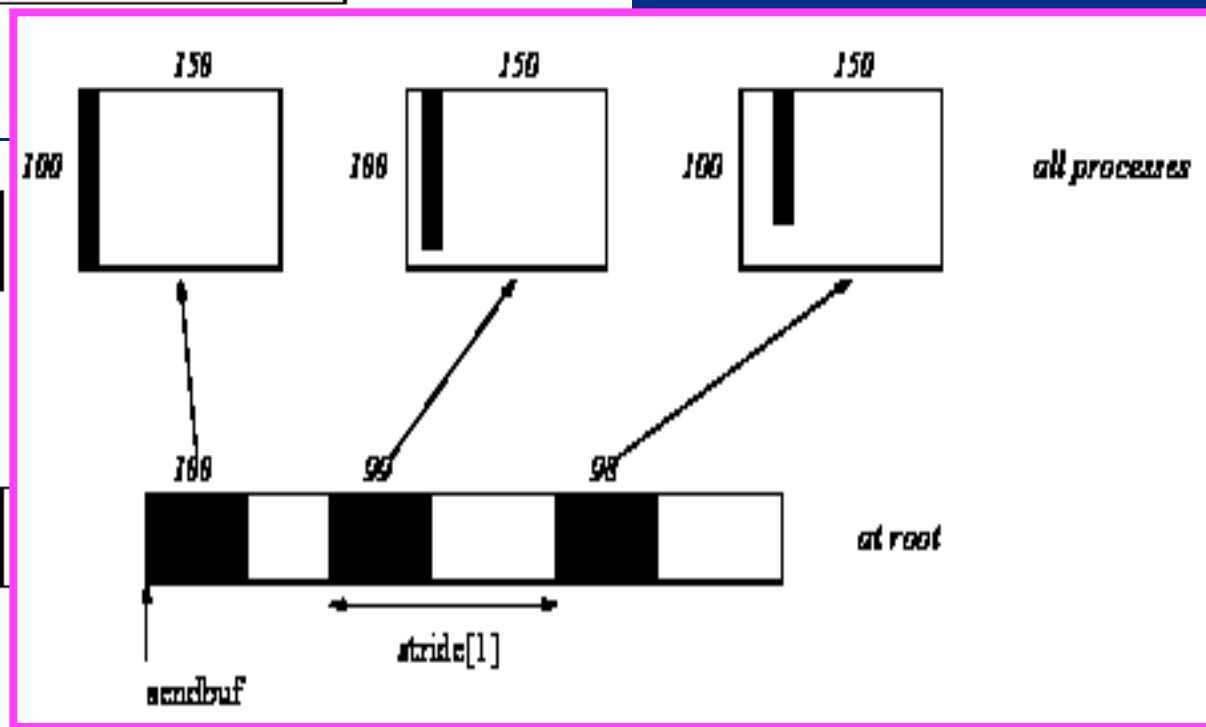
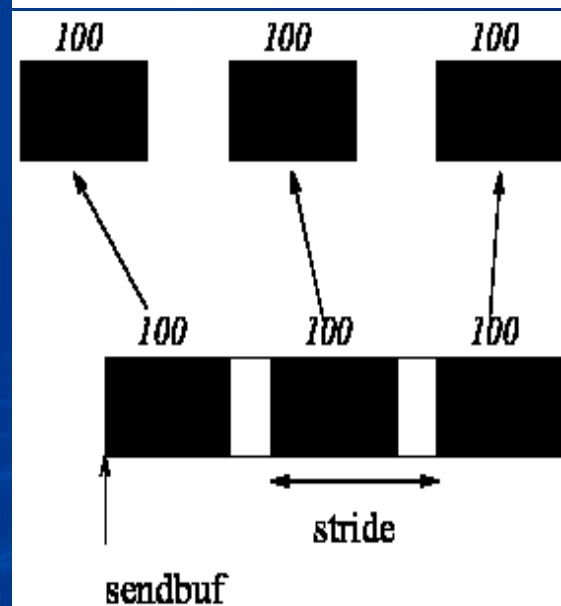
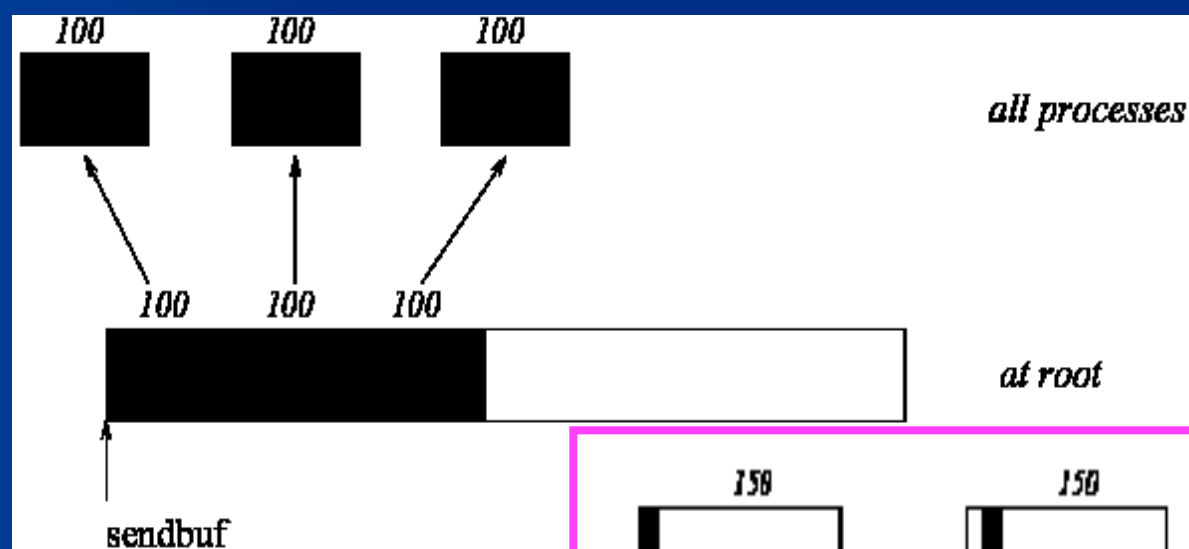


# MPI\_Scatterv 函数详解

- MPI\_Scatterv 允许 Root 向各个进程发送个数不等的数。因此要求 **sendcounts** 是一个数组同时还提供一个新的参数 **displs** 指明根进程发往其它不同进程数据在根发送缓冲区中的偏移位置
- 根进程中 **sendcount[i]** 和 **sendtype** 的类型必须和进程 **i** 的 **recvcount** 和 **recvtype** 的类型相同这就意味着在每个进程和根进程之间发送的数据量必须和接收的数据量相等
- 对于所有非根进程，发送消息缓冲区被忽略。
- 此调用中的所有参数对根进程来说都是很重要的而对于其他进程来说只有 **recvbuf**、**recvcount**、**recvtype**、**root** 和 **comm** 是有意义的
- 参数 **root** 和 **comm** 在所有进程中都必须是一致的







# 实例：MPI\_Scatterv

```
int i, size, root, *sendbuf, rbuf[100], *displs, *scounts;  
int stride = 120;
```

```
.....  
MPI_Comm_size(comm, &size);
```

```
sendbuf = (int *)malloc(size * stride * sizeof(int));  
displs = (int *)malloc(size * sizeof(int)); /* 申请缓冲区  
scounts = (int *)malloc(size * sizeof(int));*/
```

```
for (i=0; i<size; ++i) {  
    displs[i] = i*stride; scounts[i] = 100;}
```

```
MPI_Scatterv(sendbuf, scounts, displs,  
MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



# MPI\_Allgather 函数

```
int MPI_Allgather (  
    void*          sendbuf    /* in */  
    int            sendcount  /* in */  
    MPI_Datatype   sendtype   /* in */  
    void*          recvbuf    /* out */  
    int            recvcount  /* in */  
    MPI_Datatype   recvtype   /* in */  
    MPI_Comm       comm       /* in */) 
```





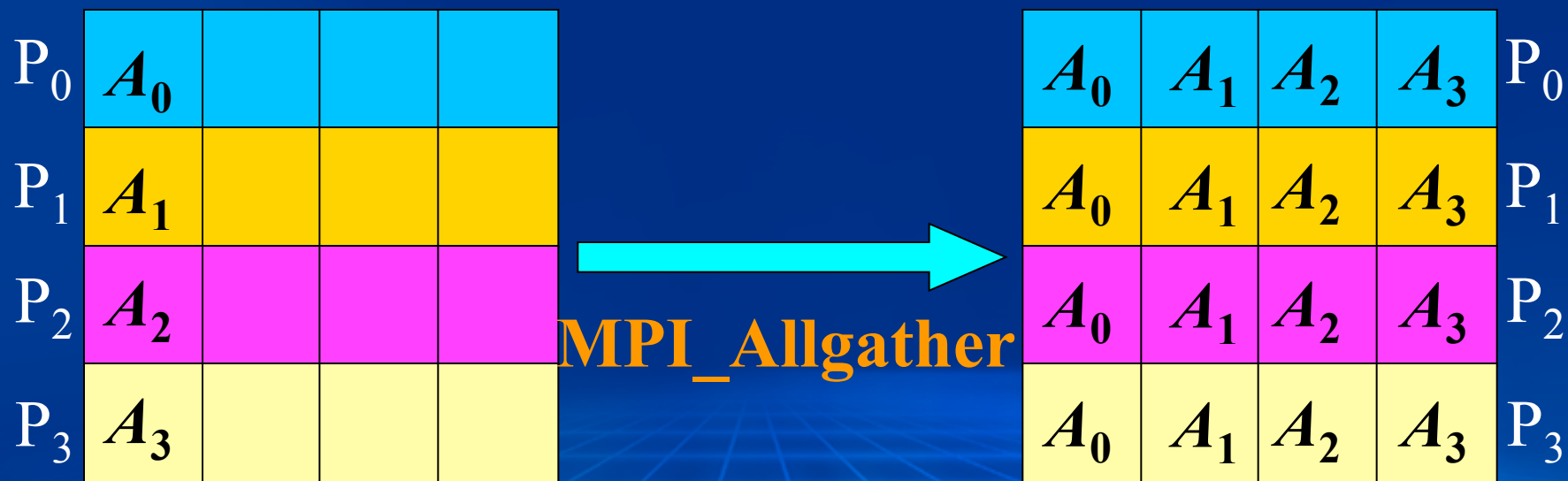
# MPI\_Allgather函数详解

- **MPI\_Gather**将数据收集到根进程, **MPI\_Allgather**相当于每一个进程都作为Root执行了一次**MPI\_Gather**调用, 即每一个进程都收集到了其它所有进程的数据
- 从参数上看**MPI\_Allgather**和**MPI\_Gather**完全相同, 只不过在执行效果上对于**MPI\_Gather**执行结束后, 只有Root进程的接收缓冲区有意义;
- **MPI\_Allgather**调用结束后所有进程的接收缓冲区都有意义。
- 它们接收缓冲区的内容是相同的。



# MPI\_Allgather函数图示





# MPI\_Allgatherv 函数

```
int MPI_Allgatherv(  
    void*          sendbuf          /* in */,  
    int            sendcount        /* in */,  
    MPI_Datatype    sendtype        /* in */,  
    void*          recvbuf          /* out */,  
    int            recvcnts[]       /* in */,  
    int            displs[]         /* in */,  
    MPI_Datatype    recvtype        /* in */,  
    MPI_Comm        comm            /* in */)
```

Root ONLY



中国科学院计算技术研究所  
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

# MPI\_Allgatherv函数详解

- **MPI\_Allgatherv**也是所有的进程都将接收结果，而不是只有根进程接收结果；
- 从每个进程发送的第j块数据将被每个进程接收，然后存放在各个进程接收消息缓冲区**recvbuf**的第j块，进程j的**sendcount**和**sendtype**的类型必须和其他所有进程的**recvcounts[j]**和**recvtype**相同



# MPI\_Allgather vs MPI\_Allgatherv

Allgather, NPROCS=3



Allgatherv, NPROCS=3





# 实例: MPI\_Allgather

```
MPI_Comm comm;  
int size, sendarray[100], *rbuf;  
.....
```

```
MPI_Comm_size(comm, &size);  
rbuf = (int *)malloc(size * 100 * sizeof(int));
```

```
MPI_Allgather(sendarray, 100, MPI_INT,  
              rbuf, 100, MPI_INT, comm);
```



# 实例: MPI\_Allgatherv

```
int size, sendarray[100], *rbuf, *displs, i, *rcounts;
int stride = 120;
.....
MPI_Comm_size(comm, &size);
rbuf = (int *) malloc(size * stride * sizeof(int));
displs = (int *) malloc(size * sizeof(int));
rcounts = (int *) malloc(size * sizeof(int)); /* 申请缓冲区 */
for (i=0; i<size; ++i) {
    displs[i] = i * stride; rcounts[i] = 100;}

MPI_Allgatherv(sendarray, 100, MPI_INT,
               rbuf, rcounts, displs, MPI_INT, comm);
```



# MPI全发散收集函数

- 每个进程散发自己的一个数据块, 并且收集拼装所有进程散发过来的数据块。称该操作为数据的“全散发收集”。
- 它既可以被认为是数据全收集的扩展, 也可以被认为是数据散发的扩展



# MPI\_Alltoall函数详解

- MPI\_Alltoall是组内进程之间完全的消息交换，每一个进程都向其它所有的进程发送消息，同时每一个进程都从其它所有的进程接收消息。
  - MPI\_Allgather每个进程散发一个相同的消息给所有的进程
  - MPI\_Alltoall散发给不同进程的消息是不同的。因此它的发送缓冲区也是一个数组
- 调用MPI\_Alltoall相当于每个进程依次将它的发送缓冲区的第i块数据发送给第i个进程，同时每个进程又都依次从第j个进程接收数据放到各自接收缓冲区的第j块数据区的位置。

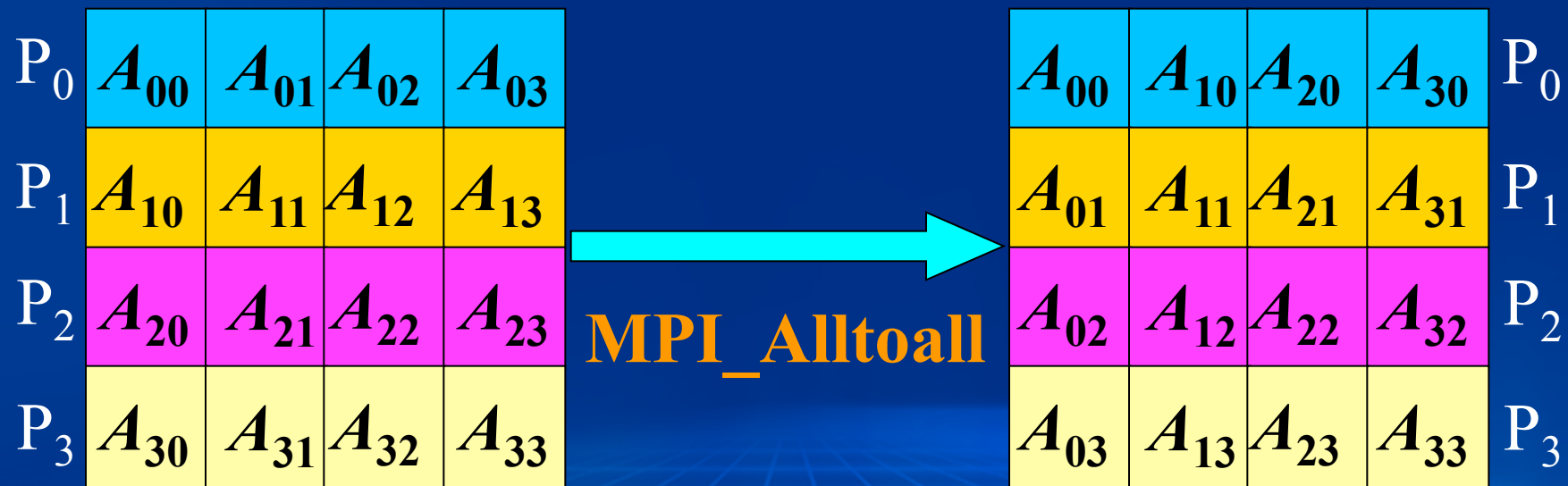


# MPI\_Alltoall函数详解

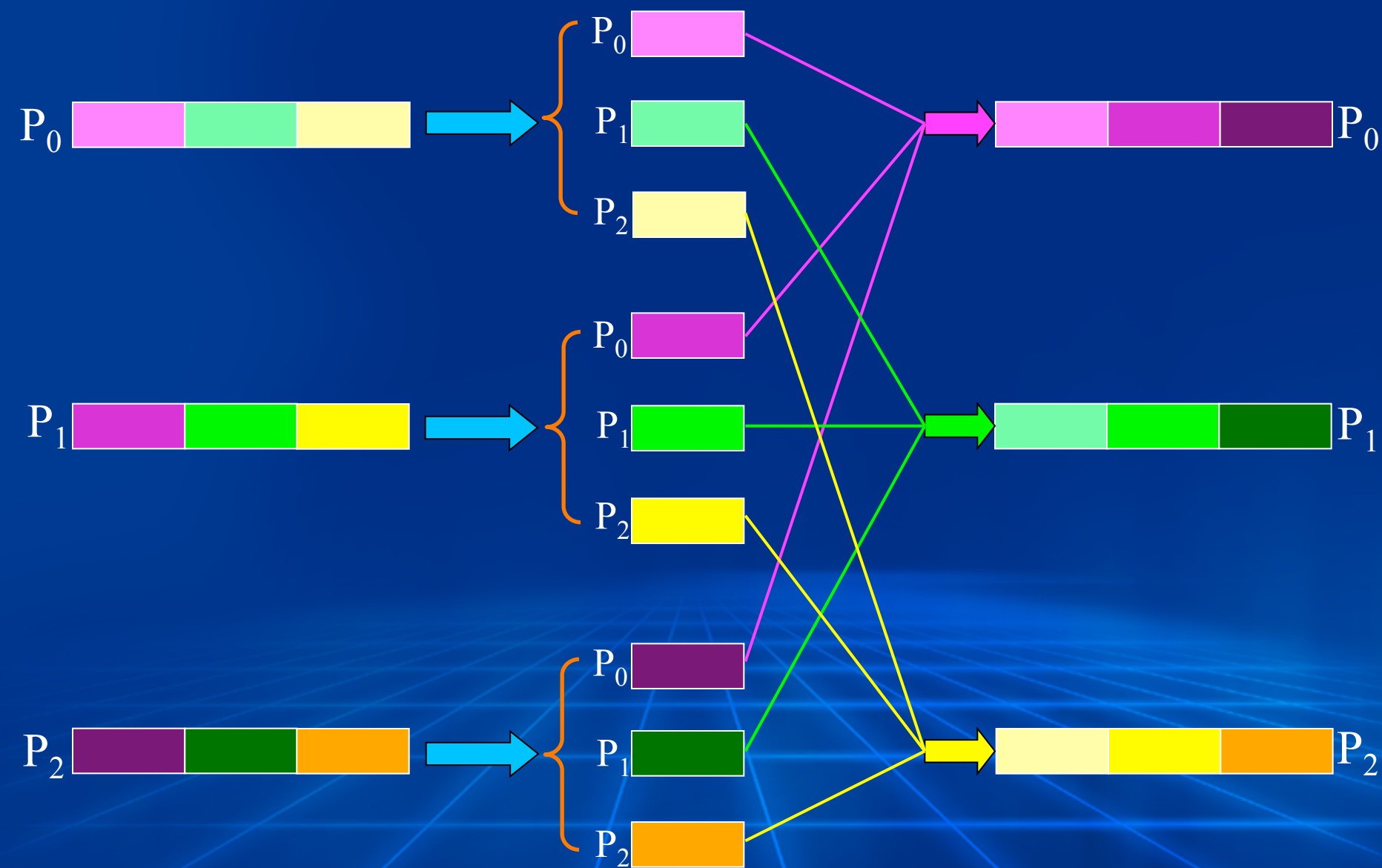
- MPI\_Alltoall的每个进程可以向每个接收者发送数目不同的数据，第i个进程发送的第j块数据将被第j个进程接收，并存放在其接收消息缓冲区recvbuf的第i块。
- 每个进程的sendcount和sendtype的类型必须和所有其他进程的recvcount和recvtype相同这就意味着在每个进程和根进程之间发送的数据量必须和接收的数据量相等。











# MPI\_Alltoall函数

```
int MPI_Alltoall(  
    void*          sendbuf      /* in */,  
    int            sendcount    /* in */,  
    MPI_Datatype    sendtype    /* in */,  
    void*          recvbuf      /* out */,  
    int            recvcount    /* in */,  
    MPI_Datatype    recvtype    /* in */,  
    MPI_Comm        comm        /* in */)
```



# MPI\_Alltoallv 函数

```
int MPI_Alltoallv(  
    void*          sendbuf          /* in */,  
    int            sendcounts[]     /* in */,  
    int            sdispls[]        /* in */,  
    MPI_Datatype   sendtype         /* in */,  
    void*          recvbuf          /* out */,  
    int            recvcounts[]     /* in */,  
    int            rdispls[]        /* in */,  
    MPI_Datatype   recvtype         /* in */,  
    MPI_Comm       comm             /* in */)
```

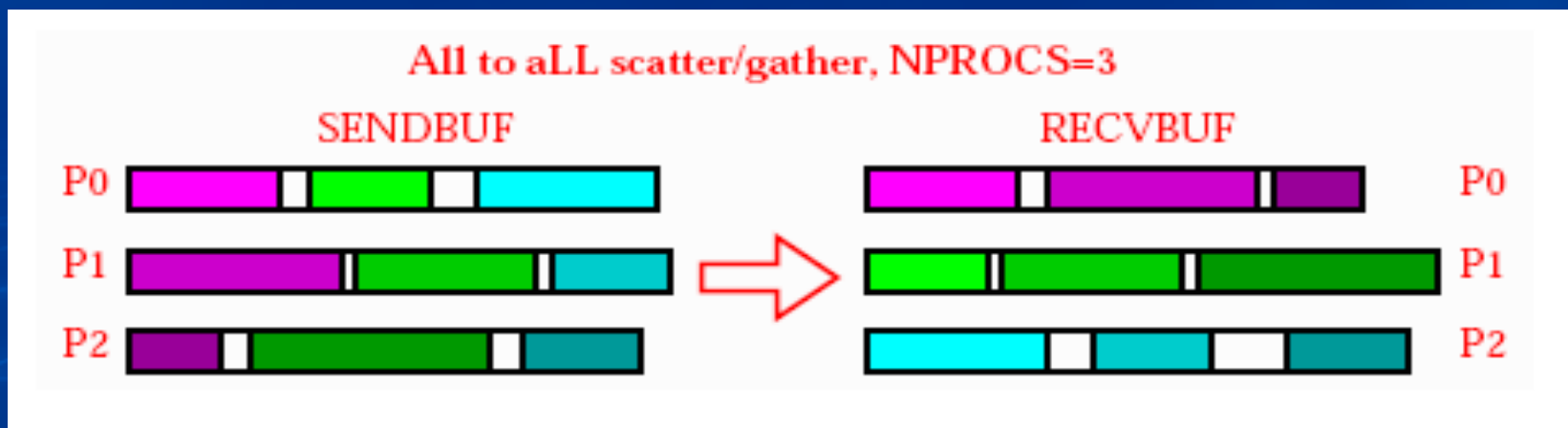
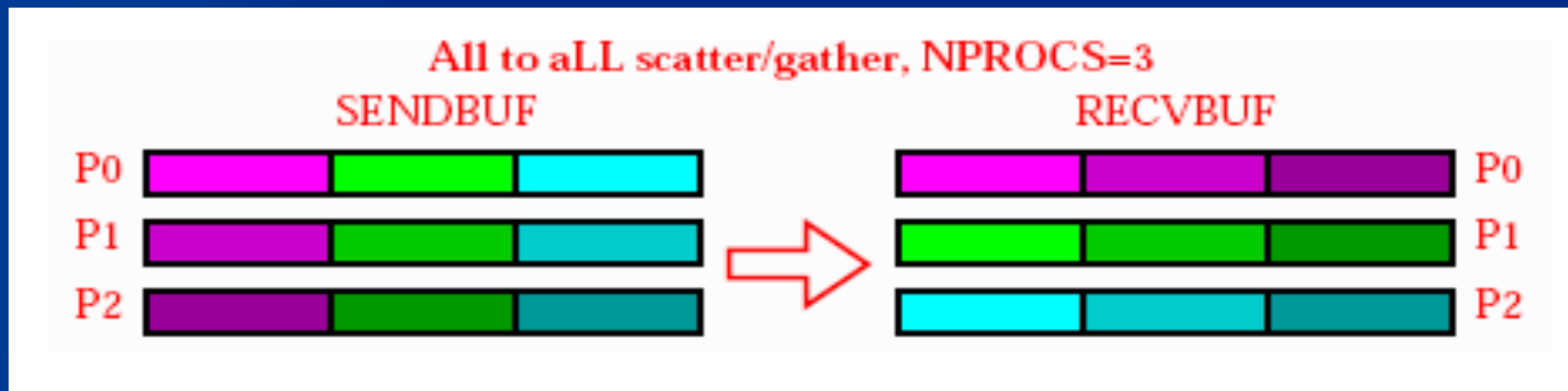


# MPI\_Alltoallv 函数详解

- 正如MPI\_Allgatherv 和MPI\_Allgather 的关系一样MPI\_Alltoallv在MPI\_Alltoall的基础上进一步增加了灵活性。它可以由sdispls指定待发送数据的位置，在接收方则由rdispls指定接收的数据存放在缓冲区的偏移量
- 所有参数对每个进程都是有意义的，并且所有进程中的comm值必须一致
- MPI\_Alltoall和MPI\_Alltoallv可以实现n次独立的点对点通信但也有限制：1)所有数据必须是同一类型；2)所有的消息必须按顺序进行散发和收集



# MPI\_Alltoall vs MPI\_Alltoallv



# 组归约函数MPI\_Allreduce

```
int MPI_AllReduce (  
    void*          openand    /* in */,  
    void*          result     /* in */,  
    int            count      /* in */,  
    MPI_Datatype   datatype   /* in */,  
    MPI_Op         operator   /*out*/,  
    MPI_Comm       comm       /* in */)
```

- MPI\_Allreduce比MPI\_Reduce少一个root参数, 其它参数及含义与后者一样





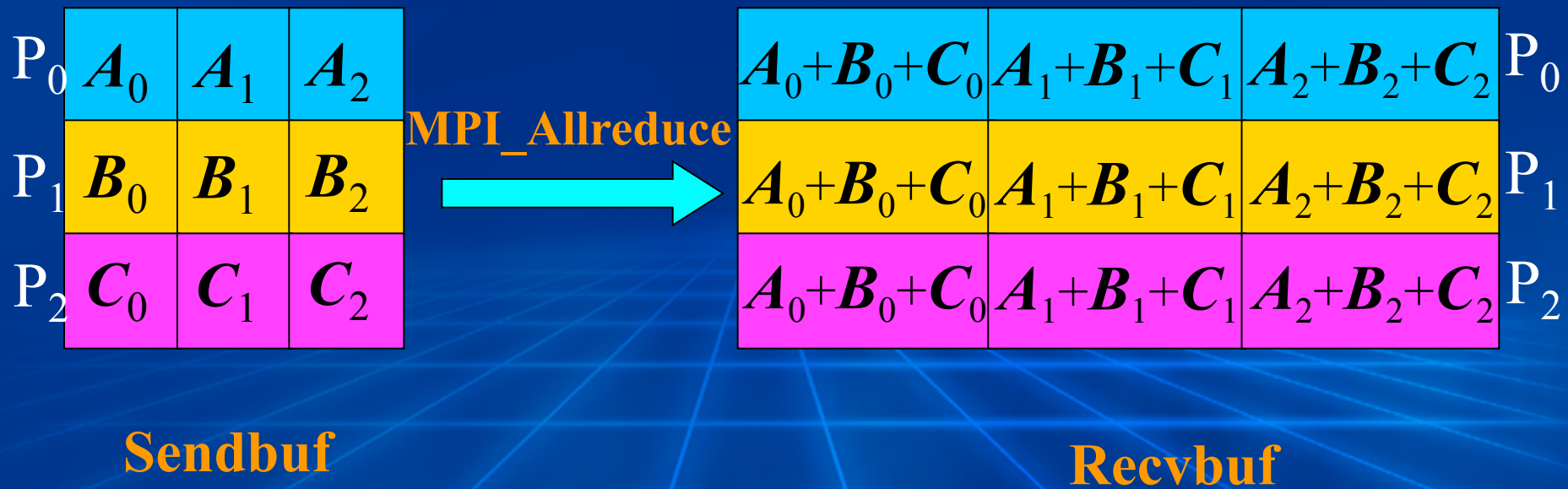
# MPI\_Allreduce函数详解

- **MPI\_Allreduce**相当于组中每一个进程都作为**root**分别进行了一次归约操作。
- **MPI\_Allreduce** 相当于在**MPI\_Reduce** 后马上再将结果进行一次广播**MPI\_Bcast**
- 归约的结果不只是某一个进程所有而是所有的进程都所有。
- 它在某种程度上和组收集与收集的关系很相似



# MPI\_Allreduce函数图示

MPI\_Allreduce: np = 3; count = 3; Op = MPI\_SUM



# 归约发散函数

## MPI\_Reduce\_scatter

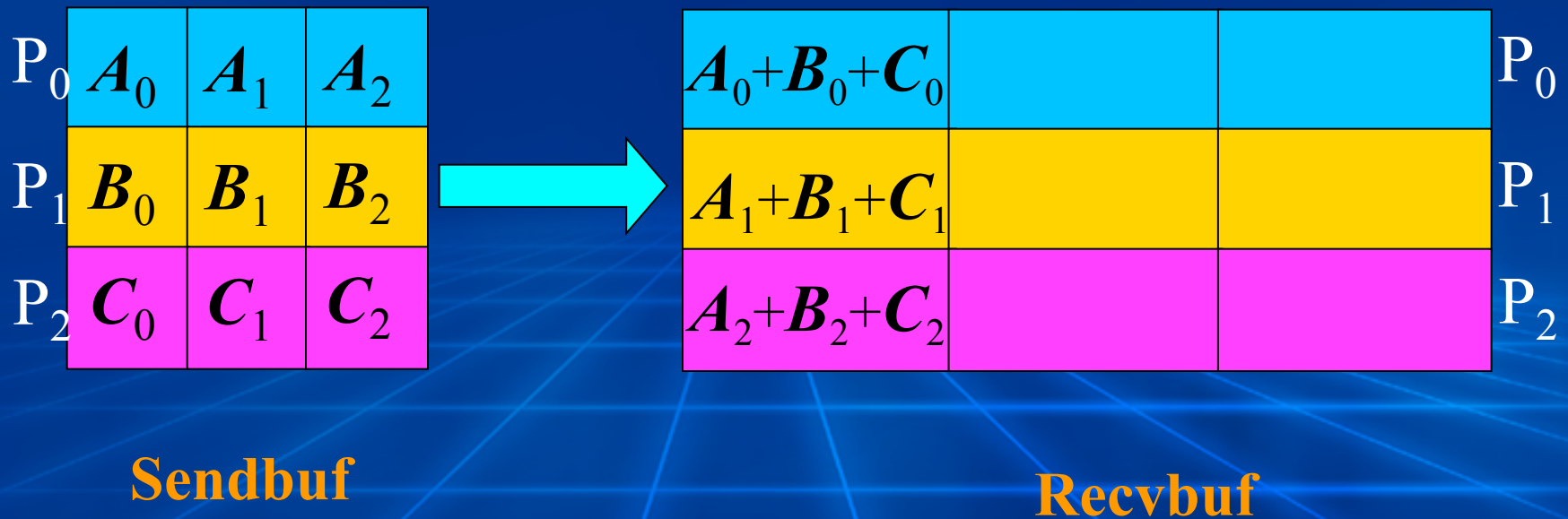
```
int MPI_Reduce_scatter (  
    void*          sendbuf      /* in */,  
    void*          recvbuf      /* in */,  
    int*           recvcounts   /* in */,  
    MPI_Datatype   datatype     /* in */,  
    MPI_Op         operator     /* out */,  
    MPI_Comm       comm         /* in */)

```



# MPI\_Reduce\_scatter函数图示

**MPI\_Reduce\_scatter: np = 3; count = 3;  
Op = MPI\_SUM**



# MPI\_Reduce\_scatter函数详解

- MPI Reduce scatter可以认为是MPI对每个归约操作的变形，它将归约结果分散到组内的所有进程中去而不是仅仅归约到root进程
- MPI Reduce scatter对由sendbuf、count和datatype定义的发送缓冲区数组的元素逐个进行归约操作，发送缓冲区数组的长度count= recvcount[i]
- 然后再对归约结果进行散发操作，散发给第*i*个进程的数据块长度为recvcounts(*i*). 其余参数的含义与MPI\_Reduce一样.



# 前缀扫描函数MPI\_Scan

```
int MPI_Scan (  
    void*                openand /* in */,  
    void*                result   /* in */,  
    int                  count    /* in */,  
    MPI_Datatype          datatype /* in */,  
    MPI_Op               operator /* out */,  
    MPI_Comm             comm     /* in */)
```





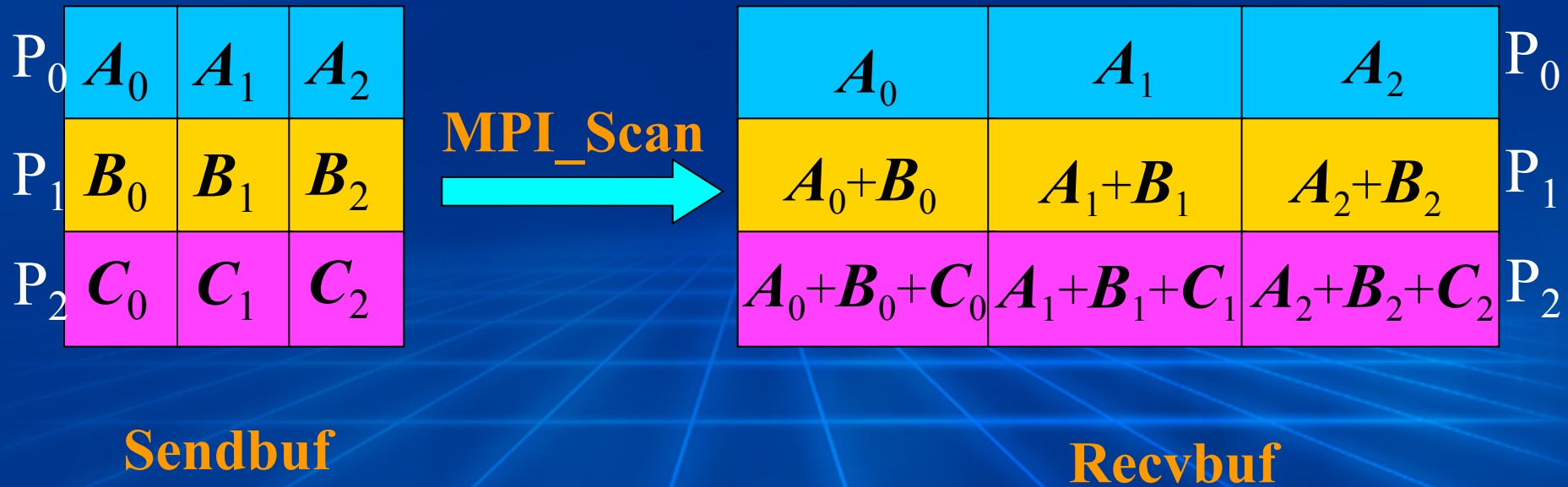
# MPI\_Scan函数详解

- **MPI\_Scan**前缀扫描, 或前缀归约, 与归约**MPI\_Reduce**操作类似, 但各处理器依次得到部分归约结果。
- 确切地说, 操作结束后第 $i$ 个处理器的recvbuf中将包含前 $i$ 个处理器的归约运算结果。
- 各参数的含义与**MPI\_Allreduce**基本相同



# MPI\_Scan函数图示

MPI\_Scan: np = 3; count = 3; Op = MPI\_SUM



# 用户自定义的归约运算

```
int MPI_Op_create (  
    MPI_User_function    *func    /* in */,  
    int                   commute /* in */,  
    MPI_Op                *op      /* out */) 
```

- MPI\_Op\_create创建一个新的运算。
- **func**:用户提供用于完成该运算的外部函数名
- **commute**:用来指明所定义的运算是否满足交换律(commute=true是可交换的)
- **op**:返回所创建的运算指针



# 用户自定义的操作函数

- 用户自定义的操作必须是可以结合的。如果 `commute=true` 则此操作同时也是可交换的，否则此操作不满足交换律
- `function` 函数必须具备四个参数: `invec`, `inoutvec`, `len` 和 `datatype`

```
void user_function(  
    void                                *invec,  
    void                                *inoutvec,  
    int                                 *len,  
    MPI_Datatype                       *datatype)
```



# 用户自定义的操作函数

- 进入函数function时, invec和inoutvec包含参与运算的操作数(operand); 函数返回时 inoutvec中应该包含运算的结果; len给出 invec和inoutvec中包含的元素个数(相当于归约函数中的count); datatype 给出操作数的数据类型(即归约函数中的datatype)
- 直观地, 函数function必须完成如下操作:

```
for ( i = 0; i < len; i++)
```

```
    inoutvec[i] = invec[i] op inoutvec[i];
```





# 撤销自定义操作

- 当一个用户定义的运算不再需要时, 可以调用MPI\_Op\_free 将其释放, 以便释放它所占用的系统资源.

```
int MPI_Op_free (  
    MPI_Op      *op      /* in */) 
```





# MPI 集合通信函数

类型	函数	功能
数据移动	<b>MPI_Bcast</b>	一到多，数据广播
	<b>MPI_Gather</b>	多到一，数据汇合
	<b>MPI_Gatherv</b>	<b>MPI_Gather</b> 的一般形式
	<b>MPI_Allgather</b>	<b>MPI_Gather</b> 的一般形式
	<b>MPI_Allgatherv</b>	<b>MPI_Allgather</b> 的一般形式
	<b>MPI_Scatter</b>	一到多，数据分散
	<b>MPI_Scatterv</b>	<b>MPI_Scatter</b> 的一般形式
	<b>MPI_Alltoall</b>	多到多，置换数据
	<b>MPI_Alltoallv</b>	<b>MPI_Alltoall</b> 的一般形式
数据收集	<b>MPI_Reduce</b>	多到一，数据归约
	<b>MPI_Allreduce</b>	上者的一般形式，结果在所有进程
	<b>MPI_Reduce_scatter</b>	结果scatter到各个进程
	<b>MPI_Scan</b>	前缀操作
同步	<b>MPI_Barrier</b>	同步操作



- All表示最后的结果存放到所有的进程中，MPI\_Allgather、MPI\_Alltoall、MPI\_Allreduce
- V:Vector,操作以及被操作的数据对象更加灵活，MPI\_Gather(v)、MPI\_Scatter(v)、MPI\_Allgather(v)、MPI\_Alltoall(v)



# 上机实习1:

- 计算向量点积  $c = \sum_{i=0}^{n-1} a_i \cdot b_i \quad n > 100000$
- 要求：计算出加速比
- 建议：N能够被节点数np整除



# 上机实习2:矩阵-向量乘法

设  $A = (a_{ij})_{n \times n}$   $X, Y$  为  $n$  维向量,  $Y = A \times X$

$$Ax = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \cdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n \end{pmatrix}$$

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$



# 矩阵-向量乘法代码1

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define N 1000000

void main(int argc, char** argv) {
    int myrank, i, j, size, row;          /* 定义变量 */
    int a[N][N], b[N], c[N], (*tmp_a)[N], *local_sum;

    MPI_Init(&argc, &argv); /* 初始化MPI */
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    row = N / size;
    tmp_a = (int*) malloc(N * row * sizeof(int));
    local_sum = (int*) malloc(row * sizeof(int));
```



# 矩阵-向量乘法代码2

```
if (myrank == 0){  
    for(i = 0; i < N; i++) { b[i] = i+1;  
        for(j = 0; j < N; j++) a[i][j] = i+1;}}
```

```
MPI_Bcast(b, N, MPI_INT, 0, MPI_COMM_WORLD);  
MPI_scatter(a, N*row, MPI_INT, tmp_a, N*row,  
            MPI_INT, 0, MPI_COMM_WORLD);
```

```
for(i = 0; i <= row; i++){  
    local_sum[i] = 0;  
    for(j = 0; j < N; j++) local_sum[i] += tmp_a[i][j]*b[j]; }
```

```
MPI_Gather(&local_sum[0], row, MPI_INT, c, row,  
/*进程0收集*/ MPI_INT, MPI_COMM_WORLD);
```

```
if (myid == 0){ /*打印结果*/}  
MPI_Finalize();  
}
```

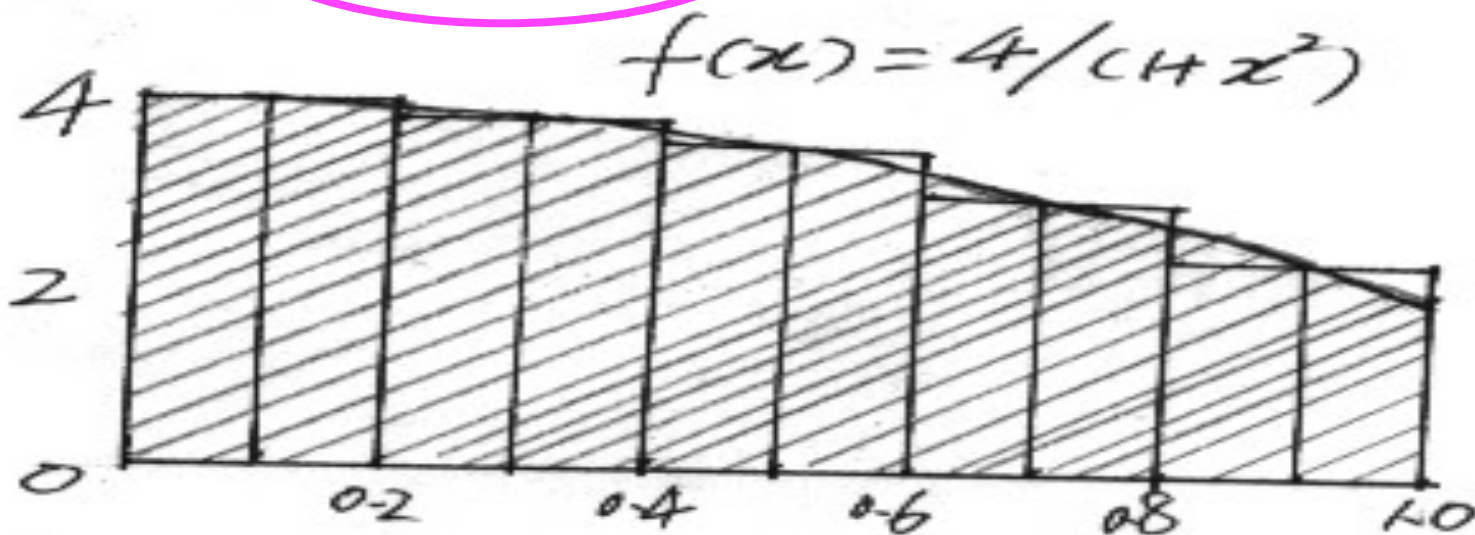




# 计算圆周率 $\pi$

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x) \Big|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

令函数  $f(x) = 4/(1+x^2)$ , 则:  $\int_0^1 f(x) dx = \pi$



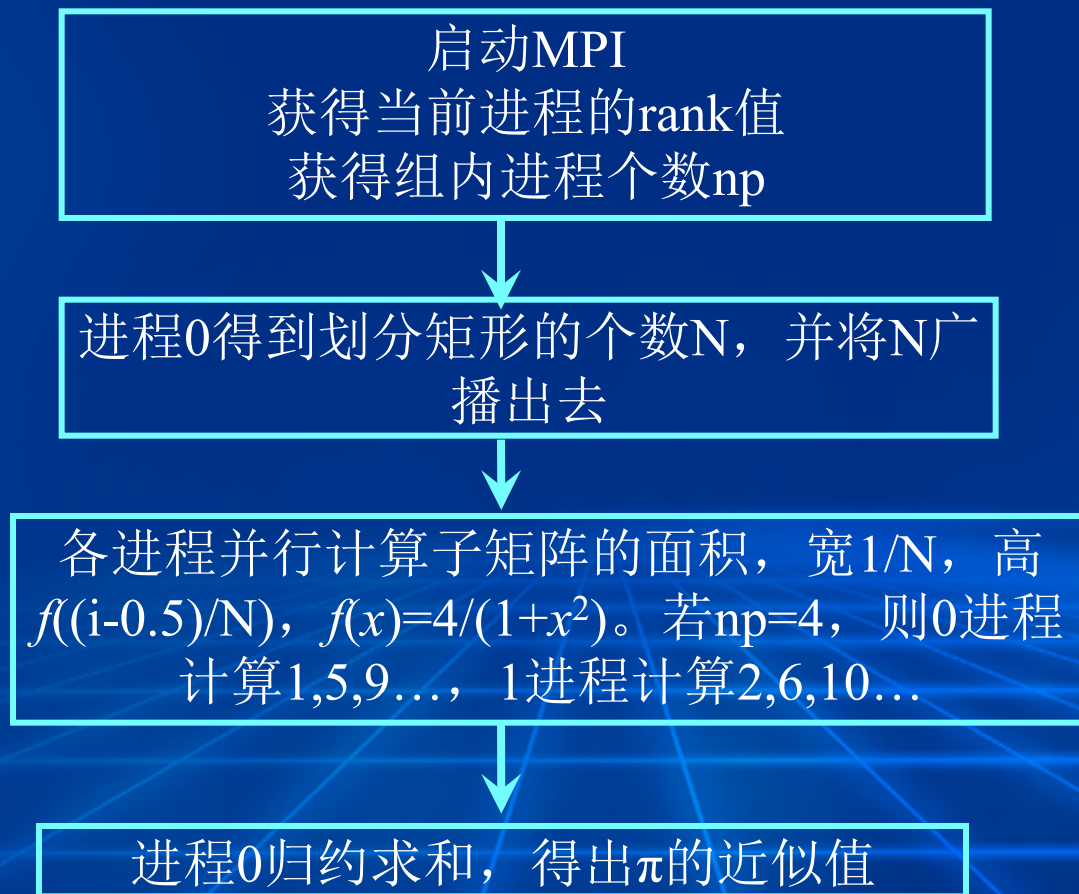
# 计算圆周率 $\pi$

- 计算 $f(x)$ 图像下面从0到1之间的面积即为 $\pi$ 的值。
- 设将0到1之间划分为 $N$ 个矩形，则：

$$\pi \approx \sum_{i=1}^N f\left(\frac{2 \times i - 1}{2 \times N}\right) \times \frac{1}{N} = \frac{1}{N} \times f\left(\frac{i - 0.5}{N}\right)$$



# 计算圆周率 $\pi$ 的计算流程



# 计算圆周率代码1

```
#define PI25DT 3.141592653589793238462643
```

```
/*计算 $f(x) = 4.0 / (1 + x * x)$ */
```

```
double f(double a){ return (4.0 / (1.0 + a*a)); }
```

```
void main(int argc, char** argv){  
    int n, myid, np, i;          /* 定义变量 */  
    double mypi = 0, pi, x, s_time, e_time;
```

```
    MPI_Init(&argc, &argv);    /* 初始化MPI */
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &np);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
    if (myid == 0){ /* 进程0获得划分矩阵的个数n */
```

```
        scanf("%d", &n); s_time = MPI_Wtime(); }
```



# 计算圆周率代码2

/\* 进程0将n进行广播 \*/

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

```
for(i = myid+1; i <= n; i += np){  
    x = ((double)i - 0.5) / (double)n; /*各进程并  
    mypi += f(x) / (double)n; }        行计算*/
```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,  
/*进程0归约求和*/ MPI_SUM, 0,  
MPI_COMM_WORLD);
```

```
if (myid == 0){ printf("pi is %.16f, Error is %e\n",  
/*打印结果*/ pi, fabs(pi - PI25DT) );  
printf("Wall time = %f\n", MPI_Wtime() - s_time);}
```

```
MPI_Finalize();  
}
```



# THANKS



中国科学院计算技术研究所  
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES