并行计算

十五、分布存储系统并行编程

分布存储系统并行编程

- 14.1 基于消息传递的并行编程
- 14.2 MPI并行编程
  - 6个基本函数组成的MPI子集
  - MPI消息
  - 点对点通信
  - 群集通信

SPMD和MPMD

- SPMD
  - 各个进程是同构的，多个进程对不同的数据执行相同的代码(一般是数据并行)
  - 常对应并行循环，数据并行结构，单代码
- MPMD
  - 各个进程是异构的， 多个进程执行不同的代码（一般是任务并行，或功能并行）
  - 常对应并行块，多代码
  - 要为有1000个处理器的计算机编写一个完全异构的并行程序是很困难的

SPMD和MPMD

SPMD程序的构造方法

用单代码方法说明SPMD

要说明以下SPMD程序:

```
parfor  (i=0; i<=N, i++) foo(i)
```

用户需写一个以下程序:

```
pid=my_process_id();
numproc=number_of _processes();
parfor  (i=pid; i<=N, i=i+numproc) foo(i)
```

此程序经编译后生成可执行程序A, 用shell脚本将它加载到N个处理结点上:

```
run A –numnodes N
```

用数据并行程序的构造方法

要说明以下SPMD程序:

```
parfor  (i=0; i<=N, i++) {
C[i]=A[i]+B[i];
}
```

用户可用一条数据赋值语句:

```
C=A+B
```

或

```
forall (i=1,N) C[i]=A[i]+B[i]
```

SPMD和MPMD

MPMD程序的构造方法

用多代码方法说明MPMD

对不提供并行块或并行循环的语言

要说明以下MPMD程序:

parbegin S1 S2 S3 parend

用户需写3个程序, 分别编译生成3个可执行程序S1 S2 S3, 用shell脚本将它们加载到3个处理结点上:

run S1 on node1

run S2 on node1

run S3 on node1

S1, S2和S3是顺序语言程序加上进行交互的库调用.

用SPMD伪造MPMD

要说明以下MPMD程序:

parbegin S1 S2 S3 parend

可以用以下SPMD程序:

```
parfor (i=0; i<3, i++) {
    if (i=0) S1
    if (i=1) S2
    if (i=2) S3
    }
```

因此, 对于可扩展并行机来说, 只要支持SPMD就足够了

分布存储系统并行编程

- 14.1 基于消息传递的并行编程
- 14.2 MPI并行编程
  - 6个基本函数组成的MPI子集
  - MPI消息
  - 点对点通信
  - 群集通信

MPI简介

- MPI(Message Passing Interface)是一个消息传递接口标准
- MPI提供一个可移植、高效、灵活的消息传递接口库
- MPI以语言独立的形式存在，可运行在不同的操作系统和硬件平台上
- MPI提供与C/C++和Fortran语言的绑定

MPI简介

- MPI的版本
  - MPICH：http://www-unix.mcs.anl.gov/mpi/mpich
  - LAM (Local Area Multicomputer): http://www.lam-mpi.org
  - Open-MPI: http://www.open-mpi.org/
  - CHIMP: ftp://ftp.epcc.ed.ac.uk/pub/chimp/release/

6个基本函数组成的MPI子集

```
#include "mpi.h" /*MPI头函数，提供了MPI函数和数据类型定义*/
int main( int argc, char** argv )
{
int rank, size, tag=1;
int senddata,recvdata;
MPI_Status status;
MPI_Init(&argc, &argv); /*MPI的初始化函数*/
MPI_Comm_rank(MPI_COMM_WORLD, &rank); /*该进程编号*/
MPI_Comm_size(MPI_COMM_WORLD, &size); /*总进程数目*/
```

6个基本函数组成的MPI子集

```
if (rank==0){
senddata=9999;
MPI_Send( &senddata, 1, MPI_INT, 1, tag, MPI_COMM_WORLD); /*发
    送数据到进程1*/
}
if (rank==1)
MPI_Recv(&recvdata, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
    &status);
/*从进程0接收数据*/
MPI_Finalize(); /*MPI的结束函数*/
return (0);
}
```

6个基本函数组成的MPI子集

- MPI初始化：通过MPI_Init函数进入MPI环境并完成所有的初始化工作。
  - int MPI_Init(int *argc, char * * * argv)
- MPI结束：通过MPI_Finalize函数从MPI环境中退出。
  - int MPI_Finalize(void)

6个基本函数组成的MPI子集

- 获取进程的编号：调用MPI_Comm_rank函数获得当前进程在指定通信域中的编号，将自身与其他程序区分。
  - int MPI_Comm_rank(MPI_Comm comm, int *rank)
- 获取指定通信域的进程数：调用MPI_Comm_size函数获取指定通信域的进程个数，确定自身完成任务比例。
  - int MPI_Comm_size(MPI_Comm comm, int *size)

6个基本函数组成的MPI子集

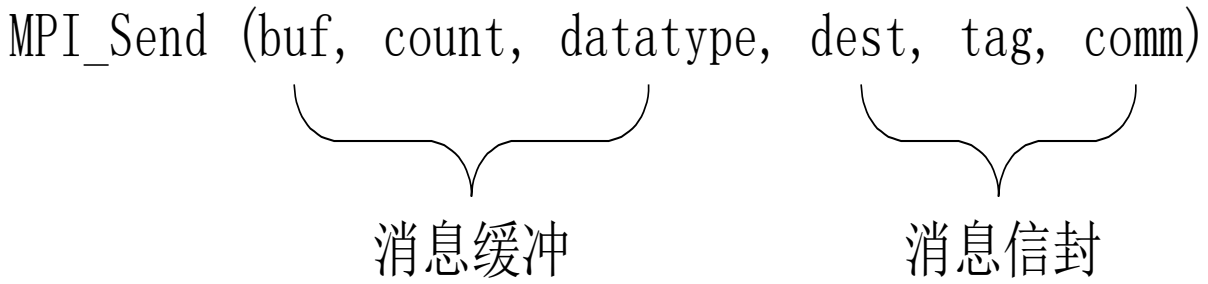
- 消息发送：MPI_Send函数用于发送一个消息到目标进程。
  - int MPI_Send(void *buf, int count, MPI_Datatype dataytpe, int dest, int tag, MPI_Comm comm)
- 消息接受:MPI_Recv函数用于从指定进程接收一个消息
  - int MPI_Recv(void *buf, int count, MPI_Datatype datatyepe,int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI消息

- 一个消息好比一封信
- 消息的内容即信的内容，在MPI中称为消息缓冲(Message Buffer)
- 消息的接收/发送者即信的地址，在MPI中称为消息信封(Message Envelop)

MPI消息

- MPI中，消息缓冲由三元组<起始地址，数据个数，数据类型>标识
- 消息信封由三元组<源/目标进程，消息标签，通信域>标识



- 三元组的方式使得MPI可以表达更为丰富的信息，功能更强大

MPI消息(数据类型)

- MPI的消息类型分为两种：预定义类型和派生数据类型(Derived Data Type)
- 预定义数据类型:MPI支持异构计算(Heterogeneous Computing)，它指在不同计算机系统上运行程序，每台计算可能有不同生产厂商，不同操作系统。
  - MPI通过提供预定义数据类型来解决异构计算中的互操作性问题，建立它与具体语言的对应关系。
- 派生数据类型：MPI引入派生数据类型来定义由数据类型不同且地址空间不连续的数据项组成的消息。

MPI消息(数据类型)

表 2.1　MPI 中预定义的数据类型

MPI(C 语言绑定)	C	MPI(Fortran 语言绑定)	Fortran
MPI_BYTE		**MPI_ BYTE**	
MPI_CHAR	signed char	MPI_CHARACTER	CHARACTER
		MPI_COMPLEX	COMPLEX
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_FLOAT	float	MPI_REAL	REAL
MPI_INT	int	MPI_INTEGER	INTEGER
		MPI_LOGICAL	LOGICAL
MPI_LONG	long		
MPI_LONG_DOUBLE	long double		
MPI_PACKED		**MPI_PACKED**	
MPI_SHORT	short		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long		
MPI_UNSIGNED_SHORT	unsigned short		

MPI消息(数据类型)

- MPI提供了两个附加类型:MPI_BYTE和MPI_PACKED 。
- MPI_BYTE表示一个字节，所有的计算系统中一个字节都代表8个二进制位。
- MPI_PACKED预定义数据类型被用来实现传输地址空间不连续的数据项 。

MPI消息(数据类型)

```
double A[100];
MPI_Pack_size (50,MPI_DOUBLE,comm,&BufferSize);
TempBuffer = malloc(BufferSize);
j = sizeof(MPI_DOUBLE);
Position = 0;
for (i=0;i<50;i++)
    MPI_Pack(A+i*j,1,MPI_DOUBLE,TempBuffer,BufferSize,&Position,comm);
MPI_Send(TempBuffer,Position,MPI_PACKED,destination,tag,comm);
```

- MPI_Pack_size函数来决定用于存放50个MPI_DOUBLE数据项的临时缓冲区的大小
- 调用malloc函数为这个临时缓冲区分配内存
- for循环中将数组A的50个偶序数元素打包成一个消息并存放在临时缓冲区

MPI消息(数据类型)

- 消息打包，然后发送

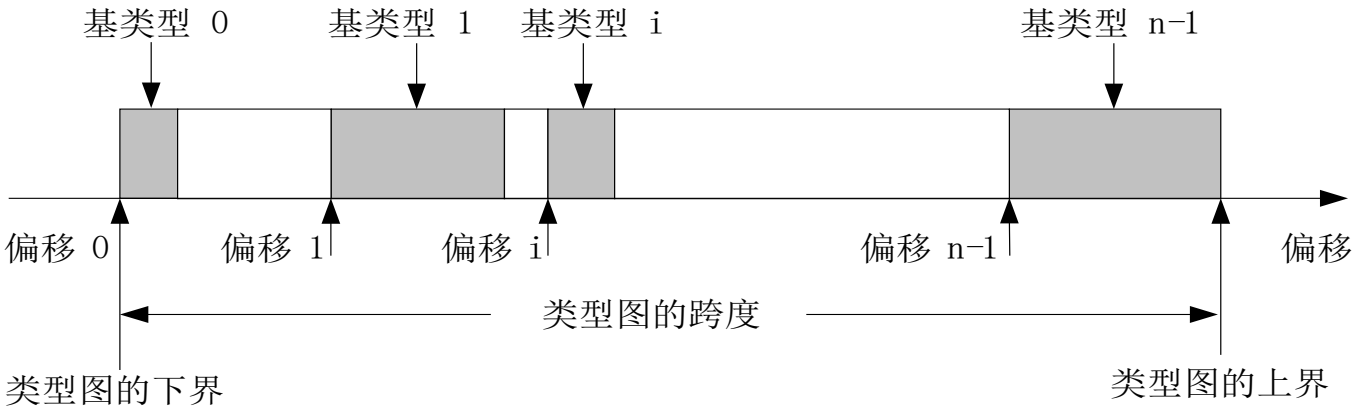
```
MPI_Pack(buf, count, dtype,
                //以上为待打包消息描述
                packbuf, packsize, packpos,
                //以上为打包缓冲区描述
                communicator)
```

- 消息接收，然后拆包

```
MPI_Unpack(packbuf, packsize, packpos,
                  //以上为拆包缓冲区描述
                 buf,  count,  dtype,
                // 以上为拆包消息描述
                communicatior)
```

MPI消息(数据类型)

- 派生数据类型可以用类型图来描述，这是一种通用的类型描述方法，它是一系列二元组<基类型，偏移>的集合，可以表示成如下格式：
  - {<基类型0,偏移0>，…，<基类型n-1,偏移n-1>}
- 在派生数据类型中，基类型可以是任何MPI预定义数据类型，也可以是其它的派生数据类型，即支持数据类型的嵌套定义。
- 阴影部分是基类型所占用的空间，其它空间可以是特意留下的，也可以是为了方便数据对齐。



MPI消息(数据类型)

- MPI提供了全面而强大的**构造函数**(Constructor Function)来定义派生数据类型。

函数名	含义
MPI_Type_contiguous	定义由相同数据类型的元素组成的类型
MPI_Type_vector	定义由成块的元素组成的类型，块之间具有相同间隔
MPI_Type_indexed	定义由成块的元素组成的类型，块长度和偏移由参数指定
MPI_Type_struct	定义由不同数据类型的元素组成的类型
MPI_Type_commit	提交一个派生数据类型
MPI_Type_free	释放一个派生数据类型

MPI消息(数据类型)

```
double A[100];
MPI_Datatype EvenElements;
…
MPI_Type_vector(50, 1, 2, MPI_DOUBLE, &EvenElements);
MPI_Type_commit(&EvenElements);
MPI_Send(A, 1, EvenElements, destination, ⋯);
```

- 首先声明一个类型为MPI_Data_type的变量EvenElements
- 调用构造函数MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)来定义派生数据类型
- 新的派生数据类型必须先调用函数MPI_Type_commit获得MPI系统的确认后才能调用MPI_Send进行消息发送

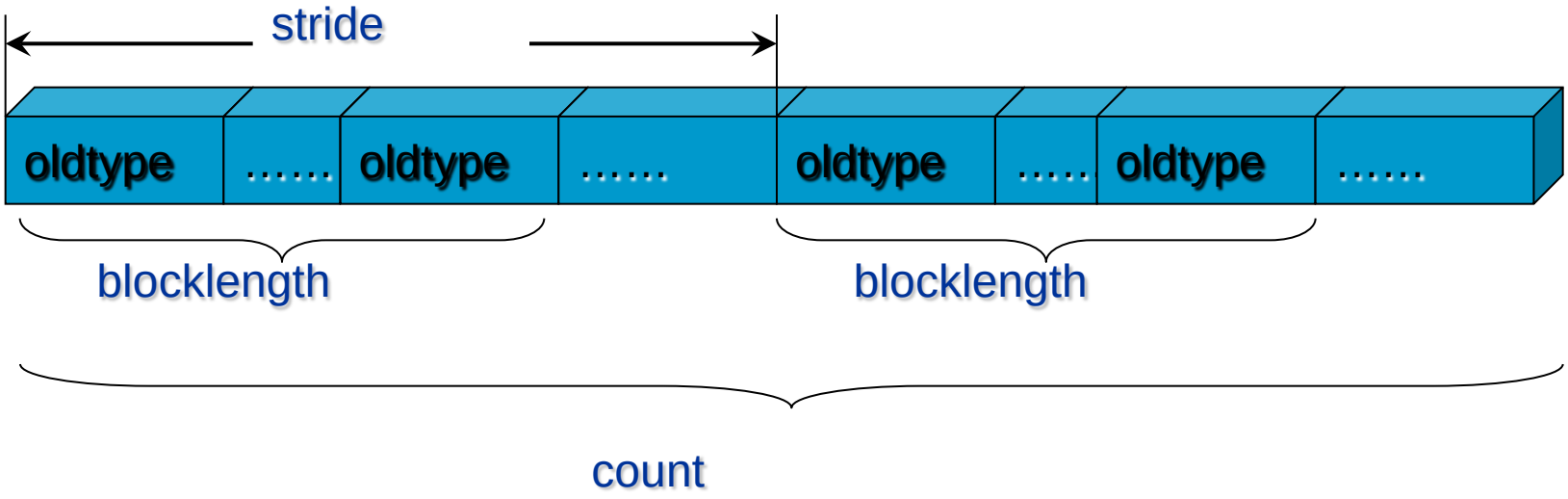
MPI消息(数据类型)

- 调用构造函数MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)来定义派生数据类型。
- 该newtype由count个数据块组成。
- 而每个数据块由blocklength个oldtype类型的连续数据项组成。
- 参数stride定义了两个连续数据块的起始位置之间的oldtype类型元素的个数。因此，两个块之间的间隔可以由(stride-blocklength)来表示。

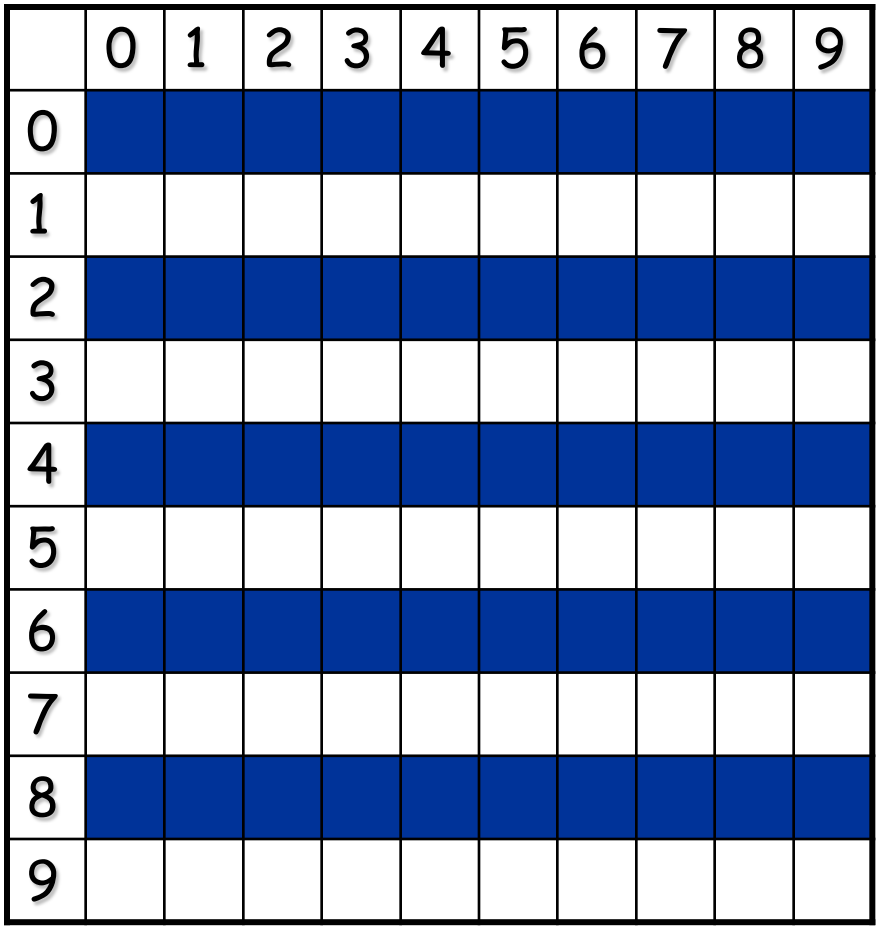
- MPI_Type_vector(50,1,2,MPI_DOUBLE,&EvenElements)函数调用产生了派生数据类型EvenElements，它由50个块组成，每个块包含一个双精度数，后跟一个(2－1)MPI_DOUBLE(8字节)的间隔，接在后面的是下一块。上面的发送语句获取数组A的所有序号为偶数的元素并加以传递。

MPI消息(数据类型)

- MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)



MPI消息(数据类型)



左图10×10整数矩阵的所有偶序号的行:

```
MPI_Type_vector(
    5, // count
    10, // blocklength
    20, // stride
    MPI_INT, //oldtype
    &newtype
)
```

MPI消息(消息标签)

- 为什么需要消息标签？
- 当发送者连续发送两个相同类型消息给同一个接收者，如果没有消息标签，接收者将无法区分这两个消息

Process P:	Process Q:
Send(A, 32, Q)	recv (X, 32, P)
Send(B, 16, Q)	recv (Y, 16, P)

- 这段代码打算传送A的前32个字节进入X，传送B的前16个字节进入Y。但是，尽管消息B后发送，但可能先到达进程Q，就会被第一个接收函数接收在X中。使用标签可以避免这个错误

Process P:	Process Q:
send(A, 32, Q, tag1)	recv (X, 32, P, tag1)
send(B, 16, Q, tag2)	recv (Y, 16, P, tag2)

MPI消息(消息标签)

- 添加标签使得服务进程可以对两个不同的用户进程分别处理，提高灵活性

```
Process P:
send (request1,32, Q)

Process R:
send (request2, 32, Q)

Process Q:
while (true) {
      recv (received_request, 32, Any_Process);
      process received_request;
      }
```

```
Process P:
send(request1, 32, Q, tag1)

Process R:
send(request2, 32, Q, tag2)

Process Q:
while (true){
      recv(received_request, 32, Any_Process, Any_Tag, Status);
      if (Status.Tag==tag1) process received_request in one way;
      if (Status.Tag==tag2) process received_request in another way;
      }
```

MPI消息(通信域)

- **通信域**(Communicator)包括**进程组**(Process Group)和**通信上下文**(Communication Context)等内容，用于描述通信进程间的通信关系。

- 通信域分为组内通信域和组间通信域，分别用来实现MPI的**组内通信**(Intra-communication)和**组间通信**(Inter-communication)。

MPI消息(通信域)

- 进程组是进程的有限、有序集。
  - 有限意味着，在一个进程组中，进程的个数n是有限的，这里的n称为**进程组大小**(Group Size)。
  - 有序意味着，进程的编号是按0，1，…，n-1排列的
- 一个进程用它在一个通信域(组)中的编号进行标识。组的大小和进程编号可以通过调用以下的MPI函数获得：
  - MPI_Comm_size(communicator, &group_size)
  - MPI_Comm_rank(communicator, &my_rank)

MPI消息(通信域)

- 通信上下文：安全的区别不同的通信以免相互干扰
  - 通信上下文不是显式的对象，只是作为通信域的一部分出现
- 进程组和通信上下文结合形成了通信域
  - MPI_COMM_WORLD是所有进程的集合

MPI消息(通信域)

- MPI提供丰富的函数用于管理通信域

函数名	含义
MPI_Comm_size	获取指定通信域中进程的个数
MPI_Comm_rank	获取当前进程在指定通信域中的编号
MPI_Comm_compare	对给定的两个通信域进行比较
MPI_Comm_dup	复制一个已有的通信域生成一个新的通信域，两者除通信上下文不同外，其它都一样。
MPI_Comm_create	根据给定的进程组创建一个新的通信域
MPI_Comm_split	从一个指定通信域分裂出多个子通信域，每个子通信域中的进程都是原通信域中的进程。
MPI_Comm_free	释放一个通信域

MPI消息(通信域)

- 一个在MPI中创建新通信域的例子

```
MPI_Comm MyWorld, SplitWorld;
int my_rank,group_size, Color, Key;
MPI_Init(&argc, &argv);
MPI_Comm_dup(MPI_COMM_WORLD,&MyWorld);
MPI_Comm_rank(MyWorld,&my_rank);
MPI_Comm_size(MyWorld,&group_size);
Color=my_rank%3;
Key=my_rank/3;
MPI_Comm_split(MyWorld,Color,Key,&SplitWorld);
```

MPI消息(通信域)

- MPI_Comm_dup(MPI_COMM_WORLD,&MyWorld)创建了一个新的通信域MyWorld，它包含了与原通信域MPI_COMM_WORLD相同的进程组，但具有不同的通信上下文。
- MPI_Comm_split(MyWorld,Color,Key,&SplitWorld)函数调用则在通信域MyWorld的基础上产生了几个分割的子通信域。原通信域MyWorld中的进程按照不同的Color值处在不同的分割通信域中，每个进程在不同分割通信域中的进程编号则由Key值来标识。

Rank in MyWorld	0	1	2	3	4	5	6	7	8	9
Color	0	1	2	0	1	2	0	1	2	0
Key	0	0	0	1	1	1	2	2	2	3
Rank in SplitWorld(Color=0)	0			1			2			3
Rank in SplitWorld(Color=1)		0			1			2		
Rank in SplitWorld(Color=2)			0			1			2	

MPI消息(通信域)

- 组间通信域是一种特殊的通信域，该通信域包括了两个进程组，分属于两个进程组的进程之间通过组间通信域实现通信。
- 一般把调用进程所在的进程组称为本地进程组，而把另外一个称为远程进程组。

函数名	含义
MPI_Comm_test_inter	判断给定的通信域是否为组间通信域
MPI_Comm_remote_size	获取指定组间通信域中远程进程组的大小
MPI_Comm_remote_group	返回给定组间通信域的远程进程组
MPI_Intercomm_creat	根据给定的两个组内通信域生成一个组间通信域。
MPI_Intercomm_merge	将给定组间通信域包含的两个进程组合并，形成一个新的组内通信域

MPI消息(消息状态)

- 消息状态(MPI_Status类型)存放接收消息的状态信息，包括:

  消息的源进程标识－－MPI_SOURCE

  消息标签－－MPI_TAG

  错误状态－－MPI_ERROR

  其他－－包括数据项个数等，但多为系统保留的。

- 是消息接收函数MPI_Recv的最后一个参数。
- 当一个接收者从不同进程接收不同大小和不同标签的消息时，消息的状态信息非常有用。

MPI消息(消息状态)

- 假设多个客户进程发送消息给服务进程请求服务，通过消息标签来标识客户进程，从而服务进程采取不同的服务

```
while (true){
    MPI_Recv(received_request,100,MPI_BYTE,MPI_Any_source,MPI_Any_tag,comm,&Status);
    switch (Status.MPI_Tag) {
    case tag_0: perform service type0;
    case tag_1: perform service type1;
    case tag_2: perform service type2;
    }
}
```

点对点通信

- MPI的点对点通信(‘Point-to-Point Communication)同时提供了阻塞和非阻塞两种通信机制 。
- 同时也支持多种通信模式。
- 不同通信模式和不同通信机制的结合，便产生了非常丰富的点对点通信函数。

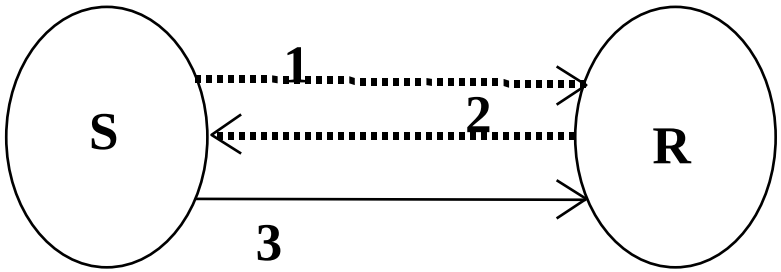
点对点通信(通信模式)

- **通信模式**(Communication Mode)指的是缓冲管理，以及发送方和接收方之间的同步方式。
- 共有下面四种通信模式
  - 同步(synchronous)通信模式
  - 缓冲(buffered)通信模式
  - 标准(standard)通信模式
  - 就绪(ready)通信模式

点对点通信(通信模式)

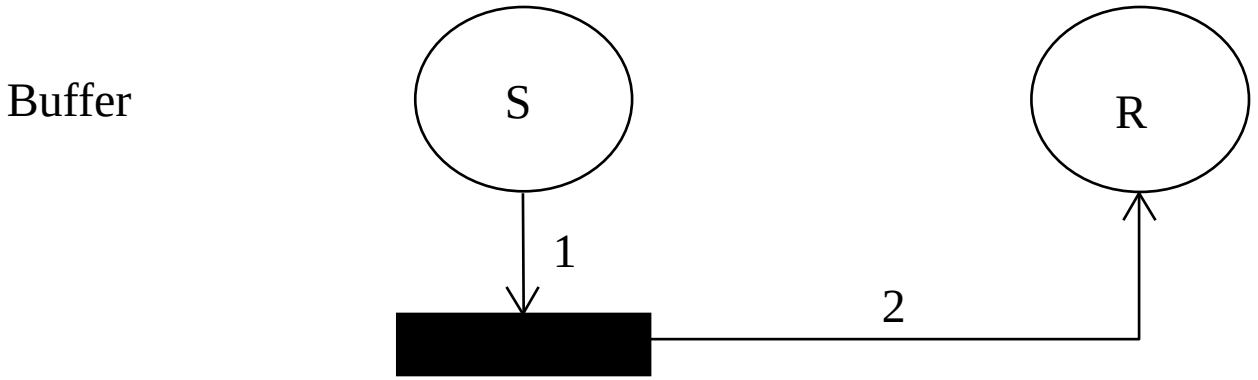
- **同步通信模式：**只有相应的接收过程已经启动，发送过程才正确返回。
- 因此，同步发送返回后，表示发送缓冲区中的数据已经全部被系统缓冲区缓存，并且已经开始发送。
- 当同步发送返回后，发送缓冲区可以被释放或者重新使用。

Synchronous



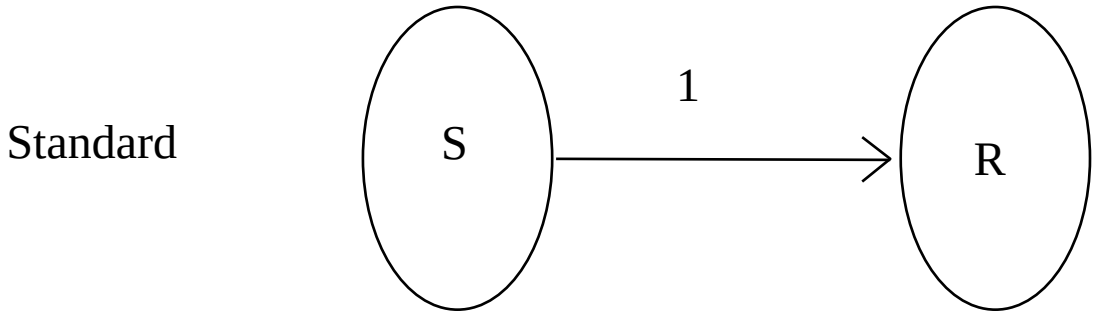
点对点通信(通信模式)

- **缓冲通信模式：**缓冲通信模式的发送不管接收操作是否已经启动都可以执行。
- 但是需要用户程序事先申请一块足够大的缓冲区，通过MPI_Buffer_attch实现，通过MPI_Buffer_detach来回收申请的缓冲区。



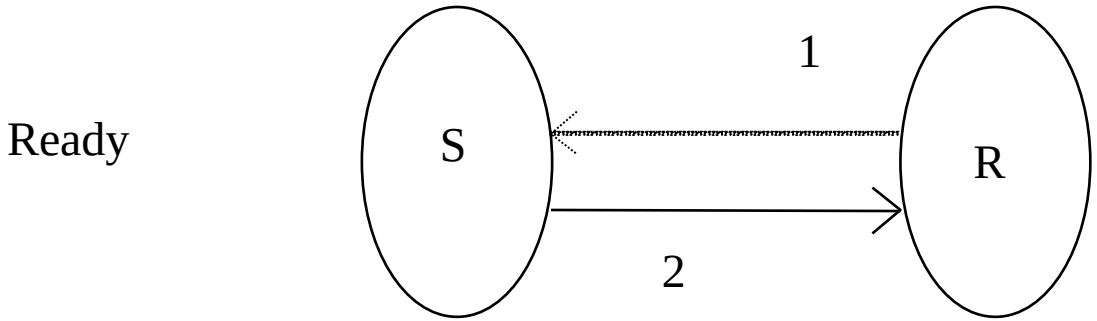
点对点通信(通信模式)

- **标准通信模式：**是否对发送的数据进行缓冲由MPI的实现来决定，而不是由用户程序来控制。
- 发送可以是同步的或缓冲的，取决于实现



点对点通信(通信模式)

- **就绪通信模式：**发送操作只有在接收进程相应的接收操作已经开始才进行发送。
- 当发送操作启动而相应的接收还没有启动，发送操作将出错。就绪通信模式的特殊之处就是接收操作必须先于发送操作启动。



点对点通信(通信模式)

- 阻塞和非阻塞通信的主要区别在于返回后的资源可用性
- 阻塞通信返回的条件：
  - 通信操作已经完成，即消息已经发送或接收
  - 调用的缓冲区可用。若是发送操作，则该缓冲区可以被其它的操作更新；若是接收操作，该缓冲区的数据已经完整，可以被正确引用。

点对点通信(通信模式)

- MPI的发送操作支持四种通信模式，它们与阻塞属性一起产生了MPI中的8种发送操作。
- 而MPI的接收操作只有两种：阻塞接收和非阻塞接收。
- 非阻塞通信返回后并不意味着通信操作的完成，MPI还提供了对非阻塞通信完成的检测，主要的有两种：MPI_Wait函数和MPI_Test函数。

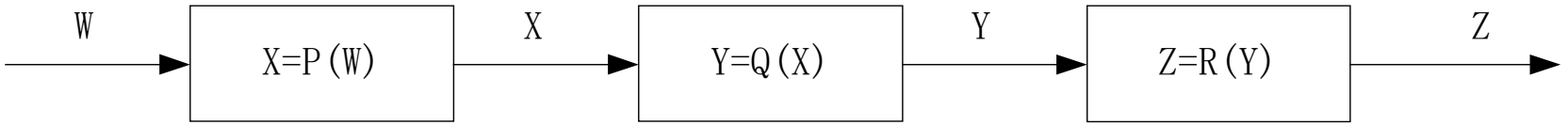
点对点通信(通信模式)

- MPI的点对点通信操作

MPI 原语	**阻塞**	**非阻塞**
Standard Send	MPI_Send	MPI_Isend
Synchronous Send	MPI_ Ssend	MPI_ Issend
Buffered Send	MPI_ Bsend	MPI_ Ibsend
Ready Send	MPI_ Rsend	MPI_ Irsend
Receive	MPI_Recv	MPI_Irecv
Completion Check	MPI_Wait	MPI_Test

点对点通信(通信模式)

- 在阻塞通信的情况下，通信还没有结束的时候，处理器只能等待，浪费了计算资源。
- 一种常见的技术就是设法使计算与通信重叠，非阻塞通信可以用来实现这一目的。
- 一条三进程的流水线，一个进程连续地从左边的进程接收一个输入数据流，计算一个新的值，然后将它发送给右边的进程。




```
while (Not_Done){
MPI_Irevc(NextX, … );
MPI_Isend(PreviousY, … );
CurrentY=Q(CurrentX);
}
```

点对点通信(通信模式)

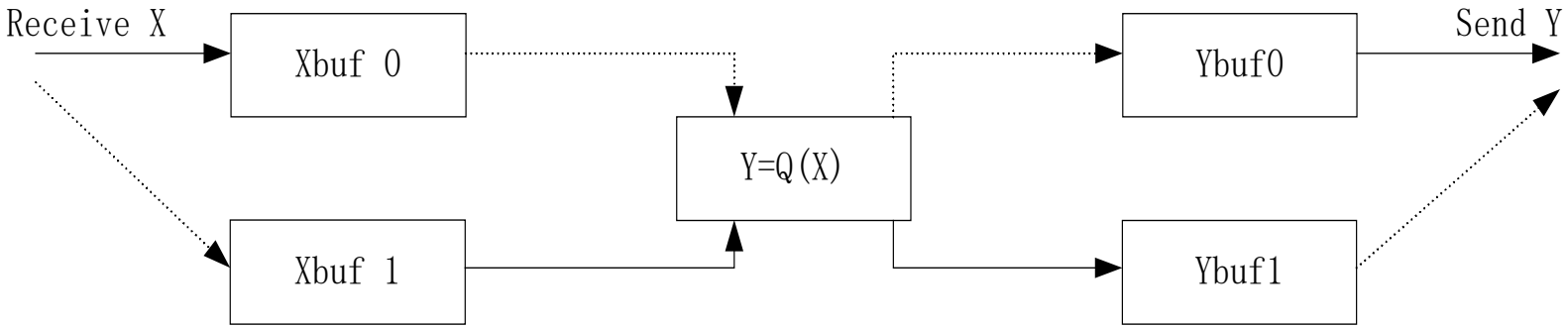
- 非阻塞通信中，双缓冲是一种常用的方法。
  - 我们需要为X和Y各自准备两个单独的缓冲，当接收进程向缓冲中放下一个X时，计算进程可能从另一个缓冲中读当前的X。
  - 我们需要确信缓冲中的数据在缓冲被更新之前使用 。

- 代码如下

```
while (Not_Done){
if (X==Xbuf0) {X=Xbuf1; Y=Ybuf1; Xin=Xbuf0; Yout=Ybuf0;}
else {X=Xbuf0; Y=Ybuf0; Xin=Xbuf1; Yout=Ybuf1;}
MPI_Irevc(Xin, …, recv_handle);
MPI_Isend(Yout, …, send_handle);
Y=Q(X);                          /* 重叠计算*/
MPI_Wait(recv_handle,recv_status);
MPI_Wait(send_handle,send_status);
}
```

点对点通信(通信模式)

- send_handle和revc_handle分别用于检查发送接收是否完成。
- 检查发送接收通过调用MPI_Wait(Handle, Status)来实现，它直到Handle指示的发送或接收操作已经完成才返回 。
- 另一个函数MPI_Test(Handle, Flag, Status)只测试由Handle指示的发送或接收操作是否完成，如果完成，就对Flag赋值True，这个函数不像MPI_Wait，它不会被阻塞。



点对点通信－Send-Recv

- 给一个进程发送消息，从另一个进程接收消息；
- 特别适用于在进程链（环）中进行“移位”操作，而避免在通讯为阻塞方式时出现死锁。

```
MPI_Sendrecv(
   sendbuf, sendcount, sendtype, dest, sendtag,
  //以上为消息发送的描述
  recvbuf, recvcount, recvtype, source, recvtag,
 // 以上为消息接收的描述
 comm, status)
```

群集通信

- **群集通信**(Collective Communications)是一个进程组中的所有进程都参加的全局通信操作。

- 群集通信实现三个功能：通信、聚集和同步
  - 通信功能主要完成组内数据的传输
  - 聚集功能在通信的基础上对给定的数据完成一定的操作
  - 同步功能实现组内所有进程在执行进度上取得一致

群集通信

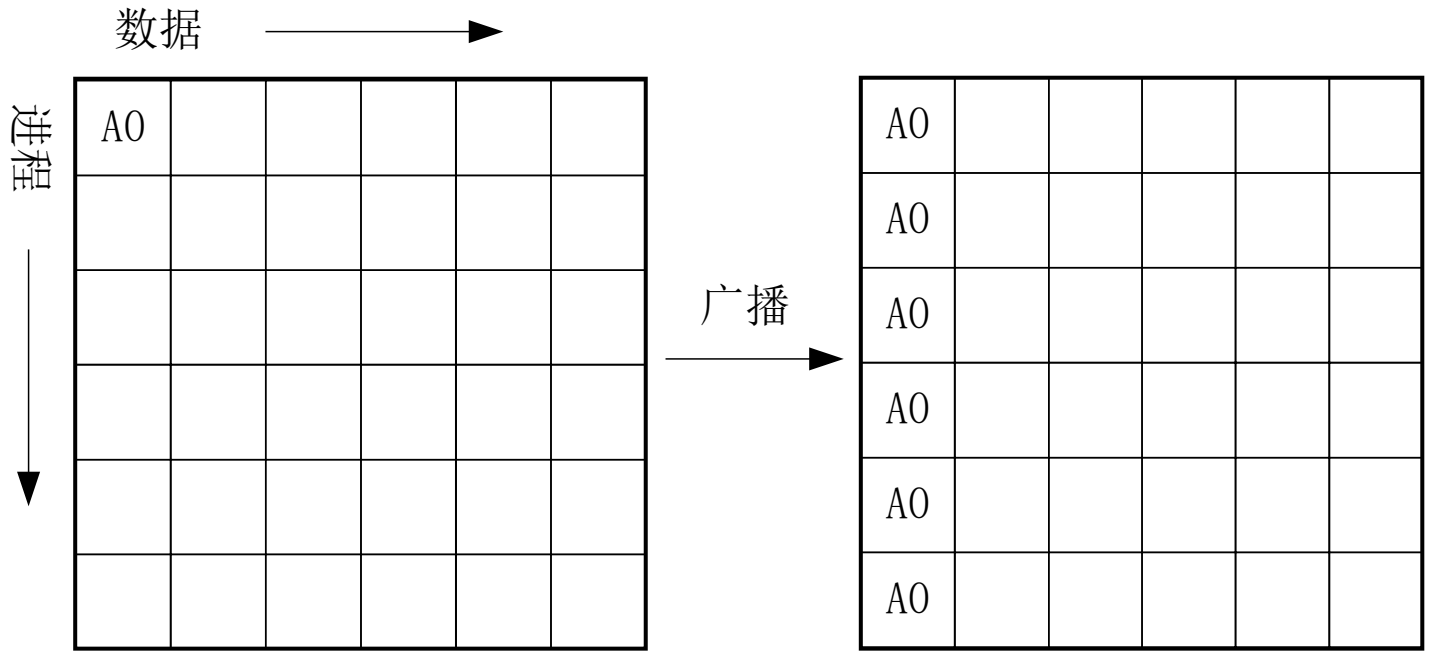
- 群集通信，按照通信方向的不同，又可以分为三种：一对多通信，多对一通信和多对多通信。
  - **一对多通信**：一个进程向其它所有的进程发送消息，这个负责发送消息的进程叫做Root进程。
  - **多对一通信**：一个进程负责从其它所有的进程接收消息，这个接收的进程也叫做Root进程。
  - **多对多通信**：每一个进程都向其它所有的进程发送或者接收消息。

群集通信

类型	函数名	含义
通信	MPI_Bcast	一对多广播同样的消息
	MPI_Gather	多对一收集各个进程的消息
	MPI_Gatherv	MPI_Gather的一般化
	MPI_Allgather	全局收集
	MPI_Allgatherv	MPI_Allgather的一般化
	MPI_Scatter	一对多散播不同的消息
	MPI_Scatterv	MPI_Scatter的一般化
	MPI_Alltoall	多对多全局交换消息
	MPI_Alltoallv	MPI_Alltoall的一般化
聚集	MPI_Reduce	多对一归约
	MPI_Allreduce	MPI_Reduce的一般化
	MPI_Reduce_scatter	MPI_Reduce的一般化
	MPI_Scan	扫描
同步	MPI_Barrier	路障同步

群集通信

- **广播**是一对多通信的典型例子，其调用格式如下：
  - MPI_Bcast(Address, Count, Datatype, Root, Comm)



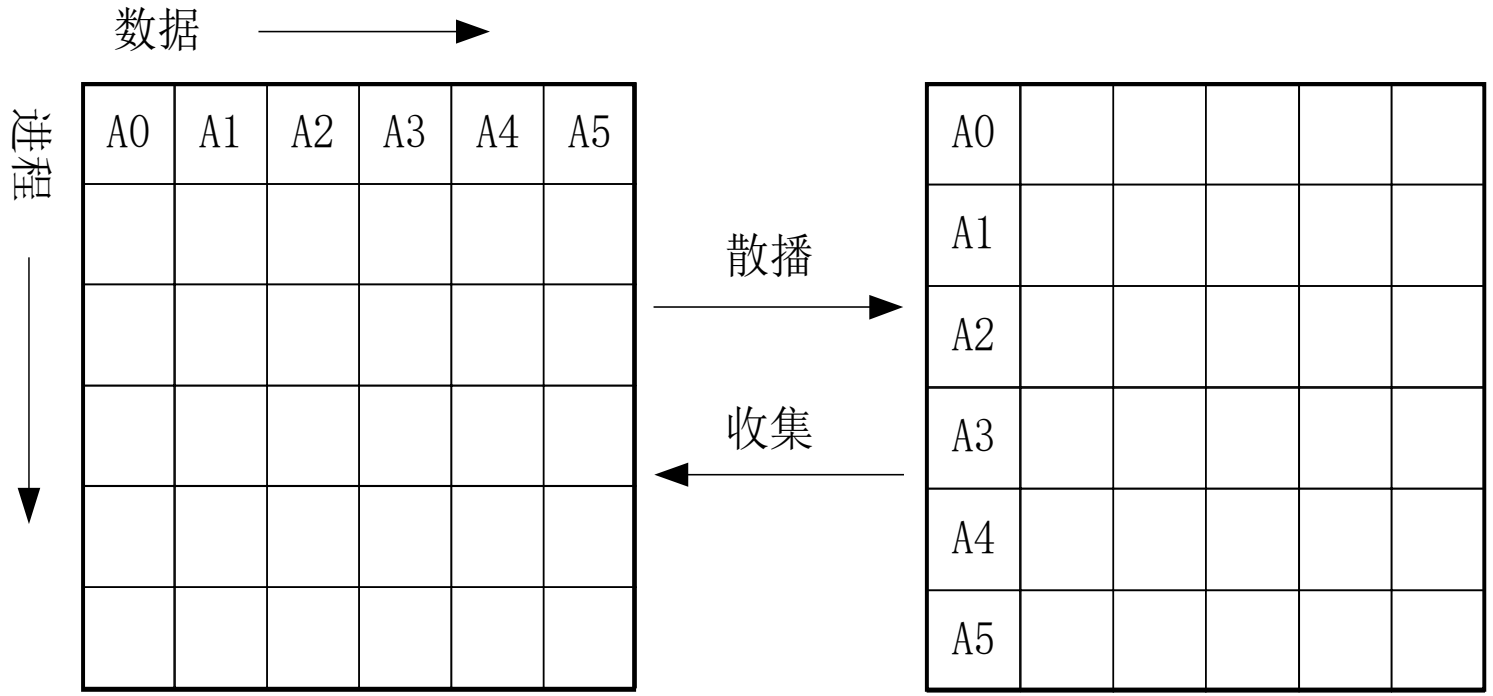
群集通信

- 广播的特点
  - 标号为Root的进程发送相同的消息给通信域Comm中的所有进程。
  - 消息的内容如同点对点通信一样由三元组<Address, Count, Datatype>标识。
  - 对Root进程来说，这个三元组既定义了发送缓冲也定义了接收缓冲。对其它进程来说，这个三元组只定义了接收缓冲

群集通信

- 收集是多对一通信的典型例子，其调用格式下:

  MPI_Gather(SendAddress, SendCount, SendDatatype,
  RecvAddress, RecvCount, RecvDatatype, Root, Comm)



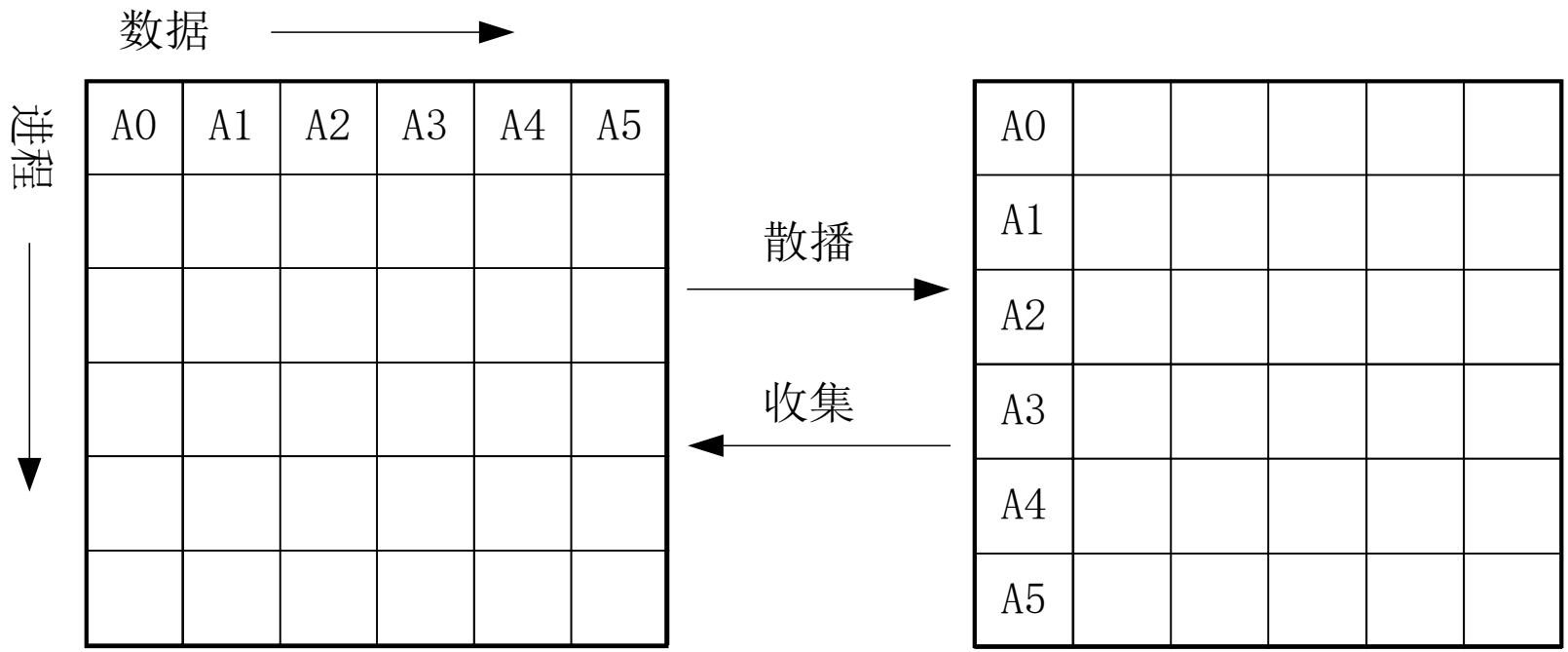
群集通信

- 收集的特点
  - 在收集操作中，Root进程从进程域Comm的所有进程(包括它自己)接收消息。
  - 这n个消息按照进程的标识rank排序进行拼接，然后存放在Root进程的接收缓冲中。
  - 接收缓冲由三元组<RecvAddress, RecvCount, RecvDatatype>标识，发送缓冲由三元组<SendAddress, SendCount, SendDatatype>标识，所有非Root进程忽略接收缓冲。

群集通信

- 散播是一个一对多操作，其调用格式如下:

  MPI_Scatter(SendAddress, SendCount, SendDatatype,
  RecvAddress, RecvCount, RecvDatatype, Root, Comm)



群集通信

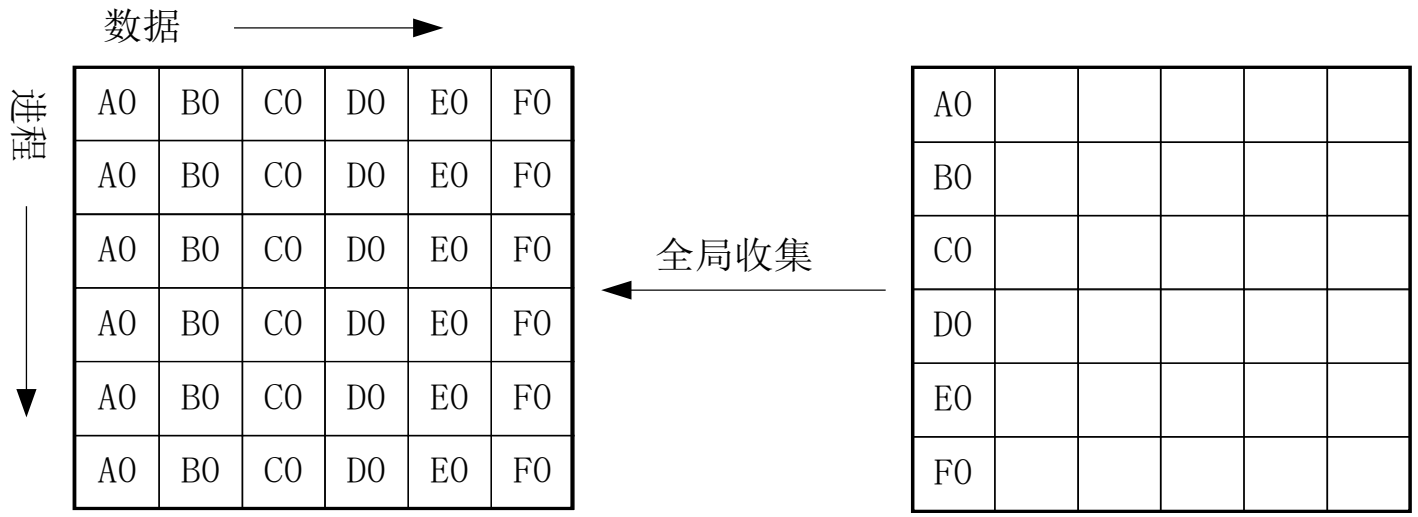
- 散播的特点
  - Scatter执行与Gather相反的操作。
  - Root进程给所有进程(包括它自己)发送一个不同的消息，这n (n为进程域comm包括的进程个数)个消息在Root进程的发送缓冲区中按进程标识的顺序有序地存放。
  - 每个接收缓冲由三元组<RecvAddress, RecvCount, RecvDatatype>标识，所有的非Root进程忽略发送缓冲。对Root进程，发送缓冲由三元组<SendAddress, SendCount, SendDatatype>标识。

群集通信

- 全局收集多对多通信的典型例子，其调用格式如下：

  MPI_Allgather(SendAddress, SendCount, SendDatatype,
  RecvAddress, RecvCount, RecvDatatype, Comm)

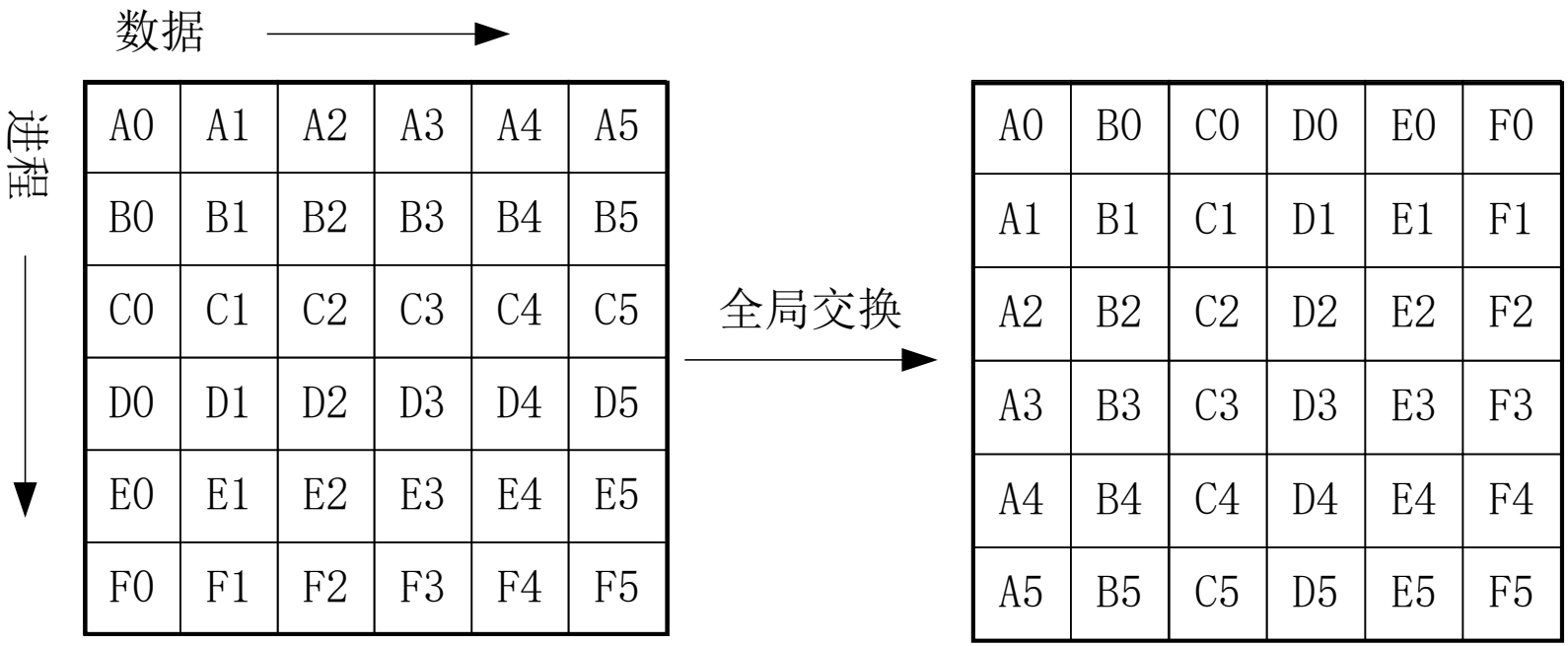
  – Allgather操作相当于每个进程都作为ROOT进程执行了一次Gather调用，即每一个进程都按照Gather的方式收集来自所有进程(包括自己)的数据。



群集通信

- 全局交换也是一个多对多操作，其调用格式如下:

  MPI_Alltoall(SendAddress, SendCount, SendDatatype, RecvAddress, RecvCount, RecvDatatype, Comm)



A0	A1	A2	A3	A4	A5
B0	B1	B2	B3	B4	B5
C0	C1	C2	C3	C4	C5
D0	D1	D2	D3	D4	D5
E0	E1	E2	E3	E4	E5
F0	F1	F2	F3	F4	F5

A0	B0	C0	D0	E0	F0
A1	B1	C1	D1	E1	F1
A2	B2	C2	D2	E2	F2
A3	B3	C3	D3	E3	F3
A4	B4	C4	D4	E4	F4
A5	B5	C5	D5	E5	F5

群集通信

- 全局交换的特点
  - 在全局交换中，每个进程发送一个消息给所有进程(包括它自己)。
  - 这n (n为进程域comm包括的进程个数)个消息在它的发送缓冲中以进程标识的顺序有序地存放。从另一个角度来看这个通信，每个进程都从所有进程接收一个消息，这n个消息以标号的顺序被连接起来，存放在接收缓冲中。
  - 全局交换等价于每个进程作为Root进程执行了一次散播操作。

群集通信

- 同步功能用来协调各个进程之间的进度和步伐。目前MPI的实现中支持一个同步操作，即**路障同步**(Barrier)。
- 路障同步的调用格式如下:
  - MPI_Barrier(Comm)
  - 在路障同步操作MPI_Barrier(Comm)中，通信域Comm中的所有进程相互同步。
  - 在该操作调用返回后，可以保证组内所有的进程都已经执行完了调用之前的所有操作，可以开始该调用后的操作。

群集通信

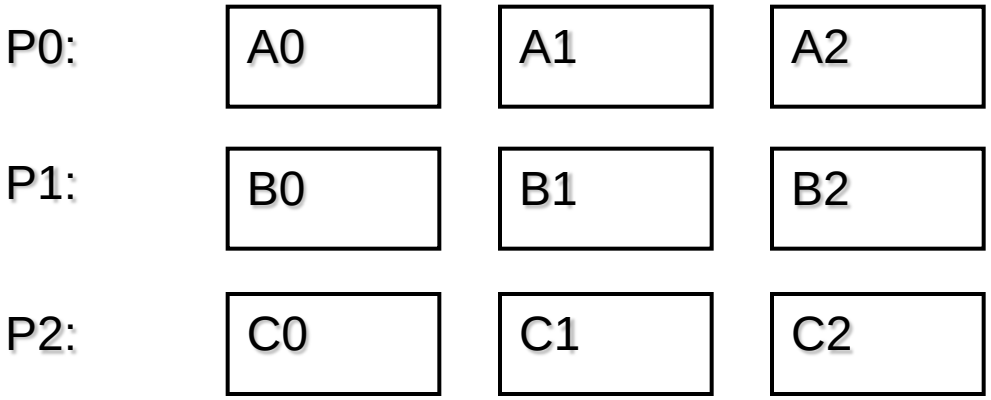
- 群集通信的聚合功能使得MPI进行通信的同时完成一定的计算。
- MPI聚合的功能分三步实现
  - 首先是通信的功能，即消息根据要求发送到目标进程，目标进程也已经收到了各自需要的消息；
  - 然后是对消息的处理，即执行计算功能；
  - 最后把处理结果放入指定的接收缓冲区。
- MPI提供了两种类型的聚合操作: 归约和扫描。

群集通信

- 归约的调用格式如下:
  - MPI_Reduce(SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm)
- 归约的特点
  - 归约操作对每个进程的发送缓冲区(SendAddress)中的数据按给定的操作进行运算，并将最终结果存放在Root进程的接收缓冲区(RecvAddress)中。
  - 参与计算操作的数据项的数据类型在Datatype域中定义，归约操作由Op域定义。
  - 归约操作可以是MPI预定义的,也可以是用户自定义的。
  - 归约操作允许每个进程贡献向量值，而不只是标量值，向量的长度由Count定义。

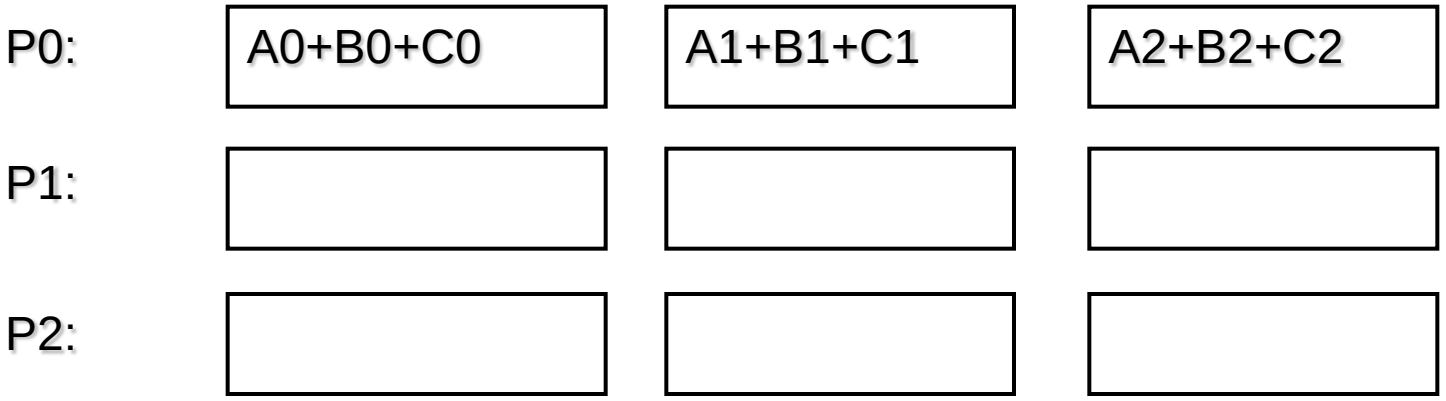
群集通信

- MPI_Reduce: root＝0，Op＝MPI_SUM
- MPI_Allreduce: Op＝MPI_SUM
- 归约前的发送缓冲区



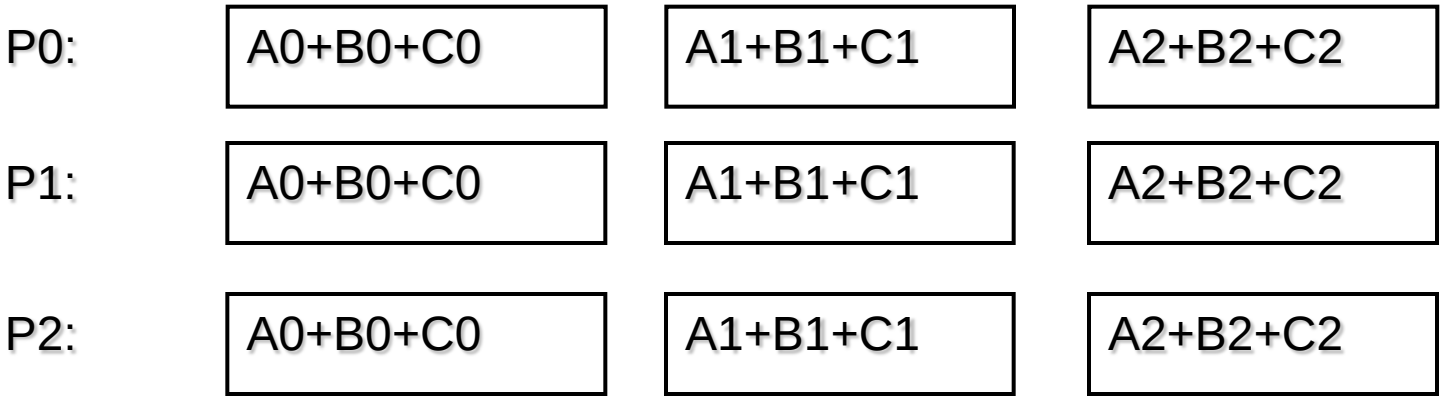
群集通信

- MPI_Reduce: root＝P0，Op＝MPI_SUM
- 归约后的接收缓冲区



群集通信

- MPI_Allreduce: Op＝MPI_SUM
- 归约后的接收缓冲区

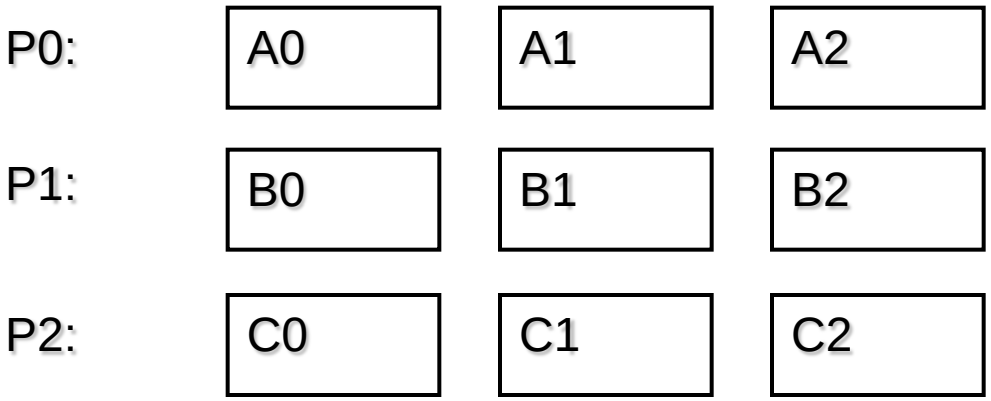


群集通信

- 扫描的调用格式如下：
  - MPI_scan(SendAddress, RecvAddress, Count, Datatype, Op, Comm)
- 扫描的特点
  - 可以把扫描操作看作是一种特殊的归约，即每一个进程都对排在它前面的进程进行归约操作。
  - MPI_SCAN调用的结果是，对于每一个进程i，它对进程0,1,…,i的发送缓冲区的数据进行了指定的归约操作。
  - 扫描操作也允许每个进程贡献向量值，而不只是标量值。向量的长度由Count定义。

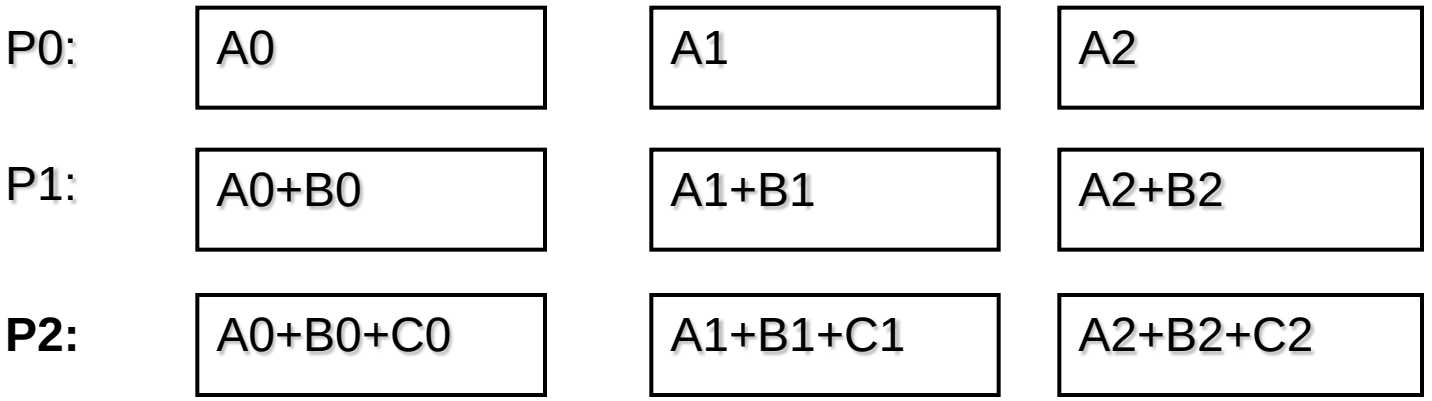
群集通信

- MPI_scan：Op＝MPI_SUM
- 扫描前发送缓冲区：



群集通信

- MPI_scan：Op＝MPI_SUM
- 扫描后接收缓冲区：



群集通信

- 所有的MPI群集通信操作都具有如下的特点:
  - 通信域中的所有进程必须调用群集通信函数。只有通信域中的一部分成员调用了群集通信函数而其它没有调用，是错误的。
  - 除MPI_Barrier以外，每个群集通信函数使用类似于点对点通信中的标准、阻塞的通信模式。也就是说，一个进程一旦结束了它所参与的群集操作就从群集函数中返回，但是并不保证其它进程执行该群集函数已经完成。
  - 一个群集通信操作是不是同步操作取决于实现。MPI要求用户负责保证他的代码无论实现是否同步都必须是正确的。 一般可认为有隐含的同步。
  - 所有参与群集操作的进程中，Count和Datatype必须是兼容的。
  - 群集通信中的消息没有消息标签参数，消息信封由通信域和源/目标定义。例如在MPI_Bcast中，消息的源是Root进程，而目标是所有进程(包括Root)。