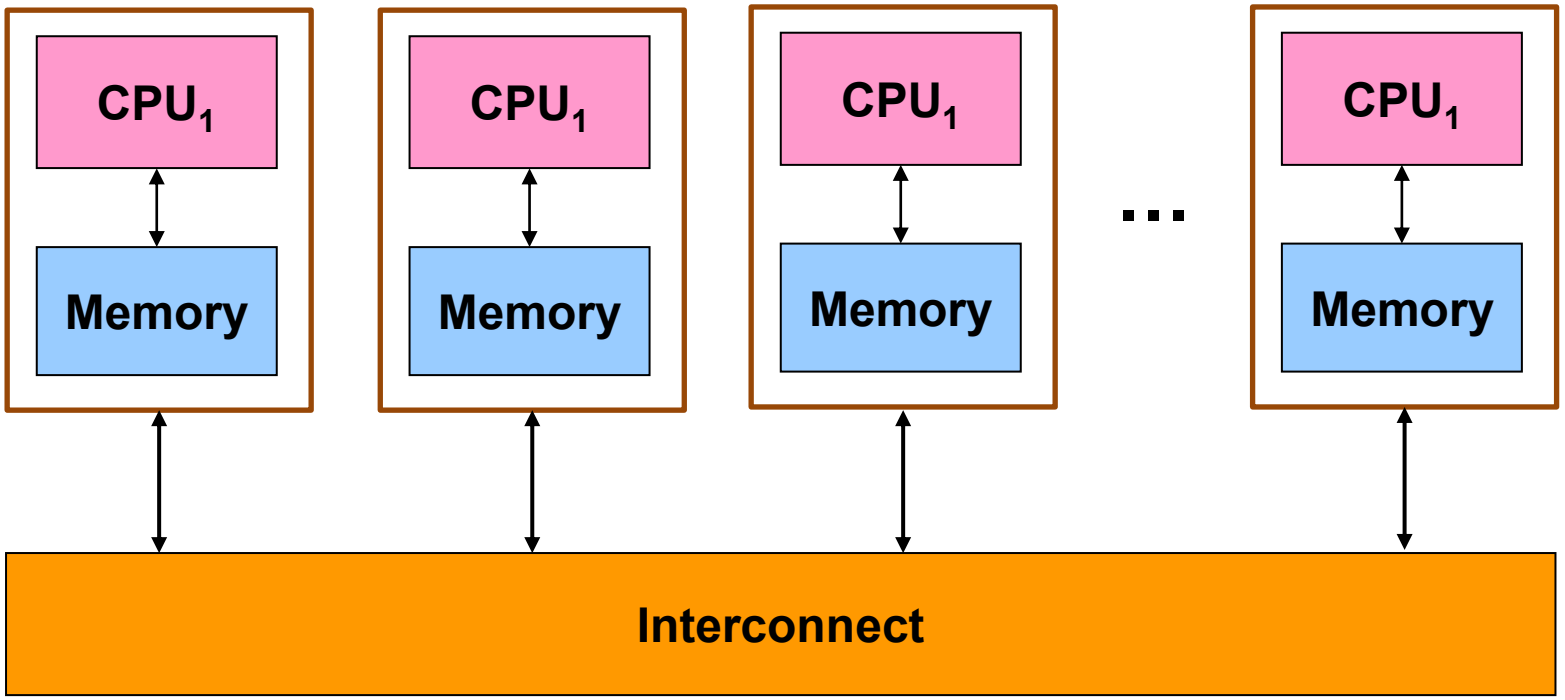
第五章
基于分布式内存的并行计算

哈尔滨工业大学
郝萌

2023, Fall Semester

分布式内存系统



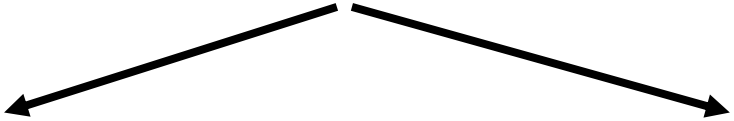
集群

节点可以通过互联网络发送和接收信息来与其他节点进行通信

消息传递编程的原理

- 在消息传递程序中，每个处理器运行一个单独的进程（子程序、任务）
  - 支持消息传递范式机器的逻辑视图是由P个进程组成，每个进程都有自己的专用地址空间
- 所有变量都是私有的
  - 每个数据元素必须属于空间的一个分区；因此，必须对于数据进行显示分区和放置
- 通过特殊的子函数调用进行通信
  - 所有交互（只读或读/写）都需要两个进程的协作——拥有数据的进程和想要访问数据的进程

消息传递编程的原理



SPMD

单程序多数据

- 同一个程序
- 每个进程只知道/操作一小部分数据

MPMD

多程序多数据

- 每个进程执行不同的功能（输入、问题设置、解决方案、输出、显示）

MPI介绍

MPI：Message Passing Interface

- **https://www.mpi-forum.org/**
- **消息传递编程标准，提供一个高效、可扩展、统一的并行编程环境，是目前最为通用的分布式并行编程方式。**
- **MPI是一种消息传递编程模型，是一种标准或规范，MPI实现通过提供库函数实现进程间通信，从而进行并行计算，目前所有并行机制造商都 提供对MPI 的支持。**
- **MPI是一个库，不是一门语言，最终目的是服务于进程间通信**

The goal of the Message-Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

MPI介绍

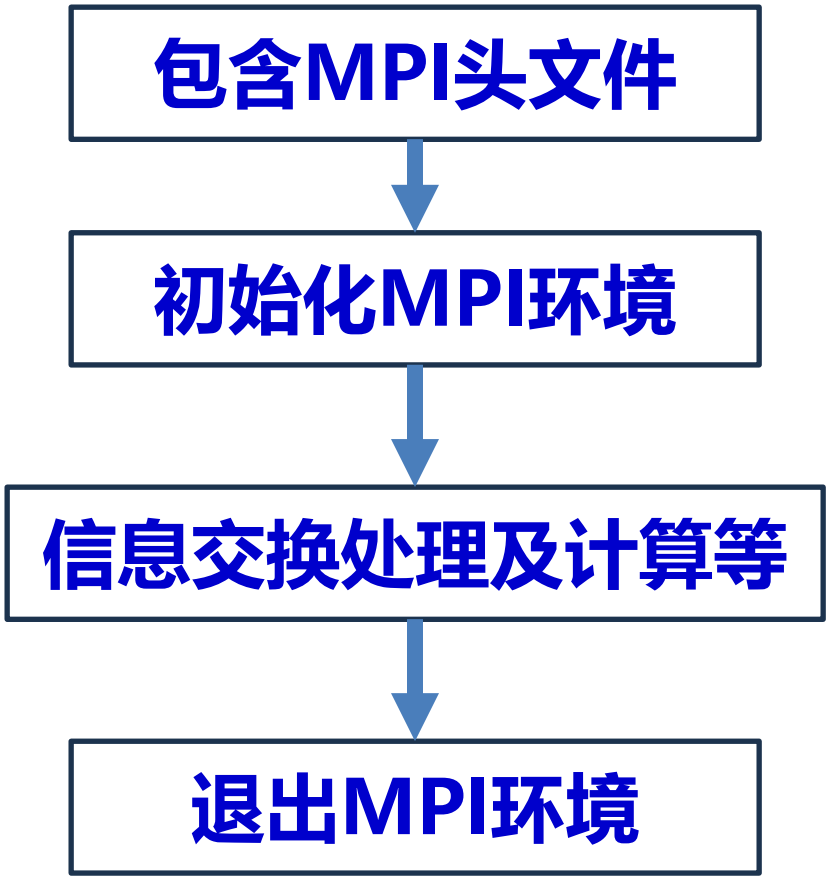
MPI 的目标

- 高通信性能，高可移植性，强大的功能
- Practical，Portable，Efficient，Flexible

MPI 标准和 MPI 实现

- 1994 年 MPI-1.0；1998 年 MPI-2.0；2012 年 MPI-3.0； MPI-3.1 (2015)；MPI-4.0 (2021)； MPI-5.0
- 支持 C/C++ 和 Fortran（目前以 Fortran 90 为主）
- MPI 实现（免费版）：MPICH 和 OpenMPI
- MPI 实现（商业版）：Intel MPI，IBM MPI，HP-MPI，. . .
- 所有版本都遵循 MPI 标准，可以不加修改地运行

MPI程序的一般结构



第一个MPI程序

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char * argv[])
{
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    MPI_Finalize();
    return 0;
}
```

```
mpicc -O2 -o hello mpi_hello.c
mpirun -np 4 ./hello
```

程序分析

```
#include <mpi.h>
```

MPI相对于C 语言的头文件

```
int MPI_Init(int *argc, char ***argv)
```

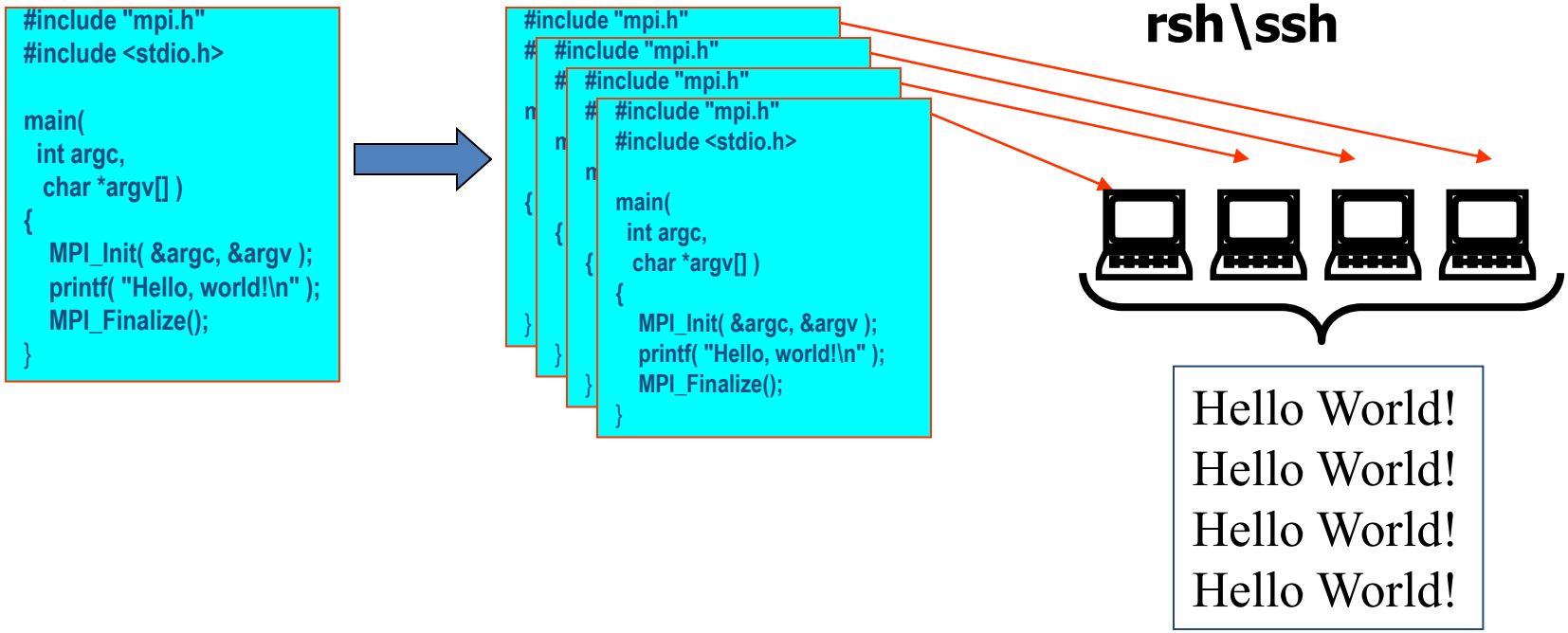
MPI程序的开始，在调用其他MPI函数之前被调用。 其目的是初始化 MPI 环境

```
MPI_Finalize()
```

MPI 程序的结束，在计算结束时被调用，它执行各种清理任务以终止 MPI 环境

Hello程序如何执行

- SPMD: Single Program Multiple Data



MPI_Comm_size 和 MPI_Comm_rank

- 在写MPI程序时，我们通常需要知道以下两个问题的答案：
  - 任务由多少个进程来进行并行计算?
  - 我是哪一个进程?

第二个MPI程序

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
int main(int argc, char * argv[])
{
    int myid, np;  int namelen;
    char proc_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Get_processor_name(proc_name,&namelen);
    fprintf(stderr,"Hello, I am proc. %d of %d on %s\n",
myid, np, proc_name);
    MPI_Finalize();
}
```

程序分析

MPI_MAX_PROCESSOR_NAME

预定义的宏，即 MPI 所允许的机器名字的最大长度

int MPI_Comm_rank(MPI_Comm comm, int *rank)

返回本进程的进程号，是一个整数，范围从零到通信器的大小减一

int MPI_Comm_size(MPI_Comm comm, int *size)

函数，返回所有参加运算的进程的个数

int MPI_Get_processor_name(char *name, int *resultlen)

函数，返回运行本进程所在的结点的主机名

MPI通信器

通信器/通信子（Communicator）

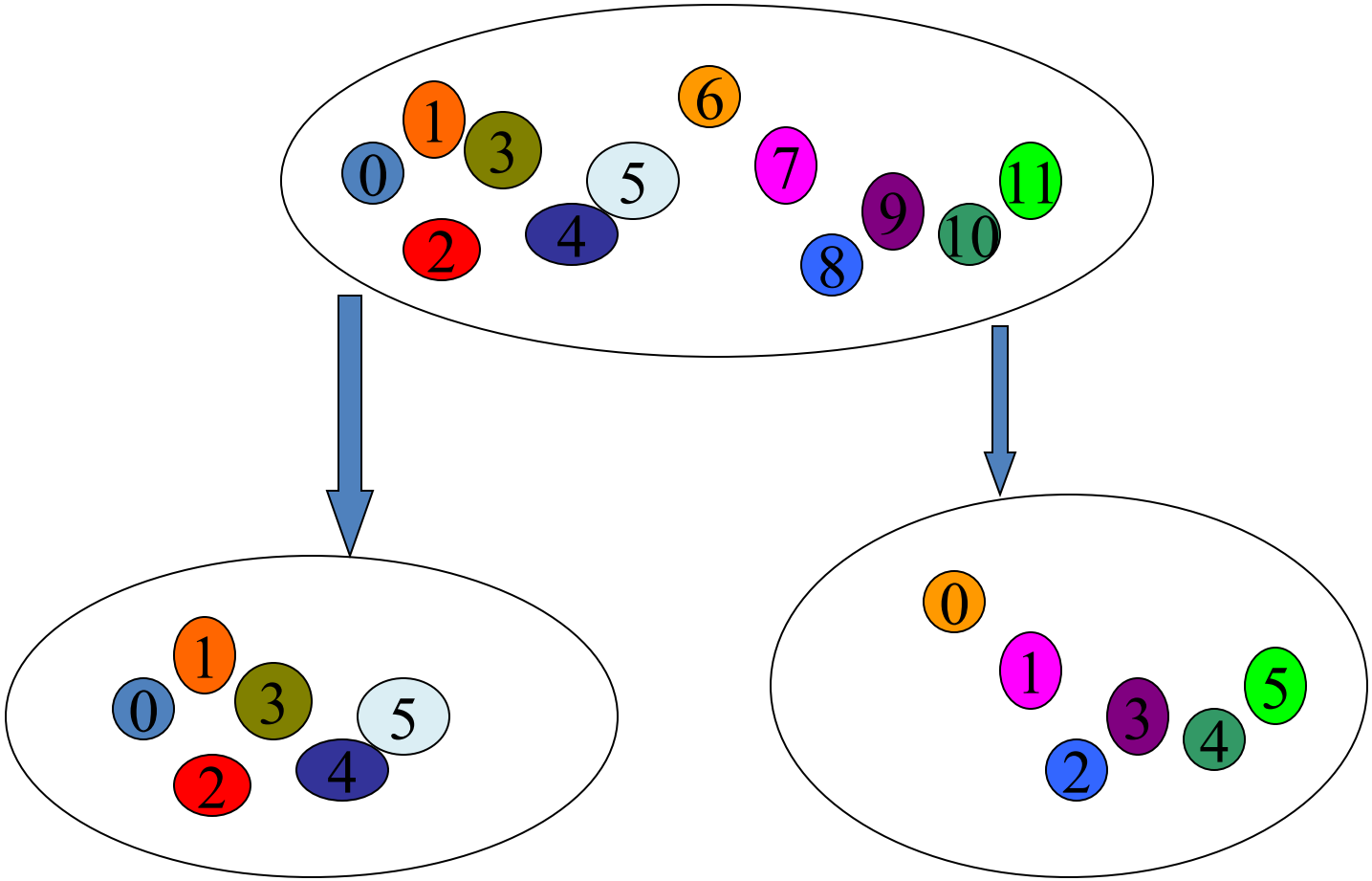
- 一个通信器定义一个通信域（communication domain ）—— 一组允许相互通信的进程
- 有关通信域的信息存储在 MPI_Comm类型的变量中
- MPI程序中进程间的通信必须通过通信器进行
- 在执行函数MPI_Init之后，一个MPI程序的所有进程形成一个缺省的组,这个组的通信域即被写作MPI_COMM_WORLD
- MPI通信操作函数中必不可少的参数，用于限定参加通信的进程的范围

MPI通信器

MPI进程

- MPI 程序中一个独立参与通信的个体
- MPI 进程组：由部分或全部进程构成的有序集合
- 进程号是相对进程组或通信器而言的，同一进程在不同的进程组或通信器中可以有不同的进程号
- 进程号是在进程组或通信器被创建时赋予的
- 进程号取值范围为 0, …, np-1

MPI通信器



程序运行结果

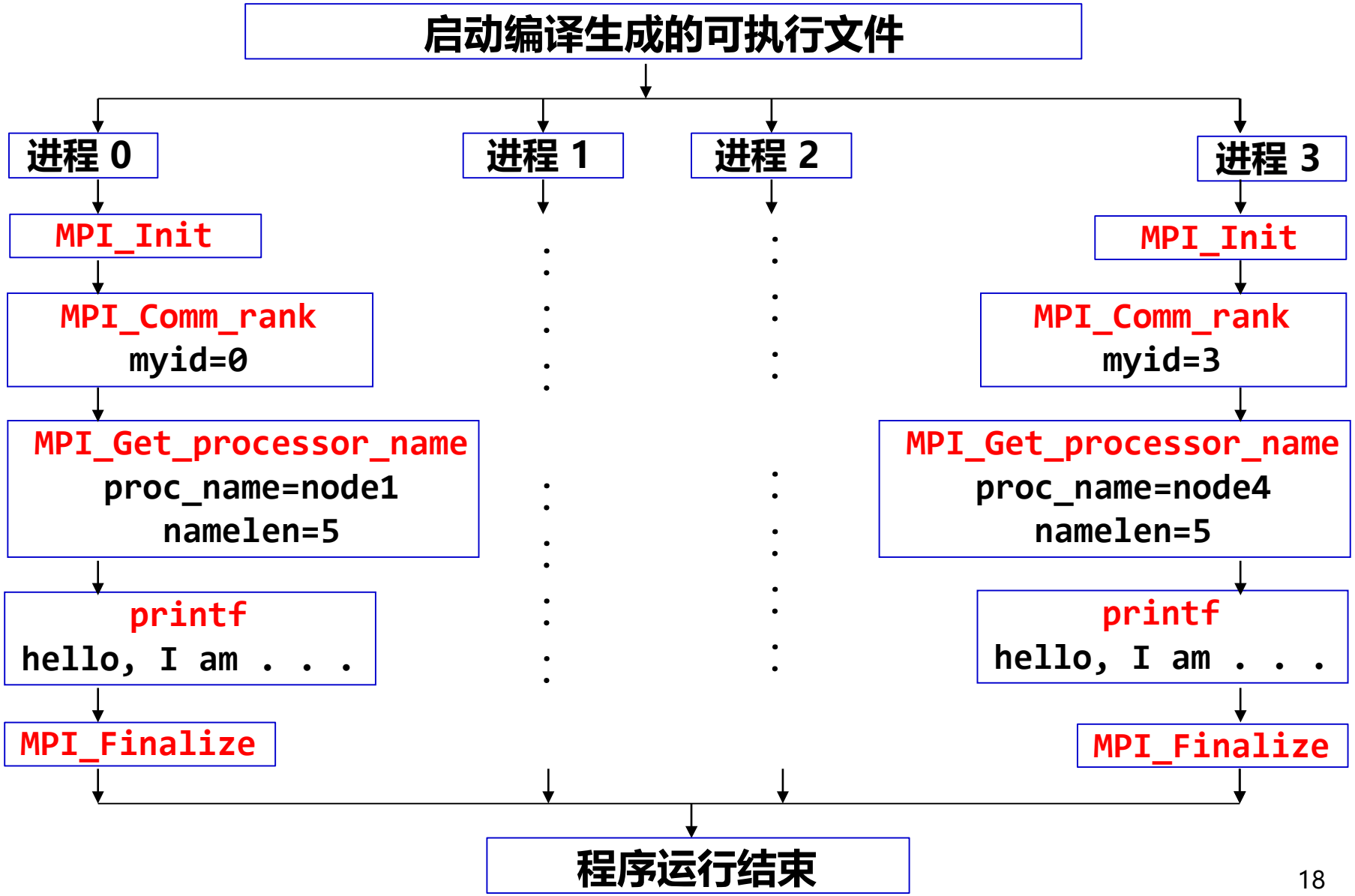
- **在单个节点上，开 4 个进程的运行结果**

```
Hello, I am proc. 1 of 4 on node1
Hello, I am proc. 0 of 4 on node1
Hello, I am proc. 2 of 4 on node1
Hello, I am proc. 3 of 4 on node1
```

- **在四个节点上，开 4 个进程的运行结果**

```
Hello, I am proc. 1 of 4 on node1
Hello, I am proc. 0 of 4 on node2
Hello, I am proc. 2 of 4 on node3
Hello, I am proc. 3 of 4 on node4
```

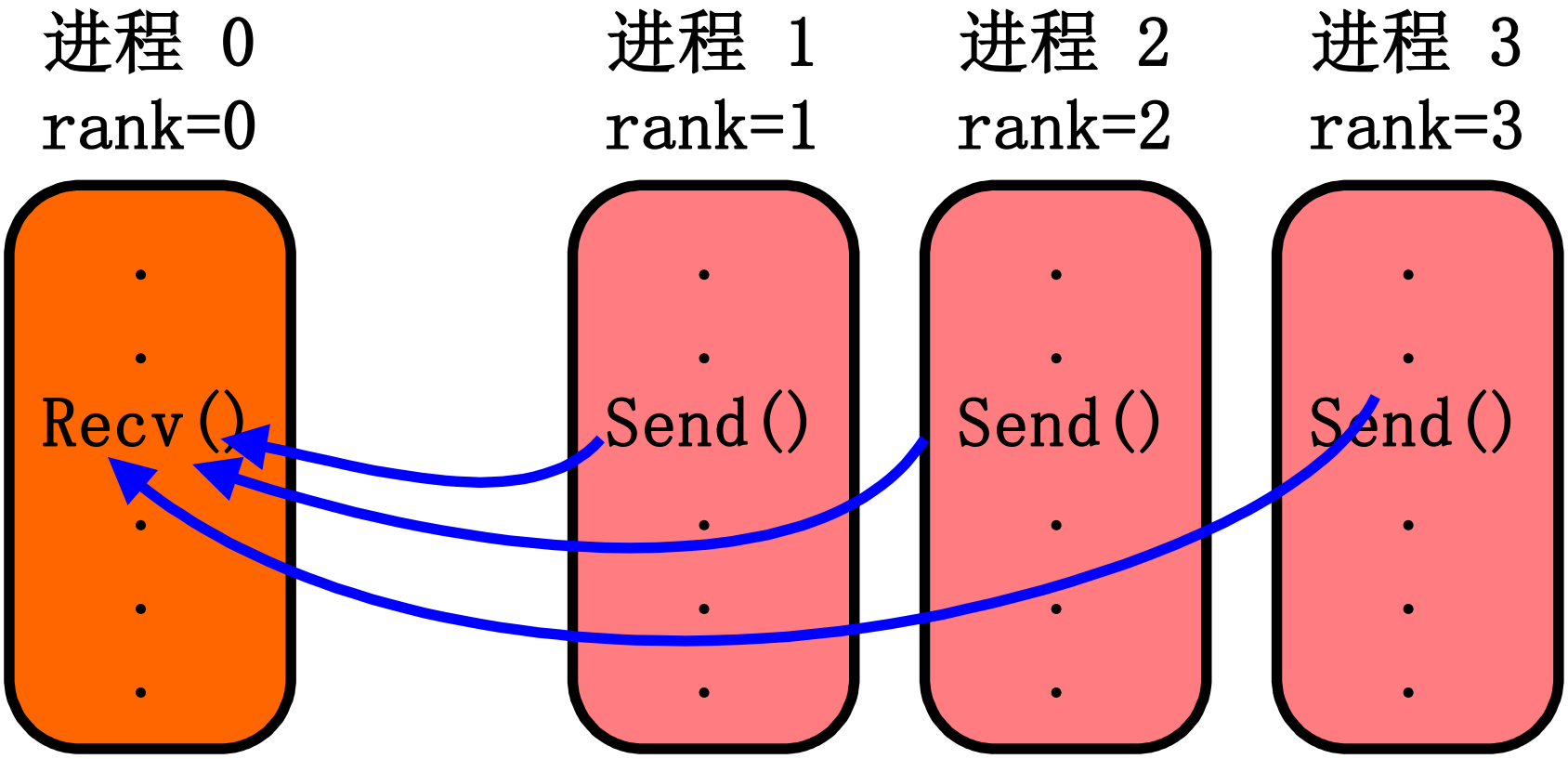
程序运行过程


启动编译生成的可执行文件
进程 0
进程 1
进程 2
进程 3
MPI_Init
MPI_Comm_rank
myid=0
MPI_Get_processor_name
proc_name=node1
namelen=5
printf
hello, I am . . .
MPI_Finalize
MPI_Init
MPI_Comm_rank
myid=3
MPI_Get_processor_name
proc_name=node4
namelen=5
printf
hello, I am . . .
MPI_Finalize
程序运行结束

MPI编程的一些惯例

- MPI 的所有常量、变量与函数均以MPI_ 开头
- 在C 程序中，所有常数的定义除下划线外一律由大写字母组成，在函数和数据类型定义中，接MPI_ 之后的第一个字母大写，其余全部为小写字 母，即MPI_Xxxx_xxx 形式
- 除MPI_Wtime 和MPI_Wtick外，所有C函数调用之后都将返回一个错误信息码
- 由于C语言的函数调用机制是值传递，所以MPI的所有C函数中的输出参数用的都是指针
- MPI 是按进程组(Process Group) 方式工作：
  - 所有 MPI 进程在开始时均是在通信器MPI_COMM_WORLD 所对应的进程组中工作，之后用户可以根据自己的需要，建立其它的进程组
- 所有 MPI 的通信一定要在通信器中进行

MPI并行通信程序



Greeting执行过程

Greetings程序

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<mpi.h>
4
5  const int MAX_STRING=100;
6
7  int main(){
8      char greeting[MAX_STRING];
9      int comm_sz;
10     int my_rank;
11
12     MPI_Init(NULL,NULL);
13     MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);
14     MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
15
16     if(my_rank!=0){
17         sprintf(greeting,"Greetings from process %d of %d!",my_rank,comm_sz);
18         MPI_Send(greeting,strlen(greeting)+1,MPI_CHAR,0,0,MPI_COMM_WORLD);
19     }else{
20         for(int q = 1;q < comm_sz;q++){
21             MPI_Recv(greeting,MAX_STRING,MPI_CHAR,q,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
22             printf("%s\n",greeting);
23         }
24     }
25     MPI_Finalize();
26     return 0;
27 }
```

消息发送和接收

- MPI 中发送和接收消息的基本函数分别是MPI_Send和MPI_Recv，也称为点对点通信

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

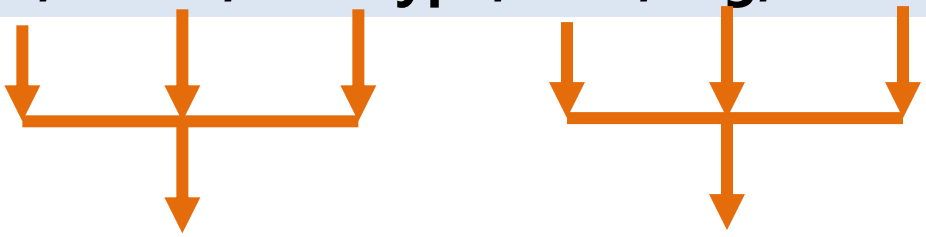
```
int MPI_Recv(void *buf, int count, MPI_Datatype
             datatype, int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

消息发送和接收

MPI 消息（message）

- MPI消息包括信封和数据两部分
- 信封：＜源/目，标识，通信域＞
- 数据：＜起始地址，数据个数，数据类型＞

```
int MPI_Send(buf, count, datatype, dest, tag, comm)
```

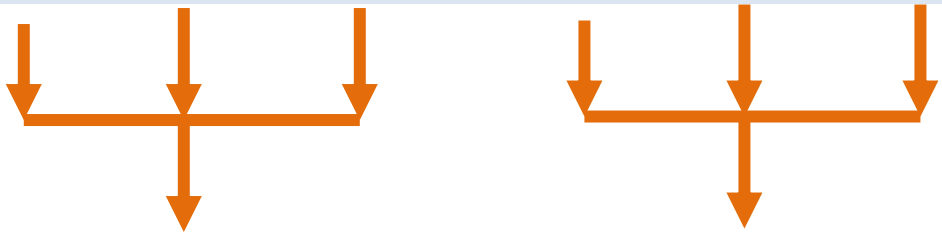


消息发送和接收

MPI 消息（message）

- MPI消息包括信封和数据两部分
- 信封：＜源/目，标识，通信域＞
- 数据：＜起始地址，数据个数，数据类型＞

int MPI_Recv(buf, count, datatype, source, tag, comm, status)



MPI消息数据

buf, count, datatype

buf:　　发送(接收)缓冲区的起始地址

发送哪些连续数据？从内存的哪个位置开始发送?
接收到的数据存放在内存的什么位置?

count:　　发送（接收）数据的个数

发送（接收）多少个数?

datatype:　　发送（接收）的数据类型

发送（接收）什么样的数据?

MPI消息信封

发送：dest, tag, comm

接收：source, tag, comm

dest：目标进程（将消息发送给哪个进程？）

source：源进程 （从哪个进程接收消息？）

tag： 消息标签（当发送者发送两个相同类型的数据给同一个接收者时，如果没有消息标识，接收者将如何区别这两个消息）

MPI数据类型

MPI 除了提供函数外，还定义了一组常量和数据类型

- MPI 常量命名规则：全部大写
- MPI 变量数据类型命名规则：
  - 以MPI_ 开头，后面跟C语言原始数据类型名称
- MPI 数据类型包含原始数据类型和自定义数据类型

† 注意：MPI 的数据类型只用于消息传递！

MPI数据类型

MPI datatype	C datatype
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_INT	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

MPI 数据类型主要用于数据的传递

消息匹配

```
int MPI_Send(send_buf_p, send_buf_sz, send_type, dest,  send_tag,
             send_comm );
```

r

MPI_Send

src = q

MPI_Recv

dest = r

```
int MPI_Recv(recv_buf_p,  recv_buf_sz,  recv_type,  src,  recv_tag,
             recv_comm, &status) ;
```

q

MPI_Send点对点通信

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

参数	含义
buf	所发送消息的内存首地址
count	传递的数据的长度
datatype	变量类型，MPI预定义的变量类型
dest	接收消息的进程的标识号
tag	消息标签，用于识别消息
comm	通信器，数据在其中进程间传输

MPI_Send点对点通信

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

- 阻塞型消息发送接口
- MPI_SEND 将缓冲区中count个datatype类型的数据发给进程号为dest 的目的进程。这里 count 是元素个数，即指定数据类型的个数，不是字节数，数据的起始地址为 buf。本次发送的消息标签是 tag，使用标签的目的是把本次发送的消息和本进程向同一目的进程发送的其它消息区别开来。其中 dest 的取值范围为 0~np-1（np 表示通信器 comm 中的进程数) 或 MPI_PROC_NULL，tag 的取值为 0~ MPI_TAG_UB
- 该函数可以发送各种类型的数据，如整型、实型、字符等
- 点对点通信是 MPI 通信机制的基础

MPI_Recv点对点通信

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

参数	含义
buf	接收消息数据的首地址
count	接收数据的最大个数
datatype	接收数据的数据类型
source	发送消息的进程的标识号
tag	消息标签，用于识别消息
comm	通信器，数据在其中进程间传输
status	返回状态

MPI_Recv点对点通信

```
MPI_Recv(buf,count,datatype,source,tag,comm,status)
```

- 阻塞型消息接收接口
- 从指定的进程source接收不超过count个datatype类型的数据，并把它放到缓冲区中，起始位置为buf，本次消息的标识为tag。这里source的取值范围为0~np-1，或MPI_ANY_SOURCE，或MPI_PROC_NULL，tag的取值为0~MPI_TAG_UB 或MPI_ANY_TAG
- 接收消息时返回的状态 STATUS，在 C 语言中是用结构定义的，在 FORTRAN 中是用数组定义的，其中包括 MPI_SOURCE，MPI_TAG 和 MPI_ERROR。此外 STATUS 还包含接收消息元素的个数，但它不是显式给出的，需要用到后面给出的函数 MPI_Get_Count

示例

```
……
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if(myid==0){
   MPI_Recv( buf1,10,MPI_INT,1,1,MPI_COMM_WORLD,&status);
   MPI_Recv (buf2,10,MPI_INT,2,1,MPI_COMM_WORLD,&status);
}
if(myid==1)
   MPI_Send (buf1,10 ,MPI_INT,0,1,MPI_COMM_WORLD );
if (myid ==2)
   MPI_Send(buf2,10 ,MPI_INT,0,1,MPI_COMM_WORLD );
……
```

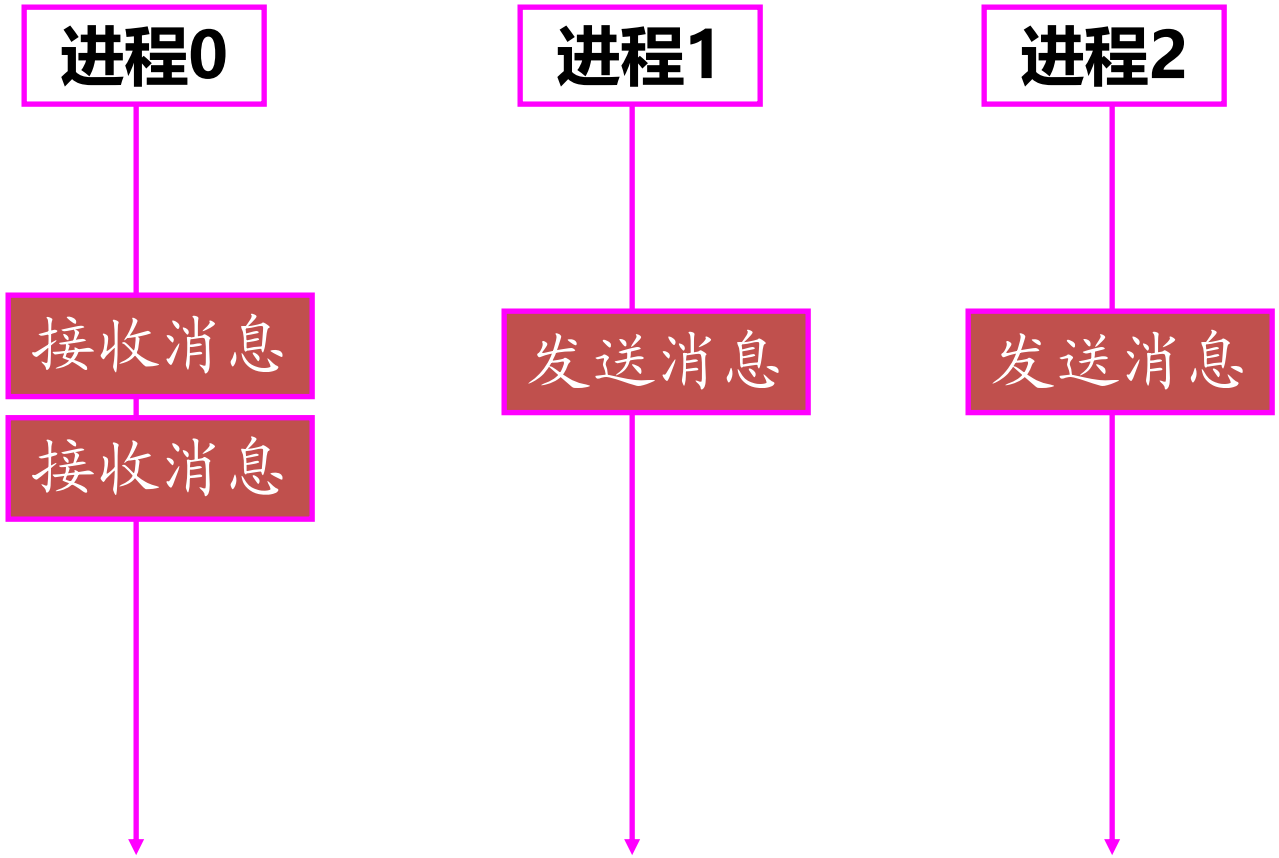
消息来源于进程1

消息来源于进程2

消息发送给进程0

消息发送给进程0

程序执行过程



任意源和任意标识

MPI_ANY_SOURCE，MPI_ANY_TAG

- MPI_ANY_SOURCE：
  任何进程发送的消息都可以接收，但其它要求必须满足
- MPI_ANY_TAG：
  任何标签的消息都可以接收，但其它要求必须满足
- 两者可同时使用，也可单独使用
- 不能给通信域comm指定任意值
- 发送操作与接收操作不对称

示例

```
……
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if(myid == 0){
    MPI_Recv(buf1, 10, MPI_INT, MPI_ANY_SOURCE, 1,
        MPI_COMM_WORLD, &status);
    MPI_Recv (buf2,10,MPI_INT, MPI_ANY_SOURCE, 1,
        MPI_COMM_WORLD, &status);
}
if(myid == 1)
    MPI_Send (buf1,10 ,MPI_INT,0,1,MPI_COMM_WORLD );
if (myid == 2)
    MPI_Send(buf2,10 ,MPI_INT,0,1,MPI_COMM_WORLD )
……
```

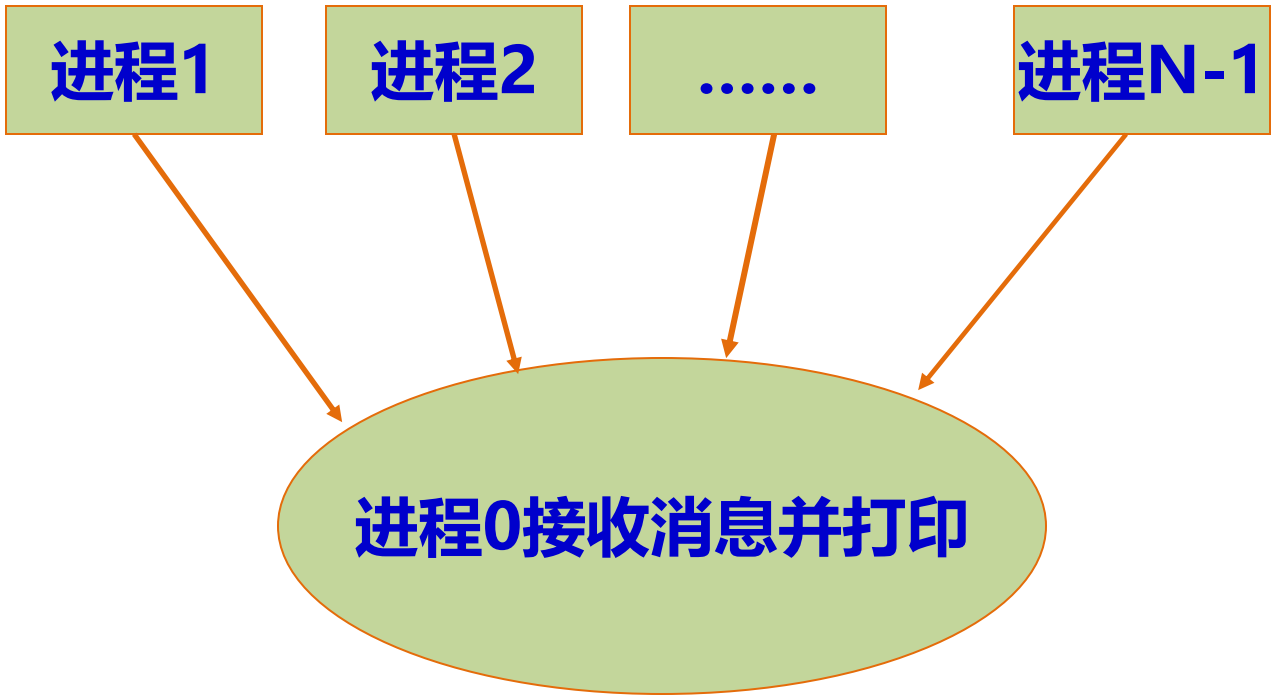
消息来源：可能是进程1，也可能是进程2

消息来源：可能是进程1，也可能是进程2

消息发送给进程0

消息发送给进程0

进程0接收任意源和任意标签消息



分析greetings

```
#include <stdio.h>
#include "mpi.h”
int main(int argc, char *argv[])
{
    int numprocs;        /*进程数,该变量为各处理器中的同名变量,存储是分布的*/
    int myid;            /*进程ID,存储也是分布的                          */
    MPI_Status status;   /*消息接收状态变量,存储也是分布的                */
    char message[100];   /*消息buffer,存储也是分布的                      */
    /*初始化MPI*/
    MPI_Init( &argc, &argv );
    /*该函数被各进程各调用一次,得到自己的进程rank值*/
    MPI_Common_rank(MPI_COMMON_WORLD,&myid);
    /*该函数被各进程各调用一次,得到进程数*/
    MPI_Common_size(MPI_COMMON_WORLD,&numprocs);
```

分析greetings

```
  if (myid != 0) {
     /*建立消息*/
     sprintf(message, "Greetings from process %d!",myid);
     /* 发送长度取strlen(message)+1,使\0也一同发送出去*/
          MPI_Send(message,strlen(message)+1,MPI_CHAR,
               0,99,MPI_COMM_WORLD);
  }
  else{ /*myrank == 0*/
    for (source = 1; source < numprocs; source++) {
       MPI_Recv(message, 100, MPI_CHAR, source, 99,
               MPI_COMM_WORLD, &status);
       printf("%s\n", message);
    }
  }
   /*关闭MPI,标志并行代码段的结束*/
  MPI_Finalize();
} /* end main */
```

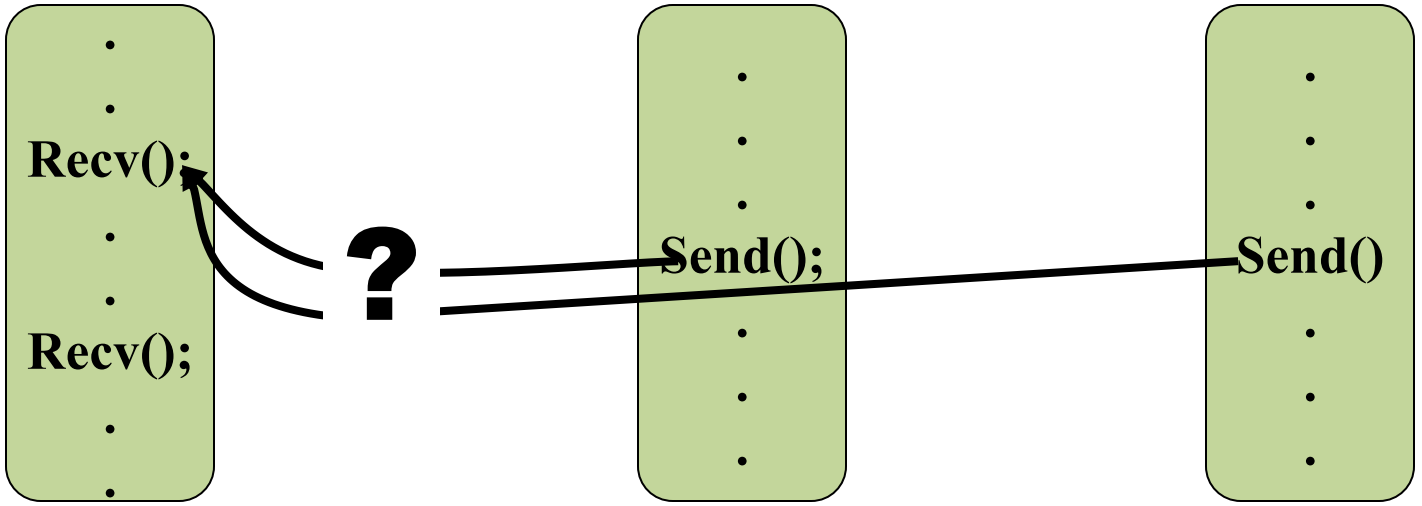
Greetings执行过程

假设进程数为3

(进程0)
(rank=0)

(进程1)
(rank=1)

(进程2)
(rank=2)



问题:进程1和2谁先开始发送消息?谁先完成发送?

运行greetings

```
[haomeng@ubuntu ~]$ mpicc –o greeting greeting.c
[haomeng@ubuntu ~]$ mpirun -np 4 ./greeting
    Greetings from process 1 of 4!
    Greetings from process 2 of 4!
    Greetings from process 3 of 4!
[haomeng@ubuntu ~]$
```

计算机打印字符

我们输入的命令

利用MPI_ANY_SOURCE

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<mpi.h>
4
5  const int MAX_STRING=100;
6
7  int main(){
8      char greeting[MAX_STRING];
9      int comm_sz; //线程的个数
10     int my_rank;
11
12     MPI_Init(NULL,NULL);
13     MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);
14     MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
15
16     if(my_rank!=0){
17         sprintf(greeting,"Greetings from process %d of %d!",my_rank,comm_sz);
18         MPI_Send(greeting,strlen(greeting)+1,MPI_CHAR,0,0,MPI_COMM_WORLD);
19     }else{
20         for(int q=1;q<comm_sz;q++){
21             MPI_Recv(greeting,MAX_STRING,MPI_CHAR,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
22             printf("%s\n",greeting);
23         }
24     }
25
26     MPI_Finalize();
27     return 0;
28 }
```

运行greetings

```
[haomeng@ubuntu ~]$ mpicc –o greeting greeting.c
[haomeng@ubuntu ~]$ mpirun -np 4 ./greeting
    Greetings from process 3 of 4!
    Greetings from process 1 of 4!
    Greetings from process 2 of 4!
[haomeng@ubuntu ~]$
```

计算机打印字符

我们输入的命令

最基本的MPI程序

- MPI函数的总数虽然庞大，但根据实际编写MPI的经验，常用的MPI调用的个数确实有限

- 下面是6个最基本也是最常用的MPI函数

  MPI_Init(…)
  MPI_Comm_size(…)
  MPI_Comm_rank(…)
  MPI_Send(…)
  MPI_Recv(…)
  MPI_Finalize()

- 下面更多的了解MPI

阻塞与非阻塞通信

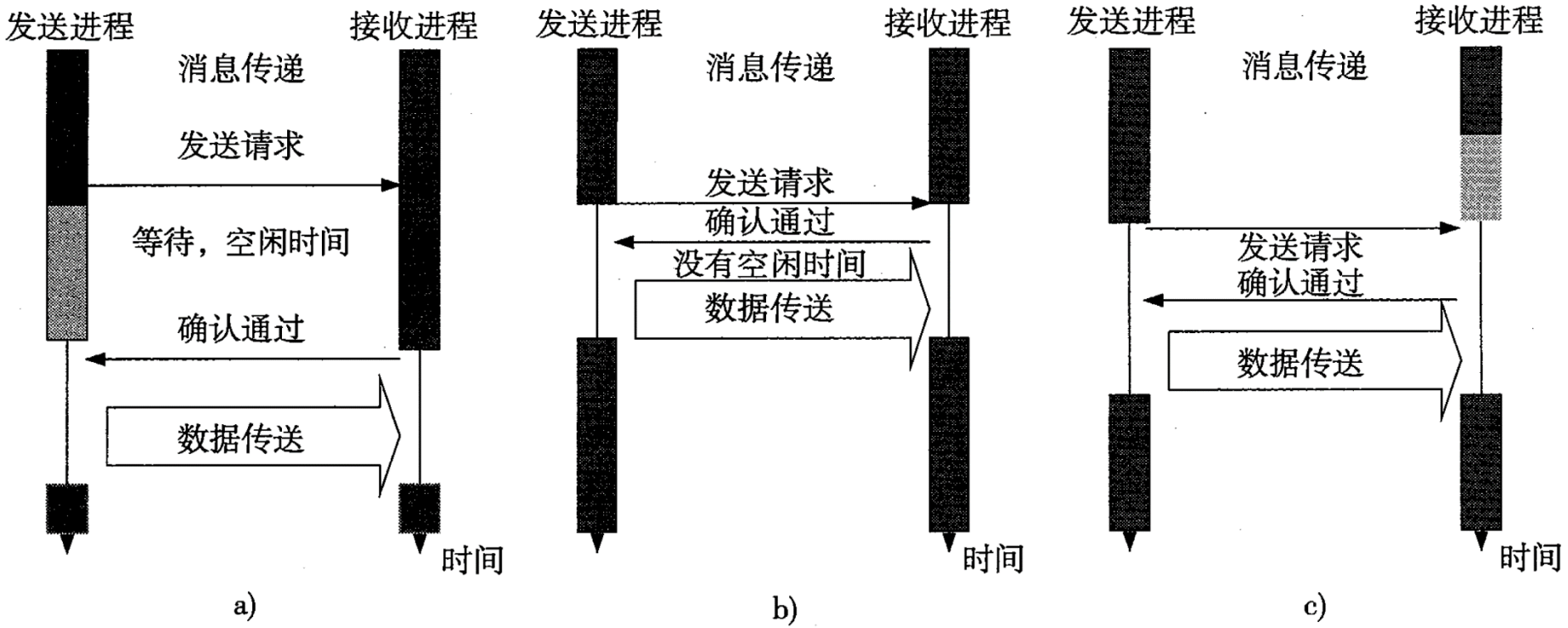
- MPI的点对点通信(Point-to-Point Communication)同时提供了阻塞和非阻塞两种通信机制，也支持多种通信模式

- 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

阻塞与非阻塞通信

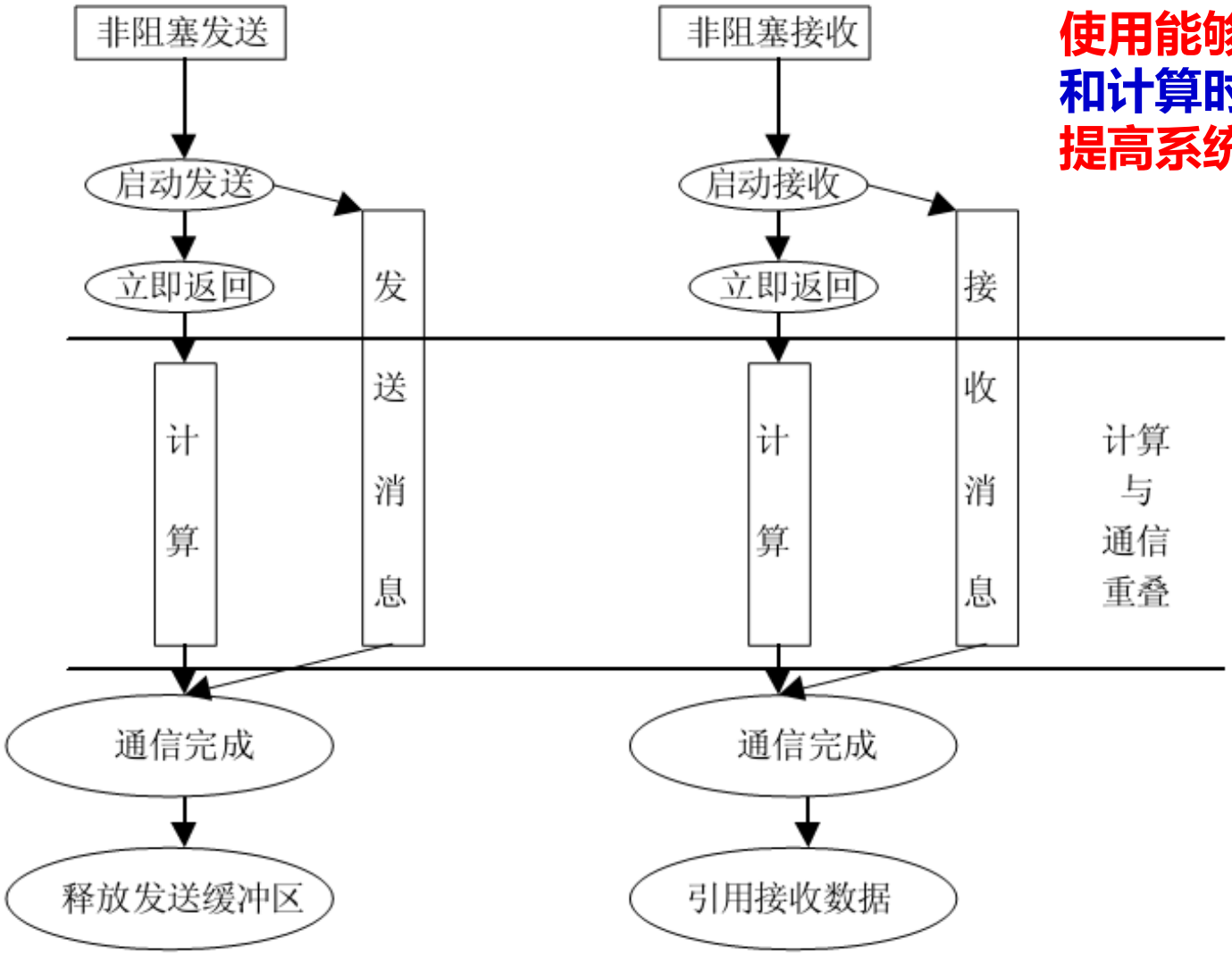
- 阻塞和非阻塞通信的主要区别在于返回后的资源可用性
- 阻塞通信返回的条件:
  - 通信操作已经完成。即消息已经发送或接收
  - 调用的缓冲区可用。若是发送操作，则该缓冲区已经可以被其它的操作更新；若是接收操作，该缓冲区的数据已经完整，可以被正确引用
- 非阻塞通信返回后并不意味着通信操作的完成，MPI还提供了对非阻塞通信完成的检测，主要的有两种：MPI_Wait函数和MPI_Test函数

阻塞通讯：握手模式


发送进程
接收进程
消息传递
发送请求
等待，空闲时间
确认通过
数据传送
时间
a)
发送进程
接收进程
消息传递
发送请求
确认通过
没有空闲时间
数据传送
时间
b)
发送进程
接收进程
消息传递
发送请求
确认通过
数据传送
时间
c)

非阻塞标准发送和接收

非阻塞通信机制的使用能够重叠通信和计算时间，从而提高系统的性能



MPI_Isend点对点通信

```
int MPI_Isend(void *buf, int count,
              MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
```

参数	含义
buf	所发送消息的内存首地址
count	传递的数据的长度
datatype	变量类型，MPI预定义的变量类型
dest	接收消息的进程的标识号
tag	消息标签，用于识别消息
comm	通信器，数据在其中进程间传输
request	非阻塞通信完成对象

MPI_Irecv点对点通信

```
int MPI_Irecv(void *buf, int count,
              MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

参数	含义
buf	接收消息数据的首地址
count	接收数据的最大个数
datatype	接收数据的数据类型
source	发送消息的进程的标识号
tag	消息标签，用于识别消息
comm	通信器，数据在其中进程间传输
request	非阻塞通信完成对象

判断非阻塞通信完成

- **发送的完成**：代表发送缓冲区中的数据已送出，发送缓冲区可以重用。它并不代表数据已被接收方接收。数据有可能被缓冲
- **接收的完成**：代表数据已经写入接收缓冲区。接收者可访问接收缓冲区

阻塞型函数，必须等待指定的通信请求完成后才能返回

```
int MPI_Wait(MPI_Request* request,
             MPI_Status * status)
```

```
int MPI_Test(MPI_Request * request, int * flag,
             MPI_Status * status)
```

检测指定的通信请求，不论通信是否完成都立刻返回。若通信已经完成则返回flag=true，如果通信还未完成则返回 flag=false

MPI_Wait应用示例

```
MPI_Request request;
MPI_Status status;
int x,y;
if(rank == 0){
        x = func1();
        MPI_Isend(&x, 1, MPI_INT, 1, 99, comm, &request)
        …
        MPI_Wait(&request,&status);
        x = func2();
        …
}else{
        MPI_Irecv(&y, 1, MPI_INT, 0, 99, comm, &request)
        …
        MPI_Wait(&request, &status);
        func3(y);
}
```

MPI_Test应用示例

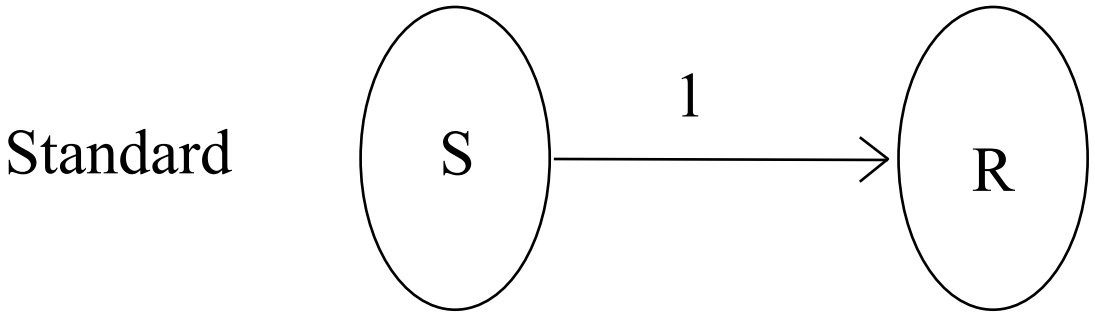
```
MPI_Request request;
MPI_Status status;
int x,y,flag;
if(rank == 0){
        x = func1();
        MPI_Isend(&x, 1, MPI_INT, 1, 99, comm, &request);
         … //computing
        while(!flag)
                MPI_Test(&request, &flag, &status);
        x = func2();
}else{
        MPI_Irecv(&y, 1, MPI_INT, 0, 99, comm, &request);
        … //computing
        while(!flag)
                MPI_Test(&request, &flag, &status);
        func3(y);
}
```

通信模式

- **通信模式**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：
  - **标准**(standard)通信模式
  - **缓冲**(buffered)通信模式
  - **同步**(synchronous)通信模式
  - **就绪**(ready)通信模式

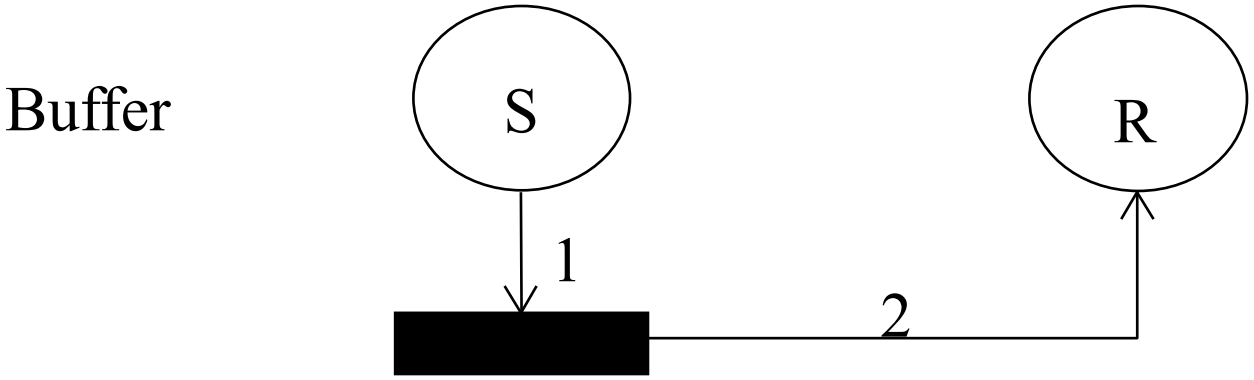
标准通信模式

- 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
- 对应MPI_Send



缓冲通信模式

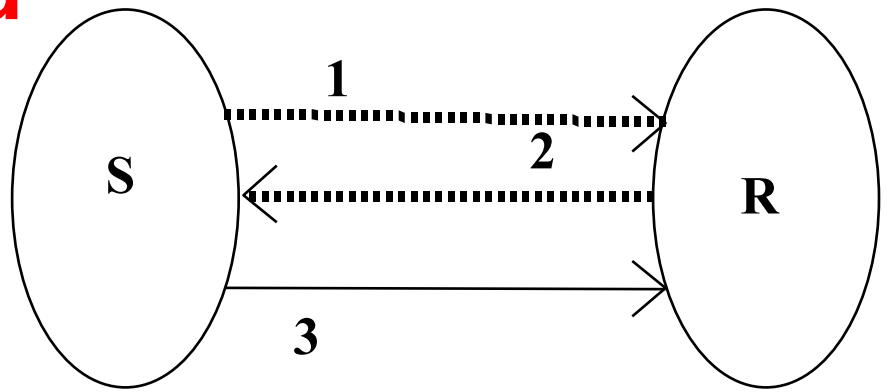
- 缓冲通信模式的发送不管接收操作是否已经启动都可以执行
- 需要用户程序事先申请一块足够大的缓冲区，通过MPI_Buffer_attach实现，通过MPI_Buffer_detach来回收申请的缓冲区
- 对应MPI_Bsend



同步通信模式

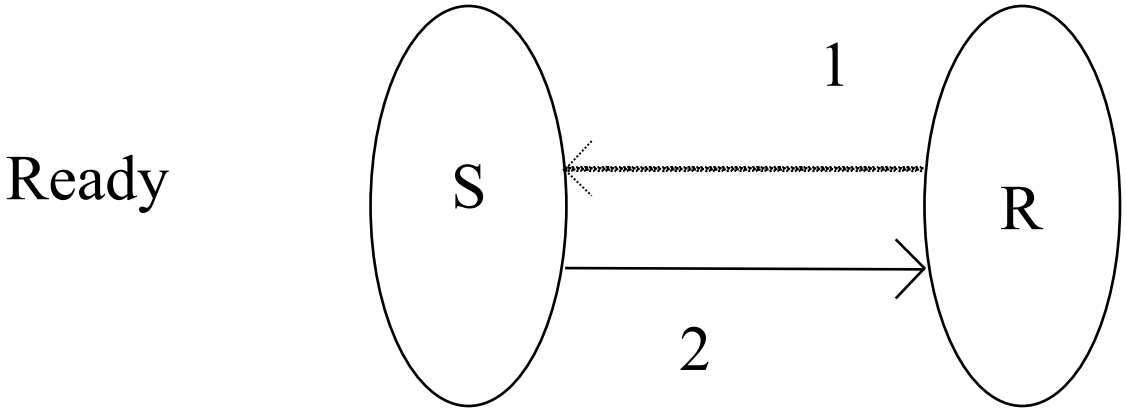
- 只有相应的接收过程已经启动，发送过程才正确返回
- 同步发送返回后，表示发送缓冲区中的数据已经全部被系统缓冲区缓存，并且已经开始发送
- 当同步发送返回后，发送缓冲区可以被释放或者重新使用
- 对应MPI_Ssend

Synchronous



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
- 对应MPI_Rsend



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
- MPI的接收操作有两种：阻塞接收和非阻塞接收

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

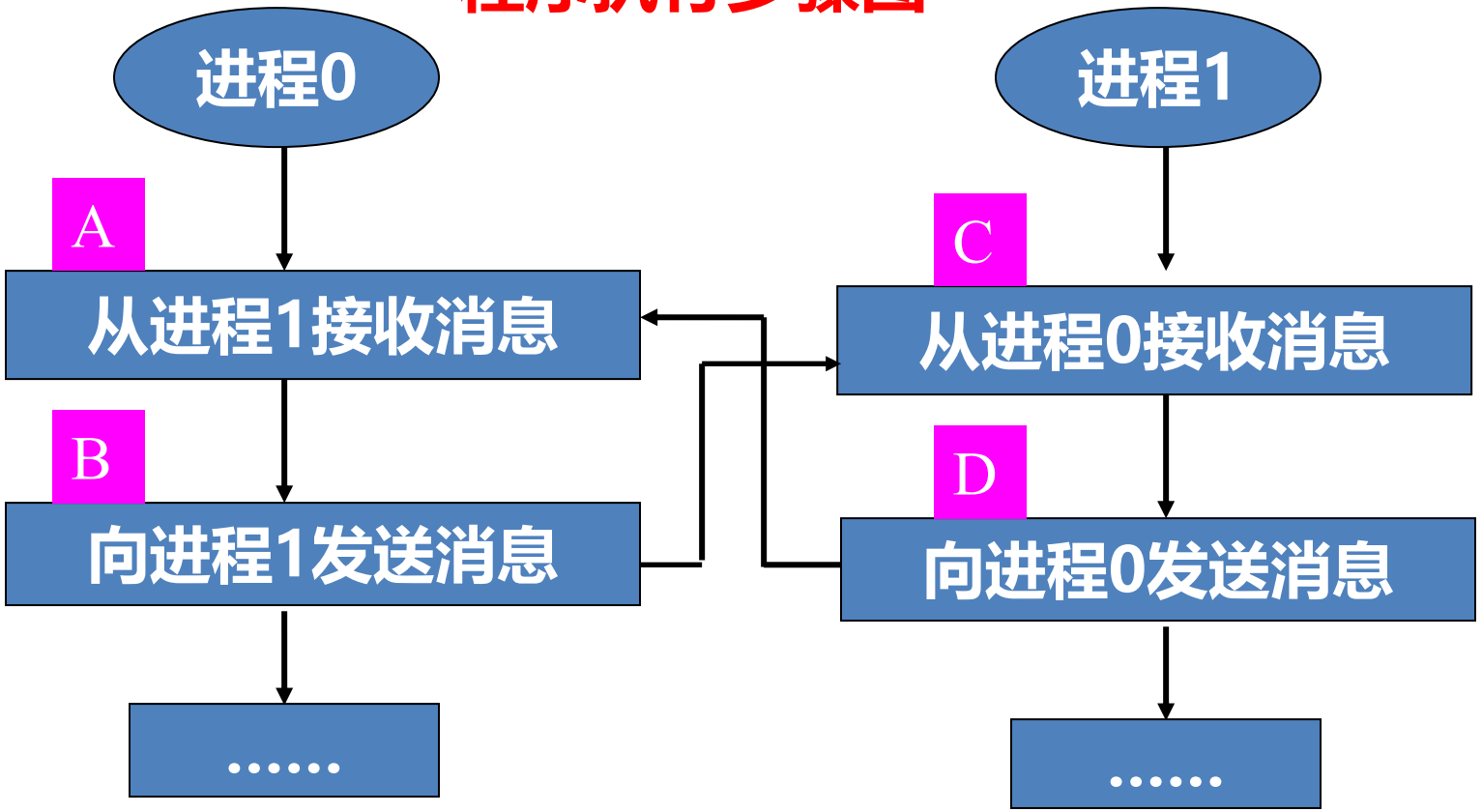
编写安全的MPI程序

假设程序启动时共产生两个进程

```
comm=MPI_COMM_WORLD;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0)
{   MPI_Recv(x2 , 3 , MPI_INT,1,tag,comm,&status);
    MPI_Send(x1 , 3 , MPI_INT,1,tag,comm);  }
if(rank==1)
{   MPI_Recv(x1, 3 ,MPI_INT,0,tag,comm,&status);
    MPI_Send(x2, 3 ,MPI_INT,0,tag,comm);  }
……
```

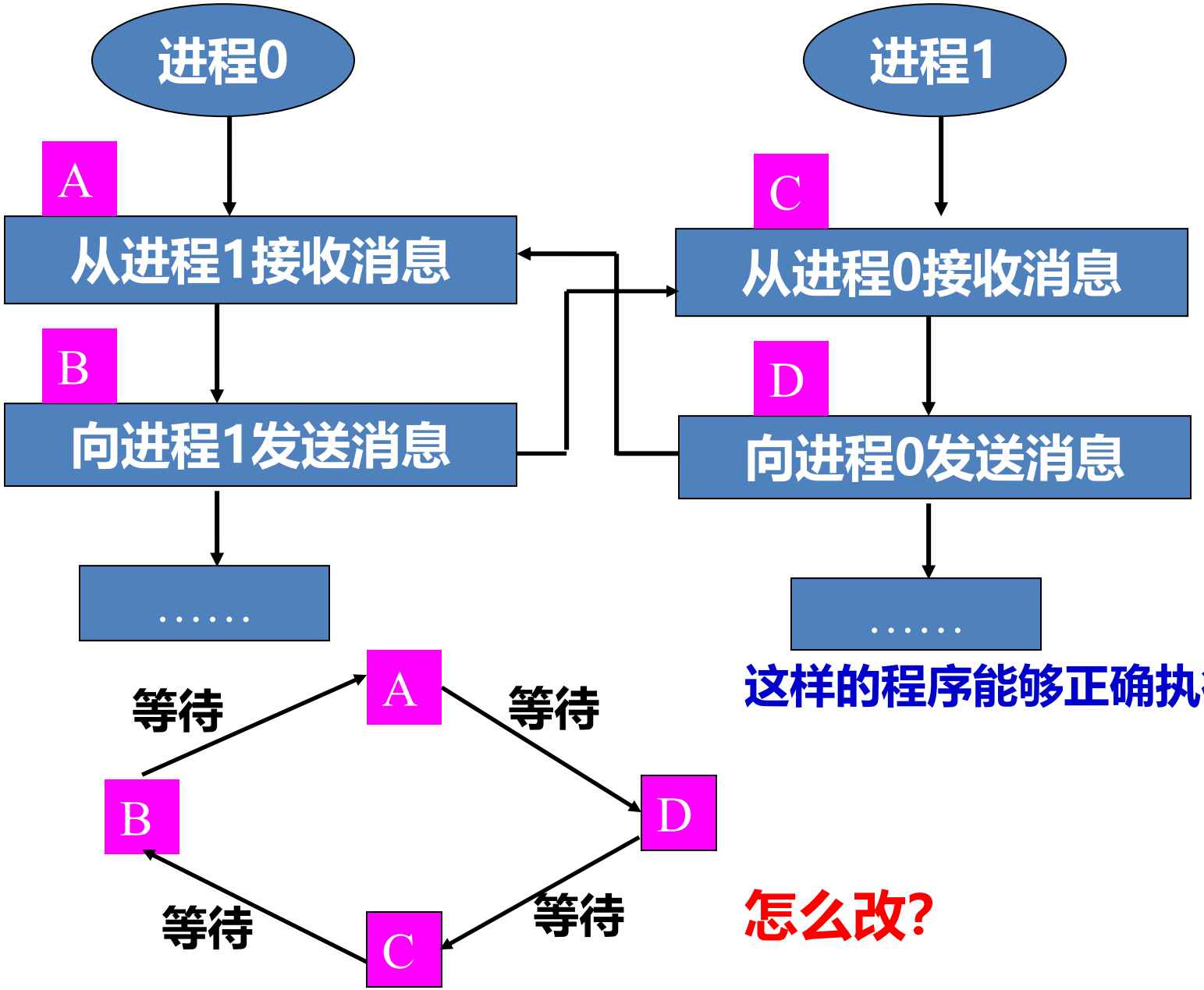
编写安全的MPI程序

程序执行步骤图



程序的预期目标：1、两个进程分别从对方接收一个消息，

2、向对方发送一个消息。


进程0
进程1
A
从进程1接收消息
C
从进程0接收消息
B
向进程1发送消息
D
向进程0发送消息
……
……
这样的程序能够正确执行吗?
等待
A
等待
B
D
等待
C
等待
怎么改?

```
comm=MPI_COMM_WORLD;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0)
{   MPI_Recv(x2 , 3 , MPI_INT,1,tag,comm,&status);
    MPI_Send(x1 , 3 , MPI_INT,1,tag,comm);  }
if(rank==1)
{   MPI_Recv(x1, 3 ,MPI_INT,0,tag,comm,&status);
    MPI_Send(x2, 3 ,MPI_INT,0,tag,comm);  }
……
```

两个进程都是先接收，后发送。

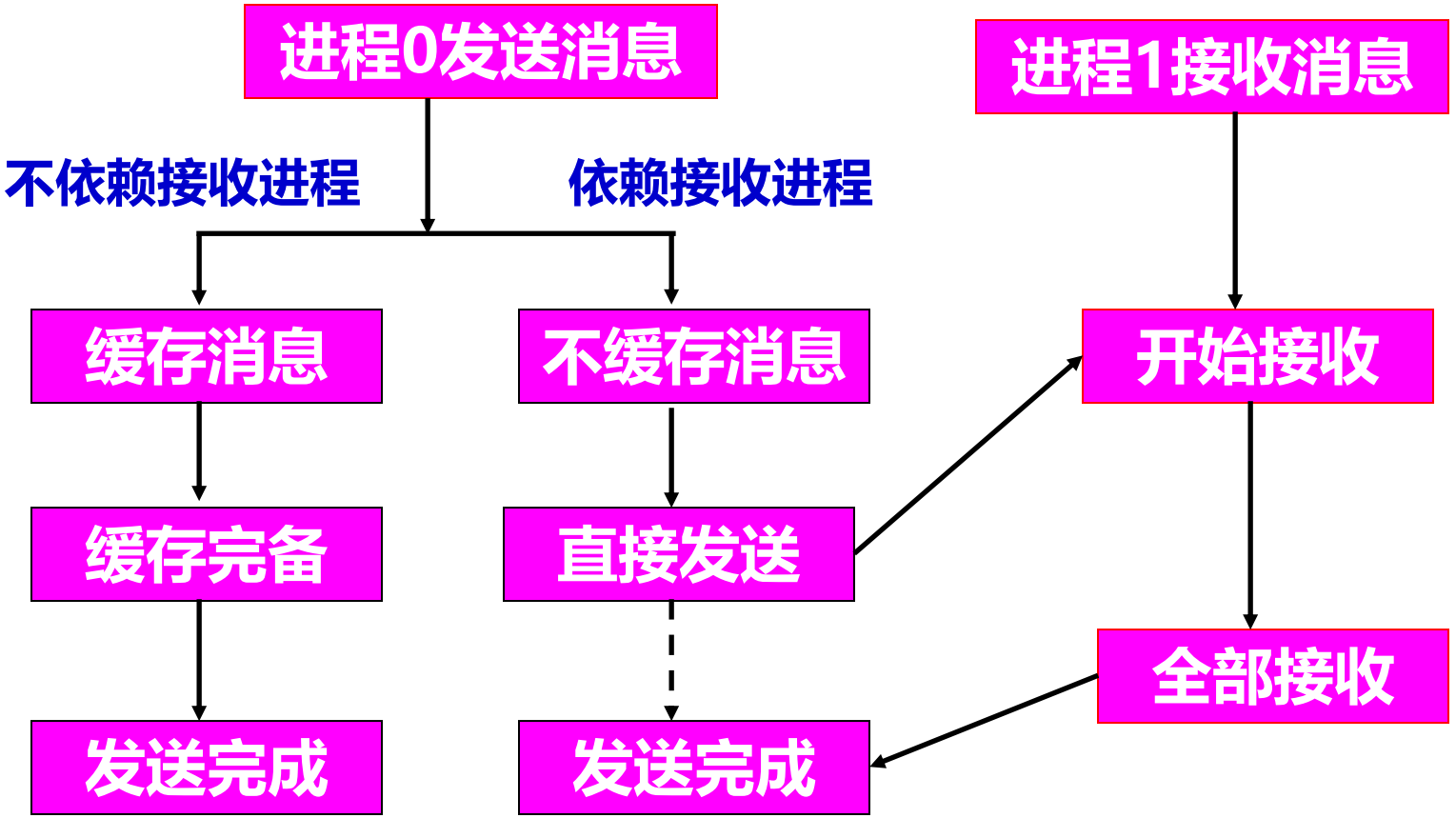
没有发送，何谈接收?　　不合逻辑!

改成：先发送，后接收

```
comm=MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD,&rank)
if(rank==0)
{    MPI_Send(sendbuf,3,MPI_INT,1,tag,comm);
     MPI_Recv(recbuf,3,MPI_INT,1,tag,comm, &status);
}
if(rank==1)
{    MPI_Send(sendbuf,3,MPI_INT,0,tag,comm);
     MPI_Recv(recbuf,3,MPI_INT,0,tag,comm, &status);
}
```

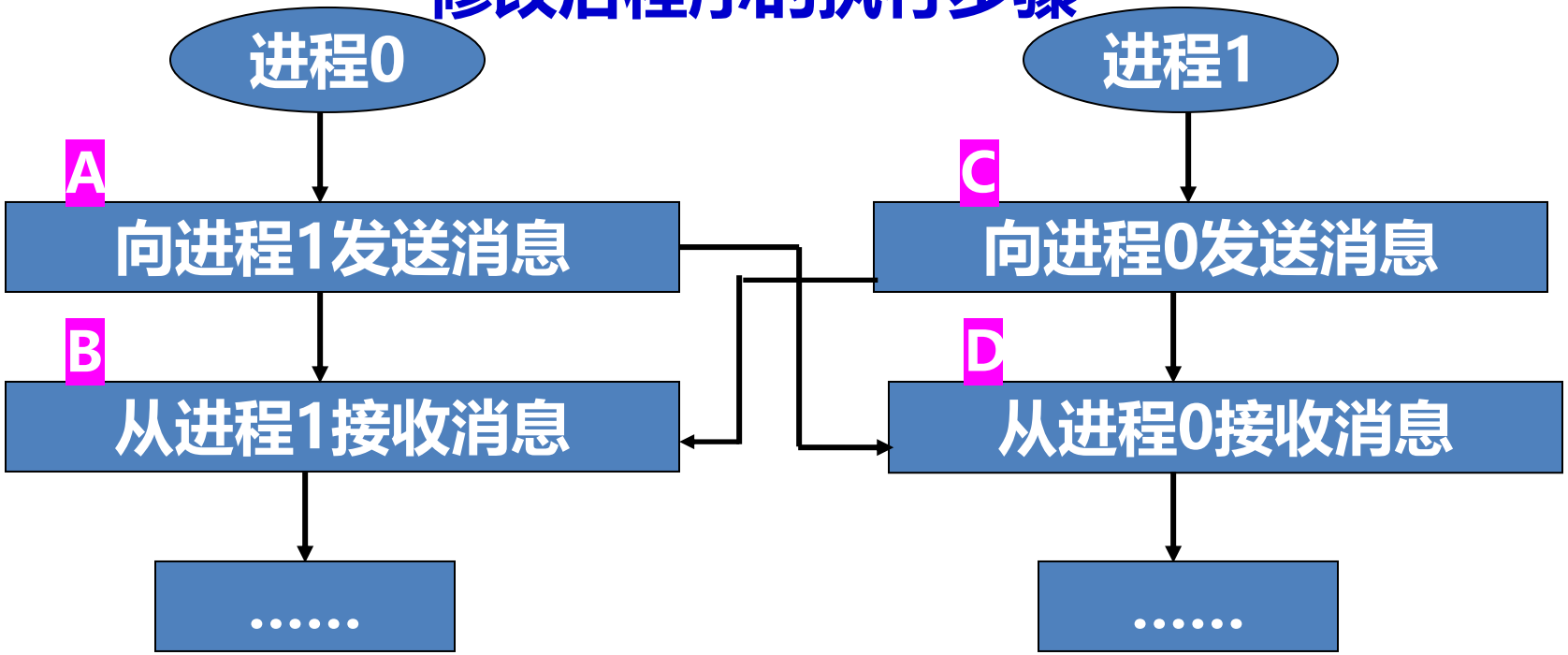
这个程序的性能如何?

欲分析该程序的性能，应先了解上述通信模式中发送与接收操作的执行过程



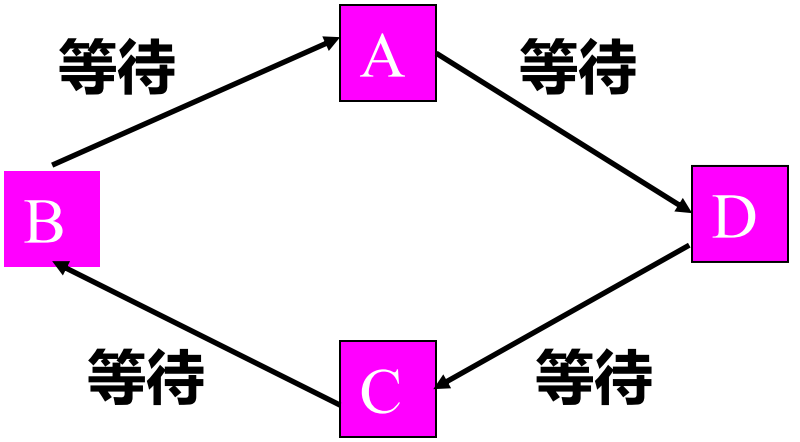
（1）是否缓存数据由MPI决定；（2）如果MPI决定缓存该数据，则发送操作可正确返回而不要求接收操作收到发送的数据；（3）缓存数据要付出代价，缓冲区并非总是可以得到的。

修改后程序的执行步骤

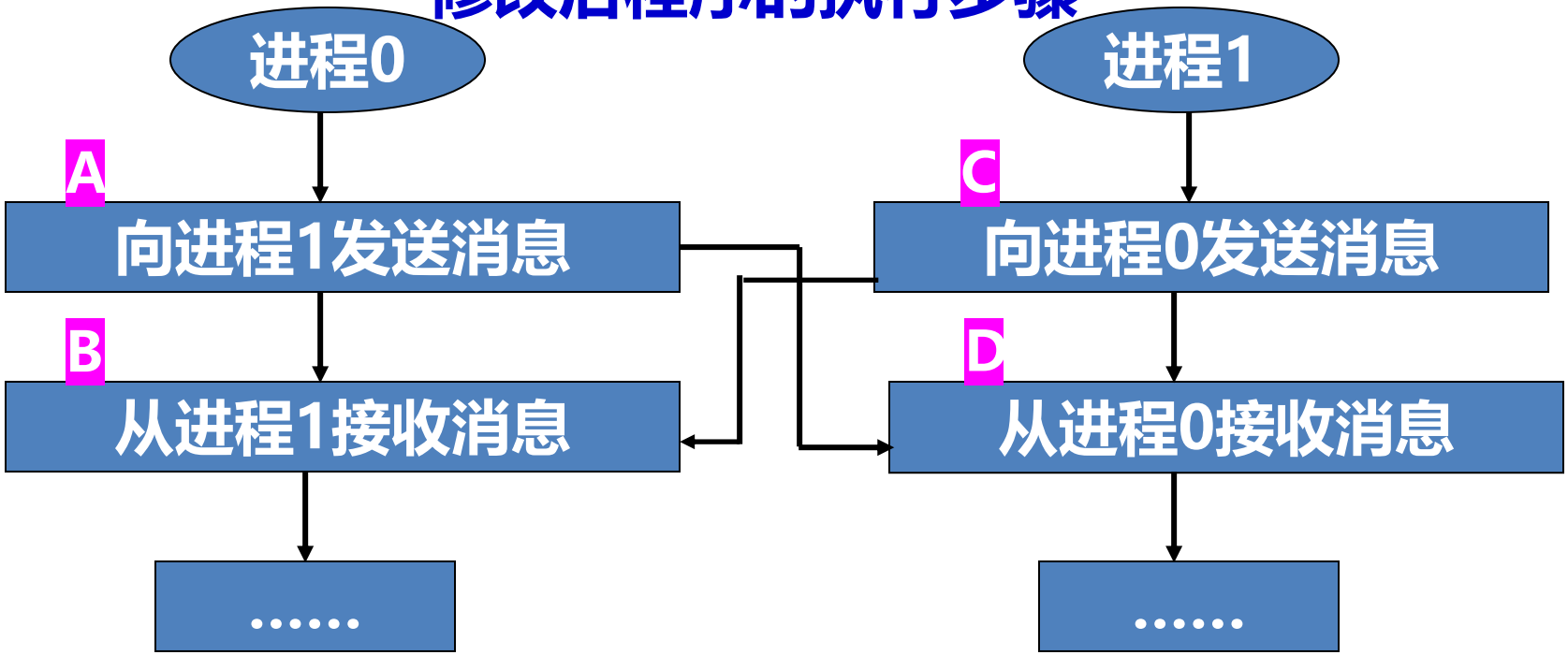


如果系统不缓存消息：

消息死锁!

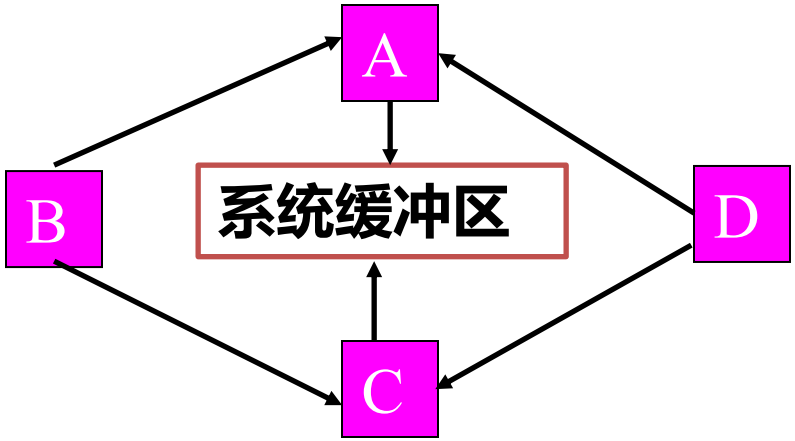


修改后程序的执行步骤



如果系统缓存消息：

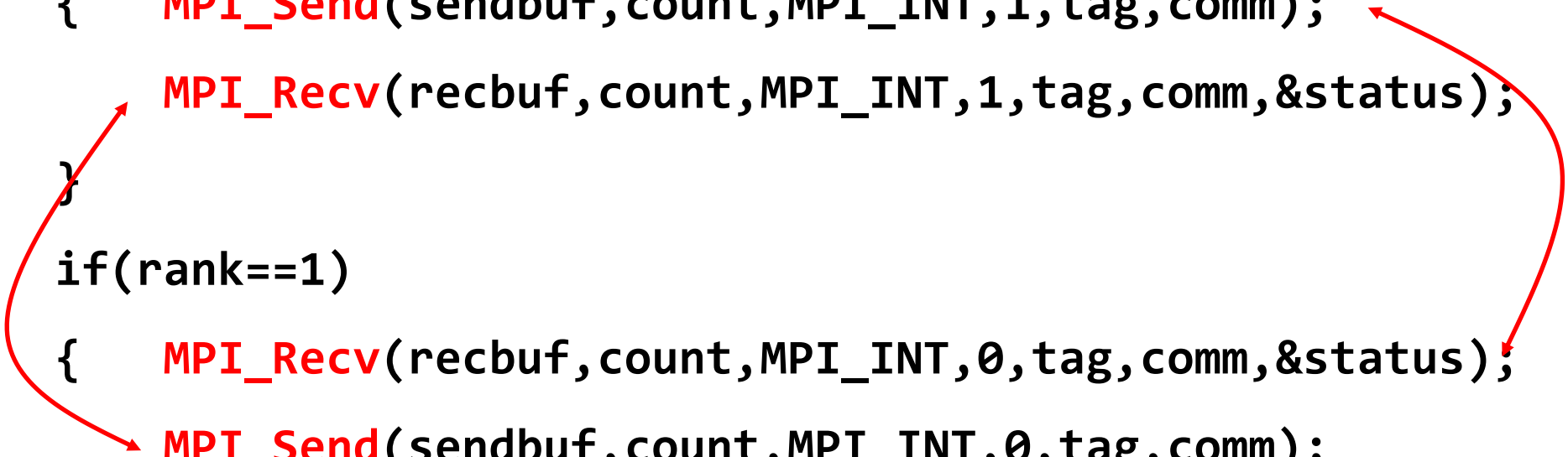
A，C两个发送操作都需要系统缓冲区，如果系统缓冲不足，消息传递将无法完成。

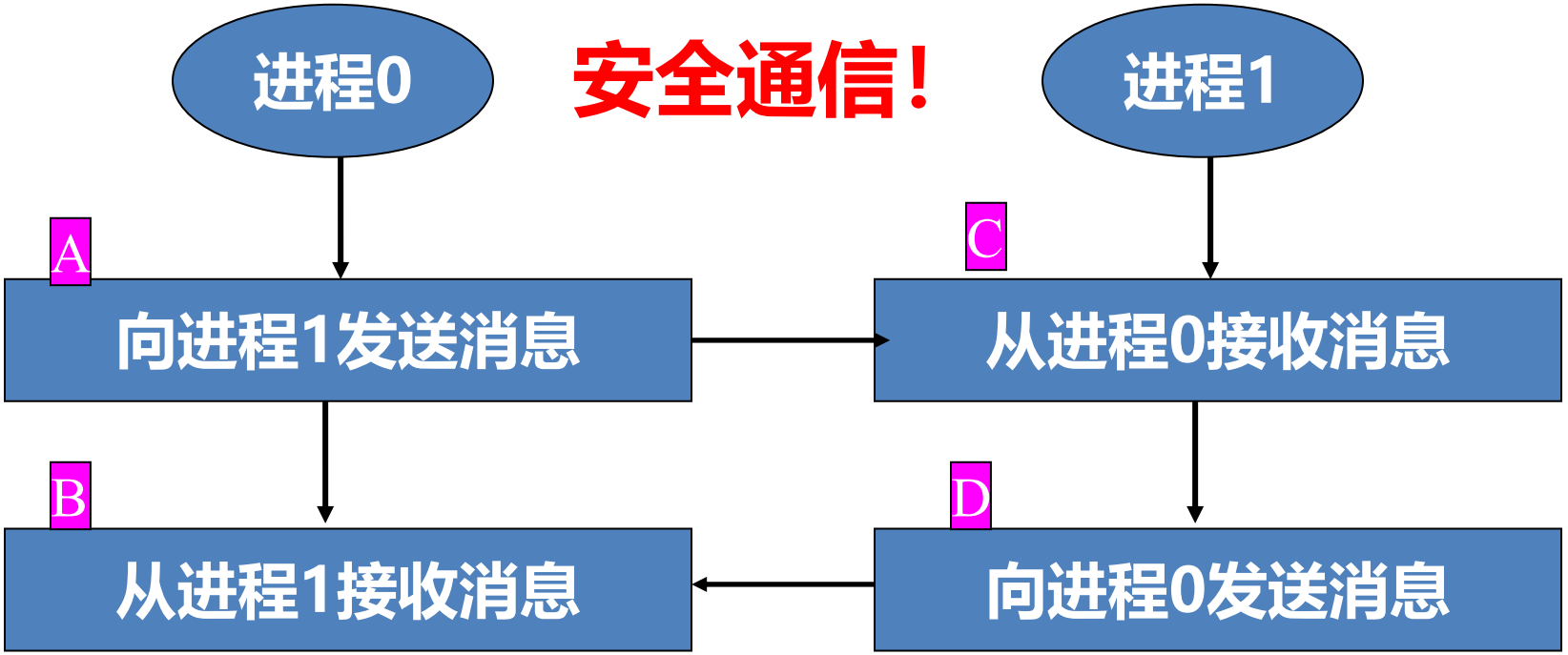


不安全的通信调用次序！

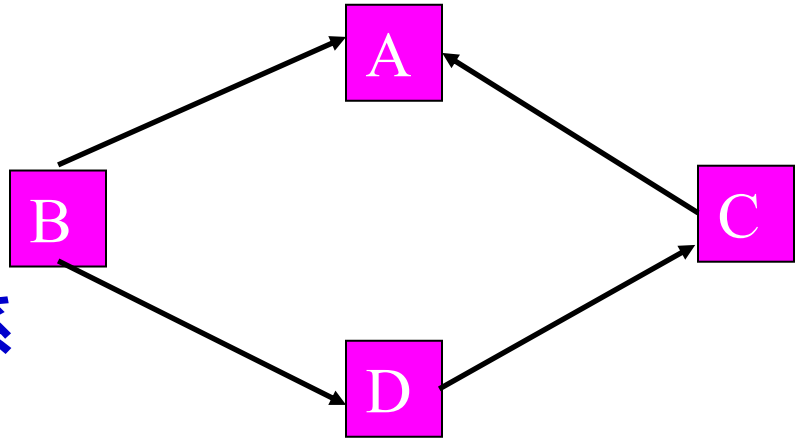
再一次修改：发送与接收操作按次序进行匹配

```
comm=MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD,&rank)
if(rank==0)
{    MPI_Send(sendbuf,count,MPI_INT,1,tag,comm);
     MPI_Recv(recbuf,count,MPI_INT,1,tag,comm,&status);
}
if(rank==1)
{    MPI_Recv(recbuf,count,MPI_INT,0,tag,comm,&status);
     MPI_Send(sendbuf,count,MPI_INT,0,tag,comm);
}
```





1. 只要C存在，则系统不提供缓冲区A也能够执行；
2. C能够执行；
3. A，C完成后，只要B存在，系统不提供缓冲区D也能够执行；
4. B能够执行。



编写安全的MPI程序

- 将发送与接收操作按照次序进行匹配
  - 一个进程的发送操作在前，接收操作在后；
  - 另一个进行的接收操作在前，发送操作在后；
- 若将两个进程的发送与接收操作次序互换，其消息传递过程仍是安全的

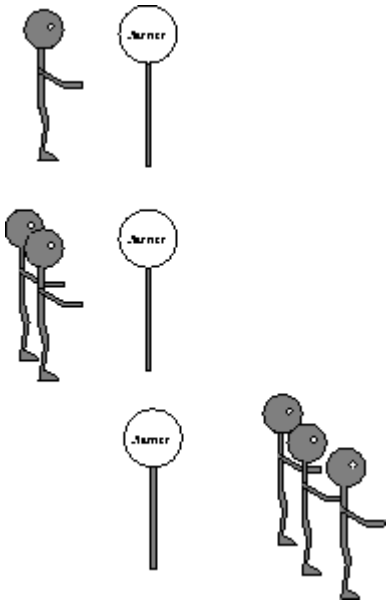
聚合通信

- MPI的聚合 (Collective) 通信，即向一组进程发送消息或从一组进程接收消息
  - 一对多通信：一个进程向其他所有的进程发送消息，这个负责发送消息的进程称为root进程
  - 多对一通信：一个进程负责从其他所有的进程接收消息，这个负责接收消息的进程也称为root进程
  - 多对多通信：每个进程都向其他所有的进程发送或接收消息
- 聚合通信包括：路障(Barrier)、广播通信(Broadcast)、收集通信(Gather)、分散通信(Scatter)、归约通信(Reduction)、全到全广播通信(All-to-all) . . .

聚合通信：路障

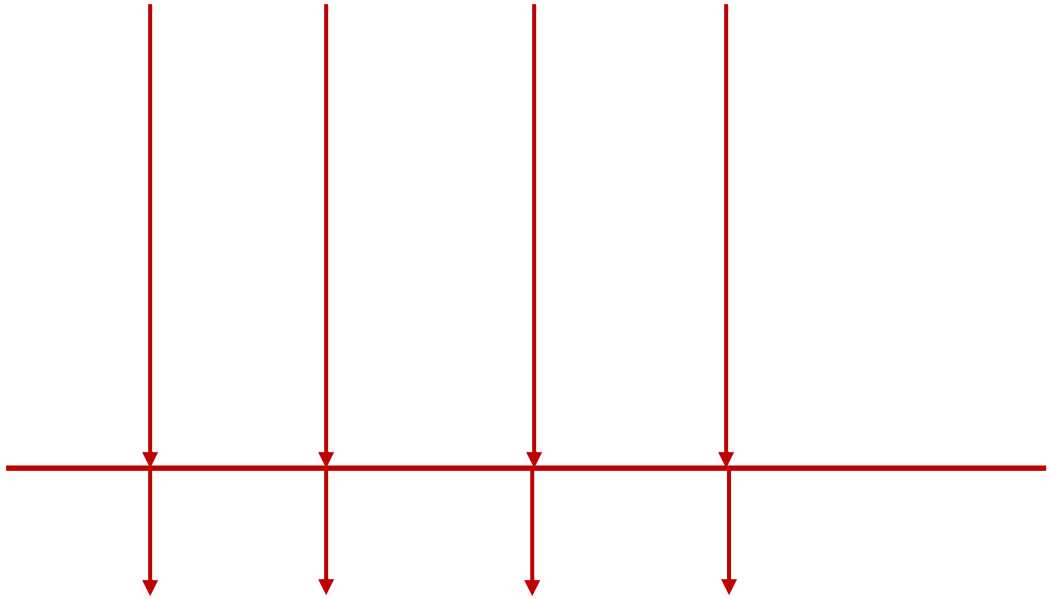
- MPI_Barrier 阻塞通信器中所有进程，直到所有的进程组成员都调用了它。仅当进程组所有的成员都进入了这个调用后，各个进程中这个调用才可以返回

```
int  MPI_Barrier ( MPI_Comm  comm )
```



路障操作会同步多个进程

聚合通信：路障



快的进程必须等待慢的进程，直到所有进程都执行到该语句后才可以向下进行

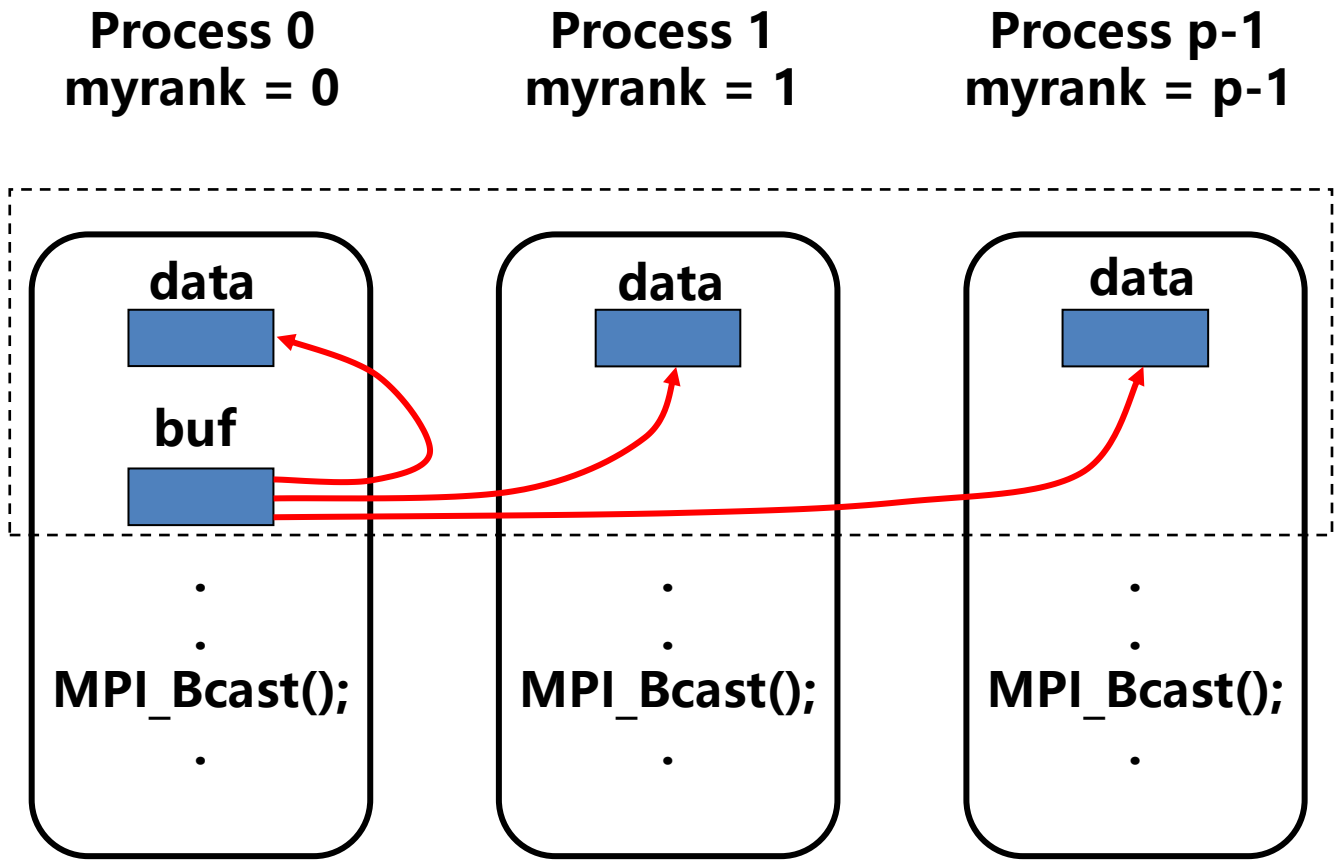
聚合通信：广播通信

- 一到全广播通信 (one-to-all broadcast)
- 从一个进程向一组中的其他进程发送数据

```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

参数	含义
buf	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	通信器，广播通信在通讯器的进程组内进行

聚合通信：广播通信


Process 0
myrank = 0
Process 1
myrank = 1
Process p-1
myrank = p-1
data
buf
data
data
MPI_Bcast();
MPI_Bcast();
MPI_Bcast();

聚合通信：广播通信

```
int p, myrank;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm, &my_rank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);

if(myrank==0)
      buf = 1.0;
MPI_Bcast(&buf, 1, MPI_FLOAT, 0, comm);
```

聚合通信：广播通信

```
int main(int argc, char* argv[]) {
    int myid, size;
    MPI_Init(&argc, &argv);
    char message[124]="my name is rank0";
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (myid == 0){ //0进程作为根进程
        MPI_Bcast(message, 124, MPI_CHAR, 0, MPI_COMM_WORLD);
    }
    else{
        //各进程打印接收到的消息
        MPI_Bcast(message, 124, MPI_CHAR, 0,MPI_COMM_WORLD);
        printf("rank %d received message is: %s\n", myid, message);
    }
    MPI_Finalize();
    return 0;
}
```

聚合通信：广播通信

```
int main(int argc, char* argv[]) {
    int myid, size;
    MPI_Init(&argc, &argv);
    char message[124]="my name is rank0";
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
(base) haomeng@ubuntu:~/projects/mytest/mpi$ mpirun -np 4 ./a.out
rank 2 received message is: my name is rank0
rank 3 received message is: my name is rank0
rank 1 received message is: my name is rank0
```

```
        //各进程打印接收到的消息
        MPI_Bcast(message, 124, MPI_CHAR, 0,MPI_COMM_WORLD);
        printf("rank %d received message is: %s\n", myid, message);
    }
    MPI_Finalize();
    return 0;
}
```

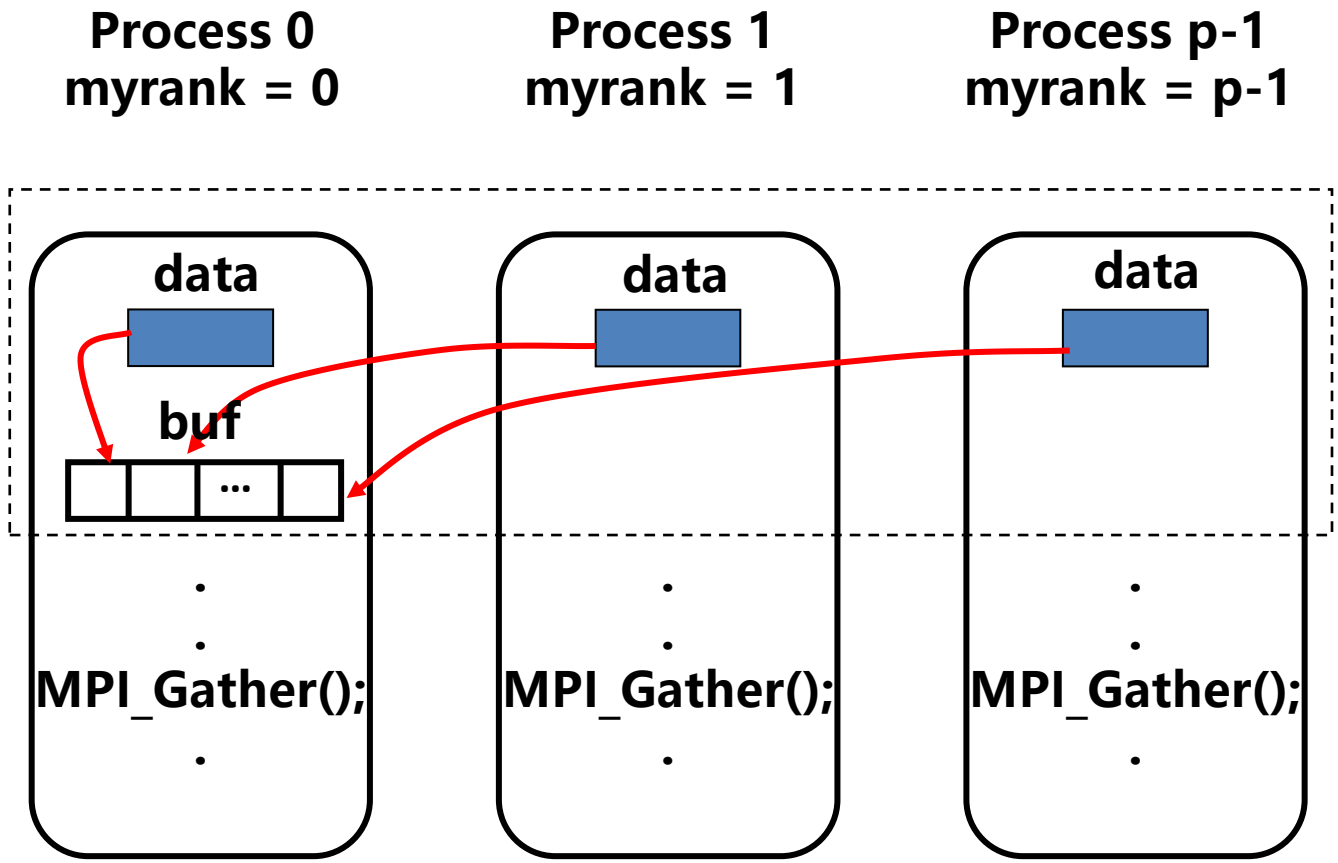
聚合通信：收集通信

- 每个进程(包括根进程)将其发送缓冲区中的内容发送到根进程，根进程根据发送这些数据的进程的序列号将它们依次存放到自己的消息缓冲区中

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype  senddatatype, void *recvbuf, int recvcount,
               MPI_Datatype recvdatatype, int root, MPI_Comm comm)
```

参数	含义
sendbuf	发送消息缓冲区的起始地址
sendcount	发送消息缓冲区中的数据个数
senddatatype	发送消息缓冲区中的数据类型
recvbuf	接收消息缓冲区的起始地址(仅对于根进程有意义)
recvcount	待接收的数据个数(仅对于根进程有意义)
recvdatatype	接收数据的数据类型(仅对于根进程有意义)
root	接收进程的标识号
comm	通信器

聚合通信：收集通信


Process 0
myrank = 0
Process 1
myrank = 1
Process p-1
myrank = p-1
data
data
data
buf
...
MPI_Gather();
MPI_Gather();
MPI_Gather();

聚合通信：收集通信

```
int p, myrank;
float data[10]; /*分布变量*/
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_rank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
...
if(myrank==0){
        /*开辟接收缓冲区*/
        buf=(float*)malloc(p*10*sizeof(float);
}
MPI_Gather(data,10,MPI_FLOAT,buf,10,MPI_FLOAT,0,comm);
```

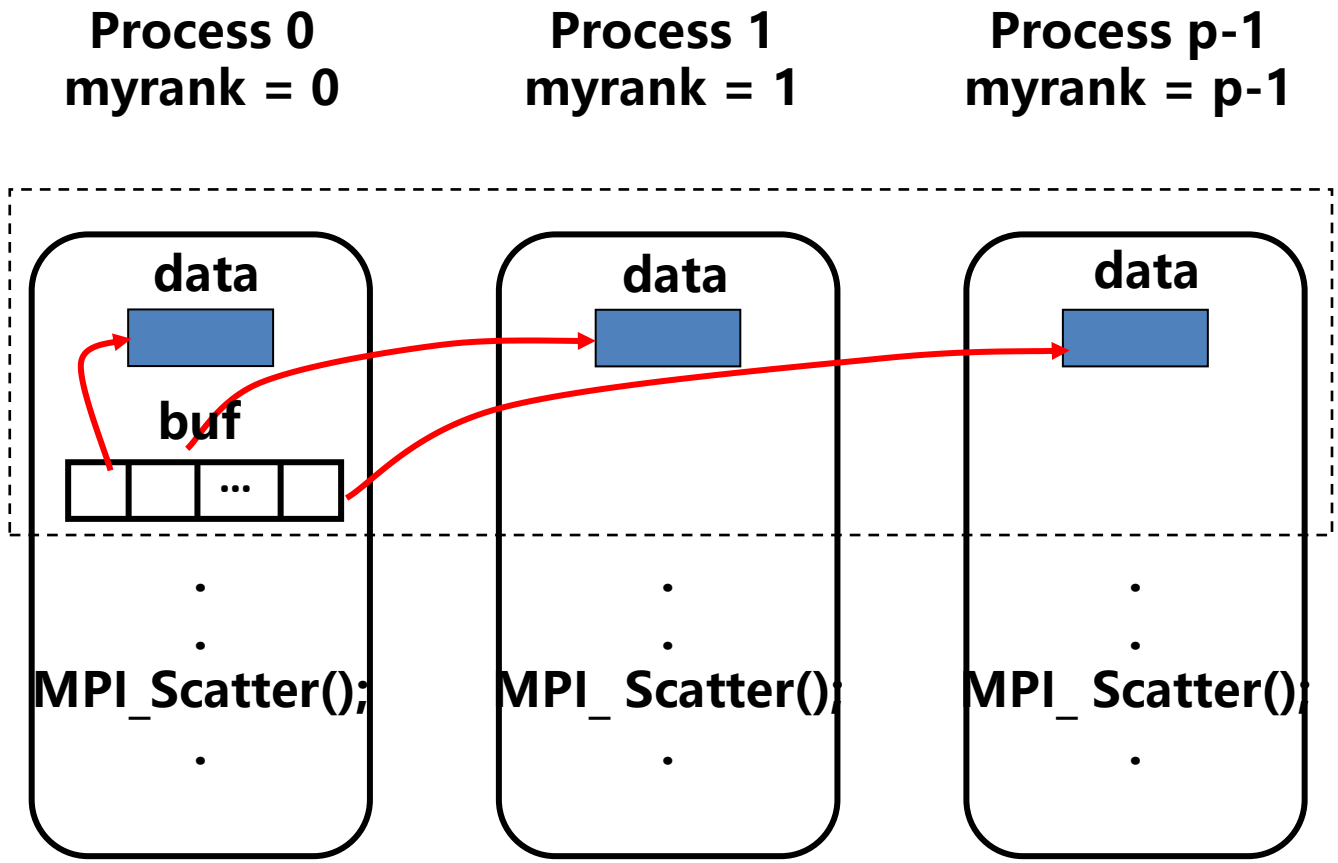
聚合通信：分散通信

- 源（根）进程一个数组中的每个元素发送到不同目标进程。分散 (Scatter)操作是收集 (Gather) 操作的逆操作

```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype  senddatatype, void *recvbuf, int recvcount,
                MPI_Datatype recvdatatype, int root, MPI_Comm comm)
```

参数	含义
sendbuf	发送消息缓冲区的起始地址
sendcount	发送到各个进程的数据个数
senddatatype	发送消息缓冲区中的数据类型
recvbuf	接收消息缓冲区的起始地址
recvcount	待接收的数据个数
recvdatatype	接收数据的数据类型
root	发送进程的标识号
comm	通信器

聚合通信：分散通信


Process 0
myrank = 0
Process 1
myrank = 1
Process p-1
myrank = p-1
data
data
data
buf
...
MPI_Scatter();
MPI_ Scatter();
MPI_ Scatter();

聚合通信：分散通信

```
int p, myrank;
float data[10];
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_rank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
...
if(myrank==0){
      /*开辟发送缓冲区*/
       buf = (float*)malloc(p*10*sizeof(float);
        ...
}
MPI_Scatter(buf,10,MPI_FLOAT,data,10,MPI_FLOAT,0,comm);
```

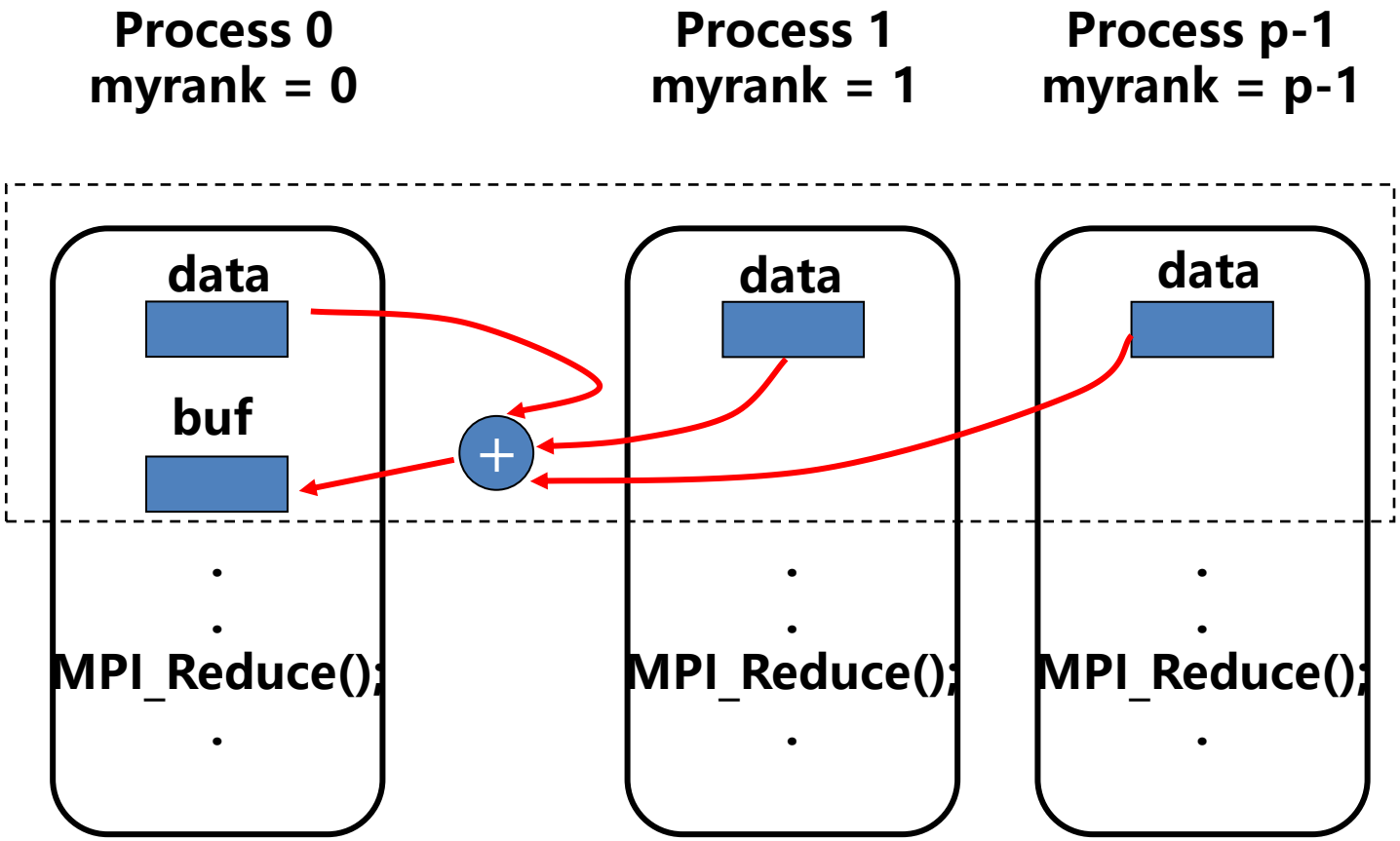
聚合通信：规约通信

- 对组中所有进程的发送缓冲区中的数据用OP参数指定的操作进行运算,并将结果送回到根进程的接收缓冲区中

```
int MPI_Reduce( void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root,
                MPI_Comm comm )
```

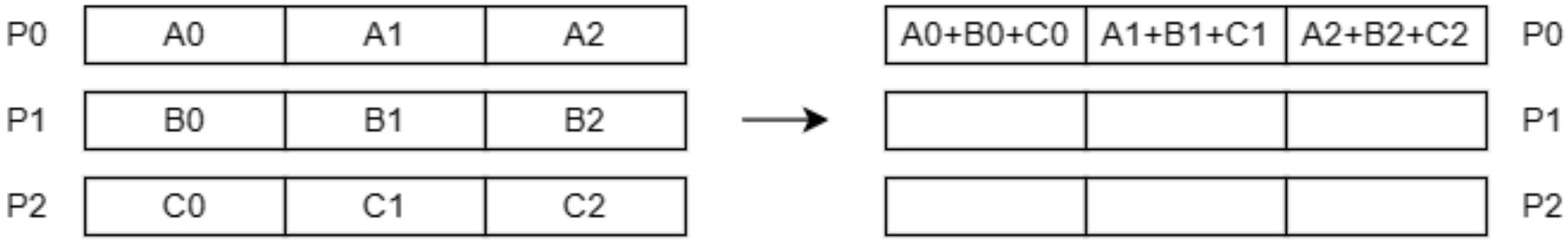
参数	含义
sendbuf	发送消息缓冲区的起始地址
recvbuf	接收消息缓冲区中的地址
count	发送消息缓冲区中的数据个数
datatype	发送消息缓冲区的数据类型
op	归约操作符
root	根进程的标识号
comm	通信器

聚合通信：规约通信


Process 0
myrank = 0
Process 1
myrank = 1
Process p-1
myrank = p-1
data
buf
data
data
+
MPI_Reduce();
MPI_Reduce();
MPI_Reduce();

聚合通信：规约通信

- 所有进程所提供的数据长度相同、类型相同



聚合通信：规约通信

- MPI 中预定义的运算操作（op）

操作名	含义
MPI_MAX	求最大值
MPI_MIN	求最小值
MPI_SUM	求和
MPI_PROD	求积
MPI_LAND	逻辑与
MPI_BAND	二进制按位与

操作名	含义
MPI_LOR	逻辑或
MPI_BOR	二进制按位或
MPI_LXOR	逻辑异或
MPI_BXOR	二进制按位异或
MPI_MAXLOC	求最大值和所在位置
MPI_MINLOC	求最小值和所在位置

- 每种归约运算操作所允许的数据类型

运算操作 OP	允许的数据类型
MPI_MAX, MPI_MIN	整型和实型
MPI_SUM, MPI_PROD	整型、实型和复型
MPI_LAND, MPI_LOR, MPI_LXOR	C的整型
MPI_BAND, MPI_BOR, MPI_BXOR	整型和二进制型(MPI_BYTE)

聚合通信：规约通信

- 归约操作可以使用MPI预定义的运算操作，也可以使用用户自定义的运算操作，必须满足结合律
  - 创建自定义归约运算操作：MPI_Op_create
  - 释放自定义的归约操作: MPI_Op_free

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

参数	含义
func	用户自定义的函数
commute	如果用户自定义的运算可交换则为true, 否则为false
op	运算操作符

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

聚合通信：规约通信

```
int p, myrank;
float data = 0.0;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_rank);

/*各进程对data进行不同的操作*/
data = data + myrank * 10;

/*将各进程中的data数相加并存入根进程的buf中 */
MPI_Reduce(&data,&buf,1,MPI_FLOAT,MPI_SUM,0,comm);
```

聚合通信：其他

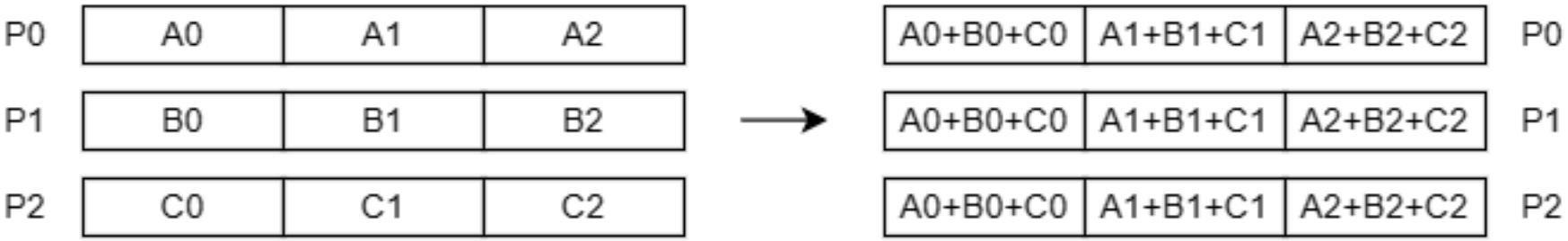
- MPI_Allreduce：全规约函数，归约并将结果发送到所有进程

```
int MPI_Allreduce (void *sendbuf, void *recvbuf,  int count,
                   MPI_Datatype datatype, MPI_Op op,
                   MPI_Comm comm)
```

参数	含义
sendbuf	发送消息缓冲区的起始地址
recvbuf	接收消息缓冲区的起始地址
count	发送消息缓冲区中的数据个数
datatype	发送消息缓冲区中的数据类型
op	归约操作符
comm	通信器

聚合通信：其他

- MPI_Allreduce：全规约函数，归约并将结果发送到所有进程
- 所有进程的recvbuf将同时获得归约运算的结果
- 相当于MPI_Reduce后再将结果进行一次广播MPI_Bcast



聚合通信：其他

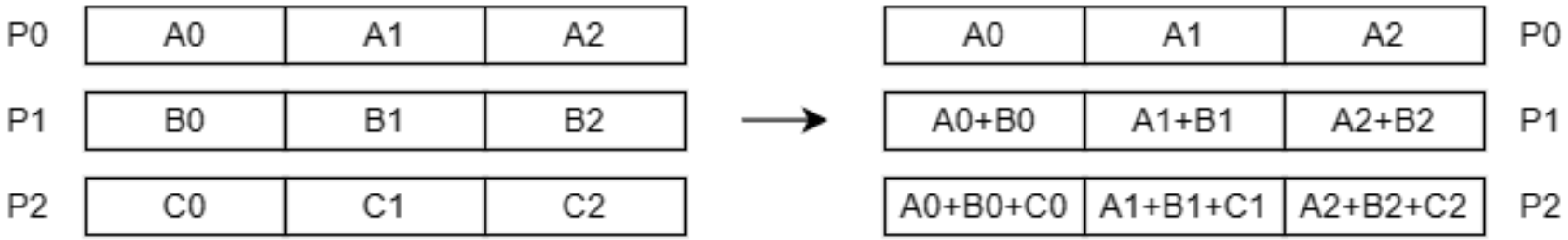
- MPI_Scan：扫描函数，特殊的规约，每一个进程都对排在它前面的进程进行归约操作

```
int MPI_Scan (void * sendbuf, void * recvbuf, int count,
              MPI_Datatype datatype,
              MPI_Op op, MPI_Comm comm)
```

参数	含义
sendbuf	发送消息缓冲区的起始地址
recvbuf	接收消息缓冲区的起始地址
count	输入缓冲区中元素的个数
datatype	输入缓冲区中元素的类型
op	归约操作符
comm	通信器

聚合通信：其他

- MPI_Scan：扫描函数
- 每一个进程都对排在它前面的进程进行归约操作，操作结束后，第i个进程中的recvbuf中将包含前i个进程的归约结果
- 0号进程接收缓冲区中的数据就是其发送缓冲区的数据



聚合通信：其他

- MPI_Allgather：全收集函数，从所有进程收集数据到所有进程（每个进程最后获得相同的所有数据）

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

参数	含义
sendbuf	发送消息缓冲区的起始地址
sendcount	发送消息缓冲区中的数据个数
sendtype	发送消息缓冲区中的数据类型
recvbuf	接收消息缓冲区的起始地址
recvcount	从其它进程中接收的数据个数
recvtype	接收消息缓冲区的数据类型
comm	通信器

聚合通信：其他

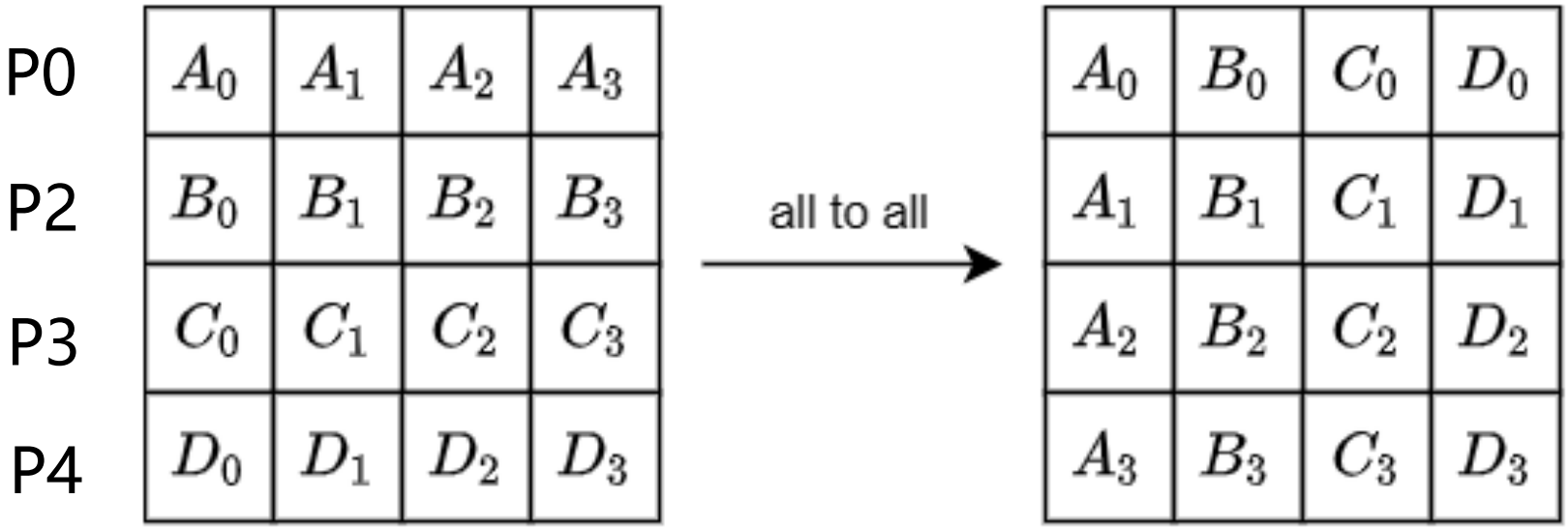
- MPI_Alltoall：全收集分散函数

```
int MPI_ Alltoall (void * sendbuf, int sendcount,
                   MPI_Datatype sendtype,
                   void* recvbuf, int recvcount,
                   MPI_Datatype recvtype, MPI_Comm comm)
```

参数	含义
sendbuf	发送消息缓冲区的起始地址
sendcount	发送到每个进程的数据个数
sendtype	发送消息缓冲区中的数据类型
recvbuf	接收消息缓冲区的起始地址
recvcount	从每个进程中接收的元素个数
recvtype	接收消息缓冲区的数据类型
comm	通信器

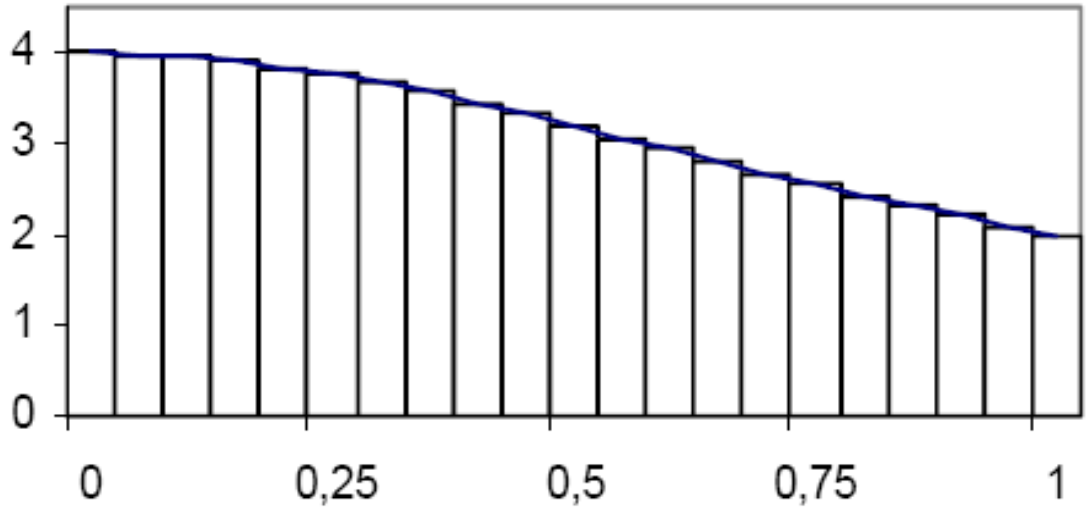
聚合通信：其他

- MPI_Alltoall：全收集分散函数
- 每个进程发送一个消息给n个进程，包括它自己，这n个消息的发送缓冲区中以标号的顺序有序地存放
- 一次全局交换中共有n^2个消息进行通信



示例：计算π

- 通过积分的方法计算 π

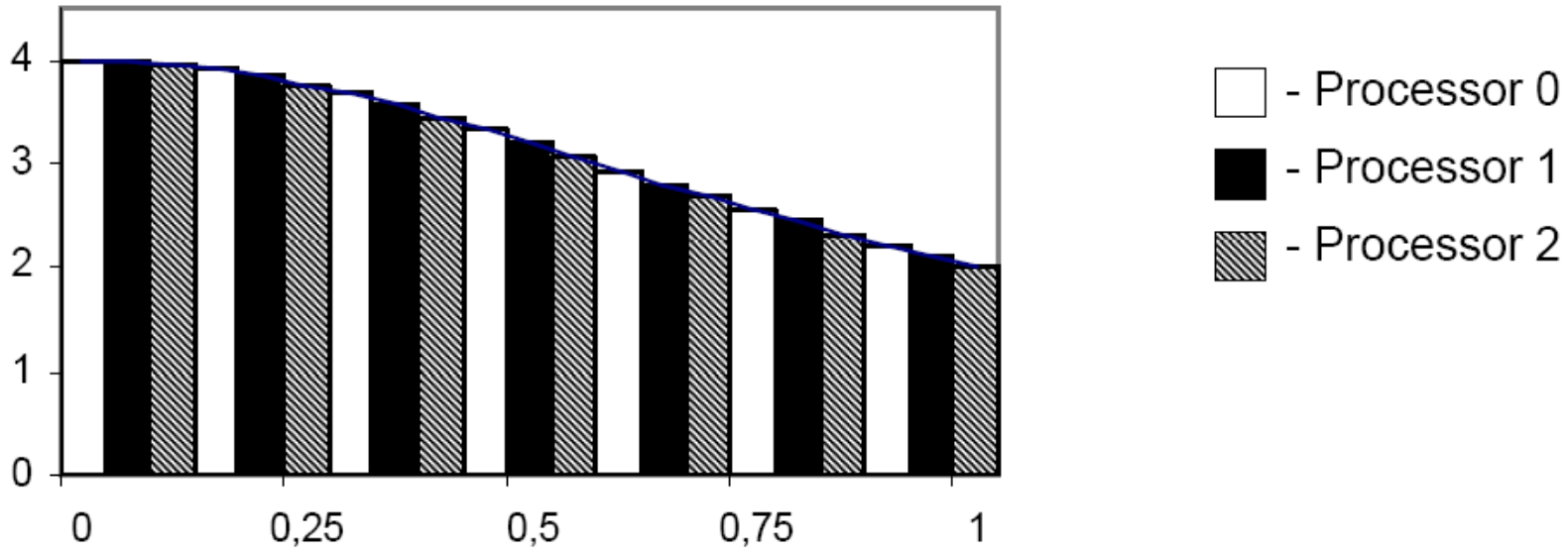


- 使用矩形法来计算数值积分

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

计算π

- 在处理器之间循环分配计算负载
- 在不同处理器上计算的部分和最终必须相加



计算π：串行程序

```
int num_steps = 1000;
double width;
int main ()
{
      int i;
      double x, pi, sum = 0.0;
      width = 1.0 / (double) num_steps;
      for (i=1; i <= num_steps; i++) {
            x = (i-0.5)* width;
            sum = sum + 4.0 / (1.0+x*x);
      }
      pi = sum * width;
      return 0;
}
```

计算π：MPI并行程序（点对点通信）

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Status st;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if(myid == 0){
        printf("Enter the number of intervals: \n");
        scanf("%d", &n);
        for(i = 1; i < numprocs; i++)
            MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);
    }
```

计算π：MPI并行程序（点对点通信）

```
    h = 1.0/(double)n;
    sum = 0.0;
    for(i = myid + 1; i <= n; i+= numprocs){
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    mypi = h * sum;
    if(myid != 0)
        MPI_Send(&mypi, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    else {
        pi = mypi;
        for(i = 1; i < numprocs; i++){
            MPI_Recv(&mypi,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&st);
            pi += mypi;
        }
        printf("pi is approximately %.16f\n", pi);
    }
    MPI_Finalize();
    return 0;
}
```

计算π：MPI并行程序 (聚合通信)

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Status st;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if(myid == 0){
        printf("Enter the number of intervals: \n");
        scanf("%d", &n);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

计算π：MPI并行程序 (聚合通信)

```
    h = 1.0/(double)n;
    sum = 0.0;
    for(i = myid + 1; i <= n; i+= numprocs){
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if(myid == 0)
        printf("pi is approximately %.16f\n", pi);
    MPI_Finalize();
    return 0;
}
```

计时：MPI_Wtime

- 返回从过去某一时刻开始所经过的秒数

```
double MPI_Wtime(void)
```

- 对一个MPI代码块进行计时：

```
double start, finish;
...
start = MPI_Wtime();
/*Code to be timed*/
...
finish = MPI_Wtime();
print("Proc %d > Elapsed time = %e seconds\n",my_rank,
                          finish - start);
```

计时：MPI_Wtime

- 并行程序会为每个进程报告一次时间，但我们需要获得一个总的单独时间
- 理想情况是，所有的进程同时开始运算，当最后一个进程完成运算时，能获取从开始到最后一个进程结束之间的时间开销。换句话讲，并行时间取决于“最慢”进程花费的时间

```
double local_start, local_finish, local_elapsed, elapsed;
...
MPI_Barrier(comm);
local_start = MPI_Wtime();
/*Code to be timed*/
...
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed,&elapsed,1,MPI_DOUBLE,MPI_MAX,0,comm);
if(my_rank == 0)
    print("Elapsed time = %e seconds\n",elapsed);
```