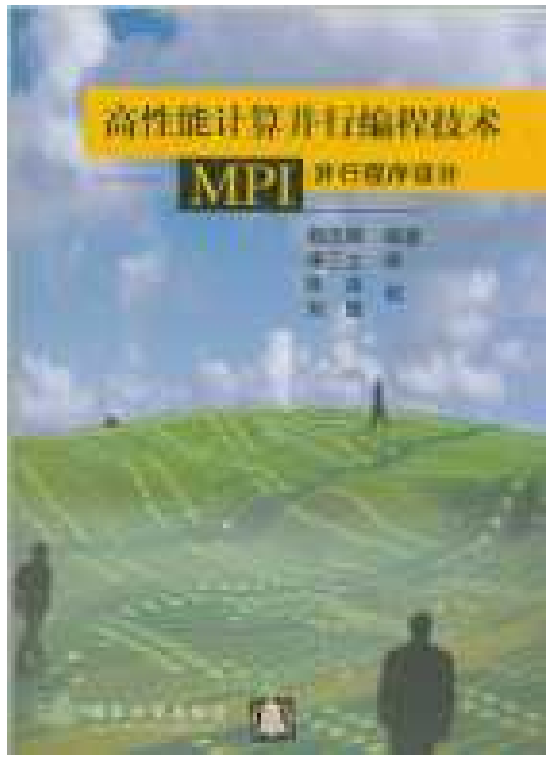
MPI并行编程入门

中国科学院计算机网络信息中心
超级计算中心

参考材料

张林波　清华大学出版社

莫则尧　科学出版社



都志辉　清华大学出版社

消息传递平台MPI

- 什么是**MPI**（Message Passing Interface）
  - 是函数库规范，而不是并行语言；操作如同库函数调用
  - 是一种标准和规范，而非某个对它的具体实现（MPICH等），与编程语言无关
  - 是一种消息传递编程模型，并成为这类编程模型的代表

- **What is the message?**

  DATA+ENVELOPE

- **MPI的目标**
  - 较高的通信性能
  - 较好的程序可移植性
  - 强大的功能

消息传递平台MPI

- **MPI的产生**
  - 1992-1994年，MPI 1.1版本问世
  - 1995-1997年，MPI 2.0版本出现
    - 扩充并行I/O
    - 远程存储访问
    - 动态进程管理等
- **MPI的语言绑定**
  - Fortran（科学与工程计算）
  - C（系统和应用程序开发）
- 主要的**MPI**实现
  - 并行机厂商提供
  - 高校、科研部门
    - **MPICH** (http://www.mcs.anl.gov/mpi/mpich)
    - LAMMPI (http://www.lam-mpi.org/)

消息传递平台MPI

- **MPI程序编译与运行**
  - 程序编译

C: %mpicc -o mpiprog mpisrc.c

Fortran 77: %mpif77 -o mpiprog mpisrc.f

  - 程序运行

%mpirun -np 4 mpiprog

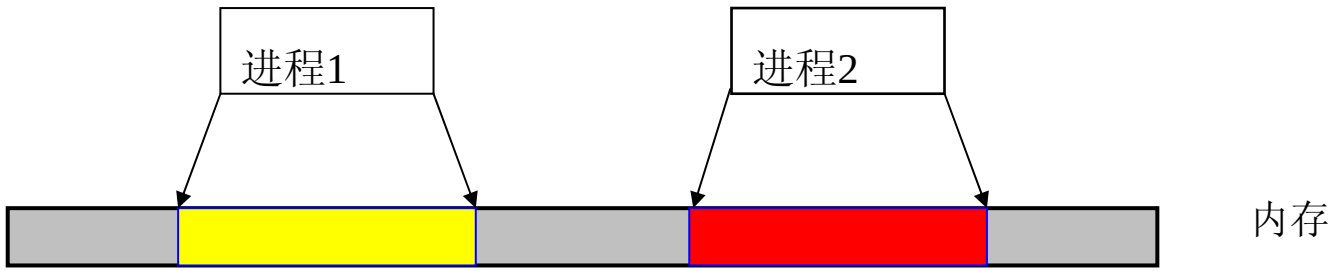
- 程序执行过程中不能动态改变进程的个数
- 申请的进程数np与实际处理器个数无关

MPI基础知识

- 进程与消息传递
- MPI重要概念
- MPI函数一般形式
- MPI原始数据类型
- MPI程序基本结构
- MPI几个基本函数
- 并行编程模式

进程与消息传递

- 单个进程（process）
  - 进程与程序相联，程序一旦在操作系统中运行即成为进程。进程拥有独立的执行环境（内存、寄存器、程序计数器等），是操作系统中独立存在的可执行的基本程序单位
  - 串行应用程序编译形成的可执行代码，分为“指令”和“数据”两个部分，并在程序执行时“独立地申请和占有”内存空间，且所有计算均局限于该内存空间。

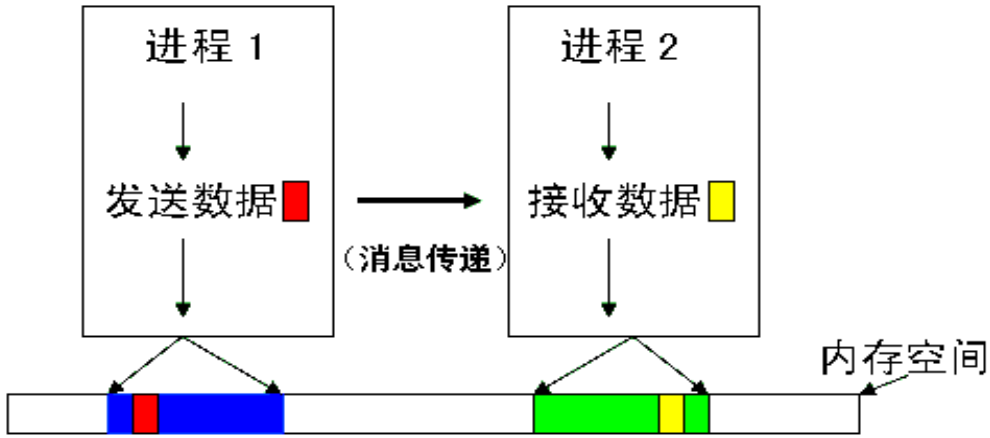


进程与消息传递

- 单机内多个进程
  - 多个进程可同时存在于单机内同一操作系统。操作系统负责调度分时共享处理机资源（CPU、内存、存储、外设等）
  - 进程间相互独立（内存空间不相交）。在操作系统调度下各自独立地运行，例如多个串行应用程序在同一台计算机运行
  - 进程间可以相互交换信息。例如数据交换、同步等待，消息是这些交换信息的基本单位，消息传递是指这些信息在进程间的相互交换，是实现进程间通信的唯一方式

进程与消息传递

- 包含于通过网络联接的不同处理器的多个进程
  - 进程独立存在，并位于不同的处理器，由各自独立的操作系统调度，享有独立的CPU和内存资源
  - 进程间相互信息交换，可依靠消息传递
  - 最基本的消息传递操作包括发送消息send、接受消息receive、进程同步barrier、归约reduction等



MPI重要概念

- 进程组（**process group**） 指MPI 程序的全部进程集合的一个有序子集且进程组中每个进程被赋于一个在该组中唯一的序号(rank)， 用于在该组中标识该进程。序号的取值范围是[0,进程数- 1]
- 通信器（**communicator**）
  - 理解为一类进程的集合即一个进程组，且在该进程组，进程间可以相互通信
  - 任何**MPI**通信函数均必须在某个通信器内发生
  - MPI系统提供省缺的通信器MPI_COMM_WORLD，所有启动的MPI进程通过调用函数MPI_Init()包含在该通信器内；各进程通过函数MPI_Comm_size()获取通信器包含的(初始启动)的MPI进程个数
  - 组内通信器和组间通信器

MPI重要概念

- 进程序号（**rank**） 用来在一个进程组或通信器中标识一个进程
  - **MPI** 程序中的进程由进程组或通信器序号唯一确定，序号相对于进程组或通信器而言（假设np个处理器，标号0…np-1）
  - 同一个进程在不同的进程组或通信器中可以有不同的序号，进程的序号是在进程组或通信器被创建时赋予的
  - MPI 系统提供了一个特殊的进程序号**MPI_PROC_NULL**，它代表空进程(不存在的进程)，与MPI_PROC_NULL 间的通信实际上没有任何作用
- 消息（**message**）
  - 分为数据（**data**）和包装（**envelope**）两个部分
  - 包装由接收进程序号/发送进程序号、消息标号和通信器三部分组成；数据包含用户将要传递的内容

MPI重要概念

- **MPI对象** MPI系统内部定义的数据结构，包括数据类型（如MPI_INT）、通信器（MPI_Comm）、通信请求（MPI_Request）等，它们对用户不透明。在FORTRAN语言中，所有MPI对象均必须说明为“整型变量INTEGER”。

- **MPI联接器（handles）** 联接MPI对象的具体变量，用户可以通过它访问和参与相应MPI对象的具体操作。例如，MPI系统内部提供的通信器MPI_COMM_WORLD。在FORTRAN语言中，所有MPI联接器均必须说明为“整型变量INTEGER”

MPI函数一般形式

C:

```
error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

- 整型错误码由函数值返回
- 除MPI_Wtime() 和MPI_Wtick()外, 所有MPI 的C 函数均返回一个整型错误码。成功时返回MPI_SUCCESS，其他错误代码依赖于执行

Fortran 77 ：

```
CALL MPI_XXXXX(parameter,...,IERROR)
```

- 整型错误码由函数的参数返回
- 除MPI_WTIME() 和MPI_WTICK()外为子函数程序（function），Fortran77的所有MPI过程都是Fortran77的子例行程序（subroutine）

MPI原始数据类型

MPI Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed log 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	
MPI_PACKED	

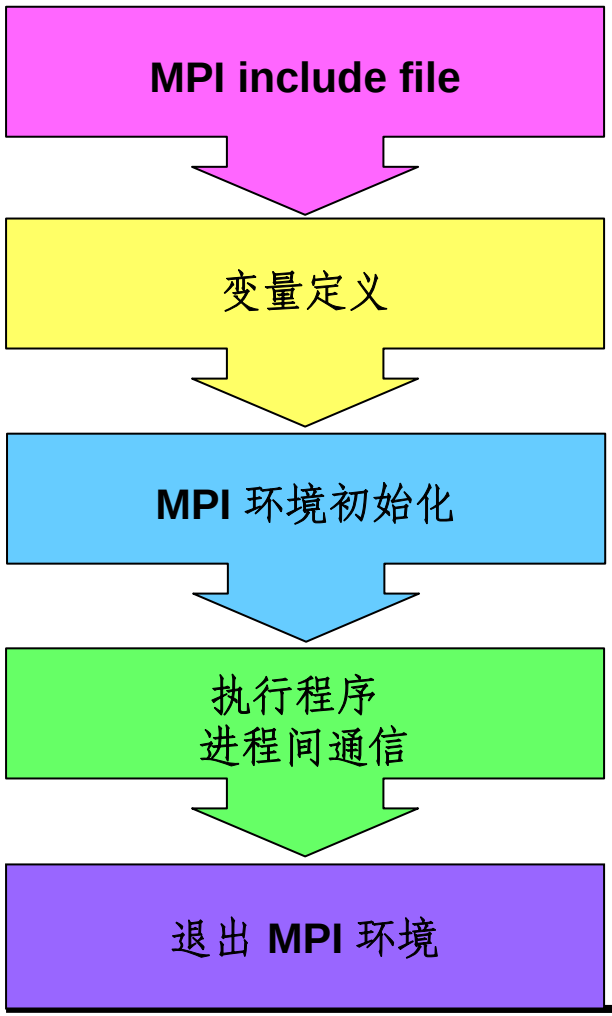
MPI_BYTE 一个字节

MPI_PACKED 打包数据

MPI原始数据类型

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

MPI程序基本结构




```
#include <mpi.h>
void main (int argc, char *argv[])
{
int np, rank, ierr;
ierr = MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&np);
/*      Do Some Works          */
ierr = MPI_Finalize();
}
```

MPI几个基本函数

- **Index**
  - MPI_Init
  - MPI_Initialized
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Finalize
  - MPI_Abort
  - MPI_Get_processor_name
  - MPI_Get_version
  - MPI_Wtime

MPI几个基本函数

- 初始化 **MPI** 系统

C:

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

Fortran 77:

```
MPI_INIT(IERROR)
INTEGER IERROR
```

- 通常为第一个调用的MPI函数，除 MPI_Initialized 外
- 在C接口中，MPI系统通过argc和argv得到命令行参数，并且会把MPI系统专用的参数删除，留下用户的解释参数

MPI几个基本函数

- 检测 **MPI** 系统是否已经初始化

C:

```
int MPI_Initialized(int *flag)
```

Fortran 77:

```
MPI_INIT(FLAG,IERROR)
LOGICAL FLAG
INTEGER IERROR
```

- 唯一可在 MPI_Init 前使用的函数
- 已经调用MPI_Init，返回flag = true，否则flag = false

MPI几个基本函数

- 得到通信器的进程数和进程在通信器中的标号

C:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran 77:

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR

MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER COMM, RANK, IERROR
```

MPI几个基本函数

- 退出 **MPI** 系统

C:

```
int MPI_Finalize(void)
```

Fortran 77:

```
MPI_FINALIZE(IERROR)
```

- 每个进程都必须调用，使用后不准许调用任何MPI函数
- 若不执行MPI退出函数，进程可能被悬挂
- 用户在调用该函数前，应确保非阻塞通讯结束

MPI几个基本函数

- 异常终止**MPI**程序

C:

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

Fortran 77:

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
INTEGER COMM, ERRORCODE, IERROR
```

- 在出现了致命错误而希望异常终止MPI程序时执行
- MPI系统会设法终止comm通信器中所有进程
- 输入整型参数errorcode，将被作为进程的退出码返回给系统

MPI几个基本函数

- 获取处理器的名称

C:

```
int MPI_Get_processor_name(char *name,
    int *resultlen)
```

Fortran 77:

```
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERR)
CHARACTER *(*) NAME
INTEGER RESULTLEN, IERROR
```

- 在返回的name中存储所在处理器的名称
- resultlen存放返回名字所占字节
- 应提供参数name不少于MPI_MAX_PRCESSOR_NAME个字节的存储空间

MPI几个基本函数

- 获取 **MPI** 版本号

C:

```
int MPI_Get_version(int *version,
    int *subversion)
```

Fortran 77:

```
MPI_GET_VERSION(VERSION,SUBVERSION,IERR)
INTEGER VERSION, SUBVERSION, IERROR
```

- 若 mpi 版本号为2.0，则返回的version=2，subversion=0

MPI几个基本函数

- 获取墙上时间

C:

```
double MPI_Wtime(void)
```

Fortran 77:

```
DOUBLE PRECISION MPI_WTIME()
```

- 返回调用时刻的墙上时间，用浮点数表示秒数
- 经常用来计算程序运行时间

Sample :Hello World - C

C+MPI

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
void main(int argc, char *argv[ ])
 {
      int myid, numprocs, namelen;
      char processor_name[MPI_MAX_PROCESSOR_NAME];
      MPI_Init(&argc,&argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&myid);
      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
      MPI_Get_processor_name(processor_name,&namelen);
      printf("Hello World! Process %d of %d on %s\n",myid, numprocs, processor_name);
      MPI_Finalize();
}
```

Sample :Hello World - Fortran

Fortran＋MPI

```
program main
include 'mpif.h'
character * (MPI_MAX_PROCESSOR_NAME) processor_name
integer myid, numprocs, namelen, rc, ierr

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
call MPI_GET_PROCESSOR_NAME(processor_name, namelen, ierr)
 write(*,*) 'Hello World! Process ',myid,' of ',numprocs,' on ', processor_name
call MPI_FINALIZE(ierr)
end
```

Sample :Hello World

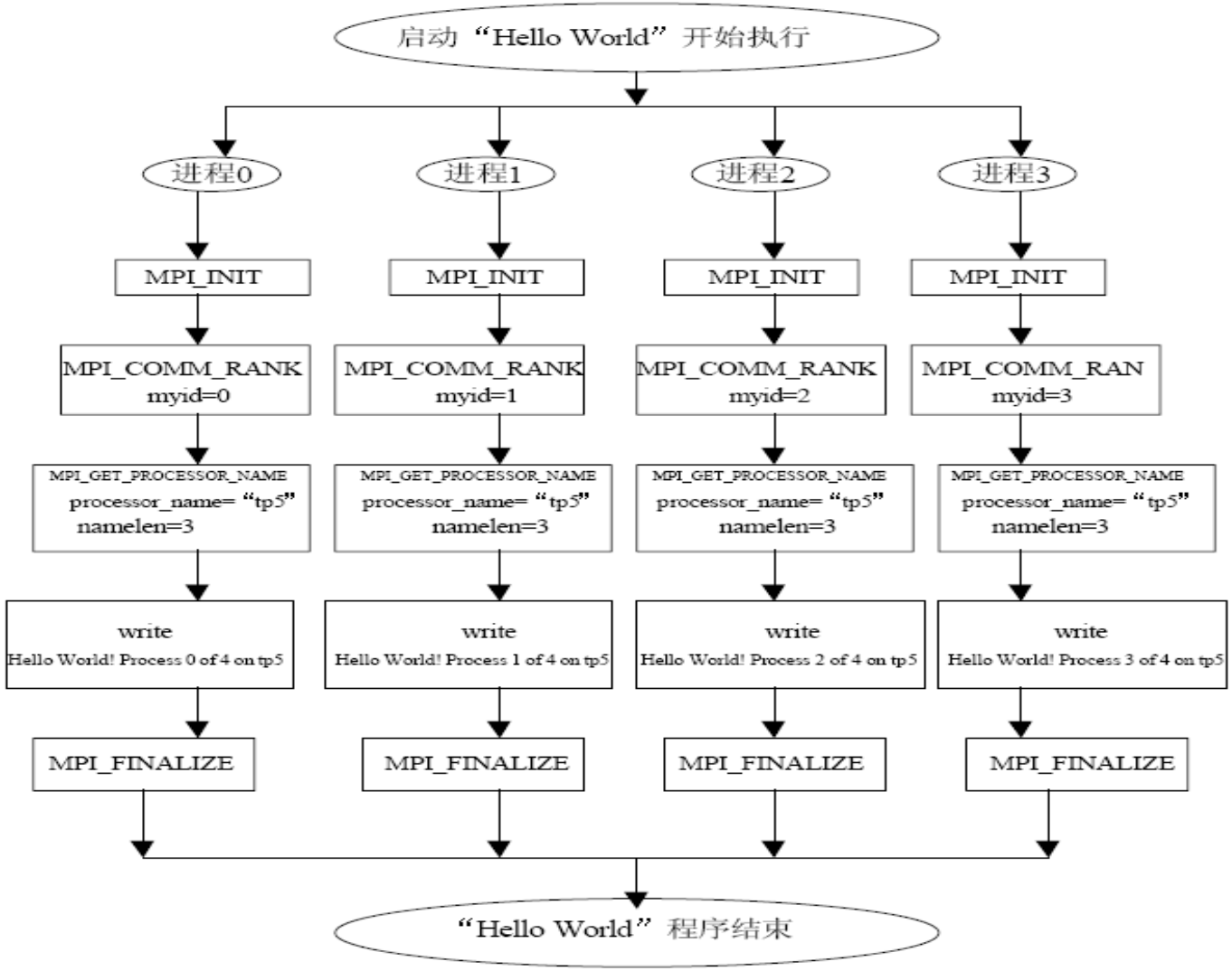
单处理器(tp5)运行4个进程

```
Hello World! Process  1 of 4 on tp5
Hello World! Process  0 of 4 on tp5
Hello World! Process  2 of 4 on tp5
Hello World! Process  3 of 4 on tp5
```

4个处理器(tp1,tp2,tp3,tp4)分别运行4个进程

```
Hello World! Process  0 of 4 on tp5
Hello World! Process  1 of 4 on tp1
Hello World! Process  2 of 4 on tp3
Hello World! Process  3 of 4 on tp4
```

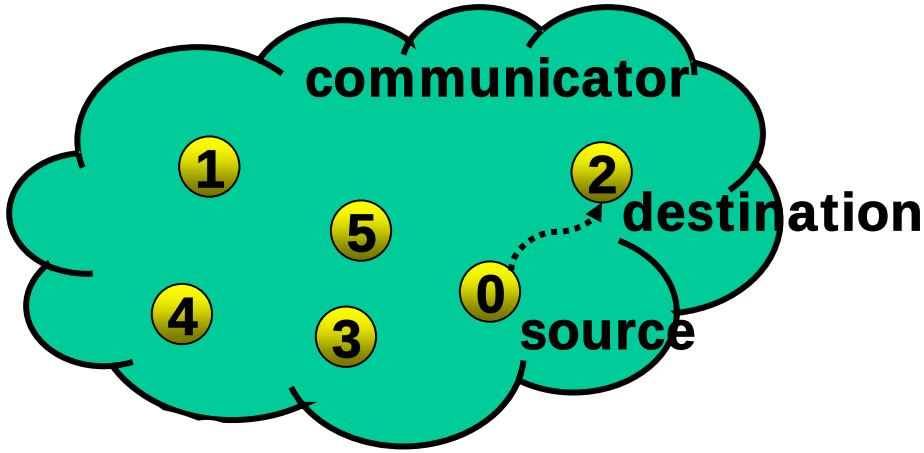
Sample :Hello World



点对点通信

- 定义
- 阻塞式点对点通信
- 编写安全的MPI程序
- 其他阻塞式点对点通信函数
- 阻塞式消息发送模式
- 非阻塞式点对点通信
- 持久通信

定义



- 两个进程之间的通信
- 源进程发送消息到目标进程
- 目标进程接受消息
- 通信发生在同一个通信器内
- 进程通过其在通信器内的标号表示

MPI系统的通信方式都建立在点对点通信之上

阻塞式点对点通信

- **Index**
  - MPI_Send
  - MPI_Recv
  - MPI_Get_count
  - MPI_Sendrecv
  - MPI_Sendrecv_replace

阻塞式点对点通信

- 阻塞式消息发送

C:

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

Fortran 77:

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type>   BUF(*)
INTEGER  COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

- count 不是字节数，而是指定数据类型的个数
- datatype可是原始数据类型，或为用户自定义类型
- dest 取值范围是 0～np－1，或MPI_PROC_NULL
  （np是comm中的进程总数）
- tag 取值范围是 0～MPI_TAG_UB，用来区分消息

阻塞式点对点通信

- 阻塞式消息接收

C:

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

Fortran 77:

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
IERROR)
<type>   BUF(*)
INTEGER  COUNT, DATATYPE, DEST, TAG, COMM, IERROR
INTEGER  STATUS(MPI_STATUS_SIZE)
```

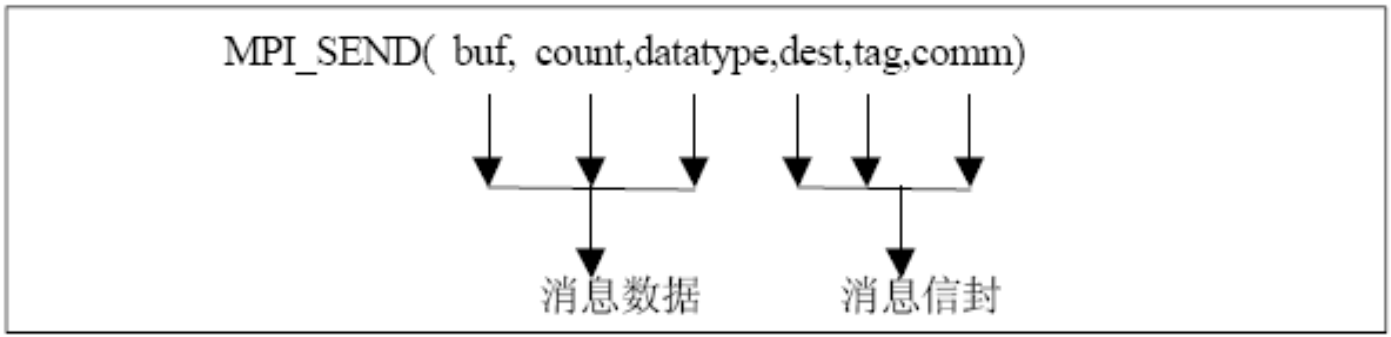
- count是接受缓存区的大小，表示接受上界，具体接受长度可用MPI_Get_count 获得
- source 取值范围是 0～np－1，或MPI_PROC_NULL和MPI_ANY_SOURCE
- tag 取值范围是 0～MPI_TAG_UB，或MPI_ANY_TAG

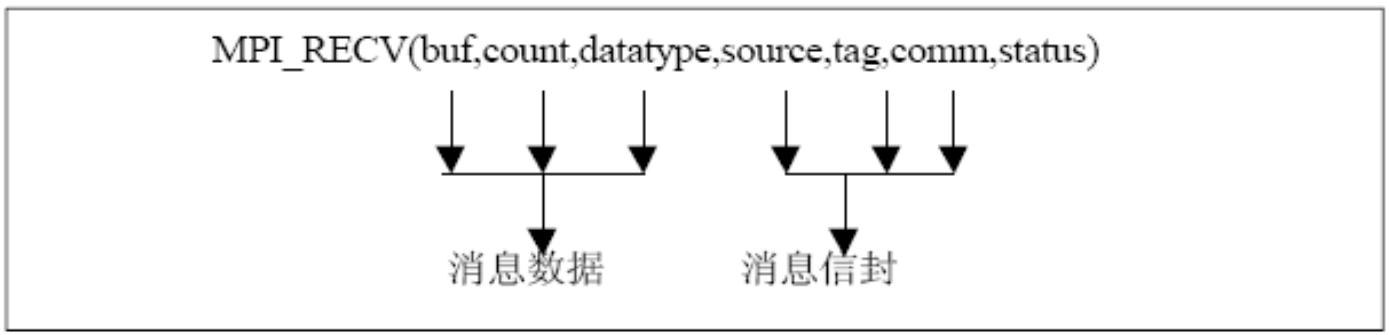
阻塞式点对点通信

- 消息（**message**）

信封：<源/目，标识，通信域>

数据：<起始地址，数据个数，数据类型>





阻塞式点对点通信

- **status的内容**
  - C中是一个数据结构为MPI_status的参数，用户可以直接访问的三个域（共5个域）
  - Fortran中是包含MPI_STATUS_SIZE个整型的数组

```
typedef struct {
... ...int MPI_SOURCE;        消息源地址        STATUS(MPI_SOURCE)
    int MPI_TAG;              消息标号          STATUS(MPI_TAG)
    int MPI_ERROR; ... ...    接收操作的错误码  STATUS(MPI_ERROR)
} MPI_Status;
```

- 使用前需要用户为其申请存储空间 (MPI_Status status;)
- C中引用时为 status.MPI_SOURCE …

阻塞式点对点通信

- 查询接受到的消息长度

C:

```
int MPI_Get_count(MPI_Status status, MPI_Datatype datatype,
                  int *count)
```

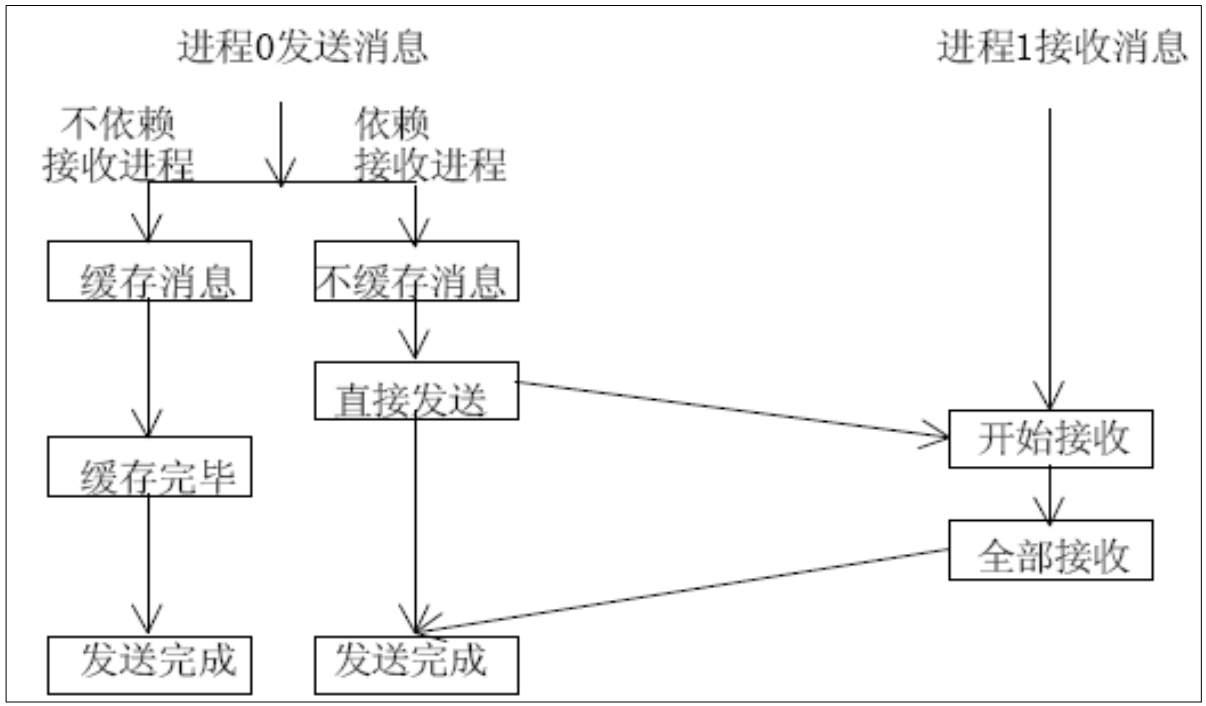
Fortran 77:

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERR)
INTEGER DATATYPE, COUNT, IERR, STATUS(MPI_STATUS_SIZE)
```

- 该函数在count中返回数据类型的个数，即消息的长度
- count属于MPI_Status结构的一个域，但不能被用户直接访问

定义

- 标准阻塞式通信



- 是否对发送数据进行缓存，由MPI系统决定，而非程序员
- 阻塞：发送成功，意味（1）消息成功发送；（2）或者消息被缓存
  接收成功，意味消息已被成功接收

阻塞式点对点通信

- 消息传递成功
  - 发送进程需指定一个有效的目标接收进程
  - 接收进程需指定一个有效的源发送进程
  - 接收和发送消息的进程要在同一个通信器内
  - 接收和发送消息的 **tag** 要相同
  - 接收缓存区要足够大

阻塞式点对点通信

- 任意源进程（接收操作可以接受任意进程的消息）

  `MPI_ANY_SOURCE`

- 任意标号（接收操作可以接受任意标号的消息）

  `MPI_ANY_TAG`

- 真实的源进程与消息标号可以访问接受函数中的**status**参数获得

其他阻塞式点对点通信函数

- 捆绑发送和接收
  - 将一次发送调用和一次接收调用合并在一起，执行无先后
  - 发送缓冲区和接收缓冲区须分开
  - 发送与接收使用同一个通信域
  - 由捆绑发送接收调用发出的消息可被普通接收操作接收；一个捆绑发送接收调用可以接受一个普通的发送操作所发送的消息

C:

```
int MPI_Sendrecv(void *sendbuff,int sendcount,MPI_Datatype sendtype,int
                 dest,int sendtag,void *recvbuff,int recvcount,
                 MPI_Datatype recvtype, int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

其他阻塞式点对点通信函数

- 捆绑发送和接收

Fortran 77:

```
MPI_SENDRECV(SENDBUFF,SENDCOUNT,SENDTYPE,DEST,SENDTAG,RECVBUFF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG,COMM, STATUS, IERR)
<type> SENDBUFF(*), RECVBUFF(*)
INTEGER SENDCOUNT,SENDTYPE,DEST,SENDTAG,RECVCOUNT,RECVTYPE,SOURCE,
        RECVTAG,COMM,IERR
INTEGER STATUS(MPI_STATUS_SIZE)
```

语义上等同于一个发送和一个接收操作结合，但此函数可以有效避免在单独发送和接收操作过程中，由于调用次序不当而造成的死锁。MPI系统会优化通信次序，从而最大限度避免错误发生。

其他阻塞式点对点通信函数

- 捆绑发送和接收，收发使用同一缓存区

C:

```
int MPI_Sendrecv_replace(void *buff,int count,MPI_Datatype datatype,
                         int dest, int sendtag,int source, int
                         recvtag,MPI_Comm comm, MPI_Status *status)
```

Fortran 77:

```
MPI_SENDRECV_REPLACE(BUFF,COUNT,DATATYPE,DEST,SENDTAG,SOURCE,RECVTAG,
                     COMM, STATUS, IERR)
<type> BUFF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, IERR
INTEGER STATUS(MPI_STATUS_SIZE)
```

- MPI系统保证其消息发出后再接收信息
- MPI_Sendrecv送收使用不同的缓存区；该函数使用同一缓存区

Sample - Fortran

- **MPI_SENDRECV代替MPI_SEND和MPI_RECV**

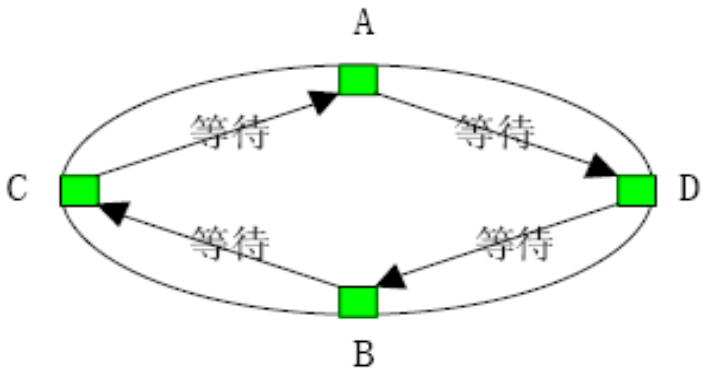
```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE （rank .EQ. 1）
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
   CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 1, tag,
 +                                recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
IF(rank.EQ.1) THEN
   CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 0, tag,
 +                                recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
```

编写安全的MPI程序

- **Pro0发送消息到Pro1，同时，Pro1发送消息到Pro0**

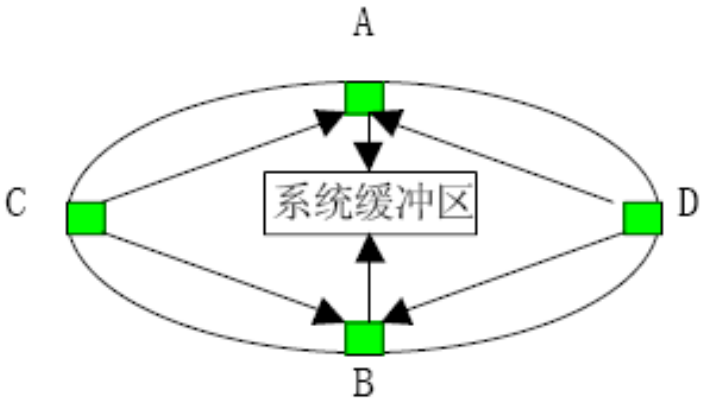
```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)    A
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)            C
ELSE IF( rank .EQ. 1)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)    B
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)            D
END IF
```



死锁

编写安全的MPI程序

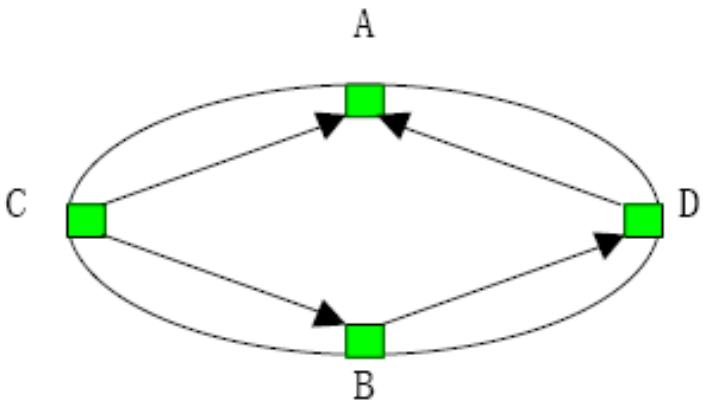
```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)        A
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)   C
ELSE （rank .EQ.1）
   CALL MPI_SEND(sendbuf, count, MPI_REAK, 0, tag, comm, status, ierr)   B
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)   D
END IF
```



不安全

编写安全的MPI程序

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)      A
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)   C
ELSE （rank .EQ. 1）
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)   D
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)      B
END IF
```



正确

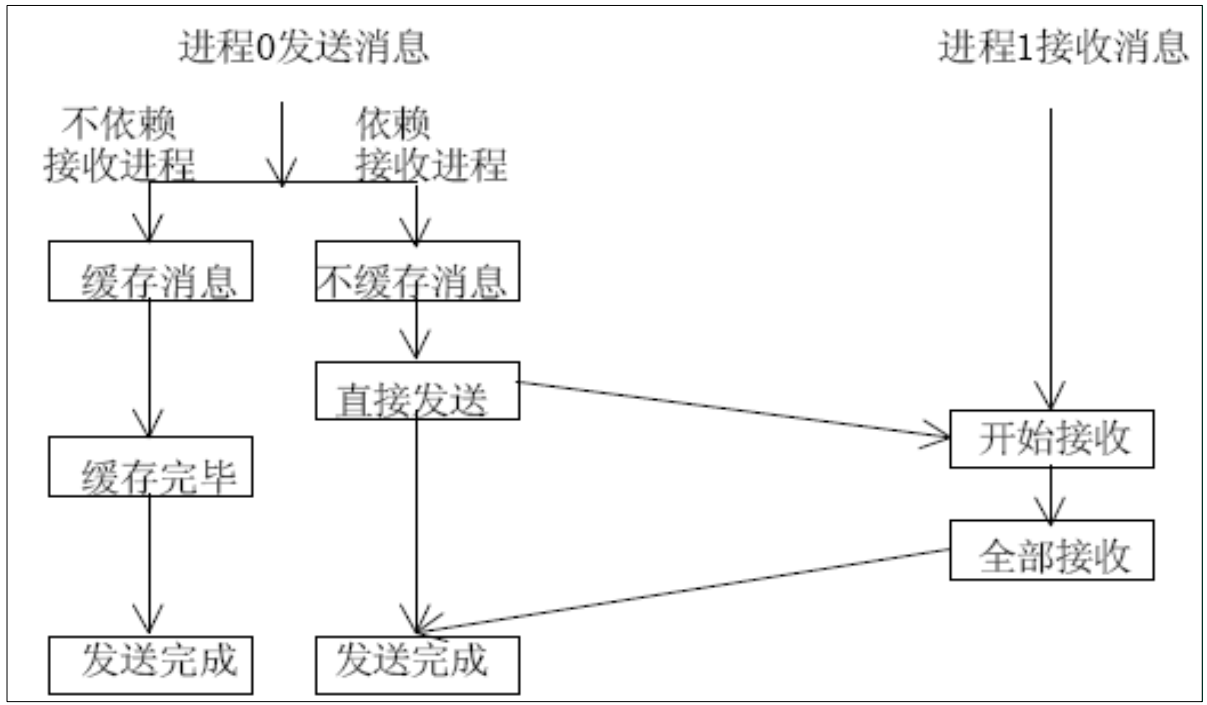
阻塞式消息发送模式

- 按着发送方式和接收方状态要求的不同分类

通信模式	发送	接收
标准通信模式	MPI_SEND	MPI_RECV
缓存通信模式	MPI_BSEND	
同步通信模式	MPI_SSEND	
就绪通信模式	MPI_RSEND	

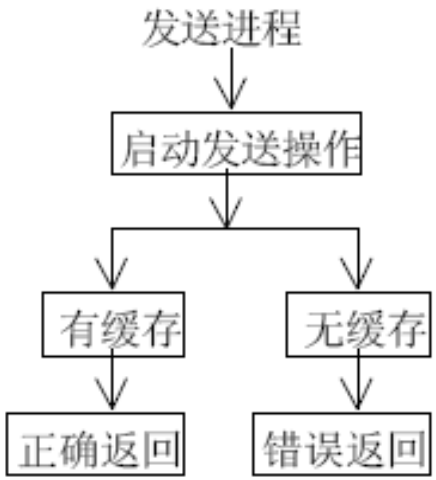
- 四个函数拥有完全一样的入口参数
- 共用一个标准的消息接收函数

标准消息发送函数 (MPI_Send)



- 发送操作不管接收操作是否启动，都可以开始
- 发送返回的条件
  - 发送数据被MPI系统存入系统缓存，此时不要求接收操作收到发送数据
  - 不缓存，则数据被接收到接收缓冲区

缓存消息发送函数 (MPI_Bsend)



- 发送操作不管接收操作是否启动，都可以开始
- 直接对缓冲区进行控制，用户直接对通信缓冲区进行申请、使用、释放
  - 发送消息前必须有足够的缓冲区可用，否则发送失败
  - 缓存发送返回后，不意味申请的缓存区可自由使用，须等待消息发送出去方可使用
- 优势：发送操作在缓存了发送数据后，可以立刻返回

缓存消息发送函数 (MPI_Bsend)

- 缓冲区申请提交

C

```
int MPI_Buffer_attach( void* buffer, int size)
```

Fortran 77

```
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERR)
<type>  BUFFER(*)
INTEGER SIZE, IERR
```

  - 可调用MPI_Type_size来确定数据类型所占字节数
  - 缓冲区大小=数据长度+常数，常数用于MPI系统管理提交的缓冲区，是必须的
  - 同一时刻，一个进程只能定义一个缓冲区

- 缓冲区释放

C

```
int MPI_Buffer_detach( void *buffer_addr, int *size)
```

Fortran 77

```
MPI_BUFFER_DETACH( BUFFER_ADDR, SIZE, IERR)
<type>  BUFFER_ADDR(*)
INTEGER SIZE, IERR
```

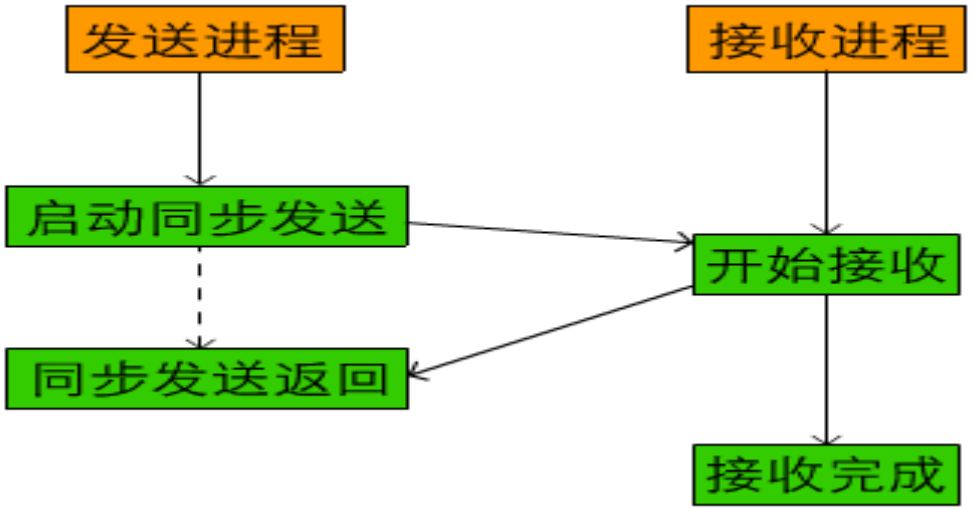
  - 此函数为阻塞式调用，等到该缓存消息发送后才释放返回

Sample - Fortran

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE （rank .EQ. 1）
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

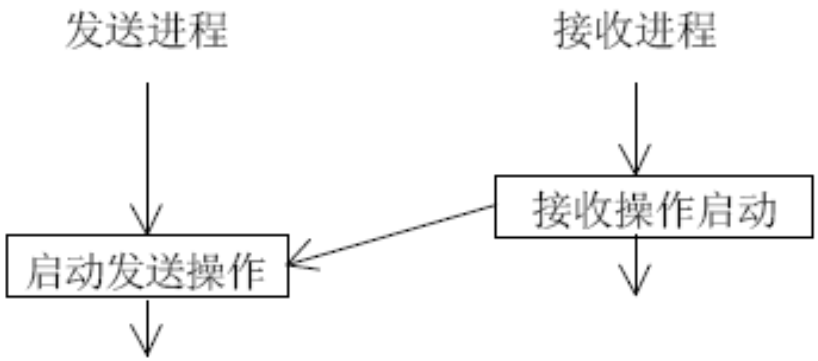
```
REAL * (*) BUF
INTEGER SIZE, TOTALSIZE
CALL MPI_TYPE_SIZE(MPI_REAL,SIZE,ierr)
TOTALSIZE=count*SIZE + 2*MPI_BSEND_OVERHEAD  ！必须如此
CALL MPI_BUFFER_ATTACH(BUF,TOTALSIZE,ierr)
IF(rank.EQ.0) THEN
   CALL MPI_BSEND(sendbuf, count, MPI_REAL, 1, tag,comm,ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
IF(rank.EQ.1) THEN
   CALL MPI_BSEND(sendbuf, count, MPI_REAL, 0, tag,comm,ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
```

同步消息发送函数 (MPI_Ssend)



- 同步通信模式的开始不依赖于接收进程相应的接收操作是否已经启动
- 发送返回条件，需在标准模式上确认接收方已经开始接收数
- 优势：这种模式发送和接收最为安全

就绪消息发送函数 (MPI_Rsend)



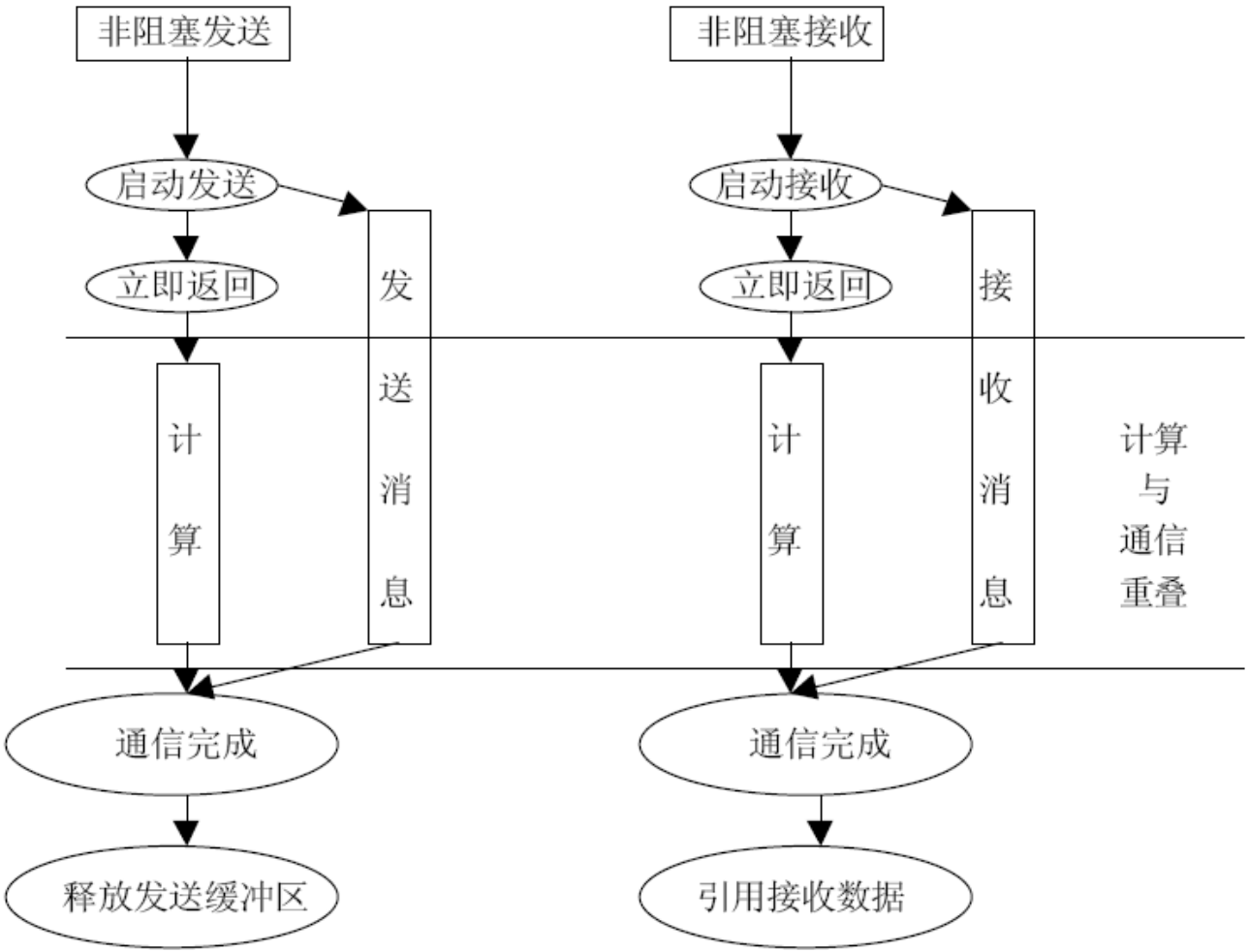
- 发送操作必须要求接收操作启动，才可以开始
  - 启动接受操作，意味着接收进程正等待接收发送的消息
  - 若发送操作启动而相应接收操作没有启动，发送操作将出错
- 优势：减少消息发送时间开销，可能获得好的计算性能

非阻塞式点对点通信

- 阻塞式通信与非阻塞式通信

通信类型	函数返回	对数据区操作	特性
阻塞式通信	1. 阻塞型函数需要等待指定操作完成返回 2. 或所涉及操作的数据要被MPI系统缓存安全备份后返回	函数返回后，对数据区操作是安全的	1. 程序设计相对简单 2. 使用不当容易造成死锁
非阻塞式通信	1. 调用后立刻返回，实际操作在MPI后台执行 2. 需调用函数等待或查询操作的完成情况	函数返回后，即操作数据区不安全。可能与后台正进行的操作冲突	1. 可以实现计算与通信的重叠 2. 程序设计相对复杂

非阻塞式点对点通信



非阻塞式点对点通信

- **Index**
  - MPI_Isend/MPI_Irecv
  - MPI_Wait/MPI_Waitany/MPI_Waitall/MPI_Waitsome
  - MPI_Test/MPI_Testany/MPI_Testall/MPI_Testsome
  - MPI_Request_free
  - MPI_Cancel
  - MPI_Test_cancelled
  - MPI_Probe/MPI_Iprobe

非阻塞式点对点通信

- 非阻塞式发送

C

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

Fortran 77

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
+         REQUEST, IERR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERR
```

  - 该函数仅提交了一个消息发送请求，并立即返回
  - MPI系统会在后台完成消息发送
  - 函数为该发送操作创建了一个请求，通过request变量返回
  - request可供之后（查询和等待）函数使用

非阻塞式点对点通信

- 非阻塞式接收

C

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

Fortran 77

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
+         REQUEST, IERR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,
+         IERR
```

  - 该函数仅提交了一个消息接收请求，并立即返回
  - MPI系统会在后台完成消息发送
  - 函数为该接收操作创建了一个请求，通过request变量返回
  - request可供之后查询和等待函数使用

非阻塞式点对点通信

- 等待、检测一个通信请求的完成

C

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

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

Fortran 77

```
MPI_WAIT(REQUEST, STATUS, IERR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERR

MPI_TEST(REQUEST, FLAG, STATUS, IERR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERR
LOGICAL FLAG
```

- MPI_Wait 阻塞等待通信函数完成后返回；MPI_Test检测某通信，不论其是否完成，都立刻返回。如果通信完成，则flag=true
- 当等待或检测的通信完成时，通信请求request被设置成MPI_REQUEST_NULL
- 考察接收请求，status返回与MPI_Recv一样；发送请求，则不确定
- MPI_Test返回时，当flag=false, status不被赋值

Sample - Fortran

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE （rank .EQ. 1）
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

```
IF(rank.EQ.0) THEN
   CALL MPI_ISEND(sendbuf, count, MPI_REAL, 1, tag,comm,request,ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
   CALL MPI_WAIT(request,status,ierr)
IF(rank.EQ.1) THEN
   CALL MPI_ISEND(sendbuf, count, MPI_REAL, 0, tag,comm,request,ierr)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_WAIT(request,status,ierr)
```

非阻塞式点对点通信

- 等待、检测一组通信请求中某一个的完成

C

```
int MPI_Waitany(int count,
        MPI_Request *array_of_requests,
        int *index, MPI_Status *status)

int MPI_Testany(int count,
        MPI_Request *array_of_requests,
        int *index, int *flag, MPI_Status *status)
```

- count表示通信请求的个数
- array_of_requests是一组非阻塞通信的请求
- index存储一个成功完成的通信在array_of_requests中的位置
- flag表示是否有任意一个通信请求完成，若有flag=true
- 完成的通信请求request被自动赋值MPI_REQUEST_NULL
- MPI_Testany返回时，当flag=false, status不被赋值

非阻塞式点对点通信

Fortran 77

```
      MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS,
     +            IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, IERR
      INTEGER STATUS(MPI_STATUS_SIZE)

      MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG,
     +            STATUS, IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, IERR
      INTEGER STATUS(MPI_STATUS_SIZE)
      LOGICAL FLAG
```

非阻塞式点对点通信

- 等待、检测一组通信请求的全部完成

C

```
int MPI_Waitall(int count,
        MPI_Request *array_of_requests,
        MPI_Status *array_of_statuses)

int MPI_Testall(int count,
        MPI_Request *array_of_requests,
        int *flag, MPI_Status *array_of_statuses)
```

- count表示通信请求的个数
- array_of_requests是一组非阻塞通信的请求
- array_of_statuses返回该组通信完成的状态
- flag表示全部通信是否完成，若完成flag=true
- MPI_Testall返回时，当flag=false, array_of_statuses不被赋值

非阻塞式点对点通信

Fortran 77

```
      MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES,
     +            IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)

      MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG,
     +            ARRAY_OF_STATUSES, IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)
      LOGICAL FLAG
```

非阻塞式点对点通信

- 等待、检测一组通信请求的部分完成

C

```
int MPI_Waitsome(int incount,
        MPI_Request *array_of_requests,
        int outcount, int *array_of_indices,
        MPI_Status *array_of_statuses)

int MPI_Testsome(int incount,
        MPI_Request *array_of_requests,
        int outcount, int *array_of_indices,
        MPI_Status *array_of_statuses)
```

- MPI_Waitsome等待至少一个通信完成才返回
- outcount表示通信成功完成的个数
- array_of_indices存储完成的通信在array_of_requests中的位置
- array_of_statuses返回完成通信的状态，其他不被赋值
- MPI_Testsome返回时若没有一个通信完成，则outcount=0

非阻塞式点对点通信

Fortran 77

```
      MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES,
     +            IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)

      MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG,
     +            ARRAY_OF_STATUSES, IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)
      LOGICAL FLAG
```

非阻塞式点对点通信

- 通信请求的释放（阻塞型）

C

```
int MPI_Request_free(MPI_Request *request)
```

Fortran 77

```
MPI_REQUEST_FREE(REQUEST, IERR)
INTEGER REQUEST, IERR
```

- 调用MPI_Wait/Test可间接释放完成的通信请求，此函数则直接释放通信请求及所占内存空间
- 如果通信尚未完成，则阻塞等待完成后返回
- 该函数返回，通信请求request被设置成MPI_REQUEST_NULL

非阻塞式点对点通信

- 通信请求的取消（非阻塞型）

C

```
int MPI_Cancel(MPI_Request *request)
```

Fortran 77

```
MPI_CANCEL(REQUEST, IERR)
INTEGER REQUEST, IERR
```

- MPI_Cancel取消已调用的非阻塞通信，用此命令来释放非阻塞操作所占用的资源
- 命令调用后立刻返回，但调用并不意味相应的通信被取消。该操作调用时，若相应非阻塞通信已经开始，它会正常完成，不受影响；若没有开始，则释放通信占用资源，该通信被取消
- 即使调用取消操作，也需等待、查询函数来释放该非阻塞通信的请求,并且在返回结果status中指明该通信已经被取消

非阻塞式点对点通信

- 检测一个通信操作是否被取消（非阻塞型）

C

```
int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

Fortran 77

```
MPI_TEST_CANCELLED(STATUS, FLAG, IERR)
INTEGER STATUS(MPI_STATUS_SIZE), IERR
LOGICAL FLAG
```

  - 函数调用后立即返回
  - 如果一个非阻塞通信已经被执行了取消操作，则该通信的MPI_Wait和MPI_Test将释放相应的通信对象，并且在返回结果status中指明该通信是否被取消
  - 如果MPI_Test_cancelled返回结果flag=true，则表明此通信已经被成功取消，否则该通信还没有被取消

非阻塞式点对点通信

- 消息探测（阻塞型）

C

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
        MPI_Status *status)
```

Fortran 77

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERR)
INTEGER SOURCE, TAG, COMM, IERR
INTEGER STATUS(MPI_STATUS_SIZE)
```

- 为接收消息前即可对接收消息进行探测，进而决定如何接收该消息
- 阻塞等待，只有当探测到符合source/tag条件的消息时才返回
- 返回的status与MPI_Recv的status完全相同
- source/tag可以取MPI_ANY_SOURCE/MPI_ANY_TAG

非阻塞式点对点通信

- 消息探测（非阻塞型）

C

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
        int *flag, MPI_Status *status)
```

Fortran 77

```
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERR)
INTEGER SOURCE, TAG, COMM, IERR
INTEGER STATUS(MPI_STATUS_SIZE)
LOGICAL FLAG
```

  - 函数调用后即返回
  - 当探测到符合source/tag条件的消息时，flag=true;
  - 若flag=true，返回的status与MPI_Recv的status完全相同；若flag=false，则对status不作定义
  - source/tag可以取MPI_ANY_SOURCE/MPI_ANY_TAG

阻塞型与非阻塞型通信函数

函数类型	通信模式	阻塞型	非阻塞型
消息发送函数	标准模式	MPI_Send	MPI_Isend
	缓冲模式	MPI_Bsend	MPI_Ibsend
	同步模式	MPI_Ssend	MPI_Issend
	就绪模式	MPI_Rsend	MPI_Irsend
消息接收函数		MPI_Recv	MPI_Irecv
消息检测函数		MPI_Probe	MPI_Iprobe
等待/查询函数		MPI_Wait	MPI_Test
		MPI_Waitall	MPI_Testall
		MPI_Waitany	MPI_Testany
		MPI_Waitsome	MPI_Testsome
释放通信请求		MPI_Request_free	
取消通信			MPI_Cancel
			MPI_Test_cancelled

持久通信

- 持久通信

持久通信请求 (persistent communication request) 可用于以完全相同的方式 (相同的通信器、收发缓冲区、数据类型与长度、源/目的地址和消息标签) 重复收发的消息. 目的是减少处理消息的开销及简化 MPI 程序.

1 通信的初始化，比如MPI_SEND_INIT
2 启动通信，MPI_START
3 完成通信，MPI_WAIT
4 释放查询对象，MPI_REQUEST_FREE

- 通信初始化没有启动消息操作，MPI_Start触发才开始真正通信
- 调用消息完成操作MPI_Wait，并不释放持久通信对象，只是将其设置为非活动状态
- 当不需要再进行重复通信时，必须用MPI_Request_free释放对象，这是和普通非阻塞通信不同之处

持久通信

- 创建持久消息发送请求

C

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

Fortran 77

```
      MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
     +        REQUEST, IERR)
      <type>  BUF(*)
      INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM,
     +        REQUEST, IERR
```

  - 创建持久消息发送请求，但并非开始实际的消息发送，request请求留待以后实际消息发送使用
  - 其对应标准模式的非阻塞发送，MPI_Bsend_init/MPI_Ssend_init/MPI_Rsend_init分别对应缓冲/同步/就绪模式的持久消息发送请求函数，不作重点说明

持久通信

- 创建持久消息接收请求

C

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

Fortran 77

```
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
        REQUEST, IERR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,
+       IERR
```

  - 创建持久消息接收请求，但并非开始实际的消息接收，request请求留待以后实际消息接收使用

持久通信

- 开始基于持久通信请求的通信

C

```
int MPI_Start(MPI_Request *request)

int MPI_Startall(int count,
        MPI_Request *array_of_requests)
```

Fortran 77

```
MPI_START(REQUEST, IERR)
INTEGER REQUEST, IERR

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
```

  - 每次调用MPI_Start相当于调用一次MPI_***_init相对应的非阻塞型通信函数（如MPI_Isend，MPI_Irecv等）
  - 当创建了多个持久通信请求时，可用一个MPI_Startall来一次性启动多个通信
  - 由上述函数调用开始的通信请求也要用MPI_Wait/Test函数来等待、检测完成。这些调用完成后，持久通信对象并不释放，等待激活。