# MPI 编程

# 1. 预备知识

MPI — Message Passing Interface 是基于消息传递编写并行程序的一种用户界面。

## 1.1 什么是 MPI

- MPI 是一个库，不是一门语言。MPI 提供库函数/过程供 C/FORTRAN 调用。
- MPI 是一种标准或规范的代表而不特指某一个对它的具体实现。
  迄今为止所有的并行计算机制造商都提供对 MPI 的支持。
- MPI 是一种消息传递编程模型。最终目的是服务于进程间通信这一目标。

**MPI 的目的**

- 较高的通信性能；
- 较好的程序可移植性；
- 强大的功能。

**目前主要的 MPI 实现**

- MPICH — 开源免费的MPI实现，支持Windows和Unix
  http://www.mpich.org
- OpenMPI — 开源免费
  http://www.open-mpi.org

一些厂商也提供商业版的 MPI 系统，主要是在 MPICH/OpenMPI 的基础上优化产生的。

## 1.2 一些名词与概念

### 1.2.1 程序代码

这里的程序不是指以文件形式存在的源代码、可执行代码等，而是指为了完成一个计算任务而进行的一次运行过程。

### 1.2.2 进程(Process)

一个 MPI 并行程序由一组运行在相同或不同计算机 /计算节点上的进程或线程构成。为统一起见，我们将 MPI 程序中一个独立参与通信的个体称为一个进程。

### 1.2.3 进程组

一个 MPI 程序的全部进程集合的一个有序子集。进程组中每个进程都被赋予一个在改组中唯一的序号（rank），用于在该组中标识该进程。序号范围从 0 到进程数－1。

### 1.2.4 通信器（communicator）

有时也译成通信子，是完成进程间通信的基本环境，它描述了一组可以互相通信的进程以及它们之间的联接关系等信息。MPI 所有通信必须在某个通信器中进行。通信器分域内通信器（intracommunicator）和域间通信器（intercommunicator）两类，前者用于同一进程中进程间的通信，后者则用于分属不同进程的进程间的通信。

MPI 系统在一个 MPI 程序运行时会自动创建两个通信器：一个称为 MPI_COMM_WORLD，它包含 MPI 程序中所有进程，另一个称为 MPI_COMM_SELF，它指单个进程自己所构成的通信器。

### 1.2.5 序号（rank）

即进程的标识，是用来在一个进程组或一个通信器中标识一个进程。MPI 的进程由进程组 /

序号或通信器 /序号唯一确定。

#### 1.2.6 消息（message）

MPI 程序中在进程间传递的数据。它由通信器、源地址、目的地址、消息标签和数据构成。

#### 1.2.7 通信（communication）

通信是指在进程之间进行消息的收发、同步等操作。

# 2. 第一个 MPI 程序

## 2.1 FORTRAN77 的 MPI 程序

```
      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(*,10) myid,numprocs,processor_name
 10   FORMAT('Hello World! Process ',I2,' of ',I1,' on ', 20A)
      call MPI_FINALIZE(ierr)
      end
```

- ✧ mpif.h 是MPI相对于FORTRAN实现的头文件
- ✧ MPI_MAX_PROCESSOR_NAME是MPI预定义的宏，即MPI所允许的机器名字的最大长度
- ✧ MPI程序的开始和结束必须是MPI_INIT和MPI_FINALIZE，分别完成MPI的初始化和结束工作
- ✧ MPI_COMM_RANK得到当前正在运行的进程的标识号
- ✧ MPI_COMM_SIZE得到所有参加运算的进程的个数
- ✧ MPI_GET_PROCESSOR_NAME得到运行本进程的机器的名称
- ✧ 进程标识号为 0，…，processor-1
- ✧ write语句是普通的FORTRAN语句

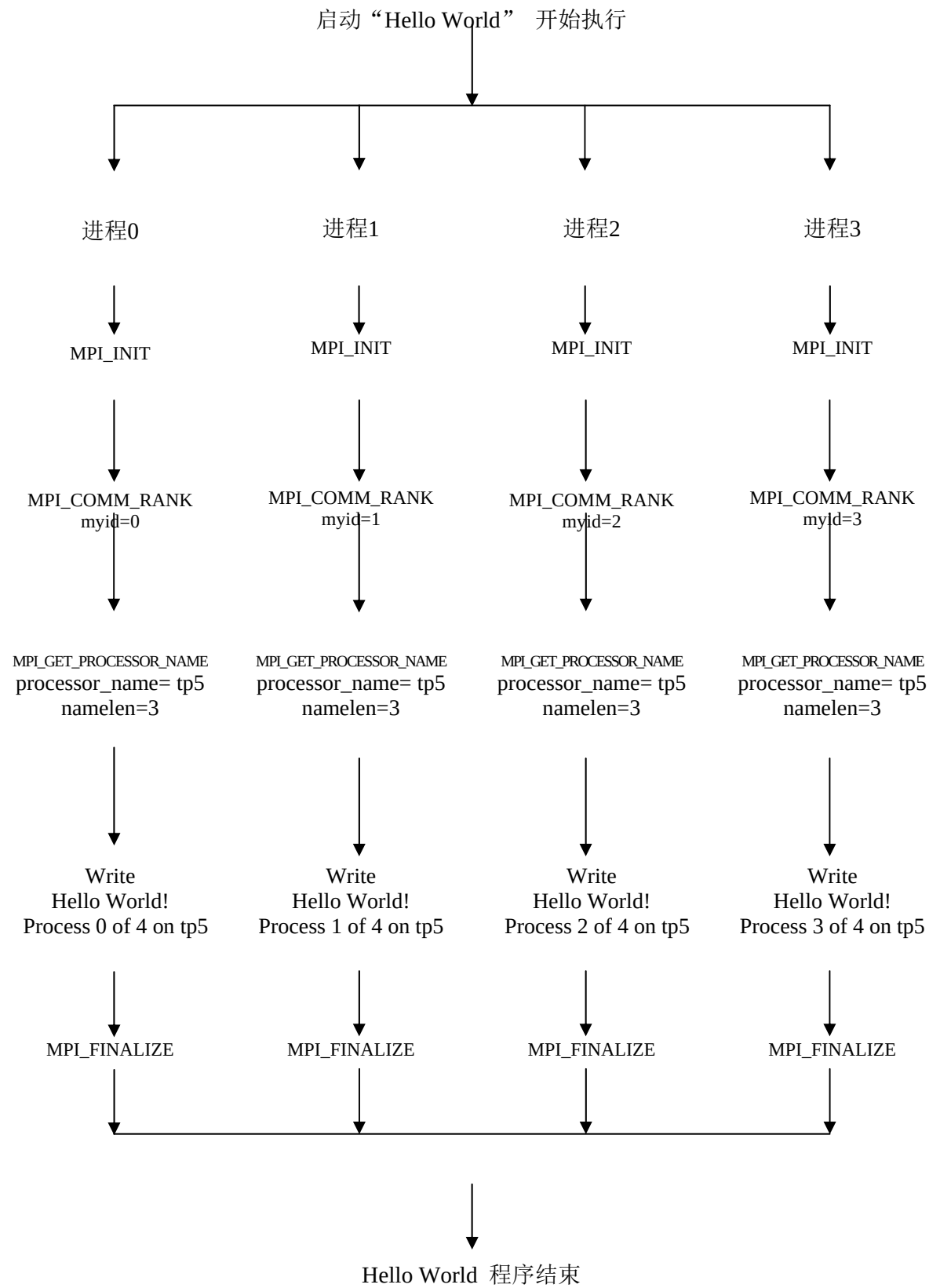
在单个机器tp5上启动4个进程同时运行，执行结果如下：

```
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
```

在四台机器tp5,tp1,tp3,tp4上启动4个进程同时运行，执行结果如下：

```
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
```

## 2.2 C 语言的 MPI 程序



```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
int myid, numprocs;
int 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);
fprintf(stderr, "Hello World! Process %d of %d on %s\n",
myid, numprocs, processor_name);
MPI_Finalize();
}
```

- ✧ mpi.h 是MPI相对于C语言的头文件
- ✧ argc，argv 是命令行参数
- ✧ fprintf 是打印语句

## 2.3 MPI 的一些惯例

- ✧ MPI的所有常量、变量与函数 /过程均以MPI_ 开头
- ✧ MPI 的 C 语言接口为函数，FORTRAN 接口为子程序，且对应接口的名称相同
- ✧ 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
- ✧ 对于FORTRAN 程序，MPI 函数全部以过程方式调用，一般全用大写字母表示，即MPI_XXXX_XXX形式（FORTRAN不区分大小写）
- ✧ 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，而 MPI 的所有 FORTRAN 子程序中都有一个哑元参数（IERR）代表调用错误码
- ✧ MPI 是按进程组(Process Group) 方式工作的，所有MPI 程序在开始时均被认为是在通信器MPI_COMM_WORLD 所拥有的进程组中工作，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(communicator) 中进行
- ✧ FORTRAN的数组下标是以1开始，而C语言的数组是以0开始
- ✧ 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。

# 3. 六个接口构成的 MPI 子集

## 3.1 MPI 原始数据类型

MPI系统中数据的发送和接收都是基于数据类型进行的。数据类型可以是MPI系统预定义的，称为原始数据类型；也可以是用户在原始数据类型的基础上自己定义的新的数据类型。

MPI 的基本数据类型定义与相应的FORTRAN 和C 的数据类型对照关系如下:

Fortran 程序中可使用的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_ DOUBLE_COMPLEX | DOUBLE COMPLEX |
| MPI_LOGICAL | LOGICAL |
| MPI_CHARACTER | CHARACTER(1) |
| MPI_BYTE | |
| MPI_PACKED | |

C 程序中可使用的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 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 | |
| MPI_PACKED | |

这里MPI_BYTE 和MPI_PACKED 不对应FORTRAN 或C 中的任何数据类型，MPI_BYTE 是由一个字节组成的，而MPI_PACKED 将在后面介绍。

此外可能还有一些附加的MPI数据类型：

| 附加的MPI数据类型 | 相应的C数据类型 |
|---|---|
| MPI_LONG_LONG_INT long | long int |

| 附加的MPI数据类型 | 相应的FORTRAN77数据类型 |
|---|---|
| MPI_DOUBLE_COMPLEX | DOUBLE COMPLEX |

| | |
|---|---|
| MPI_REAL2 | REAL*2 |
| MPI_REAL4 | REAL*4 |
| MPI_REAL8 | REAL*8 |
| MPI_INTEGER1 | INTEGER*1 |
| MPI_INTEGER2 | INTEGER*2 |
| MPI_INTEGER4 | INTEGER*4 |

## 3.2 MPI 常用的六个函数

### 3.2.1 MPI 初始化

MPI_INIT()

**参数**：无

**C ：**

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

**F77 ：**

MPI_INIT(IERR)

INTEGER IERR

这个函数初始化MPI 并行程序的执行环境，它必须在调用所有其它MPI函数(除MPI_INITIALIZED)之前被调用，并且在一个MPI 程序中，只能被调用一次。

### 3.2.2 结束 MPI 系统

MPI_FINALIZE ()

**参数**：无

**C ：**

int MPI_Finalize(void)

**F77 ：**

MPI_FINALIZE(IERR)

INTEGER IERR

这个函数清除MPI 环境的所有状态。即一但它被调用，所有MPI 函数都不能再调用，其中包括MPI_INIT。

### 3.2.3 获取当前进程的标识

MPI_COMM_RANK(comm，rank)

**参数**： IN comm 通信器

OUT rank 本进程在通信器 comm 中的标识号

**C ：**

int MPI_Comm_rank(MPI_Comm comm, int * rank)

**F77 ：**

MPI_COMM_RANK(COMM, RANK, IERR)

INTEGER COMM, RANK, IERR

### 3.2.4 通信器包含的进程数

MPI_COMM_SIZE(comm, size)

**参数**： IN comm 通信器

OUT size 该通信器所包含的进程数

**C ：**

int MPI_Comm_size(MPI_Comm comm, int * size)

**F77 ：**

MPI_COMM_SIZE(COMM, SIZE, IERR)
INTEGER COMM, SIZE, IERR

### 3.2.5 消息发送

MPI_SEND(buf, count, datatype, dest, tag, comm)

**参数：**

| | | |
|---|---|---|
| IN | buf | 所发送消息的首地址 |
| IN | count | 将发送的数据的个数 |
| IN | datatype | 发送数据的数据类型 |
| IN | dest | 接收消息的进程的标识号 |
| IN | tag | 消息标签 |
| IN | comm | 通信器 |

**C ：**

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

**F77 ：**

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

MPI_SEND将缓冲区中的count 个datatype 类型的数据发给进程号为dest 的目的进程。这里count是元素个数而不是字节数，数据空间的起始地址为buf。本次发送的消息标签是tag，使用标志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 － NPROCS-1(本文用NPROCS 或*p 表示*通信器comm 中的进程数)或MPI_PROC_NULL，tag的取值为0 － MPI_TAG_UB，count是指定数据类型的个数，而不是字节数。

在C和FORTRAN77的说明中，对void*和<type>需要进行特殊说明。MPI的库和一般的C和FORTRAN77库在语法上基本上是相同的，但是对于MPI的调用允许不同的数据类型使用相同的调用，比如对于数据的发送操作，整型、实型、字符型等都用一个相同的调用MPI_SEND。对于这样的数据类型在C和FORTRAN77的原型说明中分别用void *和<type>来表示，即用户可根据通信的要求，对不同的数据类型，可以用相同的调用。

### 3.2.6 接收消息

MPI_RECV(buf, count, datatype, source, tag, comm, status)

**参数：**

| | | |
|---|---|---|
| OUT | buf | 接收消息数据的首地址 |
| IN | count | 接收数据的最大个数 |
| IN | datatype | 接收消息的数据类型 |
| IN | source | 发送消息的进程标识号 |
| IN | tag | 消息标签 |
| IN | comm | 通信器 |
| OUT | status | 返回状态 |

**C ：**

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

**F77 ：**

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, IERR，STATUS(MPI_STATUS_SIZE)

MPI_RECV 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 － NPROCS-1，或MPI_ANY_SOURCE，或MPI_PROC_NULL，tag的取值为0 － MPI_TAG_UB或MPI_ANY_TAG。

在这个接收函数中，可以不指定SOURCE 和TAG，而分别用MPI_ANY_SOURCE 和MPI_ANY_TAG 来代替，用于接收任何进程发送的消息或任何编号的消息。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。

## 3.3 例：发送和接收消息

假设一共有$p$ 个进程，在进程编号为myid（myid= 0，1，…，$p$ -1）的进程中有一个整数 $m$，我们要把$m$ 传送到进程(myid + 1) mod $p$ 中。

```
      program ring
c
c The header file mpif.h must be included when you use MPI fuctions.
c
      include 'mpif.h'
      integer myid, p, mycomm, ierr, m, status(mpi_status_size),
     & next, front, mod, n
c
c Create MPI parallel environment and get the necessary data.
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
c
c Beginning the main parallel work on each process.
c
      m = myid
      front = mod(p+myid-1, p)
      next = mod(myid+1, p)
c
c Communication with each other.
c
      if(myid .eq. 0) then
      call mpi_recv(n, 1, mpi_integer, front, 1, mycomm, status, ierr)
      call mpi_send(m, 1, mpi_integer, next, 1, mycomm, ierr)
      m = n
      else
      call mpi_send(m, 1, mpi_integer, next, 1, mycomm, ierr)
      call mpi_recv(m, 1, mpi_integer, front, 1, mycomm, status, ierr)
      endif
```

```
c
c Ending of parallel work.
c
      print *, 'The value of m is ', m, ' on Process ', myid
      call mpi_comm_free(mycomm, ierr)
c
c Remove MPI parallel environment.
c
      call mpi_finalize(ierr)
      end
```

**注：**

**1．MPI 数据类型匹配和数据转换**

- 类型匹配规则：发送数据类型、通信函数中的数据类型、接收的数据类型要一致。
- 数据转换：1）类型转换：MPI 要求严格的类型匹配，所有不存在这个问题；
  2）数据表示的转换：主要发生在异构环境中。

**2．消息**

- 组成：1）信封：源 /目的，标识，通信器；
  2）数据：起始地址，个数，类型。
- 任意源和任意标识：MPI_ANY_SOURCE 和 MPI_ANY_TAG。

# 4. 简单的 MPI 程序示例

## 4.1 计时函数

### 4.1.1 返回墙上时间

MPI_WTIME()

**参数**：无

**C ：**

double MPI_Wtime(void)

**F77 ：**

DOUBLE PRECISION MPI_WTIME()

### 4.1.2 时钟精度

MPI_WTICK()

**参数**：无

**C ：**

double MPI_Wtick()

**F77 ：**

DOUBLE PRECISION MPI_WTICK()

在 C 中，这是唯一两个返回双精度值而非整型错误码的 MPI 函数；在 FORTRAN 中，这也是唯一两个 FUNCTION 形式的接口（其他均为 SUBROUTINE）。

## 4.2 获取结点的主机名和 MPI 版本号

### 4.2.1 MPI_GET_PROCESSOR_NAME

MPI_GET_PROCESSOR_NAME(name, namelen)

**参数**： OUT name 结点的主机名

OUT namelen 主机名的长度

**C ：**

int MPI_Get_processor_name(char *name, int *namelen)

**F77 ：**

MPI_GET_PROCESSOR_NAME(NAME, NAMELEN, IERR)

CHARACTER(*) NAME

INTEGER NAMELEN, IERR

### 4.2.2 MPI_GET_VERSION

MPI_GET_VERSION (version, subversion)

**参数**： OUT version 主版本号

OUT subversion 次版本号

**C ：**

int MPI_Get_version(int * version, int * subversion)

**F77 ：**

MPI_GET_VERSION(VERSION, SUBVERSION, IERR)

INTEGER VERSION, SUBVERSION, IERR

## 4.3 检测 MPI 是否初始化及异常中止 MPI

### 4.3.1 MPI_INITIALIED

MPI_INITIALIED(flag)

**参数：** OUT flag 如果 MPI 已经被调用，则 flag=true，否则 flag=false

**C ：**

int MPI_Initialized(int * flag)

**F77**：

MPI_INITIALIZED(FLAG, IERR)
LOGICAL FLAG
INTEGER IERR

### 4.3.2 MPI_ABORT

MPI_ABORT(comm, errorcode)

**参数：**

| | | | |
|---|---|---|---|
| IN | comm | 通信器 |
| IN | errorcode | 错误码 |

**C ：**

int MPI_Abort(MPI_Comm comm, int errorcode)

**F77 ：**

MPI_ABORT(COMM, ERRORCODE, IERR)
INTEGER COMM, ERRORCODE, IERR

调用该函数时表明因为出现了某种致命的错误而希望异常终止 MPI 程序的执行，MPI 系统会尽量设法终止通信器中的所有进程。

## 4.4 发送与接收组合进行

### 4.4.1 MPI_SENDRECV

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag,
recvbuf, recvcount, recvtype, source, recvtag, comm, status)

**参数：**

| | | | |
|---|---|---|---|
| IN | sendbuf | 发送缓冲区起始地址 |
| IN | sendcount | 发送数据的个数 |
| IN | sendtype | 发送数据的数据类型 |
| IN | dest | 目标进程的标识号 |
| IN | sendtag | 发送消息标签 |
| OUT | recvbuf | 接收缓冲区初始地址 |
| IN | recvcount | 最大接收数据个数 |
| IN | recvtype | 接收数据的数据类型 |
| IN | source | 源进程标识 |
| IN | recvtag | 接收消息标签 |
| IN | comm | 通信器 |
| OUT | status | 返回的状态 |

**C ：**

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

**F77 ：**

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,
RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG,
COMM, STATUS, IERR)

<type> SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT,RECVTYPE,
SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERR

这是发送消息和接收消息组合在一起的一个函数，好处是不用考虑先发送还是先接收消息，从而可以避免消息传递过程中的死锁。

### 4.4.2 MPI_SENDRECV_REPLACE

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag,
source, recvtag, comm, status)

**参数：**

| | | |
|---|---|---|
| INOUT | buf | 发送或接收缓冲区起始地址 |
| IN | count | 发送或接收数据的个数 |
| IN | datatype | 发送或接收数据的数据类型 |
| IN | dest | 目的进程的标识号 |
| IN | sendtag | 发送消息标签 |
| IN | source | 源进程标识 |
| IN | recvtag | 接收消息标签 |
| IN | comm | 通信器 |
| OUT | status | 返回状态 |

**C ：**

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

**F77 ：**

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG,
SOURCE, RECVTAG, COMM, STATUS, IERR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
COMM, STATUS(MPI_STATUS_SIZE), IERR

该函数的功能与 MPI_SENDRECV 类似，但发送和接收消息时使用同一个缓冲区 buf。

# 5. MPI 并行程序的两种基本模式

## 5.1 对等模式的 MPI 程序设计

### 5.1.1 Jacobi 迭代求解二维 Possion 方程

（略）

## 5.2 主从模式

### 5.2.1 矩阵向量的乘积

（略）

# 6. MPI 通信模式

MPI 有四种通信模式：标准模式，缓存模式，同步模式和就绪模式。这些消息发送函数具有完全一样的入口参数，但它们发送消息的方式或对接收方的状态要求不同。

## 6.1 标准模式 (standard mode)

由MPI 系统来决定是否对发送数据进行缓存：1）使用缓存：将发送数据拷贝至一个缓冲区后立即返回，此时消息发送由MPI在后台进行；2）不缓存数据：等待将数据发送出去后返回。大部分 MPI 系统预留了一定大小的缓冲区，当发送的消息长度小于缓冲区大小时会将数据缓存，如何立即返回，否则则当部分或全部数据发送完成后返回。标准模式是非局部的，因为它的完成可能需要与接收方联系。该模式的阻塞发送函数为 MPI_SEND。

## 6.2 缓存模式 (buffered mode)

MPI_BSEND(buf, count, datatype, dest, tag, comm)

**参数：**

| | | |
|---|---|---|
| IN | buf | 所发送消息的首地址 |
| IN | count | 将发送的数据的个数 |
| IN | datatype | 发送数据的数据类型 |
| IN | dest | 接收消息的进程的标识号 |
| IN | tag | 消息标签 |
| IN | comm | 通信器 |

**C ：**

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

**F77 ：**

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

MPI系统将发送消息拷贝至一个用户提供的缓冲区后立即返回，消息的发送由MPI系统在后台进行。用户必须确保所提供的缓冲区大小足够存放所有的发送数据。该模式是局部的。

MPI_BUFFER_ATTACH(buf, size)

**参数：**

| | | |
|---|---|---|
| IN | buf | 缓存起始地址 |
| IN | size | 大小（字节为单位） |

**C ：**

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

**F77 ：**

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

申请缓冲空间。

MPI_BUFFER_DETACH(buf, size)

**参数：**

| | | |
|---|---|---|
| IN | buf | 缓存起始地址 |
| IN | size | 大小（字节为单位） |

**C ：**

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

**F77 ：**

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

撤回由MPI_BUFFER_ATTACH申请的缓冲区。

## 6.3 同步模式(synchronous-mode)

MPI_SSEND (buf, count, datatype, dest, tag, comm)

**参数：**

| | | |
|---|---|---|
| IN | buf | 所发送消息的首地址 |
| IN | count | 将发送的数据的个数 |
| IN | datatype | 发送数据的数据类型 |
| IN | dest | 接收消息的进程的标识号 |
| IN | tag | 消息标签 |
| IN | comm | 通信器 |

**C ：**

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

**F77 ：**

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

同步发送必须确认接收方已经开始接收消息后才可以正确返回，属于非局部模式。

## 6.4 就绪模式(ready mode)

MPI_RSEND (buf, count, datatype, dest, tag, comm)

**参数：**

| | | |
|---|---|---|
| IN | buf | 所发送消息的首地址 |
| IN | count | 将发送的数据的个数 |
| IN | datatype | 发送数据的数据类型 |
| IN | dest | 接收消息的进程的标识号 |
| IN | tag | 消息标签 |
| IN | comm | 通信器 |

**C ：**

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

**F77 ：**

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

就绪模式发送时必须确保接收方已经处于就绪状态（正在等待接收该消息），否则该调用将产生一个错误。该模式设立的目的是在一些以同步方式工作的并行系统上，由于发送时可以假设接收方已经在等待接收而减少一些消息发送的开销。如果一个使用就绪模式的 MPI 程序是正确的，则将其中所有就绪模式的消息发送改为标准模式后也应该是正确的。

# 7. 阻塞型与非阻塞型通信

**阻塞型通信(blocking communication)**：阻塞型通信函数需要等待指定操作的实际完成，或者至少所涉及的数据已被 MPI 系统安全备份后才返回。MPI_SEND 和 MPI_RECV 都是阻塞型的。MPI_SEND 调用返回时表明数据已经发出或被 MPI 系统复制，随后对发送缓冲区的修改不会改变所发送的数据。而 MPI_RECV 返回时则表明数据接收已完成。阻塞型函数的操作是非局部的，使用不当很容易引起程序的死锁。

**非阻塞型通信(nonblocking communication)**：非阻塞型通信函数的调用总是立即返回，而实际操作由 MPI 系统在后台进行。用户必须随后调用其他函数来等待或查询操作完成的实际情况。在操作完成之前对相关数据区的操作是不安全的，因为随时可能与正在后台进行的操作发生冲突。非阻塞型函数的调用是局部的。在有些并行系统上，通过非阻塞型通信可以实现计算与通信的重叠进行，从而提高并行效率。

## 7.1 非阻塞发送和接收

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

| 参数： | | | |
|---|---|---|---|
| | IN | buf | 所发送消息的首地址 |
| | IN | count | 将发送的数据的个数 |
| | IN | datatype | 发送数据的数据类型 |
| | IN | dest | 接收消息的进程的标识号 |
| | IN | tag | 消息标签 |
| | IN | comm | 通信器 |
| | OUT | request | 请求句柄以备将来查询 |

**C ：**

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

**F77 ：**

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

该函数递交一个消息发送请求，要求MPI系统在后台完成消息的发送。MPI_ISEND为该发送创建一个请求并将请求的句柄通过request变量返回给MPI进程，供随后查询 /等待消息发送的完成用。

与阻塞消息发送一样，非阻塞消息发送也有四种模式：标准模式（MPI_ISEND），缓存模式（MPI_IBSEND），同步模式（MPI_ISSEND），就绪模式（MPI_IRSEND）。后三种模式较少使用，它们的发送函数与MPI_ISEND具有完全一样的入口参数，含意与阻塞消息发送相同。

MPI_IRECV(buf, count, datatype, source, tag, comm, request)

| 参数: | | | |
|---|---|---|---|
| | OUT | buf | 接收缓冲区的起始地址 |
| | IN | count | 接收数据的最大个数 |
| | IN | datatype | 数据类型 |
| | IN | source | 源进程的标识号 |
| | IN | tag | 消息标签 |
| | IN | comm | 通信器 |
| | OUT | request | 请求句柄以备将来查询 |

**C ：**

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

**F77 ：**

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

该函数递交一个消息接收请求，要求MPI系统在后台完成消息的接收。MPI_IRECV为该接收创建一个请求并将请求的句柄通过request变量返回给MPI进程，供随后查询 /等待消息接收的完成用。

## 7.2 通信请求的完成与检测

由于非阻塞通信的返回并不意味着该通信过程的完成，那么如何才能明确得知该非阻塞通信已经完成了呢？MPI提供两个调用MPI_WAIT和MPI_TEST用于这一目的。

### 7.2.1 等待、检测一个通信请求的完成

MPI_WAIT(request, status)

**参数：** INOUT request 通信请求
OUT status 发送或接收的状态

**C ：**

int MPI_Wait(MPI_Request * request, MPI_Status * status)

**F77 ：**

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

该函数是阻塞型的，它必须等待指定的通信请求完成后才能返回，与之相应的非阻塞型函数是MPI_TEST。成功返回时，status中包含关于所完成的通信的消息，相应的通信请求被释放，request被置成MPI_REQUEST_NULL。

MPI_TEST(request, flag, status)

**参数：** INOUT request 通信请求
OUT flag 操作是否完成标志
OUT status 发送或接收的状态

**C ：**

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

**F77 ：**

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

MPI_TEST 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，status 中包含关于所完成的通信的信息，相应的请求被释放，request 被置成 MPI_REQUEST_NULL。如果通信还未完成则返回 flag=false。

对于接收请求，以上两个函数中 status 返回的内容与 MPI_RECV 返回的一样；而对发送请求是没有定义的（值不确定），唯一可在发送操作中使用的 status 是查询函数 MPI_TEST_CANCELLED。

### 7.2.2 等待、检测一组通信请求中某一个的完成

MPI_WAITANY (count, array_of_requests, index, status)

**参数：** IN count 请求句柄的个数

| | | | |
|---|---|---|---|
| | INOUT | array_of_requests | 请求句柄数组 |
| | OUT | index | 已完成通信操作的句柄指标 |
| | OUT | status | 消息的状态 |

**C ：**

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

**F77 ：**

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

这个函数当所有请求句柄中至少有一个已经完成通信操作，就返回，如果有多于一个请求句柄已经完成，MPI_WAITANY 将随机选择其中的一个并立即返回。

MPI_TESTANY (count, array_of_requests, index, flag, status)

**参数：**

| | | | |
|---|---|---|---|
| | IN | count | 请求句柄的个数 |
| | INOUT | array_of_requests | 请求句柄数组 |
| | OUT | index | 已完成通信操作的句柄指标 |
| | OUT | flag | 如果有一个通信已完成，则flag=true |
| | OUT | status | 消息的状态 |

**C ：**

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

**F77 ：**

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

这个函数无论有没有通信操作完成都立即返回。

### 7.2.3 等待、检测一组通信请求的全部完成

MPI_WAITALL (count, array_of_requests, array_of_statuses)

**参数：**

| | | | |
|---|---|---|---|
| | IN | count | 请求句柄的个数 |
| | INOUT | array_of_requests | 请求句柄数组 |
| | OUT | array_of_statuses | 所有消息的状态数组 |

**C ：**

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

**F77 ：**

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

这个函数当所有的通信操作全部完成后才返回，否则将一直等待。

MPI_TESTALL (count, array_of_requests, flag, array_of_statuses)

**参数：**

| | | | |
|---|---|---|---|
| | IN | count | 请求句柄的个数 |
| | INOUT | array_of_requests | 请求句柄数组 |
| | OUT | flag | 如果有一个通信没完成，则flag=false |

OUT array_of_statuses 所有消息的状态数组

**C ：**

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

**F77 ：**

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

这个函数无论所有的通信操作是否全部完成都将立即返回。

### 7.2.4 等待、检测一组通信请求中一部分的完成

MPI_WAITSOME (incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

**参数：**

| | | |
|---|---|---|
| IN | incount | 请求句柄的个数 |
| INOUT | array_of_requests | 请求句柄数组 |
| OUT | outcount | 已完成的请求个数 |
| OUT | array_of_indices | 已完成请求的下标数组 |
| OUT | array_of_statuses | 所有消息的状态数组 |

**C ：**

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

**F77 ：**

```
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
     +             ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERR)
INTEGER  INCOUNT, OUTCOUNT, IERR
INTEGER  ARRAY_OF_REQUESTS(*), ARRAY_OF_INDICES(*)
INTEGER  ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *)
```

当指定的通信请求中至少有一个已完成或发生错误时，MPI_WAITSOME 才返回。outcount 中返回的是成功完成的通信请求个数，array_of_indices 的前 outcount 个元素给出已完成的通信请求在数组 array_of_requests 中的位置。

MPI_TESTSOME (incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

**参数：**

| | | |
|---|---|---|
| IN | incount | 请求句柄的个数 |
| INOUT | array_of_requests | 请求句柄数组 |
| OUT | outcount | 已完成的请求个数 |
| OUT | array_of_indices | 已完成请求的下标数组 |
| OUT | array_of_statuses | 所有消息的状态数组 |

**C ：**

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

**F77 ：**

```
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
     +             ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERR)
INTEGER  INCOUNT, OUTCOUNT, IERR
INTEGER  ARRAY_OF_REQUESTS(*), ARRAY_OF_INDICES(*)
```

INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *)

无论有无指定的通信请求完成，MPI_TESTSOME 都立即返回。如果都未完成则 outcount=0。

## 7.3 通信请求的释放

MPI_REQUEST_FREE(request)

**参数：** INOUT request 请求句柄，返回值为 MPI_REQUEST_NULL

**C ：**

int MPI_Request_free(MPI_Request * request)

**F77 ：**

MPI_REQUEST_FREE(REQUEST, IERR)

INTEGER REQUEST, IERR

如果能够确认一个通信操作已完成，则可以直接调用该函数将该对象所占用的资源释放。而不是通过调用通信请求的完成操作来间接地释放。若该通信请求相关联的通信操作尚未完成，则 MPI_REQUEST_FREE 会等待通信的完成，因此通信请求的释放并不影响该通信的完成。函数成功返回后 request 被置为 MPI_REQUEST_NULL。一旦执行了释放操作，该通信请求就无法再通过其它任何的调用访问。

## 7.4 消息探测和通信请求的取消

### 7.4.1 消息探测

MPI_PROBE(source, tag, comm, status)

**参数：**

| | | |
|---|---|---|
| IN | source | 发送消息的进程标识号 |
| IN | tag | 接收消息的标签 |
| IN | comm | 通信器 |
| OUT | status | 返回到达消息的状态 |

**C ：**

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

**F77 ：**

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERR)

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

该函数用于检测一个符合条件的消息是否到达。它是阻塞型函数，必须等到一个符合条件的消息到达后才返回。

MPI_IPROBE(source, tag, comm, flag, status)

**参数：**

| | | |
|---|---|---|
| IN | source | 发送消息的进程标识号 |
| IN | tag | 接收消息的标签 |
| IN | comm | 通信器 |
| OUT | flag | 如果指定消息已经达到, flag返回值为true |
| OUT | status | 返回到达消息的状态 |

**C ：**

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

**F77 ：**

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERR)

LOGICAL FLAG

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

非阻塞型函数，即不论是否有符合条件的消息，MPI_IPROBE 都立即返回。如果符合条件的消息已到达，则 flag 为非零值（C）或真（FORTRAN），否则为 0（C）或假（FORTRAN）。

这两个函数中的参数 source 可以是 MPI_ANY_SOURCE，tag 也可以是 MPI_ANY_TAG，以便用户可以检查来自不确定的源 source 以及不确定的标识 tag，但必须指定通信器 comm。

### 7.4.2 通信请求的取消

MPI_CANCEL(request)

**参数：** IN request 通信请求

**C ：**

int MPI_Cancel(MPI_Request * request)

**F77 ：**

MPI_CANCEL(REQUEST, IERR)

INTEGER REQUEST, IERR

该函数用于取消一个尚未完成的通信请求，它在MPI系统中设置一个取消该通信请求的标志后立即返回，具体的取消操作由MPI系统在后台完成。MPI_CANCEL允许取消已调用的通信请求，从而释放发送或接收操作所占用的资源，但并不意味着相应的通信一定会被取消：当取消操作调用时，若相应的通信请求已经开始，则它会正常完成，不受取消操作的影响；若相应的通信请求还没开始，则可以释放通信占用的资源。

调用MPI_CANCEL后，仍需用MPI_WAIT，MPI_TEST或MPI_REQUEST_FREE来释放该通信请求。

MPI_TEST_CANCELLED(status, flag)

**参数：** IN status 消息状态

OUT flag 如果已取消，则 flag=true

**C ：**

int MPI_Test_cancelled(MPI_Status status, int * flag)

**F77 ：**

MPI_ TEST_CANCELLED(STATUS, FLAG, IERR)

LOGICAL FLAG

INTEGER IERR, STATUS(MPI_STATUS_SIZE)

该函数用来检查一个通信请求是否已被取消。返回flag=true表示该通信请求已被取消。

## 7.4 点对点通信函数汇总

| 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 |

## 7.5 持久通信请求

持久通信(persistent communication request)请求可用于以完全相同的方式（相同的通信器、收发缓冲、数据长度、源/目地址和消息标签）重复收发的消息。目的是减少处理消息所需的开销，并简化 MPI 程序。

### 7.5.1 创建持久消息发送请求

MPI_SEND_INIT(buf,count,datatype,dest,tag,comm,request)

**参数：**

| | | |
|---|---|---|
| IN | buf | 所发送消息的首地址 |
| IN | count | 将发送的数据的个数 |
| IN | datatype | 发送数据的数据类型 |
| IN | dest | 接收消息的进程的标识号 |
| IN | tag | 消息标签 |
| IN | comm | 通信器 |
| OUT | request | 请求句柄以备将来查询 |

**C ：**

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

**F77 ：**

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

该函数并不开始消息的实际发送，而只是创建一个请求句柄，通过参数 request 返回给用户程序，留待以后实际发送时用。MPI_SEND_INIT 对应于标准的非阻塞型消息发送，相应地还有函数 MPI_BSEND_INIT，MPI_SSEND_INIT 和 MPI_RSEND_INIT，分别对应于缓冲模式、同步模式和就绪模式的非阻塞型发送。

### 7.5.2 创建持久消息接收请求

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

**参数：**

| | | |
|---|---|---|
| OUT | buf | 接收缓冲区的起始地址 |
| IN | count | 接收数据的最大个数 |
| IN | datatype | 接收数据的数据类型 |
| IN | source | 发送进程的标识号 |
| IN | tag | 消息标签 |
| IN | comm | 通信器 |
| OUT | request | 请求句柄以备将来查询 |

**C ：**

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

**F77 ：**

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

与 MPI_SEND_INIT 一样，该函数并不进行消息的实际接收，而只是创建一个请求句柄并返回给用户程序，留待以后实际接收时用。

MPI_SEND_INIT 和 MPI_RECV_INIT 是二个持久性通信函数，它们的参数和前面讲到的非阻

塞通信中的发送与接收相同，由此可以把通信以通道方式建立起来对应关系，从而提高通信效率。但是仅有这二个函数还不能够达到通信的目的，它们还需要用 MPI_START 或 MPI_START_ALL 来激活。

### 7.5.3 持久通信的激活

MPI_START(request)

**参数：** INOUT request 通信请求句柄

**C ：**

int MPI_Start(MPI_Request * request)

**F77 ：**

MPI_START(REQUEST, IERR)

INTEGER REQUEST, IERR

该函数用来激活 request 所对应的持久通信操作。

MPI_STARTALL(count, array_of_requests)

**参数：** IN count 需要激活的通信请求的个数

IN array_of_requests 通信请求句柄数组

**C ：**

int MPI_Startall(int count, MPI_Request * array_of_requests)

**F77 ：**

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERR)

INTEGER COUNT, IERR, ARRAY_OF_REQUESTS(*)

该函数的一次调用等价于用MPI_START调用array_of_requests中的每一个通信请求。

每次调用MPI_START相当于调用一次与MPI_XXXX_INIT相对应的非阻塞通信函数（如MPI_ISEND，MPI_IRECV等），即它们产生的通信效果是一样的。

### 7.5.4 持久通信请求的完成与释放

由于每次通信完成后，通信请求并未被释放，所有由一个MPI_XXXX_INIT创建的持久通信请求可反复调用MPI_START或MPI_START_ALL来完成多次通信。与普通的非阻塞通信一样，通过持久通信请求启动的通信也需要用MPI_WAIT，MPI_TEST，MPI_CANCEL等函数来等待或查询通信的完成情况或取消通信，或是用MPI_REQUEST_FREE来释放通信请求。

一个用MPI_START激发的发送操作可以和任何接收操作匹配。同样，一个用MPI_START激发的接收操作可以接收任何发送操作产生的消息。

# 8. 聚合通信

聚合通信(collective communication)是指多个进程(通常大于2)之间的通信。根据数据的流向，聚合通信可分为三种类型：一对多（一个进程对多个进程）；多对一（多个进程对一个进程）；多对多（多个进程对多个进程）。在一对多和多对一操作中，有一个进程扮演着特殊的角色，称为该操作的根进程（root）。聚合通信一般实现三个功能：通信，同步和计算。

## 8.1 障碍同步

MPI_BARRIER(comm)

**参数：** IN comm 通信器

**C ：**

int MPI_Barrier(MPI_Comm comm)

**F77 ：**

MPI_BARRIER(COMM, IERR)

INTEGER COMM, IERR

这是MPI 提供的唯一的一个同步函数，当COMM通信器中的所有进程都执行这个函数时才返回，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。

## 8.2 广播

MPI_BCAST(buf, count, datatype, root, comm)

**参数：**

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

**C ：**

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

**F77 ：**

MPI_BCAST(BUF, COUNT, DATATYPE, ROOT, COMM, IERR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, ROOT, COMM, IERR

MPI_BCAST是典型的一对多通信函数。通信器comm中的root进程将自己buf中的内容广播发送到组内的所有进程（包括它自身）。此函数在并行程序中经常出现，因此是个必须很好掌握的MPI通信函数。

## 8.3 数据收集

数据收集指各个进程（包括根进程）将自己的一块数据发送给根进程，根进程将这些数据合并成一个更大的数据块。

### 8.3.1 收集相同长度的数据块

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root , comm)

**参数：**

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| IN | sendcount | 发送消息缓冲区中的数据个数 |
| IN | sendtype | 发送消息缓冲区中的数据类型 |

| | | | |
|---|---|---|---|
| | OUT | recvbuf | 接收消息缓冲区的起始地址(仅对于根进程有意义) |
| | IN | recvcount | 待接收的数据个数(仅对于根进程有意义) |
| | IN | recvtype | 接收数据的数据类型(仅对于根进程有意义) |
| | IN | root | 接收进程的标识号 |
| | IN | comm | 通信器 |

**C** ：

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

**F77** ：

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,
+                RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
INTEGER  ROOT, COMM, IERR
```

MPI_GATHER是典型的多对一通信函数。每个进程（包括根进程）将其发送缓冲区中的消息发送给根进程，根进程将这些数据按发送进程的标识号依次存放到自己的recvbuf中。和广播调用不同的是：广播出去的数据都是相同的，但收集操作从各个进程收集到的数据一般是互不相同。但从各个进程收集到的数据的类型和个数必须相同。参数recvbuf，recvcount和recvtype仅对根进程有意义，这里的recvcount通常等于sendcount。假设通信器中有np个进程，则该函数的作用相当于所有进程（包括root）都执行了一个发送调用，同时根进程执行了np次接收调用，即：

```
CALL MPI_SEND(SENDBUF, SENDCOUNT, SENDTYPE, ROOT, ...)
IF ( MYID .EQ. ROOT ) THEN
  DO I=0, NP-1
    CALL  MPI_RECV(RECVBUF + I*RECVCOUNT*extent(RECVTYPE),
+                  RECVCOUNT, RECVTYPE, I, ...)
  ENDDO
ENDIF
```

### 8.3.2 收集不同长度的数据块

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

**参数**：

| | | | |
|---|---|---|---|
| | IN | sendbuf | 发送消息缓冲区的起始地址 |
| | IN | sendcount | 发送消息缓冲区中的数据个数 |
| | IN | sendtype | 发送消息缓冲区中的数据类型 |
| | OUT | recvbuf | 接收消息缓冲区的起始地址(仅对于根进程有意义) |
| | IN | recvcounts | 从每个进程接收的数据个数，整型数组(仅对于根进程有意义) |
| | IN | displs | 接收数据的存放位置，整型数组 |
| | IN | recvtype | 接收消息缓冲区中数据类型(仅对于根进程有意义) |
| | IN | root | 接收进程的标识号 |
| | IN | comm | 通信器 |

**C** ：

```
int  MPI_Gatherv(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                 void * recvbuf, int * recvcounts, int * displs, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

**F77** ：

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
```

```
     +                    DISPLS, RECVTYPE, ROOT, COMM, IERR)
      <type>   SENDBUF(*), RECVBUF(*)
      INTEGER   SENDCOUNT, SENDTYPE, , RECVTYPE, ROOT, COMM, IERR
      INTEGER   RECVCOUNTS(*), DISPLS(*)
```

这个函数是MPI_GATHER的扩充，它允许从不同的进程中接收不同长度的消息，并且根进程接收到的消息也不一定按进程号连续存放，而是为每一个接收消息在根进程的recvbuf中的位置提供了一个位置偏移displs数组，所以接收到的消息可以存放在接收缓冲区的不同位置，因此比MPI_GATHER更灵活。根进程中的recvcounts(*i*)必须与第*i*个进程中的sendcount相一致，0<=*i*<=np-1。该函数相当于

```
      INTEGER DISPLS(0:NP-1), RECVCOUNTS(0:NP-1)
        ... ...
      CALL MPI_SEND(SENDBUF, SENDCOUNT, SENDTYPE, ROOT, ...)
      IF ( MYID .EQ. ROOT ) THEN
        DO I=0, NP - 1
         CALL MPI_RECV(RECVBUF + DISPLS(I)*extent(RECVTYPE),
    +                  RECVCOUNTS(I), RECVTYPE, I, ...)
        ENDDO
      ENDIF
```

## 8.4 数据散发

数据散发是指根进程将一个大的数据块分成小块分别散发给各个进程（包括根进程自己）。它是数据收集的逆操作。

### 8.4.1 散发相同长度的数据块

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

**参数：**

| | | | |
|---|---|---|---|
| | IN | sendbuf | 发送消息缓冲区的起始地址 |
| | IN | sendcount | 发送到各个进程的数据个数 |
| | IN | sendtype | 发送消息缓冲区中的数据类型 |
| | OUT | recvbuf | 接收消息缓冲区的起始地址 |
| | IN | recvcount | 待接收的元素个数 |
| | IN | recvtype | 接收元素的数据类型 |
| | IN | root | 发送进程的标识号 |
| | IN | comm | 通信器 |

**C ：**

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

**F77 ：**

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
+                  RECVTYPE, ROOT, COMM, IERR)
 <type>   SENDBUF(*), RECVBUF(*)
 INTEGER   SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERR
```

根进程 root 的 sendbuf 中有 NP 个连续存放的数据块，每个数据块包含 sendcout 个 sendtype 类型的数据，根进程将这些数据块按进程的标识号依次分发给各个进程（包括 root 自己）。参数 sendbuf，sendcount 和 sendtype 仅对根进程有意义。

### 8.4.2 散发不同长度的数据块

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf,
recvcount, recvtype, root,comm)

参数：

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| IN | sendcounts | 发送到每个进程的数据个数（整数数组） |
| IN | displs | 发送到每个进程的数据起始位移（整数数组） |
| IN | sendtype | 发送消息缓冲区中数据的类型 |
| OUT | recvbuf | 接收消息缓冲区的起始地址 |
| IN | recvcount | 接收数据的个数 |
| IN | recvtype | 接收缓冲区的数据类型 |
| IN | root | 发送进程的标识号 |
| IN | comm | 通信器 |

**C ：**

```
int MPI_Scatterv(void * sendbuf, int * sendcounts, int * displs, MPI_Datatype sendtype,
                 void * recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

**F77 ：**

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
+            RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), DISPLS(*)
INTEGER SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERR
```

MPI_SCATTERV 是MPI_SCATTER 的扩展，它允许root向各个进程发送的数据块的长度不同，而且发送数据并不一定是连续存放的。数组sendcounts 和displs 的元素个数必须等于进程数，它们分别给出root 发给每个进程的数据长度和数据在sendbuf 中的偏移量（以sendtype 为单位）。这里第i 个进程中的recvcount 必须和root 的sendcounts(i) 相同。

## 8.5 多点对多点的通信

### 8.5.1 相同长度数据块的全收集

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

参数：

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

**C ：**

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

**F77 ：**

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE,
+             RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERR)
<type> SENDBUF(*), RECVBUF(*)
```

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERR

MPI_ALLGATHER 与 MPI_GATHER 类似，区别是所有进程同时将数据收集到自己的 recvbuf 中，而不仅仅是 root 进行收集，因此称为全收集。

MPI_ALLGATHER 等价于依次以每个进程为根进程进 np 次 MPI_GATHER。也可以认为是以任意进程为根进程调用一次 MPI_GATHER，紧接着再对收集到的数据进行一次广播。

### 8.5.2 不同长度数据块的全收集

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

**参数：**

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| IN | sendcount | 发送消息缓冲区中的数据个数 |
| IN | sendtype | 发送消息缓冲区中的数据类型(句柄) |
| OUT | recvbuf | 接收消息缓冲区的起始地址 |
| IN | recvcounts | 接收数据的个数（整型数组） |
| IN | displs | 接收数据的存放位置（整型数组） |
| IN | recvtype | 接收消息缓冲区的数据类型 |
| IN | comm | 通信器 |

**C ：**

```
int MPI_Allgatherv(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                   void * recvbuf, int * recvcounts, int * displs,
                   MPI_Datatype recvtype, MPI_Comm comm)
```

**F77 ：**

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE,
               RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVTYPE, COMM, IERR
INTEGER RECVCOUNTS(*), DISPLS(*)
```

该函数用于不同长度数据块的全收集，它的参数与MPI_GATHERV类似。这里进程 ***j*** 的sendcount必须与所有进程中的recvcounts(***j***)相同。

### 8.5.3 相同数据长度的全收集散发

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。我们称该操作为数据的“全收集散发”，它既可以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(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

**参数：**

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

**C ：**

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

**F77 ：**

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE,
RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERR

第 ***i*** 个进程将 sendbuf 中的第 ***j*** 块数据发送至第 ***j*** 个进程 recvbuf 中的第 ***i*** 个位置，***i***, ***j***=0,…, *np*-1。sendbuf 和 recvbuf 均有 np 个连续的数据块构成。

该操作相对于将数据/进程进行一次转置。例如，假设一个二维数组按行分块存储在各个进程中，则该调用可以很容易地将它变成按列分块存储在各个进程中。

### 8.5.4 不同数据长度的全收集散发

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype,
recvbuf, recvcounts, rdispls, recvtype, comm)

**参数：**

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| IN | sendcounts | 向每个进程发送的数据个数(整型数组) |
| IN | sdispls | 向每个进程发送数据的位移（整型数组） |
| IN | sendtype | 发送数据的数据类型 |
| OUT | recvbuf | 接收消息缓冲区的起始地址 |
| IN | recvcounts | 从每个进程中接收的数据个数(整型数组) |
| IN | rdispls | 从每个进程接收的数据在接收缓冲区的位移（整型数组） |
| IN | recvtype | 接收数据的数据类型 |
| IN | comm | 通信器 |

**C ：**

int MPI_Alltoallv(void * sendbuf, int * sendcounts, int * sdispls, MPI_Datatype sendtype,
void * recvbuf, int * recvcounts, int * rdispls, MPI_Datatype recvtype,
MPI_Comm comm)

**F77 ：**

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
+ RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE,
INTEGER RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, IERR

与 MPI_ALLTOALL 类似，但每个数据块的长度可以不等，并且在 recvbuf 中也不要求连续存放。

## 8.6 归约

### 8.6.1 普通归约

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

**参数：**

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| OUT | recvbuf | 接收消息缓冲区中的地址 |
| IN | count | 发送消息缓冲区中的数据个数 |
| IN | datatype | 发送消息缓冲区的数据类型 |
| IN | op | 归约操作符 |
| IN | root | 根进程标签号 |
| IN | comm | 通信器 |

**C ：**

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

**F77 ：**

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERR

该函数将通信器内每个进程输入缓冲区（sendbuf）中的数据按给定的操作op进行运算，并将其结果返回到root进程的输出缓冲区（recvbuf）中。所有进程都提供长度相同、类型相同的数据。op指定归约使用的运算，它在C中的类型是MPI_Op，在Fortran中则是个句柄。MPI的归约函数要求运算op满足结合律，但可以不满足交换律（MPI预定义的运算均满足交换律）。运算可以是MPI预定义的，也可以是用户自行定义的。参数recvbuf只对根进程有意义。

**MPI 中预定义的运算操作（op）**

| 操作名 | 含义 | 操作名 | 含义 |
|---|---|---|---|
| MPI_MAX | 求最大值 | MPI_LOR | 逻辑或 |
| MPI_MIN | 求最小值 | MPI_BOR | 二进制按位或 |
| MPI_SUM | 求和 | MPI_LXOR | 逻辑异或 |
| MPI_PROD | 求积 | MPI_BXOR | 二进制按位异或 |
| MPI_LAND | 逻辑与 | MPI_MAXLOC | 求最大值和所在位置 |
| MPI_BAND | 二进制按位与 | MPI_MINLOC | 求最小值和所在位置 |

这些运算是有数据类型要求的，首先对数据类型进行如下分类：

| | |
|---|---|
| C integer: | MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_LONG |
| Fortran integer: | MPI_INTEGER |
| Floating point: | MPI_REAL, MPI_DOUBLE, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE |
| Logical: | MPI_LOGICAL |
| Complex: | MPI_COMPLEX |
| Byte: | MPI_BYTE |

现在对每种操作允许的数据类型规定如下：

| 运算操作 OP | 允许的数据类型 |
|---|---|
| MPI_MAX, MPI_MIN | C integer, Fortran integer, Floating point |
| MPI_SUM, MPI_PROD | C integer, Fortran integer, Floating point, Complex |
| MPI_LAND, MPI_LOR, MPI_LXOR | C integer, Logical |
| MPI_BAND, MPI_BOR, MPI_BXOR | C integer, Fortran integer, Byte |

MPI_MAXLOC和MPI_MINLOC是两个特殊的运算，它们不仅求最值，而且还会记下最值的位置，所以需要一类特殊的数据类型－－数对。MPI为它们定义了下面数据类型。

**MPI定义的Fortran语言的值对类型**

| 名字 | 描述 |
|---|---|

| | |
|---|---|
| MPI_2REAL | 实型值对 {REAL, REAL} |
| MPI_2DOUBLE_PRECISION | 双精度变量值对 {DOUBLE_PRECISION, DOUBLE_PRECISION} |
| MPI_2INTEGER | 整型值对 {INTEGER, INTEGER} |

MPI定义的C语言的值对类型

| 名字 | 描述 |
|---|---|
| MPI_FLOAT_INT | 浮点型和整型 {float, int} |
| MPI_DOUBLE_INT | 双精度和整型 {double, int} |
| MPI_LONG_INT | 长整型和整型 {long, int} |
| MPI_2INT | 整型值对 {int, int} |
| MPI_SHORT_INT | 短整型和整型 {short, int} |
| MPI_LONG_DOUBLE_INT | 长双精度浮点型和整型 {long double, int} |

用户也可以利用MPI_OP_CREATE函数自己定义运算操作。

### 8.6.2 用户自定义的归约操作

MPI_OP_CREATE(func, commute, op)

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

**C ：**

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

**F77 ：**

```
MPI_OP_CREATE(FUNC, COMMUTE, OP, IERR)
EXTERNAL FUNC
LOGICAL COMMUTE
INTEGER OP, IERR
```

该函数创建一个新的运算，参数中 func 是用户提供的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。MPI_OP_CREATE 将用户自定义的函数 func 和操作 op 联系起来，一个运算被创建后便和 MPI 预定义的运算操作一样，可以用在各中归约和扫描函数中。

在 MPI 中，用户自定义的外部函数 func 是有严格要求的，必须具有如下形式的接口：

**C ：**

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

**F77 ：**

```
FUNCTION FUNC(INVEC, INOUTVEC, LEN, DATATYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, DATATYPE
```

invec 和inoutvec 分别指出将要被归约的数据所在的缓冲区的首址，len 指出将要归约的元素个数，datatype 指出归约对象的数据类型，函数的返回结果保存在inoutvec 中。

当一个用户自定义的运算不再需要时，可以将其释放，以释放其占用的系统资源。

MPI_OP_FREE(op)

**参数：** IN op 运算操作

**C ：**

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

**F77 ：**

```
MPI_OP_FREE(OP, IERR)
INTEGER  OP, IERR
```

该函数将用户自定义的归约操作撤消，并将op设置成MPI_OP_NULL。

### 8.6.3 全归约

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
```

**参数：**

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| OUT | recvbuf | 接收消息缓冲区的起始地址 |
| IN | count | 发送消息缓冲区中的数据个数 |
| IN | datatype | 发送消息缓冲区中的数据类型 |
| IN | op | 运算操作 |
| IN | comm | 通信器 |

**C ：**

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

**F77 ：**

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER  COUNT, DATATYPE, OP, COMM, IERR
```

MPI_ALLREDUCE 是 MPI_REDUCE 的扩充，该函数被调用后所有进程的 recvbuf 将同时获得归约运算的结果，其作用相当于在 MPI_REDUCE 后马上再将结果进行一次广播。MPI_ALLREDUCE 除了比 MPI_REDUCE 少一个参数外，其它参数及含义都与后者一样。

### 8.6.4 归约散发

```
MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)
```

**参数：**

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| OUT | recvbuf | 接收消息缓冲区的起始地址 |
| IN | recvcounts | 接收数据个数整型数组 |
| IN | datatype | 发送缓冲区中的数据类型 |
| IN | op | 运算操作 |
| IN | comm | 通信器 |

**C ：**

```
int  MPI_Reduce_scatter(void * sendbuf, void * recvbuf, int * recvcounts
                        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

**F77 ：**

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE,
+                  OP, COMM, IERROR)
<type>  SENDBUF(*), RECVBUF(*)
INTEGER  DATATYPE, OP, COMM, IERR, RECVCOUNTS(*)
```

该函数将归约结果散发给所有的进程，而不是仅仅归约到 ROOT 进程。MPI_REDUCE_SCATTER 对由 sendbuf，count 和 datatype 指定的发送缓冲区数组的元素逐个进行归约操作，这里 $\mathbf{count}=\sum_{k=1}^{np}\mathbf{recvcounts}(k)$。然后将归约结果（数组）进行散发，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。

### 8.6.5 扫描

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

**参数：**

| | | |
|---|---|---|
| IN | sendbuf | 发送消息缓冲区的起始地址 |
| OUT | recvbuf | 接收消息缓冲区的起始地址 |
| IN | count | 输入缓冲区中元素的个数 |
| IN | datatype | 输入缓冲区中元素的类型 |
| IN | op | 运算操作 |
| IN | comm | 通信器 |

**C ：**

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

**F77 ：**

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERR)
<type>    SENDBUF(*), RECVBUF(*)
INTEGER    COUNT, DATATYPE, OP, COMM, IERR
```

该函数可以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。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。也可以换一个角度，将扫描操作看作是每一个进程***i***发送缓冲区中的数据与它前面的进程***i***-1接收缓冲区中的数据进行指定的归约操作，然后将结果存入进程***i***的接收缓冲区，而进程***i***接收缓冲区中的数据用来和进程***i***+1发送缓冲区中的数据进行归约，进程0接收缓冲区中的数据就是其发送缓冲区的数据。

# 9. MPI 系统的数据类型

MPI 的消息收发函数只能处理连续存储的同一类型的数据。当需要在消息中发送或接收具有复杂结构的数据时，需要通过用户定义新的数据类型（称为派生数据类型）或将数据打包来实现。

## 9.1 与数据类型有关的一些定义

### 9.1.1 数据类型定义

一个MPI 数据类型可以由**类型图（type map）**来描述，两个数据类型是否相同，取决于它们的类型图是否相同。类型图是一系列二元组的集合：

typemap = { ($type_0$, $disp_0$), ($type_1$, $disp_1$), …, ($type_{n-1}$, $disp_{n-1}$) }。

由此可知，一个 MPI 数据类型由两个 n 元序列构成，第一个序列包含一组数据类型，称为**类型序列(type signature)**：

typesig = { $type_0$, $type_1$, …, $type_{n-1}$ }。

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

typedisp = { $disp_0$, $disp_1$, …, $disp_{n-1}$ }。

typesig 中的类型 $type_i$ (i=0, …, n-1) 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 的数据类型是可以嵌套定义的。Typedisp 中的位移 $disp_i$ 是以字节为单位的，可正可负，也没有递增或递减的要求。为了以示区别，我们称非原始数据类型为**派生数据类型（derived datatype）**。

MPI 预定义的原始数据类型可以看作是派生数据类型的一种特例，如 MPI_INT 对应的类型图为{ (int, 0) }。

**例 9.1** 假设数据类型 TYPE 的类型图为

TYPE = { (REAL, 4), (REAL, 12), (REAL, 0) }

则语句：

```
REAL    A(100)
…   …
CALL    MPI_SEND(A, 1, TYPE, … )
```

将发送 A(2), A(4), A(1)。

**例 9.2** 假设数据类型 TYPE 的类型图为

TYPE = { (REAL, －4), (REAL, 0), (REAL, 4) }

则语句：

```
REAL    A(4)
…   …
CALL    MPI_SEND(A(3), 1, TYPE, … )
```

将发送 A(2), A(3), A(4)。

### 9.1.2 数据类型的大小

数据类型的大小（size）是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。设一个数据类型的类型图为：

typemap = { ($type_0$, $disp_0$), ($type_1$, $disp_1$), …, ($type_{n-1}$, $disp_{n-1}$) }，

则该数据类型的大小为：

$$sizeof(\text{typemap}) = \sum_{i=0}^{n-1} sizeof(\text{type}_i)。$$

### 9.1.3 数据类型的下界、上界和跨度

设一个数据类型的类型图为

typemap = { ($type_0$, $disp_0$), ($type_1$, $disp_1$), …, ($type_{n-1}$, $disp_{n-1}$) }。

则该数据类型的下、上界和跨度（extent）分别为

$$\text{lb(typemap)} = \min_i \{\text{disp}_i\}$$

$$\text{up(typemap)} = \max_i \{\text{disp}_i + sizeof(\text{type}_i)\} + \varepsilon$$

$$\text{extent(typemap)} = \text{up(typemap)} - \text{lb(typemap)}$$

其中，$\varepsilon$ 是地址对界修正量，它使得 extent 能被该数据类型的对界量整除的最小非负整数。

一个数据类型的对界量定义如下：原始数据类型的对界量由编译系统决定，而派生数据类型的对界量定义为它的所有基本数据类型的对界量的最大值。地址对界要求（alignment）是指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。

### 9.1.4 MPI_LB 和 MPI_UB

MPI 系统提供了两个特殊的数据类型 MPI_LB 和 MPI_UB，称为**伪数据类型（pseudo datatype）**。它们的长度为 0，即不占任何空间，并且当它们出现在数据类型的类型图中时，对该数据类型的实际内容不起任何作用。它们的作用是让用户可以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 type 的基本数据类型中含有 MPI_LB，则它的下界定义为：

$$\text{lb(type)} = \min_i \{\text{disp}_i \mid \text{type}_i = \text{MPI\_LB}\};$$

类似地，如果一个数据类型 type 的基本数据类型中含有 MPI_UB，则它的上界定义为：

$$\text{ub(type)} = \max_i \{\text{disp}_i \mid \text{type}_i = \text{MPI\_UB}\}。$$

## 9.2 数据类型创建函数

### 9.2.1 连续复制

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

**参数：**

| | | | |
|---|---|---|---|
| | IN | count | 复制个数 |
| | IN | oldtype | 旧数据类型 |
| | OUT | newtype | 新数据类型 |

**C ：**

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype * newtype)
```

**F77 ：**

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERR
```

该函数将原数据类型 oldtype 按顺序依次连续复制后，得到一个新的数据类型。

### 9.2.2 向量数据类型的生成

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

**参数：**

| | | |
|---|---|---|
| IN | count | oldtype的块的个数 |
| IN | blocklength | 每个块中所含元素个数 |
| IN | stride | 各块第一个元素之间相隔的元素个数 |
| IN | oldtype | 旧数据类型 |
| OUT | newtype | 新数据类型 |

**C ：**

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype * newtype)
```

**F77 ：**

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERR)
INTEGER  COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERR
```

该函数首先通过连续复制 blocklength 个旧数据类型 oldtype 形成一个数据块，然后通过等间隔地复制该数据块而形成新数据类型。newtype 中包含 count 个这样的数据块，相邻两个数据块之间的位移相差 stride*extent(oldtype)个字节。

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)

**参数：**

| | | |
|---|---|---|
| IN | count | oldtype的块的个数 |
| IN | blocklength | 每个块中所含元素个数 |
| IN | stride | 各块起始位置之间相隔的字节数 |
| IN | oldtype | 旧数据类型 |
| OUT | newtype | 新数据类型 |

**C ：**

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
                     MPI_Datatype oldtype, MPI_Datatype * newtype)
```

**F77 ：**

```
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERR)
INTEGER  COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERR
```

该函数与 MPI_VECTOR 基本相同，只是 stride 不是元素个数，而是字节数。

### 9.2.3 索引数据类型的生成

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacemets, oldtype, newtype)

**参数：**

| | | |
|---|---|---|
| IN | count | oldtype的块的个数 |
| IN | array_of_blocklengths | 每个块中所含元素个数(非负整数数组) |
| IN | array_of_displacements | 各块偏移值(整数数组) |
| IN | oldtype | 旧数据类型 |
| OUT | newtypr | 新数据类型 |

**C ：**

```
int MPI_Type_indexed(int count, int * array_of_blocklengths, int * array_of_displacements,
                     MPI_Datatype oldtype, MPI_Datatype * newtype)
```

**F77 ：**

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
     +                  ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERR)
INTEGER  COUNT, OLDTYPE, NEWTYPE, IERR
```

INTEGER ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*)

该函数生成的新数据类型由 count 个数据块构成，其中第 i 个数据块包含 array_of_bloklengths(i)个连续存放的 oldtype，其字节位移是 array_of_displacements(i)*extent(oldtype)。

MPI_TYPE_INDEXED 与 MPI_TYPE_VECTOR 的区别在于每个数据块的长度可以不同，数据块之间的间隔也可以不同。

MPI_TYPE_HINDEXED(count, array_of_blocklengths, array_of_displacemets, oldtype, newtype)

**参数：**

| | | | |
|---|---|---|---|
| IN | count | oldtype的块的个数 |
| IN | array_of_blocklengths | 每个块中所含元素个数(非负整数数组) |
| IN | array_of_displacements | 各块的偏移字节数(整数数组) |
| IN | oldtype | 旧数据类型 |
| OUT | newtypr | 新数据类型 |

**C ：**

```
int MPI_Type_hindexed(int count, int * array_of_blocklengths,
                      MPI_Aint * array_of_displacements,
                      MPI_Datatype oldtype, MPI_Datatype * newtype)
```

**F77 ：**

```
MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
+                 ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERR
INTEGER ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*)
```

该函数与 MPI_TYPE_INDEXED 基本相同，唯一的区别是 array_of_displacements 以字节为单位。

### 9.2.4 结构数据类型的生成

MPI_TYPE_STRUCT(count, array_of_blocklengths,
array_of_displacemets, array_of_types, newtype)

**参数：**

| | | | |
|---|---|---|---|
| IN | count | oldtype的块的个数 |
| IN | array_of_blocklengths | 每个块中所含元素个数(整数数组) |
| IN | array_of_displacements | 各块偏移字节数(整数数组) |
| IN | array_of_types | 每个块中元素的类型 |
| OUT | newtypr | 新数据类型 |

**C ：**

```
int MPI_Type_struct(int count, int * array_of_blocklengths,
                    MPI_Aint *array_of_displacements,
                    MPI_Datatype array_of_types , MPI_Datatype * newtype)
```

**F77 ：**

```
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
+               ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES,
+               NEWTYPE, IERR)
INTEGER COUNT, NEWTYPE, IERR
INTEGER ARRAY_OF_BLOCKLENGTHS(*)
INTEGER ARRAY_OF_TYPES(*), ARRAY_OF_DISPLACEMENTS(*)
```

该函数是最一般的新数据类型的构造函数，也是使用最广泛的一个，正确使用此函数在实际应用中非常重要。生成的新数据类型由count个数据块构成，其中第i个数据块包含array_of_bloklengths(i)个连续存放、类型为array_of_types(i) 的数据，其字节位移是array_of_displacements(i)。

#### 9.2.5 地址函数

MPI_ADDRESS(location, address)

**参数：** IN location 变量在内存中的地址

OUT address 相对于位置MPI_BOTTOM的偏移

**C ：**

int MPI_Address(void * location, MPI_Aint * address)

**F77 ：**

MPI_ADDRESS(LOCATION, ADDRESS, IERR)

<type> LOCATION(*)

INTEGER ADDRESS, IERR

该函数返回给定变量在内存中相对于预定义的地址MPI_BOTTOM ( MPI系统定义的一个消息缓冲区常数)的偏移地址，主要用在Fortran 77中计算变量的位移量。C 语言中可以通过指针实现对地址的任何操作。

### 9.3 数据类型的使用

所有派生数据类型在首次使用时必须递交给 MPI 系统，而当一个派生数据类型不再需要时，应及时将其释放，以释放它所占用的系统资源。

#### 9.3.1 数据类型的递交

MPI_TYPE_COMMIT(datatype)

**参数：** INOUT datatype 递交的数据类型

**C ：**

int MPI_Type_commit(MPI_Datatype * datatype)

**F77 ：**

MPI_TYPE_COMMIT(DATATYPE, IERR)

INTEGER DATATYPE, IERR

一个新的数据类型在被递交后就可以和 MPI 的原始数据类型一样在消息的收发中使用。如果一个数据类型仅仅用于创建其它数据类型，而不直接在消息传递中使用，则不必将它递交，一旦所有基于它的新数据类型创建完毕，即可立即将其释放。

#### 9.3.2 数据类型的释放

MPI_TYPE_FREE(datatype)

**参数：** INOUT datatype 释放的数据类型(句柄)

**C ：**

int MPI_Type_free(MPI_Datatype * datatype)

**F77 ：**

MPI_TYPE_FREE(DATATYPE, IERR)

INTEGER DATATYPE, IERR

该函数将释放指定的数据类型，函数返回后，datatype 被置成 MPI_DATATYPE_NULL。正在进行的使用该数据类型的通信将会正常完成。一个数据类型的释放，并不影响在它的基础上创建的其它数据类型。

### 9.4 数据类型的查询函数

#### 9.4.1 跨度的查询

MPI_TYPE_EXTENT(datatype, extent)

**参数：** IN datatype 数据类型
OUT extent 数据类型的extent

**C ：**

int MPI_Type_extent(MPI_Datatype datatype, int * extent)

**F77 ：**

MPI_TYPE_EXTENT(DATATYPE, SIZE, ERR)
INTEGER DATATYPE, EXTENT, IERR

该函数以字节为单位返回指定数据类型的跨度extent。

### 9.4.2 大小的查询

MPI_TYPE_SIZE(datatype, size)

**参数：** IN datatype 数据类型
OUT size 数据类型的大小

**C ：**

int MPI_Type_size(MPI_Datatype datatype, int * size)

**F77 ：**

MPI_TYPE_SIZE(DATATYPE, SIZE, IERR)
INTEGER DATATYPE, SIZE, IERR

该函数以字节为单位返回指定数据类型的有用部分的大小，即跨度减去类型中的空隙。

### 9.4.3 上界的查询

MPI_TYPE_UB(datatype, displacement)

**参数：** IN datatype 数据类型
OUT displacement 上界的偏移

**C ：**

int MPI_Type_ub(MPI_Datatype datatype, int * displacement)

**F77 ：**

MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERR)
INTEGER DATATYPE, DISPLACEMENT, IERR

该函数以字节为单位返回指定数据类型的上界。

### 9.4.4 下界的查询

MPI_TYPE_LB(datatype, displacement)

**参数：** IN datatype 数据类型
OUT displacement 下界的偏移

**C ：**

int MPI_Type_lb (MPI_Datatype datatype, int * displacement)

**F77 ：**

MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERR)
INTEGER DATATYPE, DISPLACEMENT, IERR

该函数以字节为单位返回指定数据类型的下界。

## 9.5 其它与数据类型有关的函数

MPI 系统提供了两个查询接收消息长度的函数。

### 9.5.1 MPI_GET_COUNT

MPI_GET_COUNT(status, datatype, count)

**参数：**

| | | | |
|---|---|---|---|
| | IN | status | 接收操作的返回状态 |
| | IN | datatype | 接收操作使用的数据类型 |
| | OUT | count | 接收到的以指定的数据类型为单位的数据个数 |

**C ：**

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

**F77 ：**

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

该函数在 count 中返回实际接收到的消息的长度（以数据类型 datatype 为单位）

### 9.5.2 MPI_GET_ELEMENTS

MPI_GET_ELEMENTS（status, datatype, count ）

**参数：**

| | | | |
|---|---|---|---|
| | IN | status | 接收操作的返回状态 |
| | IN | datatype | 接收操作使用的数据类型 |
| | OUT | count | 接收到的基本元素个数 |

**C ：**

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

**F77 ：**

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

该函数返回的是消息中所包含的 MPI 原始数据类型的个数。

### 9.5.3 几点注意

1．MPI_SEND(buf, count, datatype, dest, tag, comm)
等价于与
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm)

2．下面的 SEND 与 RECV 都匹配，即它们可以任意匹配

```
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

## 9.6 数据的打包与拆包

在 MPI 中，通过使用特殊的数据类型 MPI_PACKED，用户可以将不同的数据进行打包，然后再一次发送出去，接收方在收到消息后再进行拆包。

### 9.6.1 数据打包

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

**参数：**

| | | |
|---|---|---|
| IN | inbuf | 输入缓冲区的起始地址 |
| IN | incount | 输入数据的个数 |
| IN | datatype | 输入数据的类型 |
| OUT | outbuf | 打包缓冲区的起始地址 |
| IN | outsize | 打包缓冲区的大小（字节数） |
| INOUT | position | 打包缓冲区的当前位置（字节数） |
| IN | comm | 通信器 |

**C ：**

```
int  MPI_Pack(void * inbuf, int incount, MPI_datatype datatype,
              void * outbuf, int outsize, int * position, MPI_Comm comm)
```

**F77 ：**

```
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
+        POSITION, COMM, IERR)
<type>  INBUF(*), OUTBUF(*)
INTEGER  INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERR
```

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），comm 是发送打包数据时将使用的通信器。Position 是打包缓冲区中的位移，每次打包第一次调用 MPI_PACK 时用户应该将其置为 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 实际上就是已打包数据的总长度。通过连续几次对不同位置的数据进行打包，就可以将不连续的数据放到一个连续的空间中。

### 9.6.2 数据拆包

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

**参数：**

| | | |
|---|---|---|
| IN | inbuf | 打包缓冲区的起始地址 |
| IN | insize | 拆包缓冲区的大小（字节数） |
| INOUT | position | 打包缓冲区的当前位置（字节数） |
| OUT | outbuf | 输出缓冲区的起始地址 |
| IN | outcount | 输出数据的个数 |
| IN | datatype | 输入数据的类型 |
| IN | comm | 通信器 |

**C ：**

```
int  MPI_Unpack(void * inbuf, int insize, int * position, void * outbuf,
                int outcount, MPI_datatype datatype, MPI_Comm comm)
```

**F77 ：**

```
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
+          DATATYPE, COMM, IERR)
<type>  INBUF(*), OUTBUF(*)
INTEGER  INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERR
```

该函数进行数据拆包操作，是 MPI_PACK 的逆操作：它从 inbuf 中拆包 outcount 个类型为 datatype 的数据到 outbuf 中。函数的各项参数与 MPI_PACK 类似，只不过这里的 inbuf 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。

# 附录 A 常见编程错误

- 缺少IERR参数：在FORTRAN源程序中MPI子程序的最后一个参数Ierr用于返回错误代码，而C形式的调用中却没有这一参数，因此该参数经常被漏掉，同时由于有些FORTRAN编译器检查得严，有些FORTRAN编译器并不检查参数是否严格匹配，因此在编译阶段不被发现导致在运行时出现一些莫名其妙的错误。
- 对status的错误声明：status是一个整数数组，而不是一个整数。在MPI_RECV调用中一些返回信息保存在status中，由于一些编译器对它的检查不严格，这样在程序运行时便经常出现写变量出界而产生不可预见的错误。
- 以MPI_开头的变量：这样很容易和MPI自身的调用名字或常量名字相同，因而造成混淆故应避免声明以MPI_开始的变量或常量。
- argcargv参数的使用：为了程序的通用性和移植性，最好不要在MPI程序中对argc和argv进行引用。在C程序的MPI调用中可以将argc和argv传递给MPI_INIT，这样就可以把argc和argv的值传递给所有的进程。但是MPI标准却并不要求一定要实现这一要求，因此对于一些MPI实现可以不将argc和argv传递给所有的进程，这样如果一个MPI程序依赖argc或argv这一参数特征，则在一定的条件下就会出错。
- 不要在MPI_INIT前和MPI_FINALIZE后写可执行程序代码：在MPI程序中在MPI_Init前和MPI_Finalize后的可执行程序如何执行MPI标准是没有定义的，因此这些位置的程序代码的执行会出现不可预料的结果。
- MPI_SEND和MPI_RECV的不合理次序：如果两个进程同时互相发送消息，都是先发送后接收，则很容易造成死锁。因为当系统内存空间缺乏时，进程1和进程2都无法将数据发送完成，因而也无法从对方接收数据。避免这种情况常采取的措施是：1） 将发送和接收重叠起来，即当一方在发送时另一方处于接收状态；2） 对于成对的交互发送和接收，鼓励使用MPI_SENDRECV，可能可以既提高效率又避免死锁问题；3） 鼓励使用非阻塞操作MPI_ISEND和MPI_IRECV来代替相应的阻塞操作。
- 数据类型不匹配：发送数据类型、发送函数中指定的数据类型和接收函数中指定的数据类型一致。
- 接收缓冲区溢出：接收缓冲区太小，但允许接收的数据容量却超过了接收缓冲区的大小。
- 行优先与列优先：C语言数组在内存中是以行存放，因此一行的数据是连续的，但同一列的数据是不连续的；FORTRAN数组在内存中以列存放，因此一列的数据是连续的，但同一行的数据是不连续的。