北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

并行编程原理与实践

7. MPI编程

王一拙、计卫星

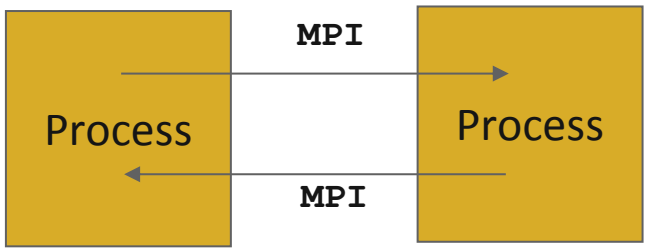
北京理工大学计算机学院

目录 | CONTENTS

1 MPI简介

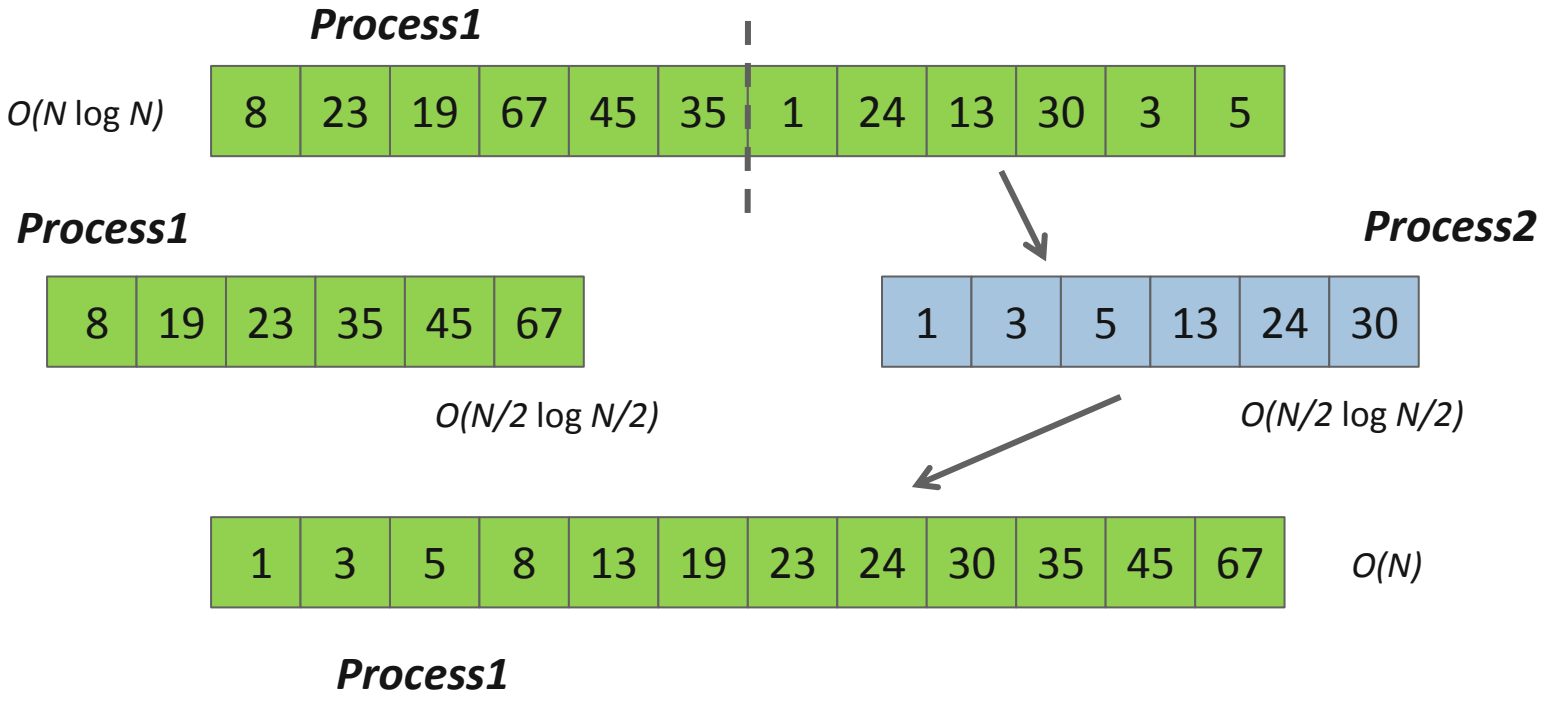
消息传递编程模型

- 进程有自己独立的地址空间，进程间通过MPI（Message Passing Interface）进行通信
- 进程间通信包括：
  - 同步
  - 数据通信（数据从一个进程的地址空间传递到另一进程地址空间）



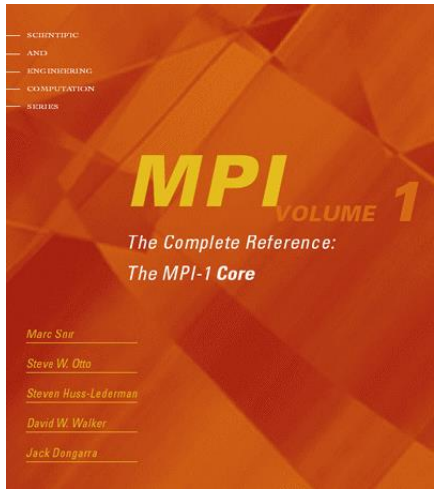
消息传递编程模型

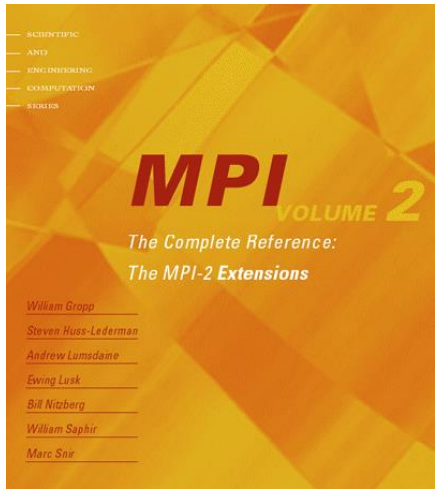
- 进程间的通信通过发送和接收消息来实现
- 消息是对数据的封装
- 示例：并行排序

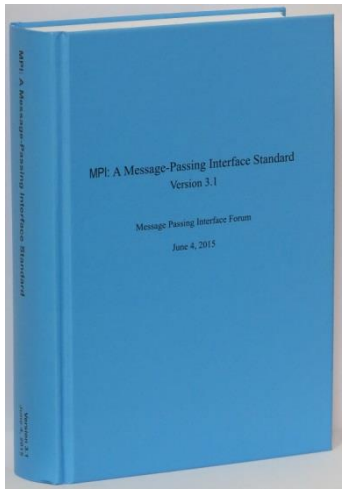


什么是MPI（Message Passing Interface）？

- 是函数库规范，而不是一种编程语言；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，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和事实上的标准







MPI简介

- **MPI 的发展历史**
  - MPI-1：1994年
    - 支持经典的消息传递编程（点对点通信、集合通信等）
    - MPICH：是MPI最流行的开源实现，由Argonne国家实验室和密西西比州立大学联合开发
  - MPI-2：1997年
    - 动态进程管理，并行I/O，远程存储访问，支持F90和C++
  - MPI-3：2012年
  - MPI-4草案：2020年
- **MPI 标准网站：**www.mpi-forum.org

为什么要用MPI？

- 标准化：MPI已成为分布式存储系统并行编程的标准，几乎所有HPC平台都支持MPI，它取代了所有以前的消息传递库
- 可移植性：应用程序在支持MPI的平台间移植时，无需修改源代码
- 性能优化：各厂商对MPI标准的实现，可以利用各自硬件特性进行性能优化
- 功能：MPI标准定义了丰富的功能
- 可用性：MPI标准有多种开源和商业软件实现
  - 开源实现主要有MPICH、LAMMPI（Open MPI）
  - 工业界在MPICH的基础上有很多MPI标准实现
    - Intel MPI, IBM Blue Gene MPI, Cray MPI, Microsoft MPI, MVAPICH, MPICH-MX

并行计算带来的挑战

- 新的算法设计、算法分析技术
- 新的编程语言、模型、开发和调试工具
- 操作系统、底层服务和I/O接口对并行的支持
- 新的硬件结构
- 程序员需要了解存储系统和互连网络的结构等

2 基本MPI编程

- 2.1 MPI基本函数
- 2.2 点对点通信
- 2.3 MPI程序的编译、运行和调试

“Hello World”示例

Hello world(C)

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

main(
   int argc,
   char *argv[] )
{
     MPI_Init( &argc, &argv );
     printf( "Hello, world!\n" );
     MPI_Finalize();
}
```

Hello world(Fortran)

```fortran
program main
include 'mpif.h'
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

2.1 MPI基本函数

C和Fortran中MPI函数约定

- C
  - 必须包含mpi.h
  - MPI 函数返回出错代码或 MPI_SUCCESS成功标志
  - MPI_前缀，且只有MPI以及MPI_标志后的第一个字母大写，其余小写
- Fortran
  - 必须包含mpif.h
  - 通过子函数形式调用MPI，函数最后一个参数为返回值
  - MPI_前缀，且函数名全部为大写
- MPI函数的参数被标志为以下三种类型：
  - IN：参数是输入的，在例程的调用中不会被修正
  - OUT：参数是输出的，在例程的调用中可能会被修正
  - INOUT：参数同时用于两个方向的数据传递

2.1 MPI基本函数

- **MPI初始化：`int MPI_Init(int *argc, char **argv)`**
  - 是MPI程序的第一个调用，它完成MPI程序的所有初始化工作。所有的MPI程序的第一条可执行语句都是这条语句
  - 启动MPI环境，标志并行代码的开始
- **MPI结束：`int MPI_Finalize(void)`**
  - 是MPI程序的最后一个调用，它结束MPI程序的运行，它是MPI程序的最后一条可执行语句，否则程序的运行结果是不可预知的
  - 标志并行代码的结束，结束除主进程外其它进程
  - 之后串行代码仍可在主进程(rank = 0)上运行(如果必须)

编译运行“Hello World”示例

- MPI作为函数库存在，因此对MPI程序的编译只是调用本机编译器并加上相关设置，这些细节被封装成mpicc可执行脚本命令
- 编译：
  - 普通程序：`gcc hello.c -o hello`
  - MPI程序：`mpicc hello.c -o hello`
- 执行：
  - 普通程序：`./hello`
  - MPI程序：`mpiexec -n 16 ./hello`

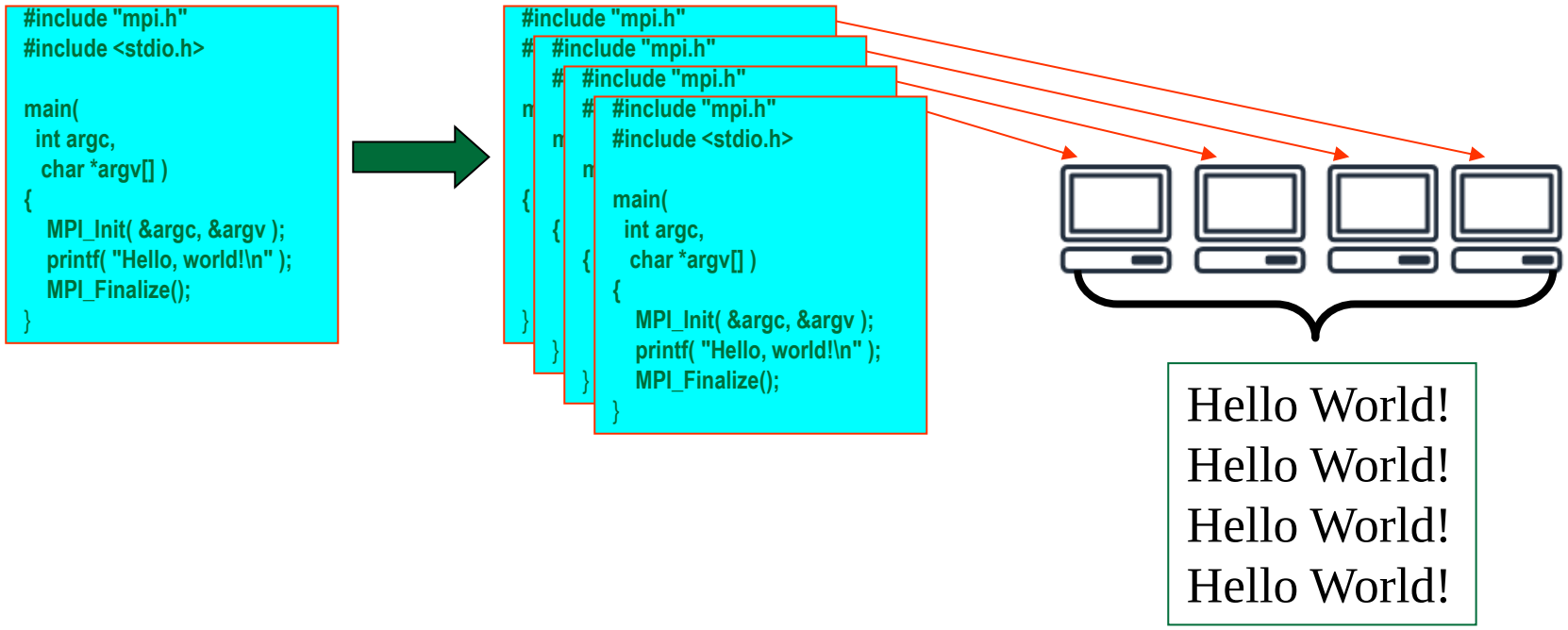
编译运行“Hello World”示例

- MPI作为函数库存在，因此对MPI程序的编译只是调用本机编译器并加上相关设置，这些细节被封装成mpicc可执行脚本命令
- 编译：
  - 普通程序：`gcc hello.c -o hello`
  - MPI程序：`mpicc hello.c -o hello`
- 执行：
  - 普通程序：`./hello`
  - MPI程序：`mpiexec -n 16 ./hello`

mpicc -show

指定进程数量

- **Hello是如何被执行的？**
  - SPMD: Single Program Multiple Data(SPMD)



改进“Hello World”示例

- 任务由多少个进程来进行并行计算？
- 我是哪一个进程?

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

main( int argc, char *argv[] )
{
    int  myid, numprocs;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myid );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    printf("I am %d of %d\n", myid, numprocs );
    MPI_Finalize();
}
```

2.1 MPI基本函数

进程组（process group）

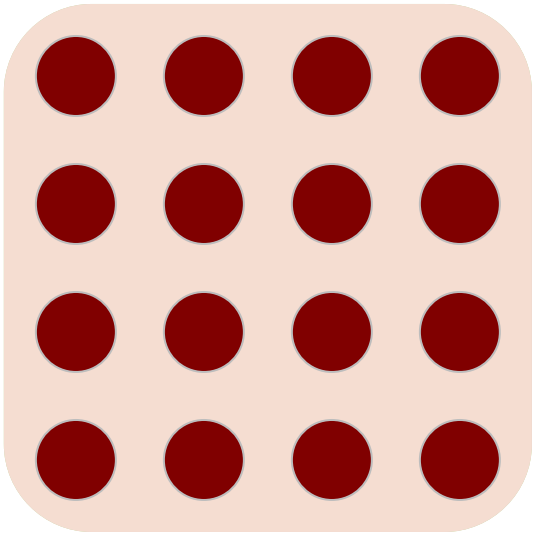
- 是全部MPI进程的有限、有序子集。进程组中每个进程被赋于一个在该组中唯一的序号(rank)，用于在该组中标识该进程。序号的取值范围是[0,进程数-1]

通信域（communicator）

- 通信域包括进程组和通信上下文等内容，用于描述通信进程间的通信关系
- 通信域分为组内通信域和组间通信域，分别用来实现MPI的组内通信和组间通信，多数 MPI用户只需进行组内通信
- MPI中通信上下文如同系统设计的超级标签，用于安全地区别不同的通信，以免相互干扰
- MPI包括几个预定义的通信域。 例如，MPI_COMM_WORLD是所有MPI进程的集合，在执行了MPI_Init函数之后自动产生，MPI_COMM_SELF是每个进程独自构成的、仅包含自己的通信域
- 任何MPI通信函数均必须在某个通信域内发生

通信域

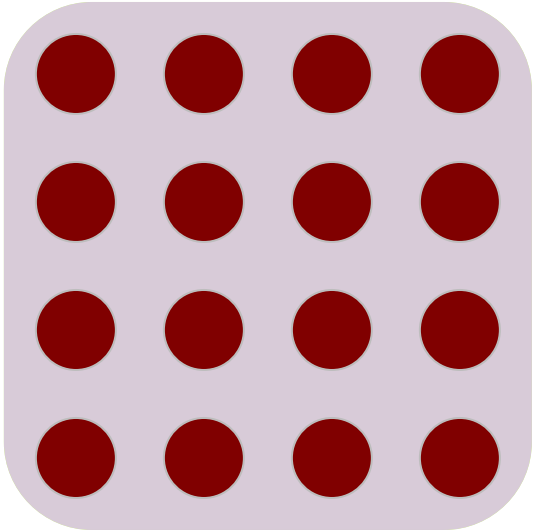
```
mpiexec  -n  16  ./test
```



启动MPI程序时自动创建
预定义通信域
MPI_COMM_WORLD

通信域

```
mpiexec  -n  16  ./test
```

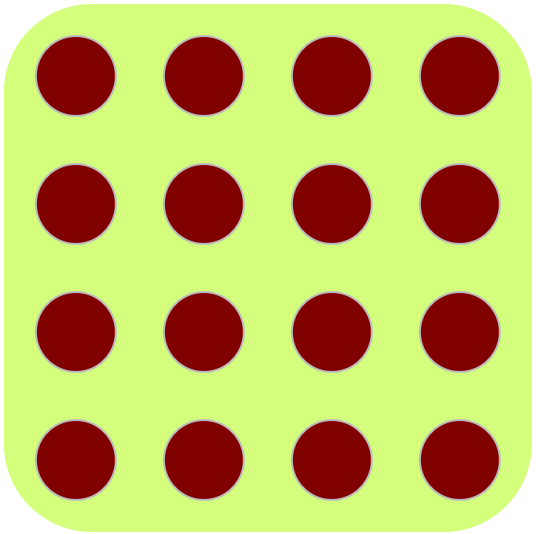


启动MPI程序时自动创建
预定义通信域
MPI_COMM_WORLD

可以对通信域进行复制

通信域

```
mpiexec  -n  16  ./test
```



启动MPI程序时自动创建
预定义通信域
MPI_COMM_WORLD

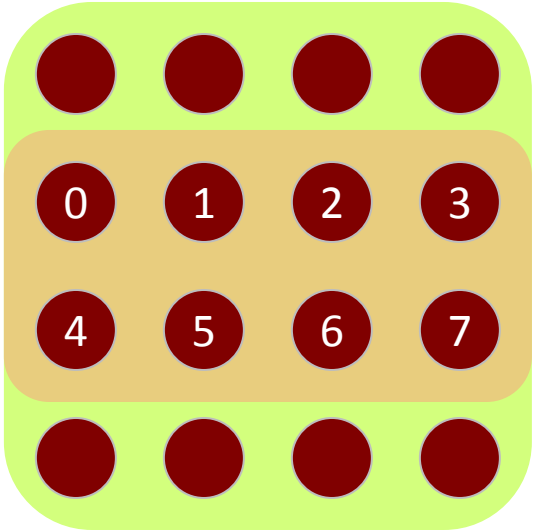
可以对通信域进行复制

通信域

```
mpiexec  -n  16  ./test
```

通信域不需要包含系统中的所有进程

通信域中的每个进程有一个序号（rank）



启动MPI程序时自动创建预定义通信域
MPI_COMM_WORLD

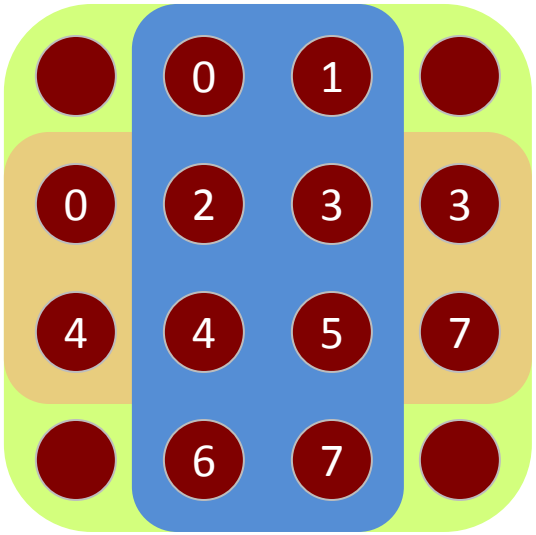
可以对通信域进行复制

通信域

```
mpiexec  -n  16  ./test
```

通信域不需要包含系统中的所有进程

通信域中的每个进程有一个序号（rank）



同一个进程在不同通信域里可能具有不同序号（rank）

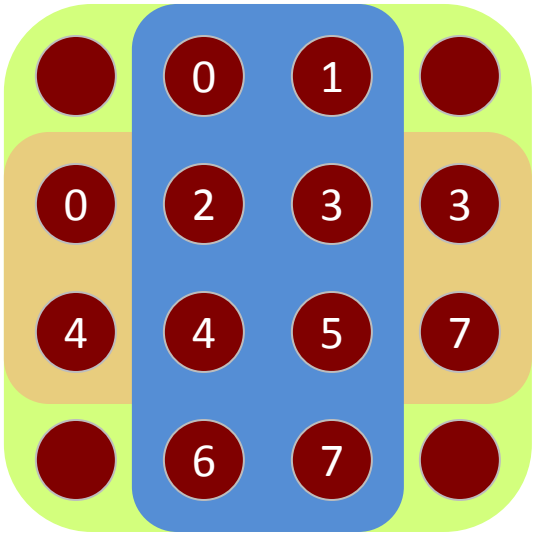
启动MPI程序时自动创建预定义通信域
MPI_COMM_WORLD

可以对通信域进行复制

通信域

```
mpiexec  -n  16  ./test
```

通信域不需要包含系统中的所有进程



启动MPI程序时自动创建预定义通信域 `MPI_COMM_WORLD`

通信域中的每个进程有一个序号（rank）

可以对通信域进行复制

同一个进程在不同通信域里可能具有不同序号（rank）

- 通信域可以在程序中手动创建或借助MPI工具软件创建
- 简单应用程序一般只用默认通信域`MPI_COMM_WORLD`即可

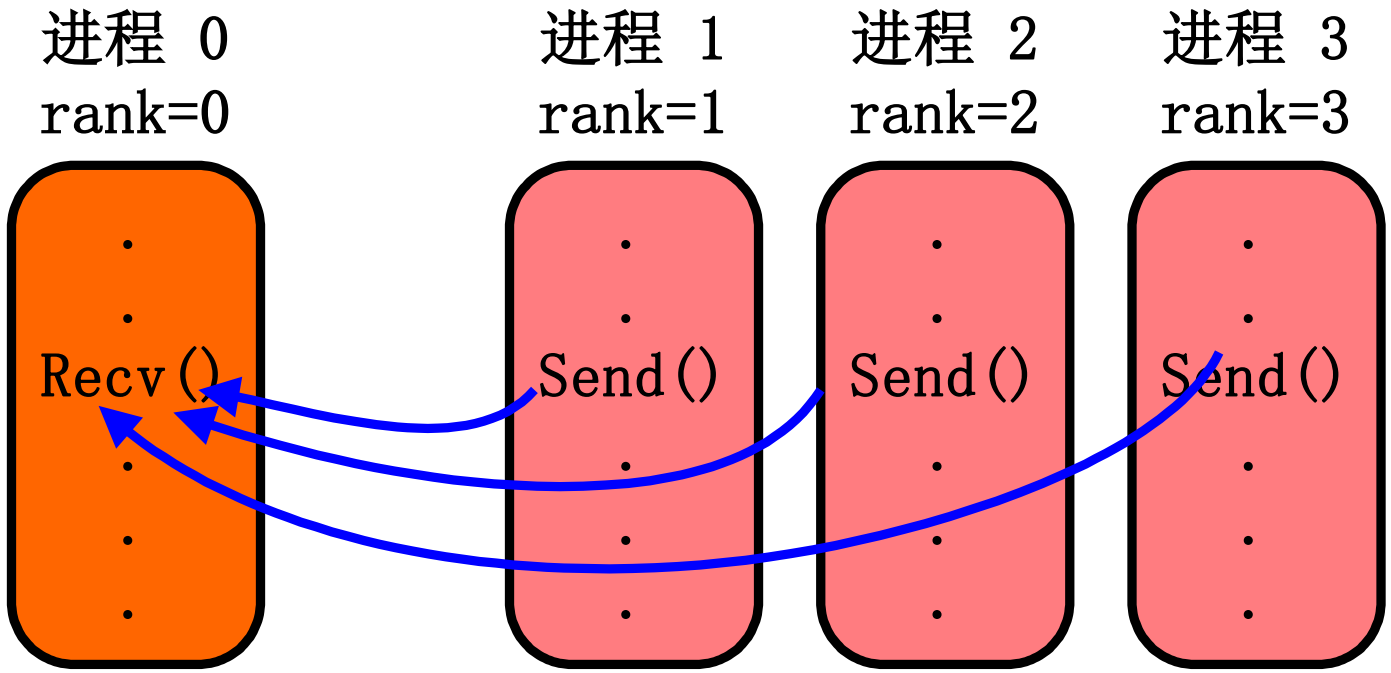
- 获取指定通信域的进程数：

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- 获取进程编号：

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

- 消息传递“Greetings”示例



消息传递“Greetings”示例

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[])
{
  int numprocs, myid, source;
  MPI_Status status;
  char message[100];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,
            &myid);
  MPI_Comm_size(MPI_COMM_WORLD,
            &numprocs);
  if (myid != 0) {
    strcpy(message, "Hello World!");
    MPI_Send(message,strlen(message)+1,
       MPI_CHAR, 0,99, MPI_COMM_WORLD);
  }
  else {/* myid == 0 */
    for(source=1; source<numprocs; source++){
      MPI_Recv(message, 100, MPI_CHAR, source,
                99, MPI_COMM_WORLD,  &status);
      printf("%s\n", message);
    }
  }
  MPI_Finalize();
} /* end main */
```

发送消息

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

- 一个进程将数据发送到另一个进程（或一组进程）
- 发送消息需要以下信息：
  - 发送的数据是什么？
    - buf为消息的地址，count是消息元素的数量，datatype为消息元素的数据类型
  - 发送给谁？
    - 通信域comm中编号为dest的进程
  - 消息的用户自定义标签（tag）

```
int → MPI_INT
double → MPI_DOUBLE
char → MPI_CHAR
```

为什么需要消息标签？

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

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

- Client/server模式中，服务进程可通过消息标签区分客户进程

为什么需要消息标签？

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

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

- Client/server模式中，服务进程可通过消息标签区分客户进程

Process P:	Process Q:
Send(A, 32, Q, tag1)	Recv (X, 32, P , tag1)
Send(B, 16, Q,tag2)	Recv (Y, 16, P , tag2)

为什么需要消息标签？

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

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

- Client/server模式中，服务进程可通过消息标签区分客户进程

Process P:	Process Q:
Send(A, 32, Q, tag1)	Recv (X, 32, P , tag1)
Send(B, 16, Q,tag2)	Recv (Y, 16, P , tag2)

```
Process P:
Send(Request1, 32, Q, tag1)

Process R:
Send(Request2, 32, Q, tag2)
```

```
Process Q:
while(true){
    Recv(request, 32, Any_process, Any_tag, status);
    if(status.Tag == tag1) Process request in one way
    if(status.Tag == tag1) Process request in another way
}
```

接收消息

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

- 接收方需要以下信息：
  - 接收的数据类型(datatype)、大小(count)和保存位置(buf)
    - 允许接收消息元素数量少于count，多于时会报错
  - 接收谁发送的消息？（通信域comm中编号为source的进程）
    - 如果接收任意进程发送的消息，令source=MPI_ANY_SOURCE
  - 消息的用户自定义标签（tag）
    - 如果接收任意标签的消息，令tag=MPI_ANY_TAG
- status是返回状态，包含实际接收的消息元素数量等更多信息

消息状态

- 消息状态(MPI_Status类型)存放接收消息的状态信息，包括：
  - MPI_SOURCE：发送消息的进程编号(rank)
  - MPI_TAG：实际接收的消息的标签
  - MPI_ERROR：错误状态
- 是消息接收函数MPI_Recv的最后一个参数
- 如不需要任何信息，可使用MPI_STATUS_IGNORE
- 实际接收到的消息元素数量通过以下函数获得：

```
typedef struct
_MPI_Status {
    int count;
    int cancelled;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status,
*PMPI_Status;
```

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

6个最基本的MPI函数：

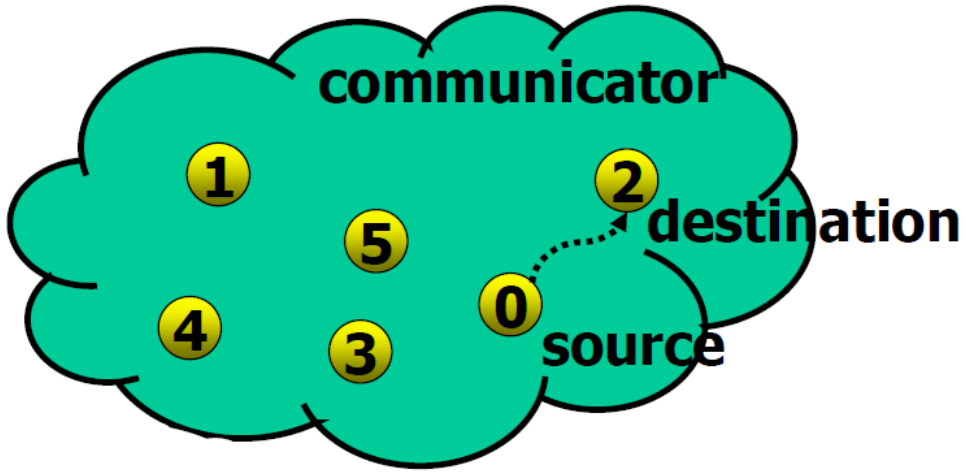
- `MPI_Init(…);`
- `MPI_Comm_size(…);`
- `MPI_Comm_rank(…);`
- `MPI_Send(…);`
- `MPI_Recv(…);`
- `MPI_Finalize();`

```
MPI_Init(…);
…
并行代码;
…
MPI_Fainalize();
只能有串行代码;
```

- 2.1 MPI基本函数
- 2.2 点对点通信
- 2.3 MPI程序的编译、运行和调试

什么是“点对点通信”（Point-to-Point Communication ）？

- 两个MPI进程之间的通信
- 源进程发送消息到目标进程
- 目标进程接收消息
- 通信发生在同一个通信域内



MPI通信函数术语

- Blocking（阻塞）：发送/接收函数调用等待操作完成才返回，返回后用户才可以重新使用调用中所占用的资源
- Non-blocking（非阻塞）：函数调用不必等待操作完成便可返回，但这并不意味着调用中所占用的资源可被重用
- Local（本地）：函数调用的完成不依赖于其它进程
- Non-local（非本地）：函数调用的完成依赖于其它进程，例如：消息发送进程等待接收进程完成接收才返回
- Collective（集合）：进程组里的所有进程都参与通信

MPI点对点通信函数概况

- MPI的点对点通信同时提供了阻塞和非阻塞两种通信机制
- 同时也支持四种通信模式
  - 同步(synchronous)通信模式
  - 缓冲(buffered)通信模式
  - 标准(standard)通信模式
  - 就绪(ready)通信模式
- 通信模式(Communication Mode)指的是缓冲管理，以及发送方和接收方之间的同步方式
- 不同通信模式和不同通信机制的结合，便产生了丰富的点对点通信函数

MPI点对点通信函数概况

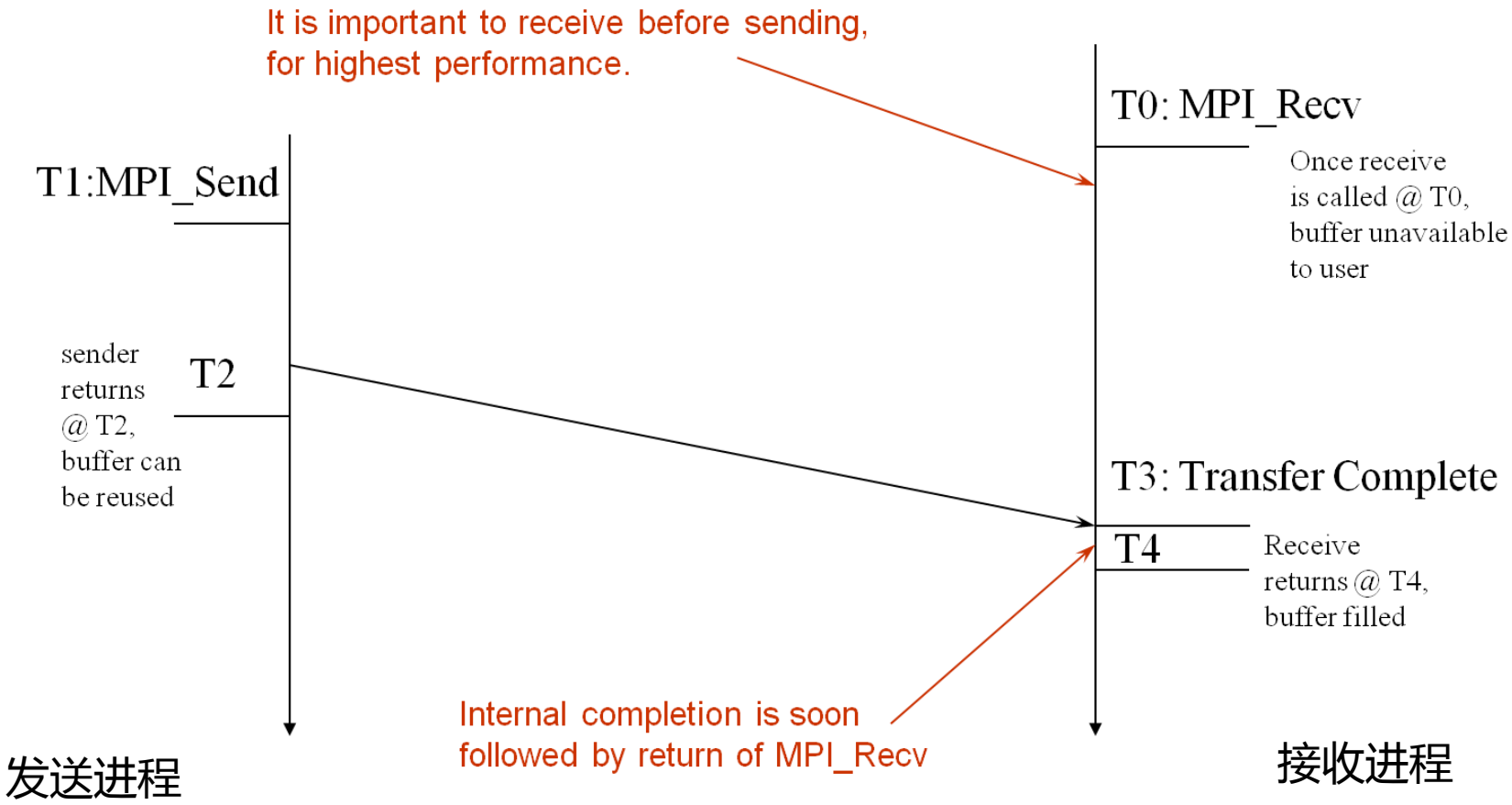
- MPI的发送操作支持四种通信模式，它们与阻塞属性一起产生了MPI中的8种发送操作
- 而MPI的接收操作只有两种：阻塞接收和非阻塞接收
- 非阻塞通信返回后并不意味着通信操作的完成，MPI还提供了对非阻塞通信完成的检测，主要的有两种：MPI_Wait和MPI_Test函数

通信模式	阻塞	非阻塞
同步	MPI_Ssend	MPI_Issend
缓冲	MPI_Bsend	MPI_Ibsend
就绪	MPI_Rsend	MPI_Irsend
标准	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv
	MPI_Sendrecv	
	MPI_Sendrecv_replace	

阻塞 vs. 非阻塞

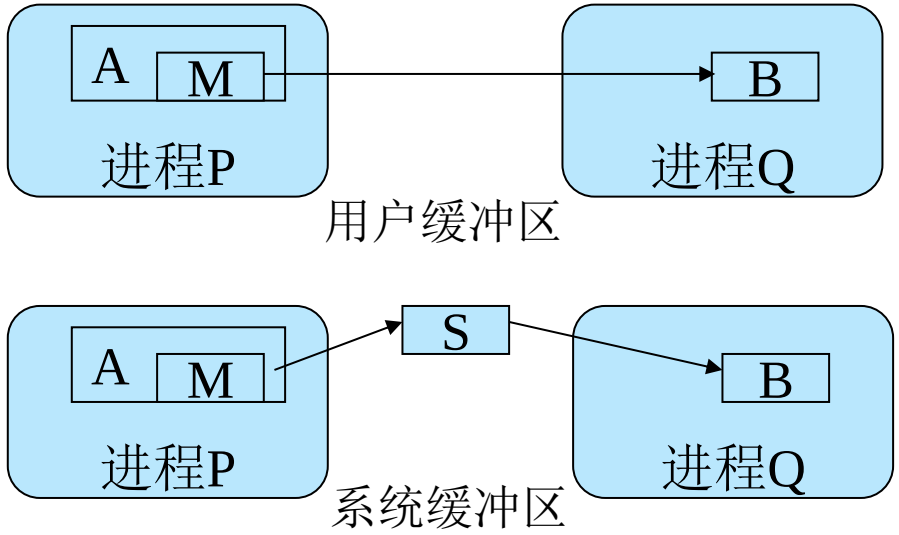
- 阻塞通信函数：**MPI_Send/MPI_Recv**
  - 进程将被阻塞，调用返回时，消息传递中所使用的内存位置可被重用
    - 对发送来说，就是发送缓冲区变量buf可被重复利用/修改，修改不会影响发送给接收者的数据
    - 对接收来说，消息已经接收到缓冲区变量buf，可以从其中读取数据了
  - 通信的确切完成语义取决于系统缓冲区设置和消息大小
  - 阻塞通信很容易使用，但容易出现死锁
- 非阻塞通信函数：**MPI_Isend/MPI_Irecv**
  - 调用立即返回，必须单独测试通信完成情况
  - 主要用于重叠计算和通信以提高性能

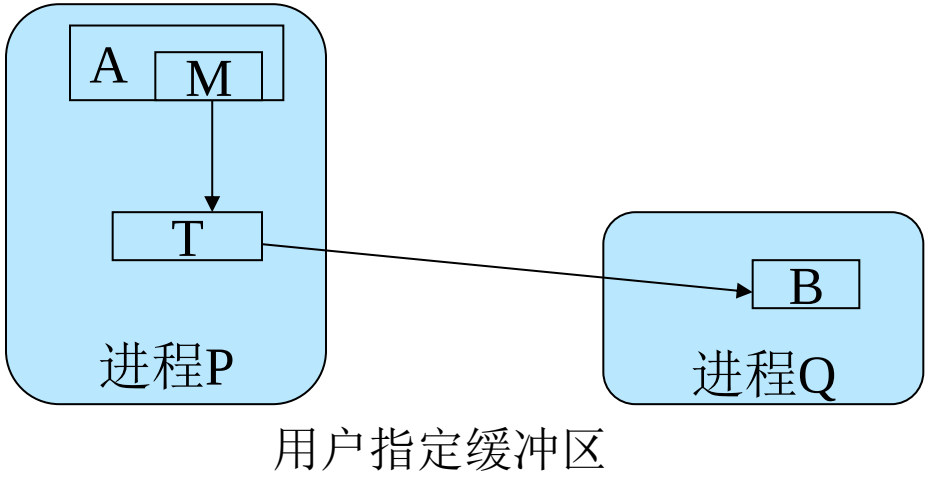
阻塞通信


It is important to receive before sending, for highest performance.
T0: MPI_Recv
Once receive is called @ T0, buffer unavailable to user
T1:MPI_Send
sender returns @ T2, buffer can be reused
T2
T3: Transfer Complete
T4
Receive returns @ T4, buffer filled
Internal completion is soon followed by return of MPI_Recv
发送进程
接收进程

MPI通信过程中的缓冲区是什么？

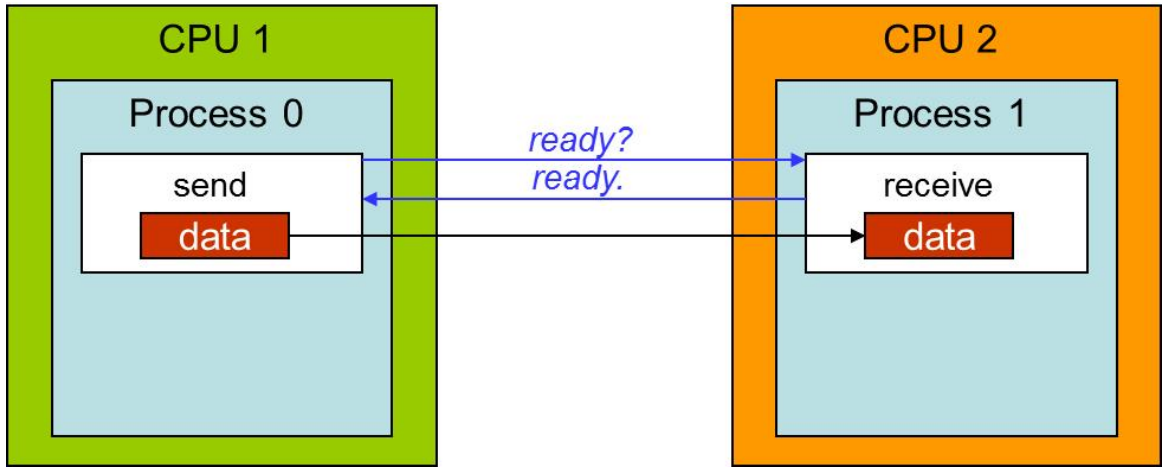
- 应用程序中声明的变量，在消息传递语句中用作缓冲区的起始位置
- 也可表示由系统(不同用户)创建和管理的某一存储区域，在消息传递过程中用于暂存放消息，也被称为系统缓冲区
- 用户可设置一定大小的存储区域，用作中间缓冲区以保留可能出现在其应用程序中的任意消息





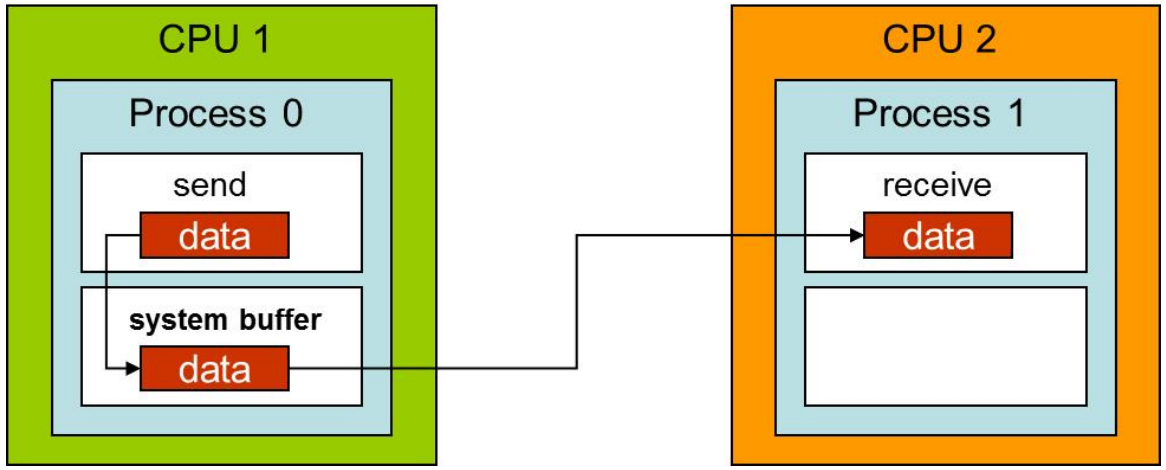
同步通信模式

- 只有相应的接收过程已经启动，发送过程才正确返回
- 同步发送调用正确返回后，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



缓冲通信模式

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



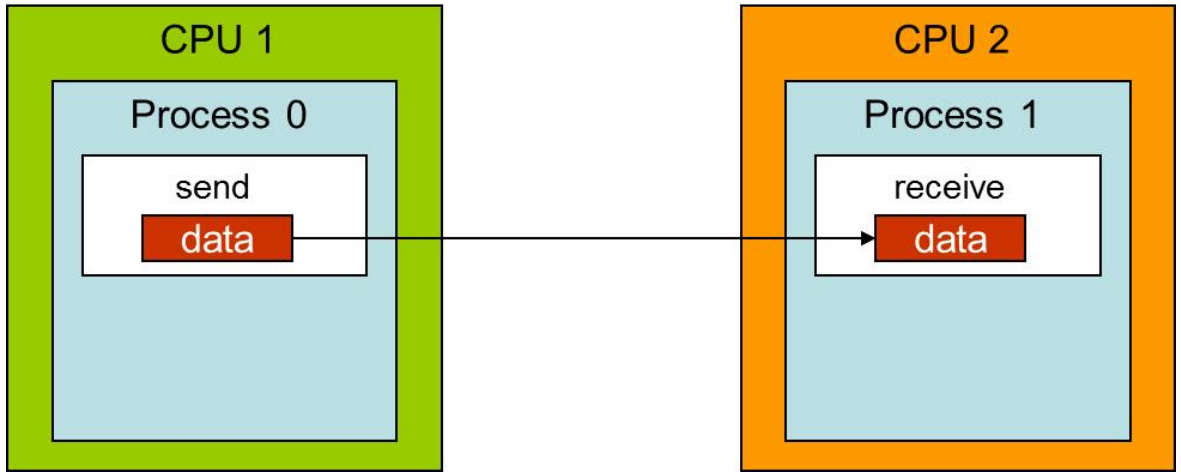
缓冲通信模式

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


```
char *buf = malloc(bufsize);
MPI_Buffer_attach( buf, bufsize );
MPI_Bsend(&outmsg, strlen(outmsg), MPI_CHAR, 1, 1, MPI_COMM_WORLD);
MPI_Buffer_detach( &buf, &bufsize );
```

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对应的接收操作还没有启动，发送操作将出错



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对应的接收操作还没有启动，发送操作将出错

```
if(rank==dest){ //接收进程
  //执行一个非阻塞的接收调用，立刻正确返回
  MPI_Irecv(buf,size,MPI_XX,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD);
  //向发送进程发送一个阻塞的发送调用。
  MPI_Send(buffer,size,MPI_XX,source,tag,MPI_COMM_WORLD);
}
else{  //发送进程
  //放在就绪发送调用之前，确保接受进程的接受调用启动后，再进行就绪发送。
  MPI_Recv(buffer,size,MPI_XX,dest,MPI_ANY_TAG,MPI_COMM_WORLD);
  //接收进程的阻塞发送结束后再开始执行。
  MPI_Rsend(buf,size,MPI_XX,dest,tag,MPI_COMM_WORLD,ieer);
}
```

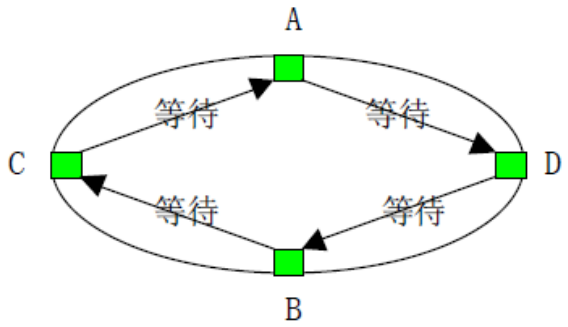
标准通信模式

- 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
  - 若缓存发送数据，则发送调用的正确返回不依赖于接收进行
  - 若不缓存发送数据，直接发送数据，则只有当相应的接收调用执行且数据完全到达接收缓冲区后，发送调用才正确返回
- 相当于“无模式”，具体采用哪种通信模式由实现决定
- OpenMPI实现对短消息采用缓冲模式，对长消息采用类似同步模式

编写安全的MPI程序

- 编写MPI程序如果通信调用的顺序使用的不当很容易造成死锁

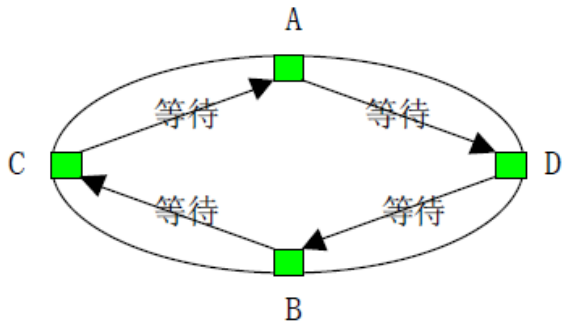
```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)     A
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)             C
ELSE IF( rank .EQ. 1)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)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



编写安全的MPI程序

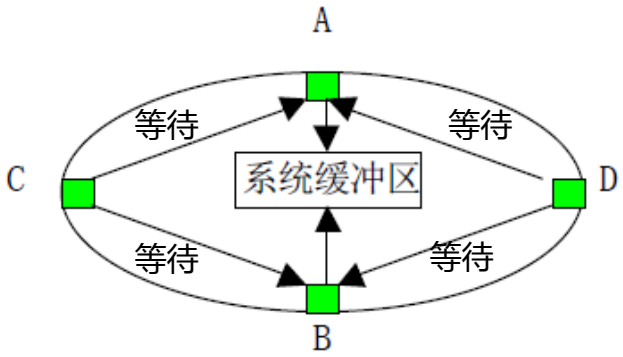
- 编写MPI程序如果通信调用的顺序使用的不当很容易造成死锁

```
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
```



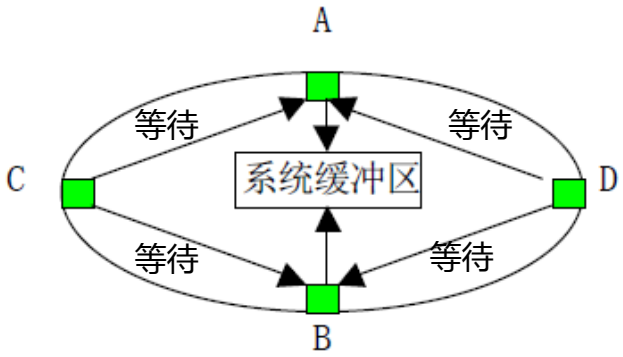
编写安全的MPI程序

```
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 IF( 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
```



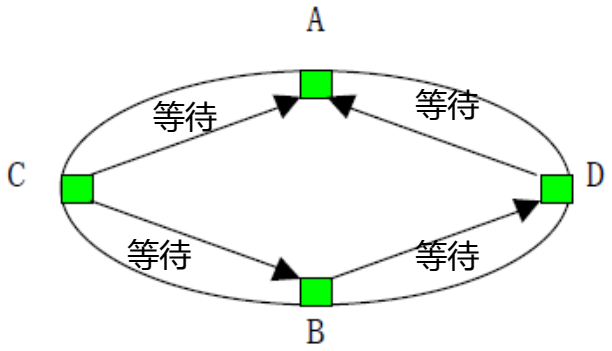
编写安全的MPI程序

```
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 IF( 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
```



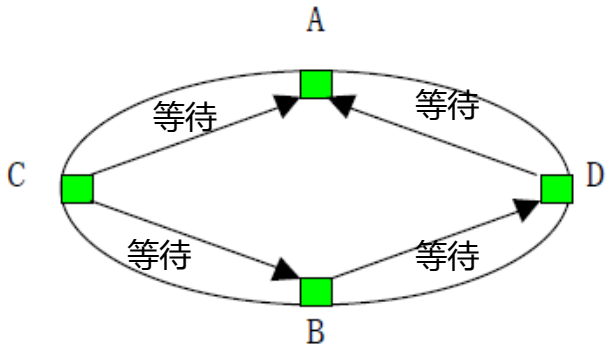
编写安全的MPI程序

```
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 IF( 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
```



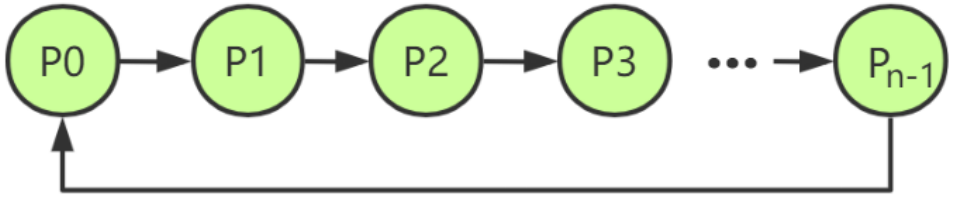
编写安全的MPI程序

```
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 IF( 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
```



捆绑发送和接收

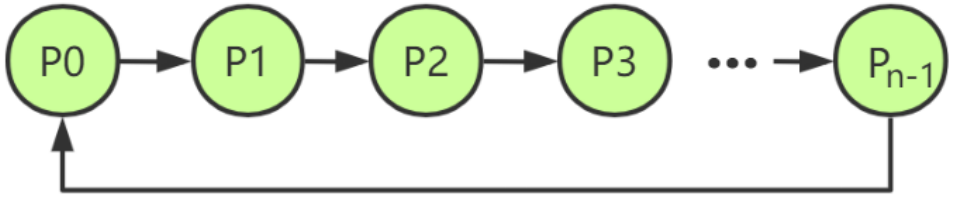
- 数据轮换




```
int MPI_Sendrecv(
    void *          sendbuf,    //发送缓冲区起始地址
    int             sendcount,  //发送数据的个数
    MPI_Datatype    sendtype,   //发送数据的数据类型
    int             dest,       //目标进程的标识号
    int             sendtag,    //发送消息标签
    void *          recvbuf,    //接收缓冲区初始地址
    int             recvcount,  //最大接收数据个数
    MPI_Datatype    recvtype,   //接收数据的数据类型
    int             source,     //源进程标识
    int             recvtag,    //接收消息标签
    MPI_Comm        comm, //通信域
    MPI_Status *    status      //返回的状态
)
```

捆绑发送和接收

- 数据轮换

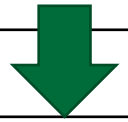



```
int MPI_Sendrecv(
    void *          sendbuf,    //发送缓冲区起始地址
    int             sendcount,  //发送数据的个数
    MPI_Datatype    sendtype,   //发送数据的数据类型
    int             dest,       //目标进程的标识号
    int             sendtag,    //发送消息标签
    void *          recvbuf,    //接收缓冲区初始地址
    int             recvcount,  //最大接收数据个数
    MPI_Datatype    recvtype,   //接收数据的数据类型
    int             source,     //源进程标识
    int             recvtag,    //接收消息标签
    MPI_Comm        comm, //通信域
    MPI_Status *    status      //返回的状态
)
```

对于成对的交互发送和接收，鼓励使用MPI_Sendrecv语句，因为该语句本身提供了优化的可能，既提高效率又避免编写单独的MPI_Send和MPI_Recv语句可能造成的死锁问题。

捆绑发送和接收

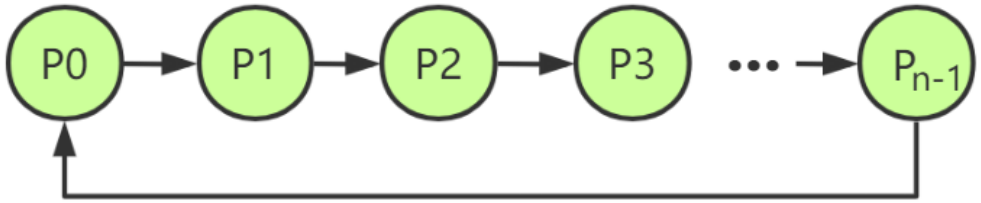
```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF( rank .EQ. 1)
  CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf,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



```
RECVCALL 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)
```

捆绑发送和接收

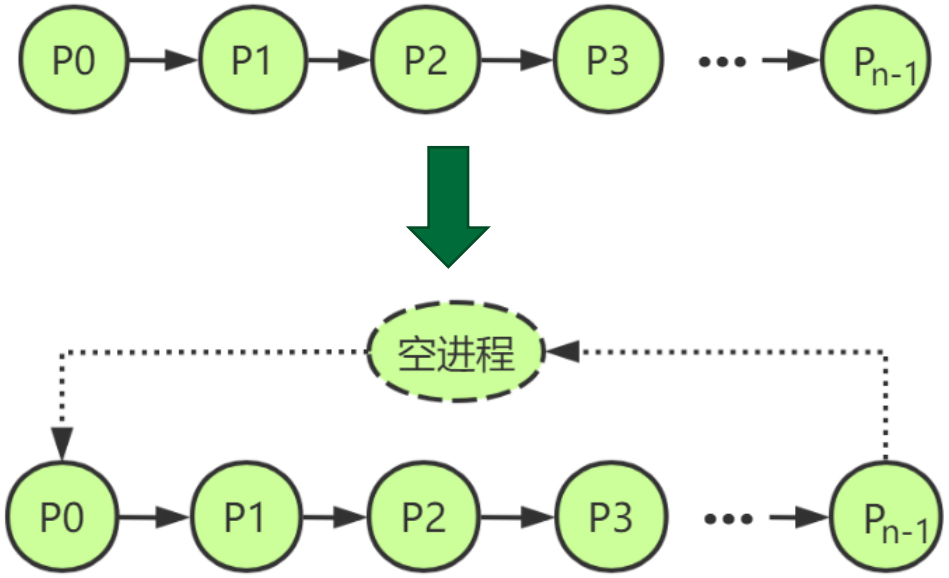
- 数据轮换示例




```
...
int a,b;
...
MPI_Status status;
int dest = (rank+1)%p;
int source = (rank + p -1)%p;   /*p为进程个数*/
MPI_Sendrecv( &a, 1, MPI_INT, dest, 99, &b 1, MPI_INT, source, 99,
                              MPI_COMM_WORLD, &status);
```

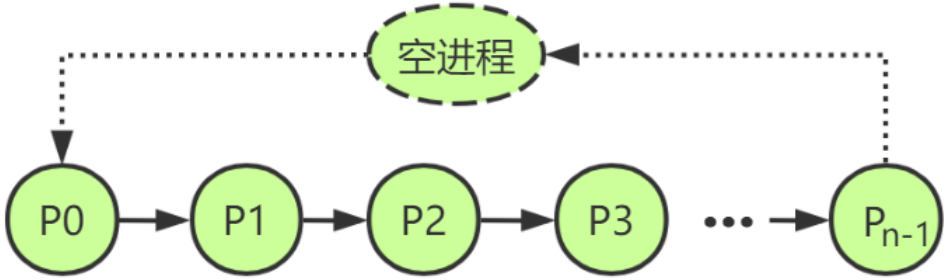
捆绑发送和接收

- 利用空进程
  - rank = MPI_PROC_NULL的进程称为空进程
  - 使用空进程的通信不做任何操作
  - 向MPI_PROC_NULL发送的操作总是成功并立即返回
  - 从MPI_PROC_NULL接收的操作总是成功并立即返回，且接收缓冲区内容为随机数



捆绑发送和接收

- 空进程示例




```
...
MPI_Status status;
int dest = (rank+1)%p;
int source = (rank + p -1)%p;
if(source == p-1) source = MPI_PROC_NULL;
if(dest == 0) dest = MPI_PROC_NULL;

MPI_Sendrecv( &a, 1, MPI_INT, dest, 99, &b 1, MPI_INT, source, 99,
                        MPI_COMM_WORLD, &status);
...
```

非阻塞通信

- 与阻塞通信的比较

通信类型	MPI函数	函数调用返回	对数据区操作	特性
阻塞通信	MPI_Send MPI_Recv	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
非阻塞通信	MPI_Isend MPI_Irecv	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

非阻塞发送

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

- 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对象，通过request变量返回
- request可供之后（查询和等待）函数使用

非阻塞接收

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

如何判断非阻塞通信的完成？

- 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
- 通过MPI_Wait()和MPI_Test()来判断通信是否已经完成

MPI_Wait的使用

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

- 以非阻塞通信请求对象request作为参数，一直等到相应的非阻塞通信完成后才成功返回，将相关信息放入status中，并释放这一非阻塞通信请求对象（request = MPI_REQUEST_NULL）

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

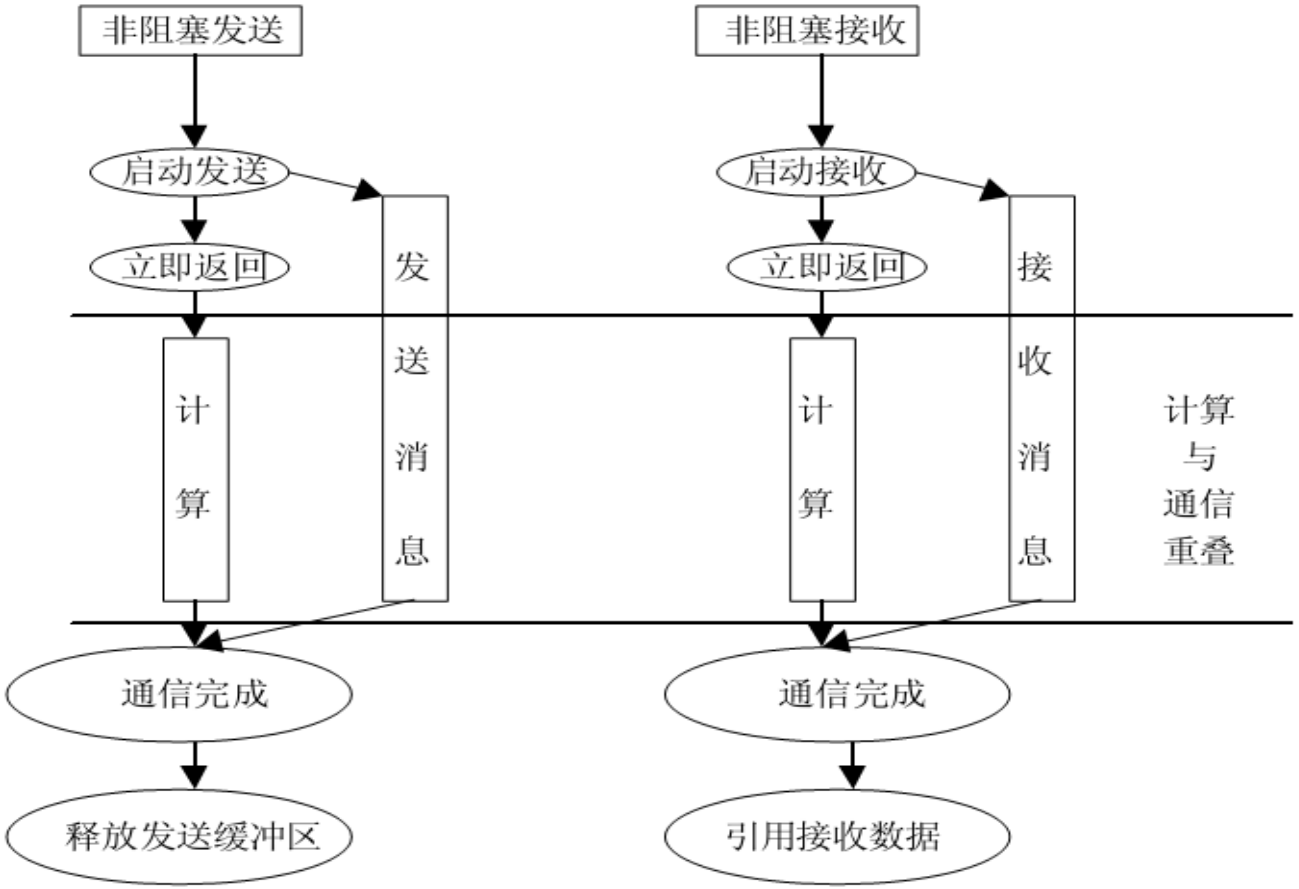
MPI_Test的使用

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

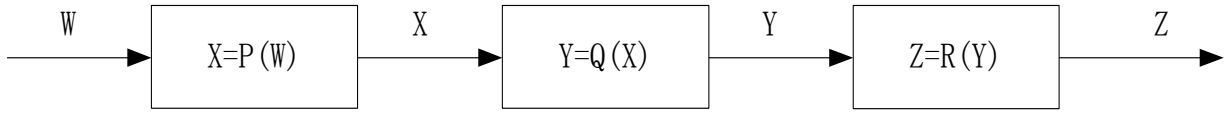
- MPI_Test在调用后会立刻返回，若相应非阻塞通信已完成，则置完成标志flag=true；反之，置完成标志flag=false

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

非阻塞通信


非阻塞发送
启动发送
立即返回
计算
发送消息
通信完成
释放发送缓冲区
非阻塞接收
启动接收
立即返回
计算
接收消息
通信完成
引用接收数据
计算与通信重叠

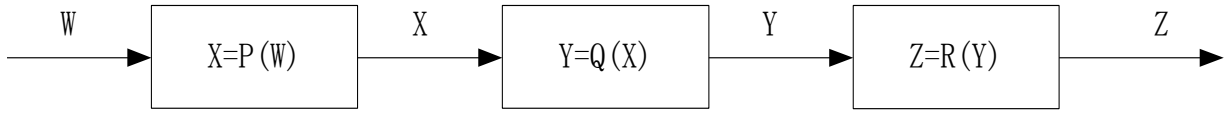
计算与通信重叠（Overlap）



- 通常利用双缓冲实现

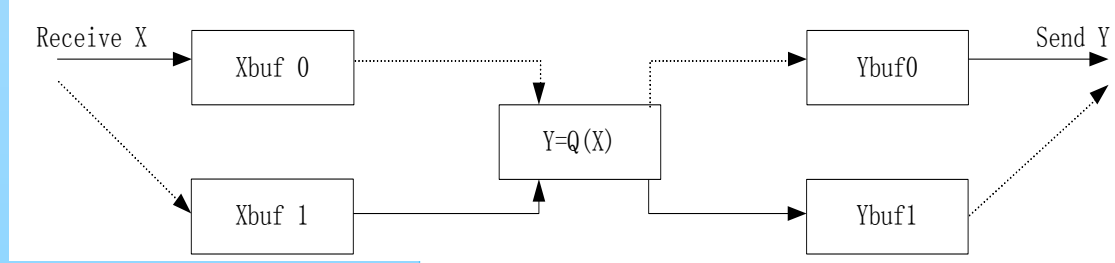
```
while (Not_Done){
  if (X==Xbuf0) {
    X=Xbuf1; Y=Ybuf1; Xin=Xbuf0; Yout=Ybuf0;
  }
  else {
    X=Xbuf0; Y=Ybuf0; Xin=Xbuf1; Yout=Ybuf1;
  }
  MPI_Irevc(Xin, …, recv_handle);
  MPI_Isend(Yout, …, send_handle);
  Y=Q(X);    /* 重叠计算*/
  MPI_Wait(recv_handle,recv_status);
  MPI_Wait(send_handle,send_status);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

计算与通信重叠（Overlap）



- 通常利用双缓冲实现

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



非阻塞通信相关MPI函数

- 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

非阻塞通信示例

```
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
          for (i=0; i< 100; i++) {
             data[i] = compute(i);
             MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                       &request[i]);
          }
         MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
    }
    else {
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```

- 2.1 MPI基本函数
- 2.2 点对点通信
- 2.3 MPI程序的编译、运行和调试

下载和安装MPICH

- Linux系统
  - Ubuntu: `sudo apt-get install mpich`
  - 从http://www.mpich.org 下载安装包自己安装
- Windows系统
  - MPICH2: https://www.mpich.org/static/tarballs/1.4.1p1/
  - MS-MPI：http://www.mpich.org
- 集群中的每个节点都要安装MPICH
- 各节点要新建相同用户名用户，设置相同密码

编译MPI程序

- For C programs: `mpicc test.c -o test`
- For C++ programs: `mpicxx test.cpp -o test`
- For Fortran 77 programs: `mpif77 test.f -o test`
- For Fortran 90 programs: `mpif90 test.f90 -o test`

➢ mpicc等是对本地编译器(gcc, gfortran, etc.)的封装，可以使用一般的编译选项，含义和原来的编译器相同

- 例如：`mpicc test.c -o test -lm`

➢ 在同构的系统上，只需编译一次；异构系统，则要在每个异构系统上都对MPI源程序进行编译

➢ 将可执行程序拷贝到各个节点机上

运行MPI程序（MPICH）

- 本地节点运行16个进程
  - `mpiexec -n 16 ./test`
- 4节点（每节点4核CPU）运行16个进程
  - `mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test`
  - `mpiexec -hosts h1,h2,h3,h4 -n 16 ./test`
- 节点数量多的时候最好先创建好一个host文件

```
cat hf
    h1:4
    h2:2
```

  - `mpiexec -hostfile hf -n 16 ./test`

运行MPI程序（MPICH）

- 本地节点运行16个进程
  - `mpiexec -n 16 ./test`
- 4节点（每节点4核CPU）运行16个进程
  - `mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test`
  - `mpiexec -hosts h1,h2,h3,h4 -n 16 ./test`
- 节点数量多的时候最好先创建好一个host文件

```
cat hf
    h1:4
    h2:2
```

  - `mpiexec -hostfile hf -n 16 ./test`

节点名可以是IP地址或DNS hostname，例如：
localhost
192.168.1.2
node1.mydomain.com

用集群系统的作业管理（资源管理）软件运行MPI程序

- 在超算等集群系统中，一般不能直接调用mpiexec 或 mpirun运行MPI程序，必须通过其上提供的作业管理系统来提交计算任务
  - PBS，LSF 和 SLURM
- 以PBS作业管理系统为例：
  - 创建PBS脚本test.sub
  - 提交作业：`qsub -l nodes=2:ppn=2 test.sub`

```
#! /bin/bash
cd $PBS_O_WORKDIR
# No need to provide -np or -hostfile options
mpiexec ./test
```

- 其它作业管理系统类似

调试MPI程序

- 调试并行程序远比调试串行程序困难
  - 不确定性：数据竞争、消息传递、动态调度、不确定的系统调用
  - 探针效应：调试工具的引入可能掩盖被调试程序中的时序错误
- 常用调试工具
  - 商业工具：TotalView、Allinea DDT
  - 开源工具：ddd、gdb、padb
- TotalView调试：`totalview -a mpiexec -n 6 ./test`
- ddd(gdb)调试：`mpiexec -n 4 ./test : -n 1 ddd ./test : -n 1 ./test`

3 深入MPI编程

- 3.1 集合通信（Collective）
- 3.2 MPI进程组和通信域管理
- 3.3 用户自定义（派生）数据类型

3.1 集合通信

- **集合通信(Collective Communications)是一个进程组中的所有进程都参加的全局通信操作**
- **集合通信一般实现三个功能：数据移动、数据聚集和同步**
  - 数据移动功能主要完成组内数据的传输
  - 数据聚集功能在通信的基础上对给定的数据完成一定的操作
  - 同步功能实现组内所有进程在执行进度上取得一致
- **集合通信，按照通信方向的不同，又可以分为三种：一对多通信，多对一通信和多对多通信**

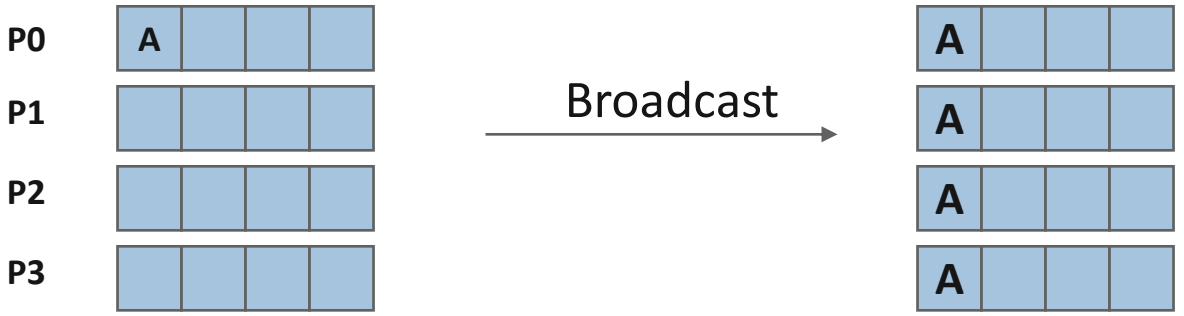
MPI集合通信函数

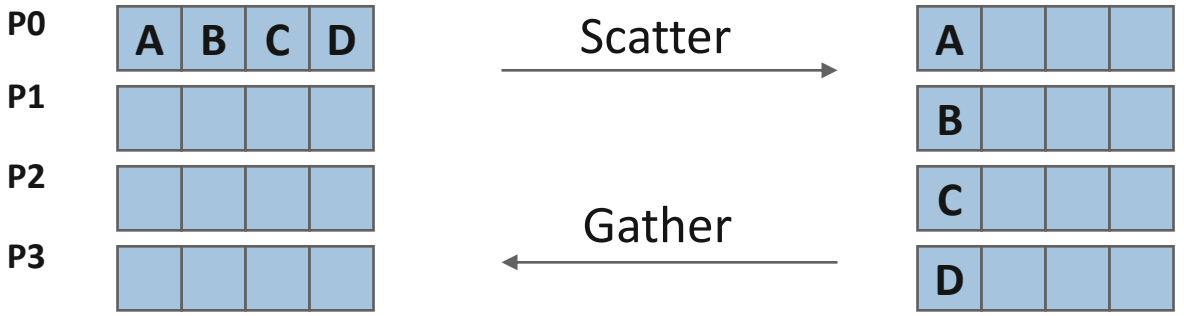
All: 表示传送结果到所有进程.

V: Variety,被操作的数据对象和操作更为灵活

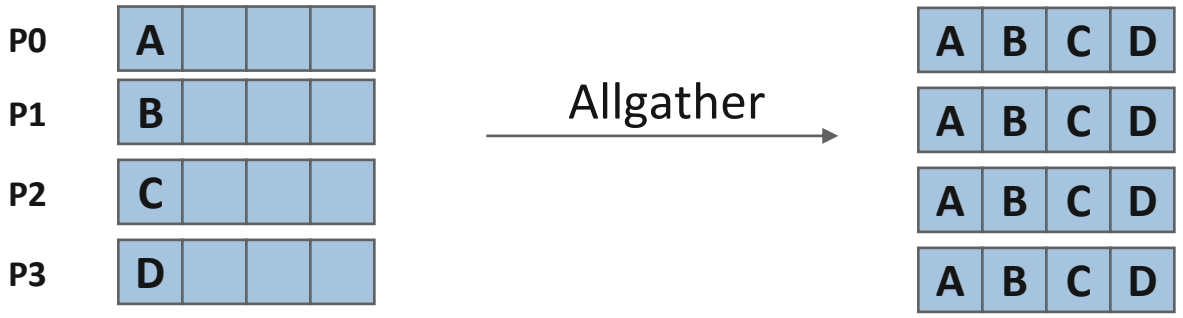
类型	函数	功能
数据移动	MPI_Bcast	一到多，数据广播
	MPI_Gather	多到一，数据汇合
	MPI_Gatherv	MPI_Gather的一般形式
	MPI_Allgather	MPI_Gather的一般形式
	MPI_Allgatherv	MPI_Allgather的一般形式
	MPI_Scatter	一到多，数据分散
	MPI_Scatterv	MPI_Scatter的一般形式
	MPI_Alltoall	多到多，置换数据(全互换)
	MPI_Alltoallv	MPI_Alltoall的一般形式
数据聚集	MPI_Reduce	多到一，数据归约
	MPI_Allreduce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

数据移动


P0
P1
P2
P3
A
Broadcast
A
A
A
A

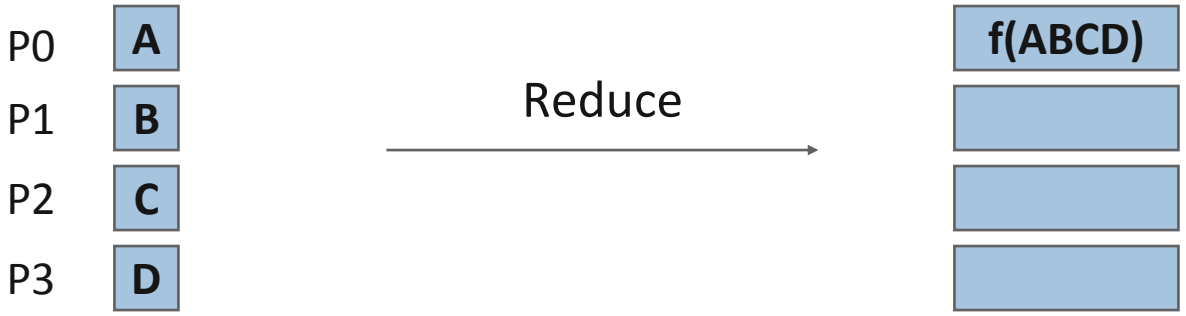

P0
P1
P2
P3
A B C D
Scatter
Gather
A
B
C
D

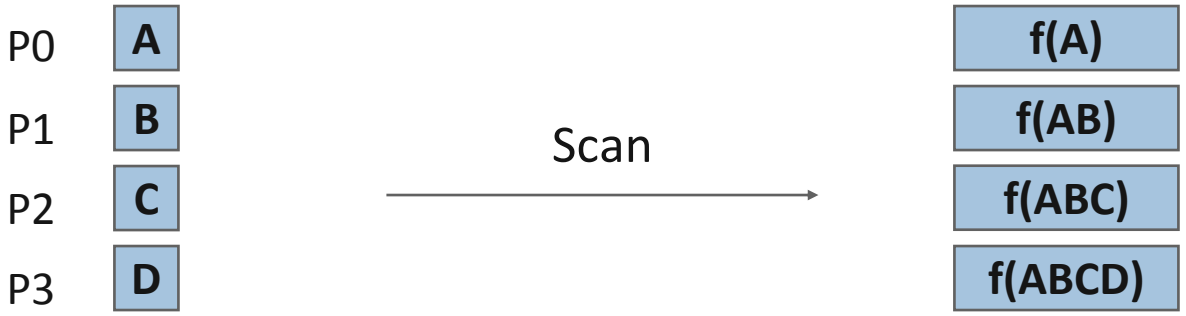
数据移动


P0
P1
P2
P3
A
B
C
D
Allgather
A B C D
A B C D
A B C D
A B C D


P0
P1
P2
P3
A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3
Alltoall
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3

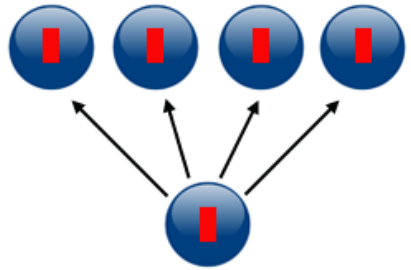
数据聚集


P0
P1
P2
P3
A
B
C
D
Reduce
f(ABCD)


P0
P1
P2
P3
A
B
C
D
Scan
f(A)
f(AB)
f(ABC)
f(ABCD)

Broadcast — 数据广播

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



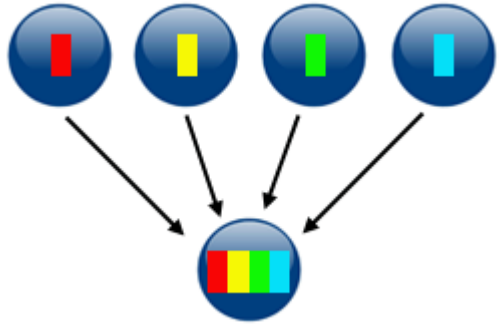
- 标号为root的进程发送相同的消息给通信域comm中的所有进程。
- 消息的内容如同点对点通信一样由三元组<buffer, count, datatype>标识
- 对root进程来说，这个三元组既定义了发送缓冲也定义了接收缓冲。对其它进程来说，这个三元组只定义了接收缓冲

```
int p, myrank;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm, &my_rank);
MPI_Comm_size(comm, &p);
if(myrank==0)
   buf = 1.0;

MPI_Bcast(&buf,1,MPI_FLOAT,0, comm);
```

Gather — 数据收集

```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype
  sendtype, void *recvbuf, int recvcnt, MPI_Datatype
  recvtype, int root, MPI_Comm comm );
```

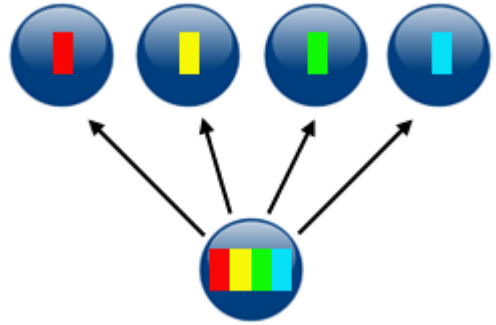


- 在收集操作中，root进程从进程域comm的所有进程(包括它自已)接收消息
- 这n个消息按照进程的标识rank排序进行拼接，然后存放在root进程的接收缓冲中
- 接收缓冲由三元组<recvbuf, recvcnt, recvtype>标识，发送缓冲由三元组<sendbuf, sendcnt, sendtype>标识，所有非root进程忽略接收缓冲
- recvcnt指根进程从每个进程接收的数据的个数，而不是进程接收的所有数据的个数

```
int p, myrank;
float data[10];
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm,&my_rank);
MPI_Comm_size(comm, &p);
if(myrank==0)
  buf=(float*)malloc(p*10*sizeof(float);
MPI_Gather(data,10,MPI_FLOAT,
       buf,10,MPI_FlOAT,0,comm);
```

Scatter — 数据分散

```
int  MPI_Scatter  (  void  *sendbuf,  int  sendcnt,
  MPI_Datatype sendtype, void *recvbuf, int recvcnt,
  MPI_Datatype recvtype, int root, MPI_Comm comm );
```



- Scatter执行与Gather相反的操作
- root进程给所有进程(包括它自已)发送一个不同的消息，这n (n为进程域comm包括的进程个数)个消息在root进程的发送缓冲区中按进程标识的顺序有序地存放
- 每个接收缓冲由三元组<recvbuf, recvcnt, recvtype>标识，所有的非root进程忽略发送缓冲。对root进程，发送缓冲由三元组<sendbuf, sendcnt, sendtype>标识

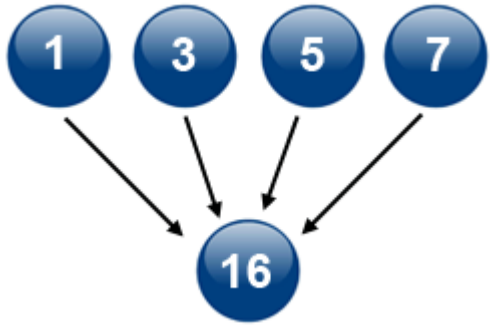
```
int p, myrank;
float data[10];
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm, &my_rank);
MPI_Comm_size(comm, &p);
if(myrank==0)
  buf = (float*)malloc(p*10*sizeof(float);
MPI_Scatter(buf,10,MPI_FLOAT,data,10,
          MPI_FlOAT,0,comm);
```

数据聚集

- 集合通信的数据聚集操作使得MPI进程通信的同时完成一定的计算
- 数据聚集操作分三步实现
  - 首先是通信的功能，即消息根据要求发送到目标进程，目标进程也已经收到了各自需要的消息；
  - 然后是对消息的处理，即执行计算功能；
  - 最后把处理结果放入指定的接收缓冲区
- MPI提供了两种类型的聚集操作函数: 归约（Reduce）和扫描（Scan）

Reduce — 数据归约

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



- 对每个进程的发送缓冲区(sendbuf)中的数据按给定的操作进行运算，并将最终结果存放在root进程的接收缓冲区(recvbuf)中
- 参与计算操作的数据项的数据类型在datatype域中定义，归约操作由op域定义
- 归约操作可以是MPI预定义的,也可以是用户自定义的
- 归约操作允许每个进程贡献向量值，而不只是标量值，向量的长度由count定义

```
int p, myrank;
float data = 0.0;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(comm,&my_rank);
data = data + myrank * 10;
MPI_Reduce(&data,&buf,1,MPI_FLOAT,MPI_SUM,
            0,comm);
```

Reduce — 数据归约

- MPI预定义的归约操作

操作	含义	操作	含义
MPI_MAX	最大值	MPI_LOR	逻辑或
MPI_MIN	最小值	MPI_BOR	按位或
MPI_SUM	求和	MPI_LXOR	逻辑异或
MPI_PROD	求积	MPI_BXOR	按位异或
MPI_LAND	逻辑与	MPI_MAXLOC	最大值且相应位置
MPI_BAND	按位与	MPI_MINLOC	最小值且相应位置

Reduce — 数据归约

- 自定义归约操作

```
MPI_OP_CREATE(user_fn, commutes, &op);
MPI_OP_FREE(&op);

user_fn(invec, inoutvec, len, datatype);
```

- 自定义的归约函数user_fn执行：

```
for i from 0 to len-1
  inoutvec[i] = invec[i] op inoutvec[i];
```

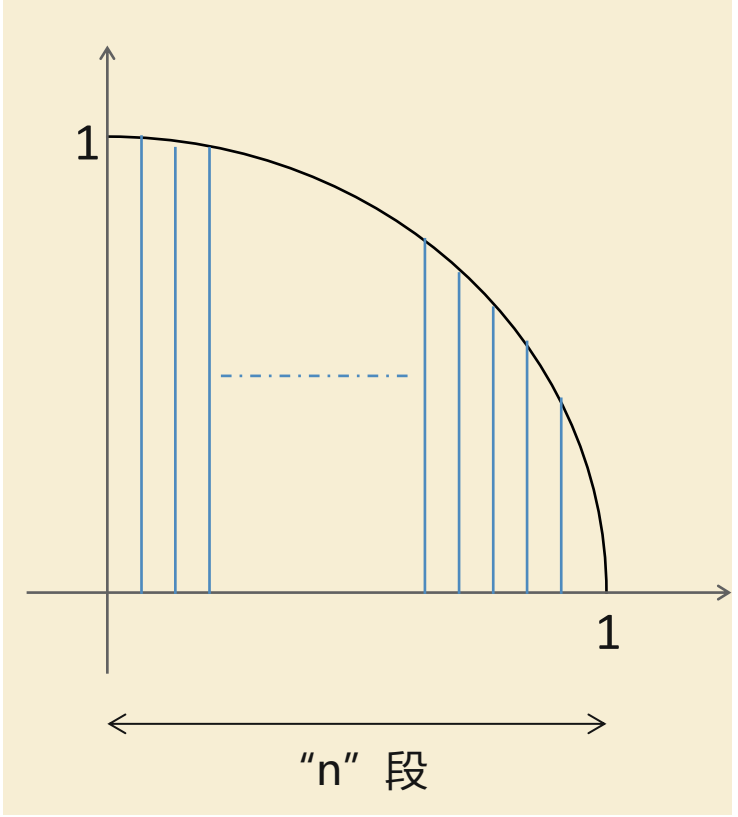
- 自定义归约操作可以不满足交换律，但必须满足结合律

集合通信示例：数值积分法求Pi

$$\pi = 4\int_0^1 \sqrt{1-x^2}\,dx \qquad \pi = 4\int_0^1 \frac{1}{1+x^2}\,dx$$

- 区间分成n小段
- n个小段均分给p个MPI进程
- 每个进程计算n/p个小段的面积和
- 将p个部分和相加得到Pi

- 每段宽度w=1/n
- 第i段起点的横坐标d(i)=i*w
- 第i段对应小矩形高度为：sqrt(1 – [d(i)]^2)



集合通信示例：数值积分法求Pi

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = rank + 1; i <= n; i += size)
        mypi += w * sqrt(1 - (((double) i / n) * ((double) i / n));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 4 * pi,
               fabs((4 * pi) - PI25DT));
    [...snip...]
}
```

栅栏同步

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

- 阻塞调用进程，直到通信域comm内所有进程调用了这个函数。当MPI_Barrier返回时，所有进程都在这个栅栏上同步了
- MPI_Barrier由软件完成，在一些机器上可能造成显著的开销

集合通信的特点：

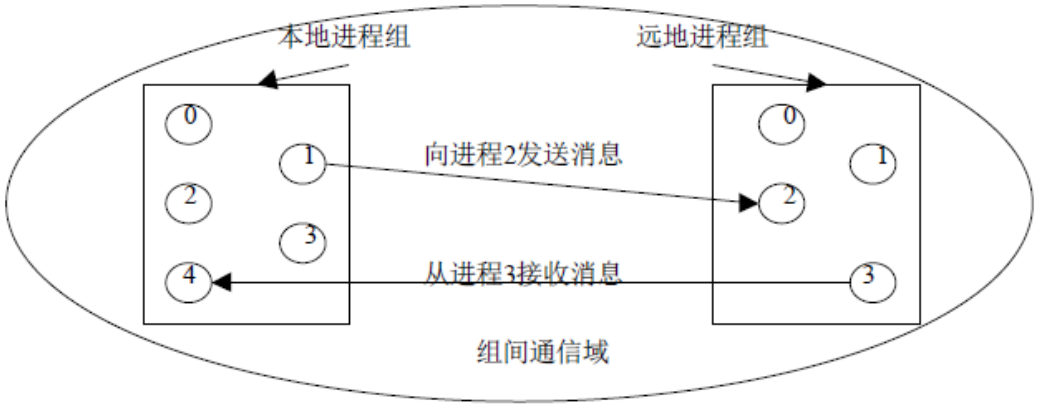
- 通信域中的所有进程必须调用集合通信函数。如果只有通信域中的一部分成员调用了集合通信函数而其它没有调用，则是错误的
- 除MPI_Barrier以外，每个集合通信函数使用类似于点对点通信中的标准、阻塞的通信模式。也就是说，一个进程一旦结束了它所参与的集合操作就从集合函数中返回，但是并不保证其它进程执行该集合函数已经完成
- 所有参与集合操作的进程中，Count和Datatype必须是兼容的
- 集合通信中的消息没有消息标签参数，消息信封由通信域和源/目标定义。例如在MPI_Bcast中，消息的源是Root进程，而目标是所有进程(包括Root)

- 3.1 集合通信（Collective）
- 3.2 MPI进程组和通信域管理
- 3.3 用户自定义（派生）数据类型

3.2 MPI进程组和通信域管理

- **通信域：描述进程间的通信关系，包括**
  - 通信上下文：区别不同的通信域，一个上下文所发送的消息不能被另一个上下文所接收
  - 进程组：多个进程的有序集合
  - 虚拟拓扑：多个进程在逻辑上的排列关系，反映了进程间的通信模型
  - 属性：用户可通过自定义的属性，将任意信息附加到通信域上
- **MPI进程的通信在通信域的控制和维护下进行**
- **所有MPI通信函数都直接或间接用到通信域这一参数**
- **对通信域的重组和划分可以方便实现任务的划分（进程的分工）**

- **通信域分为组内通信域和组间通信域，分别用来实现MPI的组内通信(Intra-communication)和组间通信(Inter-communication)**
  - 组内通信域包含一个进程组，通信在该进程组内的进程间进行
  - 组间通信域包含两个进程组：本地进程组、远程进程组。通过组间通信域可以实现两个不同进程组的进程之间的通信



3.2 MPI进程组和通信域管理

- 一个进程用它在一个通信域(进程组)中的编号进行标识。组的大小和进程编号可以通过调用以下的MPI函数获得：

```
MPI_Comm_size(communicator, &group_size)
MPI_Comm_rank(communicator, &my_rank)
```

- 一个进程可以属于多个通信域，具有不同编号(rank)
- 如何得到通信域包含的进程组？

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

预定义的进程组和通信域

- MPI_COMM_WORLD：有效通信域句柄，包括元素为所有进程
- MPI_COMM_SELF：有效通信域句柄，包括元素仅为当前进程
- MPI_GROUP_EMPTY：有效进程组句柄，包括元素个数为0
- MPI_GROUP_NULL：无效进程组句柄
- MPI_COMM_NULL：无效通信域句柄

进程组管理函数

操作	函数
并	MPI_Group_union(group1,group2,newgroup)
交	MPI_Group_intersection(group1,group2,newgroup)
差	MPI_Group_difference(group1,group2,newgroup)
子集	MPI_Group_incl(group,n,ranks,newgroup)
去掉子集	MPI_Group_excl(group,n,ranks,newgroup)
比较	MPI_Group_compare(group1,group2,result)
组中进程个数	MPI_Group_size(group,size)
当前进程rank	MPI_Group_rank(group,rank)
释放	MPI_Group_free(group)

通信域管理函数

操作	函数
创建	MPI_Comm_create(comm,group,newcomm)
复制	MPI_Comm_dup(comm,newcomm)
分裂	MPI_Comm_split(comm,color,key,newcomm)
比较	MPI_Comm_compare(comm1,comm2,result)
域中进程个数	MPI_Comm_size(comm,size)
当前进程rank	MPI_Comm_rank(comm,rank)
释放	MPI_Comm_free(comm)

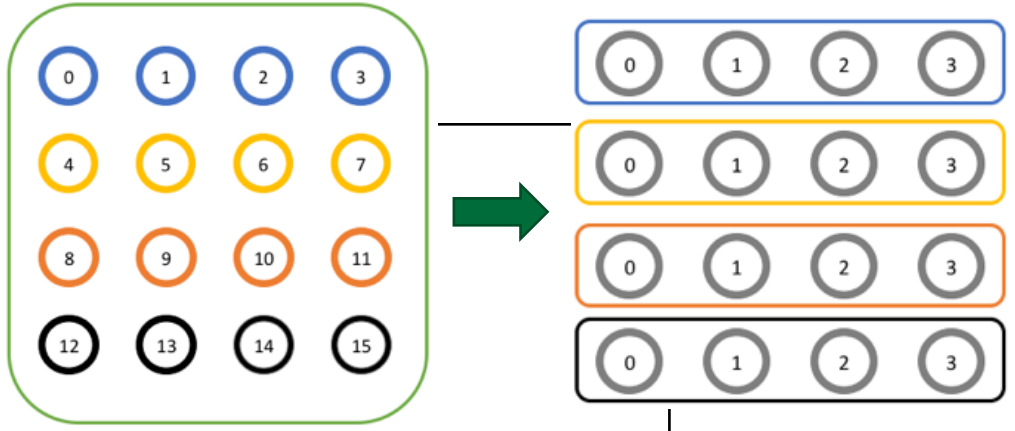
通信域的分裂

```
int MPI_Comm_split(MPI_Comm comm,int color,int key,MPI_Comm *newcomm)
```

- 根据color值的不同，形成不同的通信域，同时也就确定了当前进程落入哪个分裂后的通信域
- 根据key值的大小，决定新通信域中的进程编号顺序（不是指定rank）

- `MPI_Comm_split(comm, 0, rank, &newcomm)` 等同于 `MPI_Comm_dup`
- `MPI_Comm_split(comm, rank, rank, &newcomm)` 为每个进程返回一个只包含自己的通信域，即`MPI_COMM_SELF`

通信域的分裂




```
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4;

MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
        world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

进程组和通信域管理示例：分组求和

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

int main(int argc, char *argv[])  {
  int rank, new_rank, sendbuf, recvbuf, numtasks,
      ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
  MPI_Group  orig_group, new_group;
  MPI_Comm   new_comm;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  sendbuf = rank;
  // extract the original group handle
  MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
  if (rank < NPROCS/2)
    MPI_Group_incl(orig_group, NPROCS/2, ranks1,
       &new_group);
  else
    MPI_Group_incl(orig_group, NPROCS/2, ranks2,
       &new_group);

  MPI_Comm_create(MPI_COMM_WORLD,new_group,&new_comm);
  MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT,
         MPI_SUM, new_comm);

  MPI_Group_rank (new_group, &new_rank);
  printf("rank= %d newrank= %d recvbuf= %d\n",
          rank,new_rank,recvbuf);
  MPI_Finalize();
  return 0;
}
```

进程组和通信域管理示例：分组求和

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

int main(int a
  int rank, ne
      ranks1[4
  MPI_Group   o
  MPI_Comm    n

  MPI_Init(&ar
  MPI_Comm_ran
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  sendbuf = rank;
  // extract the original group handle
  MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

```
rank= 0 newrank= 0 recvbuf= 6
rank= 1 newrank= 1 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 5 newrank= 1 recvbuf= 22
rank= 6 newrank= 2 recvbuf= 22
rank= 7 newrank= 3 recvbuf= 22
```

```
  if (rank < NPROCS/2)
    MPI_Group_incl(orig_group, NPROCS/2, ranks1,
       &new_group);
  else
    MPI_Group_incl(orig_group, NPROCS/2, ranks2,
       &new_group);

  MPI_Comm_create(MPI_COMM_WORLD,new_group,&new_comm);
  MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT,
         MPI_SUM, new_comm);

  MPI_Group_rank (new_group, &new_rank);
  printf("rank= %d newrank= %d recvbuf= %d\n",
          rank,new_rank,recvbuf);
  MPI_Finalize();
  return 0;
}
```

组间通信域相关函数

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

- 3.1 集合通信（Collective）
- 3.2 MPI进程组和通信域管理
- 3.3 用户自定义（派生）数据类型

3.3 用户自定义（派生）数据类型

- **MPI消息的数据类型支持将任意结构的数据序列化/反序列化**
  - 数据以串行数据流的形式在一条网络信道上传输
  - 块设备等I/O设备的数据传输也需要将内存数据序列化
- **MPI消息的数据类型分为两种：预定义数据类型和派生数据类型**
  - 预定义数据类型：不同系统有不同的数据表示格式。MPI预先定义一些基本数据类型，在实现过程中通过这些基本数据类型为桥梁进行转换，来解决不同系统之间的互操作性问题
  - 派生数据类型：用来定义来自不连续的和类型不一致的存储区域的消息。它的使用可有效地减少消息传递的次数，增大通信粒度，并且在收/发消息时避免或减少数据在内存中的拷贝、复制

MPI的预定义数据类型

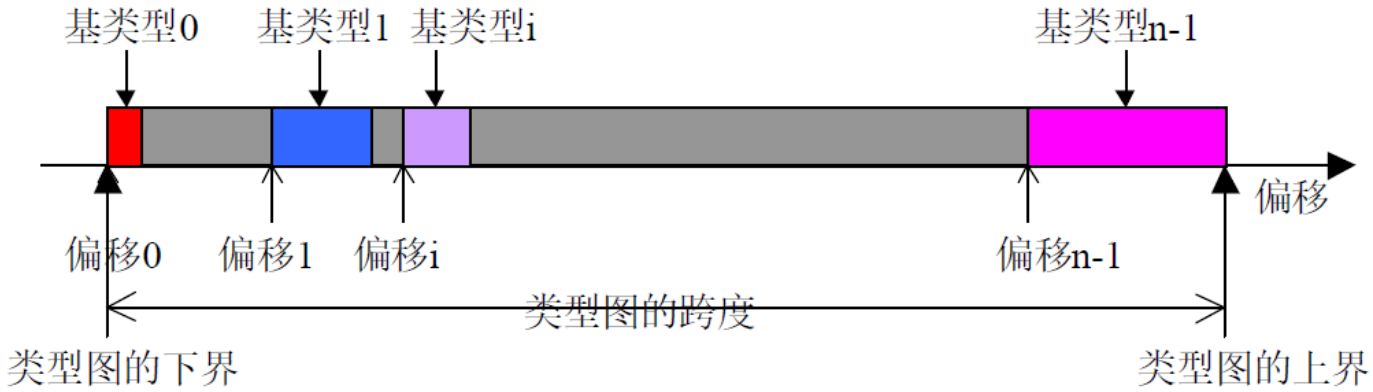
C Data Types		Fortran Data Types
MPI_CHAR	MPI_C_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_FLOAT_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_DOUBLE_COMPLEX	MPI_INTEGER1
MPI_INT	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER2
MPI_LONG	MPI_C_BOOL	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_LOGICAL	MPI_REAL
MPI_LONG_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT8_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT16_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_INT32_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_INT64_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT8_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT16_T	MPI_LOGICAL
MPI_DOUBLE	MPI_UINT32_T	MPI_BYTE MPI_PACKED
MPI_LONG_DOUBLE	MPI_UINT64_T	
	MPI_BYTE	
	MPI_PACKED	

MPI的派生数据类型

- 派生数据类型可以用类型图来描述，这是一种通用的类型描述方法，它是一系列二元组<基类型，偏移>的集合，可以表示成如下格式：

  {<基类型0,偏移0>，···，<基类型n-1,偏移n-1>}

- 在派生数据类型中，基类型可以是任何MPI预定义数据类型，也可以是其它的派生数据类型，即支持数据类型的嵌套定义
- 如图，阴影部分是基类型所占用的空间，其它空间可以是特意留下的，也可以是为了方便数据对齐

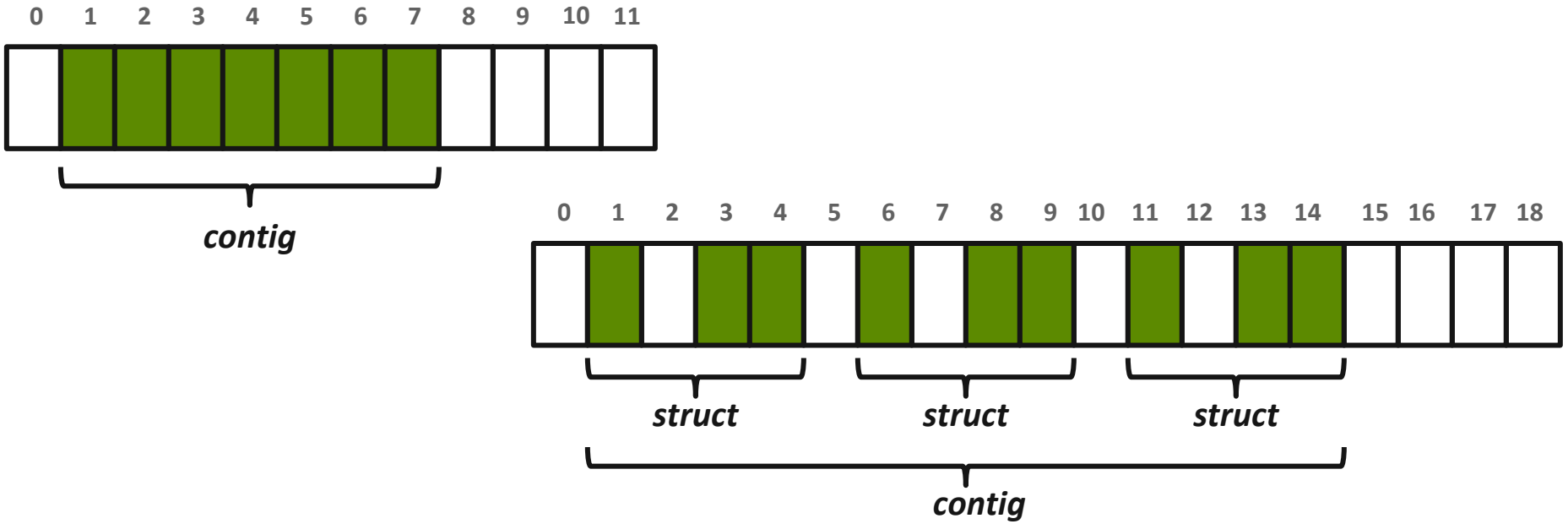


- MPI提供了许多函数来构造派生数据类型，常用的几类构造函数有：
  - Contiguous（连续）
  - Vector/Hvector（向量）
  - Indexed/Indexed_block/Hindexed/Hindexed_block（索引）
  - Struct（结构）
  - Some convenience types (e.g., subarray)（子数组）

Contiguous（连续）

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

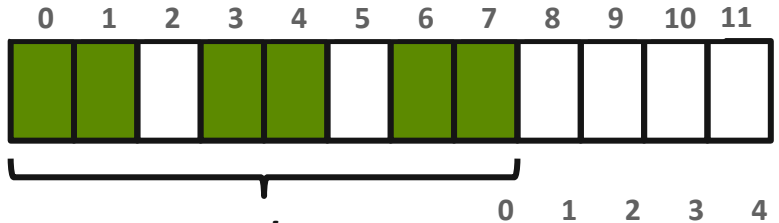
- 将一个已有的数据类型按顺序依次连续进行复制



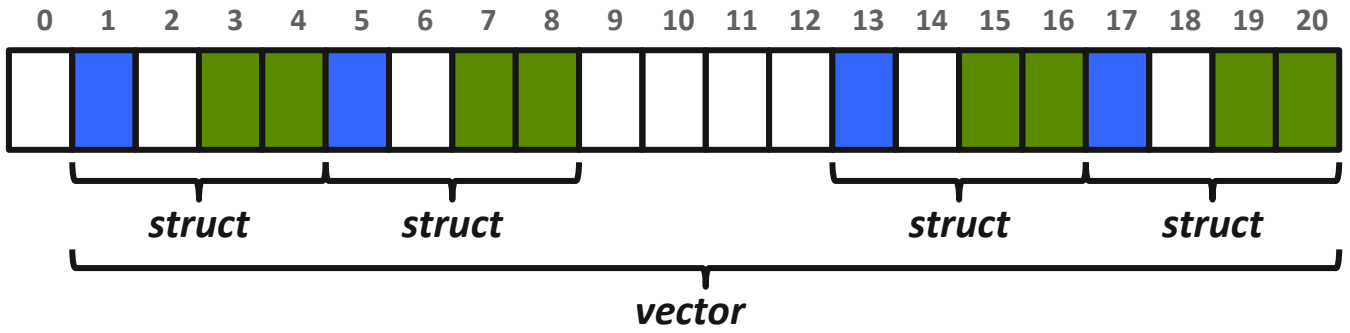
Vector/Hvector（向量）

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

- 将一个已有的数据类型按顺序依次连续进行复制
- 首先通过连续复制若干个旧数据类型形成一个“块”，然后通过等间隔地复制该块儿形成新的数据类型。块与块之间的空间是旧数据类型的倍数




```
MPI_Type_vector(2, 2, 3, mystruct, myvector);
```

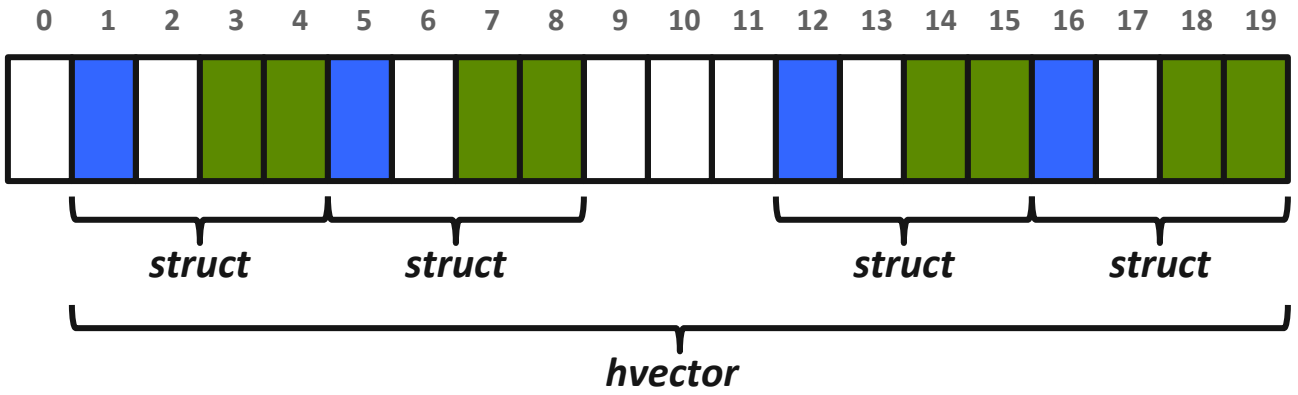


Vector/Hvector（向量）

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

- 和MPI_Type_vector一样，唯一的区别是块与块之间的间隔可以是任意字节

```
MPI_Type_create_hvector(2, 2, 11, mystruct, myhvector);
```

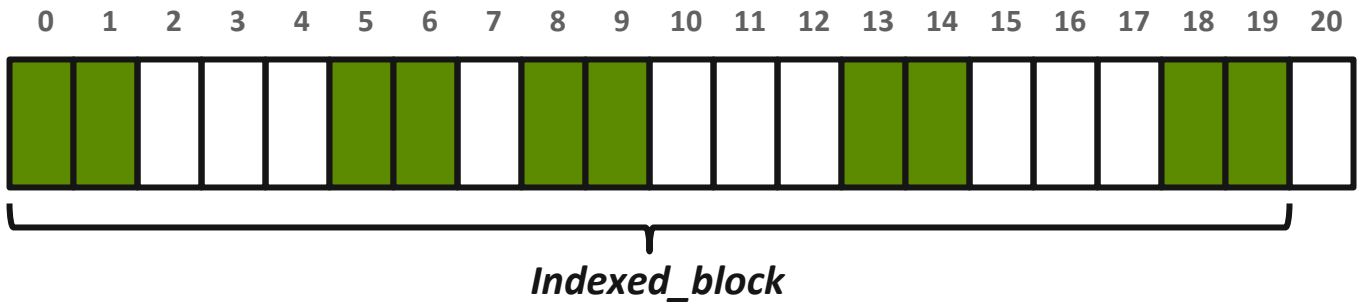


Indexed_block（索引块）

```
MPI_Type_create_indexed_block(int count, int blocklength, int
    *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- 从一个数组中提取不规则的数据子集
- 示例： displs={0,5,8,13,18}

```
MPI_Type_create_indexed_block(5, 2, displs, oldtype, newtype);
```

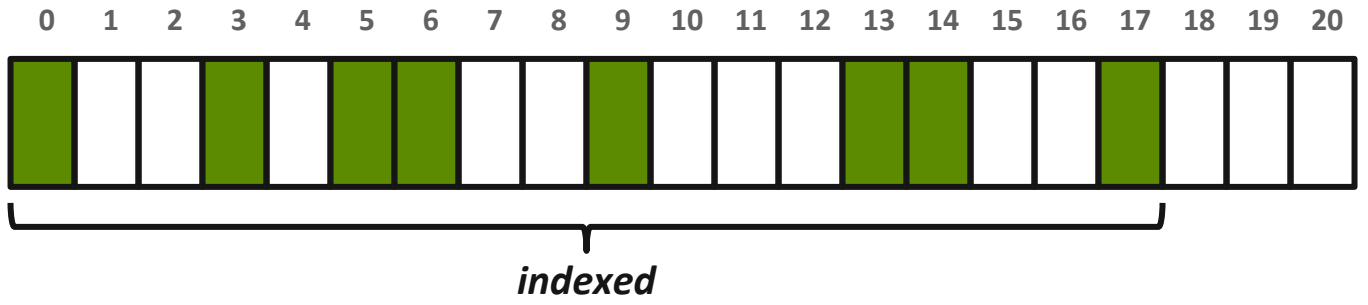


Indexed（索引）

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

- 和indexed_block类似，只不过每个块可以有不同的长度
- 示例：blen={1,1,2,1,2,1}　　displs={0,3,5,9,13,17}

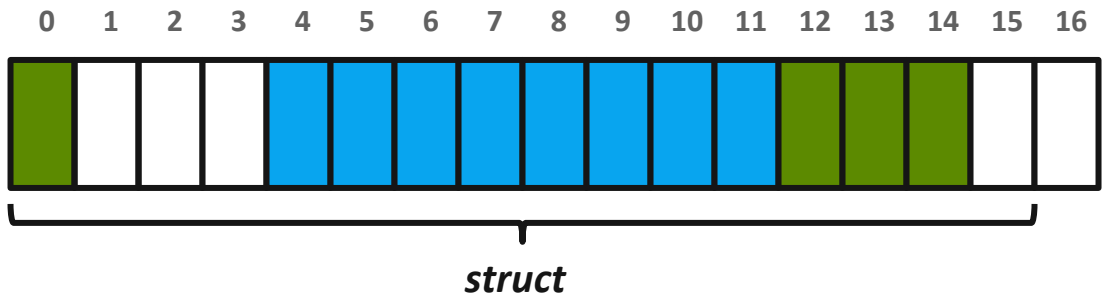
```
MPI_Type_indexed(6, blen, displs, oldtype, newtype);
```



- Struct（结构）

```
MPI_Type_create_struct(int count, int array_of_blocklengths[],
                       MPI_Aint array_of_displacements[],
                       MPI_Datatype array_of_types[],
                       MPI_Datatype *newtype);
```

  - 最通用的构造函数，允许使用不同的原类型和任意数量及间隔（也是开销最大的）
  - 示例：`type[3]={MPI_CHAR, MPI_FLOAT,MPI_CHAR};  blocklen[3]={1,2,3};`
    `disp[3]={0,sizeof(float), 3*sizeof(float)};`



Subarray（子数组）

```
MPI_Type_create_subarray(int ndims, int array_of_sizes[],
        int array_of_subsizes[], int array_of_starts[],
        int order, MPI_Datatype oldtype,
        MPI_Datatype *newtype);
```

- 截取n维数组类型的一个片段作为新的数据类型
- 示例：starts[2]={1,2}; subsizes[2]={2,2}; oldsizes[2]={4,4};

```
MPI_Type_create_subarray(2, oldsizes,
    subsizes, starts, MPI_ORDER_C,
    MPI_INT, &mysubarray);
```

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

3.3 用户自定义（派生）数据类型

- **MPI_Type_commit (MPI Datatype *datatype)**
  - 新的派生数据类型必须先调用函数MPI_Type_commit获得MPI系统的确认后才能使用
- **MPI_Type_free (MPI Datatype *datatype)**
  - 将数据类型设为MPI_DATATYPE_NULL
- **MPI_Type_dup (MPI_Datatype oldtype, MPI_Datatype * newtype)**


北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY
谢谢!
德以明理 学以精工

3.3 用户自定义（派生）数据类型

- 为保证可移植性，用MPI_BOTTOM表示派生类型的起始地址，而不是0
- 调用MPI_Address返回某一个基类型在内存中相对于派生类型起始地址MPI_BOTTOM的偏移，而不要使用&
- 常用于下列情况：
  - 建立结构数据类型
  - 数据跨越多个数组时