



# 高性能计算与云计算

## 第八讲 消息传递编程

胡金龙，董守斌

华南理工大学计算机学院  
广东省计算机网络重点实验室

Communication & Computer Network Laboratory (CCNL)

# 内容概要

---

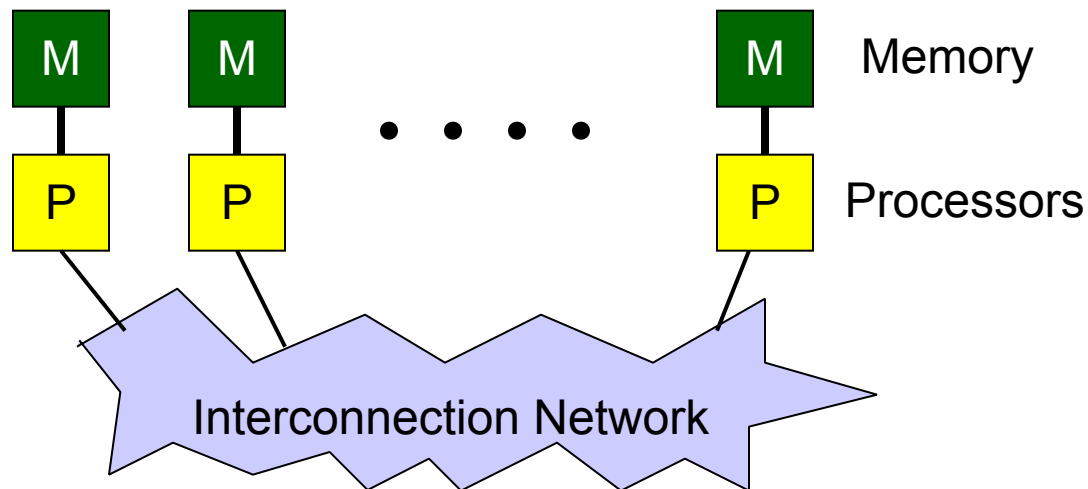
- 引言
- **MPI消息**
- 点对点通信
- 群集通信
- 总结

# 消息传递

- 分布式存储类型的并行机，一台处理器无法直接访问另一台处理器的地址空间
  - **消息传递**（Message Passing, MP）：显式地通过发送和接收消息来实现处理器之间的数据交换。子程序间通信，复杂但又灵活...
  - 其他...
- 消息传递是MPP和Cluster的主要编程方式

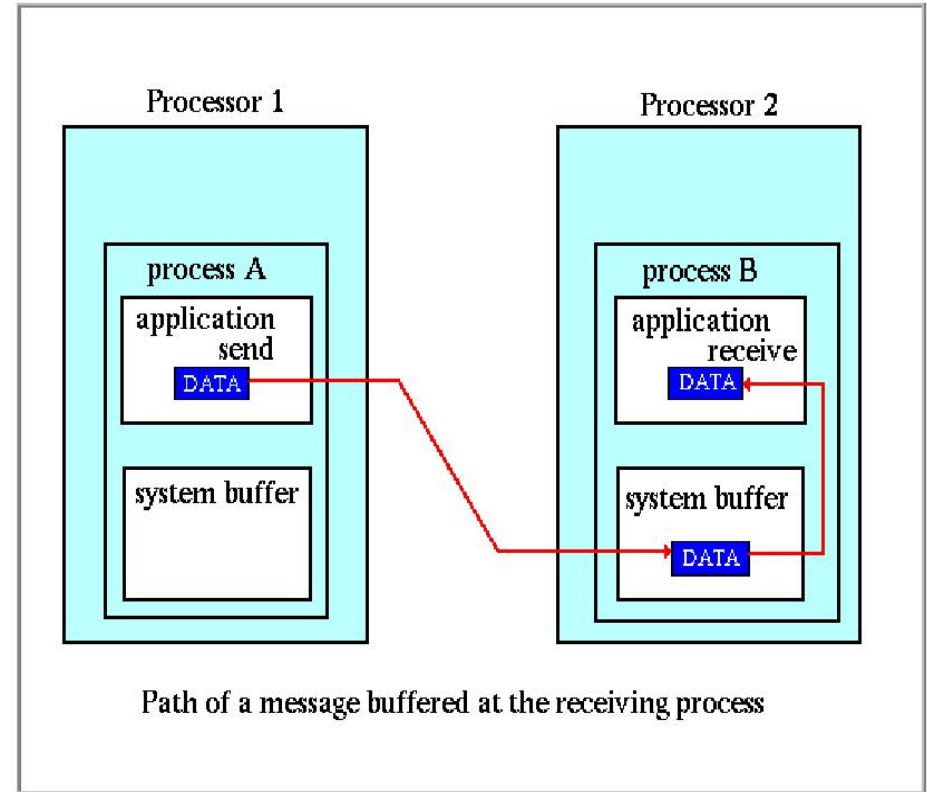
# 消息传递编程模型

- 在一个消息传递系统中，每个处理器运行一个子程序
  - 类似通常的串程序序
  - 所有变量都是私有的
  - 通过特殊的函数调用来通信



# 消息 (Messages)

- **消息**是指子程序之间传递的数据包
- 包含的信息：
  - 发送处理器 (Sending processor)
  - 源位置 (Source location)
  - 数据类型 (Data type)
  - 数据长度 (Data length)
  - 接收处理器 (Receiving processor(s))
  - 目标位置 (Destination location)
  - 目标大小 (Destination size)



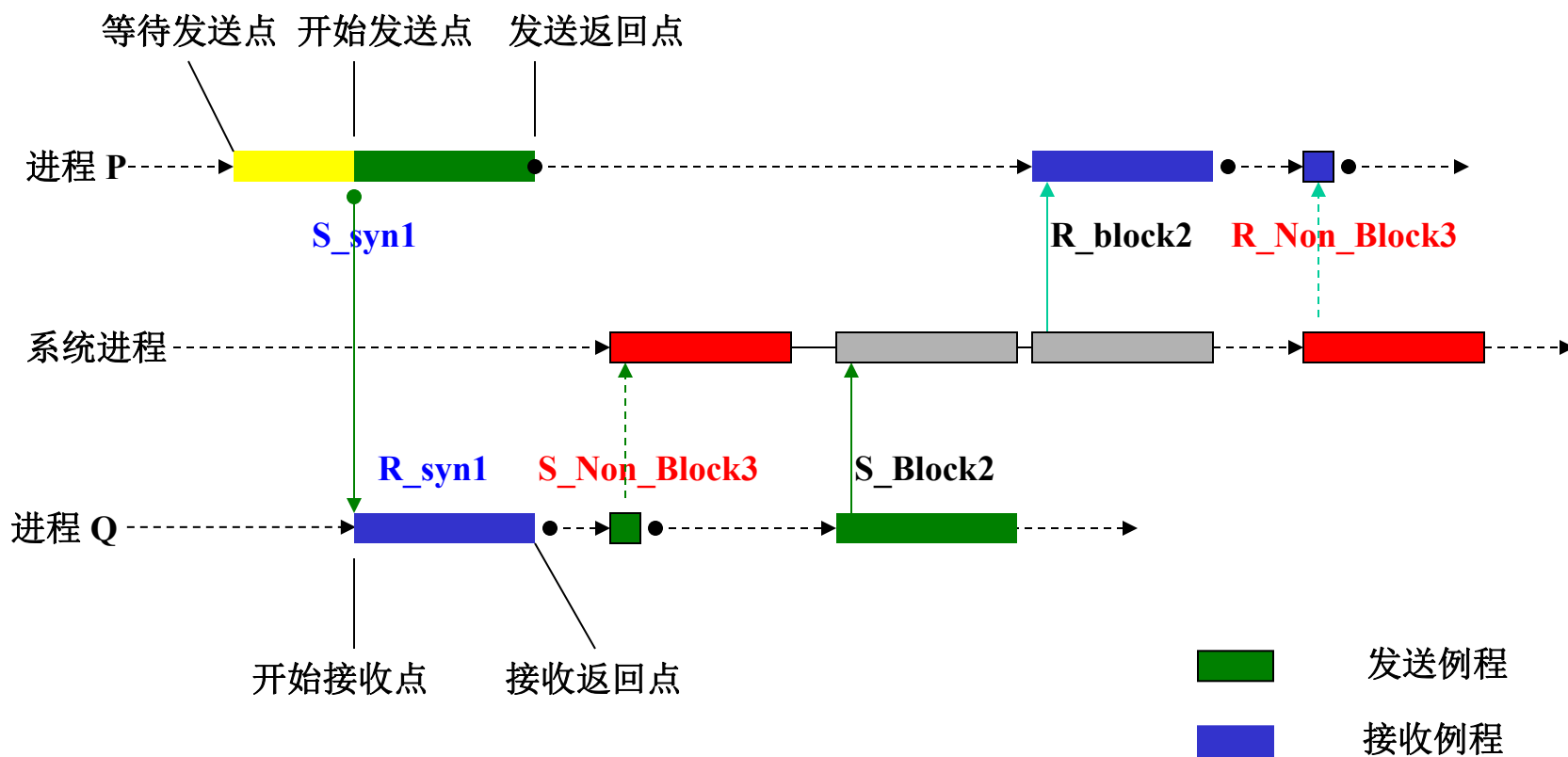
# 消息（续）

- **存取（Access）：**
  - 每个子系统需要连接到消息传递系统
- **地址（Addressing）：**
  - 消息要有地址才能传送
- **接收方（Reception）：**
  - 接收进程能够处理消息
- 一个消息传递系统类似于：
  - 邮局、电话、传真、电子邮件等
- **通信类型：**
  - 点到点（Point-to-Point），群集（Collective）

# 点到点通信

- 最简单的消息传递模式：一个进程发送消息给另一个
- 几种方式：
  - **同步发送**（Synchronous Sends）
    - ✓ 提供消息完成的信息
    - ✓ 类似：电话、传真
  - **异步发送**（Asynchronous Sends）
    - ✓ 仅知道消息已经发走
    - ✓ 类似：明信片
  - **阻塞操作**（Blocking operations）
    - ✓ 仅当传输操作完成才从调用中返回
  - **非阻塞操作**（Non-blocking operations）
    - ✓ 直接返回—以后再测试（或等待）是否完成

# 消息传递方式





# 群集通信

- 一次有多个进程参与，可以建立在点到点通信的基础上
- 例如：
  - **同步 (Barriers)**
    - 同步进程
  - **广播 (Broadcast)**
    - 一对多的通信
  - **归约操作 (Reduction operations)**
    - 从多个进程合并数据，以产生一个单一的结果

# 消息传递系统

- 早期，各发展商各自开发，特征各异，通常不兼容
- **PVM - Parallel Virtual Machine**
  - MPI出现之前的事实标准 (*de facto* standard)
  - 开源但不是public domain (公共域)
  - 提供用户界面
  - 支持动态环境
- **MPI - Message Passing Interface**

# 什么是MPI?

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

# MPI的历史

- **MPI论坛 ([www.mpi-forum.org](http://www.mpi-forum.org))**
  - 1992年11月开始，来自40个组织的60个人参加
  - 在1993的Supercomputing会议上提出草案
  - 1994年5月发布1.0版本，分为8个类别的115个函数（`routines/functions`）
  - 1995年6月发布1.1版本
  - 1997年7月发布MPI-2
  - 2015年6月发布MPI 3.1
  - 2018年11月发布新的MPI标准草案

# MPI的实现

- **MPICH** , 美国阿贡国家实验室 (Argonne National Lab) 和密执根州立大学 (and Mississippi State Univ.) 共同开发
  - <http://mpich.org>
- **OpenMPI** : 融合了之前各个开源项目 (FT-MPI, LA-MPI, LAM/MPI, PACX-MPI) 的技术
  - <http://www.openmpi.org/>
- **MS-MPI** (Microsoft MPI) , Intel MPI

<http://mpi-forum.org/slides/2016/06/mpi31-impl-status-Jun16.pdf>

# MPI通用程序结构

---

**MPI Include File**

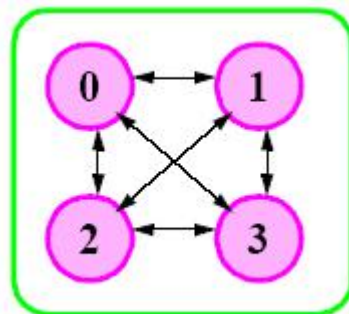
**Initialize MPI Environment**

**Do work and perform message communication**

**Terminate MPI Environment**

# 一个简单的MPI应用

- MPI应用的基本元素
  - 4个进程，编号从0到3
  - 进程之间存在通信路径



- 进程集合加上通信频道被称为通信域
  - “MPI\_COMM\_WORLD”

# 例子: Hello World

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

int main( int argc, char *argv[] )
{
    int rank, size, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Get_processor_name( processor_name, &namelen );
    printf("Hello World! I'm rank %d of %d on %s\n",
           rank, size, processor_name);

    MPI_Finalize();
    return 0;
}
```



# 编译和运行

---

- 编译：

**mpicc -o hello hello.c**

- 运行：

**mpirun -np 4 ./hello**

- 4个进程的运行结果：

**Hello World! I'm rank 0 of 4 on node01**

**Hello World! I'm rank 2 of 4 on node03**

**Hello World! I'm rank 3 of 4 on node04**

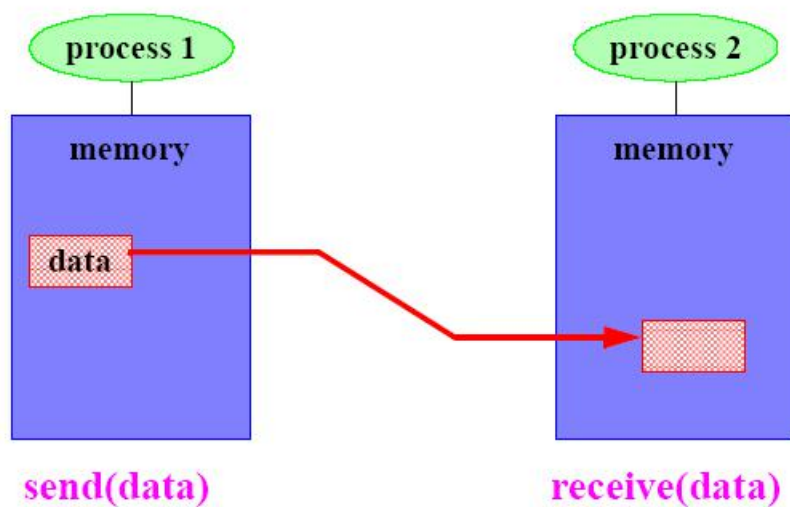
**Hello World! I'm rank 1 of 4 on node02**

# 6个基本函数（1）

- MPI初始化：通过**MPI\_Init**函数进入MPI环境并完成所有的初始化工作
  - `int MPI_Init( int *argc, char * * * argv )`
- MPI结束：通过**MPI\_Finalize**函数从MPI环境中退出
  - `int MPI_Finalize(void)`
- 获取进程的编号：调用**MPI\_Comm\_rank**函数获得当前进程在指定通信域中的编号，将自身与其他程序区分
  - `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- 获取指定通信域的进程数：调用**MPI\_Comm\_size**函数获取指定通信域的进程个数，确定自身完成任务比例
  - `int MPI_Comm_size(MPI_Comm comm, int *size)`

# 消息的发送和接收

- 双边的操作：发送者和接收者都要动作



- 在发送和接收时要考虑什么？

## 6个基本函数（2）

- 消息发送： **MPI\_Send**函数用于发送一个消息到目标进程
  - **int MPI\_Send( void \*buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)**
- 消息接受： **MPI\_Recv**函数用于从指定进程接收一个消息
  - **int MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \*status)**

### MPI消息：

- 消息缓冲（Message Buffer）： (buf, count, datatype)
- 消息封装（Message Envelop）： (dest, tag, comm)

# 内容概要

---

- 引言
- **MPI消息**
- 点对点通信
- 群集通信
- 小结

# MPI消息


---

- 一个消息好比一封信
- 消息的内容即信的内容，在MPI中成为**消息缓冲**（Message Buffer）
- 消息的接收者即信的地址，在MPI中成为**消息封装**（Message Envelop）

# MPI消息

- 消息缓冲由三元组<起始地址，数据个数，数据类型>标识
- 消息封装由三元组<源/目标进程，消息标签，通信域>标识

MPI\_Send (buf, count, datatype, dest, tag, comm)



消息缓冲                      消息信封

- 三元组的方式使得MPI可以表达丰富的信息

# 消息标签

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

<b>Process P:</b>	<b>Process Q:</b>
Send(A, 32, Q)	recv (X, 32, P)
Send(B, 16, Q)	recv (Y, 16, P)

- 使用标签可以避免这个错误

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



# 消息标签（续）

- 添加标签使得服务进程可以对两个不同的用户进程分别处理，提高灵活性

**Process P:**

```
send (request1,32, Q)
```

**Process R:**

```
send (request2, 32, Q)
```

**Process Q:**

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

**Process P:**

```
send(request1, 32, Q, tag1)
```

**Process R:**

```
send(request2, 32, Q, tag2)
```

**Process Q:**

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

# 数据类型

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

# 预定义的数据类型

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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

# 附加数据类型

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

# MPI\_PACKED的例子

```
double A[100];

MPI_Pack_size (50,MPI_DOUBLE,comm,&BufferSize);
TempBuffer = malloc(BufferSize);
j = sizeof(MPI_DOUBLE);
Position = 0;

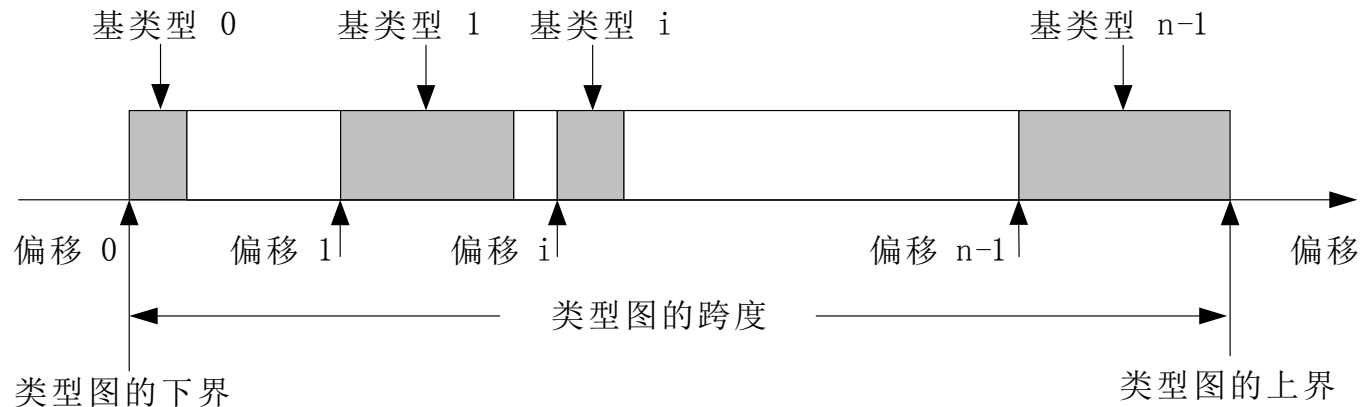
for (i=0;i<50;i++)
    MPI_Pack(A+i*j, 1, MPI_DOUBLE, TempBuffer,
            BufferSize, &Position, comm);

MPI_Send(TempBuffer, Position, MPI_PACKED, destination,
        tag, comm);
```

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

# 派生数据类型

- 派生数据类型可以用类型图来描述，它是一系列二元组  $\langle \text{基类型}, \text{偏移} \rangle$  的集合，可以表示成如下格式：
  - $\emptyset \{ \langle \text{基类型}0, \text{偏移}0 \rangle, \dots, \langle \text{基类型}n-1, \text{偏移}n-1 \rangle \}$
  - $\emptyset$  **基类型** 可以是任何MPI预定义数据类型，或其它的派生数据类型（数据类型的嵌套定义）



# 派生数据类型的构造函数

- MPI提供**构造函数**（Constructor Function）来定义派生数据类型

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

# 例子：构造函数

- 构造函数**`MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)`**定义的派生数据类型：
  - 该**`newtype`**由**`count`**个数据块组成
  - 每个数据块由**`blocklength`**个**`oldtype`**类型的连续数据项组成
  - 参数**`stride`**定义了两个连续数据块之间的**`oldtype`**类型元素的个数。因此，两个块之间的间隔可以由 **`(stride-blocklength)`** 来表示
- 例：**`MPI_Type_vector(50, 1, 2, MPI_DOUBLE, &EvenElements)`**
  - 这个函数调用产生了派生数据类型**`EvenElements`**，它由**50**个块组成，每个块由一个双精度数组成，后跟一个**`MPI_DOUBLE`**（8字节）的间隔，接在后面的是下一块



# 例子：构造派生数据类型

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

- 首先声明一个类型为MPI\_Datatype的变量EvenElements
- 调用构造函数MPI\_Type\_vector定义派生数据类型
- 新的派生数据类型必须先调用函数MPI\_Type\_commit获得MPI系统的确认后才能调用MPI\_Send进行消息发送

# 通信域

- **通信域**（Communicator）包括**进程组**（Process Group）和**通信上下文**（Communication Context）等内容，用于描述通信进程间的通信关系
- 通信域分为组内通信域和组间通信域，分别用来实现MPI的**组内通信**（Intra-communication）和**组间通信**（Inter-communication）

# 通信域—进程组

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

# 通信域—通信上下文

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

# 通信域的管理

- MPI提供丰富的函数用于管理通信域

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

# 消息状态

- **消息状态**（`MPI_Status`类型）存放接收消息的状态信息，包括消息的源进程标识、消息标签、包含的数据项个数等
- 它是消息接收函数`MPI_Recv`的最后一个参数
- 当一个接收者从不同进程接收不同大小和不同标签的消息时，消息的状态信息非常有用

# 消息状态

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

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

# 内容概要

---

- 引言
- MPI消息
- 点对点通信
- 群集通信
- 总结

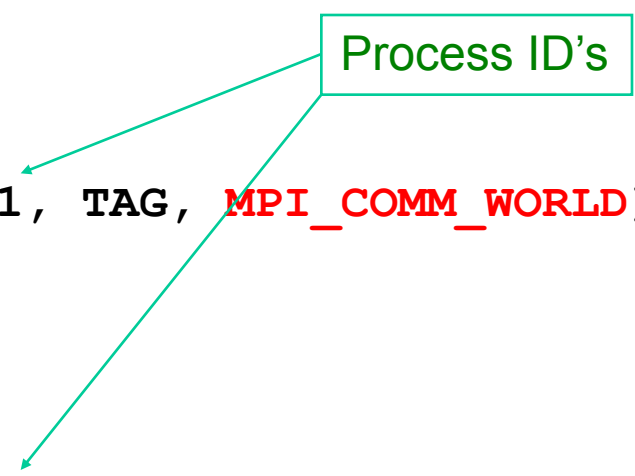


# 点到点通信的例子

Process 0 sends array “A” to process 1 which receives it as “B”

1:

```
#define TAG 123
double A[10];
MPI_Send(A, 10, MPI_DOUBLE, 1, TAG, MPI_COMM_WORLD)
```



2:

```
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD,
        &status)
```

or

```
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &status)
```

# 点对点通信

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

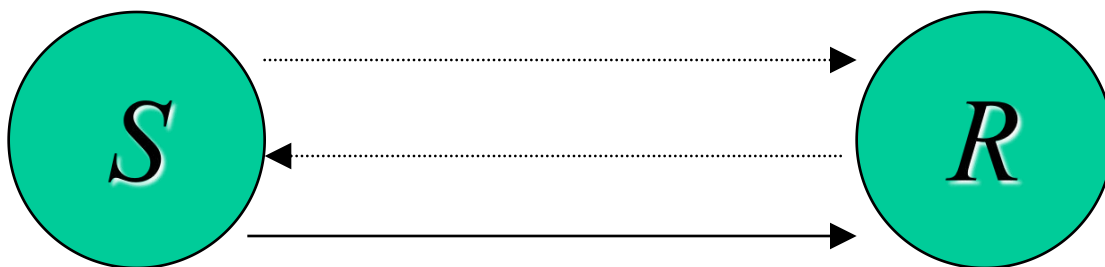
# 通信模式

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

# 同步通信模式

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

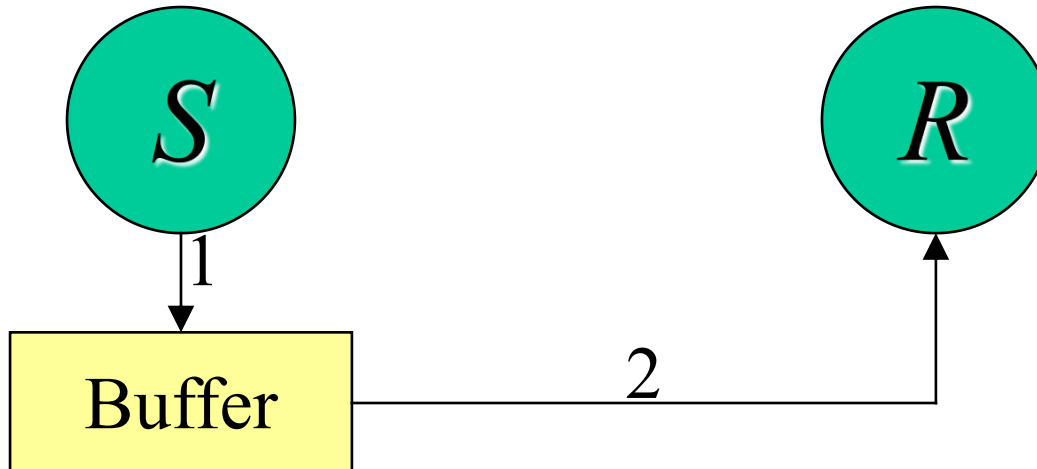
**MPI\_Ssend**(buf,count,datatype,dest,tag,comm)



# 缓冲通信模式

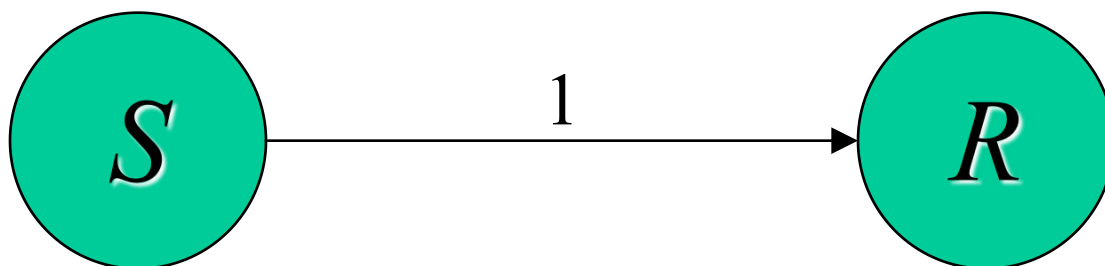
- **缓冲通信模式**：缓冲通信模式的发送不管接收操作是否已经启动都可以执行
- 但是需要用户程序事先申请一块足够大的缓冲区，通过MPI\_Buffer\_attach实现，通过MPI\_Buffer\_detach来回收申请的缓冲区

**MPI\_Bsend**(buf,count,datatype,dest,tag,comm)



# 标准通信模式

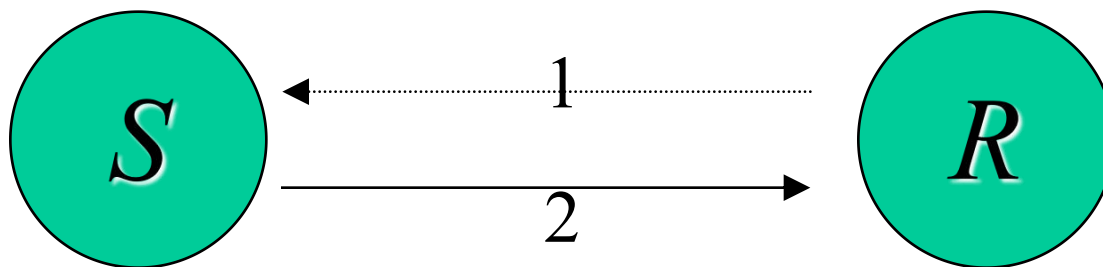
- **标准通信模式**：是否对发送的数据进行缓冲由MPI的实现来决定，而不是由用户程序来控制
- 发送可以是同步的或缓冲的，取决于实现



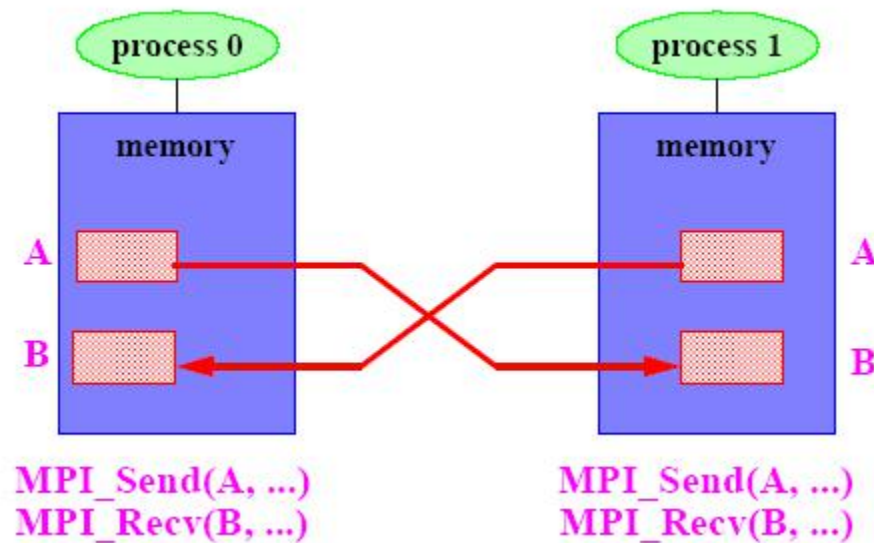
# 就绪通信模式

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

**MPI\_Rsend**(buf,count,datatype,dest,tag,comm)



# “不安全”的MPI程序

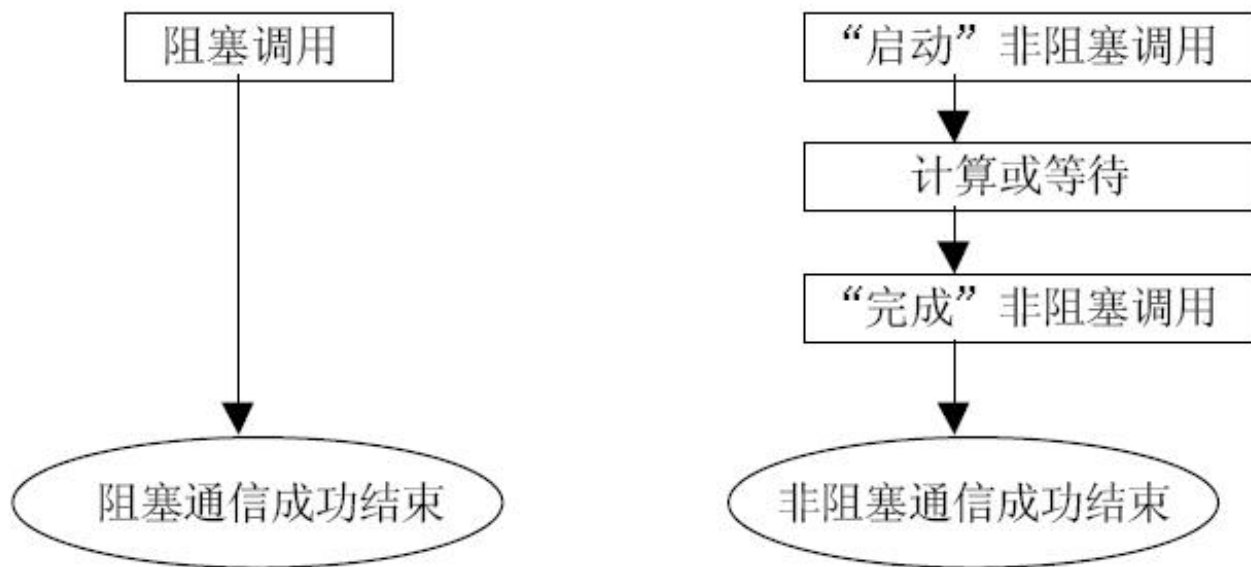


可能运行也可能不能运行，取决于是否存在可存储消息的缓存

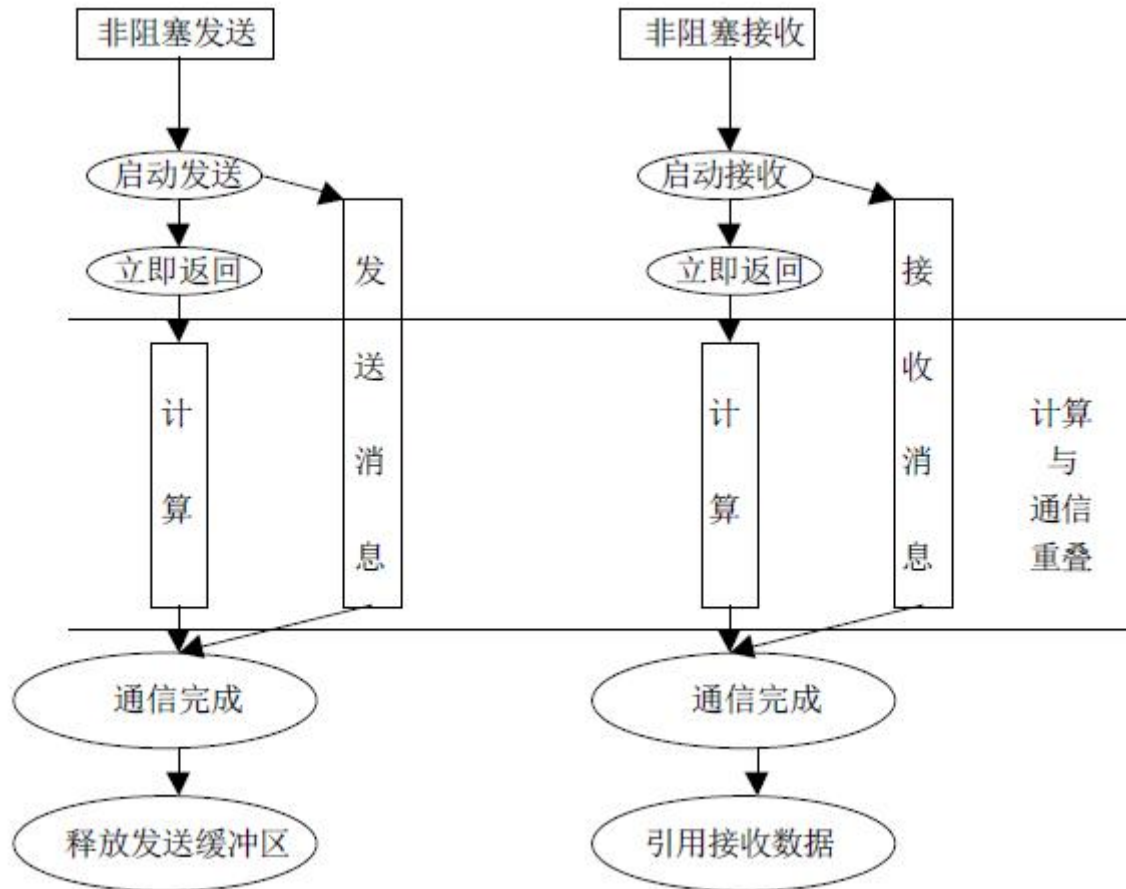
考虑非阻塞通信



# 阻塞通信与非阻塞通信



# 非阻塞发送和非阻塞接收



# 阻塞通信与非阻塞通信（续）

- 阻塞和非阻塞通信的主要区别在于返回后的资源可用性
- 阻塞通信返回的条件：
  - 通信操作已经完成，及消息已经发送或接收
  - 调用的缓冲区可用。若是发送操作，则该缓冲区可以被其它的操作更新；若是接收操作，该缓冲区的数据已经完整，可以被正确引用
- 在阻塞通信的情况下，通信还没有结束的时候，处理器只能等待，浪费了计算资源
- 非阻塞通信的好处：
  - 没有死锁—保证正确性
  - 数据可以并发地传输—提高性能

# 非阻塞通信

- 将通信操作分成两个部分
  - 第一部分初始化操作，它不阻塞
  - 第二部分等待操作完成

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
Process 0:
    MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request)
    MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
    MPI_Wait(&request, &status)
Process 1:
    MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request)
    MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
    MPI_Wait(&request, &status)
```

# 非阻塞模式的安全

非阻塞模式下, 强制进程等待直到安全时再继续执行

## Process P:

**M=10;**  
**send M to Q;**  
*do some computation which  
does not change M;*  
**wait for M to be sent;**  
**M=20;**

## Process Q:

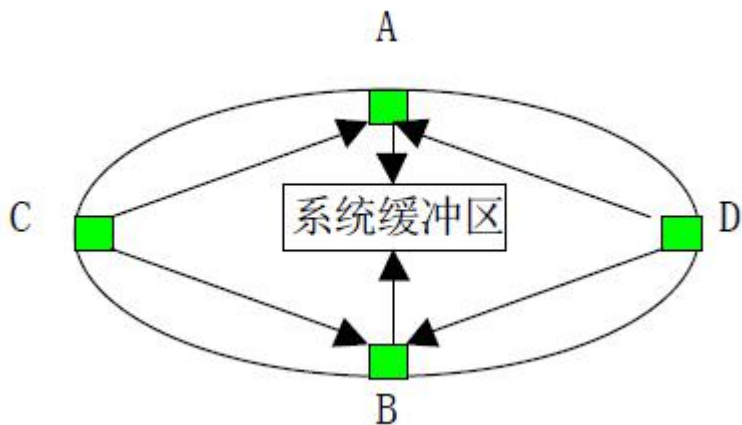
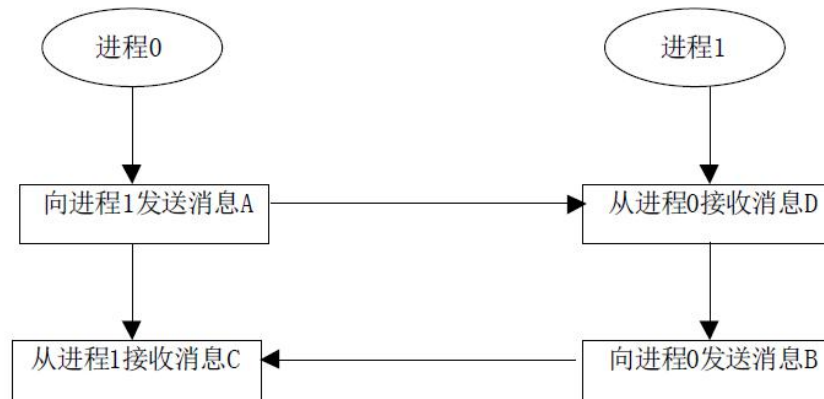
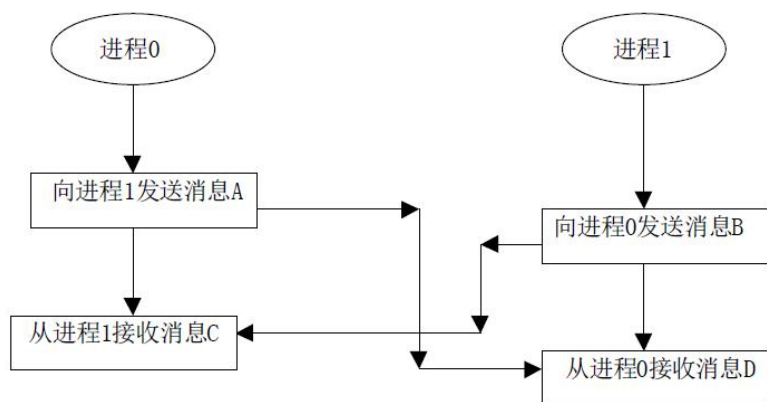
**S=-100;**  
**receive S from P;**  
*do some computation which  
does not use S;*  
**wait for S to be received;**  
**X=S+1;**

- 非阻塞模式本身也会带来一些额外开销:
  - 作为临时缓冲区用的内存空间
  - 分配缓冲区的操作
  - 将消息拷入和拷出临时缓冲区
  - 执行一个额外的检测和等待函数

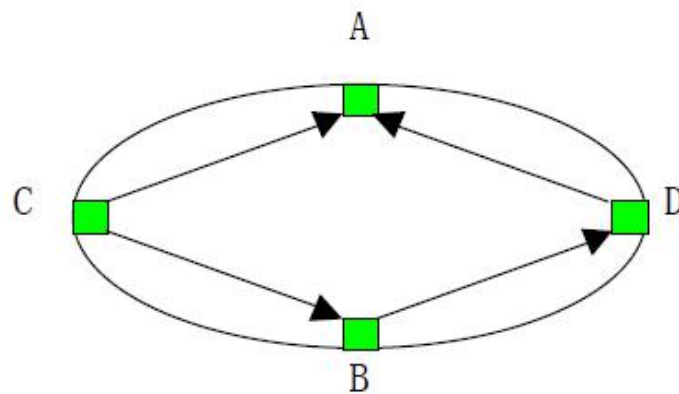
# MPI非阻塞通信的操作类型

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

# 调整通信次序



- 不安全的通信调用次序



- 安全的通信调用次序

# 捆绑发送和接收操作

- 捆绑发送和接收操作可以在一条MPI语句中同时实现向其它进程的数据发送和从其它进程接收数据操作
- **MPI\_SENDRECV**(**sendbuf**, **sendcount**, **sendtype**, **dest**, **sendtag**, **recvbuf**, **recvcount**, **recvtype**, **source**, **recvtag**, **comm**, **status**)
- 在语义上等同于一个发送操作和一个接收操作的结合
- 该操作由通信系统来实现系统会优化通信次序从而有效地避免不合理的通信次序，最大限度避免死锁的产生



# 例子： Cannon算法的初始对准

```
void init_alignment()
{
    /*将A中坐标为(i,j)的分块A(i,j)向左循环移动i步*/
    MPI_Sendrecv(a, dl2, MPI_FLOAT,
        get_index(my_row,my_col-my_row,sp), 1,
        tmp_a, dl2, MPI_FLOAT,
        get_index(my_row,my_col+my_row,sp), 1,
        MPI_COMM_WORLD, &status);
    memcpy(a, tmp_a, dl2 * sizeof(float) );

    /*将B中坐标为(i,j)的分块B(i,j)向上循环移动j步*/
    MPI_Sendrecv(b, dl2, MPI_FLOAT,
        get_index(my_row-my_col,my_col,sp), 1,
        tmp_b, dl2, MPI_FLOAT,
        get_index(my_row+my_col,my_col,sp), 1,
        MPI_COMM_WORLD, &status);
    memcpy(b, tmp_b, dl2 * sizeof(float) );
}
```

# 讨论：避免死锁的方法

- 调整send和receive的顺序：一个进程运行send，而另一进程运行 receive
- 用非阻塞函数：先运行非阻塞receive，然后测试是否结束
- 用MPI\_Sendrecv：将两个操作放在一个函数中
- 用缓冲模式：发送操作可以在数据拷贝到用户缓冲后执行

# MPI的点对点通信操作

MPI 原语	阻塞	非阻塞
Standard Send	MPI_Send	MPI_Isend
Synchronous Send	MPI_Ssend	MPI_Issend
Buffered Send	MPI_Bsend	MPI_Ibsend
Ready Send	MPI_Rsend	MPI_Irsend
Receive	MPI_Recv	MPI_Irecv
Completion Check	MPI_Wait	MPI_Test

# 总结：通信模式的比较

通信模式	优点	缺点
Synchronous	安全，容易移植 SEND/RECV的顺序不重要 无需buffer	导致潜在的同步开销
Ready	开销最小，无需SEND/RECV的握手	RECV必须先行于 SEND
Buffered	SEND和RECV松耦合 无需SEND的同步开销 SEND/RECV的循序不重要 程序员可以控制buffer大小	拷贝到buffer需要额外的系统开销
Standard	大部分情形下适用	可能不适合你的程序

**Tips: Try Standard-nonBlocking as possible**

# 内容概要

---

- 引言
- MPI消息
- 点对点通信
- 群集通信
- 总结

# 群集通信

- **群集通信**（**Collective Communications**）是一个进程组中的所有进程都参加的全局通信操作
- 群集通信一般实现三个功能：通信、聚集和同步
  - 通信功能主要完成组内数据的传输
  - 聚集功能在通信的基础上对给定的数据完成一定的操作
  - 同步功能实现组内所有进程在执行进度上取得一致

# 群集通信的类型

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

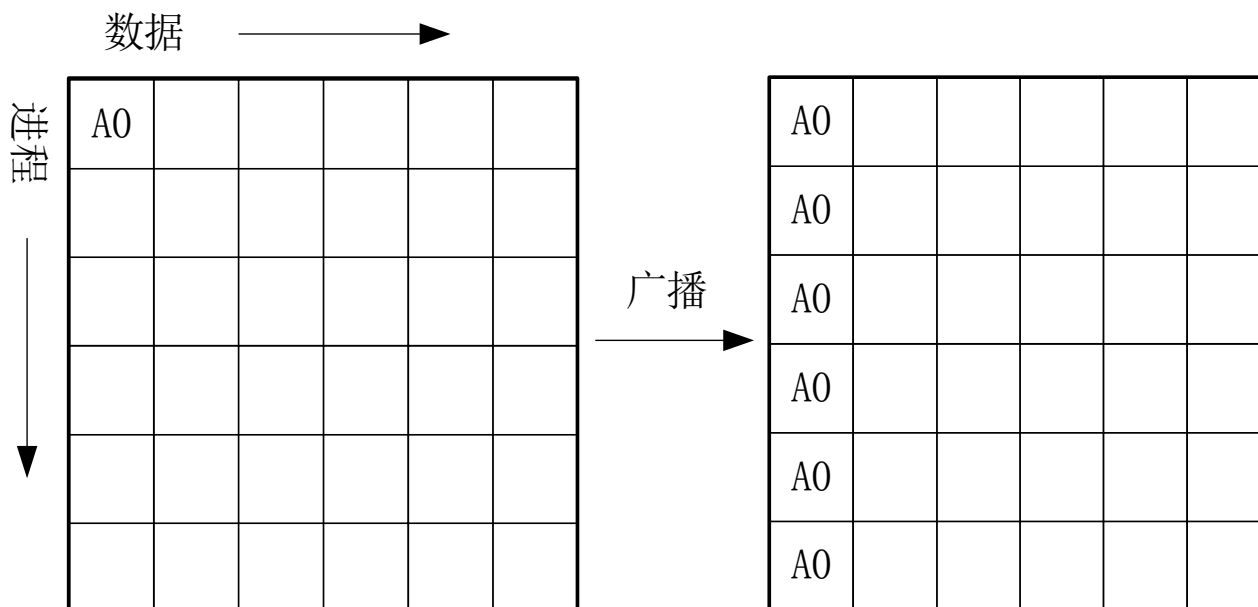
# 群集通信的函数

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



# 广播

- **广播**是一对多通信的典型例子，其调用格式如下：
  - Ø `MPI_Bcast(Address, Count, Datatype, Root, Comm)`
  - Ø 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
  - Ø 消息的内容由三元组**<Address, Count, Datatype>**标识
  - Ø 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。对它其它进程来说，这个三元组只定义了接收缓冲



# Bcast的例子

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv )
{
    int rank, value;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

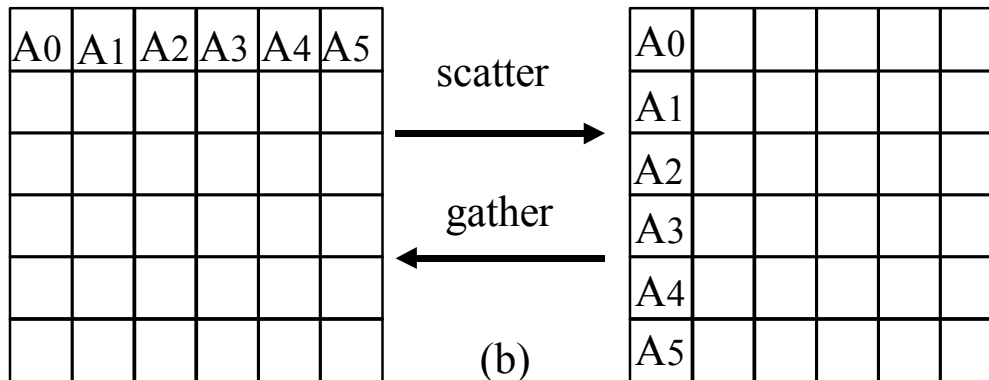
    do {
        if (rank == 0) /*进程0读入需要广播的数据*/
            scanf( "%d", &value );
        MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD ); /*将该数据广播出去*/
        Printf("Process %d got %d\n", rank, value ); /*各进程打印收到的数据*/
    } while (value >= 0);

    MPI_Finalize( );
    return 0;
}
```

# 散播 (Scatter) 和收集 (Gather)

**MPI\_Scatter** (SendAddress, SendCount, SendDatatype, RecvAddress, RecvCount, RecvDatatype, Root, Comm)

**MPI\_Gather** (SendAddress, SendCount, SendDatatype, RecvAddress, RecvCount, RecvDatatype, Root, Comm)



## MPI\_Scatter

- Root进程发送给所有n个进程发送一个不同的消息, 包括自己.
- 这n个消息在Root进程的发送缓冲区中按标号的顺序有序地存放. 每个接收缓冲由三元组 (RecvAddress, RecvCount, RecvDatatype) 标识
- 非Root进程忽略发送缓冲. 对Root进程, 发送缓冲由三元组 (SendAddress, SendCount, SendDatatype) 标识.

## MPI\_Gather

- Root进程从n个进程的每一个接收各个进程(包括它自己)的消息.
- 这n个消息的连接按序号rank进行, 存放在Root进程的接收缓冲中. 每个发送缓冲由三元组 (SendAddress, SendCount, SendDatatype) 标识
- 所有非Root进程忽略接收缓冲. 对Root进程, 发送缓冲由三元组 (RecvAddress, RecvCount, RecvDatatype) 标识.

# Gather的例子

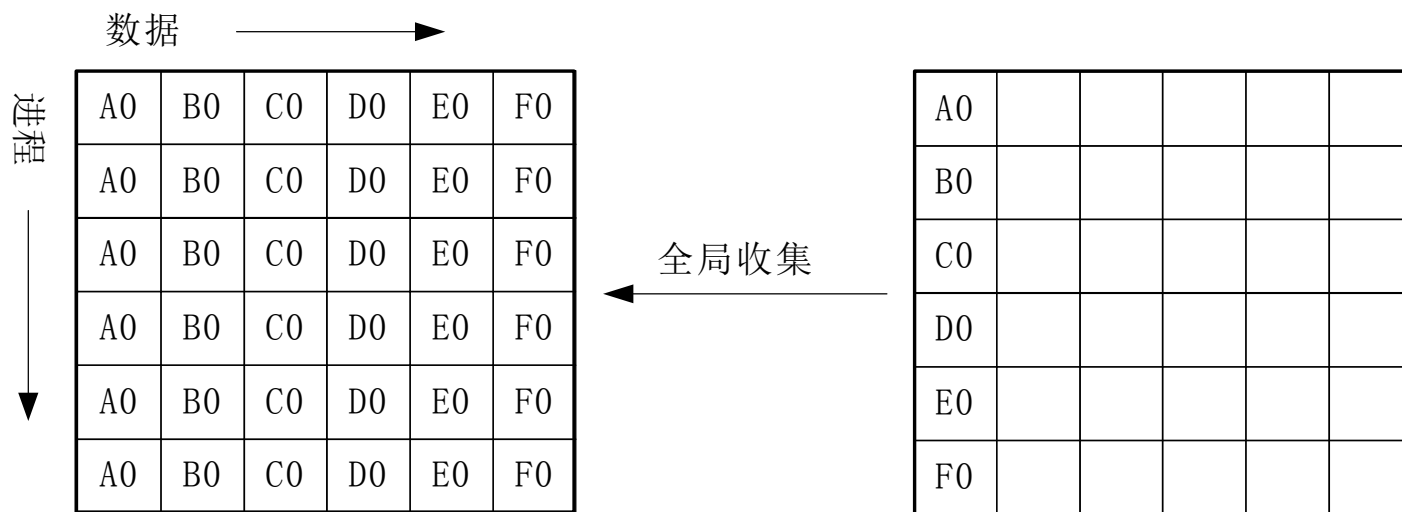
```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf;
.....

MPI_Comm_size(comm, &gsize);
rbuf = (int *) malloc(gsize * 100 * sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

- 实现从进程组中的每个进程收集100个整型数送给根进程

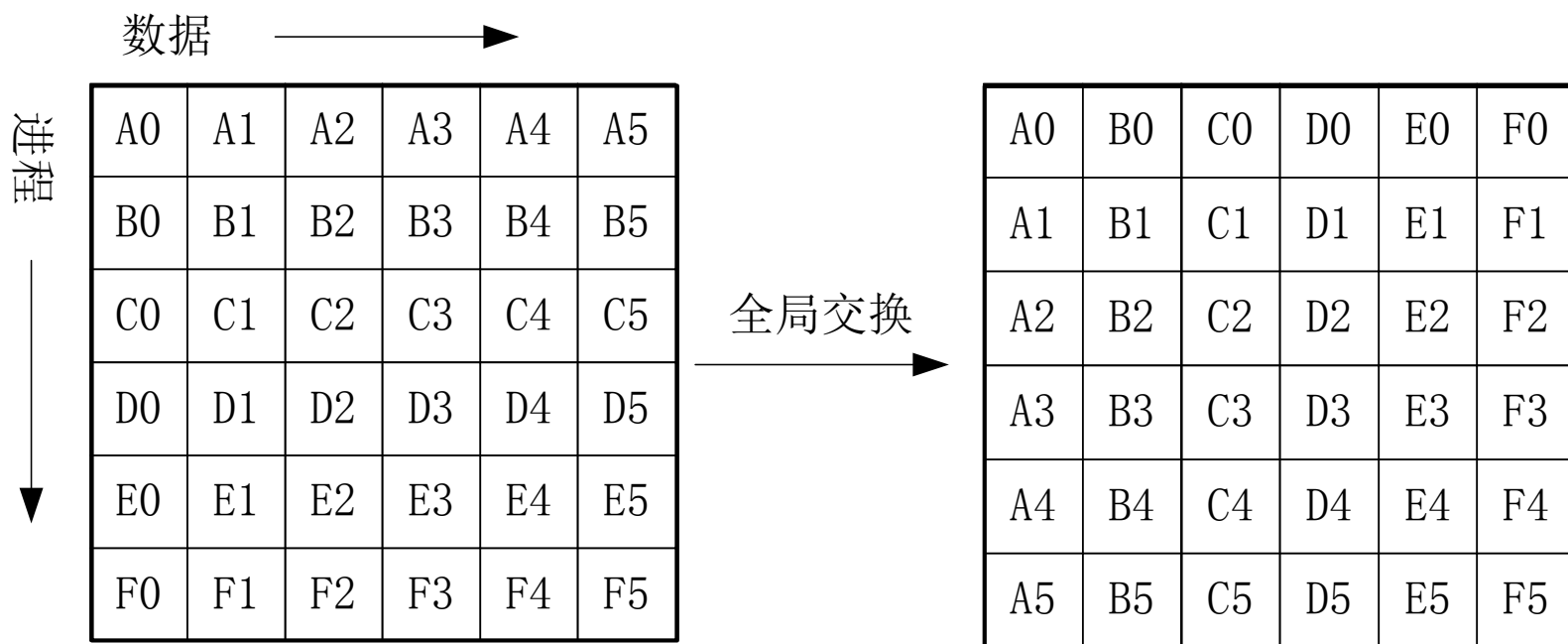
# 全局收集

- **全局收集**多对多通信的典型例子，其调用格式如下：
  - `MPI_Allgather(SendAddress, SendCount, SendDatatype, RecvAddress, RecvCount, RecvDatatype, Comm)`
  - **Allgather**操作相当于每个进程都作为**ROOT**进程执行了一次**Gather**调用，即每一个进程都按照**Gather**的方式收集来自所有进程（包括自己）的数据。



# 全局交换

- 全局交换也是一个多对多操作，其调用格式如下：
  - **MPI\_Alltoall(SendAddress, SendCount, SendDatatype, RecvAddress, RecvCount, RecvDatatype, Comm)**



# 全局交换

- 全局交换的特点

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

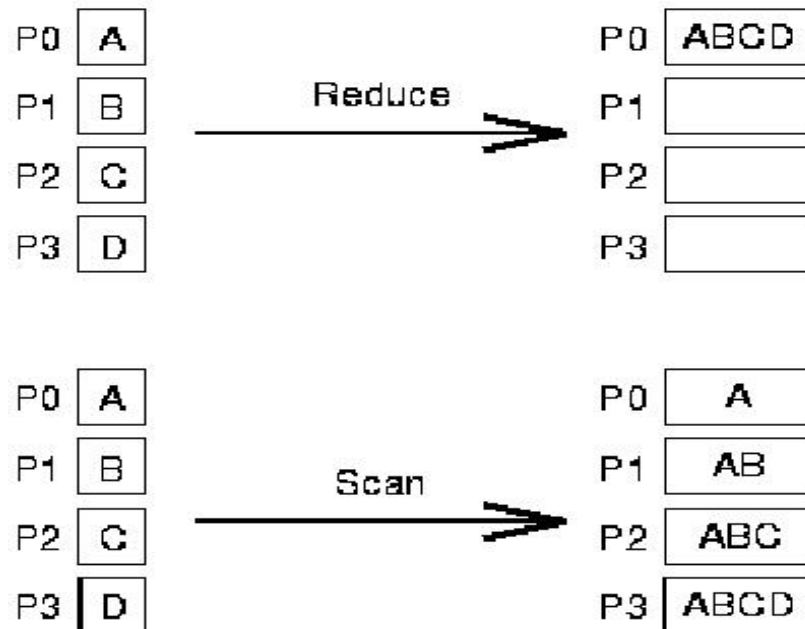
# 同步

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



# 聚合操作

- 群集通信的聚合功能使得MPI进行通信的同时完成一定的计算
- MPI提供了两种类型的聚合操作：**归约**（reduction）和**扫描**（scan）



# 归约

- 归约的调用格式如下：
  - **MPI\_Reduce(SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm)**
- 归约的特点
  - 归约操作对每个进程的发送缓冲区（SendAddress）中的数据按给定的操作进行运算，并将最终结果存放在Root进程的接收缓冲区（RecvAddress）中
  - 参与计算操作的数据项的数据类型在Datatype域中定义，归约操作由Op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

# MPI预定义的归约操作

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

# 归约的例子：点积

```
/* distribute two vectors over all processes such that
   processor 0 has elements 0...99
   processor 1 has elements 100...199
   processor 2 has elements 200...299
   etc.
*/

double dotprod(double a[100], double b[100])
{
    double gresult = lresult = 0.0;
    integer i;
    /* compute local dot product */
    for (i = 0; i < 100; i++) lresult += a[i]*b[i];
    MPI_Allreduce(&lresult, &gresult, 1, MPI_DOUBLE,
                  MPI_SUM, MPI_COMM_WORLD);
    return(gresult);
}
```

# 扫描

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

# 可变长度和位置的群集函数

- 允许消息的大小和位置是可变的
  - **MPI\_Scatterv, MPI\_Gatherv, MPI\_Allgatherv, MPI\_Alltoallv**
- 优点
  - 降低拷贝数据到临时缓冲区的要求
  - 代码更简练
  - 实现机制更为优化
- 缺点：效率不如长度/位置固定的函数

# Scatterv 和 Gatherv

```
int MPI_Scatterv(void *SendAddress, int *SendCount, int  
    *Displs, MPI_Datatype SendDatatype, void  
    *RecvAddress, int RecvCount, MPI_Datatype  
    RecvDatatype, int Root, MPI_Comm Comm)
```

```
int MPI_Gatherv(void *SendAddress, int SendCount,  
    MPI_Datatype SendDatatype, void *RecvAddress, int  
    *RecvCount, int *Displs, MPI_Datatype RecvDatatype,  
    int Root, MPI_Comm Comm)
```

- **\*SendCount 和 \*RecvCount:** 整数数组，包含每个进程要发送和接受的元素个数
- **\*Displs:** 整数数组，说明要发送的数据到buffer的起始位置的偏移量

# 群集通信的特点

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



# 内容概要

---

- 引言
- **MPI消息**
- 点对点通信
- 群集通信
- **总结**

# 例子：计算 PI

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
```

$$\pi \approx \sum_{i=1}^n \frac{4}{\left(\frac{i-0.5}{n}\right)^2}$$

## 例子：计算 PI（续）

```
h    = 1.0 / (double) n;
sum  = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);

if (myid == 0)
    printf("pi is approximately %.16f, Error is
           %.16f\n", pi, fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
}
```

# 例子：矩阵乘法Cannon

//输入:  $A_{n \times n}$ ,  $B_{n \times n}$ ; 输出:  $C_{n \times n}$

Begin

(1) for  $k=0$  to  $p/2-1$  do

for all  $P_{i,j}$  par-do

(i) if  $i > k$  then

$A_{i,j} \leftarrow A_{i,(j+1) \bmod \sqrt{p}}$   
endif

(ii) if  $j > k$  then

$B_{i,j} \leftarrow B_{(i+1) \bmod \sqrt{p}, j}$   
endif

endfor

endfor

(2) for all  $P_{i,j}$  par-do  $C_{i,j} = 0$  endfor

## 算法9.5

(3) for  $k=0$  to  $p^{1/2}-1$  do

for all  $P_{i,j}$  par-do

(i)  $C_{i,j} = C_{i,j} + A_{i,j} B_{i,j}$

(ii)  $A_{i,j} \leftarrow A_{i,(j+1) \bmod \sqrt{p}}$

(iii)  $B_{i,j} \leftarrow B_{(i+1) \bmod \sqrt{p}, j}$

endfor

endfor

End

Source: Cannon.c

# MPI和OpenMP

- 都是 SPMD模型
- 消息传递与共享存储
- 进程与线程
- **MPI**
  - 可移植性好，对硬件和编译器无特殊要求
  - All-or-nothing并行化机制
  - 没有共享，需要考虑分布式数据结构
  - 数据的划分操作比较复杂
- **OpenMP**
  - 需要共享存储的多处理器系统的支持
  - 支持增量并行化
  - 共享数据，需要考虑变量的共享与私有性
  - 编译制导方式比较简单

# 课程小结

---

- **MPI简介**
- **MPI的六个基本函数**
- **MPI的基本概念和高级特性**
  - **MPI消息，数据类型，通信域**
  - **通信模式，点对点通信，群集通信等**
- **MPI编程例子**
- **MPI和OpenMP的比较**

# 推荐网站和读物

---

- 《并行计算》（第三版）
  - 第15章：分布式存储系统并行编程
- 都志辉等，《高性能计算并行编程技术—MPI并行程序设计》，清华大学出版社，2001
- **The MPI Standard: <http://www.mpi-forum.org>**

# 下一讲

---

- **GPU与CUDA编程**