并行计算--

MPI (Message Passing Interface)

汤善江

Outline

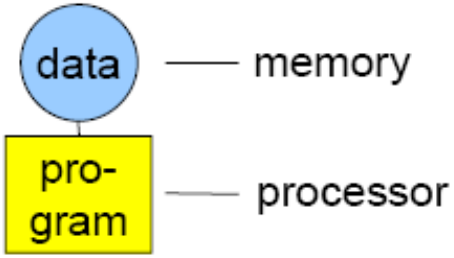
- MPI概述
- 点到点通信/组通信
- 自定义数据类型

Outline

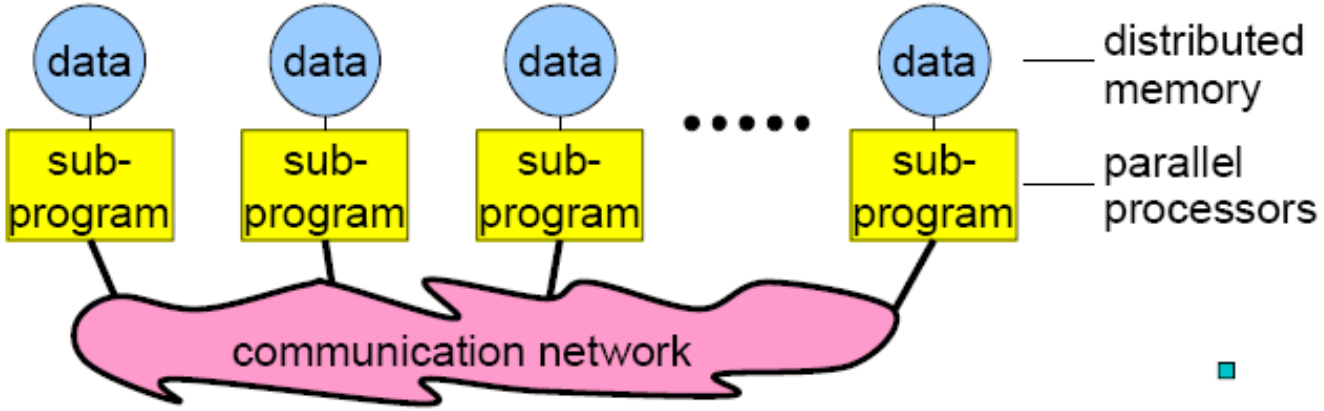
- **MPI概述**
- 点到点通信/组通信
- 自定义数据类型

MPI概述

- 串行程序



- MPI并行程序



MPI (Message passing interface)

- 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。 MPI同时也是一种消息传递编程模型，并成为这种编程模型的代表和事实上的标准。
  - 迄今为止所有的并行计算机制造商都提供对MPI的支持，可以在网上免费得到MPI在不同并行计算机上的实现。
- MPI的实现是一个库，而不是一门语言。
  - 可以把FORTRAN+MPI或C+MPI 看作是一种在原来串行语言基础之上扩展后得到的并行语言。

MPI程序示例: Hello World!

Fortran

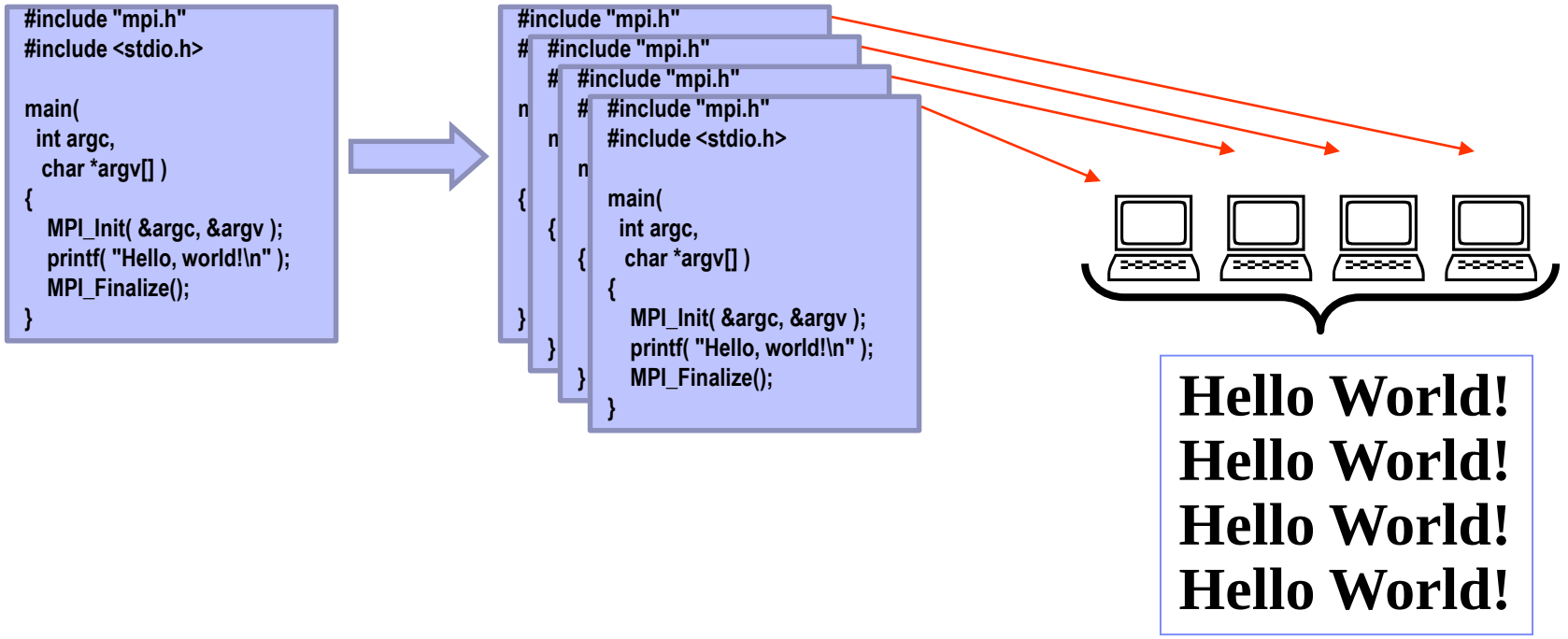
```
PROGRAM hello
  INCLUDE ‘mpif.h‘
  INTEGER err
  CALL MPI_INIT(err)
  PRINT *, “hello world!”
  CALL MPI_FINALIZE(err)

END
```

C

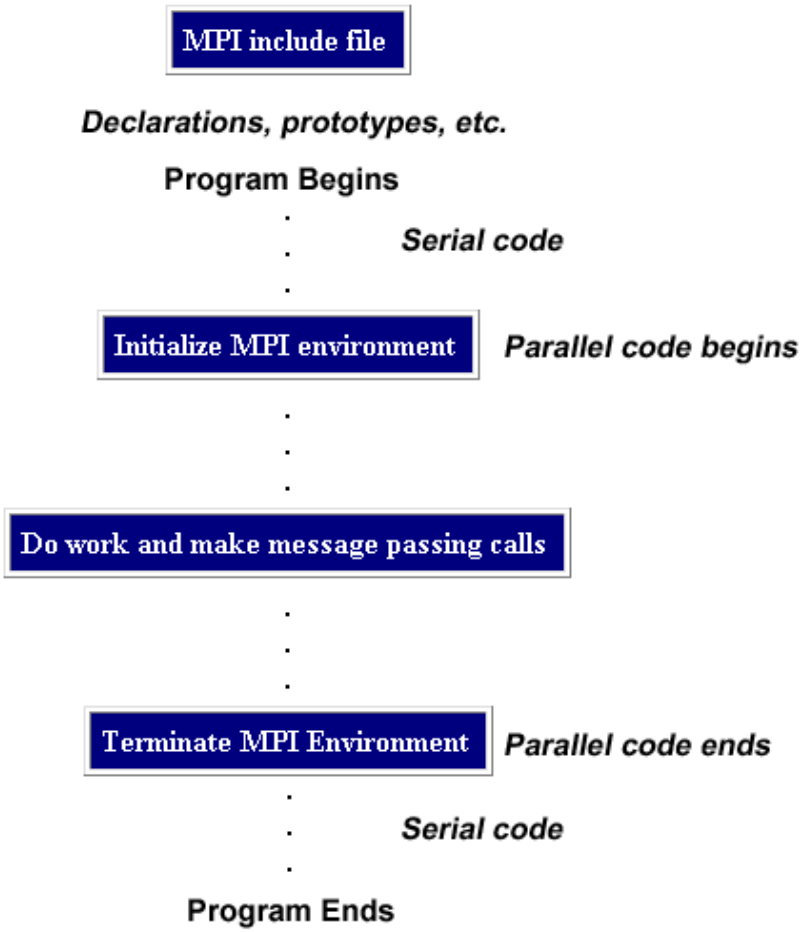
```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char * argv[])
{
  int err;

  err = MPI_Init(&argc, &argv);
  printf( “Hello world!\n” );
  err = MPI_Finalize();
}
```

MPI程序的执行

- SPMD: Single Program Multiple Data(MIMD)



MPI程序结构


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work and make message passing calls
Terminate MPI Environment
Parallel code ends
Serial code
Program Ends

MPI 的六个基本接口

- 开始与结束
  - MPI_INIT
  - MPI_FINALIZE
- 进程身份标识
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
- 发送与接收消息
  - MPI_SEND
  - MPI_RECV

MPI 程序的开始与结束

- MPI代码开始之前必须进行如下调用：

```
MPI_Init(&argc, &argv);
```

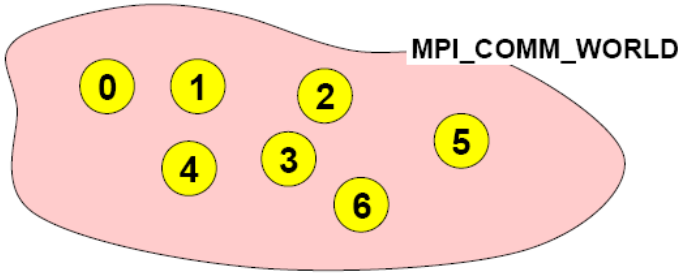
  - MPI系统将通过argc,argv得到命令行参数

- MPI代码的最后一行必须是：

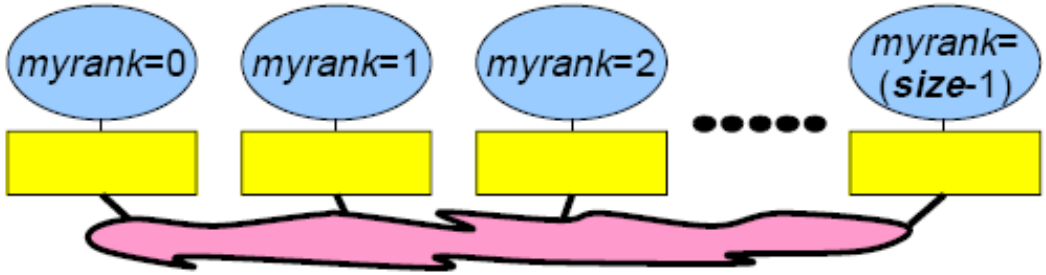
```
MPI_Finalize();
```

  - 如果没有此行，MPI程序将不会终止。

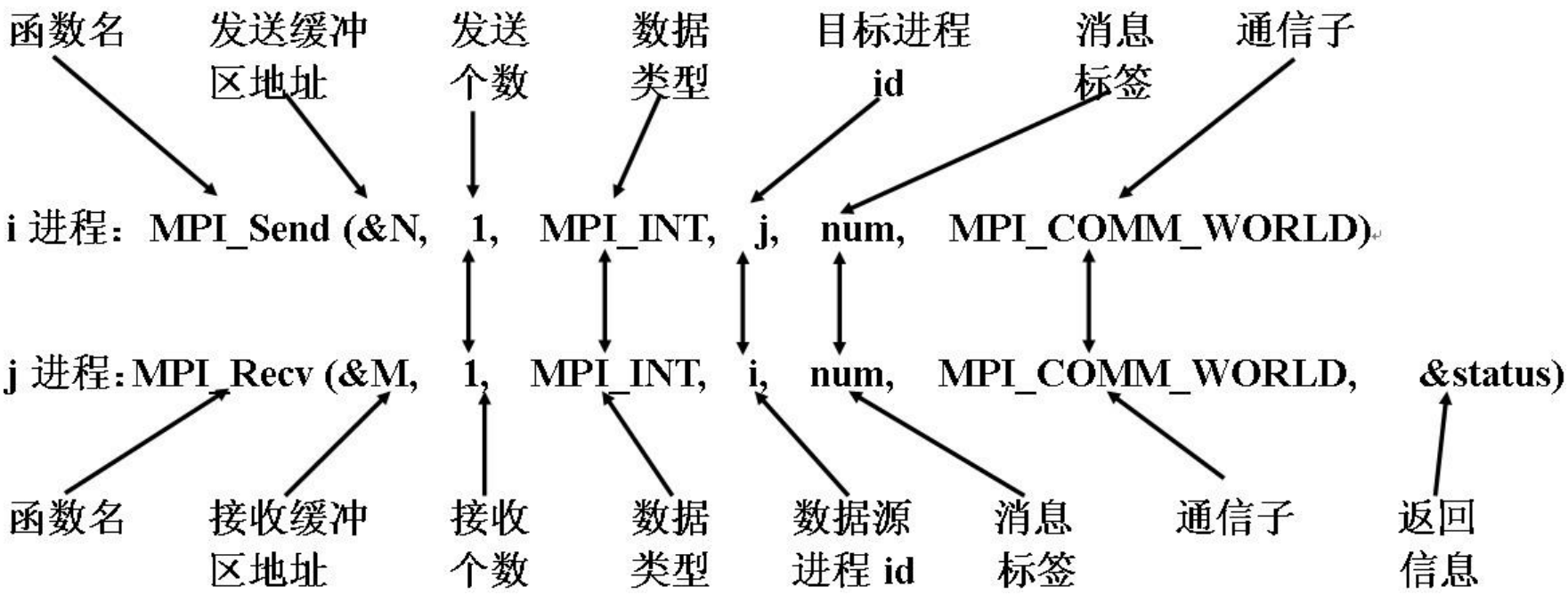
MPI进程身份标识



- 通信域
  - 缺省的通信域为MPI_COMM_WORLD
- MPI_Comm_size(MPI_COMM_WORLD, &size)
  - 获得缺省通信域内所有进程数目，赋值给size
- MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
  - 获得进程在缺省通信域的编号，赋值给myrank



发送和接收消息


函数名
发送缓冲区地址
发送个数
数据类型
目标进程id
消息标签
通信子
i 进程： MPI_Send (&N, 1, MPI_INT, j, num, MPI_COMM_WORLD)
j 进程: MPI_Recv (&M, 1, MPI_INT, i, num, MPI_COMM_WORLD, &status)
函数名
接收缓冲区地址
接收个数
数据类型
数据源进程 id
消息标签
通信子
返回信息

一个计算∑foo(i)的MPI SPMD消息传递程序

```
#include "mpi.h"
int foo(i)
int i;
{...}
main(argc, argv)
int argc;
char* argv[]
{
int i, tmp, sum=0, group_size, my_rank, N;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &group_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0) {
   printf("Enter N:");
   scanf("%d",&N);
   for (i=1;i<group_size;i++)
      MPI_Send(&N, 1, MPI_INT, i, i, MPI_COMM_WORLD);
   for (i=my_rank;i<N;i=i+group_size) sum=sum+tmp;
   for (i=1;i<group_size;i++) {
      MPI_Recv(&tmp,1,MPI_INT,i,i,MPI_COMM_WORLD,&status);
     sum=sum+tmp;
    }
   printf("\n The result = %d", sum);
}
else {
   MPI_Recv(&N,1,MPI_INT,0,i,MPI_COMM_WORLD,&status);
   for (i-my_rank;i<N;i=i+group_size) sum=sum+foo(i);
   MPI_Send(&sum,1,MPI_INT,0,i,MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

初始化MPI环境

获得总进程数

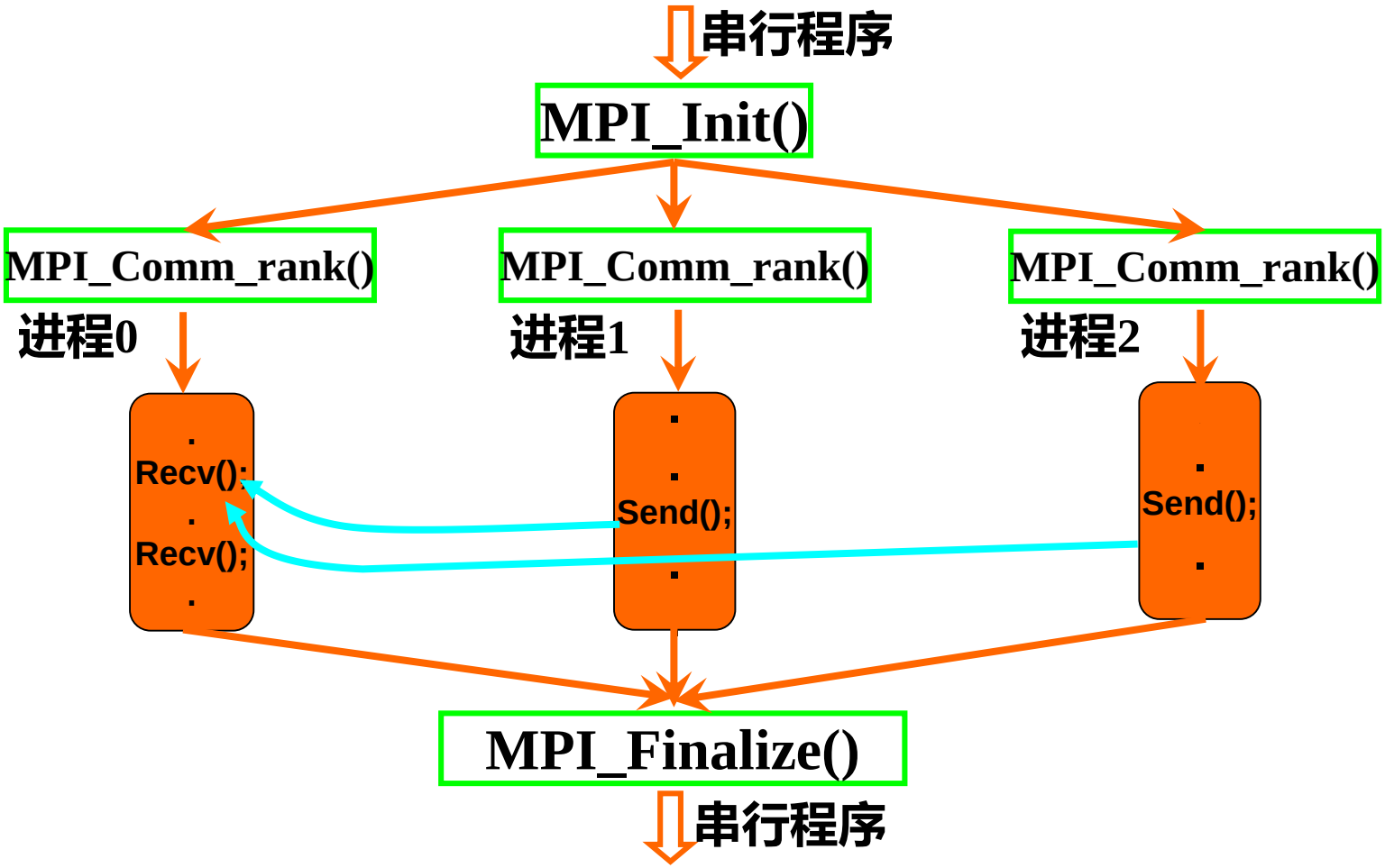
得到每个进程在组中的编号

发送消息

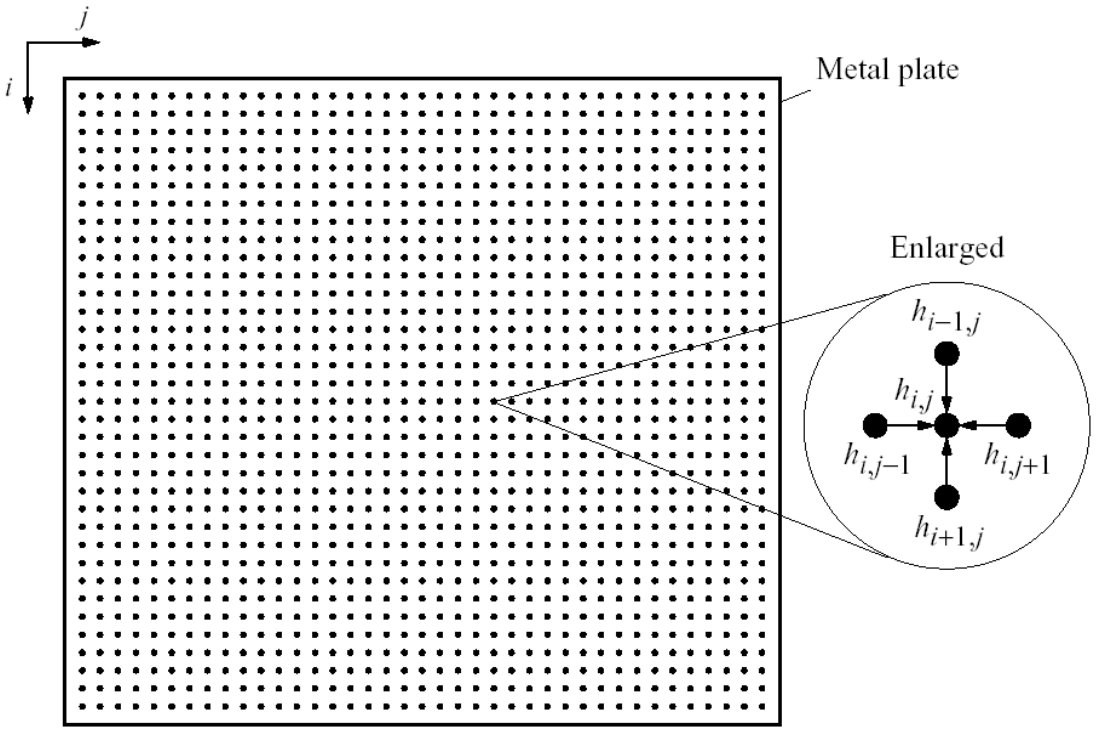
接收消息

终止MPI环境

消息传递的过程


串行程序
MPI_Init()
MPI_Comm_rank()
MPI_Comm_rank()
MPI_Comm_rank()
进程0
进程1
进程2
Recv();
Recv();
Send();
Send();
MPI_Finalize()
串行程序

问题：Jacobi迭代



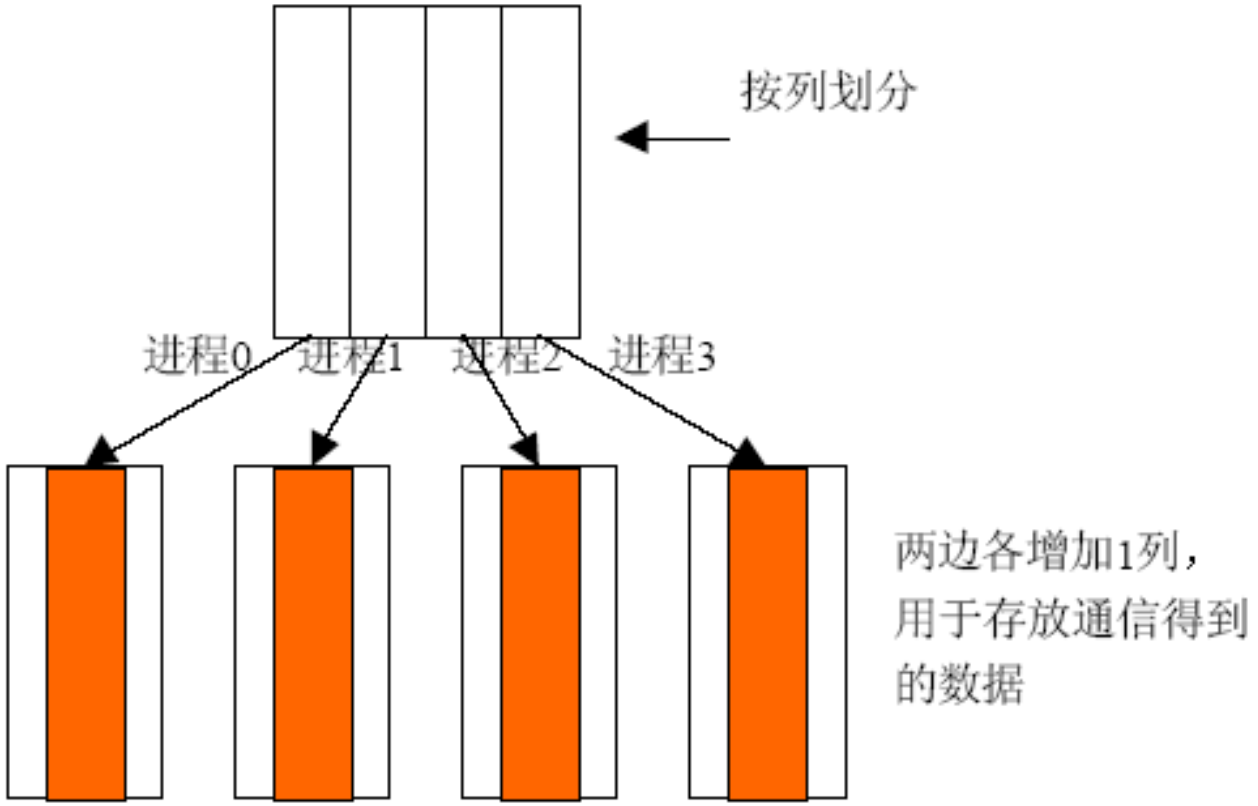
$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

Jacobi迭代

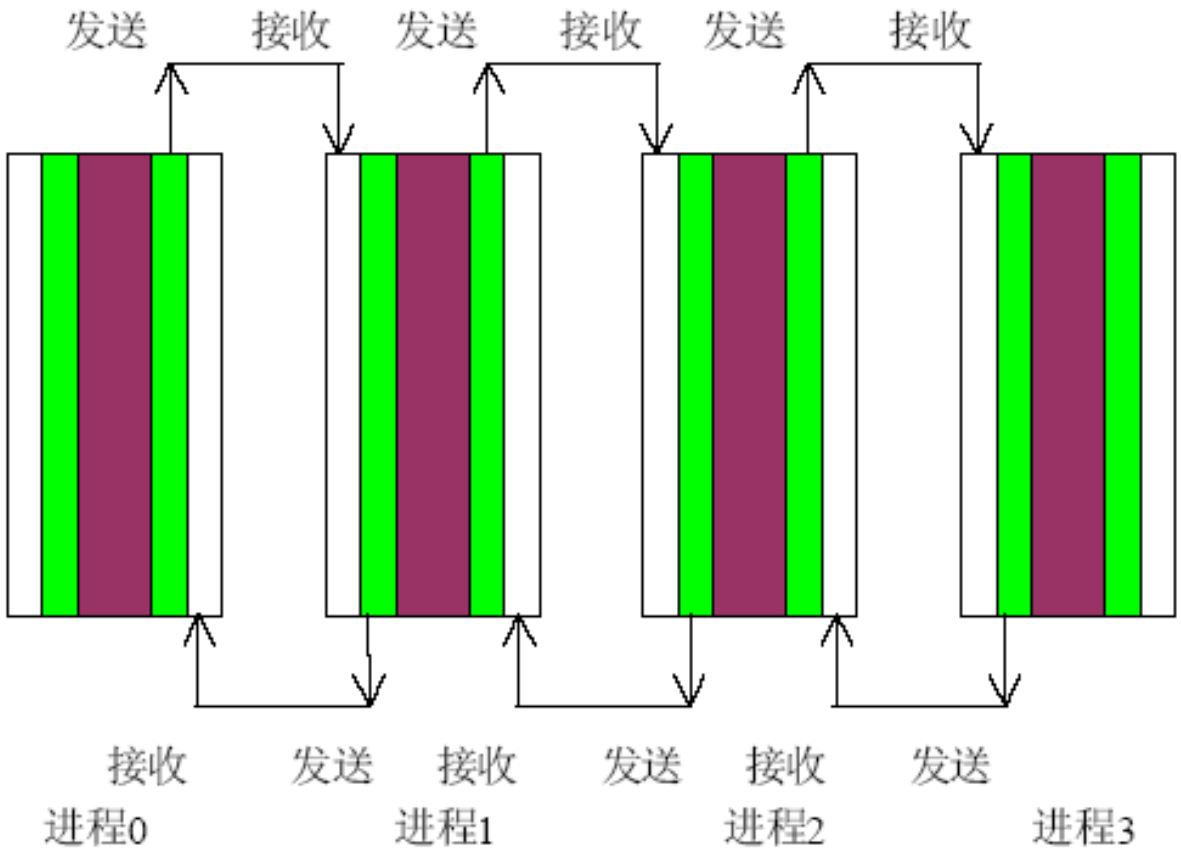
- 伪代码描述：

```
…
REAL A(N+1,N+1), B(N+1,N+1)
…
DO K=1,STEP
  DO J=1,N
        DO I=1,N
              B(I,J)=0.25*(A(I-1,J)+A(I+1,J)+A(I,J+1)+A(I,J-1))
        END DO
  END DO
  DO J=1,N
        DO I=1,N
              A(I,J)=B(I,J)
        END DO
  END DO
```

Jacobi迭代：数据划分



Jacobi迭代：通信

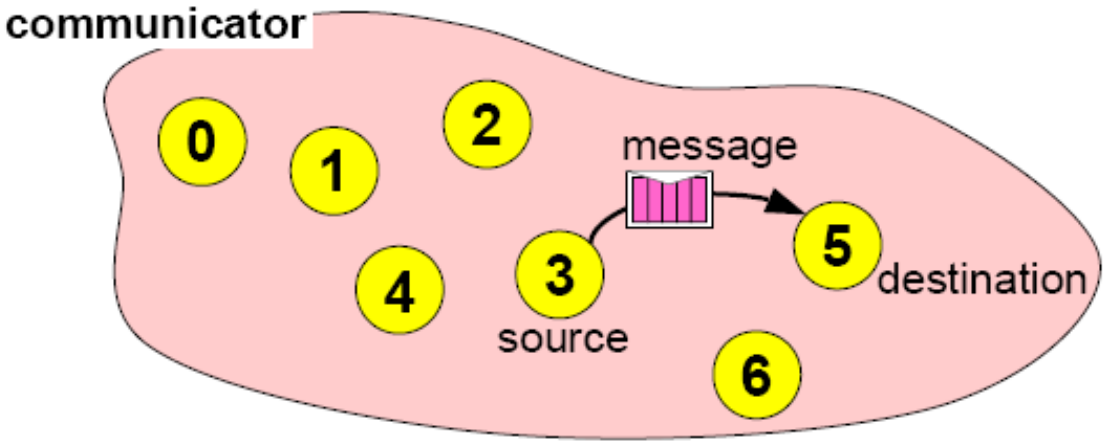


Outline

- MPI概述
- **点到点通信/组通信**
- 自定义数据类型

点到点通信

- 对于某一消息
  - 唯一发送进程
  - 唯一接收进程



MPI_Send

MPI_Send(buffer, count, datatype, destination, tag, communicator)

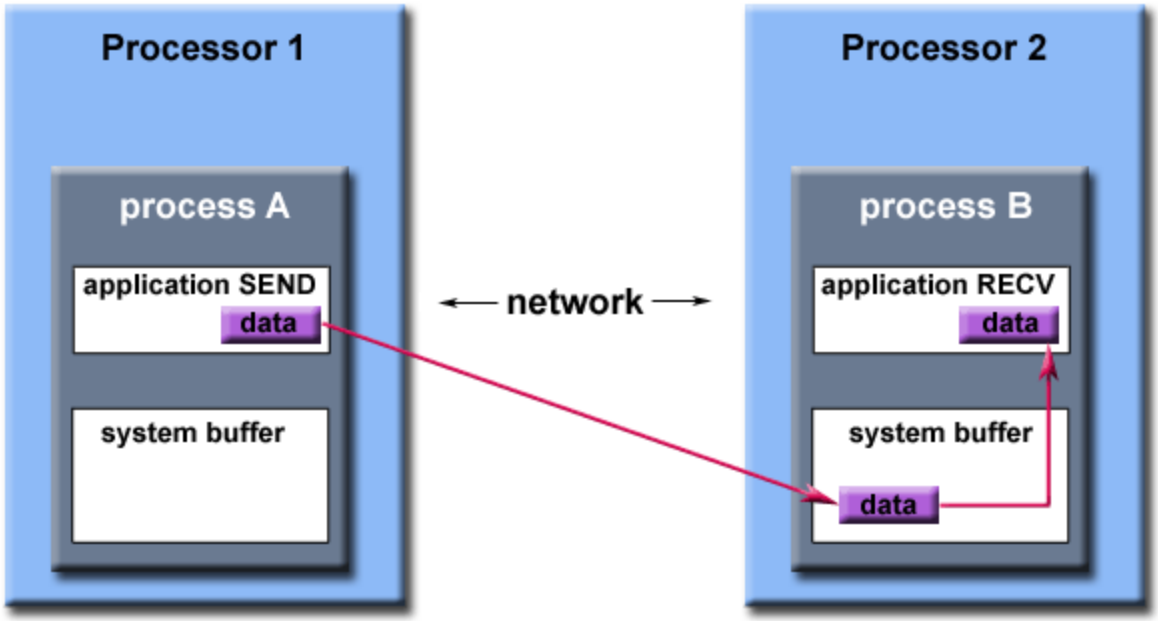
- MPI_Send(&N, 1, MPI_INT, i, i , MPI_COMM_WORLD);
- 第一个参数指明消息缓存的起始地址，即存放要发送的数据信息。
- 第二个参数指明消息中给定的数据类型有多少项，数据类型由第三个参数给定。
- 数据类型要么是基本数据类型，要么是导出数据类型，后者由用户生成指定一个可能是由混合数据类型组成的非连续数据项。
- 第四个参数是目的进程的标识符(进程编号)。
- 第五个是消息标签。
- 第六个参数标识进程组和上下文，即通信域。通常，消息只在同组的进程间传送。但是MPI允许通过intercommunicators在组间通信。

MPI_Receive

MPI_Recv(address, count, datatype,source, tag, communicator, status)

- MPI_Recv(&tmp, 1, MPI_INT, i, i, PI_COMM_WORLD,&Status)
- 第一个参数指明接收消息缓冲的起始地址，即存放接收消息的内存地址。
- 第二个参数指明给定数据类型可以被接收的最大项数。
- 第三个参数指明接收的数据类型。
- 第四个参数是源进程标识符 (编号)。
- 第五个是消息标签。
- 第六个参数标识一个通信域。
- 第七个参数是一个指针, 指向一个结构：MPI_Status Status
  - 存放有关接收消息的各种信息。(Status.MPI_SOURCE, Status.MPI_TAG)
  - MPI_Get_count(&Status, MPI_INT, &C)读出实际接收到的数据项数。

消息的接收（系统缓存）



Path of a message buffered at the receiving process

标签的使用

为什么要使用消息标签(Tag)?

这段代码需要传送A的前32个字节进入X，传送B的前16个字节进入Y。但是，如果消息B尽管后发送但先到达进程Q，就会被第一个recv()接收在X中。

使用标签可以避免这个错误。

未使用标签

Process P:	Process Q:
send(A,32,Q)	recv(X, 32, P)
send(B,16,Q)	recv(Y, 16, P)

使用了标签

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

标签的使用

```
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;
      }
```

使用标签的另一个原因是可以简化对下列情形的处理。
假定有两个客户进程P和R，每个发送一个服务请求消息给服务进程Q。

```
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;
      }
```

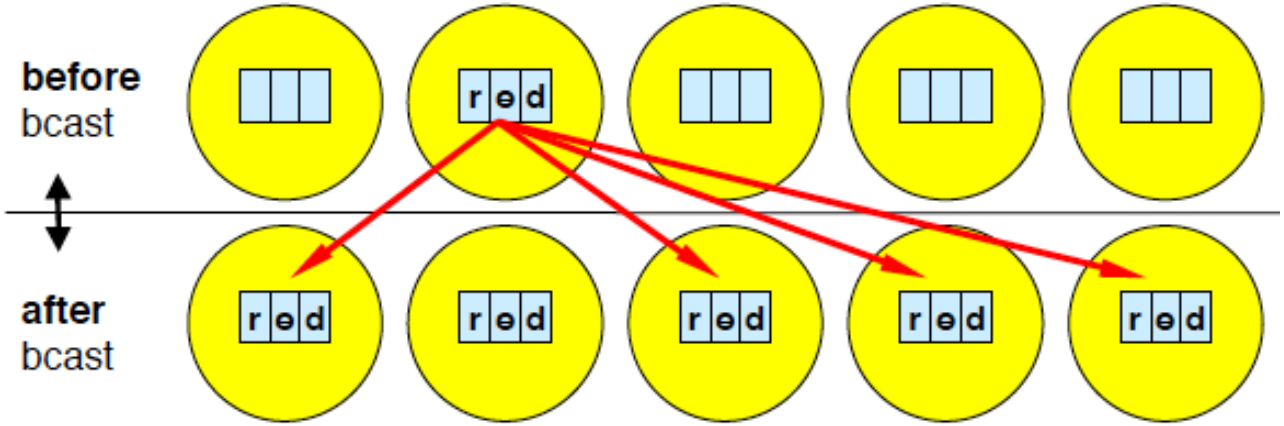
组通信

- 一到多 (Broadcast, Scatter)
- 多到一 (Reduce, Gather)
- 多到多 (Allreduce, Allgather)
- 同步 (Barrier)

广播（Broadcast）

MPI_Bcast(Address, Count, Datatype, *Root, Comm*)

- 标号为Root的进程发送相同的消息给标记为Comm的通信子中的所有进程。
- 消息的内容如同点对点通信一样由三元组(Address, 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。



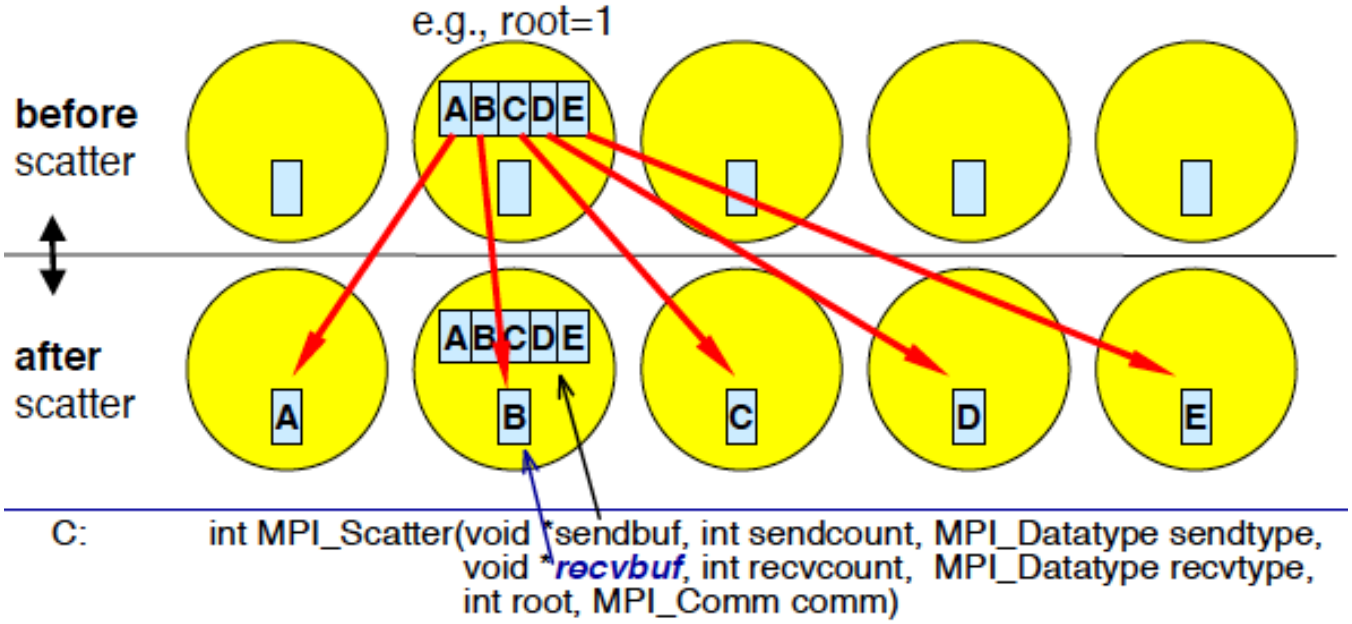
MPI_Bast

```
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;
}
```

Scatter

MPI_Scatter (SendAddress,SendCount,SendDatatype,
RecvAddress,RecvCount,RecvDatatype,Root,Comm)

- Root进程发送给所有n个进程各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_Scatter

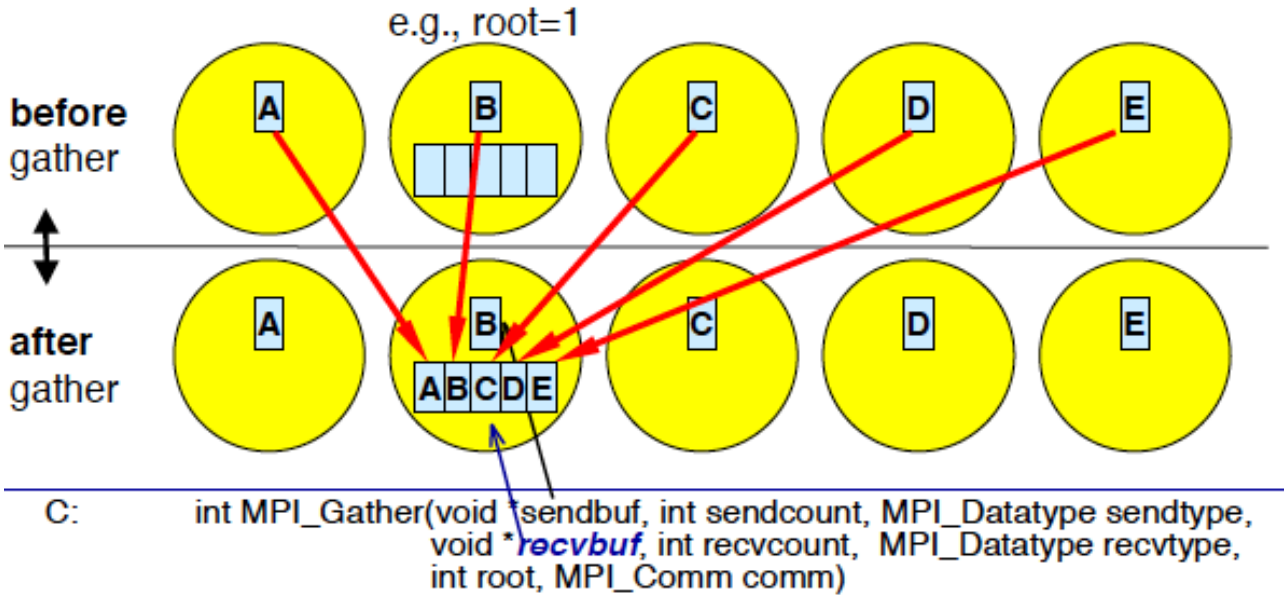
- 根进程向组内每个进程散播100个整型数据

```
MPI_Comm comm;
int gsize,*sendbuf;
int root,rbuf[100];
.....
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
.....
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Gather

MPI_Gather (SendAddress,SendCount,SendDatatype,
RecvAddress,RecvCount,RecvDatatype,Root,Comm)

- Root进程接收各个进程(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。
- 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。RecvCount是自每个进程接收数据个数。



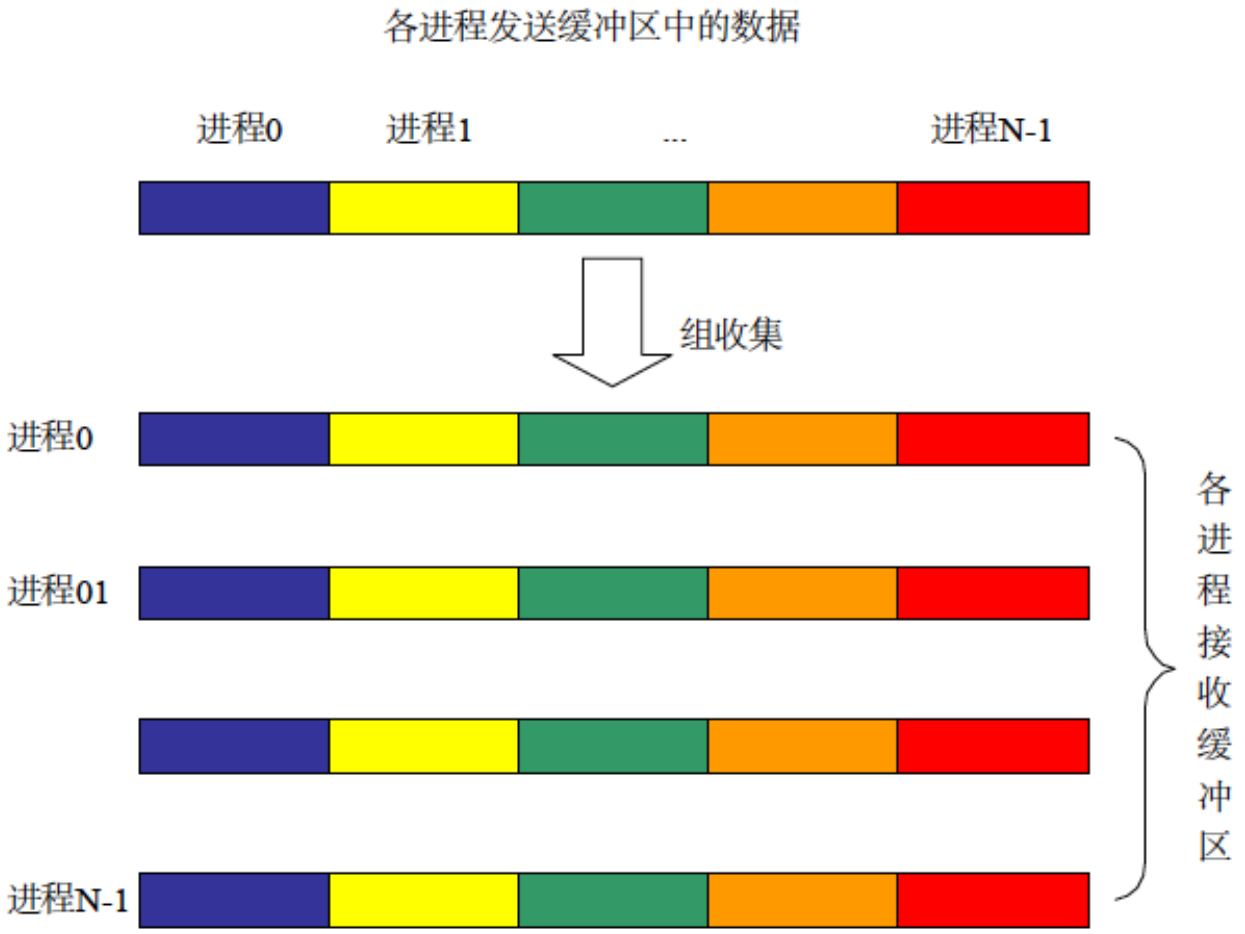
MPI_Gather

- 自进程组中每个进程收集100个整型数给根进程

```
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);
```

Allgather

MPI_Allgather (SendAddress, SendCount, SendDatatype,
RecvAddress, RecvCount, RecvDatatype, Comm **)**



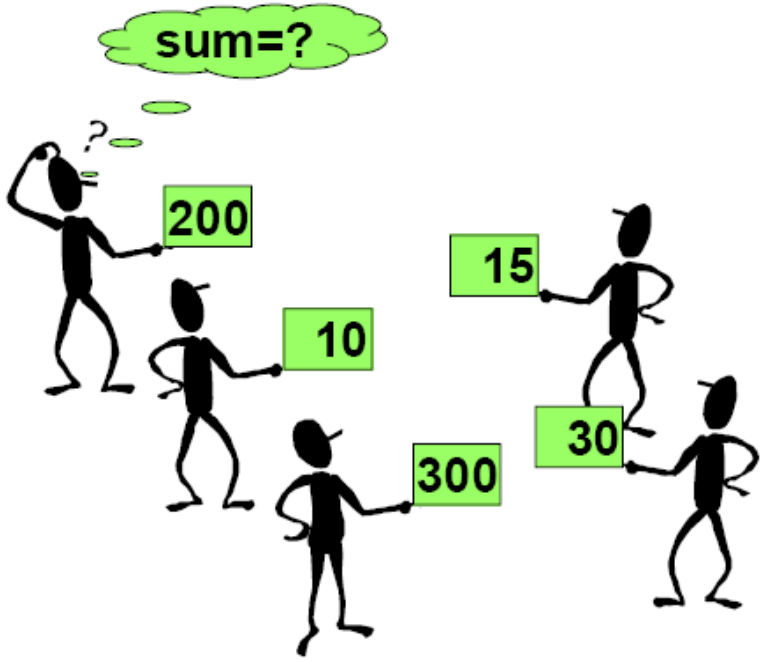
MPI_Allgather

- 每个进程都从其他进程收集100个数据，存入自己的缓冲区内

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
.....
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

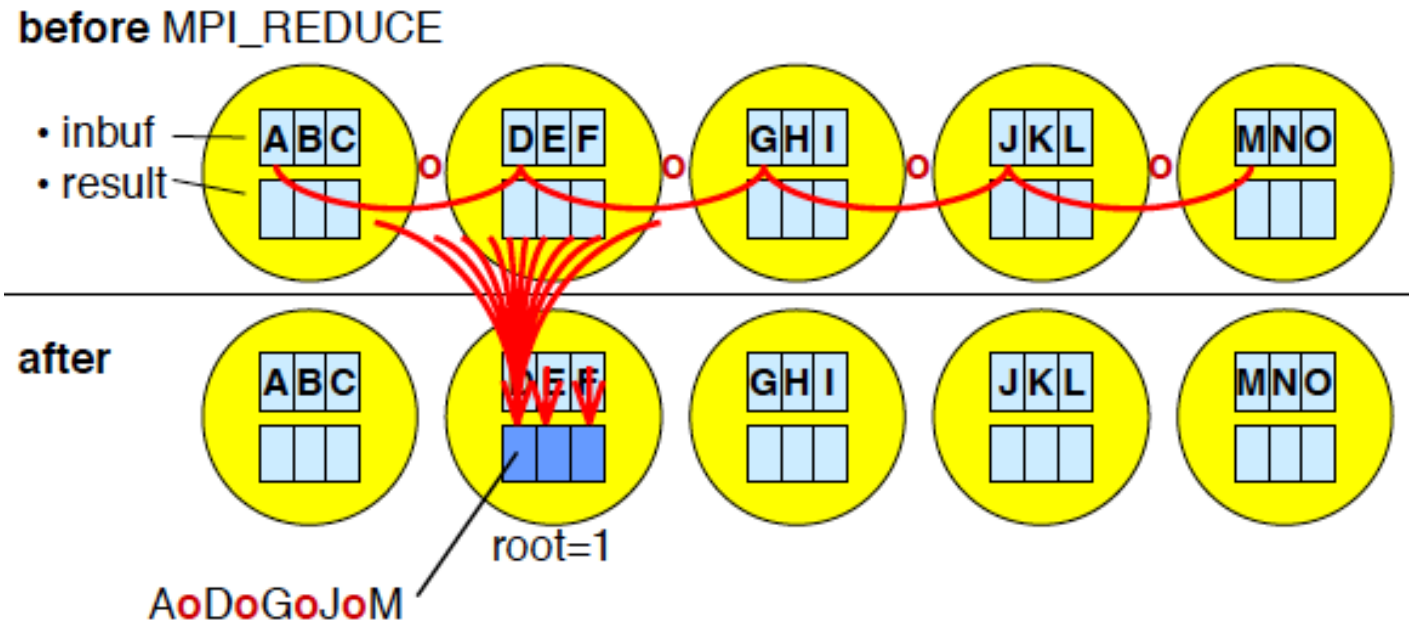
归约（Reduce）

- 所有进程向同一进程发送消息，与broadcast的消息发送方向相反。
- 接收进程对所有收到的消息进行归约处理。
- 归约操作：
  - MAX, MIN, SUM, PROD, LAND, BAND, LOR, BOR, LXOR, BXOR, MAXLOC, MINLOC



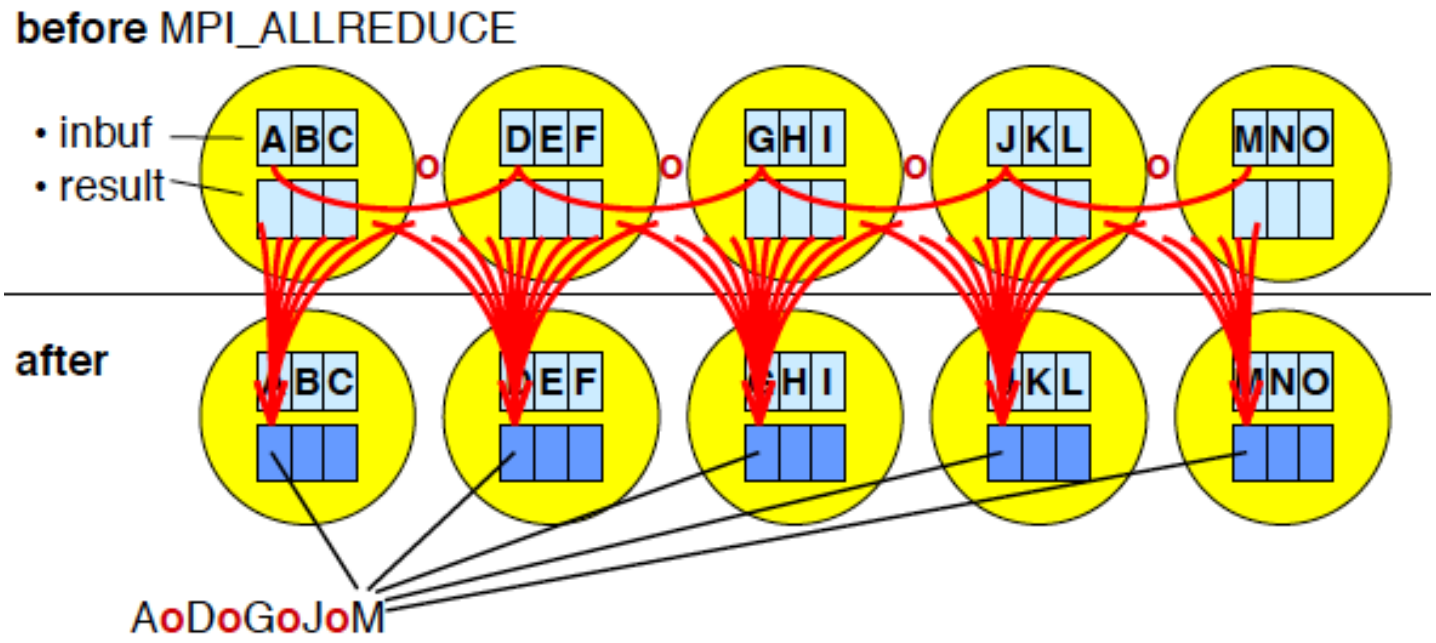
MPI_Reduce

MPI_REDUCE(inbuf,result,count,datatype,op,root,comm)

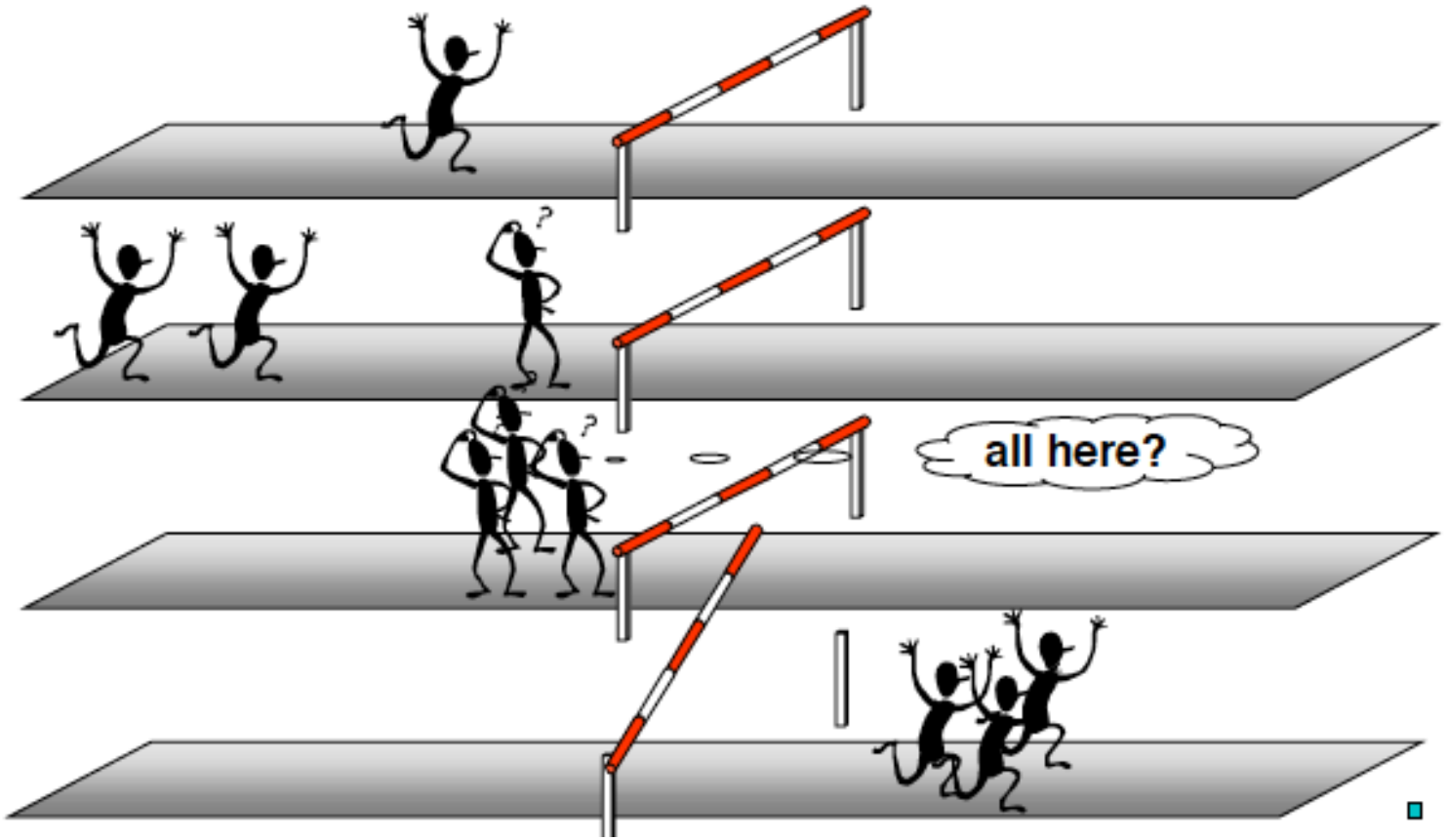


MPI_Allreduce

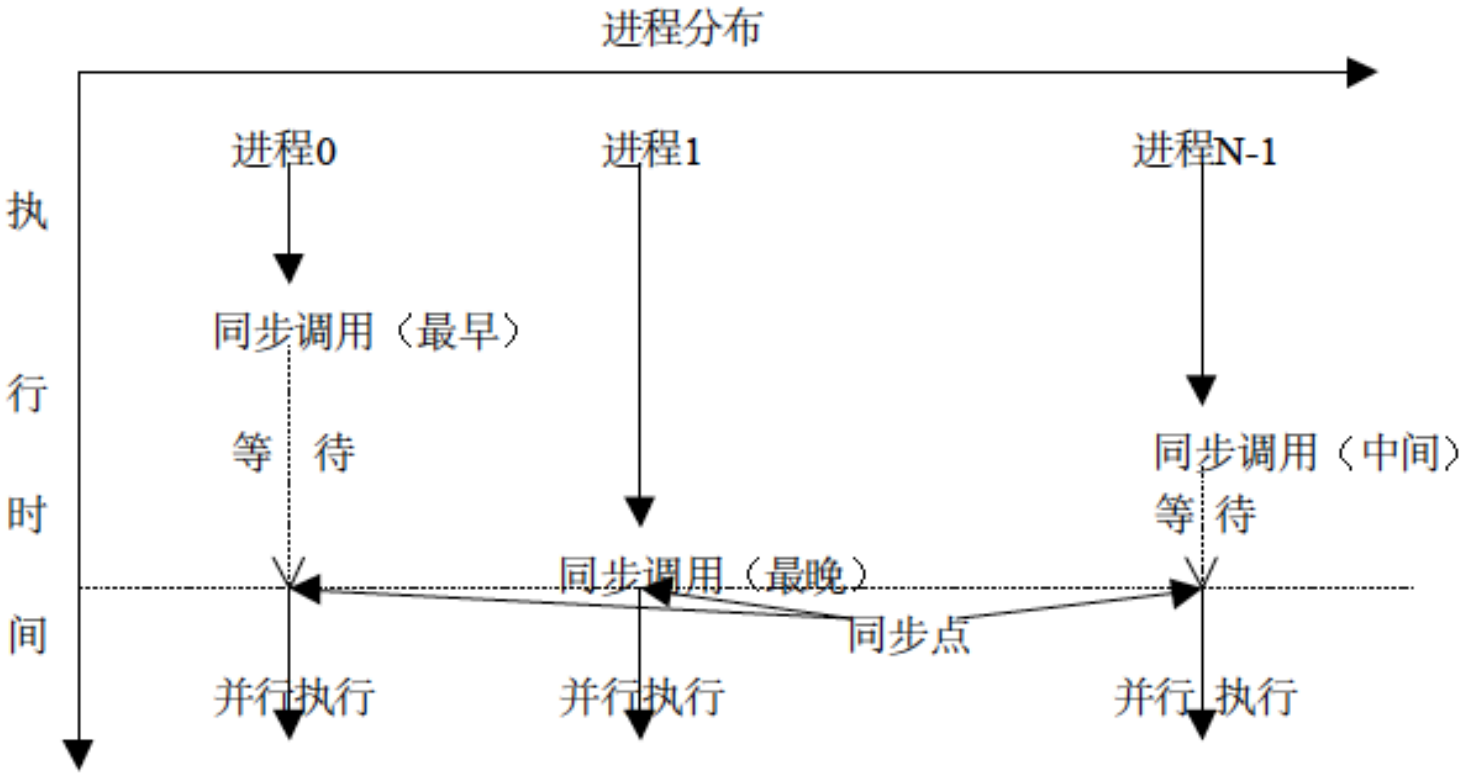
- 语法与reduce类似，但无root参数
- 所有进程都将获得结果



MPI_Barrier



MPI_Barrier


进程分布
进程0
进程1
进程N-1
执
行
时
间
同步调用（最早）
等 待
同步调用（中间）
等 待
同步调用（最晚）
同步点
并行执行
并行执行
并行 执行

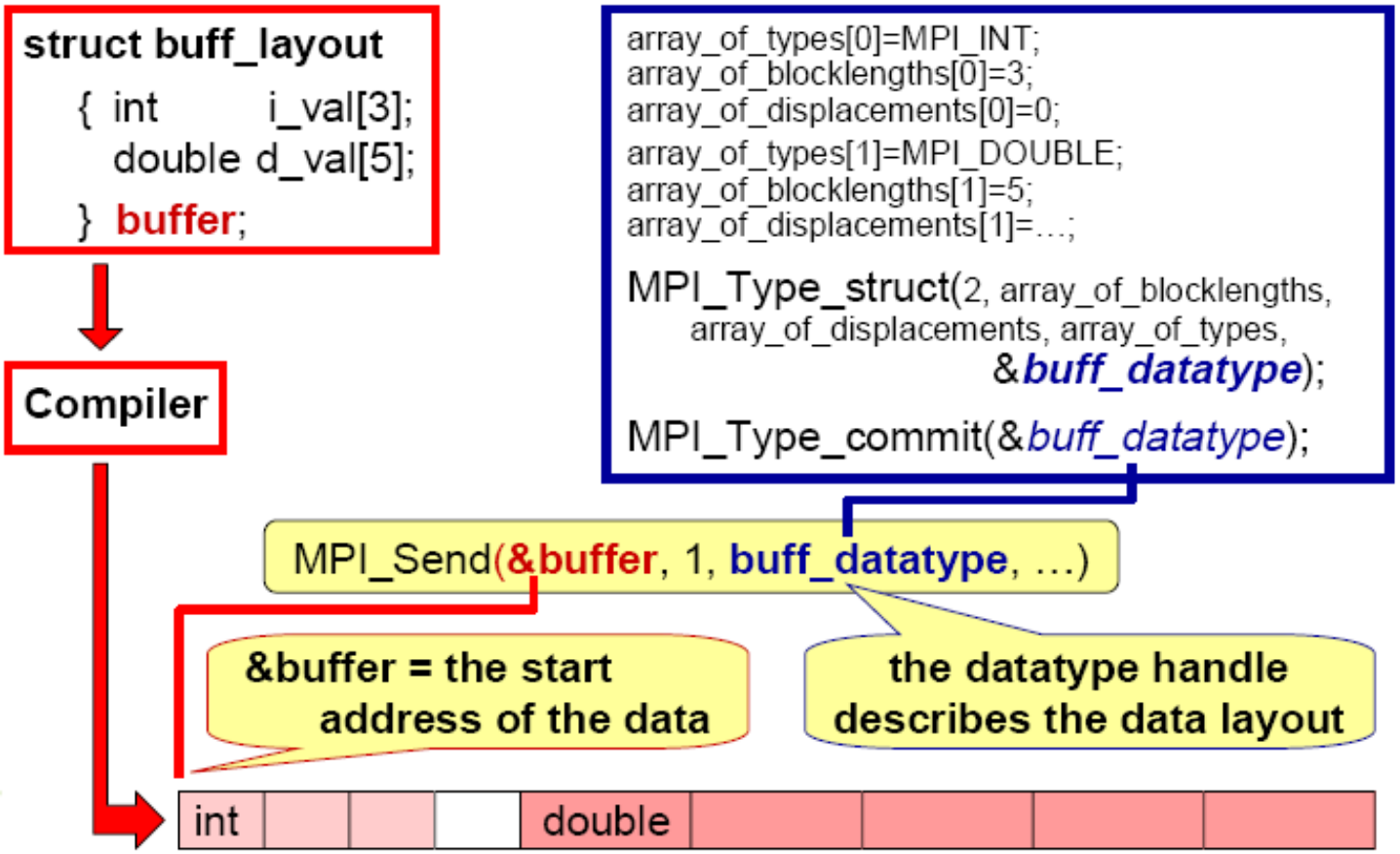
Outline

- MPI概述
- 点到点通信/组通信
- **自定义数据类型**

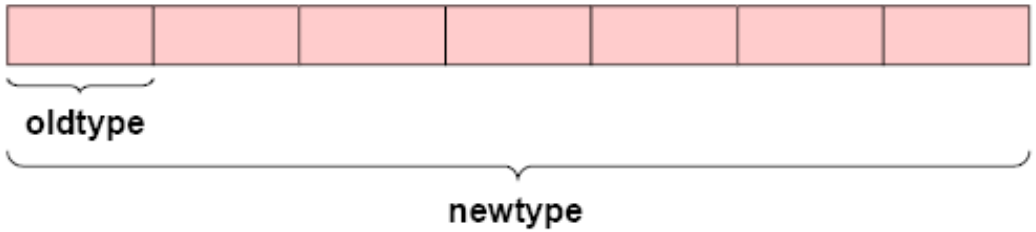
MPI基本数据类型

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	

自定义数据类型

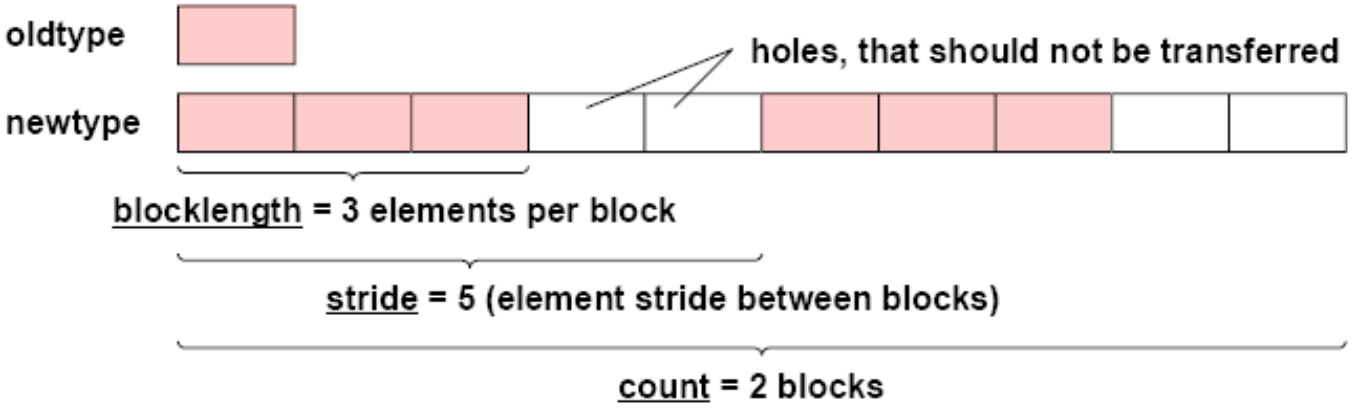

struct buff_layout
{ int i_val[3];
double d_val[5];
} buffer;
Compiler
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=…;
MPI_Type_struct(2, array_of_blocklengths, array_of_displacements, array_of_types, &buff_datatype);
MPI_Type_commit(&buff_datatype);
MPI_Send(&buffer, 1, buff_datatype, …)
&buffer = the start address of the data
the datatype handle describes the data layout
int
double

自定义数据类型：连续数据



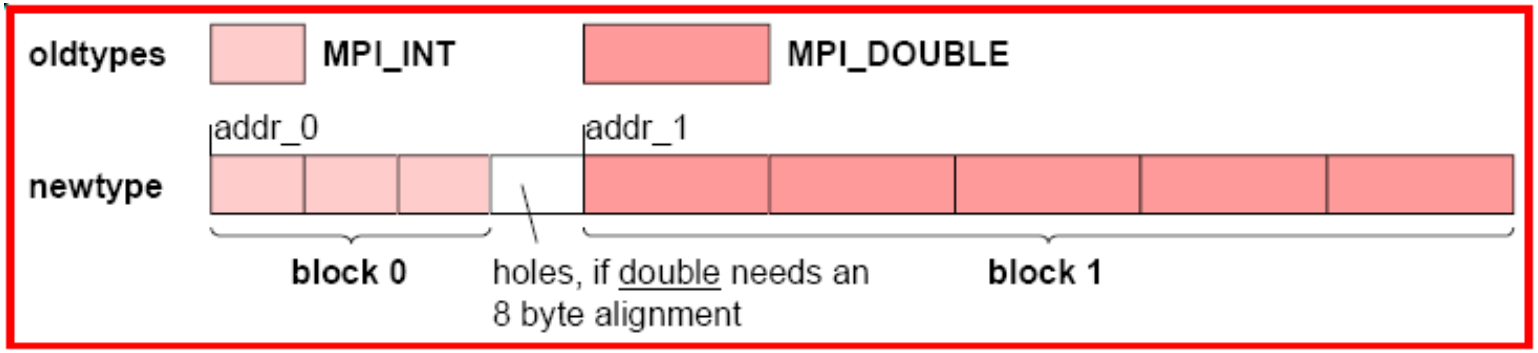
- C: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
- Fortran: MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
  INTEGER COUNT, OLDTYPE
  INTEGER NEWTYPE, IERROR

自定义数据类型：向量



- C: int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
- Fortran: MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
  INTEGER COUNT, BLOCKLENGTH, STRIDE
  INTEGER OLDTYPE, NEWTYPE, IERROR

自定义数据类型：结构体



- C: int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)
- Fortran: MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)