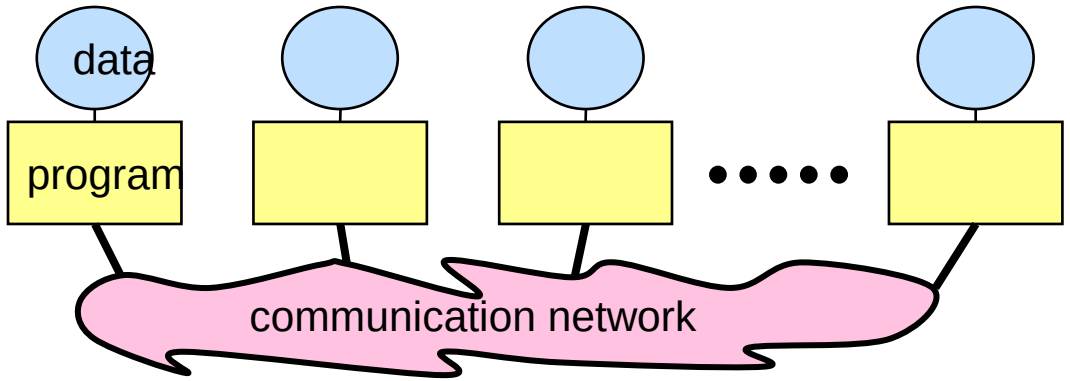
Programming Using the Message Passing Paradigm

目录

- Message-Passing Programming简介
- 消息传递库
- MPI: 消息传递接口
  - 基本概念
  - 通信模式
  - 群集通信

Message-Passing Programming简介

- 消息通信中的每一处理器/进程执行程序的一实例:
  - 以常规的语言编写程序，
  - 程序通常处理多数据集
  - 每一子程序的变量有着相同的名字，但所处位置不同，内容也不同
  - 通过send和receive进行通信 (message passing)



Message-Passing Programming简介

- 消息通信逻辑视图：
  - p个进程，每个进程有自己独立的地址空间.
  - 每一数据属于一地址空间的一区域.
  - 所有只读或读写交互至少涉及两进程：拥有数据的进程及存取该数据的进程.
- 关于消息通信, 涉及三个方面:
  - 共有多少个进程
  - 进程间如何同步
  - 如何管理通信缓冲区

进程通信：Send和Receive操作

- 函数原型如下:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- 如下代码里， send操作的语义要求进程P1收到的值必须是100而不是0.
- 需要设计send和receive的协议.

P0

```
a = 100;
send(&a, 1, 1);
a = 0;
```

P1

```
receive(&a, 1, 0)
printf("%d\n", a);
```

P0：a是发送消息缓冲 (send message buffer, or send buffer)

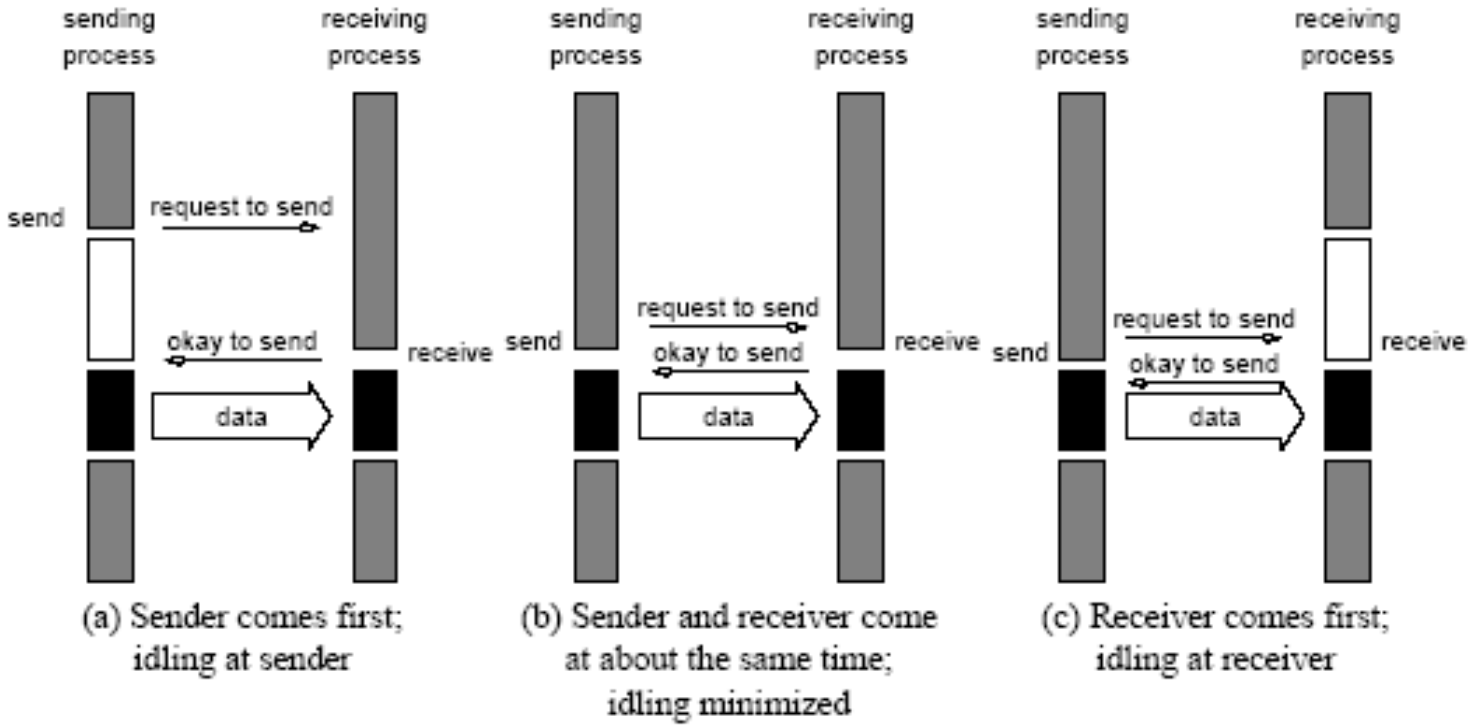
P1：a是接收消息缓冲 (receive message buffer, or receive buffer)

Message-Passing Programming简介

- 异步或同步模式.
  - 在异步模式里，并发任务异步执行.
  - 在同步模式里，任务的子集需要同步交互. 在交互之间，任务异步执行.
- 阻塞/非阻塞

无缓存阻塞消息通信(Non-Buffered Blocking Message Passing Operation)

- 实现send/receive语义的一种简单方法：当安全时send操作才返回
- 无缓存的阻塞发送不会返回，直到接收进程的与之相匹配的receive开始接收.
- 空闲和死锁是无缓存阻塞发送需要解决的两大问题.

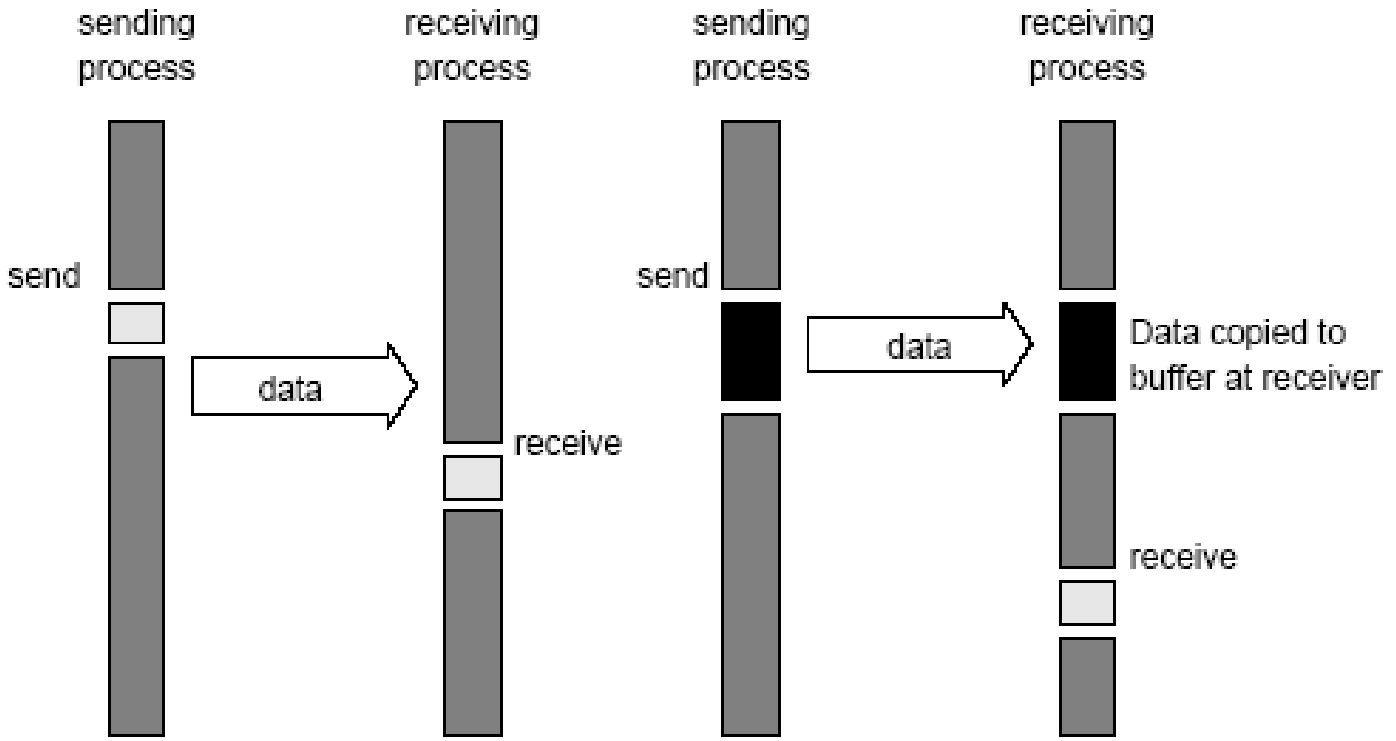


Handshake for a blocking non-buffered send/receive operation.

缓存式阻塞消息通信(Buffered Blocking Message Passing Operations)

- 对于空闲等待和死锁问题的简单解决方法：在发送和接收端缓存.
- 在缓存式阻塞发送里，发送者简单地将数据拷贝到指定的缓冲区，当拷贝操作完成后返回. 接收端也同样将数据拷贝到缓冲区.
- 缓存降低了闲置时间，但引入了拷贝开销.
- 缓存平衡空闲开销和拷贝开销.

缓存式阻塞消息通信



(a)接收端和发送端通信硬件具有buffer

(b)通信硬件不支持，发送方中断接收方，将数据写到接收方buffer

缓存式阻塞消息通信

Buffer大小对性能有极大影响，例如下例中，当消费者接收数据速度极慢时

P0

```
for (i = 0; i < 1000; i++){
    produce_data(&a);
    send(&a, 1, 1);
}
```

P1

```
for (i = 0; i < 1000; i++){
    receive(&a, 1, 0);
    consume_data(&a);
}
```

缓存式阻塞消息通信

虽然有buffer，由于接收操作阻塞，死锁依然可能存在

P0

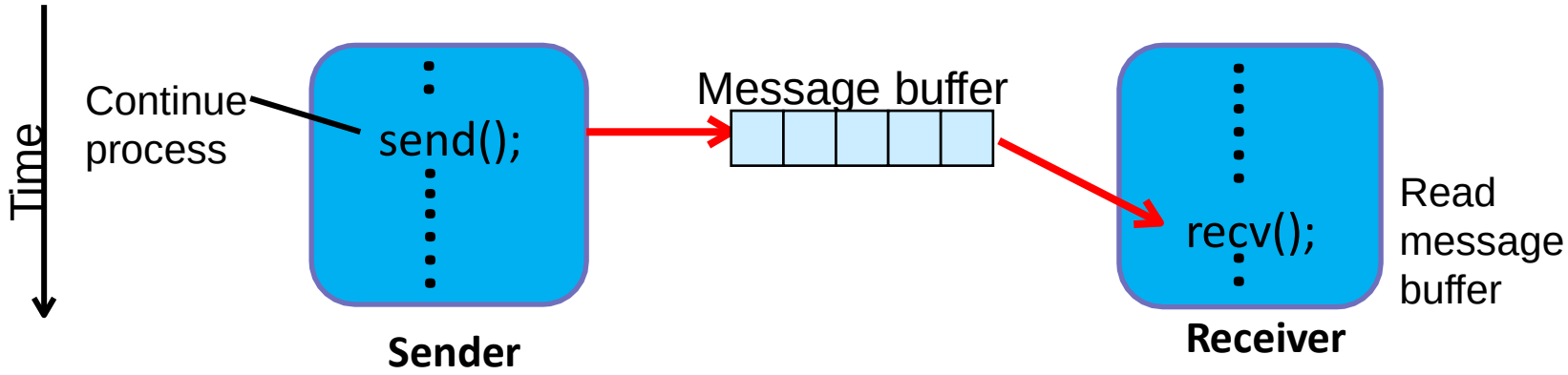
```
receive(&a, 1, 1);
send(&b, 1, 1);
```

P1

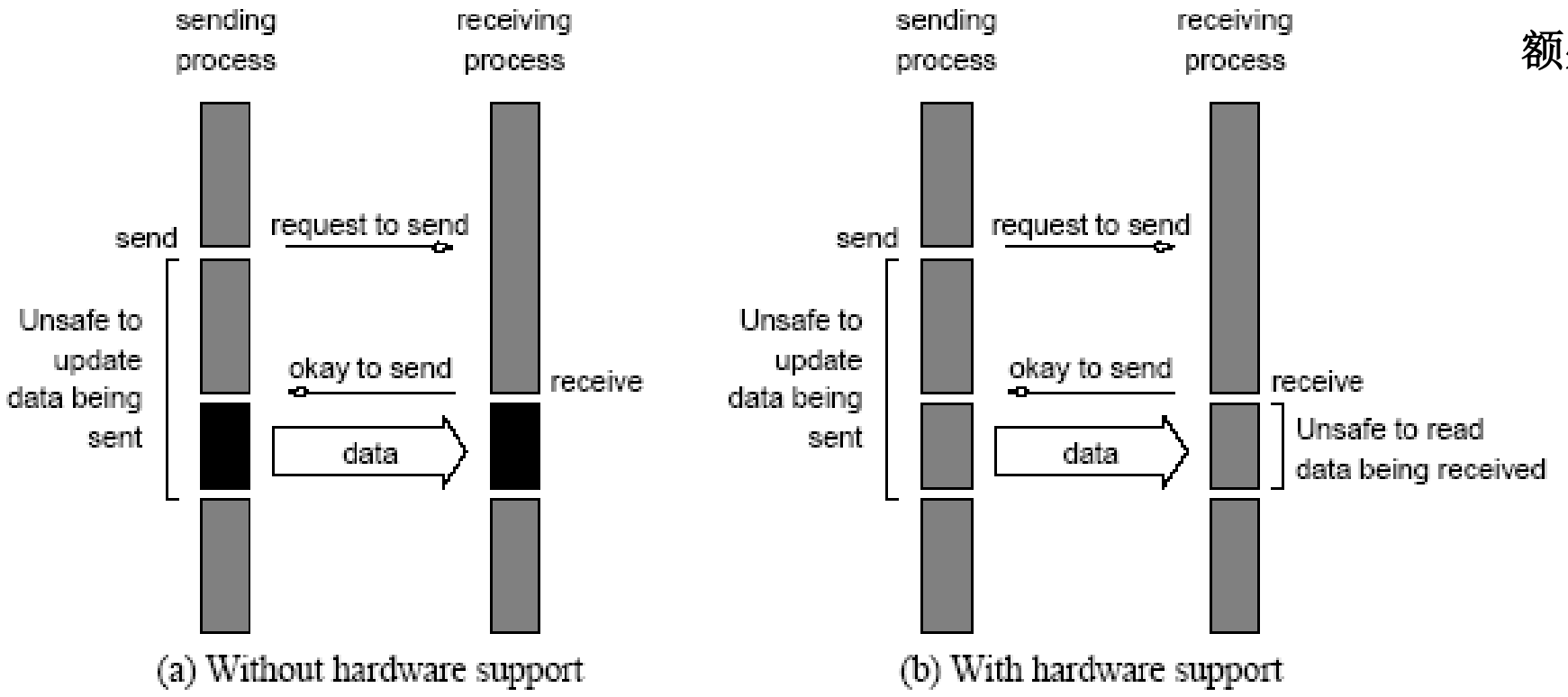
```
receive(&a, 1, 0);
send(&b, 1, 0);
```

非阻塞消息通信(Non-Blocking Message Passing Operations)

- 非阻塞操作通常伴随有状态检查操作.
- 非阻塞操作可实现通信与计算重叠.



非阻塞消息通信

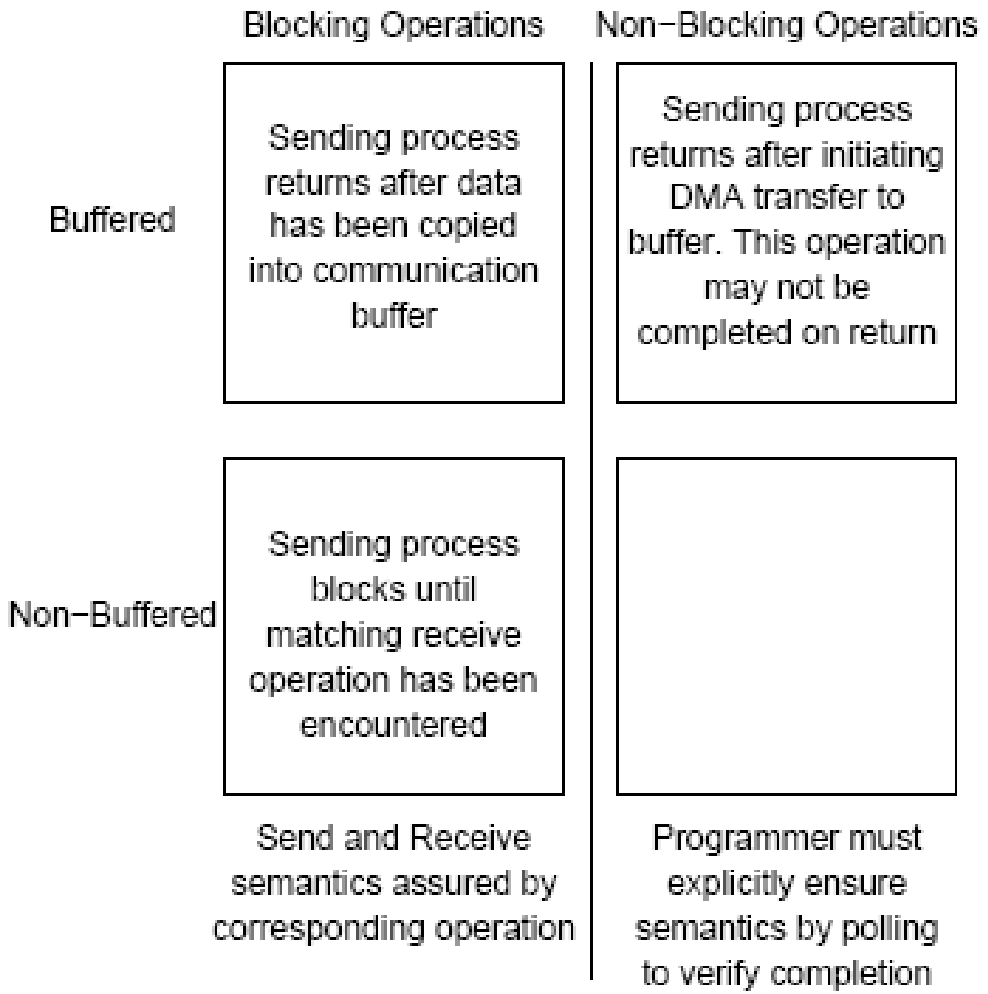


非阻塞模式本身存在一些额外开销:

- 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

非阻塞无buffer的发送和接收操作 (a) 无通信硬件支持; (b)有通信硬件支持.

发送和接收协议


Blocking Operations
Non−Blocking Operations
Buffered
Sending process returns after data has been copied into communication buffer
Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non−Buffered
Sending process blocks until matching receive operation has been encountered
Send and Receive semantics assured by corresponding operation
Programmer must explicitly ensure semantics by polling to verify completion

消息传递库

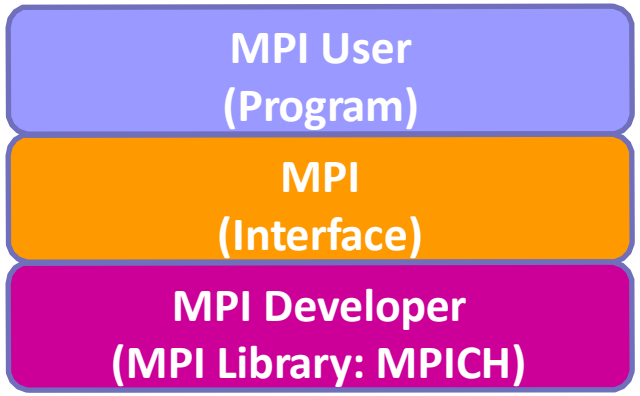
名字	Original Creator	特征
CMMD	Thinking Machines	是一个用于Thinking Machines CM-5系统的消息传递库, 其特点是基于主动消息(Active Message)机制在用户空间实现通信以减少通信延迟;
Express	Parasoft	支持点到点和群集通信以及并行I/O的程序设计环境
Fortran-M	Argonne National Lab	是对Fortran77的扩展, 它在设计上既支持共享存储也支持消息传递, 但当前只实现了对消息传递的支持. 该语言提供了许多机制用于支持开发行为确定、模块化的并行程序
MPI	MPI Forum	A widely adopted standard
NX	Intel	为Intel MPP(例如, Hypercubes和 Paragon)开发的微核系统. 现在已由用于Intel/Sandia ASCI TFLOPS 系统中的新的微核系统PUMA代替
P4	Argonne National Lab	Parallel Programs for Parallel Processors：是一组宏和子程序, 用于支持共享存储和消息传递系统中的程序设计, 它可以移植到许多体系结构上
PARMACS	ANL/GMD	Mainly used in Europe
PVM	Oak Ridge National Lab	A widely used, stand-alone system
UNIFY	Mississippi State	A system allowing both MPI and PVM calls
Zipcode	Livemore National Lab	Contributed to the context concept

What is MPI

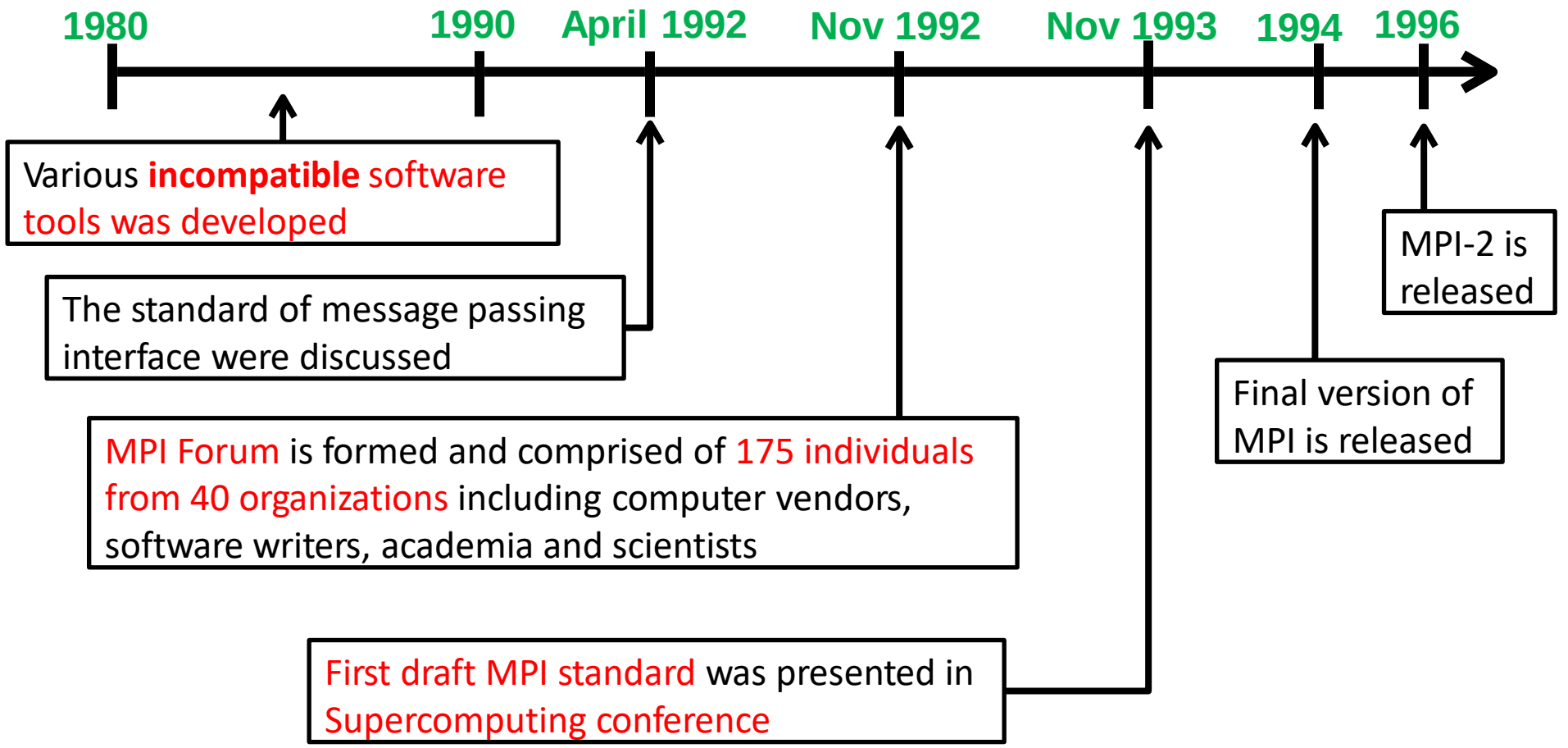
- MPI = Message Passing **Interface**
- A **specification** for the

  developers and users

  of message passing libraries
  - By itself, it is an **interface NOT a library**
- Commonly used for **distributed memory system & high-performance computing**
- Goal:
  - **Portable**: Run on different machines or platforms
  - Scalable: Run on million of compute nodes
  - Flexible: Isolate MPI developers from MPI programmers (users)



MPI发展历史


1980
1990
April 1992
Nov 1992
Nov 1993
1994
1996
Various **incompatible** software tools was developed
The standard of message passing interface were discussed
MPI Forum is formed and comprised of 175 individuals from 40 organizations including computer vendors, software writers, academia and scientists
First draft MPI standard was presented in Supercomputing conference
Final version of MPI is released
MPI-2 is released

消息传递库

- 提供了与C、Fortran和Java语言的绑定. 不包含任何专用于特定制造商、操作系统或硬件的特性. 由于这个原因, MPI在并行计算界得到广泛接受.
- MPI的实现
  - 建立在厂家专用的环境之上
    - IBM SP2的POE/MPL,
    - Intel Paragon的OSF/Nx
  - 公共的MPI环境:
    - CHIMP Edinburg 大学
    - LAN(Local Area Multicomputer) Ohio超级计算中心
    - MPICH Argonne国家实验室与Mississippi州立大学
- MPICH是MPI在各种机器上的可移植实现,可以安装在几乎所有的平台上

消息传递库

- PVM(Parallel Virtual Machine) 简介
  - 开发时间: 始于1989年
  - 开发单位: 美国Tennessee大学、Oak Ridge国家实验室和Emory大学联合研制
  - 特点: 具有较好的适应性、可扩展性、可移植性和易使用性等特点, 源代码可以免费获取, 现已被用户广泛采纳.
  - 现状: 目前对它的研究和开发工作仍在各大学和研究机构进行. 尽管已经有越来越多的人开始使用MPI, 但PVM仍然是并行处理最流行的软件之一. 随着它的不断流行, 已经被移植到多种机群系统.
- 包括资源管理、进程控制、消息传递、动态任务组、容错

消息传递库

- PVM和MPI间的主要差别:
  - PVM是一个自包含的系统, 而MPI不是. MPI依赖于支持它的平台提供对进程的管理和I/O功能. 而PVM本身就包含这些功能.
  - MPI对消息传递提供了更强大的支持.
  - PVM不是一个标准, 因此PVM可以更方便、更频繁地进行版本更新.
- MPI和PVM在功能上现在正趋于相互包含. 例如, MPI-2增加了进程管理功能, 而现在的PVM也提供了更多的群集通信函数.
- 在国产并行机神威、银河和曙光上都实现了对MPI和PVM和支持

MPI编程

- MPI中的消息
- MPI中的消息信封
- MPI中的通信模式
- 点对点通信
- 群集通信

MPI: the Message Passing Interface

- MPI 定义了核心库函数的语法及语义.
- C:

  error = MPI_Xxxxxx(parameter, ...);

  MPI_Xxxxxx(parameter, ...);
- Fortran:

  CALL MPI_XXXXXX(parameter, ..., ***IERROR***)
- MPI_..... 专用于MPI函数和常数，应用里面的过程及变量命名不应以MPI_开头

MPI: the Message Passing Interface

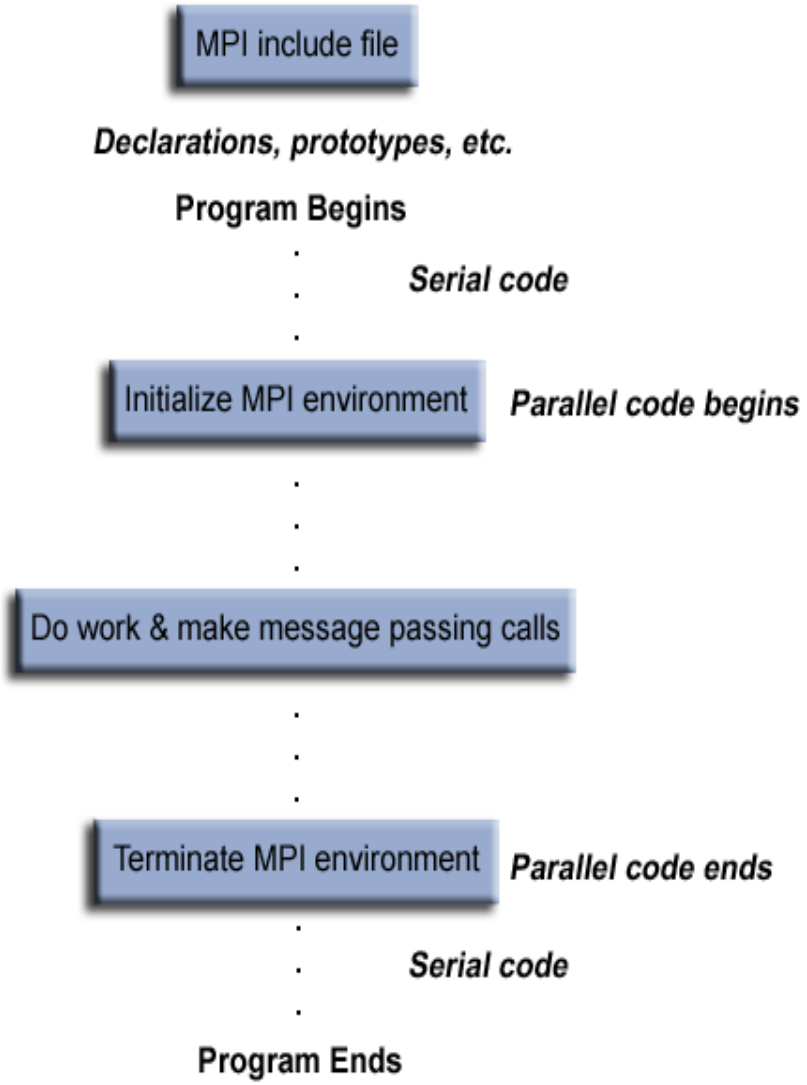
- 125函数，常用的主要函数有6个

MPI函数最小集

`MPI_Init`	初始化MPI.
`MPI_Finalize`	终止MPI.
`MPI_Comm_size`	确定进程数量.
`MPI_Comm_rank`	返回调用进程的label.
`MPI_Send`	发送消息.
`MPI_Recv`	接收消息.

MPI程序结构

- 头文件: **“mpi.h”**
- MPI调用:
  - Format: rc = MPI_Xxx(parameter, ...)
  - Example: rc = MPI_Bcast (&buffer,count,datatype,root,comm)
  - Error code: return as “rc”; rc=MPI_SUCCESS if successful



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,i,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

得到缺省的进程组大小

得到每个进程在组中的编号

发送消息

接收消息

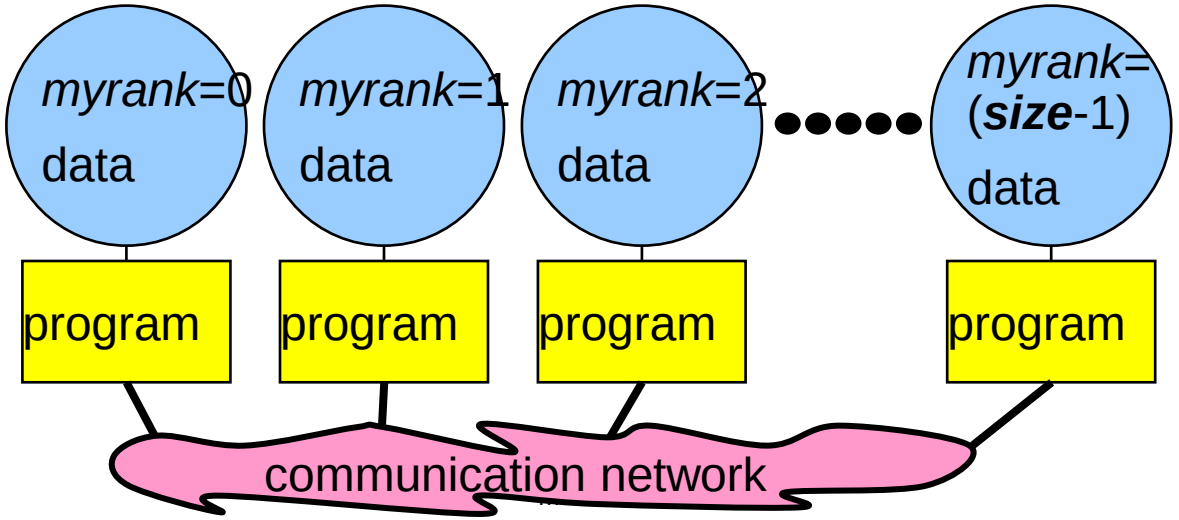
终止MPI环境

基本概念

- 消息数据类型 (message data types)
- 通信子 (communicators)
- 通信操作 (communication operations)

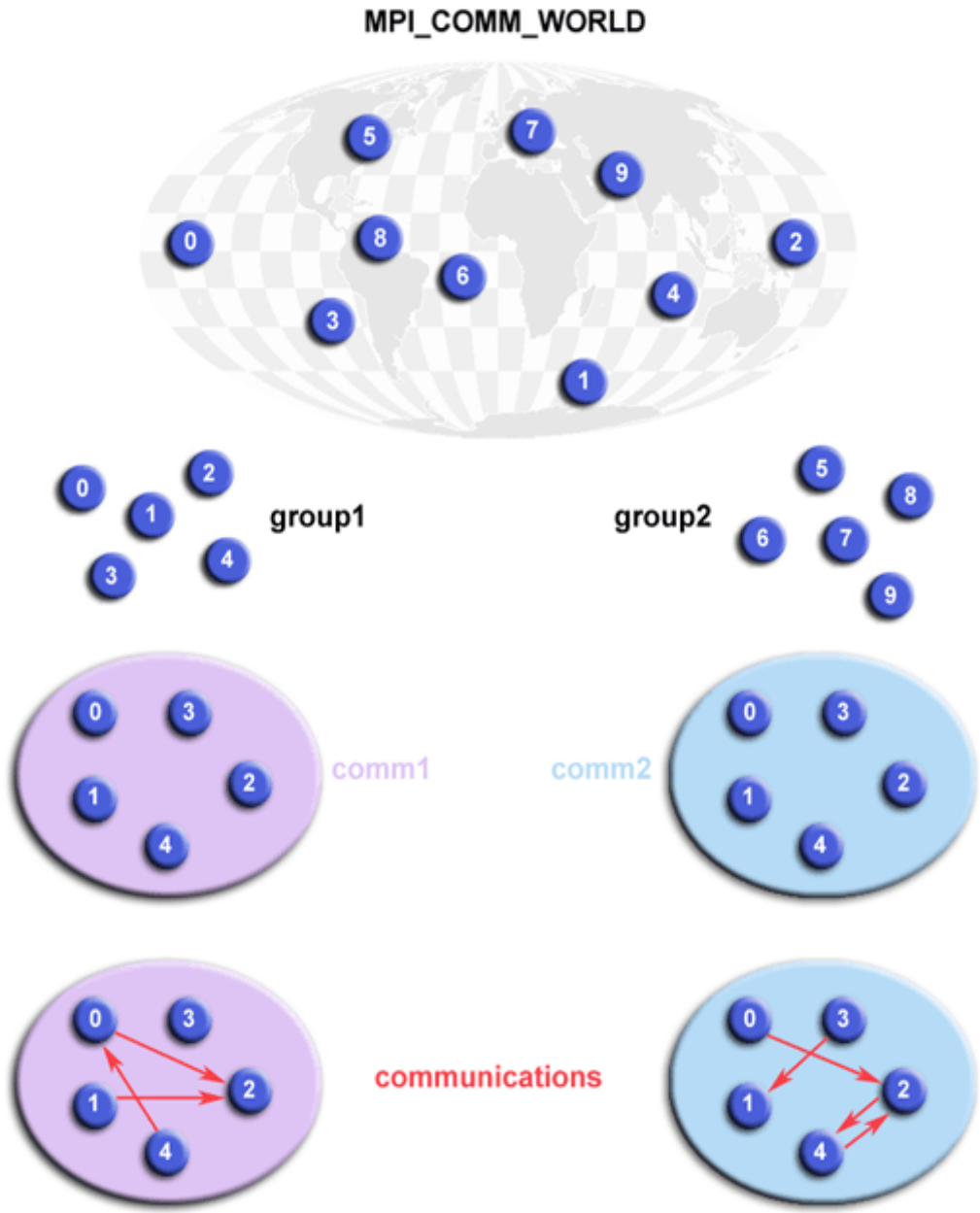
进程标识

- 为了通信，MPI进程需要标识符: **rank**
- 所有分布决策都基于***rank***
  - 例如，哪个进程处理哪一数据



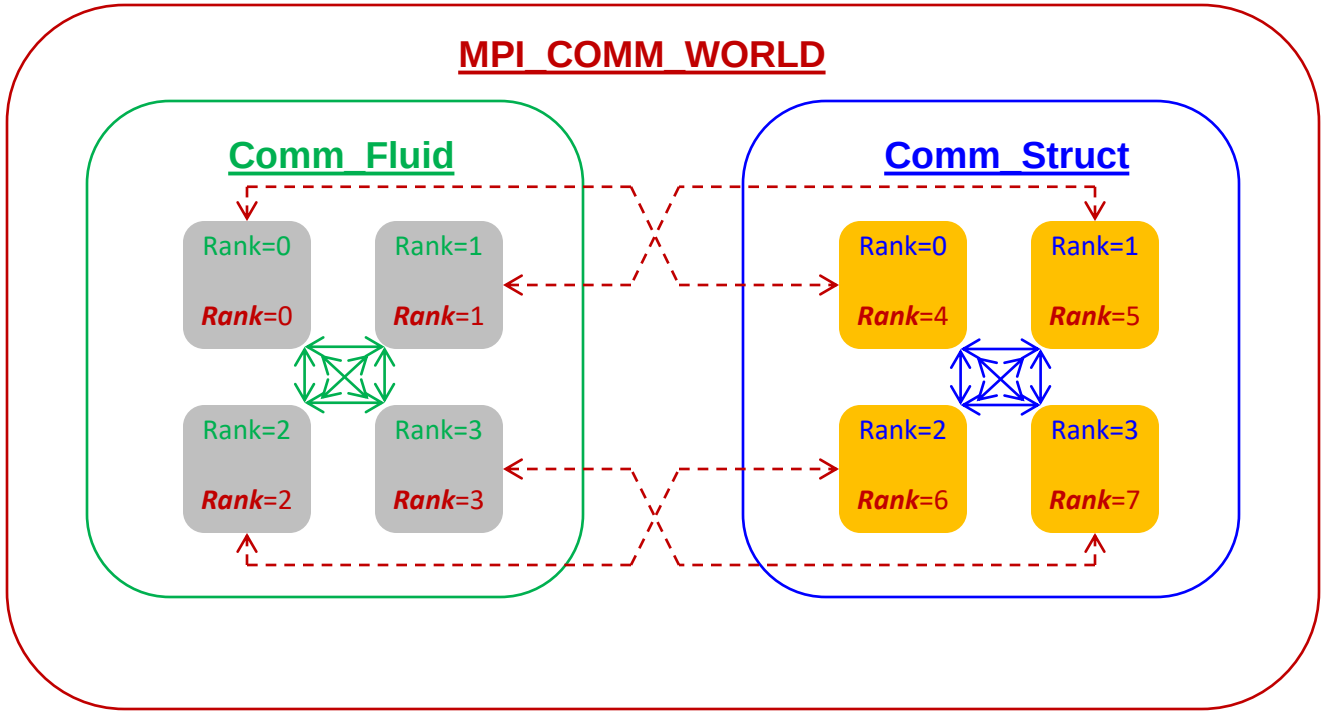
Groups & communicators

- **通信域（communicator）是一个综合的通信概念**。其包括上下文（context），进程组（group），虚拟处理器拓扑（topology）。一个通信域对应一个进程组。
- **进程（process）与进程组（group）：**同一个进程可以属于多个进程组（每个进程在不同进程组中有各自的rank号）；同一个进程可以属于不同的进程组，因此也可以属于不同的通信域。
- 创建群组动机
  - 编程方便
  - 使用群通信



Communicator

- Communicator定义了一通信域(*communication domain)*，即一组可相互通信的进程.
- 通信域的信息存储在一类型为MPI_Comm的变量里.
- 进程可以属于多个通信域.
- MPI定义了一个缺省的通信域MPI_COMM_WORLD，它包含所有进程.
- **通信子(communicator): 一个进程组(process group)+上下文(context).**



- ✓ Ranks within MPI_COMM_WORLD
- ✓ Ranks within Comm_Fluid
- ✓ Ranks within Comm_Struct

Communicator管理

- Communicator访问
  - MPI_COMM_SIZE(…)
  - MPI_COMM_RANK(…)
- Communicator创建
  - 在已有通信域基础上复制：MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
  - MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)：在已有进程组的基础上创建
  - 在已有通信域基础上划分获得：MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
- Communicator Destructors
  - MPI_COMM_FREE(comm)

Group管理

- 进程组（group）可以当成一个集合的概念，可以通过“子、交、并、补”各种方法。所有进程组产生的方法都可以套到集合的各种运算
- Group Accessors
  - MPI_Group_size(…)
  - MPI_Group_rank(…)
- Group Constructors
  - MPI_COMM_GROUP(…)
  - int MPIAPI MPI_Group_incl(MPI_Group group, int n, _In_count_(n) int *ranks, _Out_ MPI_Group *newgroup);
  - int MPIAPI MPI_Group_excl(MPI_Group group, int n, In_count_(n) int *ranks, _Out_ MPI_Group *newgroup);
- Group Destructors
  - MPI_GROUP_FREE(group)

Communicator及Group示例

考虑如下由10个进程执行的代码:

```
MPI_Comm MyWorld, SplitWorld;
int my_rank, group_size, Color, Key;
MPI_Init(&argc, &argv);
MPI_Comm_dup(MPI_COMM_WORLD, &MyWorld);
MPI_Comm_rank(MyWorld, &my_rank);
MPI_Comm_size(MyWorld, &group_size);
Color=my_rank%3;
Key=my_rank/3;
MPI_Comm_split(MyWorld, Color, Key, &SplitWorld);
```

Communicator及Group示例

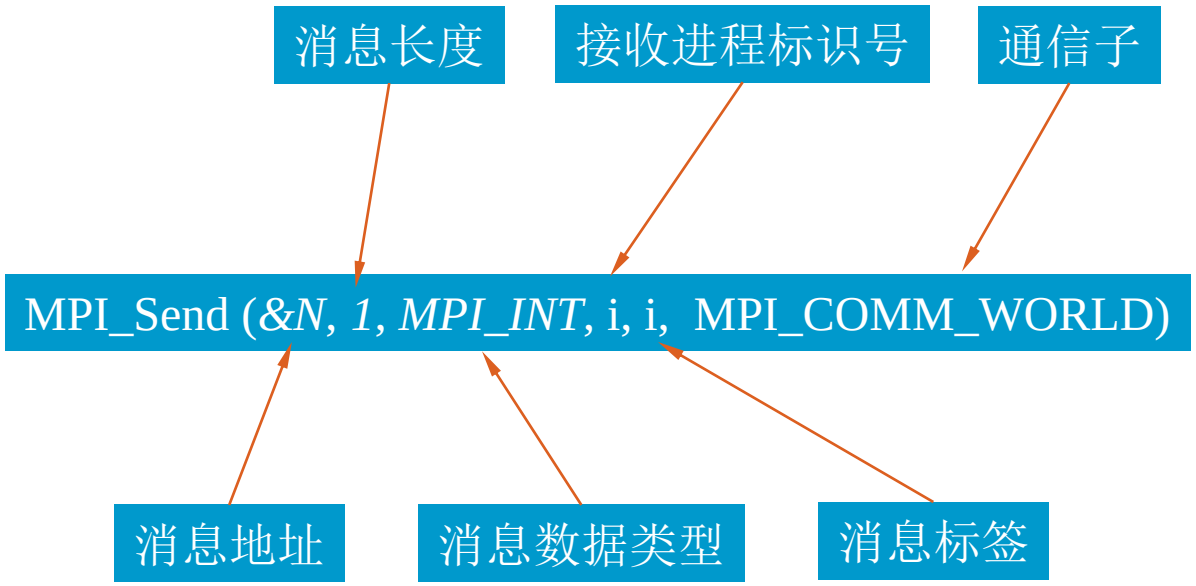
MPI_Comm_dup(MPI_COMM_WORLD, &MyWorld)

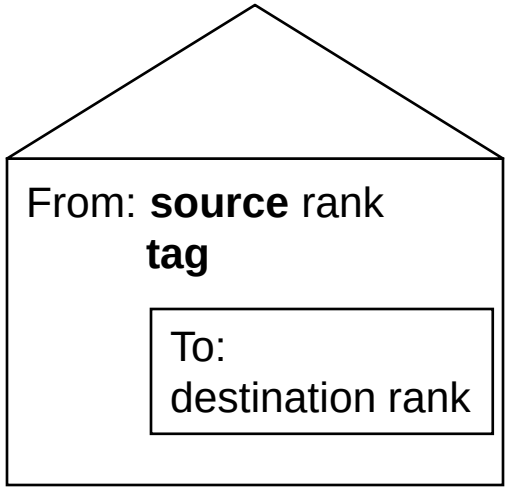
创建一个新通信子MyWorld, 它是包含与原始MPI_COMM_WORLD相同的10个进程的进程组,但有不同的上下文.

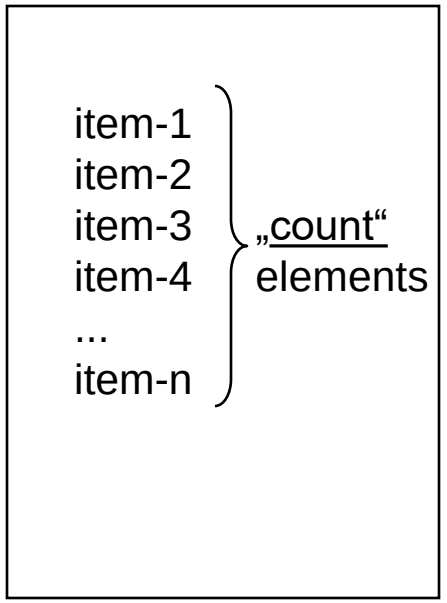
分裂一个通信子MyWorld

Rank in MyWorld	0	1	2	3	4	5	6	7	8	9
Color	0	1	2	0	1	2	0	1	2	0
Key	0	0	0	1	1	1	2	2	2	3
Rank in SplitWorld(Color=0)	0			1			2			3
Rank in SplitWorld(Color=1)		0			1			2		
Rank in SplitWorld(Color=2)			0			1			2	

MPI中的消息







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

(buffer, count, datatype) **消息缓冲**

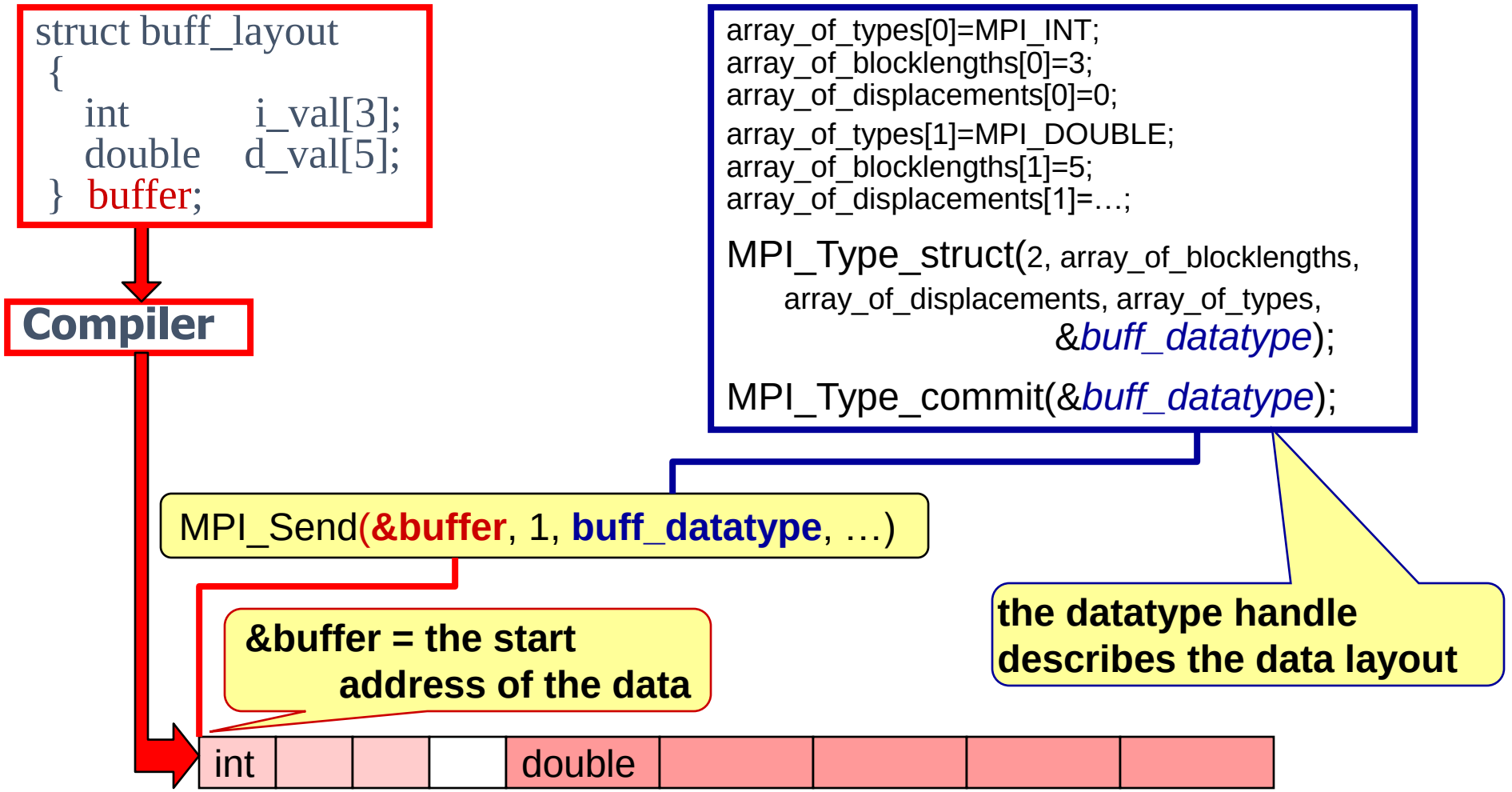
(destination, tag, communicator) **消息信封**

消息

- 消息包含具有特定数据类型的一些元素.
- 描述buffer的内存布局
- MPI数据类型:
  - 基础数据类型.
  - 派生数据类型： Vectors, structs, etc; Built from existing datatypes
- 数据类型句柄用于描述内存中的数据类型.

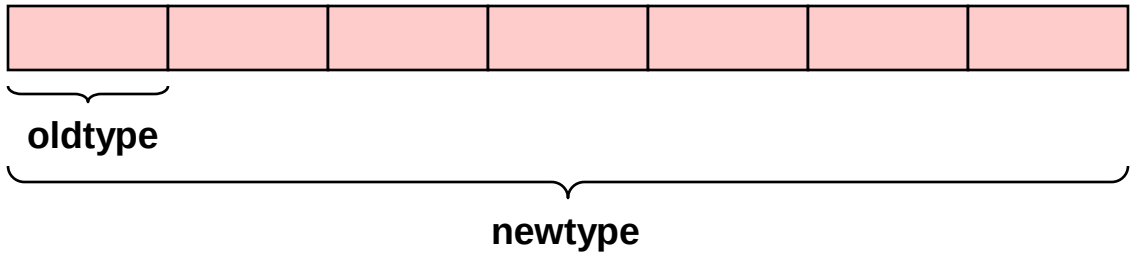
MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Data Layout and the Describing Datatype Handle



连续数据

- 最简单的派生数据类型
- 由一系列同一类型的数据项组成

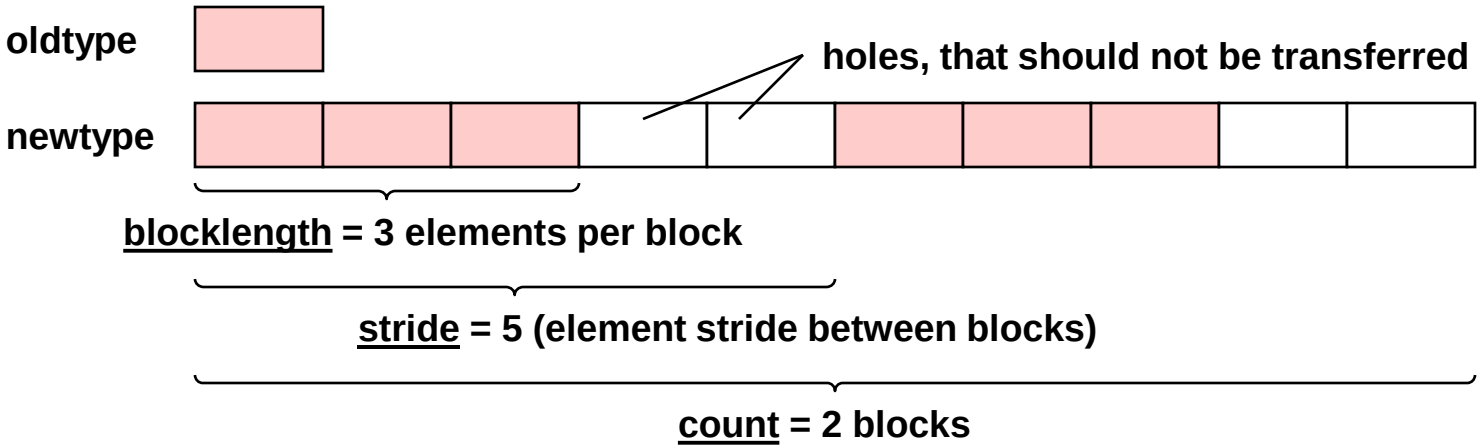


- int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

在消息传递中发送一个混合数据类型

序号	要发送的消息	消息的性质与定义方法
(1)	由数组A的所有元素组成的消息. A有100个元素, 每个元素是一个双精度数. 每一项的大小是8字节. 第i项的起始地址是A+8(i-1).	这两个消息有两个特征: . 数据项的存放是连续的; . 所有的数据项具有相同的数据类型.
(2)	由数组A的第3和第4项组成消息. 这个消息由两项A[2]和A[3]组成. 第一项始于A+16, 第二项始于A+24.	这类消息可以方便地用三元组(address, count, datatype)来定义. 第(1)个消息可以用 (A, 100, MPI_DOUBLE) 来定义. 第(2)个消息可以用 (A+16,2,MPI_DOUBLE) 来定义.
(3)	由数组A的所有偶序数项组成的消息: 由50项A[0], A[2], A[4], ..., A[98]组成. 第i项的起始地址是A+16(i-1).	第(3)个消息: 数据项没有放在一个连续的存储区中. 第(4)个消息: 数据不仅没有连续存放, 而且是混合数据类型. 上述简单的方法不能处理第(3)和第(4)个消息. MPI引入派生数据类型(derived data type)的概念, 允许定义可能是由混合数据类型、非连续存放的数据项组成的消息. 导出数据类型由用户的应用程序在运行时从基本的数据类型构造.
(4)	假定从如下数据结构: struct{ double A[100]; char b,c; int j,k;} S 取数组A的第3项, 后跟一个字符c, 再跟一个整型数k. 这个消息由三个不同类型的数据组成. 那么第一项A[2]的地址是S+16, 第二项c 的地址是S+801, 第三项k的地址是S+806.	

Vector数据类型



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

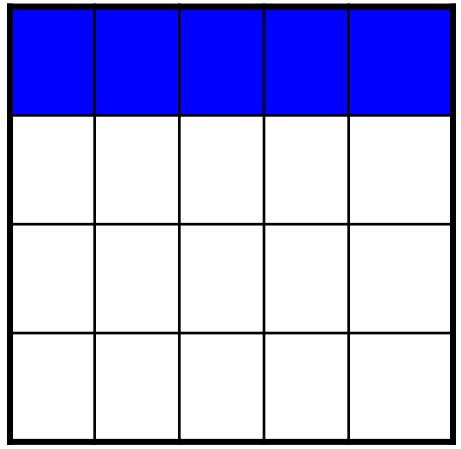
发送矩阵的一行和一列

MPI_Type_vector(**1, 5, 1**, MPI_INT, MPI_ROW)

MPI_Type_Commit(MPI_ROW)

MPI_Send(&buf …, MPI_ROW…)

MPI_Recv(&buf …, MPI_ROW…)

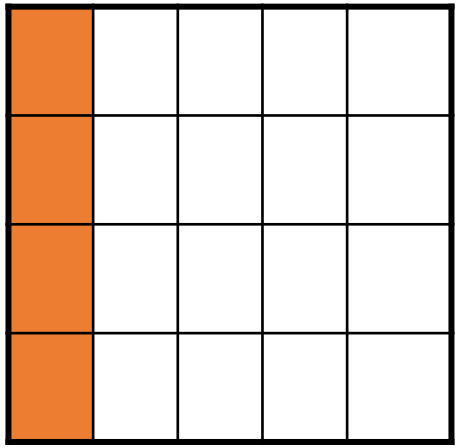


MPI_Type_vector(**4, 1, 5**, MPI_INT, MPI_COL)

MPI_Type_Commit(MPI_COL)

MPI_Send(buf …, MPI_COL…)

MPI_Recv(buf …, MPI_COL…)



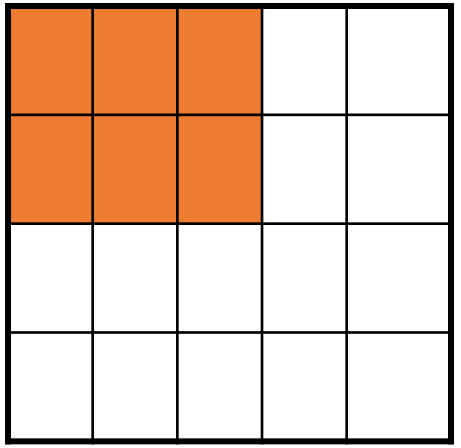
发送子矩阵

MPI_Type_vector(2, 3, 2, MPI_INT, MPI_SUBMAT)

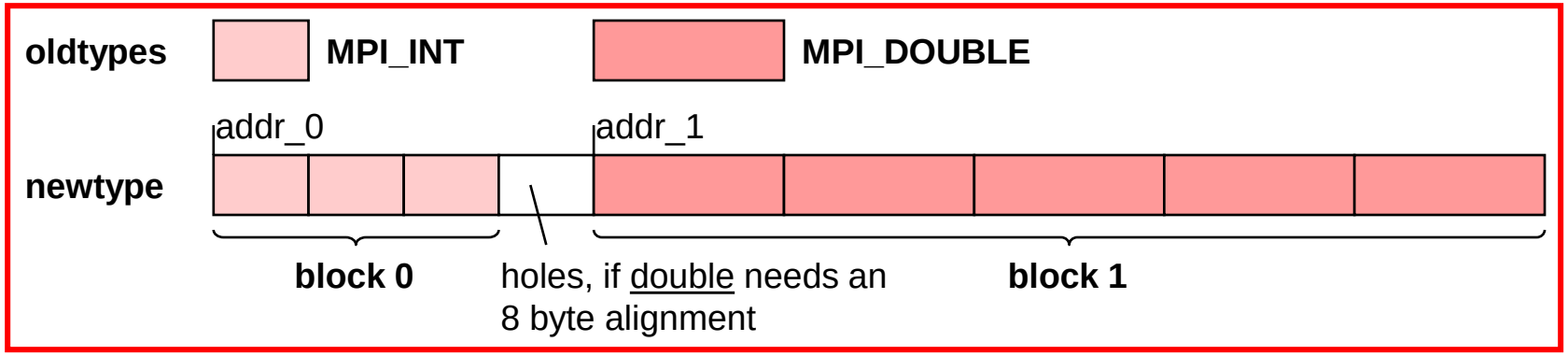
MPI_Type_Commit(MPI_SUBMAT)

MPI_Send(&buf …, MPI_SUBMAT…)

MPI_Recv(&buf …, MPI_SUBMAT…)



Struct Datatype



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

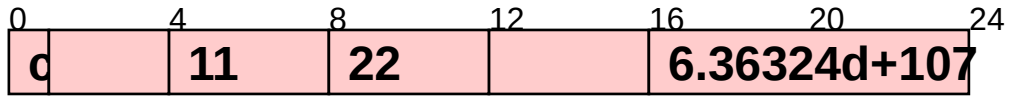
count = 2
array_of_blocklengths = (3, 5)
array_of_displacements = (0, addr_1 – addr_0)
array_of_types= (MPI_INT, MPI_DOUBLE)

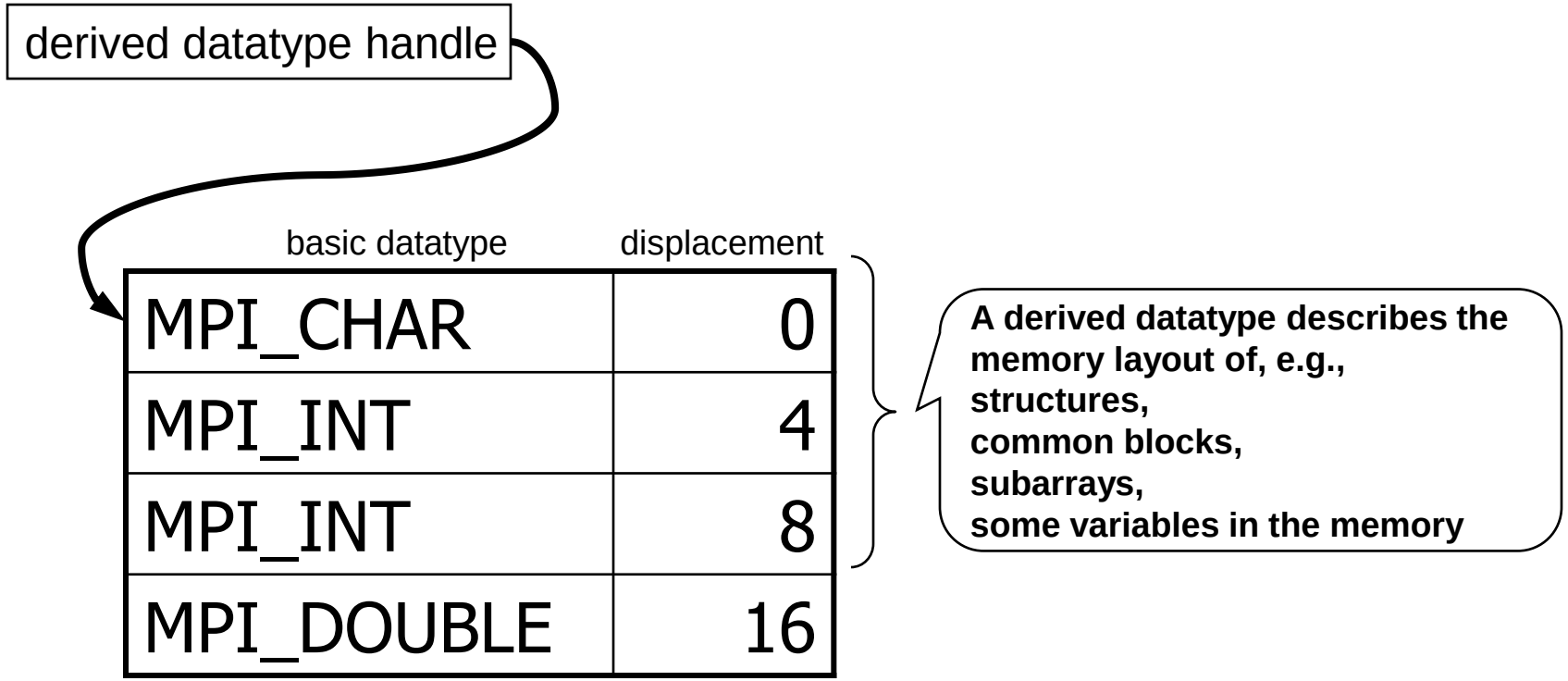
提交数据类型

- 当一数据类型的handle用在消息通信中之前，需要调用**MPI_TYPE_COMMIT**.
- 调用且只能调用一次
- C: int MPI_Type_commit(MPI_Datatype ***datatype*);

派生数据类型逻辑上是指向一系列入口的一指针:

Example:





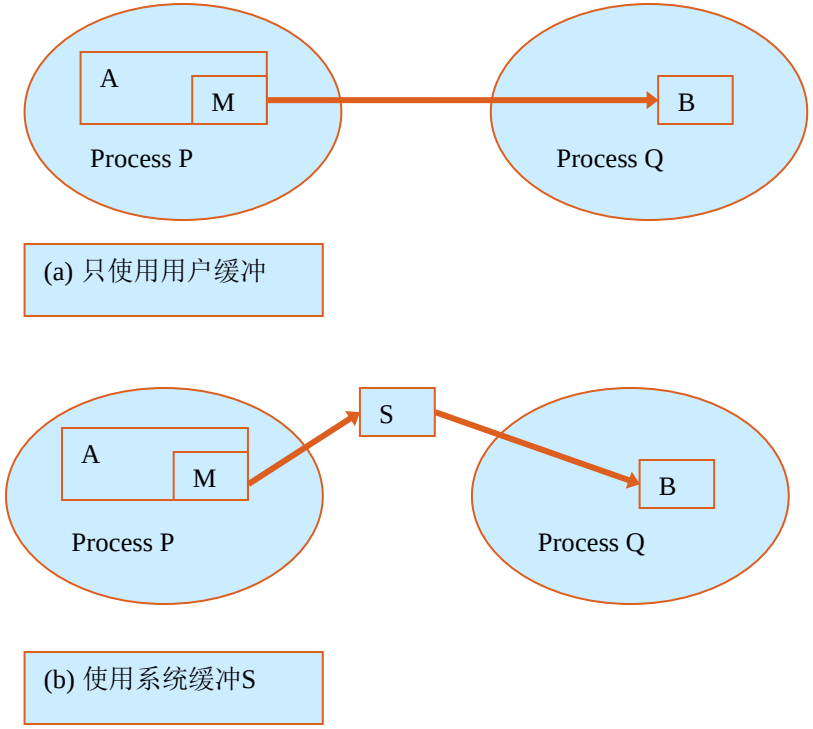
MPI中的消息

- 消息缓冲(message buffer, 简称buffer), 在不同的消息传递使用场合有不同的含义:
  - 消息缓冲指的是由程序员定义的应用程序的存储区域, 用于存放消息的数据值
  - 消息缓冲也可以由消息传递系统(而非用户)创建和管理的一些内存区, 它用于发送消息时暂存消息. 这种缓冲称为系统消息缓冲(或系统缓冲).
  - MPI允许用户可以划出一定大小的内存区, 作为出现在其应用中的任意消息的中间缓冲.

Process P:
double A[2000000];
send(A,32,Q,tag);

Process Q:
double B[32];
recv(B,32,P,tag)

由进程P传送一个存放在数组A中的消息M, 到进程Q的数组B中.



MPI中的消息信封

MPI_Send (address, count, datatype, *destination, tag, communicator*)

destination 域是一个整数, 标识消息的接收进程.

消息标签(message tag), 也称为消息类型(message type), 是程序员用于标识不同类型消息、限制消息接收者的一个整数.

MPI中的消息信封

- message-tag取值范围介于0与MPI_TAG_UB之间.
- **标签作用：**
  - **上面代码传送A的前32个字节给X，传送B的前16个字节给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) send(B, 16, Q)	recv(X, 32, P) recv(Y, 16, P)

使用了标签

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

在消息传递中使用标签

未使用标签

```
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, 分别发送服务请求消息给服务进程Q.

使用了标签

```
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;
      }
```

MPI中的消息信封

Process 0:

MPI_Send(msg1, count1, MPI_INT, 1, tag1, comm1);

parallel_fft(...);

Process 1:

MPI_Recv(msg1, count1, MPI_INT, 0, tag1, comm1);

parallel_fft(...);

包含如下代码

包含如下代码

if (my_rank==0) MPI_Send(msg2, count1, MPI_INT,1,tag2,comm2);

- 存在问题: 无法确保tag1 和tag2取了不同的值:
  - 标签是由用户定义的整数值, 用户可能会出错.
  - 或者难以确保tag1 、tag2取不同的值. 例如：当函数parallel_fft( )是库函数或由其它用户编写时， 当前用户可能不知道tag2的值.
  - 或者MPI_Recv可能使用通配(wildcard)标签MPI_Any_tag.
- 解决办法: 在parallel_fft( )中的通信使用不同的通信子, 它可能包含相同的进程组(如, 进程0和1), 但每个通信子有系统指定的不同的上下文, 与comm1的不同. 因此, MPI_Recv 不会有从parallel_fft( )的MPI_Send中接收msg2的风险.

Wildcard

- MPI允许通配参数（wildcard argument） for both source and tag.
- 如果source设为MPI_ANY_SOURCE，通信域任何进程可以是消息的源.
- 若Tag设为MPI_ANY_TAG，任何tag的消息都可接收.
- 在接收端，消息长度必须不大于长度域指定的长度

MPI中的消息信封

消息传递中的状态(Status)字

当一个接收者能从不同进程接收不同大小和标签的信息时, 状态信息就很有用.

```
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;
        }
}
```

MPI中的消息信封

MPI-1只支持组内通信 (intra-communication) :

MPI-1被设计成使不同通信子中的通信是相互分开的, 以及任何群集通信是与任何点对点通信分开的, 即使它们是在相同的通信子内. 通信子概念尤其方便了并行库的开发.

MPI-2支持组间通信 (inter-communication)

MPI消息特性总结

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

例子:

MPI_Send(&N, 1, MPI_INT, i, i , MPI_COMM_WORLD);

- 第二个参数指明消息中给定的数据类型有多少项, 这个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, 要么是派生数据类型, 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.
- 第五个是消息标签
- 第六个参数标识进程组和上下文, 即, 通信子. 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_Recv(address, count, datatype,source, tag, communicator, status)

例：

MPI_Recv(&tmp, 1, MPI_INT, i, i, MPI_COMM_WORLD,&Status)

第五个是消息标签
第六个参数标识一个通信子 } → 这两个域可以是wildcard MPI_Any_source和 MPI_Any_tag.
第七个参数是一个指针, 指向一个结构

MPI_Status Status

存放了各种有关接收消息的各种信息.

Status.MPI_SOURCE 实际的源进程编号

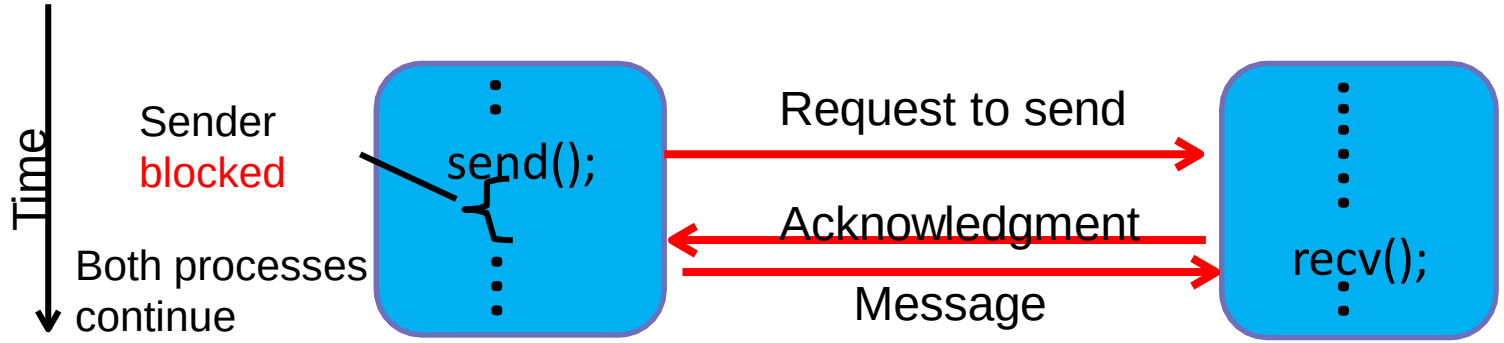
Status.MPI_TAG 实际的消息标签

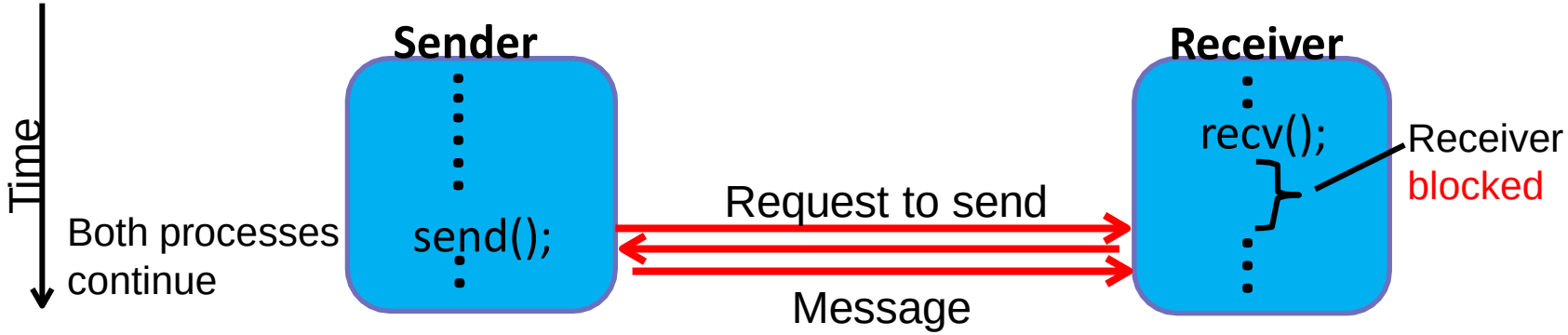
实际接收到的数据项数由MPI例程

MPI_Get_count(&Status, MPI_INT, &C)

读出. 这个例程使用Status中的信息来决定给定数据类型(在这里是MPI_INT)中的实际项数, 将这个数放在变量C中.

点对点通信(通信模式)


Time
Sender
blocked
send();
Request to send
Acknowledgment
Message
recv();
Both processes
continue


Time
Sender
Receiver
send();
recv();
Receiver
blocked
Request to send
Message
Both processes
continue

通信模式

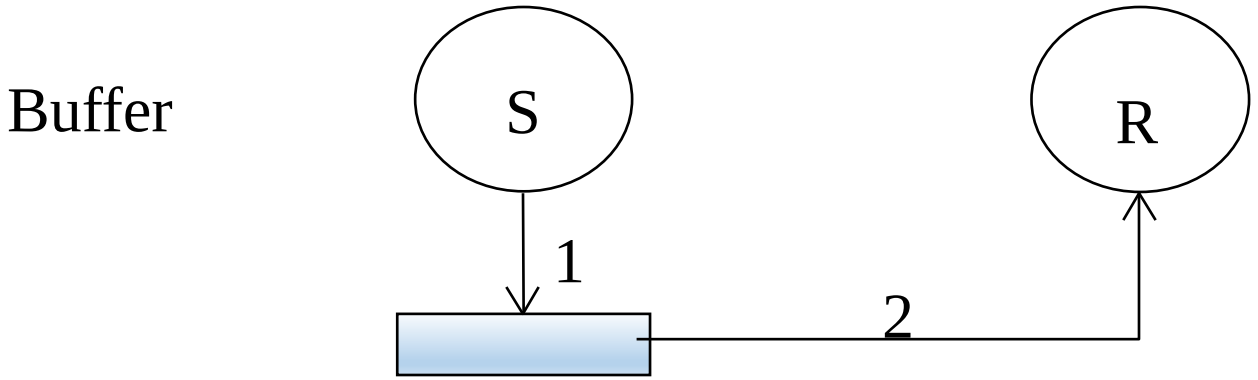
Mode	**Start**	**Completion**
Standard (MPI_Send)	Before or after recv	Before recv (buffer) or after (no buffer)
Buffered (MPI_Bsend) (Uses MPI_Buffer_Attach)	Before or after recv	Before recv
Synchronous (MPI_Ssend)	Before or after recv	Particular point in recv
Ready (MPI_Rsend)	After recv	After recv

点对点通信(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。
  - 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。
- 注意: 在MPI中可以有非阻塞的同步发送, 它的返回不意味着消息已经被发出。它的实现不需要在接收端有附加的缓冲, 但需要在发送端有一个系统缓冲. 若要消除额外的消息拷贝, 可使用阻塞的同步发送.

点对点通信(通信模式)

- **缓存通信模式**：缓存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_attch实现，通过MPI_Buffer_detach来回收申请的缓冲区。



点对点通信(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

点对点通信(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。

点对点通信要求

- 发送者必须指定有效的目标rank.
- 接收者必须指定有效的源rank.
- Communicator必须一样.
- Tag必须匹配.
- 消息数据类型必须匹配.
- Receiver的buffer必须足够大.

阻塞通信

- 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。
- 死锁

死锁

考虑:

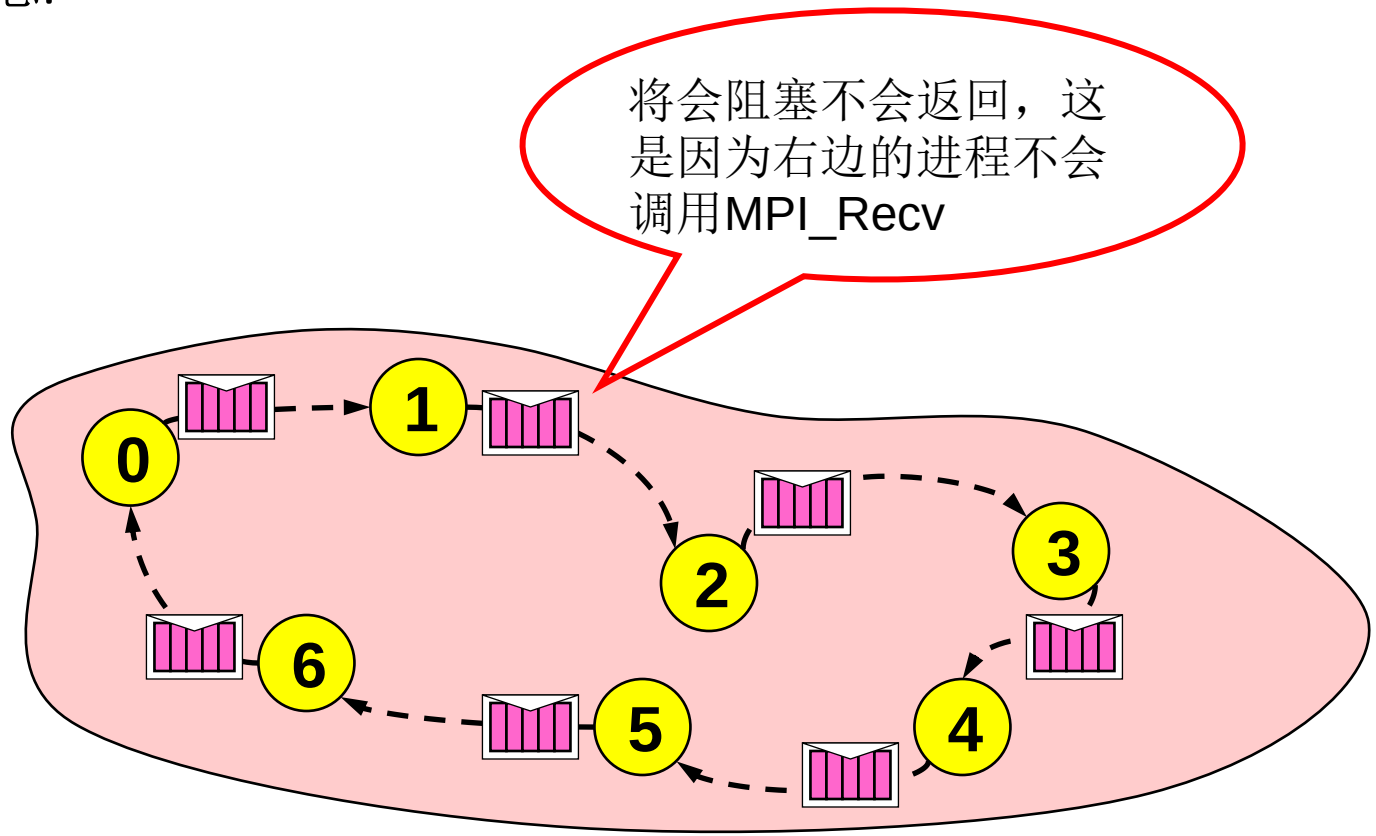
```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

若MPI_Send阻塞，会发生死锁.

死锁

进程i发送消息给进程$i+1$并从进程$i-1$接收消息.

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD,
&npes);
MPI_Comm_rank(MPI_COMM_WORLD,
&myrank);
--modulo the number of processes
MPI_Ssend(a, 10, MPI_INT,
(myrank+1)%npes, 1,
      MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-
1+npes)%npes, 1,  MPI_COMM_WORLD);
...
```



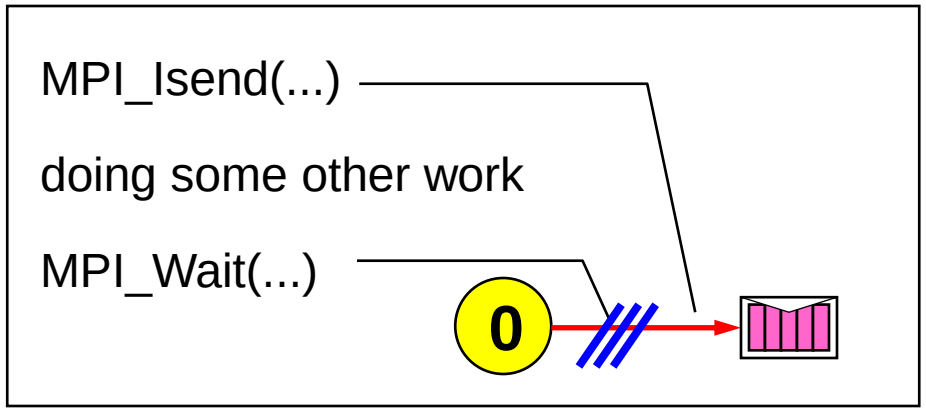
如果MPI实现采用同步通信方式，对于标准发送模式(MPI_Send),同样情况也会发生

非阻塞通信

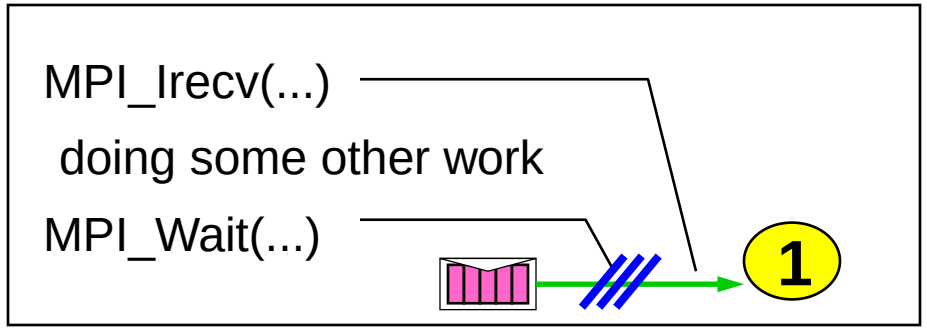
- 启动非阻塞通信
  - 立即返回
- 执行其它代码
  - “latency hiding”
- 等待非阻塞通信完成
- 函数名字以MPI_I开头

非阻塞示例

- Non-blocking send



- Non-blocking receive

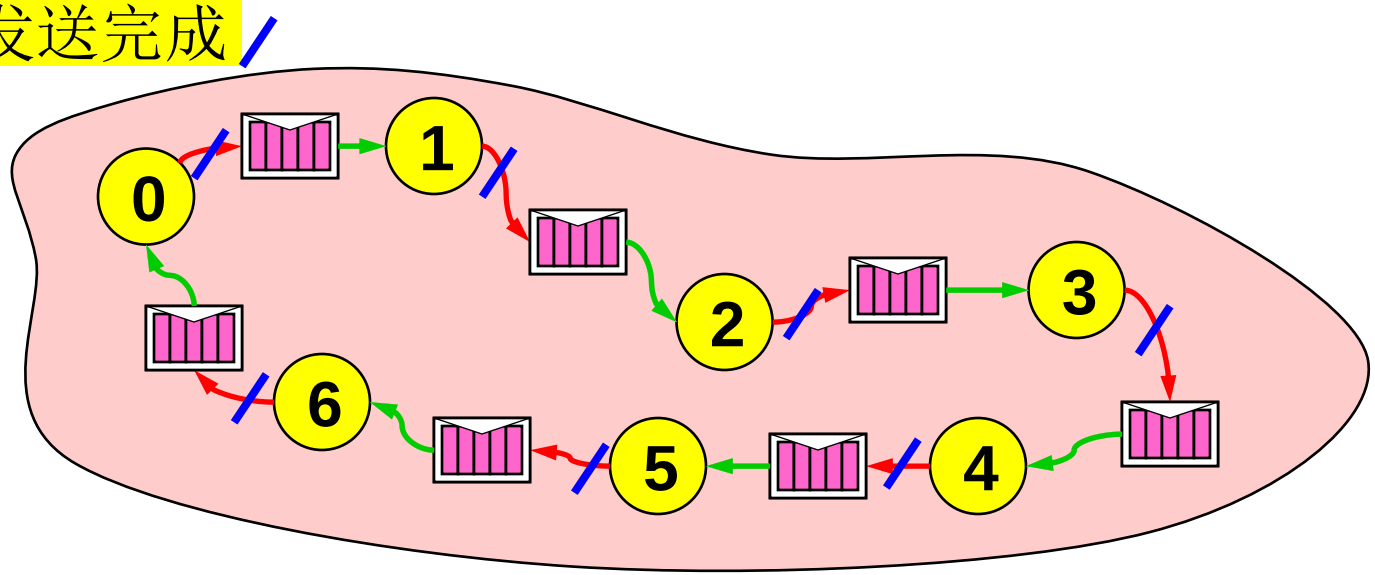


= waiting until operation locally completed

非阻塞发送

- 启动非阻塞发送
  → 以上述环为例: 向右边邻居非阻塞发送
- 执行其它工作:
  → 以环发送代码为例: 从左边邻居接收消息

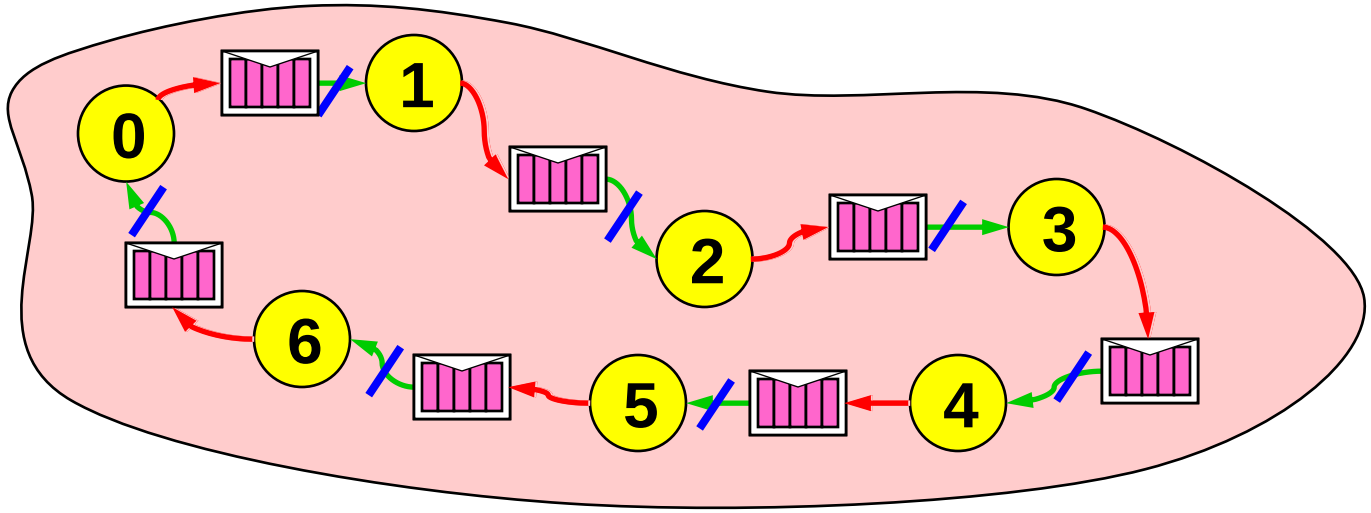
- 等待非阻塞发送完成



非阻塞接收Non-Blocking Receive

- 启动非阻塞接收
  → 从左边邻居启动非阻塞接收
- Do some work:
  → 向右边邻居发送消息

- 等待非阻塞接收完成 /



Non-blocking Synchronous Send

- C:

```
MPI_Issend(buf, count, datatype, dest, tag, comm,
           OUT &request_handle);

MPI_Wait(INOUT &request_handle, &status);
```

- buf must not be used between Issend and Wait
- "Issend + Wait directly after Issend" is equivalent to blocking call (Ssend)
- status is not used in Issend, but in Wait (with send: nothing returned)

Non-blocking Receive

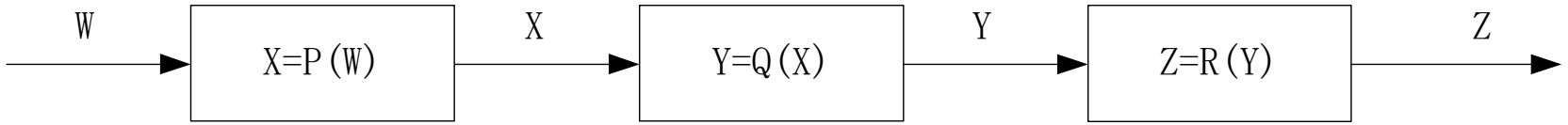
- C:

```
MPI_Irecv(buf, count, datatype, source, tag, comm,
          OUT &request_handle);

MPI_Wait(INOUT &request_handle, &status);
```

- buf must not be used between Irecv and Wait (in all progr. languages)

idle

- 非阻塞通信可以用来降低闲置时间。
- 一条三进程的流水线，一个进程连续地从左边的进程接收一个输入数据流，计算一个新的值，然后将它发送给右边的进程。



进程流水线中的数据流

```
while (Not_Done){
        MPI_Irevc(NextX, … );
        MPI_Isend(PreviousY, … );
        CurrentY=Q(CurrentX);
}
```

Blocking和Non-Blocking

- 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。

Blocking和Non-Blocking

- Send和receive可以是阻塞也可以是非阻塞.
- 阻塞发送可以与非阻塞接收一起使用，反之亦然.
- 非阻塞发送可以使用任何通信模式
  - standard – MPI_ISEND
  - synchronous – MPI_ISSEND
  - buffered – MPI_IBSEND
  - ready – MPI_IRSEND
- 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。
- MPI的接收操作只有两种：阻塞接收和非阻塞接收。

通信完成

- 非阻塞通信返回后并不意味着通信操作的完成，MPI提供了对非阻塞通信完成的检测：MPI_Wait函数和MPI_Test函数。
- C:

  MPI_Wait(&request_handle, &*status*);它直到Handle指示的发送或接收操作已经完成才返回 。

  MPI_Test(&request_handle, &*flag*, &*status*);测试由Handle指示的发送或接收操作是否完成，如果完成，就对Flag赋值True，这个函数不像MPI_Wait，它不会被阻塞。

  send_handle和revc_handle分别用于检查发送接收是否完成。

示例

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
  int x;
  MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
  compute();
  MPI_Wait(req1, status);
} else if (myrank == 1) {
  int x;
  MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

Multiple Non-Blocking Communications

May have several request handles:

- Wait or test for completion of **one** message
  - **MPI_Waitany / MPI_Testany**
- Wait or test for completion of **all** messages
  - **MPI_Waitall / MPI_Testall**
- Wait or test for completion of **as many** messages as possible
  - **MPI_Waitsome / MPI_Testsome**

点对点通信

- 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

同时发送和接收消息

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

参数包括给send和receive的参数。若想对send和receive用同样的buffer:

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

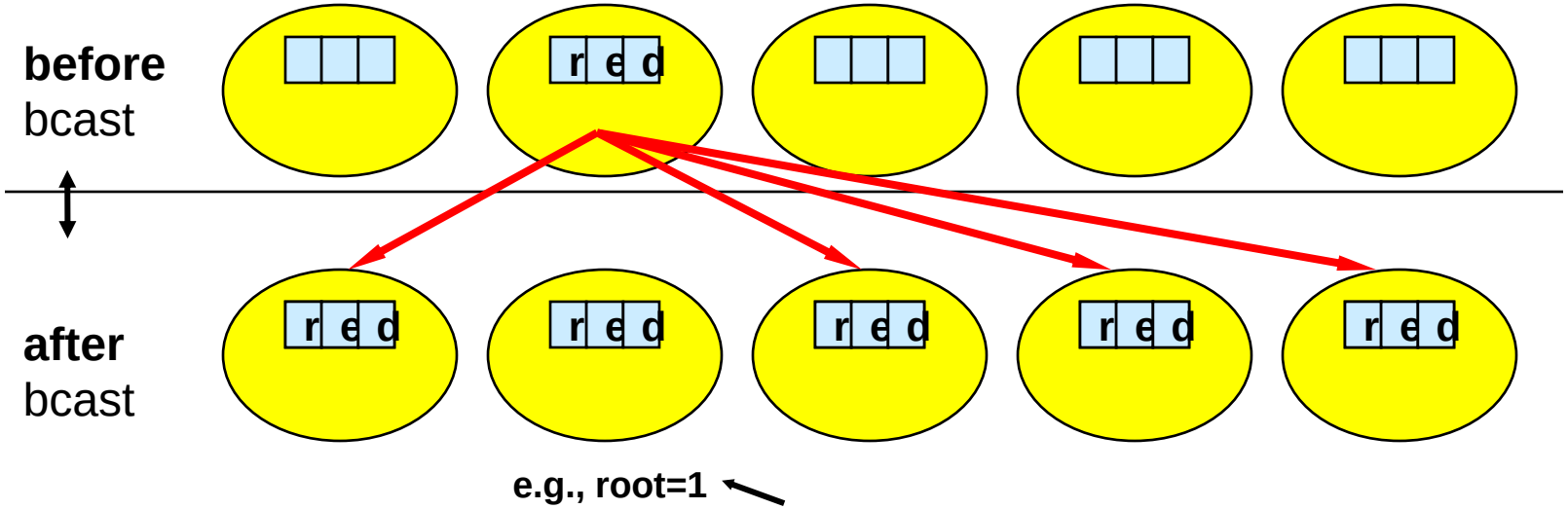
Collective Communications

群通信/组通信

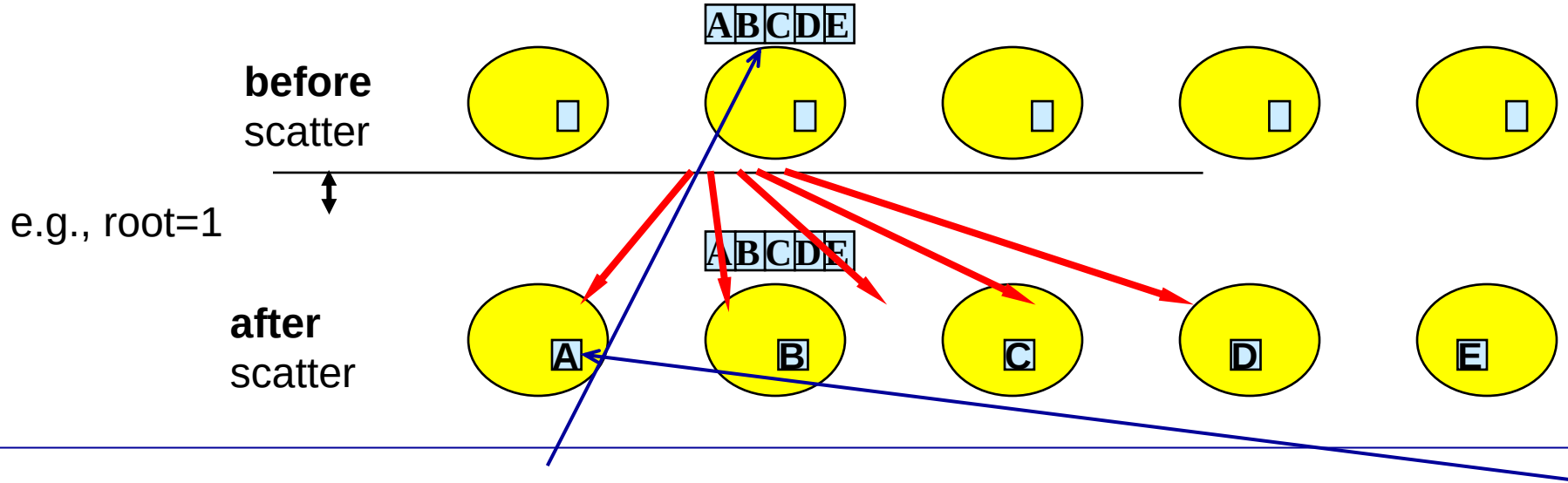
- 群通信虽然可以通过点对点通信实现，但群通信效率更高
- Only blocking; standard mode; no tags
- Some collectives have "root"s
- Different types
  - One-to-all
    - MPI_BCAST
    - MPI_SCATTER, MPI_SCATTERV
  - All-to-one
    - MPI_GATHER, MPI_GATHERV
    - MPI_REDUCE
  - All-to-all
    - MPI_ALLGATHER, MPI_ALLGATHERV
    - MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
    - MPI_ALLREDUCE, MPI_REDUCE_SCATTER

Broadcast

- 发送同样消息给所有相关进程
  Multicast – 发送同样的消息到指定的一组进程
- int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm)



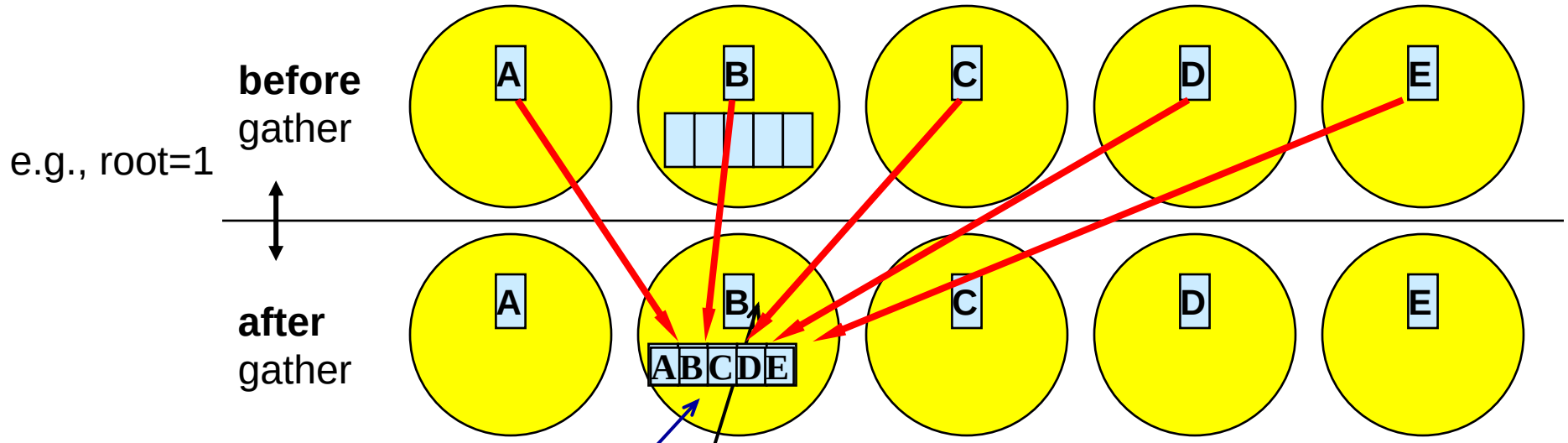
Scatter



C: int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void ***recvbuf***,

int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

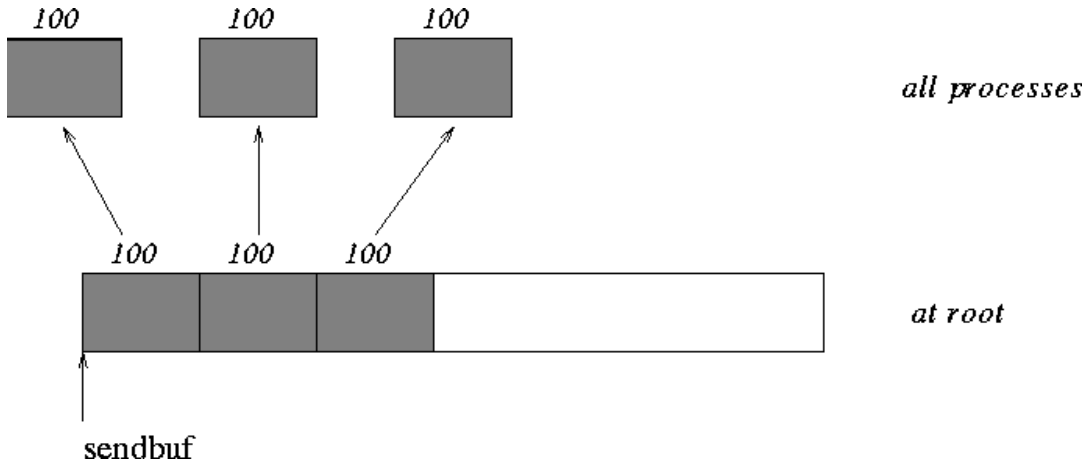
Gather



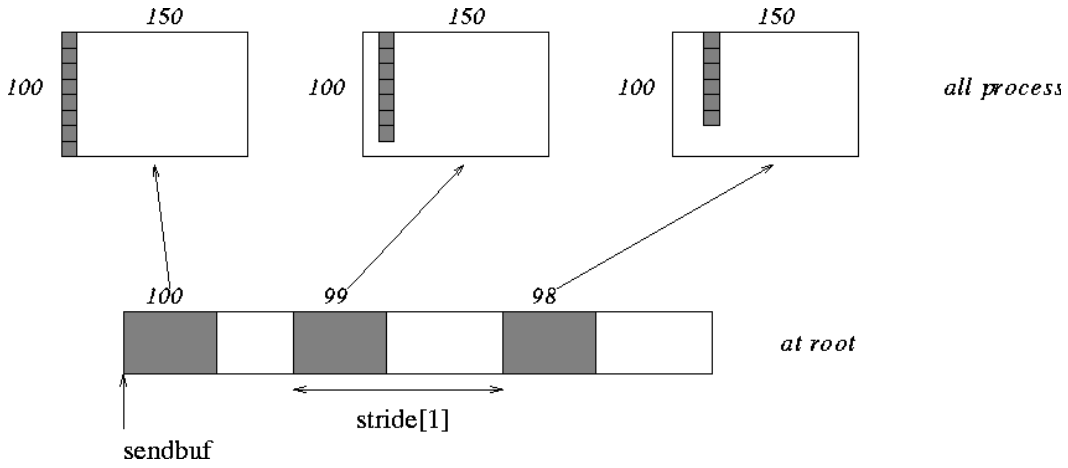
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void ***recvbuf**, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Scatter & Gather

MPI_SCATTER



MPI_SCATTERV



MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
MPI_SCATTERV(sendbuf, array_of_sendcounts, array_of_displ, sendtype, recvbuf, recvcount, **recvtype, root, comm)**

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

Collective Communications - Reduce

- MPI_Reduce(SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm)
- 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。

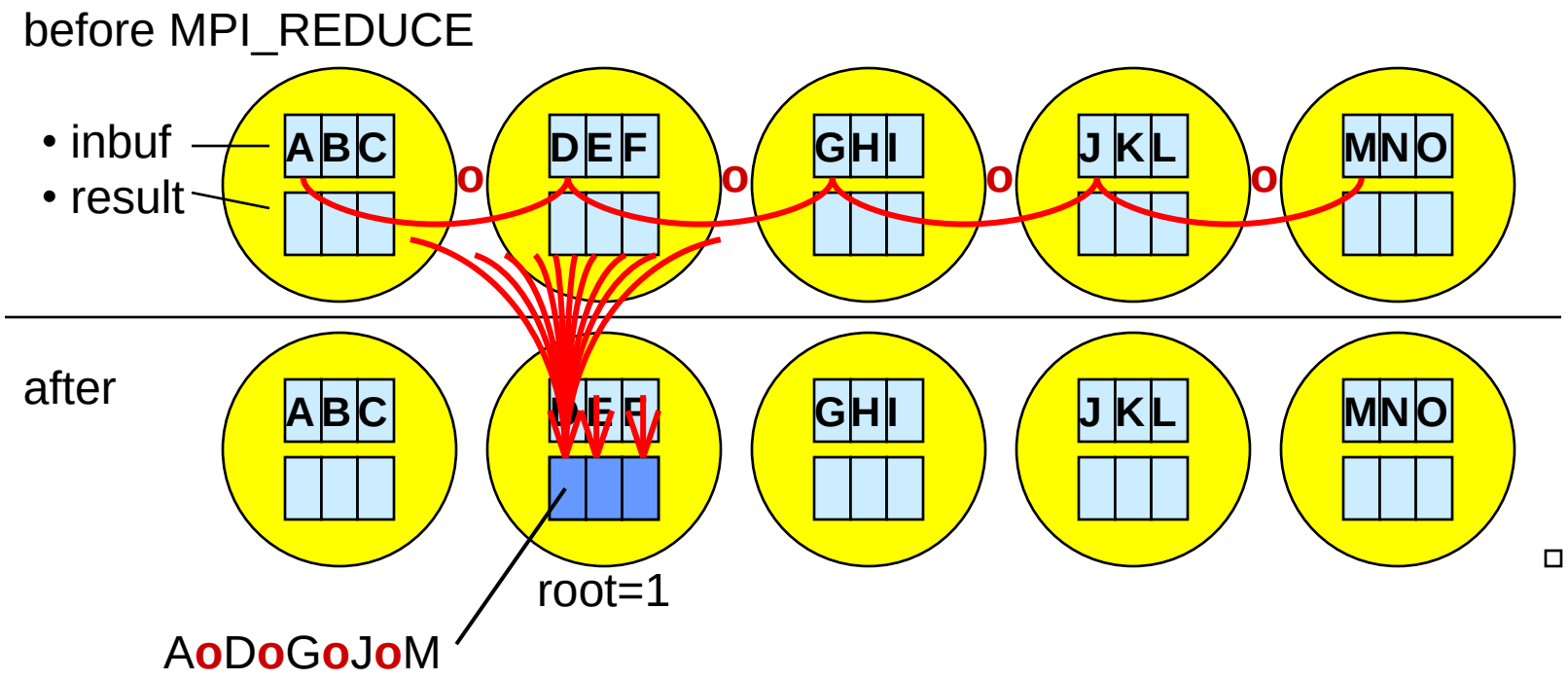
Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - d_i = data in process rank I (single variable, or vector)
  - **o** = associative operation
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation

Predefined Reduction Operation Handles

Predefined operation handle	**Function**
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

MPI_REDUCE


before MPI_REDUCE
• inbuf
• result
A B C
o
D E F
o
G H I
o
J K L
o
M N O
after
A B C
D E F
G H I
J K L
M N O
root=1
AoDoGoJoM

Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf.*
- C: root=0;
  MPI_Reduce(&inbuf, &*resultbuf*, 1, MPI_INT, MPI_SUM, root,
  MPI_COMM_WORLD);
- The result is only placed in *resultbuf* at the root process.

用户自定义归约函数

- 用户定义操作 ■:
  - 可结合
  - user-defined function must perform the operation vector_A ■ vector_B
  - 语法符合标准
- 登记用户自定义函数:
  - C: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)
  - COMMUTE声明FUNC是否可交换.

Example

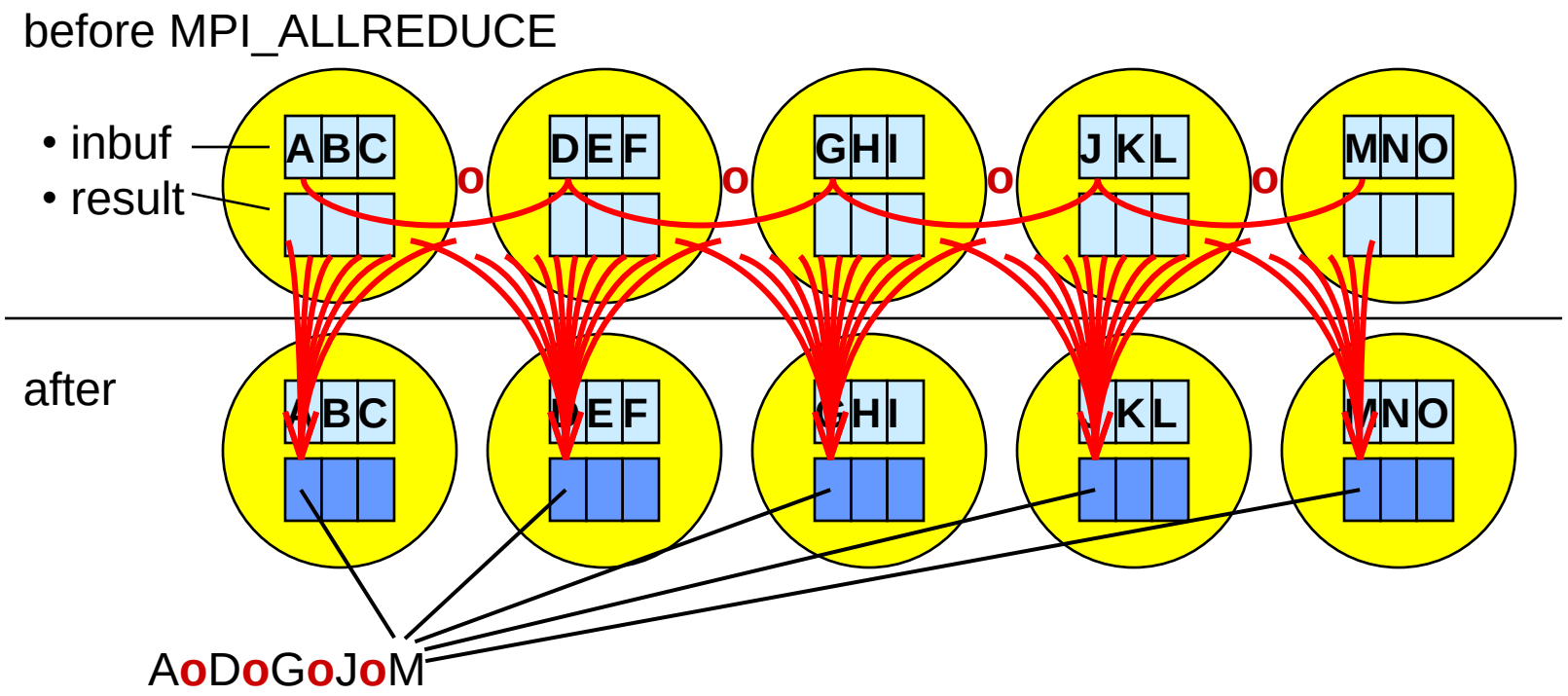
```
typedef struct {
double real, imag;
} Complex

Complex   a[100], answer[100];
MPI_Op  myOp
MPI_Datatype ctype;
MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype);
MPI_Type_commit( &ctype );
MPI_Op_create(myProd, True, &  myOp );
MPI_Reduce( a, answer, 100, ctype, myOp, root,  comm );
```

```
void   myProd  (Complex  *in, Complex *inout,  int *len, MPI_Datatype
   *dptr )
   {
            int i;
            Complex   c;
            for (i=0; i< *len; ++i) {
                 c.real = inout->real*in->real - inout->imag*in->imag;
                 c.imag = inout->real*in->imag + inout->imag*in->real;
                 *inout = c;
                 in++; inout++;
            }
   }
```

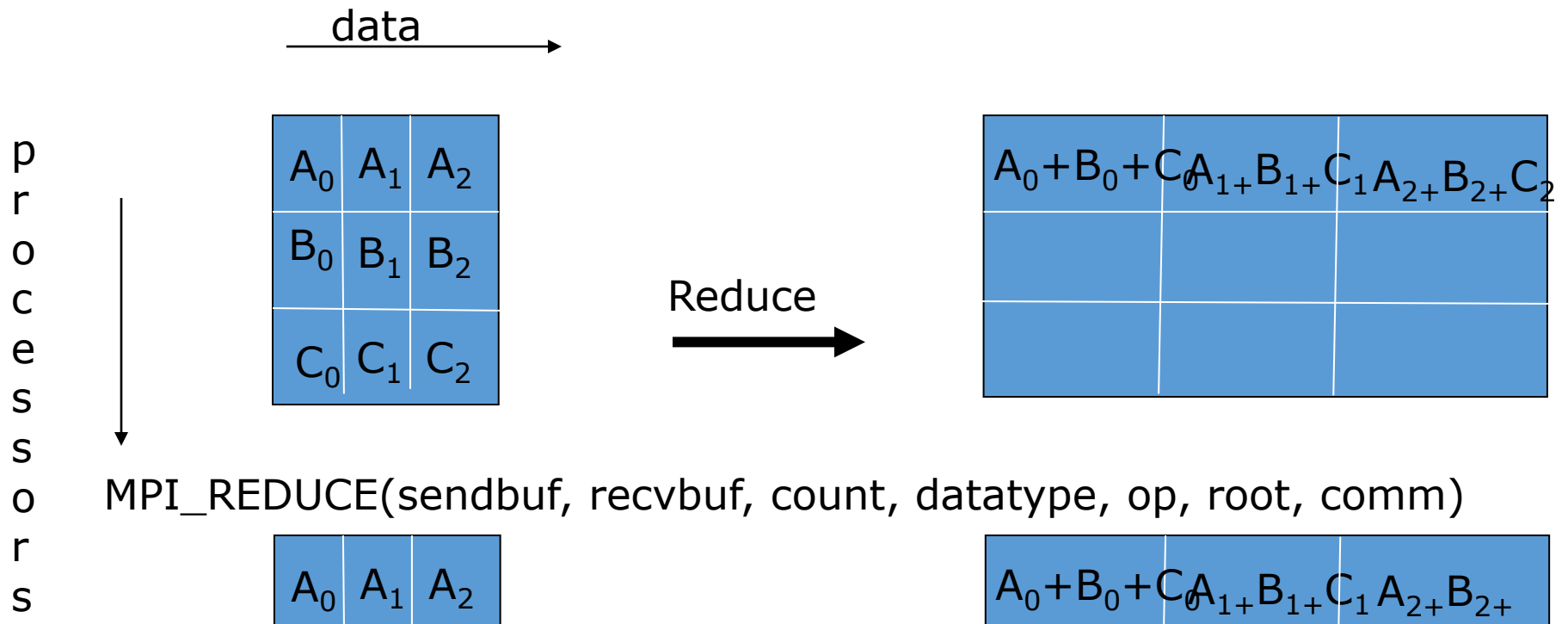
Variants of Reduction Operations

- MPI_ALLREDUCE
  - no root,
  - returns the result in all processes
- MPI_REDUCE_SCATTER
  - result vector of the reduction operation
    is scattered to the processes into the real result buffers
- MPI_SCAN
  - prefix reduction
  - result at process with rank i :=
    reduction of inbuf-values from rank 0 to rank i

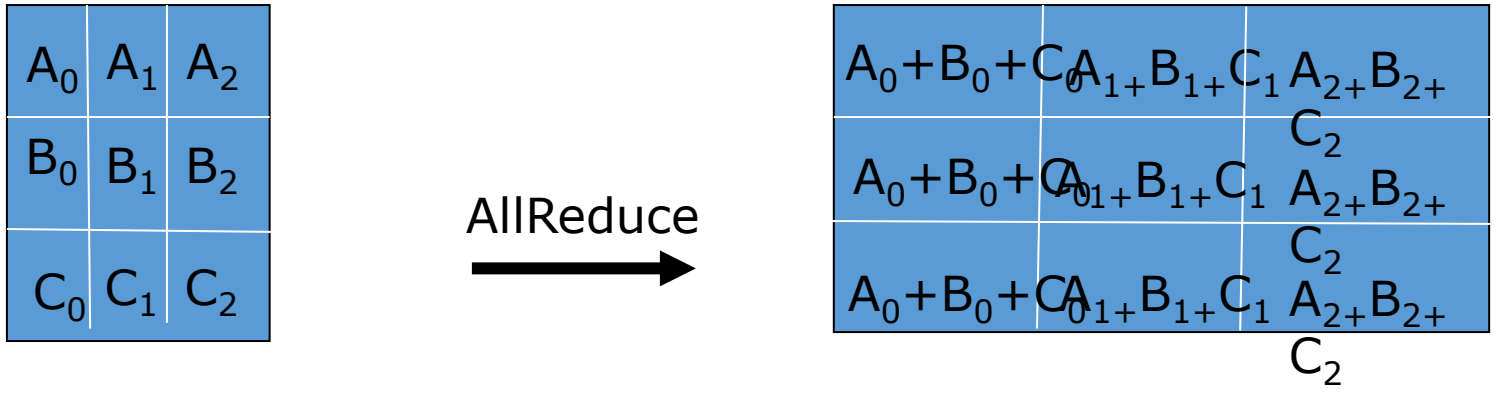
MPI_ALLREDUCE



Reduce vs Allreduce



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

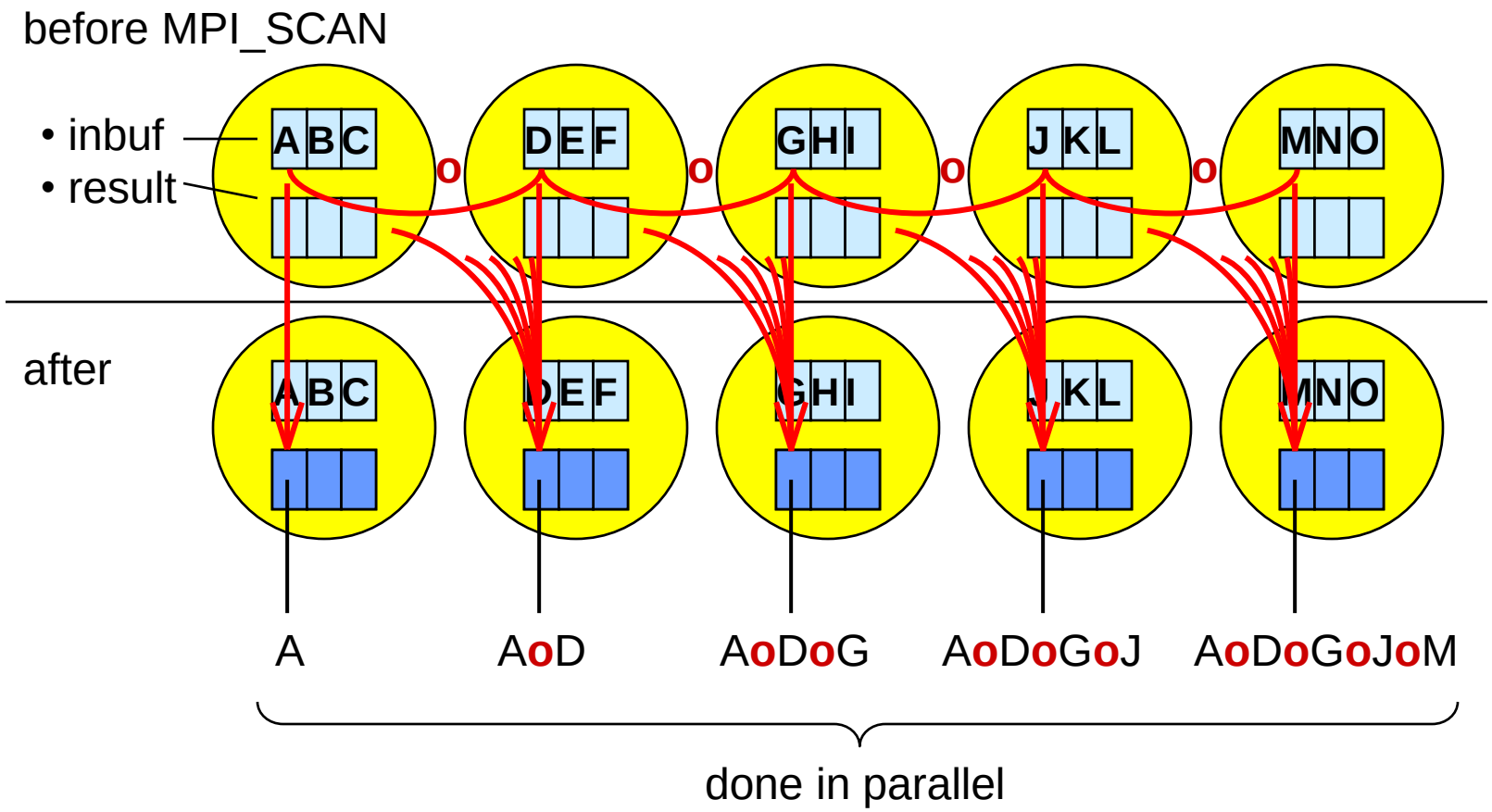


MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

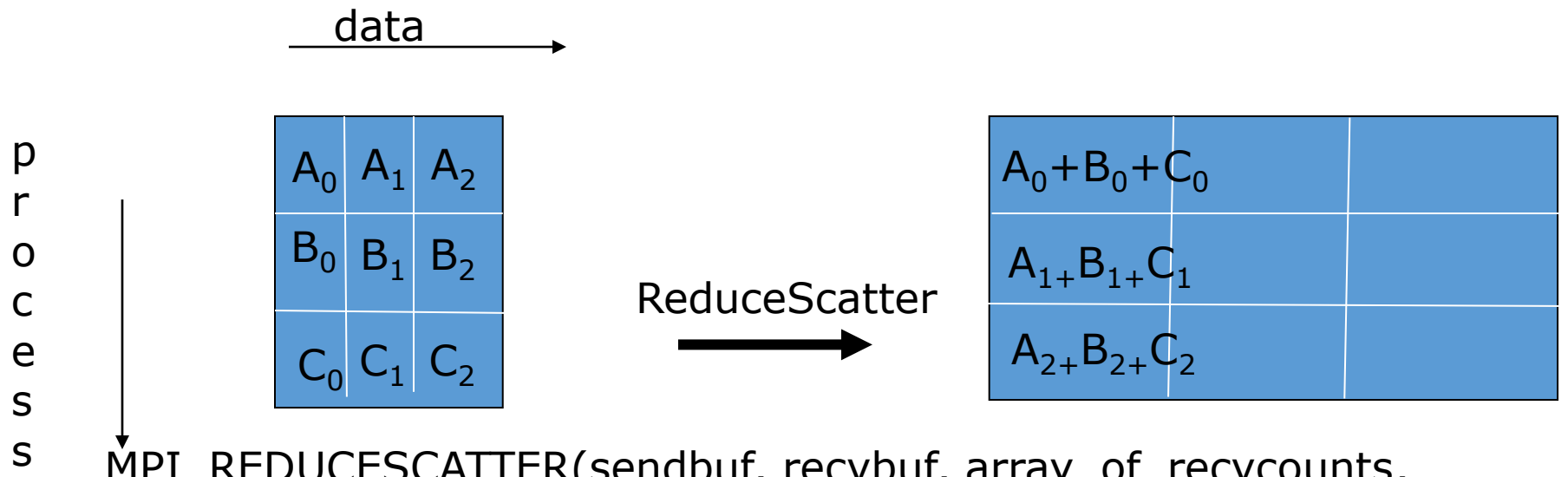
Collective Communications - Scan

- 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。

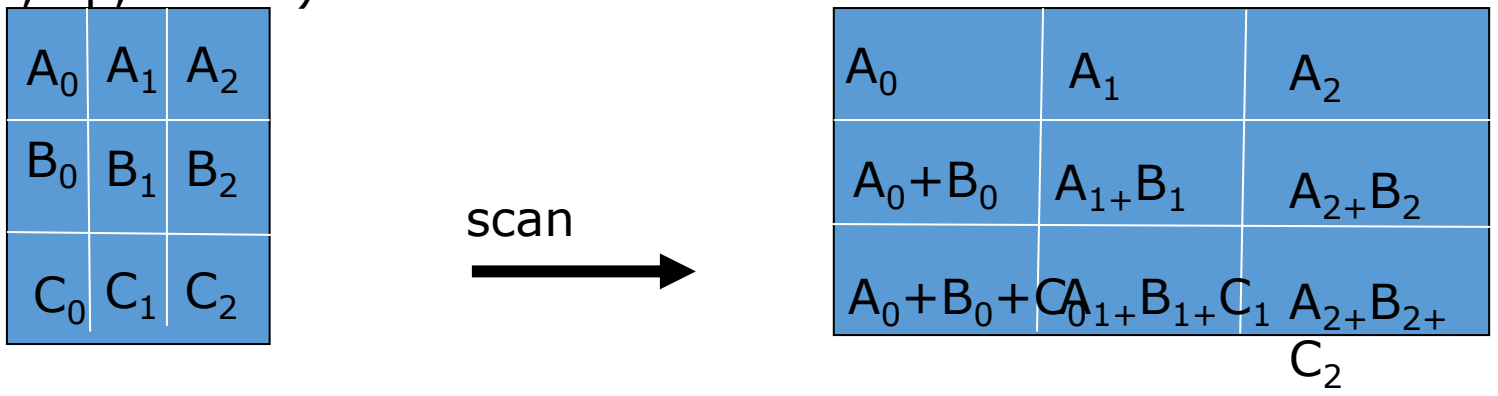
MPI_SCAN


before MPI_SCAN
• inbuf
• result
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
D E F
G H I
J K L
M N O
A
AoD
AoDoG
AoDoGoJ
AoDoGoJoM
done in parallel

Collective Communications – ReduceScatter, Scan



MPI_REDUCESCATTER(sendbuf, recvbuf, array_of_recvcounts, datatype, op, comm)



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

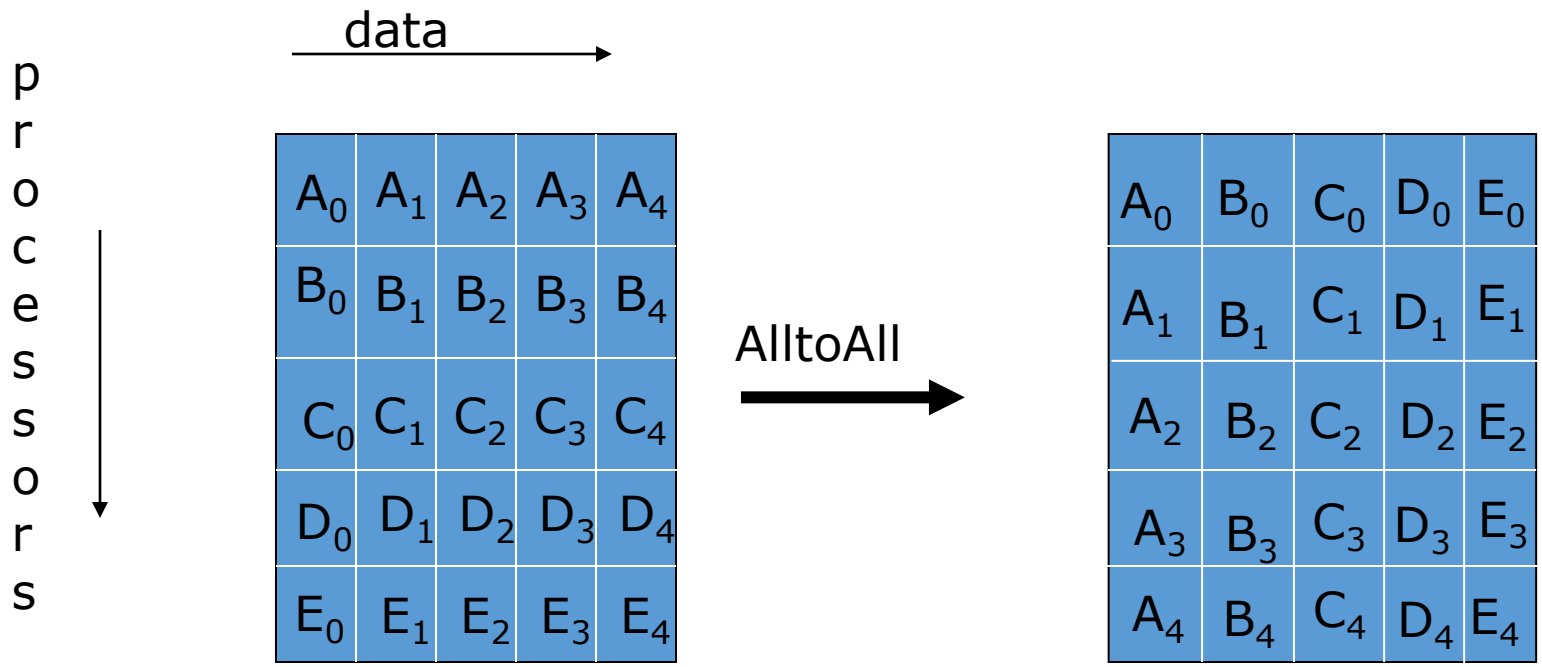
Example: 矩阵-向量乘

```
/* Summing the dot-products */
MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);

/* Now all values of x is stored in process 0. Need to scatter them */
MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0, comm);
```

Collective Communications – All-to-one

- 收集
  - 在收集操作中，Root进程从进程域Comm的所有进程(包括它自己)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<RecvAddress, RecvCount, RecvDatatype>标识，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。

Collective Communications – AlltoAll



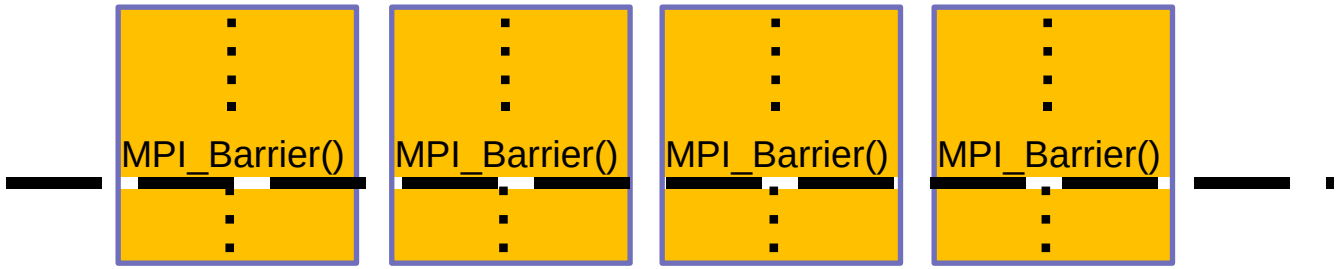
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

MPI_ALLTOALLV(sendbuf, array_of_sendcounts, array_of_displ, sendtype, array_of_recvbuf, array_of_displ, recvcount, recvtype, comm)

Collective Communications - AlltoAll

- 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。

Barrier



Blocked until synchronized

C: int MPI_Barrier(MPI_Comm comm)

MPI_Barrier is normally never needed because all synchronization is done automatically by the data communication:

A return from barrier in one process tells the process that the other processes have ***entered*** the barrier.

Collective Communications

Type	Routine	Functionality
Data movement	MPI_Bcast	一对多播送相同的信息
	MPI_Gather	多对一收集个人信息
	MPI_Gatherv	通用的MPI_Gather
	MPI_Allgather	全局收集操作
	MPI_Allgatherv	通用的MPI_Allgather
	MPI_Scatter	一对多播撒个人信息
	MPI_Scatterv	通用的MPI_Scatter
	MPI_Alltoall	多对多全交换个人信息
	MPI_Alltoallv	通用的MPI_Alltoall
Aggregation	MPI_Reduce	多对一归约
	MPI_Allreduce	通用的MPI_Reduce
	MPI_Reduce_scatter	通用的MPI_Reduce
	MPI_Scan	多对多并行prefix
Synchronization	MPI_Barrier	路障同步

Collective Communications

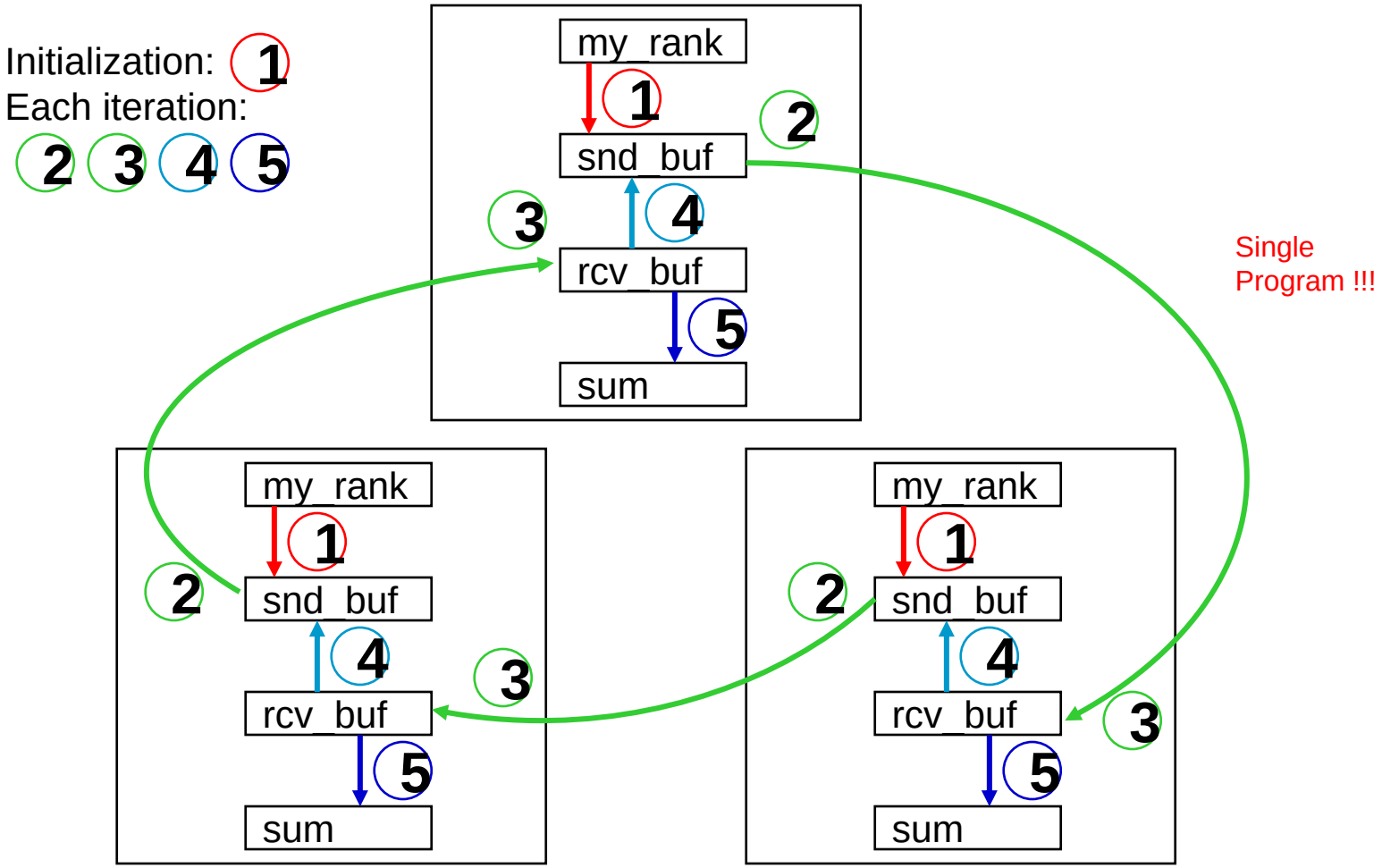
- 通信子中的所有进程必须调用群集通信例程.
  - Count 和Datatype在所有参与进程中须一致.
  - 无tag参数. 消息信封由通信子参数和源/目的进程定义. 例如, MPI_Bcast里, 消息的源是Root, 目的是所有进程(包括Root).
  - 若只有通信子中的一部分成员调用了一群集例程而其它没有调用, 则是错误的. 错误代码可能导致死锁或产生错误的结果等情况.
- 一个进程一旦结束了它所参与的群集操作就从群集例程中返回.
- 除了MPI_Barrier以外, 每个群集例程使用类似于点对点通信中的标准(standard)、阻塞的通信模式.
  - 例如, 当Root进程从MPI_Bcast中返回时, 意味着发送缓冲可以安全地再次使用.
  - 群集例程是否同步取决于实现. MPI要求用户负责保证他的代码无论实现是否是同步的都是正确的.

Example: PI in C

```
#include "mpi.h"

#include <math.h>

int main(int argc, char *argv[])

{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.14159265358979323846264643;
  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;
    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;
}
```

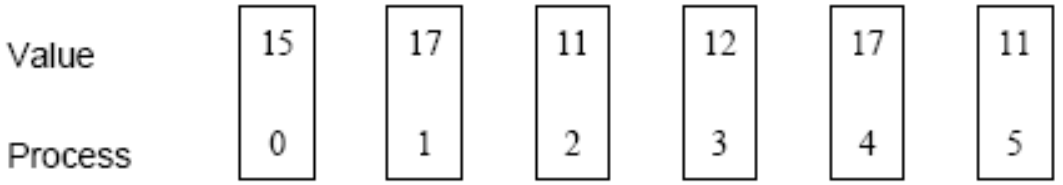
思考

- 用MPI实现：一组进程组成一环，起始时每进程将自己的rank赋给一变量buf，然后将buf值传递给右侧的邻居。进程将收到的值替换buf的值，并计算收到的值之和。程序终止条件可自定



Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.




```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.