天津大学 Tianjin University

国家超级计算天津中心 National SuperComputer Center in Tianjin

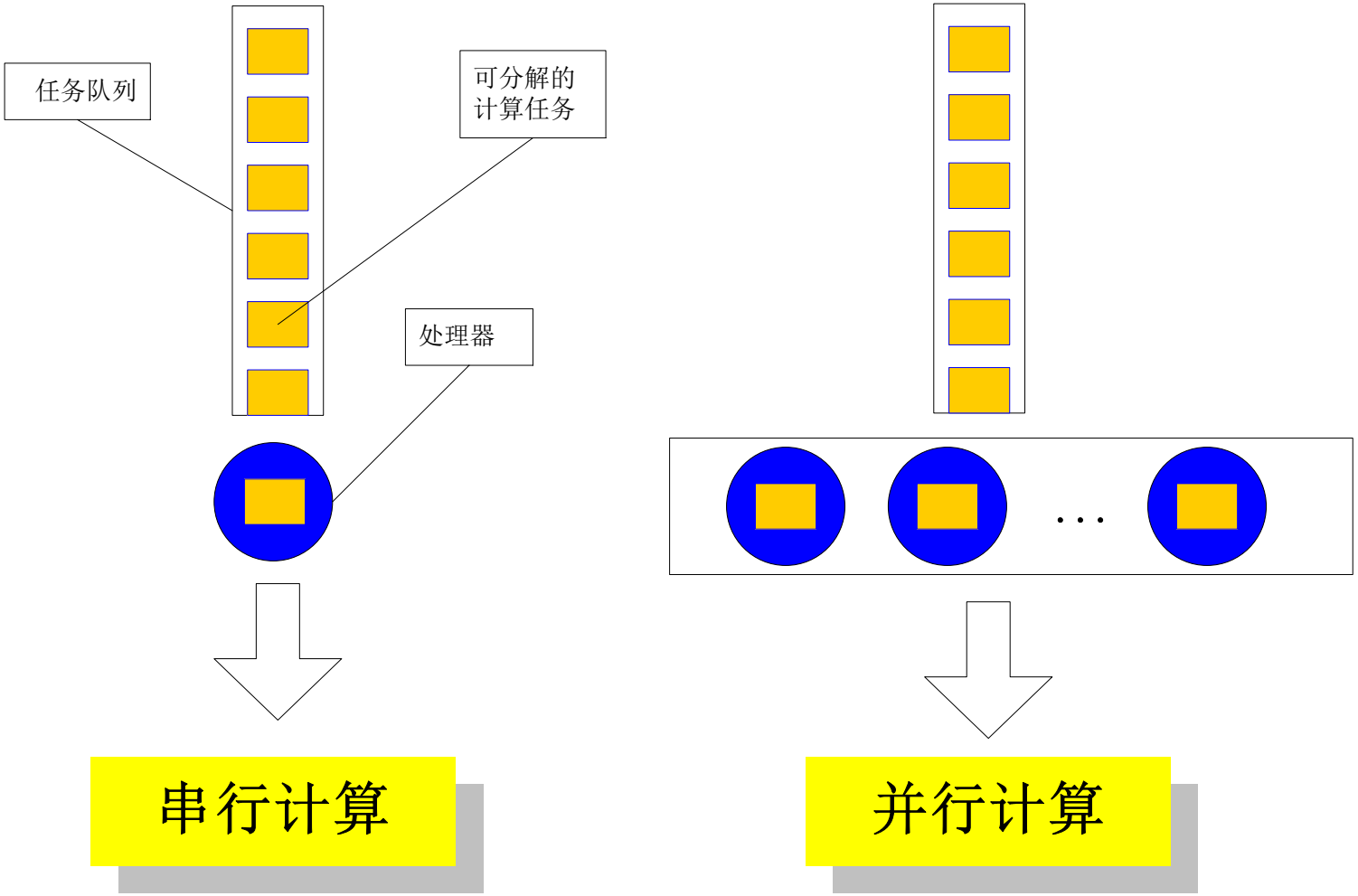
# MPI (Message Passing Interface)

于策

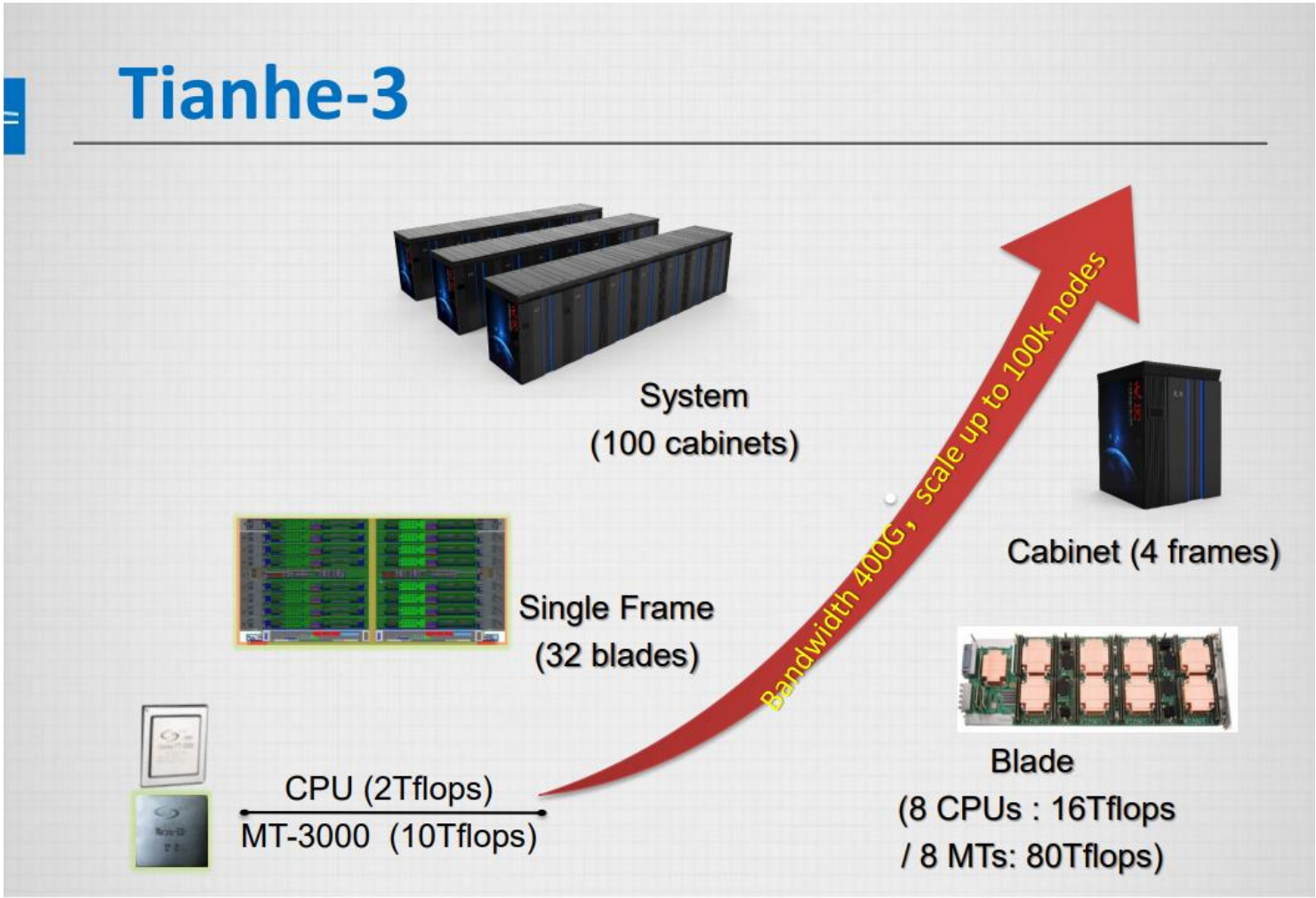
yuce@tju.edu.cn

2019.11.15

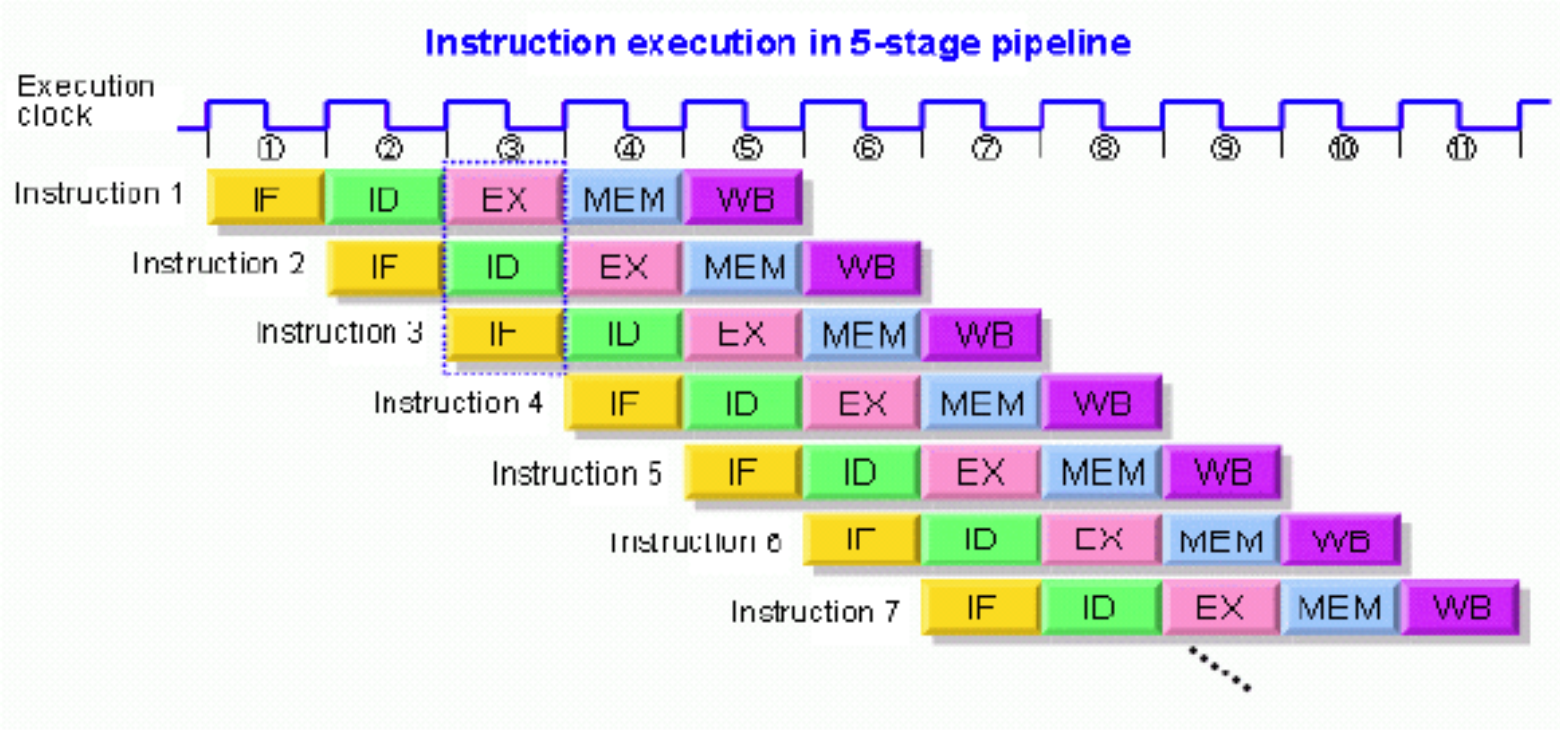
# 预备知识：并行计算



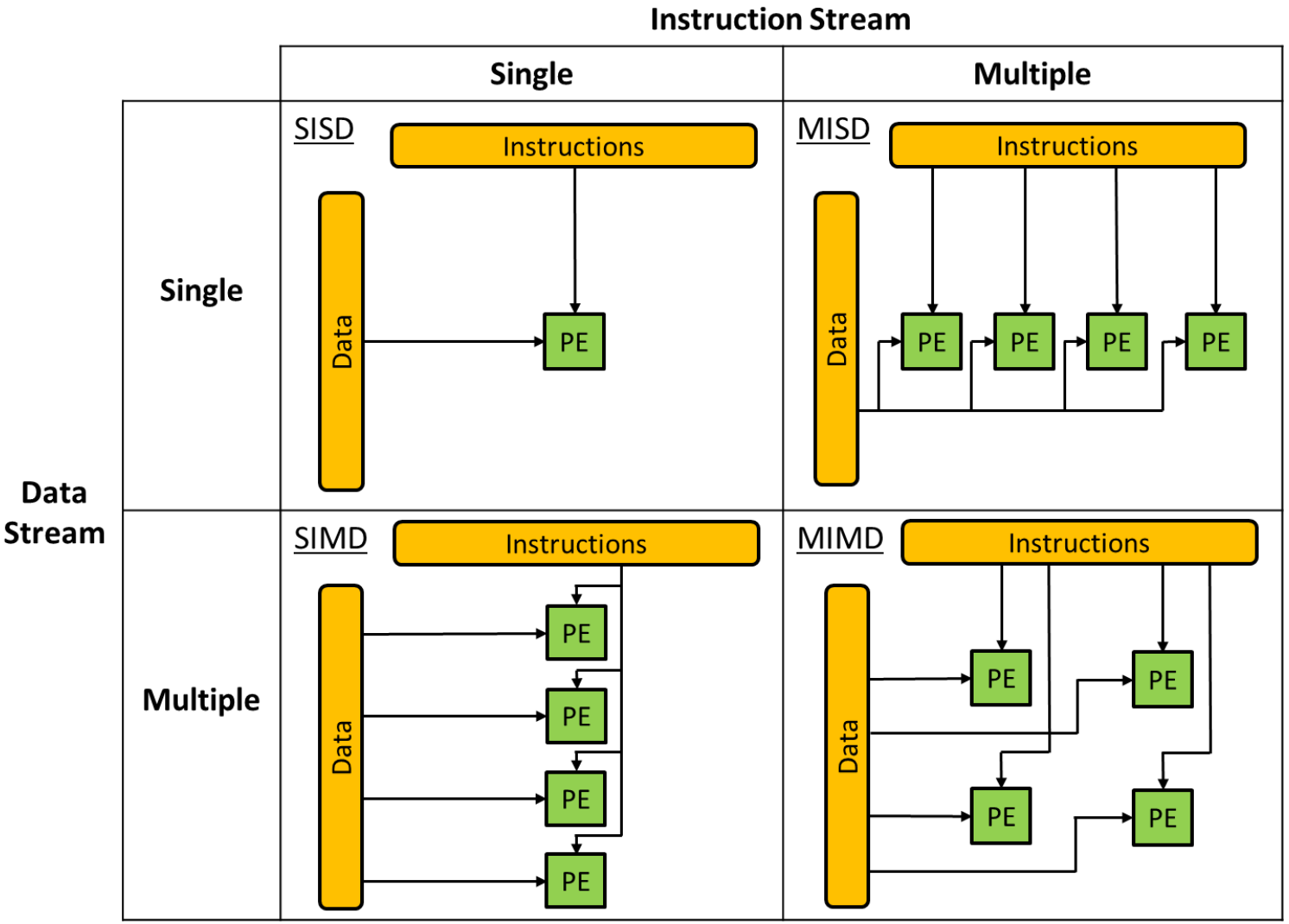
# 并行计算的层次


Tianhe-3
System
(100 cabinets)
Cabinet (4 frames)
Single Frame
(32 blades)
Bandwidth 400G，scale up to 100k nodes
Blade
(8 CPUs : 16Tflops
/ 8 MTs: 80Tflops)
CPU (2Tflops)
MT-3000 (10Tflops)
National University of Defense Technology

# 隐式指令级并行


Instruction execution in 5-stage pipeline
Execution clock
① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
Instruction 7
IF ID EX MEM WB

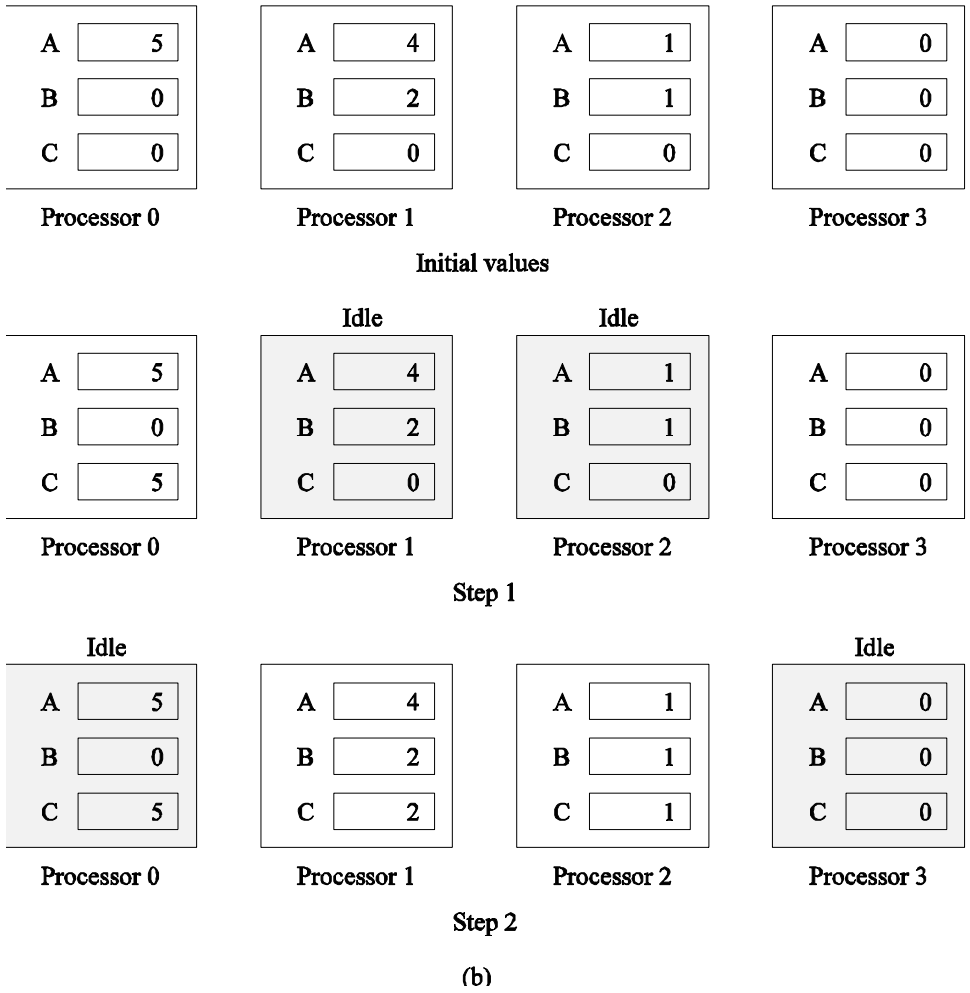
# SIMD


Instruction Stream
Single
Multiple
Data Stream
Single
Multiple
SISD
Instructions
Data
PE
MISD
Instructions
Data
PE
PE
PE
PE
SIMD
Instructions
Data
PE
PE
PE
PE
MIMD
Instructions
Data
PE
PE
PE
PE

# SIMD指令执行

```
if (B == 0)
        C = A;
else
        C = A/B;
```

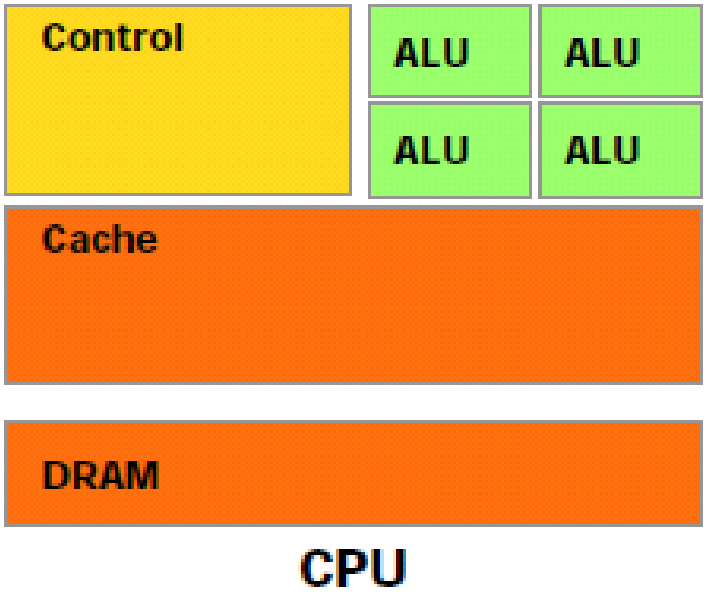
(a)

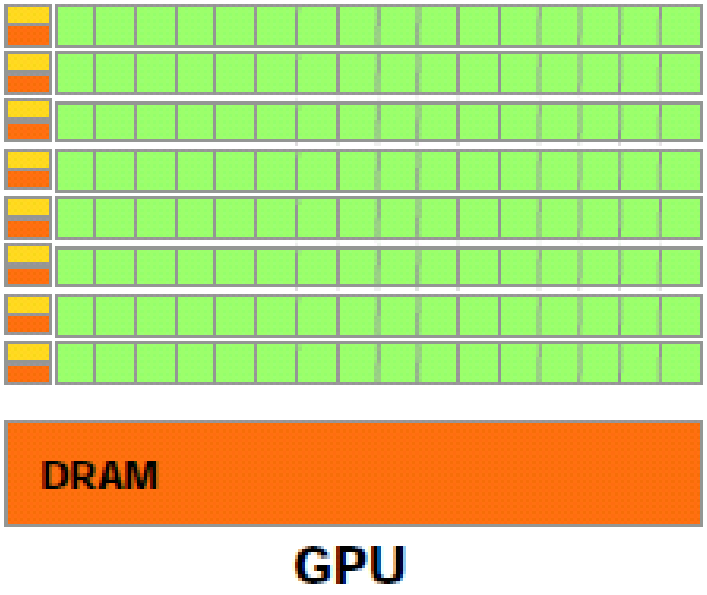



(b)

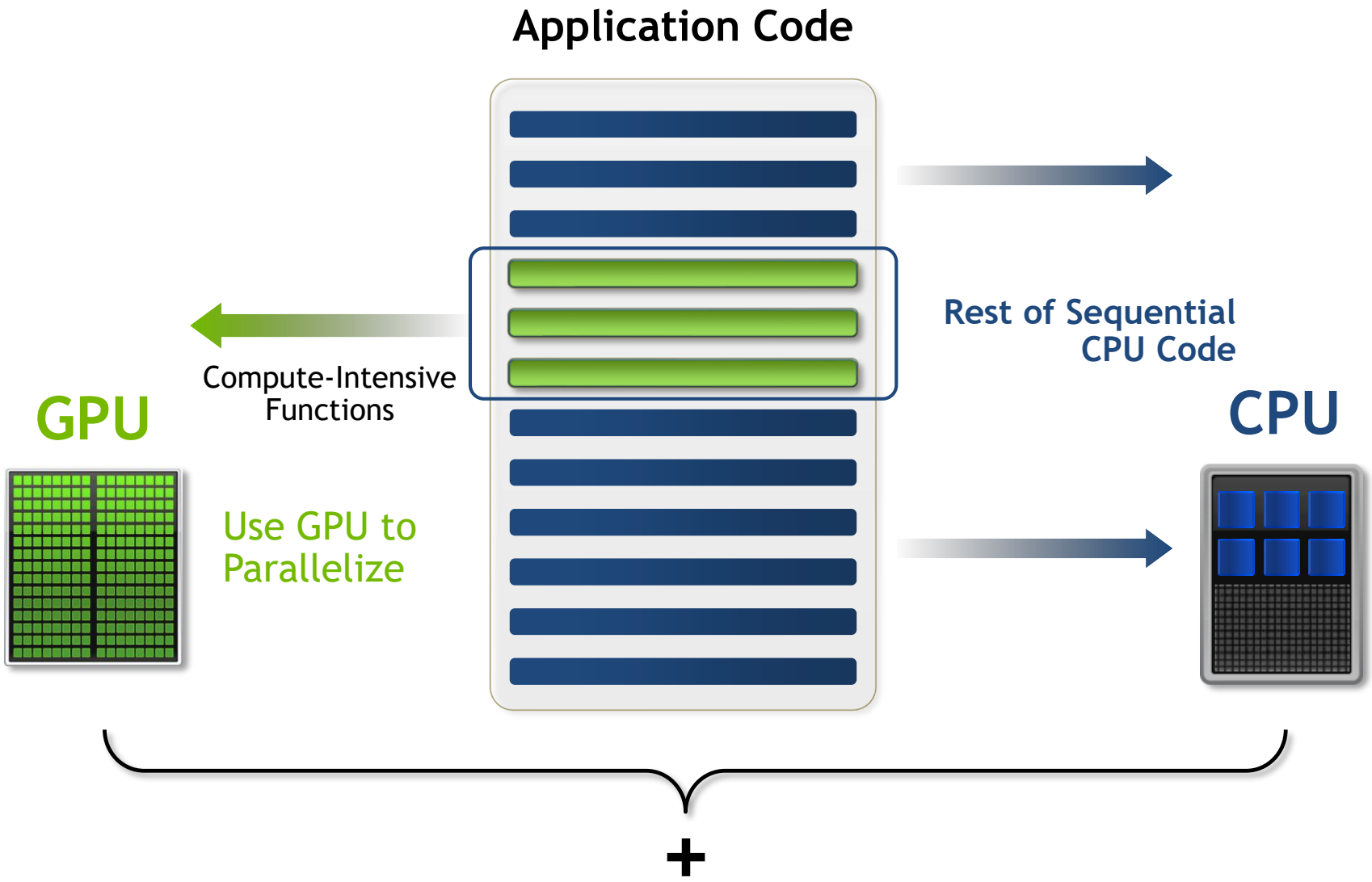
# 协处理器

- 将更多元件用于数据处理，而非控制和存储



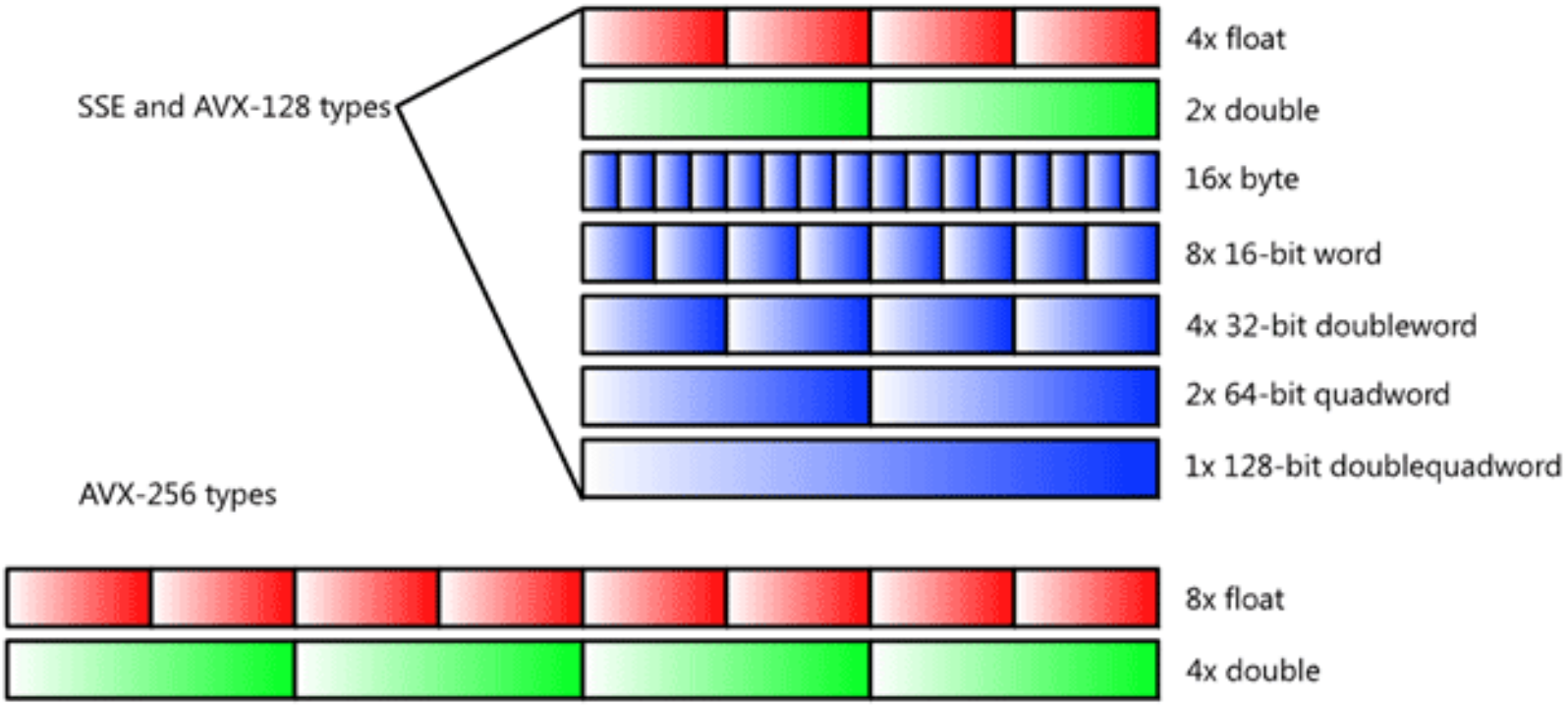



# 异构计算编程模型

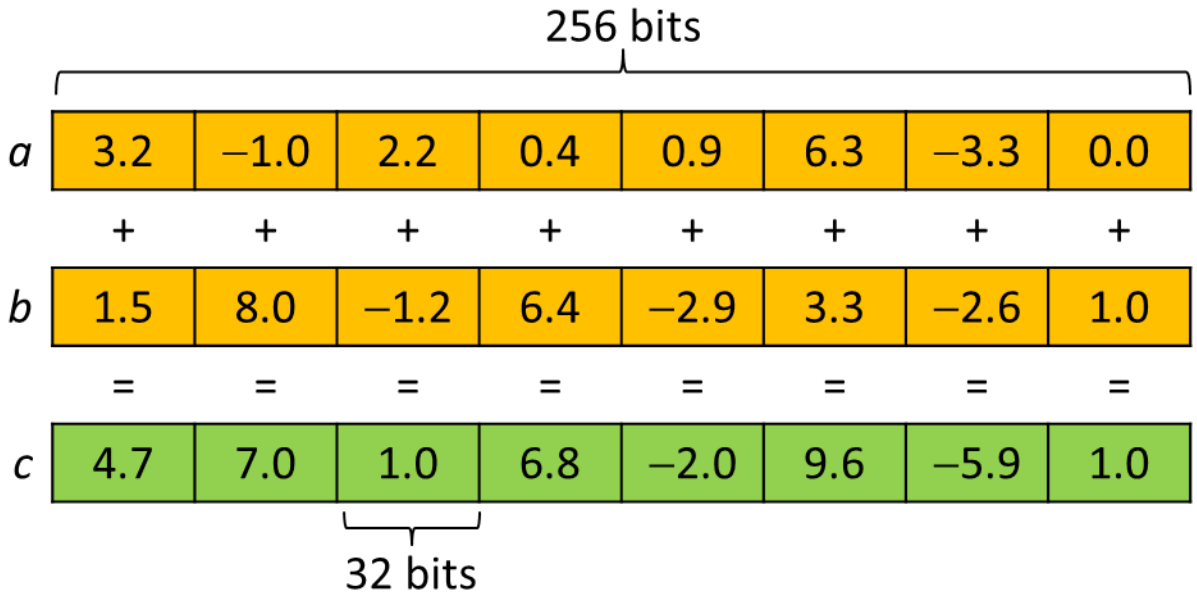

Application Code
Compute-Intensive Functions
Rest of Sequential CPU Code
GPU
CPU
Use GPU to Parallelize
+

# X86 SSE/AVX 指令集

- Intel 公司设计、对其 X86体系的 SIMD 扩展指令集
- 支持**向量化数据并行**，同时操作的数据个数由向量寄存器的长度与数据类型共同决定

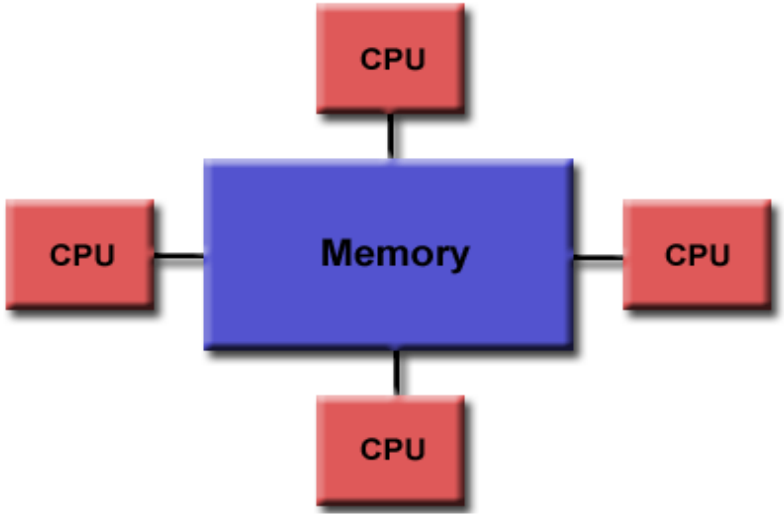


# AVX2示例

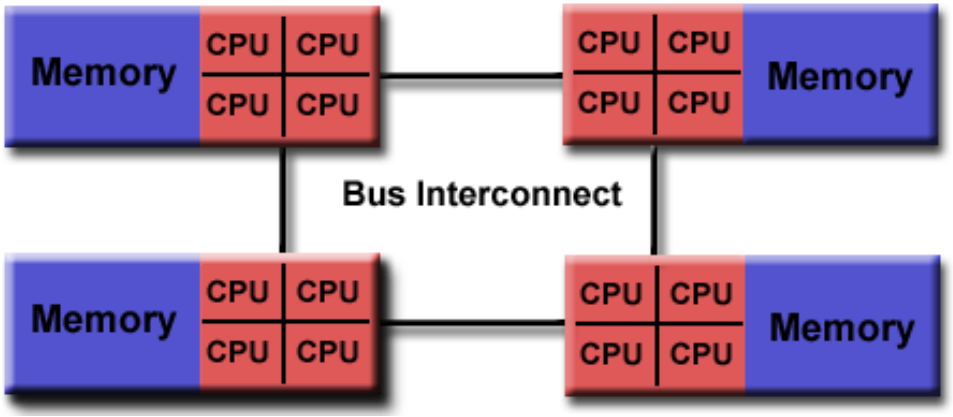



```
//AVX2-Programming with C/C++ Intrinsics
__m256 a, b, c;           // declare AVX registers
...                       // initialize a and b
c = _mm256_add_ps(a, b); // c[0:8] = a[0:8] + b[0:8]
```

# 共享存储访问

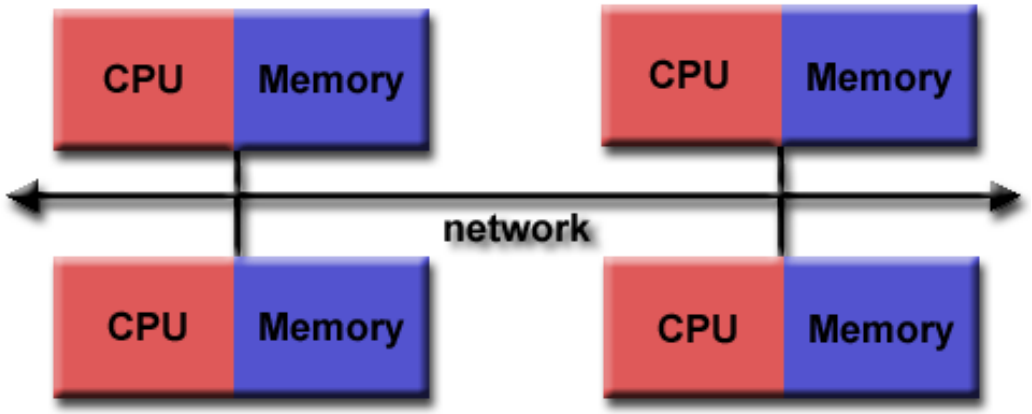


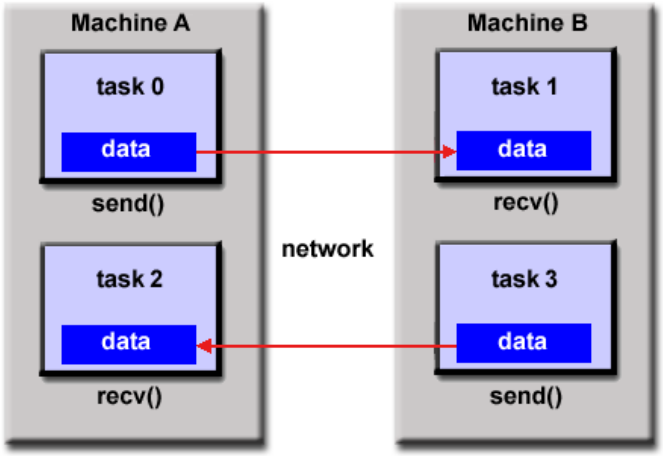

- UMA
- 均匀存储访问
- 每个处理器可以等同地访问统一的存储空间



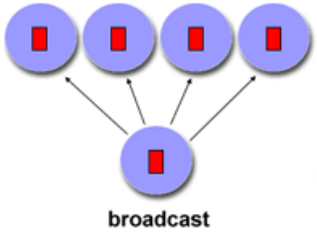

- NUMA
- 非均匀存储访问
- 存储全局统一编址，每个处理器均可直接访问
- 处理器访问本地存储速度比远程存储快

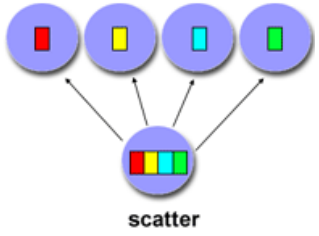
# 分布式存储访问

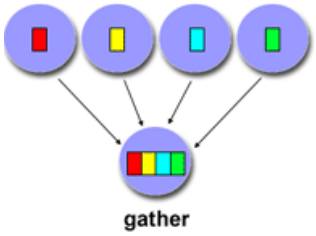


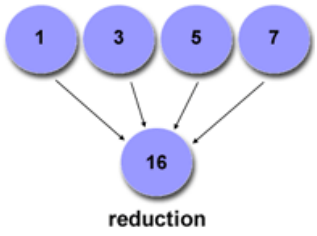




- 每个物理节点有自己私有的存储空间，各物理节点通过高速网络连接；
- 运行在不同物理节点的任务，不能直接访问其他物理节点的存储空间，相互数据交换通过消息传递实现。

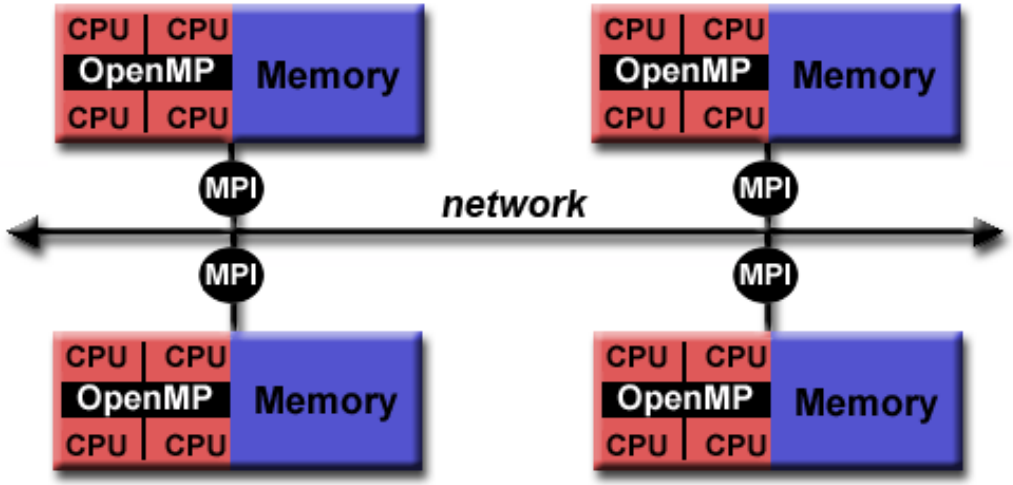


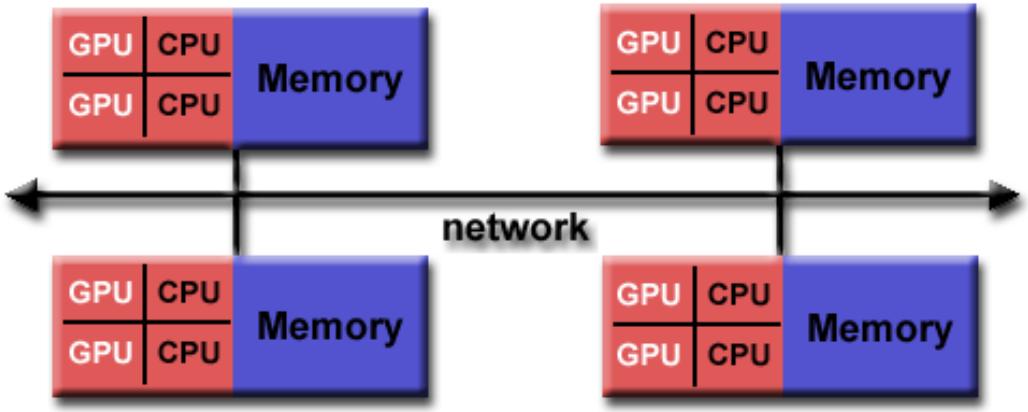
# 多层混合并行计算




- 物理节点内部任务可实现为OpenMP或多线程代码
- 运行在不同物理节点的任务通过消息传递交换数据

---

- 物理节点内部分计算分配到GPU
- 物理节点间的任务通过消息传递交换数据



# Outline

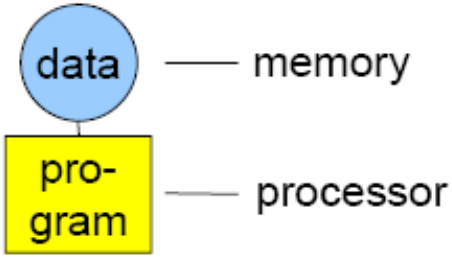
- MPI概述
- 点到点通信/组通信
- 阻塞通信/非阻塞通信
- MPI_Sendrecv和虚进程
- 自定义数据类型
- 虚拟进程拓扑

# Outline

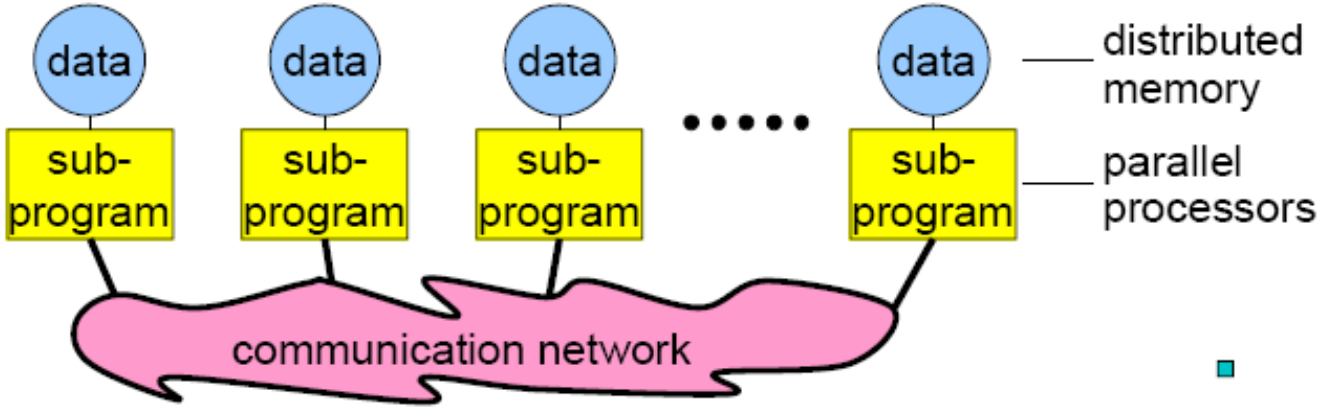
- **MPI概述**
- 点到点通信/组通信
- 阻塞通信/非阻塞通信
- MPI_Sendrecv和虚进程
- 自定义数据类型
- 虚拟进程拓扑

# 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**

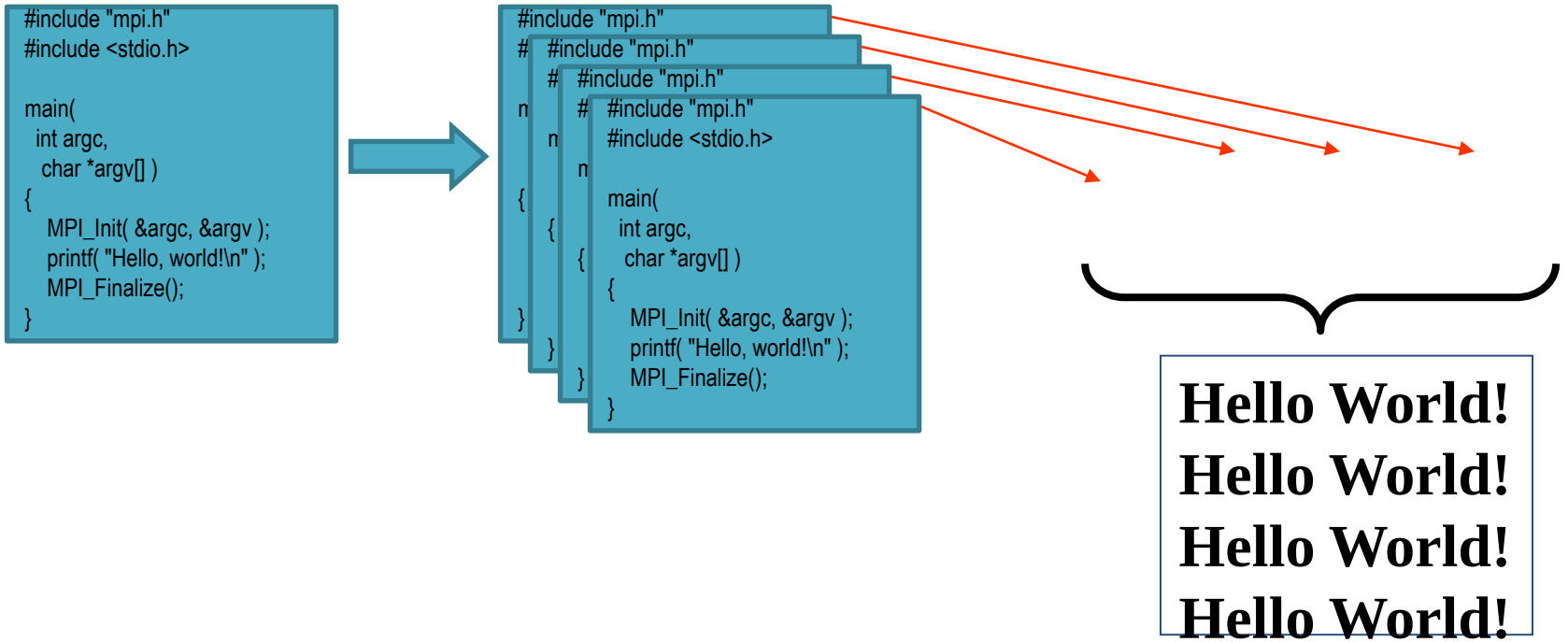
```
PROGRAM hello
  INCLUDE ‘mpif.h‘
  INTEGER err
  CALL MPI_INIT(err)
  PRINT *, “hello world!”
  CALL MPI_FINALIZE(err)

END
```

C

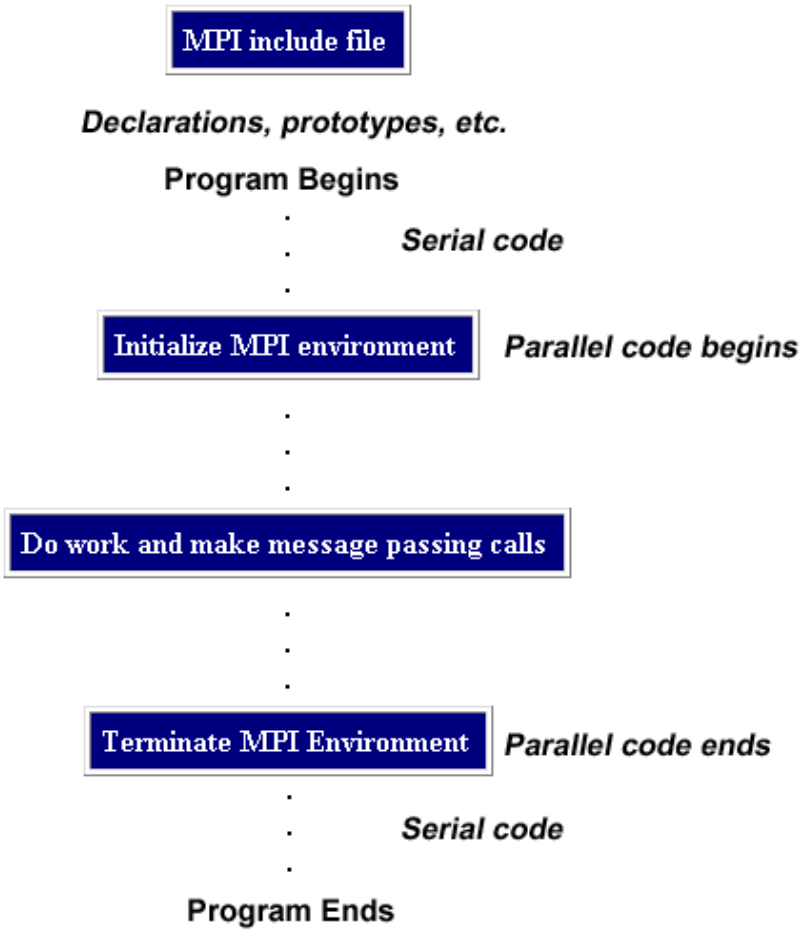
```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char * argv[])
{
  int err;

  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);
```

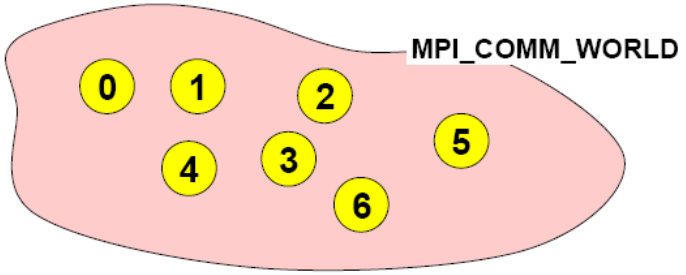
  - MPI系统将通过argc,argv得到命令行参数

- MPI代码的最后一行必须是：

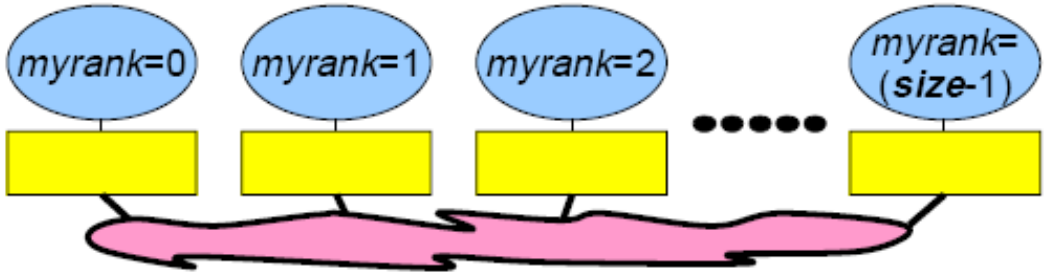
```
MPI_Finalize();
```

  - 如果没有此行，MPI程序将不会终止。

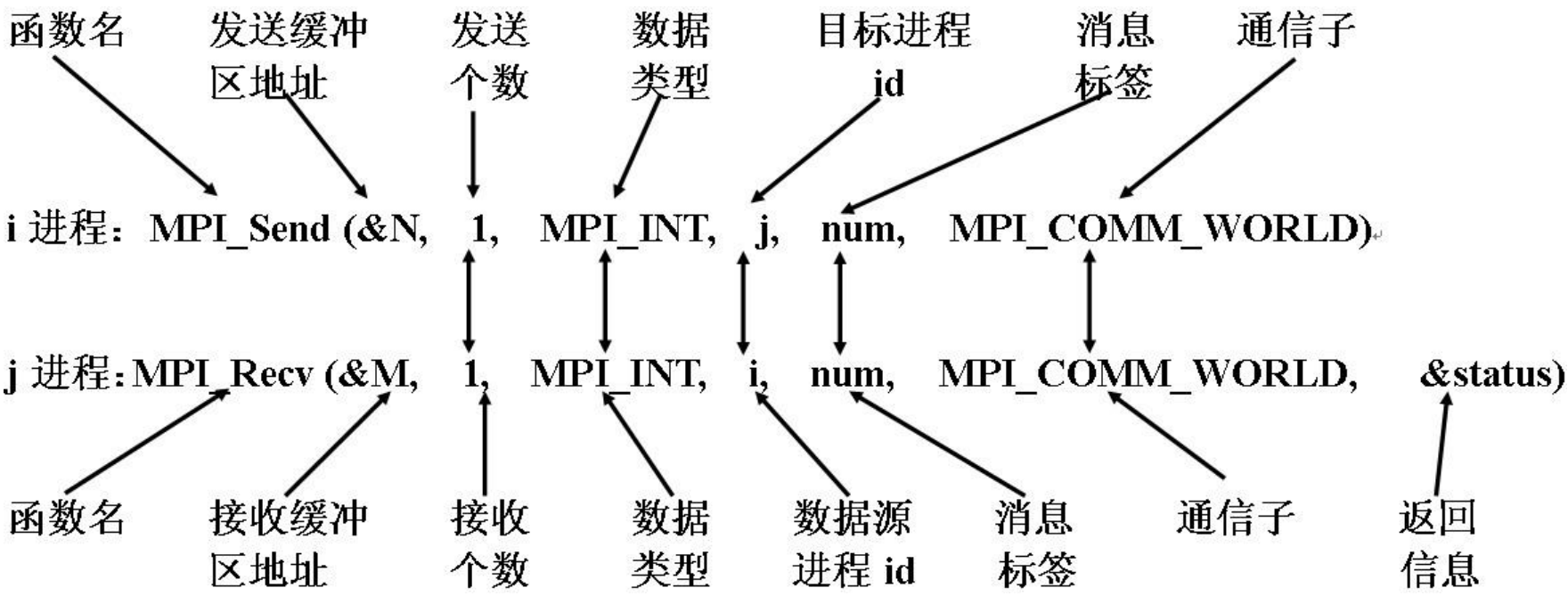
# MPI进程身份标识



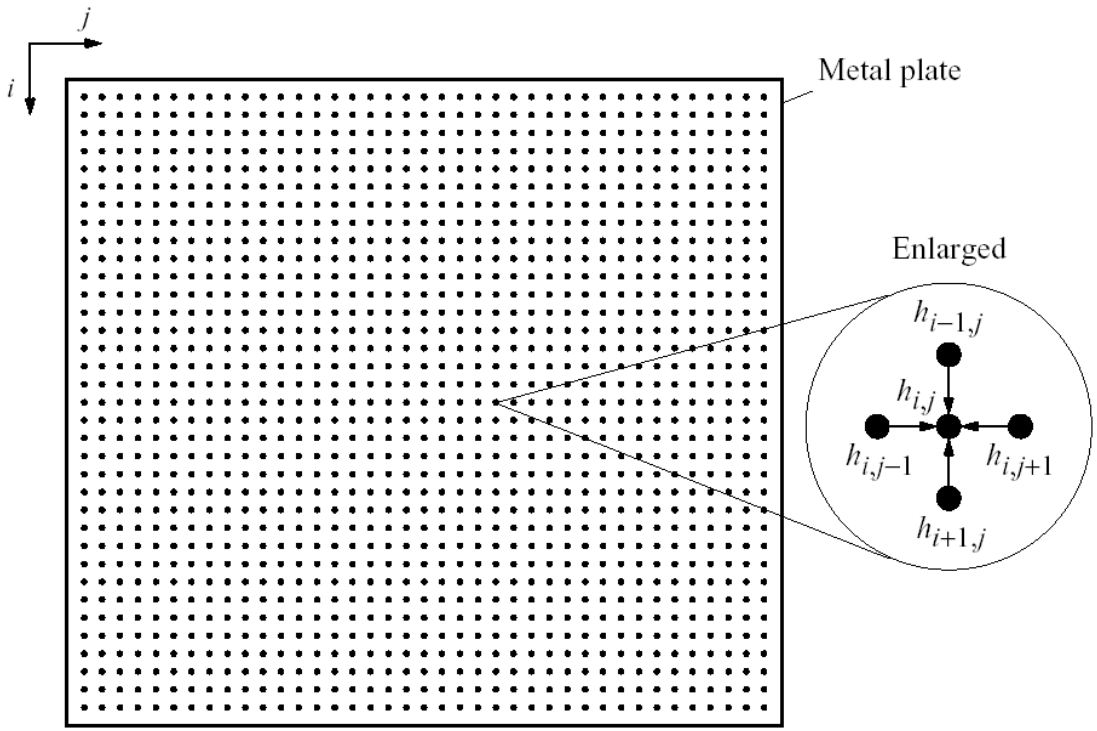

- 通信域
  - 缺省的通信域为MPI_COMM_WORL_
- MPI_Comm_size(MPI_COMM_WORLD, &size)
  - 获得缺省通信域内所有进程数目，赋值给size
- MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
  - 获得进程在缺省通信域的编号，赋值给myrank



# 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

# 问题：Jacobi迭代



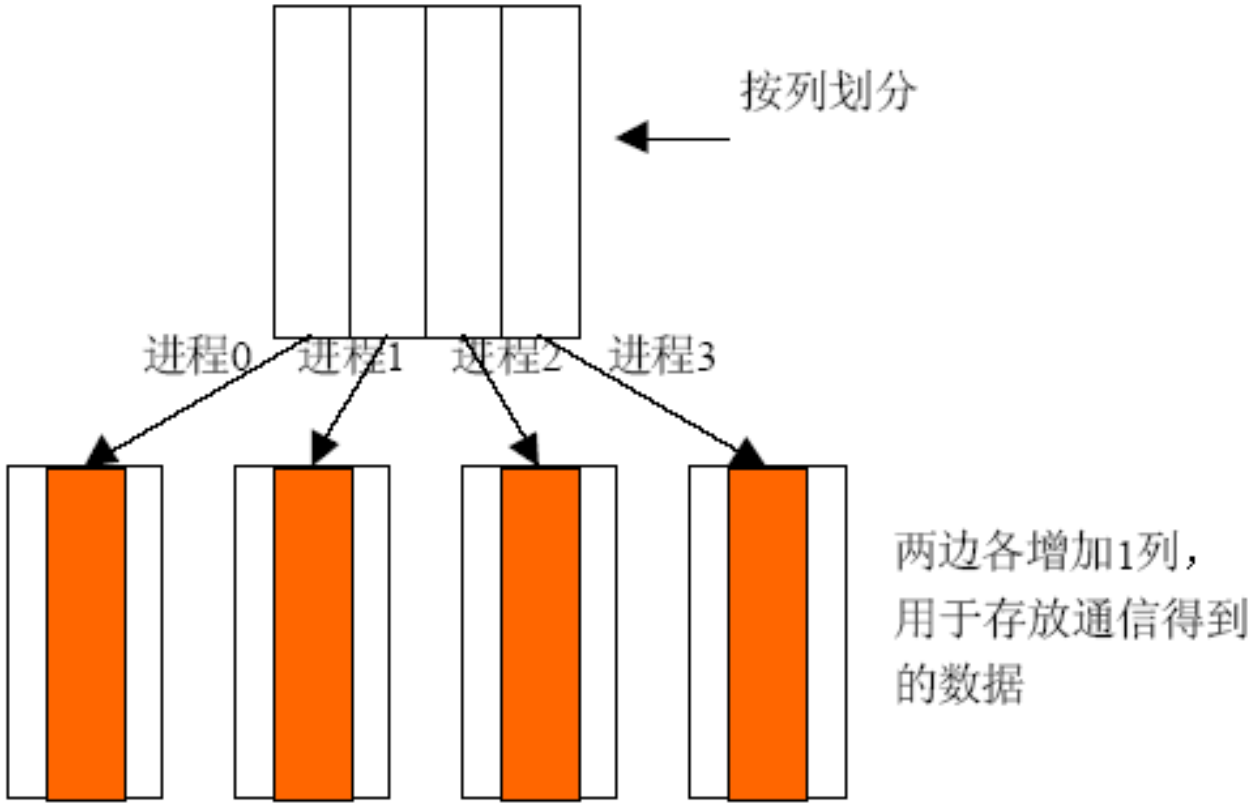

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

# Jacobi迭代

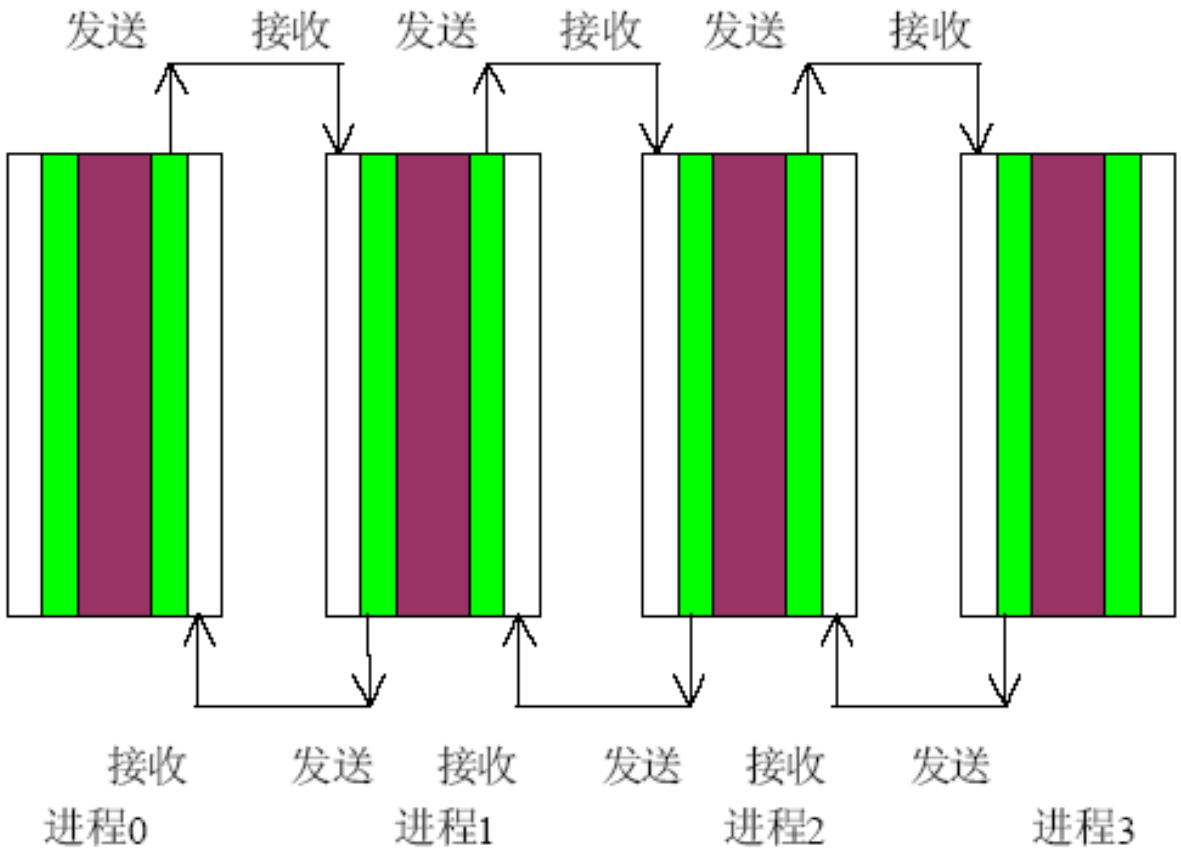
- 伪代码描述：

```
…
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

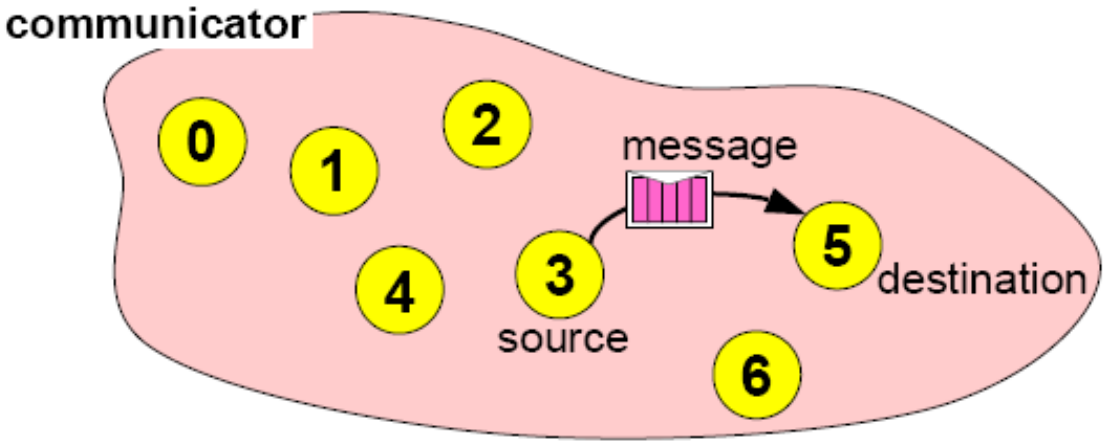


# Outline

- MPI概述
- **点到点通信/组通信**
  - 阻塞通信/非阻塞通信
- MPI_Sendrecv和虚进程
- 自定义数据类型
- 虚拟进程拓扑

# 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)**

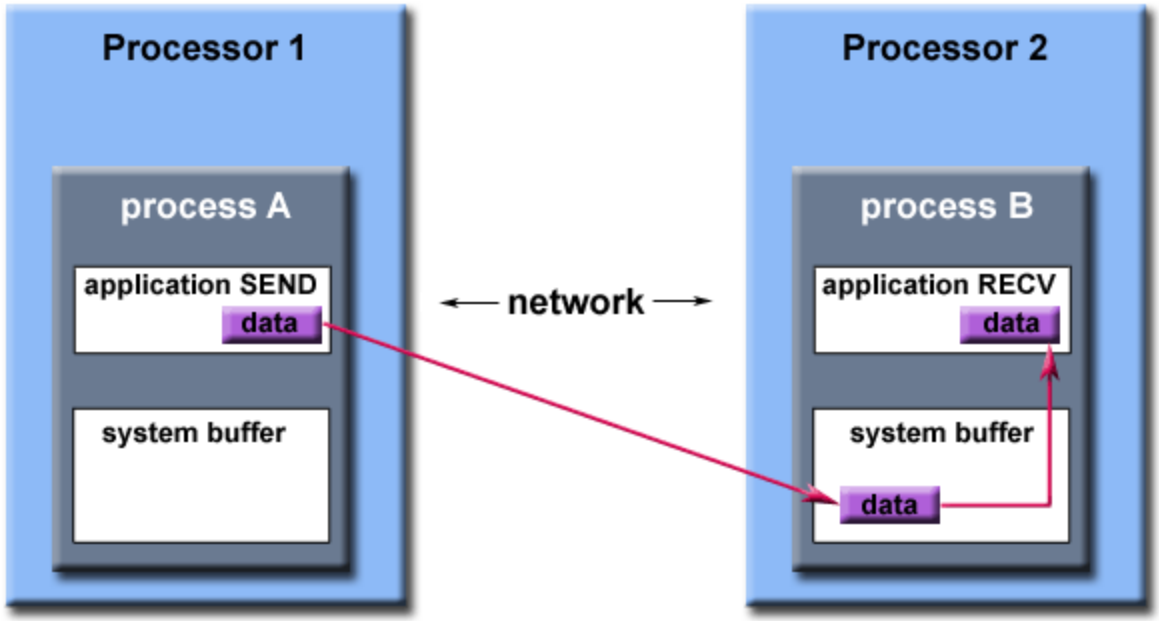
- MPI_Send(&N, 1, MPI_INT, i, i , MPI_COMM_WORLD);
- 第一个参数指明消息缓存的起始地址，即存放要发送的数据信息。
- 第二个参数指明消息中给定的数据类型有多少项，数据类型由第三个参数给定。
- 数据类型要么是基本数据类型，要么是导出数据类型，后者由用户生成指定一个可能是由混合数据类型组成的非连续数据项。
- 第四个参数是目的进程的标识符(进程编号)。
- 第五个是消息标签。
- 第六个参数标识进程组和上下文，即通信域。通常，消息只在同组的进程间传送。但是MPI允许通过intercommunicators在组间通信。

# 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;
      }
```

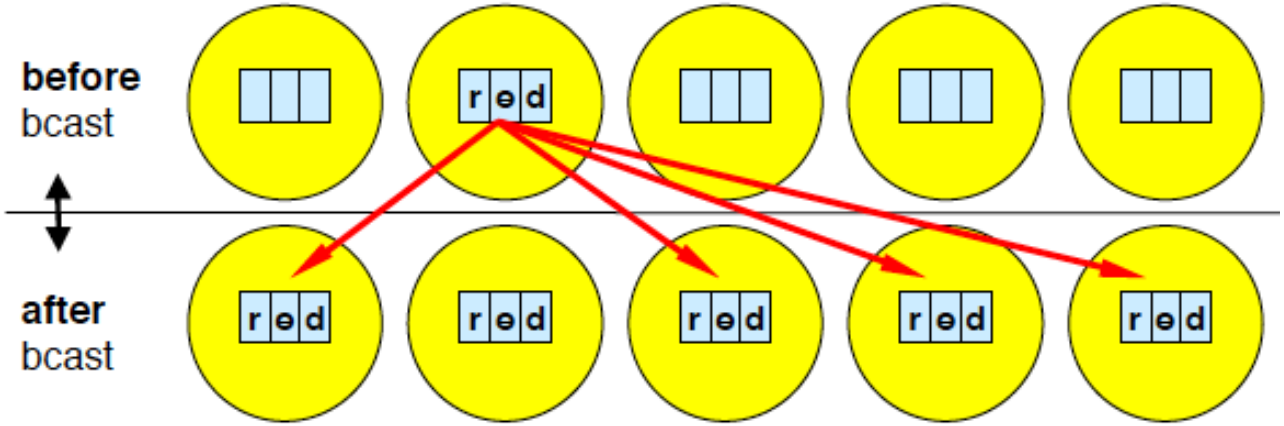
# 组通信

- 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*)

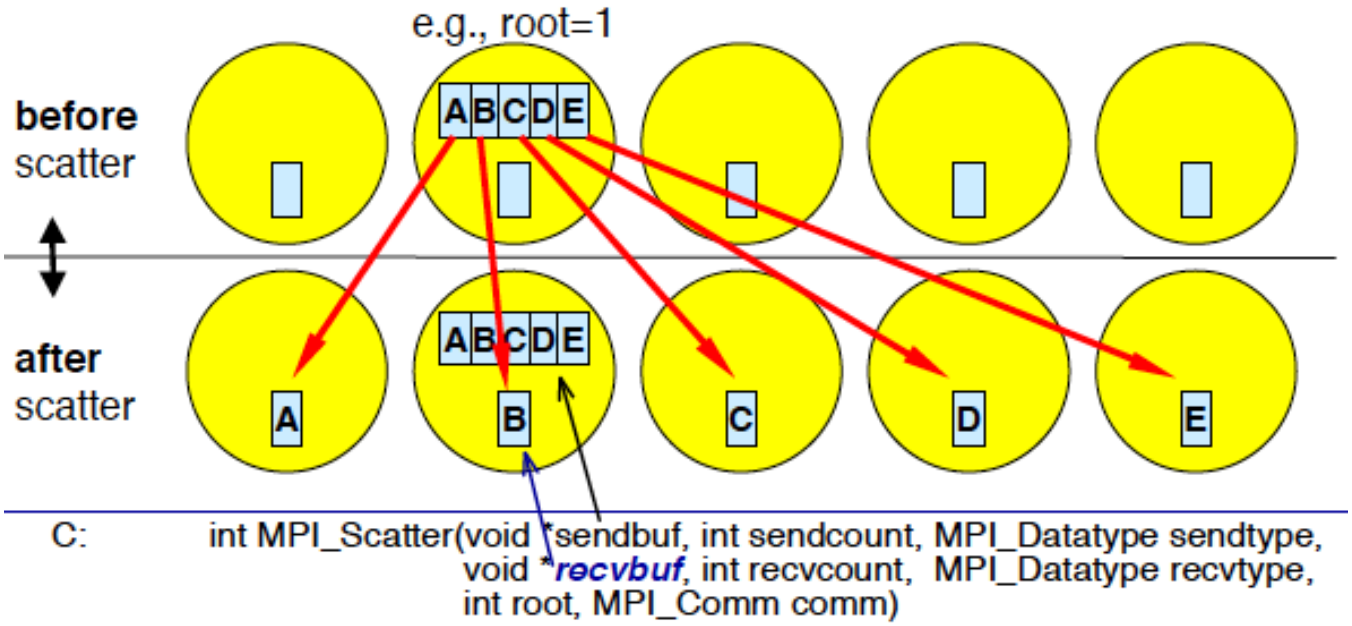
- 标号为Root的进程发送相同的消息给标记为Comm的通信子中的所有进程。
- 消息的内容如同点对点通信一样由三元组(Address, Count, Datatype)标识。对Root进程来说，这个三元组既定义了发送缓冲也定义了接收缓冲。对其它进程来说，这个三元组只定义了接收缓冲。



# Scatter

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

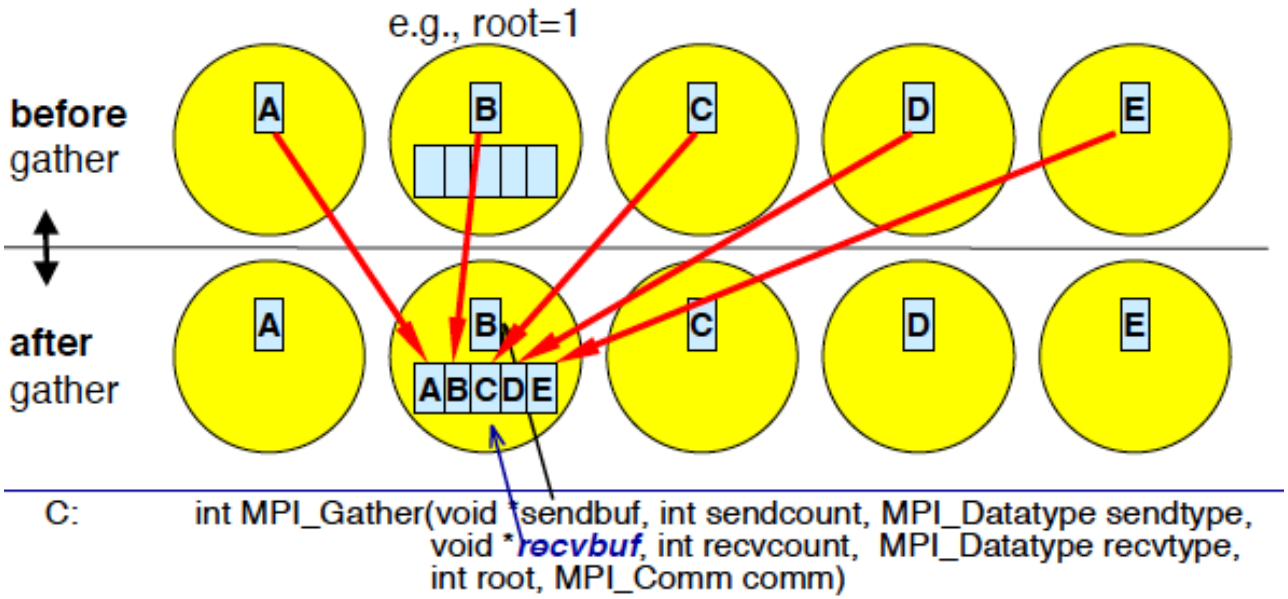
- Root进程发送给所有n个进程各发送一个不同的消息，包括自己。
- 这n个消息在Root进程的发送缓冲区中按标号的顺序有序地存放。每个接收缓冲由三元组(RecvAddress, RecvCount, RecvDatatype)标识。非Root进程忽略发送缓冲。
- 对Root进程，发送缓冲由三元组(SendAddress, SendCount, SendDatatype)标识.



# Gather

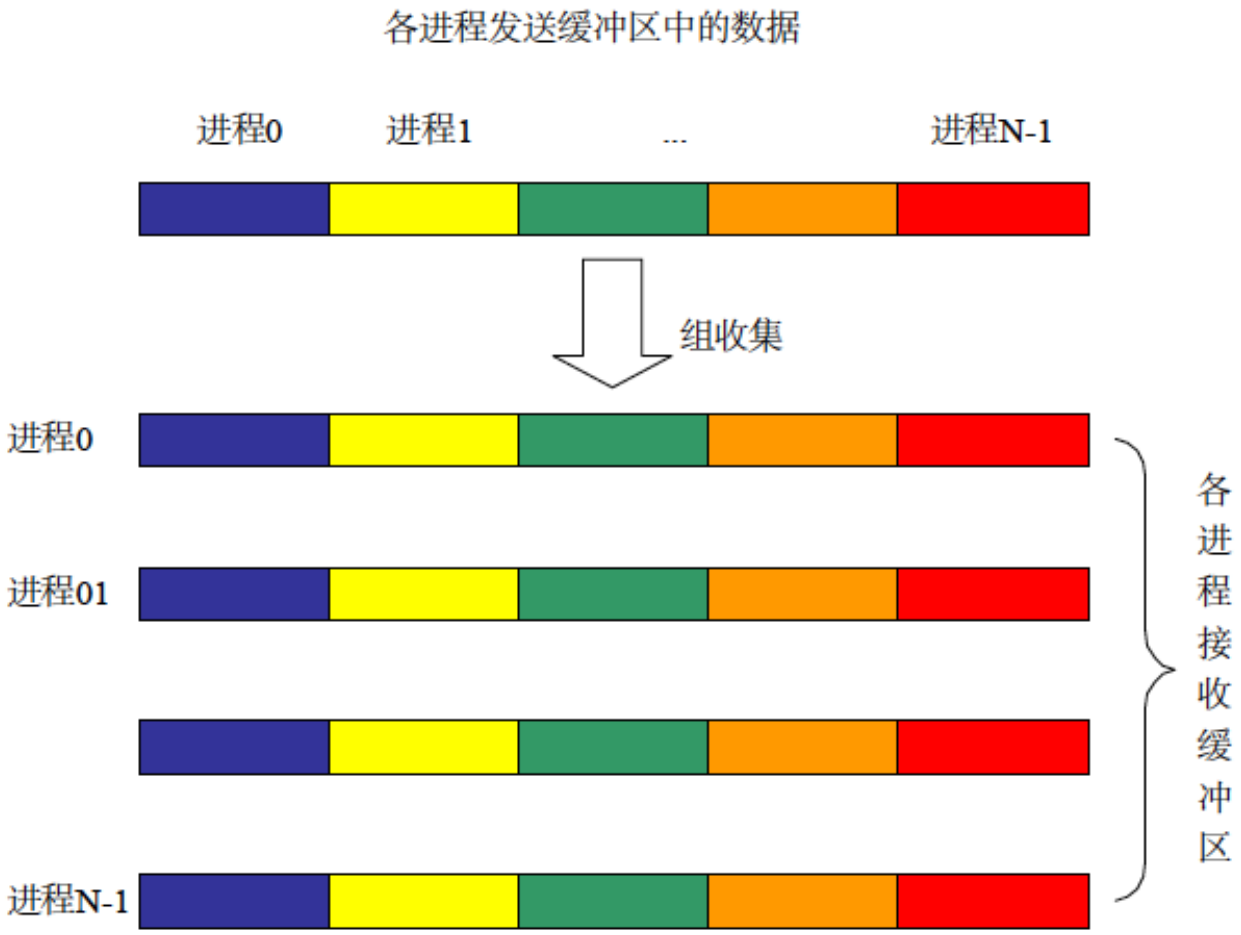
MPI_Gather (SendAddress,SendCount,SendDatatype,
*RecvAddress,RecvCount,RecvDatatype,Root,Comm*)

- 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。



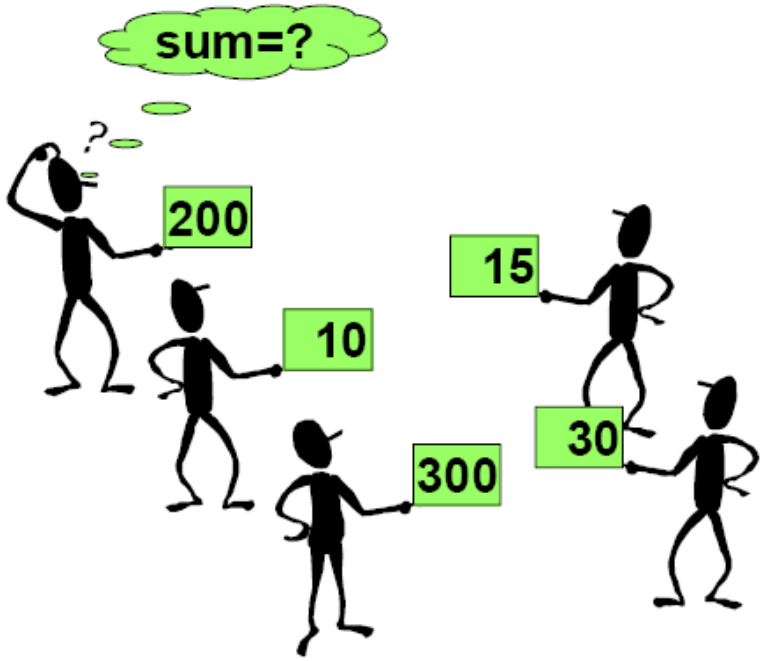
# Allgather

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



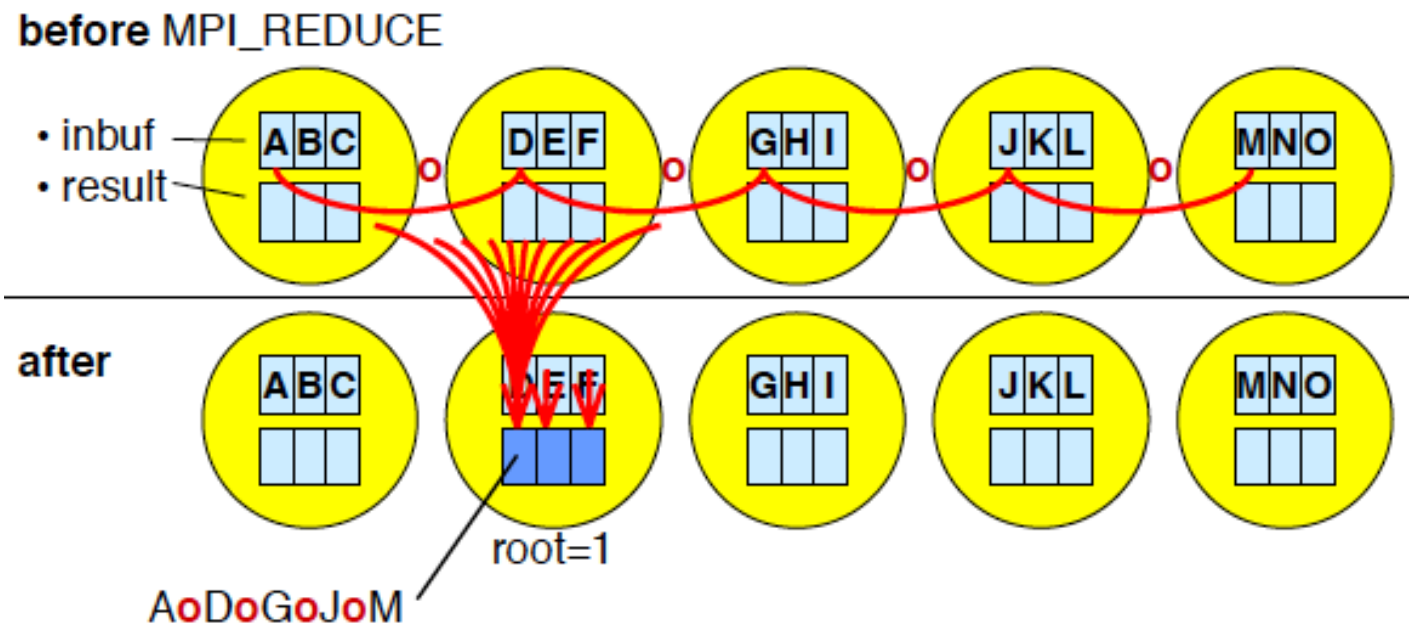
# 归约（Reduce）

- 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



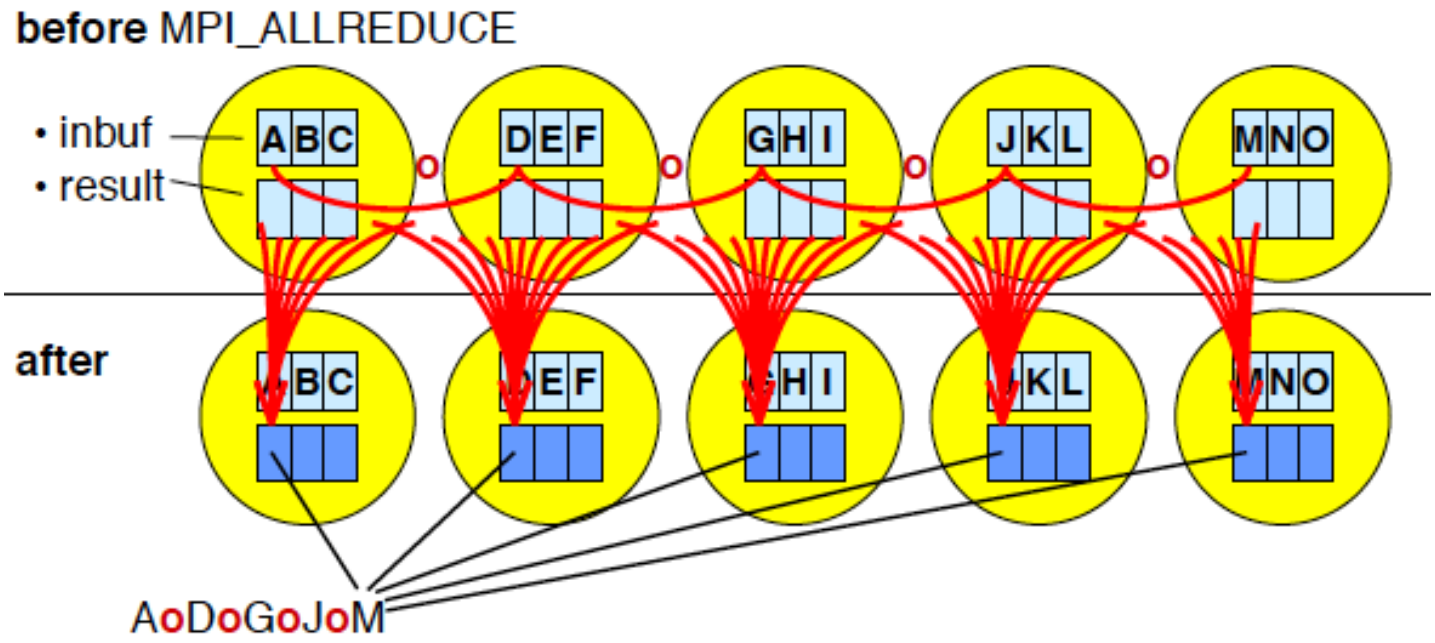
# MPI_Reduce

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

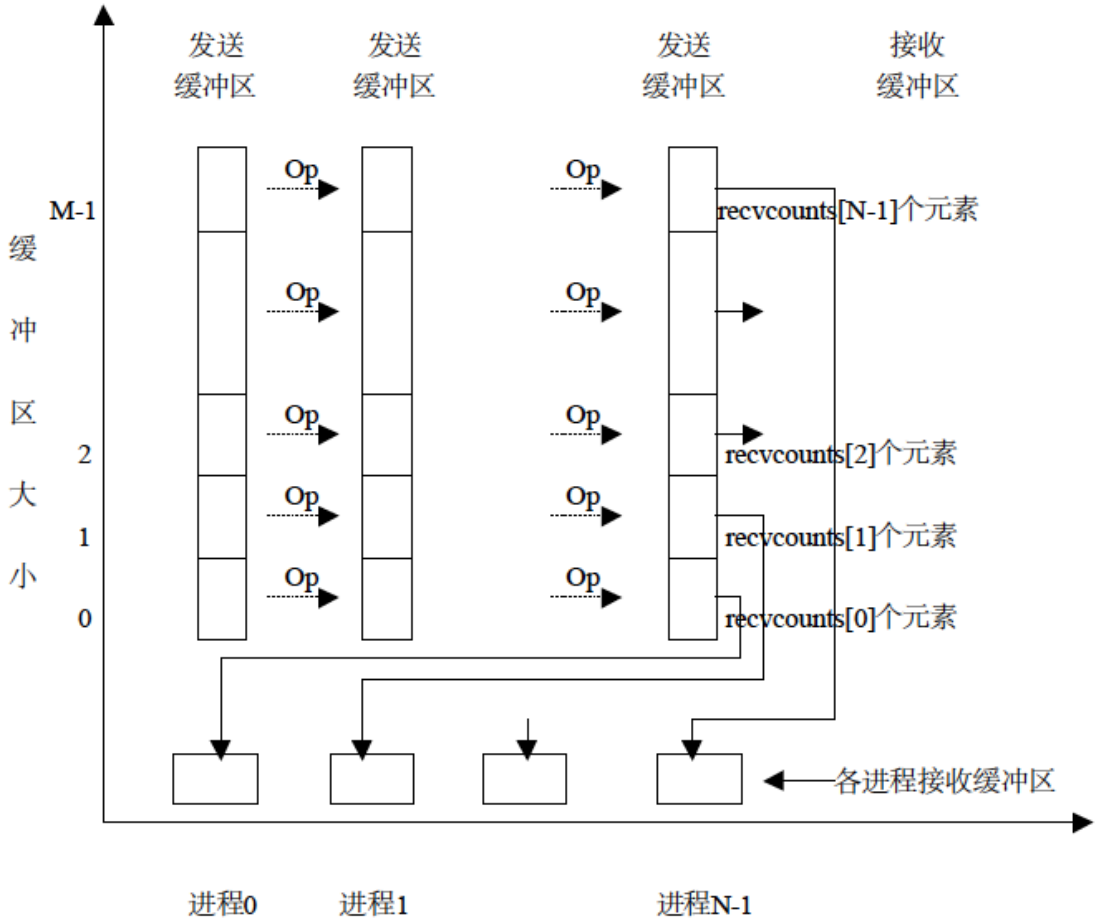


# MPI_Allreduce

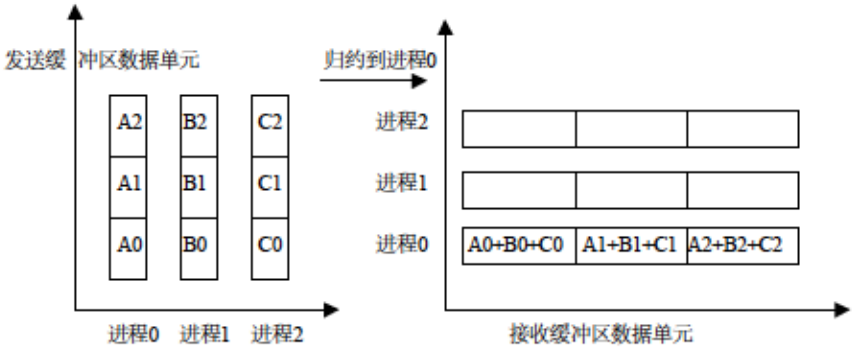
- 语法与reduce类似，但无root参数
- 所有进程都将获得结果



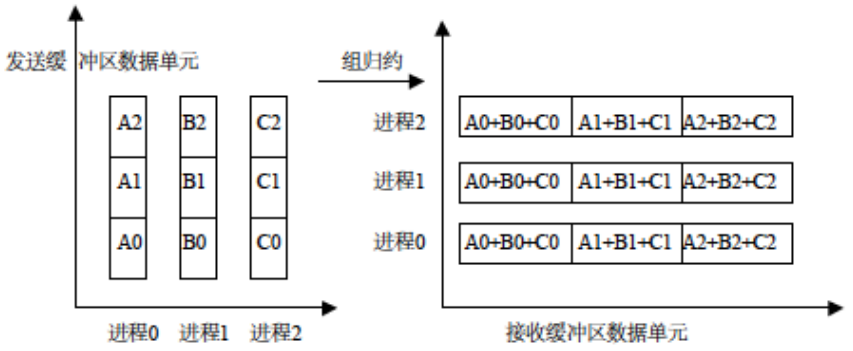
# MPI_Reduce_scatter



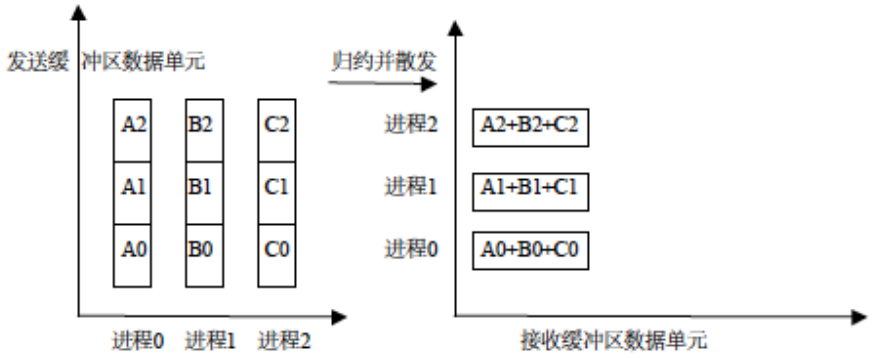
# 不同类型的归约操作对比



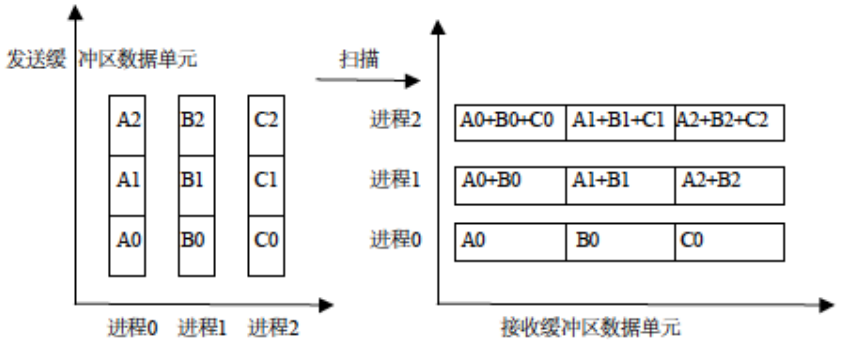

Reduce




Allreduce



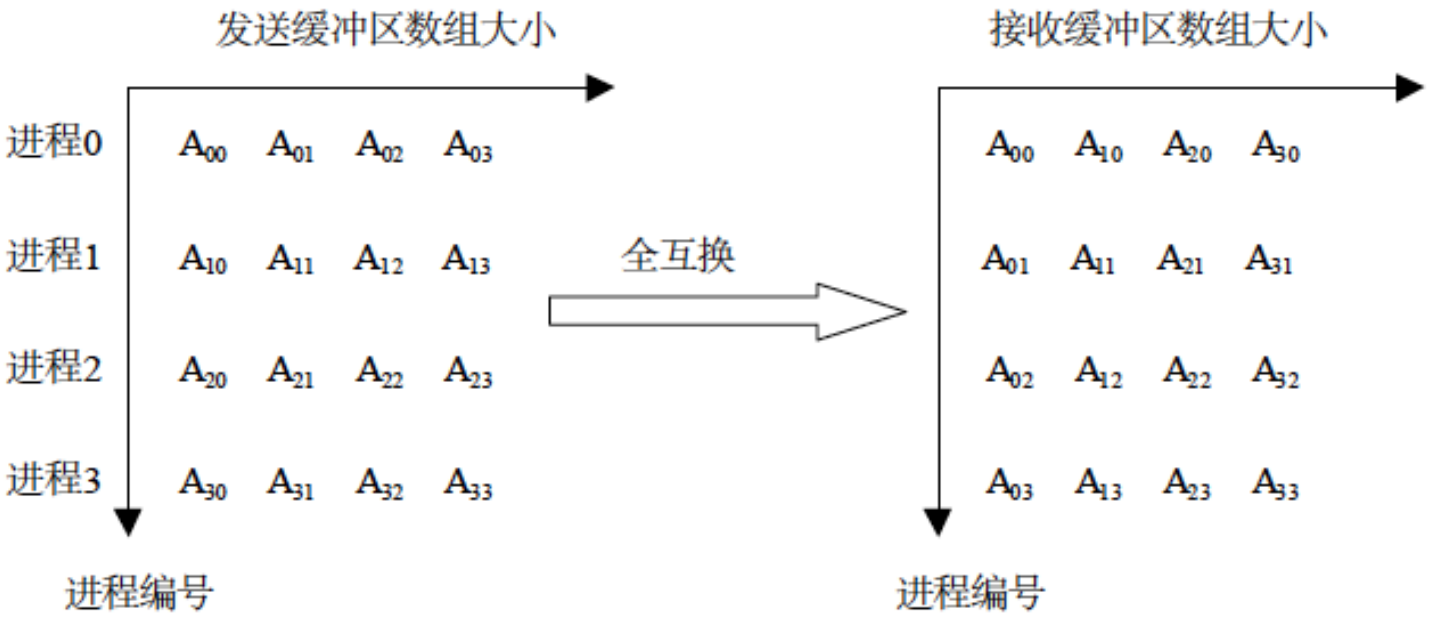

Reduce_scatter




Scan

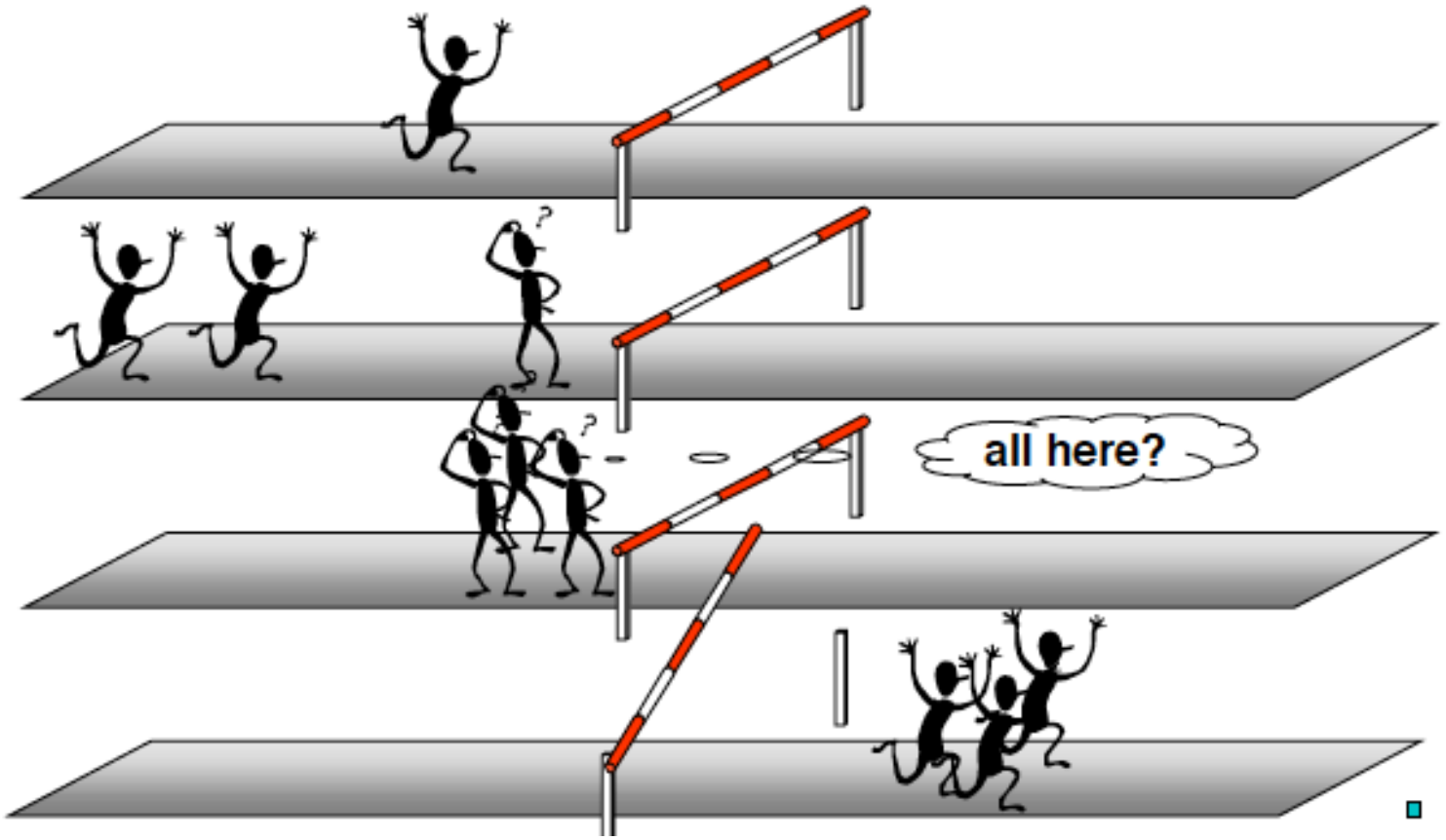
# MPI_Alltoall

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

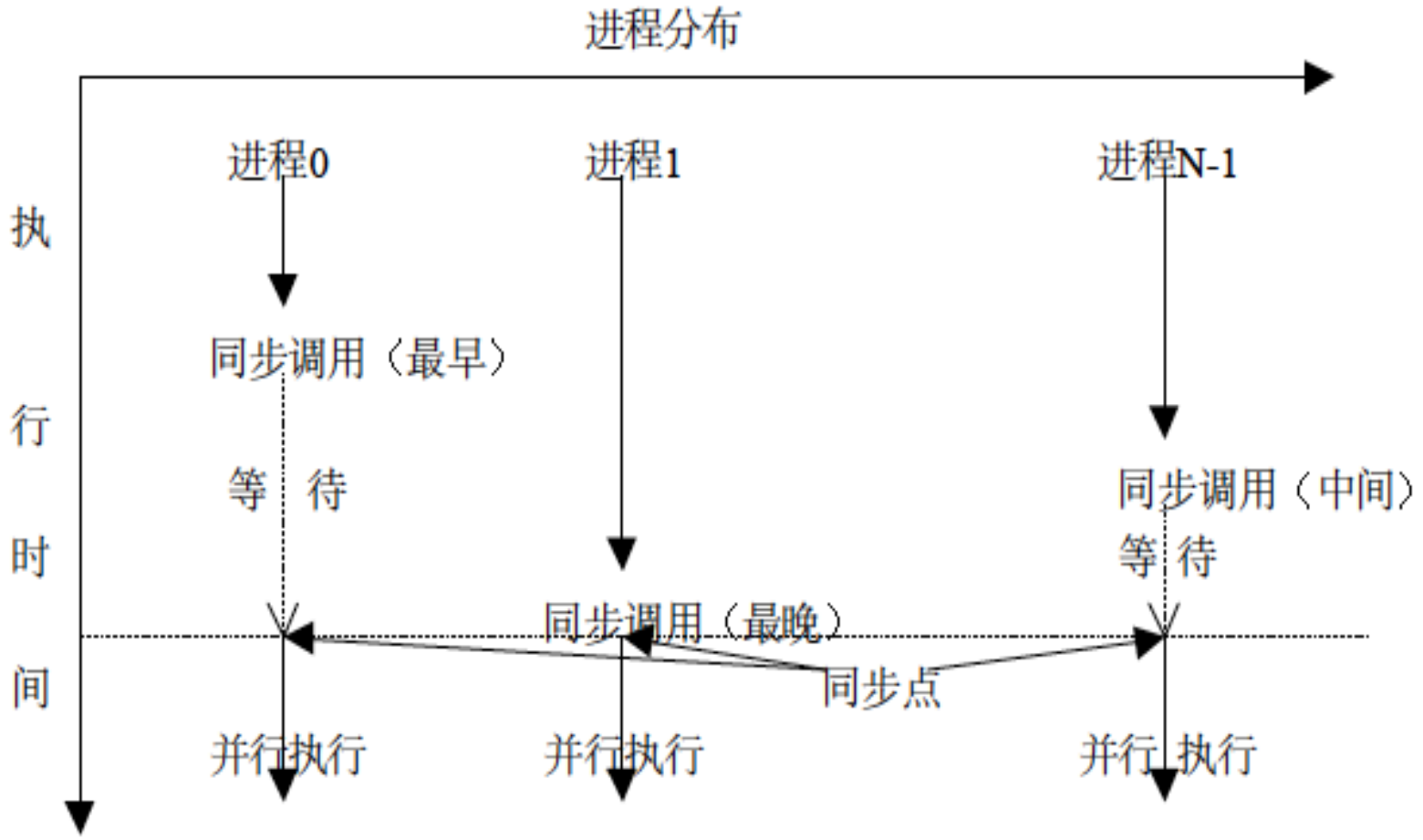



每个进程依次将它的发送缓冲区的第i块数据发送给第i个进程，同时每个进程又都依次从第j个进程接收数据放到各自接收缓冲区的第j块数据区的位置

# 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
- **阻塞通信/非阻塞通信**
- MPI_Sendrecv和虚进程
- 自定义数据类型
- 虚拟进程拓扑

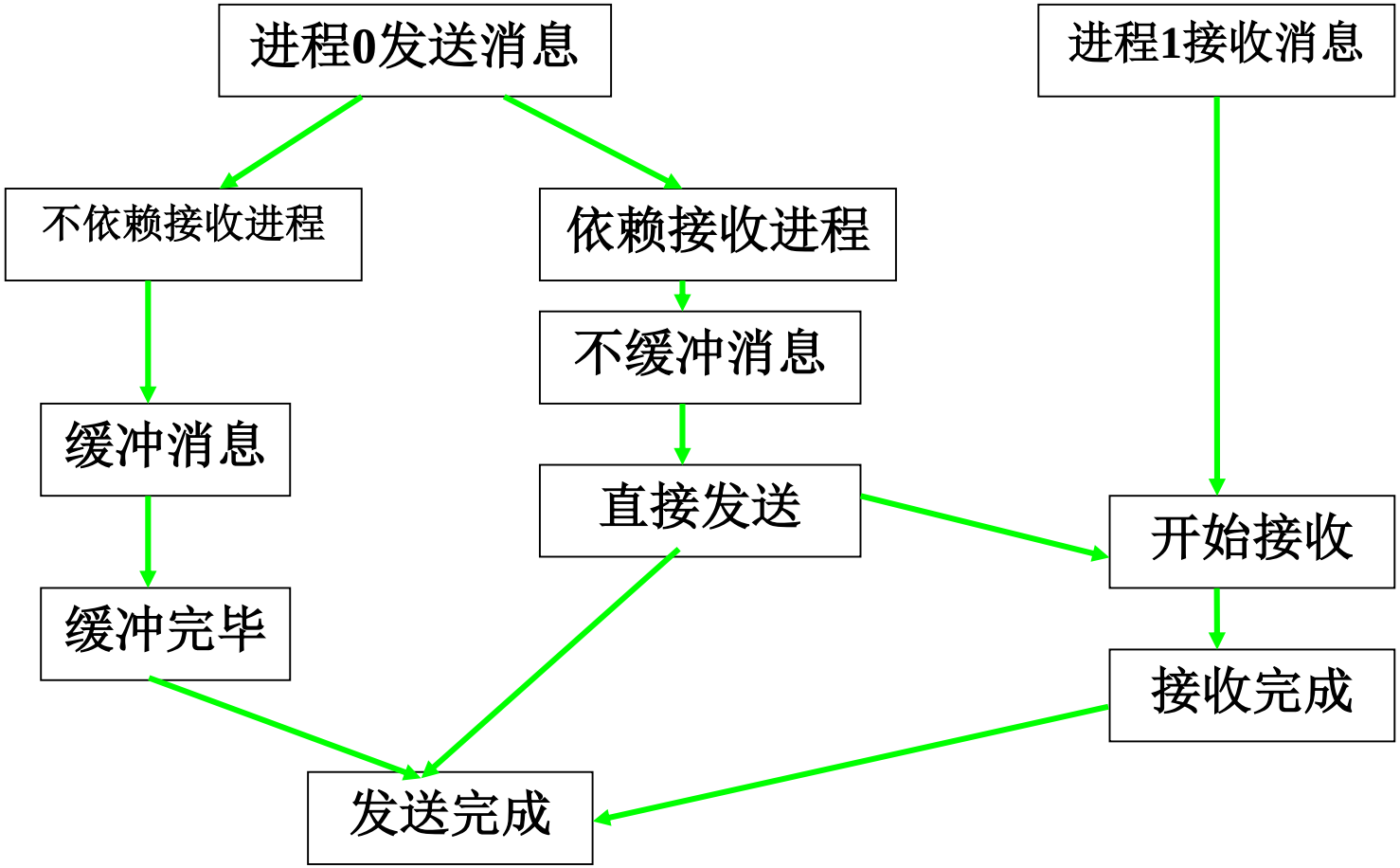
# 阻塞通信

- 标准通信模式
- 缓存通信模式
- 同步通信模式
- 就绪通信模式

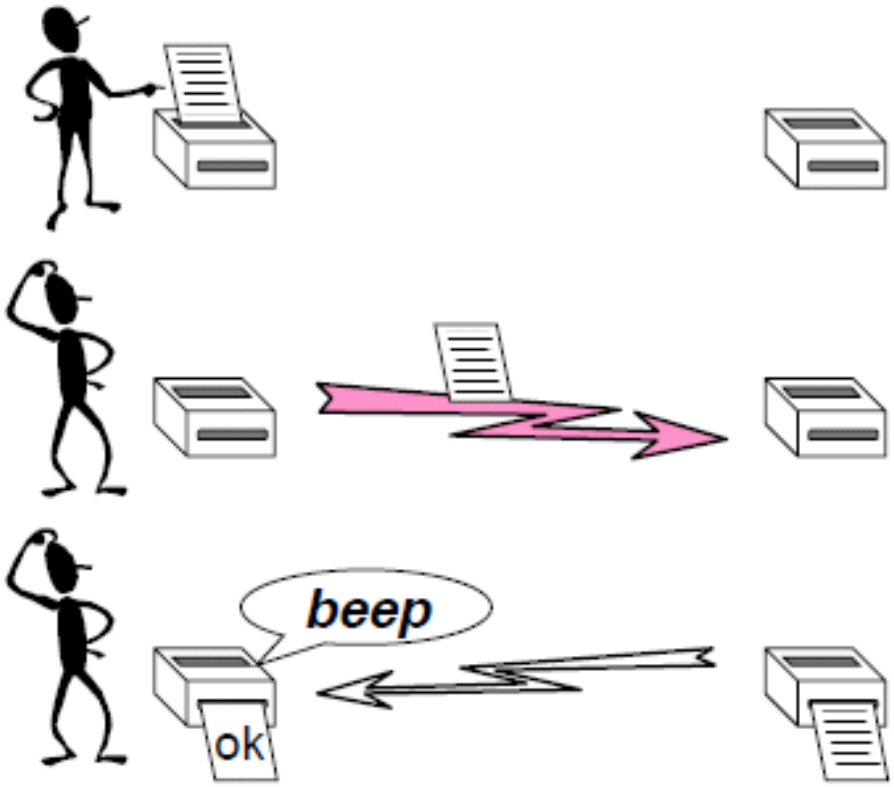
# 标准通信模式

- MPI_Send
- 在MPI采用标准通信模式时，是否对发送的数据进行缓存是由MPI自身决定的，而不是由并行程序员来控制。
- 如果MPI决定缓存将要发出的数据，发送操作不管接收操作是否执行，都可以进行，而且发送操作可以正确返回，而不要求接收操作收到发送的数据。

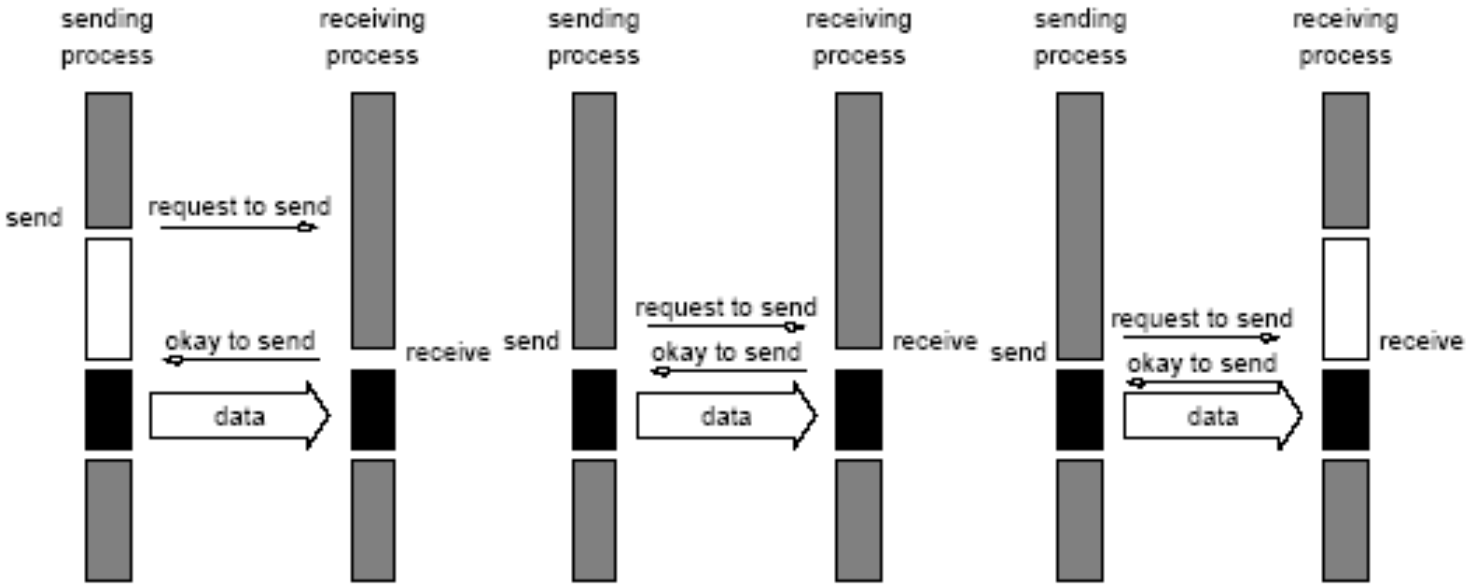
# 标准通信模式


进程0发送消息
进程1接收消息
不依赖接收进程
依赖接收进程
不缓冲消息
缓冲消息
直接发送
开始接收
缓冲完毕
接收完成
发送完成

# 同步发送



# 同步发送




(a) Sender comes first; idling at sender

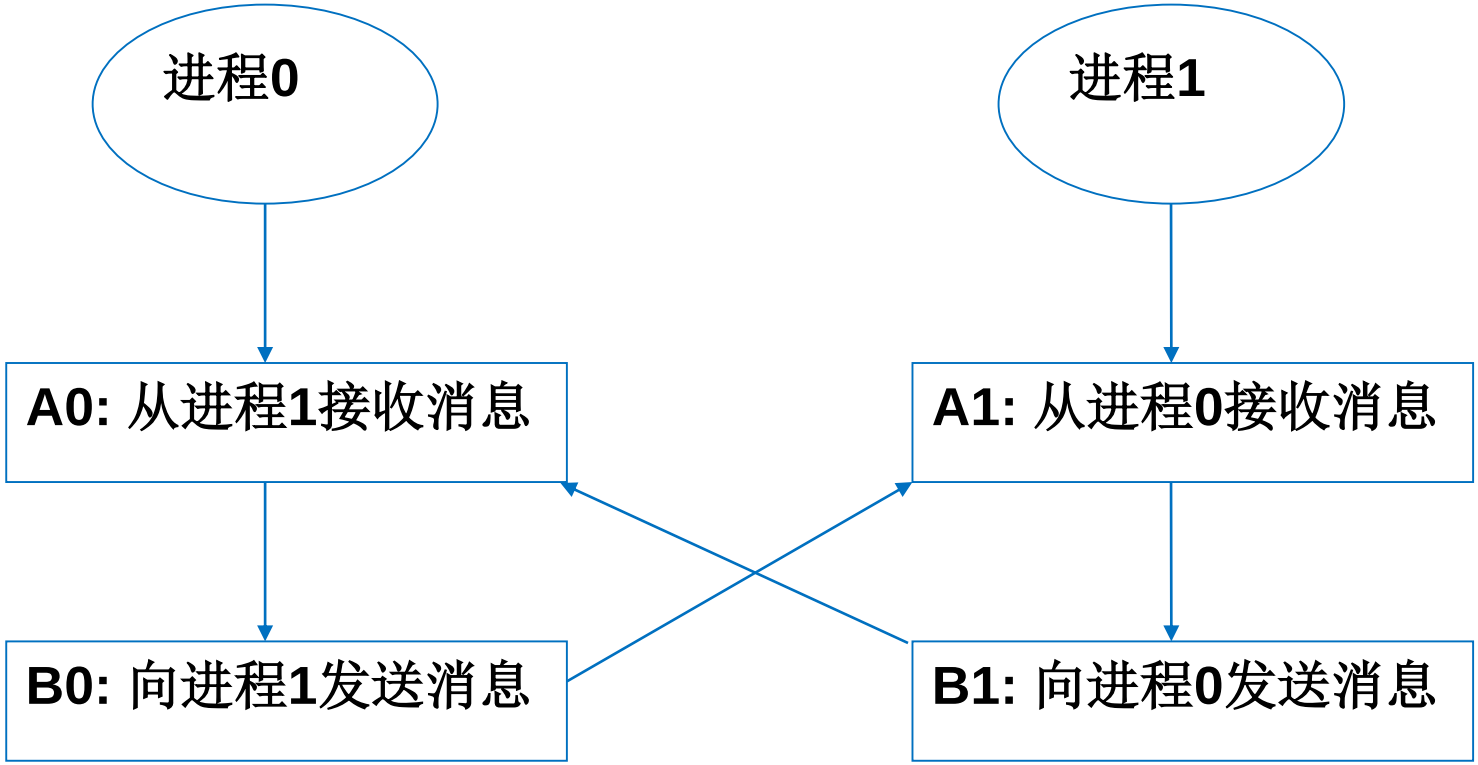
(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

# 进程间通信的组织

```
……
MPI_Comm_dup (MPI_COMM_WORLD, &comm);
If ( myid==0) {
  MPI_Recv(bufA0,1,MPI_Float,1,101,comm,status);
  MPI_Send(bufB0,1,MPI_Float,1,100,comm);}

else if (myid==1){

  MPI_Recv(bufA1,1,MPI_Float,0,100,comm,status);
  MPI_Send(bufB1,1,MPI_Float,0,101,comm);}
……
```

# 进程间通信的组织

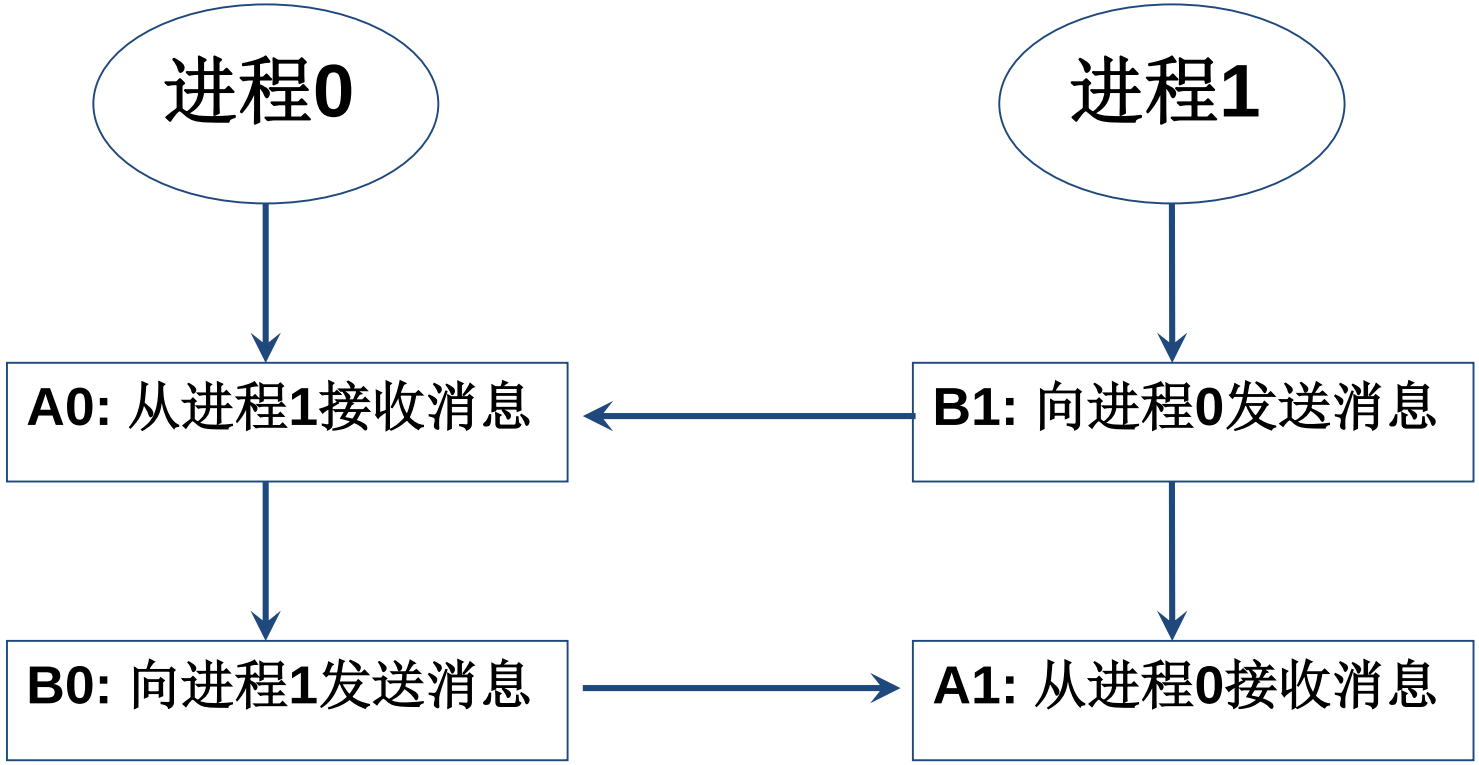


# 进程间通信的组织

```
……
MPI_Comm_dup (MPI_COMM_WORLD, &comm);
If ( myid==0) {

  MPI_Recv(bufA0,1,MPI_Float,1,101,comm,status);
  MPI_Send(bufB0,1,MPI_Float,1,100,comm);}

else if (myid==1){

  MPI_Recv(bufA1,1,MPI_Float,0,100,comm,status);
  MPI_Send(bufB1,1,MPI_Float,0,101,comm);}
……
```

死锁

# 死锁的避免



# 上例的修改

```
……
If ( myid==0) {

    MPI_Recv(bufA0,1,MPI_Float,1,101, comm, status);
    MPI_Send(bufB0,1,MPI_Float,1,100, comm);}

else if (myid==1){

    MPI_Send(bufB1,1,MPI_Float,0,101, comm);}
    MPI_Recv(bufA1,1,MPI_Float,0,100, comm, status);
    ……
```

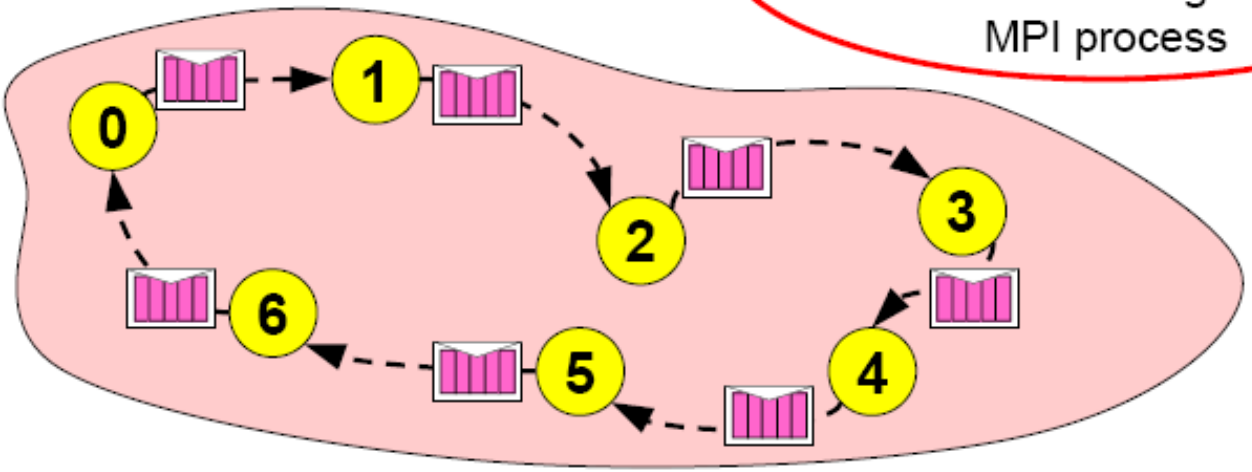
# 循环死锁

- Code in each MPI process:

  MPI_Ssend(…, right_rank, …)

  MPI_Recv( …, left_rank, …)

Will block and never return, because MPI_Recv cannot be called in the right-hand MPI process

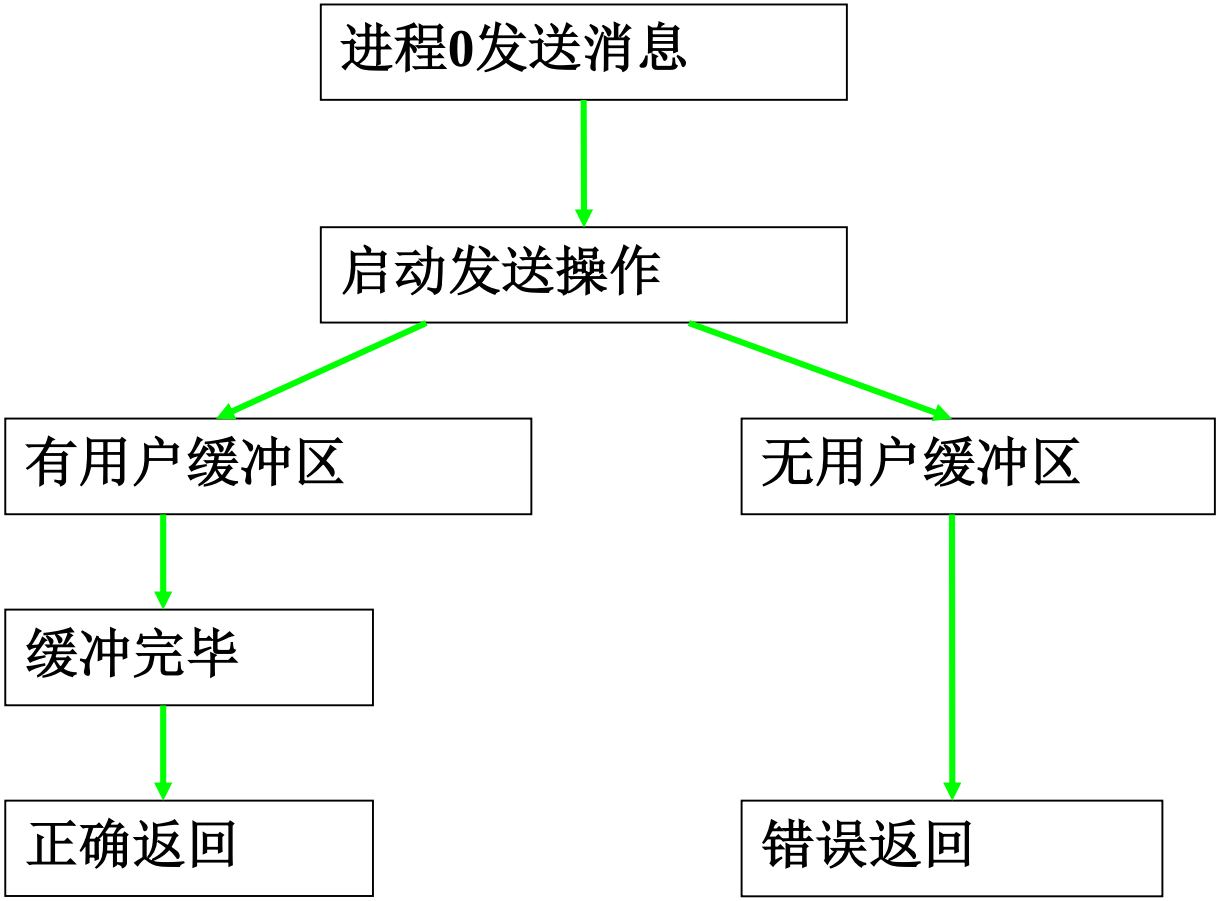



- Same problem with standard send mode (MPI_Send), if MPI implementation chooses synchronous protocol

# 缓存通信模式

- MPI_Bsend
- 由用户直接对通信缓冲区进行申请、使用和释放。
- 缓存模式下对通信缓冲区的合理与正确使用由程序设计人员自己保证。
- MPI_BSEND的各个参数的含义和MPI_SEND的完全相同，不同之处仅表现在通信时是使用标准的系统提供的缓冲区还是用户自己提供的缓冲区。

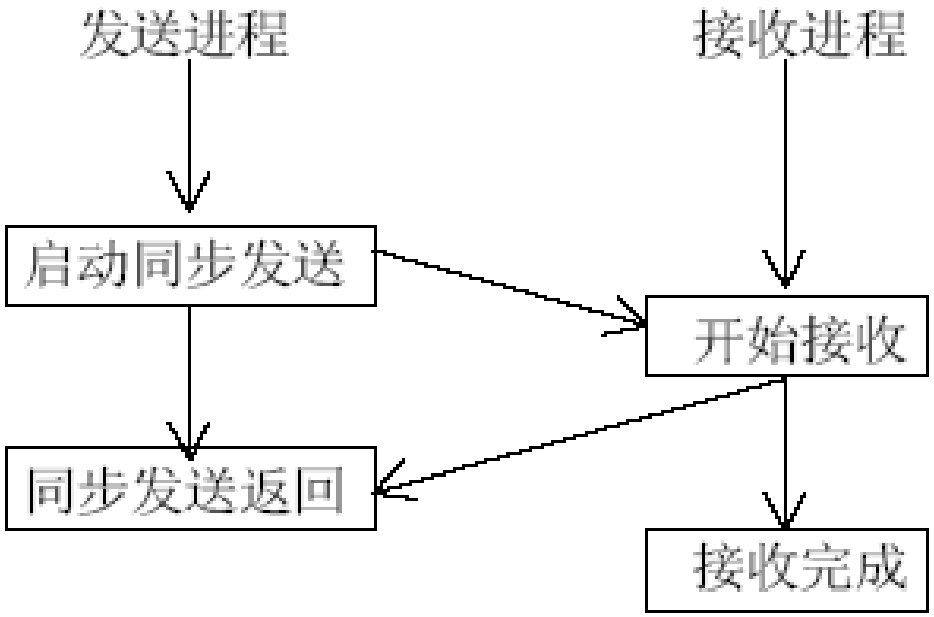
# 缓存通信模式



# 同步通信模式

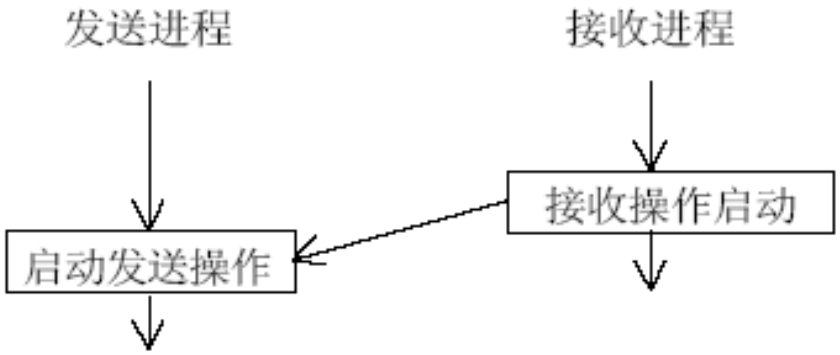
- MPI_Ssend
- 同步通信模式的开始不依赖于接收进程相应的接收操作是否已经启动，但是**同步发送**必须等到相应的接收进程开始后才可以正确返回。
- 因此，同步发送返回后，意味着发送缓冲区中的数据已经全部被系统缓冲区缓存并且已经开始发送。
- 这样，当同步发送返回后发送缓冲区可以被释放或重新使用。

# 同步通信模式



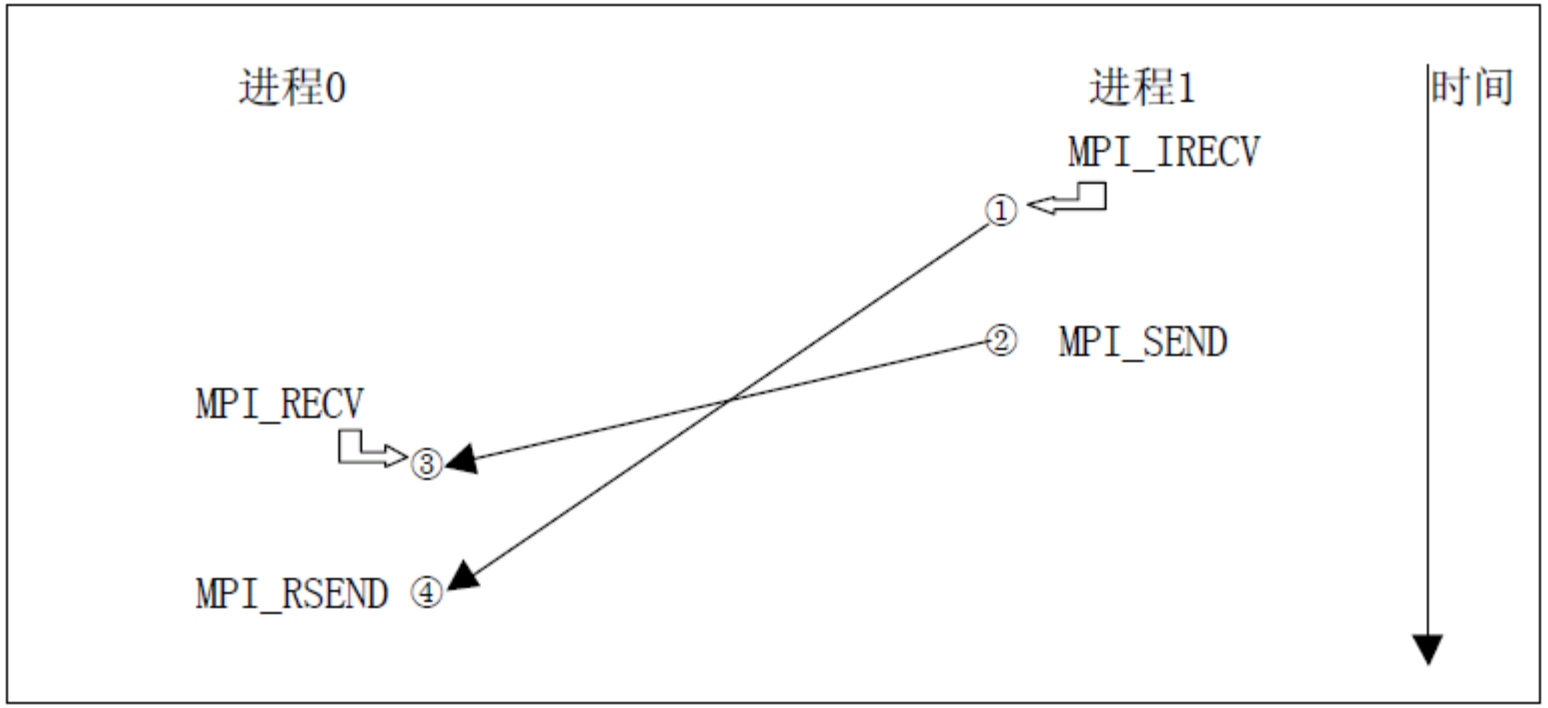
# 就绪通信模式

- MPI_Rsend
- 在就绪通信模式中，只有当接收进程的接收操作已经启动时，才可以在发送进程启动发送操作，否则，当发送操作启动而相应的接收还没有启动时，发送操作将出错。
- 对于非阻塞发送操作的正确返回，并不意味着发送已完成；但对于阻塞发送的正确返回，则发送缓冲区可以重复使用。

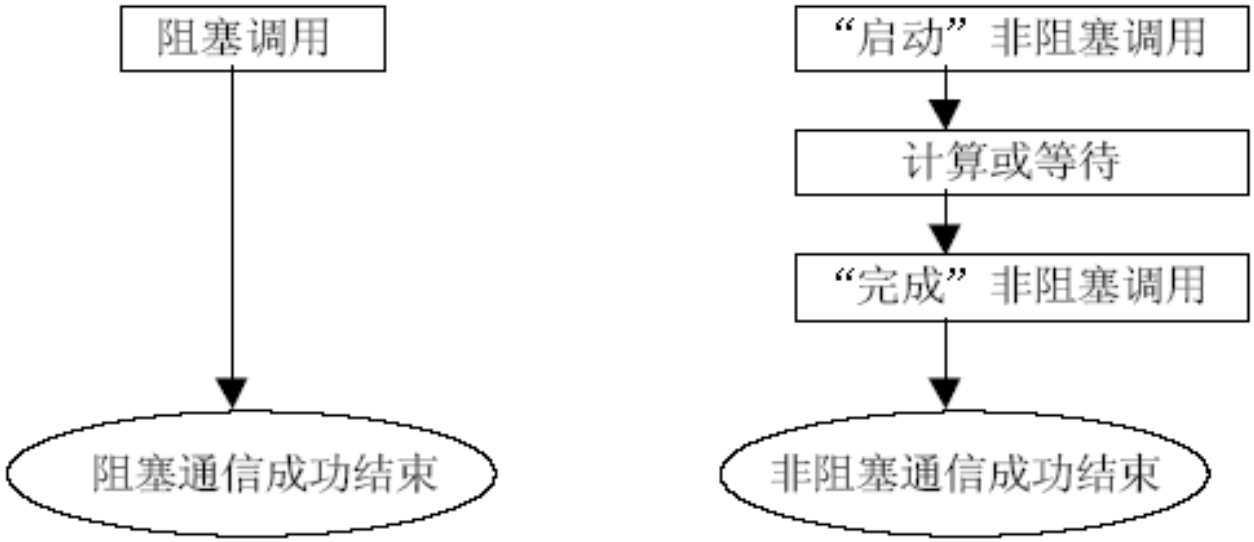


# 就绪通信模式

- 一种安全的就绪通信模式

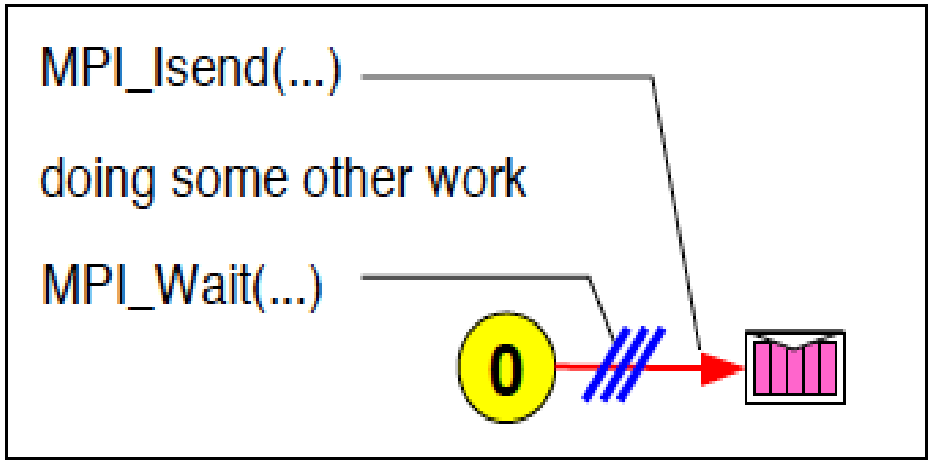


# 非阻塞通信

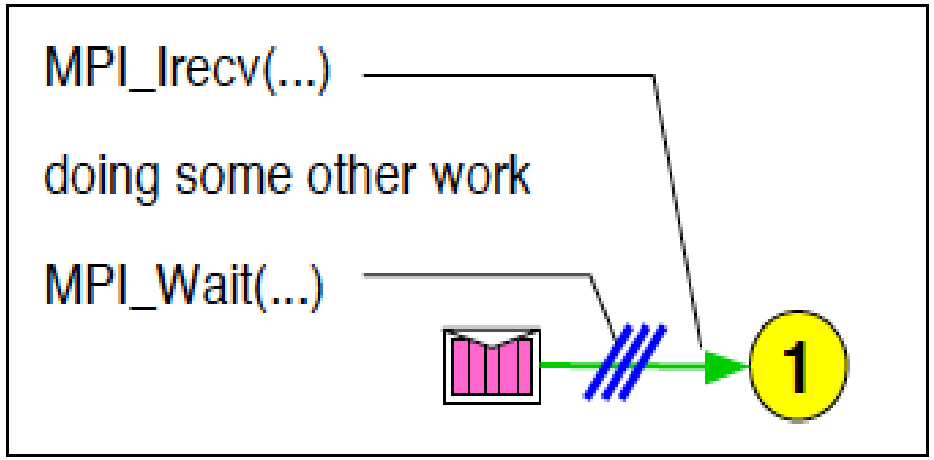


# 非阻塞操作

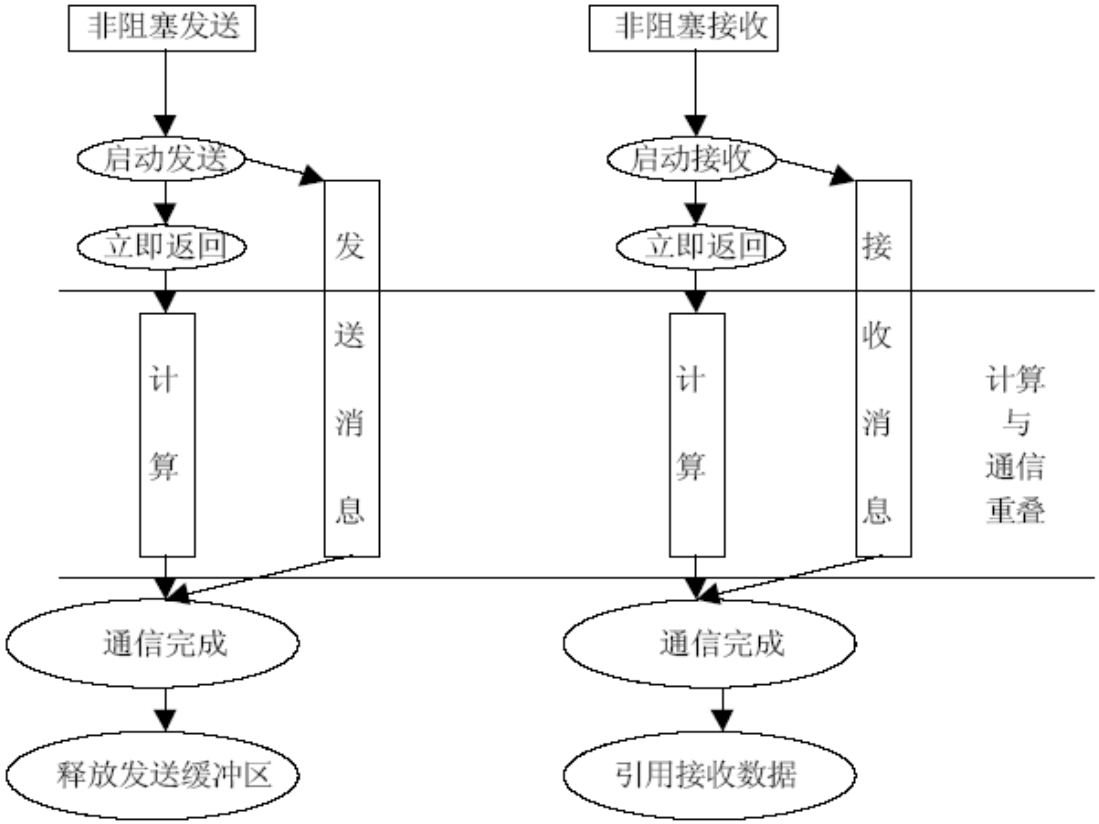
Non-blocking **send**




Non-blocking **receive**



# 非阻塞标准发送和接收


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

# MPI_Isend

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

- MPI_Request：非阻塞通信对象
  - MPI内部的对象，通过一个句柄存取。
  - 识别非阻塞通信操作的各种特性
    - 发送模式
    - 和它联结的通信缓冲区
    - 通信上下文
    - 用于发送的标识和目的参数
    - 用于接收的标识和源参数

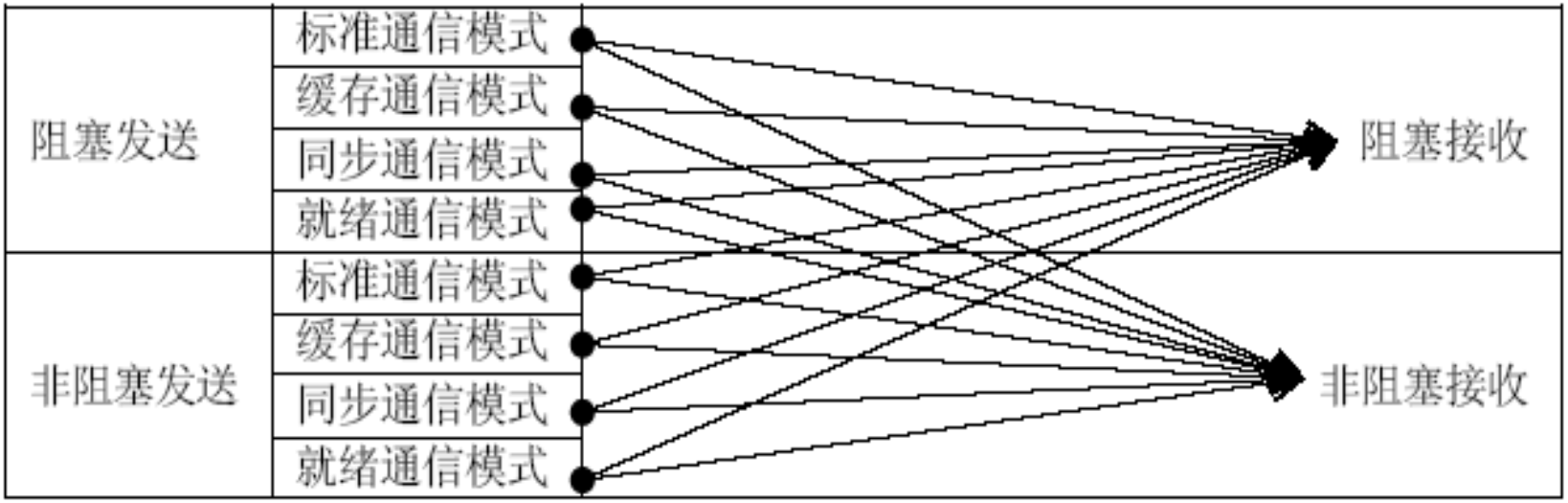
# 非阻塞通信与其它三种通信模式的组合

- 对于阻塞通信的四种消息通信模式：标准通信模式，缓存通信模式，同步通信模式和接收就绪通信模式，非阻塞通信也具有相应的四种不同模式。
- MPI使用与阻塞通信一样的命名约定，前缀B、S、R分别表示缓存通信模式、同步通信模式和就绪通信模式。
- 前缀I(immediate)表示这个调用是非阻塞的。

# 非阻塞MPI通信模式

| 通信模式 | | 发送 | 接收 |
|---|---|---|---|
| 标准通信模式 | | MPI_ISEND | MPI_IRECV |
| 缓存通信模式 | | MPI_IBSEND | |
| 同步通信模式 | | MPI_ISSEND | |
| 就绪通信模式 | | MPI_IRSEND | |
| 重复<br>非阻塞通信 | 标准通信模式 | MPI_SEND_INIT | MPI_RECV_INIT |
| | 缓存通信模式 | MPI_BSEND_INIT | |
| | 同步通信模式 | MPI_SSEND_INIT | |
| | 就绪通信模式 | MPI_RSEND_INIT | |

# 不同类型的发送与接收的匹配



# 非阻塞通信的完成

- 对于非阻塞通信，通信调用的返回并不意味着通信的完成，因此需要专门的通信语句来完成或检查该非阻塞通信。
- 不管非阻塞通信是什么样的形式，对于完成调用是不加区分的。
- 当非阻塞完成调用结束后，就可以保证该非阻塞通信已经正确完成了。

# 非阻塞通信的完成与检测

| 非阻塞通信的数量 | 检测 | 完成 |
| --- | --- | --- |
| 一个非阻塞通信 | MPI_TEST | MPI_WAIT |
| 任意一个非阻塞通信 | MPI_TESTANY | MPI_WAITANY |
| 一到多个非阻塞通信 | MPI_TESTSOME | MPI_WAITSOME |
| 所有非阻塞通信 | MPI_TESTALL | MPI_WAITALL |

# 阻塞与非阻塞操作总结

- 阻塞操作
  - 阻塞发送的返回，意味着发送缓冲区可被再次使用，而不会影响接收方，但并不意味接收方已经完成接收（有可能保存在系统缓冲区内）
  - 阻塞发送可以同步方式工作，发送方和接收方需要实施一个握手协议来确保发送动作的安全
  - 阻塞发送可以异步进行，此时需要系统缓冲区进行缓存
  - 阻塞接收操作仅当消息接收完成后才返回

# 阻塞与非阻塞操作总结

- 非阻塞操作
  - 非阻塞的发送和接收，在调用后都可以立即返回，不会等待任何与通信相关的事件
  - 非阻塞只对MPI环境提出一个要求 – 在可能的时候启动通信。用户无法预测通信何时发生
  - 在通过某种手段确定MPI环境确实执行了通信之前，修改发送缓冲区的数据是不安全的
  - 非阻塞通信的主要目的是把计算和通信重叠起来，从而改进并行效率

# Outline

- MPI概述
- 点到点通信/组通信
- 阻塞通信/非阻塞通信
- **MPI_Sendrecv和虚进程**
- 自定义数据类型
- 虚拟进程拓扑

# MPI_Sendrecv (捆绑发送接收)

- Jacobi迭代例子中，每一个进程都要向相邻的进程发送数据，同时从相邻的进程接收数据。
  - 潜在死锁，且算法逻辑复杂
- MPI提供了MPI_Sendrecv（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

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

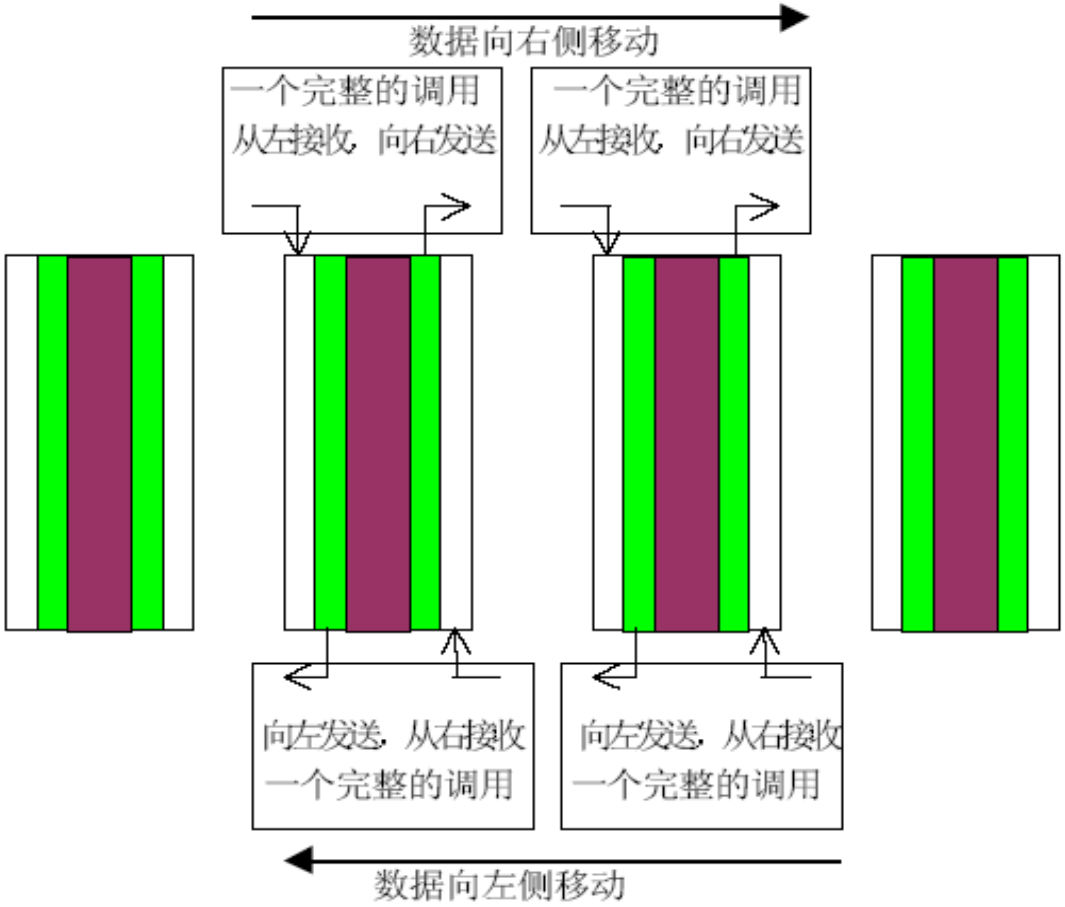
- 捆绑发送接收操作是不对称的，即一个由捆绑发送接收调用发出的消息可以被一个普通接收操作接收，一个捆绑发送接收调用可以接收一个普通发送操作发送的消息。

- 该操作执行一个阻塞的发送和接收，接收和发送使用同一个通信域。

- 发送缓冲区和接收缓冲区必须分开，可以是不同的数据长度和不同的数据类型。

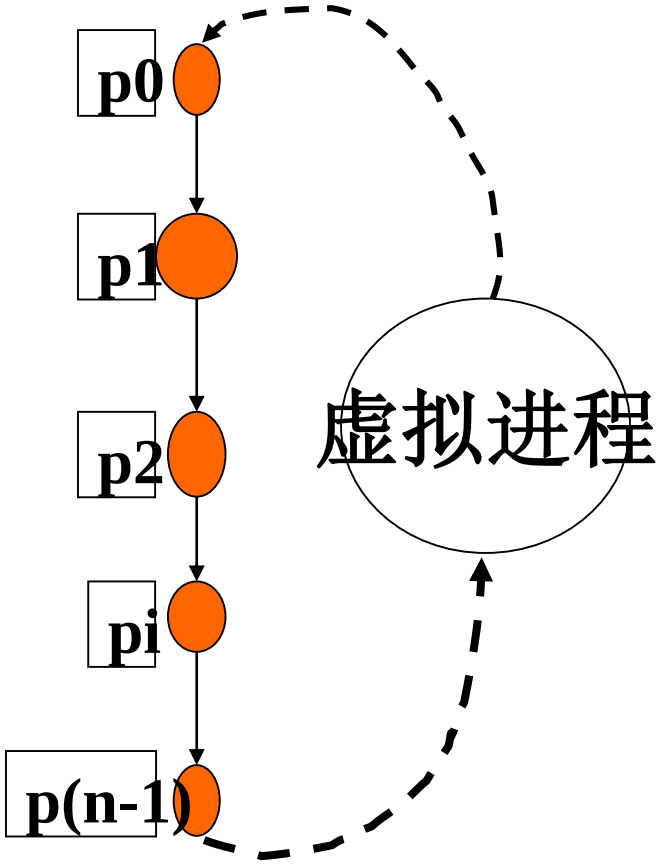
- 在语义上等同于一个发送操作和一个接收操作的结合
- 但可以有效地避免由于单独书写发送或接收操作时，由于次序的错误而造成的死锁因为该操作由通信系统来实现，系统会优化通信次序从而有效地避免不合理的通信次序，最大限度避免死锁的产生

# 用MPI_Sendrecv实现Jacobi迭代



# 虚拟进程

- 虚拟进程（MPI_PROC_NULL）是不存在的假想进程，在MPI中的主要作用是充当真实进程通信的目或源。
- 引入虚拟进程的目的是为了在某些情况下编写通信语句的方便。
- 当一个真实进程向一个虚拟进程发送数据或从一个虚拟进程接收数据时，该真实进程会立即正确返回，如同执行了一个空操作。



# 使用MPI_Sendrecv和虚拟进程的数据交换

```
if (myid > 0)
left= myid - 1;
else
left= MPI_PROC_NULL;
if (myid < n)
right= myid + 1;
else
right= MPI_PROC_NULL;

//从左向右平移数据
MPI_Sendrecv ( sendData1, sendCount, MPI_FLOAT, right, tag1, recvData1,recvCount, MPI_FLOAT,
left, tag1, MPI_COMM_WORLD, status)
//从右向左平移数据
MPI_Sendrecv ( sendData2, sendCount, MPI_FLOAT, left, tag1, recvData2,recvCount, MPI_FLOAT,
right, tag1, MPI_COMM_WORLD, status)
```

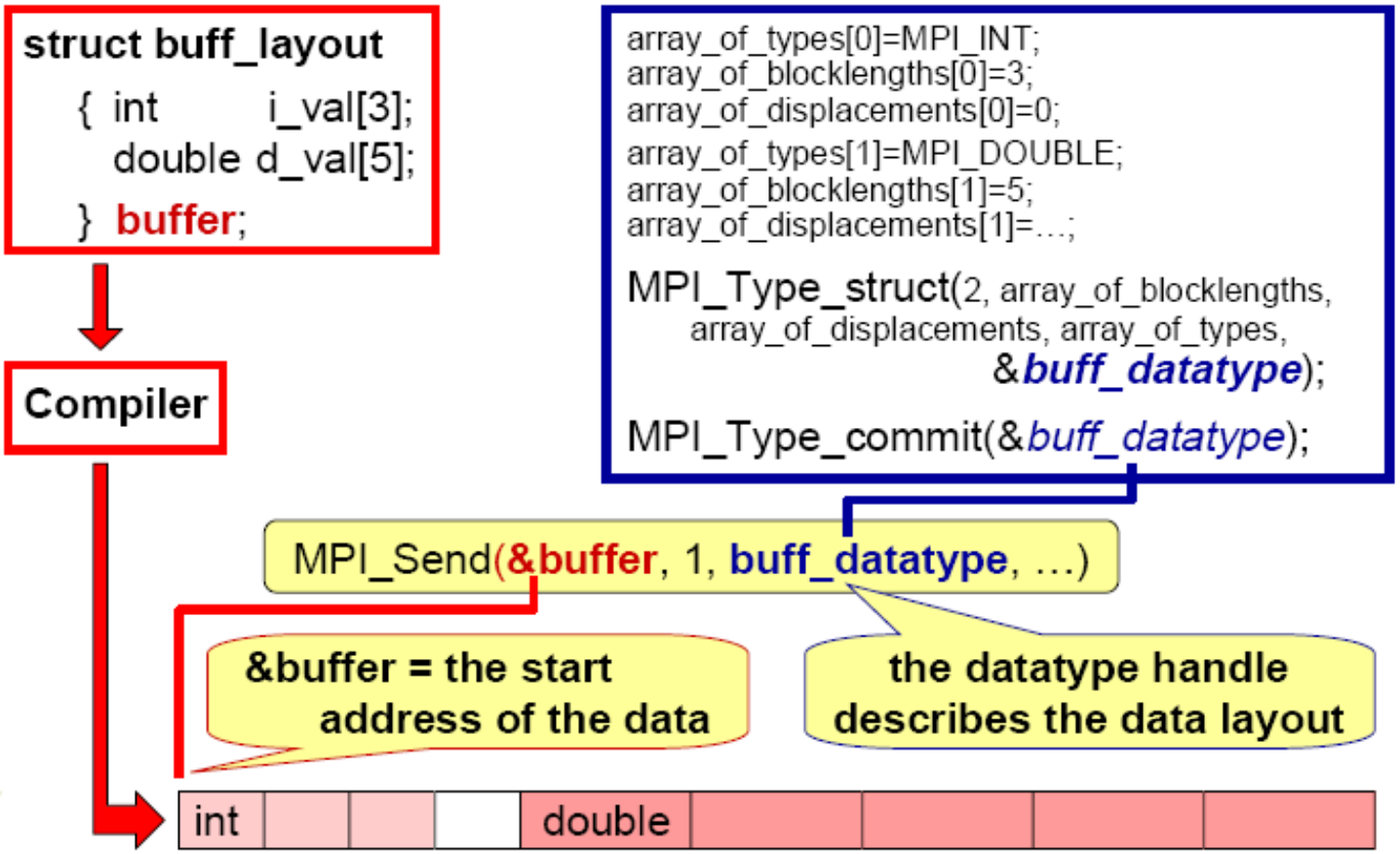
# Outline

- MPI概述
- 点到点通信/组通信
- 阻塞通信/非阻塞通信
- MPI_Sendrecv和虚进程
- **自定义数据类型**
- 虚拟进程拓扑

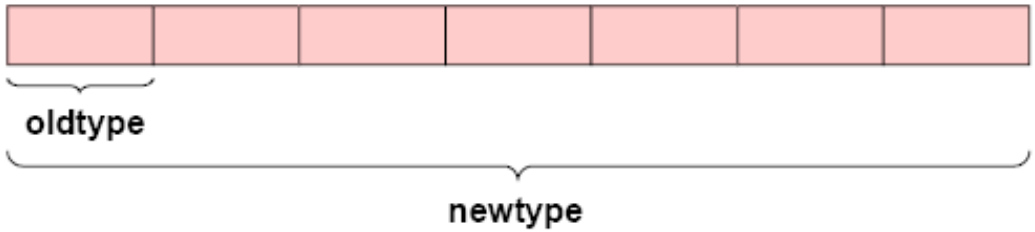
# MPI基本数据类型

| 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

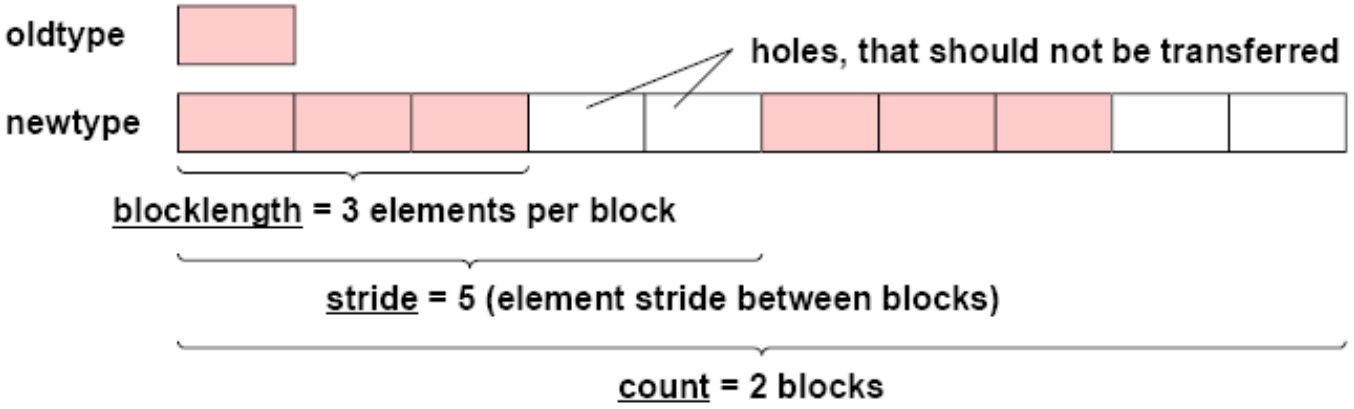

struct buff_layout
{ int i_val[3];
double d_val[5];
} buffer;
Compiler
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=…;
MPI_Type_struct(2, array_of_blocklengths, array_of_displacements, 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

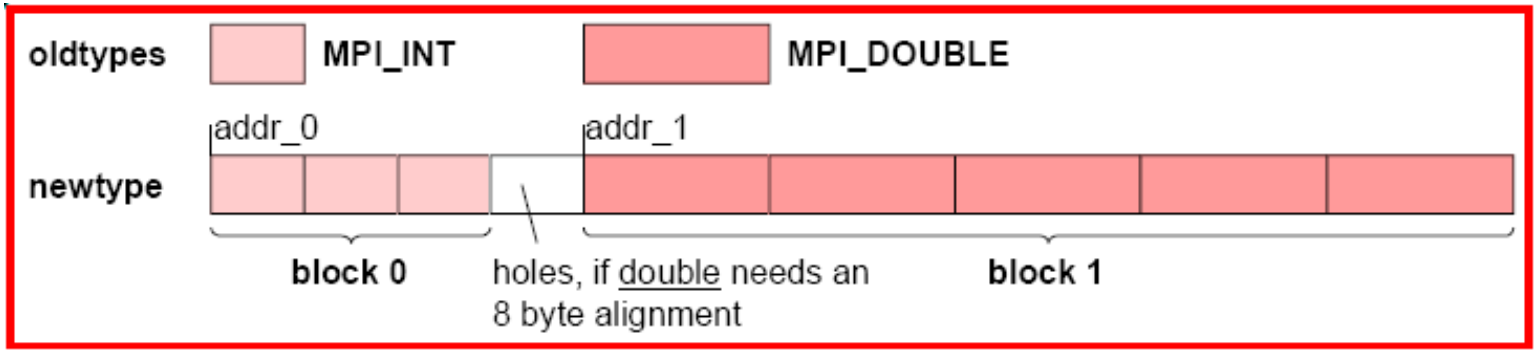
# 自定义数据类型：向量




- 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



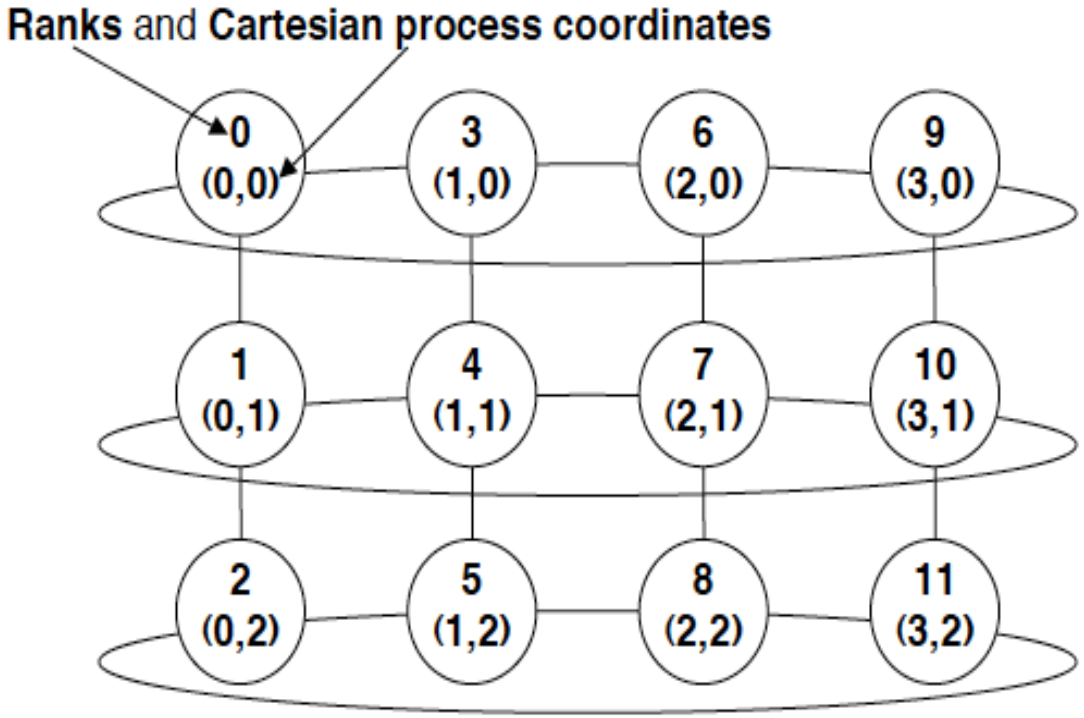

- C: int MPI_Type_struct(int count, int *array_of_blocklengths,
  MPI_Aint *array_of_displacements,
  MPI_Datatype *array_of_types, MPI_Datatype *newtype)
- Fortran: MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
  ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES,
  NEWTYPE, IERROR)

# Outline

- MPI概述
- 点到点通信/组通信
- 阻塞通信/非阻塞通信
- MPI_Sendrecv和虚进程
- 自定义数据类型
- **虚拟进程拓扑**

# 虚拟进程拓扑

- 进程的线性排列不能直接地反映进程间在逻辑上的通信模型(通常由问题几何和所用的算法决定)。进程经常被排列成二维或三维网格形式的拓扑模型，而且通常用一个图来描述逻辑进程排列。这种逻辑进程排列称为**虚拟拓扑**。



# 虚拟进程拓扑

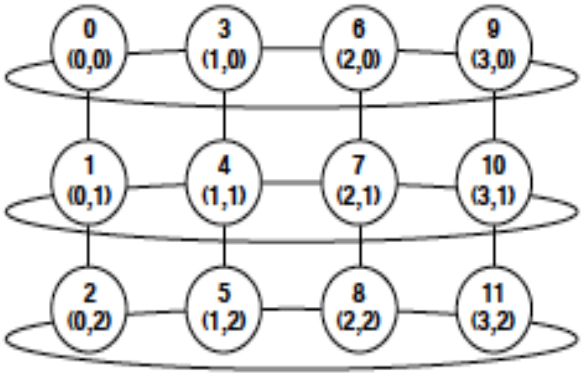
- 笛卡儿拓扑
  - 每个进程处于一个虚拟的网格内，与其邻居通信
  - 边界可以构成环
  - 通过笛卡尔坐标来标识进程
  - 任何两个进程也可以通信
- 图拓扑
  - 适用于复杂通信形

# 创建虚拟拓扑

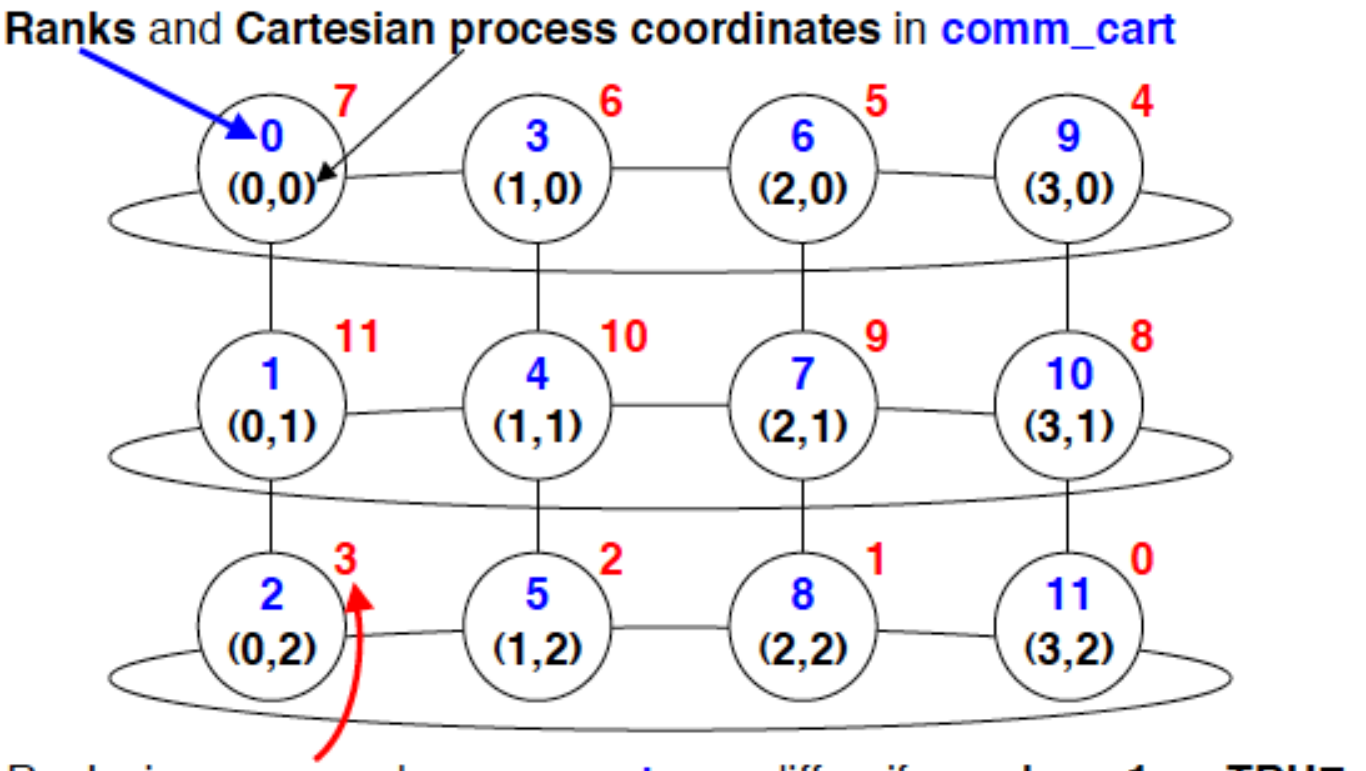
- C: int MPI_Cart_create(MPI_Comm comm_old, int ndims,
  int *dims, int *periods, int reorder,
  MPI_Comm *comm_cart)

- Fortran: MPI_CART_CREATE( COMM_OLD, NDIMS, DIMS, PERIODS,
  REORDER, COMM_CART, IERROR)
  INTEGER COMM_OLD, NDIMS, DIMS(*)
  LOGICAL PERIODS(*), REORDER
  INTEGER COMM_CART, IERROR

```
comm_old = MPI_COMM_WORLD
   ndims = 2
    dims = ( 4,        3        )
 periods = ( 1/.true., 0/.false. )
 reorder = see next slide
```

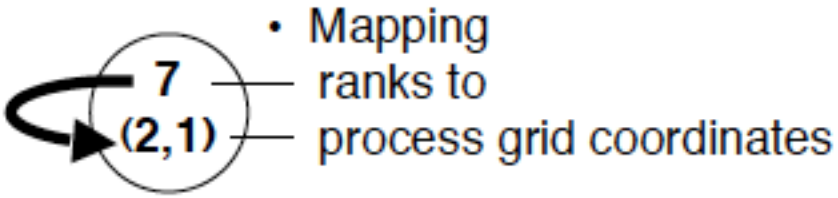


# 创建虚拟拓扑


Ranks and Cartesian process coordinates in comm_cart
0 (0,0)
7
3 (1,0)
6
6 (2,0)
5
9 (3,0)
4
1 (0,1)
11
4 (1,1)
10
7 (2,1)
9
10 (3,1)
8
2 (0,2)
3
5 (1,2)
2
8 (2,2)
1
11 (3,2)
0
Ranks in comm and comm_cart may differ, if reorder = 1 or .TRUE.
This reordering can allow MPI to optimize communications

# 进程序号到迪卡尔坐标的映射

- 给定进程号，返回该进程的迪卡尔坐标



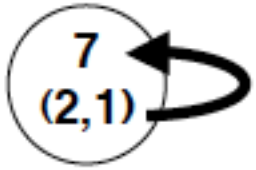

- C: int MPI_Cart_coords(MPI_Comm comm_cart, int rank,
  int maxdims, int *coords)

- Fortran: MPI_CART_COORDS(COMM_CART, RANK,
  MAXDIMS, COORDS, IERROR)
  INTEGER COMM_CART, RANK
  INTEGER MAXDIMS, COORDS(*), IERROR

# 迪卡尔坐标到进程序号的映射

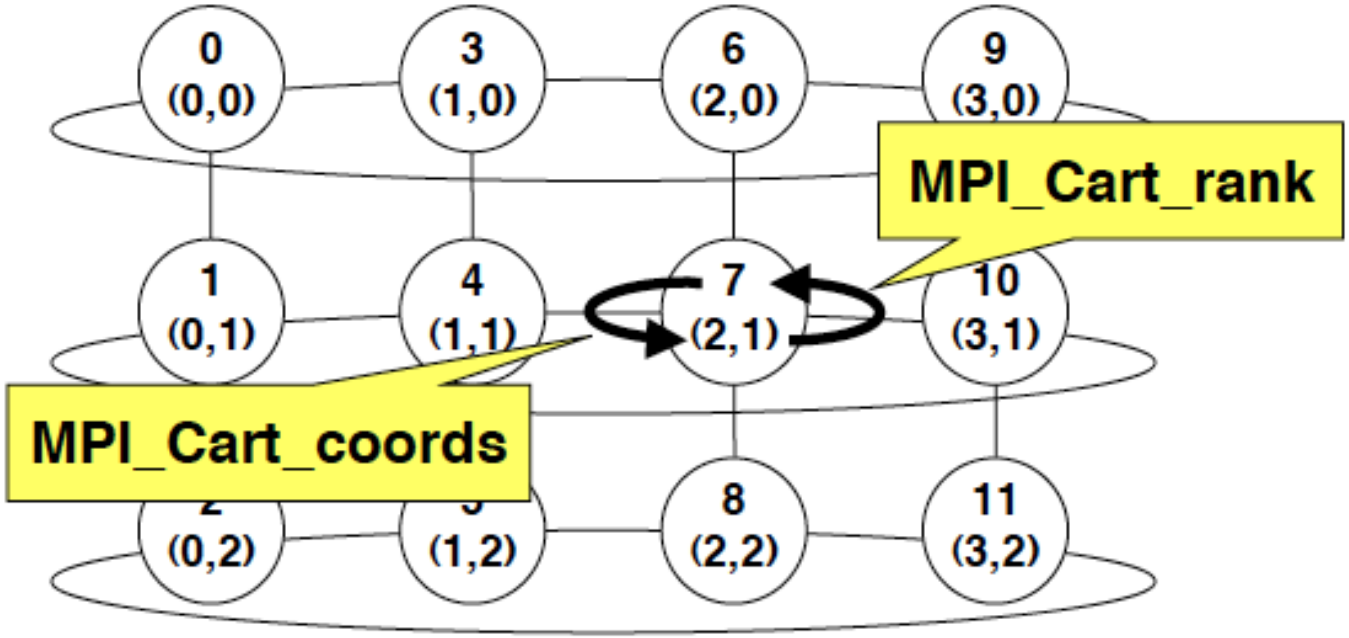
- 给定迪卡尔坐标，返回进程号

- Mapping process grid coordinates to ranks




- C: int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int **rank*)
- Fortran: MPI_CART_RANK(COMM_CART, COORDS, *RANK*, *IERROR*)
  INTEGER COMM_CART, COORDS(*)
  INTEGER RANK, IERROR

# 计算当前进程的坐标

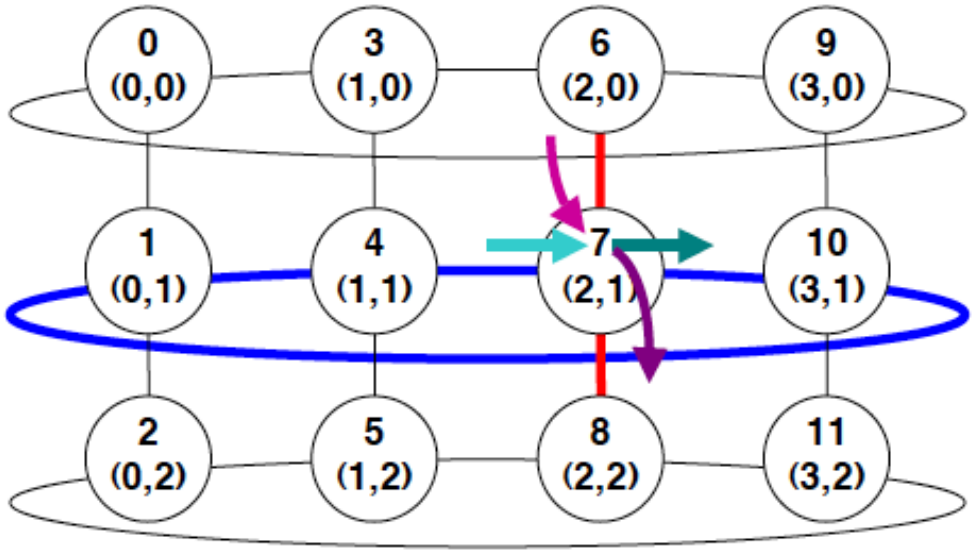



Each process gets its own coordinates with
MPI_Comm_rank(comm_cart, ***my_rank***, *ierror*)
MPI_Cart_coords(comm_cart, **my_rank**, maxdims, ***my_coords***, *ierror*)

# 数据平移

int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp,
int *rank_source, int *rank_dest)

- 计算相邻进程的rank
- 如果没有邻居，返回 MPI_PROC_NULL

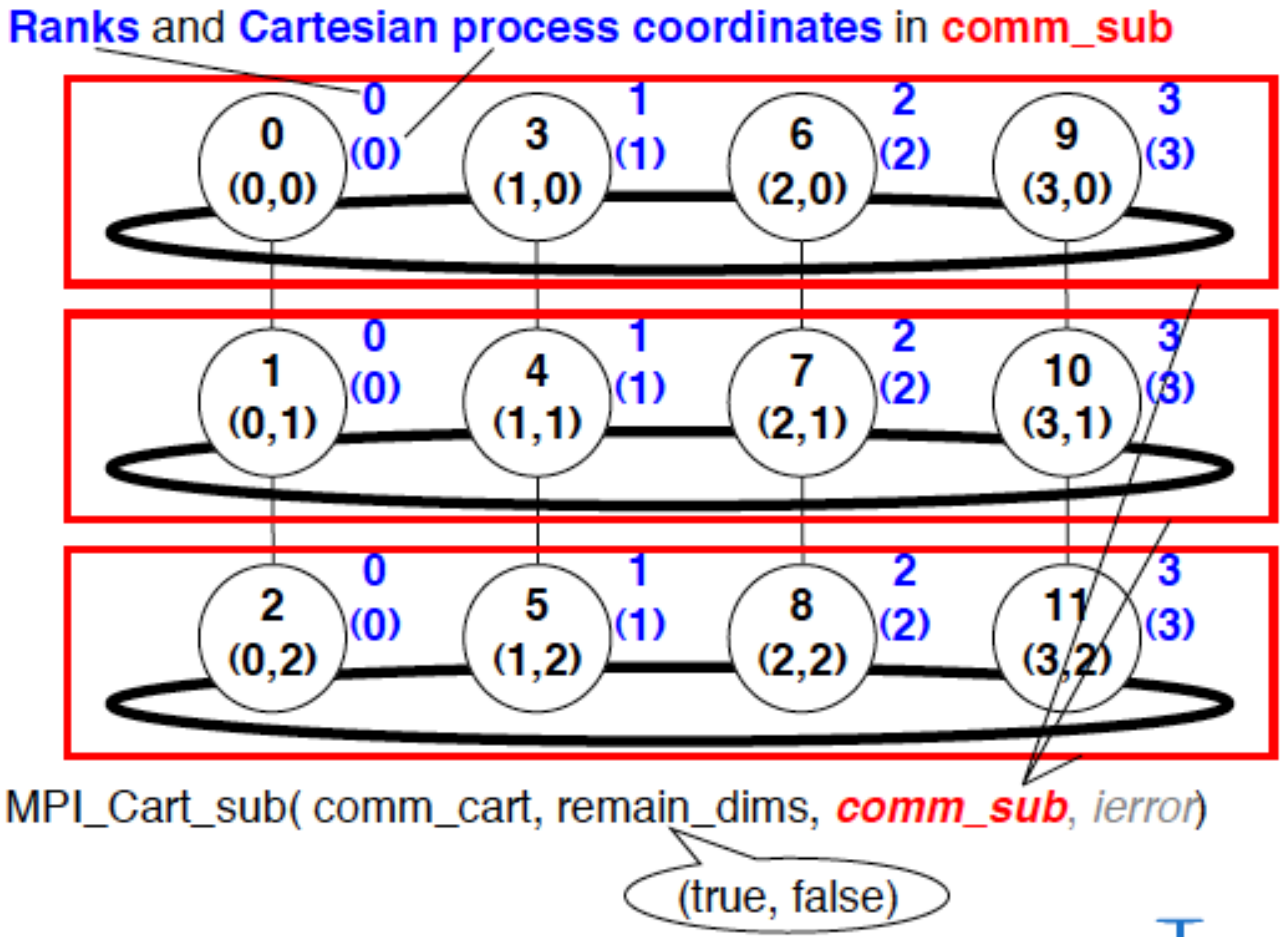



invisible input argument: **my_rank** in cart

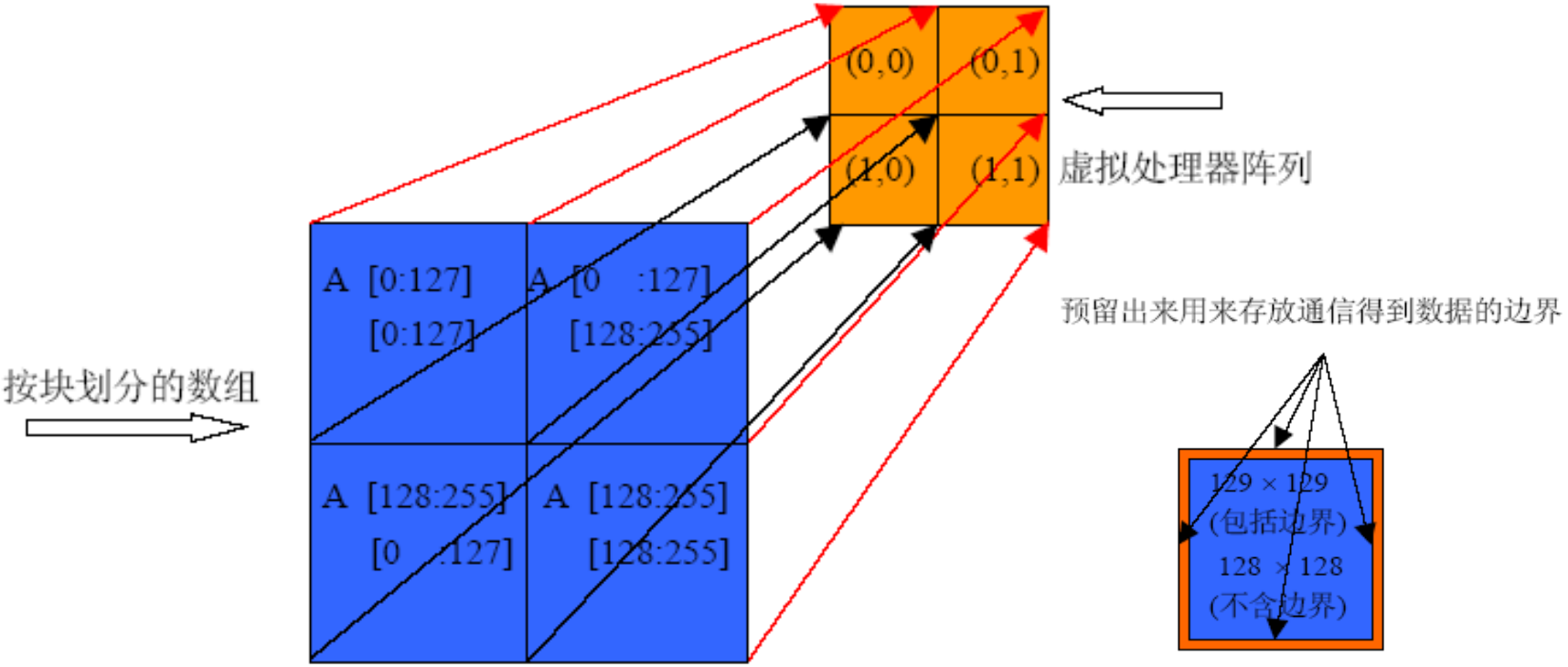
MPI_Cart_shift( cart, direction, displace, *rank_source*, *rank_dest*, *ierror*)

| | | | | |
|---|---|---|---|---|
| example on | **0** or | **+1** | 4 | 10 |
| process rank=**7** | 1 | **+1** | 6 | 8 |

# MPI_Cart_sub（划分子拓扑）



# Jacobi迭代


(0,0)
(0,1)
(1,0)
(1,1)
虚拟处理器阵列
A [0:127] [0:127]
A [0 :127] [128:255]
A [128:255] [0 :127]
A [128:255] [128:255]
按块划分的数组
预留出来用来存放通信得到数据的边界
129 × 129
(包括边界)
128 × 128
(不含边界)

# Jacobi迭代