# SIMD 、MIMD、SPMD、MPMD

SIMD Single-Instruction Multiple-Data 单指令多数据并行计算机

MIMD Multiple-Instruction Multiple-Data 多指令多数据并行计算机

SPMD Single-Program Multuple-Data 单程序多数据并行计算机

MPMD Multiple-ProgramMultiple-Data 多程序多数据并行计算机

# 共享内存的并行计算机

各个处理单元通过对共享内存的访问来交换信息 协调各

处理器对并行任务的处理 对这种共享内存的编程 实现起来相对简单 但共享内存往往成

为性能特别是扩展性的重要瓶颈

# 分布式内存的并行计算机

各个处理单元都拥有自己独立的局部存储器 由于不存

在公共可用的存储单元 因此各个处理器之间通过消息传递来交换信息 协调和控制各个处

理器的执行，复杂的消息传递语句的编写成为在这种并行计算机上进行并行程序设计的难点所在 但是 对于这种类型的并行计算机 由于它有很好的扩展性和很高的性能 因此 它的应用非常广泛

# 并行编程模式（数据并行）

数据并行即将相同的操作同时作用于不同的数据 因此适合在SIMD及SPMD并行计算机上运行，数据并行是可以高效地解决一大类科学与工程计算问题，对于编程者来说 只需要简单地指明执行什么样的并行操作和并行操作的对象 就实现了数据并行的编程。

**缺**：但是对于非数据并行类的问题 如果通过数据并行的方式来解决 一般难以取得较高的效率 数据并行不容易表达甚至无法表达其它形式的并行特征。

# 并行编程模式（消息传递）

即各个并行执行的部分之间通过传递消息来交换信息 协调步伐 控制执行，消息传递为编程

者提供了更灵活的控制手段和表达并行的方法 一些用数据并行方法很难表达的并行算法

都可以用消息传递模型来实现 灵活性和控制手段的多样化 是消息传递并行程序能提供高

的执行效率的重要原因。

**缺**：它也将各个并行执行部分之间复杂的信息交换和协调 控制的任务交给了编程者 这在一定程度上增加了编程者的负担

并行编程模型除了数据并行和消息传递之外 还有共享变量模型 函数式模型等等

# 什么是MPI

1. MPI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

1. 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。
2. MPI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

# MPI 实现的 Hello World!

1. #include<mpi.h>
2. MPI\_MAX\_PROCESSOR\_NAME：MPI预定义的宏 即某一MPI的具体实现中允许机器名字的最大长度
3. MPI程序的开始和结束必须是MPI\_INIT和MPI\_FINALIZE 分别完成MPI程序的初始化和结束工作
4. MPI程序的程序体 包括各种MPI过程调用语句。

# MPI 程序的一些惯例

所有MPI的名字都有前缀 MPI\_ 不管是常量 变量还是过程或函数调用的名字都是

这样, 而C形式的MPI调用 则为MPI\_Aaaa\_aaa的形式

# 类似java的重载（多态）

在C和FORTRAN77的说明中，对void\*和,<type>需要进行特殊说明 MPI的库和一般的C

和FORTRAN77库在语法上基本上是相同的 但是对于MPI的调用 允许不同的数据类型使

用相同的调用 比如对于数据的发送操作 整型 实型 字符型等都用一个相同的调用

MPI\_SEND 对于这样的数据类型在C和FORTRAN77的原型说明中 分别用void \*和 <type>

来表示 即用户可根据通信的要求 对不同的数据类型 可以用相同的调用

# MPI\_Max\_processor\_name()

MPI预定义的宏 即某一MPI的具体实现中允许机器名字的最大长度

# MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)

这一调用返回调用进程在给定的通信域中的进程标识号 有了这一标识号 不同的进程

就可以将自身和其它的进程区别开来 实现各进程的并行和协作MPI\_Comm\_size()

# MPI\_COMM\_SIZE(MPI\_Comm comm, int \*size)

这一调用返回给定的通信域中所包括的进程的个数 不同的进程通过这一调用得知在给

定的通信域中一共有多少个进程在并行执行

# MPI\_Get\_processor\_name()

MPI\_GET\_PROCESSOR\_NAME得到运行本进程的机器的名称\

# MPI\_Init(int \*argc, char \*\*\*argv)

MPI\_INIT是MPI程序的第一个调用 它完成MPI程序所有的初始化工作 所有MPI程序

的第一条可执行语句都是这条语句

# MPI\_Finalize(void)

MPI\_FINALIZE是MPI程序的最后一个调用 它结束MPI程序的运行 它是MPI程序的最

后一条可执行语句 否则程序的运行结果是不可预知的

# MPI\_Send(void\* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm

# comm)

IN buf 发送缓冲区的起始地址(可选类型)

IN count 将发送的数据的个数(非负整数)

IN datatype 发送数据的数据类型(句柄)

IN dest 目的进程标识号(整型)

IN tag 消息标志(整型)

IN comm 通信域(句柄)

MPI\_SEND将发送缓冲区中的count个datatype数据类型的数据发送到目的进程 目的进

程在通信域中的标识号是dest 本次发送的消息标志是tag 使用这一标志 就可以把本次发

送的消息和本进程向同一目的进程发送的其它消息区别开来

MPI\_SEND操作指定的发送缓冲区是由count个类型为datatype的连续数据空间组成 起

始地址为buf 注意这里不是以字节计数 而是以数据类型为单位指定消息的长度 这样就

独立于具体的实现 并且更接近于用户的观点

# MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype, int source, int tag,

# MPI\_Comm comm, MPI\_Status \*status)

OUT buf 接收缓冲区的起始地址(可选数据类型)

IN count 最多可接收的数据的个数(整型)

IN datatype 接收数据的数据类型(句柄)

IN source 接收数据的来源即发送数据的进程的进程标识号(整型)

IN tag 消息标识 与相应的发送操作的表示相匹配相同(整型)

IN comm 本进程和发送进程所在的通信域(句柄)

OUT status 返回状态 (状态类型)

MPI\_RECV从指定的进程source接收消息 并且该消息的数据类型和消息标识和本接收

进程指定的datatype和tag相一致 接收到的消息所包含的数据元素的个数最多不能超过

count

接收缓冲区是由count个类型为datatype的连续元素空间组成 由datatype指定其类型 起

始地址为buf 接收到消息的长度必须小于或等于接收缓冲区的长度 这是因为如果接收到

的数据过大 MPI没有截断 接收缓冲区会发生溢出错误 因此编程者要保证接收缓冲区的

长度不小于发送数据的长度

# int MPI\_Bcast(void\* buffer,int count,MPI\_Datatype datatype,int root, MPI\_Comm comm)

IN/OUT buffer 通信消息缓冲区的起始地址(可选数据类型)

IN count 将广播出去/或接收的数据个数(整型)

IN datatype 广播/接收数据的数据类型(句柄)

IN root 广播数据的根进程的标识号(整型)

IN comm 通信域(句柄)

MPI\_BCAST是一对多组通信的典型例子 它完成从一个标识为root的进程将一条消息

广播发送到组内的所有其它的进程 同时也包括它本身在内 在执行该调用时组内所有进程

不管是root进程本身还是其它的进程 都使用同一个通信域comm和根标识root 其执行结

果是将根进程通信消息缓冲区中的消息拷贝到其他所有进程中去

# 返回状态 status

返回状态变量status用途很广 它是MPI定义的一个数据类型 使用之前需要用户为它分

配空间

在C实现中 状态变量是由至少三个域组成的结构类型 这三个域分别是

MPI\_SOURCE

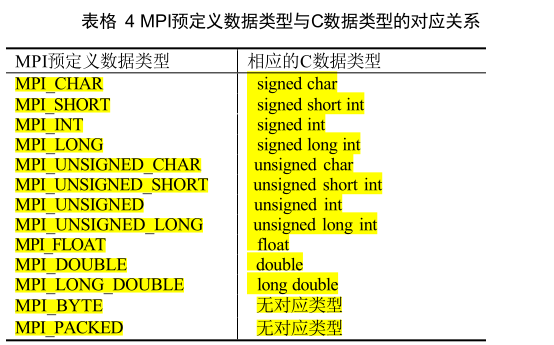
MPI\_TAG

MPI\_ERROR

它还可以包括其它的附加域

这样通过对status.MPI\_SOURCE status.MPI\_TAG和status.MPI\_ERROR的引用 就可以得到返回状态中所包含的发送数据进程的标识 发送数据使用的tag标识和本接收操作返回的错误代码，除了以上三个信息之外 对status变量执行MPI\_GET\_COUNT调用可以得到接收到的消息的长度信息

# MPI预定义数据类型与C数据类型的对应关系



# MPI的消息传递过程可以分为三个阶段

1 消息装配 将发送数据从发送缓冲区中取出 加上消息信封等形成一个完整的消息

2 消息传递 将装配好的消息从发送端传递到接收端

3 消息拆卸 从接收到的消息中取出数据送入接收缓冲区

在这三个阶段都需要类型匹配

1. 在消息装配时 发送缓冲区中变量的类型必须和相应的发送操作指定的类型相匹配
2. 在消息传递时 发送操作指定的类型必须和相应的接收操作指定的类型相互匹配
3. 在消息拆卸时 接收缓冲区中变量的类型必须和接收操作指定

的类型相匹配

在MPI中类型匹配有两个方面的意思

1. 宿主语言的类型和通信操作所指定的类型相匹配
2. 发送方和接收方的类型相匹配

上述类型匹配规则的例外是对于MPI提供的MPI\_BYTE和MPI\_PACKED 它们可以和任

何以字节为单位的存储相匹配 包含这些字节的类型是任意的 MPI\_TYPE用于不加修改地

传送内存中的二进制值 MPI\_PACK用于数据的打包和解包 MPI\_UNPACK

# 类型匹配规则可以概括为

1. 有类型数据的通信 发送方和接收方均使用相同的数据类型

2. 无类型数据的通信 发送方和接收方均以MPI\_BYTE作为数据类型

3. 打包数据的通信, 发送方和接收方均使用MPI\_PACKED

# 数据转换

数据类型的转换：指改变一个值的数据类型 比如将实型转换为整型 通过舍入操作或将整型转换为实型等

数据表示转换：指改变一个值的二进制表示 比如高字节和低字节顺序的改变

由于MPI严格要求类型匹配 所以在MPI中不存在数据类型转换的问题

但是 MPI必须实现数据表示的转换 这是因为MPI的目的之一是对异构环境的支持 在异构系统中 不同的系统其数据的内部表示往往是不同的 因此MPI必须负责实现这些不同表示之间的相互转换

# MPI 消息的组成

MPI消息包括信封和数据两个部分 信封指出了发送或接收消息的对象及相关信息 而

数据是本消息将要传递的内容 信封和数据又分别包括三个部分 可以用一个三元组来表示

信封 <源/目 标识 通信域>

数据 <起始地址 数据个数 数据类型>

为什么还有tag标识呢 这是因为 当发送者发送两个相同

类型的数据给同一个接收者时 如果没有消息标识 接收者将无法区别这两个消息

# 任意源和任意标识

接收者可以给source指定一个任意值MPI\_ANY\_SOURCE 标识任何进程发送的消息都可以接收 即本接收操作可以匹配任何进程发送的消息 但其它的要求还必须满足 比如tag的匹配 如果给tag一个任意值MPI\_ANY\_TAG 则任何tag都是可接收的 在某种程度上 类似于统配符的概念。但是不能给comm指定任意值。

即一个接收操作可以接收任何发送者的消息 但是对于一个发送操作 则必须指明一个单独的接收者MPI允许发送者=接收者 Source = destination 即一个进程可以给自己发送一个消息但是这种操作要注意死锁的产生

# 死锁

操作系统中的死锁被定义为**系统中两个或者多个进程无限期地等待永远不会发生的条件，系统处于停滞状态，这就是死锁。**

产生死锁的原因主要是：   
（1） 因为系统资源不足。   
（2） 进程运行推进的顺序不合适。   
（3） 资源分配不当等

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

# MPI 通信域

MPI通信域包括两部分 进程组和通信上下文 进程组即所有参加通信的进程的集合

如果一共有N个进程参加通信 则进程的编号从0到N-1 通信上下文提供一个相对独立的通

信区域 不同的消息在不同的上下文中进行传递 不同上下文的消息互不干涉 通信上下文

可以将不同的通信区别开来

一个预定义的通信域MPI\_COMM\_WORLD由MPI提供 MPI初始化后 便会产生这一

描述子 它包括了初始化时可得的全部进程 进程是由它们在MPI\_COMM\_WORLD组中的

进程号所标识

用户可以在原有的通信域的基础上 定义新的通信域 通信域为库和通信模式提供一种

重要的封装机制 他们允许各模式有其自己的独立的通信域 和它们自己的进程计数方案

## 进程组

进程组，每个进程组有一个领头进程。进程组是一个或多个进程的集合，通常它们与一组作业相关联，可以接受来自同一终端的各种信号。每个进程组都有一个组长进程，组长进程的进程号等于进程组ID。组长进程可以创建一个进程组、创建该组中的进程。只要某个进程组中有一个进程存在，则该进程组就存在，与组长进程是否终止无关。从进程组创建开始到其中最后一个进程离开为止的时间区间成为进程组的生存期。进程组中最后一个进程可以终止或者转移到另一个进程组中。

## 通信上下文

通信上下文提供一个相对独立的通

信区域 不同的消息在不同的上下文中进行传递 不同上下文的消息互不干涉 通信上下文

可以将不同的通信区别开来

# 简单的MPI程序示例

用 MPI 实现计时功能

C:\Users\ai\Desktop\并行&云计算\c源码\time.cpp

获取机器的名字和 MPI 版本号

C:\Users\ai\Desktop\并行&云计算\c源码\getMPIVersion.cpp

将指定的master结点杀掉

C:\Users\ai\Desktop\并行&云计算\c源码\abort.cpp

数据接力传送

C:\Users\ai\Desktop\并行&云计算\c源码\relay.cpp

# MPI\_Initialized(int \*flag)

在MPI程序中唯一一个可以用在MPI\_INIT之前的MPI调用是MPI\_INITALIZED 它的功

能就是判断MPI\_INIT是否已经执行。MPI\_INITALIZED判断当前进程是否已经调用了MPI\_INIT 若已调用 则flag=true 否

则flag=false

# MPI\_Abort(MPI\_Comm comm, int errorcode)

MPI\_ABORT使通信域comm中的所有进程退出 本调用并不要求外部环境对错误码采取

任何动作

C:\Users\ai\Desktop\并行&云计算\c源码\abort.cpp

# 互数据接力传送

C:\Users\ai\Desktop\并行&云计算\c源码\relay.cpp

# 任意进程间相互问候

C:\Users\ai\Desktop\并行&云计算\c源码\greeting.cpp

# int MPI\_Barrier(MPI\_Comm)

MPI\_Barrier 是MPI中的一个函数接口，函数定义形式为int MPI\_Barrier(MPI\_Comm);表示阻止调用直到communicator中所有进程完成调用。

# 虚拟进程（MPI\_PROC\_NULL）

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。

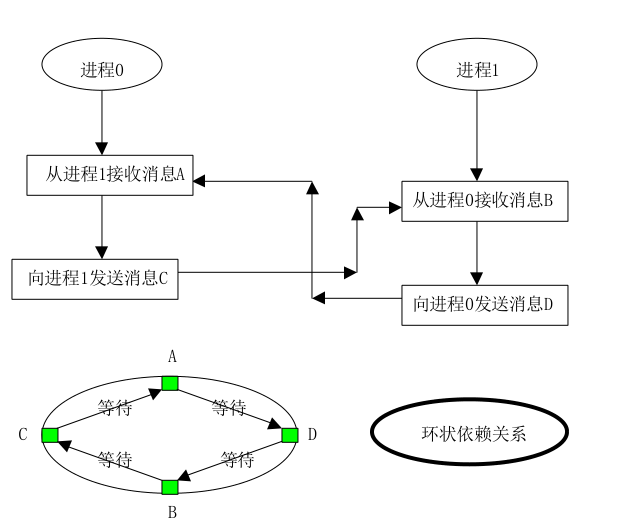
# 任意源和任意标识的使用

在接收操作中 通过使用任意源和任意tag标识 使得该接收操作可以接收任何进程以

任何标识发送给本进程的数据 但是该消息的数据类型必须和接收操作的数据类型相一致

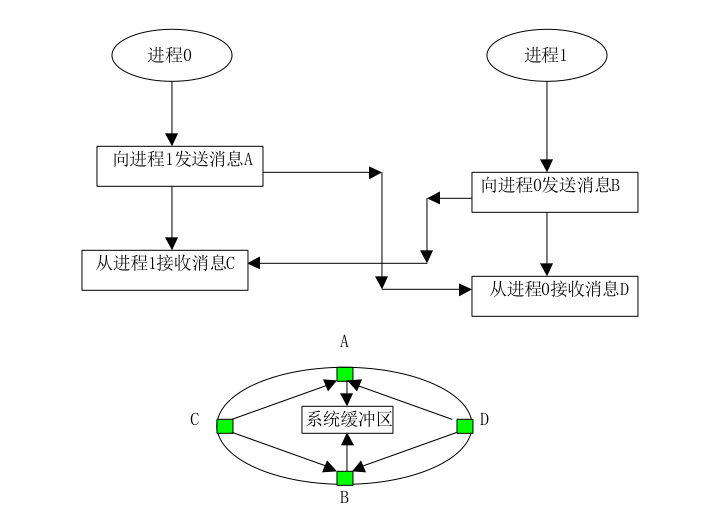
C:\Users\ai\Desktop\并行&云计算\c源码\any\_resource\_tag.cpp

# 总会死锁的发送接收序列



进程0和进程1相互等待 彼此都无法执行下去 必然导致死锁

# 不安全的发送接收序列



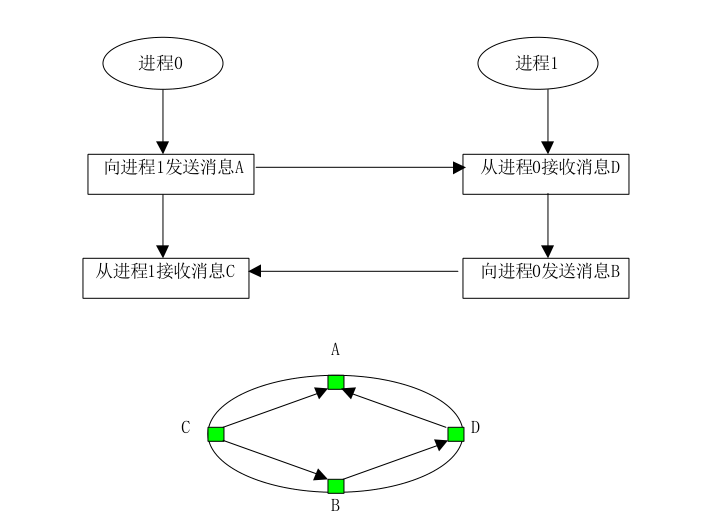
由于进程0或进程1的发送需要系统提供缓冲区 在MPI的四种通信模式中有详细的解

释 如果系统缓冲区不足 则进程0或进程1的发送将无法完成 相应的 进程1和进程0的

接收也无法正确完成.

# 安全的发送接收序列

当两个进程需要相互交换数据时 一定要将它们的发送和接收操作按照次序进行匹配，即一个进程的发送操作在前，接收操作在后。而另一个进程的接收操作在前，发送操作在后，前两个发送和接收操作要相互匹配。



# 对等模式的 MPI 程序设计

对等模式：多进程（process,not thread）并列运行，通过Recv和Send来实现进程间的数据通信；

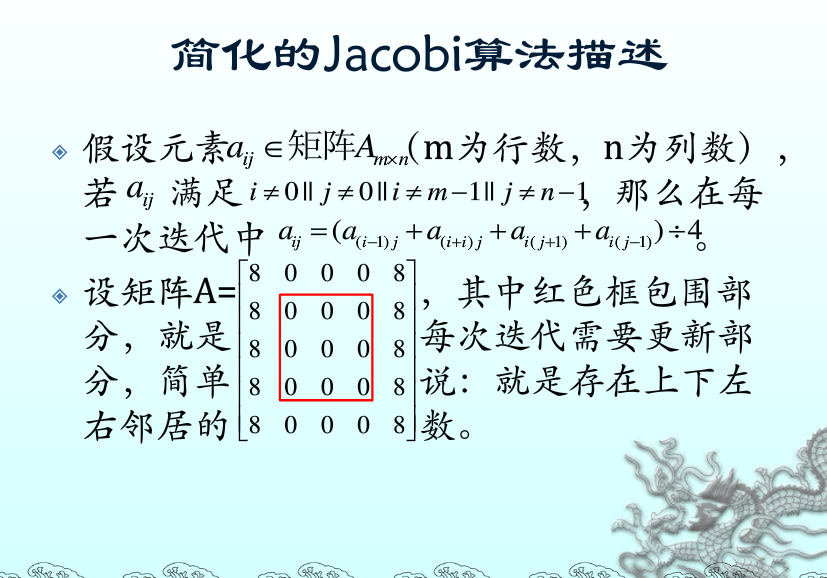
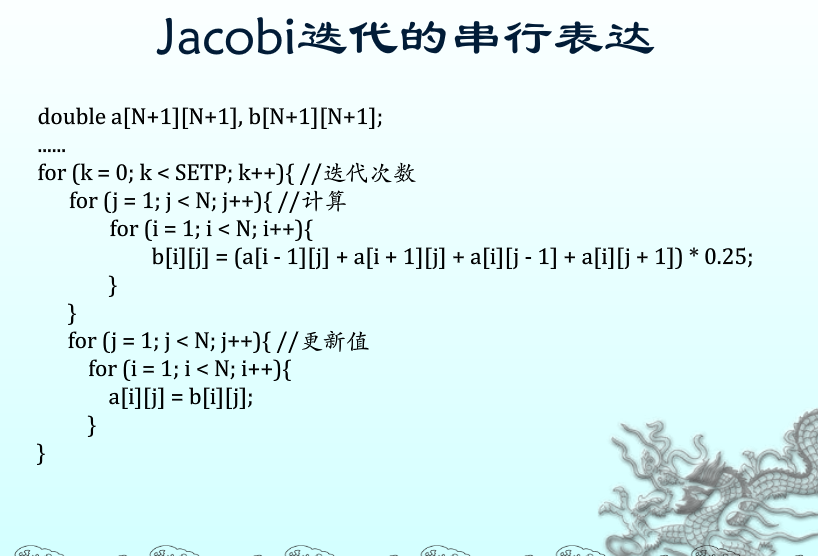
实现对等模式的问题是比较容易理解和接受的，因为各个部分地位相同，功能和代码基本一致，只不过是处理的数据或对象不同，也容易用同样的程序来实现

## Jacobi 迭代

简单地说，Jacobi迭代得到的新值是原来旧值点相邻数值点的平均

将参加迭代的数据按块分割后 各块之间除了相邻的元素需要通信外 在各块的内部可以完全独立地并行计算 随着计算规模的扩大 通信的开销相对于计算来说比例会降低 这将更有

利于提高并行效果



？？？？？？？？？？？？？？？？？？？？？？？？？？

## 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)

IN sendbuf 发送缓冲区起始地址(可选数据类型)

IN sendcount 发送数据的个数(整型)

IN sendtype 发送数据的数据类型(句柄)

IN dest 目标进程标识(整型)

IN sendtag 发送消息标识(整型)

OUT recvbuf 接收缓冲区初始地址(可选数据类型)

IN recvcount 最大接收数据个数(整型)

IN recvtype 接收数据的数据类型(句柄)

IN source 源进程标识(整型)

IN recvtag 接收消息标识(整型)

IN comm 通信域(句柄)

OUT status 返回的状态(status)

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，捆绑发送和接收操作把发送一个消息到一个目的地和从另一个进程接收一个消息合并到一个调用中，源和目的可以是相同的，捆绑发送接收操作虽然在语义上等同于一个发送操作和一个接收操作的结合，但是它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，这是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。

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，但是可能使用不同的标识，发送缓冲区和接收缓冲区必须分开，他们可以是不同的数据长度和不同的数据类型。

## MPI\_Sendrecv\_replace(void \*buf, int count, MPI\_Datatype datatype, int dest, int sendtag,int source,int recvtag, MPI\_Comm comm, MPI\_Status \*status)

一 个 与 MPI\_SENDRECV 类 似 的 操 作 是 MPI\_SENDRECV\_REPLACE 它 与

MPI\_SENDRECV的不同就在于它只有一个缓冲区 该缓冲区同时作为发送缓冲区和接收缓

冲区 这一调用的执行结果是发送前缓冲区中的数据被传递给指定的目的进程 该缓冲区被

从指定进程接收到的相应类型的数据所取代 因此从功能上说 这两者没有什么区别 只是

MPI\_SENDRECV\_REPLACE相对于MPI\_SENDRECV节省了一个接收缓冲区 和发送缓冲

区公用

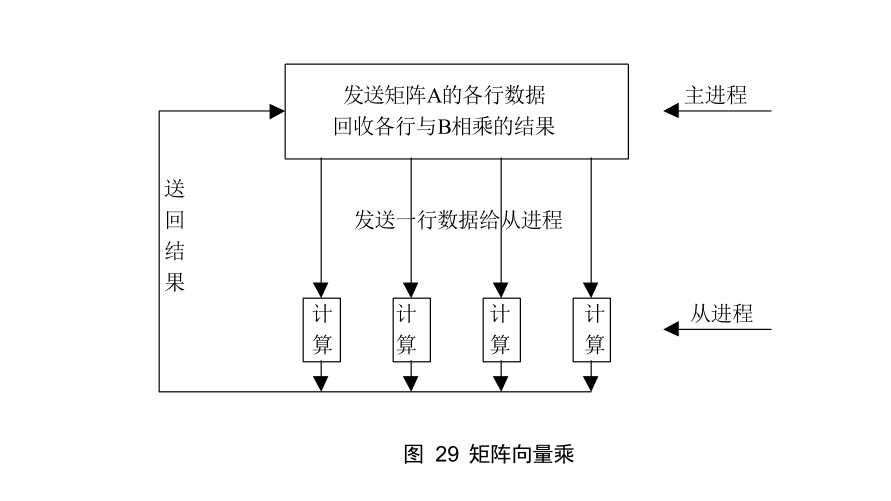
# 主从模式的 MPI 程序设计

主进程负责数据的I/O和广播（Bcast）、收集（Gather）或者归约（Reduce），一般这些已经满足常见要求。

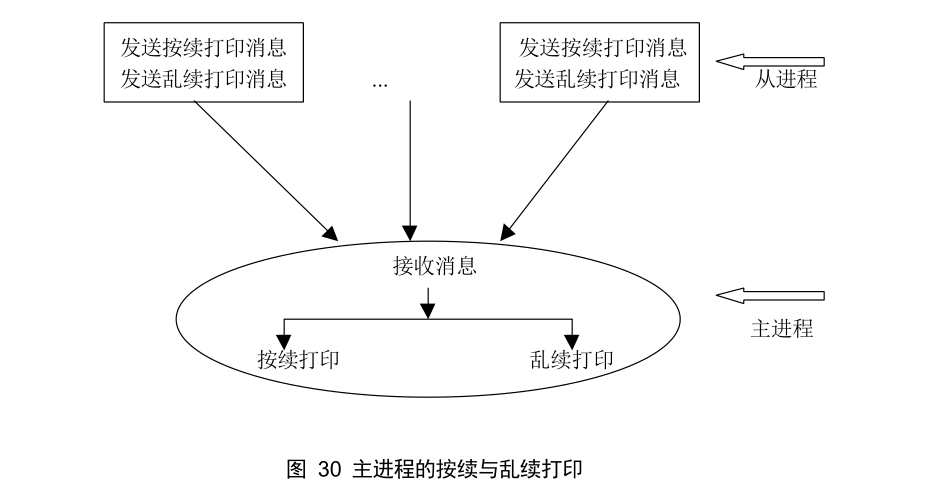
## 矩阵向量乘

D:\point\_to\_desktop\并行&云计算\c源码\arrays\_multiply.cpp

实现C=A×B 具体的实现方法是，主进程将向量B广播给所有的从进程 然后将矩阵A的各行依次发送给从进程，从进程计算一行和B相乘的结果，然后将结果发送给主进程，主进程循环向各个从进程发送一行的数据，直到将A各行的数据发送完毕一旦主进程将A的各行发送完毕 则每收到一个结果 就向相应的从进程发送结束标志 从进程接收到结束标志后退出执行，主进程收集完所有的结果后也结束。



## 主进程打印各从进程的消息



D:\point\_to\_desktop\并行&云计算\c源码\swapMsg.cpp