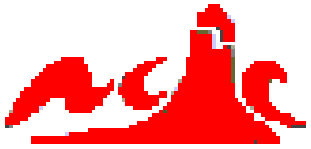


机群应用开发

并行编程原理及程序设计

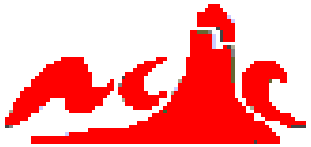
Parallel Programming: Fundamentals and Implementation

曹振南

czn@ncic.ac.cn

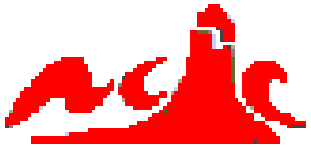
caozn@dawning.com.cn

2003.1



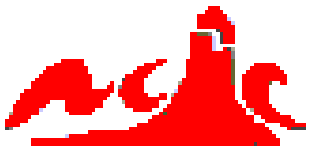
参考文献

- 黄铠,徐志伟著,陆鑫达等译. *可扩展并行计算技术,结构与编程*. 北京:机械工业出版社, P.33~56,P.227~237, 2000.
- 陈国良著.*并行计算—结构、算法、编程*. 北京:高等教育出版社,1999.
- Barry Wilkinson and Michael Allen. ***Parallel Programming(Techniques and Applications using Networked Workstations and Parallel Computers)***. Prentice Hall, 1999.
- 李晓梅,莫则尧等著. *可扩展并行算法的设计与分析*. 北京:国防工业出版社,2000.
- 张宝琳,谷同祥等著. *数值并行计算原理与方法*. 北京:国防工业出版社,1999.
- 都志辉著. *高性能计算并行编程技术—MPI并行程序设计*. 北京:清华大学出版社, 2001.



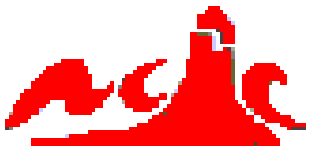
相关网址

- MPI: http://ww.mpi-forum.org，
  http://www.mcs.anl.gov/mpi
- Pthreads: http://www.oreilly.com
- PVM: http://www.epm.ornl.gov/pvm/
- OpemMP: http://www.openmp.org
- 网上搜索:www.google.com



MPI并行程序设计

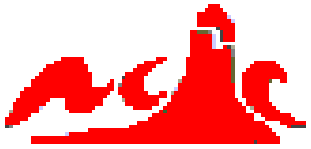
Parallel Programming with the Massage Passing Interface (MPI)



并行编程标准

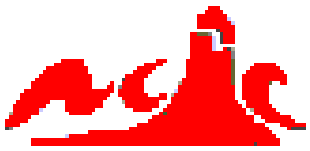
- 多线程库标准
  - – Win32 API.
  - – **POSIX** threads.
- 编译制导标准
  - – OpenMP – 可移植共享存储并行编程标准.
- 消息传递库标准
  - – MPI
  - – PVM

本讨论的重点



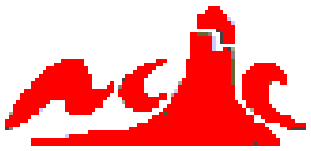
消息传递并行程序设计

- 消息传递并行程序设计
  - 指用户必须通过显式地发送和接收消息来实现处理机间的数据交换。
  - 在这种并行编程中，每个并行进程均有自己独立的地址空间，相互之间访问不能直接进行，必须通过显式的消息传递来实现。
  - 这种编程方式是大规模并行处理机（MPP）和机群（Cluster）采用的主要编程方式。
- 并行计算粒度大，特别适合于大规模可扩展并行算法
  - 由于消息传递程序设计要求用户很好地分解问题,组织不同进程间的数据交换,并行计算粒度大,特别适合于大规模可扩展并行算法.
- 消息传递是当前并行计算领域的一个非常重要的并行程序设计方式



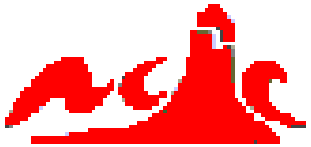
什么是MPI?

- **M**assage **P**assing **I**nterface:是消息传递函数库的标准规范，由MPI论坛开发，支持Fortran和C
  - 一种新的库描述，不是一种语言。共有上百个函数调用接口，在Fortran和C语言中可以直接对这些函数进行调用
  - MPI是一种标准或规范的代表，而不是特指某一个对它的具体实现
  - 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



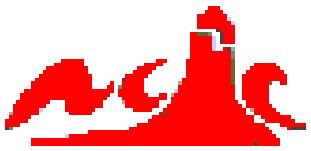
MPI的发展过程

- 发展的两个阶段
  - MPI 1.1: 1995
    - MPICH:是MPI最流行的非专利实现,由Argonne国家实验室和密西西比州立大学联合开发,具有更好的可移植性.
  - MPI 1.2~2.0:动态进程, 并行 I/O, 支持F90和C++(1997).



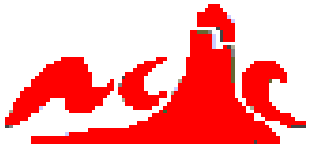
为什么要用MPI?

- 高可移植性
  - MPI已在IBM PC机上、MS Windows上、所有主要的Unix工作站上和所有主流的并行机上得到实现。使用MPI作消息传递的C或Fortran并行程序可不加改变地运行在IBM PC、MS Windows、Unix工作站、以及各种并行机上。



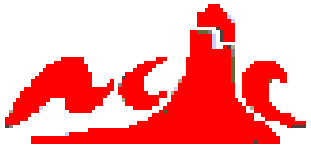
讲座内容提示

- 基本的MPI
  - 基本概念
  - 点到点通信(Point to point)
    - MPI中API的主要内容，为MPI最基本，最重要的内容
  - MPI程序的编译和运行
- 深入MPI
  - 用户自定义(/派生)数据类型(User-defined(Derived) data type)
    - 事实上MPI的所有数据类型均为MPI自定义类型
      - 支持异构系统
      - 允许消息来自不连续的或类型不一致的存储区(结构,数组散元)
  - 集合通信(Collective)
    - 数据移动，数据聚集，同步
    - 基于point to point 构建
  - MPI环境管理函数
    - 组,上下文和通信空间/通信子的管理
- 实例



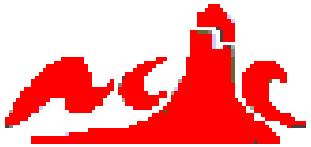
从简单入手!

- 下面我们首先分别以C语言和Fortran语言的形式给出一个最简单的MPI并行程序Hello (下页).
- 该程序在终端打印出Hello World!字样.
- “Hello World”:一声来自新生儿的问候.



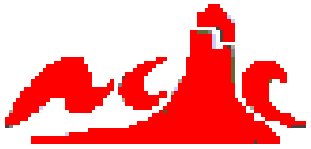
Hello world(C)

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

main(
    int argc,
    char *argv[] )
{
     MPI_Init( &argc, &argv );
     printf( "Hello, world!\n" );
     MPI_Finalize();
}
```



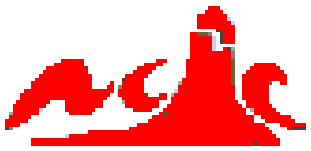
Hello world(Fortran)

```
program main
include ‘mpif.h’
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```



C和Fortran中MPI函数约定

- C
  - 必须包含mpi.h.
  - MPI 函数返回出错代码或 **MPI_SUCCESS成功标志.**
  - MPI-前缀，且只有MPI以及MPI_标志后的第一个字母大写，其余小写.
- Fortran
  - 必须包含mpif.h.
  - 通过子函数形式调用MPI，函数最后一个参数为返回值.
  - MPI-前缀，且函数名全部为大写.
- MPI函数的参数被标志为以下三种类型：
  - IN：参数在例程的调用中不会被修正.
  - OUT：参数在例程的调用中可能会被修正.
  - INOUT：参数在一些例程中为IN，而在另一些例程中为OUT.

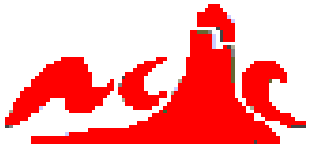


MPI初始化-MPI_INIT

int MPI_Init(int *argc, char **argv)

MPI_INIT(IERROR)

- MPI_INIT是MPI程序的第一个调用，它完成MPI程序的所有初始化工作。所有的MPI程序的第一条可执行语句都是这条语句。
- 启动MPI环境,标志并行代码的开始.
- 并行代码之前,第一个mpi函数(除MPI_Initialize()外).
- 要求main必须带参数运行,否则出错.

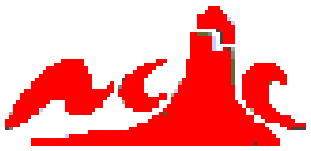


MPI结束-MPI_FINALIZE

int MPI_Finalize(void)

MPI_FINALIZE(IERROR)

- 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。
- 标志并行代码的结束,结束除主进程外其它进程.
- 之后串行代码仍可在主进程(rank = 0)上运行(如果必须).

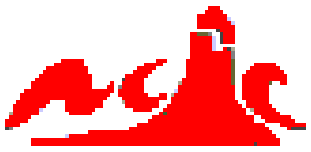


MPI程序的的编译与运行

- mpif77 hello.f 或 mpicc hello.c
  - 默认生成a.out的可执行代码.
- mpif77 –o hello hello.f 或
- mpicc –o hello hello.c
  - 生成hello的可执行代码.
- mpirun –np 4 a.out
- mpirun –np 4 hello
  - 4 指定np的实参,表示进程数,由用户指定.
  - a.out / hello 要运行的MPI并行程序.


小写o

np:
The number of process.



:运行我们的MPI程序!

⌘ server0{czn}17: mpicc -o hello hello.c

⌘ server0{czn}18: ./hello (×)

[0] Aborting program ! Could not create p4 procgroup.
Possible missing fileor program started without mpirun.

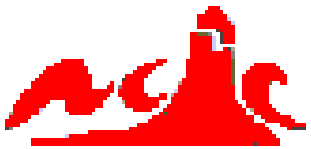
⌘ server0{czn}19: mpirun -np 4 hello (✓)

Hello World!
Hello World!
Hello World!
Hello World!

⌘ server0{czn}20:

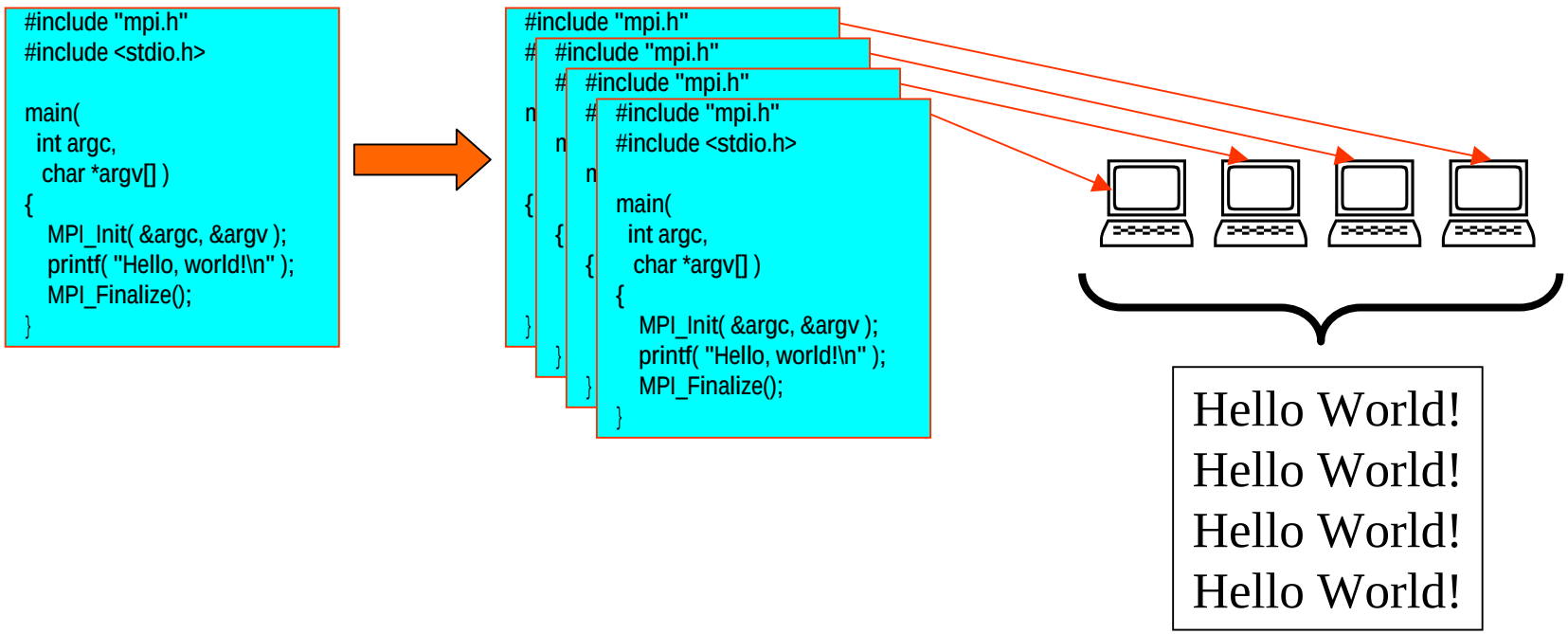
计算机打印字符

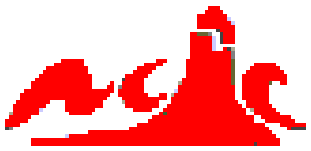
我们输入的命令



:Hello是如何被执行的?

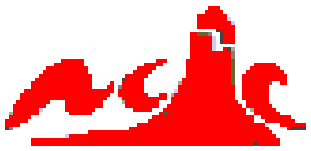
- SPMD: Single Program Multiple Data(MIMD)





:开始写MPI并行程序

- 在写MPI程序时，我们常需要知道以下两个问题的答案：
  - 任务由多少个进程来进行并行计算？
  - 我是哪一个进程?



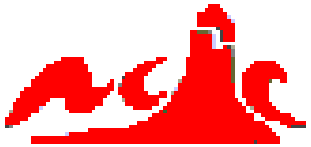
:开始写MPI并行程序

- MPI 提供了下列函数来回答这些问题：
  - 用**`MPI_Comm_size`** 获得进程个数 p

    int MPI_Comm_size(MPI_Comm comm, int *size);

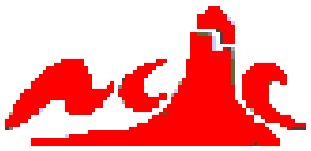
  - 用**`MPI_Comm_rank`** 获得进程的一个叫*rank*的值，该*rank*值为0到p-1间的整数,相当于进程的ID

    int MPI_Comm_rank(MPI_Comm comm, int *rank);



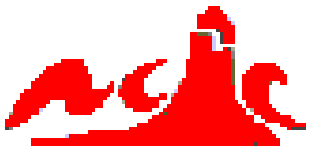
更新的Hello World(c)

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

main( int argc, char *argv[] )
{
    int  myid, numprocs;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myid );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    printf("I am %d of %d\n", myid, numprocs );
    MPI_Finalize();
}
```



更新的Hello World(Fortran

```
program main
include ‘mpif.h’
integer ierr, myid, numprocs

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
print *, ‘I am', myid, ‘of', numprocs
call MPI_FINALIZE( ierr )
end
```



👓:运行结果

⌘server0{czn}22: mpicc –o hello1 hello1.c

⌘server0{czn}23: mpirun -np 4 hello1

I am 0 of 4

I am 1 of 4

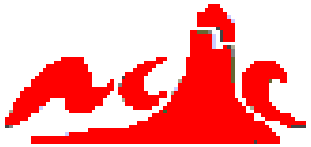
I am 2 of 4

I am 3 of 4

⌘server0{czn}24:

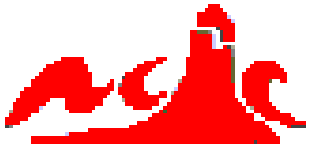
计算机打印字符

我们输入的命令



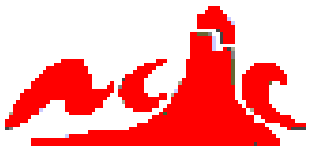
有消息传递greetings(c)

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[])
{
        int numprocs, myid, source;
        MPI_Status status;
        char message[100];

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

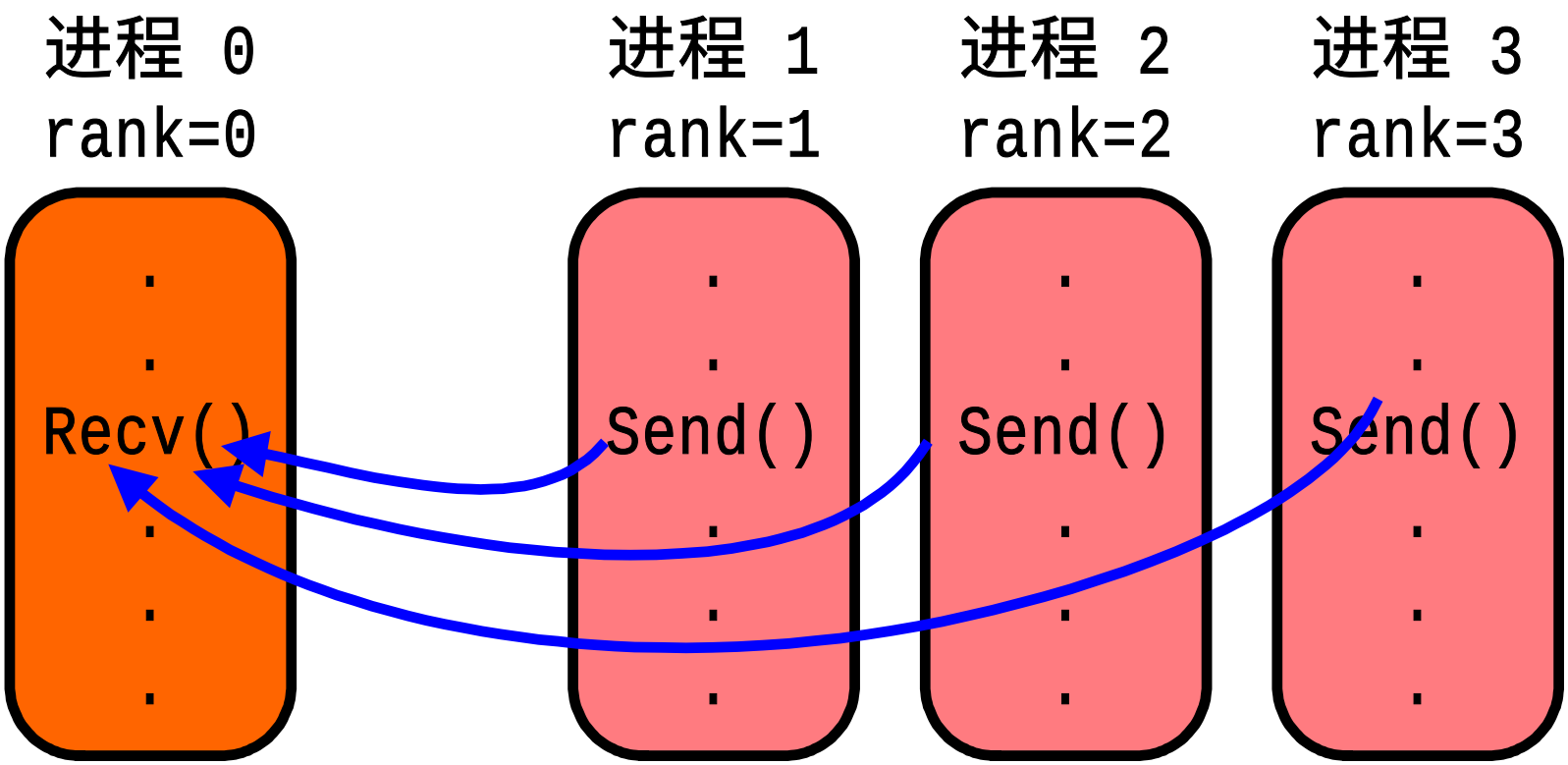


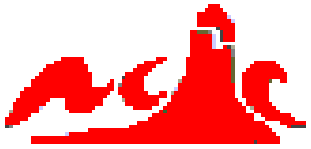
有消息传递greetings(c)

```
    if (myid != 0)  {
        strcpy(message, "Hello World!");
        MPI_Send(message,strlen(message)+1, MPI_CHAR, 0,99,
    MPI_COMM_WORLD);
    } else {/* myid == 0 */
        for (source = 1; source < numprocs; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, 99,
    MPI_COMM_WORLD,  &status);
            printf("%s\n", message);
        }
    }
    MPI_Finalize();
} /* end main */
```



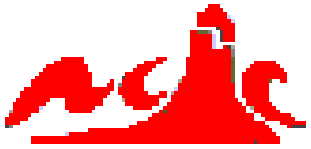
Greeting执行过程





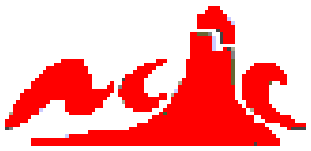
解剖greetings程序

- 头文件: mpi.h/mpif.h.
- `int MPI_Init(int *argc, char ***argv)`
  - 启动MPI环境,标志并行代码的开始.
  - 并行代码之前,第一个mpi函数(除MPI_Initialize()外).
  - 要求main必须带能运行,否则出错.
- 通信子(通信空间): MPI_COMM_WORLD：
  - 一个通信空间是一个进程组和一个上下文的组合.上下文可看作为组的超级标签,用于区分不同的通信子.
  - 在执行函数MPI_Init之后,一个MPI程序的所有进程形成一个缺省的组,这个组的通信子即被写作MPI_COMM_WORLD.
  - 该参数是MPI通信操作函数中必不可少的参数,用于限定参加通信的进程的范围.



解剖greetings程序

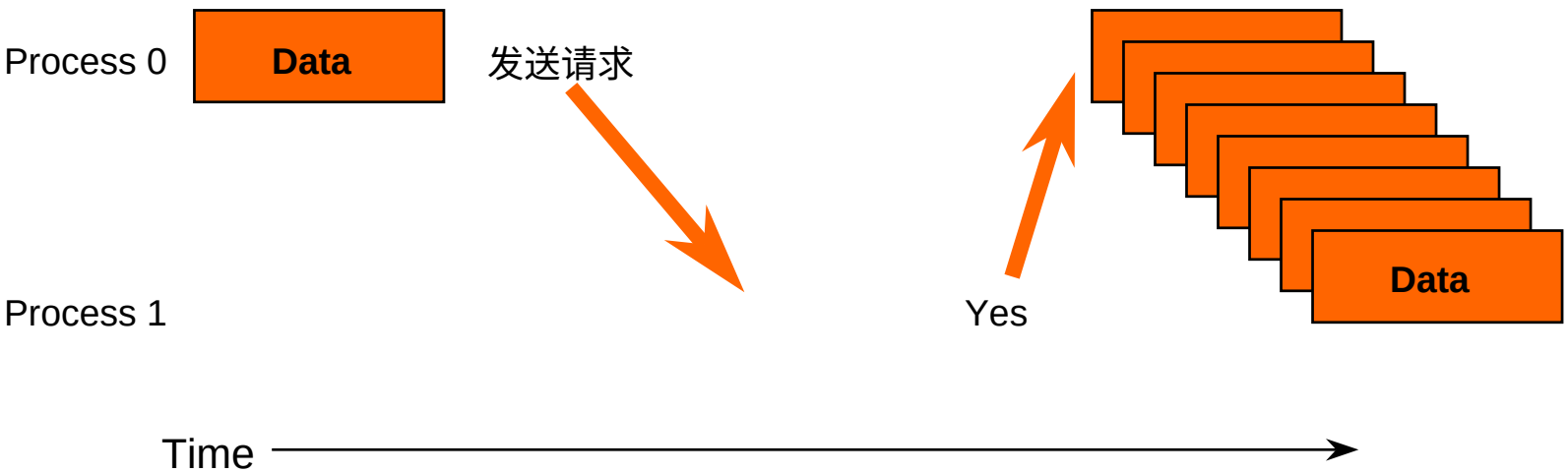
- `int MPI_Comm_size ( MPI_Comm comm, int *size )`
  - 获得通信空间comm中规定的组包含的进程的数量.
  - 指定一个communicator,也指定了一组共享该空间的进程, 这些进程组成该communicator的group.
- `int MPI_Comm_rank ( MPI_Comm comm, int *rank )`
  - 得到本进程在通信空间中的rank值,即在组中的逻辑编号(从0开始).
- `int MPI_Finalize()`
  - 标志并行代码的结束,结束除主进程外其它进程.
  - 之后串行代码仍可在主进程(rank = 0)上运行(如果必须).



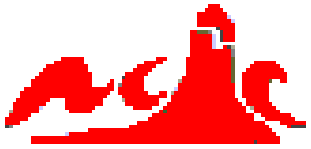
消息传送(先可不关心参数含义)

```
MPI_Send(A, 10, MPI_DOUBLE, 1,99, MPI_COMM_WORLD);

MPI_Recv(B, 20, MPI_DOBULE, 0, 99, MPI_COMM_WORLD,  &status);
```

数据传送 + 同步操作



- 需要发送方与接收方合作完成.

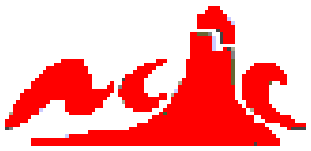


最基本的MPI

MPI调用借口的总数虽然庞大，但根据实际编写MPI的经验，常用的MPI调用的个数确什么有限。下面是6个最基本的MPI函数。

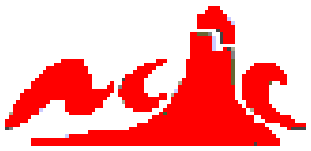
1. MPI_Init(...);
2. MPI_Comm_size(...);
3. MPI_Comm_rank(...);
4. MPI_Send(...);
5. MPI_Recv(...);
6. MPI_Finalize();

MPI_Init(…);
…
并行代码;
…
MPI_Fainalize();
只能有串行代码;



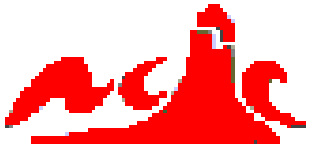
讲座内容提示

- 基本的MPI
  - 基本概念
  - 点到点通信(Point to point)
    - MPI中API的主要内容，为MPI最基本，最重要的内容
  - MPI程序的编译和运行
- 深入MPI
  - 用户自定义(/派生)数据类型(User-defined(Derived) data type)
    - 事实上MPI的所有数据类型均为MPI自定义类型
      - 支持异构系统
      - 允许消息来自不连续的或类型不一致的存储区(结构,数组散元)
  - 集合通信(Collective)
    - 数据移动，数据聚集，同步
    - 基于point to point 构建
  - MPI环境管理函数
    - 组,上下文和通信空间/通信子的管理
- 实例



Point to Point

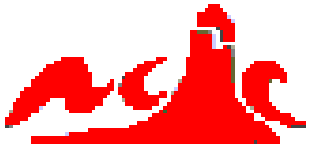
- 单个进程对单个进程的通信,重要且复杂
- 术语
  - Blocking(阻塞) :一个例程须等待操作完成才返回,返回后用户可以重新使用调用中所占用的资源.
  - Non-blocking(非阻塞):一个例程不必等待操作完成便可返回,但这并不意味着所占用的资源可被重用.
  - Local(本地):不通信.
  - Non-local(非本地):通信.



Blocking Send

int 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
IN	**count**	要发送信息的元素个数
IN	**datatype**	发送信息的数据类型
IN	**dest**	目标进程的rank值
IN	**tag**	消息标签
IN	**comm**	通信子



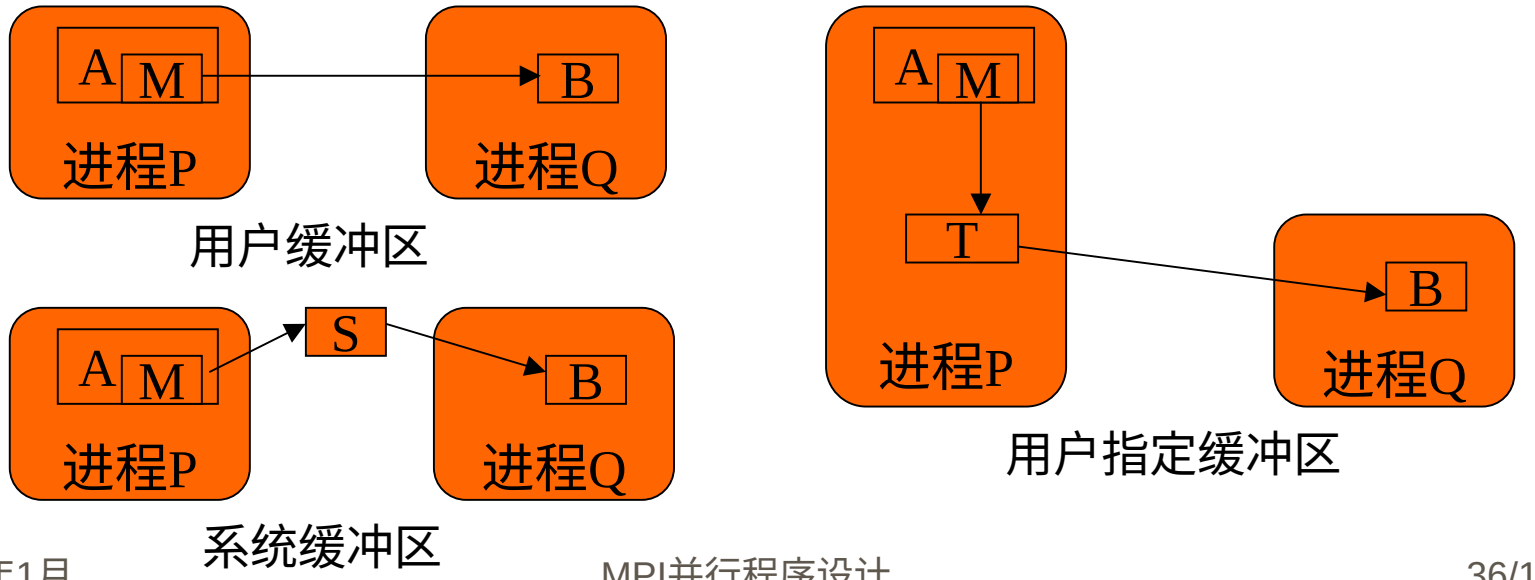
Blocking Receive

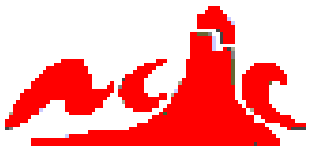
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);

OUT buf	发送缓冲区的起始地址
IN count	要发送信息的元素个数
IN datatype	发送信息的数据类型
IN dest	目标进程的rank值
IN tag	消息标签
IN comm	通信子
OUT status	status对象,包含实际接收到的消息的有关信息

什么是缓冲区？

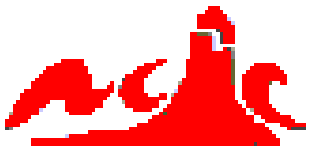
1. 应用程序中说明的变量,在消息传递语句中又用作缓冲区的起始位置.
2. 也可表示由系统(不同用户)创建和管理的某一存储区域,在消息传递过程中用于暂存放消息.也被称为系统缓冲区.
3. 用户可设置一定大小的存储区域,用作中间缓冲区以保留可能出现在其应用程序中的任意消息.





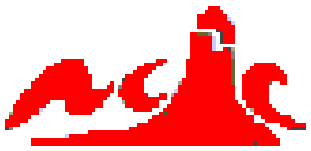
消息标识(Message Envelope):信封 消息buffer:内容

- **MPI标识一条消息的信息包含四个域:**
  - Source: 发送进程隐式确定,由进程的rank值唯一标识
  - Destination: Send函数参数确定
  - Tag: Send函数参数确定, (0,UB),UB:MPI_TAG_UB>=32767.
  - Communicator: 缺省MPI_COMM_WORLD
    - Group:有限/N，有序/Rank∈ [0,1,2,...N-1]
    - Contex:Super_tag,用于标识该通讯空间.
- **数据类型**
  - 异构计算:数据转换.
  - 派生数据类型:结构或数组散元传送.



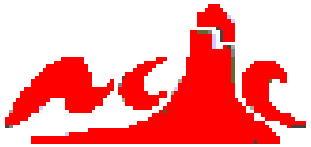
消息匹配

- 接收buffer必须至少可以容纳count个由datatype参数指明类型的数据. 如果接收buf太小, 将导致溢出、出错.
- 消息匹配
  - 参数匹配 dest,tag,comm/ source,tag,comm
  - Source == MPI_ANY_SOURCE：接收任意处理器来的数据(任意消息来源).
  - Tag == MPI_ANY_TAG：匹配任意tag值的消息(任意tag消息).
- 在阻塞式消息传送中不允许Source==Dest,否则会导致deadlock.
- 消息传送被限制在同一个communicator.
- 在send函数中必须指定唯一的接收者(Push/pull通讯机制).



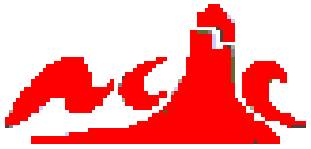
status参数

- 当使用**MPI_ANY_SOURCE**或/和**MPI_ANY_TAG**接收消息时如何确定消息的来源**source** 和 **tag**值呢？
  - 在C中,status.MPI_SOURCE, status.MPI_TAG.
  - 在Fortran中, source=status(MPI_SOURCE), tag=status(MPI_TAG).
- **Status**还可用于返回实际接收到消息的长度
  - int MPI_Get_count(MPI_Status status, MPI_Datatype datatype,int* count)
    - IN status 接收操作的返回值.
    - IN datatype 接收缓冲区中元素的数据类型.
    - OUT count 接收消息中的元素个数.



分析greetings

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

main(int argc, char* argv[])
{
    int numprocs;          /*进程数,该变量为各处理器中的同名变量, 存储是分布的  */
    int myid;              /*我的进程ID,存储也是分布的                          */
    MPI_Status status;     /*消息接收状态变量,存储也是分布的                    */
    char message[100];     /*消息buffer,存储也是分布的                          */

    /*初始化MPI*/
    MPI_Init(&argc, &argv);
    /*该函数被各进程各调用一次,得到自己的进程rank值*/
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    /*该函数被各进程各调用一次,得到进程数*/
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

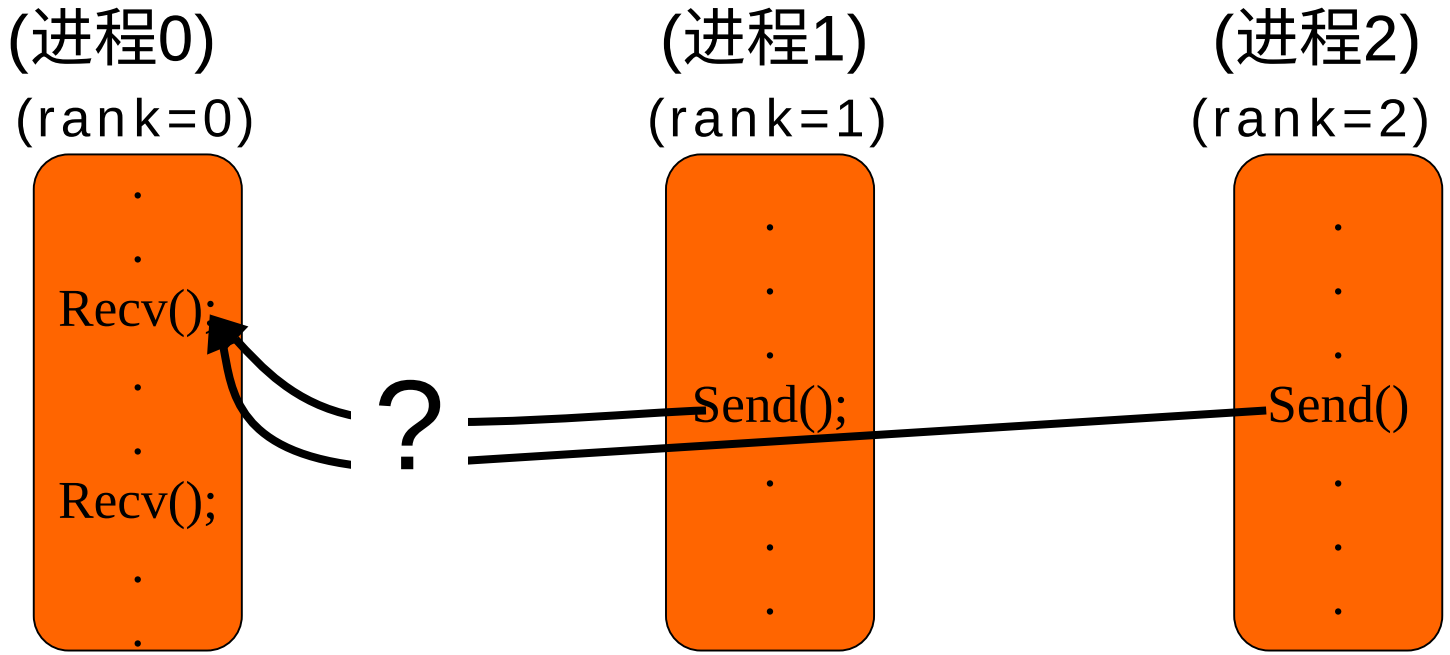


分析greetings

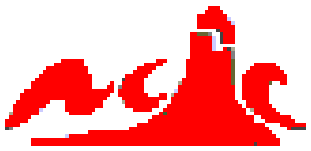
```
if (myid != 0)
{
    /*建立消息*/
    sprintf(message, "Greetings from process %d!",myid);
    /* 发送长度取strlen(message)+1,使\0也一同发送出去*/
    MPI_Send(message,strlen(message)+1, MPI_CHAR, 0,99,MPI_COMM_WORLD);
}
else
{ /* my_rank == 0 */
     for (source = 1; source < numprocs; source++)
     {
       MPI_Recv(message, 100, MPI_CHAR, source, 99, MPI_COMM_WORLD,&status);
       printf("%s\n", message);
     }
}
    /*关闭MPI,标志并行代码段的结束*/
     MPI_Finalize();
} /* End main */
```

Greetings执行过程

假设进程数为3



问题:进程1和2谁先向根进程发送消息?



运行greetings

- server0{czn}28: mpicc –o greeting greeting.c
- server0{czn}29: mpirun -np 4 greeting

  Greetings from process 1!

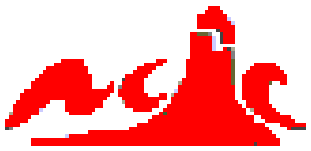
  Greetings from process 2!

  Greetings from process 3!
- server0{czn}30:

计算机打印字符

我们输入的命令

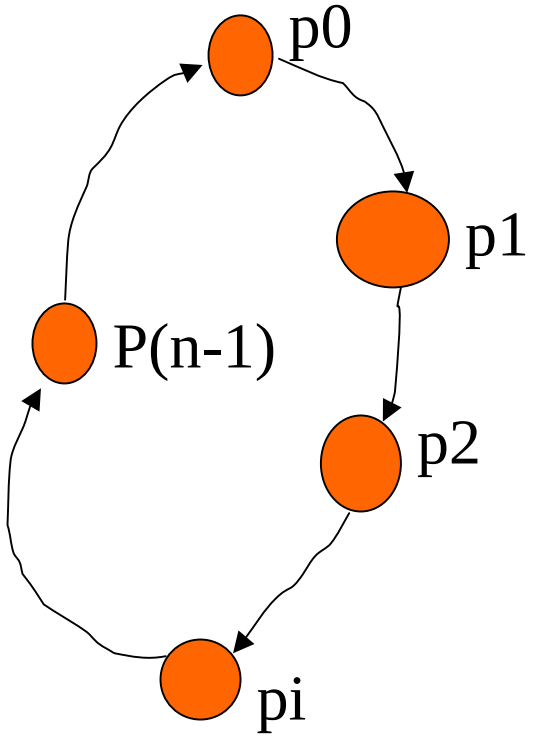
现在您已经能够用MPI进行并行编程了!

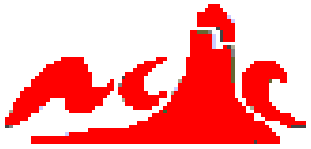


MPI_Sendrecv函数原型

```
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)
```

- 数据轮换





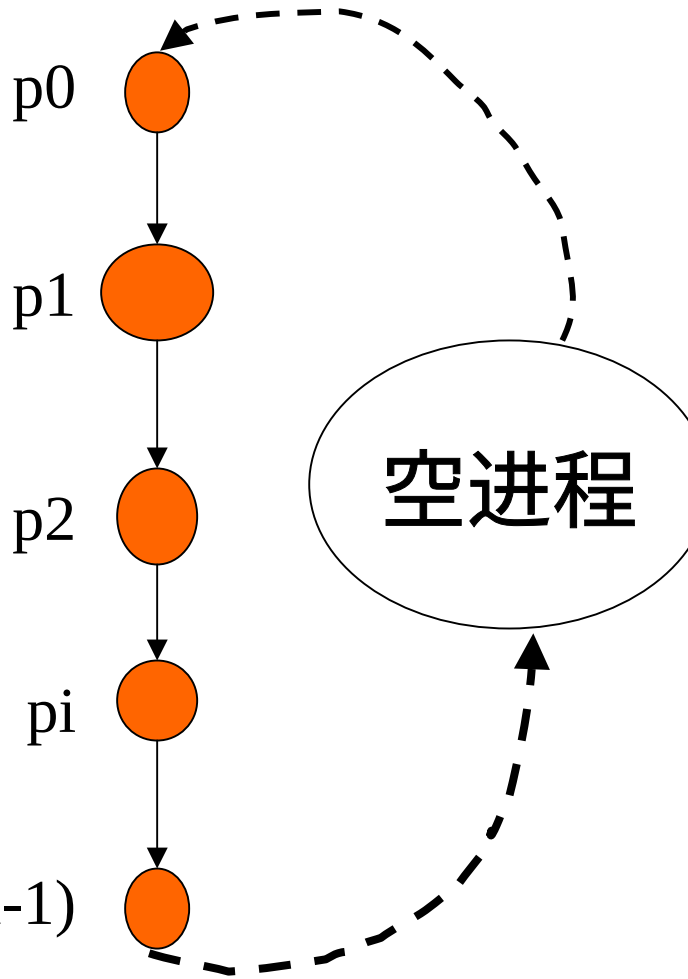
MPI_Sendrecv用法示意

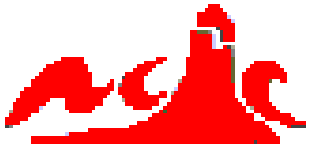
```
…
int a,b;
…
MPI_Status status;
int dest = (rank+1)%p;
int source = (rank + p -1)%p;    /*p为进程个数*/
MPI_Sendrecv( &a, 1, MPI_INT, dest, 99, &b 1,
   MPI_INT, source, 99, MPI_COMM_WORLD, &status);
```

该函数被每一进程执行一次.

空进程

- rank = MPI_PROC_NULL的进程称为空进程
  - 使用空进程的通信不做任何操作.
  - 向MPI_PROC_NULL发送的操作总是成功并立即返回.
  - 从MPI_PROC_NULL接收的操作总是成功并立即返回，且接收缓冲区内容为随机数.
  - status
    - status.MPI_SOURCE = MPI_PROC_NULL
    - status.MPI_TAG = MPI_ANY_TAG
    - MPI_Get_count(&status,MPI_Datatype datatype, &count) =>count = 0



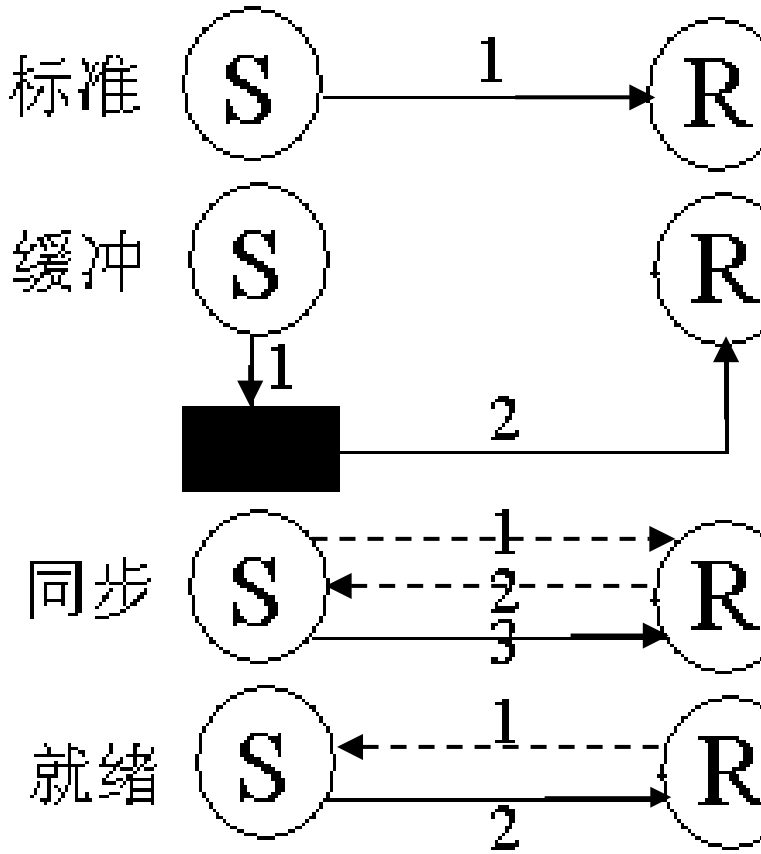


空进程应用示意

```
…
MPI_Status status;
int dest = (rank+1)%p;
int source = (rank + p -1)%p;
if(source == p-1) source = MPI_PROC_NULL;
if(dest == 0) dest = MPI_PROC_NULL;

MPI_Sendrecv( &a, 1, MPI_INT, dest, 99, &b 1,
   MPI_INT, source, 99, MPI_COMM_WORLD, &status);
…
```

阻塞通信模式

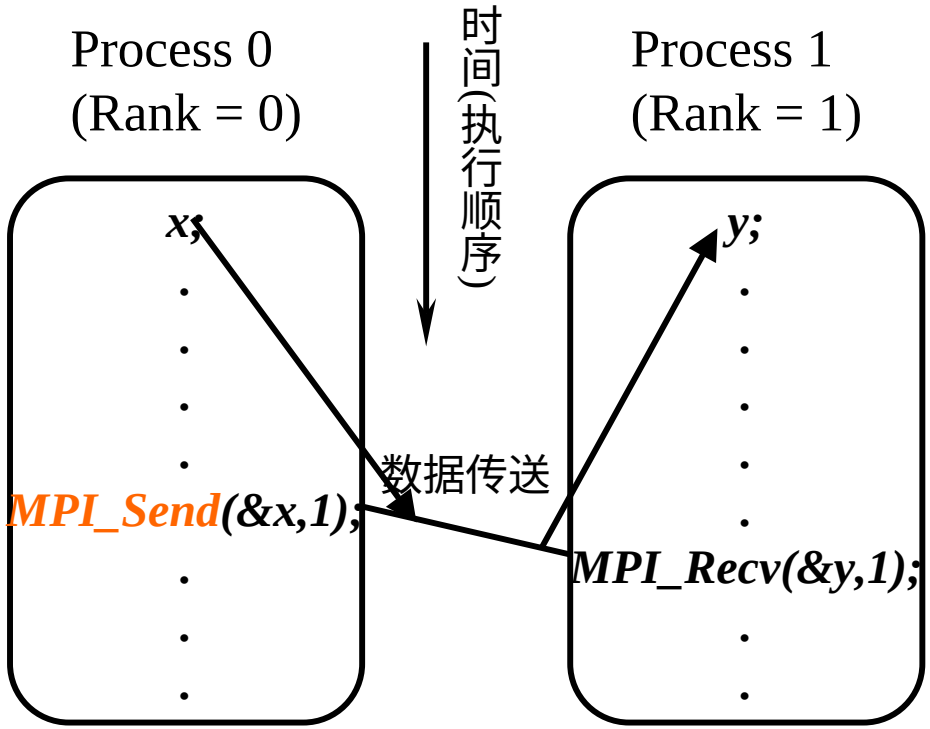
- 由发送方体现(send语句).
- 阻塞通信中接收语句相同，MPI_Recv
- 按发送方式的不同，消息或直接被copy到接收者的buffer中或被拷贝到系统buffer中。
  - 标准模式Standard
    - 最常用的发送方式 **MPI_Send(...)**
  - B:缓冲模式Buffer
    - 发到系统缓冲区
    - **MPI_Bsend(...)**
  - S:同步模式Synchronous
    - 任意发出，不需系统缓冲区
    - **MPI_Ssend(...)**
  - R:就绪模式Ready
    - 就绪发出，不需系统缓冲区
    - **MPI_Rsend(...)**



标准模式Standard

--直接送信或通过邮局送信

- 由MPI决定是否缓冲消息
  - 没有足够的系统缓冲区时或出于性能的考虑，MPI可能进行直接拷贝：仅当相应的接收开始后，发送语句才能返回
  - MPI缓冲消息：发送语句地相应的接收语句完成前返回
- 发送的结束 == 消息已从发送方发出，而不是滞留在发送方的系统缓冲区中
- 非本地的：发送操作的成功与否依赖于接收操作
- 最常用的发送方式

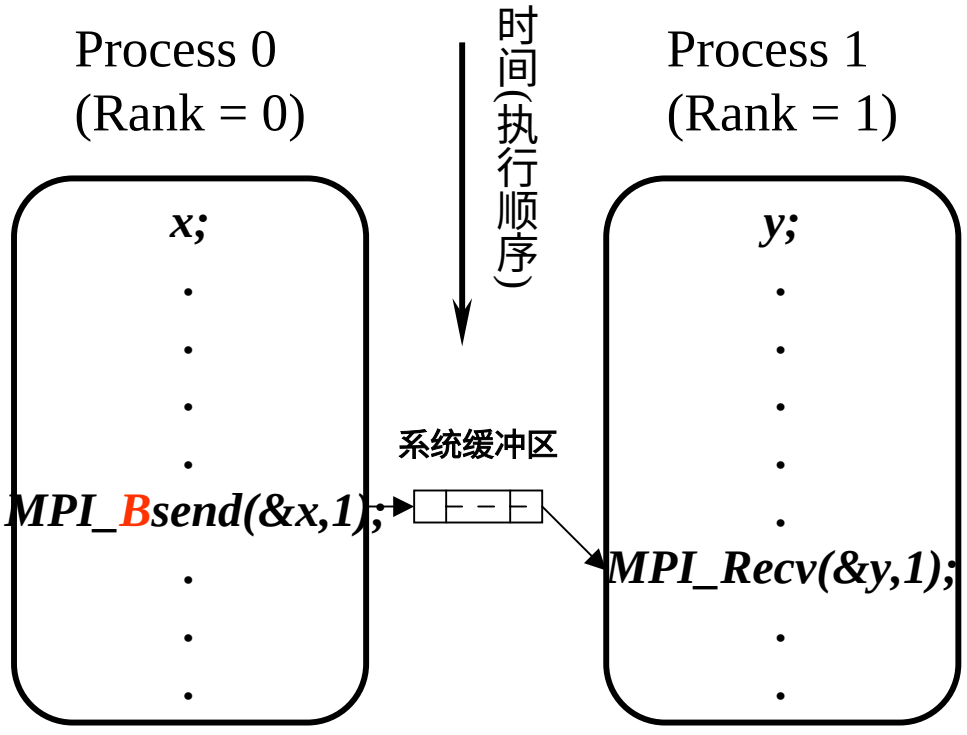


数据在发送方buffer与接收方buffer间直接拷贝

缓冲模式Buffer

--通过邮局送信(应用系统缓冲区)

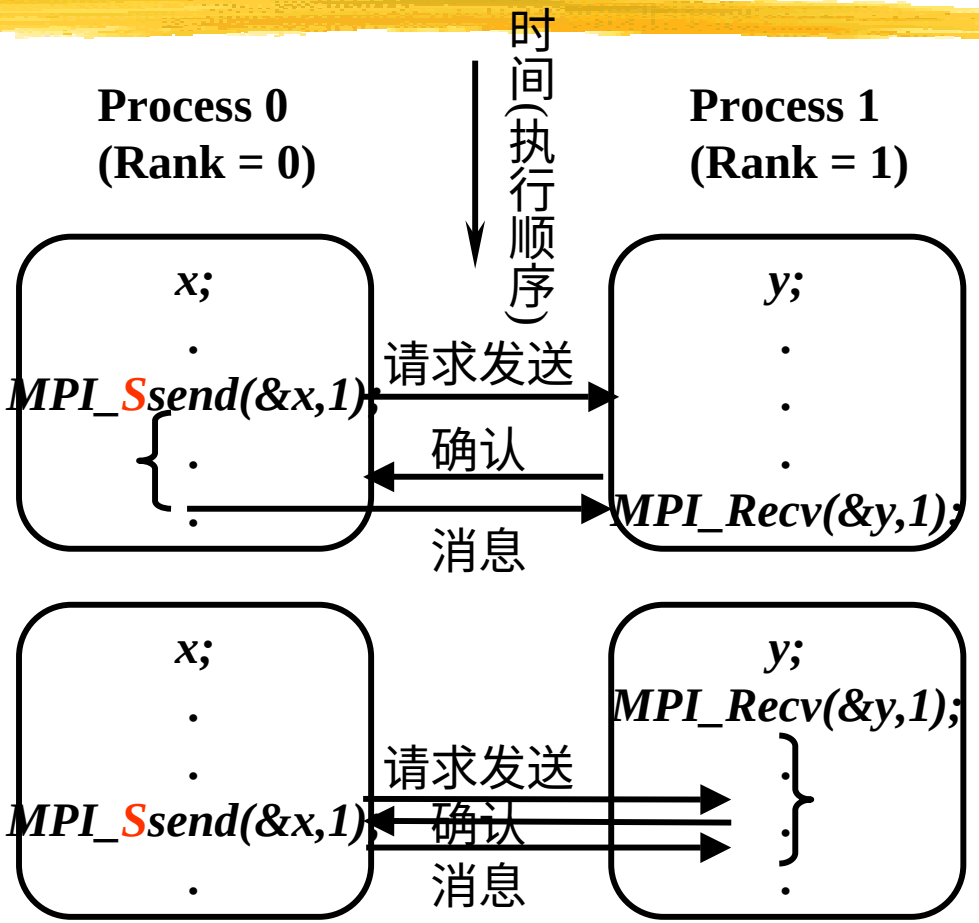
- **前提:** 用户显示地指定用于缓冲消息的系统缓冲区
  - MPI_Buffer_attach(*buffer, *size) 。
- **发送是本地的**: 完成不依赖于与其匹配的接收操作。发送的结束仅表明消息进入系统的缓冲区中，发送方缓冲区可以重用，而对接收方的情况并不知道。
- 缓冲模式在相匹配的接收未开始的情况下，总是将送出的消息放在缓冲区内，这样发送者可以很快地继续计算,然后由系统处理放在缓冲区中的消息。
- 占用内存，一次内存拷贝。
- 其函数调用形式为：MPI_BSEND(...)。B代表缓冲.



通过系统缓冲区传送消息

同步模式Synchronous
--握手后才送出名片(遵从three-way协议)

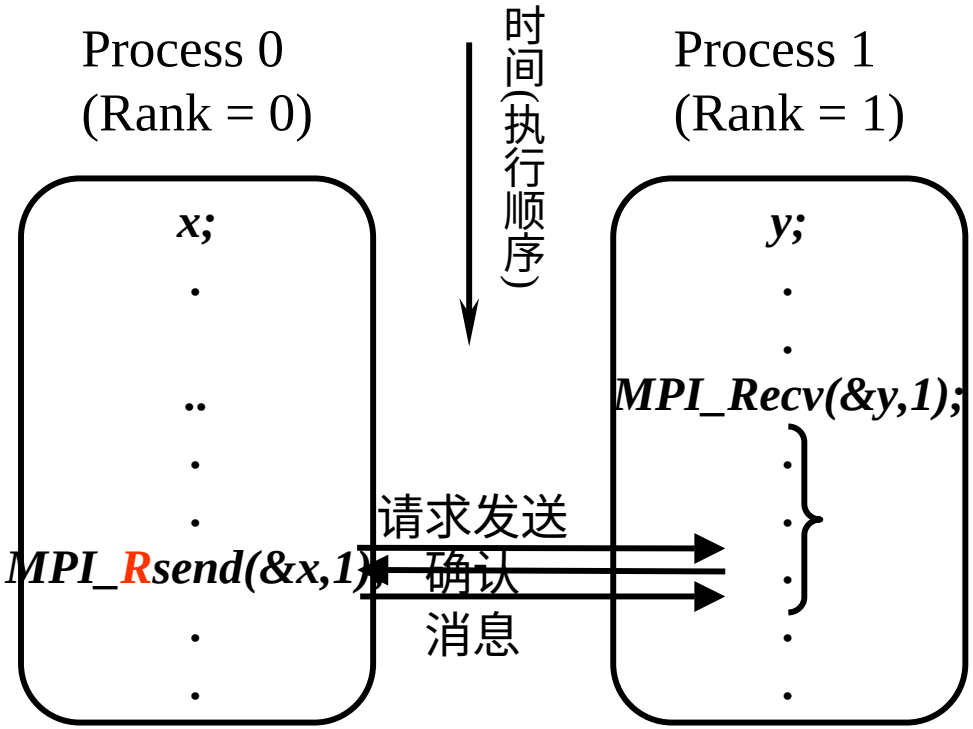
- 本质特征：收方接收该消息的缓冲区已准备好，不需要附加的系统缓冲区
- 任意发出：发送请求可以不依赖于收方的匹配的接收请求而任意发出
- 成功结束：仅当收方已发出接收该消息的请求后才成功返回，否则将阻塞。意味着：
  - 发送方缓冲区可以重用
  - 收方已发出接收请求
- **是非本地的**
- 其函数调用形式为：MPI_SSEND(...)。S代表缓冲



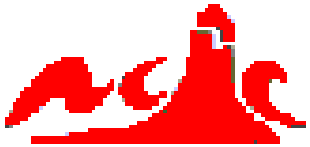
同步发送与接收
上图：发送起前于接收
下图：发送滞后于接收

就绪模式Ready
--有客户请求,才提供服务

- 发送请求仅当有匹配的接收后才能发出，否则出错。在就绪模式下，系统默认与其相匹配的接收已经调用。接收必须先于发送。
- 它不可以不依赖于接收方的匹配的接收请求而任意发出
- 其函数调用形式为：MPI_RSEND(...)。R代表缓冲

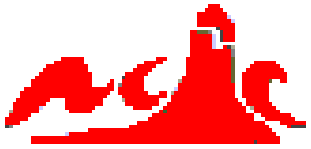


接收必须先于发送
(只有客户发出服务请求,才提供服务)



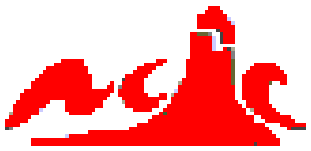
阻塞与非阻塞的差别

- 用户发送缓冲区的重用:
  - 非阻塞的发送：仅当调用了有关结束该发送的语句后才能重用发送缓冲区，否则将导致错误；对于接收方，与此相同，仅当确认该接收请求已完成后才能使用。所以对于非阻塞操作，要先调用等待MPI_Wait()或测试MPI_Test()函数来结束或判断该请求，然后再向缓冲区中写入新内容或读取新内容。
- 阻塞发送将发生阻塞,直到通讯完成.
- 非阻塞可将通讯交由后台处理，通信与计算可重叠.
- 发送语句的前缀由MPI_改为MPI_I, I:immediate:
  - 标准模式:MPI_Send(...)->MPI_Isend(...)
  - Buffer模式:MPI_Bsend(...)->MPI_Ibsend(...)
  - ...



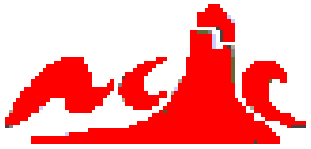
非阻塞发送与接收

- int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  - IN buf 发送缓冲区的起始地址
  - IN count 发送缓冲区的大小(发送元素个数)
  - IN datatype 发送缓冲区数据的数据类型
  - IN dest 目的进程的秩
  - IN tag 消息标签
  - IN comm 通信空间/通信子
  - OUT request 非阻塞通信完成对象(句柄)
- MPI_Ibsend/MPI_Issend/MPI_Irsend:非阻塞缓冲模式/非阻塞同步模式/非阻塞就绪模式
- int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request* request)



通信的完成(常用于非阻塞通信)

- 发送的完成: 代表发送缓冲区中的数据已送出，发送缓冲区可以重用。它并不代表数据已被接收方接收。数据有可能被缓冲;
  - 同步模式：发送完成==接收方已初始化接收，数据将被接收方接收;
- 接收的完成:代表数据已经写入接收缓冲区。接收者可访问接收缓冲区，status对象已被释放。它并不代表相应的发送操作已结束。
- 通过MPI_Wait()和MPI_Test()来判断通信是否已经完成;

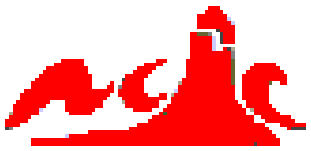


MPI_Wait()及应用示例

- int MPI_Wait(MPI_Request* request, MPI_Status * status);

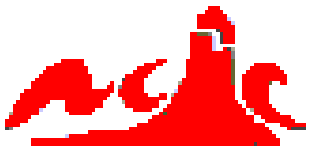
  当request标识的通信结束后，MPI_Wait()才返回。如果通信是非阻塞的，返回时request = MPI_REQUEST_NULL;函数调用是非本地的;

```
MPI_Request request;
MPI_Status status;
int x,y;
if(rank == 0){
    MPI_Isend(&x,1,MPI_INT,1,99,comm,&request)
    ...
    MPI_Wait(&request,&status);
}else{
    MPI_Irecv(&y,1,MPI_INT,0,99,comm,&request)
    ...
    MPI_Wait(&request,&status);
}
```



MPI_Test()及应用示例

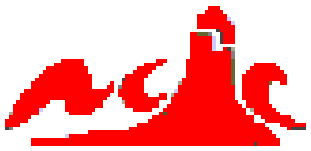
```
//int MPI_Test(MPI_Request *request,int *flag, MPI_Status *status);

MPI_Request request;
MPI_Status status;
int x,y,flag;
if(rank == 0){
    MPI_Isend(&x,1,MPI_INT,1,99,comm,&request)
    while(!flag)
        MPI_Test(&request,&flag,&status);
}else{
    MPI_Irecv(&y,1,MPI_INT,0,99,comm,&request)
    while(!flag)
        MPI_Test(&request,&flag,&status);
}
```



消息探测

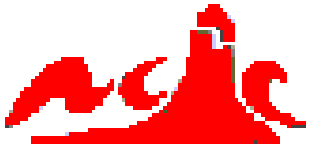
--Probe函数(适用于阻塞与非阻塞)

- MPI_Probe()和MPI_Iprobe()函数探测接收消息的内容。用户根据探测到的消息内容决定如何接收这些消息，如根据消息大小分配缓冲区等。前者为阻塞方式,即只有探测到匹配的消息才返回;后者为非阻塞,即无论探测到与否均立即返回.
- int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)
- int MPI_Iprobe(int source, int tag, MPI_Comm comm, int*flag, MPI_Status* status)
  - IN source 数据源的rank，可以是MPI_ANY_SOURCE
  - IN tag 数据标签，可以是MPI_ANY_TAG
  - IN comm 通信空间/通信子
  - OUT flag 布尔值，表示探测到与否(只用于非阻塞方式)
  - OUT status status对象，包含探测到消息的内容



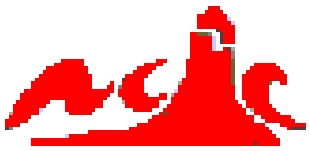
MPI_Probe应用示例

```
int x;
float y;
MPI_Comm_rank(comm, &rank);
if(rank ==0)                    /*0->2发送一int型数*/
    MPI_Send(100,1,MPI_INT,2,99,comm);
else if(rank == 1)              /*1->2发送一float型数*/
    MPI_Send(100.0,1,MPI_FLOAT,2,99,comm);
else                            /* 根进程接收 */
    for(int i=0;i<2;i++) {
        MPI_Probe(MPI_ANY_SOURCE,0,comm,&status);/*Blocking
          */
        if (status.MPI_SOURCE == 0)
            MPI_Recv(&x,1,MPI_INT,0,99,&status);
        else if(status.MPI_SOURCE == 1)
            MPI_Recv(&y,1,MPI_FLOAT,0,99,&status);
    }
```



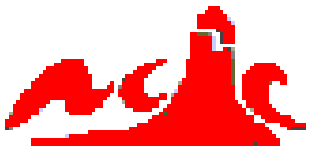
讲座内容提示

- 基本的MPI
  - 基本概念
  - 点到点通信(Point to point)
    - MPI中API的主要内容，为MPI最基本，最重要的内容
  - MPI程序的编译和运行
- 深入MPI
  - 用户自定义(/派生)数据类型(User-defined(Derived) data type)
    - 事实上MPI的所有数据类型均为MPI自定义类型
      - 支持异构系统
      - 允许消息来自不连续的或类型不一致的存储区(结构,数组散元)
  - 集合通信(Collective)
    - 数据移动，数据聚集，同步
    - 基于point to point 构建
  - MPI环境管理函数
    - 组,上下文和通信空间/通信子的管理
- 实例



MPI程序的编译

- mpicc编译并连接用C语言编写的MPI程序
- mpiCC编译并连接用C++编写的MPI程序
- mpif77编译并连接用FORTRAN 77编写的MPI程序
- mpif90编译并连接用Fortran 90编写的MPI程序
- 这些命令可以自动提供MPI需要的库，并提供特定的开关选项（用-help查看）。

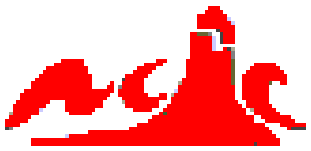


MPI程序的编译

- 用mpicc编译时，就像用一般的C编译器一样。还可以使用一般的C的编译选项，含义和原来的编译器相同

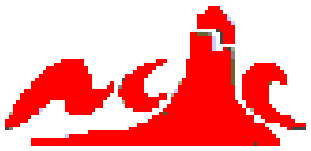
- 例如：

```
./mpicc -c foo.c
./mpicc -o foo foo.o
```



MPI程序的运行

- MPI程序的执行步骤一般为：
  - 编译以得到MPI可执行程序（若在同构的系统上，只需编译一次；若系统异构，则要在每一个异构系统上都对MPI源程序进行编译）
- 将可执行程序拷贝到各个节点机上
- 通过mpirun命令并行执行MPI程序



最简单的MPI运行命令

mpirun –np N <program>

其中：

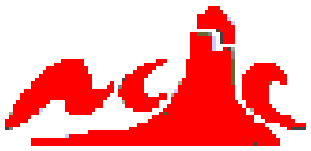
N： 同时运行的进程数

<program>： 可执行MPI程序名

例如：

mpirun –np 6 cpi

mpirun –np 4 hello



一种灵活的执行方式

mpirun –p4pg <pgfile> <program>

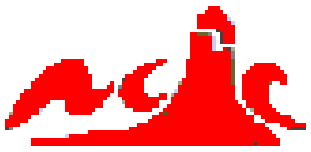
- **<pgfile>为配置文件，其格式为：**

  <机器名> <进程数> <程序名>
  <机器名> <进程数> <程序名>
  <机器名> <进程数> <程序名>

- **例如：(注：第一行的0并不表示在node0上没有进程，这里的0特指在node0上启动MPI程序)**

  node0 0 /public0/czn/mpi/cpi
  node1 1 /public0/czn/mpi/cpi
  node2 1 /public0/czn/mpi/cpi

- **这种方式允许可执行程序由不同的名字和不同的路径**



完整的MPI运行方式

- **MPI程序的一般启动方式：**

  mpirun –np <number of processor> <program name and argument>

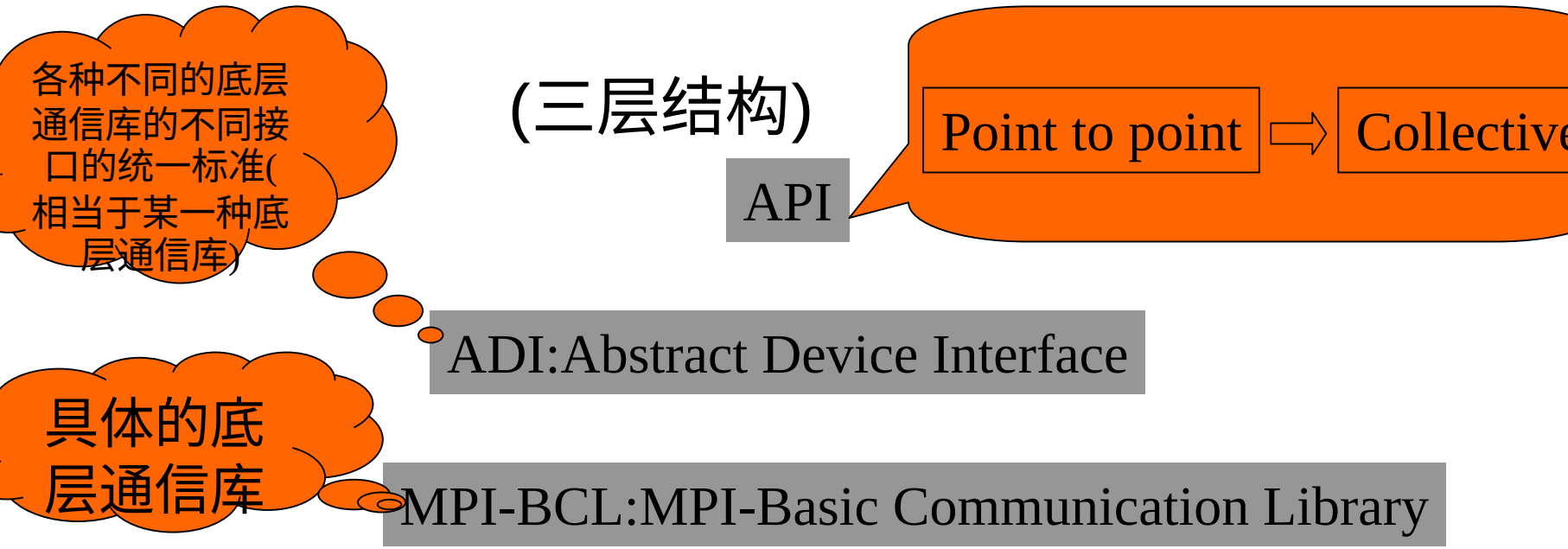
- **完整的MPI运行方式：**

  mpirun [mpirun_options] <program> [options...]

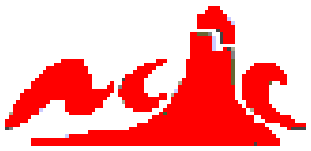
详细参数信息执行mpirun -help

曙光4000L: MPI-BCL

- MPI-BCL:为曙光4000l优化的MPI通信库，和基于TCP/IP的网络版MPICH1.2

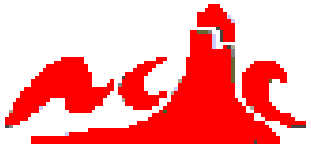


移植:根据不同的device(即平台或底层通信)实现不同的ADI接口



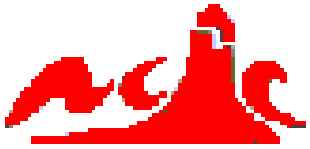
曙光4000L: run

- run -h|help|-? -sz <size>|-sz <hXw> -pn <pnode list> -np <nprocs> -pl <poolname> -cpu program <parameters>
  - -h|help|-? 得到run的帮助信息,需要帮助时用
  - -sz <size> 指定物理节点数,默认与np数相等,可忽略,
    - run –sz 4 –np 8 a.out 表示在**4**个物理节点上运行a.out的**8**个进程
  - -sz <hXw> 指定物理节点的长和宽,选定mesh结构,
    - run –sz 4X4 a.out表示在一个4*4的mesh结构上组织通信
  - -np <nprocs> 指定进程数
  - -on <pnode list> 指定物理节点列表
    - run –on node0..3 a.out 表示在节点0,1,2,3上运行a.out
  - -pl <plname> 指定节点池名p1,p2,默认为p1/(0-15)
  - -cpu 载入要执行程序的cpu模式 –ep独占通讯端口,-en独占节点
  - program <parameters> 要执行程序名以及其参数
- **常用形式：run –np 4 a.out**



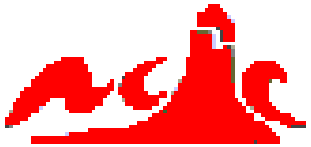
讲座内容提示

- 基本的MPI
  - 基本概念
  - 点到点通信(Point to point)
    - MPI中API的主要内容，为MPI最基本，最重要的内容
  - MPI程序的编译和运行
- 深入MPI
  - 用户自定义(/派生)数据类型(User-defined(Derived) data type)
    - 事实上MPI的所有数据类型均为MPI自定义类型
      - 支持异构系统
      - 允许消息来自不连续的或类型不一致的存储区(结构,数组散元)
  - 集合通信(Collective)
    - 数据移动，数据聚集，同步
    - 基于point to point 构建
  - MPI环境管理函数
    - 组,上下文和通信空间/通信子的管理
- 实例



MPI数据类型

```
.....
if (my_rank != 0)
{
    /*建立消息*/
    sprintf(message, "Greetings from process %d!",my_rank);
    /* 发送长度取strlen(message)+1,使\0也一同发送出去*/
    MPI_Send(message,strlen(message)+1, MPI_CHAR, 0,99,MPI_COMM_WORLD);
}
else
{ /* my_rank == 0 */
     for (source = 1; source < p; source++)
     {
       MPI_Recv(message, 100, MPI_CHAR, source, 99, MPI_COMM_WORLD,&status);
       printf("%s\n", message);
     }
}
    /*关闭MPI,标志并行代码段的结束*/
     MPI_Finalize();
} /* main */
```

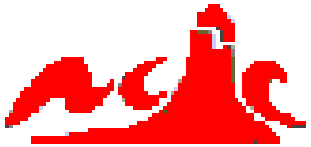


用户自定义数据类型/派生数据类型

- 目的
  - 导构计算: 不同系统有不同的数据表示格式。MPI预先定义一些基本数据类型，在实现过程中在这些基本数据类型为桥梁进行转换。
  - 派生数据类型:允许消息来自不连续的和类型不一致的存储区域，如数组散元与结构类型等的传送。
- **MPI中所有数据类型均为MPI自定义类型**
  - 基本数据类型,如MPI_INT,MPI_DOUBLE...
  - 用户定义数据类型或派生数据类型.

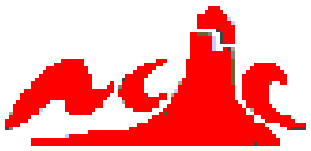
MPI基本数据类型

MPI(C Binding)	C	MPI(Fortran Binding)	Fortran
MPI_BYTE		MPI_BYTE	
MPI_CHAR	signed char	MPI_CHARACTER	CHARACTER(1)
		MPI_COMPLEX	COMPLEX
MPI_DOUBLE	double	MPI_DOUBLE_ PRECISION	DOUBLE_ PRECISION
MPI_FLOAT	float	MPI_REAL	REAL
MPI_INT	int	MPI_INTEGER	INTEGER
		MPI_LOGICAL	LOGICAL
MPI_LONG	long		
MPI_LONG_DOUBLE	long double		
MPI_PACKED		MPI_PACKED	
MPI_SHORT	short		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long		
MPI_UNSIGNED_SHORT	unsigned short		



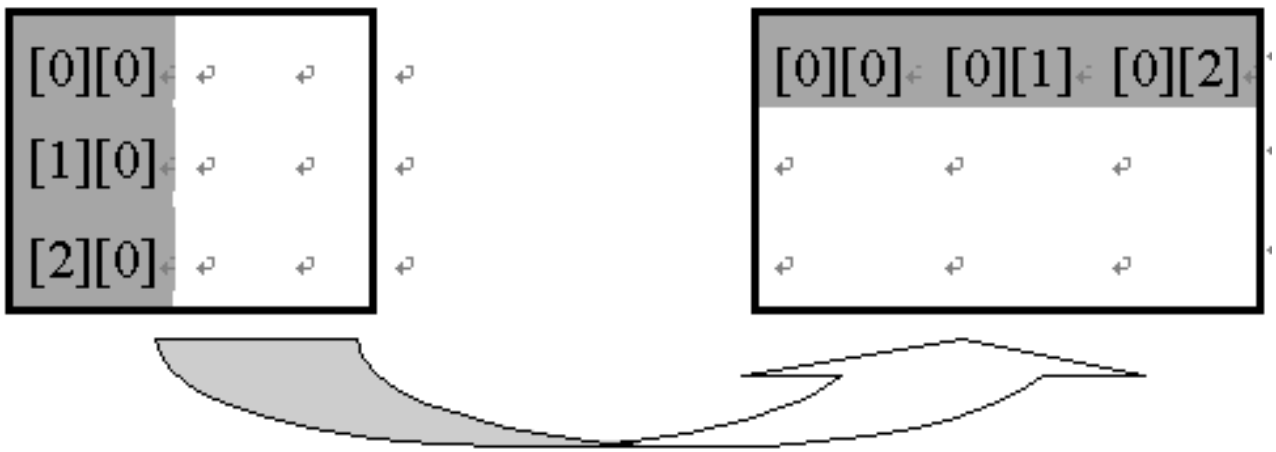
Derived Datatype(派生)

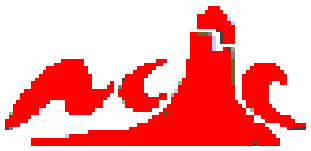
- 常用
  - MPI_Type_vector
  - MPI_Type_indexed



用MPI_Vector进行矩阵的行列置换

- 以C语言的数组表示为例：将矩阵的一列送到另一数组的一行中

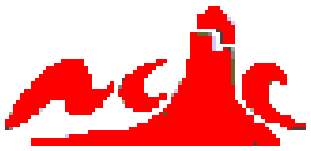




MPI_Vector函数原型

- MPI_Vector()函数首先通过连续复制若干个旧数据类型形成一个“块”，然后通过等间隔地复制该块儿形成新的数据类型。块与块之间的空间时旧数据类型的倍数。

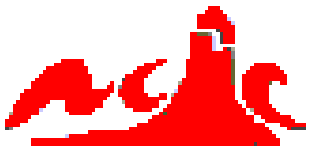
```
#include "mpi.h"
int MPI_Type_vector  (
    int count,                  /*数据块个数 (非负整数)*/
    int blocklen,               /*块中元素个数 (非负整数)*/
    int stride,                 /*块间起始地址间隔 (非负整数)*/
    MPI_Datatype   old_type,        /*原始数据类型(句柄)*/
    MPI_Datatype   *newtype         /*派生数据类型指针*/
)
```



MPI_Type_vector应用示意

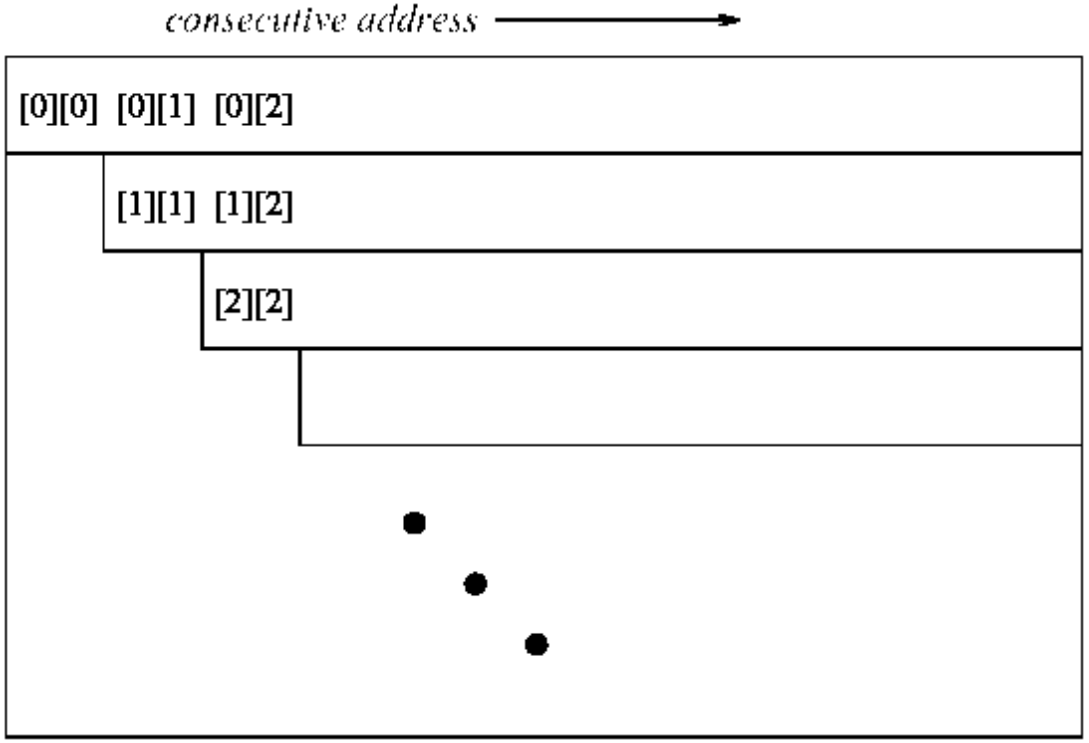
用MPI_Vector进行矩阵的行列置换

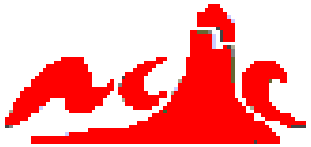
```
...
float A[10][10];
MPI_Datatype column_mpi_t;

MPI_Type_vector(10, 1, 10, MPI_FLOAT,  &column_mpi_t);
MPI_Type_commit(&column_mpi_t);

if (my_rank == 0)
   MPI_Send(&(A[0][0]), 1, column_mpi_t, 1, 0, MPI_COMM_WORLD);
else {    /* my_rank = 1 */
   MPI_Recv(&(A[0][0]),  10,  MPI_FLOAT,  0,  0,
                    MPI_COMM_WORLD,  &status);
 ...
```



用MPI_Type_indexed发送矩阵的上三角部分

- 以C语言表示的数组为例,数组按行连续存储



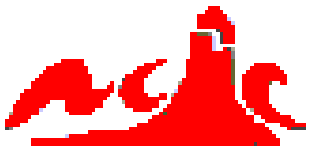


MPI_Type_indexed函数原型

```
#include "mpi.h"

int MPI_Type_indexed  (
    int count,                  /*数据块的个数，数据块间不连续*/
    int blocklens[],  /*每一数据块中元素的个数，为一个非负整型数组*/
    int indices[],      /*每一块数据在原始数据类型中的起始位置,整型数组*/
    MPI_Datatype old_type,          /*原始数据类型(名柄)*/
    MPI_Datatype* newtype           /*派生数据类型指针*/
)
```

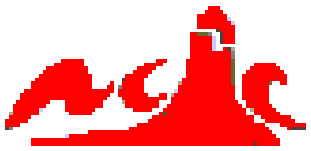
MPI_Type_indexed应用示意(将A矩阵的上三角部分送到另一个处理器中的T矩阵的对应位置)

```
float        A[n][n];    /* Complete Matrix */
float        T[n][n];    /* Upper Triangle  */
int          displacements[n];
int          block_lengths[n];
MPI_Datatype  index_mpi_t;

for (i = 0; i < n; i++) {
   block_lengths[i] = n-i;
   displacements[i] = (n+1)*i;
}
MPI_Type_indexed(n, block_lengths, displacements,MPI_FLOAT, &index_mpi_t);
MPI_Type_commit(&index_mpi_t);

if (my_rank == 0)
     MPI_Send(A, 1, index_mpi_t, 1, 0, MPI_COMM_WORLD);
else /* my_rank == 1 */
     MPI_Recv(T, 1, index_mpi_t, 0, 0, MPI_COMM_WORLD,  &status);
```



其它派生类型

- MPI_Hvector
- MPI_Hindexed
- MPI_Type_struct:结构
  - MPI_Hindexed更普遍的类型
- MPI_Pack/MPI_Unpack:数据打包/解包
  - 是其它数据派生数据类型的基础，MPI不建议用户进行显式的数据打包
  - 为了与早期其它并行库兼容

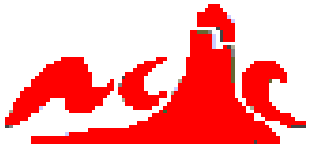


MPI_Pack ()

```
int MPI_Pack (
    void          *inbuf,        /* 输入缓冲区起始地址*/
    int           incount,       /* 输入数据项个数 */
    MPI_Datatype  datatype,      /* 输入数据项的数据类型 */
    void          *outbuf,       /* 输出缓冲区起始地址 */
    int           outcount,      /* 输出缓冲区大小 */
    int           *position,     /* 输出缓冲区当前位置 */
    MPI_Comm      comm           /* 通信域 */
)
```

例：

```
packsize=0;
MPI_Pack(&a,1,MPI_INT,packbuf,100,&packsize,MPI_COMM_WORLD);
MPI_Pack(&b,1,MPI_DOUBLE, packbuf,100,&packsize,MPI_COMM_WORLD);
```

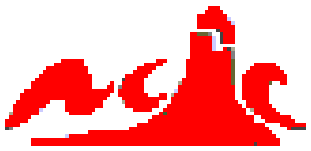


MPI_Unpack()

```
int MPI_Unpack (
    void          *inbuf,          /* 输入缓冲区起始地址*/
    int           incount,         /* 输入数据项大小*/
    int           *position,        /* 缓冲区当前位置 */
    void          *outbuf,         /* 输出缓冲区起始地址 */
    int           outcount,        /* 输出缓冲区大小 */
    MPI_Datatype      datatype, /* 输出数据项的数据类型 */
    MPI_Comm          comm /* 通信域 */
)
```

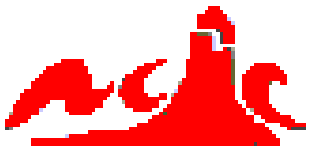
例：

```
pos=0;
MPI_Unpack(packbuf,packsize,&pos,&a,1,MPI_INT,MPI_COMM_WROLD);
MPI_Unpack(packbuf,packsize,&pos,&b,1,MPI_FLOAT,MPI_COMM_WROLD);
```



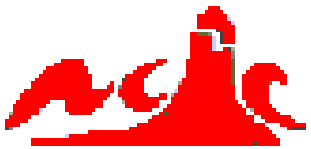
派生数据类型的应用

- 提交:int MPI_Type_commit(MPI Datatype *datatype)
  - 将数据类型映射进行转换或“编译”
  - 一种数据类型变量可反复定义，连续提交
- 释放:int MPI_Type free(MPI_Datatype *datatype)
  - 将数据类型设为MPI_DATATYPE_NULL



讲座内容提示

- 基本的MPI
  - 基本概念
  - 点到点通信(Point to point)
    - MPI中API的主要内容，为MPI最基本，最重要的内容
  - MPI程序的编译和运行
- 深入MPI
  - 用户自定义(/派生)数据类型(User-defined(Derived) data type)
    - 事实上MPI的所有数据类型均为MPI自定义类型
      - 支持异构系统
      - 允许消息来自不连续的或类型不一致的存储区(结构,数组散元)
  - 集合通信(Collective)
    - 数据移动，数据聚集，同步
    - 基于point to point 构建
  - MPI环境管理函数
    - 组,上下文和通信空间/通信子的管理
- 实例



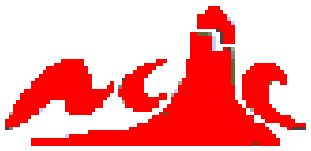
集合通信
Collective Communication

- 特点
  - 通信空间中的所有进程都参与通信操作
  - 每一个进程都需要调用该操作函数
- 一到多
- 多到一
- 同步

All:表示结果到所有进程.
V:Vector,被操作的数据对象和操作更为灵活.

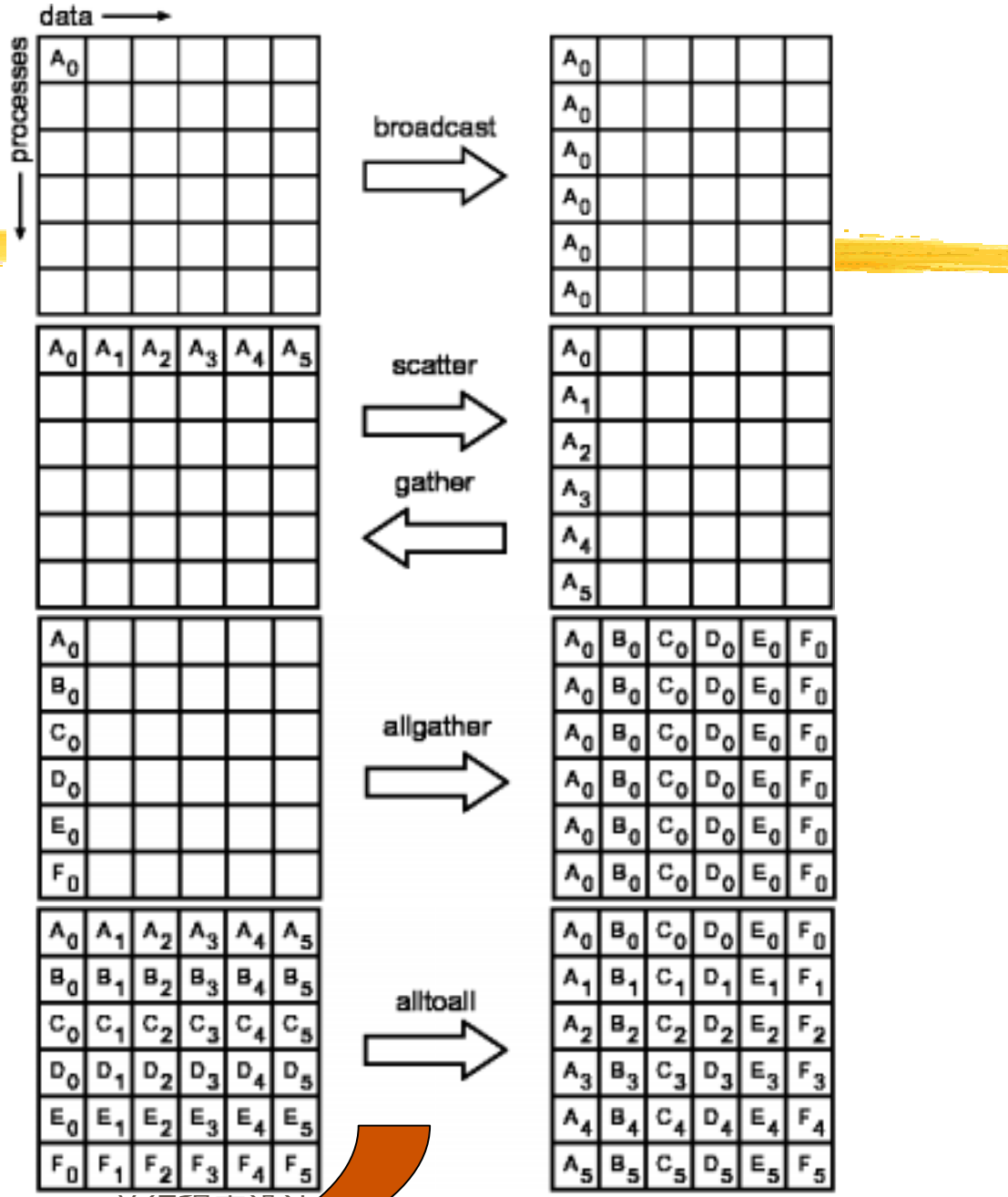
MPI集合通信函数

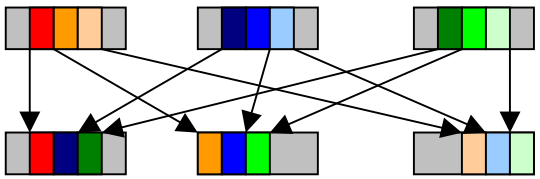
类型	函数	功能
数据移动	MPI_Bcast	一到多，数据广播
	MPI_Gather	多到一，数据汇合
	MPI_Gatherv	MPI_Gather的一般形式
	MPI_Allgather	MPI_Gather的一般形式
	MPI_Allgatherv	MPI_Allgather的一般形式
	MPI_Scatter	一到多，数据分散
	MPI_Scatterv	MPI_Scatter的一般形式
	MPI_Alltoall	多到多，置换数据(全互换)
	MPI_Alltoallv	MPI_Alltoall的一般形式
数据聚集	MPI_Reduce	多到一，数据归约
	MPI_Allreduce	上者的一般形式，结果在所有进程
	MPI_Reduce_scatter	结果scatter到各个进程
	MPI_Scan	前缀操作
同步	MPI_Barrier	同步操作



数据移动

- Broadcast
- Scatter
- Gather
- Allgather
- Alltoall



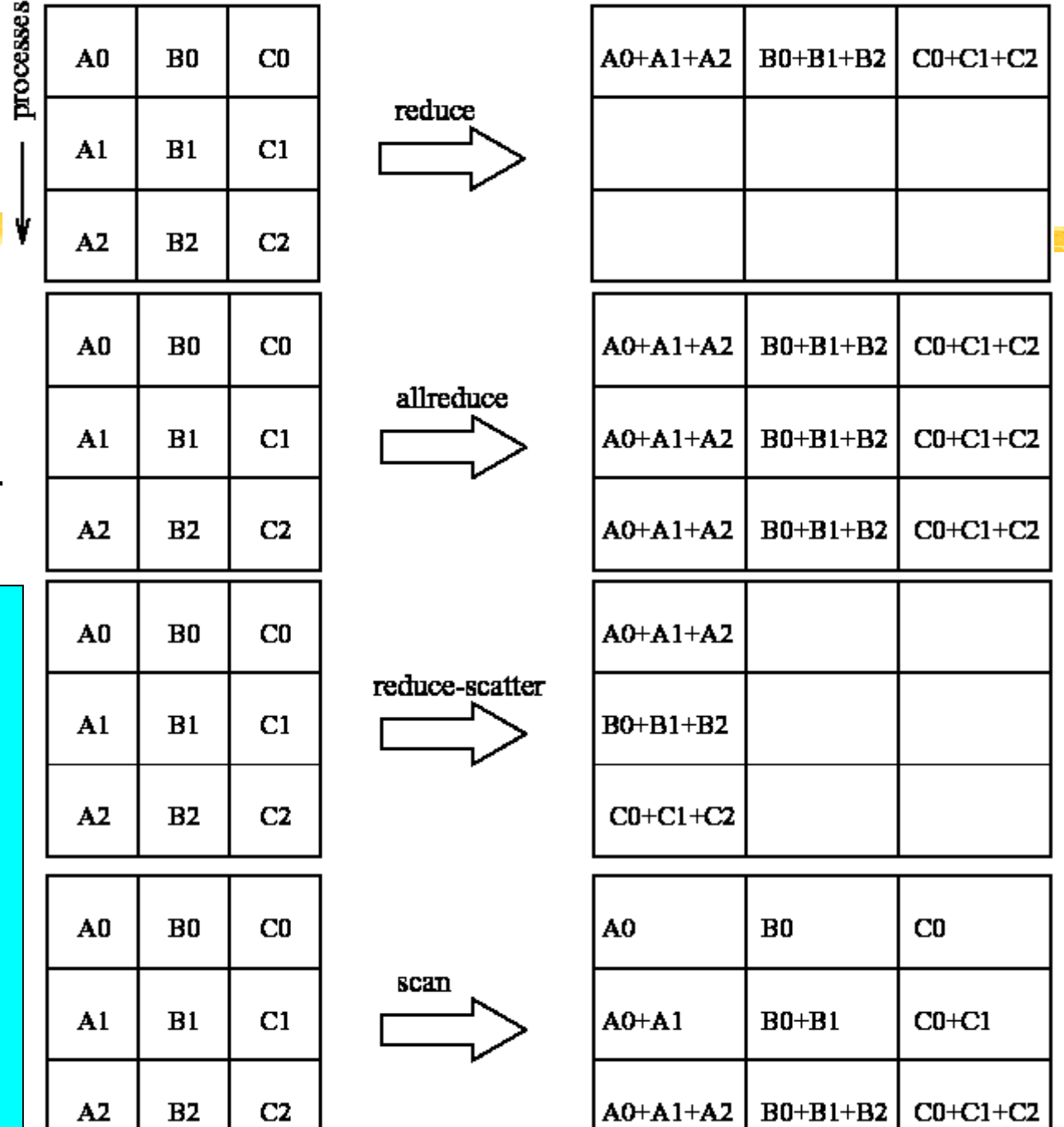


数据聚集

Reduce
Allreduce
Reduce-scatter
Scan

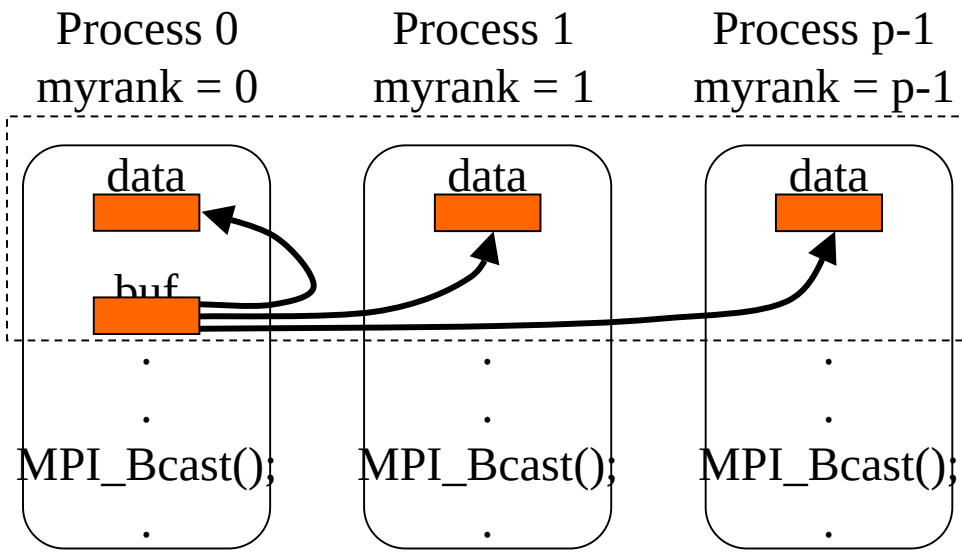
MPI 预定义全局数据运算符:

MPI_MAX / MPI_MIN;
MPI_SUM 求和
MPI_PROD 求积
MPI_LAND 逻辑与
MPI_LOR 逻辑或
MPI_MAXLOC/MPI_MINLOC 最大/小值求下相应位置



Broadcast -- 数据广播

```
nt p, myrank;
loat buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,
    &my_rank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
f(myrank==0)
     buf = 1.0;
MPI_Bcast(&buf,1,MPI_FLOAT,
    0, comm);
```

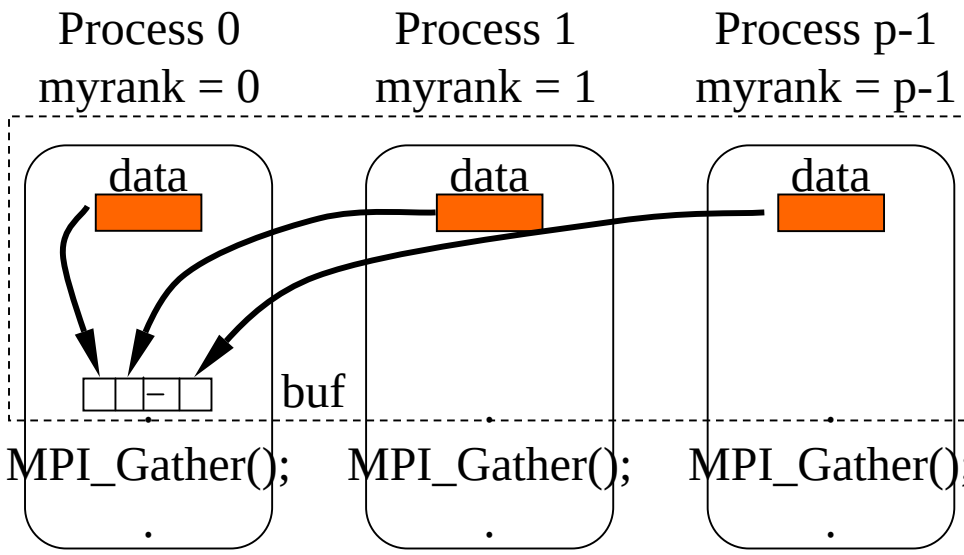



```
int MPI_Bcast (
        void *buffer,/*发送/接收buf*/
        int     count, /*元素个数*/
        MPI_Datatype datatype,
        int root,                   /*指定
根进程*/
        MPI_Comm comm)
```

根进程既是发送缓冲区也是接收缓冲区

Gather -- 数据收集

```
nt p, myrank;
loat data[10];/*分布变量*/
loat* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_ra
    nk);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
f(myrank==0)
    buf=(float*)malloc(p*10*size
    of(float);/*开辟接收缓冲区*/

MPI_Gather(data,10,MPI_FLOAT,
    buf,10,MPI_FlOAT,0,comm);
```

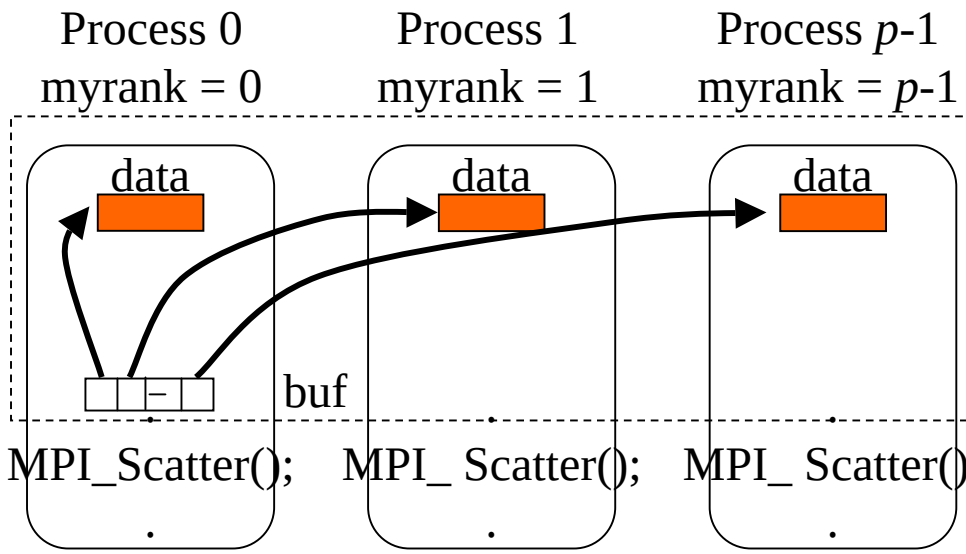


根进程接收其他进程来的消息(包括根进程),按每在进程在通信组中的编号依次联接在一下,存放在要进程的接收缓冲区中.

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

Scatter -- 数据分散

```
nt p, myrank;
loat data[10];
loat* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_ra
    nk);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
f(myrank==0)
    buf =
    (float*)malloc(p*10*sizeof(flo
    at);/*开辟接收缓冲区*/
MPI_Scatter(buf,10,MPI_FLOAT,
    data,10,MPI_FlOAT,0,comm);
```

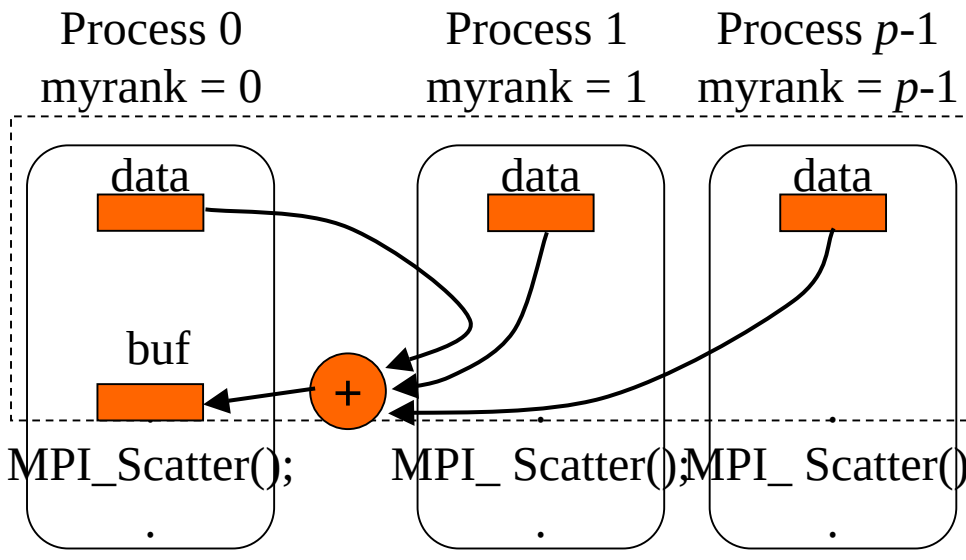


根进程中存储了*p*个消息,第*i*个消息将传给第*i*个进程.

```
int MPI_Scatter ( void *sendbuf, int
sendcnt, MPI_Datatype sendtype, void
*recvbuf, int recvcnt, MPI_Datatype
recvtype, int root, MPI_Comm comm )
```

Reduce -- 全局数据运算

```
nt p, myrank;
loat data = 0.0;
loat buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_ra
    nk);

/*各进程对data进行不同的操作*/
data = data + myrank * 10;

/*将各进程中的data数相加并存入根
    进程的buf中 */
MPI_Reduce(&data,&buf,1,MPI_F
    LOAT,MPI_SUM,0,comm);
```

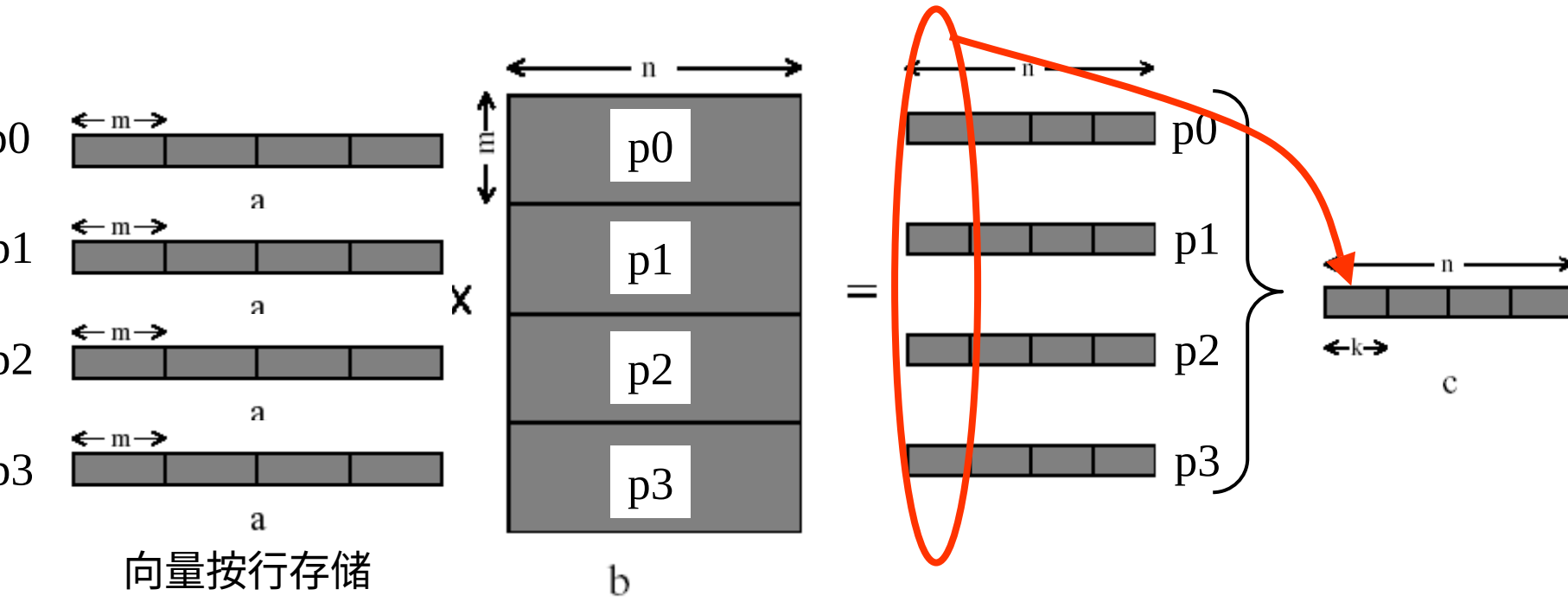


对组中所有进程的发送缓冲区中的数据用OP参数指定的操作进行运算,并将结果送回到根进程的接收缓冲区中.

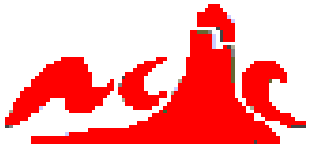
```
int MPI_Reduce ( void *sendbuf, void
*recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm
comm )
```

Reduce_scatter

$$A_{1\times m} \cdot B_{m\times n} = C_{1\times n}$$

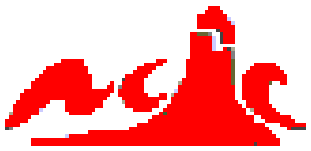


Reduce-scatter在向量-矩阵乘中的应用



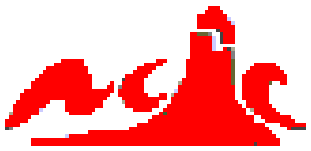
后缀V:更灵活的集合通信

- 带后缀V的集合通信操作是一种更为灵活的集合通信操作
  - 通信中元素块的大小可以变化
  - 发送与接收时的数据位置可以不连续



MPI_Gather

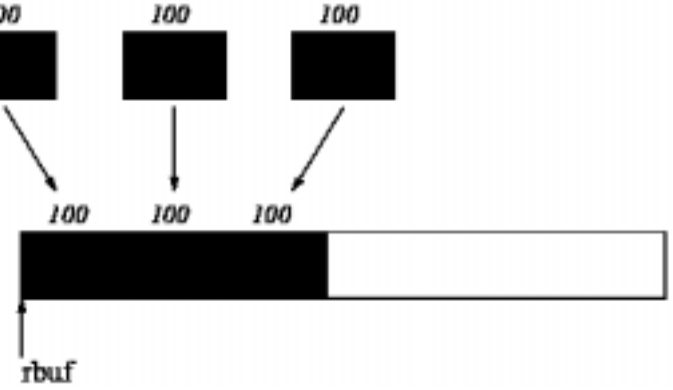
- `int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm )` 参数:
- **sendbuf** 发送缓冲区起始位置
- **sendcount** 发送元素个数
- **sendtype** 发送数据类型
- **recvcount** 接收元素个数**(所有进程相同)** (该参数仅对根进程有效)
  recvtype 接收数据类型(仅在根进程中有效)
- **root** 通过rank值指明接收进程
- **comm** 通信空间

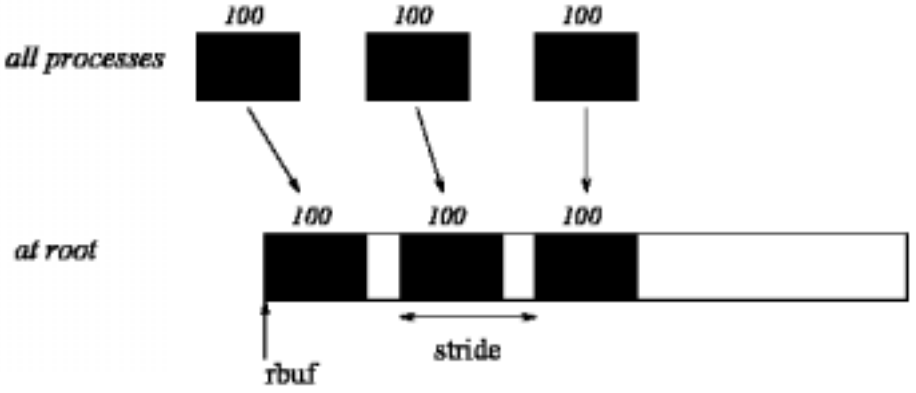


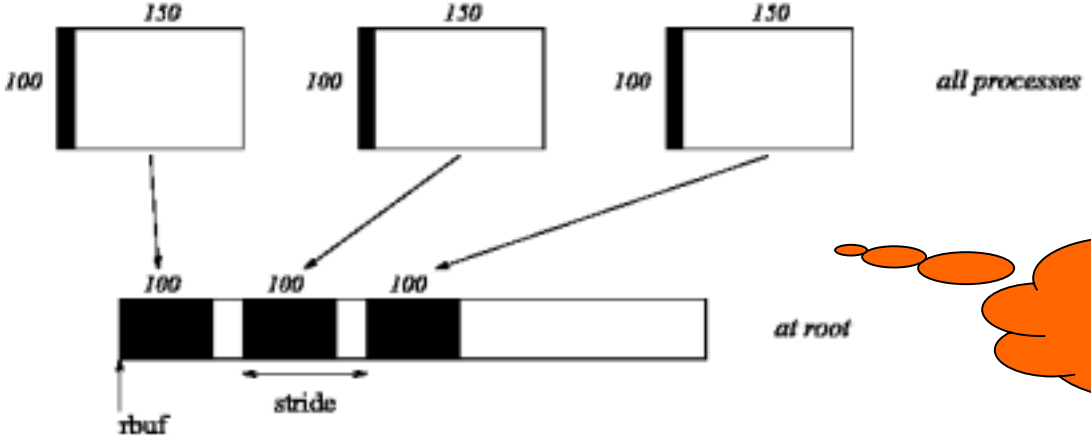
MPI_Gatherv

- `int MPI_Gatherv ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm )`
- 参数:
- sendbuf 发送缓冲区的起始位置
- sendcount 发送元素个数
- sendtype 发送数据类型
  recvcounts 整型数组(大小等于组的大小),用于指明从各进程要接收的元素的个数(仅对根进程有效)
- displs 整型数组(大小等于组的大小). 其元素 i指明要接收元素存放位置相对于接收缓冲区起始位置的偏移量 (仅在根进程中有效)
- recvtype 接收数据类型
- root 通过rank值指明接收进程
  comm 通信空间

Gather与GatherV

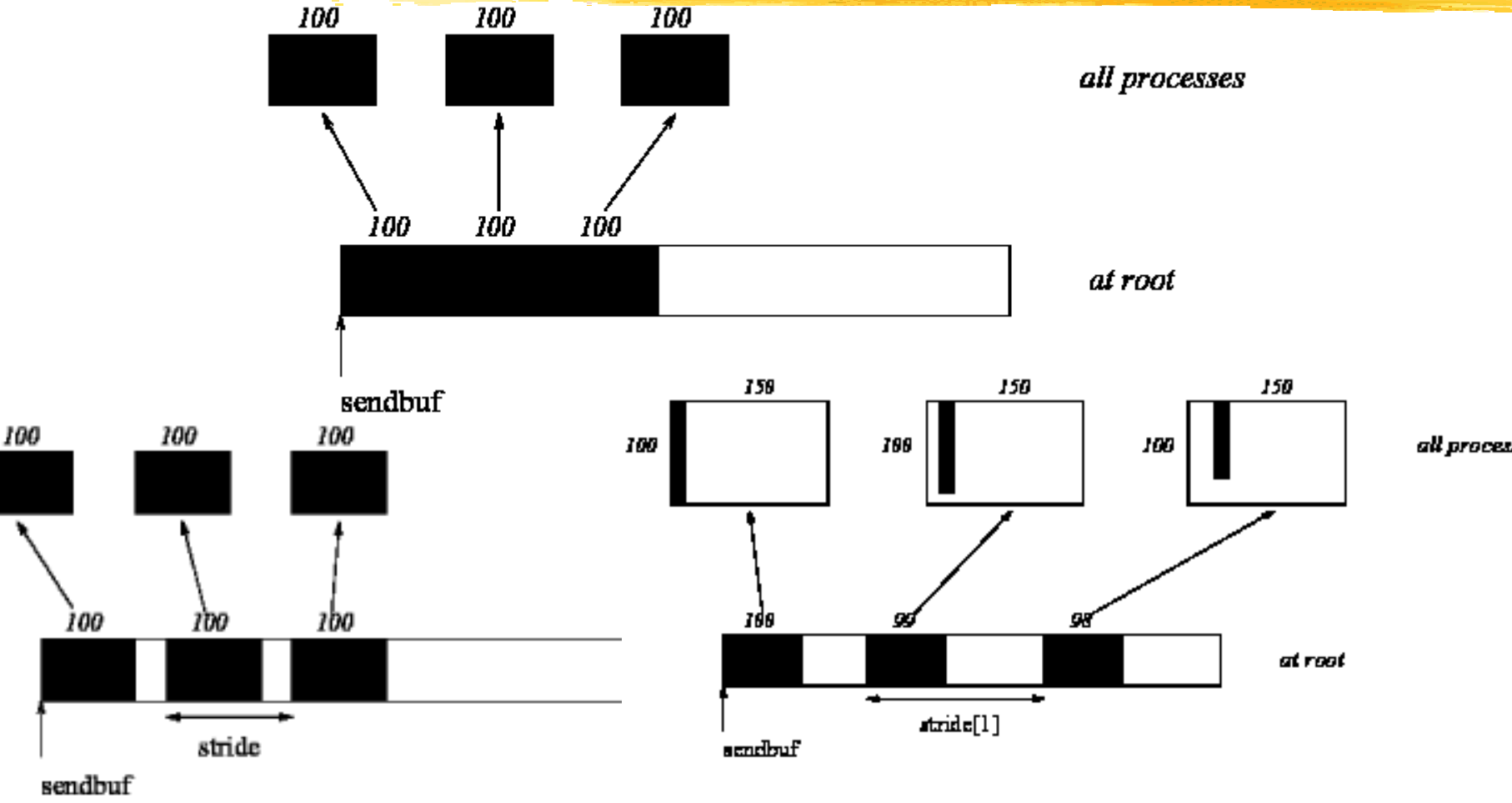


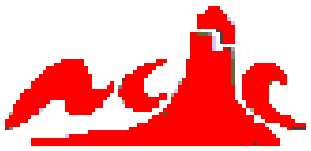




应用Vector派生数据类型

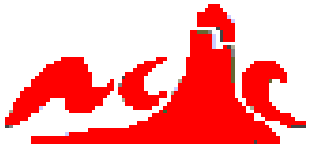
Scatter与ScatterV





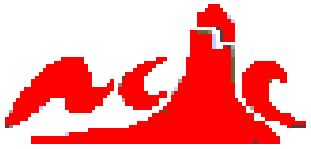
讲座内容提示

- 基本的MPI
  - 基本概念
  - 点到点通信(Point to point)
    - MPI中API的主要内容，为MPI最基本，最重要的内容
  - MPI程序的编译和运行
- 深入MPI
  - 用户自定义(/派生)数据类型(User-defined(Derived) data type)
    - 事实上MPI的所有数据类型均为MPI自定义类型
      - 支持异构系统
      - 允许消息来自不连续的或类型不一致的存储区(结构,数组散元)
  - 集合通信(Collective)
    - 数据移动，数据聚集，同步
    - 基于point to point 构建
  - MPI环境管理函数
    - 组,上下文和通信空间/通信子的管理
- 实例



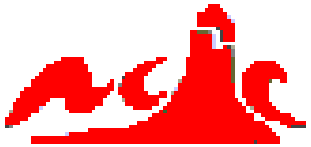
MPI环境管理

- MPI起动与结束:
  - MPI_Init();
  - MPI_Initialized();测试是否已执行MPI_Init();
  - MPI_Finalize();
- MPI计时函数
  - double MPI_Wtime();返回自过去某一时刻调用时的时间间隔,以秒为单位.
  - double MPI_Wtick();返回用作硬作计时的两次脉冲间的间隔时间，以秒为单位.
- 组,上下文和通信空间管理(略).



讲座内容提示

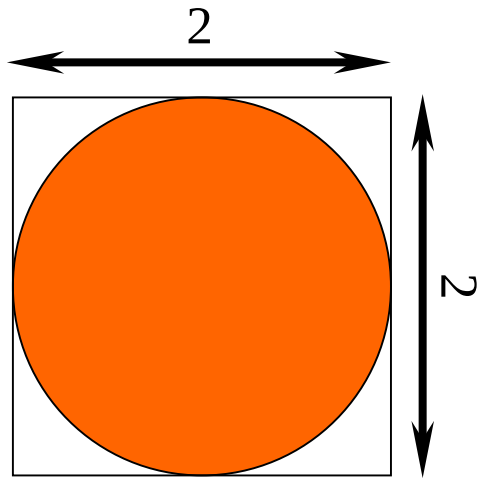
- 基本的MPI
  - 基本概念
  - 点到点通信(Point to point)
    - MPI中API的主要内容，为MPI最基本，最重要的内容
  - MPI程序的编译和运行
- 深入MPI
  - 用户自定义(/派生)数据类型(User-defined(Derived) data type)
    - 事实上MPI的所有数据类型均为MPI自定义类型
      - 支持异构系统
      - 允许消息来自不连续的或类型不一致的存储区(结构,数组散元)
  - 集合通信(Collective)
    - 数据移动，数据聚集，同步
    - 基于point to point 构建
  - MPI环境管理函数
    - 组,上下文和通信空间/通信子的管理
- 实例

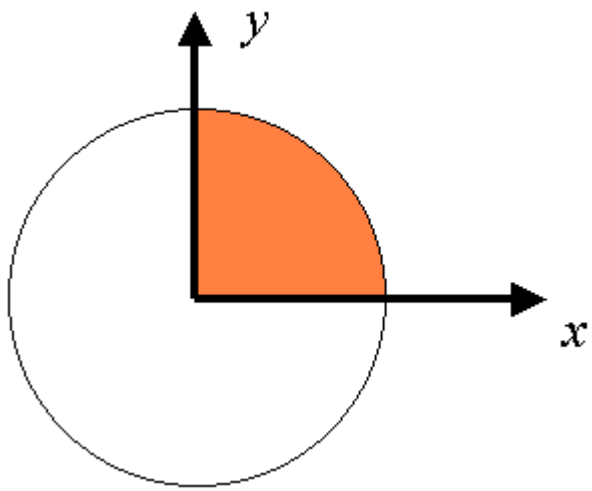


实例分析

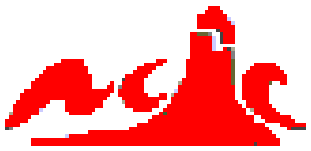
- 求PI
- 向量点积算法及程序
- 矩阵向量相乘算法及程序
- 矩阵乘积算法及程序

实例分析:求PI





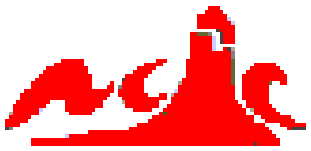
$$\int_0^1 \sqrt{1-x^2}\,\mathrm{d}x = \frac{\pi}{4}$$



串行代码

```
h=1.0/(double)n;
sum=0.0;
for (i=1; i<=n; i++) {
   x=h*((double)i – 0.5);
   sum += f(x);
}
pi=h*sum;
```

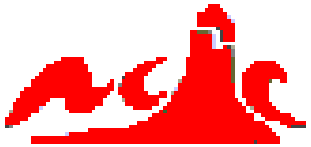
```
double f(double a)
{
  return (4.0/(1.0+a*a));
}
```



并行代码

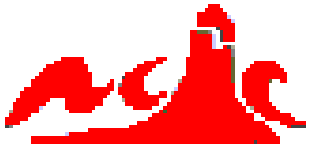
```
h=1.0/(double)n;
sum=0.0;
for (i=myid+1; i<=n;
   i+=numprocs) {
   x=h*((double)i – 0.5);
   sum += f(x);
}
mypi=h*sum;
 MPI_Reduce(&mypi, &pi, 1,
   MPI_DOUBLE, MPI_SUM, 0,
   MPI_COMM_WORLD);
```

```
double f(double a)
{
   return (4.0/(1.0+a*a));
}
```



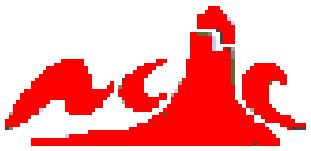
cpi.c

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a );
{
    return (4.0 / (1.0 + a*a));
}
```



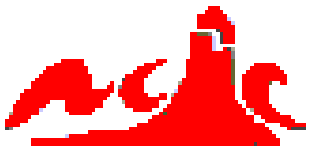
cpi.c

```
nt main( int argc, char *argv[])
  int done = 0, n, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi, h, sum, x;
  double startwtime = 0.0, endwtime;
  int  namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Get_processor_name(processor_name,&namelen)
  fprintf(stderr,"Process %d on %s\n",  myid,
  processor_name);
```



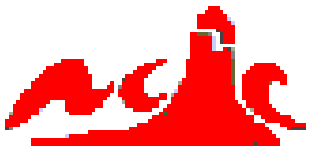
cpi.c

```
n = 0;
while (!done)
{
   if (myid == 0)
   {
      if (n==0) n=100; else n=0;
      startwtime = MPI_Wtime();
   }
   MPI_Bcast(&n, 1, MPI_INT, 0,
MPI_COMM_WORLD);
```



cpi.c

```
if (n == 0)
    done = 1;
else  {
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
    {
        x = h * ((double)i - 0.5);
        sum += f(x);
    }
    mypi = h * sum;
```



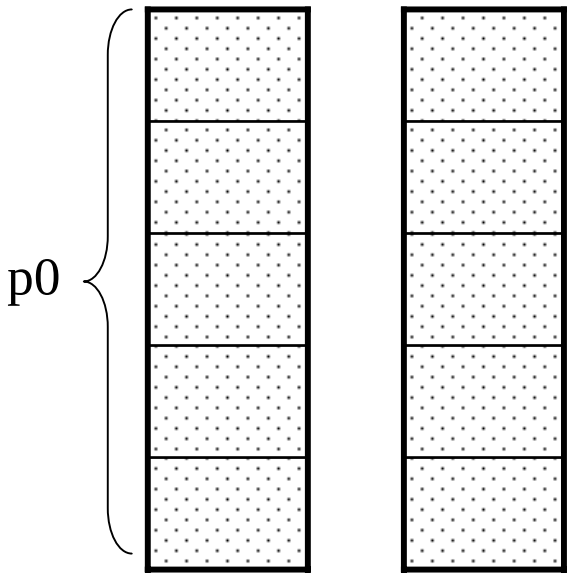
cpi.c

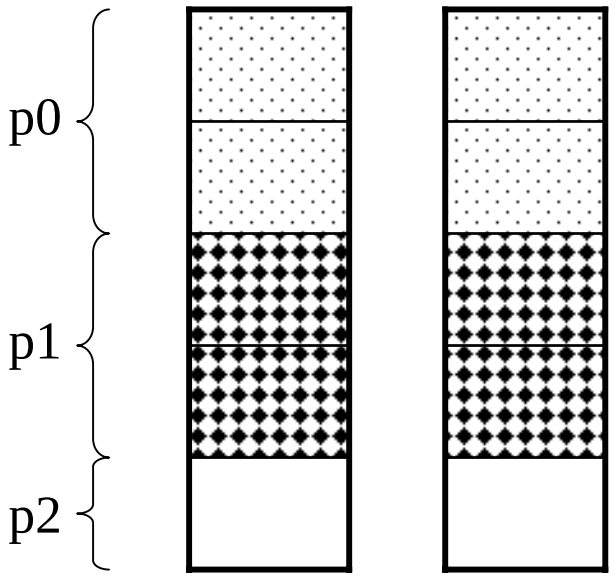
```
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
MPI_SUM, 0, MPI_COMM_WORLD);
      if (myid == 0)  {
          printf("pi is approximately %.16f, Error is
%.16f\n", pi, fabs(pi - PI25DT));
          endwtime = MPI_Wtime();
          printf("wall clock time = %f\n",
endwtime-startwtime);
      }
   }
}
MPI_Finalize();
return 0;
```

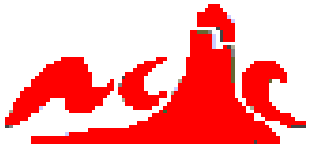
实例分析:点积运算

$$c = \sum_{i=0}^{n-1} a_i \cdot b_i$$

$$c = \sum_{j=0}^{n/p} \sum_{i=0}^{n_j - 1} a_i \cdot b_i$$







Parallel_dot.c

```
/* parallel_dot.c -- compute a dot product of a
*  vector distributed among the processes.
*  Uses a block distribution of the vectors.
*  Input:
*     n: global order of vectors
*     x, y:  the vectors
*  Output:
*     the dot product of x and y.
*
* Note:  Arrays containing vectors are statically allocated.  Assumes
*     n, the global order of the vectors, is divisible by p, the number
*     of processes.
*/
```

```
include <stdio.h>
include "mpi.h"
define MAX_LOCAL_ORDER 100
ain(int argc, char* argv[]) {
  float  local_x[MAX_LOCAL_ORDER];
  float  local_y[MAX_LOCAL_ORDER];
  int    n;
  int    n_bar;  /* = n/p */
  float  dot;
  int    p;
  int    my_rank;
  void Read_vector(char* prompt, float local_v[], int n_bar, int p,
          int my_rank);
  float Parallel_dot(float local_x[], float local_y[], int n_bar);
```

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == 0) {
    printf("Enter the order of the vectors\n");
    scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
n_bar = n/p;

Read_vector("the first vector", local_x, n_bar, p, my_rank);
Read_vector("the second vector", local_y, n_bar, p, my_rank);

dot = Parallel_dot(local_x, local_y, n_bar);

if (my_rank == 0)
    printf("The dot product is %f\n", dot);
MPI_Finalize();
/* main */
```

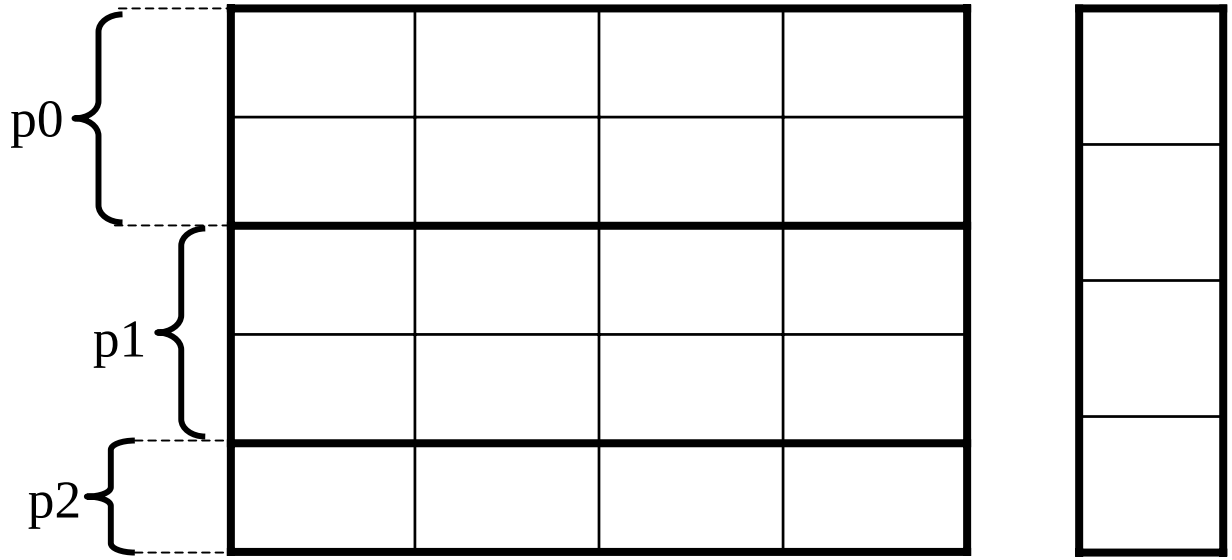
```
oid Read_vector(char*  prompt/* in */,float local_v[]/* out */,
nt    n_bar/* in  */,int    p/* in  */, int    my_rank/* in  */)

  int i, q;
  float temp[MAX_LOCAL_ORDER];
  MPI_Status status;
  if (my_rank == 0) {
     printf("Enter %s\n", prompt);
     for (i = 0; i < n_bar; i++)
        scanf("%f", &local_v[i]);
     for (q = 1; q < p; q++) {
        for (i = 0; i < n_bar; i++)
           scanf("%f", &temp[i]);
        MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
     }
  } else {
     MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                   &status);
  }
  /* Read_vector */
```

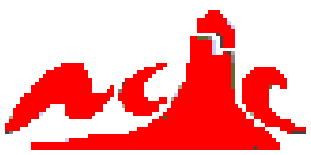
```
oat Serial_dot(
        float  x[]  /* in */,
        float  y[]  /* in */,
        int    n    /* in */) {
  int    i;
  float  sum = 0.0;
  for (i = 0; i < n; i++)
      sum = sum + x[i]*y[i];
  return sum;
/* Serial_dot */
```

```
oat Parallel_dot(
        float  local_x[]  /* in */,
        float  local_y[]  /* in */,
        int  n_bar        /* in */)

  float  local_dot;
  float  dot = 0.0;
  float  Serial_dot(float x[], float y[], int m);

  local_dot = Serial_dot(local_x, local_y, n_bar);
  MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
      MPI_SUM, 0, MPI_COMM_WORLD);
  return dot;
/* Parallel_dot */
```

实例分析:矩阵向量相乘

$$C_i = \sum_{j=0}^{n-1} a_{ij} b_j \, (i = 0,1,\cdots,m-1)$$





parallel_mat_vect.c

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

define MAX_ORDER 100

ypedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];

ain(int argc, char* argv[]) {
  int           my_rank;
  int           p;
 LOCAL_MATRIX_T  local_A;
  float         global_x[MAX_ORDER];
  float         local_x[MAX_ORDER];
  float         local_y[MAX_ORDER];
  int           m, n;
  int           local_m, local_n;
```

```
void Read_matrix(char* prompt, LOCAL_MATRIX_T local_A, int local_m,
      int n, int my_rank, int p);
void Read_vector(char* prompt, float local_x[], int local_n, int my_rank,
      int p);
void Parallel_matrix_vector_prod( LOCAL_MATRIX_T local_A, int m,
      int n, float local_x[], float global_x[], float local_y[],
      int local_m, int local_n);
void Print_matrix(char* title, LOCAL_MATRIX_T local_A, int local_m,
      int n, int my_rank, int p);
void Print_vector(char* title, float local_y[], int local_m, int my_rank,
      int p);

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == 0) {
   printf("Enter the order of the matrix (m x n)\n");
   scanf("%d %d", &m, &n);
}
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
local_m = m/p;
local_n = n/p;

Read_matrix("Enter the matrix", local_A, local_m, n, my_rank, p);
Print_matrix("We read", local_A, local_m, n, my_rank, p);

Read_vector("Enter the vector", local_x, local_n, my_rank, p);
Print_vector("We read", local_x, local_n, my_rank, p);

Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x,
   local_y, local_m, local_n);
Print_vector("The product is", local_y, local_m, my_rank, p);

MPI_Finalize();

/* main */
```

```
void Read_matrix(char  prompt  /* in  */,
      LOCAL_MATRIX_T  local_A  /* out */,
      int             local_m  /* in  */,
      int             n        /* in  */,
      int             my_rank  /* in  */,
      int             p        /* in  */) {

int           i, j;
LOCAL_MATRIX_T  temp;
/* Fill dummy entries in temp with zeroes */
for (i = 0; i < p*local_m; i++)
   for (j = n; j < MAX_ORDER; j++)
      temp[i][j] = 0.0;
if (my_rank == 0) {
   printf("%s\n", prompt);
   for (i = 0; i < p*local_m; i++)
      for (j = 0; j < n; j++)
         scanf("%f",&temp[i][j]);
}
MPI_Scatter(temp, local_m*MAX_ORDER, MPI_FLOAT, local_A,
   local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);
/* Read_matrix */
```

```
oid Read_vector(
      char*  prompt     /* in  */,
      float  local_x[]  /* out */,
      int    local_n    /* in  */,
      int    my_rank    /* in  */,
      int    p          /* in  */) {

int   i;
float temp[MAX_ORDER];

if (my_rank == 0) {
    printf("%s\n", prompt);
    for (i = 0; i < p*local_n; i++)
        scanf("%f", &temp[i]);
}
MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n, MPI_FLOAT, 0,
 MPI_COMM_WORLD);

/* Read_vector */
```

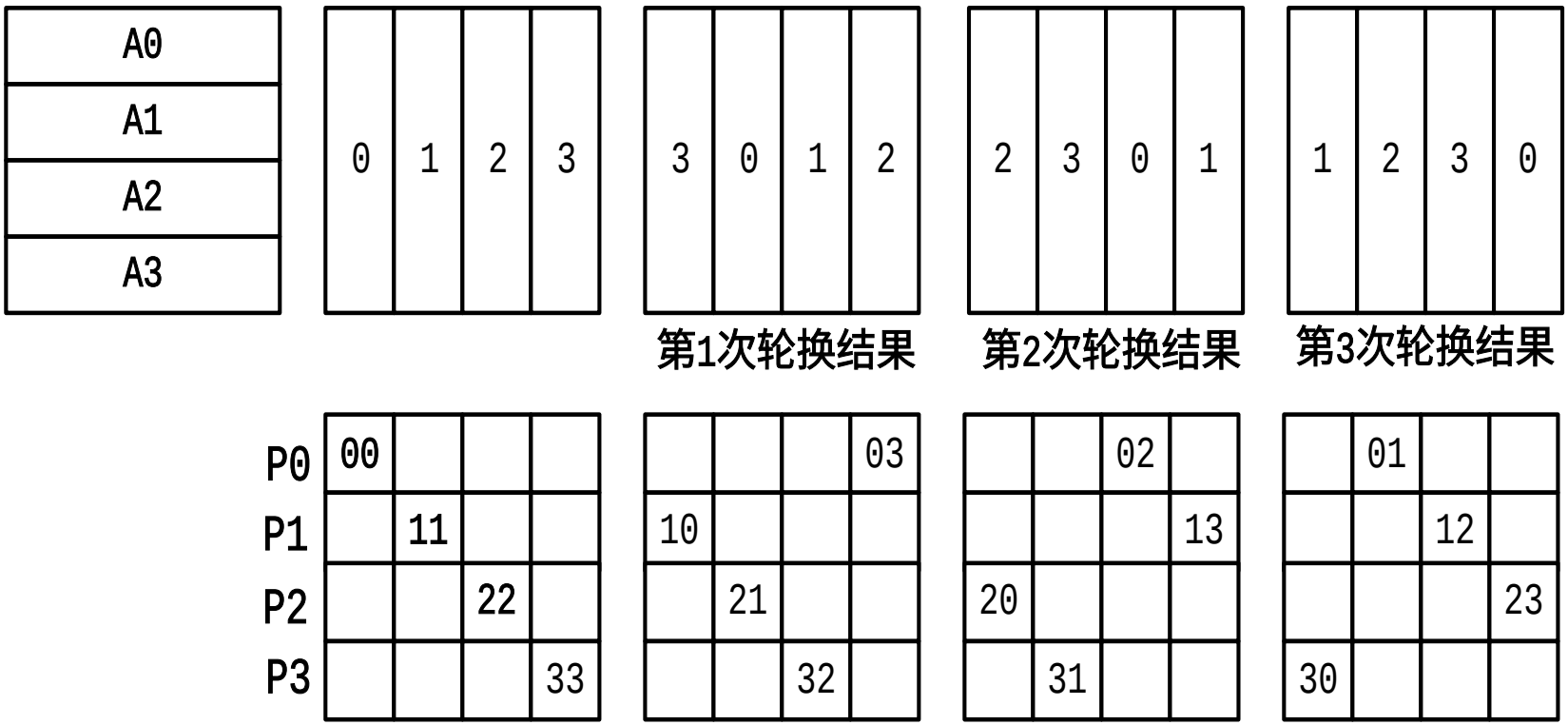
```
 * All arrays are allocated in calling program.  */
/* Note that argument m is unused                 */
void Parallel_matrix_vector_prod(
      LOCAL_MATRIX_T  local_A     /* in  */,
      int             m           /* in  */,
      int             n           /* in  */,
      float           local_x[]   /* in  */,
      float           global_x[]  /* in  */,
      float           local_y[]   /* out */,
      int             local_m     /* in  */,
      int             local_n     /* in  */) {
   /* local_m = m/p, local_n = n/p */
   int i, j;
   MPI_Allgather(local_x, local_n, MPI_FLOAT,
                 global_x, local_n, MPI_FLOAT, MPI_COMM_WORLD);
   for (i = 0; i < local_m; i++) {
      local_y[i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[i] = local_y[i] + local_A[i][j]*global_x[j];
   }
   /* Parallel_matrix_vector_prod */
```

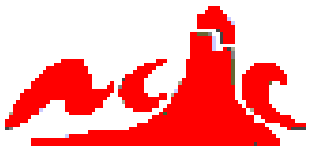
```
void Print_matrix(
      char*         title                 /* in */,
      LOCAL_MATRIX_T  local_A             /* in */,
      int           local_m               /* in */,
      int           n                   /* in */,
      int           my_rank               /* in */,
      int           p                   /* in */) {

int   i, j;
float temp[MAX_ORDER][MAX_ORDER];

MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT, temp,
    local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);
if (my_rank == 0) {
   printf("%s\n", title);
   for (i = 0; i < p*local_m; i++) {
      for (j = 0; j < n; j++)
         printf("%4.1f ", temp[i][j]);
      printf("\n");
   }
}
/* Print_matrix */
```

```
oid Print_vector(
    char*  title       /* in */,
    float  local_y[] /* in */,
    int    local_m    /* in */,
    int    my_rank  /* in */,
    int    p             /* in */) {

int   i;
float temp[MAX_ORDER];

MPI_Gather(local_y, local_m, MPI_FLOAT, temp, local_m, MPI_FLOAT,
   0, MPI_COMM_WORLD);

if (my_rank == 0) {
   printf("%s\n", title);
   for (i = 0; i < p*local_m; i++)
      printf("%4.1f ", temp[i]);
   printf("\n");
}
/* Print_vector */
```

实例分析:矩阵相乘

4处理器 m=n=4





并行程序设计的一些建议

- 优化并行算法
- 大并行粒度
- 顾及负载平衡
- 尽量减少通信次数
- 避免大消息(1M)
  - 避免消息缓冲区的溢出，且效率较低
- 避免大消息打包
  - 内存拷贝开销大


谢谢！