



并行编程初步

张丹丹

上海超级计算中心

2011-3-4



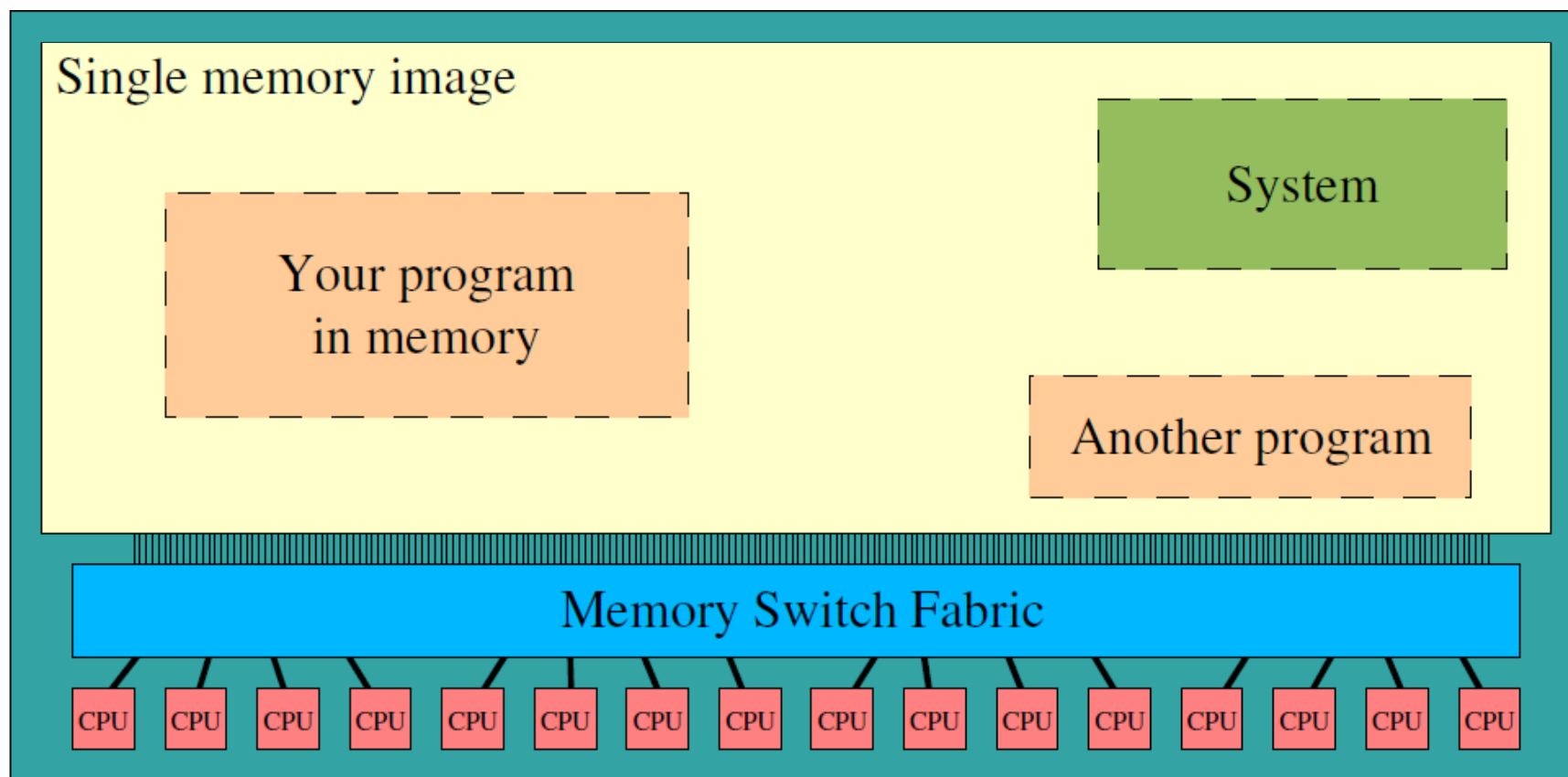
提 纲

- 引言
- 认识MPI编程
- MPI编程简介
- 实例



并行计算机体系架构

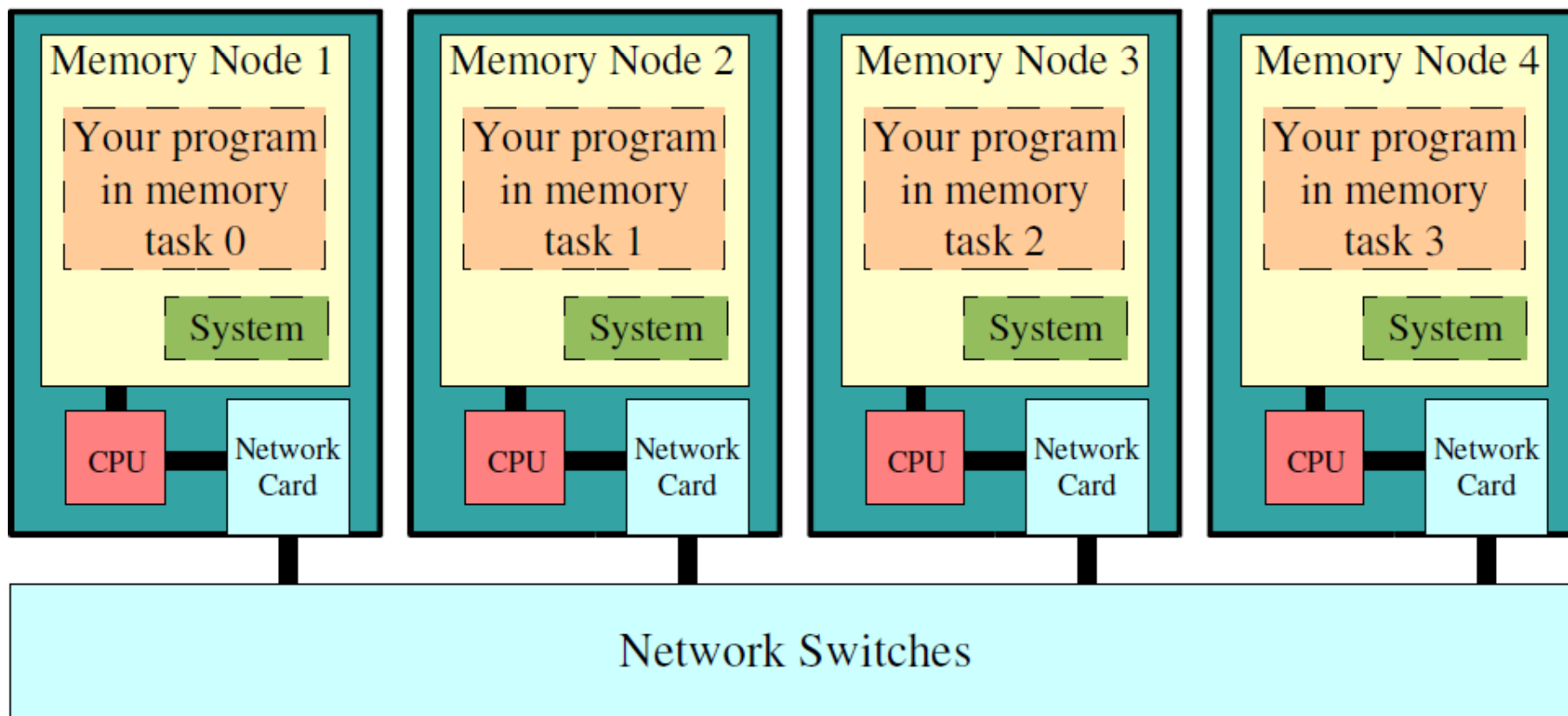
- 共享存储(Shared Memory)





并行计算机体系架构

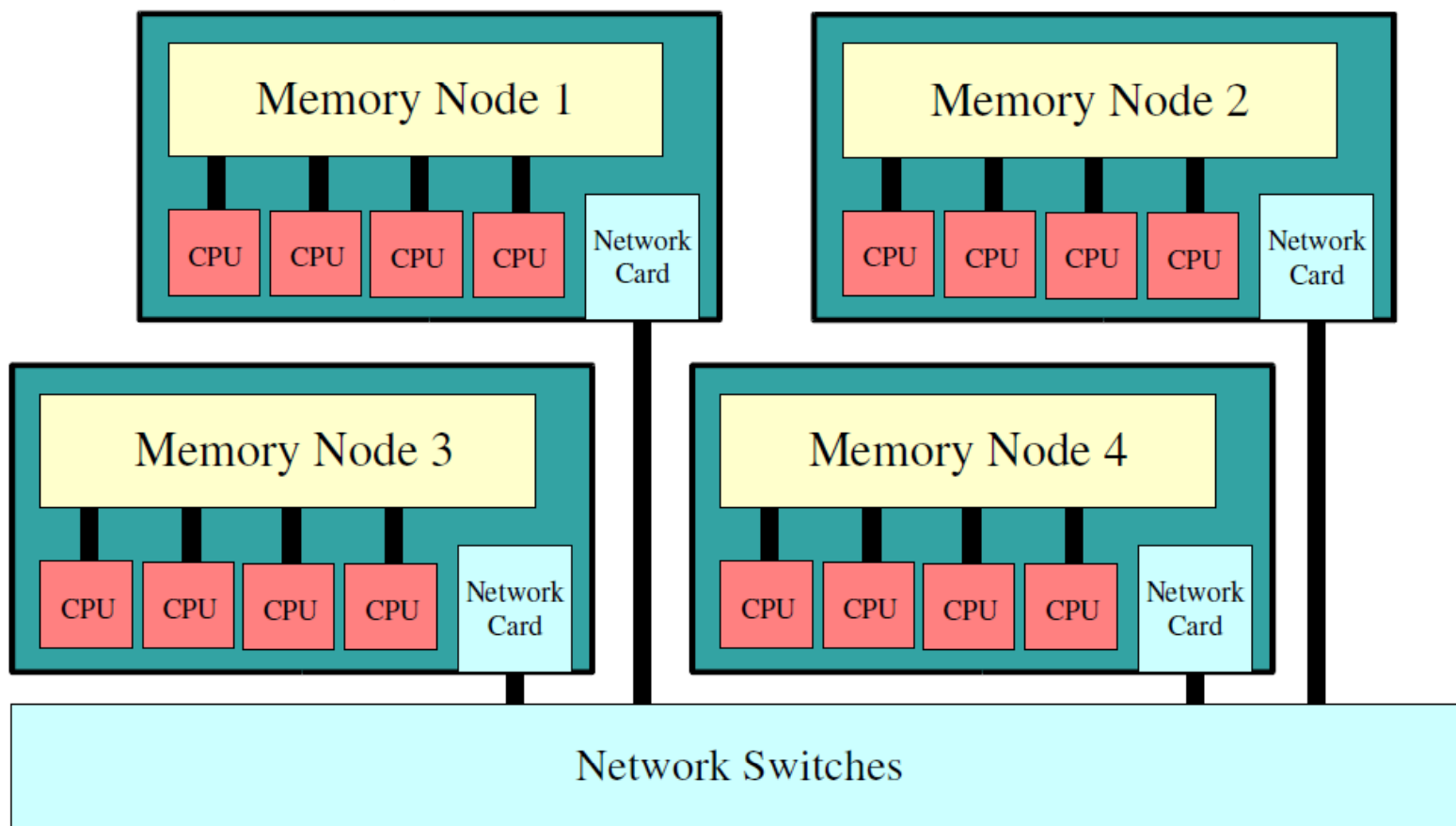
- 分布式存储（Distributed Memory）





并行计算机体系架构

- 混合架构 (Hybrid)





并行编程模型

- 数据并行模型

相同的操作同时作用于不同的数据

- 共享变量模型

用共享变量实现并行进程间的通信

- 消息传递模型

在消息传递模型中，驻留在不同节点上的进程可以通过网络传递消息相互通信，实现进程之间的信息交换、协调步伐、控制执行等。



并行编程模型

特征	数据并行	共享变量	消息传递
典型代表	HPF	OpenMP	MPI, PVM
可移植性	SMP, DSM, MPP	SMP, DSM	所有流行并行 计算机
并行粒度	进程级细粒度	线程级细粒度	进程级大粒度
并行操作方式	松散同步	异步	异步
数据存储模式	共享存储	共享存储	分布式存储
数据分配方式	半隐式	隐式	显示
学习入门难度	偏易	容易	较难
可扩展性	一般	较差	好



什么是MPI?

- Message Passing Interface: 是消息传递函数库的标准规范，由MPI论坛开发，支持Fortran和C/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/C++语言中可以直接对这些函数进行调用
 - 是一种标准或规范，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



为什么要使用MPI?

- 高可移植性
 - MPI已在PC机、MS Windows以及所有主要的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/C++或Fortran并行程序可不加改变地在上述平台实现
- 没有更好的选择



常用的MPI 版本

- 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
 - 当前最新版本有MPICH 1.2.7p1和MPICH2 1.3.2p1
- OpenMPI
 - LAMMPI的下一代MPI实现
 - 当前最新版本1.4.3
- 更多的商业版本MPI
 - HP-MPI, MS-MPI,
- 所有的版本遵循MPI标准, MPI程序可以不加修改的运行



提 纲

- 引言
- 认识MPI编程
- MPI编程简介
- 实例



从简单入手

- 下面我们首先分别以C语言的形式给出一个最简单的MPI并行程序 hello.c
- 该程序在终端打印出Hello World!字样.



Hello.c (C语言)

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

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



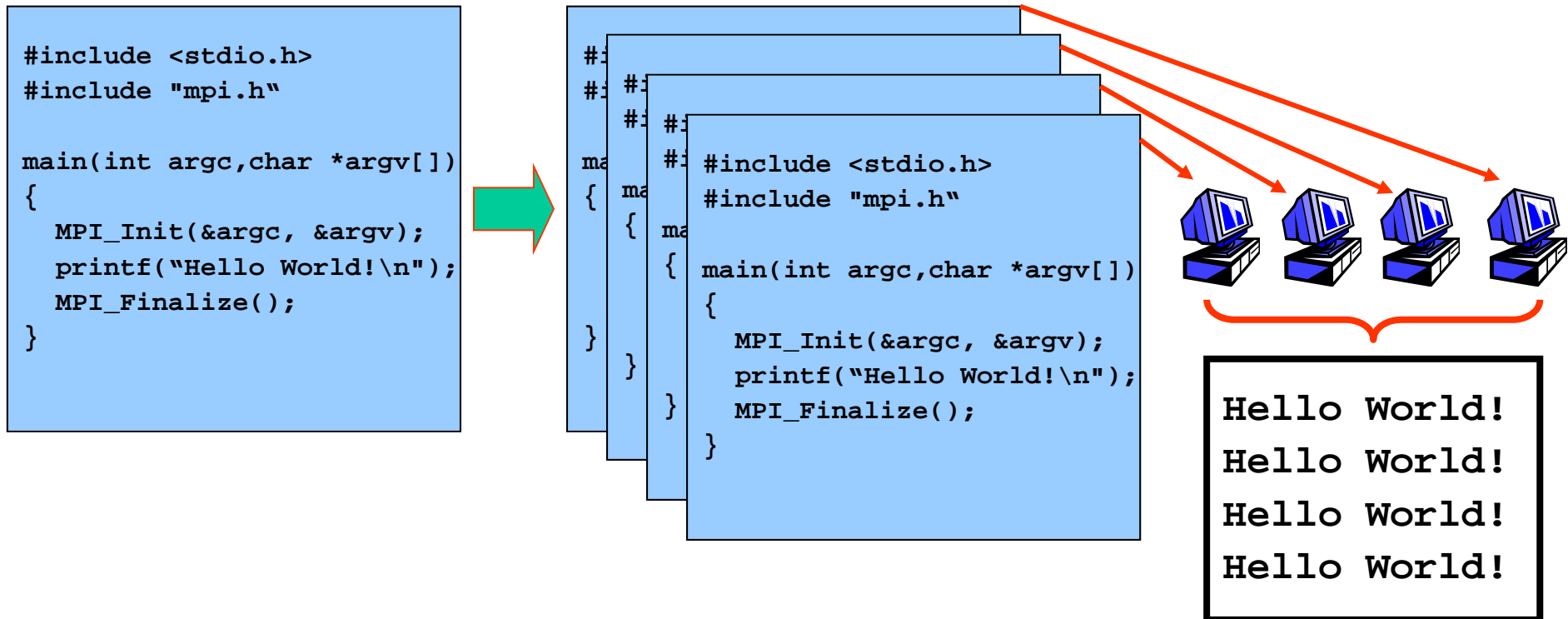
MPI 程序的编译和运行

- `mpicc -O2 -o hello hello.c`
 - 生成hello的可执行代码
- `mpirun -np 4 hello`
 - 4, 指定np的值,表示进程数,由用户指定
 - hello, 要运行的MPI并程序



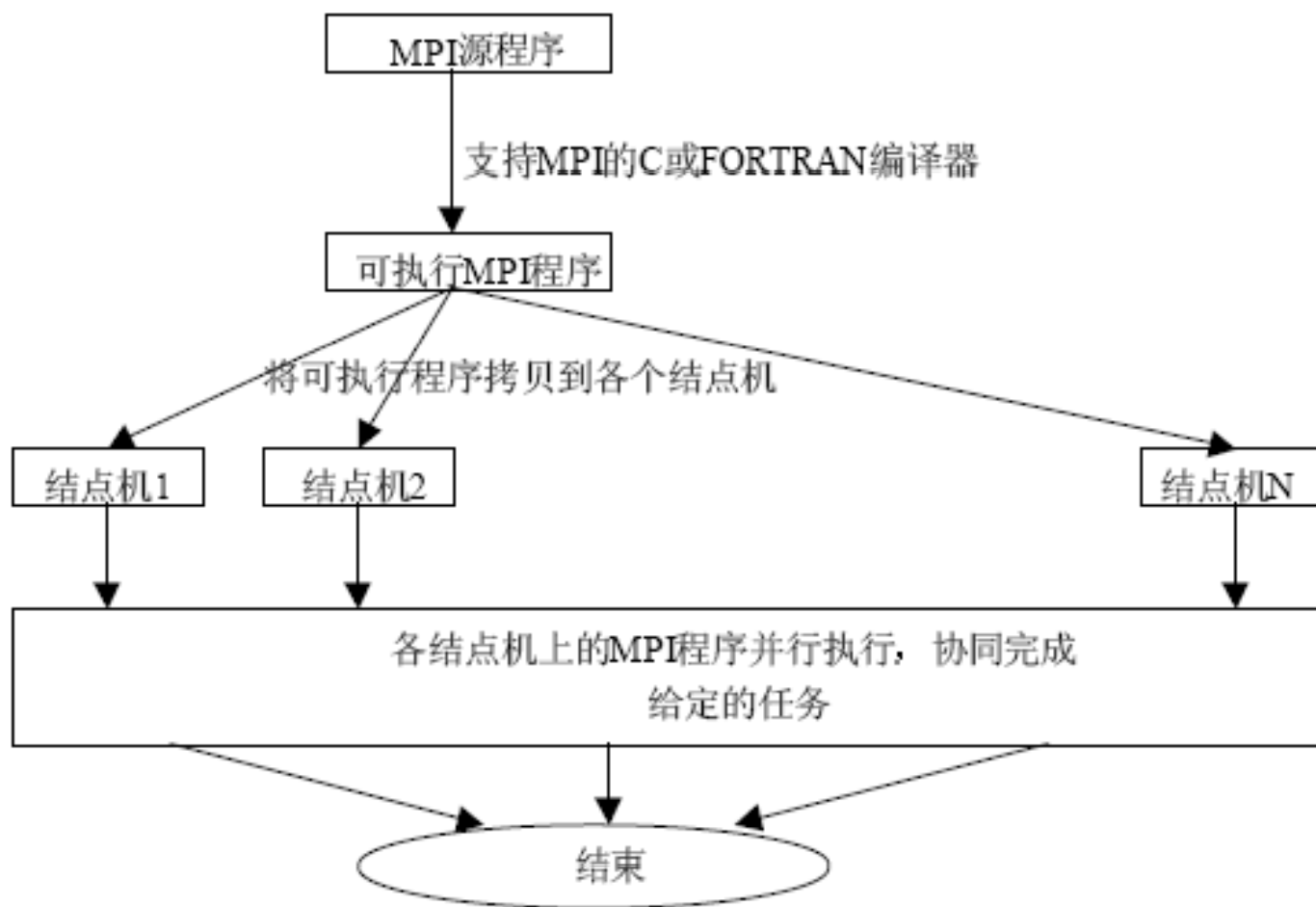
Hello是如何被执行的?

- SPMD: Single Program Multiple Data(MIMD)





MPI 程序运行模式





提 纲

- 引言
- 认识MPI编程
- MPI编程简介
- 实例



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，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_INIT`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)



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成功代码MPI_SUCCESS
 - 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，例：MPI_Init
- Fortran
 - 必须包含mpif.h
 - 通过子函数形式调用MPI，函数最后一个值为返回值
 - MPI-前缀，且函数名全部大写，例：MPI_INIT



开始写MPI程序

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



开始写MPI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语言)

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

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Hello World!Process %d of %d on
    %s\n",myid,numprocs,processor_name);

    MPI_Finalize();
}
```



更新后程序的运行结果

```
mpirun -np 4 ./hellompi
```

```
Hello World!Process 0 of 4 on blade01.ssc
```

```
Hello World!Process 1 of 4 on blade01.ssc
```

```
Hello World!Process 2 of 4 on blade01.ssc
```

```
Hello World!Process 3 of 4 on blade01.ssc
```



问题

- 有 p 个进程，除0号进程以外的所有进程都向0号进程发送消息“hello”，0号进程接收来自其他各个进程的问候。



有消息传递greetings(C语言)

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

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

```
if (myid != 0) {
    sprintf(message, "Hello! From process %d",myid);
    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 */
```



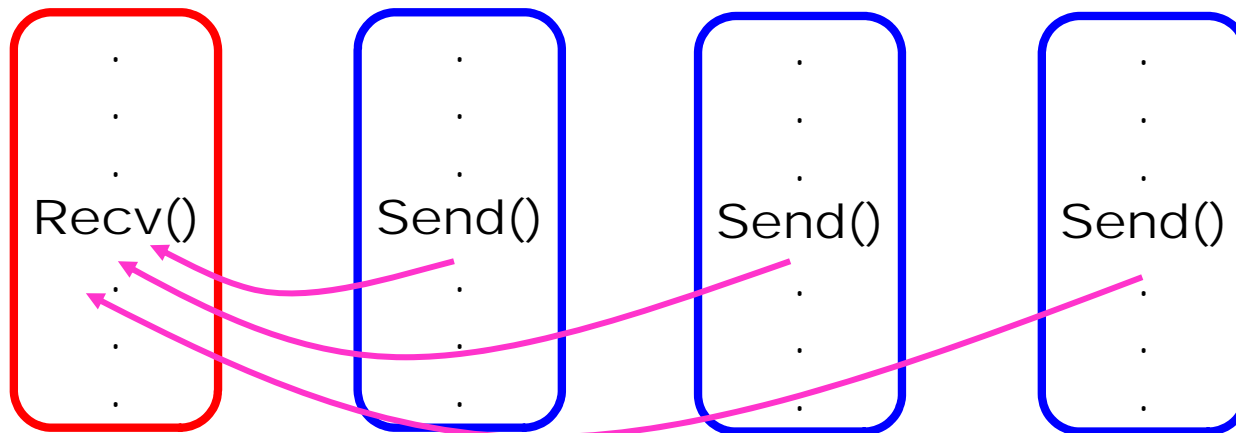
Greeting 执行过程

进程 0
rank=0

进程 1
rank=1

进程 2
rank=2

进程 3
rank=3





解剖greeting程序

- 头文件: `mpi.h/mpif.h`
- `int MPI_Init(int *argc, char ***argv)`
- 通信组/通信子: `MPI_COMM_WORLD`
 - 一个通信组是一个进程组的集合。所有参与并行计算的进程可以组合为一个或多个通信组
 - 执行MPI_Init后, 一个MPI程序的所有进程形成一个缺省的组, 这个组被写作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greeting程序

- `int MPI_Comm_size (MPI_Comm comm, int *size)`
 - 获得通信组comm中包含的进程数
- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`
 - 得到本进程在通信组中的rank值,即在组中的逻辑编号(从0开始)
- `int MPI_Finalize()`



消息传送

```
MPI_Send(A, 10, MPI_DOUBLE, 1, 99, MPI_COMM_WORLD);  
MPI_Recv(B, 20, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD, &status);
```

- 数据传送 + 同步操作
- 需要发送方和接受方合作完成

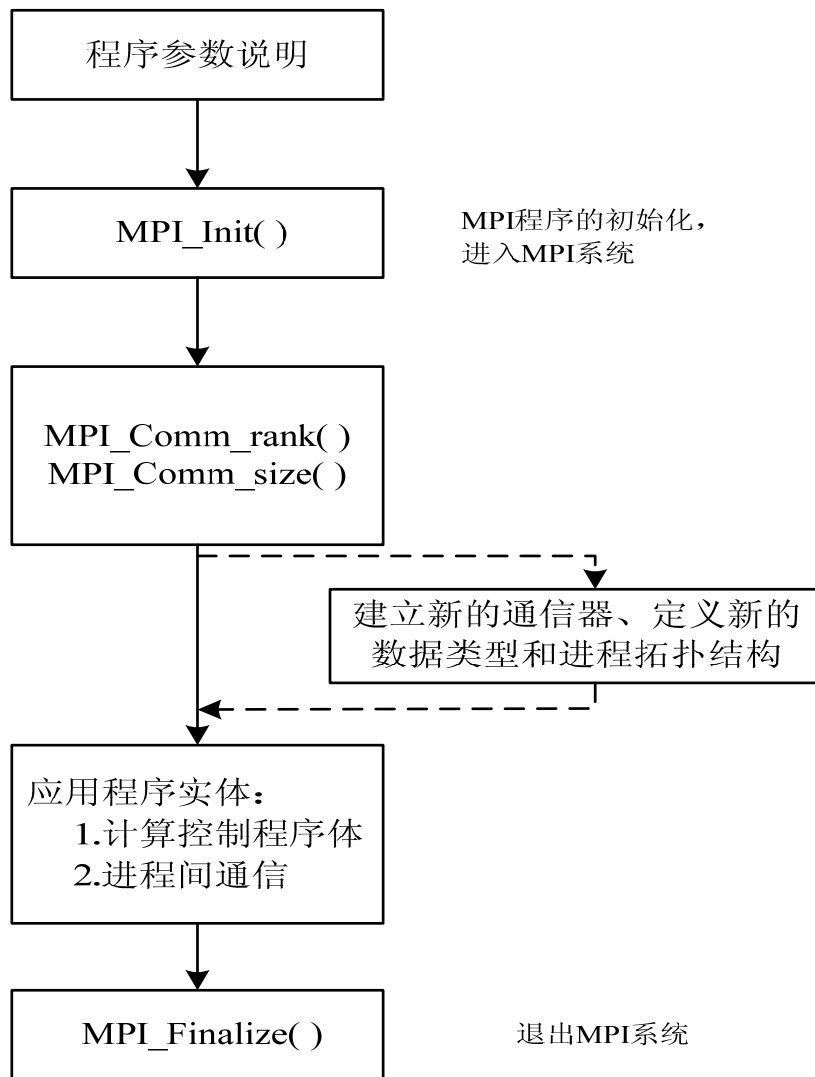


最基本的MPI程序

- MPI函数的总数虽然庞大，但根据实际编写MPI的经验，常用的MPI调用的个数确实有限。
- 下面是6个最基本也是最常用的MPI函数
 - `MPI_Init(...)`;
 - `MPI_Comm_size(...)`
 - `MPI_Comm_rank(...)`
 - `MPI_Send(...)`
 - `MPI_Recv(...)`
 - `MPI_Finalize()`



MPI 程序设计流程





点对点通信 (P to P)

- 单个进程对单个进程的通信,重要且复杂
- 术语
 - Blocking(阻塞) : 一个函数须等待操作完成才返回,返回后用户可以重新使用所占用的资源
 - Non-blocking(非阻塞): 一个函数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



阻塞发送 (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 通信组

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。



阻塞接受 (Blocking Receive)

- `int 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
 - 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
 - IN source 接收数据的来源
 - IN tag 消息标签
 - IN comm 本进程和发送进程所在的通信域
 - OUT status status对象, 包含实际接收到的消息的有关信息



MPI 数据传送

- 数据类型相匹配
- Send和recv匹配，避免死锁
- 消息标签匹配



消息信封 (Message Envelope)

- 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: $MPI_TAG_UB \geq 32767$
 - 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



为什么使用消息标签(Tag)?

为了说明为什么要用标签,我们先来看右面一段没有使用标签的代码:

这段代码打算传送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:

```
send(A,32,Q)
send(B,16,Q)
```

Process Q:

```
recv(X, 32, P)
recv(Y, 16, P)
```

使用了标签

Process P:

```
send(A,32,Q,tag1)
send(B,16,Q,tag2)
```

Process Q:

```
recv (X, 32, P, tag1)
recv (Y, 16, P, tag2)
```

- 使用标志可将本次发送的消息与本进程向同一目的进程发送的其他消息区别开来。



在消息传递中使用标签

Process P:

```
send (request1,32, Q)
```

Process R:

```
send (request2, 32, Q)
```

Process Q:

```
while (true) {  
    rcv (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){  
    rcv(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;  
}
```



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, 否则会导致死锁
- 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



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?
 - 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程序

```
#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程序

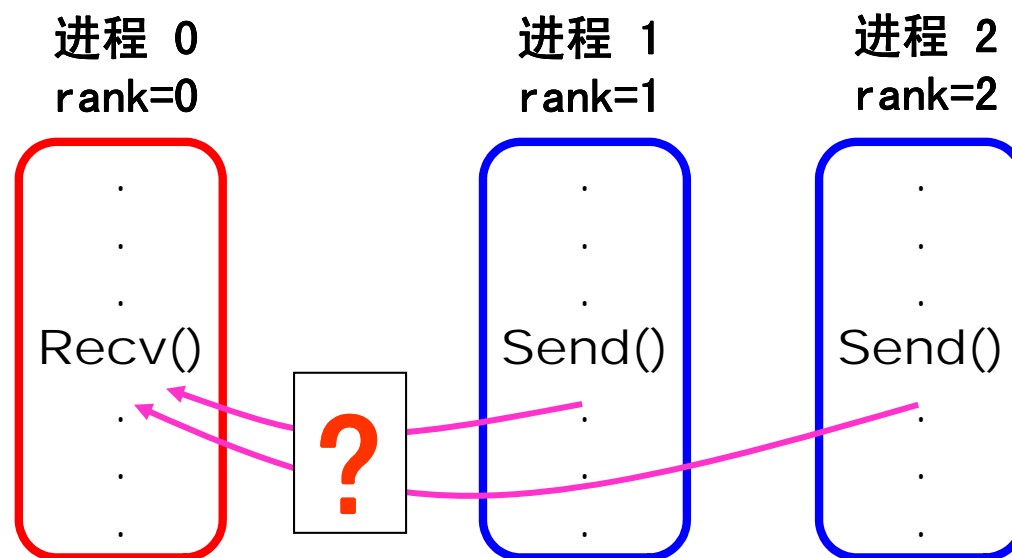
```
if (myid != 0)
{
    /*建立消息*/
    sprintf(message, "Hello!");
    /* 发送长度取strlen(message)+1,使\0也一同发送出去*/
    MPI_Send(message,strlen(message)+1, MPI_CHAR, 0,99,MPI_COMM_WORLD);
}
else
{ /* my_rank == 0 */
    for (source = 1; source < numprocs; source++)
    {
        MPI_Recv(message, 100, MPI_CHAR, 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和进程2谁先向进程0发送消息？

执行结果如何？



分析greetings程序

```
source=0;
//char buffer[10];

if (myid != 0) {
    sprintf(message, "Hello");
    MPI_Send(message,strlen(message)+1,MPI_CHAR,0,99,MPI_COMM_WORLD);
}
else {/* myid == 0 */
    do
    {
        MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, 99,MPI_COMM_WORLD,
        &status);
        source++;
        printf("%s from process %d \n",message,status.MPI_SOURCE);
    }while(source<numprocs-1);
}
MPI_Finalize();
```

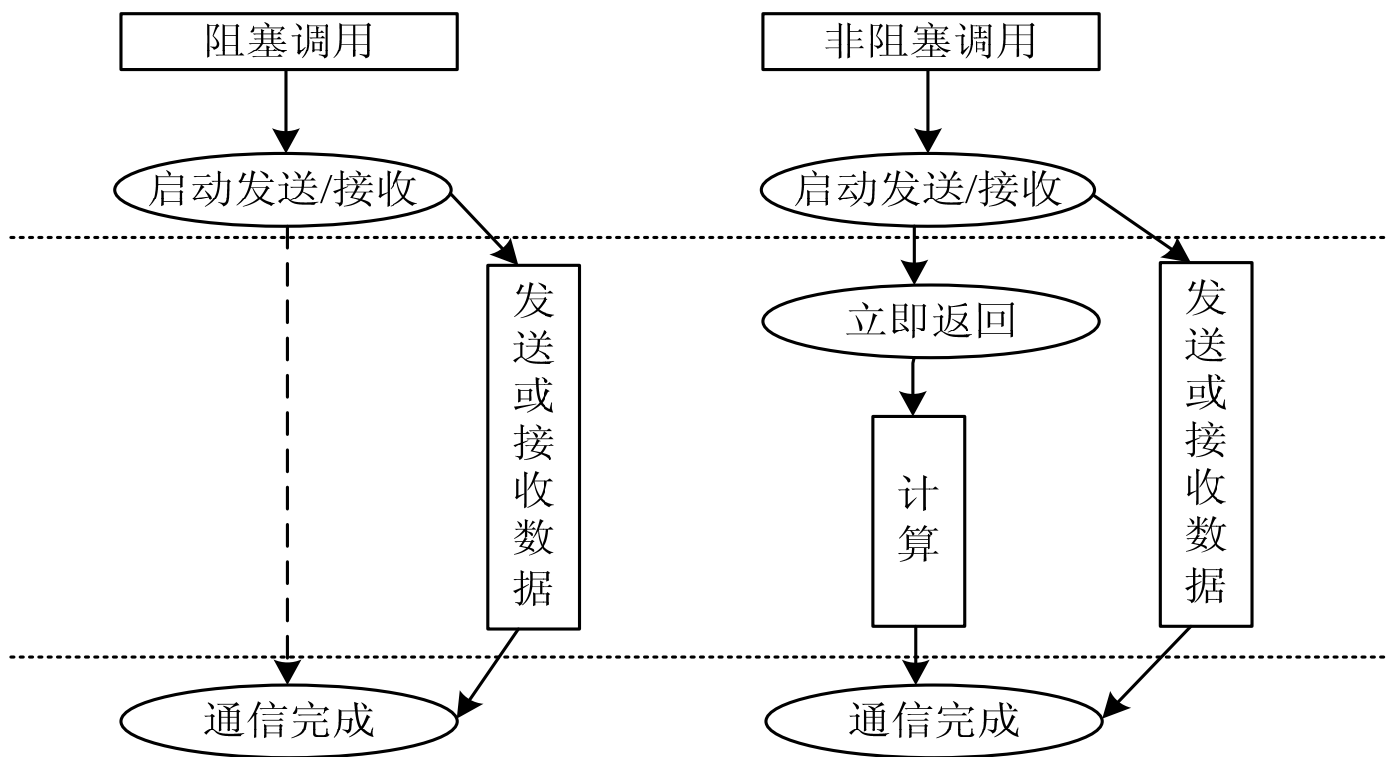


非阻塞发送与接收

- `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)
- `int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request* request)`



阻塞通信与非阻塞通信





MPI 数据类型

```
.....  
if (my_rank != 0)  
{  
    /*建立消息*/  
    sprintf(message, "Hello!");  
    /* 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

- 异构计算: 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↵	<u>signed char</u> ↵	MPI_CHARACTER↵	<u>CHARACTER(1)</u> ↵
↵	↵	MPI_COMPLEX↵	COMPLEX↵
MPI_DOUBLE↵	<u>double</u> ↵	MPI_DOUBLE_ PRECISION↵	DOUBLE_ PRECISION↵
MPI_FLOAT↵	<u>float</u> ↵	MPI_REAL↵	REAL↵
MPI_INT↵	<u>int</u> ↵	MPI_INTEGER↵	INTEGER↵
↵	↵	MPI_LOGICAL↵	LOGICAL↵
MPI_LONG↵	<u>long</u> ↵	↵	↵
MPI_LONG_DOUBLE↵	<u>long double</u> ↵	↵	↵
MPI_PACKED↵	↵	MPI_PACKED↵	↵
MPI_SHORT↵	<u>short</u> ↵	↵	↵
MPI_UNSIGNED_CHAR↵	<u>unsigned char</u> ↵	↵	↵
MPI_UNSIGNED↵	<u>unsigned int</u> ↵	↵	↵
MPI_UNSIGNED_LONG↵	<u>unsigned long</u> ↵	↵	↵
MPI_UNSIGNED_SHORT↵	<u>unsigned short</u> ↵	↵	↵



数据类型匹配规则

- 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；
- 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；
- 打包数据的通信，发送方和接收方均使用MPI_PACKED。



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



MPI集合通信函数

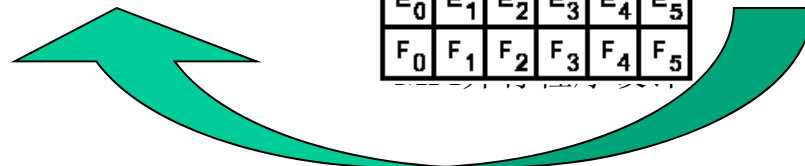
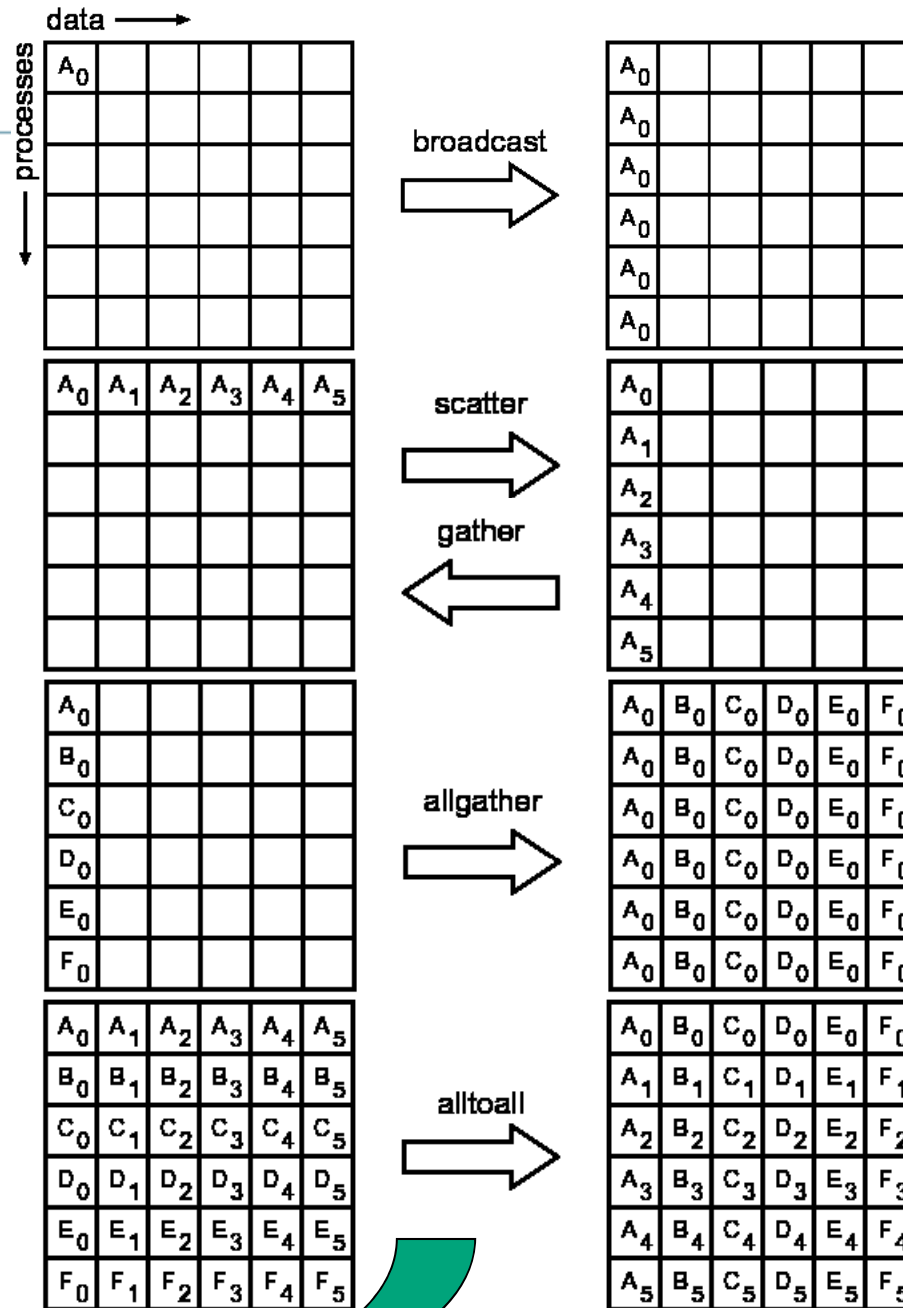
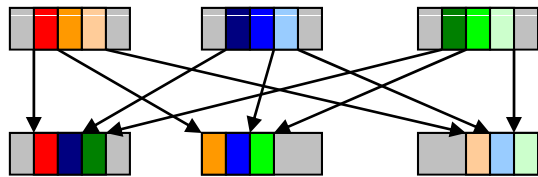
-All:表示结果到**所有**进程.

-V:Variety,被操作的数据对象和操作更为灵活.

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(全交换)
	MPI_Alltoallv	MPI_Alltoall的一般形式
数据聚集	MPI_Reduce	多到一, 数据归约
	MPI_Allreduce	MPI_Reduce的一般形式, 结果在所有进程
	MPI_Reduce_scatter	结果scatter到每个进程
	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

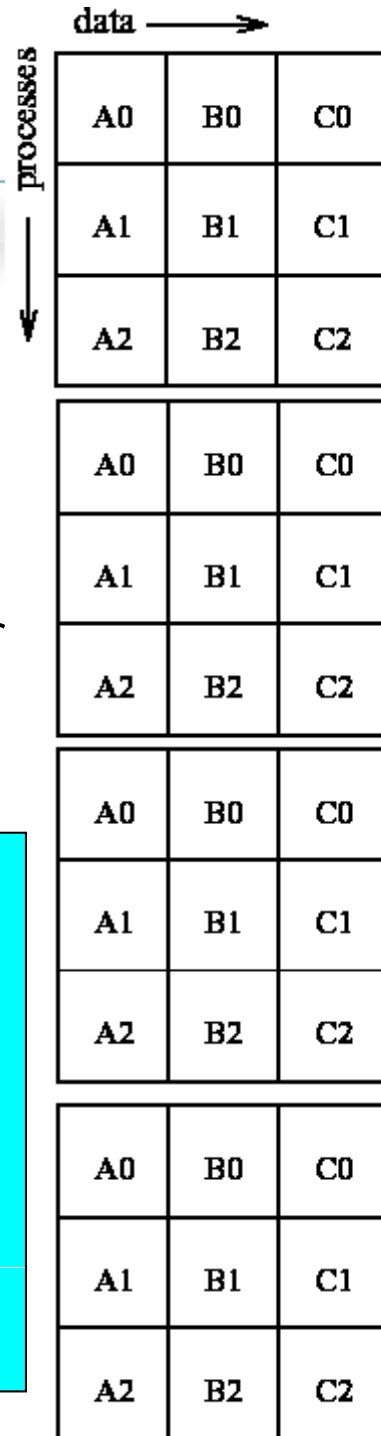
MPI_LAND 逻辑与

MPI_LOR 逻辑或

MPI_MAXLOC/MPI_MINL

OC 最大/小值求下相应位置

... ..



reduce →

A0+A1+A2	B0+B1+B2	C0+C1+C2

allreduce →

A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2

reduce-scatter →

A0+A1+A2		
B0+B1+B2		
C0+C1+C2		

scan →

A0	B0	C0
A0+A1	B0+B1	C0+C1
A0+A1+A2	B0+B1+B2	C0+C1+C2

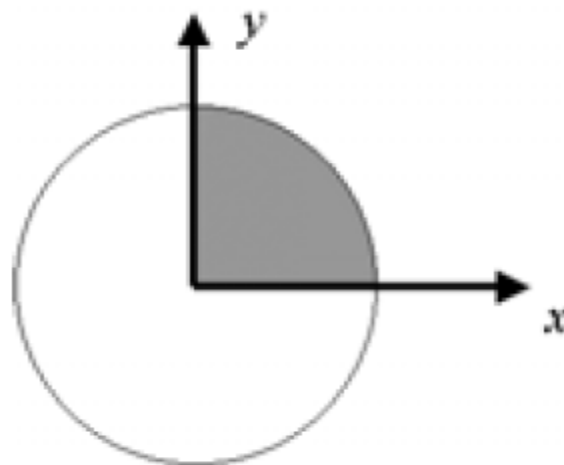
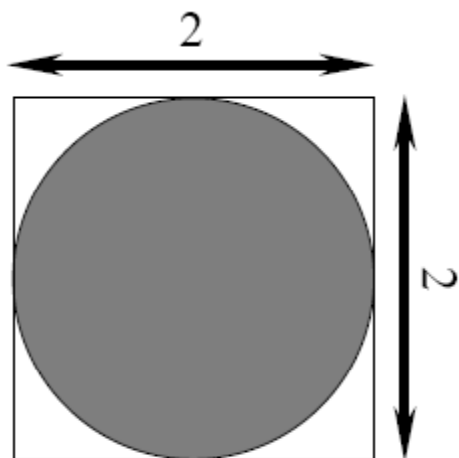


提 纲

- 引言
- 认识MPI编程
- MPI编程简介
- 实例



实例分析：求pi



$$\int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$



$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

计算 π 值. 令 $f(x) = 4/(1+x^2)$. 将区间 $[0, 1]$ 分成 n 等分, 并令 $x_i = (i - 0.5)/n$, 则:

$$\pi \approx \frac{1}{n} \sum_{i=1}^n f(x_i)$$

假设并行程序中共有 P 个进程参与计算, 则第 k 个进程负责计算:

$$\sum_{1 \leq i \leq n, (i-1) \bmod P = k} f(x_i)$$



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));  
}
```

$$\int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$



并行代码

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
long  n,          /*number of slices */
      i;          /* slice counter */
double sum,       /* running sum */
       pi,        /* approximate value of pi */
       mypi,
       x,         /* independent var. */
       h;        /* base of slice */
int group_size,my_rank;

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

```

{  int group_size,my_rank;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size( MPI_COMM_WORLD, &group_size);

    n=2000;
    /* Broadcast n to all other nodes */
    MPI_Bcast(&n,1,MPI_LONG,0,MPI_COMM_WORLD);
    h = 1.0/(double) n;
    sum = 0.0;
    for (i = my_rank+1; i <= n; i += group_size) {
        x = h*(i-0.5);
        sum = sum +4.0/(1.0+x*x);
    }
    mypi = h*sum;
    /*Global sum */
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    if(my_rank==0) { /* Node 0 handles output */
        printf("pi is approximately : %.16lf\n",pi);
    }
    MPI_Finalize();
}

```



参考资料

- Kai Hwang, Zhiwei Xu, 《可扩展并行计算：技术、结构与编程》
- Michael J. Quinn, 《MPI与OpenMP并行程序设计》
- 陈国良, 中国科技大学, 《并行计算 - 结构、算法和编程》
- 都志辉, 清华大学, 《MPI并行程序设计》
- 迟学斌, 中科院网络中心, 《高性能并行计算》课程讲义
- 张林波, 中科院计算数学所, 《并行计算》课程讲义
- 安虹, 中科大计算机系, 《并行计算》课程讲义
- 曹振南, 曙光公司, 《MPI并行程序设计讲义》



谢 谢



附录

- **隐式并行模型**

程序员未作明确地指定并行性，而让编译器和运行（时）支持系统自动地开拓它

- **显式并行模型**

程序的并行性由程序员利用专门的语言结构，编译知道和库函数调用等在源代码中给予明显的制定



什么是缓冲区?

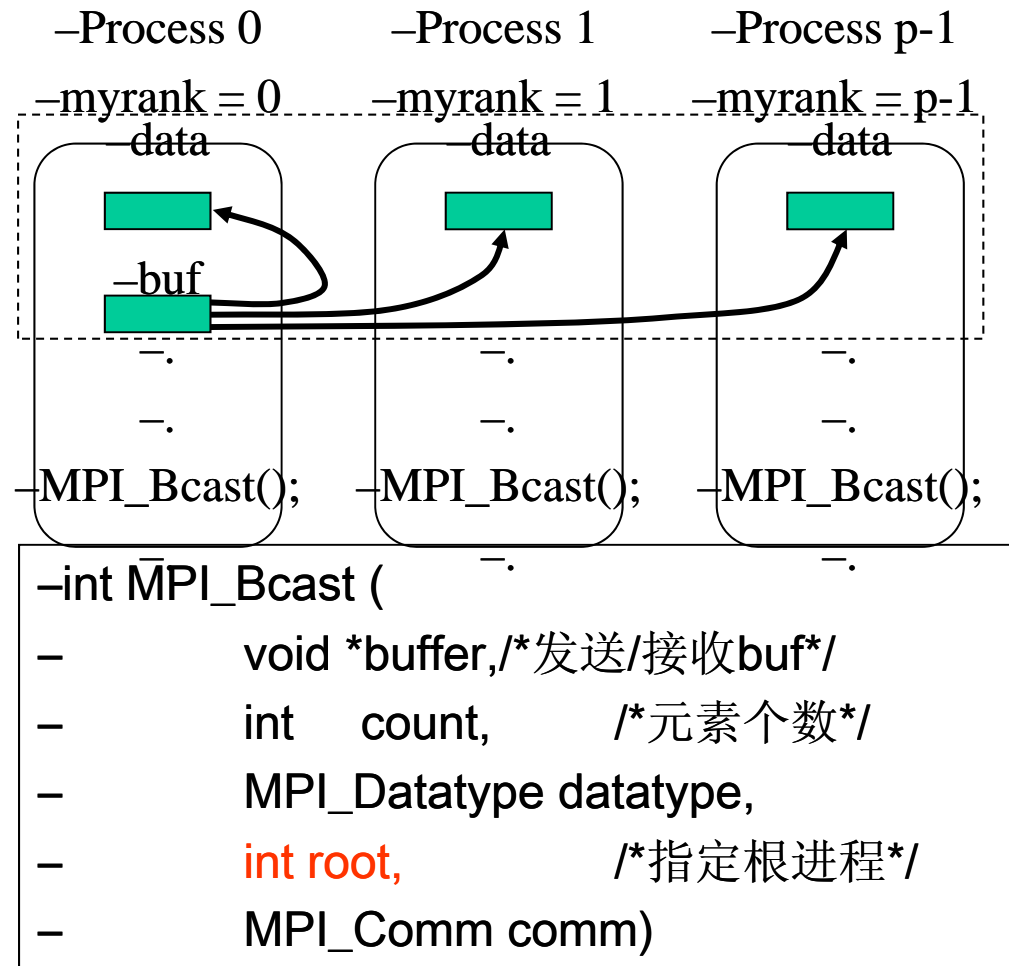
- 应用程序中说明的变量, 在消息传递语句中又用作缓冲区的起始位置
- 也可表示由系统创建和管理的某一存储区域, 在消息传递过程中用于暂存放消息. 也被称为系统缓冲区
- 用户可设置一定大小的存储区域, 用作中间缓冲区以保留可能出现在其应用程序中的任意消息.



Broadcast -- 数据广播

```

int p, myrank;
float 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);
if(myrank == 0)
    buf = 1.0;
MPI_Bcast(&buf, 1, MPI_FLOAT,
          0, comm);
    
```

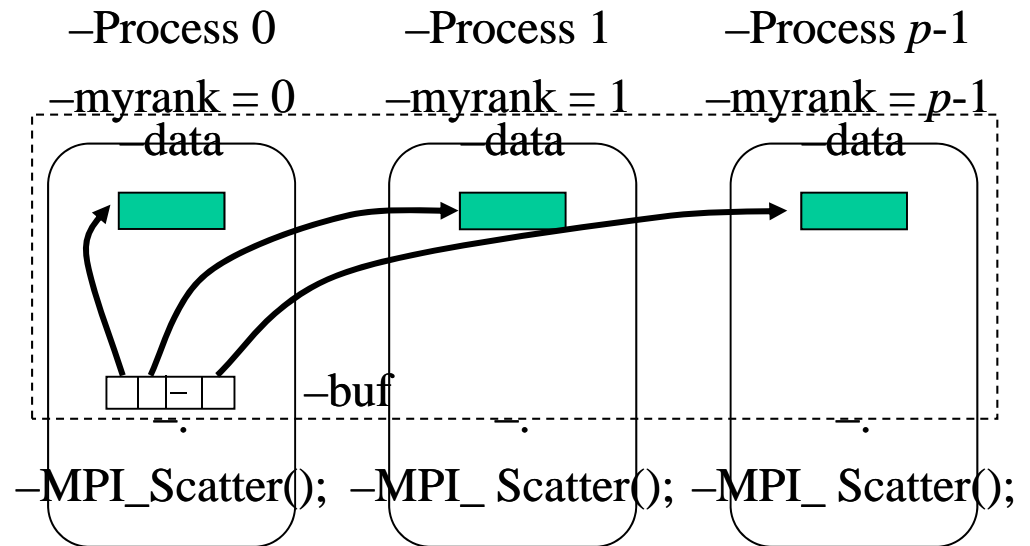


-根进程既是发送缓冲区也是接收缓冲区



Scatter -- 数据分散

```
int p, myrank;  
float data[10];  
float* buf;  
MPI_Comm comm;  
MPI_Init(&argc, &argv);  
/*得进程编号*/  
MPI_Comm_rank(comm, &myrank);  
/*得进程总数*/  
MPI_Comm_size(comm, &p);  
if(myrank == 0)  
    buf =  
    (float*)malloc(p*10*sizeof(float)); /*开辟发送缓冲区*/  
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)

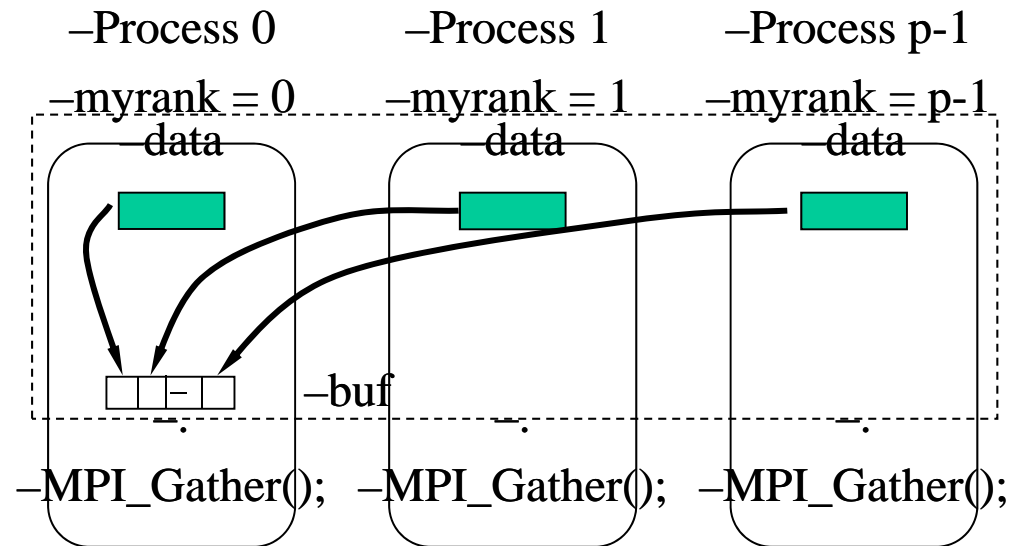


Gather -- 数据收集

```

int p, myrank;
float data[10]; /* 分布变量 */
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/* 得进程编号 */
MPI_Comm_rank(comm, &myrank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
if(myrank == 0)
    buf = (float*) malloc(p * 10 * sizeof(float)); /* 开辟接收缓冲区 */

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



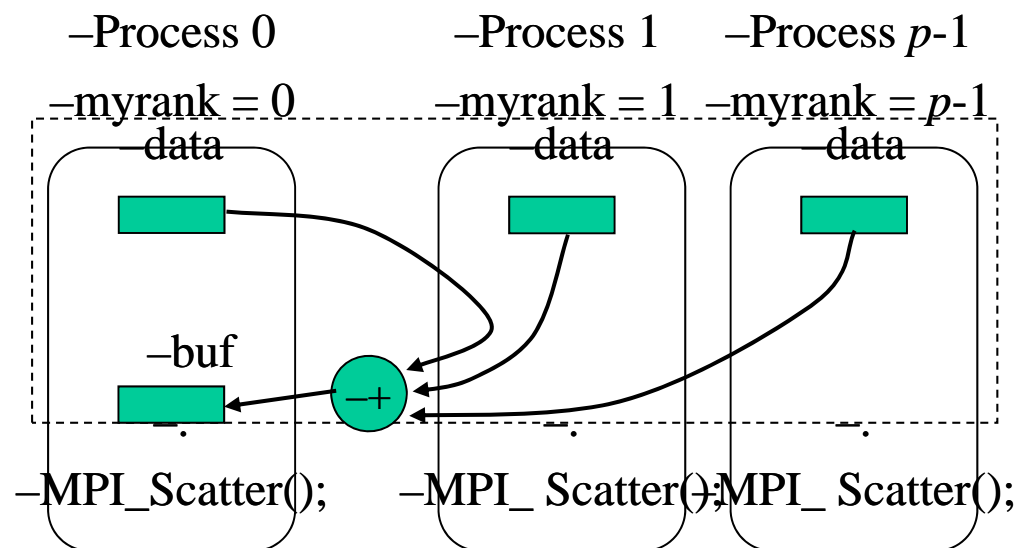
–根进程接收其他进程来的消息(包括根进程),按每在进程在通信组中的编号依次联接在一下,存放在根进程接收缓冲区中。

–int MPI_Gather (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

```
int p, myrank;  
float data = 0.0;  
float buf;  
MPI_Comm comm;  
MPI_Init(&argc, &argv);  
/* 得进程编号 */  
MPI_Comm_rank(comm, &myrank);  
  
/* 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_FLOAT, 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 )
```



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 recvcnt, 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
- **recvcnt** 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

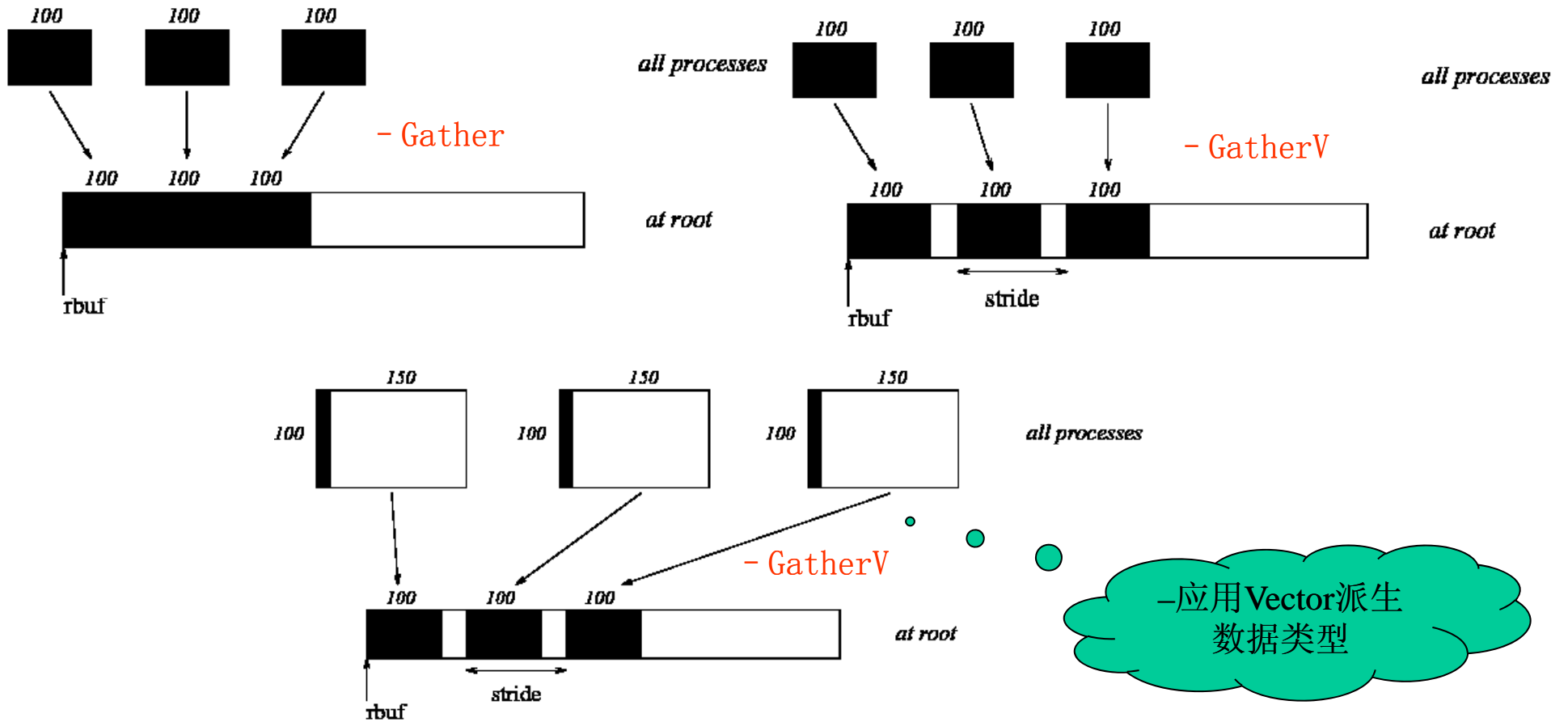


MPI_Gatherv

- `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
- `recvcnts` 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