
基本的MPI编程技术

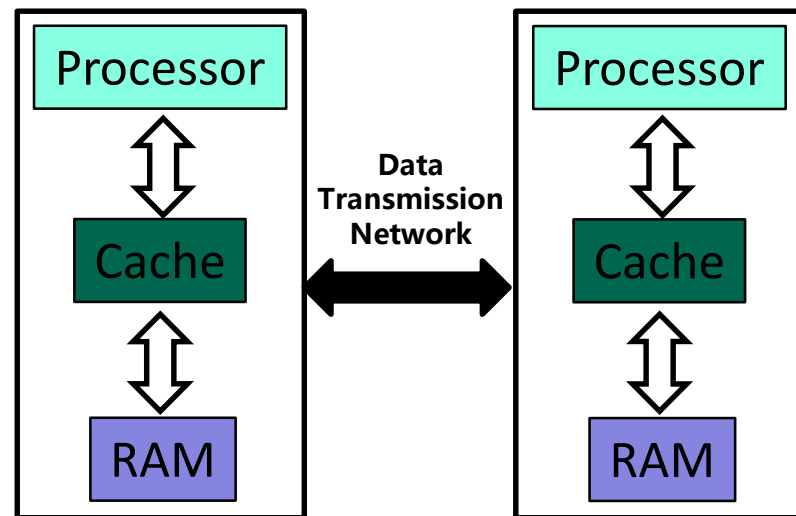
Lecture 08: MPI编程——点到点通信

肖俊敏

中国科学院计算技术研究所

分布式处理的简介

- 具有分布式存储器的计算机系统
系统中的处理器是独立运行的
- 有可能性是必要的:
 - 为了分配计算负载
 - 为了组织处理机之间的信息通信
(数据传输)



MPI (message passing interface)
为上述问题提供了解决方案。

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**、**C**和**C++**
- 一种新的库描述，不是一种语言

<https://mpitutorial.com/tutorials/>

消息传递并程序序设计

■ 消息传递并程序序设计

- 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。
- 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程序序设计方式

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
```

编译、安装MPI

- `cd /home/yourname`
- `tar -xvzf openmpi-1.6.3.tar.gz`
- `cd openmpi-1.6.3`
- `./configure --prefix=/opt/openmpi`
- `make`
- `sudo make install`
- **`vi ~/.bashrc` 末尾添加两行**
- **`export PATH=/opt/openmpi/bin:$PATH`**
- **`export LD_LIBRARY_PATH=/opt/openmpi/lib:$LD_LIBRARY_PATH`**

MPI程序的编译与运行

■ **mpif77 -o hello hello.f** 或

■ **mpicc -o hello hello.c**

- 生成hello的可执行代码.

 小写o

■ **mpirun -np 4 -hostfile nodes 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并程序序.
- nodes 文件的每一行是结点主机名

 np: The number of process.

🔄:Hello是如何被执行的?

■ SPMD: Single Program Multiple Data(MIMD)

程序

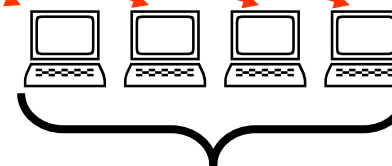
```
#include "mpi.h"
#include <stdio.h>

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



可执行文件

```
#include "mpi.h"
# #include "mpi.h"
# #include "mpi.h"
m # #include "mpi.h"
m #include <stdio.h>
m
{
  {
    main(
      int argc,
      char *argv[] )
    {
      MPI_Init( &argc, &argv );
      printf( "Hello, world!\n" );
      MPI_Finalize();
    }
  }
}
```



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

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.

🔧: 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？

田: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);

MPI基本函数

■ **int MPI_Init (**
 int*
 char** **argc** **/* in/out */**
 argv[] **/* in/out */**)

- 通常应该是第一个被调用的MPI函数
- 除MPI_Initialized()外，其余所有的MPI函数应该在其后被调用
- MPI系统将通过argc, argv得到命令行参数

MPI基本函数

■ **int MPI_Finalize (void)**

- 退出MPI系统，所有进程正常退出都必须调用。表明并行代码的结束,结束除主进程外其它进程.
- 串行代码仍可在主进程(rank = 0)上运行，但不能再有MPI函数（包括**MPI_Init()**）

MPI基本函数

- `int MPI_Comm_size (MPI_Comm comm, int* size) /* in */ /* out */`
 - 获得进程个数 size
 - 指定一个通信子,也指定了一组共享该空间的进程, 这些进程组成该通信子的group.
 - 获得通信子comm中规定的group包含的进程的数量.

MPI基本函数

- `int MPI_Comm_rank (MPI_Comm comm /* in */, int* rank /* out */)`
 - 得到本进程在通信空间中的rank值,即在组中的逻辑编号(该rank值为0到p-1间的整数,相当于进程的ID。)

MPI 程序的基本结构

```
#include "mpi.h"

... ..

int main(int argc, char *argv[])
{
    int myrank, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    ... ..
    MPI_Finalize();
}
```

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

- `mpicc -o hello1 hello1.c`
- `mpirun -np 4 hello1`

I am 0 of 4

I am 1 of 4

I am 2 of 4

I am 3 of 4

有消息传递greetings(c)

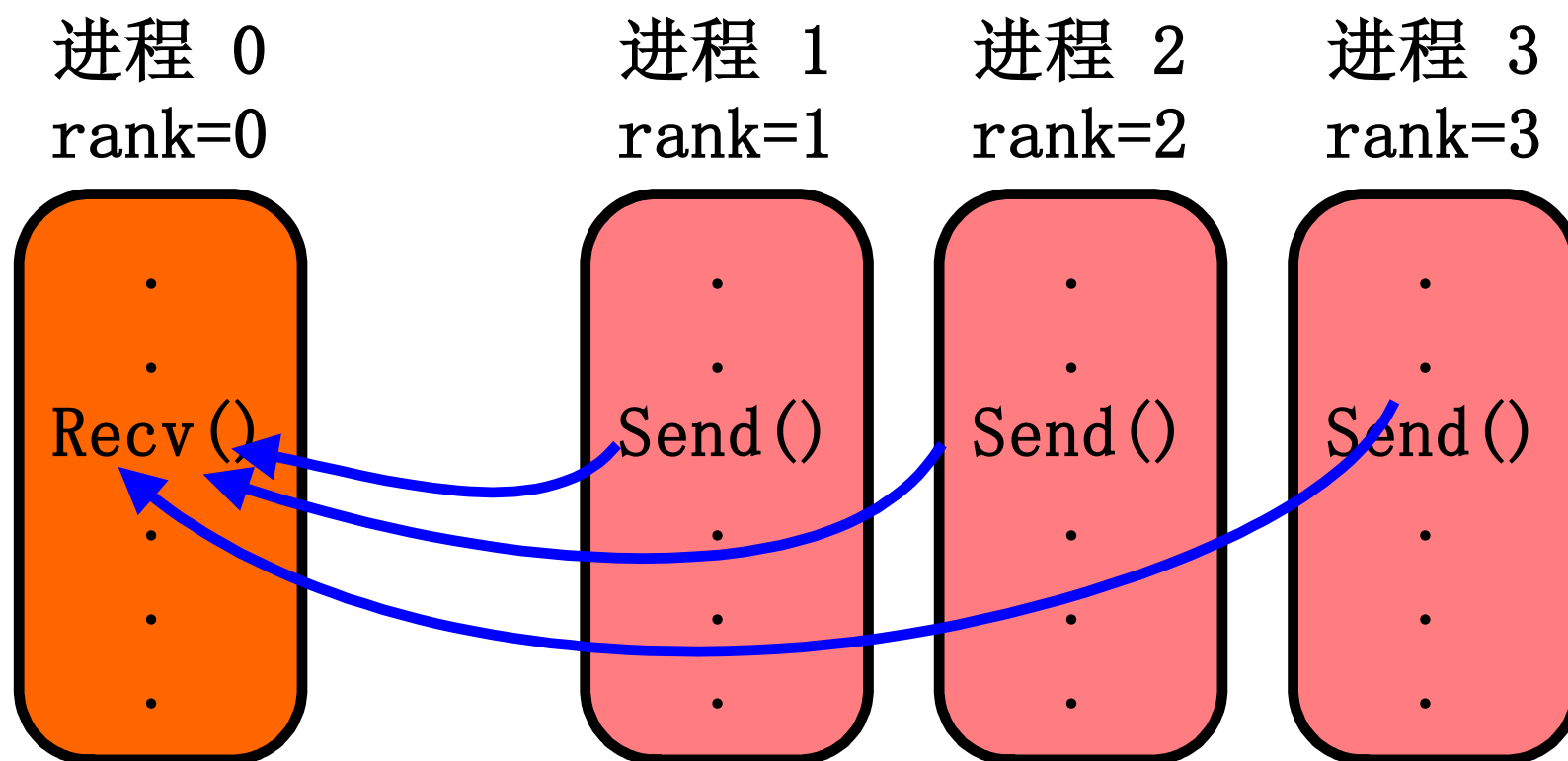
```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[])
{
    int numprocs, myid, source;
    MPI_Status status;
    char message[100];

    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/mpi.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默认的组,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

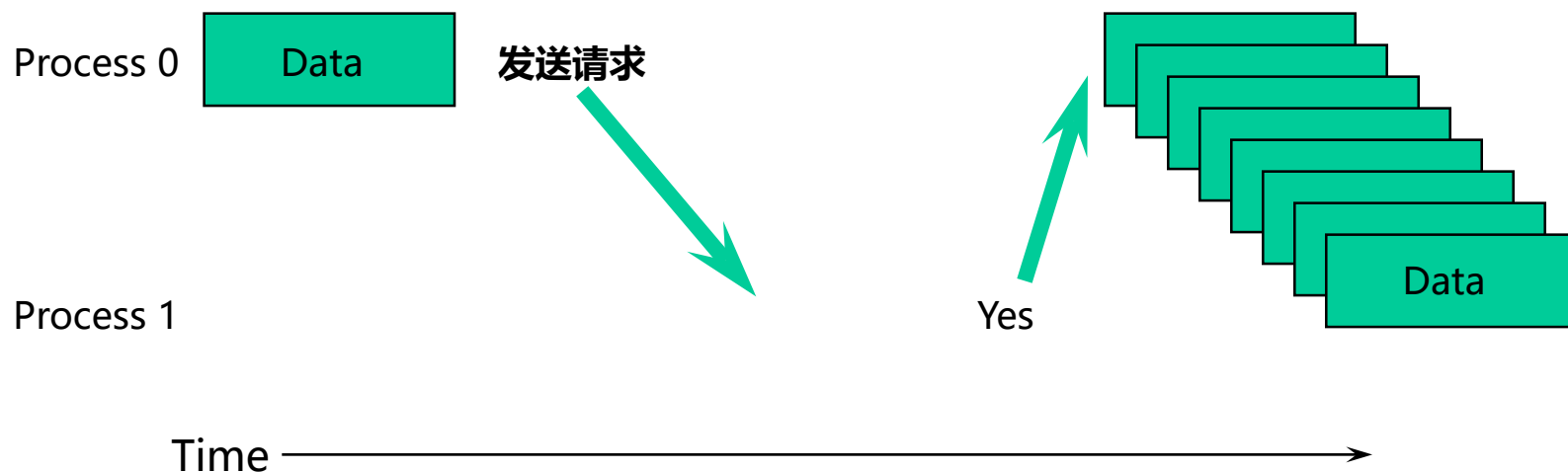
- **int MPI_Comm_size (MPI_Comm comm, int *size)**
 - 获得通信空间comm中规定的组包含的进程的数量.
 - 指定一个communicator,也指定了一组共享该空间的进程, 这些进程组成该communicator的group.
- **int MPI_Comm_rank (MPI_Comm comm, int *rank)**
 - 得到本进程在通信空间中的rank值,即在组中的逻辑编号(从0开始).
- **int MPI_Finalize()**
 - 标志并行代码的结束,结束除主进程外其它进程.
 - 之后串行代码仍可在主进程(rank = 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_DOUBLE, 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调用函数的总数虽然庞大，但根据实际编写MPI的经验，常用MPI函数的个数却十分有限。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_Finalize();  
只能有串行代码;
```

几个有用的函数 —— 查询处理器名称


■ `int MPI_Get_processor_name(char *name, int *resultlen)`

- 该函数在name 中返回进程所在的处理器的名称. 参数name 应该提供不少于MPI_MAX_PROCESSOR_NAME 个字节的存储空间用于存放处理器名称。

几个有用的函数——获取墙上时间及时钟精度

- **double MPI_Wtime(void)**
- **double MPI_Wtick(void)**
 - MPI_Wtime 返回当前墙上时间, 以从某一固定时刻算起的秒数为单位. 而MPI_Wtick 则返回MPI_Wtime 函数的时钟精度, 也是以秒为单位. 例如, 假设MPI_Wtime使用的硬件时钟计数器每1/1000秒增加1, 则MPI_Wtick的返回值为 10^{-3} , 表示用函数MPI_Wtime得到的时间精度为千分之一秒.

MPI内容目录

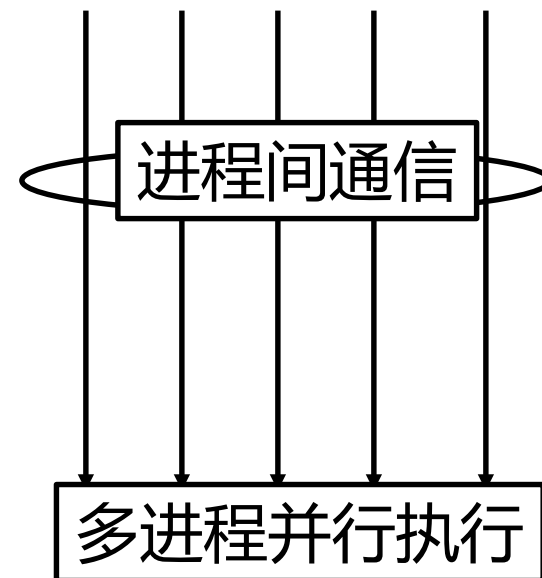
- **基本概念** 
- **点到点通信**
- **自定义数据类型**
- **集合通信**
- **虚拟拓扑**
- **文件IO**

MPI基本概念(一)

- **进程(process)**: 我们将MPI 程序中一个独立参与通信的个体称为一进程.
- **进程组(process group)**: 唯一的序号(rank), 用于在该组中标识该进程.
- **通信(communication)** : 通信指在进程之间进行消息的收发、同步等操作
- **通信子(communicator)**: 是完成进程间通信的基本环境。

MPI的进程

- MPI通信的对象
- MPI并行的单位
- MPI进程的标识
 - 区别不同的进程



一个MPI 并行程序由一组运行在相同或不同计算机/计算结点上的进程或线程构成。为统一起见, 我们将MPI 程序中一个独立参与通信的个体称为一个进程。在Unix 系统中, MPI的进程通常是一个Unix 进程。在共享内存/消息传递混合编程模式中, 一个MPI 进程可能代表一组Unix 线程。

MPI基本概念(二)

■ 序号(rank)

- 序号用来在一个进程组或通信子中标识一个进程. MPI 程序中的进程由进程组/序号或通信子/序号所唯一确定. 序号是相对于进程组或通信子而言的: 同一个进程在不同的进程组或通信子中可以有不同的序号. 进程的序号是在进程组或通信子被创建时赋予的.

MPI基本概念(三)

■ 通信子(communicator):

- 是完成进程间通信的基本环境, 它描述了一组可以互相通信的进程以及它们之间的联接关系等信息. MPI 的所有通信必须在某个通信子中进行. 通信子分域内通信子和域间通信子两类, 前者用于属于同一进程组的进程间的通信, 后者用于分属两个不同进程组的进程间的通信. 域内通信子由一个进程组和有关该进程组的进程间的拓扑联接关系构成.

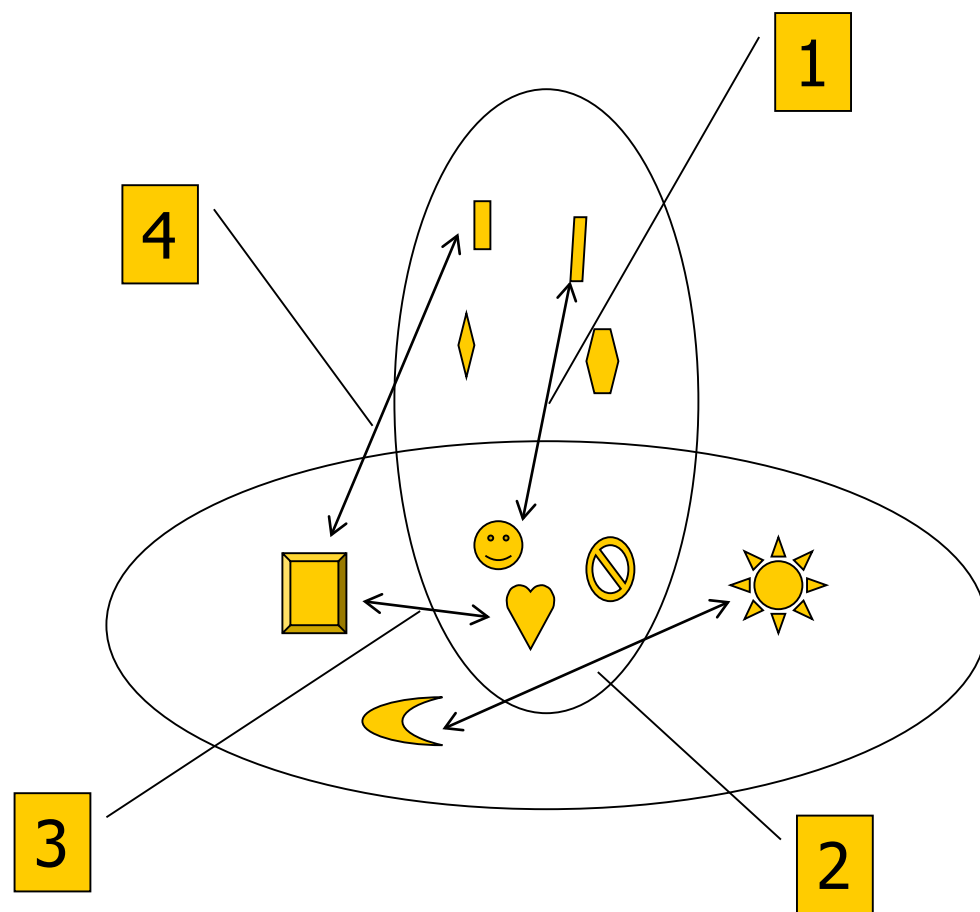
MPI的通信域

■ 组成

- 通信环境
- 通信对象

■ 预定义通信域

- `MPI_COMM_WORLD`



MPI的消息

■ 消息(message)

- MPI 程序中在进程间传送的数据称为消息. 一个消息由通信器、源地址、目的地址、消息标签和数据构成.

■ 作用

- 定义了MPI通信的方式与表示方法
- 有助于实现移植性

■ 组成

- 消息信封
- 消息内容

MPI消息目的和源

- **目的 (Destination)**
 - 消息的接收者
- **源 (Source)**
 - 消息的提供者
- **隐含目/源**
 - 组通信

MPI消息的标识TAG

■ 作用

- 区别同一进程的不同消息，使程序员以一种有序的方式处理到达的消息。
- 必要的，但不充分，因为“tag”的选择具有一定的随意性。
- MPI用一个新概念“上下文”(context)对“tag”进行扩展。系统运行时分配，不允许统配。

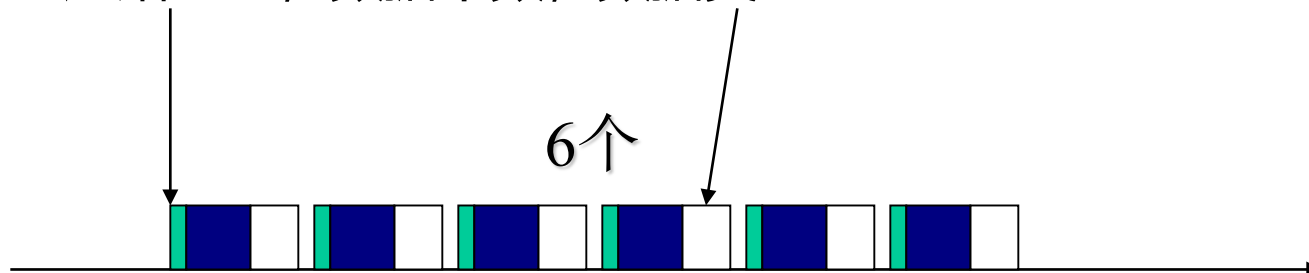
■ 隐含TAG

- 组通信，由通信语句的序列决定消息的匹配

消息内容

■ 内容 (data) , 在组通信中有变化形式

- <起始地址, 数据个数, 数据类型>



■ <起始地址, 长度>?

- 移植性
- 不连续数据的发送
- 抽象

MPI的数据类型

■ 定义

- 一个或多个，相同或不同的基本数据类型，以连续或不连续的方式形成的一种排列

■ 目的：移植，方便使用（抽象，不连续数据的传送）

■ MPI是强数据类型的

- 整型与实型匹配？



MPI 的原始数据类型

- MPI 系统中数据的发送与接收都是基于数据类型进行的. 数据类型可以是MPI 系统预定义的,称为原始数据类型,也可以是在原始数据类型的基础上自己定义的数据类型。
- MPI 为Fortran 77 和C 定义的原始的数据类型在下面两个表格中给出. 除表中列出的外, 有的MPI 实现可能还提供了更多的原始数据类型, 如MPI_INTEGER2, MPI_LONG_LONG_INT, 等等。

MPI预定义数据类型

MPI预定义数据类型	相应的C数据类型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	无对应类型
MPI_PACKED	无对应类型

MPI内容目录

- 基本概念
- 点到点通信
- 自定义数据类型
- 集合通信
- 虚拟拓扑
- 文件IO

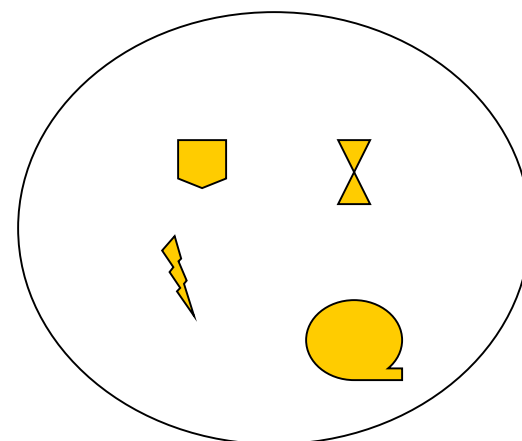


点到点通信(Point to Point)

- 有且仅有两方（发送方和接收方）参与
- 双方必须有匹配的消息信封和通信语句

组通信(Collective)

- 组内所有进程都参与才能完成
- 各进程的调用形式都相同
- 常见问题
 - 有的调用，有的不调用（主进程广播，从进程没有广播）
 - 广播语句和接收语句对应
 - 调用参数不对应



标准阻塞发送MPI_Send()

- **int MPI_Send(void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
 - tag 的取值范围为0 - MPI_TAG_UB.
 - dest 的取值范围为0 - np-1 (np 为通信器comm 中的进程数) 或 MPI_PROC_NULL.
 - count 是指定数据类型的个数, 而不是字节数.

标准阻塞接收MPI Recv ()

- **int MPI_Recv(void *buff, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**
 - count 给出接收缓冲区的大小(指定数据类型的个数), 它是接收数据长度的上界. 具体接收到的数据长度可通过调用 **MPI_Get_count** 函数得到

基本的发送与接收语句

■ MPI标识一条消息的信息包含四个域:

- **源:** 发送进程隐式确定,进程rank值唯一标识.
- **目的:** Send函数参数确定.
- **Tag:** Send函数参数确定, $(0, UB) 2^{32}-1$.
- **通信子:** 缺省MPI_COMM_WORLD
 - Group: 有限/N, 有序/Rank $[0, 1, 2, \dots, N-1]$
 - 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.

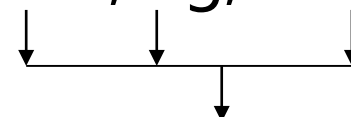
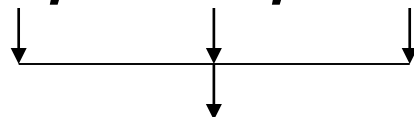
基本的发送与接收语句

- **buffer**必须至少可以容纳count个由datatype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source,tag,comm/dest,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.**

基本的发送与接收语句

■ MPI_Send

MPI_Send(**buf, count, datatype**, dest, tag, comm)

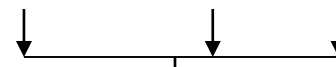
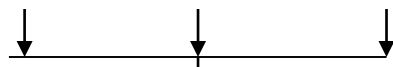


消息数据

消息信封

■ MPI_Recv

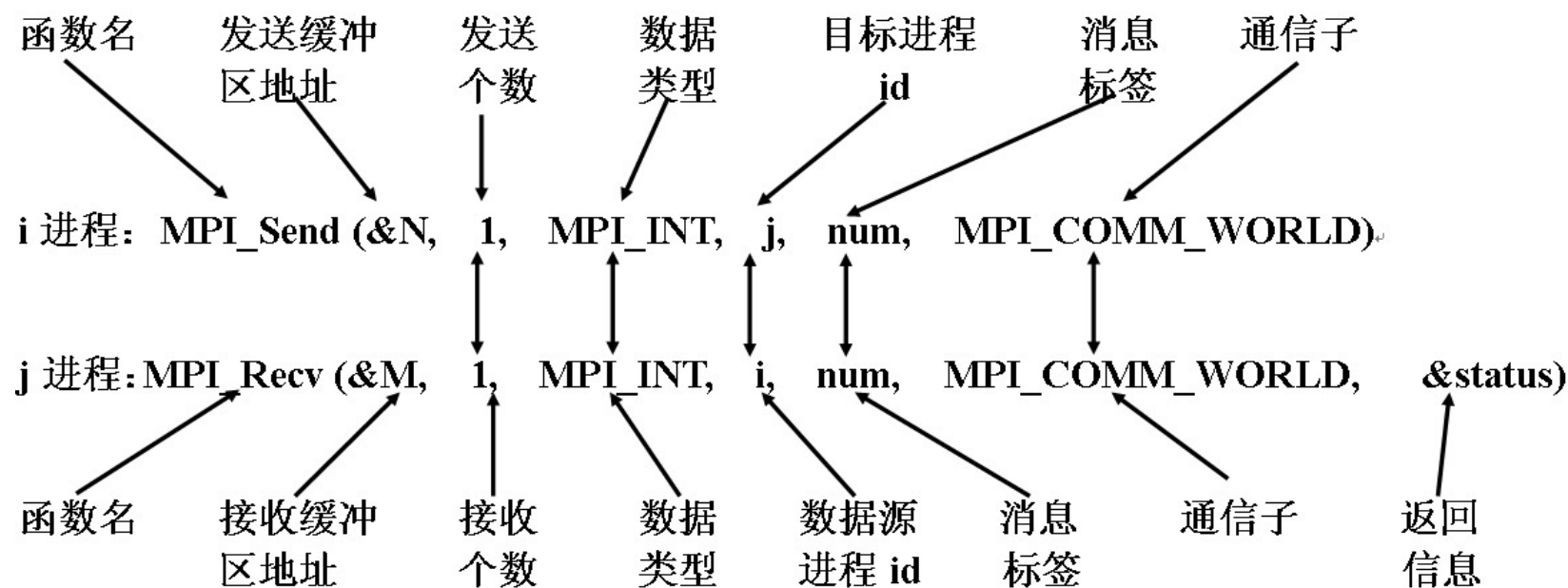
MPI_Recv(**buf, count, datatype**, **source, tag, comm**, status)



消息数据

消息信封

基本的发送与接收语句



回头来分析greetings

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

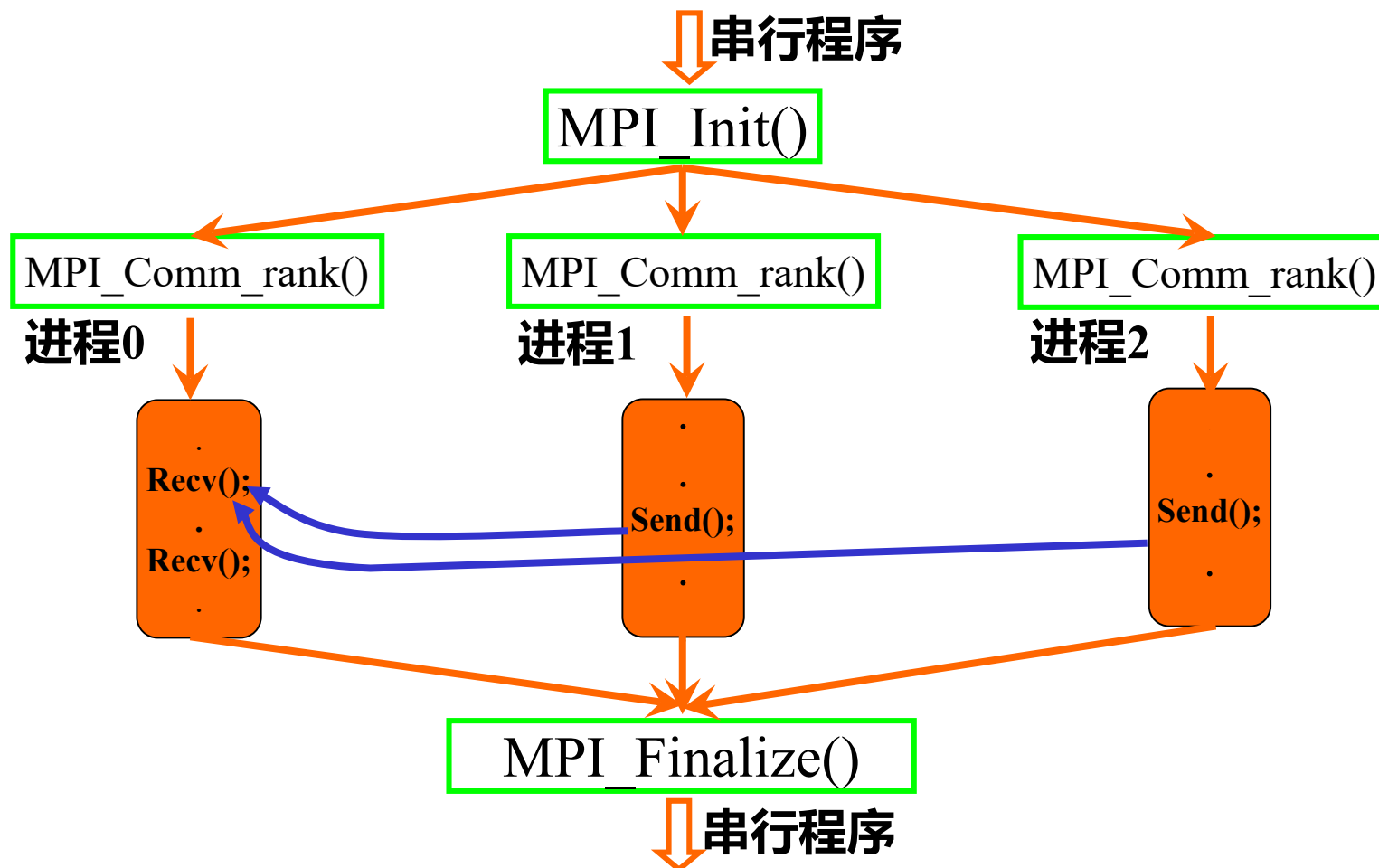
main(int argc, char* argv[])
{
    int p;                /*进程数,该变量为各处理器中的同名变量 */
    int my_rank;          /*我的进程ID,存储也是分布的 */
    MPI_Status status;    /* 消息接收状态变量,存储也是分布的 */
    char message[100];    /* 消息buffer,存储也是分布的 */

    MPI_Init(&argc, &argv); /*初始化MPI*/
    /*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, &my_rank);
    /*该函数被各进程各调用一次,得到进程数*/
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    ...
}
```

回头来分析greetings (续)

```
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);
    }
}
MPI_Finalize(); /*关闭MPI,标志并行代码段的结束*/
} /* main */
```

Greetings执行过程(进程数为3)



注意

- SPMD程序只需要一个源程序
- SPMD程序是多进程并行执行
- 每个进程都执行相同源程序的语句
- 不同进程的执行结果是不同

status 中的内容

- **status** 返回有关接收到的消息的信息, 它的结构如下:
- 在C 中**status** 是一个结构, 它包含下面三个用户可以使用域:

```
typedef struct {  
    ...  
    int MPI_SOURCE; 消息源地址  
    int MPI_TAG; 消息标签  
    int MPI_ERROR; 错误码  
    ...  
} MPI_Status;
```

查询接收到的消息长度

- Status状态参数也返回关于被接收消息的长度信息。但不能被直接得到，要求使用**MPI_Get_count**函数“解码”这个信息。
- **int MPI_Get_count (**
 MPI_Status* status /* in */,
 MPI_Datatype datatype /* in */,
 int* count /* out */)
- **以元素而不是字节来计数**

阻塞

- 需要等待操作的实际完成，或至少等待MPI系统安全地备份后才返回；
- MPI_Send与MPI_Recv都是阻塞的；
- MPI_Send调用返回时表明数据已经发出或被MPI系统复制，随后对发送缓冲区的修改不会改变所发送的数据；
- 而MPI_Recv返回，则表明已完成数据接收。
- 函数调用是非局部的

阻塞型消息传递实例

mpirun -np 2

Process 0

.....

MPI_Comm_rank(...)

.....

MPI_Send(A,...)

MPI_Recv(B,...)

.....

Process 1

.....

MPI_Comm_rank(...)

.....

MPI_Send(B,...)

MPI_Recv(A,...)

.....



进程通信的组织

.....

```
MPI_Comm_dup (MPI_COMM_WORLD, &comm);
```

```
If ( myid==0) {
```

```
    MPI_Recv(bufA0, 1, MPI_Float, 1, 101, comm, status);
```

```
    MPI_Send(bufB0, 1, MPI_Float, 1, 100, comm); }
```

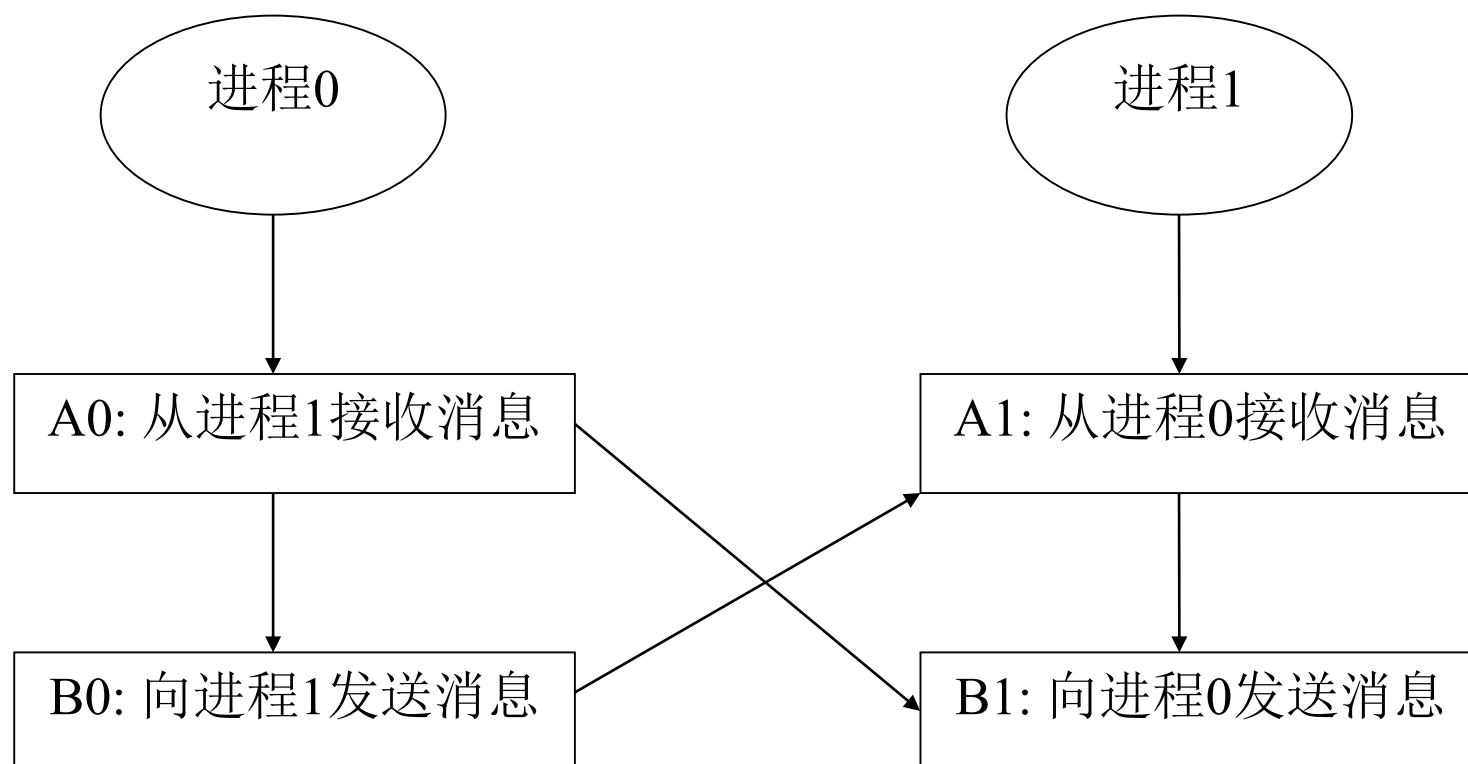
```
else if (myid==1) {
```

```
    MPI_Recv(bufA1, 1, MPI_Float, 0, 100, comm, status);
```

```
    MPI_Send(bufB1, 1, MPI_Float, 0, 101, comm); }
```

.....

进程间通信的组织

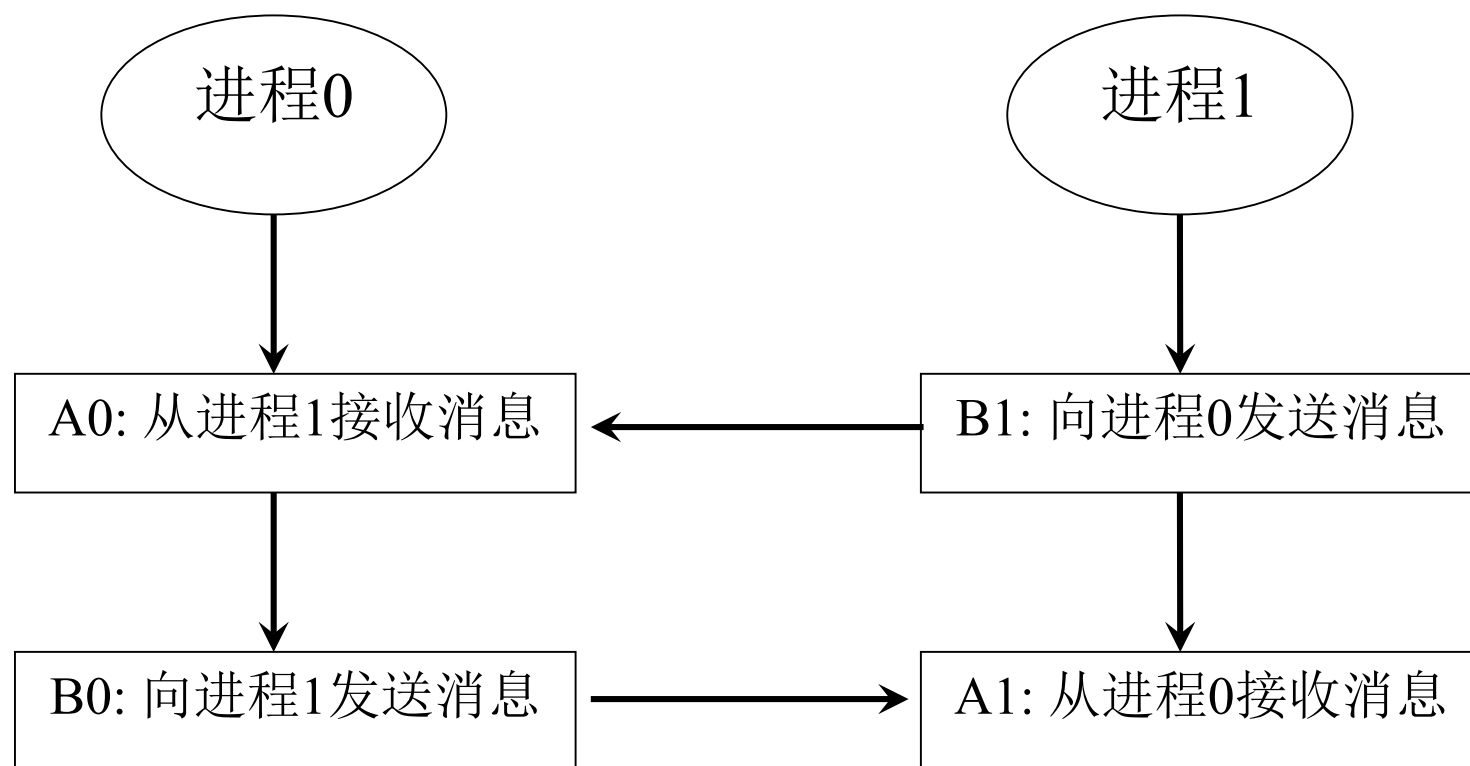


进程通信的组织

```
.....  
MPI_Comm_dup (MPI_COMM_WORLD, &comm);  
If ( myid==0) {  
  
    MPI_Recv(bufA0, 1, MPI_Float, 1, 101, comm, status);  
    MPI_Send(bufB0, 1, MPI_Float, 1, 100, comm); }  
else if (myid==1) {  
    MPI_Recv(bufA1, 1, MPI_Float, 0, 100, comm, status);  
    MPI_Send(bufB1, 1, MPI_Float, 0, 101, comm); }  
.....
```

死锁

死锁的避免



接收的有序性

- 在阻塞通信中对于接收进程在接收消息时除了要求接收到的消息的**消息信封**和接收操作自身的消息信封相**一致**
- 还要求它接收到的消息是**最早发送给自己的消息**
- 若两个消息的消息信封都和自己的消息信封吻合则**必须先接收首先发送的消息**
- 哪怕后发送的消息首先到达该接收操作也**必须等待第一个消息**

上例的修改

.....

```
If ( myid==0) {
```

```
    MPI_Recv(bufA0,1,MPI_Float,1,101,MPI_COMM_WORLD,  
             status);
```

```
    MPI_Send(bufB0,1,MPI_Float,1,100,MPI_COMM_WORLD  
             );}
```

```
else if (myid==1){
```

```
    MPI_Send(bufB1,1,MPI_Float,0,101,MPI_COMM_WORLD  
             );}
```

```
    MPI_Recv(bufA1,1,MPI_Float,0,100,MPI_COMM_WORLD,  
             status);
```

.....

非阻塞

- 函数的调用总是立即返回，而实际的操作则由MPI系统在后台进行；
- 用户必须随后调用其他函数来等待或查询完成情况；
- 在操作完成之前对相关数据区的操作是不安全的；
- 函数调用是局部的

阻塞与非阻塞的差别

■ 用户发送缓冲区的重用:

- 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。
- 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.

非阻塞 - 发送

- 一个非阻塞send start调用发起一个发送，但并不完成，而且send start调用会在消息拷贝出发送缓冲区以前返回；另一个send complete调用来完成发送操作，即证实数据已被拷贝出发送缓冲区，缓冲区可重用。通过合适的硬件，在send start完成之后，send complete完成之前，数据向外传送的过程可以和计算同时进行。

非阻塞 - 接收

- 一个非阻塞的receive start调用发起一个接收操作，这个调用会在消息存入接收缓冲区之前返回，但其返回并不表示接收已完成；另一个receive complete调用来完成接收操作，并证实数据已写入接收缓冲区。通过合适的硬件，在receive start完成后，receive complete完成之前，数据的写入过程可以和计算同时进行。
- 非阻塞通信可以避免系统缓冲以及内存到内存的复制，计算和通信可以重叠，从而使性能得到改善。

非阻塞发送 MPI_Isend()

■ int MPI_Isend (
void* message /* in */,
int count /* in */,
MPI_Datatype datatype /* in */,
int dest /* in */,
int tag /* in */,
MPI_Comm comm /* in */,
MPI_Request* request /*out*/)

The “I” stands for “Immediate”

非阻塞发送 MPI_Isend()

- 提交一个消息发送请求，要求MPI系统在后台完成。
- MPI_Isend为该发送创建一个请求，并将请求的句柄通过request变量返回给MPI进程。
- 随后MPI_Test/MPI_Wait(查询/等待)等函数可以使用request来判断发送是否完成。

非阻塞接收 MPI_Irecv()

■ int MPI_Irecv (

void*	message	/*out*/,
int	count	/* in */,
MPI_Datatype	datatype	/* in */,
int	source	/* in */,
int	tag	/* in */,
MPI_Comm	comm	/* in */,
MPI_Request*	request	/*out*/)

非阻塞通信的完成

- 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。
 - 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。
- 通过**MPI_Wait**和**MPI_Test**来判断通信是否已经完成;

MPI_Wait()函数

- **int MPI_Wait (**
 MPI_Request* request /*in/out*/,
 MPI_Status* status /* out */)
- 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_Wait一直等到非阻塞操作真正完成之后才返回，可以认为阻塞通信等于非阻塞通信加上MPI_Wait。

MPI_Wait()应用示例1

```
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_Wait()应用示例2

```
int    x;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank==0) {
    MPI_Isend(x, 1, MPI_INT, 1, msgtag,
              MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank==1) {
    MPI_Recv(x, 0, MPI_INT, 1, msgtag,
             MPI_COMM_WORLD, status);
```

MPI_Test()函数

- **int MPI_Test (**
 MPI_Request* request /* in /out*/,
 int* flag /* out */,
 MPI_Status* status /* out */)
- MPI_Test用来检测非阻塞操作是否真正结束
- MPI_Test不论通信是否完成，立即返回。如果通信已经完成，flag=true;否则flag=false

MPI_Test()应用示例

```
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);
}
```

Request对象的释放

- Request参数用来描述非阻塞通信状况的对象，通过对其查询，可以知道与之相应的非阻塞发送和接收是否完成。
- Request对象只有在通信完成后才会被释放。MPI中提供了函数**MPI_Request_free**,可不必等待通信完成便可释放request对象。
- **int MPI_Request_free (**
 MPI_Request* request /*in/out*/)

消息探测probe

- **MPI_Probe()**和**MPI_Iprobe()**函数探测接收消息的内容,但并不实际接收消息。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。
- 在接收未知长度的消息时,可先用**MPI_Probe** 和 **MPI_Get_count** 得到消息的长度。
- 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。

MPI_Probe()函数

- **int MPI_Probe (**

int	source	/* in */,
------------	---------------	------------------

int	tag	/* in */,
------------	------------	------------------

MPI_Comm	comm	/* in */,
-----------------	-------------	------------------

MPI_Status*	status	/*out*/)
--------------------	---------------	-----------------

- 阻塞型探测，直到有一个符合条件的消息到达，返回
- MPI_ANY_SOURCE
- MPI_ANY_TAG

MPI_Iprobe()函数

■ int MPI_Iprobe (

int	source	/* in */,
int	tag	/* in */,
MPI_Comm	comm	/* in */,
int*	flag	/*out*/,
MPI_Status*	status	/*out*/)

- 非阻塞型探测，无论是否有一个符合条件的消息到达，返回。有flag=true；否则flag=false

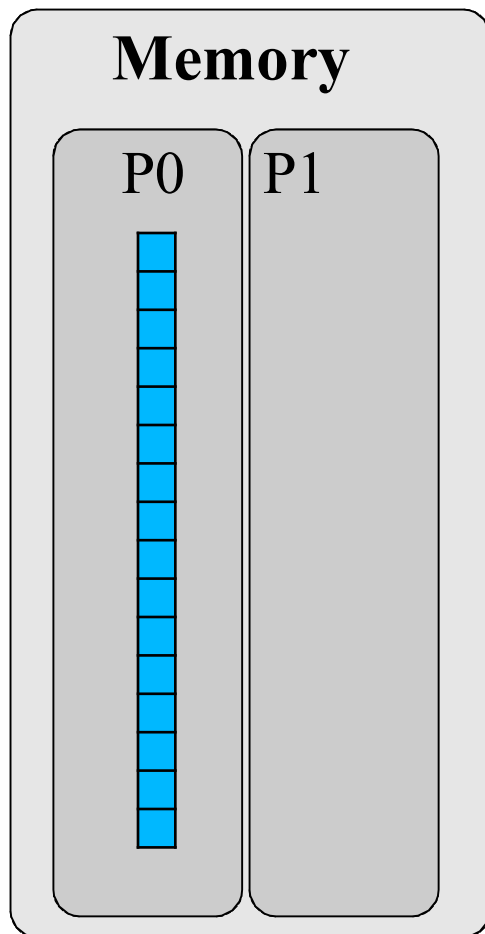
MPI_Probe()应用示例1

```
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);
    }
```

MPI_Probe()应用示例2

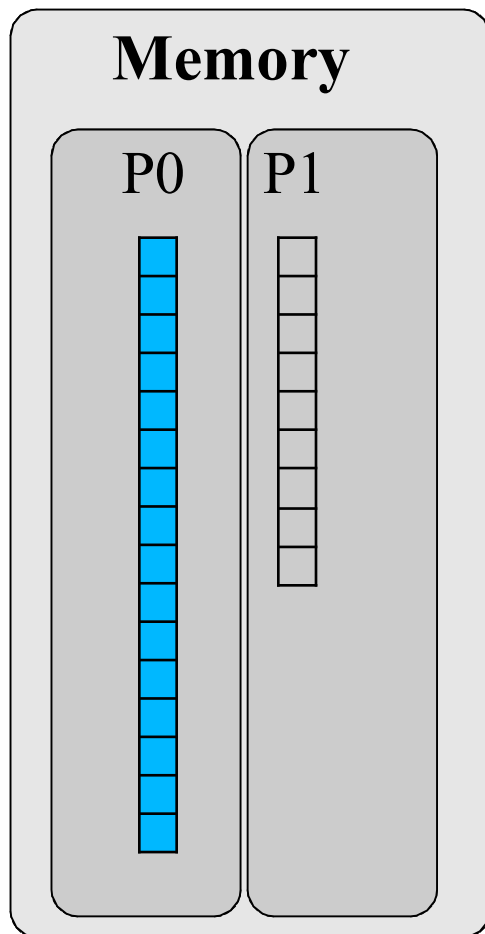
```
int x[2], y, count;
int z[2]=[100,100];
MPI_Comm_rank(comm, &rank);
if(rank ==0) /*0->2发送两int型数*/
    MPI_Send(z,2,MPI_INT,2,99,comm);
else if(rank == 1) /*1->2发送一int型数*/
    MPI_Send(100,1,MPI_INT,2,99,comm);
else /* 根进程接收 */
    for(int i=0;i<2;i++){
        MPI_Probe(MPI_ANY_SOURCE,0,comm,&status);
        MPI_Get_count(&status, MPI_INT, &count);
        /*Blocking*/
        if(count == 2) {
            MPI_Recv(x, 2, MPI_INT,
                    MPI_ANY_SOURCE, 99, &status); }
        else {MPI_Recv(&y,1,MPI_INT,
                    MPI_ANY_SOURCE, 99, &status); }
    }
```

Parallel Sum

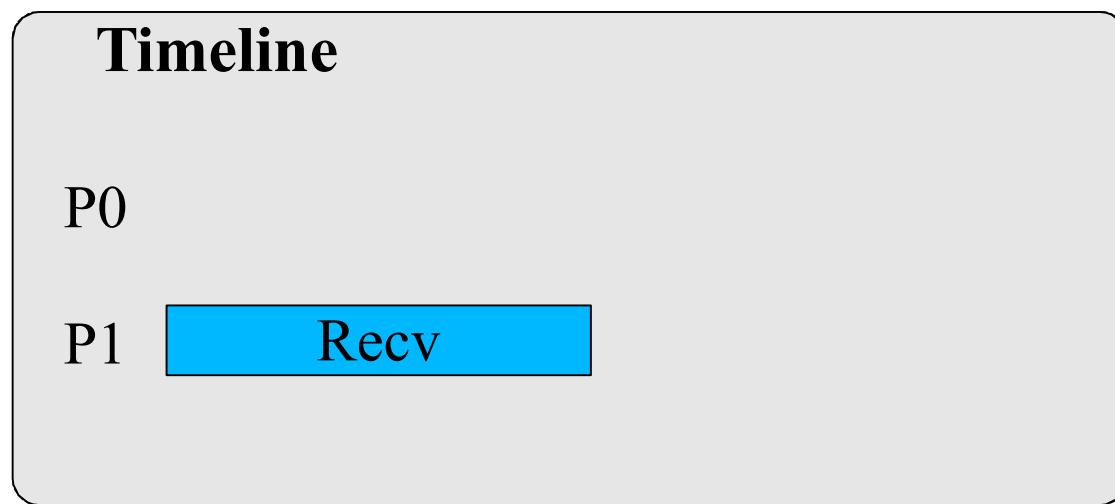


- **Array is originally on process 0 (P0)**
- **Parallel algorithm**
 - Scatter
 - Half of the array is sent to process 1
 - Compute
 - P0 & P1 sum independently their segment
 - Reduction
 - Partial sum on P1 sent to P0
 - P0 sums the partial sums

Parallel Sum

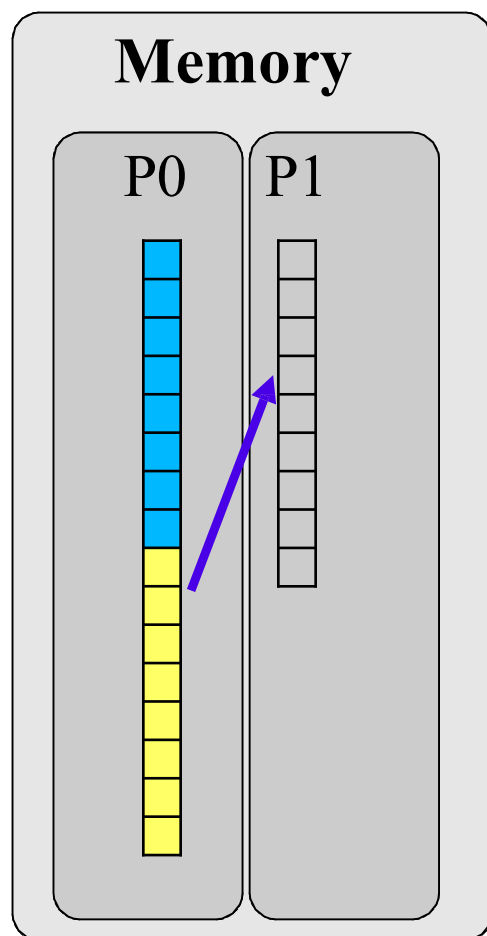


Step 1: Receive operation in scatter

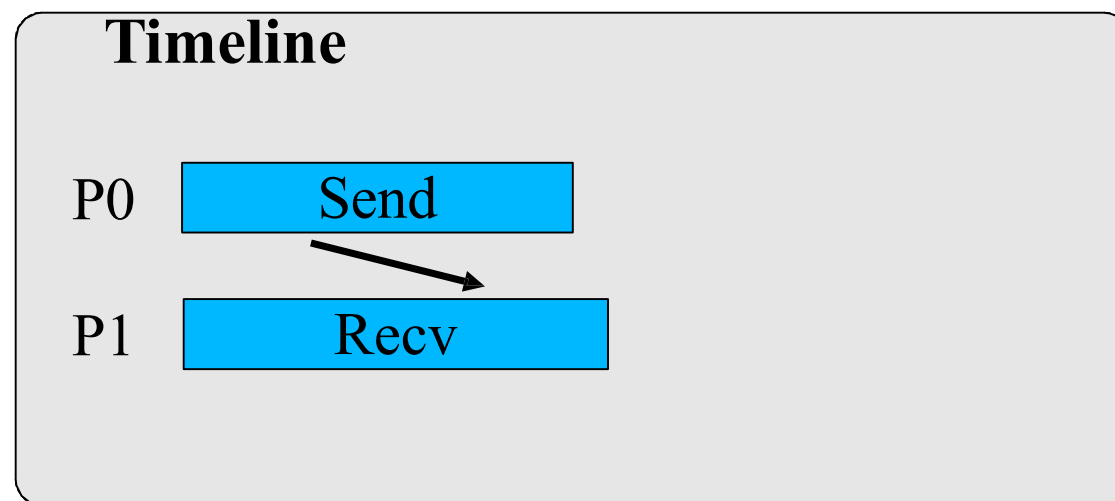


P1 posts a receive to receive half of the array from process 0

Parallel Sum

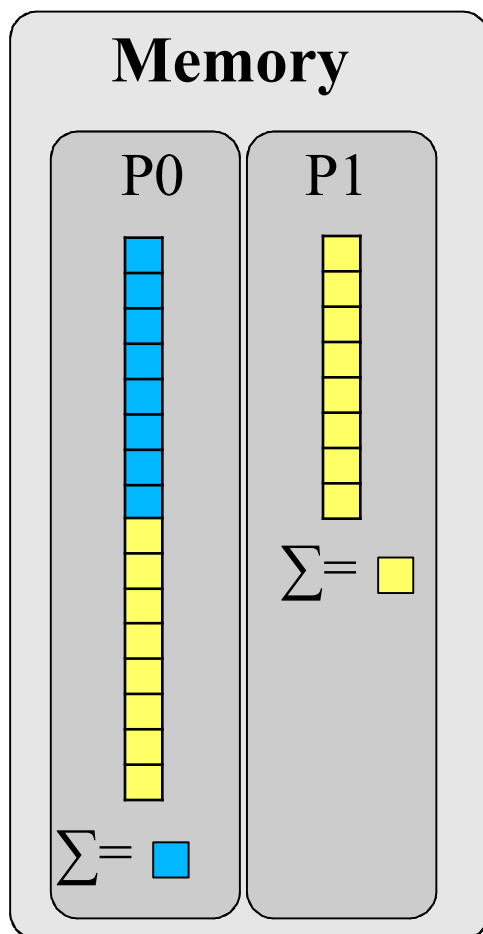


Step 2: Send operation in scatter

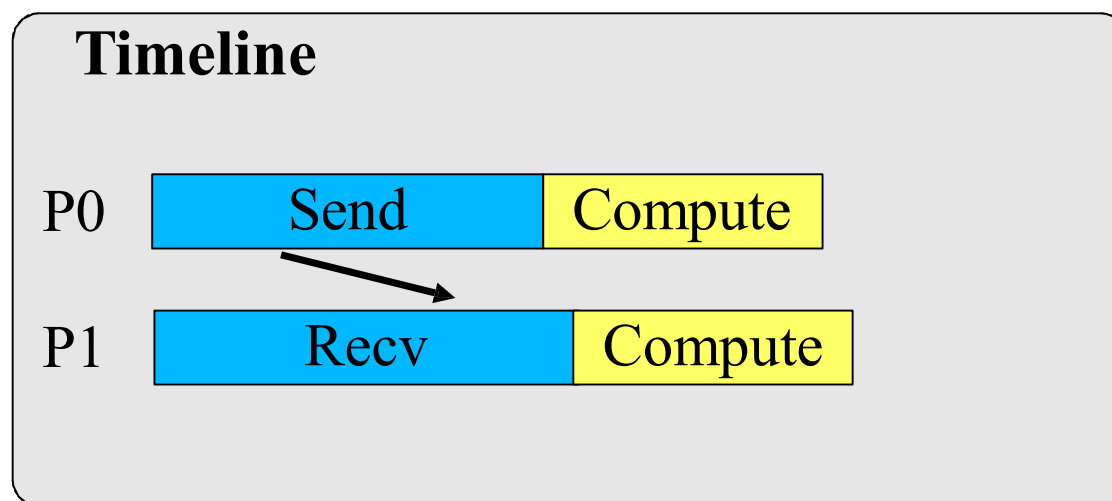


P0 posts a send to send the lower part of the array to P1

Parallel Sum

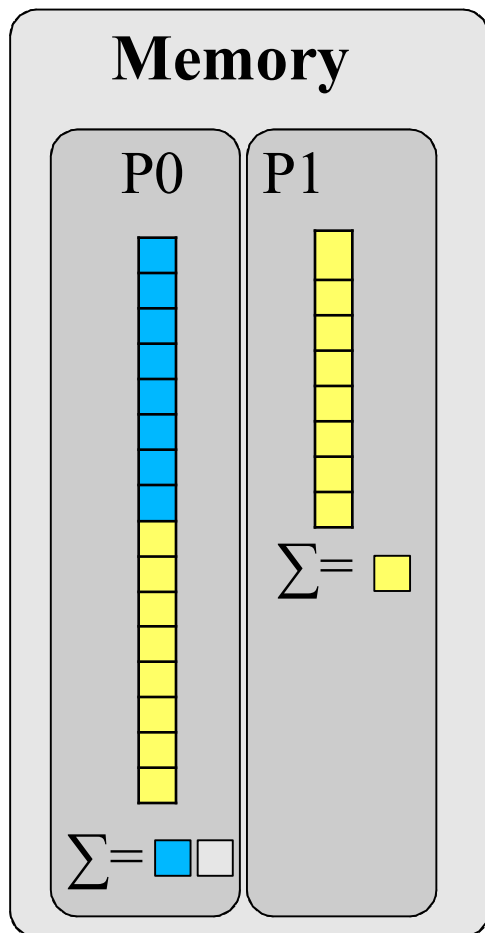


Step 3: Compute the sum in parallel

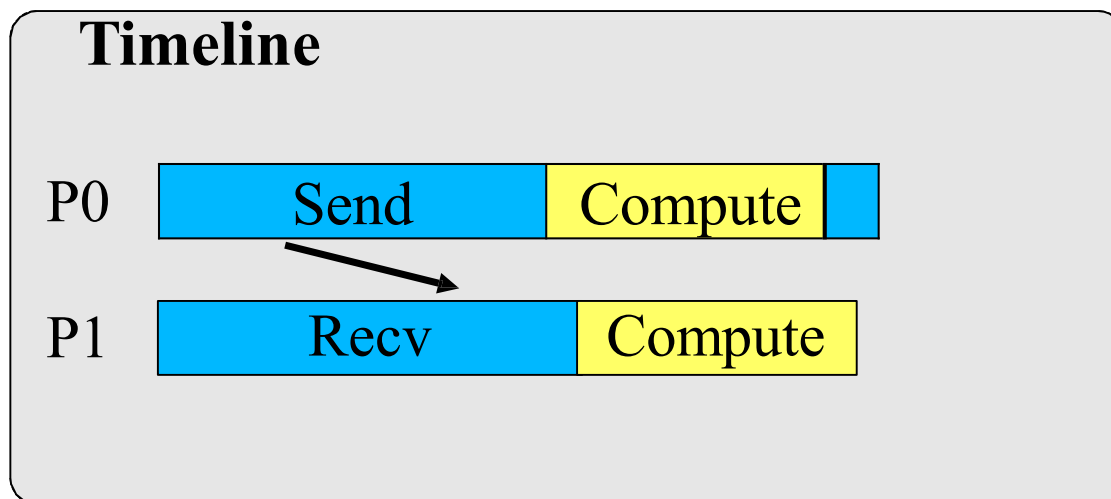


P0 & P1 computes their partial sums and store them locally

Parallel Sum

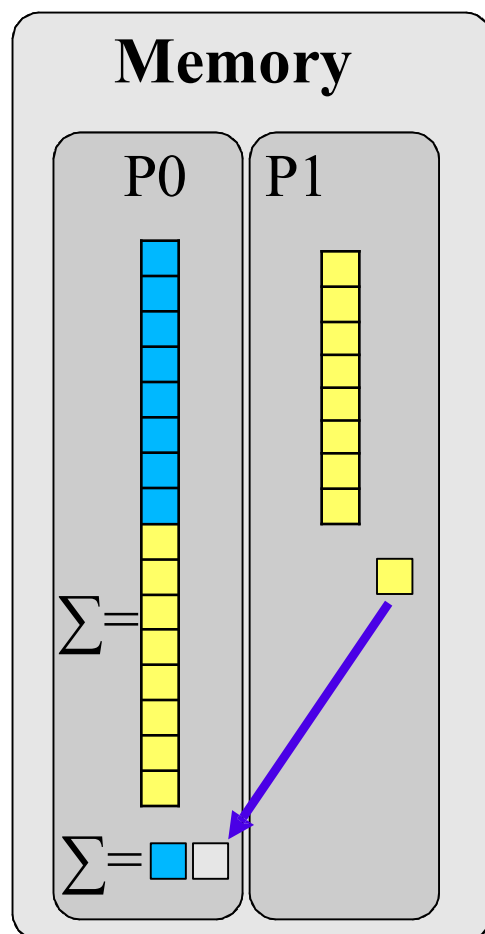


Step 4: Receive operation in reduction

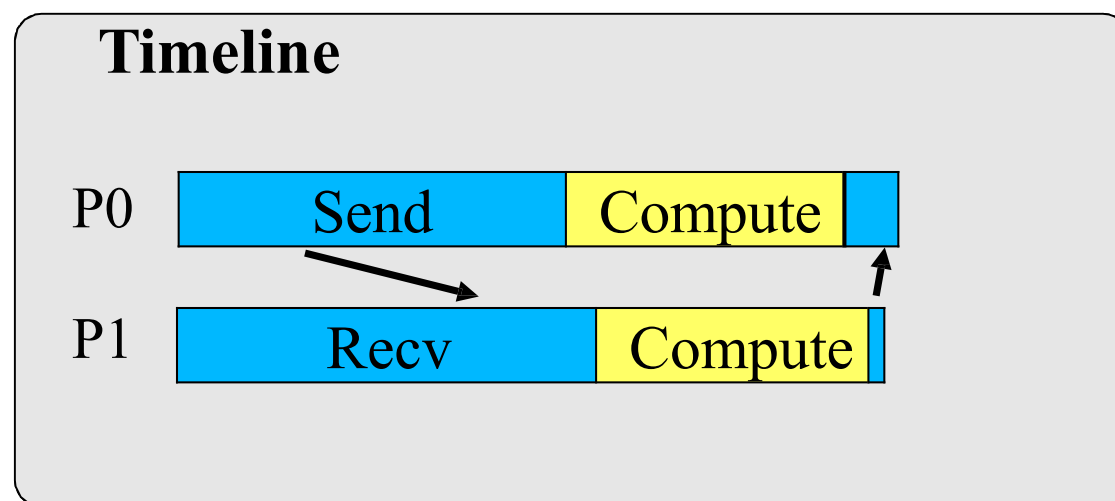


P0 posts a receive to receive partial sum

Parallel Sum

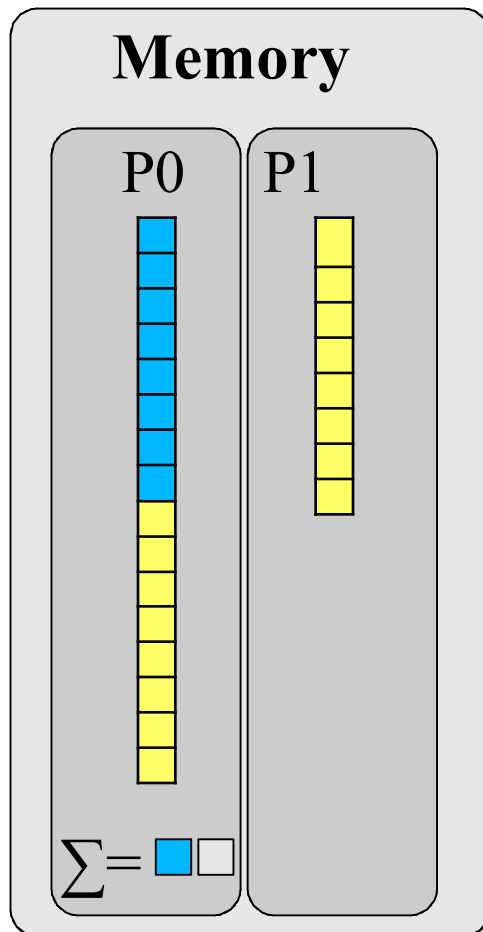


Step 5: Send operation in reduction

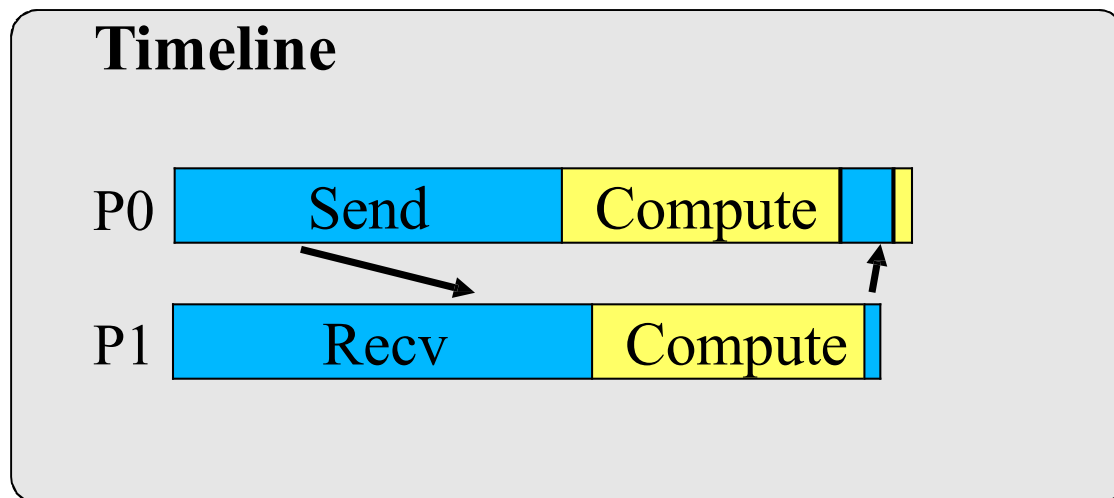


P1 posts a send with partial sum

Parallel Sum



Step 6: Compute final answer



P0 sums the partial sums

Exercise: ParallelSum

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){  int i,N;
double *array;
double sum;  N=100;
array=malloc(sizeof(double)*N);
for(i=0;i<N;i++){
array[i]=1.0;
}
sum=0;  for(i=0;i<N;i++){
sum+=array[i];
}
printf("Sum is %g\n",sum);
}
```

Exercise: ParallelSum

1. Parallelize the sum.c program with MPI

- The relevant MPI commands can be found back in the README
- run this program with two MPI-tasks

2. Use MPI_status to get information about the message received

- print the count of elements received

3. Using MPI_Probe (details on next slide) to find out the message size to be received

- Allocate an arrays large enough to receive the data
- call MPI_Recv()

***_sol.c contains the solution of the exercise.**

发送与接收组合

- **MPI_Sendrecv()** 函数将一次发送调用和一次接收调用合并进行. 它使得MPI 程序更为简洁.
- 更重要的是, MPI 的实现通常能够保证使用 MPI_SENDRECV 函数的程序不会出现由于消息收发配对不好而引起的程序死锁.

MPI_Sendrecv()

■ int MPI_Sendrecv (

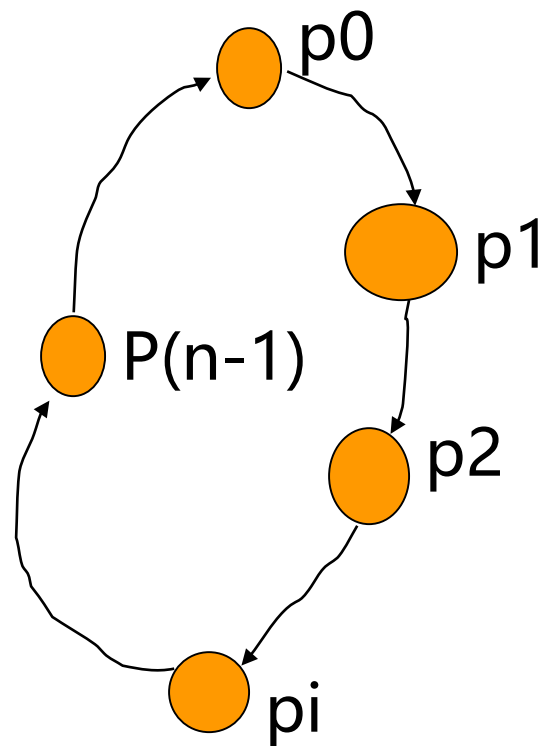
send	void*	send_buf	/* in */,
	int	send_count	/* in */,
	MPI_Datatype	send_type	/* in */,
	int	dest	/* in */,
	int	send_tag	/* in */,
recv	void*	recv_buf	/*out*/,
	int	recv_count	/* in */,
	MPI_Datatype	recv_type	/* in */,
	int	source	/* in */,
	int	recv_tag	/* in */,
	MPI_Comm	comm	/* in */,
	MPI_Status*	status	/*out*/)

MPI_Sendrecv()

- send-recv操作把发送一个消息到一个目的地和从另一个进程接收一个消息合并到一个调用中。
- 源和目的可以是同一个地方。
- 一个由send-receive发出的消息可以被一个普通接收操作接收，或被一个检查操作检查；一个send-receive操作可以接收一个普通发送操作发送的消息。

MPI_Sendrecv()

- 一个send-receive操作对穿过一个进程链的切换操作非常有用。如果阻塞的发送和接收被用于这种切换，则需要正确排列发送和接收的顺序(例如偶数进程发送，然后接收，奇数进程先接收，然后发送)以避免循环依赖导致死锁。



MPI Sendrecv 例1

```
MPI_Comm_dup(MPI_COMM_WORLD, &comm);

if (myid == 0) {

    MPI_Send(&a, 1, MPI_FLOAT, 1, tag1, comm);
    MPI_Recv(&b, 1, MPI_FLOAT, 1, tag2, comm, &status);

} else if (myid == 1) {

    MPI_Send(&b, 1, MPI_FLOAT, 0, tag2, comm);
    MPI_Recv(&a, 1, MPI_FLOAT, 0, tag1, comm, &status);
}
```

MPI Sendrecv 例2

```
MPI_Comm_dup (MPI_COMM_WORLD, &comm) ;

if (myid == 0) {

    MPI_Sendrecv (&a, 1, MPI_FLOAT, 1, tag1,
                  &b, 1, MPI_FLOAT, 1, tag2, comm, &status) ;

} else if (myid == 1) {

    MPI_Sendrecv (&b, 1, MPI_FLOAT, 0, tag2,
                  &a, 1, MPI_FLOAT, 0, tag1, comm, &status) ;

}
```

MPI Sendrecv 例3

```
MPI_Comm_dup(MPI_COMM_WORLD, &comm);

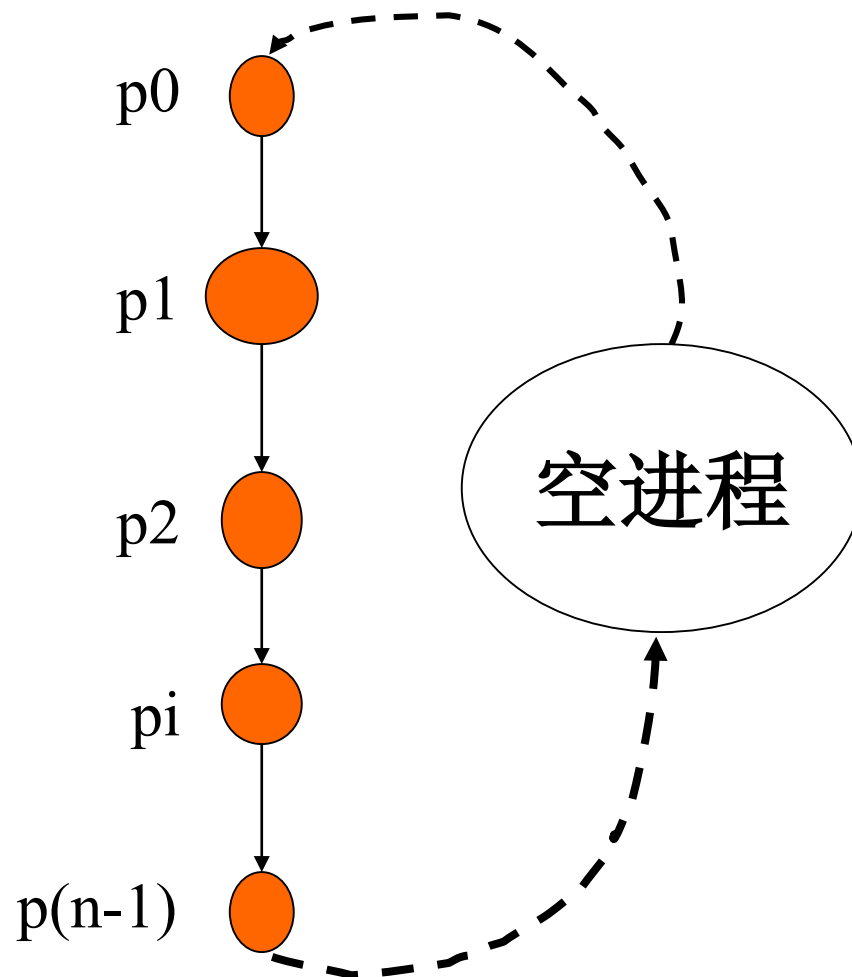
if (myid == 0) {
    MPI_Sendrecv(&a, 1, MPI_FLOAT, 1, tag1,
                 &b, 1, MPI_FLOAT, 2, tag2, comm, &status);
} else if (myid == 1) {
    MPI_Recv(&a, 1, MPI_FLOAT, 0, tag1, comm, &status);
} else if (myid == 2) {
    MPI_Send(&b, 1, MPI_FLOAT, 0, tag2, comm);
}
```

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.
- 在很多情况下为通信指定一个“假”的源或目标是非常方便的。这可以简化处理边界的代码,例如,用调用 MPI_Sendrecv实现非循环切换的时候。
- MPI_PROC_NULL这个特殊的值可以用来替换一个调用所要求的源或目标。
- 一个使用MPI_PROC_NULL进程的通信没有动作。一个给MPI_PROC_NULL的发送会立即成功返回。一个从MPI_PROC_NULL的接收会立即成功返回,对接收缓冲区没有任何改变.

空进程

- `MPI_Get_count(&status, MPI_Datatype datatype, &count) => count = 0`
- `status.MPI_SOURCE = MPI_PROC_NULL`
- `status.MPI_TAG = MPI_ANY_TAG`



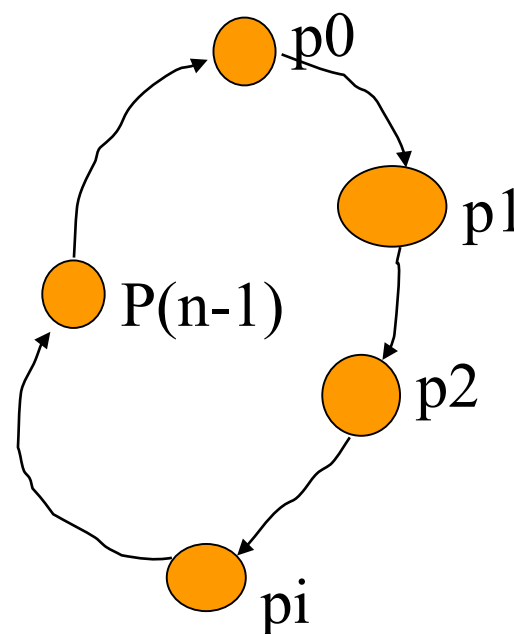
函数类型	通信模式	阻塞型	非阻塞型
消息发送函数	标准模式	MPI_Send	MPI_Isend
	缓冲模式	MPI_Bsend	MPI_Ibsend
	同步模式	MPI_Ssend	MPI_Issend
	就绪模式	MPI_Rsend	MPI_Irsend
消息接收函数		MPI_Recv	MPI_Irecv
发送接收函数		MPI_Sendrecv	
		MPI_Sendrecv_replace	
消息检测函数		MPI_Probe	MPI_Iprobe
等待查询函数		MPI_Wait	MPI_Test
		MPI_Waitall	MPI_Testall
		MPI_Waitany	MPI_Testany
		MPI_Waitsome	MPI_Testsome
释放通信请求		MPI_Request_free	
			MPI_Cancel
取消通信请求			MPI_Test_cancelled

上机练习——循环消息传递

1. 实现结点间的循环消息传递

0—1—2—3—4—5--...--
...n-1—0

发送-接收组合方式



循环通信

```
#include "mpi.h"

main(int argc, char* argv[])
{
    int                p, my_rank, tag, source, dest, tag = 0;
    MPI_Status         status;
    char               send_m[100],  recv_m[100];
    double              start_time, end_time;
    MPI_Comm    comm;

    MPI_Init(&argc, &argv);                /*初始化MPI*/
    MPI_Comm_dup (MPI_COMM_WORLD, &comm);
    MPI_Comm_rank (comm, &my_rank);          /*获得进程的ID*/
    MPI_Comm_size (comm, &p);                /*获得进程数*/

    dest = (my_rank + 1) % p;                /*消息发送的目的*/
    source = (my_rank - 1 + p) % p;          /*消息接收的源*/
```

```
if (my_rank == 0) {
    sprintf(send_m, "Greetings from process%d!", my_rank); /*建立消息*/
    start_time = MPI_Wtime();
    MPI_Send(send_m, strlen(send_m)+1, MPI_CHAR, dest, tag, comm);
    MPI_Recv(recv_m, 100, MPI_CHAR, source, tag, comm, &status);
    end_time = MPI_Wtime();
    Printf("the running time is %f\n", end_time - start_time);
}
else{
    MPI_Recv(recv_m, 100, MPI_CHAR, source, tag, comm, &status);
    MPI_Send(recv_m, strlen(recv_m) +1, MPI_CHAR, dest, tag,
comm);
}

MPI_Finalize(); /*关闭MPI,标志并行代码段的结束*/
} /* main */
```

THANKS