

# 并行计算

## (Parallel Computing)

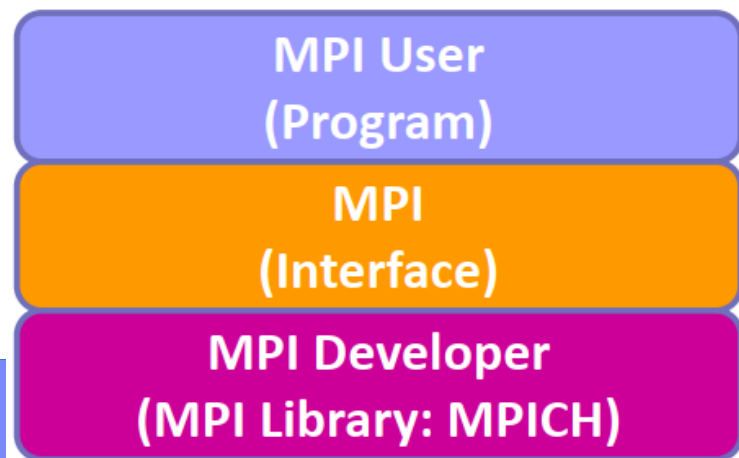
# 分布式内存编程 – MPI（一）

## 学习内容：

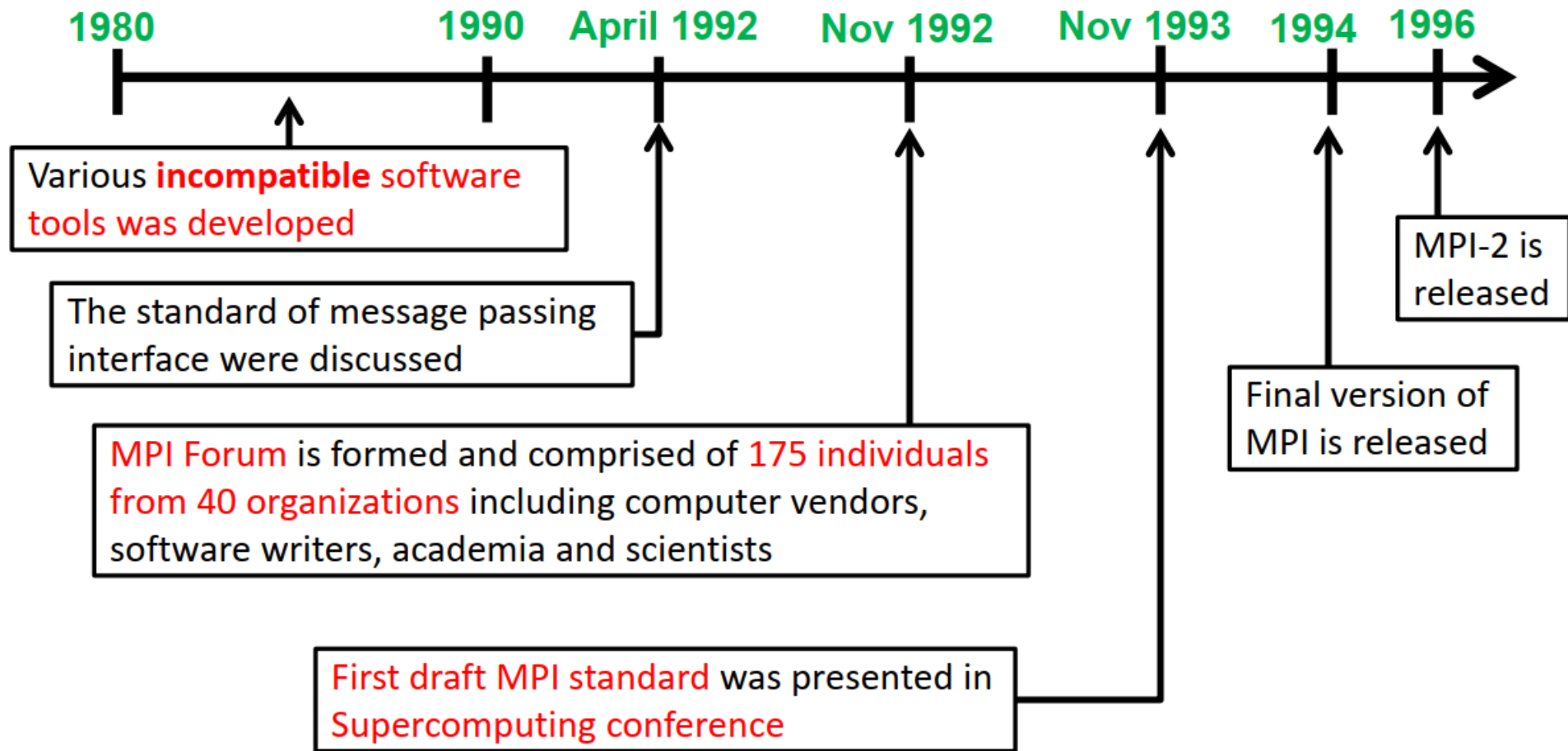
- 什么是MPI
- 第一个 MPI 程序

# 1. 什么是 MPI

- MPI = Message Passing Interface
- 常用于分布式存储系统和高性能计算（HPC）
- 目标：
  - 可移植性：在不同的机器和平台上运行
  - 可扩展性：在百万级计算机节点上运行
  - 灵活性：将MPI开发者和MPI编程者（用户）相分离

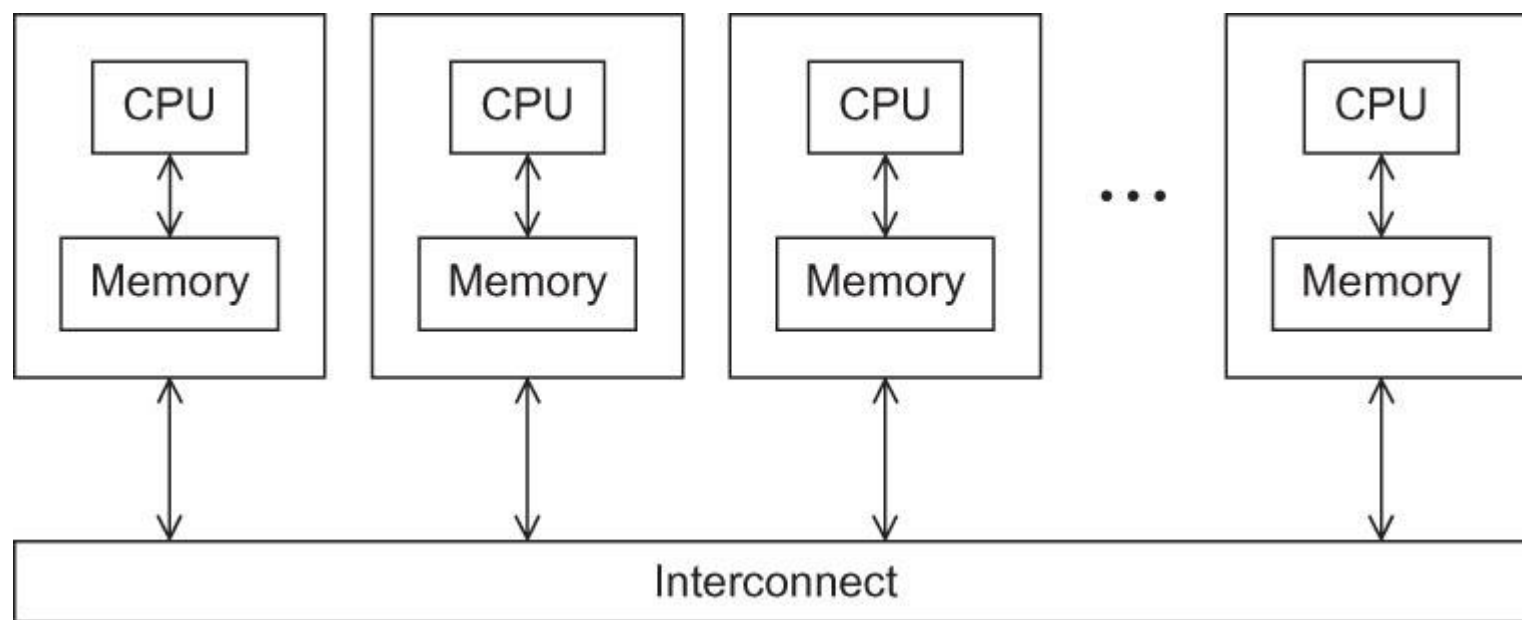


# 1. 什么是 MPI



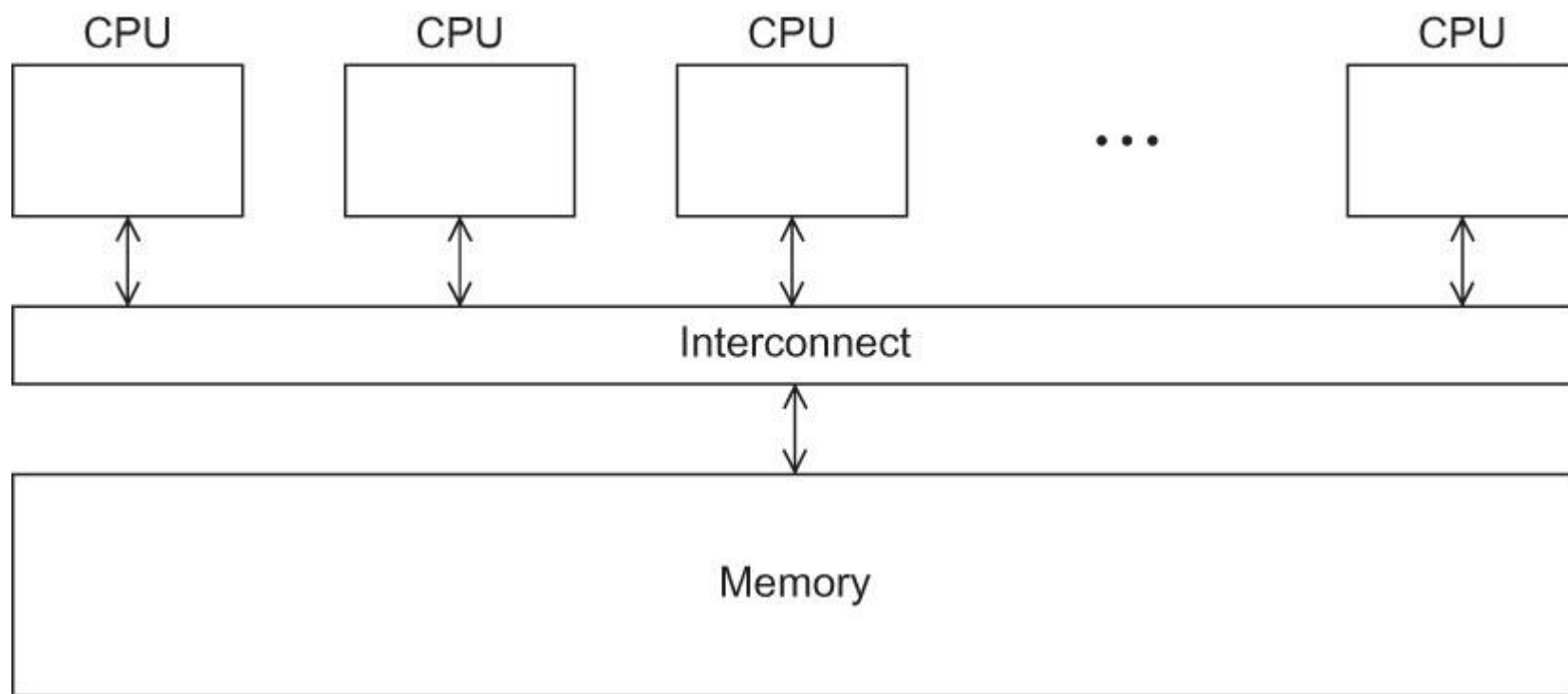
# 1. 什么是 MPI

- 从程序员的角度来看，分布式内存系统由一组通过网络连接的 core-memory 对组成，与 core 相关联的内存只有该 core 可以直接访问



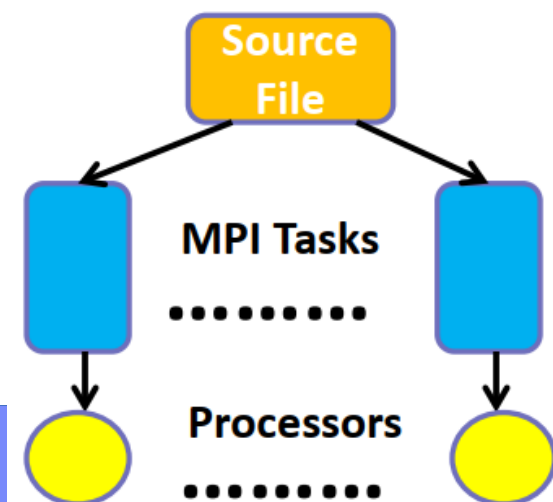
# 1. 什么是 MPI

- 从程序员的角度来看，共享内存系统由一组连接到全局可访问内存的 core 组成，其中每个 core 都可以访问任何内存位置

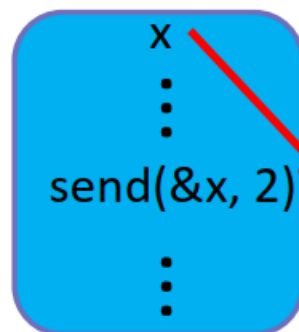


# 1. 什么是 MPI

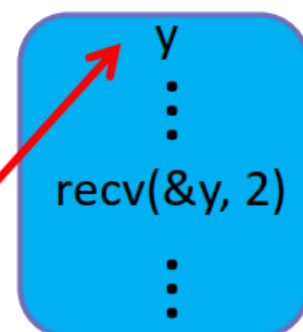
- 在消息传递程序中，运行在 core-memory 上的程序为进程
- 两个进程可以通过调用函数来通信
- 一个调用 send 函数，另一个调用 receive 函数
- MPI 库函数，可以被 C、C++、Fortran 程序调用



Task/process 0



Task/process 1



## 2.第一个 MPI 程序

```
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```





## 2. 第一个 MPI 程序

- 一个进程负责输出，其他进程向其发送消息
- 通常用非负整数来标识进程 (rank)
- $p$  个进程:  $0, 1, 2, \dots, p-1$

```
1 #include <stdio.h>
2 #include <string.h> /* For strlen */
3 #include <mpi.h> /* For MPI functions , etc */
4
5 const int MAX_STRING = 100;
6
7 int main(void) {
8     char greeting[MAX_STRING];
9     int comm_sz; /* Number of processes */
10    int my_rank; /* My process rank */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17        sprintf(greeting, "Greetings from process %d of %d!",
18                my_rank, comm_sz);
19        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20                MPI_COMM_WORLD);
21    } else {
22        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23        for (int q = 1; q < comm_sz; q++) {
24            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25                    0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            printf("%s\n", greeting);
27        }
28    }
29
30    MPI_Finalize();
31    return 0;
32 } /* main */
```

## 2. 第一个 MPI 程序

### ● 编译

*wrapper script to compile*

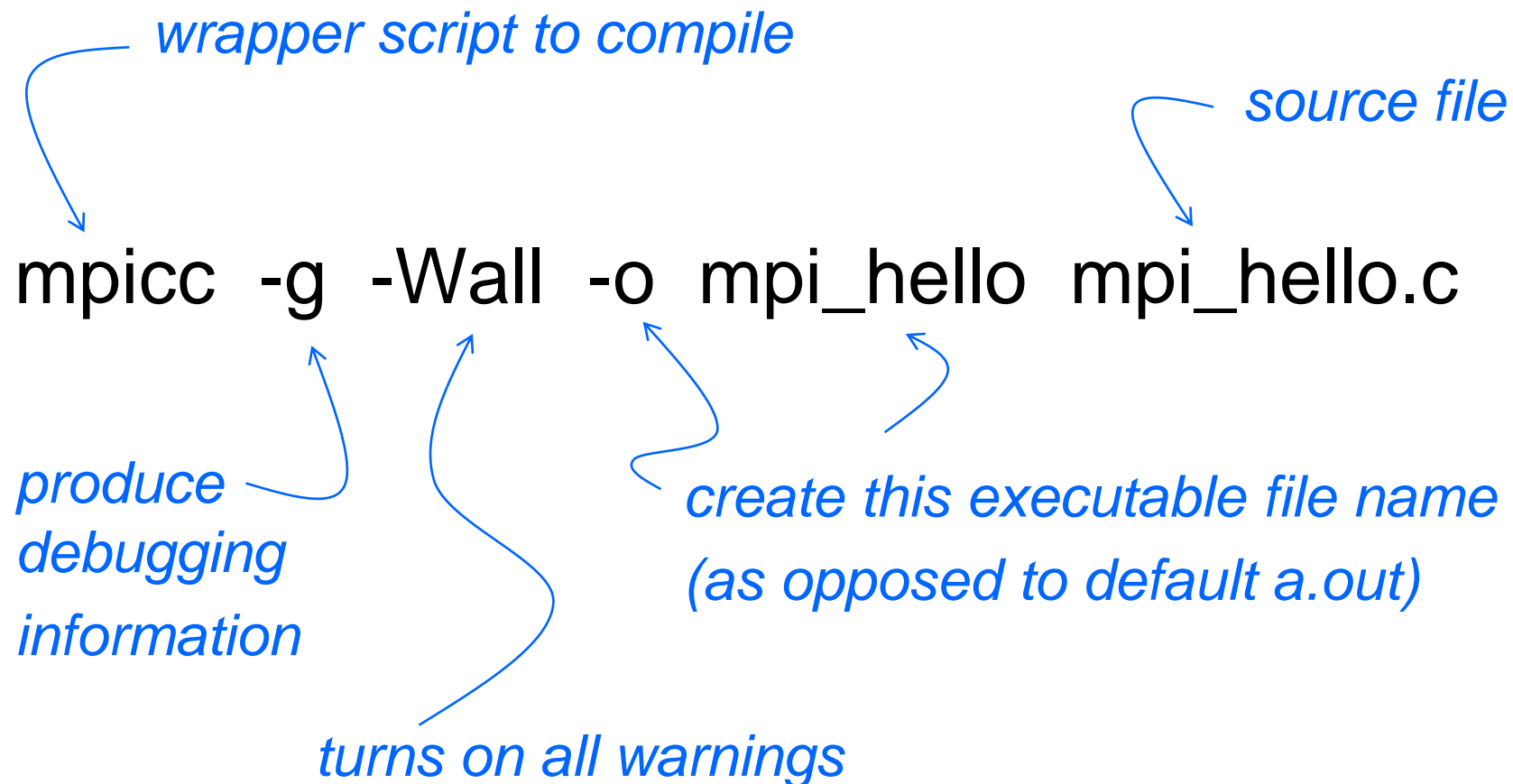
*source file*

```
mpicc -g -Wall -o mpi_hello mpi_hello.c
```

*produce debugging information*

*create this executable file name (as opposed to default a.out)*

*turns on all warnings*



## 2. 第一个 MPI 程序

- 执行

`mpiexec -n <number of processes> <executable>`

---

`mpiexec -n 1 ./mpi_hello`

 *run with 1 process*

`mpiexec -n 4 ./mpi_hello`

 *run with 4 processes*

## 2. 第一个 MPI 程序

- 执行

```
mpiexec -n 1 ./mpi_hello
```

Greetings from process 0 of 1 !

```
mpiexec -n 4 ./mpi_hello
```

Greetings from process 0 of 4 !

Greetings from process 1 of 4 !

Greetings from process 2 of 4 !

Greetings from process 3 of 4 !

## 2.第一个 MPI 程序

### ●MPI 程序

#### ➤C语言程序

- main函数
- stdio.h, string.h .....

#### ➤需要 mpi.h 头文件

#### ➤MPI 标识符：MPI\_

#### ➤下划线后第一个字母大写：函数名和 MPI 定义的类型

## 2. 第一个 MPI 程序

### ● MPI\_Init

- 进行所需的准备工作，如：为消息缓冲区分配内存、为进程分配 ID 等
- 在它之前，不应调用其他 MPI 函数

```
int MPI_Init(  
    int*      argc_p  /* in/out */,  
    char***   argv_p  /* in/out */);
```

## 2. 第一个 MPI 程序

### ● MPI\_Finalize

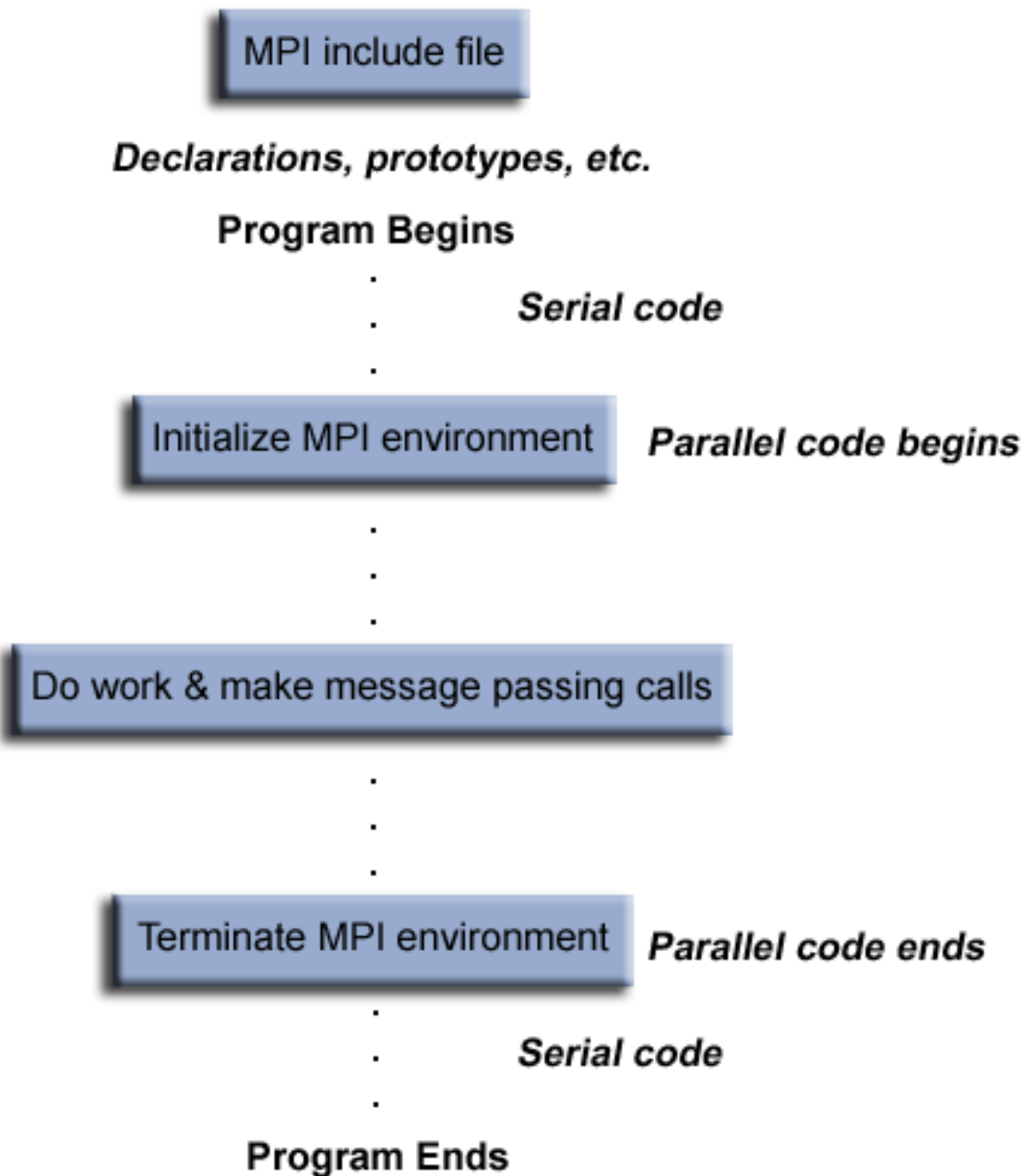
- 告知 MPI 工作结束，可以清理为程序分配的资源
- 在它之后，通常不再调用 MPI 函数

```
int MPI_Finalize(void);
```



## 2.第一个 MPI 程序

●典型的 MPI 程序



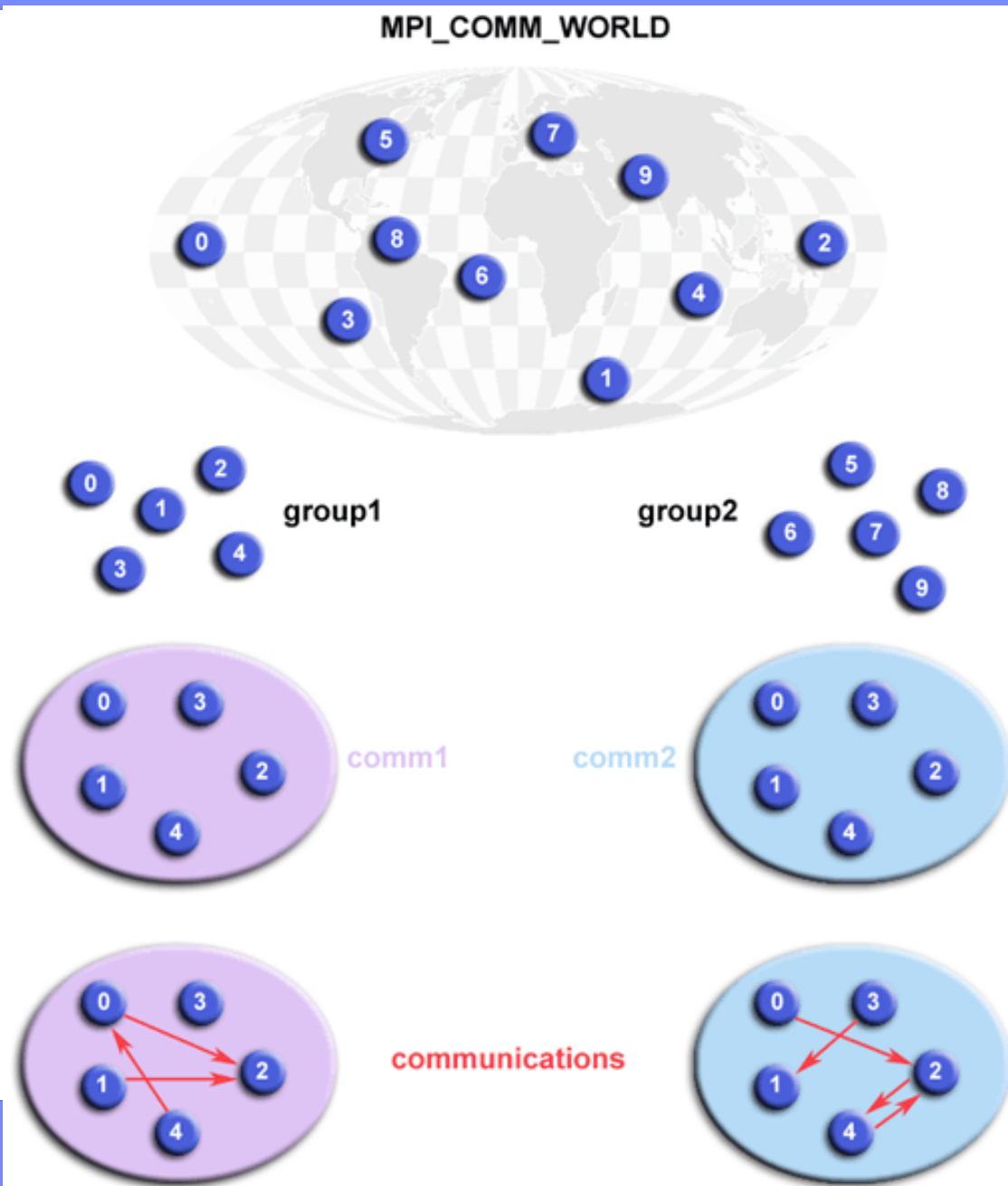
## 2.第一个 MPI 程序

### ●通信器（Communicators）

- 可以互相发送消息的进程集合
- MPI\_Init 定义了一个通信器，它由程序启动时创建的所有进程组成
- **MPI\_COMM\_WORLD**

## 2.第一个 MPI 程序

- 通信器 (Communicators)





## 2. 第一个 MPI 程序

### ● 通信器 (Communicators)

```
int MPI_Comm_size(  
    MPI_Comm comm      /* in */,  
    int* comm_sz_p     /* out */);
```

*number of processes in the communicator*

```
int MPI_Comm_rank(  
    MPI_Comm comm      /* in */,  
    int* my_rank_p     /* out */);
```

*my rank*  
*(the process making this call)*

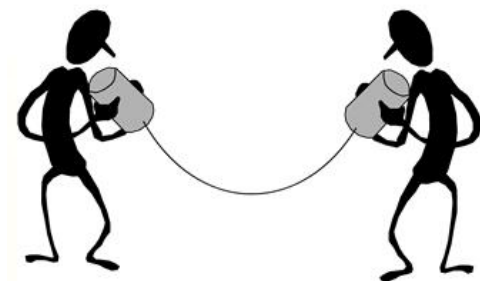
## 2. 第一个 MPI 程序

### ● SPMD

- Single-Program Multiple-Data
- 我们编译了一个程序
- 进程 0 的工作与其他进程不同，接收并打印消息
- 其他进程组织并发送消息
- **if-else** 结构使得我们的程序成为 SPMD

## 2. 第一个 MPI 程序

- 通信 (Communication)



```
int MPI_Send(  
  
    void*          msg_buf_p      /* in */,  
    int            msg_size       /* in */,  
    MPI_Datatype    msg_type      /* in */,  
    int            dest           /* in */,  
    int            tag            /* in */,  
    MPI_Comm        communicator /* in */);
```

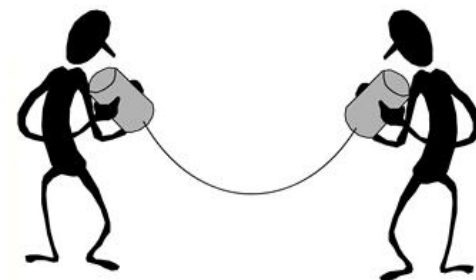
## 2.第一个 MPI 程序

- 通信（Communication）

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

## 2. 第一个 MPI 程序

### ● 通信 (Communication)



```
int MPI_Recv(  
    void*          msg_buf_p    /* out */,  
    int           buf_size     /* in  */,  
    MPI_Datatype   buf_type     /* in  */,  
    int           source       /* in  */,  
    int           tag          /* in  */,  
    MPI_Comm       communicator /* in  */,  
    MPI_Status*   status_p     /* out */);
```



## 2. 第一个 MPI 程序

### ● 消息匹配 (Message matching)

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,  
         send_comm);
```

*MPI\_Send*  
*src = q*



*MPI\_Recv*  
*dest = r*

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,  
         recv_comm, &status);
```

## 2. 第一个 MPI 程序

### ● 接收消息

- 如果 `recv_type = send_type` 且 `recv_buf_sz >= send_buf_sz`, 则进程 `q` 发送的消息能够被进程 `r` 成功接收

```
for (i = 1; i < comm_sz; i++) {  
    MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,  
             result_tag, comm, MPI_STATUS_IGNORE);  
    Process result(result);  
}
```

## 2. 第一个 MPI 程序

### ● 接收消息

- 如果 `recv_type = send_type` 且 `recv_buf_sz >= send_buf_sz`, 则进程 `q` 发送的消息能够被进程 `r` 成功接收

```
for (i = 1; i < comm_sz; i++) {  
    MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,  
             MPI_ANY_TAG, comm, MPI_STATUS_IGNORE);  
    Process result(result);  
}
```

## 2.第一个 MPI 程序

### ●接收消息

- 只有接收者可以使用通配符参数，发件者必须指定进程 rank 和 tag。因此，MPI使用的是“push”通信机制而不是“pull”机制
- 通信器参数没有通配符，发送方和接收方都必须指定通信器

## 2.第一个 MPI 程序

### ●status\_p 参数

➤接收方可以在不知道以下信息的情况下接收信息

- 消息中的数据量
- 信息的发送方
- 信息的标签 (tag)

➤ *接收方如何能获取这些信息？*

## 2. 第一个 MPI 程序

### ● status\_p 参数

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,  
         recv_comm, &status);
```

*MPI\_Status\**



```
MPI_Status* status;
```

```
status.MPI_SOURCE
```

```
status.MPI_TAG
```

*MPI\_SOURCE*

*MPI\_TAG*

*MPI\_ERROR*



## 2. 第一个 MPI 程序

### ● status\_p 参数

```
int MPI_Get_count(  
    MPI_Status* status_p    /* in */,  
    MPI_Datatype type       /* in */,  
    int* count_p            /* out */);
```



## 2.第一个 MPI 程序

### ●MPI\_Send 和 MPI\_Recv 的语义

- 发送进程将组装消息
- 发送进程缓冲消息或阻塞，如果缓冲消息，MPI 系统将消息放入自己的内部存储，MPI\_Send 调用返回；如果阻塞，发送进程在开始发送消息前，MPI\_Send 调用阻塞
- 因此，当 MPI\_Send 返回，并不实际知道消息是否被传输，我们只知道用来存储消息的缓冲区可以被程序重新使用



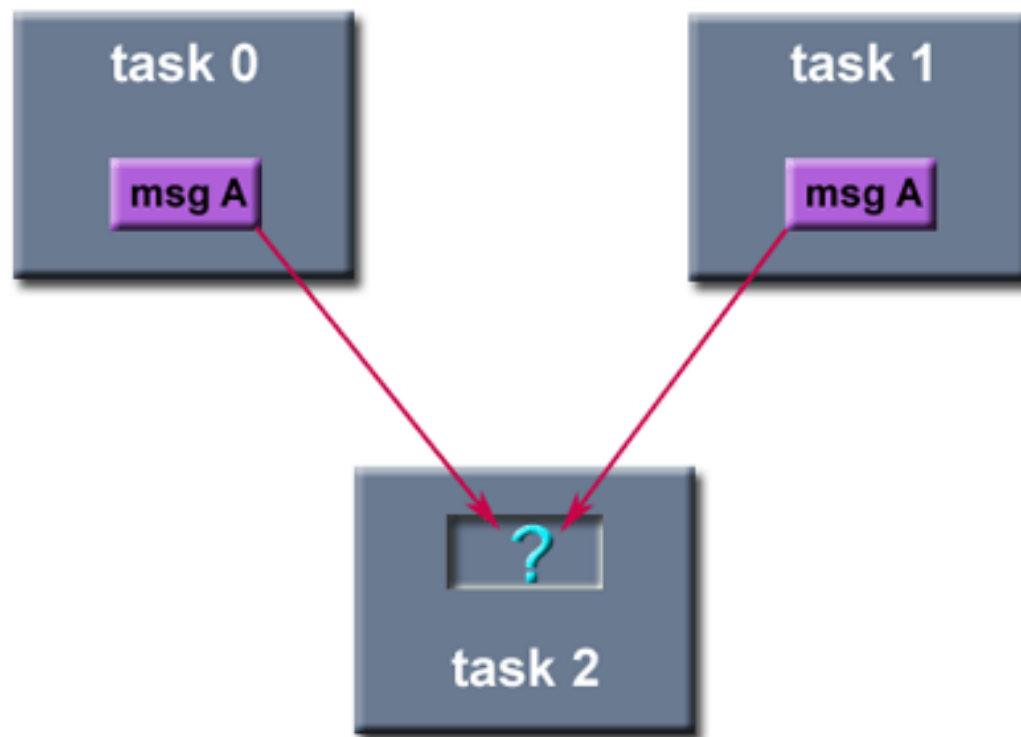
## 2. 第一个 MPI 程序

### ● MPI\_Send 和 MPI\_Recv 的语义

- MPI\_Send 的确切行为由 MPI 的实现决定。通常会有一个默认的信息大小的阈值，如果消息的长度小于该阈值，则被缓冲；如果消息的长度大于该阈值，则 MPI\_Send 被阻塞
- MPI\_Recv 通常会被阻塞，直到匹配的消息被接收。因此，当调用 MPI\_Recv 返回，可以认为有一条消息被存储在接收缓冲区（除非出现了错误）
- MPI 要求消息不超车（nonovertaking）。如果进程 q 发送两条消息给进程 r，则 q 发送的第一条消息要先于第二条消息到达 r。但是对不同进程发送的消息，没有限制。

## 2. 第一个 MPI 程序

- MPI\_Send 和 MPI\_Recv 的语义



## 2. 第一个 MPI 程序

### ● MPI\_Send 和 MPI\_Recv 潜在的问题

- 如果接收方没有匹配的发送方？
- 如果发送方没有匹配的接收方？
- 如果接收方和发送方的参数不匹配？

## 练习

- 配置MPI运行环境
- 运行mpi\_hello.c
- 思考发送和接收过程
- 打印顺序是怎样的？
- 打印的顺序和接收到数据的顺序是一致的吗？