An Introduction to Parallel Programming

Peter Pacheco

Chapter 3

用MPI进行分布式内存编程

Roadmap

- 编写第一个MPI程序.
- 常见的MPI函数.
- 用MPI来实现梯形积分法.
- 集合通信.
- MPI的派生数据类型.
- MPI程序的性能评估.
- 并行排序算法.
- MPI程序的安全性.

参考资料

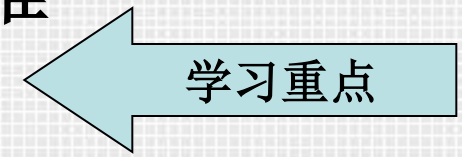
- 黄铠,徐志伟著,陆鑫达等译. 可扩展并行计算技术,结构与编程. 北京:机械工业出版社, P.33~56,P.227~237, 2000.
- 陈国良著.并行计算—结构、算法、编程. 北京:高等教育出版社,1999.
- Barry Wilkinson and Michael Allen. Parallel Programming(Techniques and Applications using Networked Workstations and Parallel Computers). Prentice Hall, 1999.
- 李晓梅,莫则尧等著. 可扩展并行算法的设计与分析. 北京: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

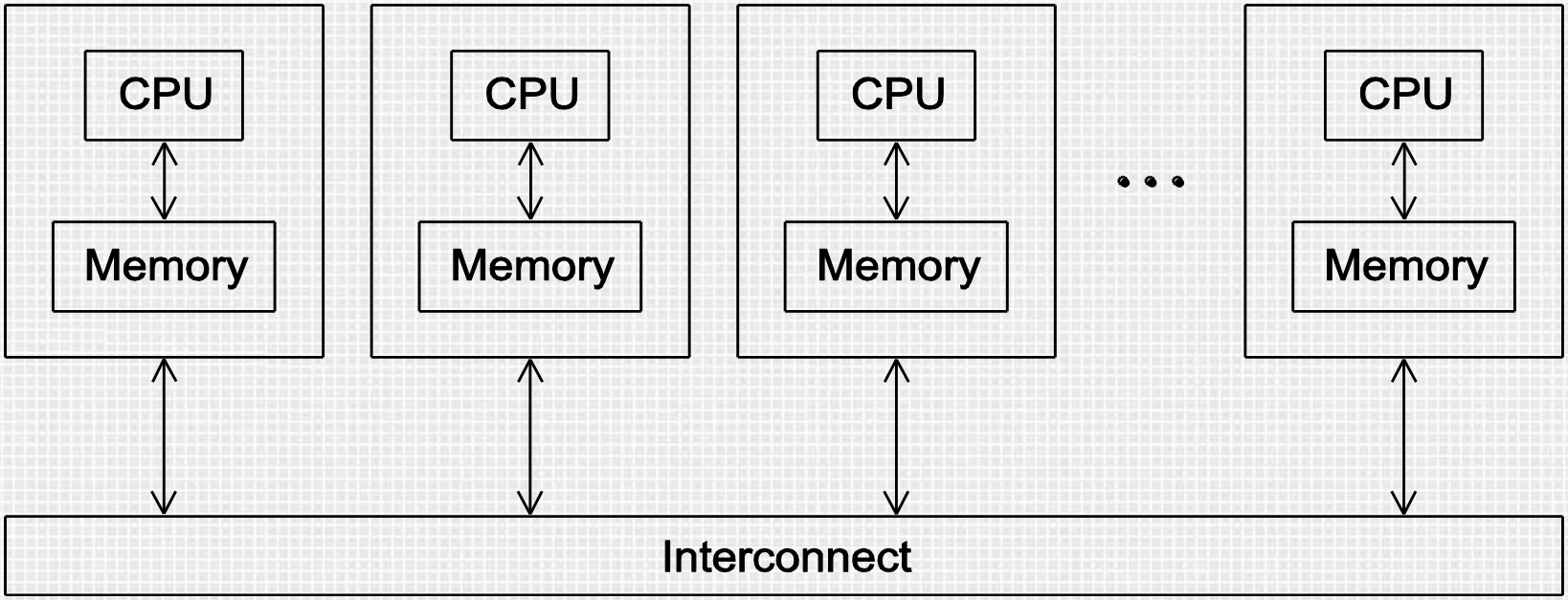
- **MPI:** http://ww.mpi-forum.org, http://www.mcs.anl.gov/mpi
- **Pthreads:** http://www.oreilly.com
- **PVM:** http://www.epm.ornl.gov/pvm/
- **OpemMP:** http://www.openmp.org

并行编程标准

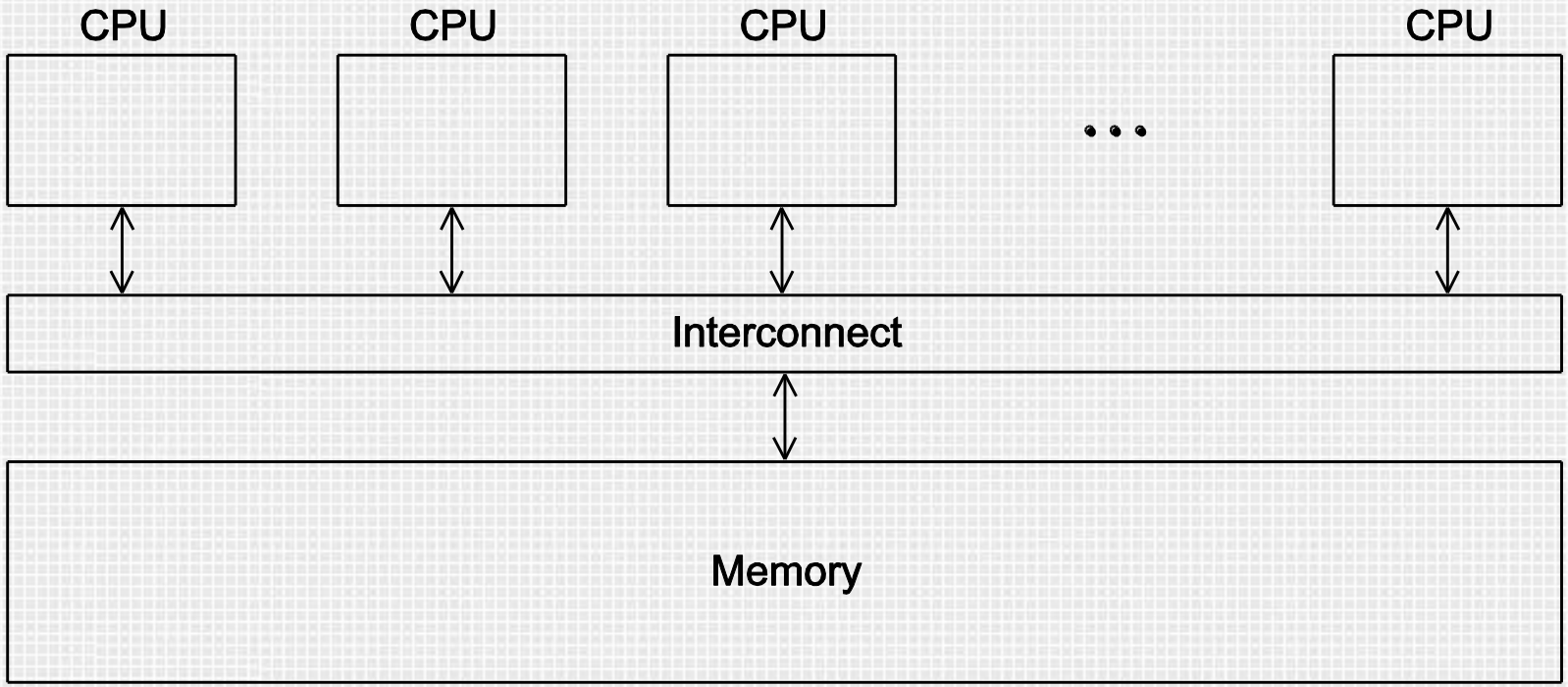
- 多线程库标准
  - – Win32 API.
  - – POSIX threads.
- 编译制导标准
  - – OpenMP – 可移植共享存储并行编程标准.
- 消息传递库标准
  - – MPI
  - – PVM



分布式内存系统



共享内存系统

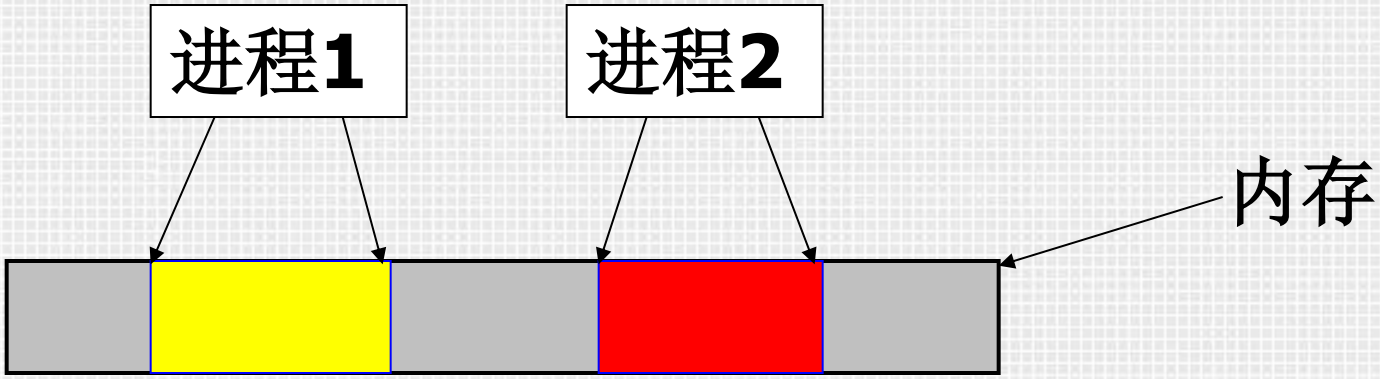


进程

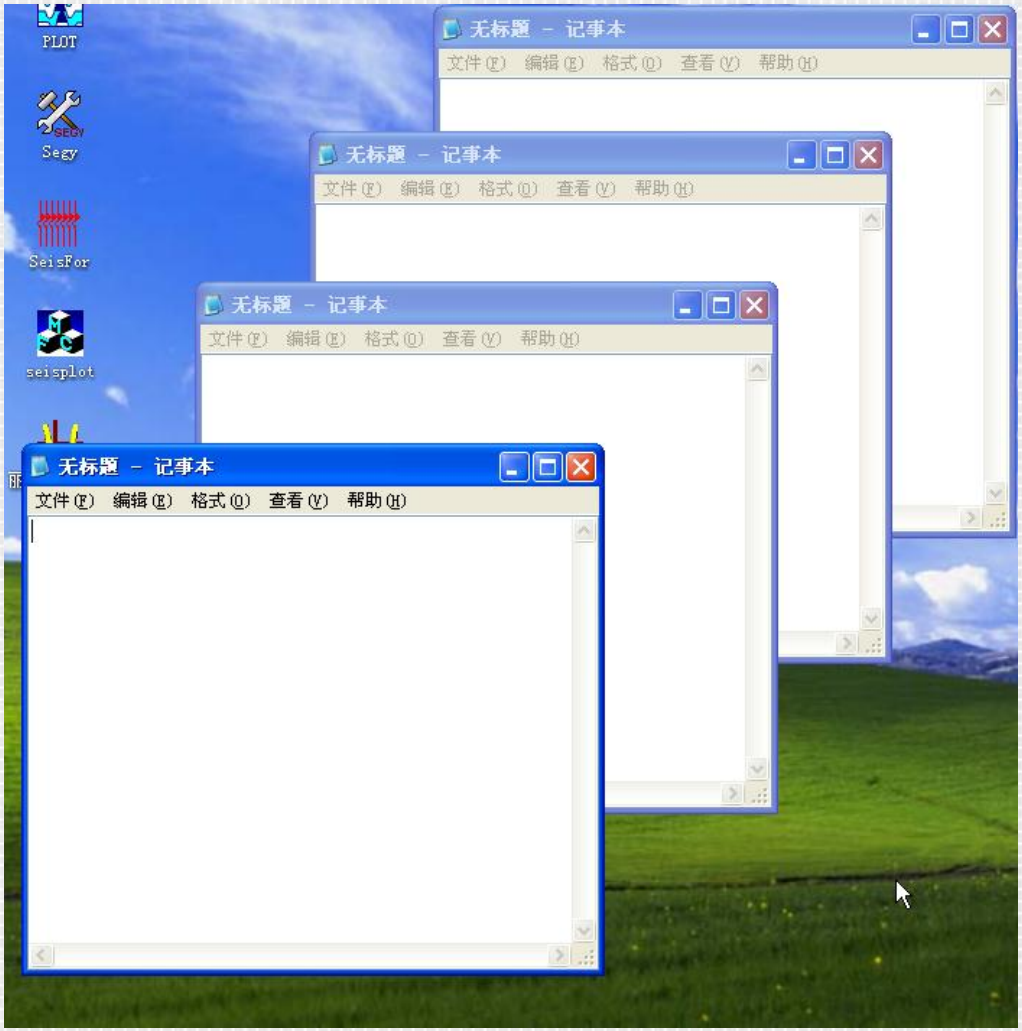
- 进程是一个程序，同时包含它的执行环境（内存、寄存器、程序计数器等），是操作系统中独立存在的可执行的基本程序单位。
- 通俗理解：串行应用程序编译形成的可执行代码，分为“指令”和“数据”两个部分，并在程序执行时“独立地申请和占有”内存空间，且所有计算均局限于该内存空间。

单机内的多个进程

- 多个进程可以同时存在于单机内同一操作系统：由操作系统负责调度分时共享处理机资源（CPU、内存、存储、外设等）。
- 进程间相互独立（内存空间不相交）：在操作系统调度下各自独立地运行，例如多个串行应用程序在同一台计算机中运行。



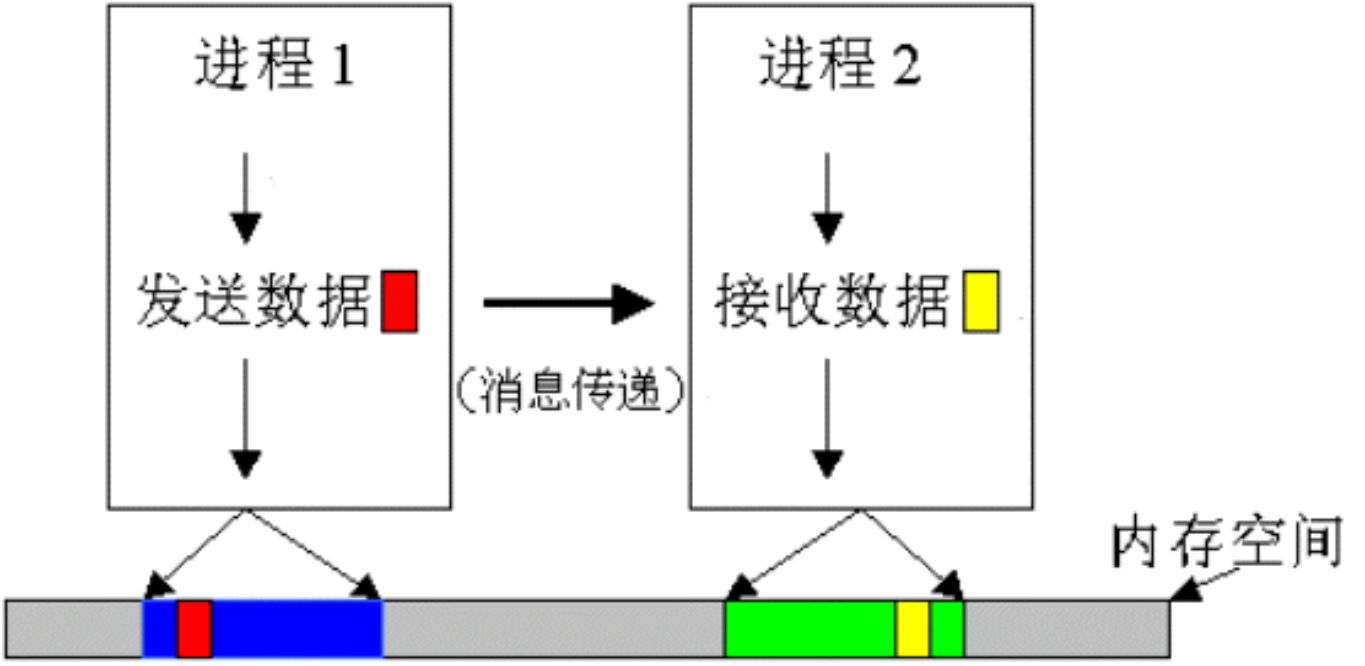
正在微机上运行的进程


PLOT
Segy
SeisFor
seisplot
无标题 - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
无标题 - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
无标题 - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
无标题 - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

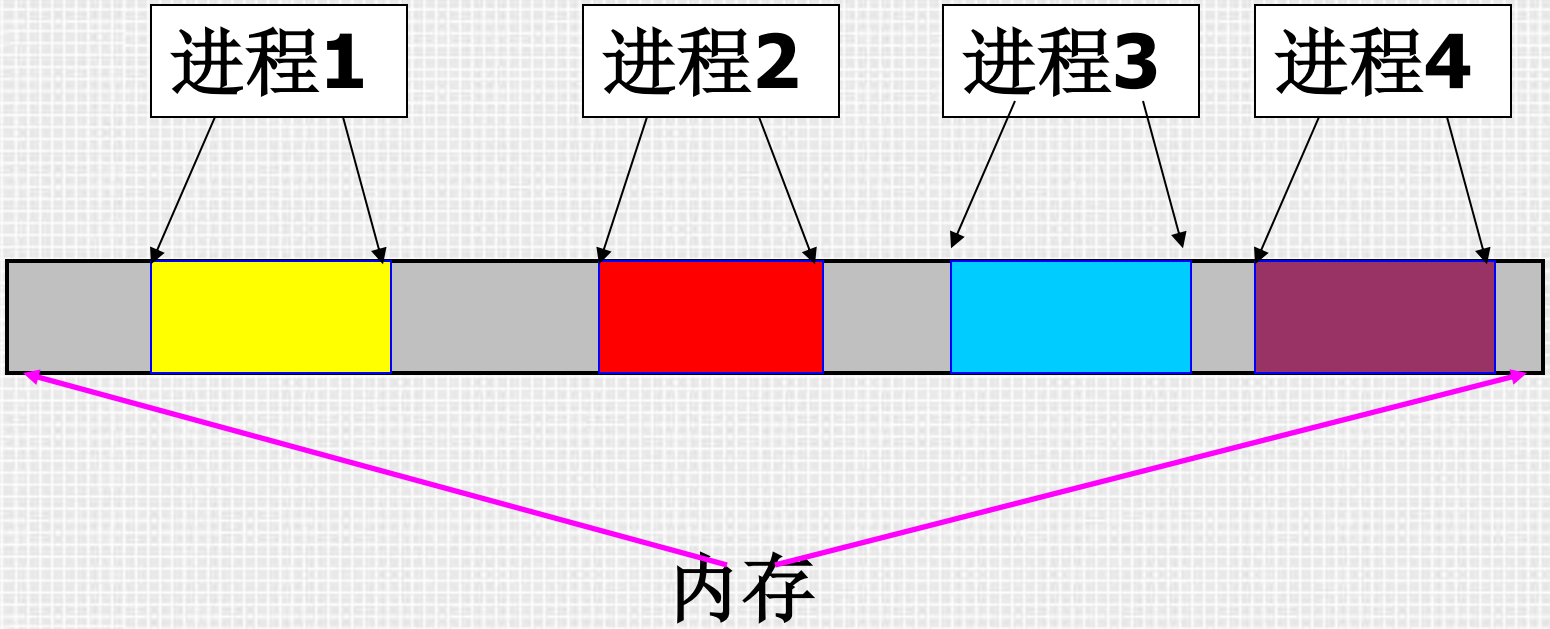
同时运行于单机上的多个进程

多个进程间消息传递

- 最基本的消息传递操作：发送消息（send）、接受消息（receive）、进程同步（barrier）、规约（reduction）。
- 消息传递的实现：共享内存或信号量，用户不必关心。
- 进程间可以相互交换信息：例如数据交换、同步等待，消息是这些交换信息的基本单位，消息传递是指这些信息在进程间的相互交换，是实现进程间通信的唯一方式。



并行程序的单机运行方式



消息传递并行程序设计MPI

- 消息传递并行程序设计
  - 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
  - 适用于大规模并行处理机（MPP）和机群（Cluster）

- 并行计算粒度大，适合大规模可扩展并行算法
  - 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?

- **Massage 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。**
  - 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
  - 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或Fortran并行程序可不加改变地在上述平台实现

MPI的发展过程

- 发展的两个阶段
  - 1994年5月完成1.0版
    - 支持C和Fortran77
    - 制定大部分并行功能
  - 1997年4月完成2.0版
    - 动态进程
    - 并行I/O
    - 支持Fortran 90和C++

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 3.2
  - http://www.mpich.org/
- LAMMPI
  - 美国Indiana 大学Open Systems 实验室实现
  - http://lammps.sandia.gov
- 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

Hello World!

```c
#include <stdio.h>

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

    return 0;
}
```

(a classic)

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

- 该程序在终端打印出Hello World!字样.

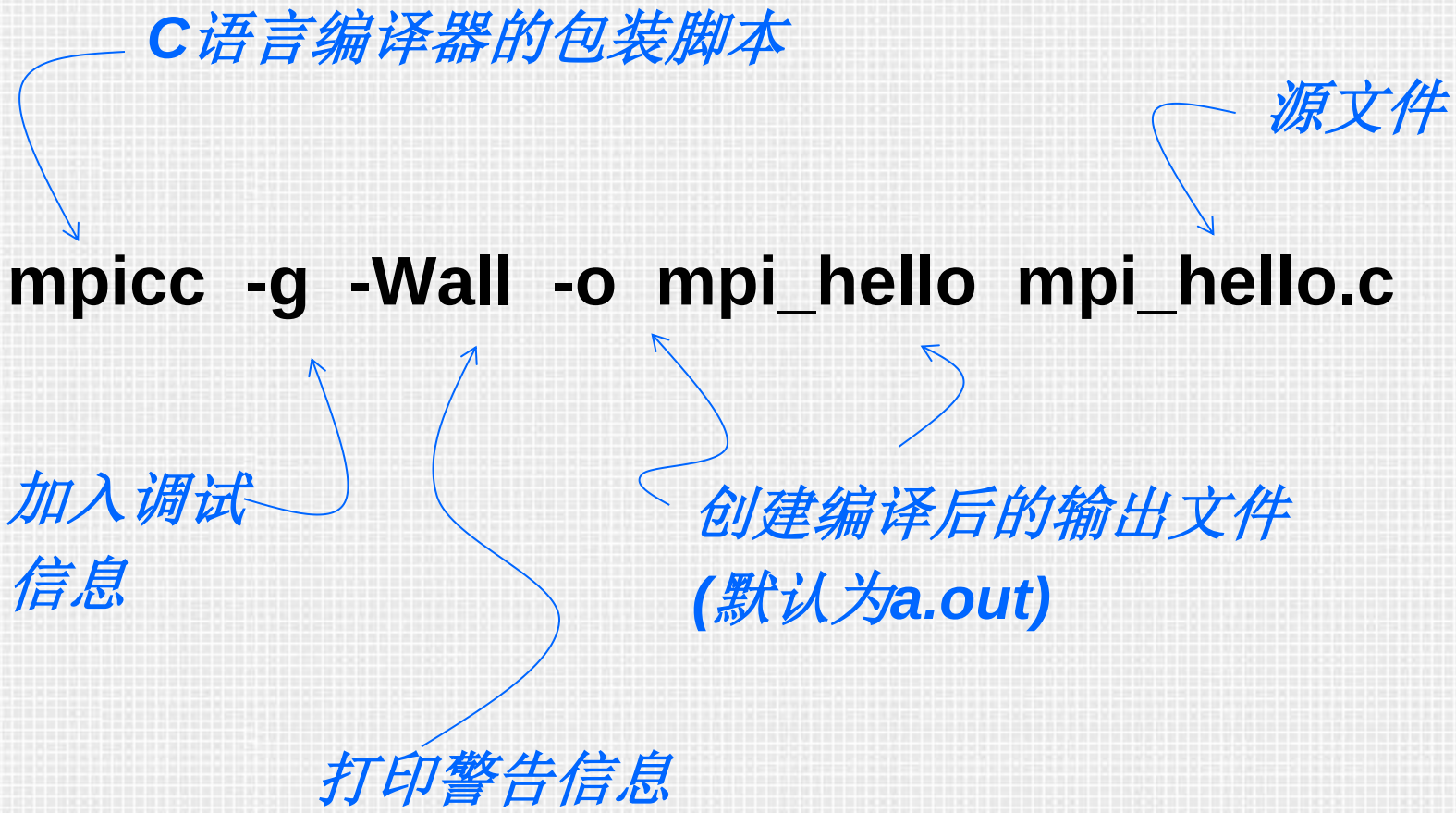
识别 MPI 进程

- 在并行编程中，常见的是将进程按照非负整数来进行标注.
- *p* 个进程被编号为*0, 1, 2, .. p-1*

第一个 MPI 程序

```
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 */
```

编译，生成hello的可执行代码



执行

mpiexec -n <number of processes>
<executable>

---

mpiexec -n 1 ./mpi_hello

用1个进程运行程序

mpiexec -n 4 ./mpi_hello

用4个进程运行程序

Execution

```
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 !
```

3.1.2 MPI 程序

- C语言.
  - 包含了main函数.
  - 标准头文件 stdio.h, string.h, etc.
- 包含 mpi.h 头文件.
- 所有MPI定义的标识符都由字符串 “MPI_” 开始.
- 下划线后的第一字母大写。
  - 表示函数名和MPI定义的类型
  - 避免混淆

MPI程序的框架结构

头文件	包含MPI库
相关变量的声明	定义与通信有关的变量
程序开始	调用MPI初始化函数
程序体 计算与通信	调用MPI其它函数
程序结束	调用MPI结束函数

用C+MPI实现hello world!

```
#include "mpi.h"
#include <stdio.h>     #include <math.h>
void main(int argc,char* argv[])
{    int myid, numprocs namelen;
     char processor_name[MPI_MAX_PROCESSOR_NAME];
     MPI_Init(&argc,&argv);/*程序初始化*/
     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
                                   /*得到当前进程号*/
     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
                                   /*得到总的进程数*/
     MPI_Get_processor_name(processor_name,&namelen);
                                   /*得到机器名*/
     printf("hello world! Process %d of %d on %s\n",
            myid, numprocs, processor_name);
     MPI_Finalize();               /*结束*/   }
```

第一部分

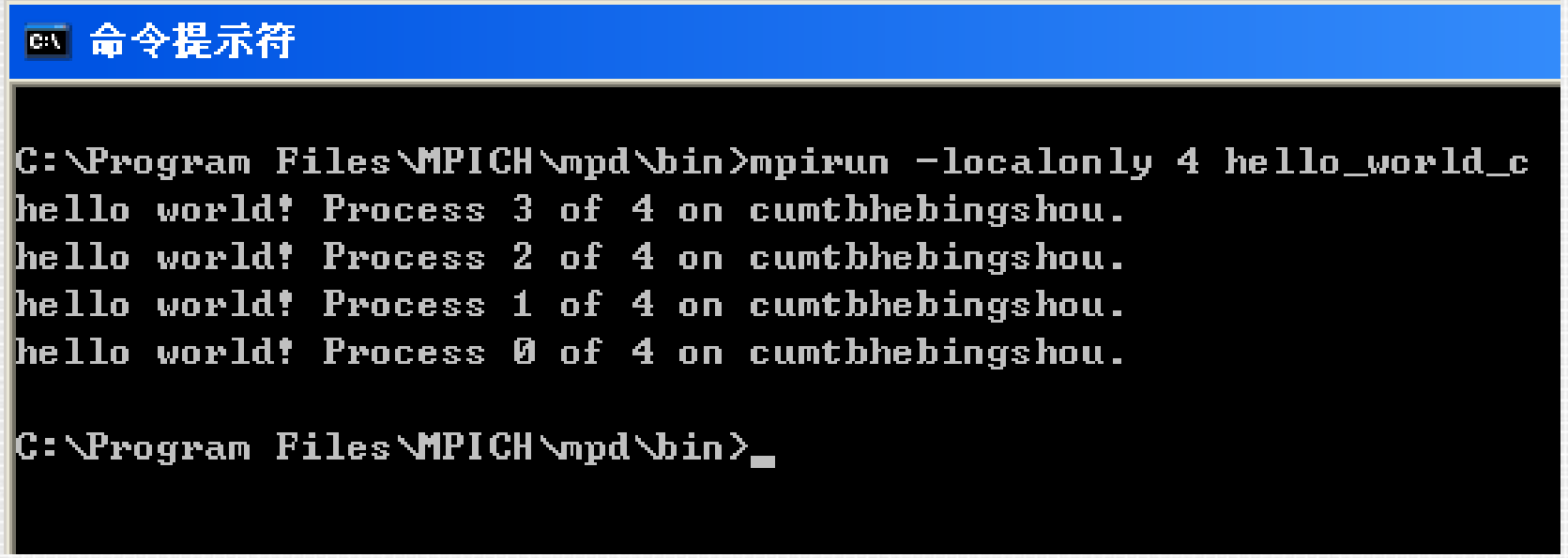
第二部分

第三部分

第四部分

第五部分

执行结果


命令提示符
C:\Program Files\MPICH\mpd\bin>mpirun -localonly 4 hello_world_c
hello world! Process 3 of 4 on cumtbhebingshou.
hello world! Process 2 of 4 on cumtbhebingshou.
hello world! Process 1 of 4 on cumtbhebingshou.
hello world! Process 0 of 4 on cumtbhebingshou.
C:\Program Files\MPICH\mpd\bin>

MPI基本调用函数（1）

- MPI为程序员提供一个并行环境库，程序员通过调用MPI的库程序来达到程序员所要达到的并行目的，可以只使用其中的6个最基本的函数就能编写一个完整的MPI程序去求解很多问题。这6个基本函数，包括启动和结束MPI环境，识别进程以及发送和接收消息：

MPI基本调用函数（2）

- 从理论上说，MPI所有的通信功能可以用它的6个基本的调用来实现:

MPI_INIT:	启动MPI环境
MPI_COMM_SIZE:	确定进程数
MPI_COMM_RANK:	确定自己的进程标识符
MPI_SEND:	发送一条消息
MPI_RECV:	接收一条消息
MPI_FINALIZE:	结束MPI环境

MPI基本调用函数（3）

(1) MPI初始化：通过MPI_Init函数进入MPI环境并完成所有的初始化工作。

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

参数描述：argc为变量数目，argv为变量数组，两个参数均来自main函数的参数

(2) MPI结束：通过MPI_Finalize函数从MPI环境中退出。

int MPI_Finalize(void)

MPI基本调用函数（4）

(3) 获取进程的编号：调用**MPI_Comm_rank**函数获得当前进程在指定通信域中的编号，将自身与其他程序区分。

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

(4) 获取指定通信域的进程数：调用**MPI_Comm_size**函数获取指定通信域的进程个数，确定自身完成任务比例。

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

MPI基本调用函数（5）

(5) 消息发送：MPI_Send函数用于发送一个消息到目标进程。

int MPI_Send(void *buf, int count, MPI_Datatype dataytpe, int dest, int tag, MPI_Comm comm)

(6) 消息接受:MPI_Recv函数用于从指定进程接收一个消息

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

最简单的hello.c

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

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

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

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

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)

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

运行MPI程序 hello.c

- 编译：mpicc -o hello hello.c
- 运行：./hello

  [0] Aborting program ! Could not create p4 procgroup.Possible missing fileor program started without mpirun.
- 运行：mpiexec -np 4 hello

  Hello World!

  Hello World!

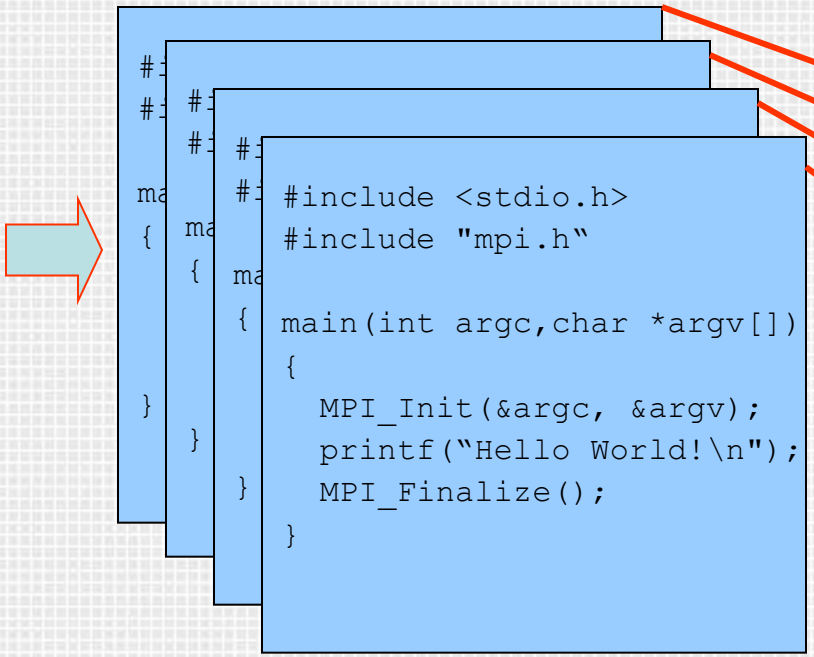
  Hello World!

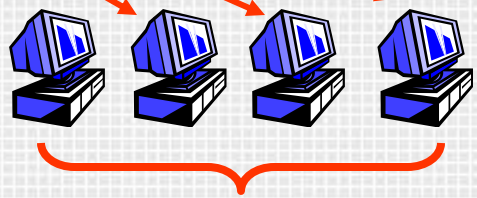
  Hello World!

Hello是如何被执行的？

- **SPMD: Single Program Multiple Data(MIMD)**

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

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

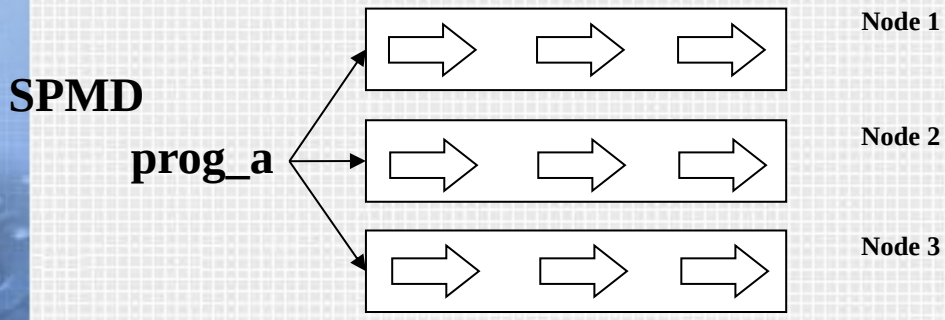




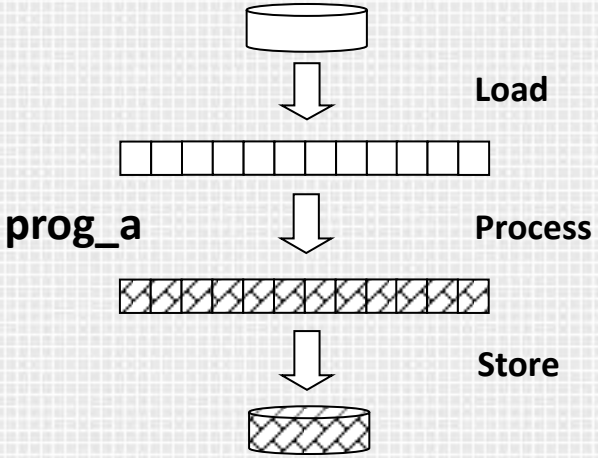
```
Hello World!
Hello World!
Hello World!
Hello World!
```

MPI程序特点

- 消息传递指的是并行执行的各个进程具有自己独立的堆栈和代码段，进程之间的信息交互完全通过显式地调用通信函数来完成。
- 分类：
  - 单程序多数据（Single Program Multiple Data，SPMD）

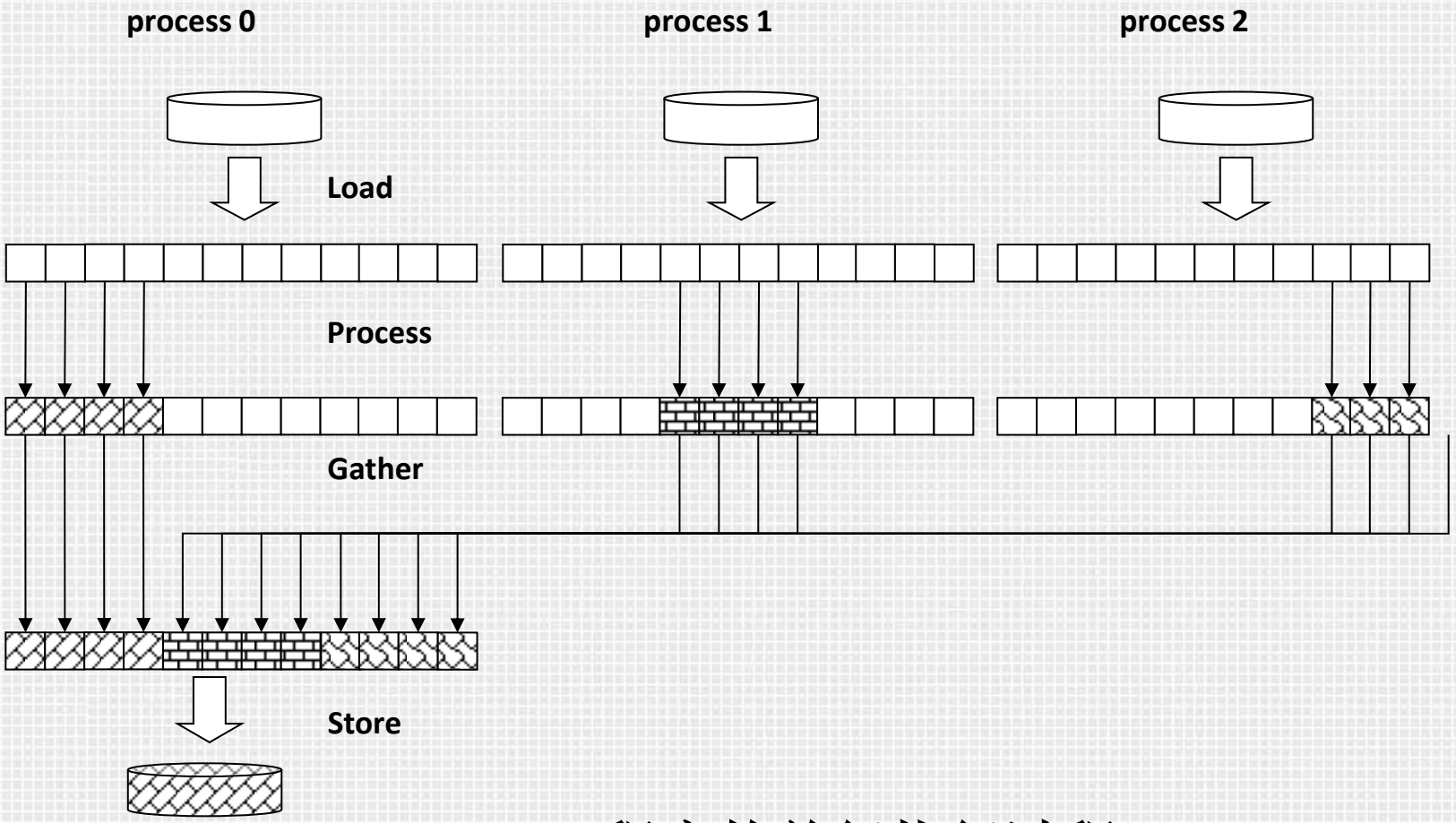


SPMD执行模型



串行处理程序的运行过程

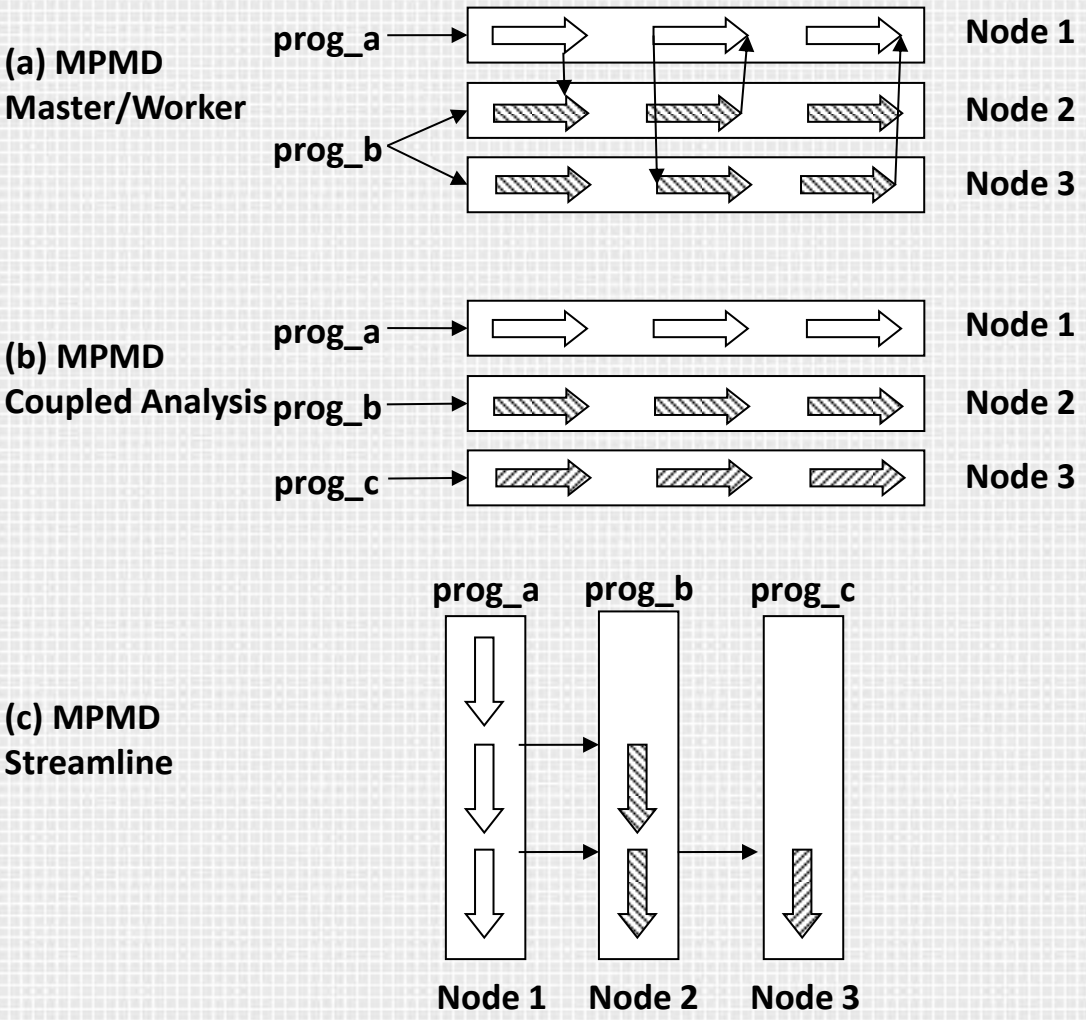
SPMD



SPMD程序的并行执行过程

MPMD

- 多程序多数据MPMD



MPMD执行模型

MPMD（续）

- （a）是一个管理者(Master)/工人(Worker)类型的MPMD程序，由一个Master(prog_a)来控制整个程序的执行，将任务分配给Worker(prog_b)来完成工作。

- （b）联合数据分析程序MPMD程序。在大部分的时间内，不同的程序各自独立的完成自己的任务，并在特定的时候交换数据。耦合性最少，通信也少，更容易获得好的并行加速效果。

- （c）是流式的MPMD程序，程序运行由prog_a、prog_b和prog_c组成，这三个程序之间是典型的串行执行。在这种情况下，并行性的取得依赖于执行大量的任务，通过这种流水线获得性能加速。

基本框架

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
    . . .
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    . . .
    MPI_Finalize();
    /* No MPI calls after this */
    . . .
    return 0;
}
```

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？

通信子

- 一组可以相互发送消息的进程集合.
- **MPI_Init** 在用户启动程序时，定义由用户启动的所有进程所组成的通信子.
- 称为 **MPI_COMM_WORLD**.

- **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 *comm_sz_p)

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

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

int MPI_Comm_rank(MPI_Comm comm, int *my_rank_p)

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

更新的Hello World(C语言)

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

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

    MPI_Init( &argc, &argv );
    MPI_Common_rank(MPI_COMMON_WORLD,&myid);
    MPI_Common_size(MPI_COMMON_WORLD,&numprocs);
    printf("I am %d of %d \n", myid,numprocs);
    MPI_Finalize();
}
```

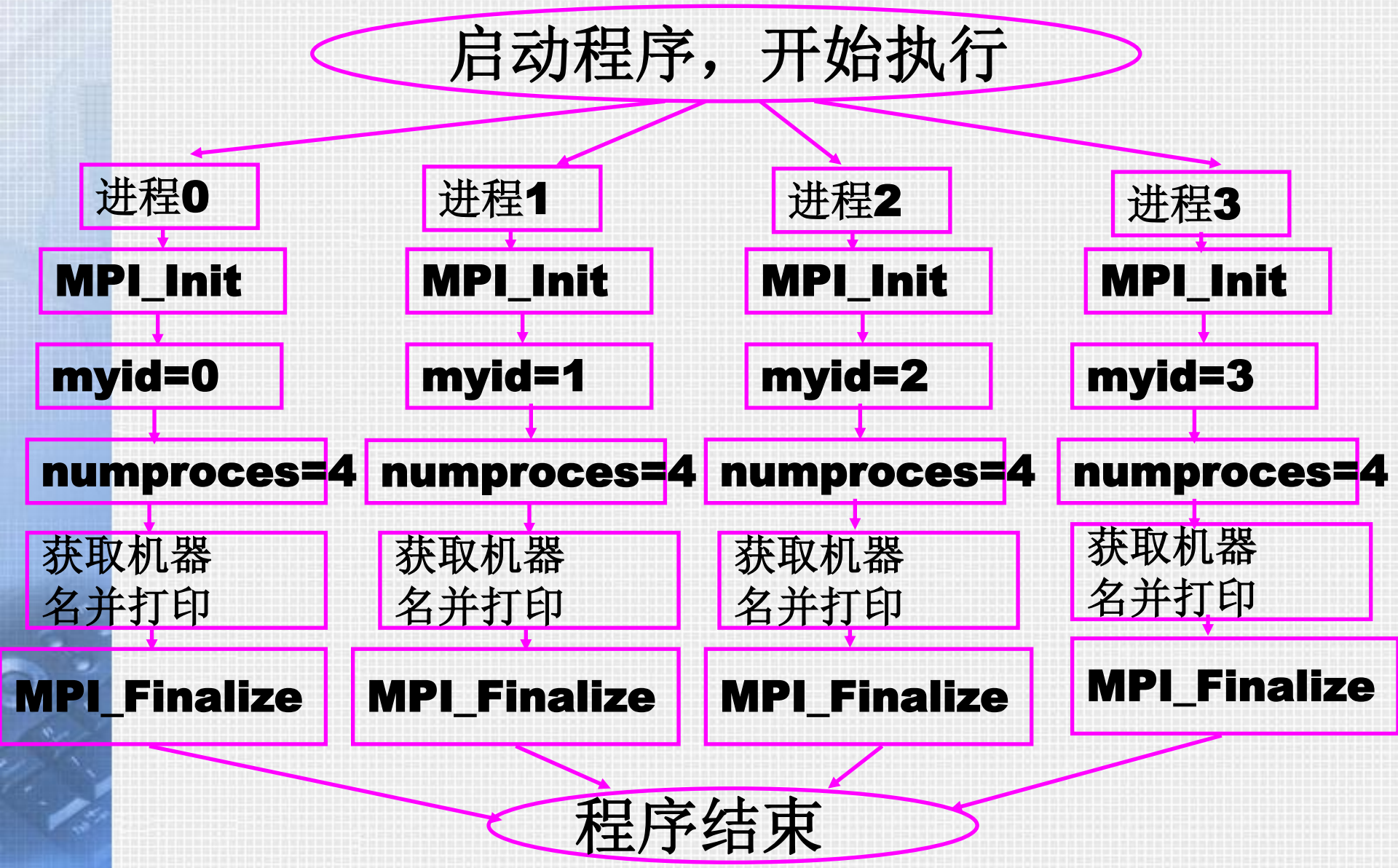
运行结果

- mpicc –o hello1 hello1.c
- mpiexec -np 4 hello1

结果：

I am 0 of 4
I am 1 of 4
I am 2 of 4
I am 3 of 4

Hello程序在单机上的运行方式

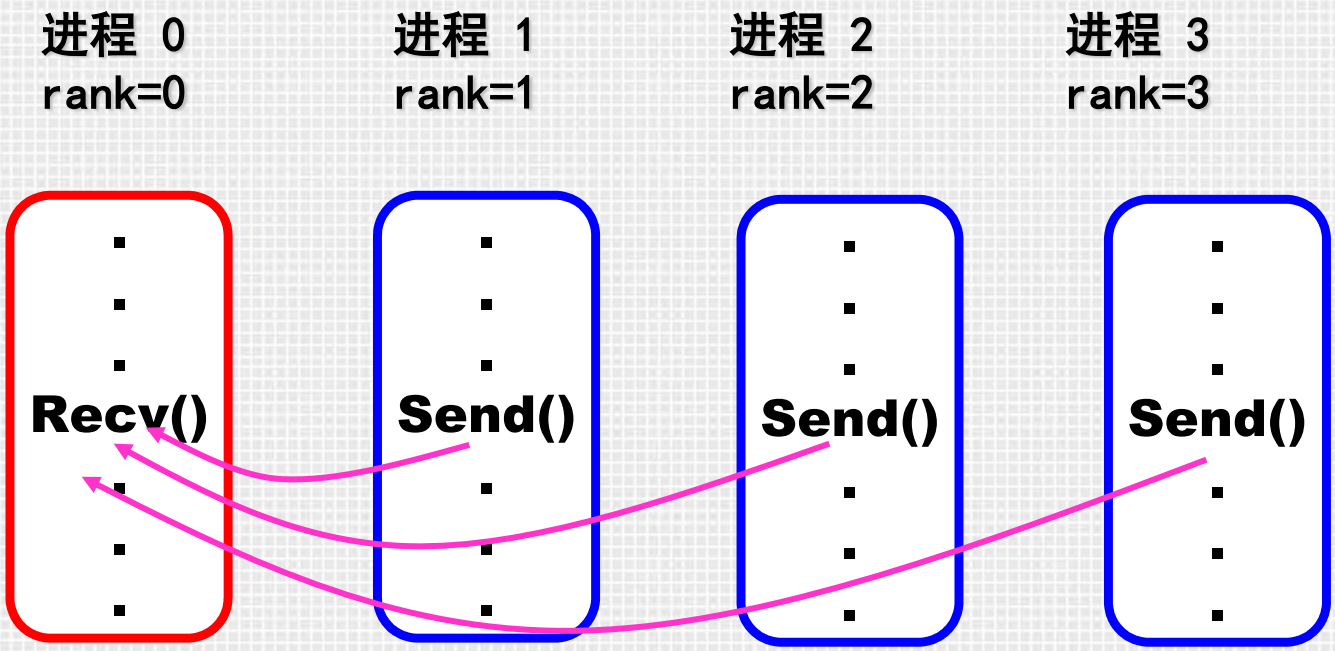


3.1.6 通信
有消息传递greetings(C语言)

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

main( int argc, char *argv[] )
{
    int myid,numprocs,source;
    MPI_Status status;
    char message[100];

    MPI_Init( &argc, &argv );
    MPI_Common_rank(MPI_COMM_WORLD,&myid);
    MPI_Common_size(MPI_COMM_WORLD,&numprocs);
```

```
  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

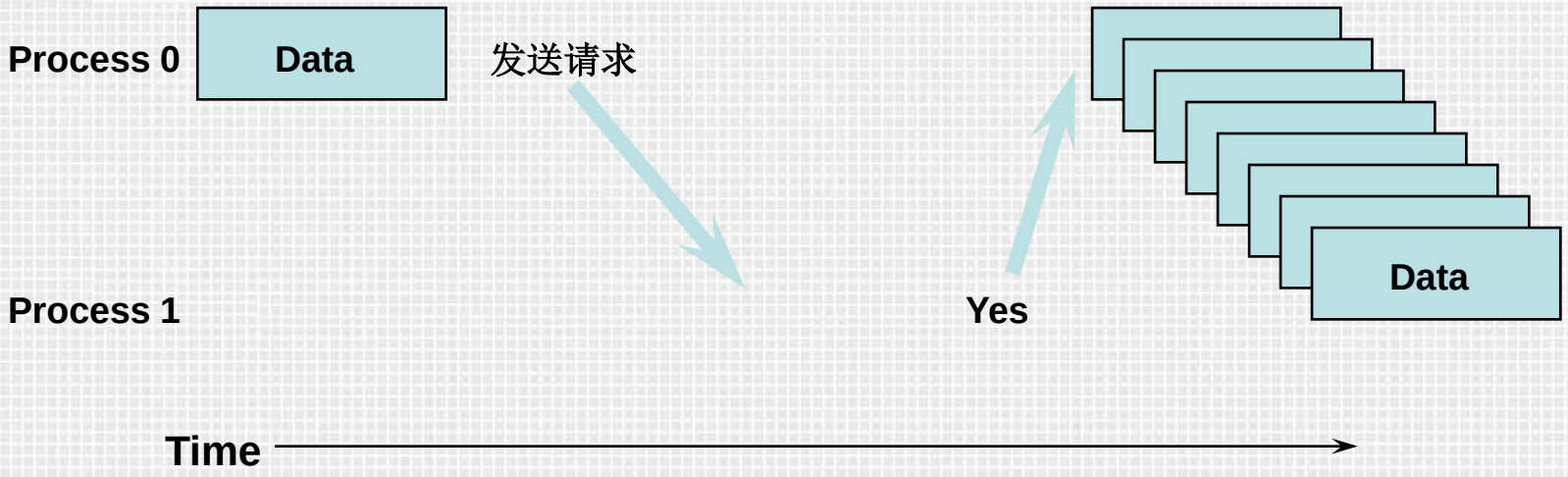
- 头文件：**mpi.h/mpif.h**
- **int MPI_Init(int *argc, char ***argv)**
- 通信组/通信子: **MPI_COMM_WORLD**
  - 一个通信组是一个进程组的集合。所有参与并行计算的进程可以组合为一个或多个通信组
  - 执行**MPI_Init**后，一个**MPI**程序的所有进程形成一个缺省的组,这个组被写作**MPI_COMM_WORLD**
  - 该参数是**MPI**通信操作函数中必不可少的参数,用于限定参加通信的进程的范围

- **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()**

消息传递

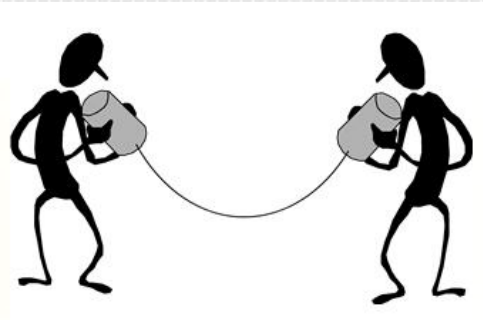
```
MPI_Send(A, 10, MPI_DOUBLE, 1, 99, MPI_COMM_WORLD);

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



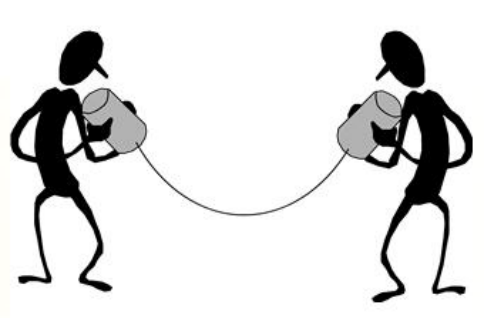
MPI_SEND

```
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 */);
```



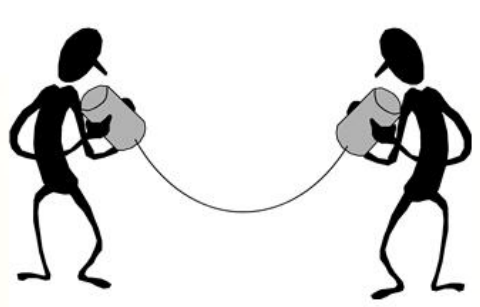
MPI_SEND

- **int MPI_Send(void* msg_buf_p, int msg_size, MPI_Datatype msg_type,int dest, int tag, MPI_Comm communicator);**
  - **IN msg_buf_p** 发送缓冲区的起始地址
  - **IN msg_size** 要发送信息的元素个数
  - **IN msg_type** 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 communicator** 通信组



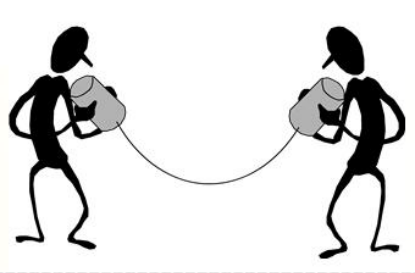
MPI_RECV

```
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 */);
```



MPI_RECV

- **int MPI_Recv(void* msg_buf_p, int buf_size, MPI_Datatype buf_type, int source, int tag, MPI_Comm comminicator, MPI_Status *status_p);**
  - **OUT msg_buf_p** 接收缓冲区的起始地址
  - **IN buf_size** 要接收信息的元素个数
  - **IN buf_type** 接收缓信息的数据类型
  - **IN source** 发送进程的rank值
  - **IN tag** 消息标签
  - **IN comminicator** 通信子
  - **OUT status_p** status_p对象,包含实际接收到的消息的有关信息



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):不依赖于其它进程.**
  - **Non-local(非本地):依赖于其它进程.**

MPI消息

- MPI消息包括信封和数据两个部分，信封指出了发送或接收消息的对象及相关信息，而数据是本消息将要传递的内容

- 数据：<起始地址、数据个数、数据类型>
- 信封：<源/目的、标识、通信域>

消息数据

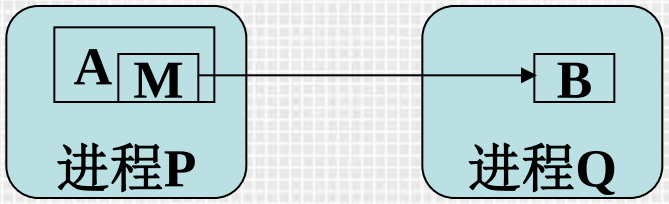
- 由**count**个类型为**datatype**的连续数据空间组成, 起始地址为**buf**
- 不是以字节数, 而是以元素的个数指定消息的长度
- **count**可以是零, 这种情况下消息的数据部分是空的
- **MPI**基本数据类型相应于宿主语言的基本数据类型

一些预先定义的MPI数据类型

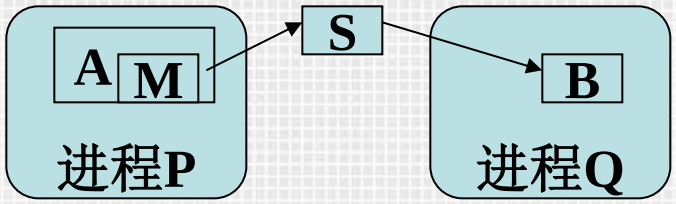
MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned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

什么是缓冲区？

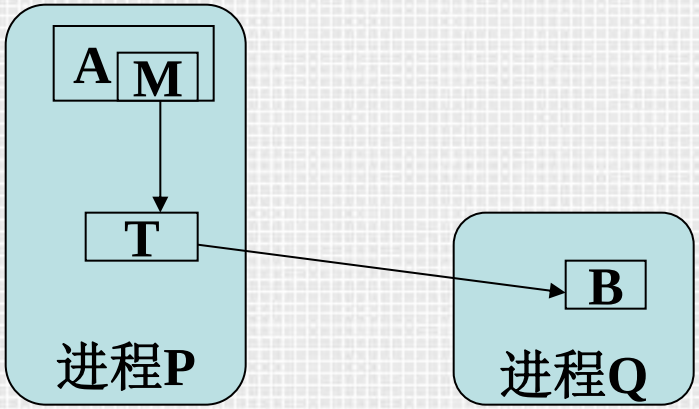
1. 应用程序中说明的变量，在消息传递语句中又用作缓冲区的起始位置。
2. 也可表示由系统(不同用户)创建和管理的某一存储区域，在消息传递过程中用于暂存放消息，也被称为系统缓冲区。
3. 用户可设置一定大小的存储区域，用作中间缓冲区以保留可能出现在其应用程序中的任意消息。



用户缓冲区



系统缓冲区

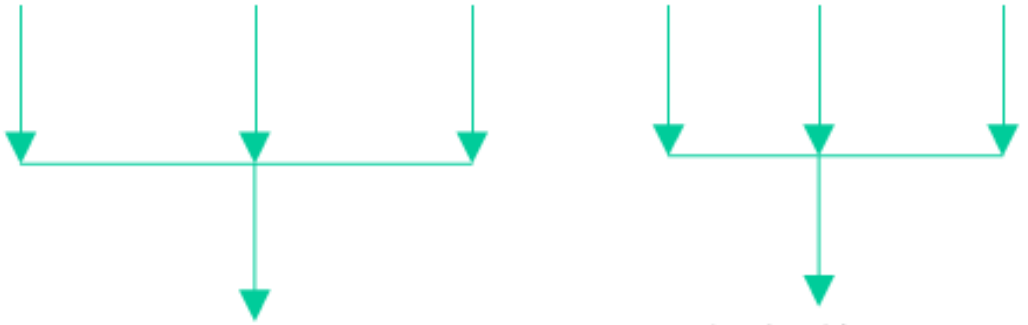


用户指定缓冲区

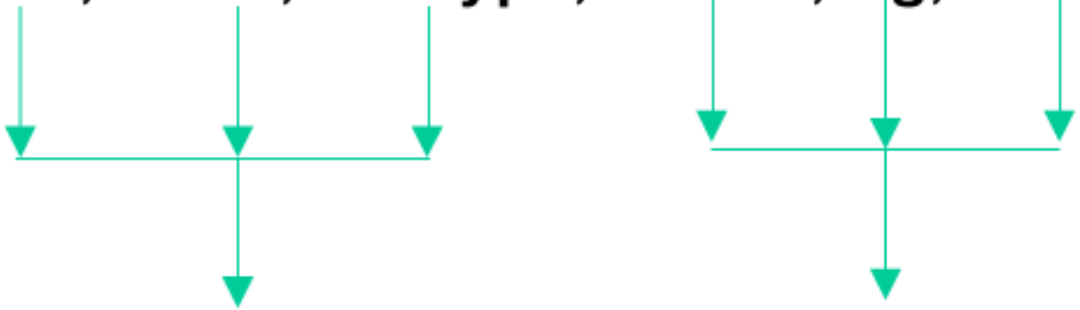
消息信封

- 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,用于识别不同的消息(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.

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



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



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:	Process Q:
send(A,32,Q)	recv(X, 32, P)
send(B,16,Q)	recv(Y, 16, P)

使用了标签

Process P:	Process Q:
send(A,32,Q,tag1)	recv (X, 32, P, tag1)
send(B,16,Q,tag2)	recv (Y, 16, P, tag2)

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) {
      recv (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){
      recv(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;
      }
```

3.1.9 消息匹配

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

r

MPI_Send

src = q

MPI_Recv

dest = r

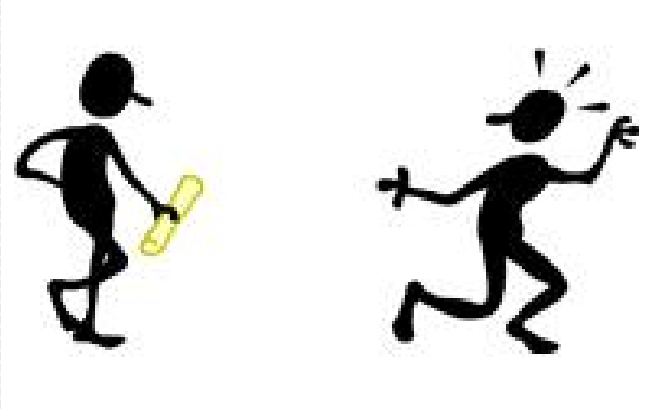
```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

- 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(Push/pull通讯机制).

接收消息

- 接收者可以在不知道以下信息的情况下接收消息:
  - 消息中的数据量,
  - 消息的发送者,
  - 消息的标签.



status_p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.
- Status还可用于返回实际接收到消息的长度
  - int MPI_Get_count(MPI_Status status_p,
    MPI_Datatype type,
    int* count.p)

IN status_p	接收操作的返回值.
IN type	接收缓冲区中元素的数据类型
OUT count.p	接收消息中的元素个数

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

How much data am I receiving?

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

分析greetings

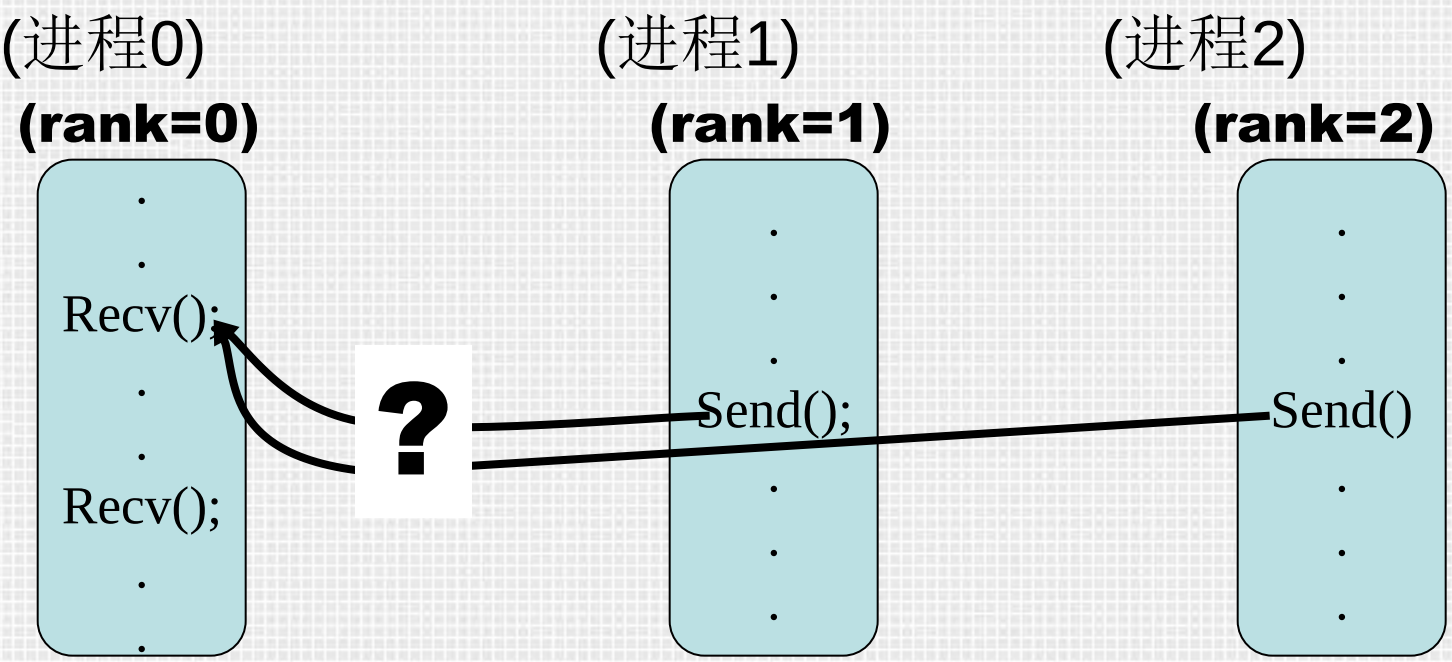
```
#include <stdio.h>
#include "mpi.h"
main( int argc, char *argv[] )
{
    int numprocs;          /*进程数,该变量为各处理器中的同名变量,存储是分布的*/
    int myid;              /*进程ID,存储也是分布的                          */
    MPI_Status status;     /*消息接收状态变量,存储也是分布的                */
    char message[100];     /*消息buffer,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
{/*myrank == 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

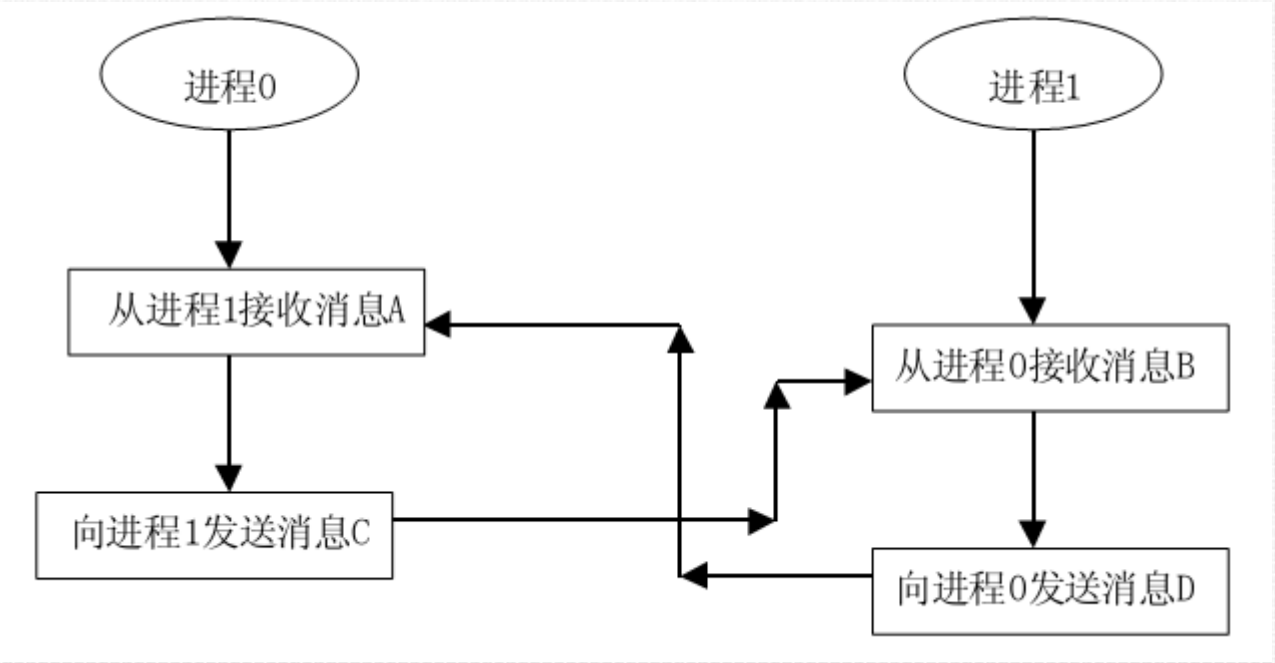
假设进程数为3



问题:进程1和2谁先向根进程发送消息?

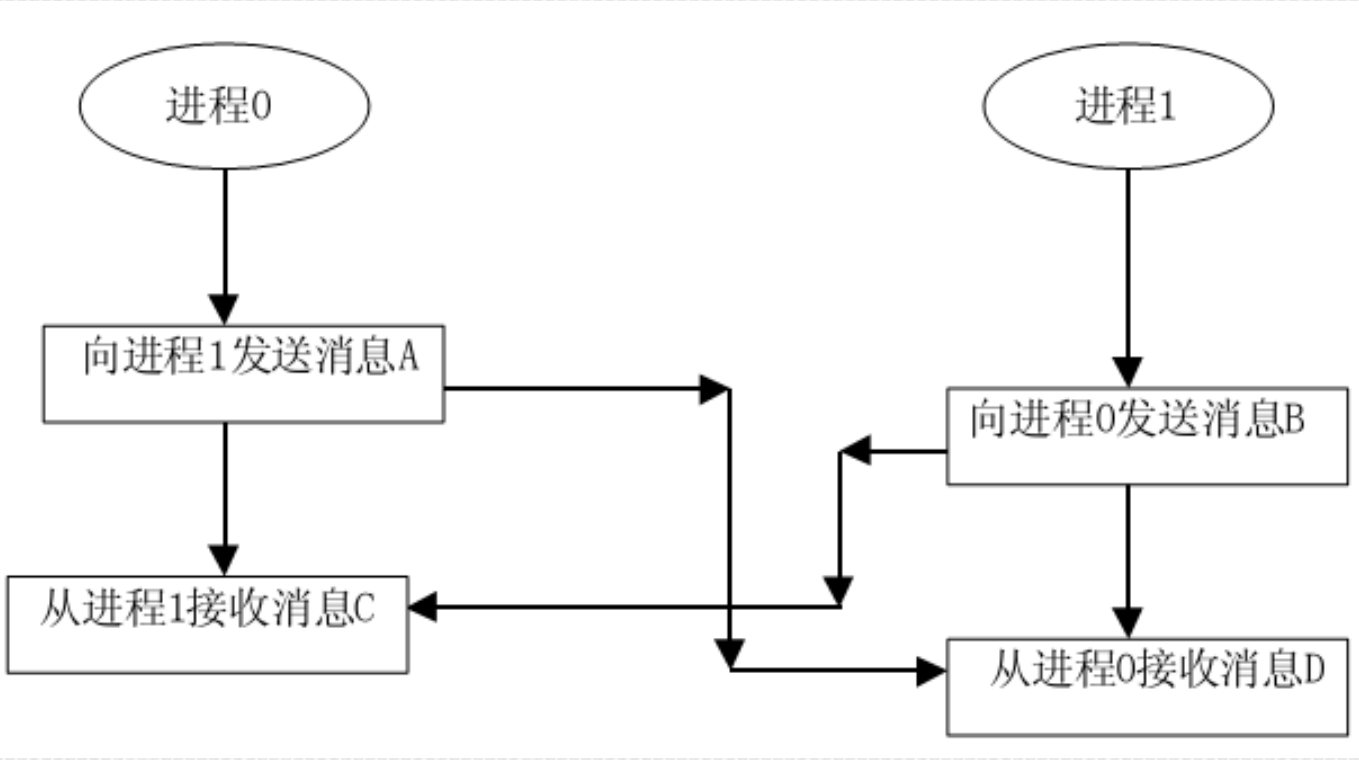
避免死锁deadlock

- 发送和接收是成对出现的，忽略这个原则很可能会产生死锁

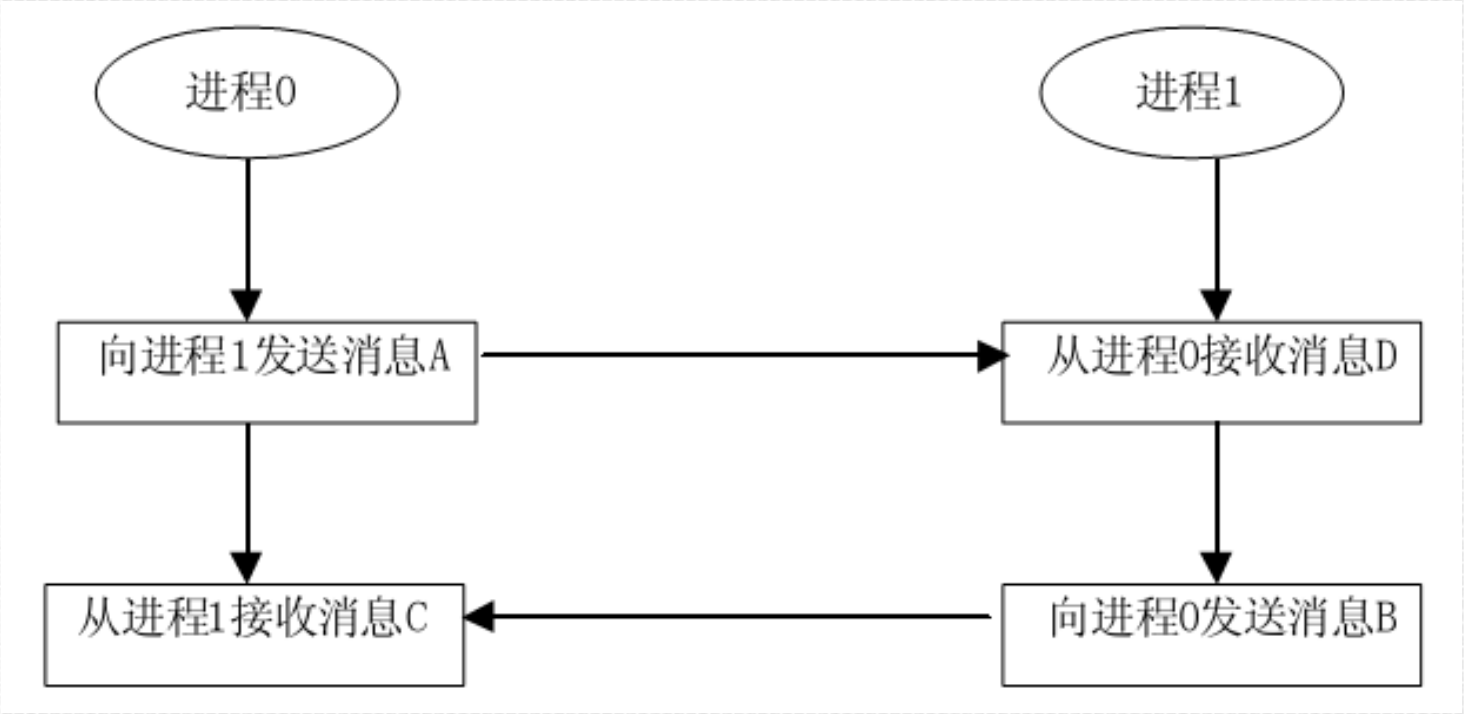


总会死锁的通信调用次序

不安全的通信调用次序

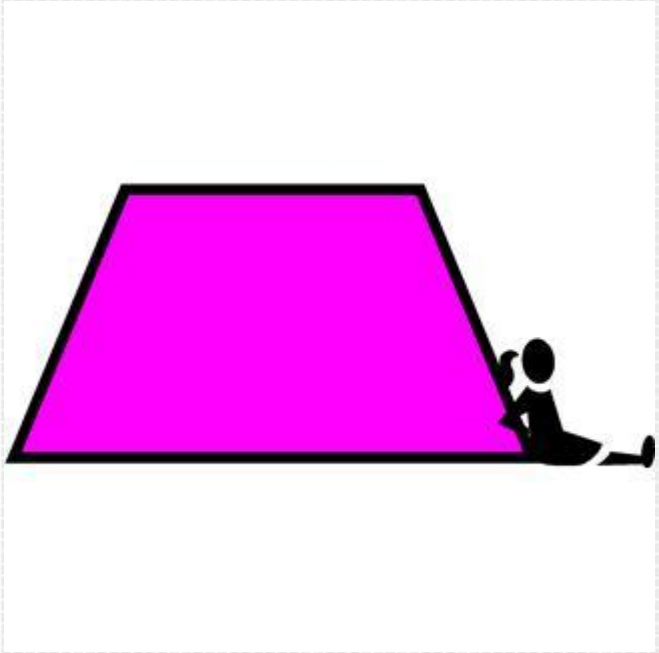


安全的通信调用次序



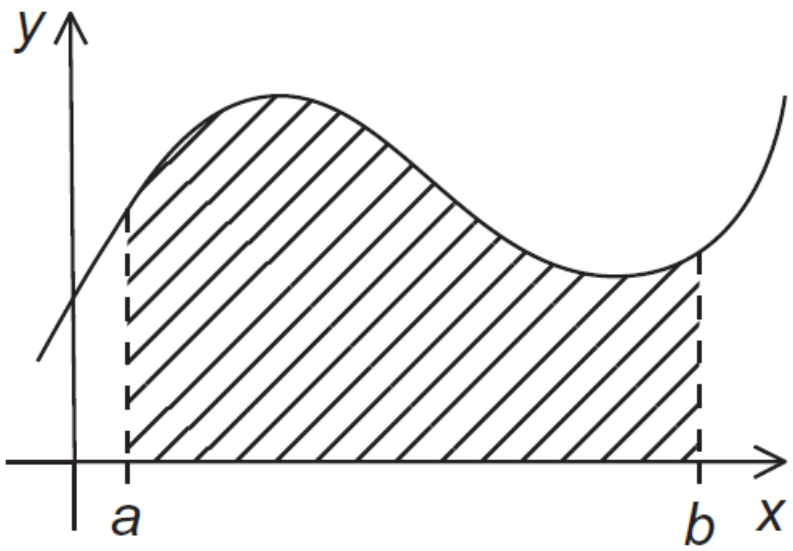
MPI_Send和MPI_Recv的问题

- MPI_Send的精确行为是由MPI实现决定的
- MPI_Send可能有不同大小的缓冲区，是缓冲还是阻塞可以由一个消息“截止”大小决定（cutoffs message size）
- MPI_Recv总是被阻塞的，直到接收到一条匹配的消息
- 了解你的执行情况；不要做假设！

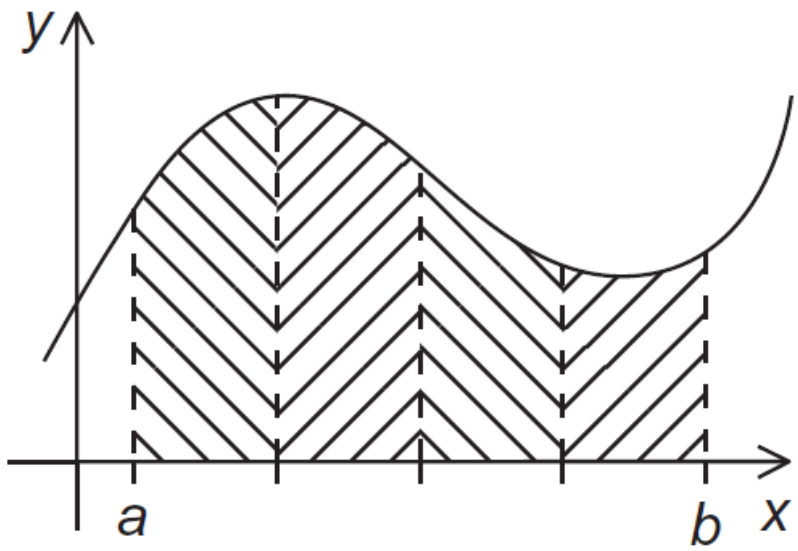


3.2 用MPI实现梯形积分法

梯形积分法

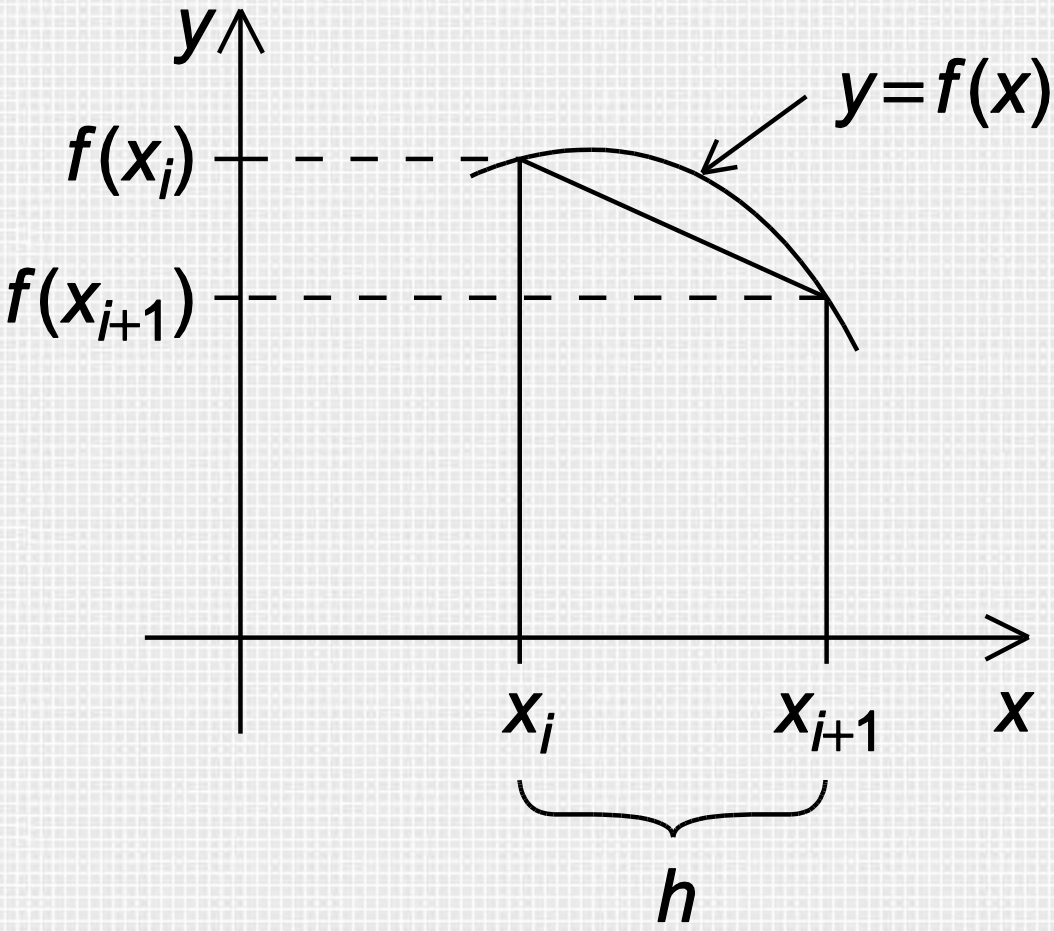


(a)



(b)

一个梯形



梯形积分法

$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h,\ \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b$$

$$\text{Sum of trapezoid areas } = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

梯形积分法的串行伪代码

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 0; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

并行化的梯形积分法

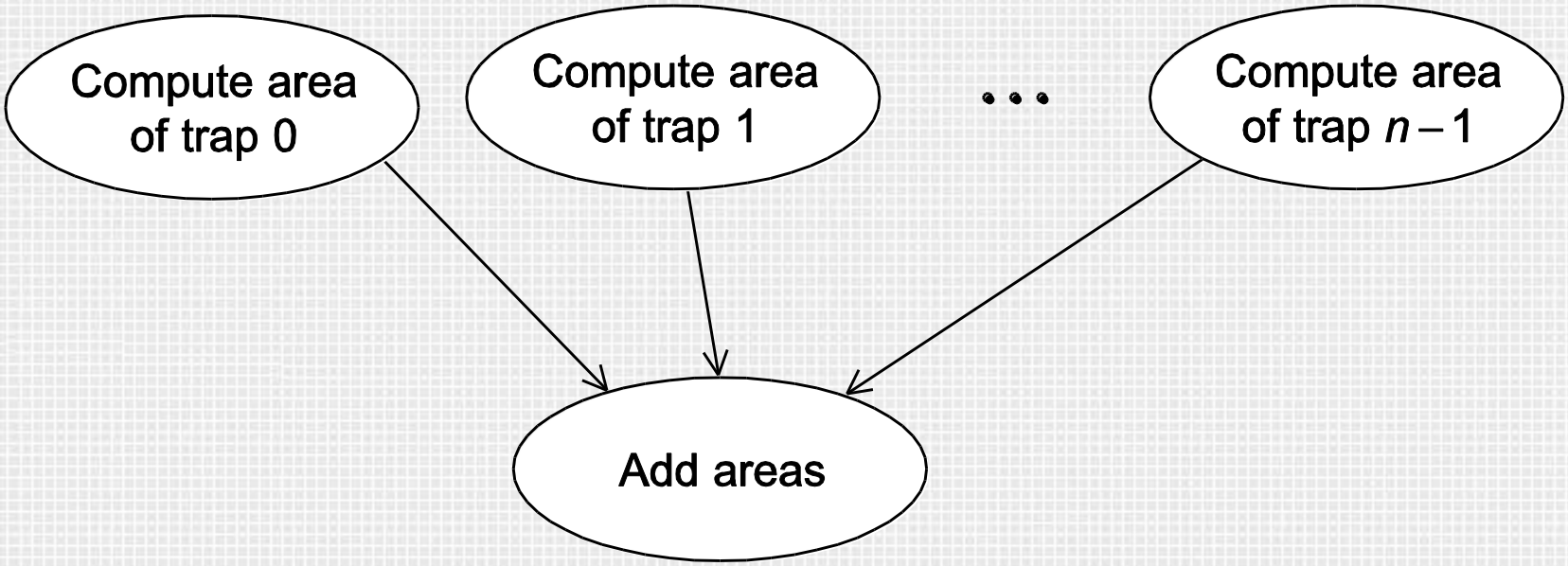
1. 将问题的解决方案划分成多个任务。

2. 在任务间识别出需要的通信信道。

3. 将任务聚合成复合任务。

4. 在核上分配复合任务。

并行梯形积分法的任务和通信



梯形积分法的并行伪代码

```
1     Get a, b, n;
2     h = (b-a)/n;
3     local_n = n/comm_sz;
4     local_a = a + my_rank*local_n*h;
5     local_b = local_a + local_n*h;
6     local_integral = Trap(local_a, local_b, local_n, h);
7     if (my_rank != 0)
8        Send local_integral to process 0;
9     else /* my_rank == 0 */
10       total_integral = local_integral;
11       for (proc = 1; proc < comm_sz; proc++) {
12          Receive local_integral from proc;
13          total_integral += local_integral;
14       }
15    }
16    if (my_rank == 0)
17       print result;
```

并行代码C语言

```
1  int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;           /* h is the same for all processes */
12    local_n = n/comm_sz;   /* So is the number of trapezoids  */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19       MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
```

并行代码C语言

```
21      } else {
22         total_int = local_int;
23         for (source = 1; source < comm_sz; source++) {
24            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            total_int += local_int;
27         }
28      }
29
30      if (my_rank == 0) {
31         printf("With n = %d trapezoids, our estimate\n", n);
32         printf("of the integral from %f to %f = %.15e\n",
33             a, b, total_int);
34      }
35      MPI_Finalize();
36      return 0;
37   } /*  main  */
```

Trap函数

```
1  double Trap(
2         double left_endpt  /* in */,
3         double right_endpt /* in */,
4         int    trap_count  /* in */,
5         double base_len    /* in */) {
6     double estimate, x;
7     int i;
8
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10    for (i = 1; i <= trap_count-1; i++) {
11       x = left_endpt + i*base_len;
12       estimate += f(x);
13    }
14    estimate = estimate*base_len;
15
16    return estimate;
17 } /*  Trap  */
```

3.3 I/O处理

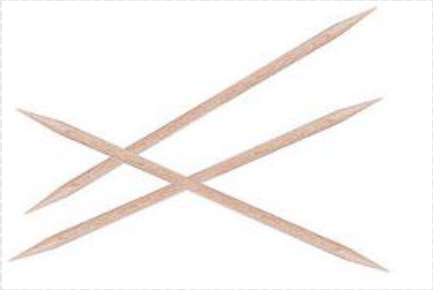
```
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
          my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

每个进程只打印一条消息

Running with 6 processes

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

输出不确定性:
输出的顺序是无法预测的



输入

- 大部分的MPI实现只允许MPI_COMM_WORLD 中的0号进程访问标准输入stdin。
- 0号进程负责读取数据(scanf)，并将数据发送给其他进程。

Get_input →

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

一个用于读取用户输入的函数

```
void Get_input(
      int       my_rank   /* in  */,
      int       comm_sz   /* in  */,
      double*   a_p       /* out */,
      double*   b_p       /* out */,
      int*      n_p       /* out */) {
   int dest;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
   } else { /* my_rank != 0 */
      MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
   }
}  /* Get_input */
```

3.4 集合通信

通信类型

- **MPI点对点通信**
  - 阻塞型和非阻塞型**Blocking，Non-Blocking**
- **MPI集合通信**

MPI点对点通信

- 阻塞型通信
  - 阻塞型通信函数需要等待指定的操作实际完成，或所涉及的数据被 **MPI** 系统安全备份后才返回。
    - **Memory referenced is ready for reuse, Non local operation**
    - **MPI_SEND, MPI_RECV**

MPI点对点通信

- 非阻塞通信
  - 非阻塞型通信函数总是立即返回，实际操作由 **MPI** 后台进行，需要调用其它函数来查询通信是否完成。
    - **Local operation**
    - 在实际操作完成之前对相关数据区域的操作是不安全的
    - 在有些并行系统上**(Communication processors)**，使用非阻塞型函数
    - 可以实现计算与通信的重叠进行
    - **MPI_ISEND, MPI_IRECV**

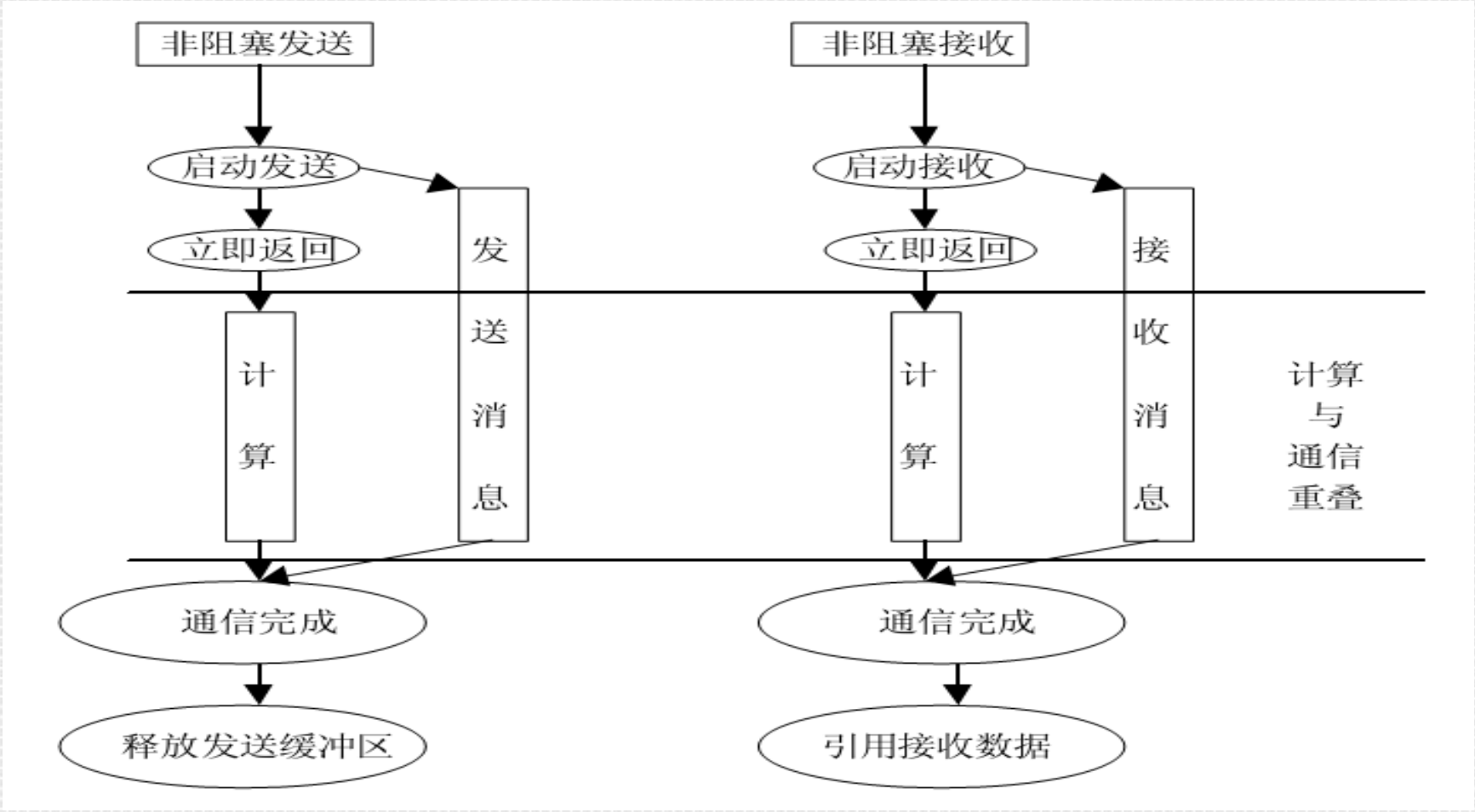
阻塞与非阻塞的差别

- 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rank值
  - 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)

非阻塞标准发送和接收


非阻塞发送
启动发送
立即返回
计算
发送消息
通信完成
释放发送缓冲区
非阻塞接收
启动接收
立即返回
计算
接收消息
通信完成
引用接收数据
计算与通信重叠

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

MPI集合通信

- 集合通信 (collective communication)是一个进程组中的所有进程都参加的全局通信操作。

- 按照通信方向的不同，集合通信可分为三种类型：
  - 一对多：一个进程向其它所有的进程发送消息，这个负责发送消息的进程叫做Root进程。
  - 多对一：一个进程负责从其它所有的进程接收消息，这个接收的进程也叫做Root进程。
  - 多对多：每一个进程都向其它所有的进程发送或者接收消息。

MPI集合通信

- 集合通信一般实现三个功能：
  - 通信，同步和计算

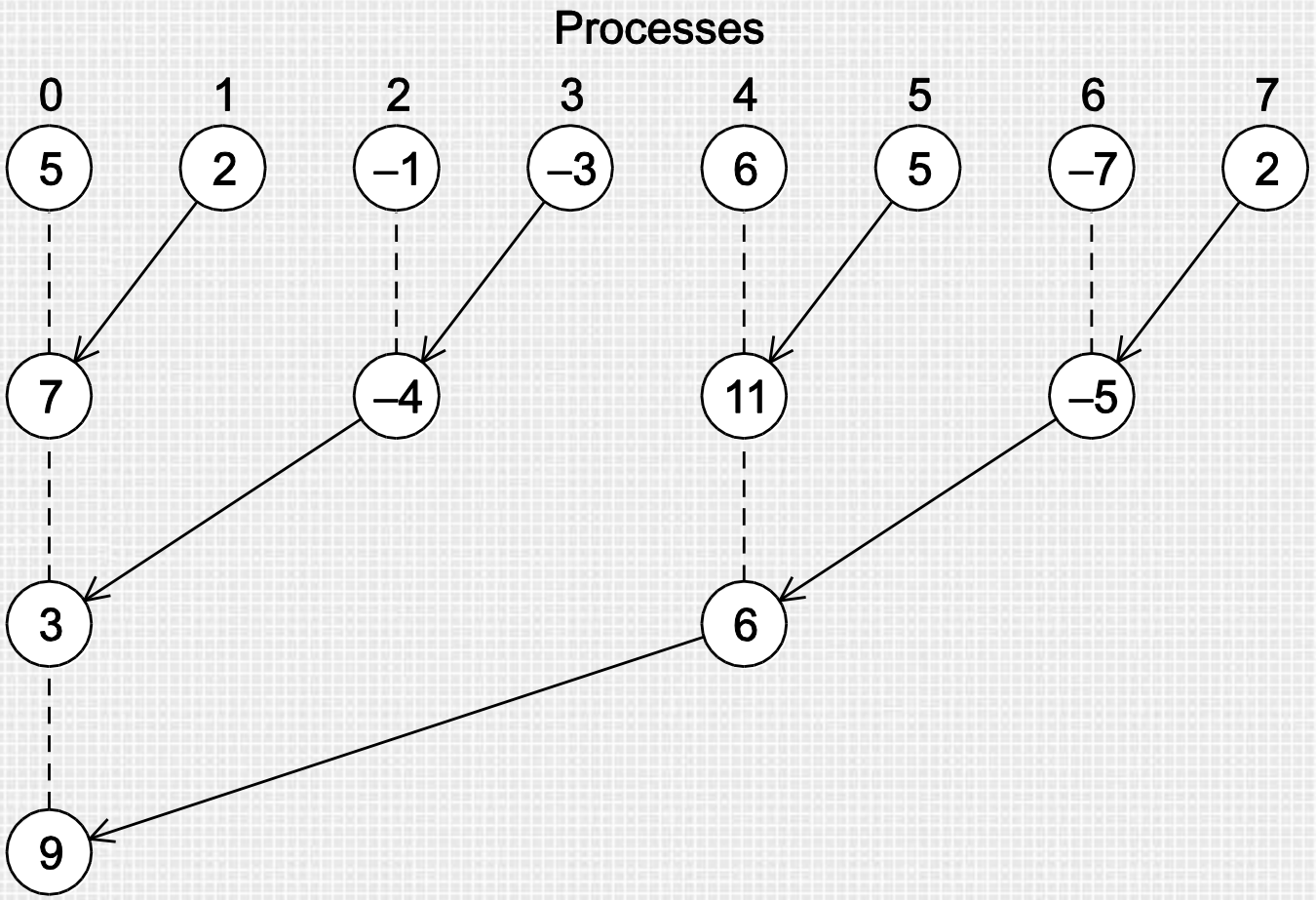
    - 通信功能主要完成组内数据的传输
    - 聚集功能在通信的基础上对给定的数据完成一定的操作
    - 同步功能实现组内所有进程在执行进度上取得一致

3.4.1 树形结构通信

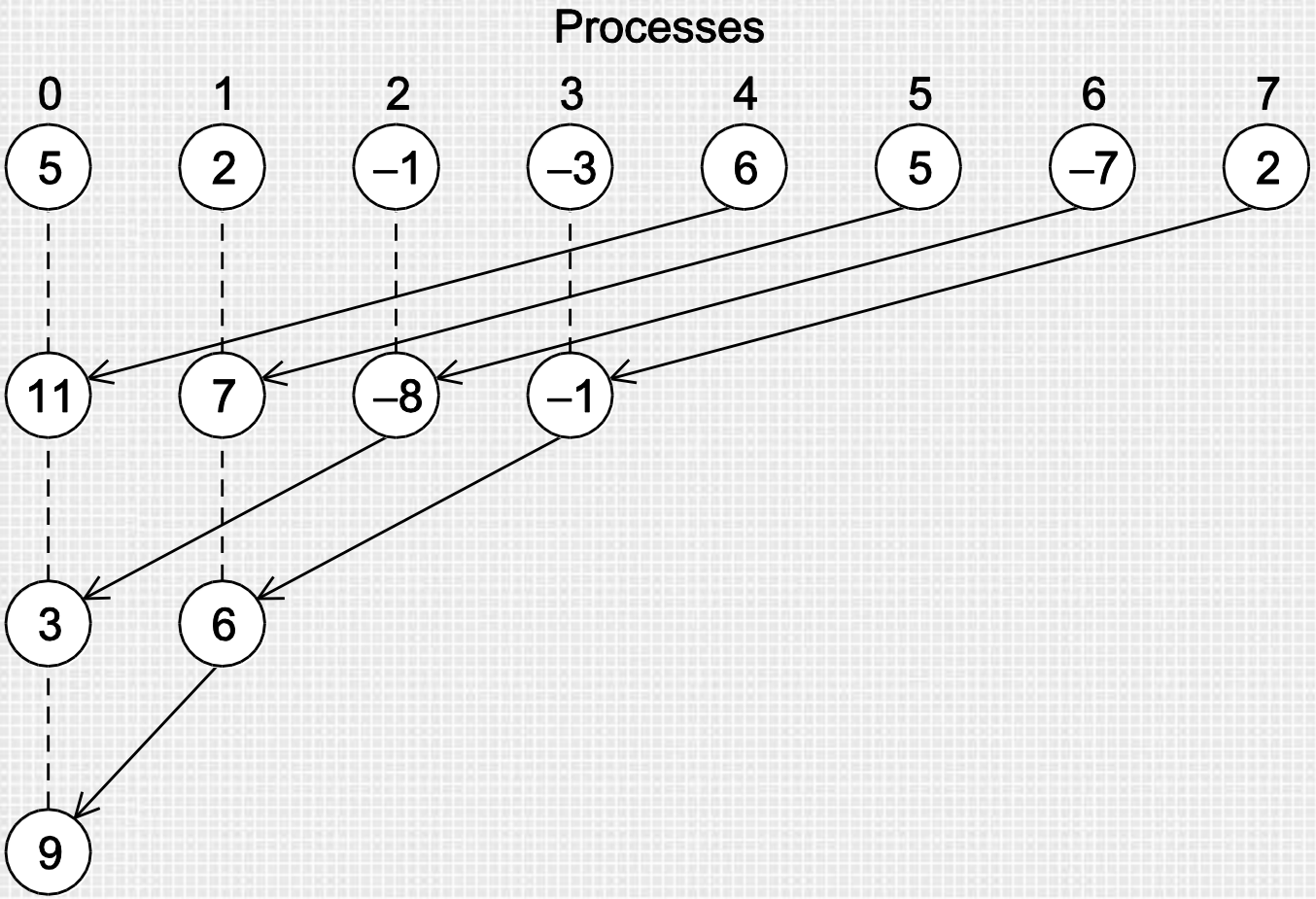
1. 第一阶段:
   (a) 1号，3号，5号，7号进程将他们的值分别发送给0号，2号，4号，6号进程。
   (b) 0, 2, 4和6号进程将接收到的值加到他们自己原有的值上。
   (c) 2号核6号进程将新值分别发送给0号和4号进程。
   (d) 0号和4号进程将接收到的值加到它们的新值上。

2. (a) 4号进程将最新的值发送给0号进程。
   (b) 0号进程将接收到的值相加。

树形结构全局求和



树形结构全局求和的另一种方法



3.4.2 MPI_Reduce

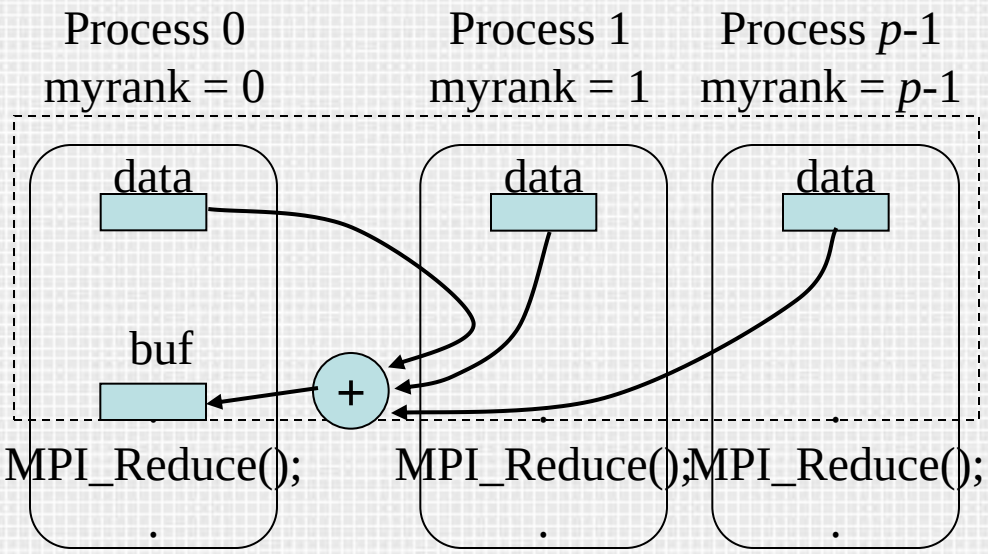
```
int MPI_Reduce（void * input_data_p,
                void * output_data_p,
                int count,
                MPI_Datatype datatype,
                MPI_Op operator,
                int dest_process,
                MPI_Comm comm）
```

这里每个进程的待处理数据存放在input_data_p中，所有进程将这些值通过输入的操作子operator计算为最终结果并将它存入dest_process进程的output_data_p。数据项的数据类型在Datatype域中定义。

Reduce -- 全局数据运算

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)



并行代码C语言

```
1  int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;            /* h is the same for all processes */
12    local_n = n/comm_sz;    /* So is the number of trapezoids  */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19       MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
```

```
21      } else {
22         total_int = local_int;
23         for (source = 1; source < comm_sz; source++) {
24            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            total_int += local_int;
27         }
28      }
29
30      if (my_rank == 0) {
31         printf("With n = %d trapezoids, our estimate\n", n);
32         printf("of the integral from %f to %f = %.15e\n",
33             a, b, total_int);
34      }
35      MPI_Finalize();
36      return 0;
37   } /*  main  */
```

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

MPI中预定义的规约操作符

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

3.4.3 集合通信与点对点通信的不同

- 在通信子中的所有进程都必须调用相同的集合通信函数。
  - 例如，试图将一个进程中的**MPI_Reduce**调用与另一个进程的**MPI_Recv**调用相匹配的程序会出错，此时程序会被悬挂或者崩溃。

集合通信与点对点通信的不同

- 每个进程传递给MPI集合通信函数的参数必须是“相容的”。
  - 例如，如果一个进程将0作为dest_process的值传递给函数，而另一个传递的是1，那么对MPI_Reduce调用所产生的结果就是错误的，程序可能被悬挂起来或者崩溃。

集合通信与点对点通信的不同

- 参数**output_data_p**只用在**dest_process**上。
  - 然而，所有进程仍需要传递一个与**output_data_p**相对应的实际参数，即使它的值只是**NULL**。

集合通信与点对点通信的不同

- 点对点通信函数是通过标签和通信子来匹配的。
- 集合通信函数不使用标签，只通过通信子和调用的顺序来进行匹配。

示例

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

对MPI_Reduce的多个调用

示例

- 假设每个进程调用 MPI_Reduce函数的运算符都是MPI_SUM ，那么目标进程为0号进程。
- 表中，在两次调用 MPI_Reduce后，b的值是3，而d的值是6。

示例

- 但是，内存单元的名字与 MPI_Reduce的调用匹配无关，函数调用的顺序决定了匹配方式。

- 所以b中所存储的值将是1+2+1 = 4， d中存储的值将是2+1+2 = 5。

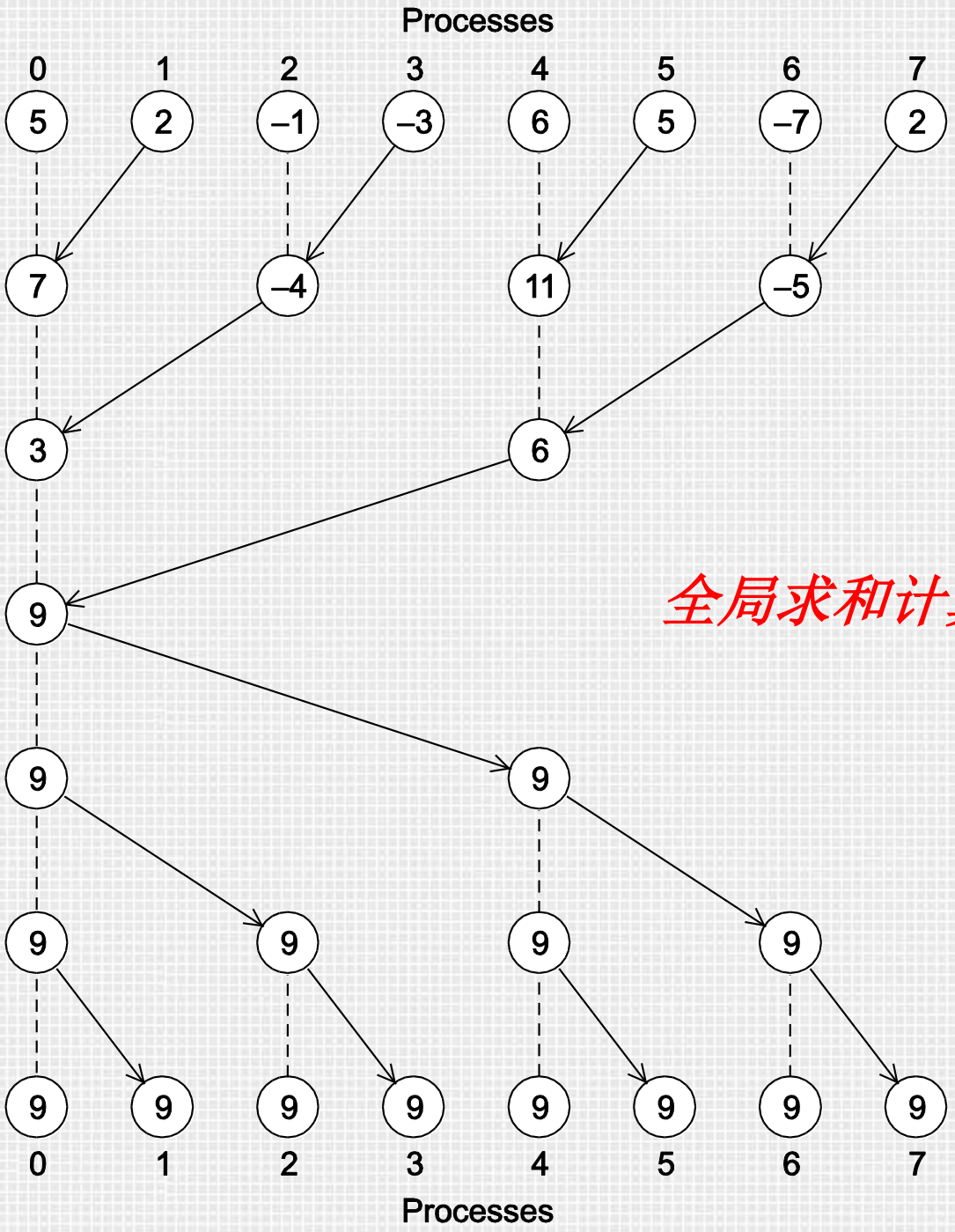
注意

- 调用**MPI_Reduce**，不能使用同一个缓冲区同时作为输入和输出。
- **MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm)**
- **MPI**这样调用方式是非法的：
- 主要原因是**MPI**禁止输入或输出参数作为其他参数的别名。

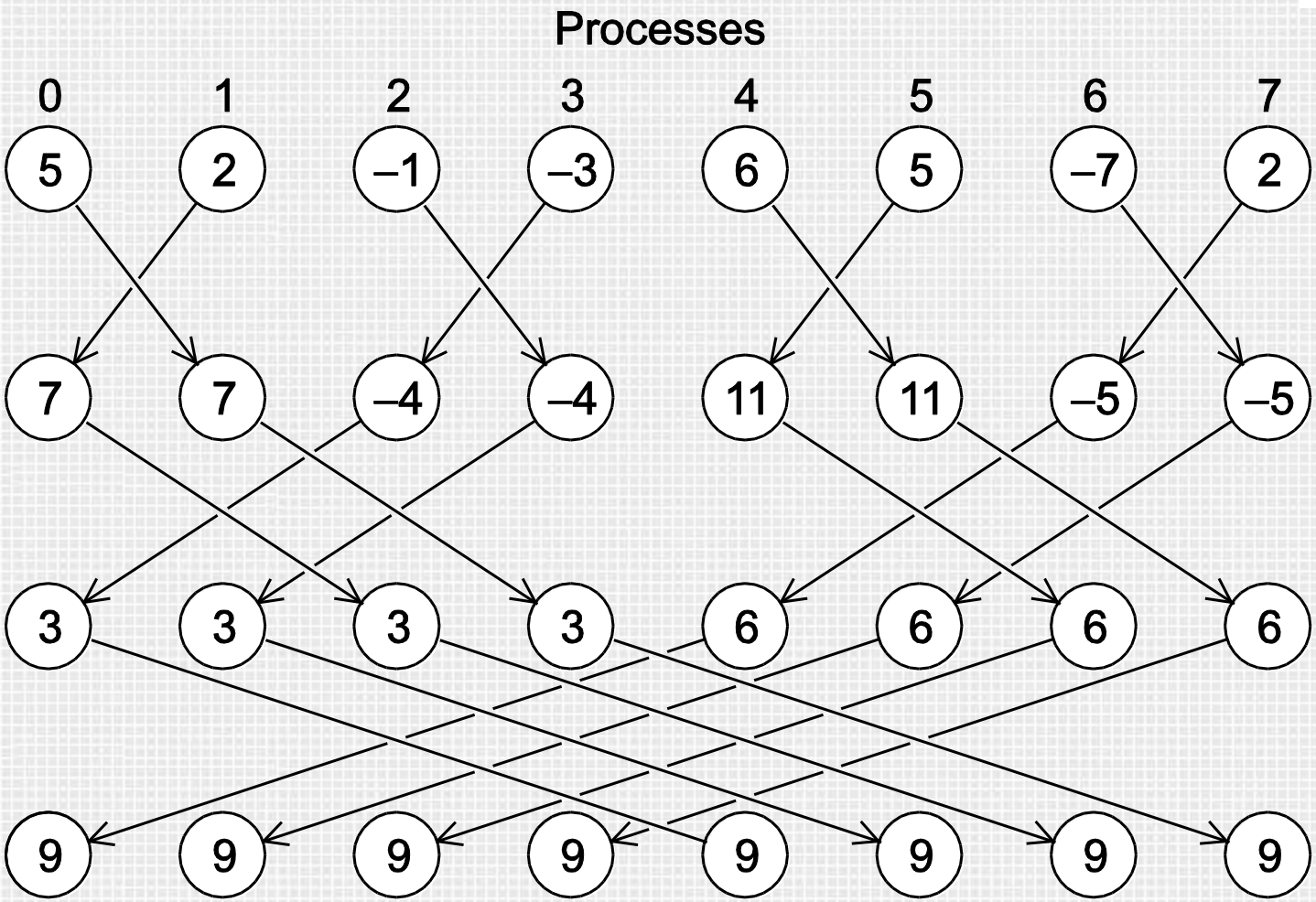
3.4.4 MPI_Allreduce

- 不难想象这样一种情况，即所有进程都想得到全局总和的结果，以便可以完成一个更大规模的计算。

```
int MPI_Allreduce(
        void*         input_data_p   /* in  */,
        void*         output_data_p  /* out */,
        int           count          /* in  */,
        MPI_Datatype  datatype       /* in  */,
        MPI_Op        operator       /* in  */,
        MPI_Comm      comm           /* in  */);
```



全局求和计算结果的发布



蝶形结构的全局总和计算

3.4.5 广播

Root

- 在一个集合通信中，如果属于一个进程的数据被发送到通信子中的所有进程，这样的集合通信就叫做广播。

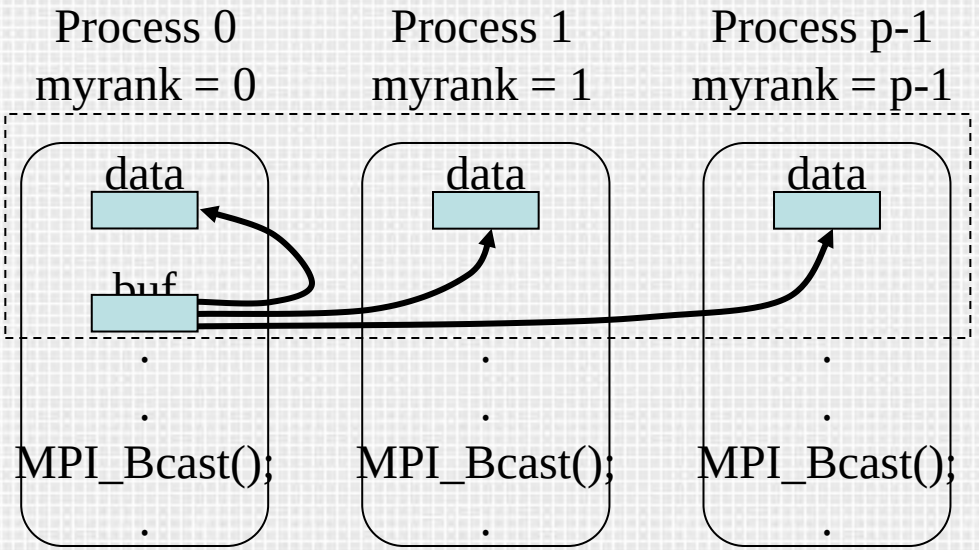
```
int MPI_Bcast(
        void*           data_p          /* in/out */,
        int             count           /* in     */,
        MPI_Datatype    datatype        /* in     */,
        int             source_proc     /* in     */,
        MPI_Comm        comm            /* in     */);
```

- 进程号为source_proc的进程将data_p所引用的内存内容发送给通信子comm中的所有进程。

Broadcast -- 数据广播

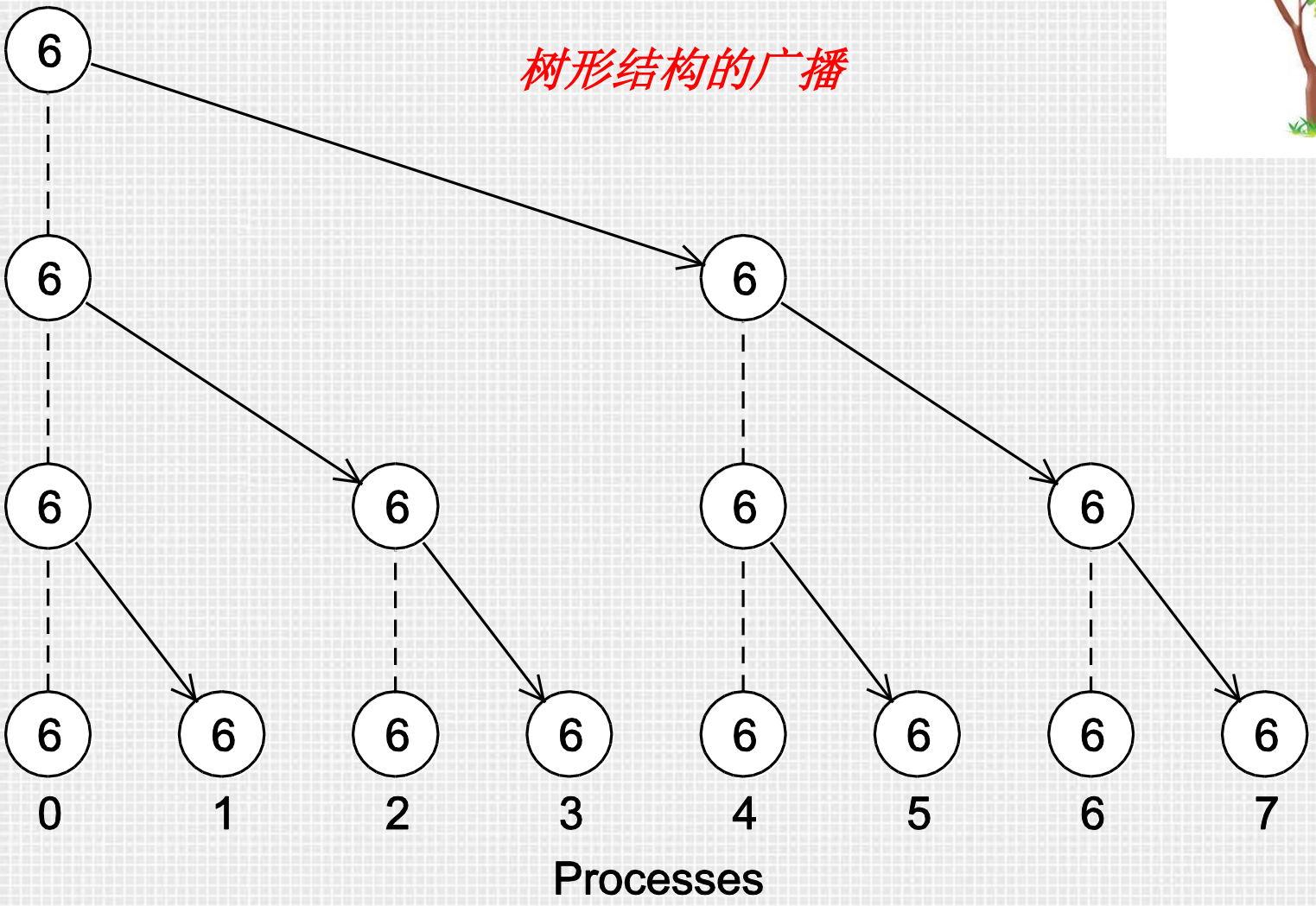
```
int MPI_Bcast (
        void *buffer,       /*发送/接收buf*/
        int     count,      /*元素个数*/
        MPI_Datatype datatype,
        int root,           /*指定根进程*/
        MPI_Comm comm)
```

根进程既是发送缓冲区也是接收缓冲区



- 广播的特点
  - 标号为**Root**的进程发送相同的消息给通信域**Comm**中的所有进程。
  - 消息的内容如同点对点通信一样由三元组**<Address, Count, Datatype>**标识。
  - 对**Root**进程来说，这个三元组既定义了发送缓冲也定义了接收缓冲。对其它进程来说，这个三元组只定义了接收缓冲

树形结构的广播



一个使用MPI_Bcast的Get_input函数版本

```
void Get_input(
      int        my_rank   /* in  */,
      int        comm_sz   /* in  */,
      double*    a_p       /* out */,
      double*    b_p       /* out */,
      int*       n_p       /* out */) {

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

3.4.6 数据分发

$$
\begin{aligned}
\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= \mathbf{z}
\end{aligned}
$$

计算向量和

向量求和的串行实现

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

在3个进程中，对有12个分量的向量的不同划分方式

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

划分方式

- 块划分**Block partitioning**
  - 简单的将连续**N**个分量所构成的块，分配到每个进程中。
- 循环划分**Cyclic partitioning**
  - 采用轮转的方式去分配向量分量。
- 块**-**循环划分**Block-cyclic partitioning**
  - 用一个循环来分发向量分量所构成的块，而不是分发单个向量分量。

向量求和的并行实现

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

3.4.7 散射

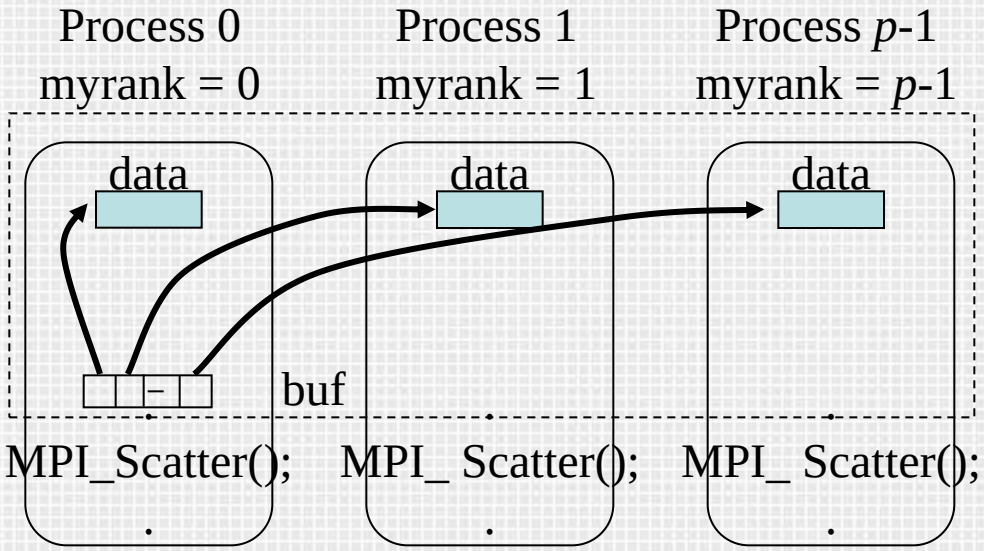
- **MPI_Scatter**
  - 0号进程读入整个向量，但只将分量发送给需要分量的其他进程。

```
int MPI_Scatter(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            src_proc     /* in  */,
      MPI_Comm       comm         /* in  */);
```

Scatter -- 数据分散

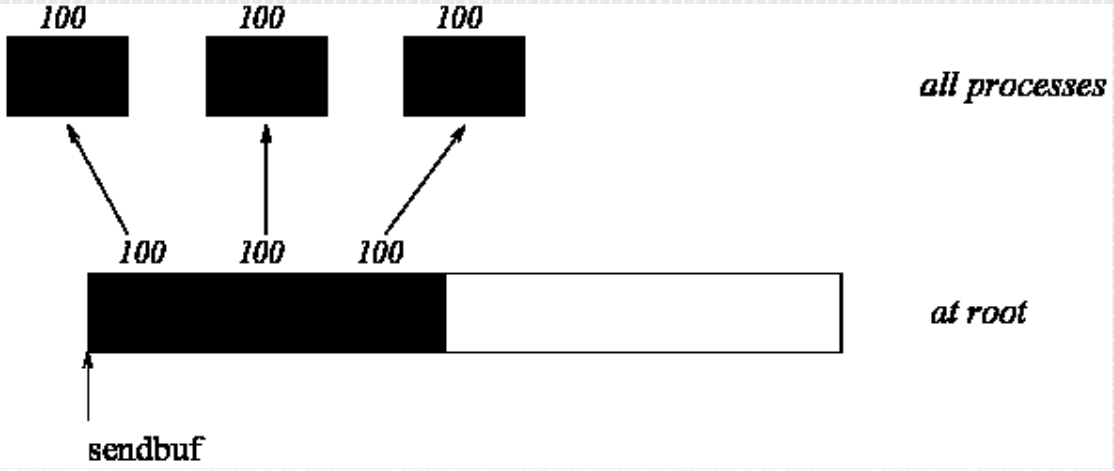
根进程中存储了*p*个消息,第*i*个消息将传给第*i*个进程.

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



MPI_ Scatter

- int MPI_Scatter (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_Scatter是一对多的传递消息。但是它和广播不同，root进程向各个进程传递的消息是可以不同的。
- 散播的特点
  - Scatter执行与Gather相反的操作。
  - Root进程给所有进程(包括它自己)发送一个不同的消息，这n (n为进程域comm包括的进程个数)个消息在Root进程的发送缓冲区中按进程标识的顺序有序地存放。
  - 每个接收缓冲由三元组<RecvAddress, RecvCount, RecvDatatype>标识，所有的非Root进程忽略发送缓冲。对Root进程，发送缓冲由三元组<SendAddress, SendCount, SendDatatype>标识。

一个读取并发向量的函数

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

3.4.8 聚集Gather

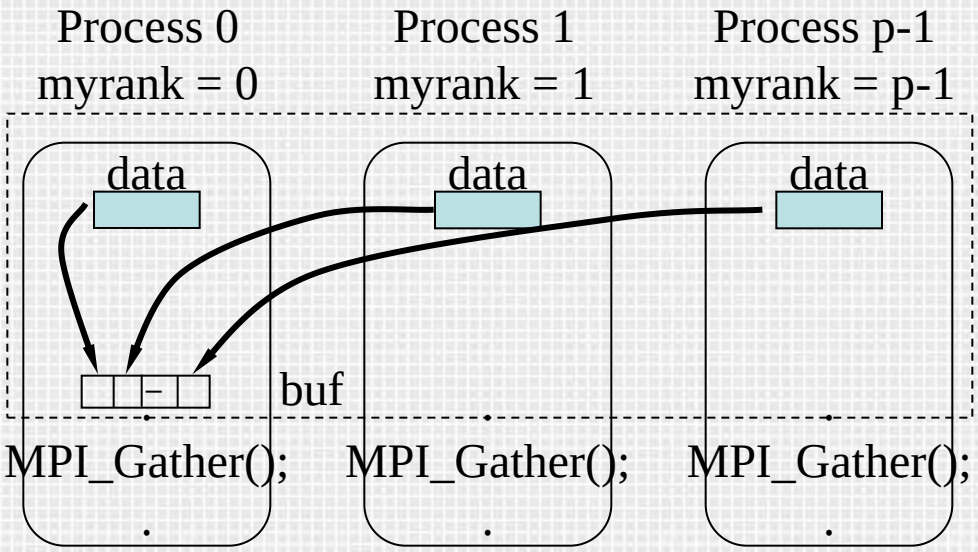
- 这个函数将向量的所有分量都收集到0号进程上，然后由0号进程将所有分量都打印出来。

```
int MPI_Gather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            dest_proc   /* in  */,
      MPI_Comm       comm        /* in  */);
```

Gather -- 数据收集

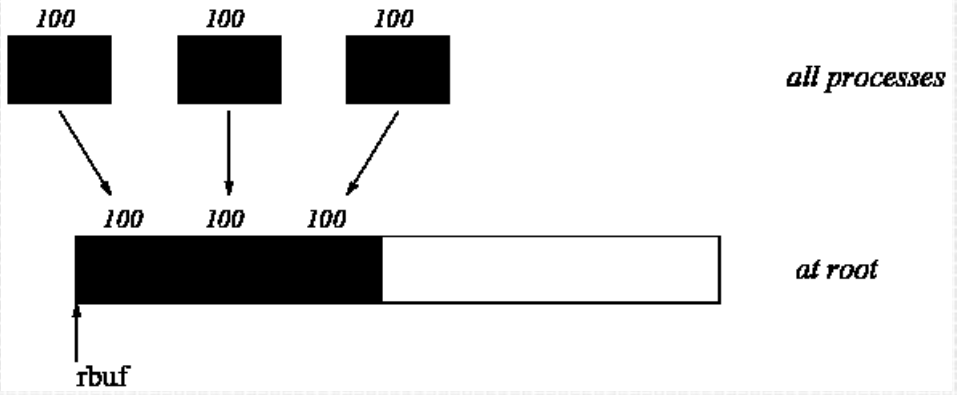
根进程接收其他进程来的消息(包括根进程),按每在进程在通信组中的编号依次联接在一下,存放在根进程的接收缓冲区中.

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



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



- 聚集的特点
  - 在收集操作中，**Root**进程从进程域**Comm**的所有进程(包括它自己)接收消息。
  - 这**n**个消息按照进程的标识**rank**排序进行拼接，然后存放在**Root**进程的接收缓冲中。
  - 接收缓冲由三元组**<RecvAddress, RecvCount, RecvDatatype>**标识，发送缓冲由三元组**<SendAddress, SendCount, SendDatatype>**标识，所有非**Root**进程忽略接收缓冲。

一个打印分布式向量的函数

```
void Print_vector(
      double     local_b[]   /* in */,
      int        local_n     /* in */,
      int        n           /* in */,
      char       title[]     /* in */,
      int        my_rank     /* in */,
      MPI_Comm   comm        /* in */) {

   double* b = NULL;
   int i;
```

```
    if (my_rank == 0) {
        b = malloc(n*sizeof(double));
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
              0, comm);
        printf("%s\n", title);
        for (i = 0; i < n; i++)
           printf("%f ", b[i]);
        printf("\n");
        free(b);
    } else {
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
              0, comm);
    }
}   /* Print_vector */
```

3.4.9 全局聚集Allgather

- 这个函数将每个进程的send_buf_p内容串联起来，存储到每个进程的recv_buf_p参数中。
- 大部分情况下，recv_count指的是每个进程接收的数据量。

```
int MPI_Allgather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

矩阵-向量乘法

$A = (a_{ij})$ is an $m \times n$ matrix

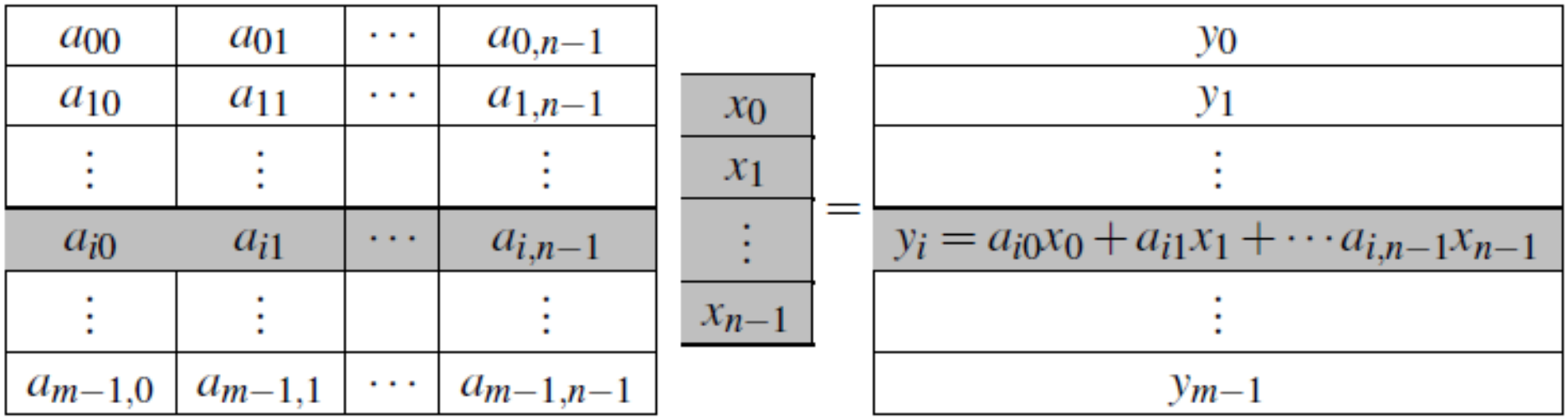
$\mathbf{x}$ is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i-th component of y

Dot product of the ith row of A with x.


a_{00} a_{01} $\cdots$ $a_{0,n-1}$
a_{10} a_{11} $\cdots$ $a_{1,n-1}$
a_{i0} a_{i1} $\cdots$ $a_{i,n-1}$
$a_{m-1,0}$ $a_{m-1,1}$ $\cdots$ $a_{m-1,n-1}$
x_0
x_1
x_{n-1}
=
y_0
y_1
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
y_{m-1}

串行矩阵-向量乘法伪代码

```
/* For each row of A */
for (i = 0; i < m; i++) {
    /* Form dot product of ith row with x */
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

C 语言的风格

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

矩阵-向量的串行乘法

```
void Mat_vect_mult(
        double  A[]   /* in  */,
        double  x[]   /* in  */,
        double  y[]   /* out */,
        int     m     /* in  */,
        int     n     /* in  */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
}  /* Mat_vect_mult */
```

MPI矩阵-向量乘法函数

```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;
```

```
   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

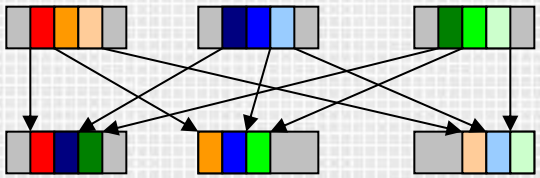
MPI集合通信函数

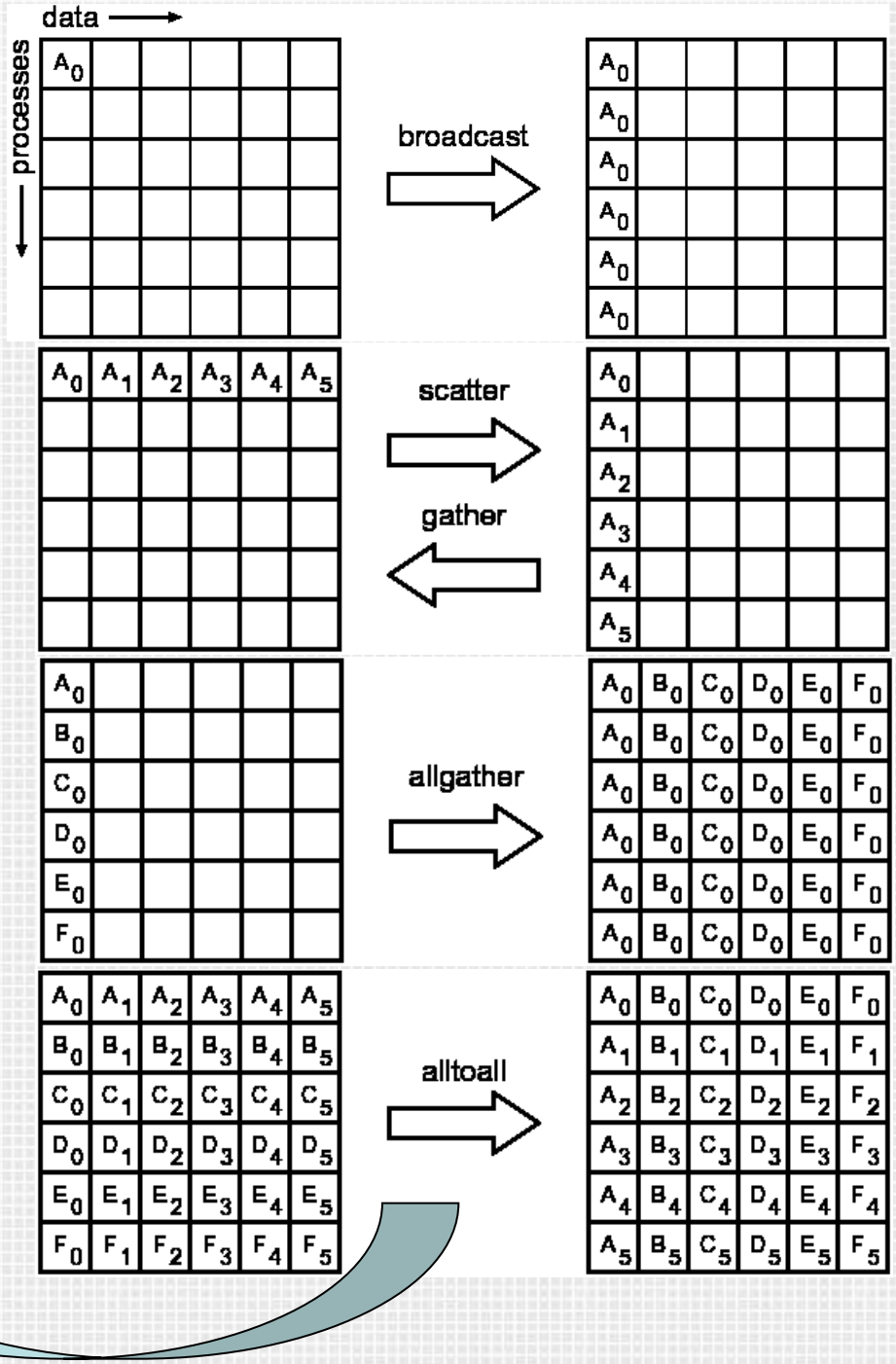
类型	函数	功能
数据移动	MPI_Bcast	一到多，数据广播
	MPI_Gather	多到一，数据汇合
	MPI_Gatherv	MPI_Gather的一般形式
	MPI_Allgather	MPI_Gather的一般形式
	MPI_Allgatherv	MPI_Allgather的一般形式
	MPI_Scatter	一到多，数据分散
	MPI_Scatterv	MPI_Scatter的一般形式
	MPI_Alltoall	多到多，置换数据(全互换)
	MPI_Alltoallv	MPI_Alltoall的一般形式
数据聚集	MPI_Reduce	多到一，数据归约
	MPI_Allreduce	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

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.

数据移动

Broadcast
Scatter
Gather
Allgather
Alltoall



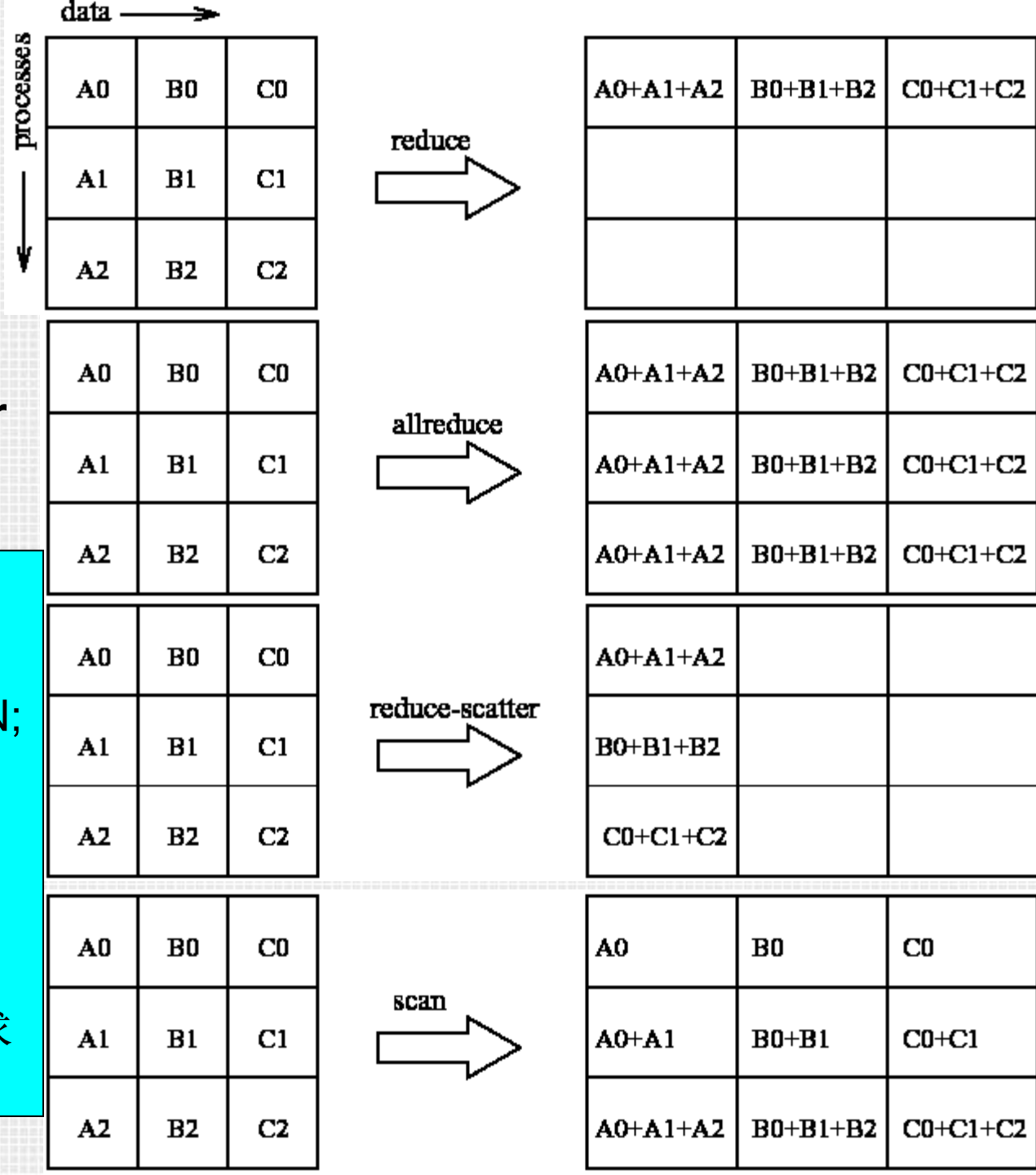


数据聚集

- **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… …



注意事项

- 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（若使用一对多或者多对一通信，则各个进程所使用的ROOT值必须都是相同的）

3.5 MPI的派生数据类型

派生数据类型

- 在MPI中，通过同时存储数据项的类型以及他们在内存中的相对位置，派生数据类型可以表示内存中数据项的任意集合。
  - 其主要思想：如果发送数据的函数知道数据项的类型以及在内存中数据项集合的相对位置，就可以在数据项被发送出去之前在内存中将数据项聚集起来。
  - 类似的，接收数据的函数可以在数据项被接收后将数据项分发到它们在内存中正确的目标地址。

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.

派生数据类型

- 一个派生数据类型是由一系列的**MPI**基本数据类型和每个数据类型的偏移所组成。

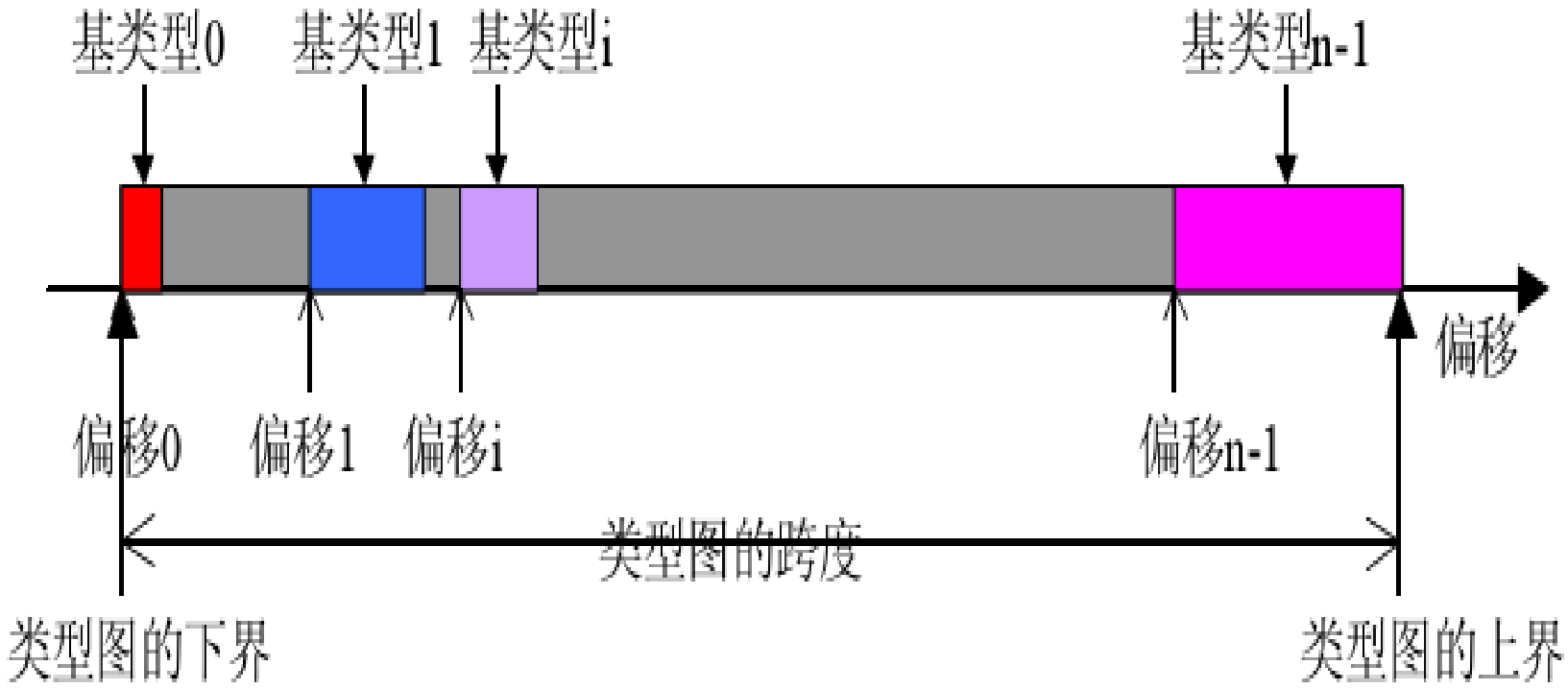
- 在梯形积分法的例子中:

Variable	Address
a	24
b	40
n	48

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}$$

数据类型图

类型图={<基类型0，偏移0>,<基类型1，偏移1>,...,<基类型n-1，偏移n-1>}



MPI_Type_create_struct

- 由这个函数创建由不同基本数据类型的元素所组成的派生数据类型：

```
int MPI_Type_create_struct(
      int              count                          /* in  */,
      int              array_of_blocklengths[]        /* in  */,
      MPI_Aint         array_of_displacements[]       /* in  */,
      MPI_Datatype     array_of_types[]               /* in  */,
      MPI_Datatype*    new_type_p                     /* out */);
```

MPI_Get_address

- 它返回的location_p是所指向的内存单元的地址。
- 这个特殊类型的MPI_Aint是整数型，它的长度足以表示系统地址。

```
int MPI_Get_address(
      void*       location_p   /* in  */,
      MPI_Aint*   address_p    /* out */);
```

MPI_Type_commit

- **MPI_Type_commit** 存储的是元素的**MPI**数据类型。它允许**MPI**实现为了在通信函数内使用这一数据类型，优化数据类型的内部表示。
  - 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

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

MPI_Type_free

- 当我们使用新的数据类型时，可以用 **MPI_Type_free** 函数去释放额外的存储空间。
  - 将数据类型设为**MPI_DATATYPE_NULL**

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

使用派生数据类型实现输入（1）

```
void Build_mpi_type(
      double*         a_p              /* in  */,
      double*         b_p              /* in  */,
      int*            n_p              /* in  */,
      MPI_Datatype*   input_mpi_t_p    /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};
```

使用派生数据类型实现输入（2）

```
    MPI_Get_address(a_p, &a_addr);
    MPI_Get_address(b_p, &b_addr);
    MPI_Get_address(n_p, &n_addr);
    array_of_displacements[1] = b_addr-a_addr;
    array_of_displacements[2] = n_addr-a_addr;
    MPI_Type_create_struct(3, array_of_blocklengths,
          array_of_displacements, array_of_types,
          input_mpi_t_p);
    MPI_Type_commit(input_mpi_t_p);
}   /* Build_mpi_type */
```

使用派生数据类型实现输入（3）

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

3.6 MPI程序的性能评估

并行代码运行时间

- **MPI_Wtime**，返回从过去某一时刻开始所经历的时间

```
double MPI_Wtime(void);
```

```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
       my_rank, finish-start);
```

串行代码运行时间

- 计算串行代码运行时间不需要连接**MPI**库。
- **GET_TIME**函数返回从过去某一时刻开始所经历的时间

```
#include "timer.h"
. . .
double now;
. . .
GET_TIME(now);
```

串行代码运行时间

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

MPI_Barrier

- **MPI_Barrier** 函数确保同一个通信子中的所有进程都完成调用改函数之前，没有进程能够提前返回。

```
int MPI_Barrier(MPI_Comm    comm    /* in */);
```

MPI_Barrier

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

矩阵-向量乘法的串行和并行程序的运行时间

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

加速比

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}$$

效率

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}$$

矩阵-向量乘法的加速比

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

矩阵-向量乘法的效率

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

可扩展性

- 如果问题的规模以一定的速率增大，但效率没有随着进程数的增加而降低，那么就可认为程序是可扩展的。

可扩展性

- 如果程序可以在不增加问题的规模的前提下维持恒定的效率，那么此程序称为强可扩展的。
- 当问题的规模增加，通过增大进程/线程个数来维持恒定的效率的程序称为弱可扩展的。

3.7 并行排序算法

排序

- n 个键值， p = comm sz个进程 .
- n/p 个键值分配给每个进程。
- 不对哪些键值分配到哪个进程上加以限制。
- 结果：
  - 当算法结束时：
    - 每个进程上的键值应该以升序的方式存储。
    - 如果 $0 \le q < r < p$, 则分配给进程q的每一个键值应该小于等于分配给进程r的每一个键值。

串行冒泡排序

```
void Bubble_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    int list_length, i, temp;

    for (list_length = n; list_length >= 2; list_length--)
        for (i = 0; i < list_length-1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }

}   /* Bubble_sort */
```

奇偶交换排序

- 这个算法由一系列阶段组成：
- 在偶数阶段，比较-交换由以下数对执行：

$$(a[0],a[1]),(a[2],a[3]),(a[4],a[5]),\ldots$$

- 在奇数阶段，比较-交换由以下数对执行：

$$(a[1],a[2]),(a[3],a[4]),(a[5],a[6]),\ldots.$$

示例

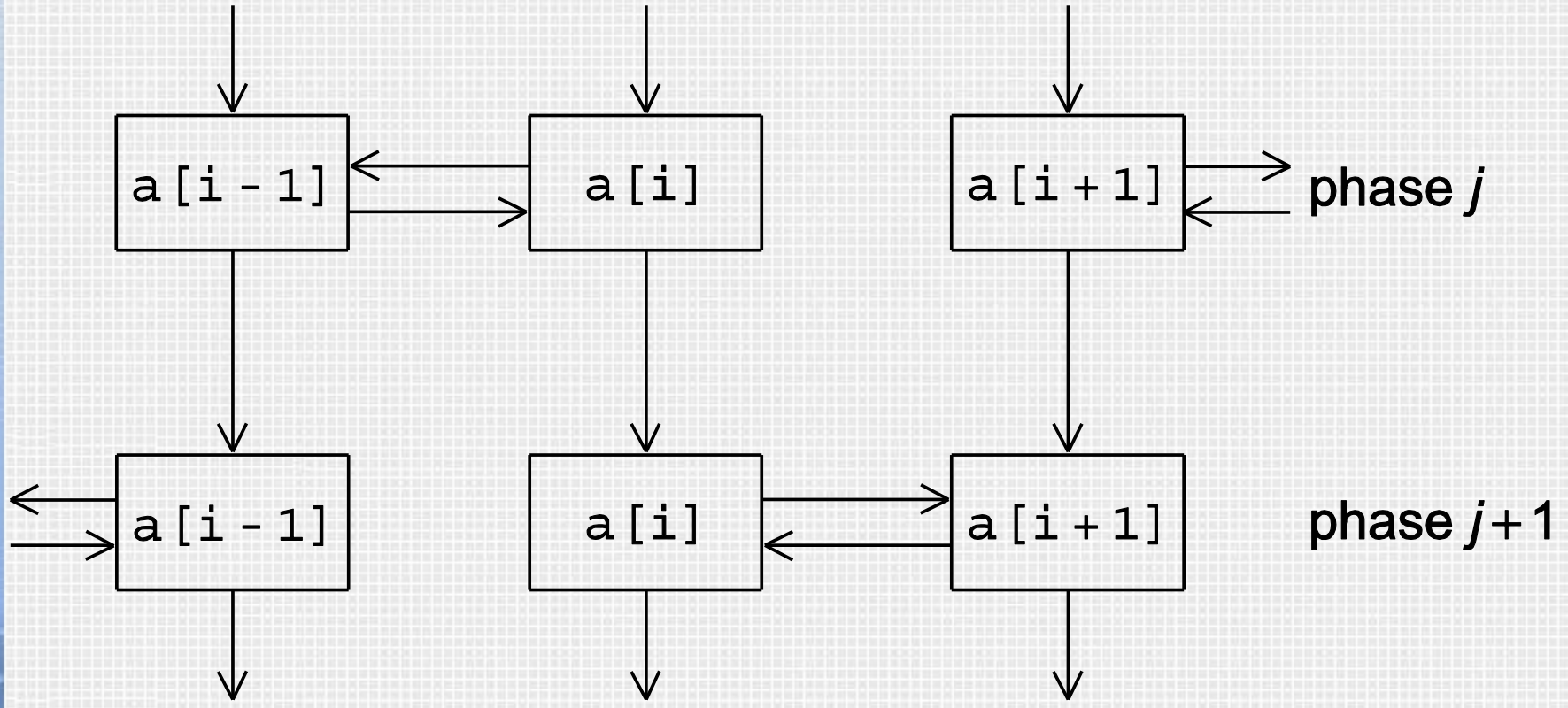
Start: 5, 9, 4, 3

1. **Even phase: compare-swap (5,9) and (4,3) getting the list 5, 9, 3, 4**
2. **Odd phase: compare-swap (9,3) getting the list 5, 3, 9, 4**
3. **Even phase: compare-swap (5,3) and (9,4) getting the list 3, 5, 4, 9**
4. **Odd phase: compare-swap (5,4) getting the list 3, 4, 5, 9**

串行奇偶交换排序

```
void Odd_even_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    int phase, i, temp;

    for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0) { /* Even phase */
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        } else { /* Odd phase */
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
        }
}  /* Odd_even_sort */
```

一次奇偶排序中任务间的通信



用a[i]来标记确定a[i]值的任务.

并行奇偶交换排序

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

伪代码

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

Compute_partner

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)       /* Odd rank */
      partner = my_rank - 1;
   else                        /* Even rank */
      partner = my_rank + 1;
else                         /* Odd phase */
   if (my_rank % 2 != 0)       /* Odd rank */
      partner = my_rank + 1;
   else                        /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

MPI程序的安全性

- MPI标准允许MPI_Send以两种不同的方式来实现：
  - 简单地将消息复制到MPI设置的缓冲区并返回；
  - 或者直到对应的MPI_Recv出现前都阻塞。

- 许多MPI函数都设置了使系统从缓冲到阻塞间切换的阀值。
  - 即相对较小的消息就交由MPI_Send缓冲。
  - 但对于大型数据就选择阻塞模式。

- 如果每个进程都阻塞在MPI_Send上，则没有进程回去调用MPI_Recv，此时程序就会发生死锁。
  - 每个进程都在等待一个不会发生的事件发生。
- 依赖于MPI提供的缓冲机制是不安全的。
  - 这样的程序在运行一些输入集时没有问题，但有可能在运行其他输入集时导致崩溃或者挂起。

引出几个问题！

- 怎么才能说一个程序时安全的？
- 怎样修改并行奇偶交换排序程序中的通信过程，使其安全？

MPI_Ssend

- MPI标准提供的一个函数来替代MPI_Send。
- 这个额外的字母“s”代表同步，保证了直到对应的接收开始前，发送端一直阻塞。

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

重构通信

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
      0, comm, MPI_STATUS_IGNORE.
```

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
} else {
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE.
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```

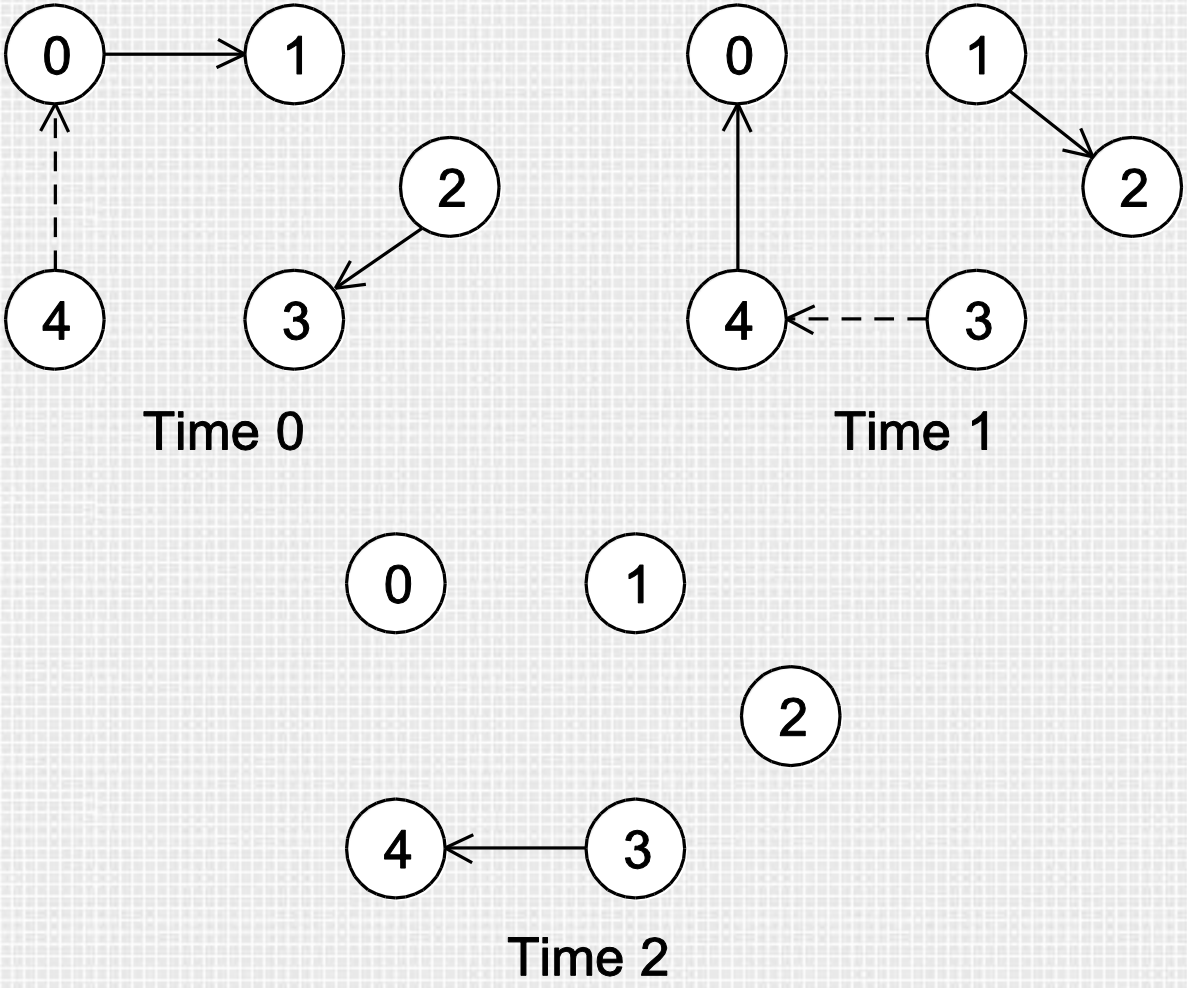
MPI_Sendrecv

- MPI提供的自己调度的方法。
- 他会分别执行一次阻塞式消息发送和一次消息接收。
- dest 和source 参数可以不同也可以相同。
- 优点在于，MPI库实现了通信调度，使程序不再挂起或崩溃。

MPI_Sendrecv

```
int MPI_Sendrecv(
        void*           send_buf_p      /* in  */,
        int             send_buf_size   /* in  */,
        MPI_Datatype    send_buf_type   /* in  */,
        int             dest            /* in  */,
        int             send_tag        /* in  */,
        void*           recv_buf_p      /* out */,
        int             recv_buf_size   /* in  */,
        MPI_Datatype    recv_buf_type   /* in  */,
        int             source          /* in  */,
        int             recv_tag        /* in  */,
        MPI_Comm        communicator    /* in  */,
        MPI_Status*     status_p        /* in  */);
```

5个进程之间的安全通信



并行奇偶排序算法中的Merge_low函数

```
void Merge_low(
        int  my_keys[],      /* in/out     */
        int  recv_keys[],    /* in         */
        int  temp_keys[],    /* scratch    */
        int  local_n         /* = n/p, in  */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

并行奇偶排序算法的运行时间

Processes	Number of Keys (in thousands)				
	200	400	800	1600	3200
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

(times are in milliseconds)

本章小结

- 消息传递接口（MPI）：是一个可以被C, C++和Fortran程序调用的函数库。
- 通信子：一组进程的集合，该集合中的进程之间可以相互发送消息。
- 单程序多数据流（SPMD）：许多并行程序通过根据不同的进程号转移到不同的分支语句。

- 大多数串行程序时确定性的：若用同一个程序运行相同的数据得到的结果是一样的。
- 大多数并行程序时非确定性的：同样的输入会得到不同的输出。
- 集合通信：不同于只涉及两个进程的MPI_Send和MPI_Recv，它涉及一个通信子中的所有进程。

- **MPI的6个基本函数调用:**

MPI_INIT:	启动MPI环境
MPI_COMM_SIZE:	确定进程数
MPI_COMM_RANK:	确定自己的进程标识符
MPI_SEND:	发送一条消息
MPI_RECV:	接收一条消息
MPI_FINALIZE:	结束MPI环境

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

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.

- 墙上时钟时间：即运行一段代码所需要的时间，它包括用户级代码，库函数，用户代码调用系统函数的运行时间以及空闲时间。
- 加速比：串行运行时间与并行运行时间之比。
- 效率：加速比除以进程总数。.

- 可扩展的：如果增加问题的规模(n)，随着p的增加，效率却没有递减，则称该并行程序时可扩展的。
- 不安全的：MPI_Send既可以阻塞也可以缓冲输入。