



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# 并行程序设计 with 算法

## MPI程序设计

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

中山大学 计算机学院  
国家超级计算广州中心

## ◦ 引言

- MPI Hello World
- MPI程序基本结构
- MPI通信基础

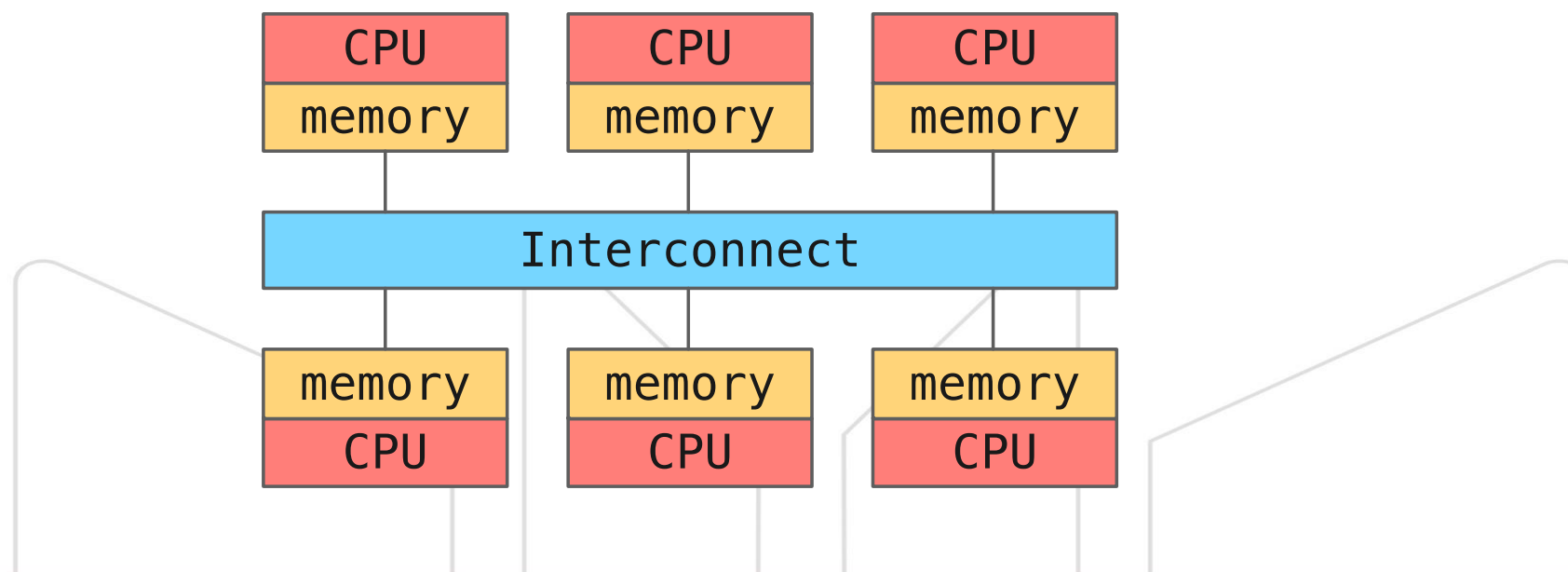
## ◦ MPI梯形积分法

## ◦ MPI集合通信

## ◦ MPI数据打包

## ◦ MPI并行排序

- MPI (Message Passing Interface) : 消息传递接口
  - 多进程并行编程API (应用程序编程接口)
    - 支持C、C++、Fortran语言
  - API使得进程间能通过发送消息进行数据通信
    - 不假设进程能够访问统一内存空间
    - 通常用于分布式内存环境 (如高性能计算)

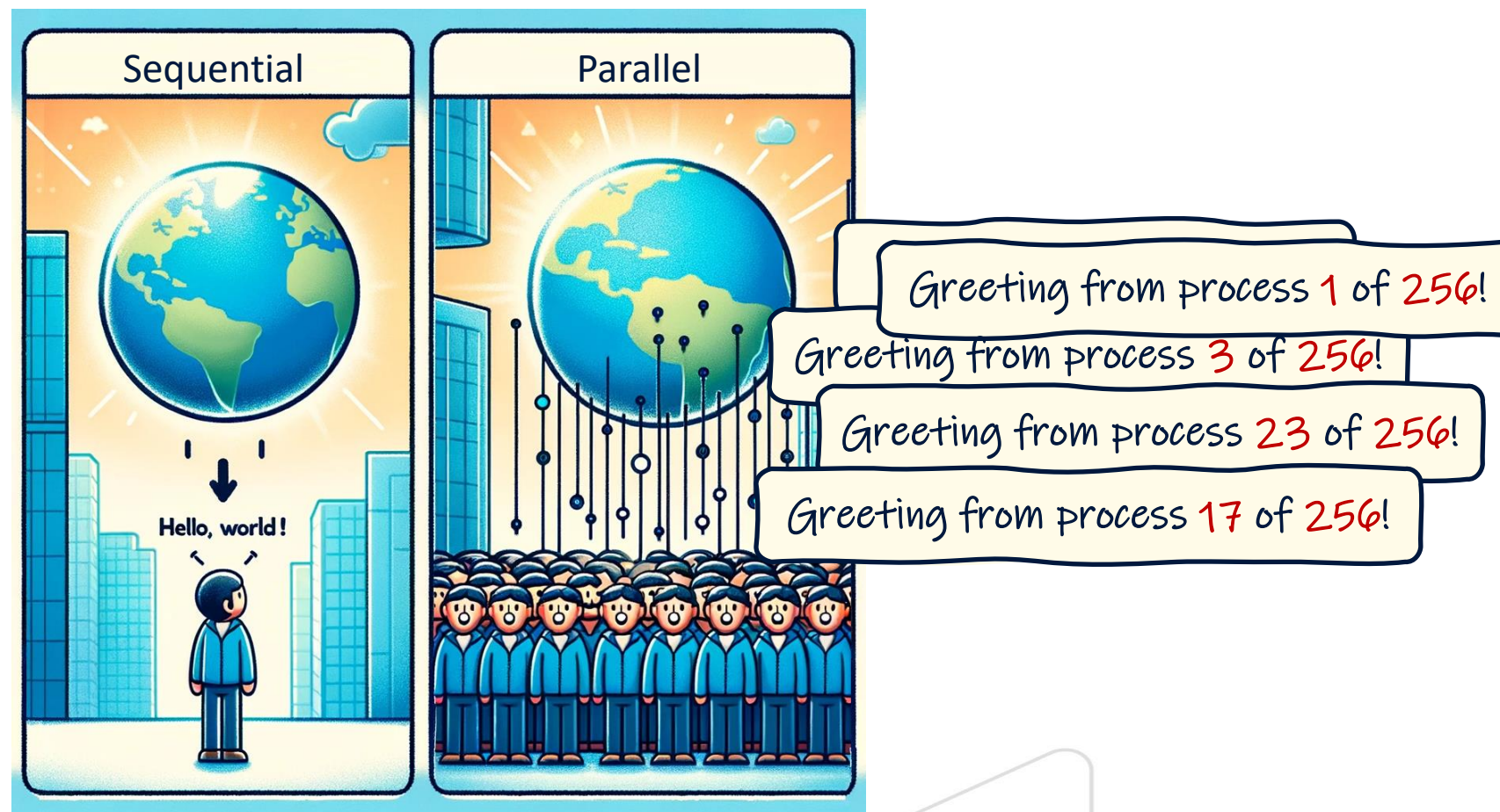


- 第一个程序: 并行Hello world!
  - 先从串行开始

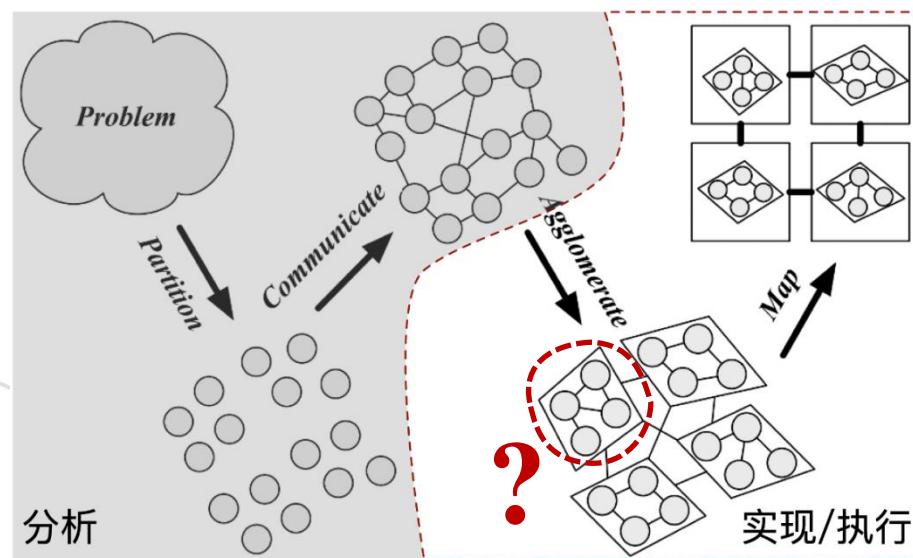
```
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");

    return 0;
}
```



- 并行程序执行过程中每个进程首先需要回答的两个问题
  - 一共有多少进程参与这个计算任务?
    - 答: 通过**`MPI_Comm_size()`**返回通信子中的进程数量
    - Comm (通信子) 为MPI中一组可以互相发送消息的进程集合
  - 我在这些进程中的编号 (负责处理哪部分任务) ?
    - 答: 通过**`MPI_Comm_rank()`**返回进程在通信子中的编号
    - $p$ 个进程返回编号分别为 $0, 1, 2, \dots, p - 1$





## • MPI Hello world实现

```
#include <mpi.h>      /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void){
    char greeting[MAX_STRING];
    int  comm_sz; /* Number of processes */
    int  my_rank; /* My process rank      */

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

    if(my_rank != 0){
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for(int q = 1; q < comm_sz; q++){
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }
    MPI_Finalize();
    return 0;
} /* main */
```

## ● MPI Hello world实现

```
#include <mpi.h>      /* For MPI functions, etc */  MPI头文件

const int MAX_STRING = 100;

int main(void){
    char greeting[MAX_STRING];
    int  comm_sz; /* Number of processes */
    int  my_rank; /* My process rank      */

    MPI_Init(NULL, NULL);          MPI初始化
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  获取进程数量
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  获取进程编号

    if(my_rank != 0){
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        发送 MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for(int q = 1; q < comm_sz; q++){
            接收 MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

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

## • MPI程序编译

– **mpicc** -g -Wall -o mpi\_hello mpi\_hello.c

- **mpicc**: C语言编译器的包装脚本 (wrapper)
  - 在C语言编译器的基础上增加了MPI相关参数
  - 课程假定包装的是GNU C编译器gcc
- **-g**: gcc编译器选项, 表明需要产生调试信息
- **-Wall**: gcc编译选项, 表明编译器将输出所有警告 (warning)
- **-o mpi\_hello**: 指明编译器输出文件名为
- **mpi\_hello.c**: 源文件
- 更多编译选项及参数可参考gcc编译
  - <https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>
  - 输入, 输出, 链接文件/目录, 调试, 警告, 优化, flag, 等



## • MPI程序运行

– **mpirexec** -n <number of process> <executable>

- -n <number of process>: 指明进程数量
- <executable>: 指明进程执行的程序

Greetings from process 0 of 1!

• **mpirexec** -n 1 ./mpi\_hello: 由1个进程执行mpi\_hello

• **mpirexec** -n 4 ./mpi\_hello: 由4个进程执行mpi\_hello

- ./: 确保系统在执行时使用当前目录下的可执行文件
- Windows执行时搜索当前路径及PATH环境变量
- Linux执行时只搜索PATH环境变量, 不搜索当前路径

Greetings from process 0 of 4!  
Greetings from process 1 of 4!  
Greetings from process 2 of 4!  
Greetings from process 3 of 4!

## ● MPI程序基本结构

### – 使用C语言编写

- 执行**main**(**void**)函数
- 可以使用C语言的标准头文件

- #include <stdio.h>
  - #include <string.h>

### – 需要包含**mpi.h**头文件

### – MPI标识符都以**MPI\_**开头

- 库函数及MPI定义的类型在**MPI\_**后的第一个字母都是大写
- **MPI\_**宏和常量所有字母都是大写
- 可以以此区分MPI定义/用户定义

```
...  
#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;  
}
```

## • MPI程序基本结构

### – MPI初始化

- `int MPI_Init(int *argc_p, char*** argv_p);`
- 通知MPI系统进行必要初始化设置
  - 分配消息缓冲区存储、指定进程号，等
- 应为第一个调用的MPI函数
- 返回错误值（通常可以忽略）

### – MPI结束

- `int MPI_Finalize(void);`
- 通知MPI系统执行结束，可以释放资源
- 应为最后一个调用的MPI函数

```
...  
#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;  
}
```

## • MPI程序基本结构

### – 通信子 (communicator)

- 一组可以互相发送消息的进程集合
- **MPI\_Init**定义由用户启动的所有进程所组成的通信子
  - 最为常见的通信子为全体通信子（所有进程）：MPI\_COMM\_WORLD
- 与通信子相关的函数及变量都以**MPI\_COMM**开头
  - 获取通信子的进程数，如**MPI\_Comm\_size**(MPI\_COMM\_WORLD ,&comm\_sz);

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

- 获取进程在通信子中的编号，如**MPI\_Comm\_rank**(MPI\_COMM\_WORLD, &my\_rank);

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

## • SPMD单程序多数据流

- 一个程序处理多个任务，而非每个任务编译一个程序
  - 大多数MPI程序采用的策略
- 此前MPI Hello World例子
  - 通过if-else语句实现多任务
  - 0号进程收集信息并打印，其他进程产生消息并发送给0号进程

```
if(my_rank != 0){  
    sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);  
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);  
}  
else {  
    printf("Greetings from process %d of %d!\n", my_rank, comm_sz);  
    for(int q = 1; q < comm_sz; q++){  
        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        printf("%s\n", greeting);  
    }  
}
```

## ● MPI\_Send 发送消息

- 点对点通信
- 指明发送消息：位置，大小，类型
- 指明接收方信息：编号，通信子
  - 消息标志用于区分相同发送方与接收方之间的多条消息

```
int MPI_Send(  
    void*          msg_buf_p, ----- 发送缓冲区地址  
    int            msg_size, ----- 发送数据大小  
    MPI_Datatype   msg_type, ----- 发送数据类型  
    int            dest, ----- 目的地进程编号  
    int            tag, ----- 消息标志  
    MPI_COMM       communicator); -- MPI进程所在的通信子
```

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	



## ◉ MPI\_Recv接收消息

### – 前6个参数与MPI\_Send一一对应

- 指明用于接收消息的内存位置，大小，类型
  - 注意内存将被写入
- 指明发送方的编号，通信子，及消息标志

int MPI_Recv( void* int MPI_Datatype int int MPI_COMM MPI_Status*	buf_buf_p, buf_size, buf_type, source, tag, communicator, status_p);	int MPI_Send( void* int MPI_Datatype int int MPI_COMM	msg_buf_p, msg_size, msg_type, dest, tag, communicator);
	-----		
	-----		
	-----		
	-----		
	-----		
	-----		

## ◉ MPI\_Recv接收消息

— 前6个参数与MPI\_Send一一对应

• 指明用于接收消息的内存位置，大小，类型

MPI Hello World程序中的消息收发

— 注意内存将被写入

• 指明发送方的编号，通信子，及消息标志

发送给0号进程

其他进程: `MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);`

0号进程: `MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`

缓冲区大小而非消息大小

使用循环依次从q号线程接收

发送成功条件

- 消息匹配：数据类型相同，接收缓冲区大于消息大小
- 收发方匹配：通信子相同，消息标签相同，发送方目标为接收方，接收方来源为发送方

潜在问题：0号进程使用循环接收消息：`for(int q = 1; q < comm_sz; q++) MPI_Recv(... q ...);`

`MPI_Status status_p),`

## ◉ MPI\_Recv接收消息

- 接收方可以不知道消息大小，数据来源，消息标签
  - 使用MPI\_ANY\_SOURCE作为source从任意发送方接收消息
  - 使用MPI\_ANY\_TAG作为tag接收发送方的任意消息
- 最后1个参数用于返回接收消息的状态（来源）
  - 包含三个成员MPI\_SOURCE, MPI\_TAG, MPI\_ERROR
  - 由用户分配空间并将其指针该参数传递给系统
  - 当消息来源及标签确定时，可不返回

– MPI\_STATUS\_IGNORE

- 获取接收消息大小

```
int MPI_Get_count(  
    MPI_Status* status_p,  
    MPI_Datatype type,  
    int* count); -- 返回大小
```

```
int MPI_Recv(  
    void* buf, int buf_size,  
    MPI_Datatype buf_type,  
    int source, int tag,  
    MPI_COMM communicator,  
    MPI_Status* status_p); 17
```

## ◉ 通信与阻塞

- 阻塞指程序执行过程中某个操作等待条件满足而暂停执行
  - 常见于资源等待、同步操作中
- 在MPI通信的语境下，指**函数调用完成后是否能马上返回**
  - 非阻塞：调用完成后立刻返回执行下一语句，无论通信是否实际完成
  - 阻塞：需暂停执行，等通信完成后才能继续执行
- **MPI\_Send**消息发送的精确行为往往**依赖于MPI实现**
  - 通常由“截止（cutoff）”大小决定：超过截止大小则阻塞
  - 一些特定实现中，甚至需要等待接收方开始接收后才返回
  - 了解你使用的实现，**不要预先假设**！
- **MPI\_Recv**消息接收都是阻塞的
- 小心**进程悬挂**：标签/收发方不匹配等导致通信无法完成而永远阻塞

## 通信中的顺序

- MPI规范要求消息**不可超越**（non-overtaking）
- 同一对发送方与接收方之间的消息接收顺序与发送顺序一致
  - 进程A向进程B依次发送消息1与消息2，则消息1必须在消息2前可用
  - 不出现“后发先至”
- 不同的发送方和接收方之间消息的到达顺序没有限制
  - 进程A先向C发送消息1，进程B再向C发送消息2，消息2可能在1前到达
  - 进程A依次向B发送消息1，向C发送消息2，消息2可能在1前到达



- 引言
- MPI梯形积分法
- MPI集合通信
- MPI数据打包
- MPI并行排序



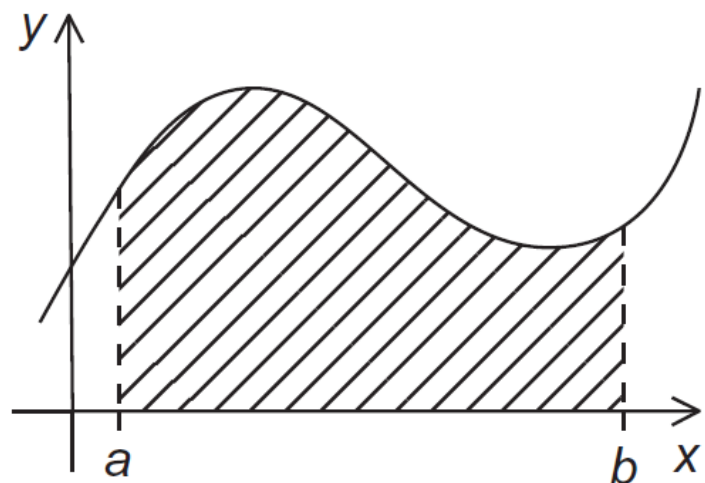
## ● 梯形积分法估算 $y = f(x)$ 在 $[a, b]$ 区间上的积分

– 将 $[a, b]$ 分为 $n$ 个等长子区间

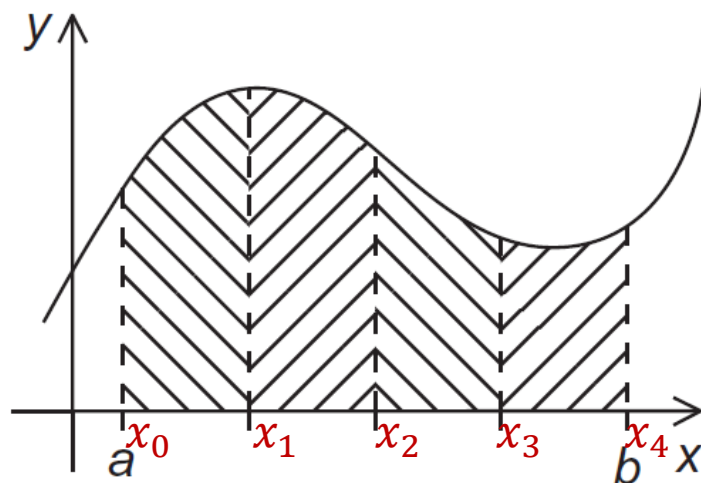
- 每段长度 $h = (b - a)/n$

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

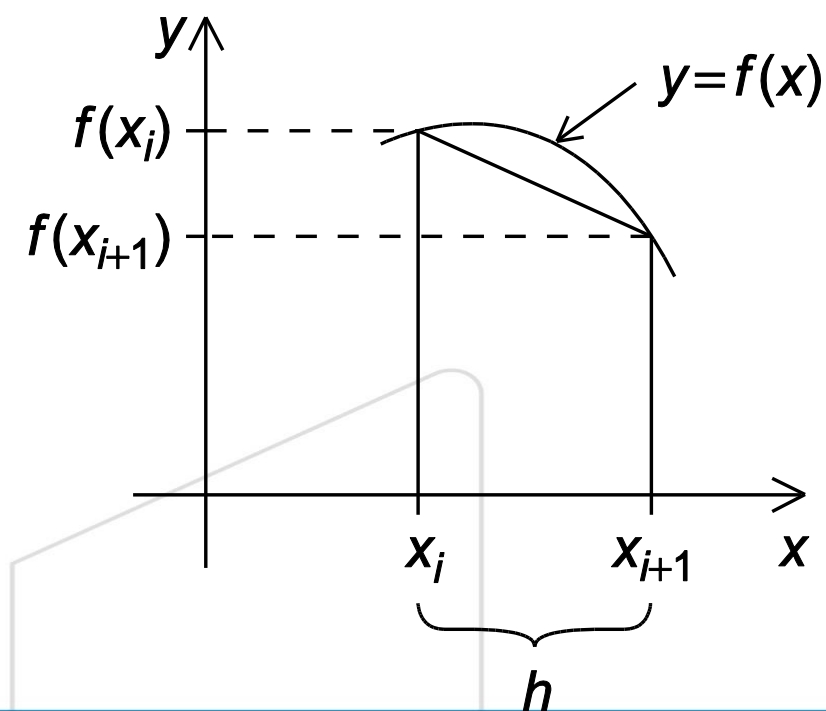
– 其中， $[x_i, x_{i+1}]$ 段的面积为 $\frac{h}{2} (f(x_i) + f(x_{i+1}))$



(a)

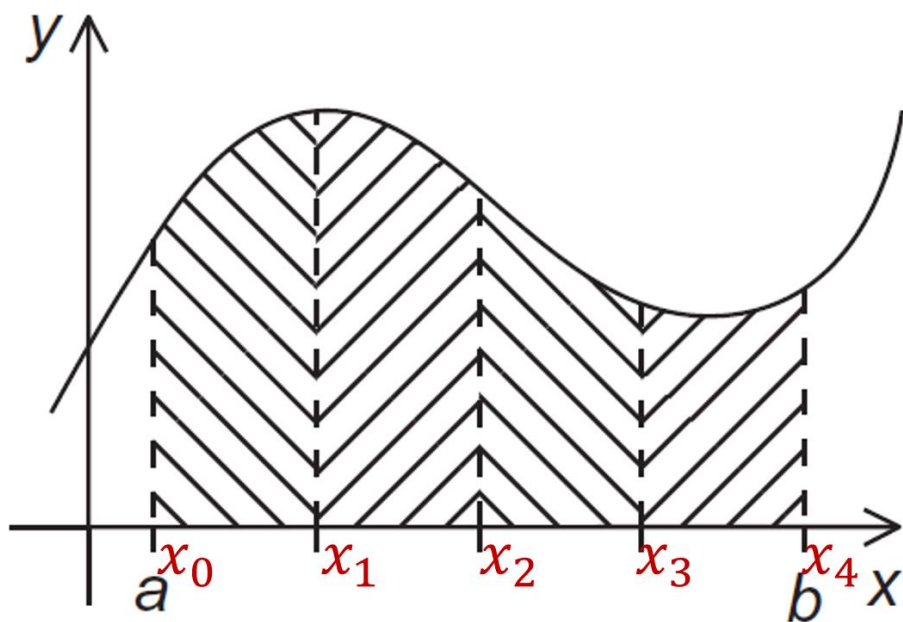


(b)



- 梯形积分法估算 $y = f(x)$ 在 $[a, b]$ 区间上的积分

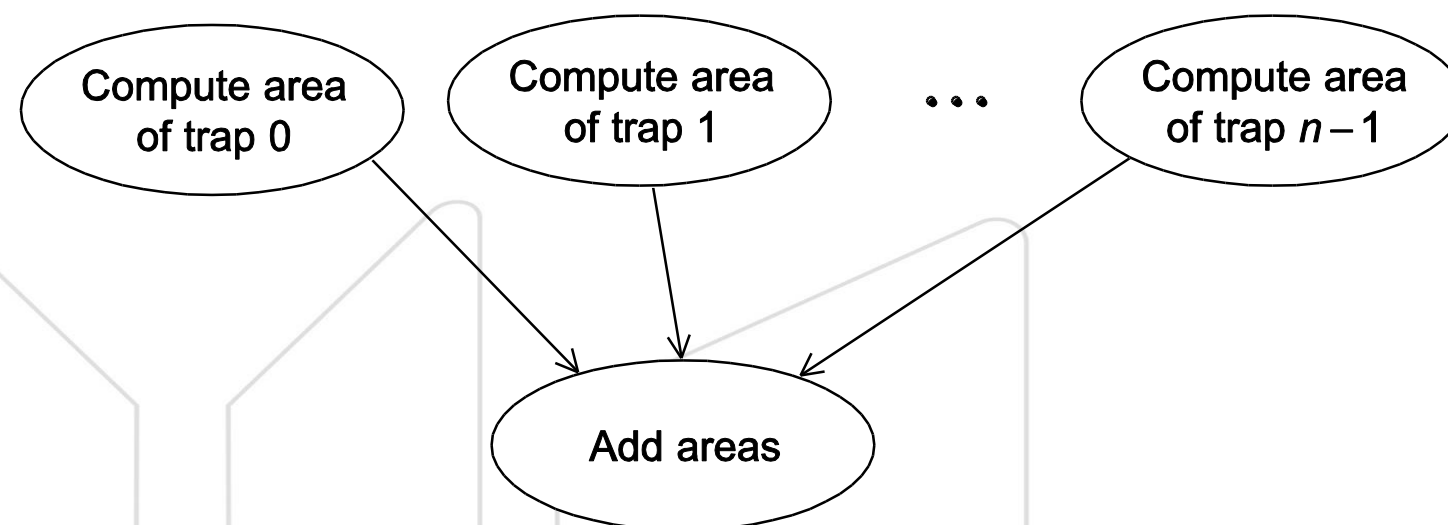
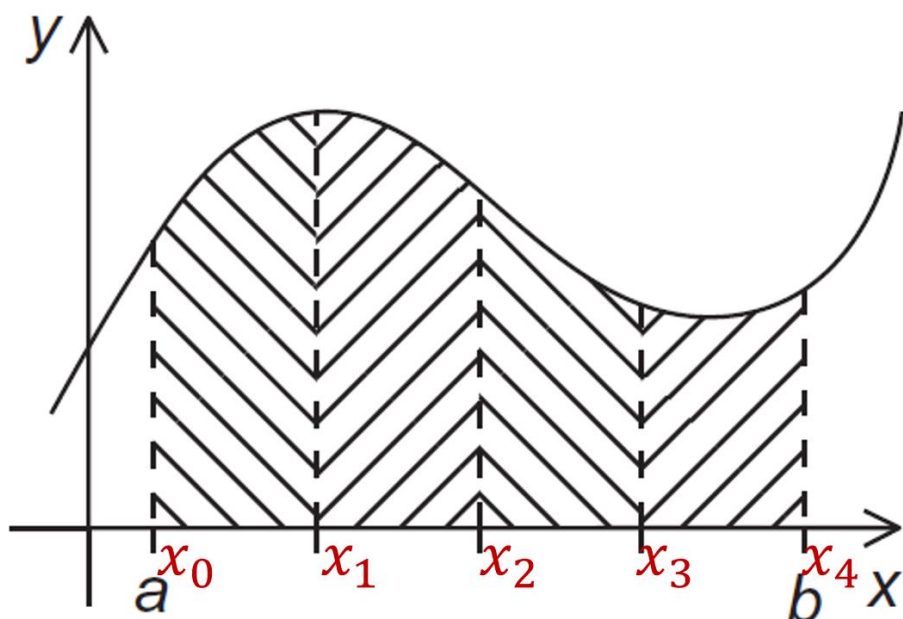
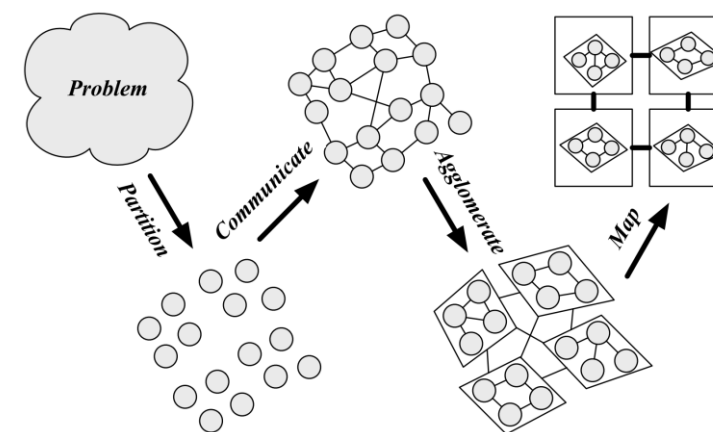
– 总面积为 $h(\frac{1}{2}f(x_0) + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n))$



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

## 梯形积分法的并行设计

- 划分：最细粒度子任务为单个梯形面积计算  $area([x_i, x_{i+1}])$
- 通信：（发送/交换）梯形面积求和
- 聚集：一个区间内的梯形面积和  $area([x_i, x_{i+k}])$
- 映射：每个进程计算一个区间



- 并行伪码

```
Get a, b, n;

h = (b-a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;

local_integral = Trap(local_a, local_b, local_n, h);

if(my_rank != 0)
    Send local_integral to process 0;
else { /* my_rank == 0 */
    total_integral = local_integral;
    for( proc = 1; proc < comm_sz; proc++){
        Receive local_integral from proc;
        total_integral += local_integral;
    }
}

if(my_rank == 0) print result;
```

## • MPI实现-第一版

变量定义

MPI初始化

局部计算

通信

输出结果

MPI结束

```
int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;

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

    h = (b-a)/n;          /* h is the same for all processes */
    local_n = n/comm_sz; /* So is the number of trapezoids */

    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    local_int = Trap(local_a, local_b, local_n, h);

    if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD);
    } else {
        total_int = local_int;
        for (source = 1; source < comm_sz; source++) {
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_int += local_int;
        }
    }

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n",
               a, b, total_int);
    }

    MPI_Finalize();

    return 0;
} /* main */
```

## ● MPI实现-第一版

### — 变量定义

```
int my_rank, comm_sz, n = 1024, local_n;  
double a = 0.0, b = 3.0, h, local_a, local_b;  
double local_int, total_int;  
int source;
```

### — MPI基本调用

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

```
int main(void) {  
    int my_rank, comm_sz, n = 1024, local_n;  
    double a = 0.0, b = 3.0, h, local_a, local_b;  
    double local_int, total_int;  
    int source;  
  
    MPI_Init(NULL, NULL);  
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
    h = (b-a)/n;          /* h is the same for all processes */  
    local_n = n/comm_sz; /* So is the number of trapezoids */  
  
    local_a = a + my_rank*local_n*h;  
    local_b = local_a + local_n*h;  
    local_int = Trap(local_a, local_b, local_n, h);  
  
    if (my_rank != 0) {  
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,  
                 MPI_COMM_WORLD);  
    } else {  
        total_int = local_int;  
        for (source = 1; source < comm_sz; source++) {  
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,  
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
            total_int += local_int;  
        }  
    }  
  
    /* Print the result */  
    if (my_rank == 0) {  
        printf("With n = %d trapezoids, our estimate\n", n);  
        printf("of the integral from %f to %f = %.15e\n",  
               a, b, total_int);  
    }  
  
    MPI_Finalize();  
  
    return 0;  
} /* main */
```



## • MPI实现-第一版

### – 局部计算

```
h = (b-a)/n;  
local_n = n/comm_sz;  
  
local_a = a + my_rank*local_n*h;  
local_b = local_a + local_n*h;  
local_int = Trap(local_a, local_b, local_n, h);
```

### – 进程0输出结果

```
if (my_rank == 0) {  
    printf("With n = %d trapezoids, our estimate\n", n);  
    printf("of the integral from %f to %f = %.15e\n",  
        a, b, total_int);  
}
```

```
int main(void) {  
    int my_rank, comm_sz, n = 1024, local_n;  
    double a = 0.0, b = 3.0, h, local_a, local_b;  
    double local_int, total_int;  
    int source;  
  
    MPI_Init(NULL, NULL);  
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
    h = (b-a)/n;          /* h is the same for all processes */  
    local_n = n/comm_sz; /* So is the number of trapezoids */  
  
    local_a = a + my_rank*local_n*h;  
    local_b = local_a + local_n*h;  
    local_int = Trap(local_a, local_b, local_n, h);  
  
    if (my_rank != 0) {  
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,  
            MPI_COMM_WORLD);  
    } else {  
        total_int = local_int;  
        for (source = 1; source < comm_sz; source++) {  
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,  
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
            total_int += local_int;  
        }  
    }  
  
    /* Print the result */  
    if (my_rank == 0) {  
        printf("With n = %d trapezoids, our estimate\n", n);  
        printf("of the integral from %f to %f = %.15e\n",  
            a, b, total_int);  
    }  
    MPI_Finalize();  
  
    return 0;  
} /* main */
```

## ● MPI实现-第一版

### – 局部计算 – 区间梯形面积计算

```
double Trap(
    double left_endpt /* in */,
    double right_endpt /* in */,
    int trap_count /* in */,
    double base_len /* in */)
{
    double estimate, x;
    int i;

    estimate = (f(left_endpt) + f(right_endpt))/2.0;
    for (i = 1; i <= trap_count-1; i++) {
        x = left_endpt + i*base_len;
        estimate += f(x);
    }
    estimate = estimate*base_len;

    return estimate;
} /* Trap */
```

是否可使用  
x += base\_len ?

```
int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;

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

    h = (b-a)/n; /* h is the same for all processes */
    local_n = n/comm_sz; /* So is the number of trapezoids */

    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    local_int = Trap(local_a, local_b, local_n, h);

    if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
                MPI_COMM_WORLD);
    } else {
        total_int = local_int;
        for (source = 1; source < comm_sz; source++) {
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_int += local_int;
        }
    }

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n",
                a, b, total_int);
    }

    MPI_Finalize();

    return 0;
} /* main */
```

## • MPI实现-第一版

### – 进程通信

- 进程0接收消息对结果进行汇总
- 其他进程将结果发送给进程0

```
if (my_rank != 0) {  
    MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,  
             MPI_COMM_WORLD);  
} else {  
    total_int = local_int;  
    for (source = 1; source < comm_sz; source++) {  
        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,  
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        total_int += local_int;  
    }  
}
```

```
int main(void) {  
    int my_rank, comm_sz, n = 1024, local_n;  
    double a = 0.0, b = 3.0, h, local_a, local_b;  
    double local_int, total_int;  
    int source;  
  
    MPI_Init(NULL, NULL);  
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
    h = (b-a)/n;          /* h is the same for all processes */  
    local_n = n/comm_sz; /* So is the number of trapezoids */  
  
    local_a = a + my_rank*local_n*h;  
    local_b = local_a + local_n*h;  
    local_int = Trap(local_a, local_b, local_n, h);
```

```
if (my_rank != 0) {  
    MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,  
             MPI_COMM_WORLD);  
} else {  
    total_int = local_int;  
    for (source = 1; source < comm_sz; source++) {  
        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,  
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
        total_int += local_int;  
    }  
}
```

```
/* Print the result */  
if (my_rank == 0) {  
    printf("With n = %d trapezoids, our estimate\n", n);  
    printf("of the integral from %f to %f = %.15e\n",  
           a, b, total_int);  
}  
MPI_Finalize();  
  
return 0;  
} /* main */
```

## ◉ 输入输出

— 每个进程输出一句话

```
#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 */
```

## ● 输入输出

— 每个进程输出一句话

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

int main(void) {
    int my_rank, comm_sz;

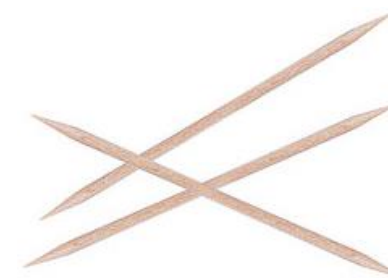
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

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

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

不可预测的输出顺序！

```
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?
```



- MPI实现 – 输入参数 $a, b, n$

- 进程0读入数据，并发送给其他进程

- 大多MPI实现都只允许MPI\_COMM\_WORLD中的进程0访问stdin

```
...  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
Get_input(my_rank, comm_sz, &a, &b, &n);  
  
h = (b-a)/n;  
...
```



## ● MPI实现 – 输入参数 $a, b, n$

– 进程0读入数据，并发送给其他进程

```
...  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
Get_input(my_rank, comm_sz, &a, &b, &n);  
  
h = (b-a)/n;  
...
```

```
void Get_input(int my_rank, int comm_sz, /* input */  
               double* a_p, double* b_p, int* n_p /* output */)   
{  
    if (my_rank == 0) {  
        printf("Enter a, b, and n\n");  
        scanf("%lf %lf %d", a_p, b_p, n_p);  
        for (int 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 */
```

## ● MPI实现 – 输入参数 $a, b, n$

– 进程0读入数据，并发送给其他进程

```
...  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
Get_input(my_rank, comm_sz, &a, &b, &n);  
  
h = (b-a)/n;  
...
```

```
void Get_input(int my_rank, int comm_sz, /* input */  
               double* a_p, double* b_p, int* n_p /* output */)  
{  
    if (my_rank == 0) {  
        printf("Enter a, b, and n: ");  
        scanf("%lf %lf %d", a_p, b_p, n_p);  
        for (int 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);  
        }  
    }  
}
```

繁忙的0号进程！

从其他进程接收运算结果并汇总

```
for (source = 1; source < comm_sz; source++) {  
    MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,  
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
    total_int += local_int;  
}
```

将输入参数发送给其他进程

```
for (int 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);  
}
```

```
    MPI_Status status;  
    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);  
    MPI_Send(&total_int, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);  
} /* Get_input */
```

- 引言
- MPI梯形积分法
- MPI集合通信
- MPI数据打包
- MPI并行排序

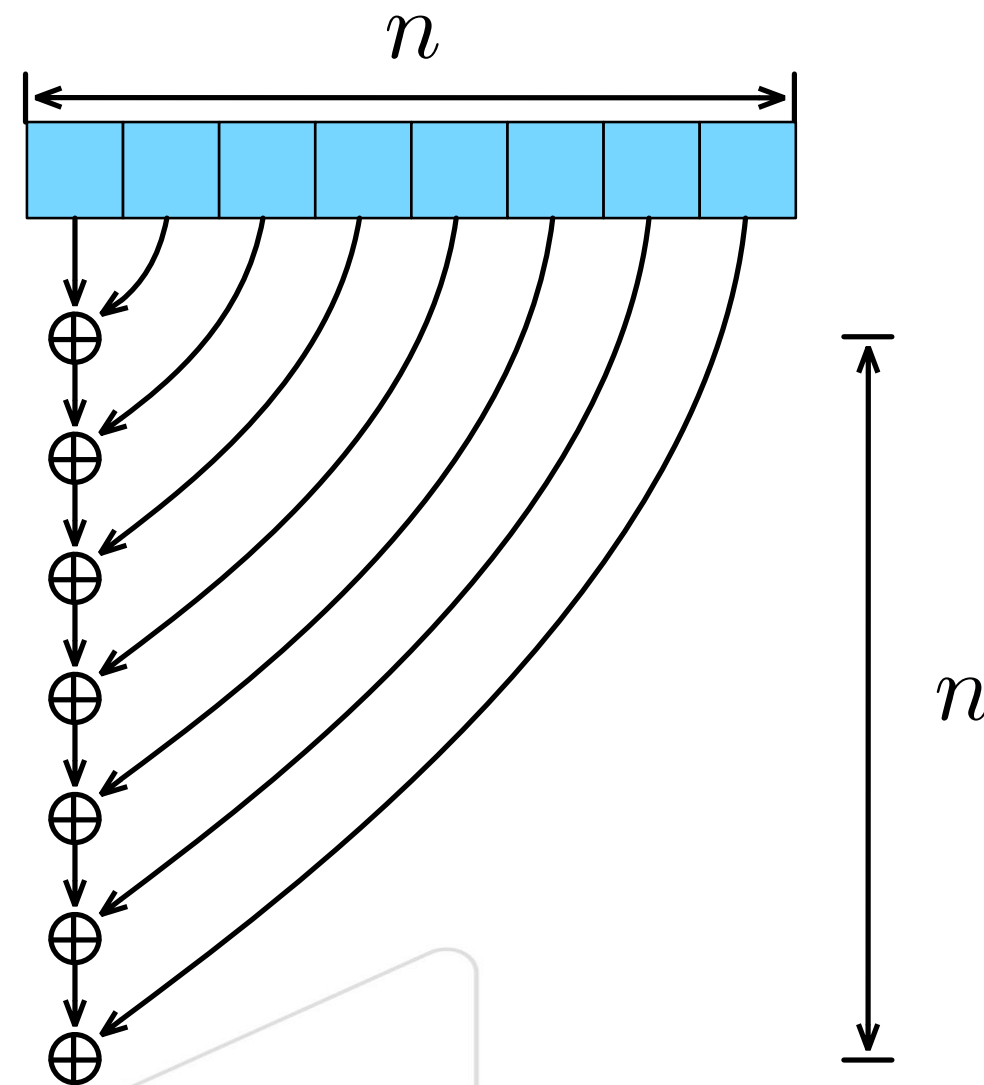
## 梯形积分法的结果汇总

进程0：从其他进程接收运算结果并汇总

```
for (source = 1; source < comm_sz; source++) {  
    MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,  
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
    total_int += local_int;  
}
```

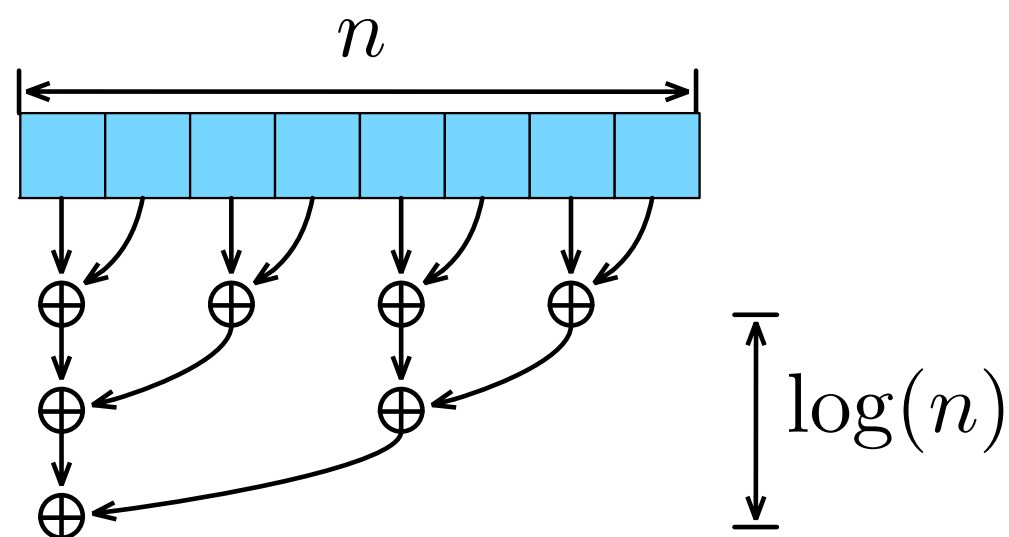
其他进程：向进程0发送运算结果

```
MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,  
        MPI_COMM_WORLD);
```

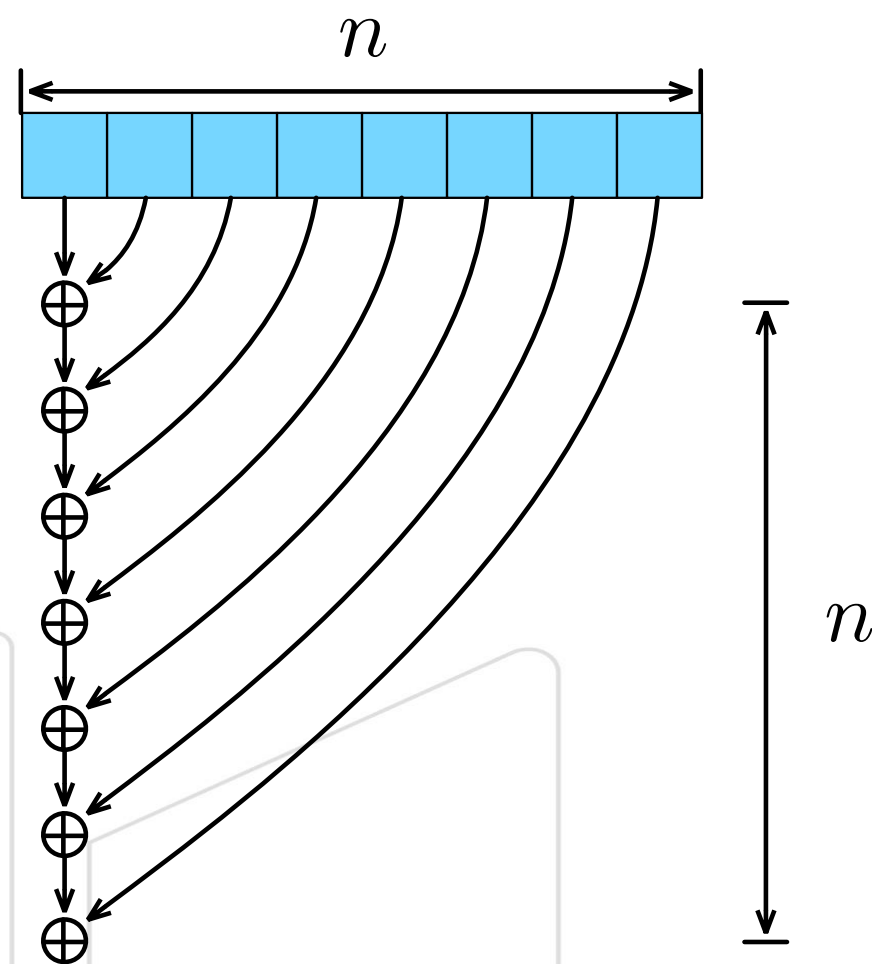


## 并行求和

- 使用二叉树的通信结构
- 执行时间  $O(\log n)$



二叉树结构是否唯一？



## 并行求和

- 使用二叉树的通信结构
- 执行时间  $O(\log n)$

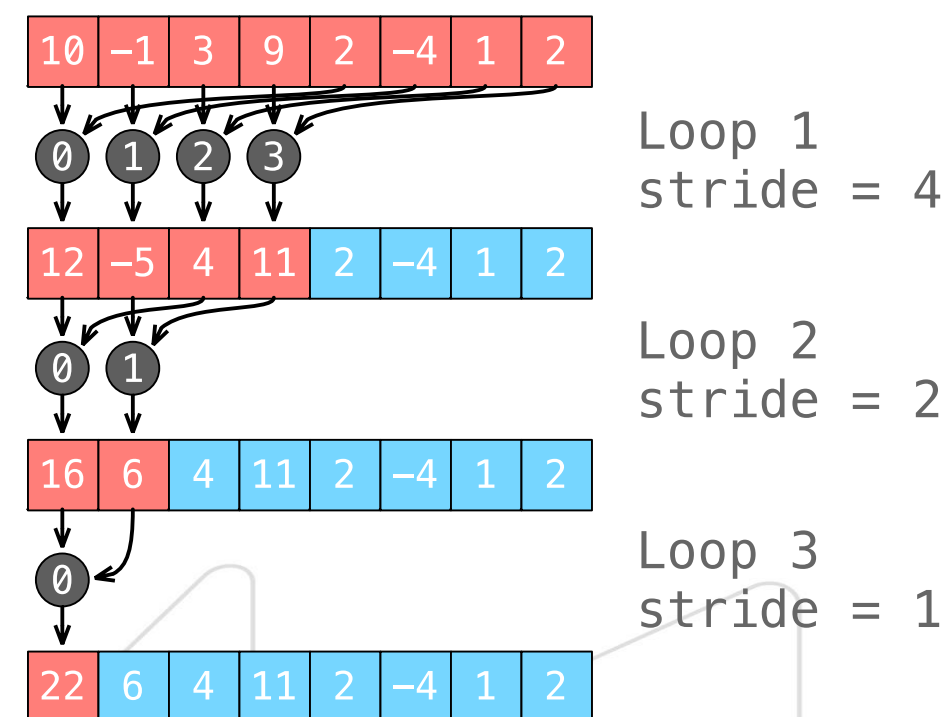
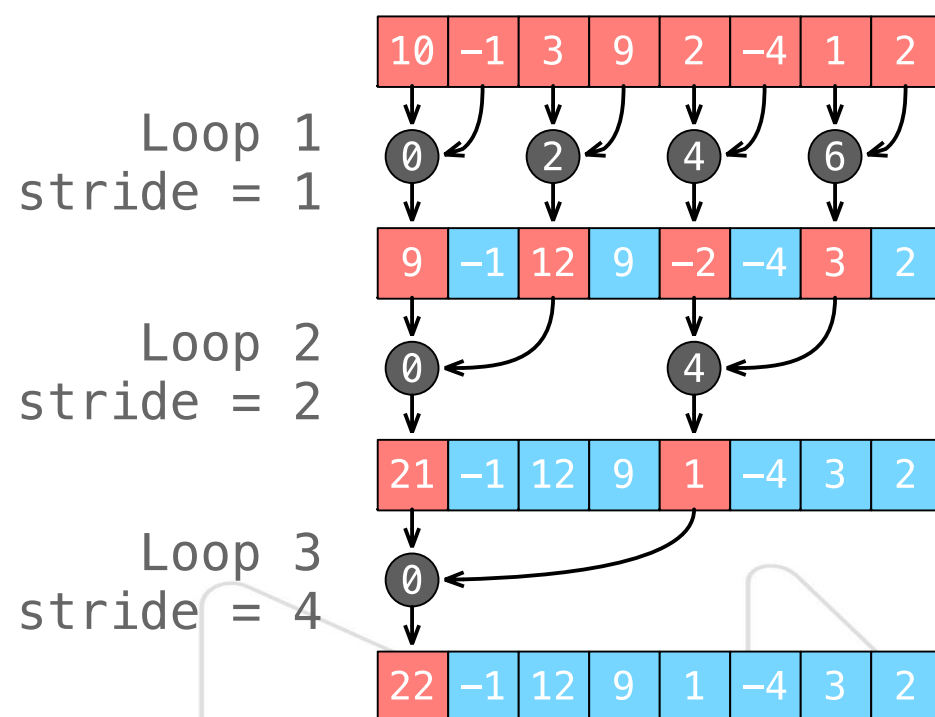


可以有多种等价的二叉树！



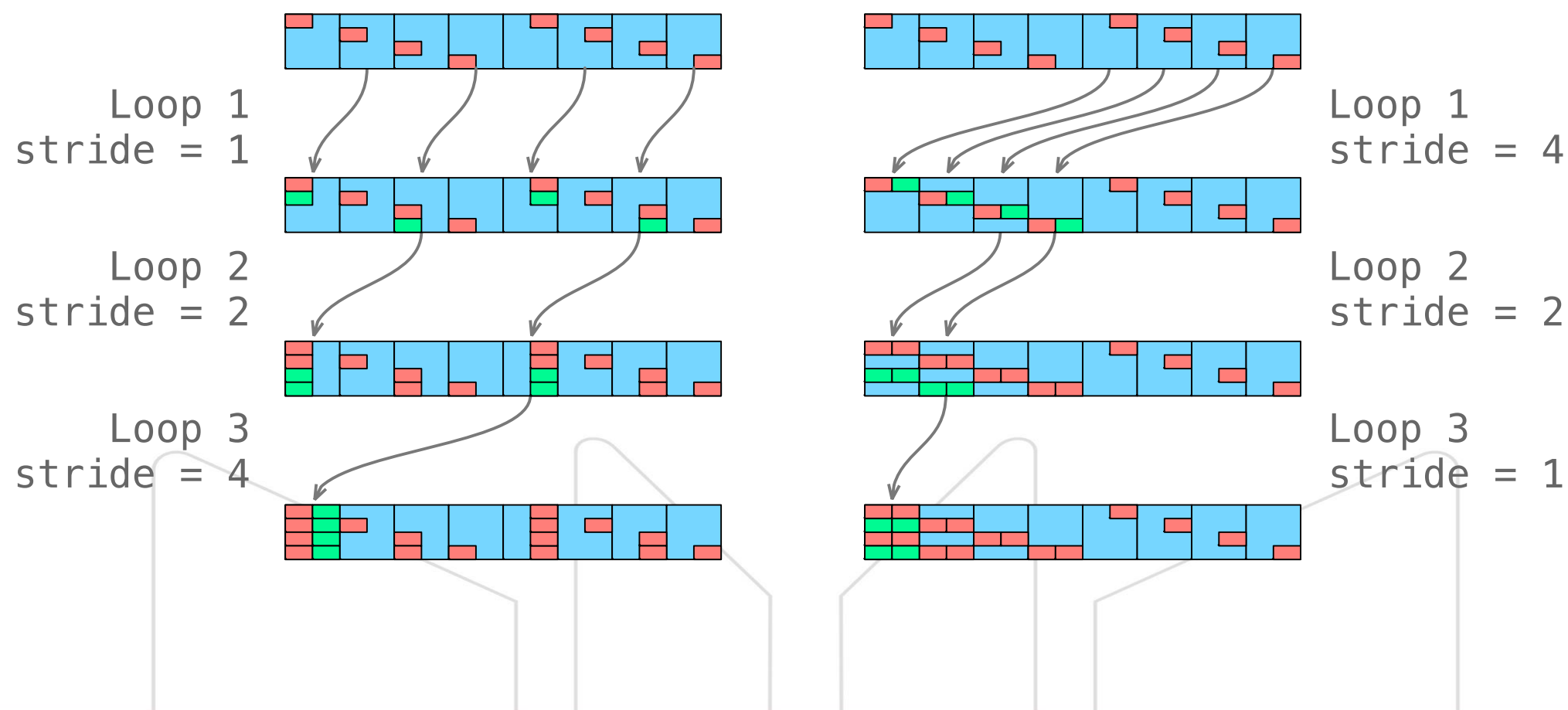
## 并行求和

- 使用二叉树的通信结构
- 执行时间  $O(\log n)$



## 并行求和

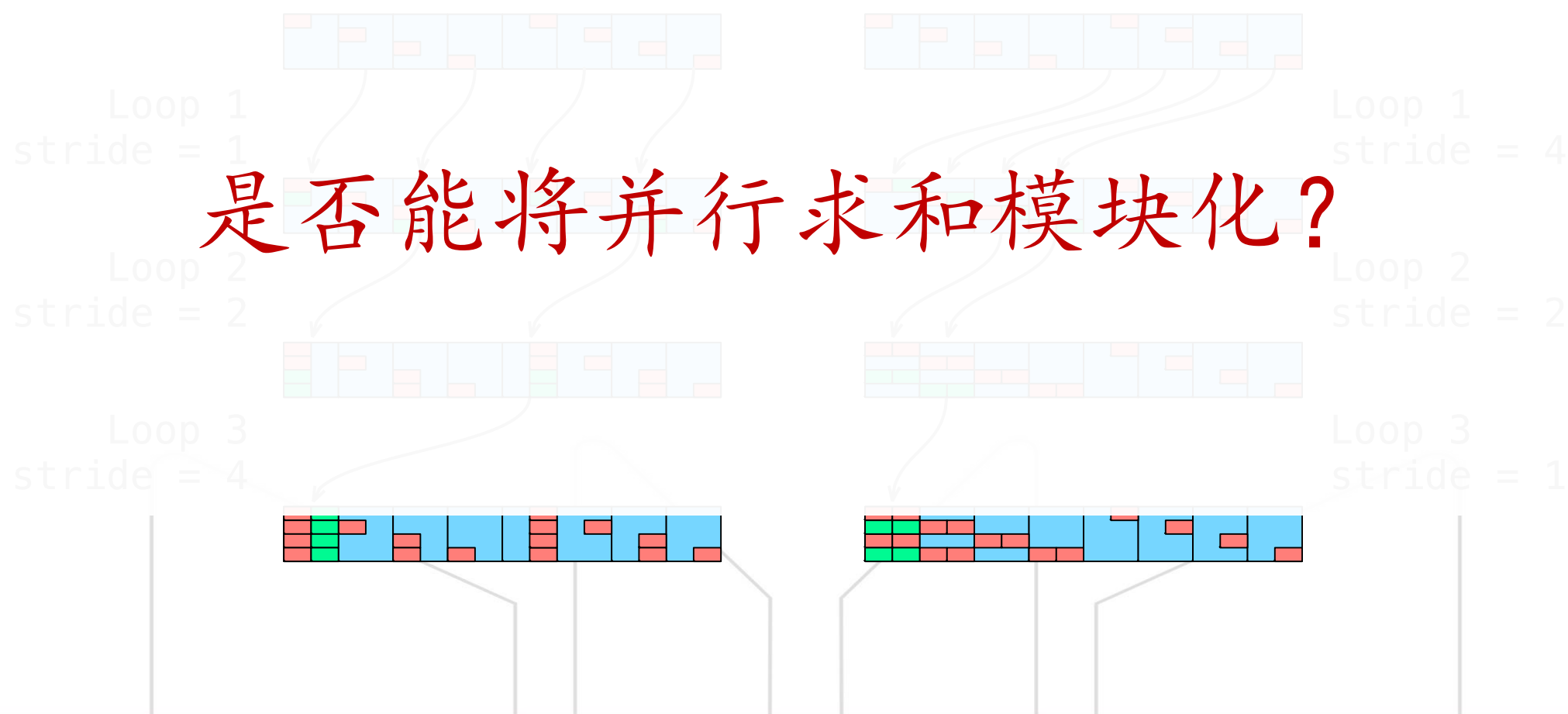
- 使用二叉树的通信结构
- 执行时间  $O(\log n)$



## 并行求和

- 使用二叉树的通信结构
- 执行时间  $O(\log n)$

是否能将并行求和模块化？



## • 点到点通信→集合通信

- 点对点通信：两个进程间一对一的通信，如**MPI\_Send/Recv**
- 集合通信：通信子中所有进程共同参与的通信
  - 需要对常见的通信模式与运算进行抽象
  - 前述并行求和可抽象为并行规约（reduce）

```
int MPI_Reduce(  
    void*      input_data_p, ----- 输入数据指针  
    void*      output_data_p, ----- 输出数据指针  
    int        count, ----- 数据数量  
    MPI_Datatype datatype, ----- 数据类型  
    MPI_Op      operator, ----- 规约操作符  
    int        dest_process, ----- 目标进程  
    MPI_Comm    comm); ----- 通信子
```

## ◉ MPI\_Reduce并行规约

- 规约运算符为**二元运算符**：输入两个对象，输出一个对象
- 规约运算符需满足结合律
- MPI提供了常见的规约运算符
- 可自定义数据类型及规约运算符
  - **MPI\_Type\_create\_struct**
  - **MPI\_Type\_commit**
  - **MPI\_Op\_create**

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

## ◉ MPI\_Reduce并行规约

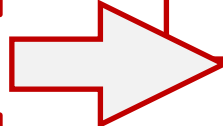
- 使用并行规约实现梯形积分法中的通信
  - 使用**MPI\_SUM**指明规约运算符为求和运算

进程0：从其他进程接收运算结果并汇总

```
for (source = 1; source < comm_sz; source++) {  
    MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,  
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
    total_int += local_int;  
}
```

其他进程：向进程0发送运算结果

```
MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,  
        MPI_COMM_WORLD);
```



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



## ◉ MPI\_Reduce并行规约

- MPI\_Reduce也可以作用于数组上（参数count>1）
- 使用并行规约实现并行直方图计算

```
double local_hist[N], global_hist[N];  
...//compute local histogram  
MPI_Reduce(local_hist, global_hist, N, MPI_DOUBLE,  
           MPI_SUM, 0, MPI_COMM_WORLD);
```

int	MPI_Reduce(		
void*	input_data_p,	-----	输入数据指针
void*	output_data_p,	-----	输出数据指针
int	count,	-----	数据数量
MPI_Datatype	datatype,	-----	数据类型
MPI_Op	operator,	-----	规约操作符
int	dest_process,	-----	目标进程
MPI_Comm	comm);	-----	通信子

## ◉ 集合通信的匹配

- 所有进程都必须调用**相同**的集合通信**函数**
  - 进程A调用**MPI\_Reduce**而B调用**MPI\_Recv**会导致错误（程序悬挂或崩溃）
- 所有进程传递给集合通信的**参数必须兼容**
  - 进程A调用**MPI\_Reduce**并指明目标进程**dest\_process**为0，而进程B调用**MPI\_Reduce**并指明目标进程为1，同样会导致错误（程序悬挂或崩溃）
  - 参数**output\_data\_p**只对**dest\_process**有效，但其他进程仍需传递该参数
- MPI禁止输入**input\_data\_p**输出**output\_data\_p**作为其他参数别名
  - 如**MPI\_Reduce**(&x, &x, **1**, MPI\_DOUBLE, MPI\_SUM, **0**, comm);
  - 使用同一缓冲区作为输入输出，结果不可预测
  - 可能产生错误，可能导致程序崩溃，也可能输出正确结果

## 集合通信的匹配

– 集合通信按顺序匹配，不需要消息标签（tag）

Time	Process 0	Process 1	Process 2
0	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>
1	<code>MPI_Reduce(&amp;a, &amp;b, ...)</code>	<code>MPI_Reduce(&amp;c, &amp;d, ...)</code>	<code>MPI_Reduce(&amp;a, &amp;b, ...)</code>
2	<code>MPI_Reduce(&amp;c, &amp;d, ...)</code>	<code>MPI_Reduce(&amp;a, &amp;b, ...)</code>	<code>MPI_Reduce(&amp;c, &amp;d, ...)</code>

- 假设目标进程为0号进程，按上述方式运行
- 结果并非  $b_0 = \sum a_i = 3, d_0 = \sum c_i = 6$
- 而是  $b_0 = a_0 + c_1 + a_2 = 4, d_0 = c_0 + a_1 + c_2 = 5$

– 目前，只有一个进程获得最终结果...

## 集合通信的匹配

– 集合通信按顺序匹配，不需要消息标签（tag）

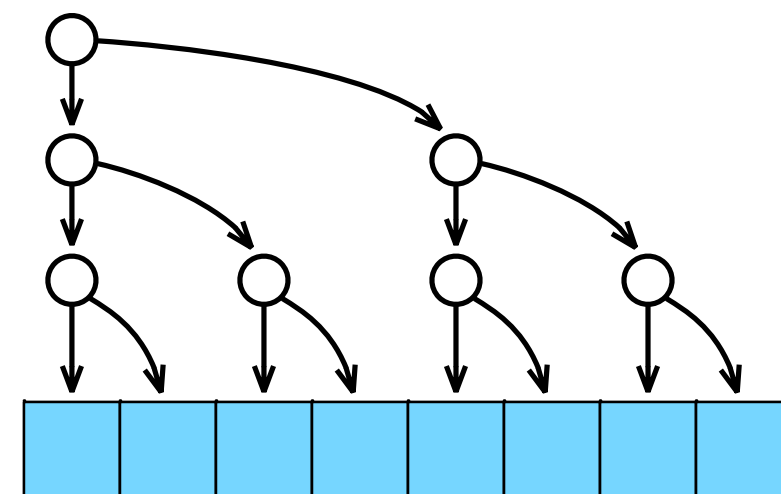
Time	Process 0	Process 1	Process 2
0	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>
1	<code>MPI_Reduce(&amp;a, &amp;b, ...)</code>	<code>MPI_Reduce(&amp;c, &amp;d, ...)</code>	<code>MPI_Reduce(&amp;a, &amp;b, ...)</code>
2	<code>MPI_Reduce(<sup>data_p</sup>&amp;c, &amp;d, ...)</code>	<code>MPI_Reduce(&amp;a, &amp;b, ...)</code>	<code>MPI_Reduce(&amp;c, &amp;d, ...)</code>

- 假设目标进程为0号进程，按上述方式运行
- 结果并非  $b_0 = \sum a_i = 3, d_0 = \sum c_i = 6$
- 而是  $b_0 = a_0 + c_1 + a_2 = 4, d_0 = c_0 + a_1 + c_2 = 5$

– 目前，只有一个进程获得最终结果...

## ● MPI\_Bcast广播

- 将属于一个进程的数据发送到通信子中的所有进程
  - 注意`data_p`既用作输入，也用作输出
  - 对于源进程，指明了需要发送数据的指针
  - 对于其他进程，指明了接收数据的缓冲区指针
- 使用树形结构可在  $\log n$  时间内完成



```
int MPI_Bcast(  
    void*      data_p, ----- 指向数据/接收缓冲区的指针  
    int        count, ----- 数据大小  
    MPI_Datatype datatype, ----- 数据类型  
    int        source_proc, ----- 数据源进程编号  
    MPI_Comm   comm); ----- 参与广播的通信子
```

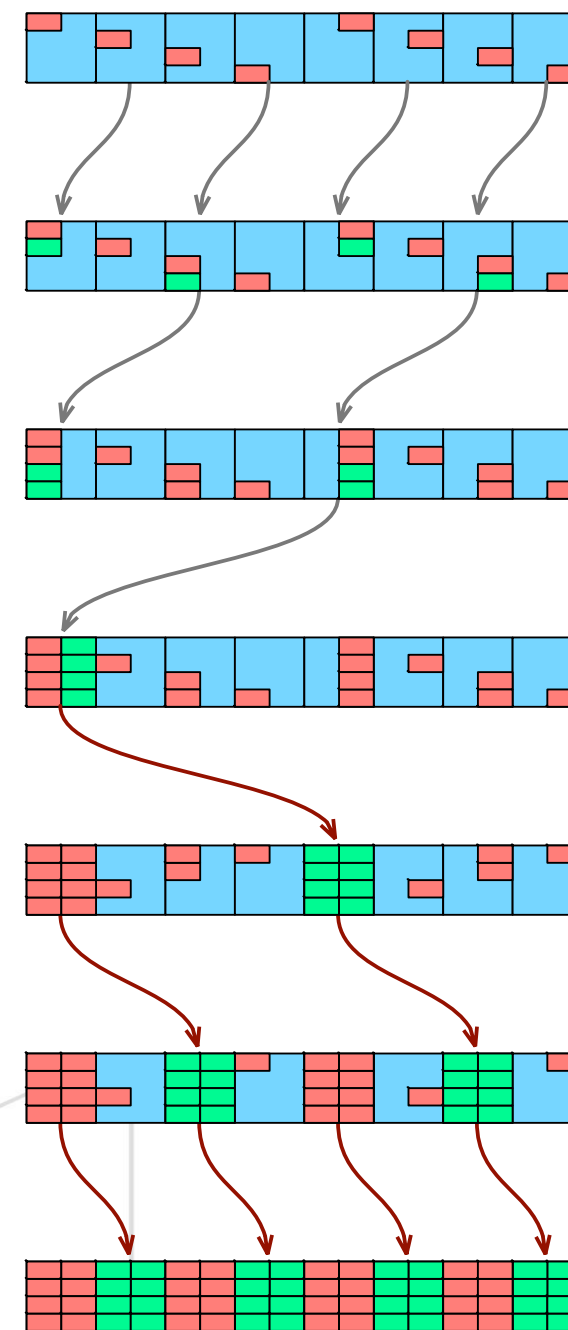
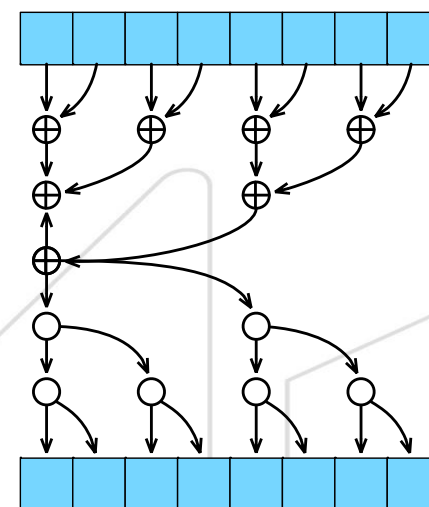
## • MPI\_Allreduce并行全局规约

– 执行并行规约，并保证结果对所有进程可见

```
int MPI_AllReduce(  
    void*      input_data_p, ----- 输入数据指针  
    void*      output_data_p, ----- 输出数据指针  
    int        count, ----- 数据数量  
    MPI_Datatype datatype, ----- 数据类型  
    MPI_Op      operator, ----- 规约操作符  
    MPI_Comm    comm); ----- 通信子
```

– 先规约，再广播

- $\oplus$ : 规约运算符
- $\bigcirc$ : 分发操作
- 是否是最高效的实现方式?

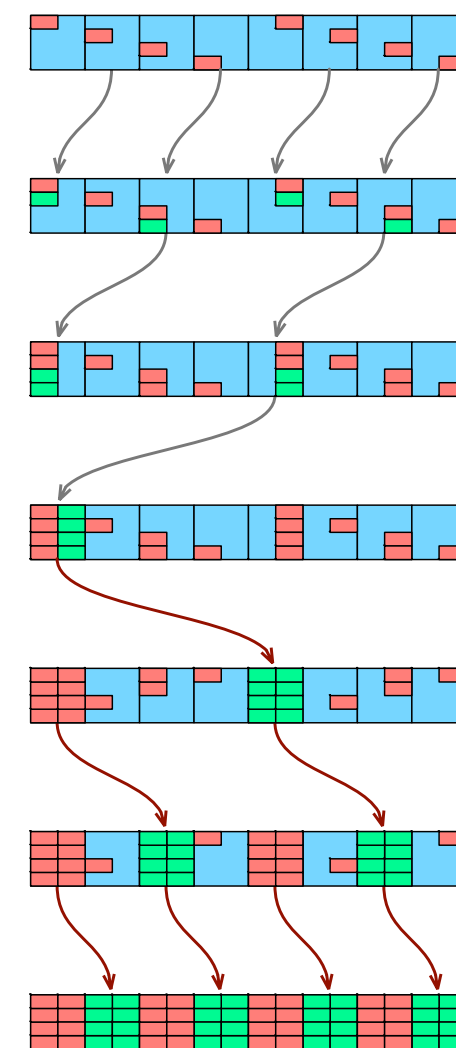
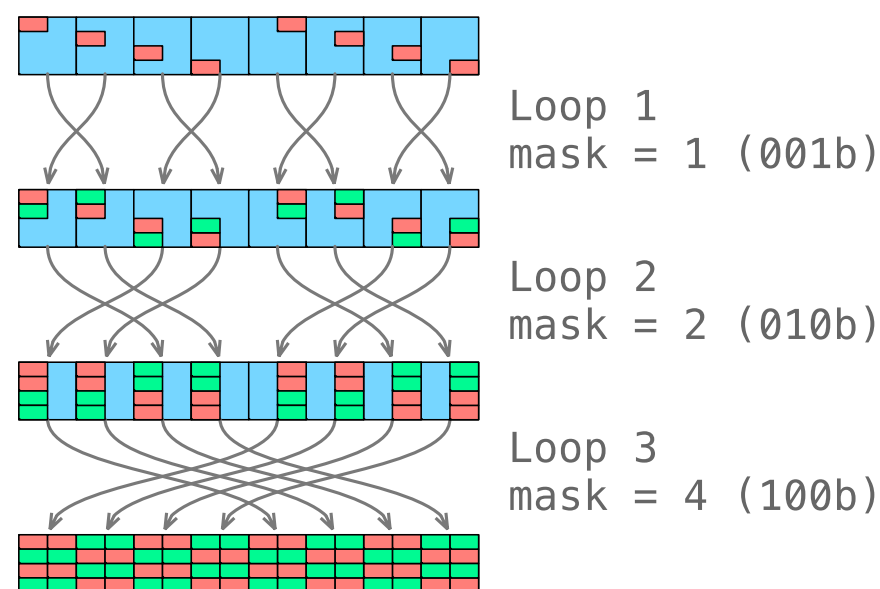
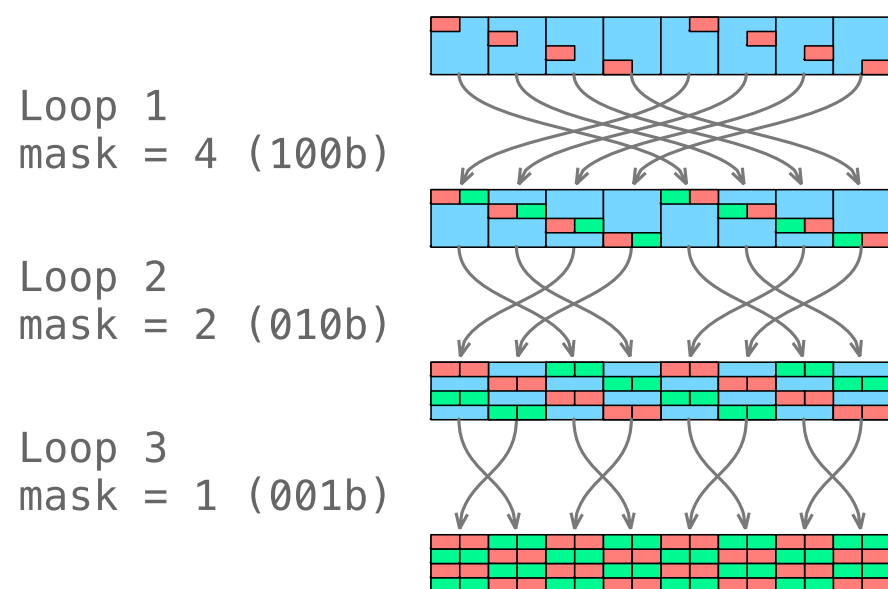




## • MPI\_Allreduce并行全局规约

– 执行并行规约，并保证结果对所有进程可见

- 蝶形通信：每轮中所有进程都参与通信
  - 进程与配对进程两两交换数据；配对方式不唯一
  - 总通信次数  $n \cdot \log n$ ；轮数  $\log n$
- 树形通信（规约+广播）：每轮中只有指定进程参与通信
  - 总通信次数  $2 \cdot (n - 1)$ ；轮数  $2 \cdot \log n$



## ● 回顾此前的参数分发（使用点对点通信）

```
void Get_input(int my_rank, int comm_sz, /* input */
               double* a_p, double* b_p, int* n_p /* output */)
{
    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (int 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 */
```

## 改进的参数分发（使用广播）

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

是否能进一步提升效率？

```
void Get_input(int my_rank, int comm_sz, /* input */
               double* a_p, double* b_p, int* n_p /* output */)
{
    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (int 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 */
```

## ◉ 数据分发策略

- 考虑向量和:  $z_i = x_i + y_i$
- 划分: 对应元素相加
- 通信: 结果汇总至一个进程 (一段内存)
- 聚合与映射: ?

```
void vector_sum(double x[], double y[], double z[],  
                int n)  
{  
    for (int i = 0; i < n; i++)  
        z[i] = x[i] + y[i];  
} /* vector_sum */
```

## ◉ 数据分发策略

– 映射：划分方式决定后，由每个进程处理其负责的数据项即可

```
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 */
```

## 数据分发策略

- 块划分：进程处理一段连续的数据块
- 循环划分：进程依次获取一个数据，重复该过程
  - Round Robin（循环/轮转策略）
- 循环-块划分：进程依次获取一个固定大小数据块，重复该过程

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



## ◉ 数据分发: **MPI\_Scatter** 散射

— 源进程将**数据分块**分别发送至通信子中**不同的对应**进程

- 数据块0发送给进程0，数据块1发送给进程1，以此类推
- 源进程自身也参与数据接收
- 注意: **send\_size**为发送的单个数据块大小，而非发送缓冲大小

```
int MPI_Scatter(  
    void*      send_buf_p, ----- 发送缓冲地址  
    int        send_size, ----- 发送数据块大小  
    MPI_Datatype send_type, ----- 发送数据类型  
    void*      recv_buf_p, ----- 接收缓冲地址  
    int        recv_size, ----- 接收数据块大小  
    MPI_Datatype recv_type, ----- 接收类型  
    int        source_proc, ----- 数据来源进程编号  
    MPI_COMM   communicator); ----- 通信子
```

## ◉ 数据分发: MPI\_Scatter 散射

– 使用MPI\_Scatter实现向量加法中的块划分

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

    if (my_rank == 0) {
        a = malloc(n*sizeof(double));
        printf("Enter the vector %s\n", vec_name);
        for (int 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 */
```

## ◉ 数据接收: **MPI\_Gather** 聚集

– 将通信子中各进程的数据块聚集到一个目标进程中

- 进程0的数据块占据目标进程接收缓冲的第0段，进程1数据占据第1段...
- 目标进程本身也参与发送
- 同样，**recv\_size**指的是接收的单个数据块大小，而非接收数据总大小

```
int MPI_Gather(  
    void*      send_buf_p, ----- 发送缓冲地址  
    int        send_size, ----- 发送数据块大小  
    MPI_Datatype send_type, ----- 发送数据类型  
    void*      recv_buf_p, ----- 接收缓冲地址  
    int        recv_size, ----- 接收数据块大小  
    MPI_Datatype recv_type, ----- 接收类型  
    int        dest_proc, ----- 目标进程编号  
    MPI_COMM   communicator); ----- 通信子
```

## ◉ 数据接收: MPI\_Gather 聚集

– 使用MPI\_Gather实现向量加法结果的汇总与打印

```
void Print_vector(double local_b[], int local_n, int n,
                  int my_rank, MPI_Comm comm)
{
    double* b = NULL;

    if (my_rank == 0) {
        b = malloc(n*sizeof(double));
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
        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 */
```

- ◉ 数据收发: **MPI\_Allgather**全局聚集
  - 从所有进程的发送缓冲`send_buf_p`中接收数据
  - 并分发至所有进程的接收缓冲`recv_buf_p`
  - 无需指定源/目标进程
  - 同样, `send_size`为单个数据块大小, 而非总大小

```
int MPI_Allgather(  
    void*      send_buf_p, ----- 发送缓冲地址  
    int        send_size, ----- 发送数据块大小  
    MPI_Datatype send_type, ----- 发送数据类型  
    void*      recv_buf_p, ----- 接收缓冲地址  
    int        recv_size, ----- 接收数据块大小  
    MPI_Datatype recv_type, ----- 接收类型  
    MPI_COMM    communicator); ----- 通信子
```

## 数据收发: MPI\_Allgather 全局聚集

– 例: 矩阵向量乘法  $y = A \cdot x$

- $A = (a_{ij})$  为  $m \times n$  的矩阵,  $x$  为  $n$  维向量
- $y = A \cdot x$  为  $m$  维向量
  - $y$  的第  $i$  个元素  $y_i = \sum a_{ij}x_j$  ( $A$  的第  $i$  行与向量  $x$  的点积)
- 划分: 子任务为向量点积  $y_i = \sum a_{ij}x_j$  (标量乘法作为点积划分过细)
- 聚合: 每个进程处理多个连续行 (块划分)
  - 是否可以使用循环划分/块-循环划分?

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

$x_0$
$x_1$
$\vdots$
$x_{n-1}$

 $=$ 

$y_0$
$y_1$
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
$\vdots$
$y_{m-1}$

```

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



## ◉ 数据收发: **MPI\_Allgather**全局聚集

— 例: 矩阵向量乘法  $y = A \cdot x$

- 假设输入  $A$ ,  $x$  都分块, 可通过 **MPI\_Allgather** 使所有进程能访问完整  $x$

```
void Mat_vect_mult(double local_A[], double local_x[],  
                  double local_y[], /* output */  
                  int local_m, int n, int local_n, MPI_Comm comm)  
{  
    double* x = malloc(n*sizeof(double));  
    MPI_Allgather(local_x, local_n, MPI_DOUBLE,  
                  x, local_n, MPI_DOUBLE, comm);  
  
    for (int local_i = 0; local_i < local_m; local_i++) {  
        local_y[local_i] = 0.0;  
        for (int j = 0; j < n; j++)  
            local_y[local_i] += local_A[local_i*n+j]*x[j];  
    }  
    free(x);  
} /* Mat_vect_mult */
```

## 数据收发: **MPI\_Allgather**全局聚集

- 全局聚集的实现: ring vs. recursive doubling
- Ring: 每次从右邻居接收一个数据块, 并向左邻居发送一个数据

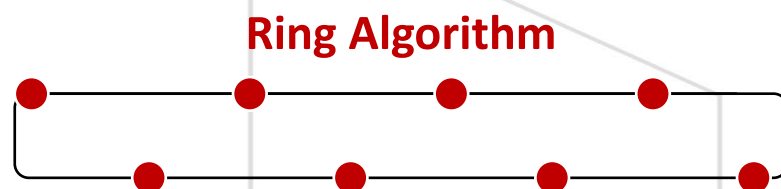
- 时间消耗:  $t_{ring} = l \cdot (p - 1) + b \cdot \frac{n}{p} (p - 1)$

- Recursive doubling: 每次交换当前所有数据

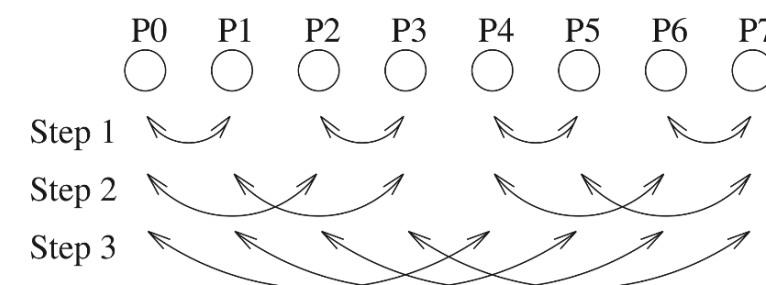
- 时间消耗:  $t_{double} = l \cdot \log p + b \cdot \frac{n}{p} (p - 1)$

- 总交换数据  $\frac{n}{p} (1 + 2 + \dots + 2^{\log p - 1}) = \frac{n}{p} (2^{\log p} - 1) = \frac{n}{p} (p - 1)$

- $l$ : 延迟;  $b$ : 带宽反比



### Recursive Doubling Algorithm



- 数据收发: **MPI\_Allgather**全局聚集
  - 全局聚集的实现: ring vs. recursive doubling
    - 实测时间对比

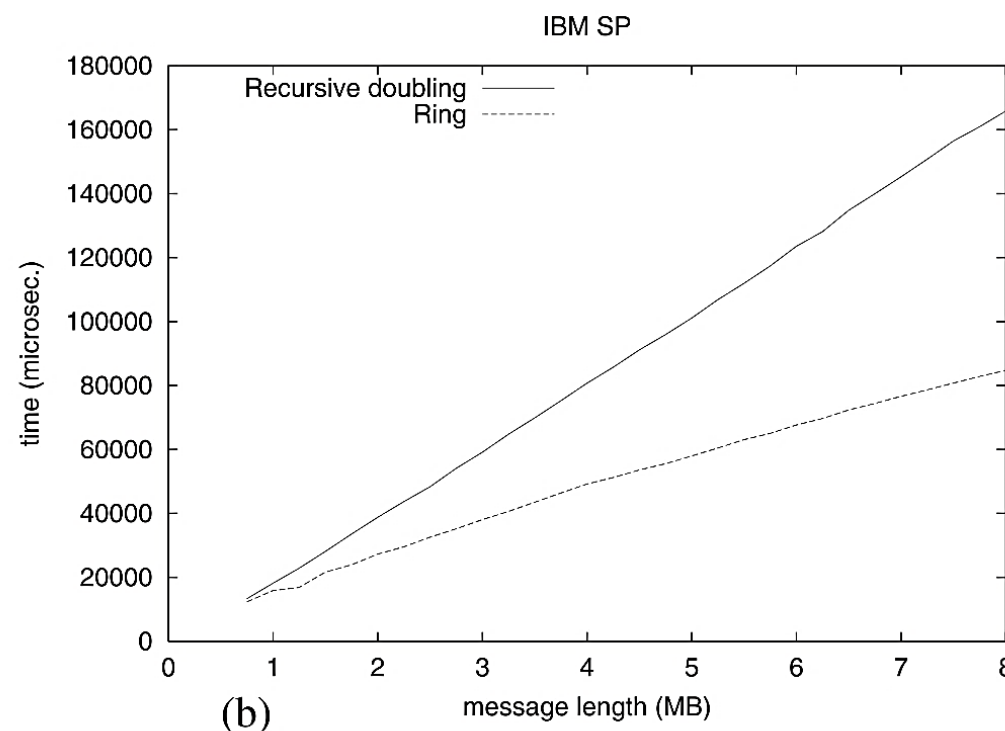
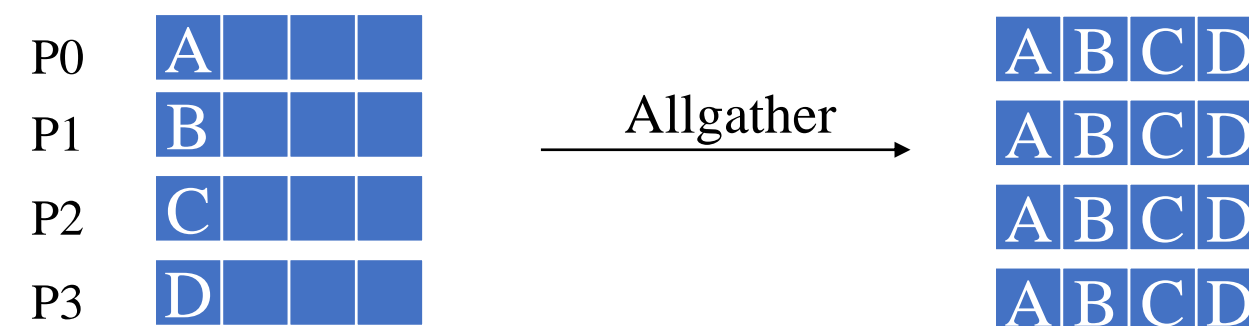
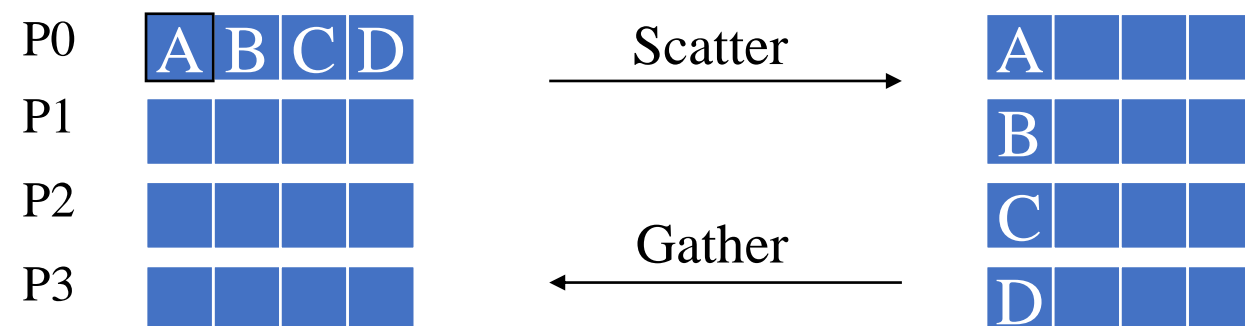
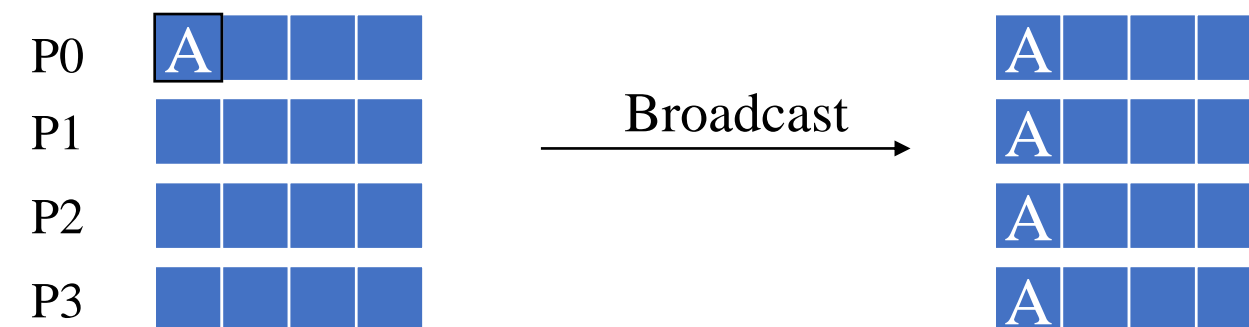
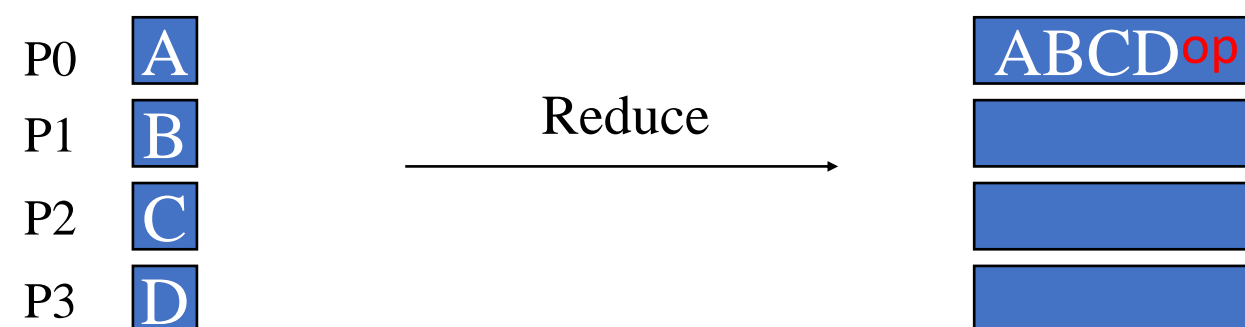


Fig. 5 Ring algorithm versus recursive doubling for long-message allgather (64 nodes). The size on the x-axis is the total amount of data gathered on each process.

## ● MPI集合通信回顾



- 引言
- MPI梯形积分法
- MPI集合通信
- MPI数据打包
- MPI并行排序

## 如何发送多个数据？

– 假设需要发送一个长度为1000的数组，使用循环显然不是好方式

```
if (my_rank == 0) {  
    for (int i = 0; i < 1000; ++i)  
        MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm)  
} else {  
    for (int i = 0; i < 1000; ++i)  
        MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status)  
}
```

– 一次性发送整个数组显然更有效率

```
if (my_rank == 0) {  
    MPI_Send(&x[0], 1000, MPI_DOUBLE, 1, 0, comm)  
} else {  
    MPI_Recv(&x[0], 1000, MPI_DOUBLE, 0, 0, comm, &status)  
}
```

–  $t_1 = 1000 \cdot l + 1000 \cdot b; t_{1000} = l + 1000 \cdot b$

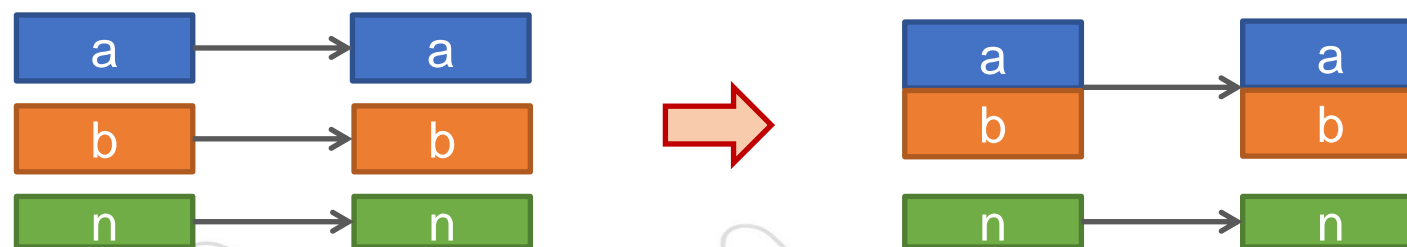


## 如何发送多个不同类型数据？

— 如，此前梯形积分法中的参数分发

```
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);
```

— 将变量a\_p, b\_p置于一个数组中，可将广播次数从3次降为2次



— 是否能使用一次广播？

## • 如何发送多个不同类型数据？

- 一次广播：将a\_p, b\_p, n\_p置于一块连续内存空间
  - 使用C语言中的强制类型转换
    - 需要进行手动编解码，可读性不高

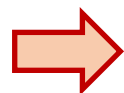
```
char data[20];  
if (my_rank == 0){  
    printf("Enter a, b, and n\n");  
    scanf("%lf %lf %d", a_p, b_p, n_p);  
    *((double*)(data)) = a_p;  
    *((double*)(data+8)) = b_p;  
    *((int*)(data+16)) = n_p;  
    MPI_Bcast(data, 20, MPI_CHAR, 0, MPI_COMM_WORLD);  
} else {  
    MPI_Bcast(data, 20, MPI_CHAR, 0, MPI_COMM_WORLD);  
    a_p = *((double*)(data));  
    b_p = *((double*)(data+8));  
    n_p = *((int*)(data+16));  
}
```

a:	0
b:	8
n:	16

## 如何发送多个不同类型数据？

- 一次广播：将a\_p, b\_p, n\_p置于一块连续内存空间
  - 使用内联，无需手动编解码
  - 是否有MPI解决方案？

```
char data[20];
if (my_rank == 0){
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", a_p, b_p, n_p);
    *((double*)(data)) = a_p;
    *((double*)(data+8)) = b_p;
    *((int*)(data+16)) = n_p;
    MPI_Bcast(data, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
} else {
    MPI_Bcast(data, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
    a_p = *((double*)(data));
    b_p = *((double*)(data+8));
    n_p = *((int*)(data+16));
}
```



```
union param {
    char data[20];
    struct {
        double a, b;
        int n;
    };
};

param p;
if (my_rank){
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", p.a, p.b, p.n);
}
MPI_Bcast(p.data, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
```

a:	0
b:	8
n:	16

## 如何发送多个不同类型数据？

– 如何描述一个数据类型？：有什么，有多少，在哪儿？

- 2个变量
- 1: double类型，偏移量0，长度2
- 2: int类型，偏移量16，长度1

```
struct {  
    double ab[2];  
    int n;  
};
```

– 使用**MPI\_Type\_create\_struct**定义MPI派生结构体类型

```
int MPI_Type_create_struct(  
    int          count, ----- 类型中的变量个数  
    int          array_of_blocklengths[], ----- 变量包含的元素个数  
    MPI_Aint     array_of_displacements[], ----- 变量的偏移量  
    MPI_Datatype array_of_types[], ----- 变量中元素的类型  
    MPI_Datatype* new_type_p); ----- 返回的派生类型
```

## 定义MPI派生类型

– 如何获取**MPI\_Type\_create\_struct**所需信息？

- **count**: 自行指明变量个数 (3)
- **array\_of\_blocklengths**: 自行指明每个变量的元素个数 ({1, 1, 1})
- **array\_of\_displacements**: 使用**MPI\_Get\_address**获取

a:	0
b:	8
n:	16

```
int MPI_Get_address(  
    void*      location_p, ----- 输入: 指向变量的指针  
    MPI_Aint*  address_p); ----- 输出: 地址偏移量
```

- **array\_of\_types**: 自行指明每个变量的数据类型
  - {MPI\_DOUBLE, MPI\_DOUBLE, MPI\_INT}

## ◉ 定义MPI派生类型

– 使用创建的派生类型前，还需要向MPI系统**指定**该类型

- `int MPI_Type_commit(MPI_Datatype* new_mpi_t_p)`
- 允许MPI实现在通信函数内使用该类型
- 优化数据类型的内部表示

– 使用完成后，需要释放为该类型分配的相关资源

- `int MPI_Type_free(MPI_Datatype* old_mpi_t_p)`

## 定义MPI派生类型：创建派生类型

```
void Build_mpi_type(double* a_p, double* b_p, 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};

    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 */
```



- 定义MPI派生类型：使用派生类型通信

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

## ◉ 定义MPI派生类型

- 多个相同类型元素组成的连续数据类型
  - 输入的原有类型可以是MPI定义的基本类型，也可以是派生类型

```
int MPI_Type_contiguous(  
    int count, ----- 数量  
    MPI_Datatype old_type, ----- 原有类型  
    MPI_Datatype* new_type_p); ----- 返回新创建类型
```

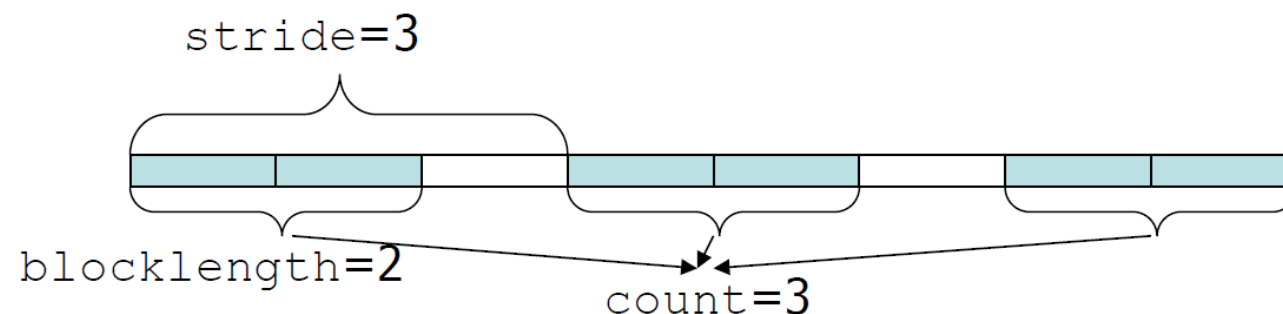
## – 使用举例

```
MPI_Datatype newtype;  
MPI_Type_contiguous(3, MPI_INT, &newtype);  
MPI_Type_commit(&newtype);  
  
int data[3] = {1, 2, 3};  
  
MPI_Send(data, 1, newtype, 1, 0, MPI_COMM_WORLD);  
MPI_Type_free(&newtype);
```

## 定义MPI派生类型

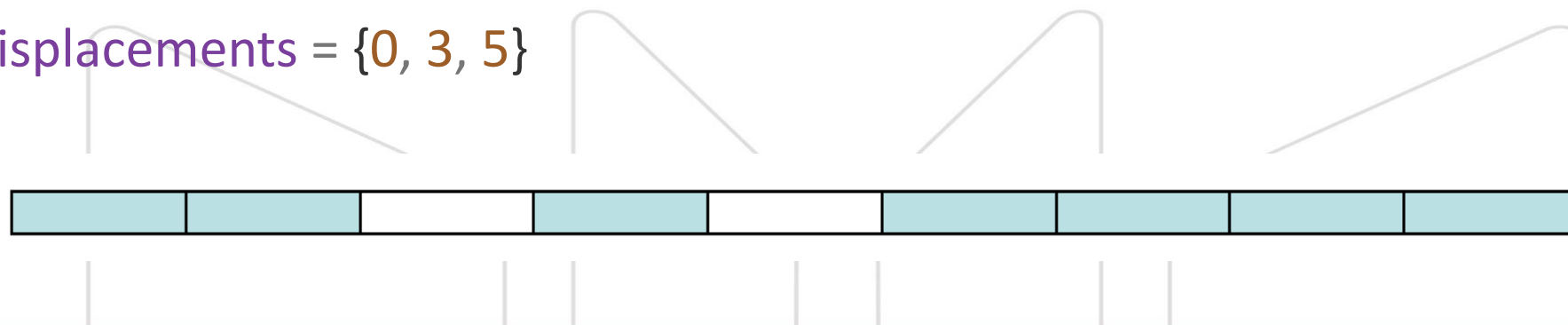
### – 多个相同类型向量组成的向量数据类型

- **MPI\_Type\_vector**(count, block\_length, stride, ...);



### – 多个相同类型数据块组成的索引数据类型

- **MPI\_Type\_indexed**(count, block\_lengths, displacements, ...);
  - `block_lengths = {2, 1, 4}`
  - `displacements = {0, 3, 5}`



## 定义MPI派生类型

### – 从原有数据类型中重新定义边界

- **MPI\_Type\_create\_resized** (data\_type, left\_bound, extent, ...);
- left\_bound = 4, extent = 12



### – 多维数组的子数组

```
int MPI_Type_create_subarray(int ndims, const int *array_of_sizes,  
    const int *array_of_subsizes, const int *array_of_starts,  
    int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- ndims, array\_of\_sizes: 指明数组维度，每个维度上的大小
- array\_of\_subsizes/starts: 指明子数组每个维度的大小/起点
- order: 行/列主序 (MPI\_ORDER\_C, MPI\_ORDER\_FORTRAN)

## ● 使用MPI\_Pack打包MPI\_Unpack解包

```
int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,  
             void *outbuf, int outsize, int *position, MPI_Comm comm);
```

- incount为输入数组元素数量
- outsize为输出缓冲区大小（字节）

```
int MPI_Unpack(const void *inbuf, int insize, int *position,  
               void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm);
```

- insize为输入缓冲区大小（字节）
- outcount为输出数据数组数量
- position为缓冲区内的偏移量，在打包/解包过程中将被更新

## ● 使用MPI\_Pack打包MPI\_Unpack解包

```
// 打包数据
int position = 0;
MPI_Pack(sendbuf, 3, MPI_INT, recvbuf, 12, &position,
MPI_COMM_WORLD);

// 打包后position将被更新, 下次打包时无需手动设置position

// 解包数据
position = 0;
MPI_Unpack(recvbuf, 12, &position, recvbuf, 3, MPI_INT,
MPI_COMM_WORLD);
```

## ● 使用MPI\_Wtime获取墙钟时间

- 使用获取进程中运行时间的最大/最小/平均值
- 进程可能不同时开始，难以评估总体并行性能

```
local_time = MPI_Wtime(); /*get time just before work section */
work();
local_time = MPI_Wtime() - local_time; /*get time just after work section*/ /*compute max, min, and average timing
statistics*/

MPI_Reduce(&local_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&local_time, &min_time, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
MPI_Reduce(&local_time, &avg_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (myrank == 0) {
    avg_time /= num_procs;
    printf("Min: %lf Max: %lf Avg: %lf\n", min_time, max_time, avg_time);
}
```



## ● 使用MPI\_Barrier同步通信子中进程

– 通信子中进程全部调用MPI\_Barrier后，函数调用才返回

➔ MPI\_Barrier(comm);

```
local_time = MPI_Wtime(); /*get time just before work section */
```

```
work();
```

```
local_time = MPI_Wtime() - local_time; /*get time just after work section*/ /*compute max, min, and average timing statistics*/
```

```
MPI_Reduce(&local_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

```
MPI_Reduce(&local_time, &min_time, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
```

```
MPI_Reduce(&local_time, &avg_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
if (myrank == 0) {
```

```
    avg_time /= num_procs;
```

```
    printf("Min: %lf Max: %lf Avg: %lf\n", min_time, max_time, avg_time);
```

```
}
```

## ● 计时与性能分析（矩阵向量乘法）

–  $p$  进程并行运行时间  $T_p$ ，串行运行时间  $T_1$

– 加速比:  $S_p = \frac{T_1}{T_p}$

– 效率:  $E_p = \frac{S_p}{p}$

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

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

并行效率

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

- 引言
- MPI梯形积分法
- MPI集合通信
- MPI数据打包
- MPI并行排序

## ◉ 问题定义

- 给定一个由 $n$ 个元素组成的序列
  - 可表示为数组或链表
    - 排序算法可能依赖于具体数据结构
    - 大多数算法都能使用数组
    - 部分算法不适合链表：如，快速排序，希尔排序（需要index）
  - $A=[2, 5, 4, 3, 6, 1, 3]$
- 将其中元素按一定顺序排列
  - 从序列中任选一对元素都是有序的，则序列为已排序的
  - $\text{sorted}(A)=[1, 2, 3, 3, 4, 5, 6]$

## ◉ 稳定性

– 稳定的排序算法中具有相同值的元素相对顺序**不会**发生变化

- 稳定:  $\text{sorted}(A)=[1, 2, \textcolor{red}{3}, \textcolor{blue}{3}, 4, 5, 6]$
- 不稳定:  $\text{sorted}(A)=[1, 2, \textcolor{blue}{3}, \textcolor{red}{3}, 4, 5, 6]$

– 稳定性的意义

- 对复杂对象的多个属性分别进行处理时，需要在排序中保持原有顺序
- 不稳定排序算法可以通过对元素值进行处理而符合稳定性要求

– 稳定的排序

- 冒泡排序、插入排序、基数排序、桶排序等

– 不稳定的排序

- 快速排序、希尔排序、选择排序等

## 插入排序

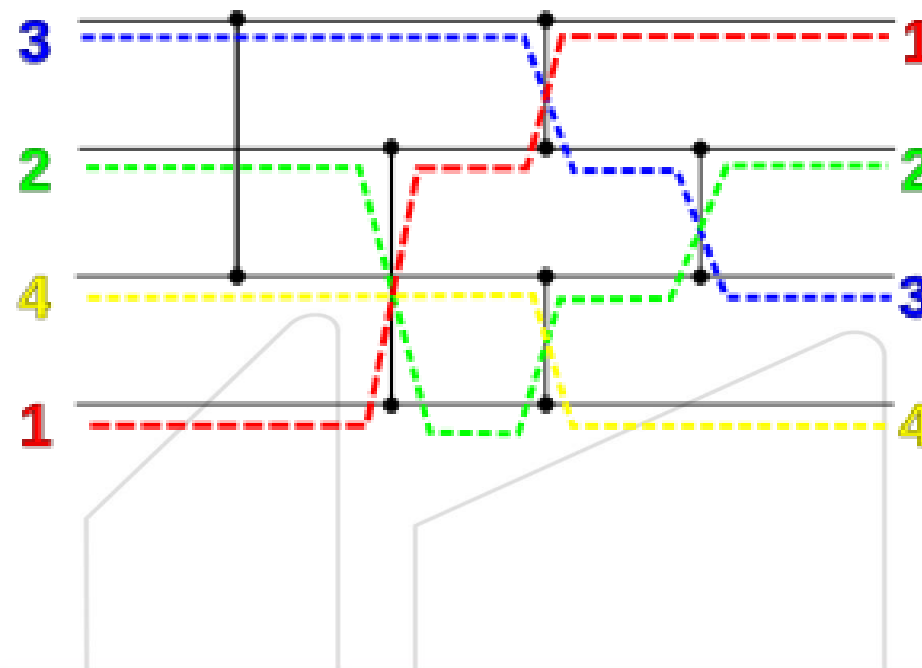
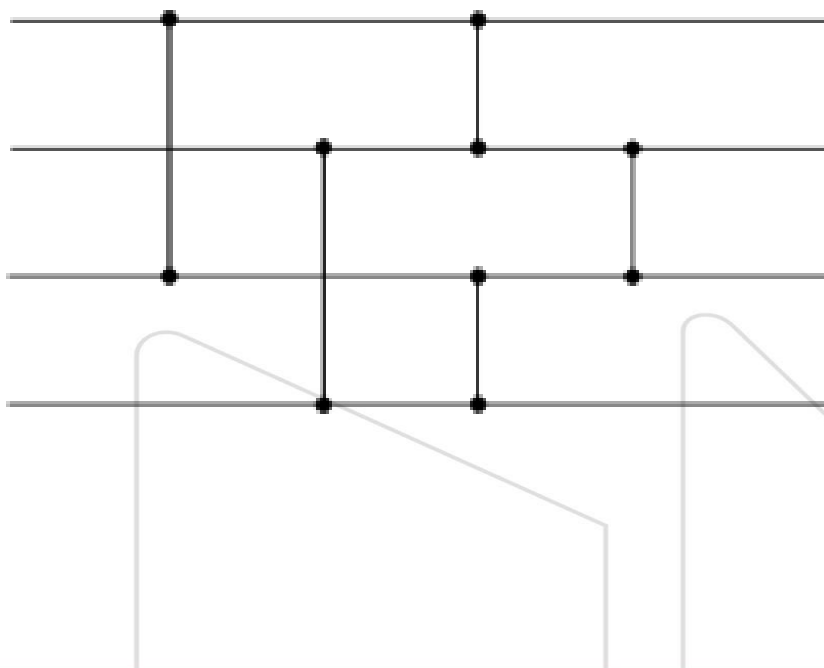
- 每次将一个新的元素插入到之前已排序的序列中
  - 如，对[2, 5, 4, 3, 6, 1]进行插入排序：
    - [2] → [2, 5] → [2, 4, 5] → [2, 3, 4, 5] → [2, 3, 4, 5, 6] → [1, 2, 3, 4, 5, 6]
  - 外层循环不变量：i次循环后，前i个元素为已排序
- 时间复杂度 $O(n^2)$ ，空间复杂度 $O(1)$ ，是否数据无关？

## 冒泡排序

- 每次检查相邻的两个元素是否有序，如无序则交换
  - 如，对[2, 5, 4, 3, 6, 1]进行插入排序（前两次外层循环）：
    - [2, 5, 4, 3, 6, 1] → [2, 4, 5, 3, 6, 1] → [2, 4, 3, 5, 6, 1] → [2, 4, 3, 5, 6, 1] → [2, 4, 3, 5, 1, 6]
    - [2, 4, 3, 5, 1, 6] → [2, 3, 4, 5, 1, 6] → [2, 3, 4, 5, 1, 6] → [2, 3, 4, 1, 5, 6]...
  - 外层循环不变量：i次循环后，后i个元素为已排序
- 时间复杂度 $O(n^2)$ ，空间复杂度 $O(1)$ ，是否数据无关？

## 排序网络

- 由一系列比较器组成的网络，对元素进行比较及交换，从而达到有序状态
  - 具有固定的执行过程
  - 现有数据无关排序算法多可表示为排序网络（如插入排序、冒泡排序）
  - 排序网络举例：

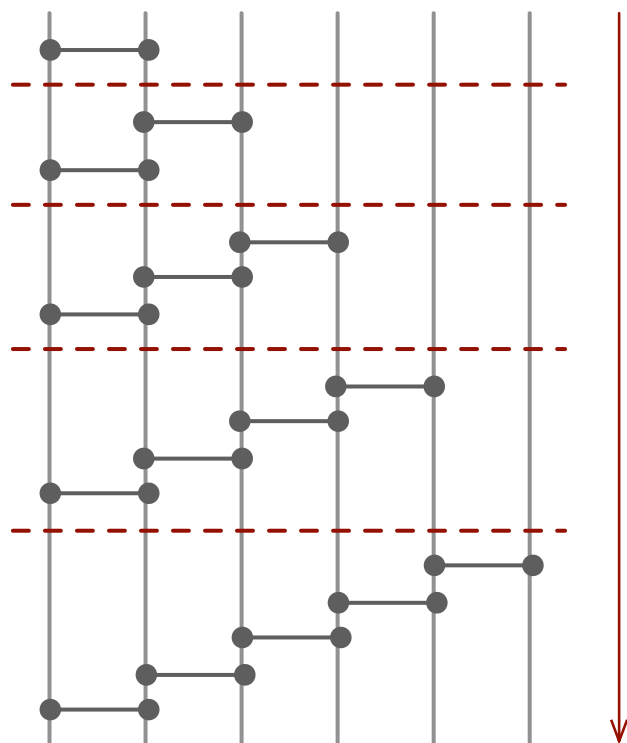




## 插入排序的排序网络

— 每层中进行比较的序列长度加1

[2 5 4 3 6 1]



[1 2 3 4 5 6]

[2 5 4 3 6 1]

[2 4 5 3 6 1] → [2 4 5 3 6 1]

[2 4 3 5 6 1] → [2 3 4 5 6 1] → [2 3 4 5 6 1]

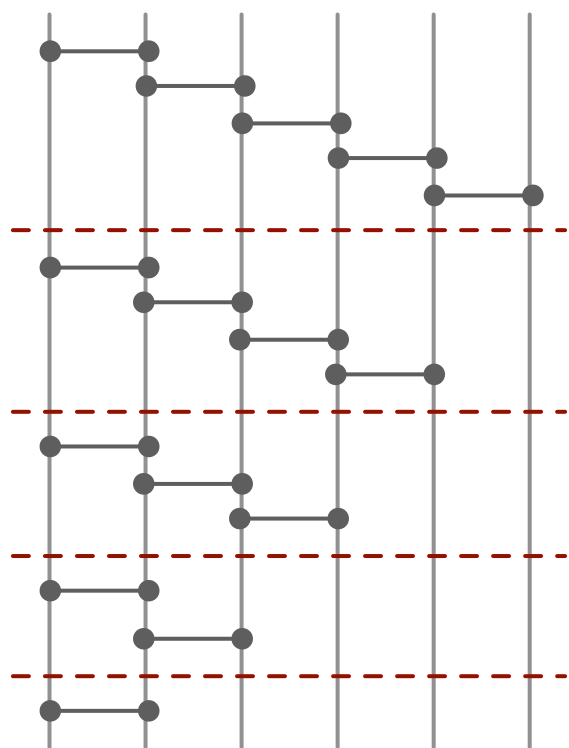
[2 3 4 5 6 1] → [2 3 4 5 6 1] → [2 3 4 5 6 1] → [2 3 4 5 6 1]

[2 3 4 5 1 6] → [2 3 4 1 5 6] → [2 3 1 4 5 6] → [2 1 3 4 5 6] → [1 2 3 4 5 6]

## 冒泡排序的排序网络

— 每层中进行比较的序列长度减1

[2 5 4 3 6 1]



[1 2 3 4 5 6]

[2 5 4 3 6 1] → [2 4 5 3 6 1] → [2 4 3 5 6 1] → [2 4 3 5 6 1] → [2 4 3 5 1 6]

[2 4 3 5 1 6] → [2 3 4 5 1 6] → [2 3 4 5 1 6] → [2 3 4 1 5 6]

[2 3 4 1 5 6] → [2 3 4 1 5 6] → [2 3 1 4 5 6]

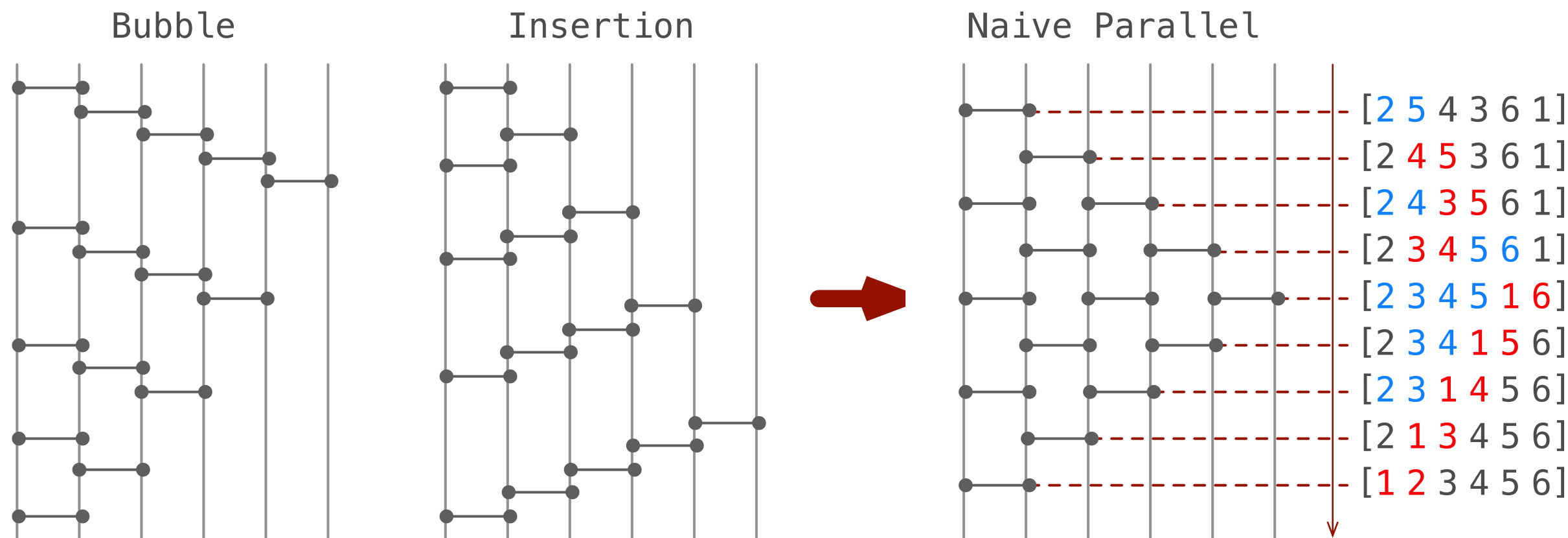
[2 3 1 4 5 6] → [2 1 3 4 5 6]

[1 2 3 4 5 6]

## 并行排序网络

– 插入排序与冒泡排序的网络高度相似！

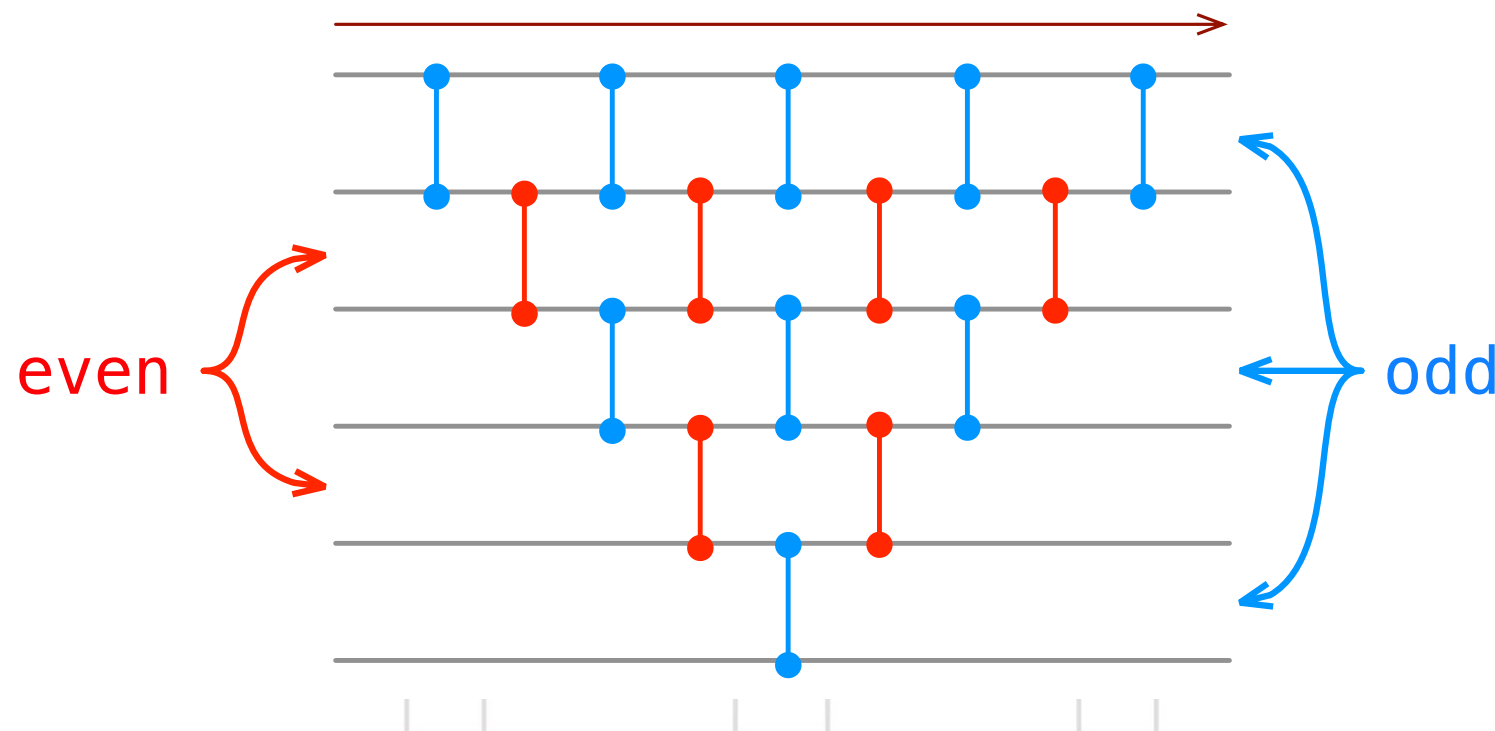
- 在不改变原有执行顺序的情况下，通过同时执行作用于操作可并行
- 总运算量不变： $n(n-1)/2$ 次比较
- 时间复杂度： $O(n^2)$



## 使用排序网络将插入排序与冒泡排序并行化

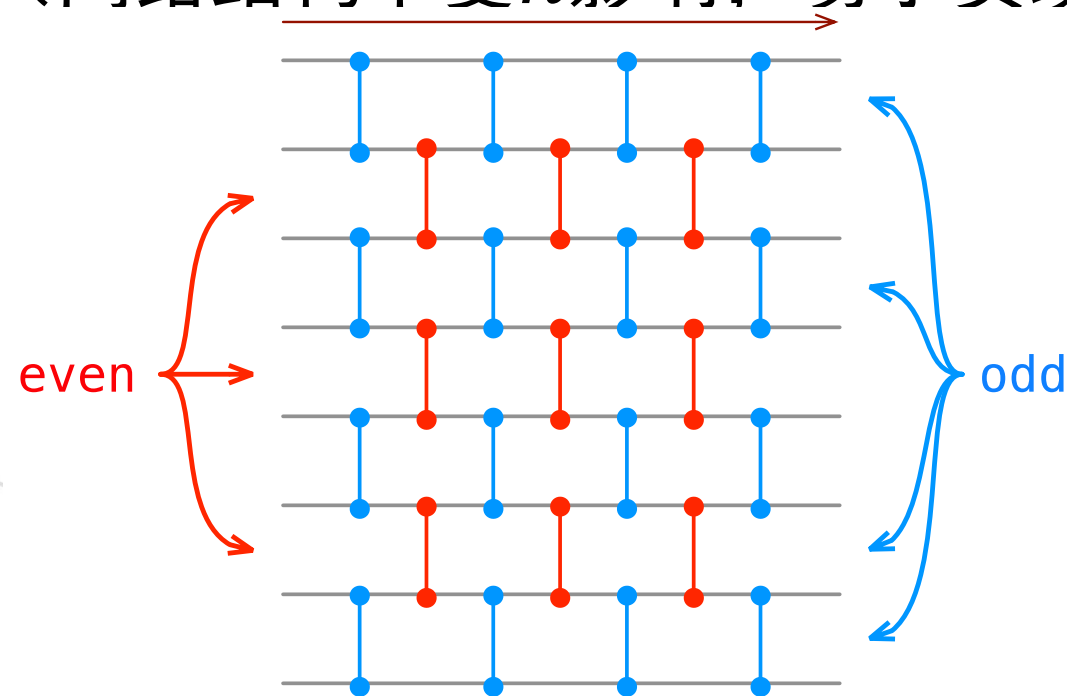
### – 插入排序与冒泡排序的网络高度相似！

- 在不改变原有执行顺序的情况下，通过同时执行作用于操作可并行
- 总运算量不变： $n(n-1)/2$ 次比较
- 时间复杂度： $O(n^2)$
- 共 $2n-3$ 层，其中第 $i$ 层处理 $i/2$ 对元素（全为奇数对，或全为偶数对）
  - 假设运算核心足够多的情况下，其并行所需时间 $2n-3$



## 奇偶移项排序网络 (odd-even transposition sort)

- 由插入排序及冒泡排序改进的排序网络深度为  $2n - 3$  层，在起始及结束阶段并没有充分利用计算资源
- 改进：将三角形网络改为正方形网络
  - 深度从  $2n - 3$  降至  $n - 1$  (为什么  $n - 1$  层已经足够?)
- 优点：空间局部性强 (所有比较都作用于相邻元素上)  
可扩展性强 (网络结构不受  $n$  影响，易于实现)



- 奇偶移项排序网络 (odd-even transposition sort)

- 网络深度为  $n - 1$  (为什么  $n - 1$  层已经足够?)

- 直观地看:  $n$  层已经足够将任意元素移动到序列中的任意位置
  - 从序列一端移动到另一端需要  $n - 1$  次操作
- 从运算量看:  $n - 1$  层中总计进行  $O(n^2)$  次比较运算

### 0-1-principle

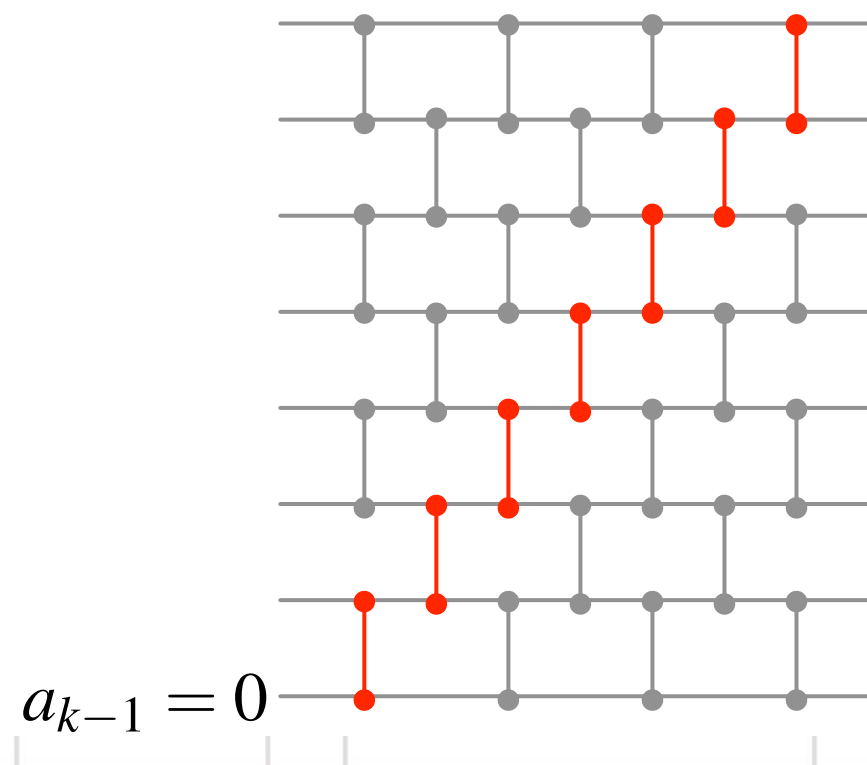
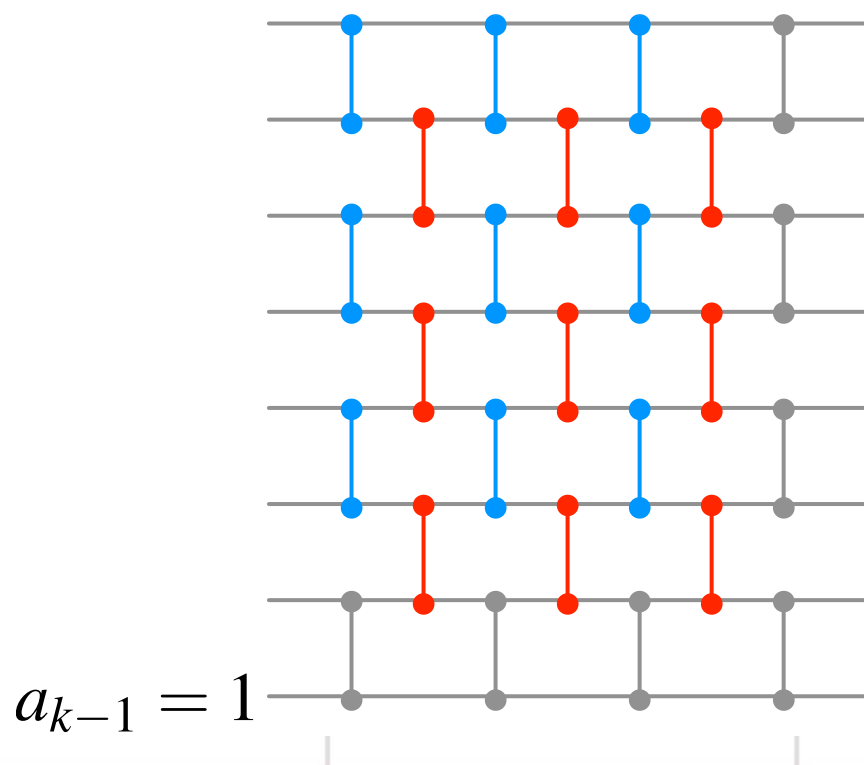
由Donald Ervin Knuth提出: 一个比较网络如果能对长度为  $2^n$  的任意0-1序列进行排序, 则该比较网络为排序网络 (能对任意数字组成的序列进行正确的排序)。



## 奇偶移项排序网络 (odd-even transposition sort)

– 网络深度为  $n - 1$  (为什么  $n - 1$  层已经足够?)

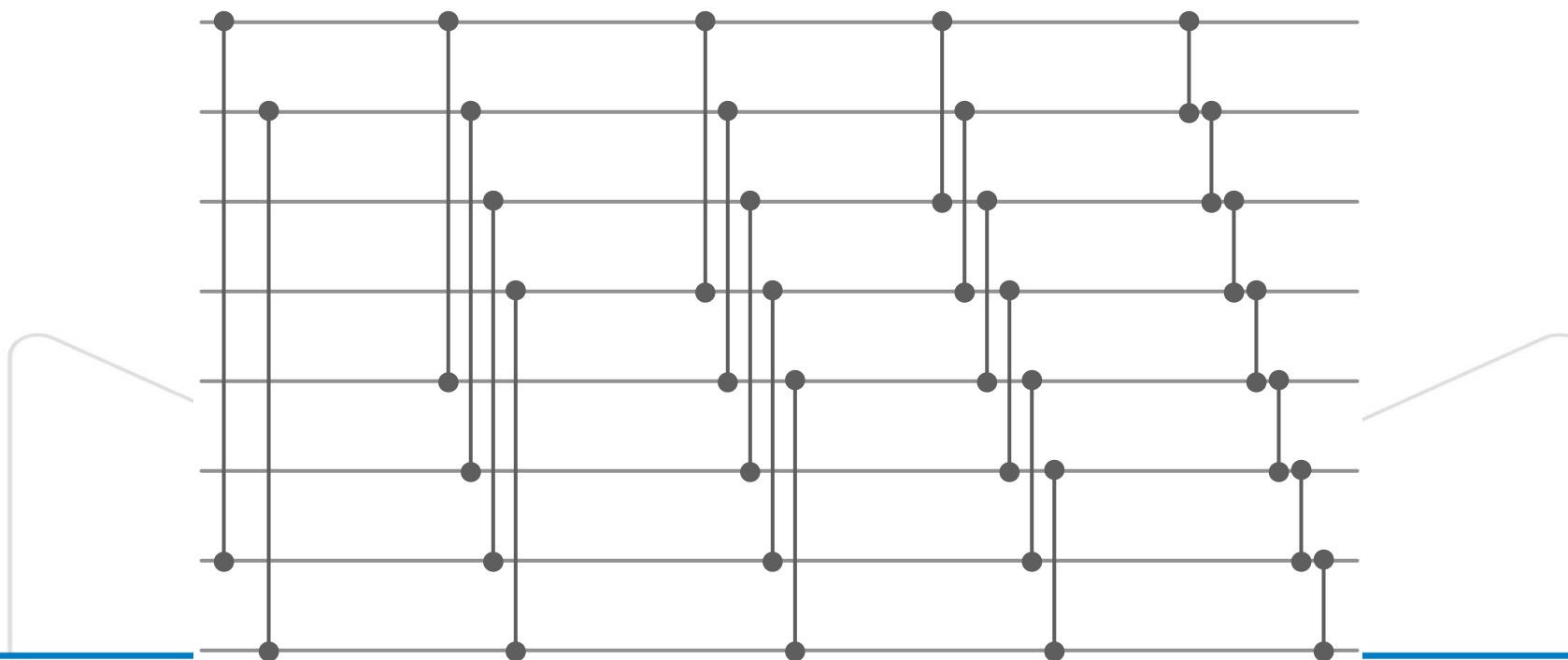
- 使用 0-1-principle 及数学归纳法证明
- 当  $n = 2$  时, 显然  $n - 1 = 1$  层已经足够
- 假设当  $n = k - 1$  时, 深度为  $k - 2$  的网络为排序网络, 由下图可知在添加一层后, 深度为  $k - 1$  的网络为对  $n = k$  数组的排序网络



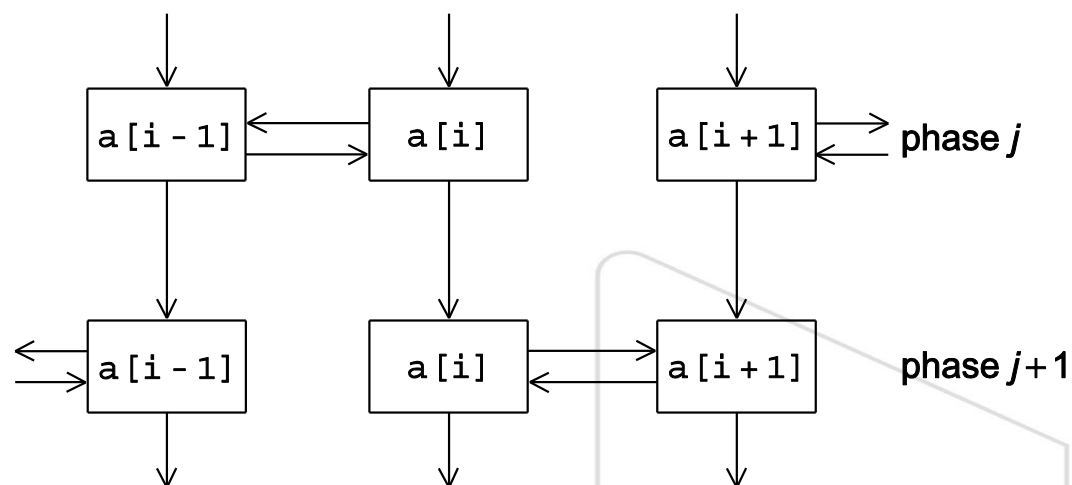


## ◉ 奇偶移项排序网络 (odd-even transposition sort)

- 存在问题：每次只交换相邻元素，对于移动距离较长的元素，需要进行多次交换
  - 必须有  $n - 1$  步才能保证排序完成（从一端至另一端）
- 希尔排序 (shellsort) : 时间复杂度  $O(n \log^2 n)$ 
  - 步长变化的插入排序，可用排序网络实现

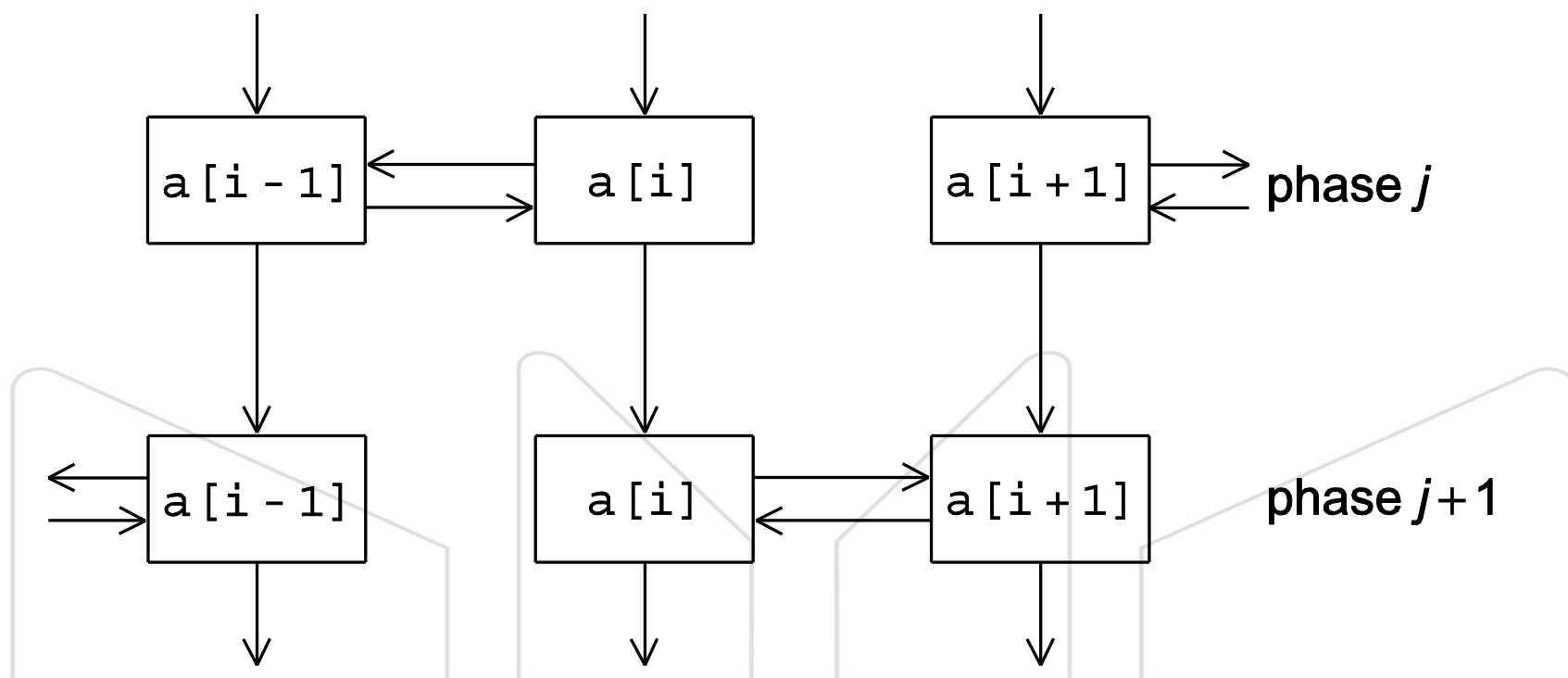


## 奇偶移项排序实现



```
void Odd_even_sort(int a[], int n) {  
    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 */  
}
```

- 奇偶移项排序实现：串行→并行
  - 划分：相邻两个数据交换
  - 通信：数据交换；phase间同步
  - 聚合&映射：每个进程处理一段数据
    - 数据远比进程多
    - 可对段内先进行排序



## 奇偶移项排序的并行实现

- 每个进程先对一段数据排序，此后进程间在数据段之间进行交换
- 注意与串行奇偶移项排序、归并排序的区别

```
sort local keys;  
for (phase = 0; phase < comm_sz; phase++){  
    partner = compute_partner(phase, my_rank);  
    if (not idle){  
        send my keys to partner;  
        receive keys from partner;  
        if (my_rank < partner)  
            keep smaller keys;  
        else  
            keep larger keys;  
    }  
}
```

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

## 奇偶移项排序的并行实现

– 每个进程先对一段数据排序，此后进程间左数据段之间进行交换

– 注意与

```
compute_partner(int phase, int my_rank){
    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;
}
```

```
sort local keys;
for (phase = 0;
    partner = compute_partner(phase, my_rank);
    if (not idle)
        send my data to partner;
        receive data from partner;
        if (my_rank == partner)
            keep data;
        else
            keep data;
}
```

2	3
12, 10	5, 2, 13, 1
10, 12	1, 2, 5, 13
2, 4, 5	6, 10, 12, 13
4, 15, 16	6, 10, 12, 13
, 11, 12	13, 14, 15, 16
, 11, 12	13, 14, 15, 16

## 进程安全性

– 以下实现方式是否能保证程序正确运行？

- 取决于MPI实现中**MPI\_Send**的阻塞行为
- 在调用**MPI\_Send**并将数据拷贝至MPI管理的缓存中就返回则能正确运行
- 需要等待**MPI\_Recv**开始后才返回则程序将悬挂

```
sort local keys;  
for (phase = 0; phase < comm_sz; phase++){  
    partner = compute_partner(phase, my_rank);  
    if (not idle){
```

```
MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &status);
```

```
        if (my_rank < partner)  
            keep smaller keys;  
        else  
            keep larger keys;  
    }  
}
```

注：MPI\_Ssend具有确定的阻塞行为

## 进程安全性

### – 如何确保程序正确运行？

- 方案1：对于参与通信的一对进程，交换其中一个进程的收发函数调用顺序
  - 对于奇偶移项排序能保证正确，但不一定适用于所有算法

```
MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &status);
```



```
if (my_rank % 2 == 0){  
    MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
    MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &status);  
} else {  
    MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &status);  
    MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
}
```



## 进程安全性

– 如何确保程序正确运行？

- 方案2：使用**MPI\_Sendrecv** “同时” 完成数据收发（MPI保证进程安全性）

```
MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &status);
```



```
MPI_Sendrecv(local_A, local_n, MPI_INT, partner, 0,  
             temp_B, local_n, MPI_INT, partner, 0, comm, &status);
```



## 进程安全性

### – 如何确保程序正确运行？

- 方案3：使用无阻塞通信（**I**: immediate, 即时）
  - 需要保证通信完成时可

```
MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &status);
```



```
MPI_Isend(local_A, local_n, MPI_INT, partner, 0, comm, &send_reqs);  
MPI_Irecv(temp_B, local_n, MPI_INT, partner, 0, comm, &recv_reqs);  
MPI_Wait(&send_reqs, &send_status);  
MPI_Wait(&recv_reqs, &recv_status);
```

还可以使用**MPI\_Test**测试通信请求状态，但不等待请求完成  
或使用 **MPI\_Waitall** / **MPI\_Waitany** 等待多个请求

## 完成数据交换后，在本地对接受数据段进行合并

```
void Merge_low(  
    int my_keys[],      /* in/out    */  
    int rcv_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] <= rcv_keys[r_i]) {  
            temp_keys[t_i] = my_keys[m_i];  
            t_i++; m_i++;  
        } else {  
            temp_keys[t_i] = rcv_keys[r_i];  
            t_i++; r_i++;  
        }  
    }  
  
    memcpy(my_keys, temp_keys, local_n*sizeof(int));  
} /* Merge_Low */
```

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

- 并行奇偶移项排序性能
  - 效率? 可扩展性?

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

## • MPI是用于多进程间通信的编程接口

- 支持C, C++, Fortran语言
- MPI程序编译 (mpicc) 及运行 (mpiexec)
- MPI程序通常是单进程多数据的SPMD
- MPI程序通过通信实现进程间的协作
  - 通信部分需放在开始 (MPI\_Init) 和结束 (MPI\_Finalize) 之间
  - 进程的执行顺序具有不确定性

## • MPI通信子 (MPI\_Comm)

- 通信子为参与通信的所有进程
- 通过MPI\_Comm\_size获取通信子中的进程数量
- 通过MPI\_Comm\_rank获取进程在通信子中的编号

## ◉ MPI点对点通信

- 着重理解通信中的阻塞
- MPI\_Send/MPI\_Recv: 阻塞行为根据具体实现可能不同
  - Send可能阻塞, Recv确定阻塞
- MPI\_Isend/MPI\_Irecv: 非阻塞
- MPI\_Sendrecv: 发送和接收都完成前会阻塞进程

## ◉ MPI集合通信

- 并行规约、散射、聚集、广播
- 主要通信结构: 树形、蝶形
  - 理解通信结构与开销间的关系
  - 实际开销依赖于实现

- MPI进程同步
  - MPI\_Barrier
- MPI程序性能
  - 性能分析：时间、加速比、并行效率
  - 使用MPI\_Wtime记录时间
  - 使用barrier同步进程
  - 使用规约统计多个进程的时间
- MPI进程安全性
  - 主要考虑通信操作可能导致的悬挂



# Questions?

