

并行程序设计



佛山科学技术学院 数学与大数据学院 许红龙

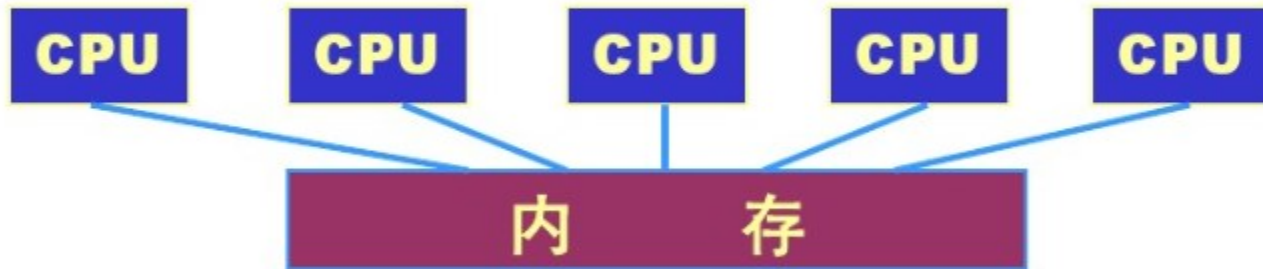
longer597@163.com

第3章 用MPI进行分布式内存编程

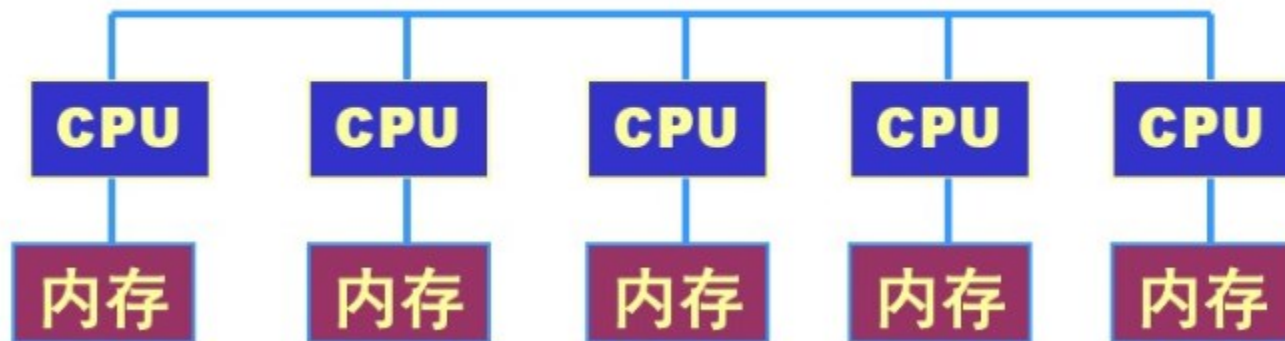
- 3.1 预备知识
- 3.2 用MPI来实现梯形积分法
- 3.3 I/O处理
- 3.4 集合通信
- 3.5 MPI的派生数据类型
- 3.6 MPI程序的性能评估
- 3.7 并行排序算法

3.1 预备知识

- 共享存储



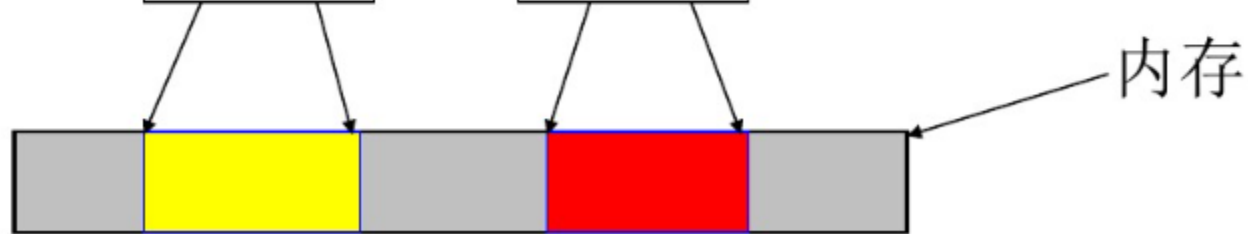
- 分布式存储



3.1 预备知识

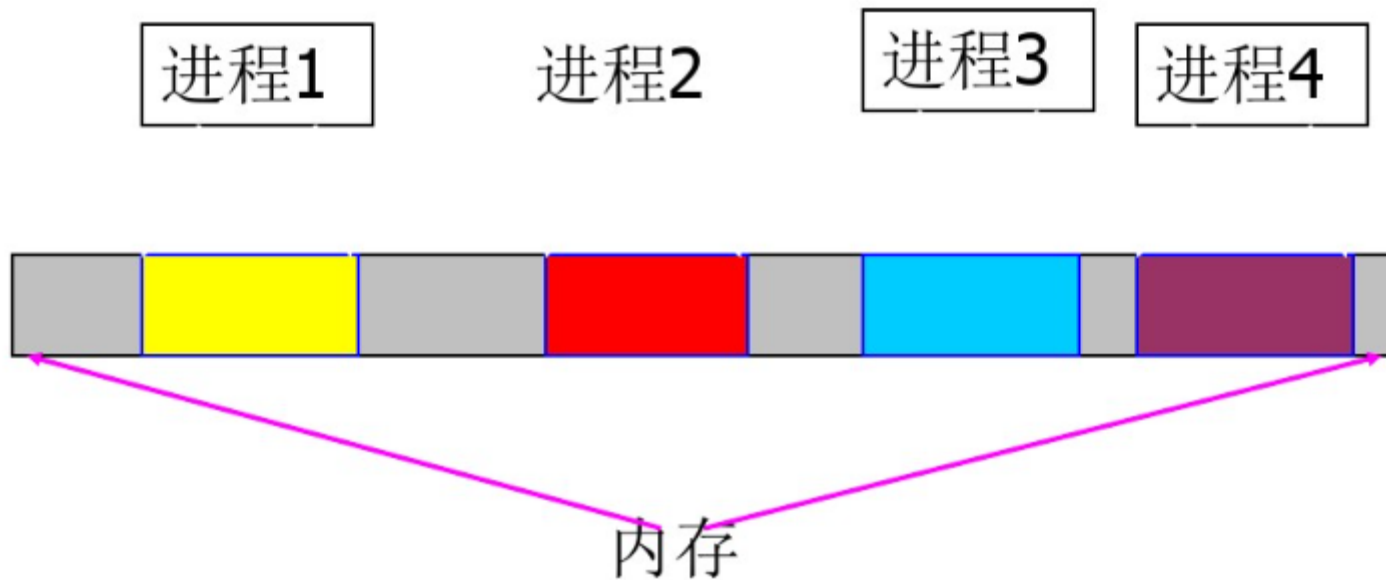
多个进程可以同时存在于单机内同一操作系统：由操作系统负责调度分时共享处理机资源（**CPU**、内存、存储、外设等）。

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，例如多个串行程序在计算机中运行。



单机内的多个进程

3.1 预备知识



并行程序的单机运行方式

什么是MPI?

- **Message Passing Interface**:是消息传递函数库的标准规范，由**MPI**论坛开发，支持**Fortran**和**C**。
 - 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++

常见的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

www.mpich.org/downloads/

Hello World!

```
#include <stdio.h>

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

    return 0;
}
```



认识MPI进程

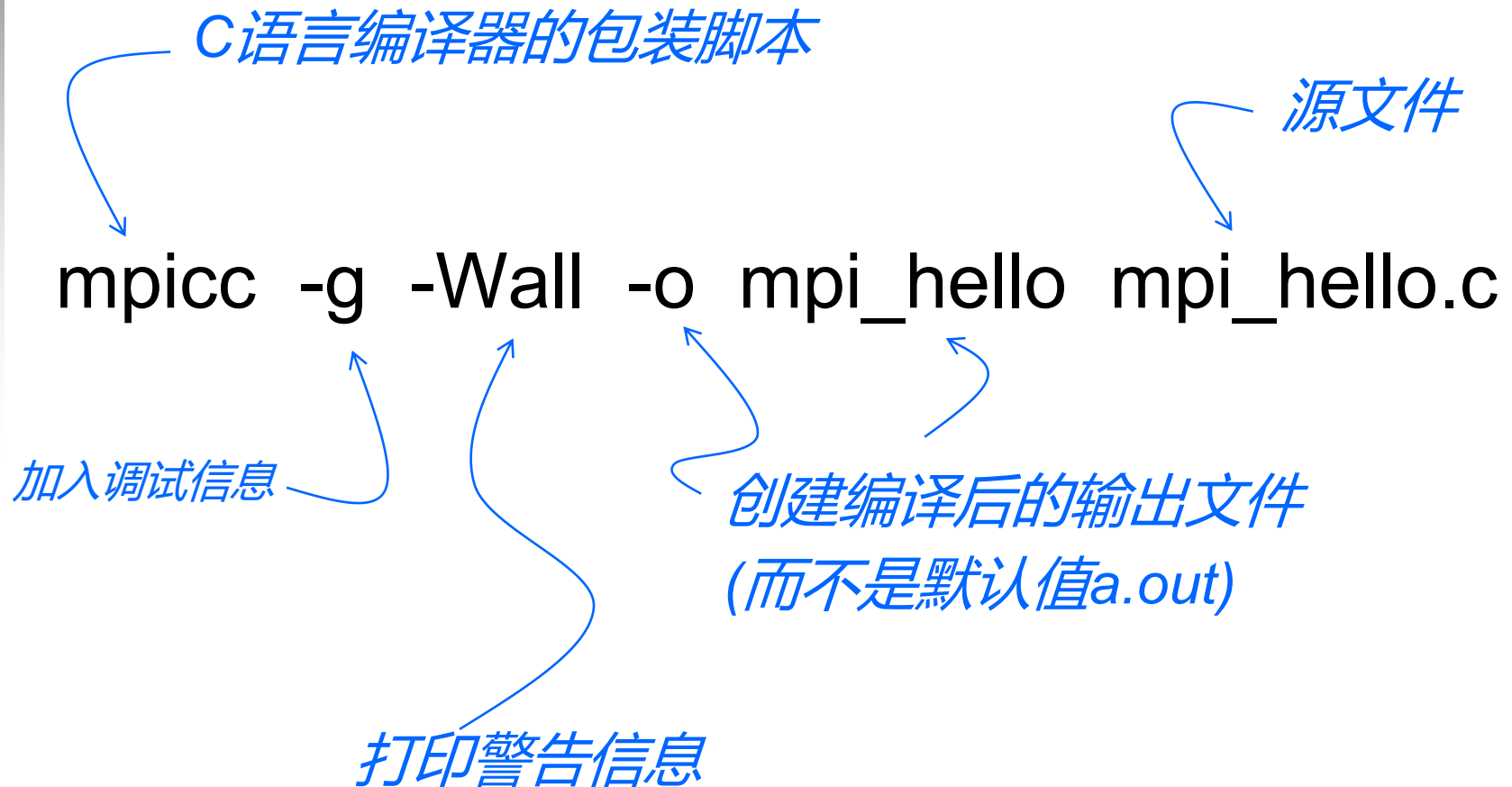
- 在并行编程中，常见的是将进程按照非负整数来标注。
- P 个进程，编号分别为 $0, 1, 2, \dots, p-1$



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

第一个 MPI程序

编译



执行

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

`mpiexec -n 1 ./mpi_hello`

 用1个进程运行

`mpiexec -n 4 ./mpi_hello`

 用4个进程运行

执行

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

Greetings from process 0 of 1 !

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

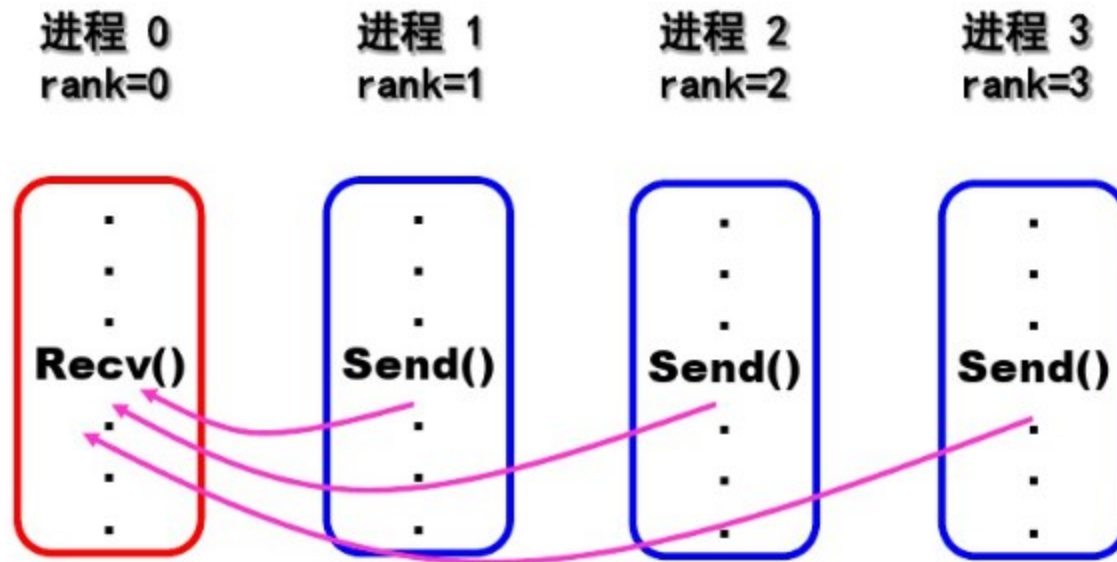
Greetings from process 0 of 4 !

Greetings from process 1 of 4 !

Greetings from process 2 of 4 !

Greetings from process 3 of 4 !

执行过程



3.1.2 MPI程序

- C语言
 - 包含了main函数
 - 标准头文件 `stdio.h`, `string.h`等
- 包含**mpi.h**头文件
- 所有MPI定义的标识符以 "MPI_" 开头
- 下划线后的第一个字母大写
 - 表示函数名和 MPI定义的类型
 - 避免混淆

MPI组件

■ MPI_Init

- 进入MPI环境并完成所有的初始化工作。

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

■ MPI_Finalize

- 从MPI环境中退出。

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

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;  
}
```

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，由用户启动的所有进程组成组成的通信子。
- 称为**`MPI_COMM_WORLD`**.

通信子



用MPI_Comm_size获得进程个数

```
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

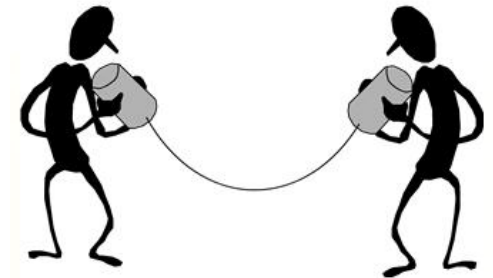
```
int MPI_Comm_rank(  
    MPI_Comm comm          /* in */,  
    int* my_rank_p        /* out */);
```

3.1.5 SPMD

- 单程序多数据流
- 我们编译一个程序。
- 进程0执行一些不同的操作。
 - 接收消息并打印它们, 而其他进程执行工作。
- **if-else**语句构建SPMD。

3.1.7 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 */);
```

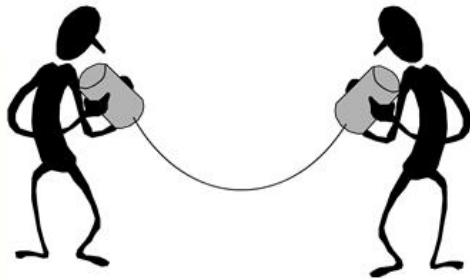


数据类型

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	

3.1.8 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 */);
```



3.1.9 消息匹配

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

MPI_Send

src = q



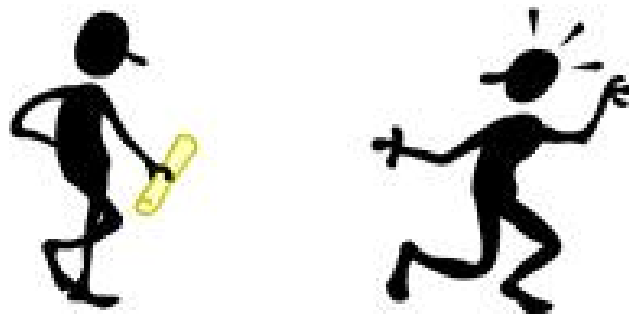
MPI_Recv

dest = r

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


3.1.9 status_p参数

- 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。



3.1.9 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

收到了多少数据？

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



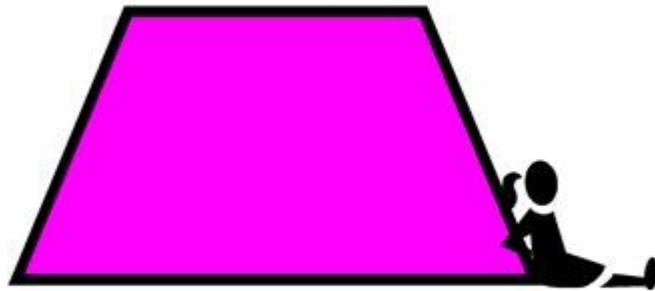
`MPI_Get_count(&status, recv_type, &count);`
count参数：接收到的元素数量；
需要**多一次**计算。

当使用**`MPI_ANY_SOURCE`**或/和**`MPI_ANY_TAG`**接收消息时使用。

3.1.11 MPI_Send和MPI_Recv的问题

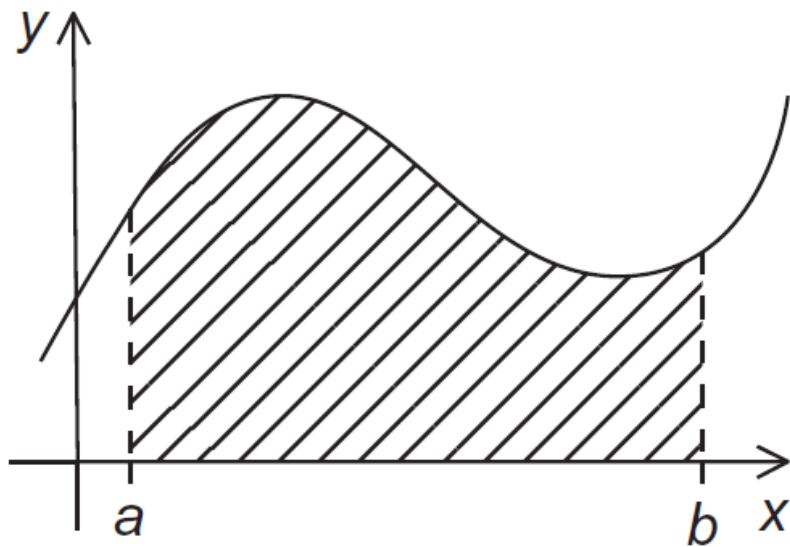
- MPI_Send的精确行为由 MPI实现决定。
- MPI_Send可能有大小不同的缓冲区，是缓冲还是阻塞可由一个消息“截止”大小决定。
- MPI_Recv 总是被阻塞的，直到收到一条匹配的消息。



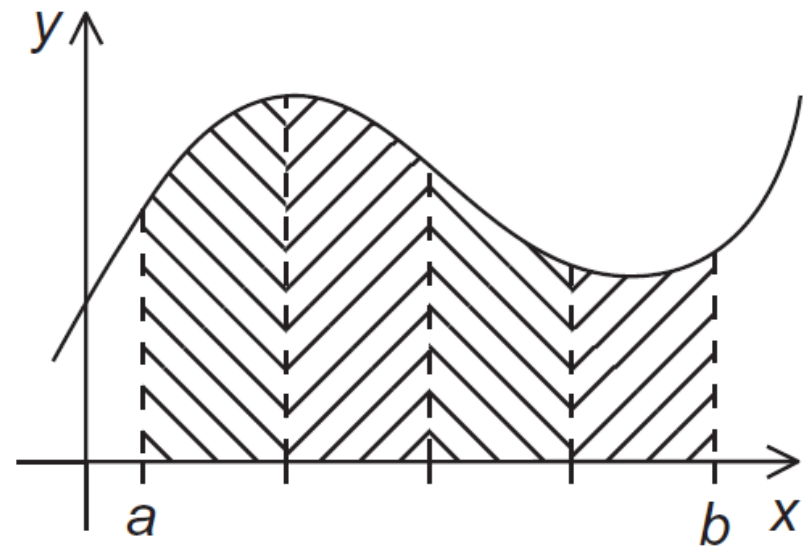


3.2 用MPI实现梯形积分法

3.2.1 梯形积分法

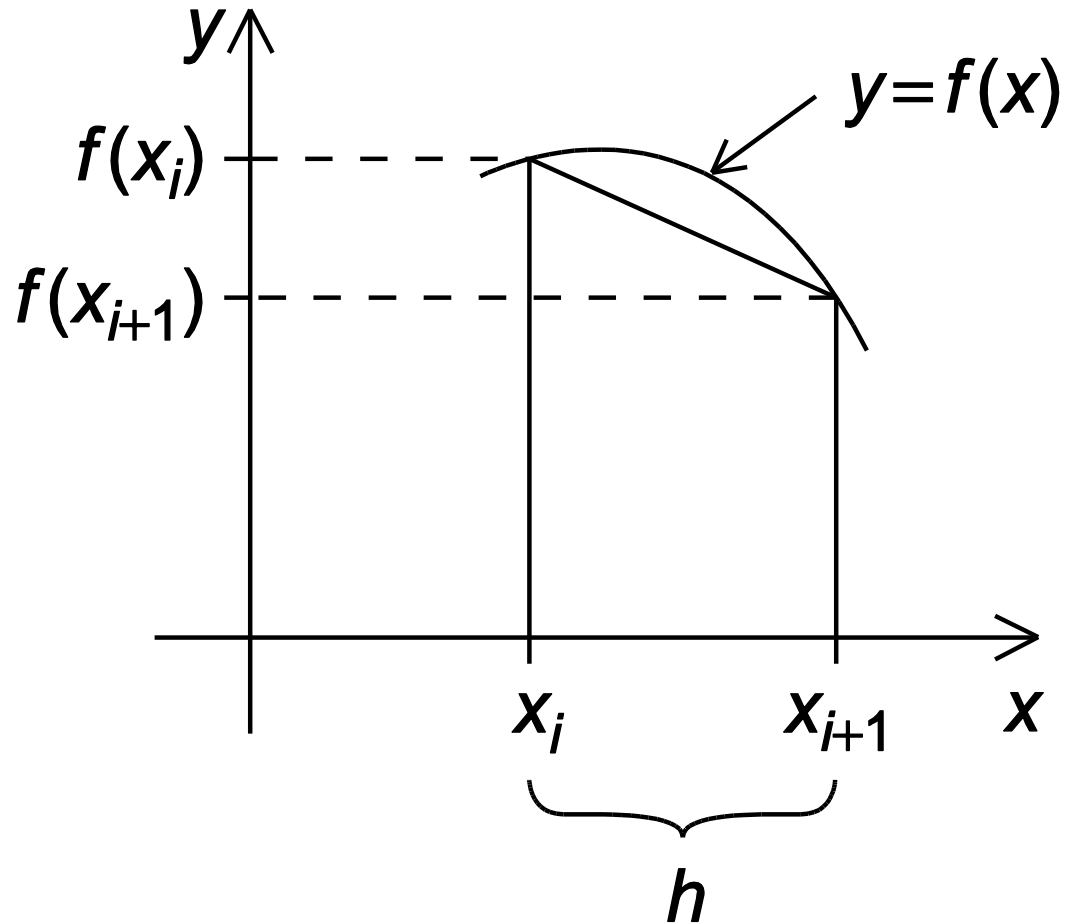


(a)



(b)

梯形



梯形积分法

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

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

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

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

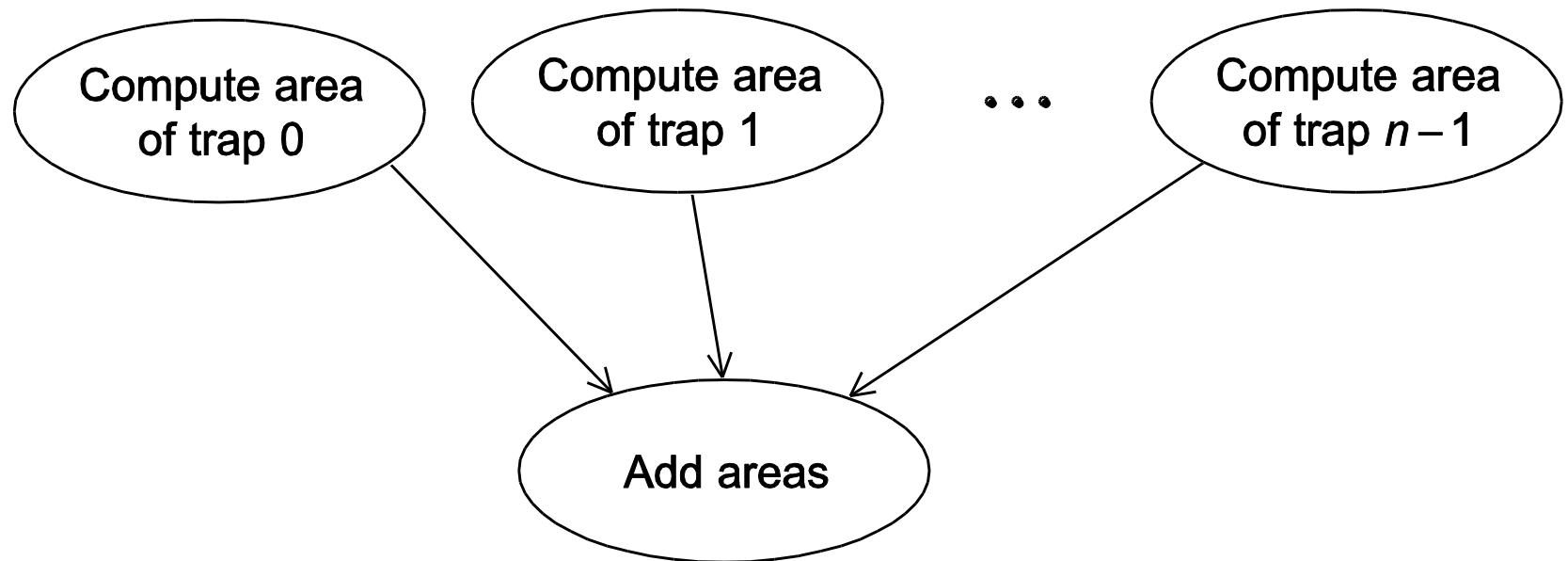
串程序的伪代码

```
/* 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;
```

梯形积分法的并行化

1. 将问题的解决方案划分成多个任务。
2. 确定任务之间的通信信道。
3. 将任务聚合成复合任务。
4. 在核上分配复合任务。

梯形积分法的任务和通信



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;
```

MPI并行实现(1)

```
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);
```

MPI并行实现(2)

```
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  */
```


MPI并行实现(3)

```
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 */
```

I/O处理

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

每个进程只打印一条消息。

```
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);

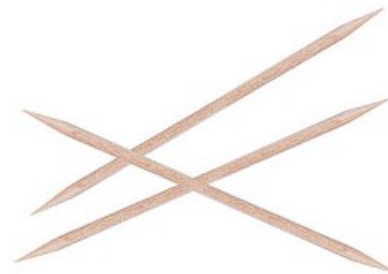
    printf("Proc %d of %d > Does anyone have a toothpick?\n",
           my_rank, comm_sz);

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

运行6个进程

```
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?
```

无法预测输出顺序



输入

- 大多数 MPI 实现只允许 MPI_COMM_WORLD 中的进程0访问stdin.
- 进程0负责读取数据 (scanf) 并发送到其他进程。

```
. . .  
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;  
. . .
```

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

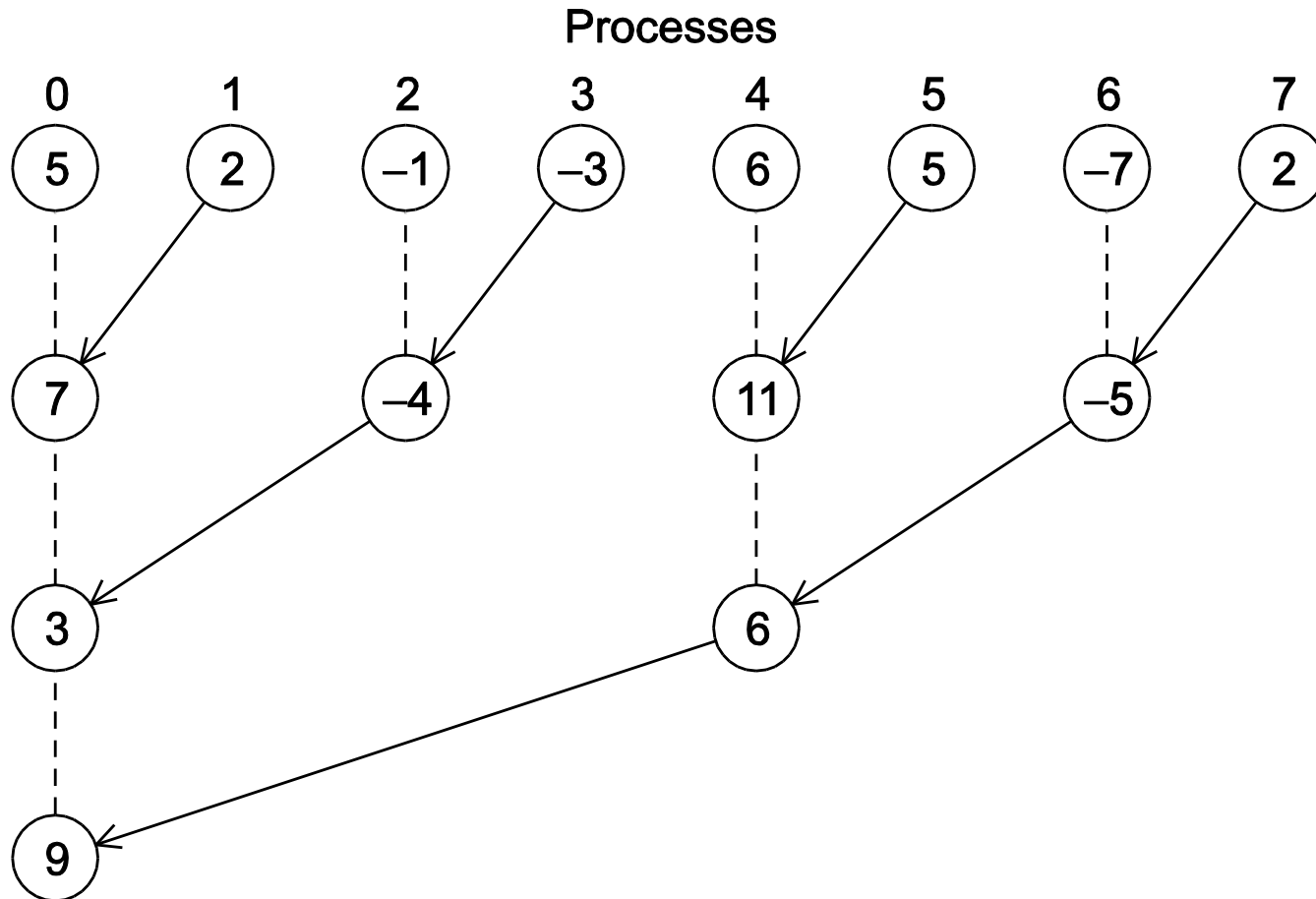


3.4.1 树形结构通信

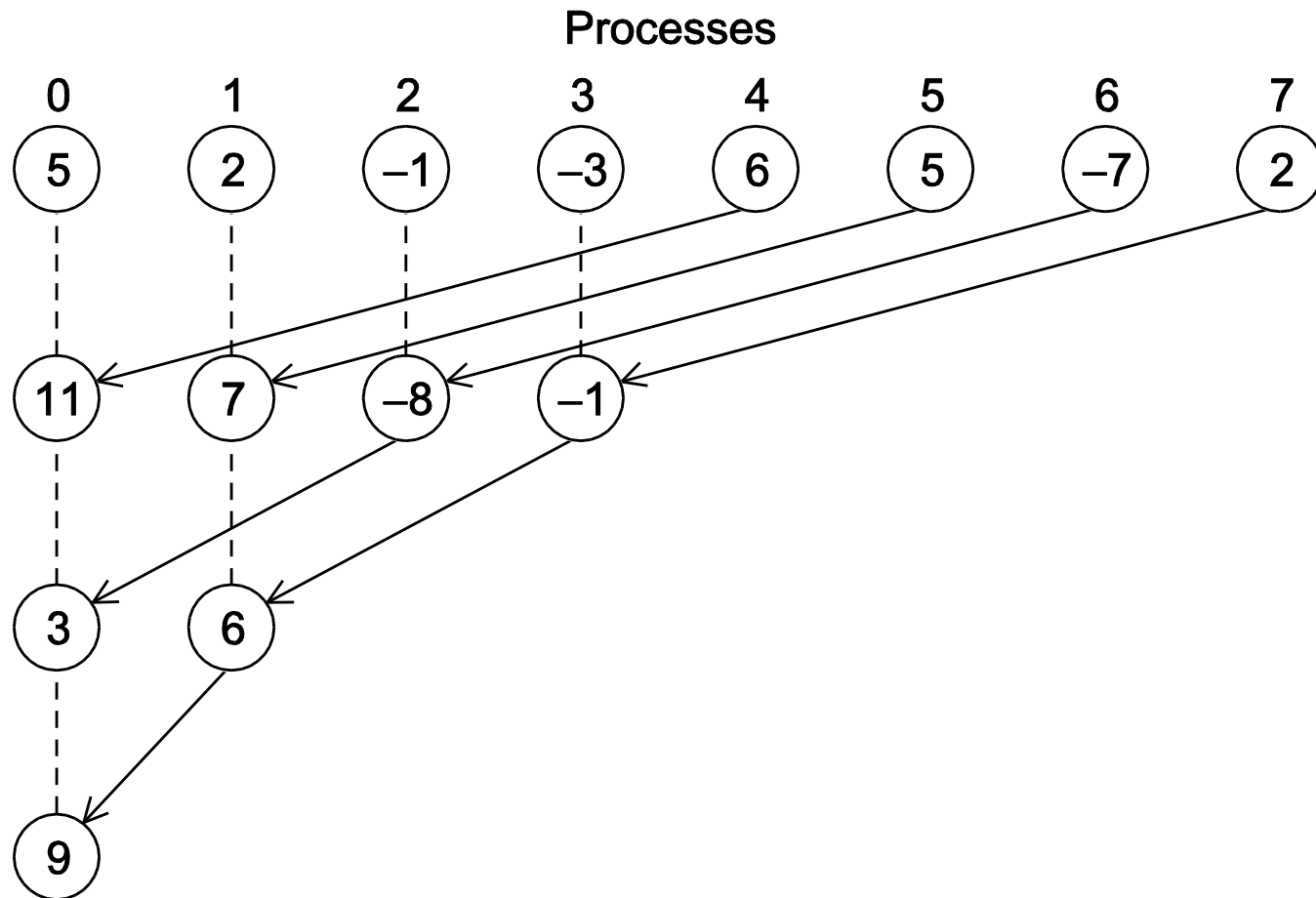
1. 在第一阶段:
 - (a) 进程1、3、5、7分别将其值发送到进程0、2、4、6。
 - (b) 进程0、2、4和6将接收到的值加到自己原有的值上。
 - (c) 进程2和6分别将其新值发送到进程0和4。
 - (d) 进程0和4将接收到的值加到新值上。

2.
 - (a) 进程4将其最新值发送到进程0。
 - (b) 进程0将接收到的值加到其最新值上。

树形结构的全局求和



另一种树形结构全局求和方法



3.4.2 MPI_Reduce

```
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

- 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**调用与另一个进程中的**MPI_Recv**调用相匹配的程序会出错, 此时程序将挂起或崩溃。

3.4.3 集合通信与点对点通信

- 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**调用所产生的结果是错误的, 程序很可能会挂起或崩溃。

3.4.3 集合通信与点对点通信

- 参数 `output_data_p` 仅用在 `dest_process` 上.
- 然而, 所有进程仍需要传递一个与 `output_data_p` 对应的实际参数, 即使它只是 `NULL`.

3.4.3 集合通信与点对点通信

- 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。
- 集合通信函数只通过通信子和被调用的顺序来匹配。

示例 (1)

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(&a, &b, ...)</code>	<code>MPI_Reduce(&c, &d, ...)</code>	<code>MPI_Reduce(&a, &b, ...)</code>
2	<code>MPI_Reduce(&c, &d, ...)</code>	<code>MPI_Reduce(&a, &b, ...)</code>	<code>MPI_Reduce(&c, &d, ...)</code>

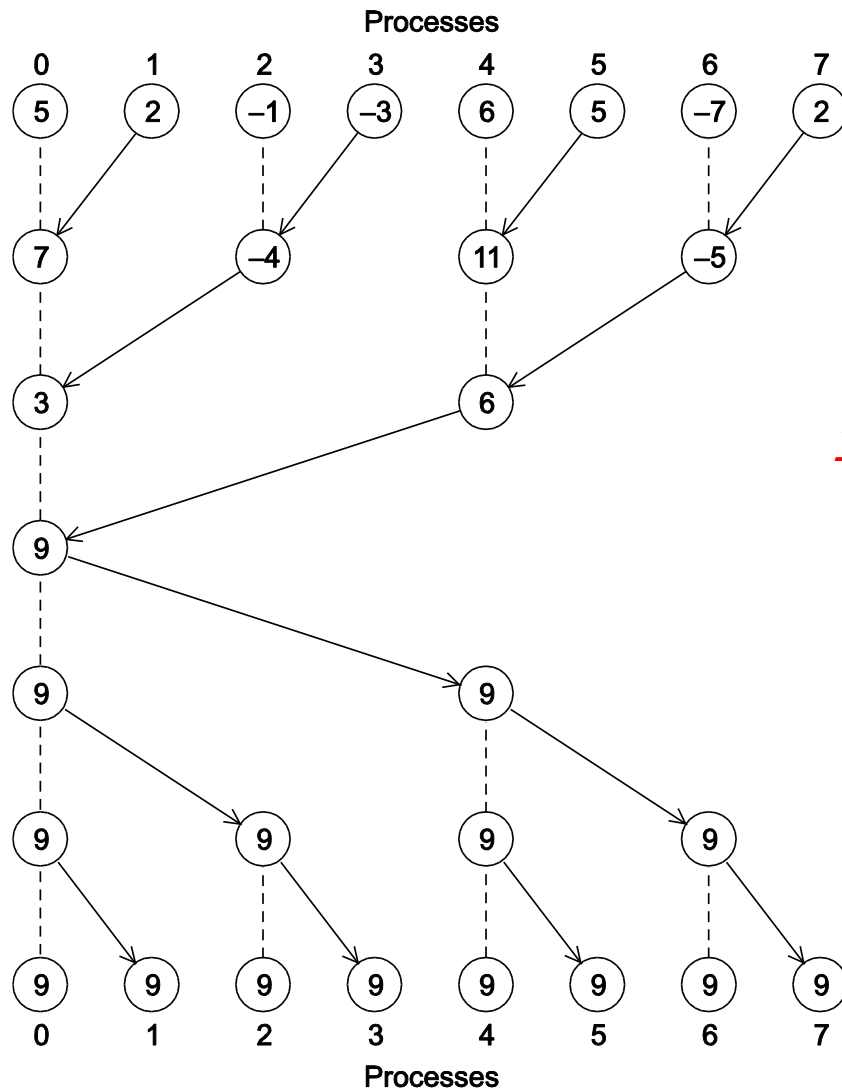
对MPI_Reduce的多个调用

- 内存单元的名称与MPI_Reduce的调用匹配无关。
- 函数调用的顺序决定匹配方式, 因此:
b中的值将为 $1 + 2 + 1 = 4$,
d中的值将为 $2 + 1 + 2 = 5$ 。

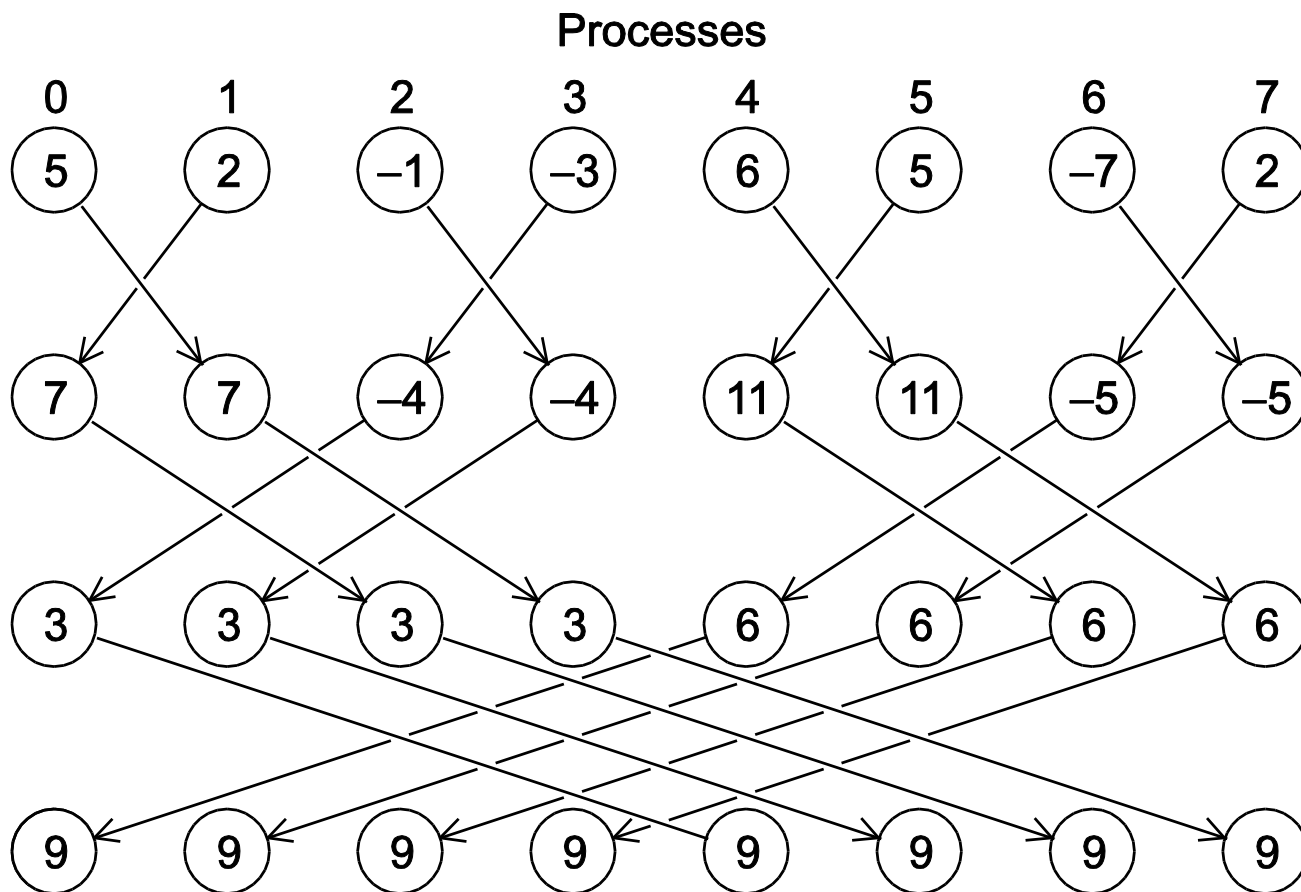
3.4.4 MPI_Allreduce

- 有时所有进程都需要全局总和的结果，才能完成一些更大规模的计算。

```
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



蝶形结构的全局求和计算。

3.4.5 广播

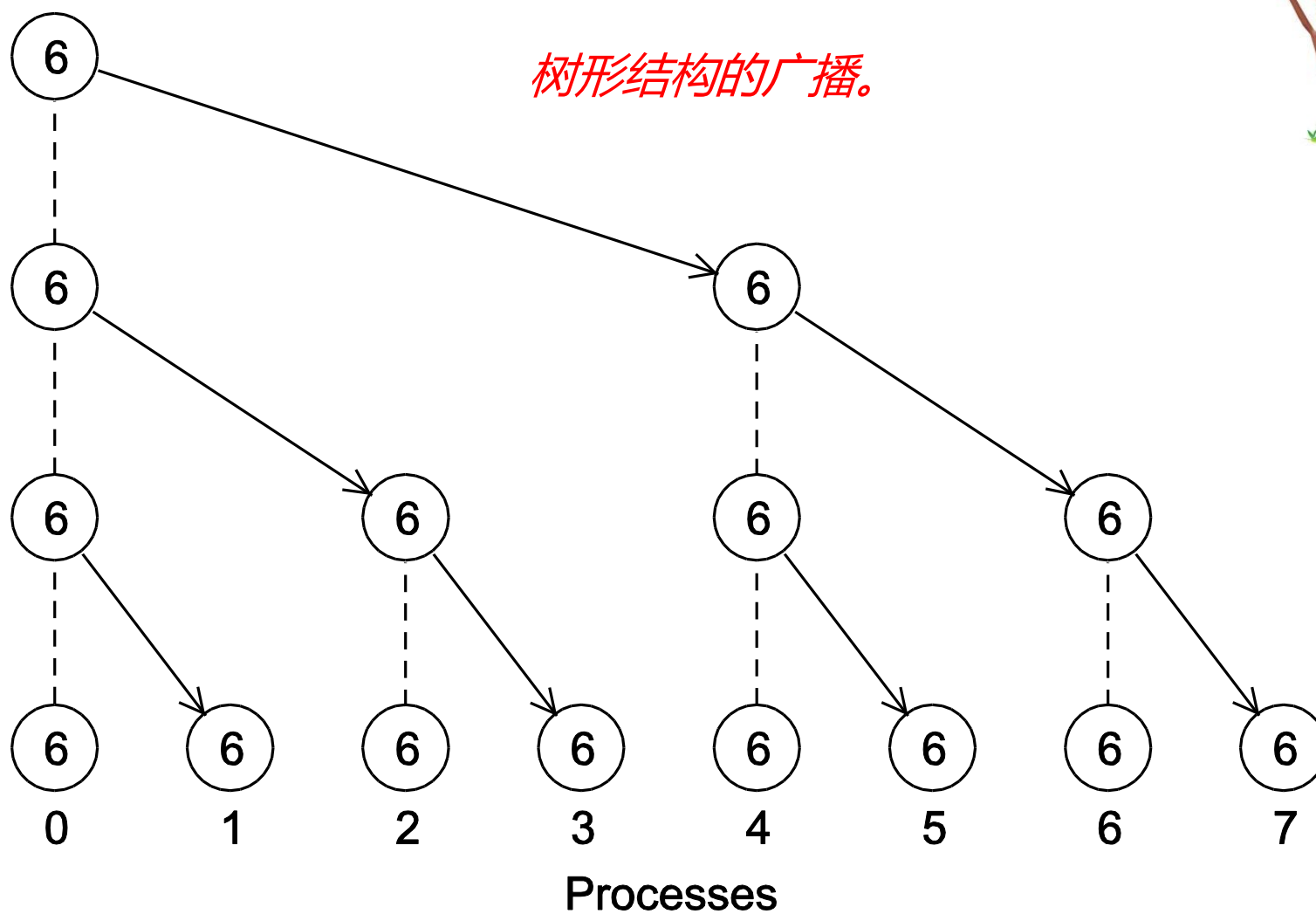
- 在一个集合通信中，属于一个进程的数据被发送到通信子中的所有进程。

```
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



树形结构的广播。



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

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

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 */
```


3个进程中12个分量向量的不同划分

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
 - 为每个进程分配连续 N 个分量所构成的块。
- 循环划分
 - 采用轮转的方式分配向量分量。
- 块-循环划分
 - 使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号进程读取整个向量, 但只将所需的分量发送到其他每个进程。

```
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 */);
```

读取和分发向量的函数

```
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

- 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 */);
```

打印分布式向量 (1)

```
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;
```

打印分布式向量 (2)

```
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

- 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。
- recv_count指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

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

y 的第 i 个分量

A 的第 i 行与 \mathbf{x} 的点积。

矩阵-向量乘法

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}

矩阵-向量乘法的串行伪代码

```
/* 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语言的数组

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

存储为

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 (1)

```
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;
```

MPI矩阵-向量乘法函数 (2)

```
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 */
```




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。
- 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**发送之前**在内存中将数据项聚集起来。
- 同样，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。

派生数据类型

- 由一系列的MPI基本数据类型以及每个数据类型的偏移组成。
- 梯形积分法示例:

Variable	Address
a	24
b	40
n	48

$\{(MPI_DOUBLE, 0), (MPI_DOUBLE, 16), (MPI_INT, 24)\}$

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

- 返回指向的内存单元地址`location_p`。
- 特殊类型`MPI_Aint`是一个整数型, 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

- 用于存储元素的MPI数据类型。
- 允许MPI实现优化数据类型的内部表示，以在通信函数内使用这一数据类型。

```
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** 函数释放额外的存储空间。

```
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 */
```

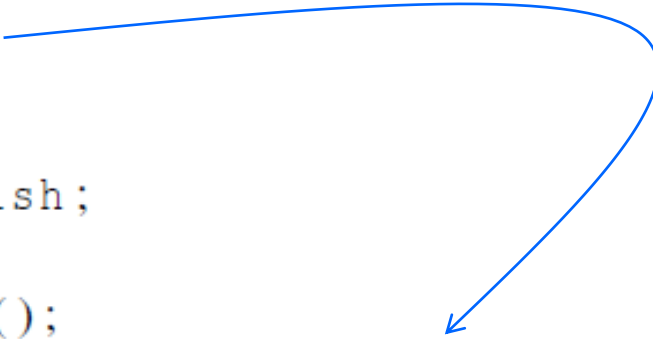


3.6 MPI程序的性能评估

并行代码的运行时间

- **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);
```



串行代码的运行时间

- 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 (以毫秒为单位)。

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



串行代码的运行时间

```
#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

- 确保同一个通信子中的所有进程都完成调用该函数之前, 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 串行和并行的运行时间

comm_sz	Order of Matrix (秒)				
	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

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

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规模的前提下维持恒定的效率，那么程序称为**强可扩展性**。
- 当问题的规模增加，通过增大进程/线程个数来维持恒定效率的程序称为**弱可扩展性**。

3.7 并行排序算法

排序

- n 个键值, $p = \text{comm_sz}$ 个进程.
- n/p 个键值分配给每个进程。
- 不限制哪些键值分配给哪些进程。
- 当算法终止时:
 - 每个进程上的键值应以升序的方式存储。
 - 如果 $0 \leq q < r < p$, 则分配给进程 q 的每个键值应小于等于分配给进程 r 的每一个键值。

3.7.1 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

- 由一系列阶段组成。
- 在偶数阶段，比较-交换由以下数对执行：

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

- 在奇数阶段，比较-交换由以下数对执行：

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

例子

开始: 5, 9, 4, 3

偶数阶段: 比较-交换 (5, 9) 和 (4, 3)

得到: 5, 9, 3, 4

奇数阶段: 比较-交换 (9, 3)

得到: 5, 3, 9, 4

偶数阶段: 比较-交换 (5, 3) 和 (9, 4)

得到: 3, 5, 4, 9

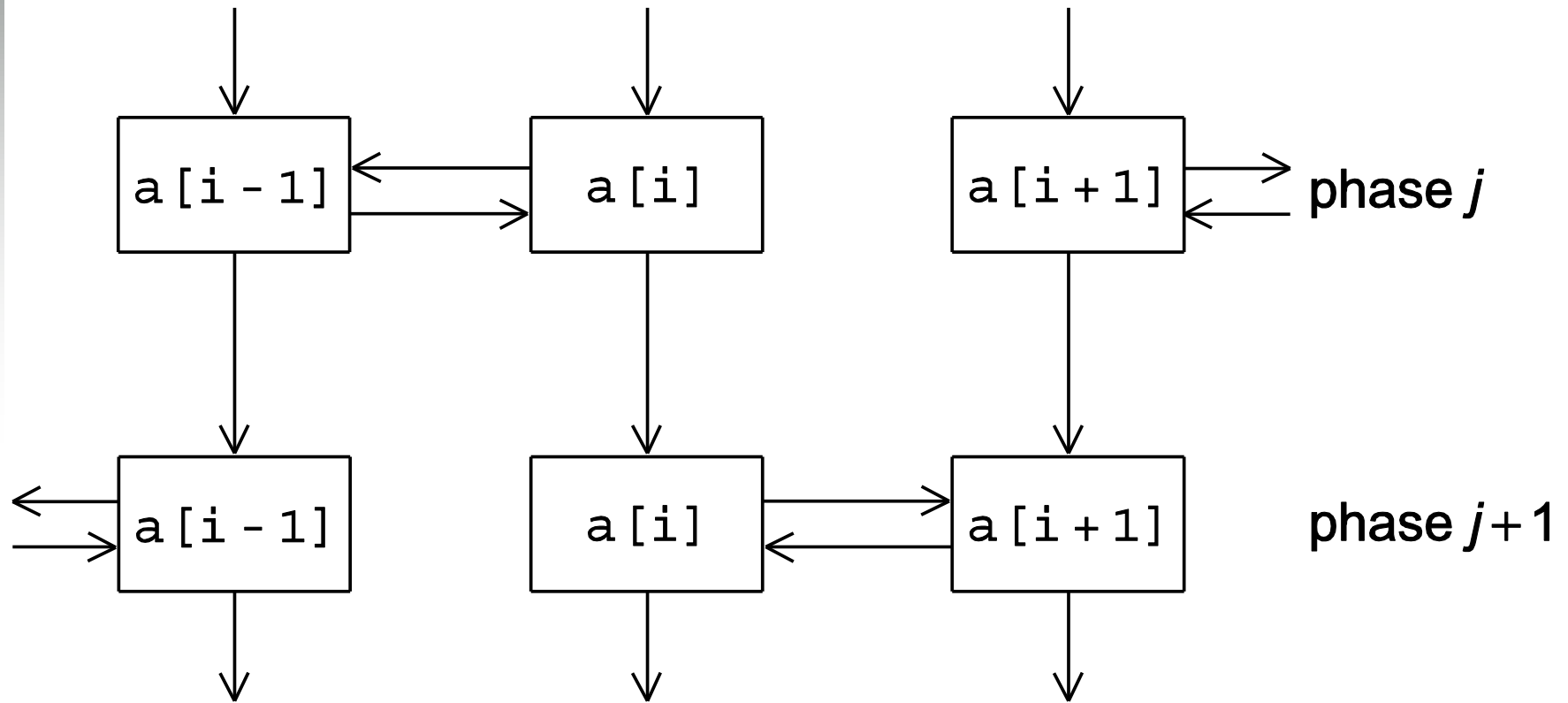
奇数阶段: 比较-交换 (5, 4)

得到: 3, 4, 5, 9

串行奇偶变换排序

```
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

并行奇偶变换排序

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以两种不同的方式实现：
 - 简单地将消息复制到 MPI设置的缓冲区并返回,
 - 或直到对应的MPI_Recv出现前都阻塞。
- 许多MPI函数的实现都设置了系统从缓冲切换到阻塞的阈值。
 - 相对较小的消息交由MPI_Send缓冲。
 - 对于大型数据, 选择阻塞模式。

MPI程序的安全性

- 如果每个进程都阻塞在MPI_Send上, 就没有进程会去调用MPI_Recv, 此时程序将挂起或死锁.
 - 每个进程都在等待永远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挂起或崩溃。

MPI_Ssend

- MPI标准提供的一个函数来替代MPI_Send。
- 额外的字母“s”代表同步, MPI_Ssend 保证在匹配的接收开始之前, 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);  
}
```

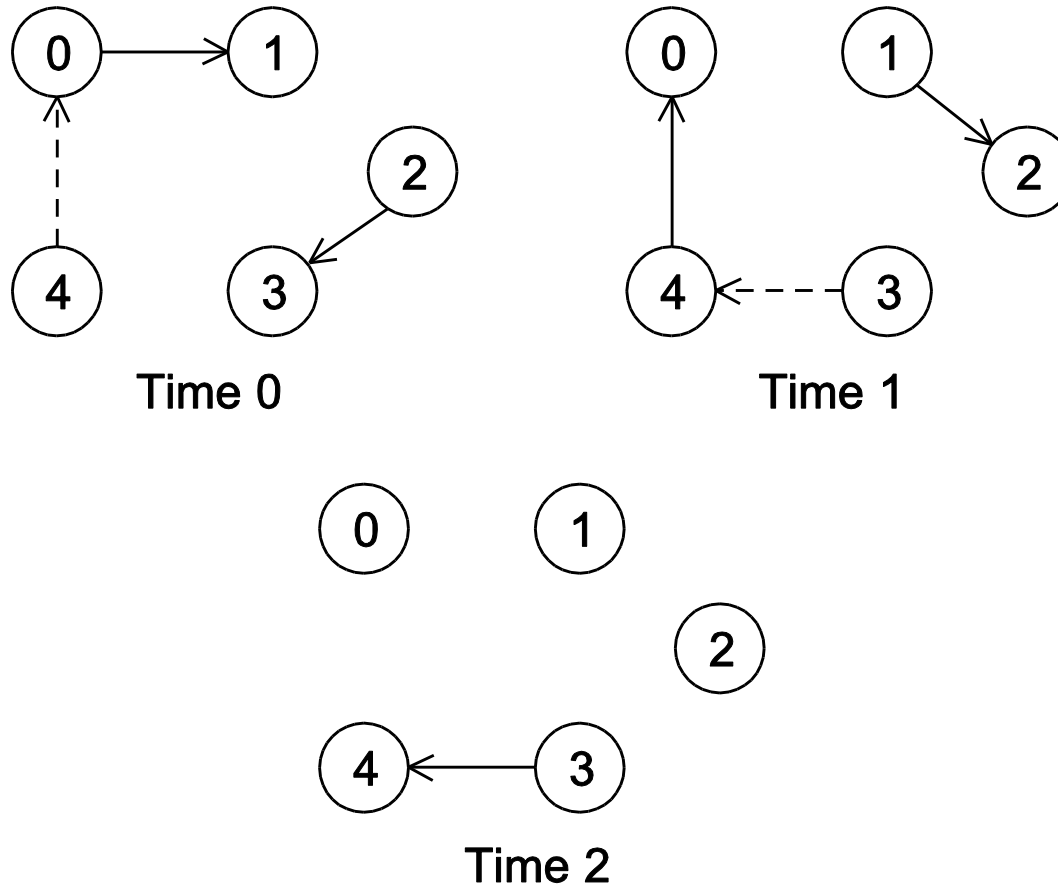
MPI_Sendrecv

- MPI提供的自己调度的方法。
- 分别执行一次阻塞式消息发送和一次消息接收。
- **dest**和**source**参数可相同,也可不同。
- 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



并行奇偶变换排序算法中的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 */
```

并行奇偶交换排序的运行时间

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

(单位: 毫秒)

总结

- **消息传递接口（MPI）**：是一个,可以被C、C++或 Fortran程序调用的函数库。
- **通信子**：一组进程的集合，该集合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。

总结

- 大多数串行程序是**确定性的**: 如果我们使用**相同的输入**运行**相同的程序**, 我们将获得**相同的输出**。
- 大多数并行程序是**非确定性的**。
- **集合通信**: 涉及通信子中的所有进程。

总结

- **墙上时钟时间**：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。

总结

- **可扩展的**: 如果增加问题规模 (n), 随着 p 的增加, 效率没有递减, 则称并行程序是可扩展的。
- **不安全的**: `MPI_Send` 既可以阻塞也可以缓冲输入。