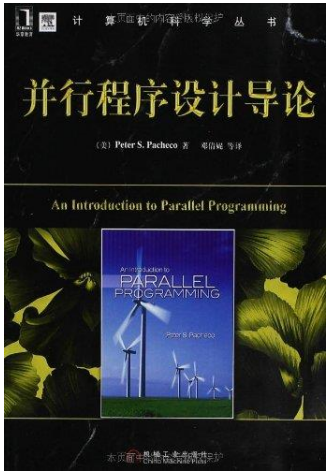


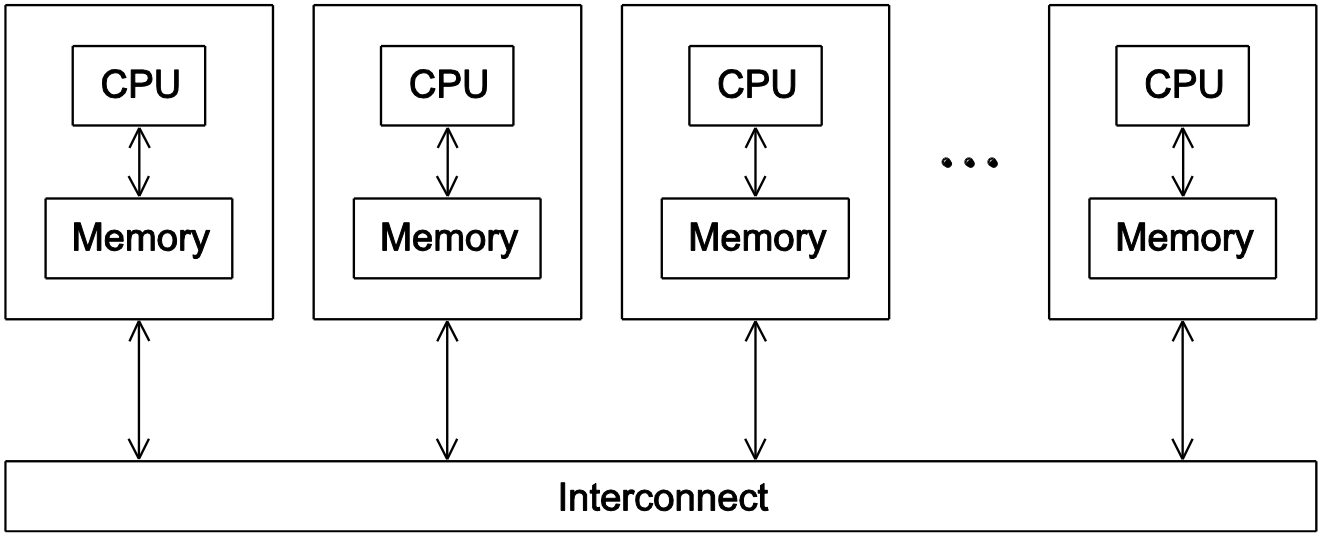
第三章

用 MPI 进行分布式内存编程

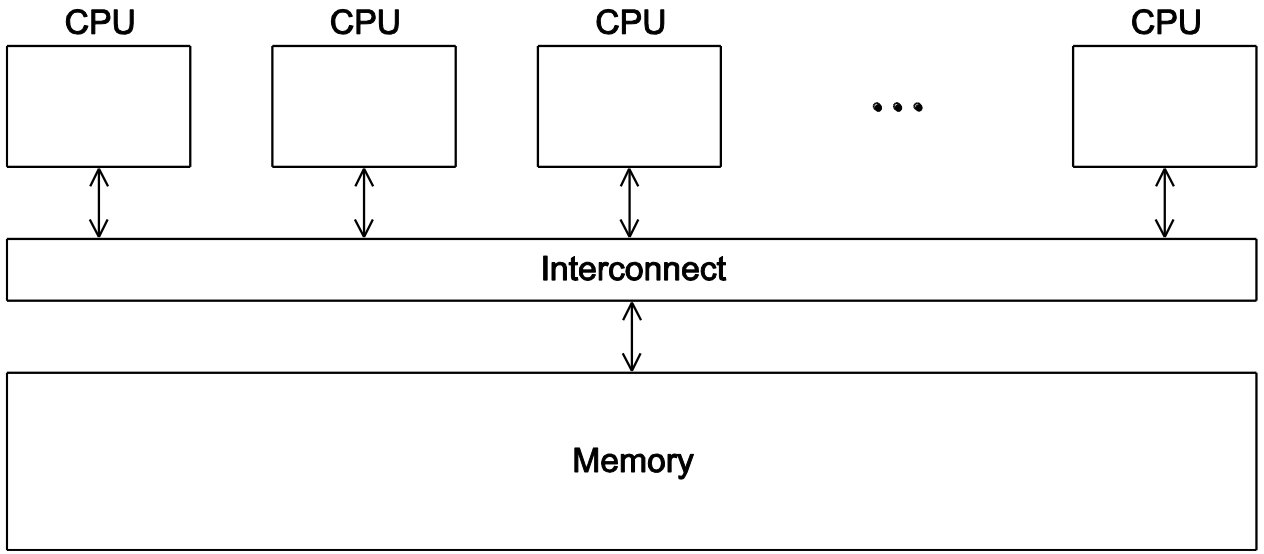
目录

- 编写你的第一个 MPI 程序.
- 使用常用的 MPI 函数.
- 用MPI来实现梯形积分法
- 集合通信.
- MPI 派生数据类型.
- MPI程序的性能评估.
- 并行排序算法.
- MPI程序的安全性.

分布式存储系统



共享内存系统



Hello World!

```
#include <stdio.h>

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

    return 0;
}
```

(经典程序)

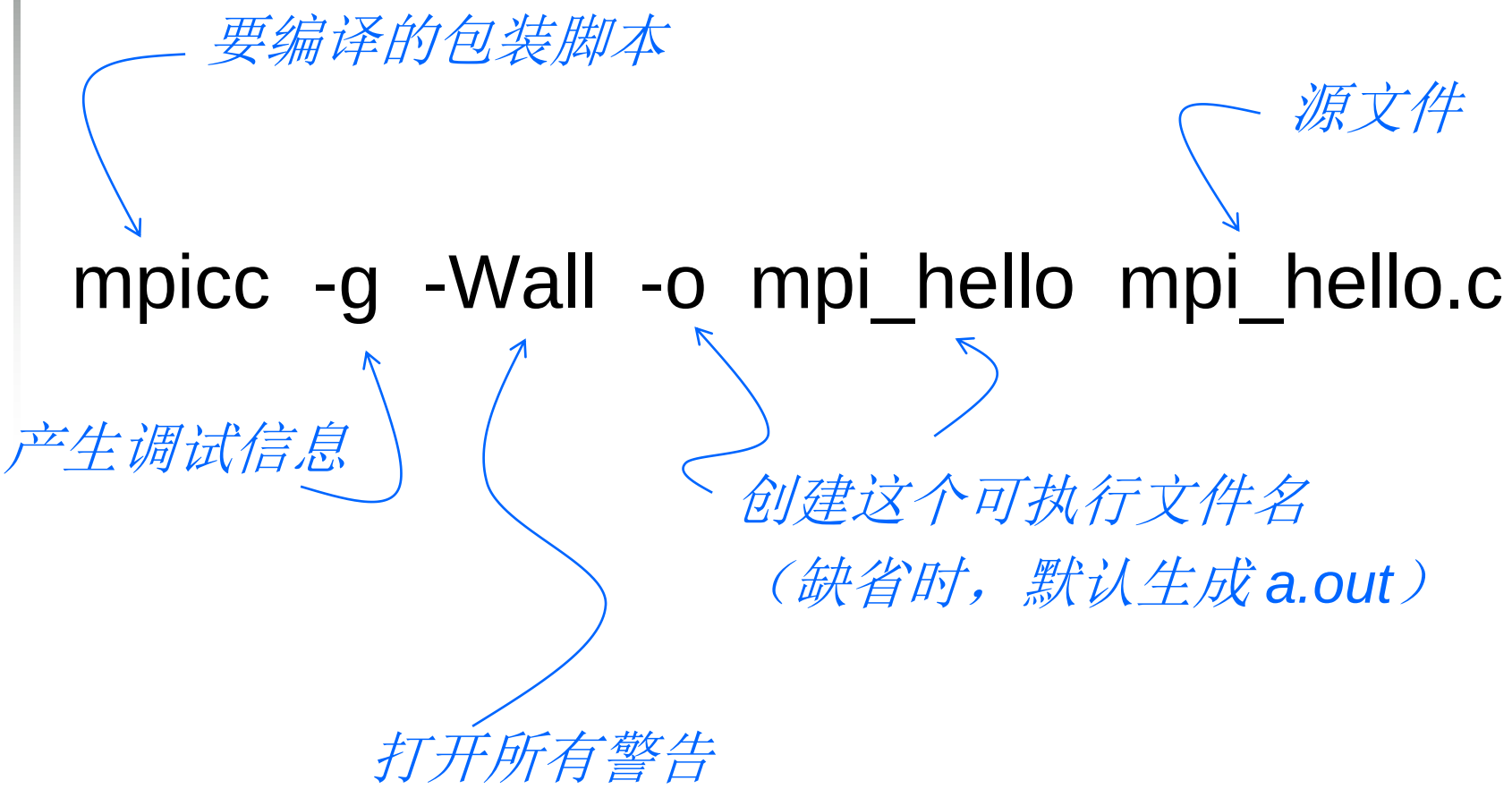
识别 MPI 进程

- 将进程按照非负整数来进行标注是常见做法
- 进程被编号为 0, 1, 2, .. p-1

第一个 MPI 程序

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

编译


要编译的包装脚本
源文件
mpicc -g -Wall -o mpi_hello mpi_hello.c
产生调试信息
创建这个可执行文件名
（缺省时，默认生成 a.out）
打开所有警告

执行

mpiexec -n <number of processes> <executable>

---

mpiexec -n 1 ./mpi_hello

运行 1 个进程

mpiexec -n 4 ./mpi_hello

运行 4 个进程

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 !
```

MPI 程序

- 写一个C程序.
  - main函数.
  - 使用 stdio.h、string.h等.
- 需要添加 mpi.h 头文件.
- MPI 定义的标识符以“MPI_”开头.
- 下划线后的第一个字母是大写.
  - 表示函数名和 MPI 定义的类型.
  - 有助于区分MPI定义与用户程序定义.

MPI 组件

- MPI_Init
  - 告诉 MPI 做所有必要的设置.

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

- MPI_Finalize
  - 告诉 MPI 我们已经完成，所以释放分配给这个程序的任何资源.

```
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（Communicators）

- 可以相互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
- 称为 MPI_COMM_WORLD.

通信子（Communicators）

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

通信子中的进程数

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

进程编号

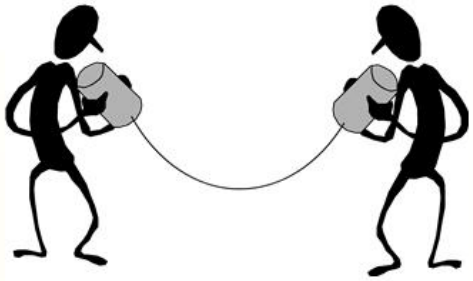
SPMD

- 单程序多数据流
- 只编译一个程序.
- 进程 0 做了一些不同的事情.
  - 在其他进程完成工作时接收消息并打印.

- if-else 结构使我们的程序是SPMD的.

通信

```
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 */);
```

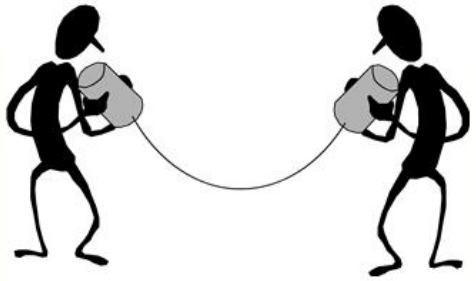


数据类型

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	

通信

```
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 */);
```



消息匹配

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

r

MPI_Send

src = q

MPI_Recv

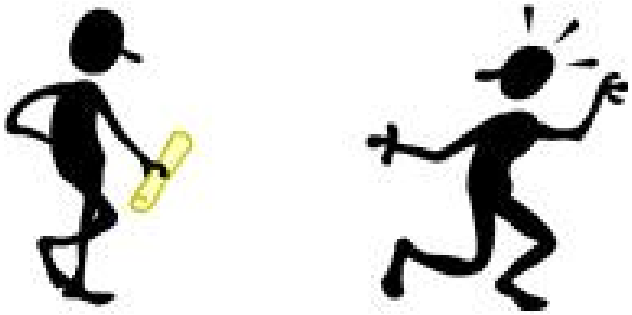
dest = r

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

q

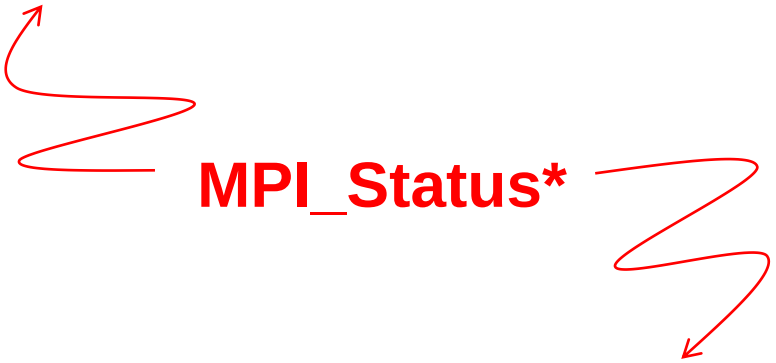
接收消息

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



status_p 参数

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



MPI_SOURCE
MPI_TAG
MPI_ERROR

MPI_Status* status;

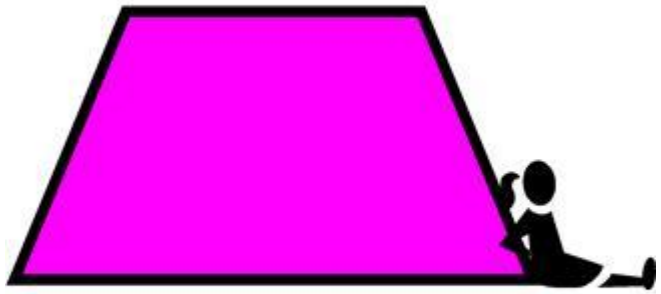
status.MPI_SOURCE
status.MPI_TAG

收到了多少数据？

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

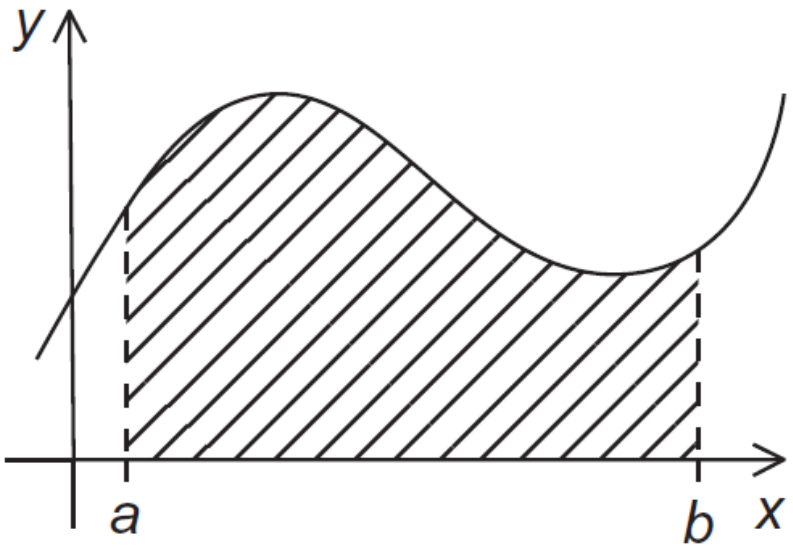
发送和接收问题

- MPI_Send的精确行为由MPI 实现所决定.
- MPI_Send 可能在缓冲区大小、截止和阻塞方面表现不同.
- MPI_Recv 始终阻塞，直到收到匹配的消息

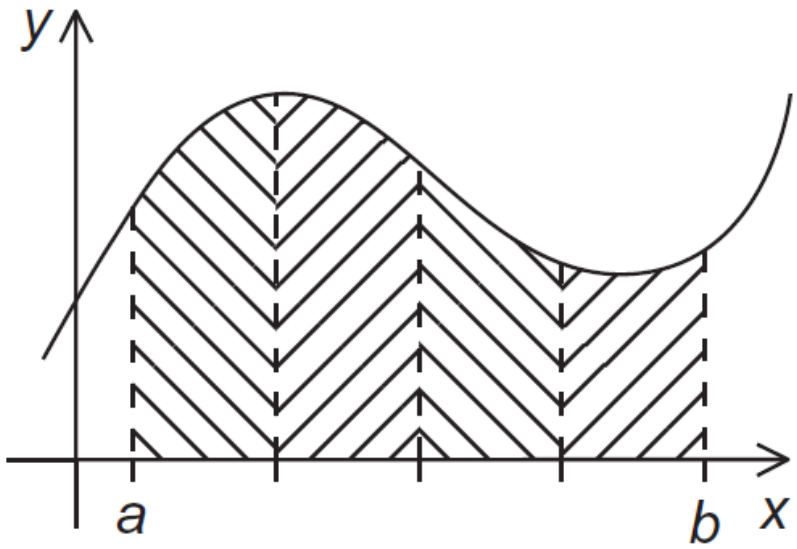


用MPI来实现梯形积分法

梯形积分法



(a)



(b)

梯形积分法

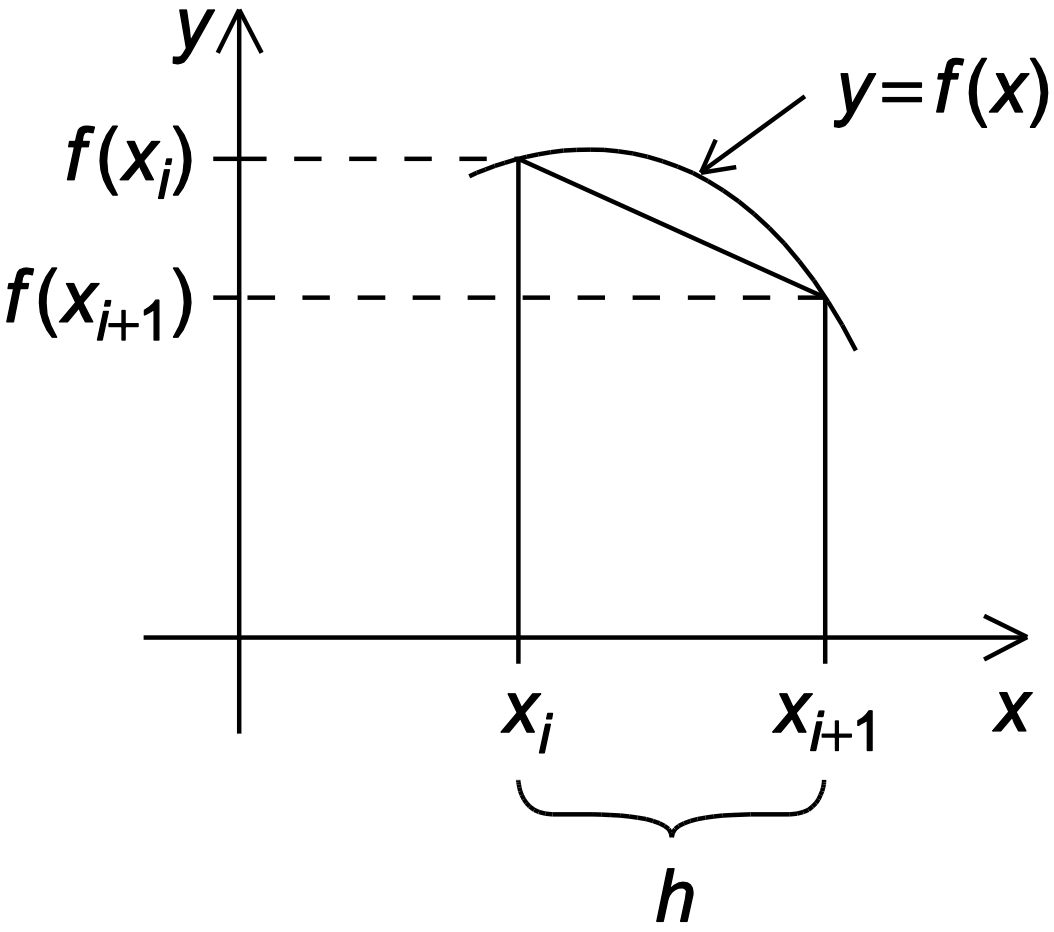
$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

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

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

一个梯形



串行程序的伪代码

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

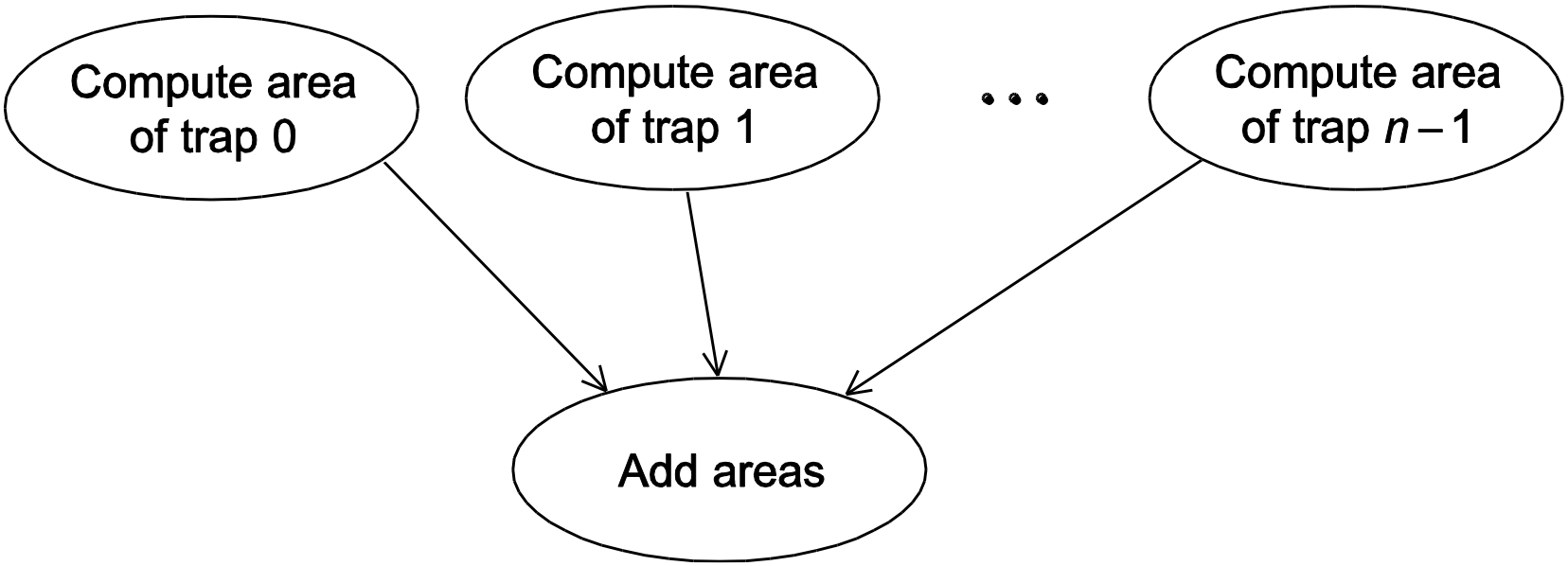
并行化梯形积分法

- 将问题解决方案划分为任务.
- 在任务间识别出需要的通信信道.
- 将任务聚合成复合任务.
- 在核上分配复合任务.

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;
```

梯形积分法的任务与通信



第一个版本 (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);
```

第一个版本 (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  */
```

第一个版本 (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

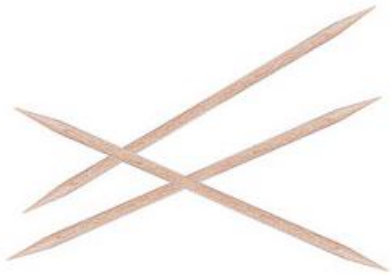
```
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz;

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

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

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

每个进程只打印一条消息。

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?
```

不可预测的输出



输入

- 大多数 MPI 实现只允许 MPI_COMM_WORLD 中的进程 0 访问 stdin.
- 0号进程负责读取数据（scanf）并将数据发给其它进程.

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

读取用户输入的函数

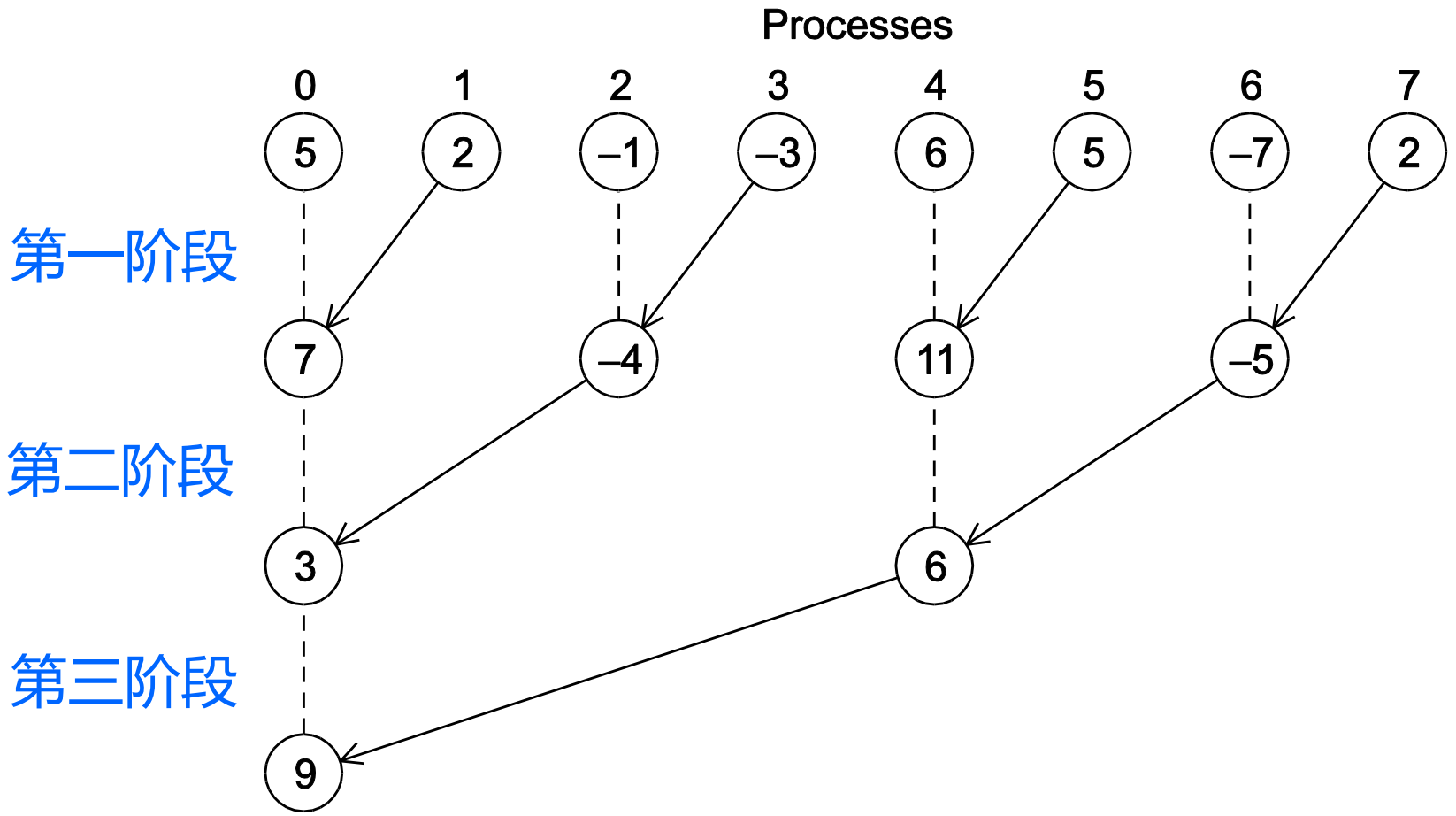
```
void Get_input(
      int       my_rank    /* in  */,
      int       comm_sz    /* in  */,
      double*   a_p        /* out */,
      double*   b_p        /* out */,
      int*      n_p        /* out */) {
   int dest;

   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 */
```

集合通信

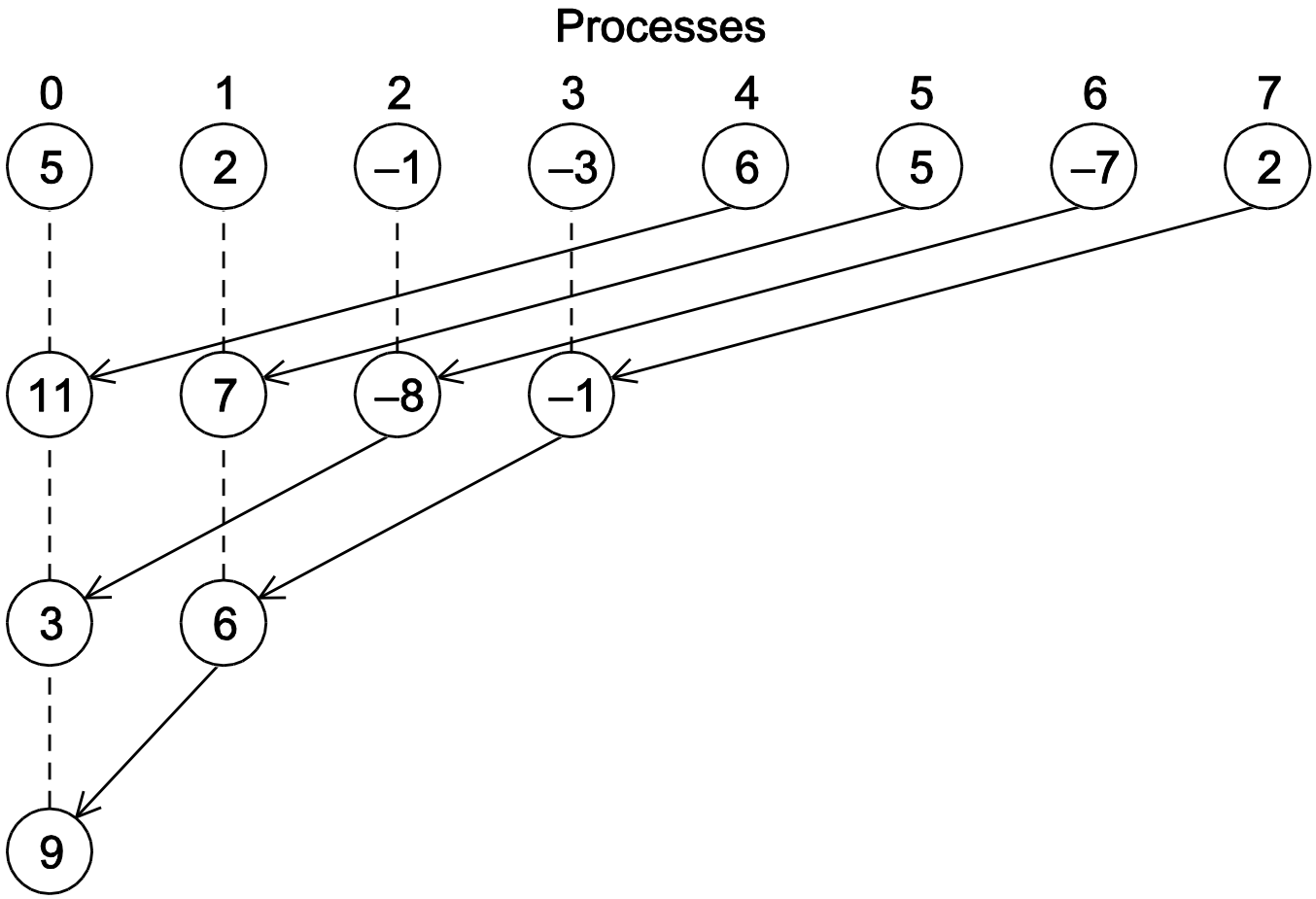
树结构通信

- 第一阶段:
  (a) 进程1发送给0，3发送给2，5发送给4，7发送给6.
  (b) 进程 0、2、4 和 6 将接收到的值相加到原有值上.
- 第二阶段:
  (a) 进程 2 和 6 分别将它们的新值发送给进程 0 和 4.
  (b) 进程 0 和 4 将接收到值加到它们新值上.
- 第三阶段:
  (a) 进程 4 将其最新值发送给进程 0.
  (b) 进程 0 将接收到的值加到它最新的值上.

树结构的全局总和



树形结构全局求和的另一种方法



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 中预定义的归约操作符

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

集合通信与点对点通信

- 通信子中的所有进程必须调用相同的集合通信函数.
  - 例如，一个程序试图将一个进程中的MPI_Reduce调用与另一个进程的MPI_Recv调用相匹配，程序会出错，此时程序会被挂起或崩溃.

集合通信与点对点通信

- 每个进程传递给 MPI 集合通信的参数必须是“相容的”。
  - 例如，如果一个进程将0作为dest_process的值传递给函数，另一个进程传递的是1，MPI_Reduce调用产生的结果是错误的，程序可能被挂起或崩溃.

集合通信与点对点通信

- output_data_p 参数只用在dest_process上. 但是，所有的进程仍然需要传入一个与output_data_p对应的实参，即使它只是NULL.

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
- 集合通信不使用标签.
- 只通过通信子和调用的顺序来进行匹配

示例 (1)

对MPI_Reduce的多个调用

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

- 假设每个进程调用MPI_Reduce函数的运算符都是MPI_SUM，目标进程为0号进程.
- 乍一看，在两次调用MPI_Reduce之后，b的值为3，d的值为6.

例子 (2)

- 但是，内存单元的名字与MPI_Reduce的调用匹配无关.

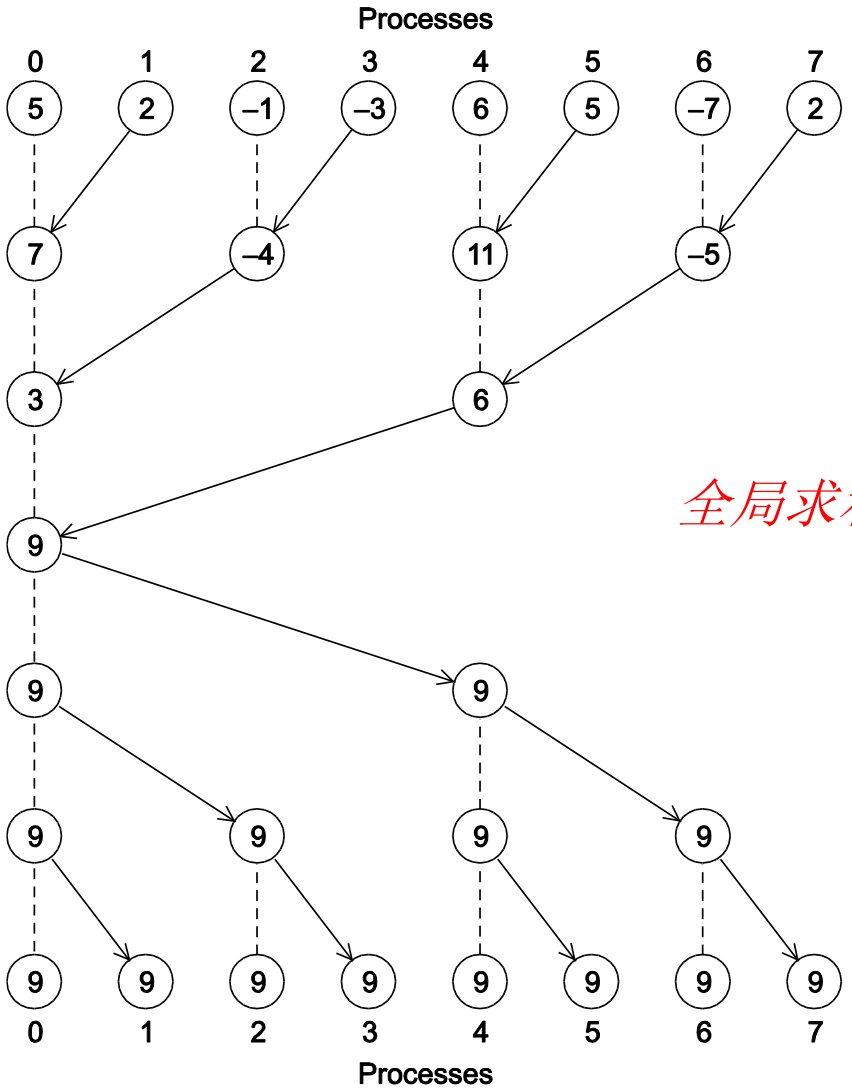
- 函数调用的顺序决定了匹配方式，所以b中存储的值将是1+2+1= 4，d 中存储的值将是 2+1+2 = 5.

MPI_Allreduce

- 在所有进程都需要全局求和的结果，以完成一些更大计算的情况下很有用.

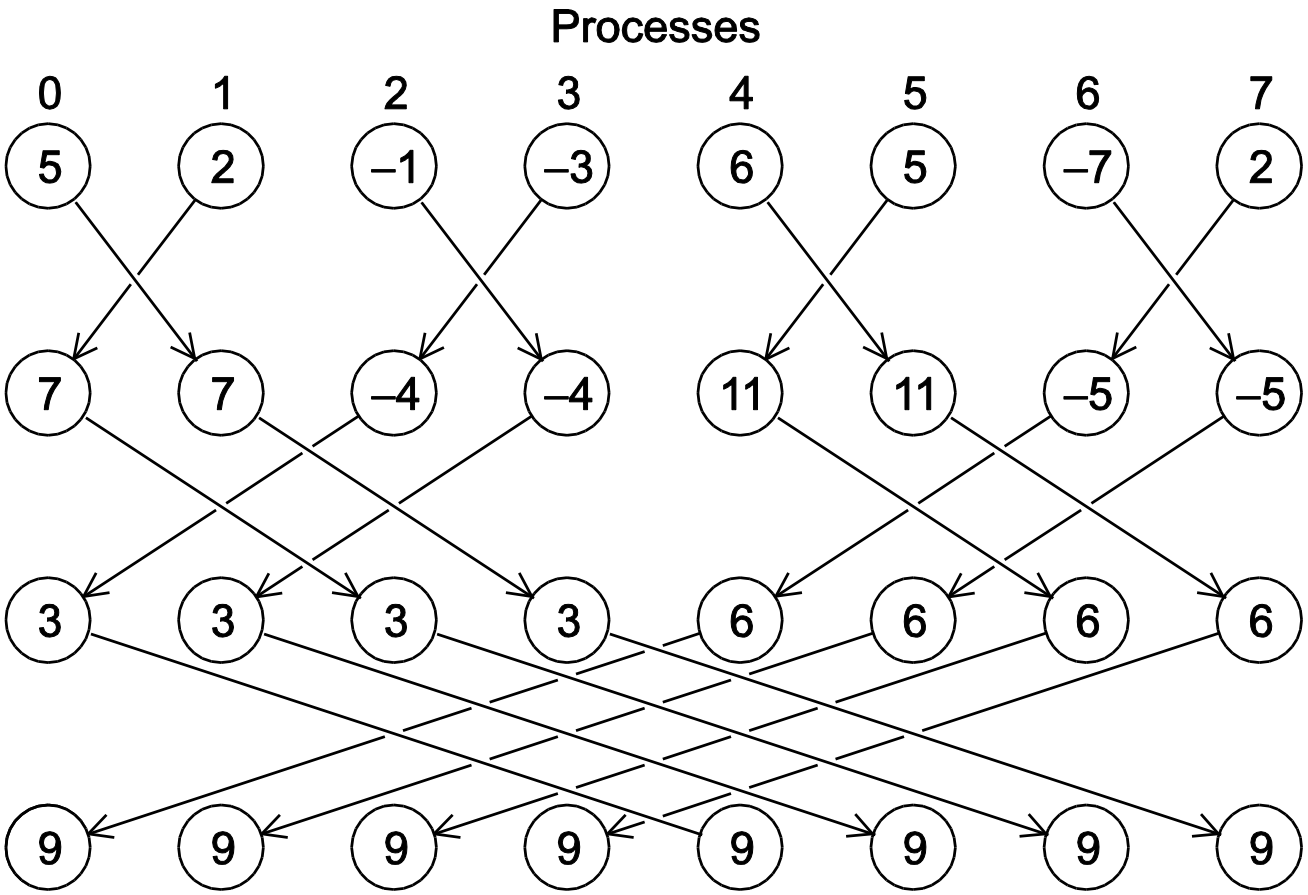
```
int MPI_Allreduce(
          void*          input_data_p    /* in  */,
          void*          output_data_p   /* out */,
          int            count           /* in  */,
          MPI_Datatype   datatype        /* in  */,
          MPI_Op         operator        /* in  */,
          MPI_Comm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

MPI_Allreduce



全局求和计算结果的发布

MPI_Allreduce



蝴蝶结构的全局求和

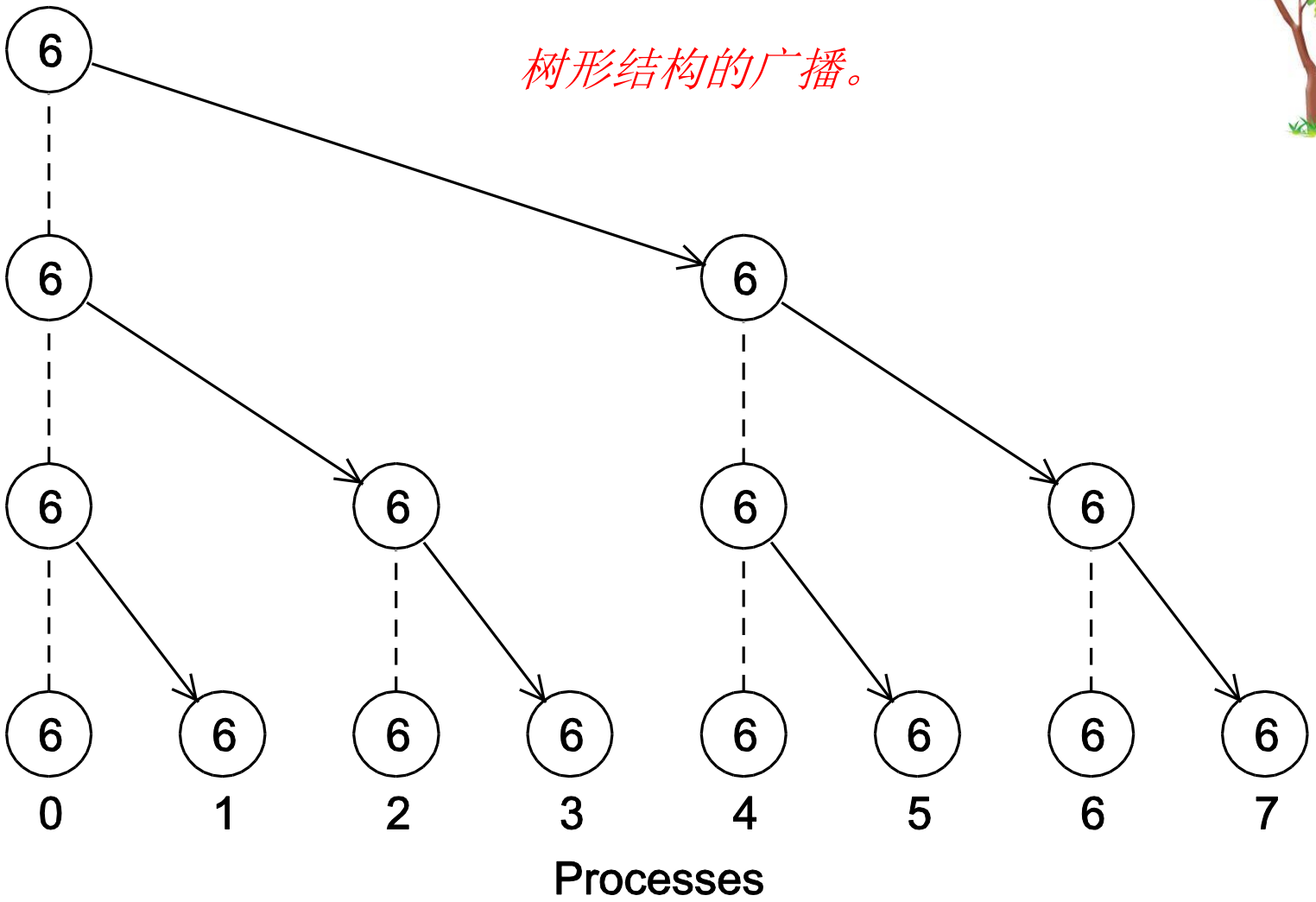
广播

- 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     */);
```

广播

树形结构的广播。



使用 MPI_Bcast 的 Get_input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 */
```

数据分发

向量求和

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

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 */
```

划分方式

- 块划分
  - 将连续向量分量所构成的块分配给每个进程.
- 循环划分
  - 以轮转的方式方式分配向量分量所构成的块.
- 块-循环划分
  - 使用一个循环来分发向量分量所构成的块.

3个进程对12 分量的向量的不同划分

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

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 */
```

散射（Scatter）

- MPI_Scatter可以在一个函数中使用，该函数0号进程读取整个向量，但只将分量发送给需要分量的其它进程。

```
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  */);
```

读取并分发向量的函数

```
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 */
```

聚集（Gather）

- 将向量的所有分量收集到进程0上，然后进程0就可以处理所有的分量了。

```
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  */);
```

打印分布式向量的函数 (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;
```

打印分布式向量的函数 (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 */
```

全局聚集（Allgather）

- 将每个进程的send_buf_p的内容串联起来，存储到每个进程的recv_buf_p参数中.
- 通常，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 x 的点积。

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

将矩阵乘以向量

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数组类型

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

存储为

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

串行矩阵向量乘法

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

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

MPI矩阵-向量乘法函数

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

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

MPI的派生数据类型

派生数据类型

- 通过存储数据项的类型及其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.
- 主要思想是，如果发送数据的函数知道有关数据项集合的信息，它可以在发送之前在内存中将数据项聚集起来.
- 类似地，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的正确目标位置.

派生数据类型

- 形式上，由一系列基本MPI数据类型以及每个数据类型的偏移所组成.
- 在梯形积分法的例子:

Variable	Address
a	24
b	40
n	48

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

MPI 数据类型的构造

- 构造由不同基本类型的单个元素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 location_p 所指向的内存单元的地址.
- 特殊类型 MPI_Aint 是整数类型，它的长度足以表示系统地址.

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

MPI_Type_commit

- 允许 MPI 实现在通信函数内使用这一数据类型，还优化了数据类型的内部表示.

```
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，可以用该函数调用去释放额外的存储空间.

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

使用派生数据类型获取输入函数(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};
   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 */
```

使用派生数据类型获取输入函数 (2)

```
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 */
```

性能评估

已用并行时间

- 返回过去一段时间以来经过的秒数.

```
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);
```

已用串行时间

- 在这种情况下，不需要链接 MPI 库。
- 返回过去某时间点开始流逝时间(微秒为单位)。

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

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

MPI_Barrier

- 确保同一通信子中的所有进程，在调用它之前，没有进程能提前返回。

```
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);
```

串行和并行矩阵向量乘法的运行时间

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

(秒)

加速比和效率

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

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

并行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

并行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

- 如果问题的规模可以以一定的速度增加，那么程序是可扩展的，这样效率不会随着进程数量的增加而降低。
- 可以在不增加问题规模的情况下保持恒定效率的程序有时被认为具有强可扩展性.
- 如果问题规模以与进程数量相同的速度增长，则可以保持恒定效率的程序有时被称为弱可扩展性.

并行排序算法

排序

- n 个键值和 p = comm_sz 个进程。
- 给每个进程分配 n/p 个键值。
- 哪些键分配给哪些进程没有限制。
- 当算法终止时：
  - 分配给每个进程的键值应按增序排序.
  - 如果 0 ≤ q < r < p，那么分配给进程 q 的每个键值应该小于或等于分配给进程 r 的每个键值。

串行冒泡排序

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

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

}   /* Bubble_sort */
```

并行奇偶交换排序

- 一系列阶段.
- 偶数阶段，比较互换:

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

- 奇数阶段，比较互换:

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

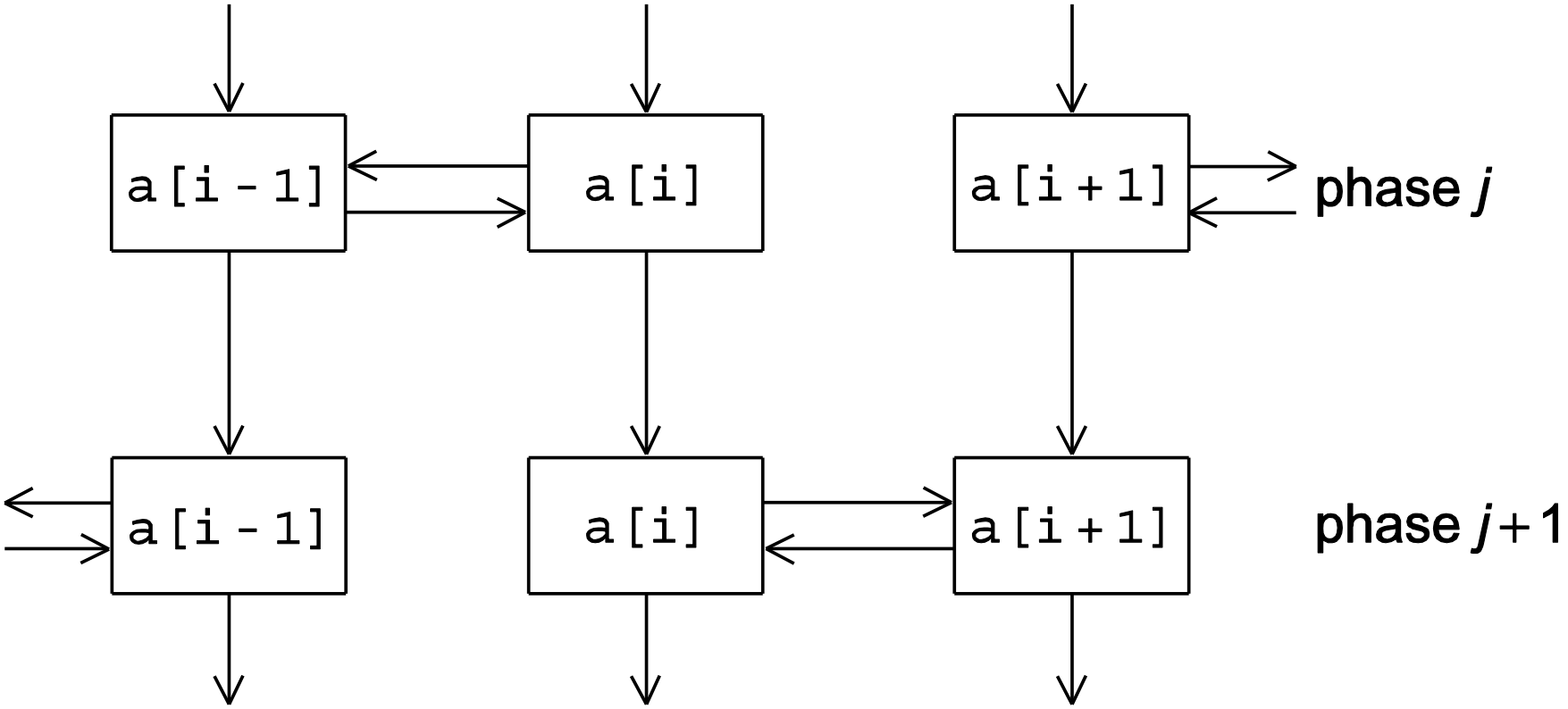
例子

- 开始： 5, 9, 4, 3
- 偶数阶段：比较交换 (5,9) 和 (4,3) 得到列表 5, 9, 3, 4
- 奇数阶段：比较交换 (9,3) 得到列表 5, 3, 9, 4
- 偶数阶段：比较交换 (5,3) 和 (9,4) 得到列表 3, 5, 4, 9
- 奇数阶段：比较交换 (5,4) 得到列表 3, 4, 5, 9

串行奇偶交换排序

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

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

奇偶排序任务间的通信



确定 a[i] 的任务用 a[i] 标记。

并行奇偶交换排序

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

伪代码

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

Compute_partner

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

MPI 程序的安全性

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

MPI 程序的安全性

- 许多MPI 函数设置了使系统从缓冲切换到阻塞的阈值.
- 相对较小的消息将被 MPI_Send 缓冲.
- 较大的消息，将导致它被阻塞.

MPI 程序的安全性

- 如果每个进程阻塞在MPI_Send ，则没有进程可以开始执行对MPI_Recv 的调用，程序将挂起或死锁.
- 每个进程都被阻塞，等待一个永远不会发生的事件.

(见伪代码)

MPI 程序的安全性

- 依赖于MPI提供的缓冲机制的程序被认为是不安全的
- 这样的程序在运行一些输入集时没有问题，但有可能在运行其它输入集时会挂起或者崩溃

MPI_Ssend

- MPI 标准定义的 MPI_Send 的替代方案.
- 额外的“s”代表同步并且 MPI_Ssend 保证了直到对应的接收开始前，发送端一直阻塞.

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

重构通信

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

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

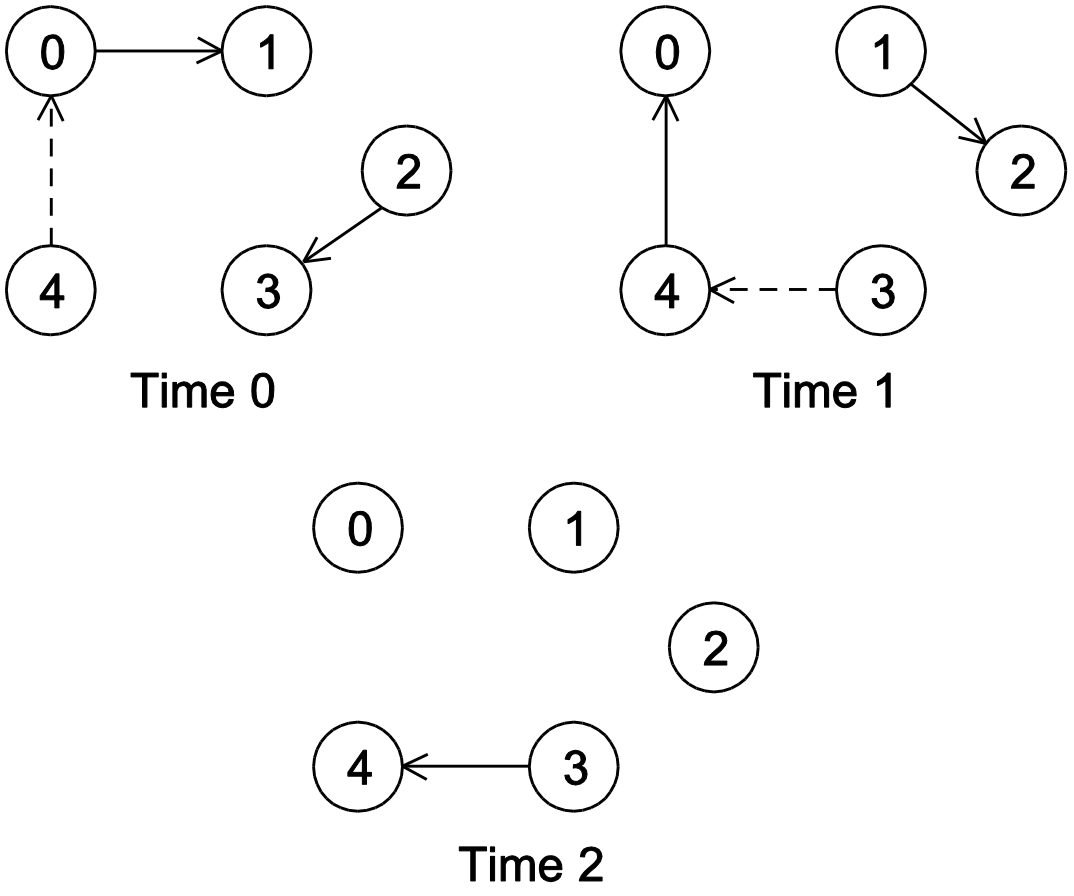
MPI_Sendrecv

- MPI提供的自己调度通信的方法.
- 在单个调用中执行阻塞发送和接收.
- dest 和 source 可以相同也可以不同.
- 有用之处是，MPI实现了通信调度，程序不再会挂起或崩溃.

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  */);
```

五个进程之间的安全通信



并行奇偶转置排序

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

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

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

并行奇偶排序的运行时间

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

(times are in milliseconds)

小结 (1)

- MPI 或消息传递接口是一个可以从 C、C++ 或 Fortran 程序调用的函数库.
- 通信子是可以相互发送消息的进程的集合.
- 许多并行程序使用单程序多数据流（SPMD）方法.

小结(2)

- 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.
- 并行程序通常不具备这个特性.
- 集合通信涉及通信子中的所有进程.

小结(3)

- 当我们为并行程序计时时，我们通常对经过的时间或“wall clock time”感兴趣.
- 加速比是串行运行时间与并行运行时间的比率.
- 效率是加速比除以并行进程数.

小结(4)

- 如果可以增加问题大小 (n) 使得效率不会随着 p 的增加而降低，那么并行程序被称为可扩展的.
- 如果 MPI 程序的正确行为取决于 MPI_Send 正在缓冲的输入，则该 MPI 程序是不安全的.

编程案例（课后题3.2）

假设我们向一个正方形飞镖板随机地投掷飞镖，飞镖板的边长为2英尺，靶心在正中央。再在正方形板上画了一个半径为1英尺的圆，面积为 π 平方英尺。如果飞镖击中靶子后的得分是平均分布的（我们总能投进正方形区域），则击中圆形区域内的数量应该大致满足等式：

击中圆内的投掷次数 / 全部的投掷次数=π/4

编程案例

我们可以使用这个公式配合随机数生成器来估计 π 的值：

```
number_in_circle = 0;
for(toss = 0; toss < number_of_tosses; toss++){
    x = random double between - 1 and 1;
    y = random double between - 1 and 1;
    distance_squared = x * x + y * y;
    if(distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4 * number_in_circle / ((double) number_of_tosses);
```

这称为蒙特卡洛方法，因为它使用了随机特性（飞镖投掷）

编程案例

编写一个用蒙特卡洛方法估计 π 的 MPI 程序，进程0读入总的投掷次数，并把它们广播给各个进程。使用MPI_Reduce求出局部变量number_in_cycle的全局总和，并让进程0打印它。击中圆内部的次数和投掷总数可能要使用Long Long int 类型的数值来表示，为了获得较精确的 π 估计值，这两个数值应该要大一些。

```
int main(void) {
   long long int number_of_tosses;
   long long int local_number_of_tosses;
   long long int number_in_circle;
   long long int local_number_in_circle;

   double pi_estimate;
   int my_rank, comm_sz;
   MPI_Comm comm;

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

   Get_input(&number_of_tosses, my_rank, comm);
   local_number_of_tosses = umber_of_tosses/comm_sz;

#  ifdef DEBUG
   printf("Proc %d > number of tosses = %lld, local
number = %lld\n", my_rank, number_of_tosses,
local_number_of_tosses);
#  endif
   local_number_in_circle =
Monte_carlo(local_number_of_tosses, my_rank);

   MPI_Reduce(&local_number_in_circle,
&number_in_circle, 1, MPI_LONG_LONG, MPI_SUM, 0,
comm);

   if ( my_rank == 0 ){
      pi_estimate =
4*number_in_circle/((double)number_of_tosses);
      printf("pi estimate = %f\n", pi_estimate);
   }

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

```
void Get_input(
      long long int* number_of_tosses  /* out */,
      int my_rank                      /* in  */,
      MPI_Comm comm                    /* in  */) {

   if (my_rank == 0 ) {
      printf("Enter the total number of tosses\n");
      scanf("%lld", number_of_tosses);
   }

   MPI_Bcast(number_of_tosses, 1, MPI_LONG_LONG,
0, comm);
}  /* Get_input */
```

```
long long int Monte_carlo(long long
local_number_of_tosses, int my_rank) {
   long long int i;
   double x,y;
   double distance_squared;
   long long int number_in_circle = 0;

   srandom(my_rank+1);
   for ( i=0 ; i< local_number_of_tosses ; i++) {
      x =  2*random()/((double)RAND_MAX) - 1.0;
      y =  2*random()/((double)RAND_MAX) - 1.0;
      distance_squared = x*x + y*y;
#     ifdef DEBUG
      printf("Proc %d > x = %f, y = %f, dist squared =
%f\n", my_rank, x, y, distance_squared);
#     endif
      if (distance_squared <= 1) {
         number_in_circle++;
      }
   }
   return number_in_circle;
}  /* Monte_carlo */
```

编程案例

编写一个MPI程序，采用树形通信结构来计算全局总和。首先计算comm_sz是2的幂的特殊情况，若能够正确运行，改变该程序使其适用于所有comm_sz的值。

```
const int MAX_CONTRIB = 20;
int main(void) {
   int i, sum, my_int;
   int my_rank, comm_sz;
   MPI_Comm comm;
   int* all_ints = NULL;

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

   srandom(my_rank + 1);
   my_int = random() % MAX_CONTRIB;

   sum = Global_sum(my_int, my_rank, comm_sz,
comm);
```

```
 if ( my_rank == 0) {
      all_ints = malloc(comm_sz*sizeof(int));
      MPI_Gather(&my_int, 1, MPI_INT, all_ints, 1,
MPI_INT, 0, comm);
      printf("Ints being summed:\n   ");
      for (i = 0; i < comm_sz; i++)
         printf("%d ", all_ints[i]);
      printf("\n");
      printf("Sum = %d\n",sum);
      free(all_ints);
   } else {
      MPI_Gather(&my_int, 1, MPI_INT, all_ints, 1,
MPI_INT, 0, comm);
   }

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

```
int Global_sum(
      int my_int    /* in */,
      int my_rank   /* in */,
      int comm_sz   /* in */,
      MPI_Comm comm /* in */) {

   int partner, recvtemp;
   int my_sum = my_int;
   unsigned bitmask = 1;

   while (bitmask < comm_sz) {
      partner = my_rank ^ bitmask;
      if (my_rank < partner) {
         MPI_Recv(&recvtemp, 1, MPI_INT, partner, 0,
comm, MPI_STATUS_IGNORE);
         my_sum += recvtemp;
         bitmask <<= 1;
      } else {
         MPI_Send(&my_sum, 1, MPI_INT, partner, 0,
comm);
         break;
      }
   }  /* while */

   return my_sum;
}  /* Global_sum */
```