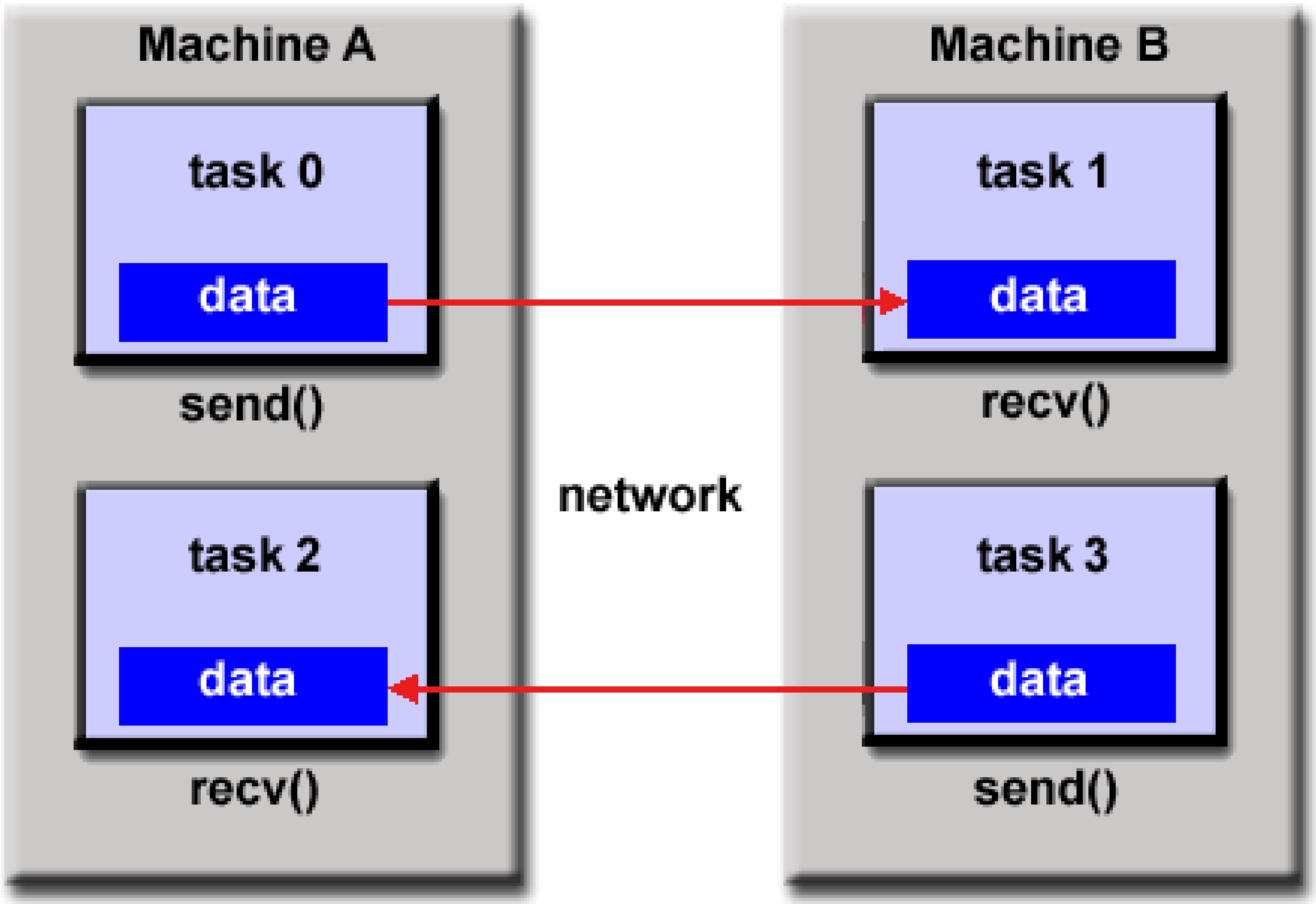
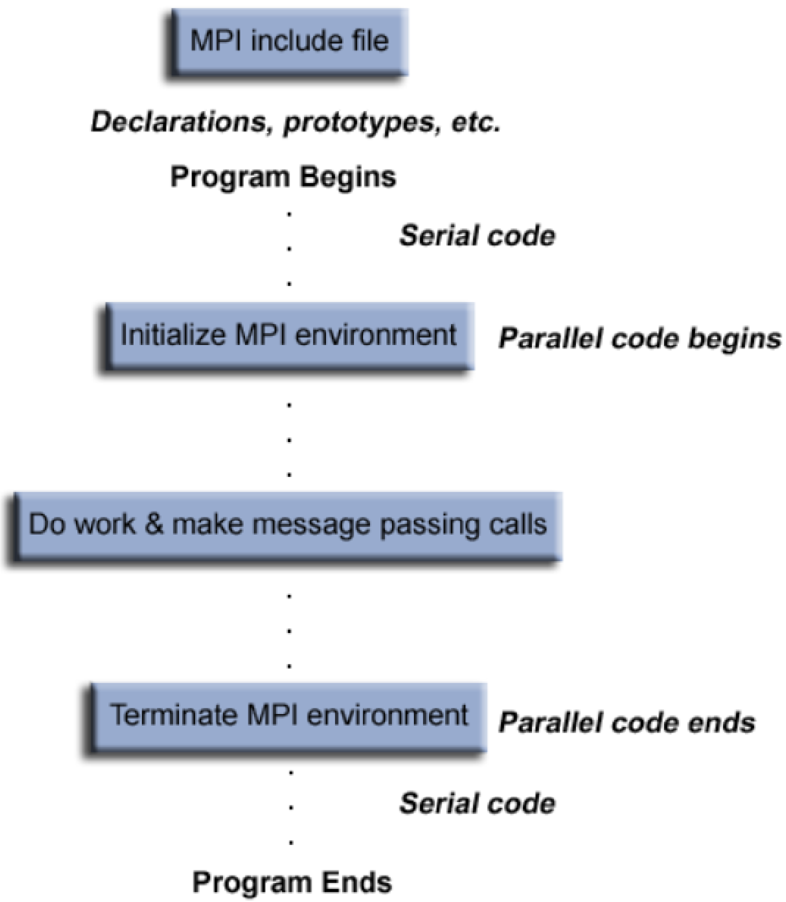
并行计算

（Parallel Computing）


Machine A
task 0
data
send()
task 2
data
recv()
network
Machine B
task 1
data
recv()
task 3
data
send()



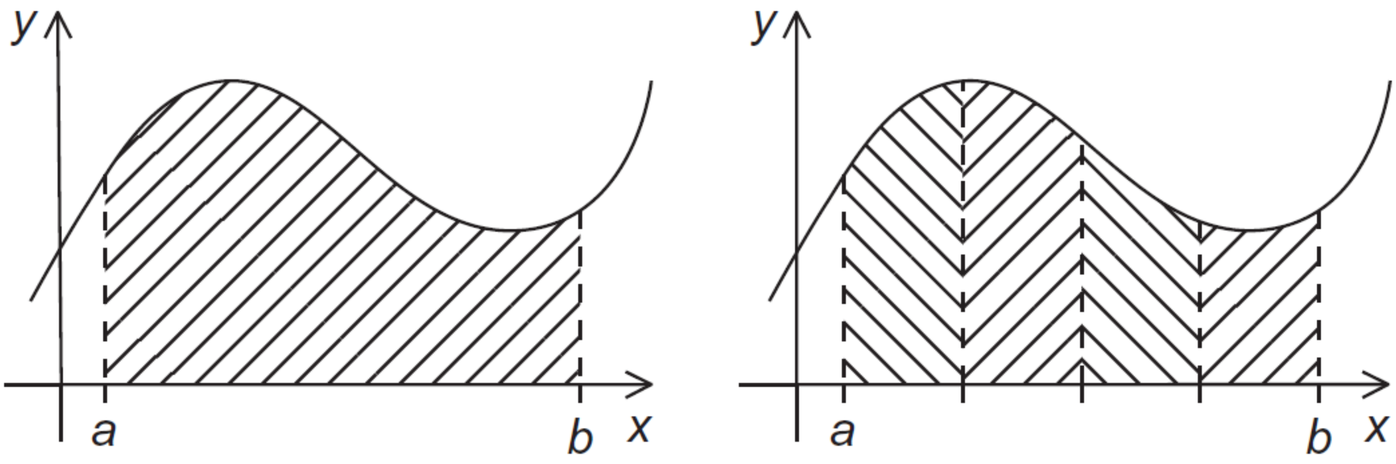

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

分布式内存编程 – MPI（二）

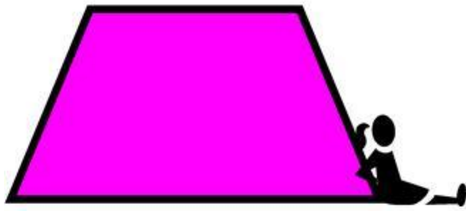
学习内容：

- 梯形法则（ Trapezoidal Rule ）求面积
- I/O处理
- 集体通信（Collective communication）

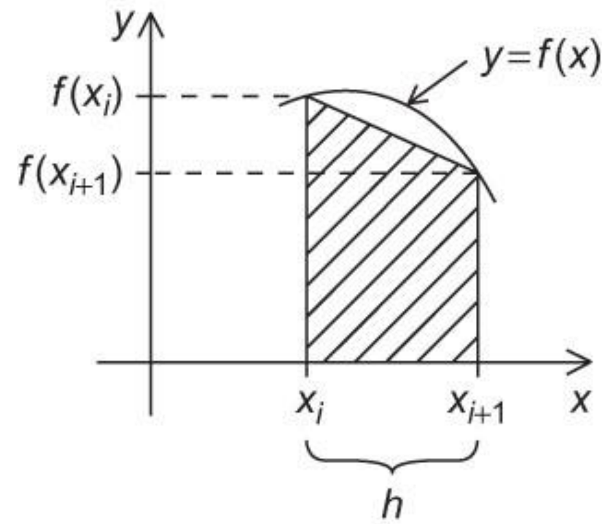
1.梯形法则（ Trapezoidal Rule ）数值积分



(a) (b)



1.梯形法则（ Trapezoidal Rule ）数值积分



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

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

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

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

1.梯形法则（Trapezoidal Rule）数值积分

- 串行程序的伪代码

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

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

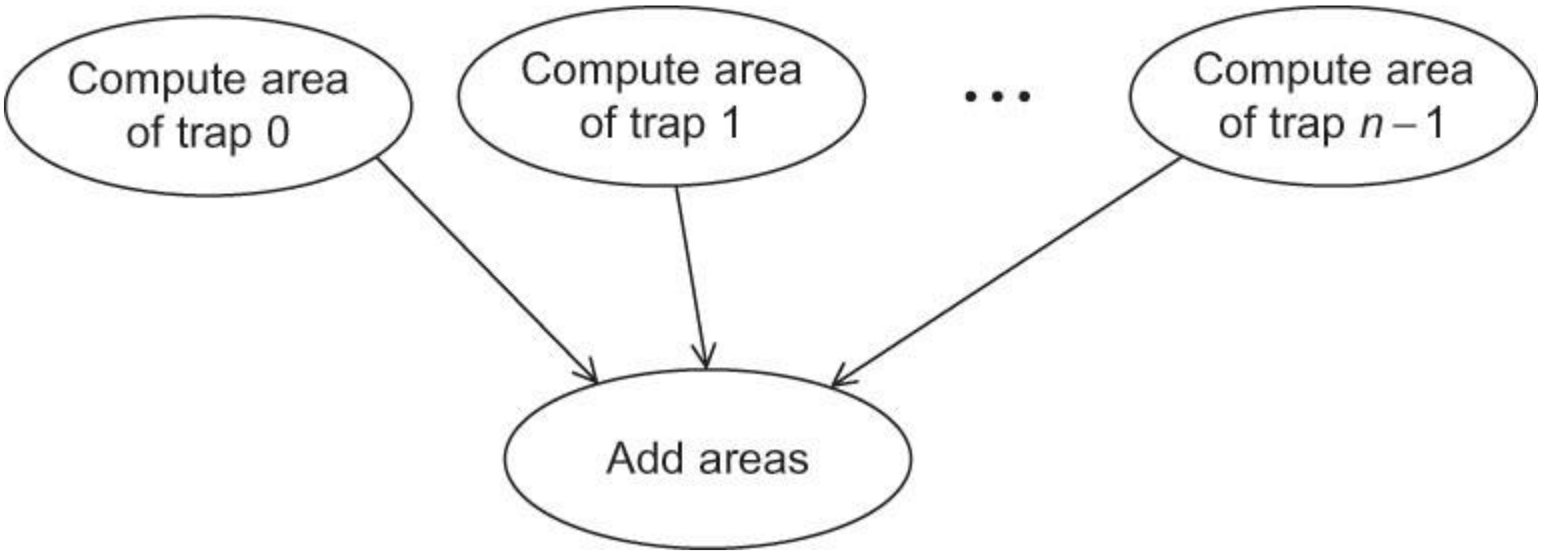
```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

1.梯形法则（ Trapezoidal Rule ）数值积分

- 并行化
  - 将问题解决方案划分为任务
  - 确认任务之间的通信
  - 聚合任务为组合任务
  - 将确定的任务映射到core

1.梯形法则（ Trapezoidal Rule ）数值积分

- 并行化



1.梯形法则（ Trapezoidal Rule ）数值积分

- 并行化

```
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;
```

1.梯形法则（ Trapezoidal Rule ）数值积分

●并行化

```
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);
```

1.梯形法则（ Trapezoidal Rule ）数值积分

● 并行化

```
   } 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;
      }
   }

   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 */
```

1.梯形法则（Trapezoidal Rule）数值积分

●并行化

```
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  */
```

2.处理 I/O

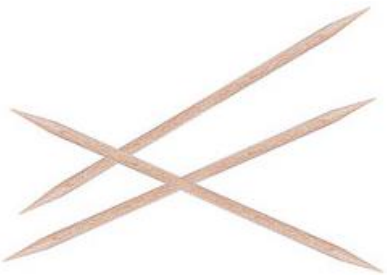
```
#include <stdio.h>
#include <mpi.h>

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 */
```

Each process just prints a message.

2.处理 I/O

```
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?
```

unpredictable output



2.处理 I/O

- 输入
  - 大多数 MPI 的实现仅允许 MPI_COMM_WORLD 中的进程 0 访问 stdin
  - 进程 0 必须读取数据 (scanf) 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;
. . .
```

2.处理 I/O

- 输入

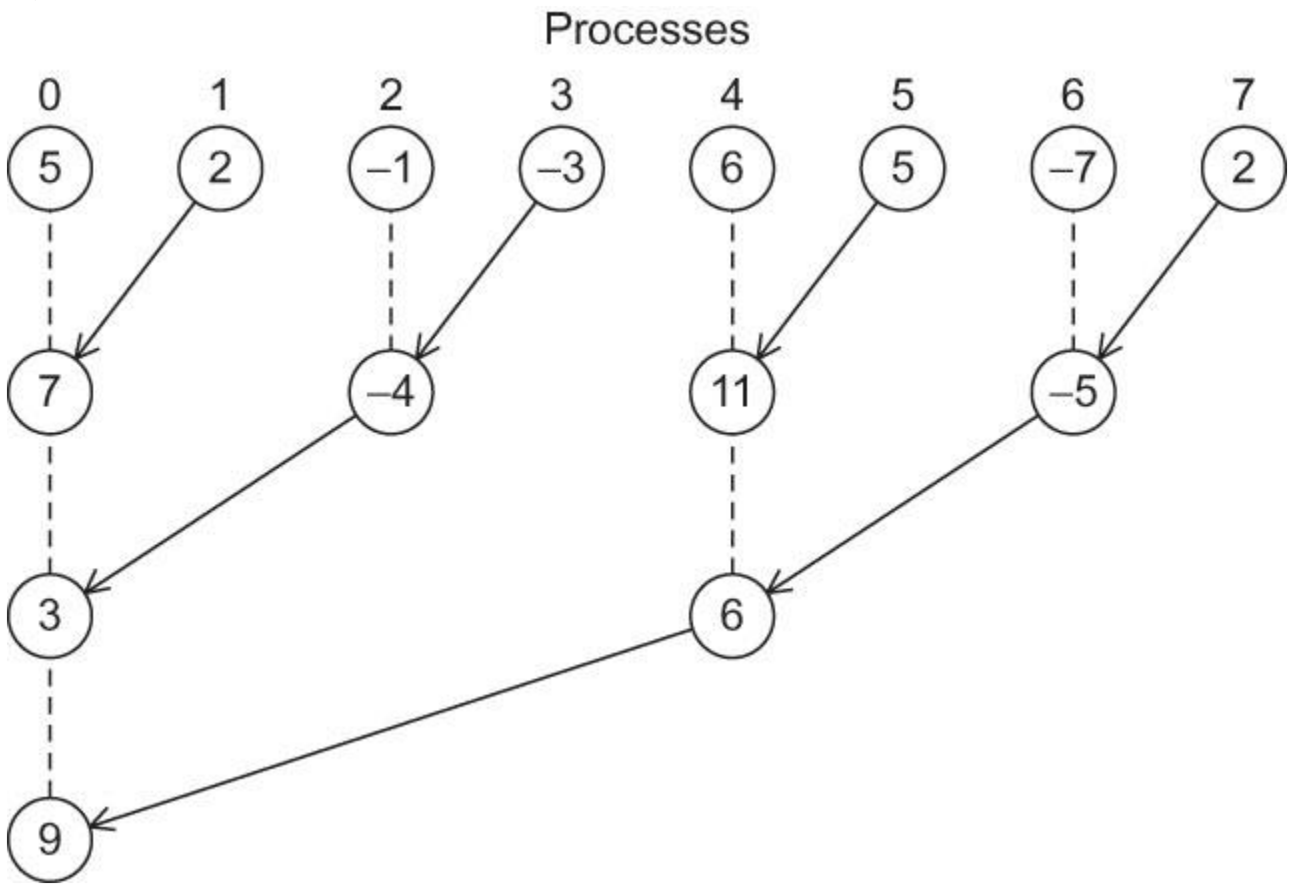
```
void Get_input(
      int      my_rank  /* in  */,
      int      comm_sz  /* in  */,
      double*  a_p      /* out */,
      double*  b_p      /* out */,
      int*     n_p      /* out */) {
   int dest;

   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.集体通信（Collective communication）

- *有哪些可以改进之处？*
  - 其他进程将部分面积和发送给进程0后退出
  - 导致进程0在工作时，其他进程空闲

3.集体通信（Collective communication）

●树结构的通信



3.集体通信（Collective communication）

●树结构的通信

a. 进程 1, 3, 5, 7 分别发送数据给进程 0, 2, 4, 6

b. 进程 0, 2, 4, 6 累加接收到的值

c. 进程 2 和 6 分别发送累加后的值给进程 0和 4

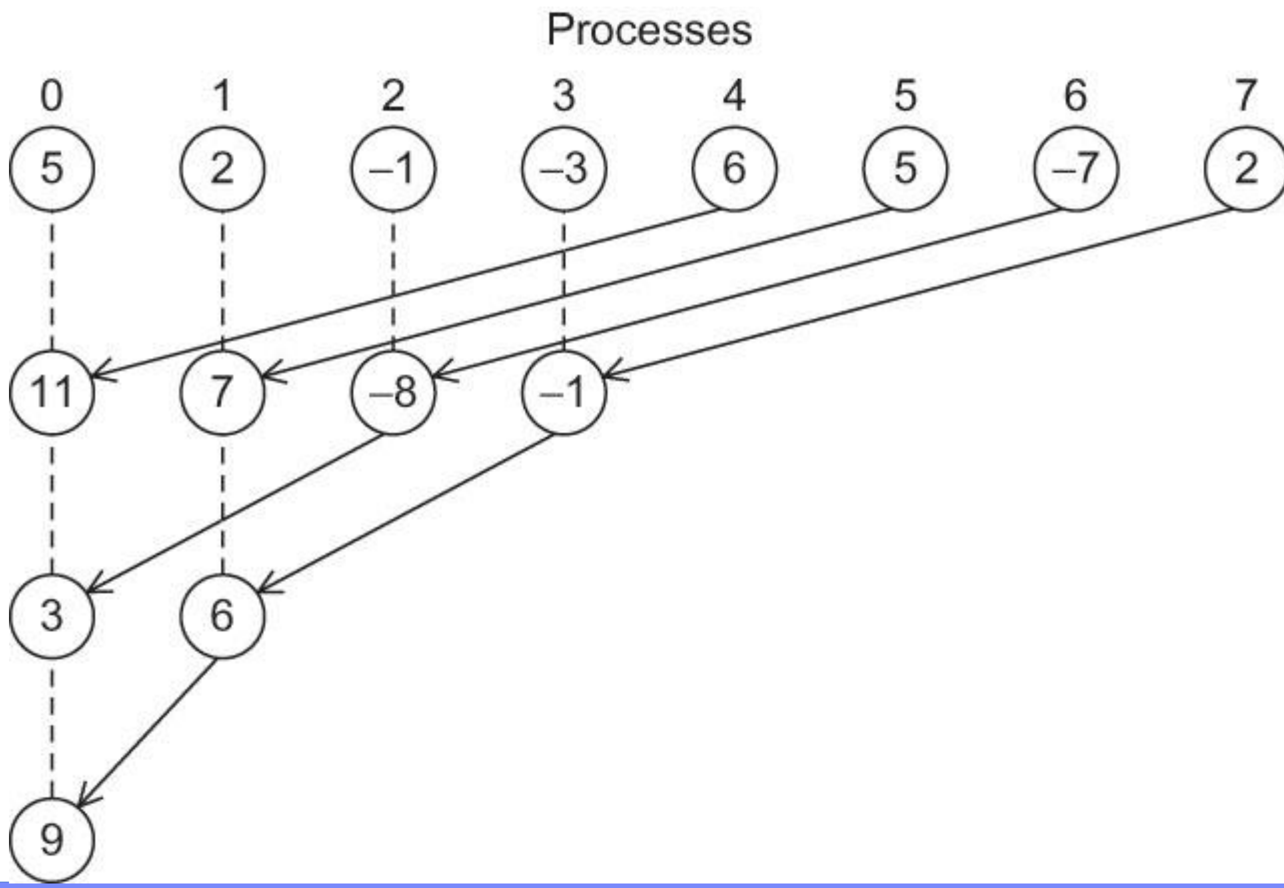
d. 进程 0 和 4 累加接收到的值

e. 进程 4 发送新的累加值给进程 0

f. 进程 0 累加接收到的值

3.集体通信（Collective communication）

●树结构的通信



3.集体通信（Collective communication）

- MPI_Reduce
  - 实现 global_sum 的方法有很多，如何比较它们哪个最优？
  - MPI 提供了 global sum 函数，由 MPI 的实现来完成优化工作
  - 不同于 MPI_Send – MPI_Recv，global_sum 函数需要多于两个进程之间进行通信，在梯形法则求面积的例子中，包含了 MPI_COMM_WORLD 中所有的进程
  - 这种通信称为 collective commnunication
  - MPI_Send – MPI_Recv 称为点对点通信

3.集体通信（Collective communication）

- 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);
```

3.集体通信（Collective communication）

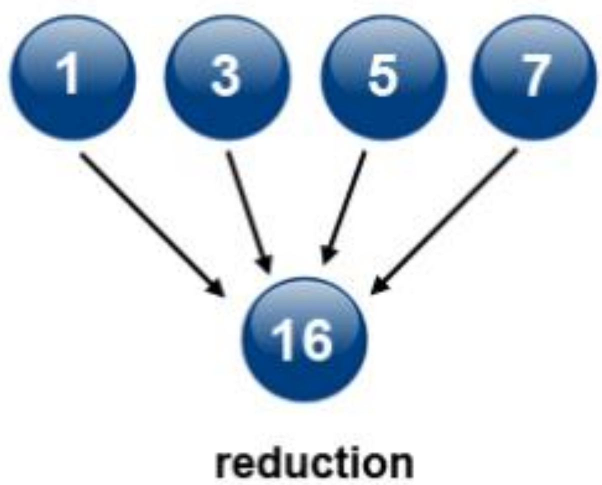
●MPI_Reduce

➢count 参数大于1，MPI_Reduce可以在数组上操作

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

3.集体通信（Collective communication）

●MPI_Reduce



3.集体通信（Collective communication）

- MPI_Reduce

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.集体通信（Collective communication）

- 集体通信 vs. 点对点通信
  - 通信器中的所有进程必须调用同一 collective 通信函数
    - 例如：一个进程中调用 MPI_Reduce，另一个进程中调用 MPI_Recv，将会产生错误
  - 每个进程传递给集体通信的参数必须“兼容”
    - 例如：如果一个进程传递 0 作为 dest_process 而另一个进程传递 1，则 MPI_Reduce 的结果是错误的
  - output_data_p 参数只在 dest_process 中使用。其他进程也需传递实参给 output_data_p，即使只传递 NULL

3.集体通信（Collective communication）

- 集体通信 vs. 点对点通信
  - 点对点通信通过 tag 和通信器匹配
  - 集体通信不使用 tag
  - 它们仅通过通信器和调用的顺序来匹配

3.集体通信（Collective communication）

●集体通信 vs. 点对点通信

- ➢ 每个进程以运算符 MPI_SUM 调用 MPI_Reduce，目的进程为进程 0
- ➢ ***b = 3，d = 6？***

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, ...)`

3.集体通信（Collective communication）

●集体通信 vs. 点对点通信

- 内存位置的名称与匹配 MPI_Reduce 调用无关
- 调用的顺序决定了 MPI_Reduce 的匹配
- 因此，b = 1+2+1 = 4，d = 2+1+2 = 5

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, ...)`

3.集体通信（Collective communication）

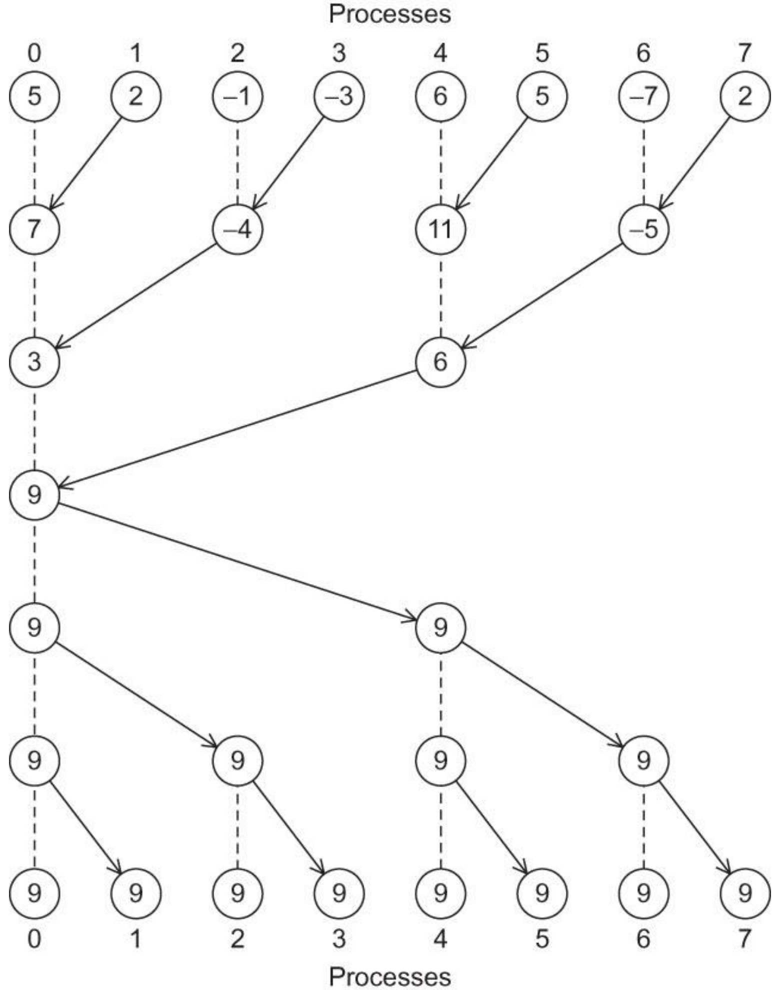
- 集体通信 vs. 点对点通信
  - 输入和输出使用同一缓冲区，为非法调用，结果不可预测

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

3.集体通信（Collective communication）

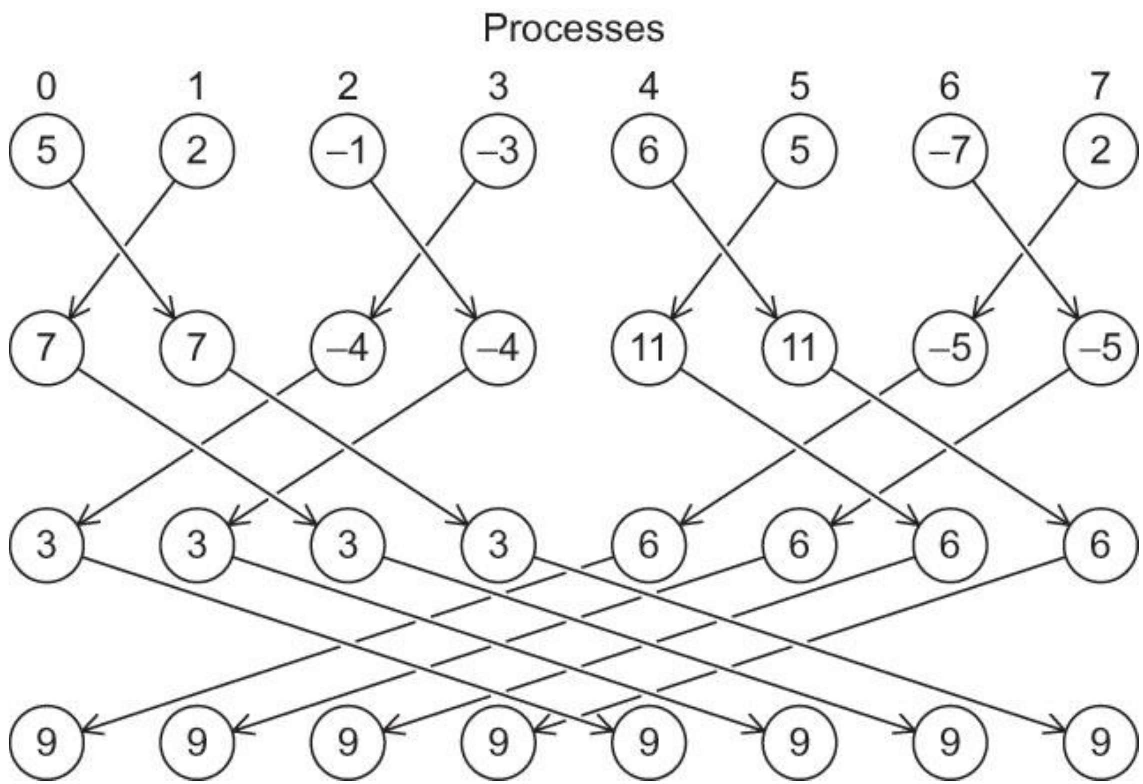
- MPI_Allreduce
  - 当所有的进程都需要计算结果（如：sum）以进行后续的计算

A global sum followed by distribution of the result.



3.集体通信（Collective communication）

- MPI_Allreduce



A butterfly-structured global sum.

3.集体通信（Collective communication）

●MPI_Allreduce

- 同样的，我们不希望去决定使用哪种通信结构，如何去优化代码
- MPI 提供了 MPI_Reduce 的变种，在所有进程中存放结果

```
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   */);
```

3.集体通信（Collective communication）

- *有哪些可以改进之处？*
  - 如果可以通过树形通信的方式优化求和，那么应该同样适用于分发输入数据
  - 广播功能

3.集体通信（Collective communication）

●广播（Broadcast）

- 将属于一个进程的数据发送到通信器中的所有进程

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

- source_proc 进程将 data_p 指向的内容发送给 comm 通信器中的所有进程

3.集体通信（Collective communication）

●广播（Broadcast）

```
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.集体通信（Collective communication）

●数据分发（Data distributions）

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

Compute a vector sum.

3.集体通信（Collective communication）

●数据分发（Data distributions）

```
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 */
```

串行代码

3.集体通信（Collective communication）

●数据分发（Data distributions）

➢并行化

- 对应元素相加
- 任务之间不需要通信
- 映射到核心（core）：如何划分?
  - 块划分（block partition）
  - 循环划分（cyclic partition）
  - 块-循环划分（block-cyclic partition）

3.集体通信（Collective communication）

● 数据分发（Data distributions）

➢ 并行化

- 将12维向量映射到3个进程的不同划分

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

3.集体通信（Collective communication）

● 数据分发（Data distributions）

➢ 并行化

```
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.集体通信（Collective communication）

●分发（Scatter）

- *进程0 读入向量x 和y，广播给其他进程?*
  - 如果10个进程，向量为10000维
  - 每个进程为向量分配10000维空间，仅操作其中的1000维
  - 如果采用块划分，我们希望进程0仅发送1000-1999维给进程1,2000-2999给进程2，……

3.集体通信（Collective communication）

●分发（Scatter）

- MPI_Scatter 可以用来在进程 0 读入整个向量，向其他进程发送它们所需的数据

```
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   */);
```

3.集体通信（Collective communication）

●分发（Scatter）

- 如果通信器 comm 包含 comm_sz 个进程，则 MPI_Scatter 将 send_buf_p 指向数据分成 comm_sz 份，第一份发给进程 0，第二份发给进程 1，以此类推
- 其他进程将本地向量作为 rece_buf_p 参数，将 local_n 作为 recv_count 参数

```
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  */);
```

3.集体通信（Collective communication）

●分发（Scatter）

- ➢假设使用块划分，进程 0 读入向量的所有 n 维数据到 send_buf_p 中
- ➢进程 0 获得第一块 local_n = n / comm_sz 维数据，进程1获得下一块 local_n
- ➢每个进程传递本地向量作为 recv_buf_p 参数，recv_count 应为 local_n
- ➢send_count 应为发送给每个进程的数据量，即：local_n

```
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   */);
```

3.集体通信（Collective communication）

```
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.集体通信（Collective communication）

●收集（Gather）

➢进程 0 收集向量的所有分量进行后续处理

```
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  */);
```

3.集体通信（Collective communication）

●收集（Gather）

- ➢进程 0 中由 send_buf_p 引用的数据存放在 recv_buf_p 中第一块位置，进程 1 中由 send_buf_p 引用的数据存放在 recv_buf_p 中第二块位置，以此类推
- ➢ recv_count 为从每个进程接收到的数据量

```
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   */);
```

3.集体通信（Collective communication）

- 收集（Gather）

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

   double* b = NULL;
   int i;

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

3.集体通信（Collective communication）

● 全收集（Allgather）：矩阵与向量相乘

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

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

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

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

i-th component of y

Dot product of the i-th row of A with x.

3.集体通信（Collective communication）

●全收集（Allgather）：矩阵与向量相乘

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

3.集体通信（Collective communication）

●全收集（Allgather）：矩阵与向量相乘

```
/* 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];
}
```

Serial pseudo-code

3.集体通信（Collective communication）

●全收集（Allgather）：矩阵与向量相乘

C style arrays

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

stored as

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

3.集体通信（Collective communication）

●全收集（Allgather）：矩阵与向量相乘

```
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 */
```

3.集体通信（Collective communication）

●全收集（Allgather）：矩阵与向量相乘

➢并行化

- y[i] += A[i * n + j] * x[j];
- 将 A 按行划分
- y[i] 的计算包含了 A 中第 i 行的所有元素和 x 中的所有元素

3.集体通信（Collective communication）

- 全收集（Allgather）：矩阵与向量相乘
  - 并行化
    - 为减少通信，将 x 中的所有元素分配给每个进程
    - 实际应用中，矩阵与向量相乘的结果 y 可能是下一次相乘的输入
    - 如何使得每个进程获得相乘的结果？
    - MPI_Gather + MPI_Bcast ？

3.集体通信（Collective communication）

● 全收集（Allgather）：矩阵与向量相乘

➢ 并行化

- 将每个进程的 send_buf_p 的内容拼接起来存放到每个进程的 recv_buf_p
- recv_count 为从每个进程接收到的数据量

```
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  */);
```

3.集体通信（Collective communication）

●全收集（Allgather）：矩阵与向量相乘

➢并行化

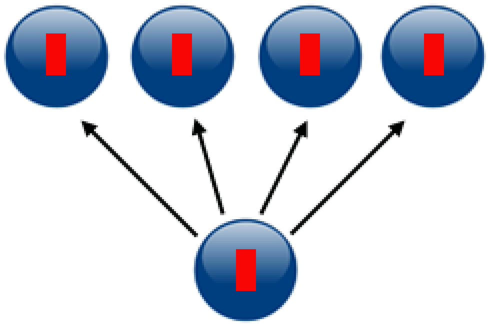
```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* 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 */
```

3.集体通信（Collective communication）

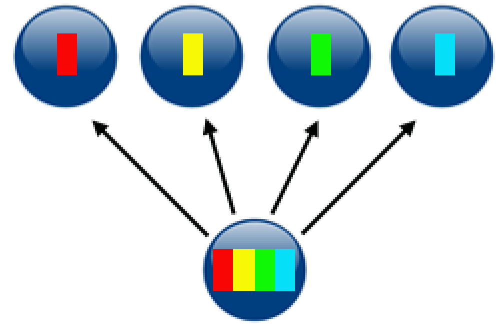
●集体操作的类型

- 同步（Synchronization）：进程等待组内其他进程到达同步点
- 数据移动（Data Movement）：broadcast, scatter/gather
- 集体计算（Collective Computation）：一个进程收集其他进程的数据，并对该数据进行操作（min, max, add, multiply 等）

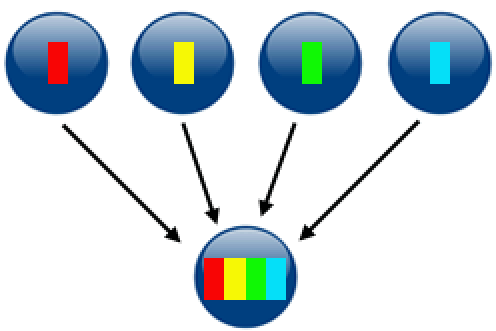
3.集体通信（Collective communication）



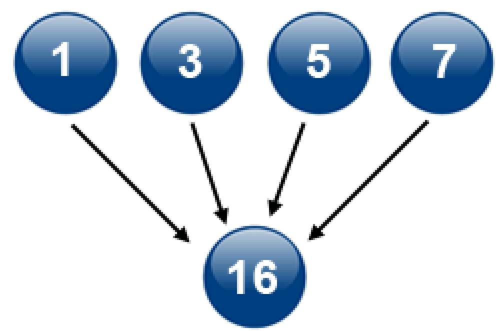
broadcast



scatter



gather



reduction

小结

●梯形法求面积

- 累加局部和操作采用树形通信 -> MPI_Reduce, MPI_Allreduce
- 进程0读入数据后分发给其他进程 -> MPI_Bcast

●矩阵相加

- 数据分发 -> MPI_Scatter
- 数据收集 -> MPI_Gather

●矩阵与向量相乘

- 结果发送给所有进程进行后续计算 -> MPI_Allgather

学习资料

- MPI Tutorial
  - https://computing.llnl.gov/tutorials/mpi/
- MPI API
  - https://www.mpich.org//static/docs/latest/www3/index.htm