主要内容

1. MPI简介及环境搭建
2. MPI点对点通信
3. **MPI集合通信**
4. MPI 派生数据类型
5. MPI程序的性能评估
6. MPI实例——并行排序算法
7. 作业

MPI集合通信

Collective communication

岳俊宏

E-mail:yuejunhong@tyut.edu.cn; Tel:18234095983

目录

集合通信概述

集合通信概述

Collective communication：

集合通信；组通信；群通信

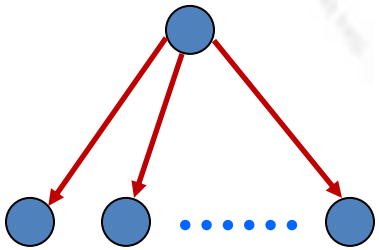
集合通信一般实现三个功能：

- 通信
- 同步
- 计算

通信

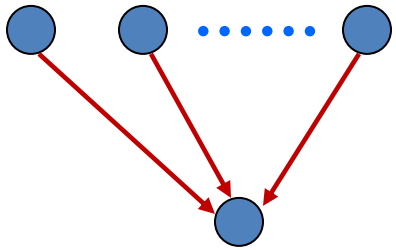
一对一

按通信方向的不同，集合通信可分为三种



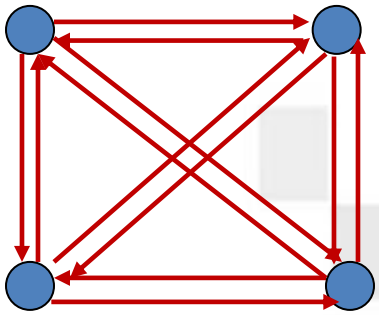
一对多

广播



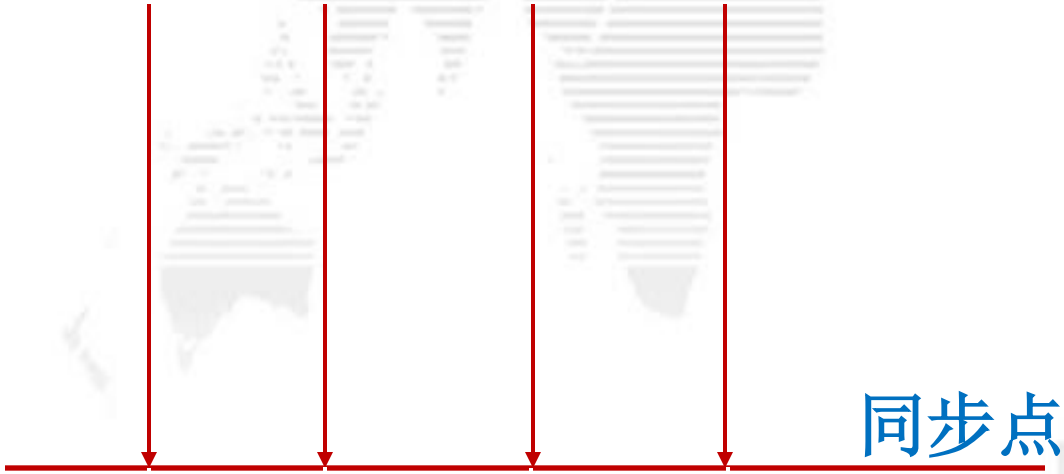
多对一

收集



多对多

同步



快的进程必须等待慢的进程，直到所有进程都执行到该语句后才可以向下进行

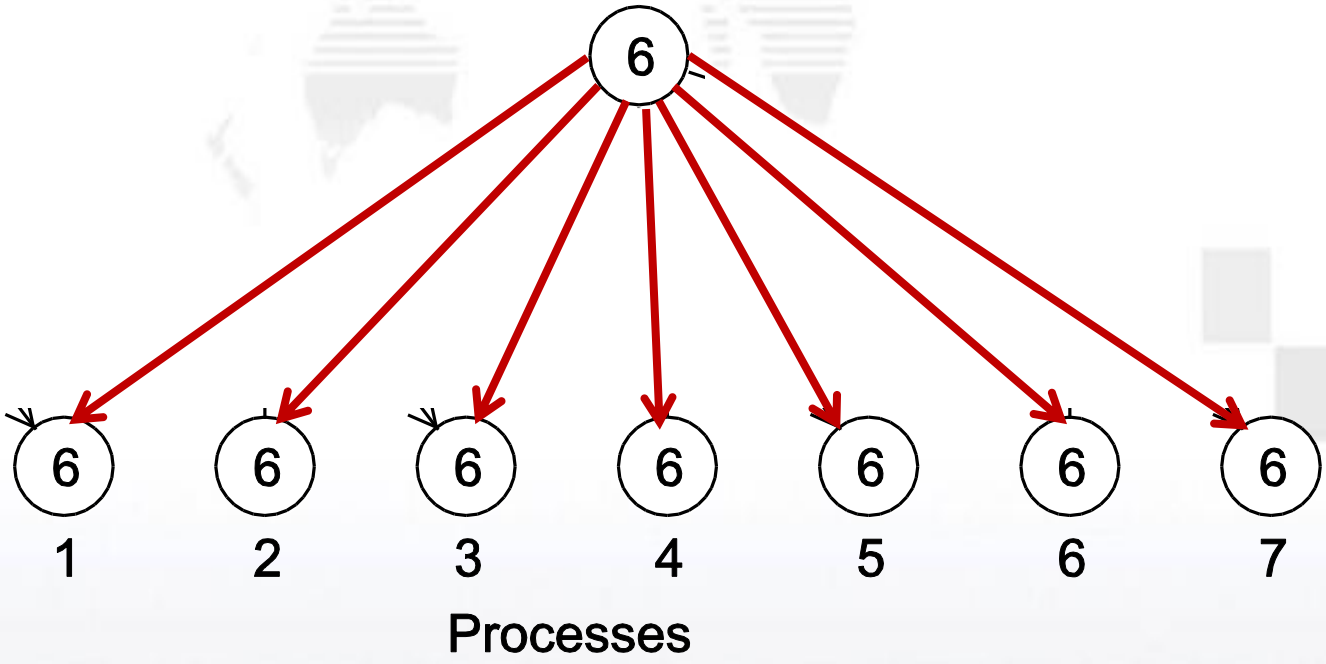
进程完成同步调用后，所有的进程都已执行同步点前面的操作

计算

MPI集合通信就是消息的处理，即计算部分

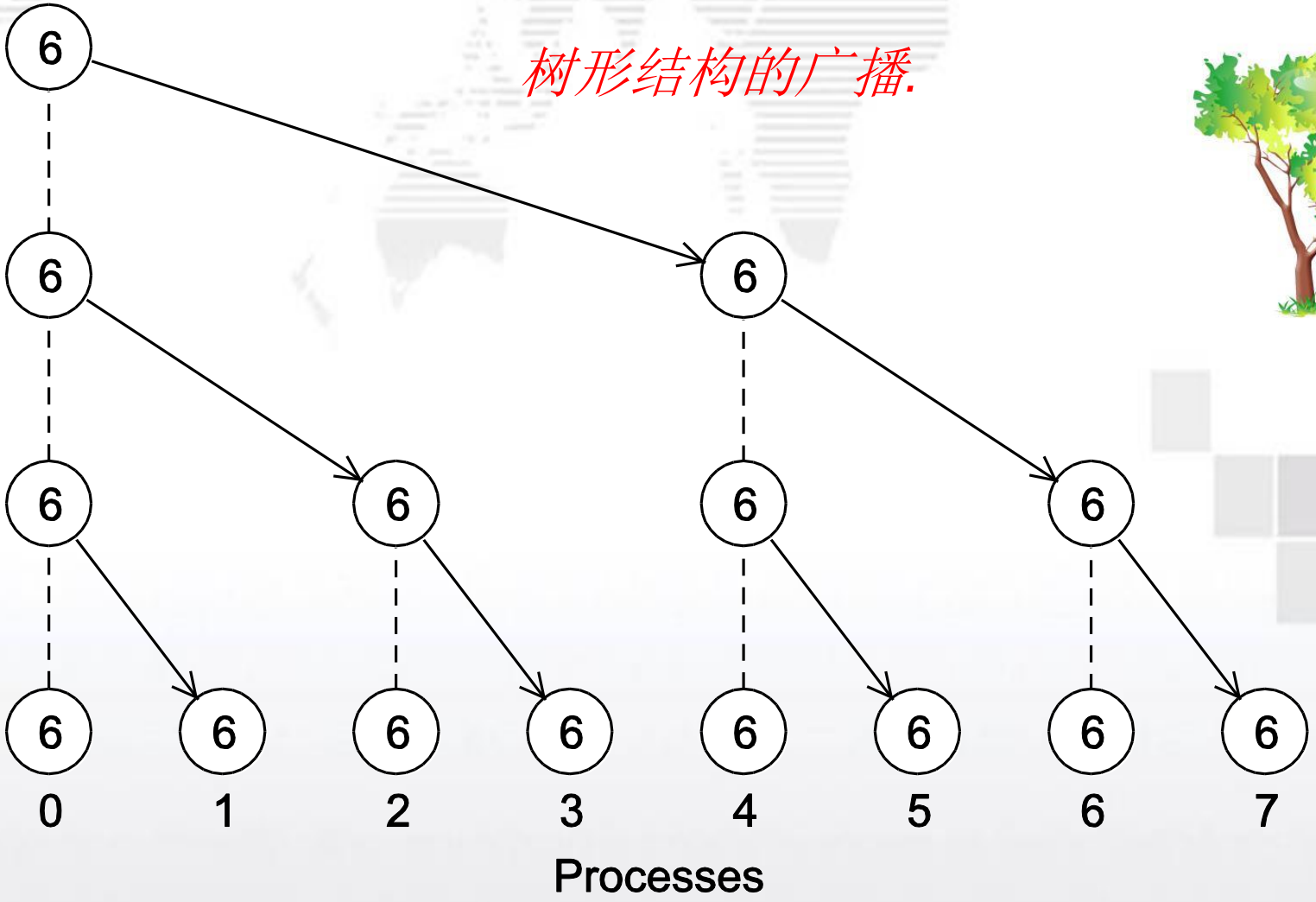

通信方式1
广播规约

通信——广播（一对多）



通信——广播（一对多）

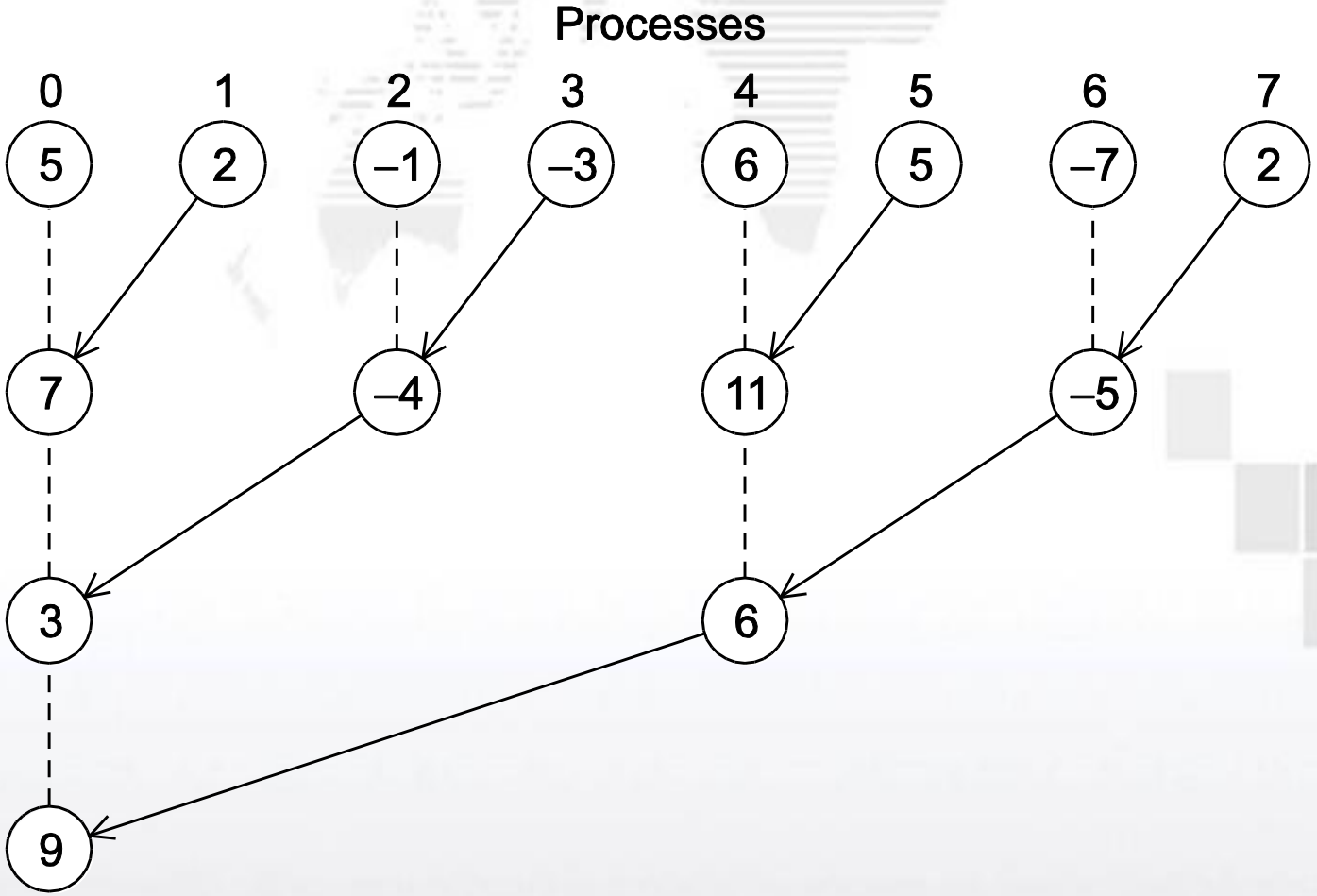
树形结构的广播.



通信——广播（一对多）

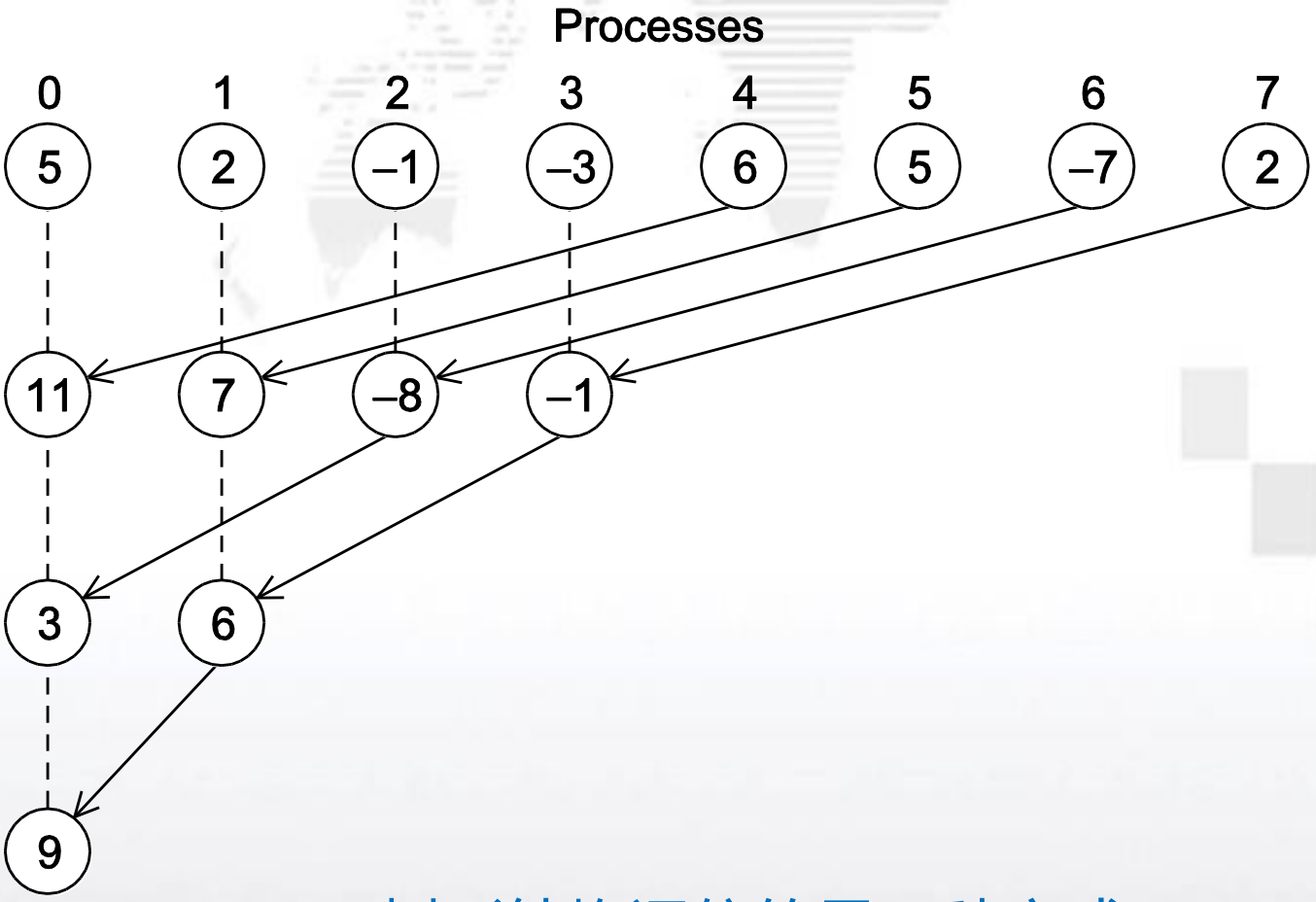
树形结构的广播.

通信——规约（多对一）



树形结构通信

通信——规约（多对一）



树形结构通信的另一种方式

思考

如何更好的实现树形结构通信?

通信——广播（一对多）

一个进程的数据被发送到通信子中的所有进程，即广播

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

通信——广播（一对多）

MPI_Bcast（buffer,count,datatype,root,comm）

buffer 通信消息缓冲区的起始地址

count 将广播出去/或接收的数据个数

datatype 广播/接收数据的数据类型

root 广播数据根进程的标识号

comm 通信子

举例——改进输入数据函数

```
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 */
```

举例——改进输入数据函数

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

   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 */
```

通信——规约（多对一）

除完成消息传递功能之外，还能对所传递的数据进行一定的操作或运算

在进行通信的同时完成一定的计算

MPI_Reduce函数

```
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函数

MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)

sendbuf 发送消息缓冲区的起始地址

recvbuf 接收消息缓冲区的起始地址

count 发送消息缓冲区中的数据个数

datatype 发送消息缓冲区的元素类型

op 归约操作符

root 根进程号

comm 通信子

功能：将组内每个进程输入缓冲区中的数据按给定的操作op进行运算，并将其结果返回到序列号为root进程的输出缓冲区中。

MPI_Reduce调用

MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)

输入缓冲区由参数sendbuf、 count和datatype定义;

输出缓冲区由参数recvbuf、 count和datatype定义;

所有进程输入和输出缓冲区的长度、元素类型相同.

各操作允许的数据类型

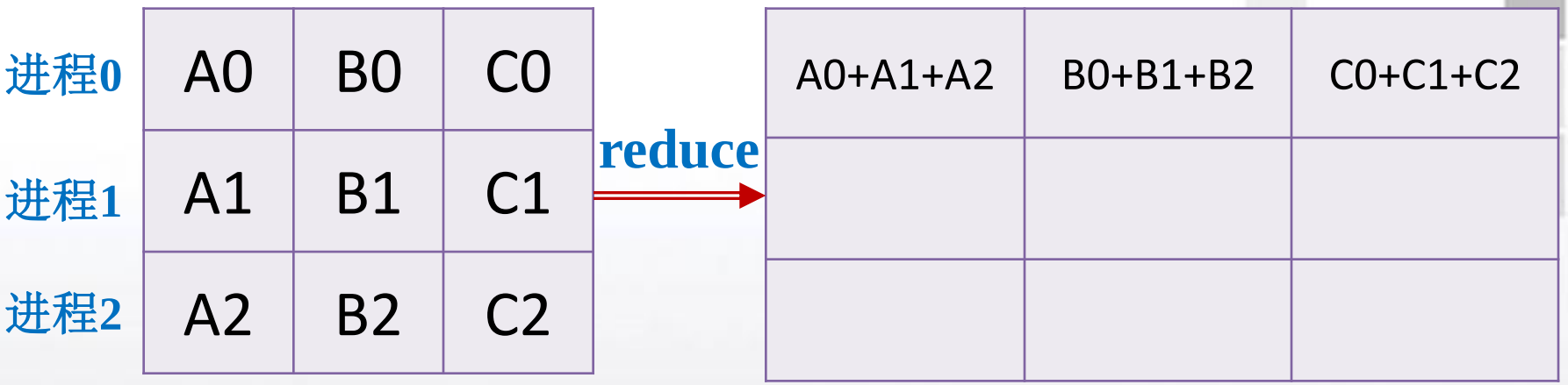
MPI 规约操作		C 语言数据类型	Fortran 语言数据类型
MPI_MAX	求最大值	int, float	integer, real, complex
MPI_MIN	求最小值	int, float	integer, real, complex
MPI_SUM	和	int, float	integer, real, complex
MPI_PROD	乘积	int, float	integer, real, complex
MPI_LAND	逻辑与	int	logical
MPI_BAND	按位与	int, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	逻辑或	int	logical
MPI_BOR	按位或	int, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	逻辑异或	int	logical
MPI_BXOR	按位异或	int, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	最大值和存储单元	float, double long double	real,complex,double precision
MPI_MINLOC	最小值和存储单元	float, double long double	real,complex,double precision

举例

```
int MPI_Reduce(sendbuf,recvbuf,1,MPI_Int,
               MPI_Sum,1,MPI_Comm_world)
```

举例

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

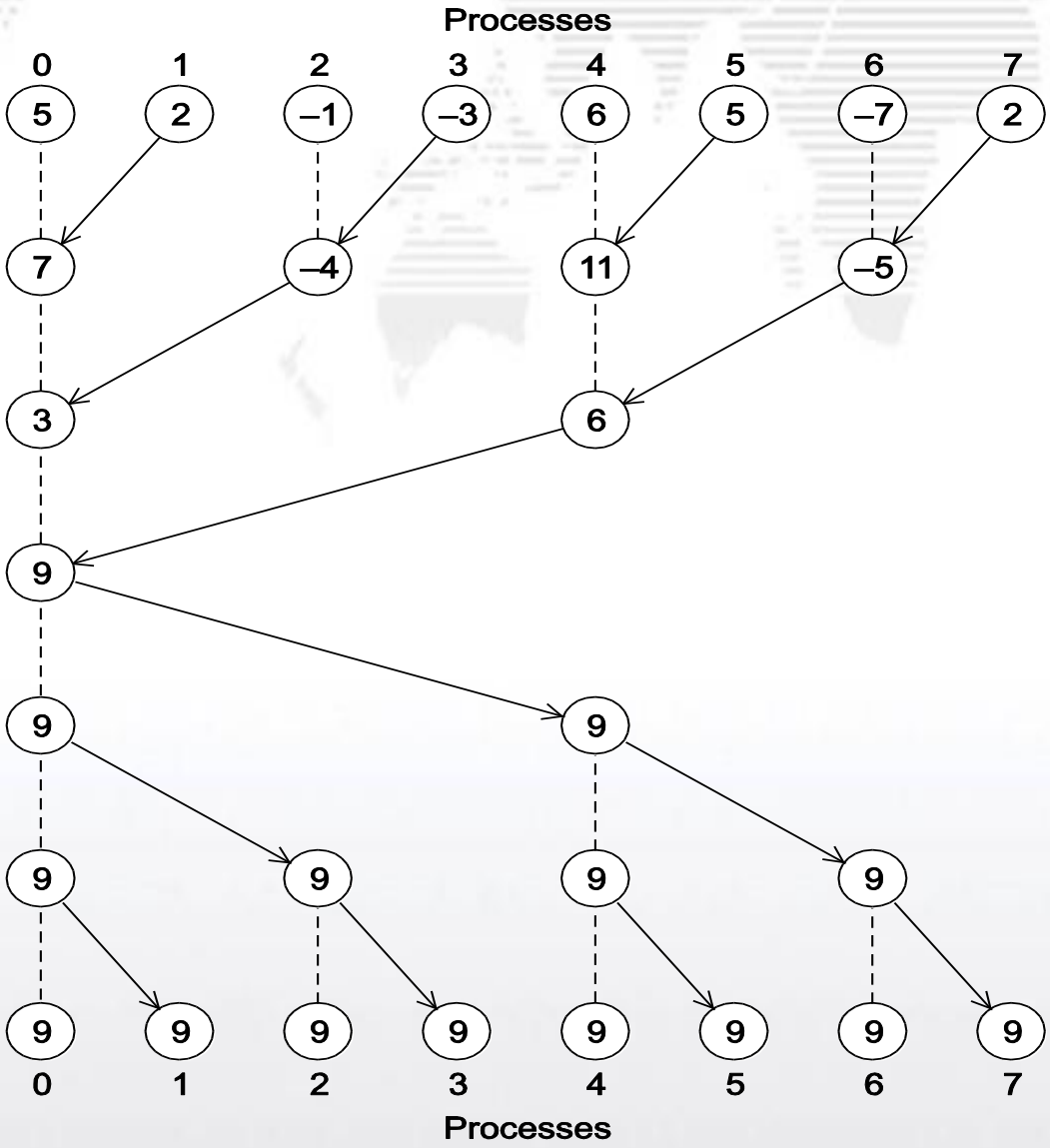


思考

当所有进程都想要全局总和结果怎么办?

规约+广播

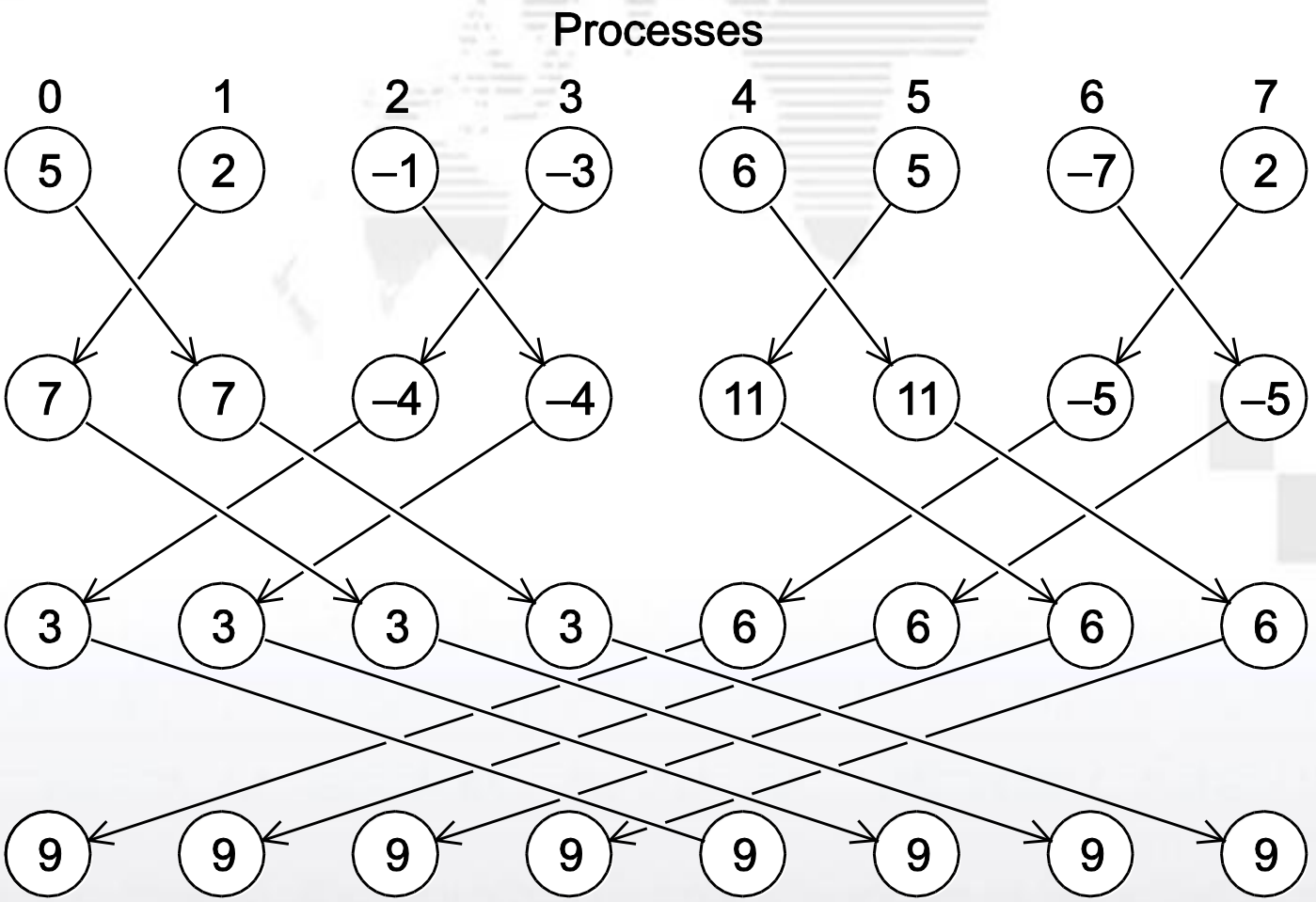
规约+广播



全局求和

结果发布

规约+广播



蝶形结构全局求和

通信——规约—广播（N—N）

```
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函数

- 区别：与MPI_Reduce相比，缺少dest_process这个参数，因为所有进程都能得到结果。
- 通信子内所有进程都作为目标进程执行一次规约操作，操作完毕后所有进程接收缓冲区的数据均相同。这个操作等价于首先进行一次MPI_Reduce,然后再执行一次MPI_Bcast。

思考

集合通信与点对点通信的区别

集合通信 vs.点对点通信

- 所有的进程都必须调用相同的集合通信函数
- For example：

  MPI_Reduce不能用MPI_Recv

集合通信 vs.点对点通信

- 每个进程传送给MPI的集合通信函数的参数必须相容
- For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

集合通信 vs.点对点通信

- 参数recvbuf只用在dest_process上。
- MPI_Reduce(sendbuf,recvbuf,count,datatype,op, dest_process,comm)
- 所有进程仍需要传递一个与recvbuf相对应的实际参数，即使它的值是NULL.

集合通信 vs.点对点通信

- 点对点通信通过标签和通信子匹配，集合通信通过通信子和调用顺序匹配

Example

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, ...)

假设每个进程调用的运算符都是MPI_SUM，目标进程都是0，b和d的值?

思考

1 使用同一缓冲区同时作为输入和输出调用

求得所有进程里x的全局总和，并且将x的结果放在0号进程里：

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

2 如果通信子只包含一个进程，不同的MPI集合通信函数分别会做什么？

上机作业

1 编写使用广播函数的梯形积分法的MPI程序（要求：将点对点通信求和改为规约）<使用MPI_Bcast和MPI_Reduce>

2 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，若能正常运行，改变该程序使其适用于所有comm_sz.

上机作业

编写使用广播函数的梯形积分法的MPI程序（要求：将点对点通信求和改为规约）<使用MPI_Bcast和MPI_Reduce>

将上述作业中MPI_Reduce换成MPI_Allreduce，然后输出结果并对比分析。

结束!


通信方式2
散射聚集

通信——散射（一对多）

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

计算向量求和

通信——散射（一对多）

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

计算向量求和

串行向量求和

```
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 */
```

思考

如何指定各核求和任务的对应分量?

(划分数据)

三种划分方式

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

向量求和的并行实现

```
1  void Paralle_vector_sum(
2       double local_x[] /* in */
3       double local_y[] /* in */
4       double local_z[] /* out */
5       int local_n /* in */){
6       int local_i;
7
8       for(local_i=0;local_i<local_n;local_i++)
9           local_z[local_i]=local_x[local_i]+local_y[local_i];
10      }
```

思考

- 对于**梯形积分**，由**进程0**读取数据(a,b,n)，然后**分发（广播）**给通信子中其它进程。
- 对于**向量加法**，由**进程0**读取向量x和向量y，然后**分发（广播）**给通信子中的其它进程。

当输入数据特别大，并且其实每个进程只有需要一小部分输入数据，并不需要全部。

1万个分量的向量！

散射（1-N）

- MPI_Scatter可以使0号进程读取整个向量，但是只将分量发送给需要分量的其他进程

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

MPI_Scatter函数

MPI_Scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,src_proc,comm)

sendbuf:发送消息缓冲区的起始地址

sendcount: 发送到每个进程的数据个数

sendtype:发送消息缓冲区中的元素类型

recvbuf: 接收消息缓冲区的起始地址

recvcount: 接收到的数据个数

recvtype: 发送消息缓冲区的元素类型

src_proc: 负责发送消息的源进程

comm: 通信子

MPI_Scatter调用

MPI_Scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,src_proc,comm)

发送缓冲区由参数sendbuf,sendcount和sendtype定义;

接收缓冲区由参数recvbuf,recvcount和recvtype定义;

所有进程发送和接收缓冲区的长度、元素类型相同.

功能：将src_proc进程中sendbuf指向的数据按照**块大小为sendcount**的**块划分**的方式发送给通信子中的其它进程。

注意：仅适用于块划分和向量分量个数n可以整除以comm_sz的情况。

举例—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 */
```

思考

如何验证向量加法的结果正确！

打印分布式向量的函数！

聚集（N-1）

- 将向量所有分量都收集到0号进程上，然后由0号进程将所有分量输出

```
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  */);
```

MPI_Gather函数

MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,dest_proc,comm)

sendbuf:发送消息缓冲区的起始地址

sendcount: 发送到每个进程的数据个数

sendtype:发送消息缓冲区中的元素类型

recvbuf: 接收消息缓冲区的起始地址

recvcount: 每个进程接收到的数据个数

recvtype: 发送消息缓冲区的元素类型

dest_proc: 负责发送消息的源进程

comm: 通信子

MPI_Gather函数

MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,dest_proc,comm)

发送缓冲区由参数sendbuf,sendcount和sendtype定义;

接收缓冲区由参数recvbuf,recvcount和recvtype定义;

所有进程发送和接收缓冲区的长度、元素类型相同.

功能：按照进程编号将每个进程中sendbuf指向的数据按照**块**依次存储在dest_proc的recvbuf指向的地址。

注意：仅适用于块划分和每个块大小相同的情况。

举例—输出分布式向量的函数

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

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

举例—输出分布式向量的函数

```
    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 */
```

作业

- 编写一个向量加法的MPI程序
- 3-9；3-10

思考：矩阵—向量乘法

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

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

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

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

i-th component of y

Dot product of the ith row of A with x.

举例——矩阵—向量乘法

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

举例——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];
}
```

矩阵存储

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

stored as

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

串行代码

```
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 */
```

思考

矩阵向量乘法——如何进行并行?

对A的行进行块划分（假设comm_sz整除行数m），每个进程都存储向量x；相应的y也进行行划分。

思考

对Ax=b

$$A=\begin{bmatrix} 0 & & & & \\ a_{21} & 0 & & & \\ a_{31} & a_{32} & 0 & & \\ \vdots & \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & 0 \end{bmatrix}+\begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix}+\begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ & 0 & a_{23} & \cdots & a_{2n} \\ & & 0 & \cdots & a_{3n} \\ & & & \ddots & \vdots \\ & & & & 0 \end{bmatrix}$$

下三角阵　　对角阵　　上三角阵

$$=L+D+U$$

$\because a_{ii}\neq 0$(Jacobi假设)　　$\therefore D$可逆

进一步，有：　$(L+D+U)x=b$

$$Dx=-(L+U)x+b$$

$$x=-D^{-1}\left(L+U\right)x+D^{-1}b$$

思考

- 上述雅克比迭代中，上一步矩阵乘法得到的向量x或y会是下一次迭代中向量x的输入。
- 向量x和y的划分一样，也就是无论是输入x，还是迭代一次得到x，它们都会是分散在各个进程中。
- 对于这样的循环迭代，每个进程在执行如下语句时，就必须使得**每个进程都能访问x中的所有分量**。

```
for(j=0;j<n;j++)
    y[i]+=A[i*n+j]*x[j];
```

全局聚集（N-N）

当所有进程都想要全局结果怎么办?

```
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  */);
```

MPI_Allgather函数

- ✓ 区别：与MPI_Gather相比，缺少dest_process这个参数，因为所有进程都能得到结果。
- ✓ 功能：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。
- ✓ recv_count:每个进程接收的数据个数。
- ✓ send_count=recv_count

并行代码

```
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;
```

并行代码

```
    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 */
```

✓ 该函数如果被调用多次，可以将x作为一个附加的参数传递给调用函数，使得程序性能进一步提高。

扩展

MPI_Reduce_scatter(

const void* sendbuf, // 指向输入数据的指针

void* recvbuf, // 指向接收数据的指针(输出)

const int recvcounts[], // 各进程接收规约结果的元素个数

MPI_Datatype datatype, // 数据类型

MPI_Op op, // 规约操作类型

MPI_Comm comm // 通信子);

功能：首先对各进程所保有的输入向量实施规约操作，再将结果向量散发到各个进程。

扩展

```
MPI_Scan(
const void* sendbuf, // 指向参与规约数据的指针
void* recvbuf, // 指向接收规约结果的指针
int count, // 每个进程中参与规约的数据量
MPI_Datatype datatype, // 数据类型
MPI_Op op, // 规约操作类型
MPI_Comm comm // 通信子);
```

前缀和函数 MPI_Scan()，将通信子内各进程的同一个变量参与前缀规约计算，并将得到的结果发送回每个进程，使用与函数 MPI_Reduce() 相同的操作类型

上机作业

编写一个并行的向量加法程序

编写一个完整的并行矩阵向量乘法程序

1.使用MPI_Gather和MPI_Bcast

2. 使用MPI_Allgather

结束!

作业

编写使用广播函数的梯形积分法的MPI程序（要求：将点对点通信求和改为规约）<使用MPI_Bcast和MPI_Reduce>

将上述作业中MPI_Reduce换成MPI_Allreduce，然后输出结果并对比分析。

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，若能正常运行，改变该程序使其适用于所有comm_sz.

上机作业

编写使用广播函数的梯形积分法的MPI程序

（要求： uses collective communications to distribute the input data and compute the global sum.）