3. MPI点对点通信、广播及例子

罗翔宇
中国人民大学统计与大数据研究院

此课件内容主要基于Blaise Barney的网络资料[Message Passing Interface (MPI)]及Wes Kendall的[MPI Tutorial]

上周课堂复习

- 通信器(communicator)定义了一组能够相互通信的进程。在这一组进程中，每个进程有一个唯一的编号(rank)，它们通过这个rank来进行交流

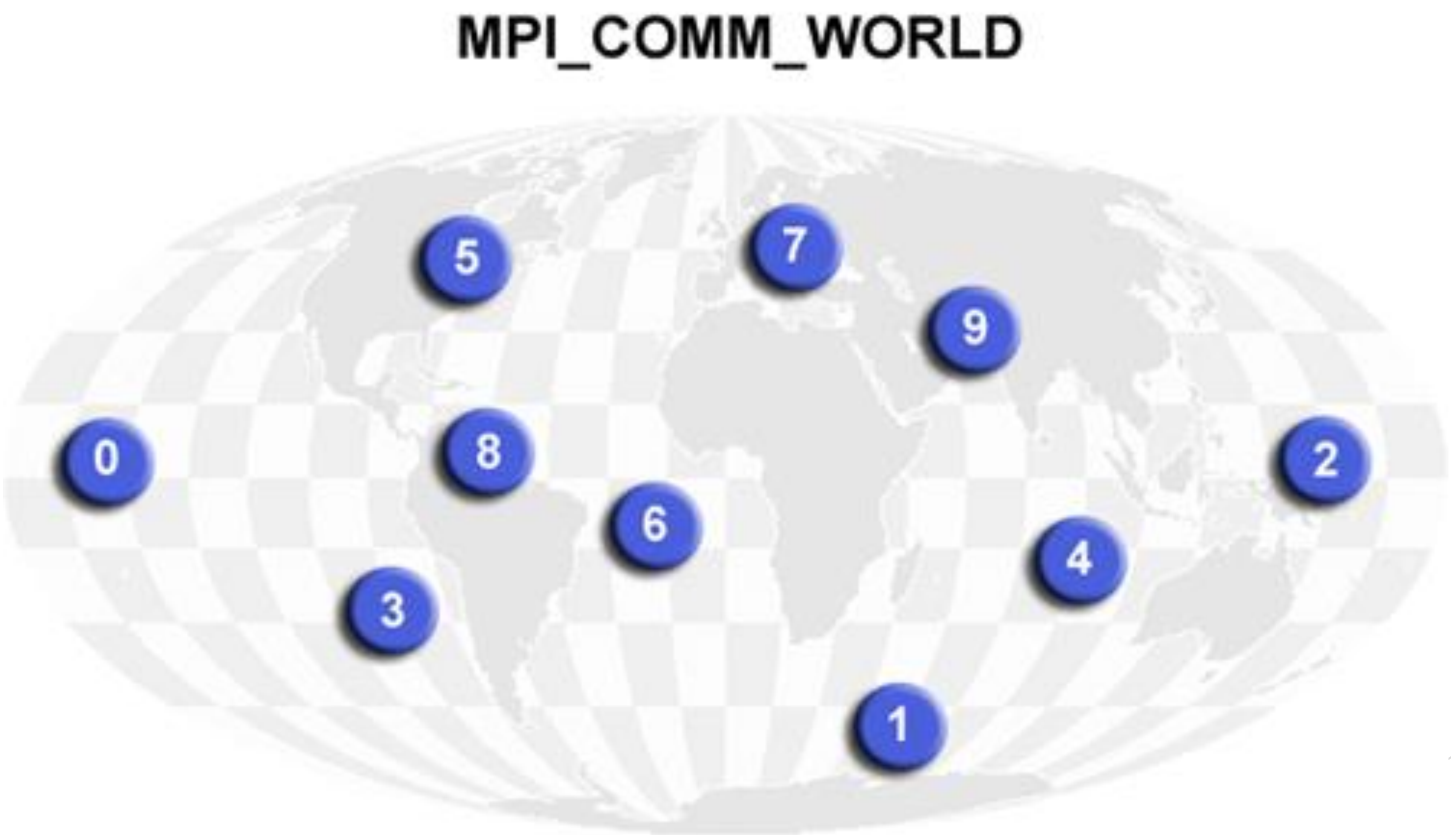


Figure credit: https://computing.llnl.gov/tutorials/mpi/

上周课堂复习

- 获取进程总个数
- int world_size;
- MPI_Comm_size(MPI_COMM_WORLD, &world_size);

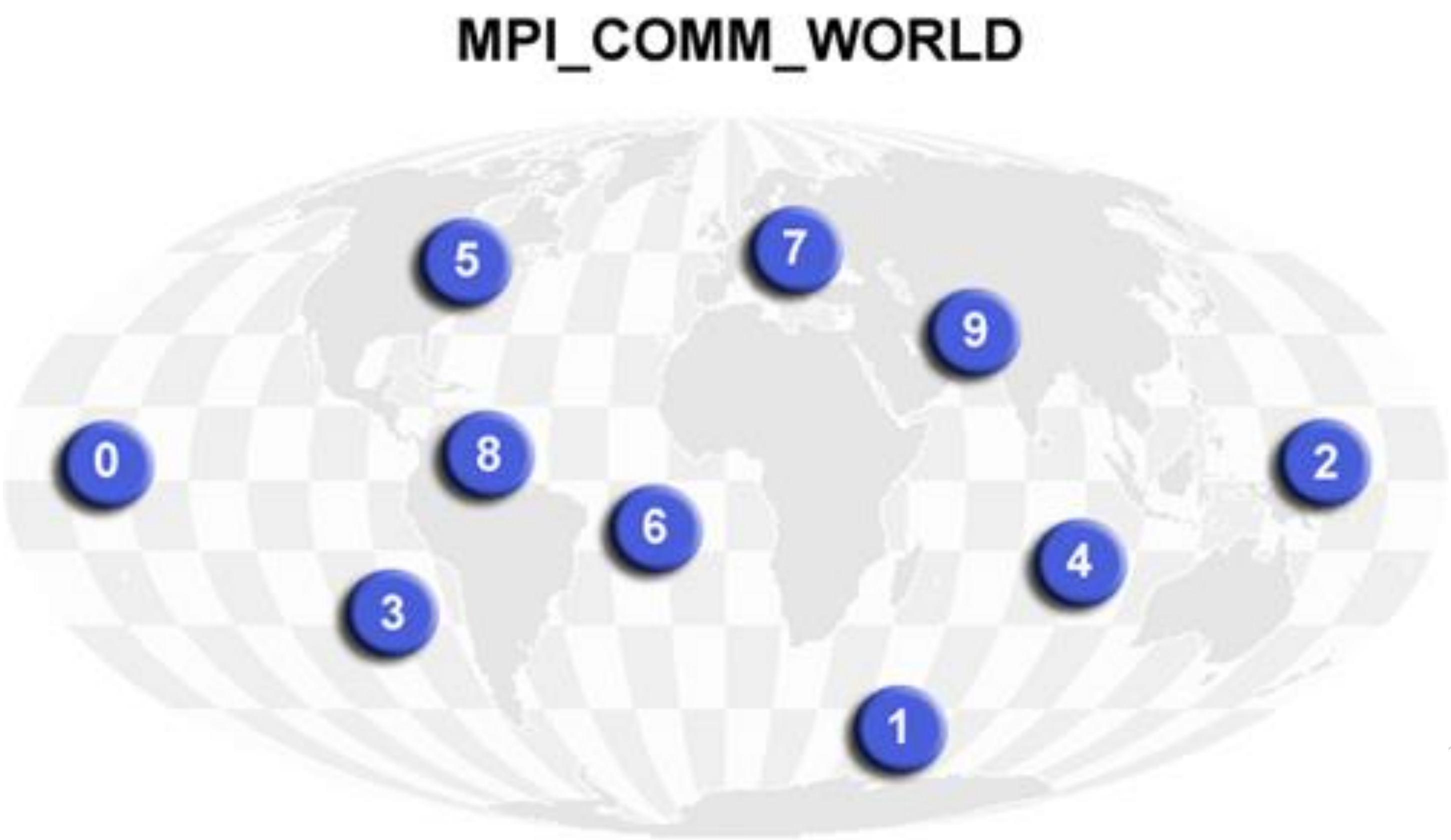


Figure credit: https://computing.llnl.gov/tutorials/mpi/

上周课堂复习

- 获取每个进程的编号(rank)
- int world_rank;
- MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

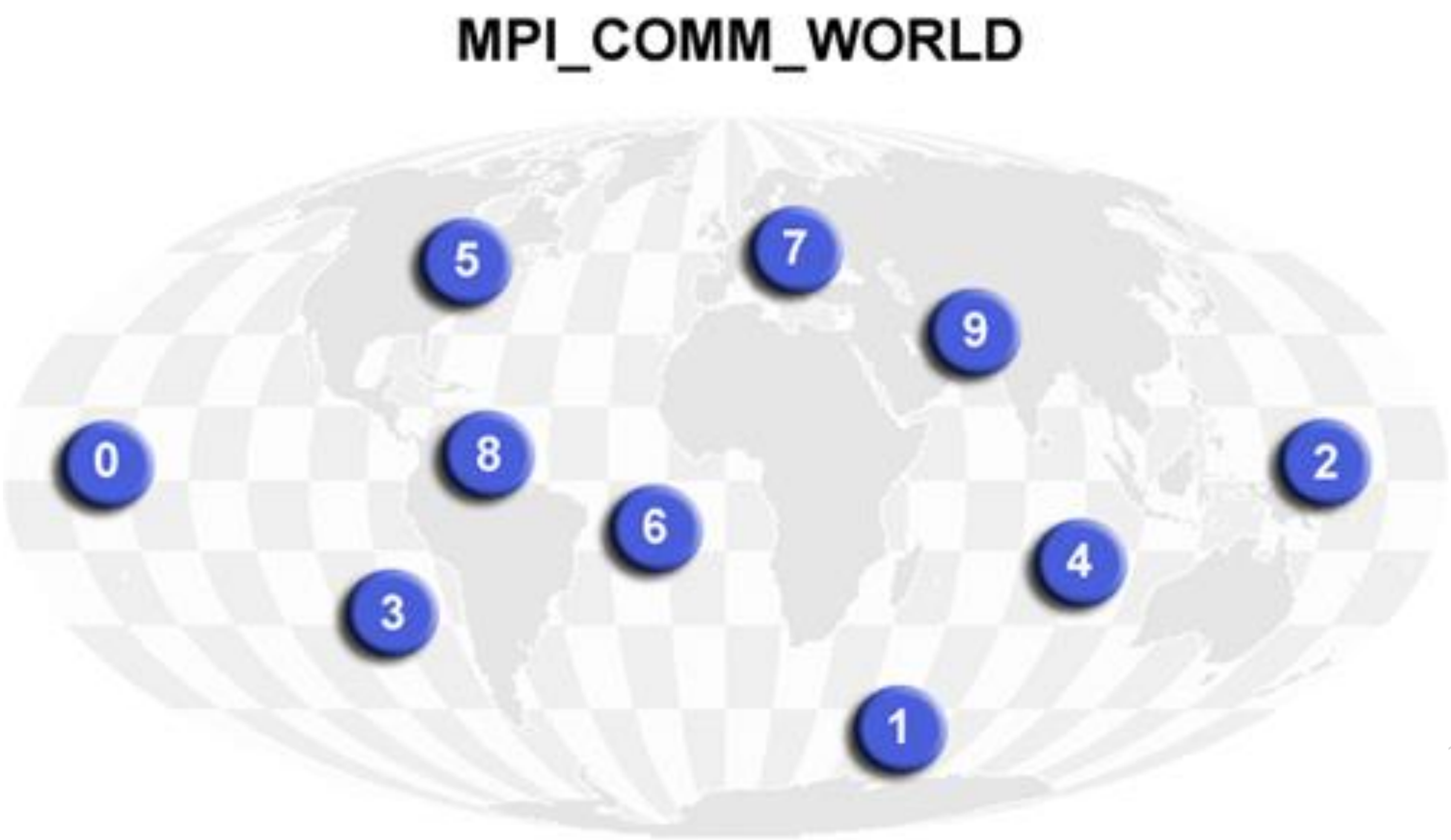


Figure credit: https://computing.llnl.gov/tutorials/mpi/

上周课堂复习

- MPI的第一个程序：MPI Hello World!

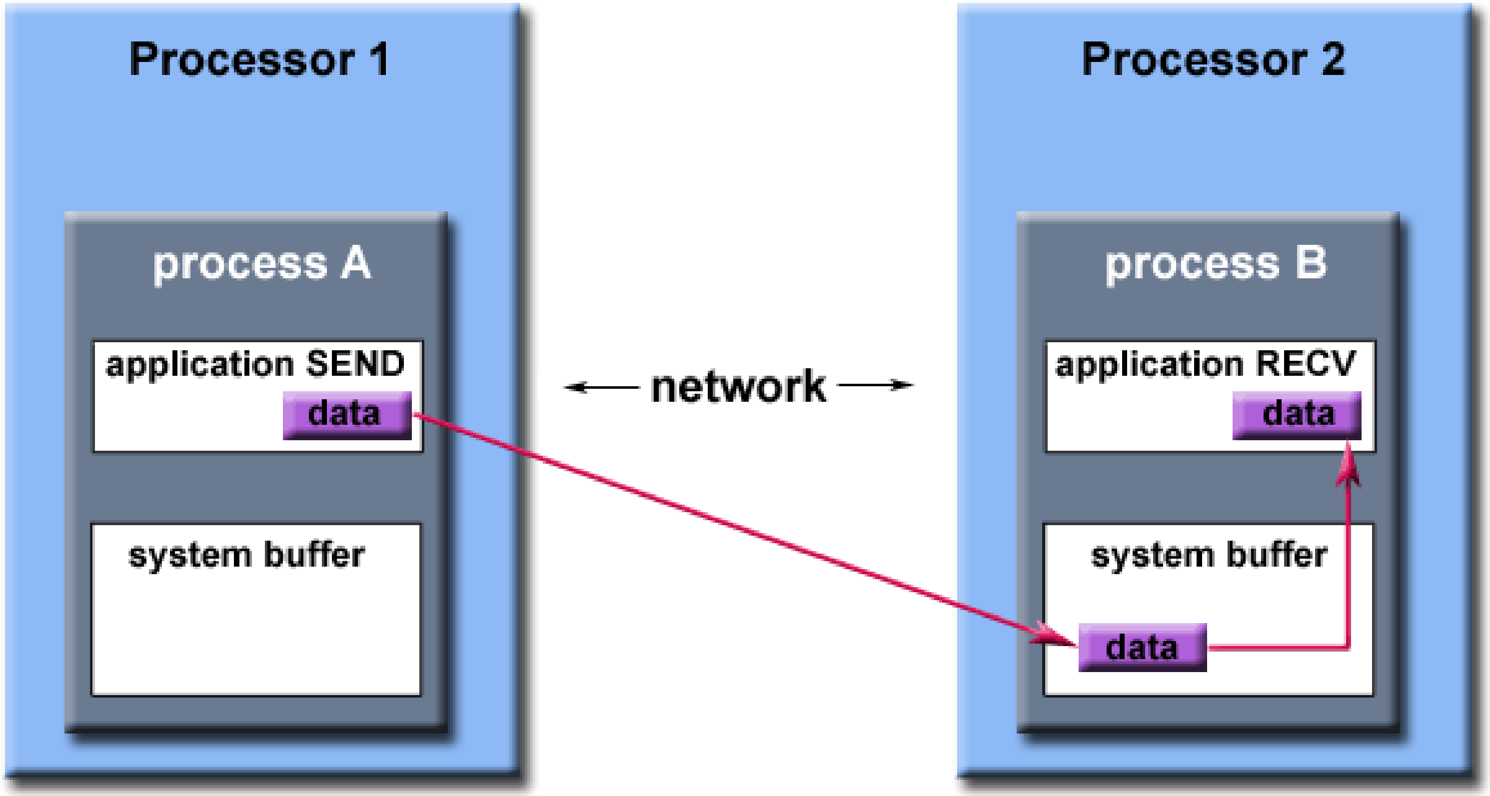
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        char processor_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(processor_name, &name_len);

        printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size);

        MPI_Finalize();
}
```

上周课堂复习

- MPI编译命令：
- mpicc –o mpi_hello_world mpi_hello_world.c
- MPI执行命令(调用3个进程)：
- mpirun –np 3 ./mpi_hello_world

上周课堂复习

- MPI阻塞式发送、阻塞式接收



Path of a message buffered at the receiving process

上周课堂复习

- 阻塞式点对点通信函数
- int MPI_Send(void* buf, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)
- int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)

上周课堂复习

- MPI中的数据类型

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	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_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

上周课堂复习

- MPI 传送/接收程序：进程0将数字-1传输给进程1

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

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if(world_size < 2){
                printf("The number of arguments is %d\n", argc);
                printf("The seconde argument is %s\n", argv[1]);
                fprintf(stderr, "World size must be greater than 1 for %s\n",\
                        argv[0]);

                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        int number;

        if(world_rank == 0){
                number = -1;
                MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        }else if(world_rank == 1){
                MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, \
                        MPI_STATUS_IGNORE);
               printf("Process 1 received number %d from process 0\n",\
                        number);
       }

       MPI_Finalize();
}
```

上周课堂复习

- MPI 传送/接收程序：进程0将数字-1传输给进程1

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 1 ./send_recv
The number of arguments is 1
The seconde argument is (null)
World size must be greater than 1 for ./send_recv
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 1 ./send_recv
 test
The number of arguments is 2
The seconde argument is test
World size must be greater than 1 for ./send_recv
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 2 ./send_recv
Process 1 received number -1 from process 0
```

MPI是否加快了运算速度?

- 例子：1+2+...+10亿
- 串行，并行(利用MPI，调用两个进程，分别计算奇数和、偶数和)
- 先在一个**双核**电脑上试试

型号名称：	MacBook Pro
型号标识符：	MacBookPro14,2
处理器名称：	Dual-Core Intel Core i5
处理器速度：	3.1 GHz
处理器数目：	1
核总数：	2
L2缓存（每个核）：	256 KB
L3缓存：	4 MB
超线程技术：	已启用
内存：	8 GB

MPI是否加快了运算速度?

- 串行、并行结果

```
mackies-MacBook-Pro:examp mackie$ ./seq_comp
The summation is 5000000050000000
Sequential computing cost 1.621922 seconds
mackies-MacBook-Pro:examp mackie$ mpirun -np 1 ./para_comp
The summation is 5000000050000000
Parallel computing cost 1.615748 seconds
mackies-MacBook-Pro:examp mackie$ mpirun -np 2 ./para_comp
The summation is 5000000050000000
Parallel computing cost 0.825165 seconds
```

MPI是否加快了运算速度？

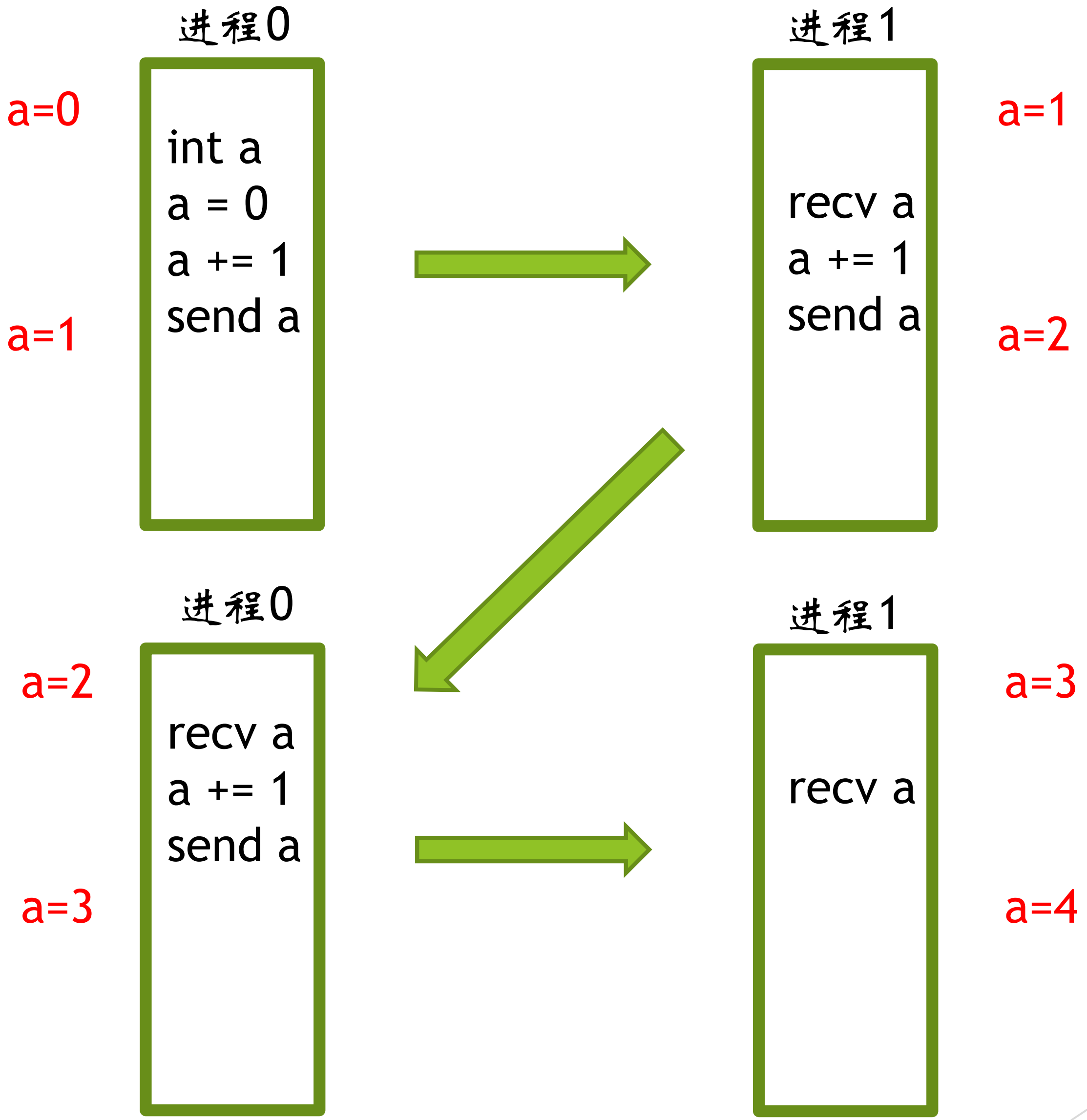
- 若在一个单核电脑上呢？

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files/hw2$ cat /proc/cpuinfo |
grep "cpu cores" | uniq
cpu cores       : 1
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files/hw2$ ./seq_comp
The summation is 500000000500000000
Sequential computing cost 3.296579 seconds
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files/hw2$ mpirun -np 1 ./para_
comp
The summation is 500000000500000000
Parallel computing cost 3.073138 seconds
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files/hw2$ mpirun -np 2 ./para_
comp
The summation is 500000000500000000
Parallel computing cost 3.073321 seconds
```

- 调用进程的个数并不是越多越好，而依赖于个人计算机的核的数目

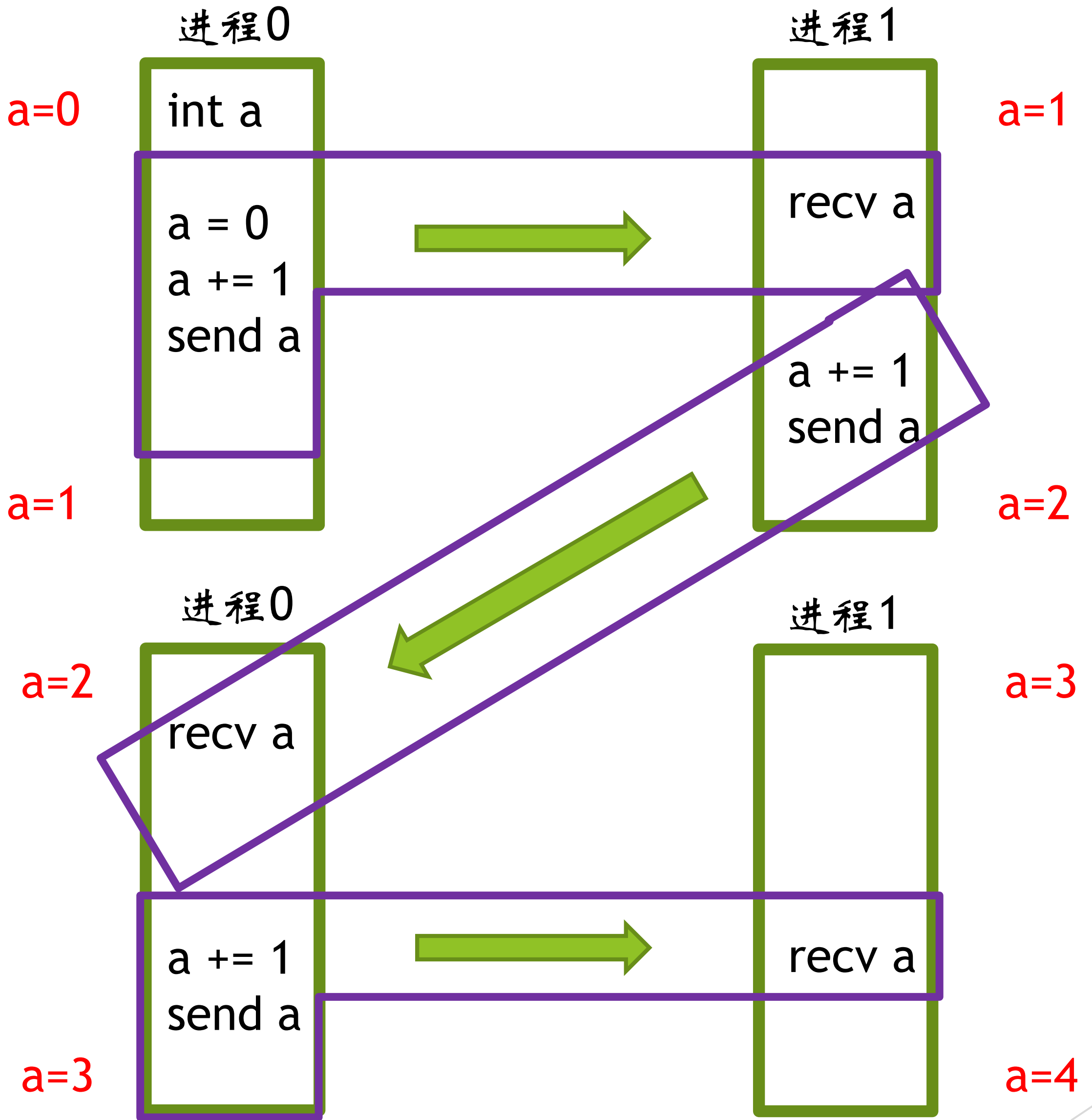
MPI乒乓程序

- 进程0将“乒乓”(数字0加一)传给进程1，进程1接收到“乒乓”后，自加1再传给进程0,重复此过程，直到乒乓到达10.



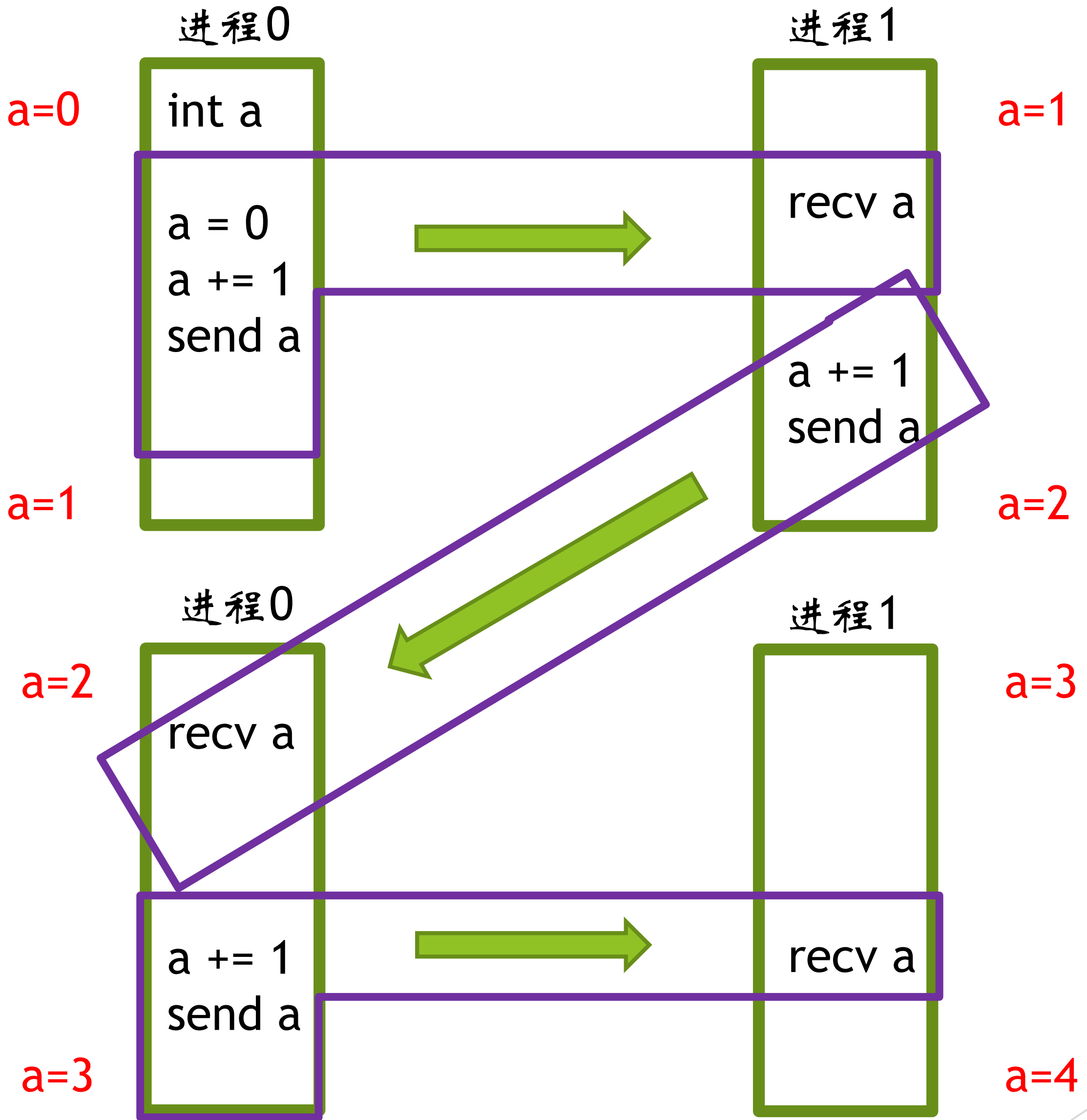
MPI乒乓程序

- 设计算法：一开始，若a为偶数，进程0：让a自加1，发送；进程1：接收a. 若a为奇数，进程1：让a自加1，发送；进程0：接收a.



MPI乒乓程序

- 设计算法：换句话说，让进程编号为a除以2的余数：a自加1，发送a；另一个进程：接收a.



MPI乒乓代码

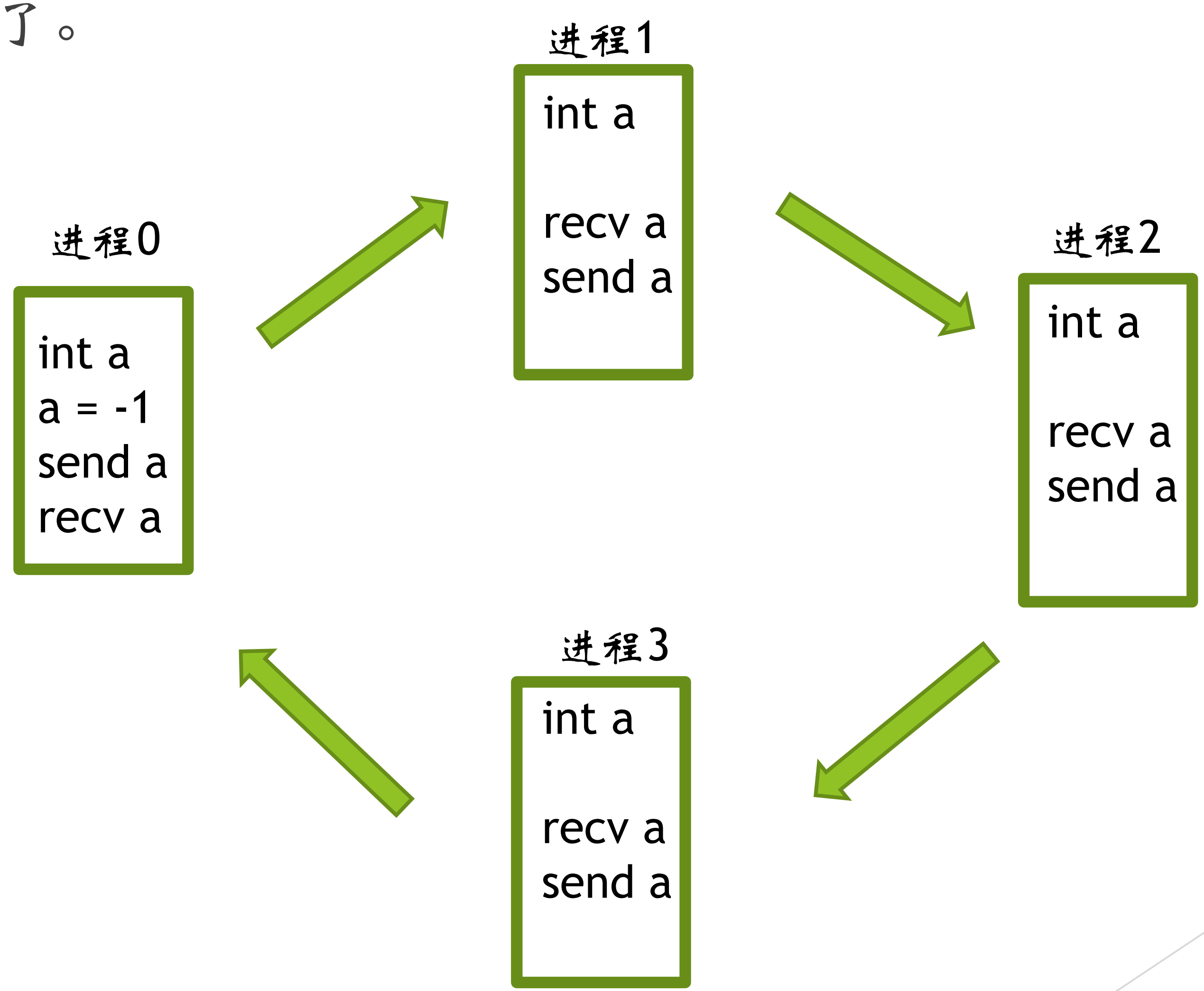
```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
        const int PING_PONG_LIMIT = 10;
        MPI_Init(NULL, NULL);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if(world_size != 2){
                fprintf(stderr, "World size must be two for %s\n", argv[0]);
                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        int ping_pong_count = 0;
        int partner_rank = (world_rank + 1) % 2;
        while(ping_pong_count < PING_PONG_LIMIT){
                if(world_rank == ping_pong_count % 2){
                        ping_pong_count++;
                        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0\
                                , MPI_COMM_WORLD);
                        printf("%d sent and incremented ping_pong_count %d to %
d\n", world_rank, ping_pong_count, \
                                partner_rank);
                }else{
                        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank,\
                                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        printf("%d received ping_pong_count %d from %d\n",\
                                world_rank, ping_pong_count, partner_rank);
                }
        }

        MPI_Finalize();
}
```

MPI乒乓代码输出结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o ping_pong ping_
pong.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 3 ./ping_pong
World size must be two for ./ping_pong
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 2 ./ping_pong

0 sent and incremented ping_pong_count 1 to 1
1 received ping_pong_count 1 from 0
1 sent and incremented ping_pong_count 2 to 0
0 received ping_pong_count 2 from 1
0 sent and incremented ping_pong_count 3 to 1
1 received ping_pong_count 3 from 0
1 sent and incremented ping_pong_count 4 to 0
0 received ping_pong_count 4 from 1
0 sent and incremented ping_pong_count 5 to 1
1 received ping_pong_count 5 from 0
1 sent and incremented ping_pong_count 6 to 0
0 received ping_pong_count 6 from 1
0 sent and incremented ping_pong_count 7 to 1
1 received ping_pong_count 7 from 0
1 sent and incremented ping_pong_count 8 to 0
0 received ping_pong_count 8 from 1
0 sent and incremented ping_pong_count 9 to 1
1 received ping_pong_count 9 from 0
1 sent and incremented ping_pong_count 10 to 0
0 received ping_pong_count 10 from 1
```

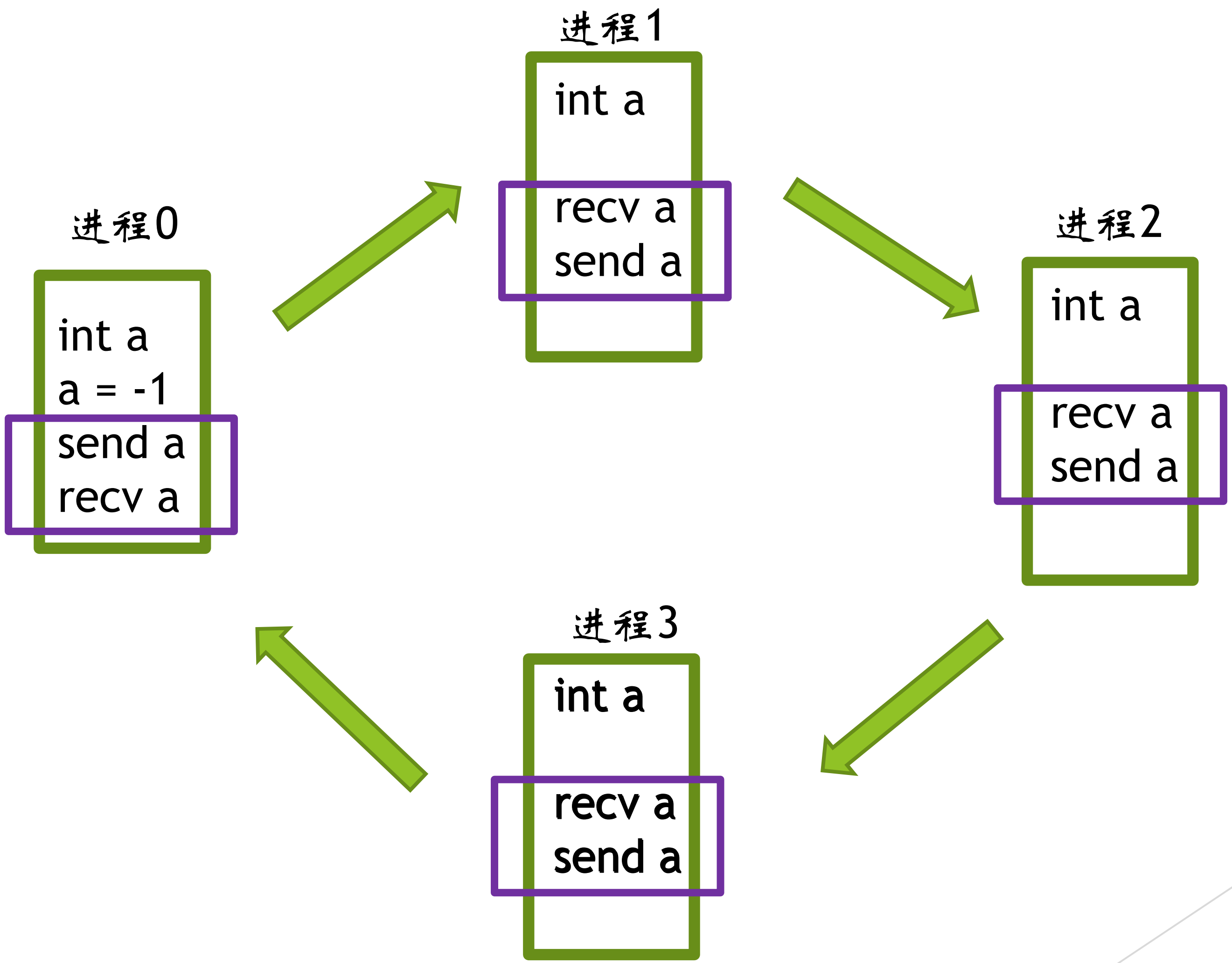
MPI指环程序

- 指环程序在进程0中初始化一个值，然后这个值被依次传递到另外的进程中。当此值从最后一个进程传递到进程0的时候，整个程序便终止了。



MPI指环程序

- 算法设计：对于进程0：先发送a,最后接收a.对于其他进程先接收从上一进程发过来的a,再发送a给下一个进程。



指环程序代码

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

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        int token, prev, next;
        prev = world_rank - 1;
        next = world_rank + 1;

        if(world_rank == 0){
                prev = world_size - 1;
        }

        if(world_rank == world_size - 1){
                next = 0;
        }
```

指环程序代码

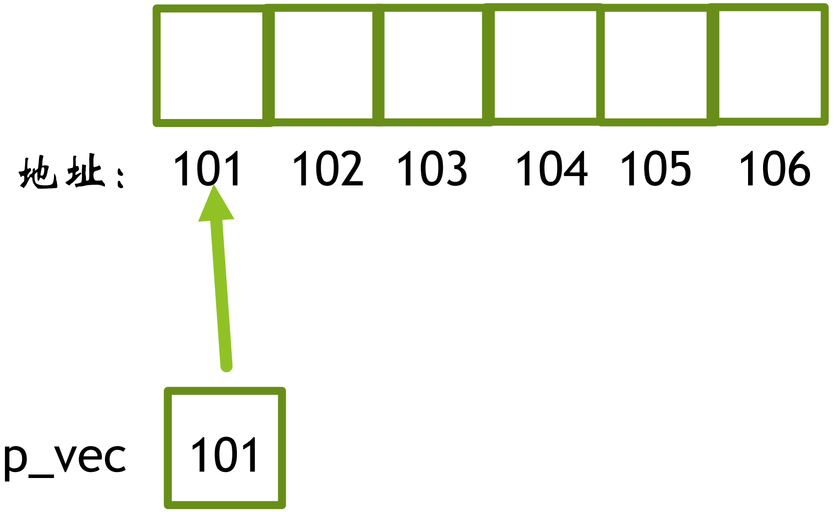
```c
        if(world_rank == world_size - 1){
                next = 0;
        }

        if(world_rank == 0){
                token = -1;
                MPI_Send(&token, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
                printf("Process %d sent token %d to process %d\n",
                        world_rank, token, next);
                MPI_Recv(&token, 1, MPI_INT, prev, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                printf("Process %d received token %d from process %d\n",
                        world_rank, token, prev);
        }else{
                MPI_Recv(&token, 1, MPI_INT, prev, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                printf("Process %d received token %d from process %d\n",
                        world_rank, token, prev);
                MPI_Send(&token, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
                printf("Process %d sent token %d to process %d\n",
                        world_rank, token, next);
        }

        MPI_Finalize();

}
```

指环程序代码结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ vi ring.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o ring ring.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 6 ./ring
Process 0 sent token -1 to process 1
Process 1 received token -1 from process 0
Process 1 sent token -1 to process 2
Process 2 received token -1 from process 1
Process 2 sent token -1 to process 3
Process 3 received token -1 from process 2
Process 3 sent token -1 to process 4
Process 4 received token -1 from process 3
Process 4 sent token -1 to process 5
Process 5 received token -1 from process 4
Process 5 sent token -1 to process 0
Process 0 received token -1 from process 5
```

例子：点对点通信传递一个矩阵

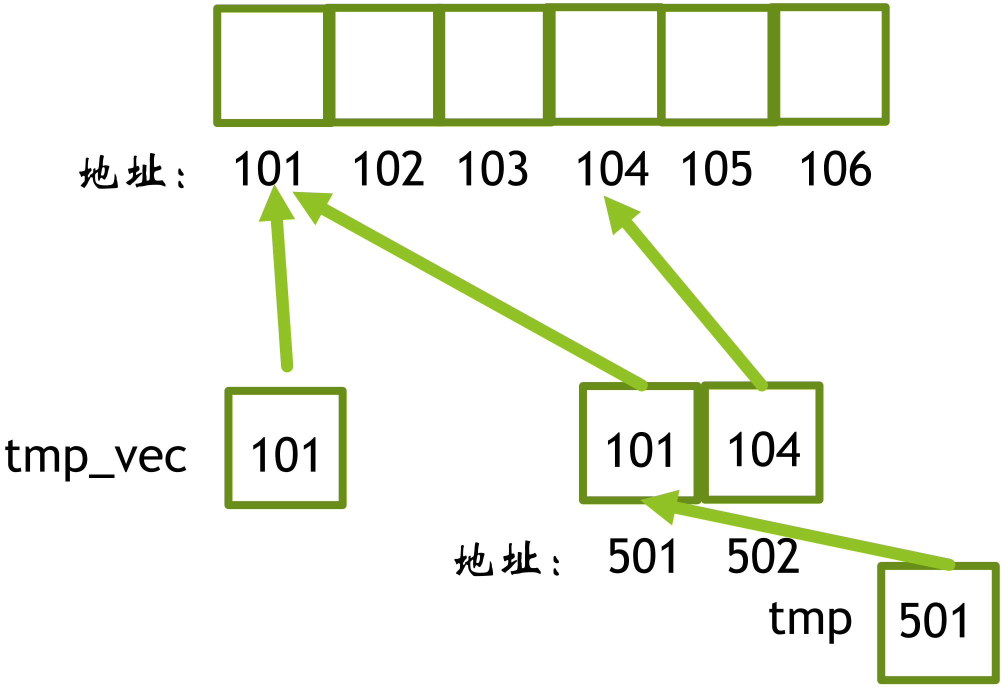
- 利用malloc动态定义一个矩阵（二维数组）
- 动态定义一个2乘3的矩阵




```
double *tmp_vec = (double *) malloc(a1*a2*sizeof(double));
```

例子：点对点通信传递一个矩阵

- 利用malloc动态定义一个矩阵（二维数组）
- 动态定义一个2乘3的矩阵




```
double ** tmp =
  (double **)malloc(a1*sizeof(double *));
 for(int i=0; i < a1; i++){
        tmp[i] = &(tmp_vec[i*a2]);
 }
```

例子：点对点通信传递一个矩阵

- 利用malloc动态定义一个矩阵（二维数组）

```
double ** make2Darray(int a1, int a2){
    double *tmp_vec = (double *) malloc(a1*a2*sizeof(double));
    double **tmp;
    tmp = (double **)malloc(a1*sizeof(double *));
    for(int i=0; i < a1; i++){
        tmp[i] = &(tmp_vec[i*a2]);
     }
     return tmp;
}
```

- 这样定义的优点：矩阵中的每个元素的地址是**连续**的。
- 思考：和R中矩阵的区别？

例子：点对点通信传递一个矩阵

- 消去给此矩阵分配的内存
- void delete2Darray(double **tmp, int a1, int a2){
-         free(tmp[0]);
-         free(tmp);
- }

例子：点对点通信传递一个矩阵

```
	int nrow = 2, ncol = 3;
	double ** matr = make2Darray(nrow, ncol);

	if(world_rank == 0){
		for(int i=0; i < nrow; i++){
			for(int j=0; j < ncol; j++){
				matr[i][j] = i*j;
			}
		}
		MPI_Send(&(matr[0][0]), nrow*ncol, MPI_DOUBLE, 1, 888, MPI_COMM_WORLD);
	}

	if(world_rank == 1){
		MPI_Recv(&(matr[0][0]), nrow*ncol, MPI_DOUBLE, 0, 888, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		for(int i=0; i<nrow; i++){
			for(int j=0; j<ncol; j++){
				printf(" %f ", matr[i][j]);
			}
			printf("\n");
		}
	}

	delete2Darray(matr, nrow, ncol);
```

例子：点对点通信传递一个矩阵

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 2 ./transfer_
matr
 0.000000  0.000000  0.000000
 0.000000  1.000000  2.000000
```

动态的消息传递：MPI_Recv中的MPI_Status

- 我们讨论了通过MPI_Send和MPI_Recv来实现标准的点对点通信，但仅仅是如何发送事先已知长度的消息。MPI能够支持动态的消息传递（接收进程并不知道发送过来多少长度的消息）。
- int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)
- MPI_Recv操作将MPI_Status结构作为最后一个参数，在例子中我们用MPI_STATUS_IGNORE忽视掉了它。如果我们将其设置为一个MPI_Status变量，那么在接收完成后，它能提供额外的信息。三大主要的信息为
  - 发送进程的秩：发送进程的秩储存在MPI_SOURCE。秩可以通过stat.MPI_SOURCE得到。
  - 消息的标签：消息标签可以通过stat.MPI_TAG得到。
  - 消息的长度：消息的长度没有一个事先定义的status的元素，但可以通过函数MPI_Get_count得到。

动态的消息传递：MPI_Recv中的MPI_Status

- MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count)
- 在此函数中，用户提供MPI_Status结构和消息的datatype，返回值为count。count变量指有多少个datatype元素的总数。
- 为什么这些信息是必要的呢？MPI_Recv可以用MPI_ANY_SOURCE作为发送进程的秩，用MPI_ANY_TAG作为消息的标签。在这个例子中，MPI_Status是唯一的方式来找出实际的发送者和消息的标签。

动态的消息传递：MPI_Recv中的MPI_Status

- 一个例子：进程0传输随机个数字给进程1，而且进程1需要弄清楚多少个数字被发送了及发送来源的信息。

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

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if(world_size != 2){
                fprintf(stderr, "Must use two processes for this example\n");
                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        const int MAX_NUMBERS = 100;
        int numbers[MAX_NUMBERS];
        int number_amount, i;
```

动态的消息传递：MPI_Recv中的MPI_Status

```
        if(world_rank == 0){
                srand(time(NULL));
                number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

                for(i=0; i<number_amount; i++){
                        numbers[i] = 1 + 2*i;
                }

                MPI_Send(numbers, number_amount, MPI_INT, 1, 888, MPI_COMM_WORL
D);
                printf("0 sent %d numbers to 1\n", number_amount);
        }else if(world_rank == 1){
                MPI_Status status;
                MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, MPI_ANY_SOURCE, MPI_ANY
_TAG, MPI_COMM_WORLD, &status);
                MPI_Get_count(&status, MPI_INT, &number_amount);
                printf("1 received %d numbers from the source process = %d, and
 the message tag is %d\n", number_amount, status.MPI_SOURCE, status.MPI_TAG);
        }

        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
}
```

动态的消息传递：MPI_Recv中的MPI_Status

- 结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o check_status ch
eck_status.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 2 ./check_sta
tus
0 sent 35 numbers to 1
1 received 35 numbers from the source process = 0, and the message tag is 888
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$
```

MPI_Get_count的返回值是相对于输入的数据类型。如果用户用MPI_CHAR作为数据类型的话，返回的数字将会是实际的4倍（假设整数占四个字节，字符占一个字节）

动态的消息传递：MPI_Probe

- 其实，在接收消息**之前**，可以利用MPI_Probe来查询消息的大小
- 这样的话可以不必要设置MPI_Recv的参数为MAX_NUMBERS
- MPI_Probe可以看成是一种除了接收数据外的MPI_Recv
- int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)

动态的消息传递：MPI_Probe

- 例子

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

int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if(world_size != 2){
                fprintf(stderr, "Must use two processes for this example\n");
                MPI_Abort(MPI_COMM_WORLD ,1);
        }

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        int number_amount, i;
```

动态的消息传递：MPI_Probe

```
        if(world_rank == 0){
                const int MAX_NUMBERS = 100;
                int numbers[MAX_NUMBERS];
                srand(time(NULL));
                number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

                for(i=0; i<number_amount; i++){
                        numbers[i] = i+1;
                }

                MPI_Send(numbers, number_amount, MPI_INT, 1, 888, MPI_COMM_WORL
D);
                printf("Process 0 sent %d numbers to process 1\n", number_amoun
t);
        }else{
                MPI_Status stat;
                MPI_Probe(0, 888, MPI_COMM_WORLD, &stat);
                MPI_Get_count(&stat, MPI_INT, &number_amount);

                int* number_buff = (int*)malloc(sizeof(int)*number_amount);

                MPI_Recv(number_buff, number_amount, MPI_INT, 0, 888,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                printf("Process 1 dynamically received %d numbers from process
0.\n", number_amount);
                free(number_buff);
        }
```

动态的消息传递：MPI_Probe

- 结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o probe probe.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 2 ./probe
Process 0 sent 60 numbers to process 1
Process 1 dynamically received 60 numbers from process 0.
```

- 大部分代码都类似，有两处不同。第一，此例子中，用MPI_Probe和MPI_Get_count组合来找到发送的个数。第二，通过调用动态储存(合适的大小)的方式来接收消息。

动态的消息传递：MPI_Probe

- 尽管在这个例子中MPI_Probe的运用很平凡，MPI_Probe构成了很多动态MPI应用。比如，当需要交换变化大小的工作消息时，进程之间经常大量利用到MPI_Probe。因此，你可以定义一个新的类似于MPI_Recv的函数来接收动态消息。

MPI点对点通信：动态的消息传递

```
#define MYTYPE int


void MPI_Recv_dynamic(void *buf, MPI_Datatype datatype, int source,
                        int tag, MPI_Comm comm){
    MPI_Status stat;
    MPI_Probe(source, tag, comm, &stat);
    int count_number;
    MPI_Get_count(& stat, datatype, & count_number);
    buf = (MYTYPE *) malloc(sizeof(MYTYPE) * count_number);
    MPI_Recv(buf, count_number, datatype, source, tag, comm,
            MPI_STATUS_IGNORE);
}
```

MPI集体通信（collective communication）

- MPI集体通信是指通信器中所有进程都参与的通信方式
- 集体操作的类型
  - 同步操作：所有进程进行等待，直到所有进程完成某任务达到同步点
  - 数据移动操作：广播(broadcast)、分散(scatter)、集合(gather)等
  - 聚集计算操作：缩减(reduction),一个进程收集其他进程的数据并进行计算操作,比如求最大值、相加、相乘等
- 集体通信作用在某个通信器中所有的进程
  - 默认条件下，针对MPI_COMM_WORLD中的所有进程
  - 额外的通信器可以由编程人员自行定义
- 让通信器中的所有进程参与集体通信是程序员的责任，否则会程序可能产生错误

MPI集体通信的一些特点

- 集体通信不需要信息标签(tag)
- 为了将集体通信作用在某些进程，可以首先把这些进程分为若干组，然后将组作为通信器
- 用在MPI事先定义的数据类型
- MPI-2 扩展版本允许多个通信器之间的数据移动
- MPI-3中的集体操作可以是阻塞性的也可以是非阻塞性的
- 这门课着重于阻塞性集体通信

MPI集体通信（collective communication）

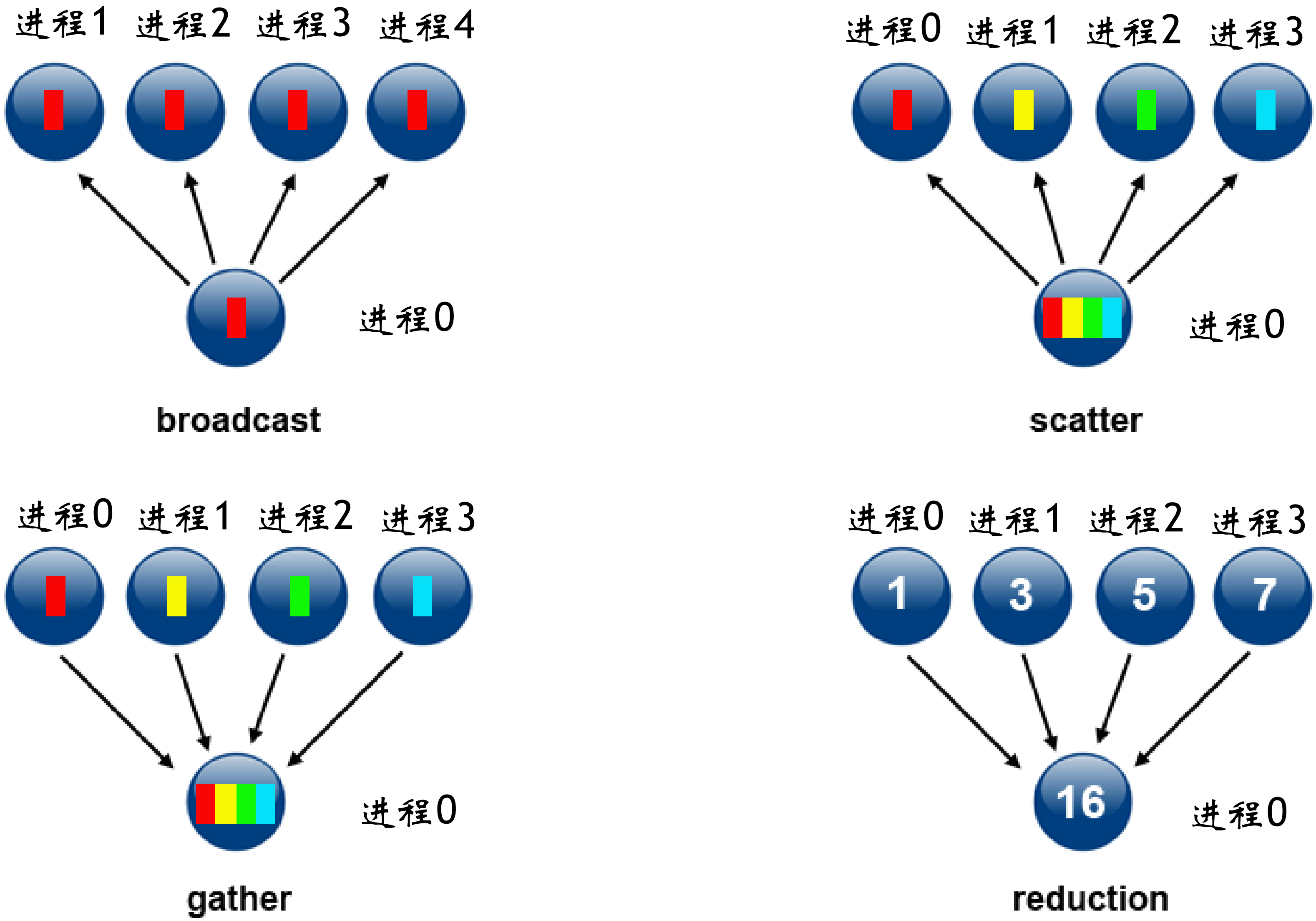
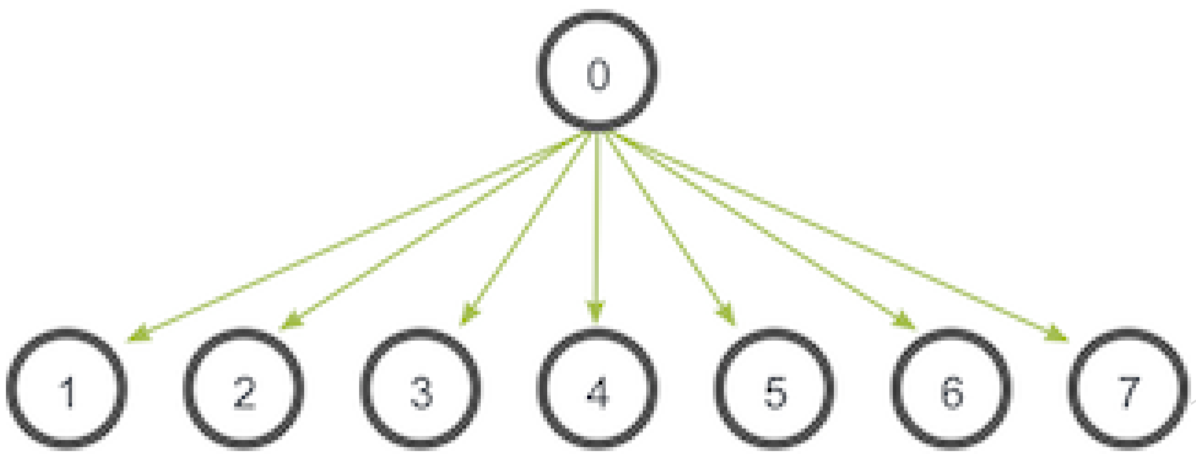


Figure credit: https://computing.llnl.gov/tutorials/mpi/

MPI广播通信

- MPI广播(broadcast)是将一个进程的数据传播到其他各个进程
- MPI内置函数 int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
  - 参数root指的是发送消息的进程rank



MPI广播通信

- MPI广播通信也可以用MPI点对点通信来自定义
- 当进程数量足够大时，自定义广播的速度是比MPI_Bcast慢得多的

MPI自定义广播通信

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

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
                MPI_Comm communicator){
        int world_rank;
        MPI_Comm_rank(communicator, &world_rank);
        int world_size;
        MPI_Comm_size(communicator, &world_size);
```

作业：

基本思路是，若进程编号等于root，发送消息给其他所有进程；若进程编号不等于root，则接收由进程root发来的消息。

```
}
```

MPI自定义广播通信

```
int main(int argc, char** argv){
        MPI_Init(NULL, NULL);

        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        int data;
        if(world_rank == 0){
                data = 100;
        }

        my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if(world_rank == 0){
                printf("Process 0 broadcasted data\n");
        }else{
                printf("Process %d successfully received data %d from root proc
ess\n", world_rank, data);
        }

        MPI_Finalize();
}
```