

COMP4007: 并行处理和体系结构

第六章：基于MPI的并行编程I

授课老师：王强、施少怀
助 教：刘虎成、田超

哈尔滨工业大学（深圳）

大纲

▶ MPI概览

- ✎ 基本概念
- ✎ 六大核心功能

▶ MPI 基础

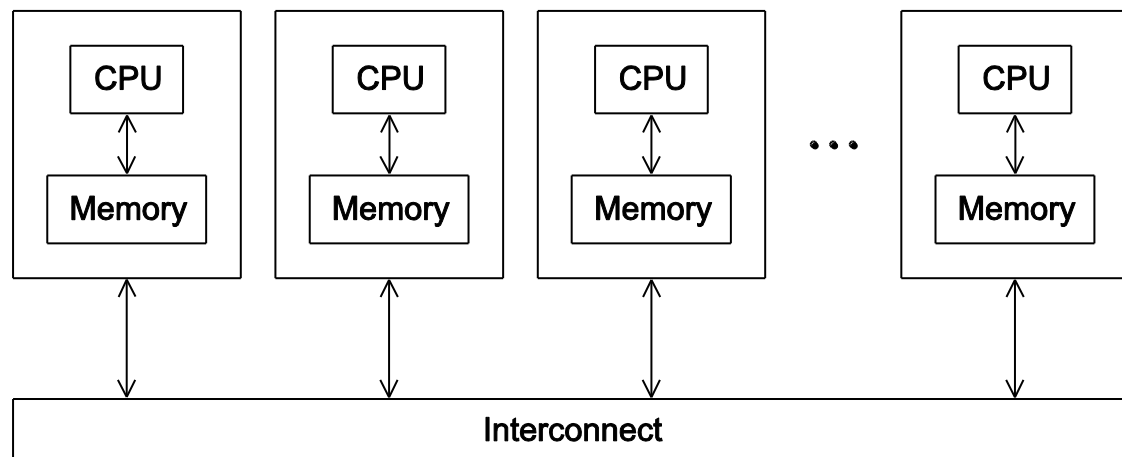
- ✎ 初始化和最终完成
- ✎ 一个简单的例子

概览: 消息传递接口[Message Passing Interface, MPI]

- ▶ 共享内存 vs. 分布式内存
- ▶ MPI是什么?
- ▶ MPI 基本概念
- ▶ MPI 的六大核心功能

分布式内存系统

- ▶ 每个 CPU 都有自己的私有内存空间
- ▶ CPU 通过高速网络（如以太网、InfiniBand）以收发信息的方式进行通信
 - ✎ 优点：CPU 数量可以非常多
 - ✎ 缺点：吞吐量比共享内存系统低，延迟比共享内存系统高
- ▶ MPI 是目前最流行的面向分布式内存系统的编程技术
- ▶ 新技术包括 Hadoop MapReduce 和 Apache Spark



MPI是什么？



▶ MPI 是一种广泛使用的编写消息传递程序的标准

✎ <http://www.mpi-forum.org>

✎ 是规范，不是实现

▶ 它被实现为一个库而不是一种编程语言

✎ 常见的MPI库有：MPICH, Open MPI, Microsoft MPI (MS-MPI), Intel MPI

✎ C、C++、Fortran、Java、Python 等语言都支持MPI

▶ MPI 的历史

✎ 第一个 MPI 标准MPI-1 于 1994 年 5 月发布

✎ MPI-1.1 (1995), MPI-1.2 (1997), MPI_1.3 (1998)

✎ 第二个 MPI 标准 MPI-2 于 1997 年发布

✎ MPI-2.1 (2008), MPI-2.2 (2009)

✎ 2012 年发布了 MPI-3 (800 多页)

✎ MPI-4于2021年发布(1,139 页)

▶ MPI 程序由许多进程组成

✎ 这些进程由一组物理处理器执行，这些处理器通过内部总线或网络交换数据

▶ 并行执行的进程拥有独立的地址空间

✎ 假设程序中有一条语句 “ $y = a + b$ ”

✎ 当进程 A 和进程 B 同时执行上述语句时，每个进程都有自己的变量集 {a、b、y}

▶ 消息传递：一个进程的部分地址空间被复制到另一个进程的地址空间中

✎ “消息”指“数据”

✎ “消息传递”指“数据传输”

✎ 通常通过发送操作和接收操作来完成

▶ 发送方需指定：

- ✎ 谁是接收方（或目的地）？
- ✎ 如何定义信息？

▶ 接收方需指定：

- ✎ 接收到的消息存储在哪？
- ✎ 谁是发送方，或者在哪里存储“发送方”信息？

▶ 发送方和接收方之间的匹配

- ✎ 一对发送方和接收方可以使用“信息标签”来控制接收哪条信息

如何识别进程？

▶ 通信域(Communicator)

- ✎ 在 MPI 中，一组进程可以组成一个“组(group)”，由一个通信域标识
- ✎ 默认通信域MPI_COMM_WORLD包括所有进程

▶ 序号(Rank)

- ✎ 如果group中包含 n 个进程，那么其进程在组内由rank标识，取值为 0 到 $n-1$ 之间的整数
- ✎ 通常，一个进程在 MPI_COMM_WORLD的rank为进程本身的rank

如何定义消息？



▶ 简单的解决方案：（地址、长度）

- ✎ 消息连续存储于内存空间中
- ✎ “地址”指消息的起始内存地址
- ✎ “长度”指消息的长度（以字节为单位）

▶ MPI 的解决方案：（地址、数量、数据类型）

- ✎ 消息是一个具有相同数据类型的数组
- ✎ “地址”指消息的起始内存地址
- ✎ “数量”指以一个数据类型为单位的数量总量
- ✎ “数据类型”指单个数据存储的类型，可以是简单的基本数据类型，如整数、浮点数，也可以是用户定义的复杂数据类型

▶ 点对点通信

✎ 数据由一个进程发送，另一个进程接收

▶ 集合通信

✎ 通信涉及一组或多组进程

▶ 单边通信

✎ 一种 "远程内存访问" 方式

✎ 一个进程指定发送端和接收端的所有通信参数。

六个MPI核心函数

| 函数名 | 描述 |
|---------------|-------------|
| MPI_Init | 初始化MPI |
| MPI_Comm_size | 返回进程总数 |
| MPI_Comm_rank | 返回当前进程的rank |
| MPI_Send | 发送一个消息 |
| MPI_Recv | 接收一个消息 |
| MPI_Finalize | 终止MPI |

MPI 基础

- ▶ 示例 1: 计算 π 的值
- ▶ 示例 2: 矩阵-向量乘法
- ▶ 性能度量
- ▶ 通信域

示例 1：计算 π 的值

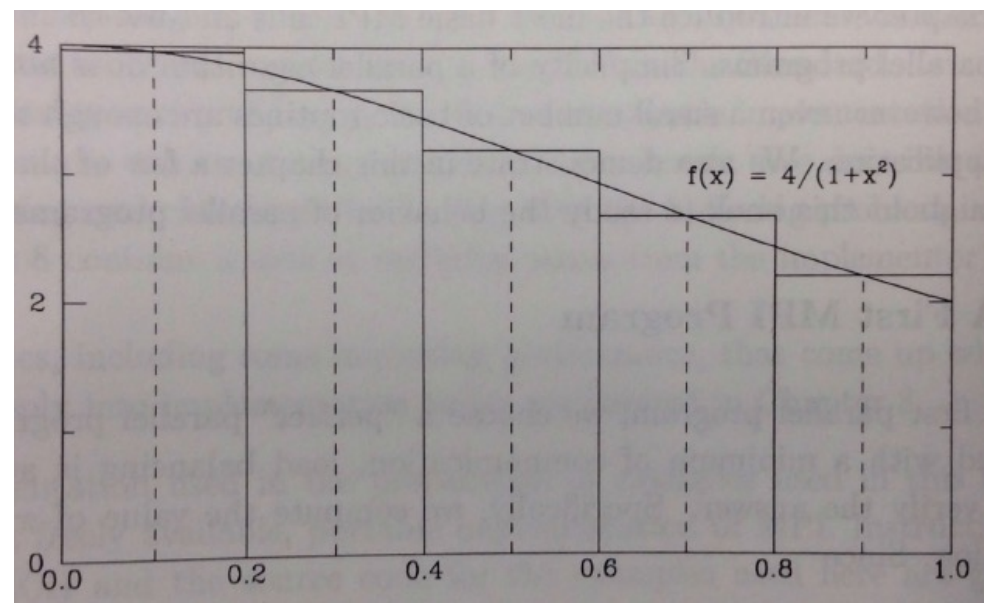


▶ $\int_0^1 \frac{1}{1+x^2} dx = \arctan(x) \Big|_0^1 = \arctan(1) - \arctan(0) = \frac{\pi}{4}$

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

▶ 数值解法：

- ✎ 将 0 到 1 的区间划分为 n 个子区间
- ✎ 累加这些矩形的面积
- ✎ 右图显示的是 $n = 5$ 的情况



第一个 MPI 程序: main()



```
1 #include "mpi.h"
2 #include <stdio.h>
3 #include <math.h>
4
5 int main( int argc, char *argv[] )
6 {
7     int n, myid, numprocs, i;
8     double PI25DT = 3.141592653589793238462643;
9     double mypi, pi, h, sum, x;
10
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
13     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14
15     while (1) {
16         /* see next page */
17     }
18
19     MPI_Finalize();
20     return 0;
21 }
```

第一个 MPI 程序: while()主体



```
15  while (1) {
16      if (myid == 0) {
17          printf("Enter the number of intervals: (0 quits) ");
18          scanf("%d",&n);
19      }
20      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
21      if (n == 0)
22          break;
23      else {
24          h = 1.0 / (double) n;
25          sum = 0.0;
26          for (i = myid + 1; i <= n; i += numprocs) {
27              x = h * ((double)i - 0.5);
28              sum += (4.0 / (1.0 + x*x));
29          }
30          mypi = h * sum;
31          MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
32          if (myid == 0)
33              printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
34      }
35  }
```


C/C++ 语言中使用MPI



- ▶ 需要包含 `mpi.h` 头文件
- ▶ 由 MPI 定义的标识符以 “MPI_” 开头
- ▶ 下划线后的第一个字母为大写
 - ✎ 用于函数名称和 MPI 定义的类型
 - ✎ 有助于避免混淆


MPI 初始化和释放

▶ MPI_Init()

 告诉 MPI 进行所有必要的设置

```
int MPI_Init(int *argc_p, char **argv_p);
```

▶ MPI_Finalize()

 告诉 MPI 运行完成了，需要清理为该程序分配的资源

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

MPI 初始化和释放



▶ `int MPI_Comm_size(MPI_Comm comm, int* comm_sz_p)`

✎ 获取组内进程的数量（通信域）

▶ `int MPI_Comm_rank(MPI_Comm comm, int* my_rank_p)`

✎ 获取当前进程的rank

MPI 中的消息广播

- ▶ 在进程组内广播消息是一种常见模式

✎ 集合通信

- ▶ `int MPI_Bcast(void *buf,
 int count,
 MPI_Datatype datatype,
 int root,
 MPI_Comm comm)`

✎ `<buf、count、datatype>` 指定了“消息”

✎ `root` 指定源进程的rank

✎ `comm` 指定进程组(通信域)

- ▶ 示例:

✎ `MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);`

MPI 数据类型



| MPI数据类型 | C数据类型 | C++数据类型 |
|--------------------------|------------------|----------------------|
| MPI::CHAR | char | char |
| MPI::SHORT | signed short | signed short |
| MPI::INT | signed int | signed int |
| MPI::LONG | signed long | signed long |
| MPI::LONG_LONG | signed long long | signed long long |
| MPI::SIGNED_CHAR | signed char | signed char |
| MPI::UNSIGNED_CHAR | unsigned char | unsigned char |
| MPI::UNSIGNED_SHORT | unsigned short | unsigned short |
| MPI::UNSIGNED_LONG | unsigned long | unsigned long int |
| MPI::FLOAT | float | float |
| MPI::DOUBLE | double | double |
| MPI::LONG_DOUBLE | long double | long double |
| MPI::BOOL | | bool |
| MPI::COMPLEX | | Complex<float> |
| MPI::DOUBLE_COMPLEX | | Complex<double> |
| MPI::LONG_DOUBLE_COMPLEX | | Complex<long double> |
| MPI::BYTE | | |
| MPI::PACKED | | |

- ▶ 在进程组内将一组报文缩减为一条报文是另一种常见的操作

- ✎ 集合通信

- ✎ 操作示例包括求最大、最小、和、乘积等

- ✎ 稍后将详细讨论

- ▶ `int MPI_Reduce(void *sendbuf, void *recvbuf, int count,`
`MPI_Datatype datatype, MPI_Op op,`
`int root, MPI_Comm comm)`

- ✎ `<sendbuf, count, datatype>` 指定“消息”

- ✎ `<recvbuf, root>` 指定存储归约结果的位置（例如，根进程的recvbuf）

- ✎ `op`指定归约操作符

- ✎ `comm`指定进程组(通信域)

- ▶ 示例：

- ✎ `MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)`

MPI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 |

- ▶ 用 C 语言编译 MPI:

- ✎ \$mpicc -o mpi_pi mpi_pi.c

- ▶ 单核（多核）计算机上运行 MPI 程序:

- ✎ \$mpiexec -n 4 ./mpi_pi

- ▶ 计算机集群上运行 MPI 程序

- ✎ 创建一个包含集群中计算机名称的文本文件（如 my_cluster）

- ✎ \$mpiexec -f my_cluster -n 16 ./mpi_pi

- ▶ 如何测量程序的执行时间？
 - ✎不同的操作系统需使用不同的函数
- ▶ MPI 提供独立于平台的解决方案
- ▶ `double MPI_Wtime()`
 - ✎返回从过去任意时间开始的时间（以秒为单位）
 - ✎在程序段开始和结束时调用它然后相减
- ▶ `double MPI_Wtick()`
 - ✎它以秒为单位返回 `MPI_Wtime()` 的分辨率

运行示例



```
[shaohuais@node1 mpi]$ mpiexec -f mpimachine -n 4 ./a.out
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535896128, Error is 0.00000000000001803
It takes 3.878083 seconds.
```

mpimachine:

node1
node2
node3
node4

```
[shaohuais@node1 mpi]$ mpiexec -f mpimachine -n 2 ./a.out
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535905170, Error is 0.00000000000007239
It takes 7.729113 seconds.
```

```
[shaohuais@node1 mpi]$ mpiexec -f mpimachine -n 1 ./a.out
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535921401, Error is 0.00000000000023470
It takes 15.485958 seconds.
```

| 进程数 | 1 | 2 | 4 |
|---------|--------|-------|-------|
| 运行时间(s) | 15.486 | 7.729 | 3.878 |

示例 2: 矩阵-向量乘法

$$\mathbf{A} \in \mathbb{R}^{m \times n} \quad \mathbf{x} \in \mathbb{R}^{n \times 1}$$

$$\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{R}^{m \times 1}$$

矩阵-向量乘法



| | | | |
|-------------|-------------|----------|---------------|
| a_{00} | a_{01} | \cdots | $a_{0,n-1}$ |
| a_{10} | a_{11} | \cdots | $a_{1,n-1}$ |
| \vdots | \vdots | | \vdots |
| a_{i0} | a_{i1} | \cdots | $a_{i,n-1}$ |
| \vdots | \vdots | | \vdots |
| $a_{m-1,0}$ | $a_{m-1,1}$ | \cdots | $a_{m-1,n-1}$ |

| |
|-----------|
| x_0 |
| x_1 |
| \vdots |
| x_{n-1} |

 $=$

| |
|---|
| y_0 |
| y_1 |
| \vdots |
| $y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$ |
| \vdots |
| y_{m-1} |

```
1 void Mat_vect_mul(double A[], double x[], double y[], int m, int n) {
2     int i, j;
3     /* For each row of A */
4     for (i = 0; i < m; i++) {
5         /* From dot product of i-th row with x */
6         y[i] = 0.0;
7         for (j = 0; j < n; j++) {
8             y[i] += A[i*n+j] * x[j];
9         }
10    }
11 }
```

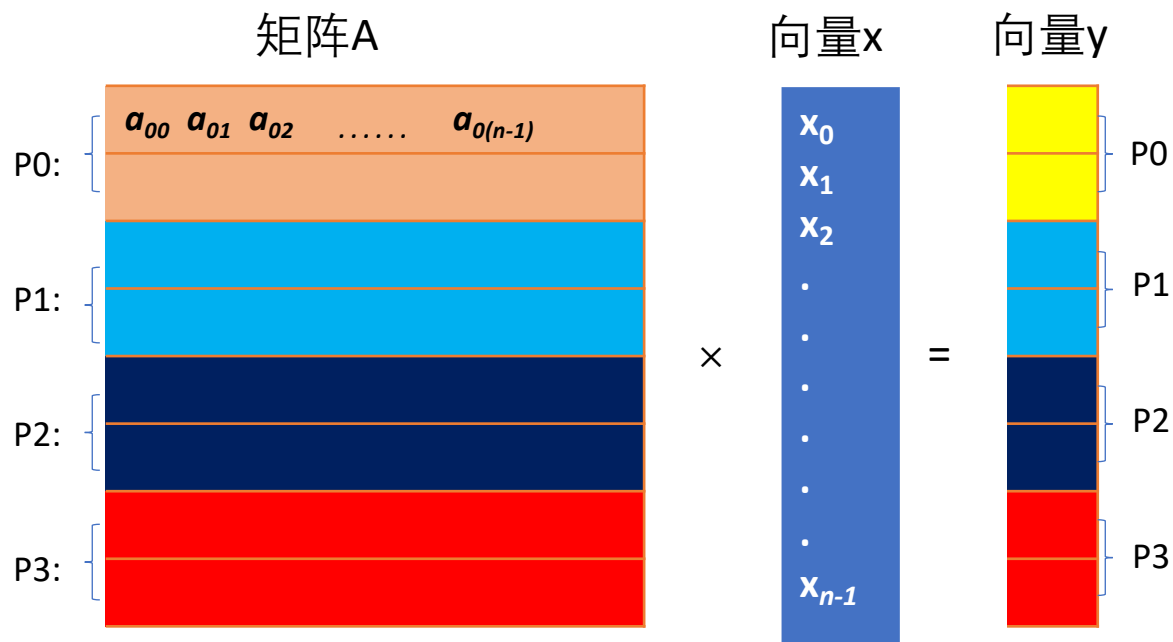
- ▶ 包含 $m \times n$ 个元素的矩阵A按1维数组方式存储
- ▶ 可通过 $A[i * n + j]$ 访问 $A[i][j]$

串行矩阵-向量乘法

按行划分

- ▶ 给定 p 个进程，矩阵 $A(m \times n)$ 被分割成 p 个较小的矩阵，每个矩阵的维数为 $(\frac{m}{p} \times n)$

✎ 简单起见，假设 p 可被 m 整除





并行矩阵-向量乘法： 框架

▶ 假设

✎ 共有 p 个进程

✎ 矩阵 $A(m \times n)$ 和向量 $x(n \times 1)$ 在进程 0 创建

☞ 进程0称为 "主进程", 因为它协调其他进程 (即 "从进程") 的工作

▶ 消息传递:

✎ 进程 0 将 $(p-1)$ 个子矩阵发送给对应的从进程

✎ 进程 0 将向所有其他 $p-1$ 个进程发送向量 x

▶ 计算:

✎ 每个进程用本地数据计算矩阵-向量乘法

▶ 消息传递:

✎ 进程 1 至 $(p-1)$ 将结果 (即向量 y 的一部分) 发送回进程 0

并行矩阵-向量乘法： 代码



```
5 int main(int argc, char** argv)
6 {
7     int m = 0, n = 0, myid, numprocs, srow = 0, i;
8     double *A, *x, *y;
9     MPI_Init(&argc, &argv);
10    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
11    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
12    if(myid == 0) {
13        while ( m <= 0 || n <= 0 || m % numprocs != 0 ) {
14            printf("Please input positive integers m and n: ");
15            scanf("%d %d", &m, &n);
16        }
17    }
18    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
19    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
20    srow = m / numprocs;
21    if (myid == 0) {
22        /* master code */
23    } else {
24        /* slave code */
25    }
26    free(A); free(x); free(y);
27    MPI_Finalize();
28    return 0;
29 }
```

主要参数:

(m, n): 矩阵维数

myid: 身份标识

numprocs: 进程总数

srow: 分配给单个进程的矩阵行数

A: 用于存储矩阵 (0 进程) 或子矩阵 (从进程) 的数据

x: 存储向量x的数据

y: 存储向量 y (用于 0 号进程) 或 y 的子向量 (用于从进程) 的数据

```
21     if (myid == 0) {
22         /* master code */
23         /* allocate memory for matrix A, vectors x and y, and initialize them */
24         A = (double*) malloc( m * n * sizeof(double) );
25         x = (double*) malloc(n * sizeof(double) );
26         y = (double*) malloc(m * sizeof(double) );
27         init_array(A, m * n); // Remark: this function is written by ourselves
28         init_array(x, n);
29
30         /* broadcast vector x */
31         MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
32
33         /* send sub-matrices to other processes */
34         for(i = 1; i < numprocs; i++)
35             MPI_Send(A+i*srow*n, srow * n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
36
37         /* perform its own calculation for the 1st sub-matrix */
38         Mat_vect_mul(A, x, y, srow, n); // Remark: this function is written by ourselves
39
40         /* collect results from other processes */
41         for(i = 1; i < numprocs; i++)
42             MPI_Recv(y+i*srow, srow, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
43     } else {
44         /* slave code */
45         /* allocate memory for sub-matrix A, vector x, and sub-sector y */
46         A = (double*) malloc( srow * n * sizeof(double) );
47         x = (double*) malloc( n * sizeof(double) );
48         y = (double*) malloc( srow * sizeof(double) );
49
50         /* receive x from process 0 */
51         MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
52
53         /* receive sub-matrix from process 0 */
54         MPI_Recv(A, srow * n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
55
56         /* perform the calculation on the sub-matrix */
57         Mat_vect_mul(A, x, y, srow, n);
58
59         /* send the results to process 0 */
60         MPI_Send(y, srow, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
61     }
```


消息传递：MPI_Send和MPI_Recv



▶ 点对点通信

✎ 发送进程调用 MPI_Send(), 接收进程调用 MPI_Recv()

▶ `int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`

✎ `<buf, count, datatype>` 指定消息

✎ `dest` 指定接收消息的进程的rank

✎ `tag` (非负整数) 用于区分信息

✎ `comm` 指定进程组(通信域)

▶ 示例:

✎ `MPI_Send(A+i*srow*n, srow * n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);`

✎ `MPI_Send(y, srow, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);`

消息传递：MPI_Send和MPI_Recv



▶ `int MPI_Recv(void* buf, int maxsize, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status_p);`

✎ `<buf, maxsize, datatype>`指定消息的存储位置

✎ `source` 指定接收的消息的发送进程的rank

✎ `tag`应与发送方指定的“tag”相匹配

✎ `comm`指定进程组(通信域)

✎ `status_p` 可以获取更多关于接收到的消息的信息。可使用MPI_STATUS_IGNORE忽略

▶ 示例：

✎ `MPI_Recv(A, srow * n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`

✎ `MPI_Recv(y+i*srow, srow, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`

- ▶ 考虑进程 q 使用如下参数调用 MPI_Send

- ✎ MPI_Send(q_buf, q_size, q_type, dest, q_tag, q_comm);

- ▶ 进程 r 使用如下参数调用 MPI_Recv

- ✎ MPI_Recv(r_buf, r_size, r_type, src, r_tag, r_comm);

- ▶ 如果以下条件全部满足，那么进程r就能接收来自进程q的消息：

- ✎ $r_comm = q_comm$

- ✎ $r_tag = q_tag$ or MPI_ANY_TAG

- ✎ $dest = r$

- ✎ $src = q$ or MPI_ANY_SOURCE

- ✎ 进程q和进程r中的缓冲区应该兼容

- ✎ 例如， $q_type = r_type$ ，且 $r_size \geq q_size$

MPI_Status是什么？

- ▶ 接收方可能会提出以下问题：
 - ✎ 消息中的数据量是多少？
 - ✎ 谁是发送方？(如果使用 `MPI_ANY_SOURCE`)
 - ✎ 消息的tag是什么（如果使用 `MPI_ANY_TAG`）？
- ▶ MPI 定义了名为 `MPI_Status` 的结构保存上述三项信息
- ▶ 例如，`MPI_Status status`；
 - ✎ `status.MPI_SOURCE` 存储了发送方的信息
 - ✎ `status.MPI_TAG` 存储了tag信息
 - ✎ 要获取消息中的数据项数量，请执行以下操作：
 - ✎ `int count;`
 - ✎ `MPI_Get_count(&status, r_type, &count);`

- ▶ 如何从输入端（如键盘）读取数据，以及如何将数据写入输出端（如屏幕）？
- ▶ 在 MPI 中，大多数实现都允许所有进程完全访问 stdout 和 stderr。
 - ✎ 如果多个进程访问同一输出，顺序将无法预测
- ▶ 只有 MPI_COMM_WORLD 中的进程 0 才允许访问 stdin

- ▶ Thomas Sterling, Matthew Anderson and Maciej Brodowicz (2018), “High Performance Computing: Modern Systems and Practices,” Morgan Kaufmann, Chapter 8. [PDF: <https://www.sciencedirect.com/book/9780124201583/high-performance-computing>]