清華大學
Tsinghua University

高性能计算导论
第3讲：消息传递编程模型-1

翟季冬

计算机系

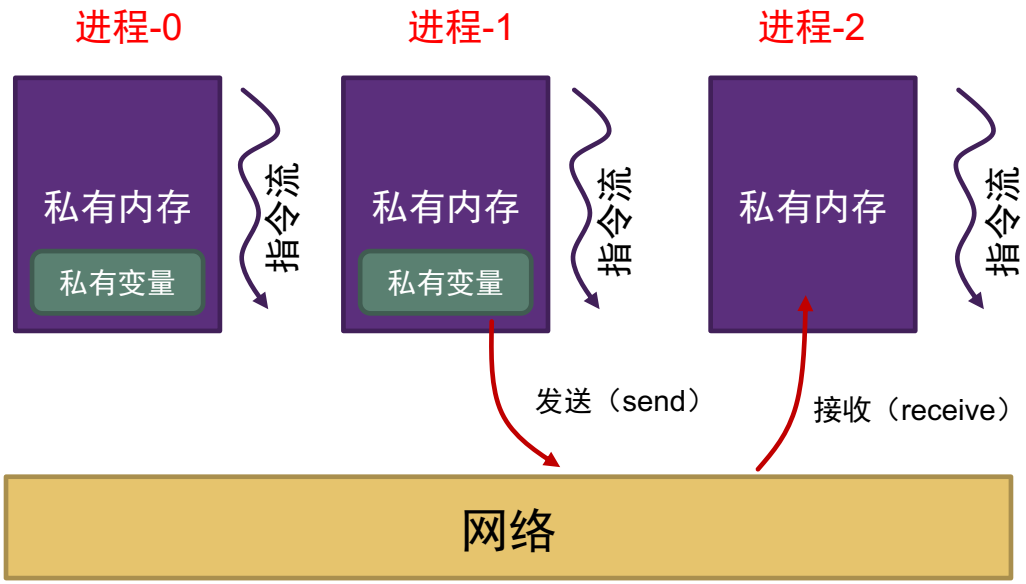
目录

- MPI介绍
- 如何写一个基本MPI程序
- 点对点通信
- 集合通信
- MPI-IO
- MPI程序编译与运行

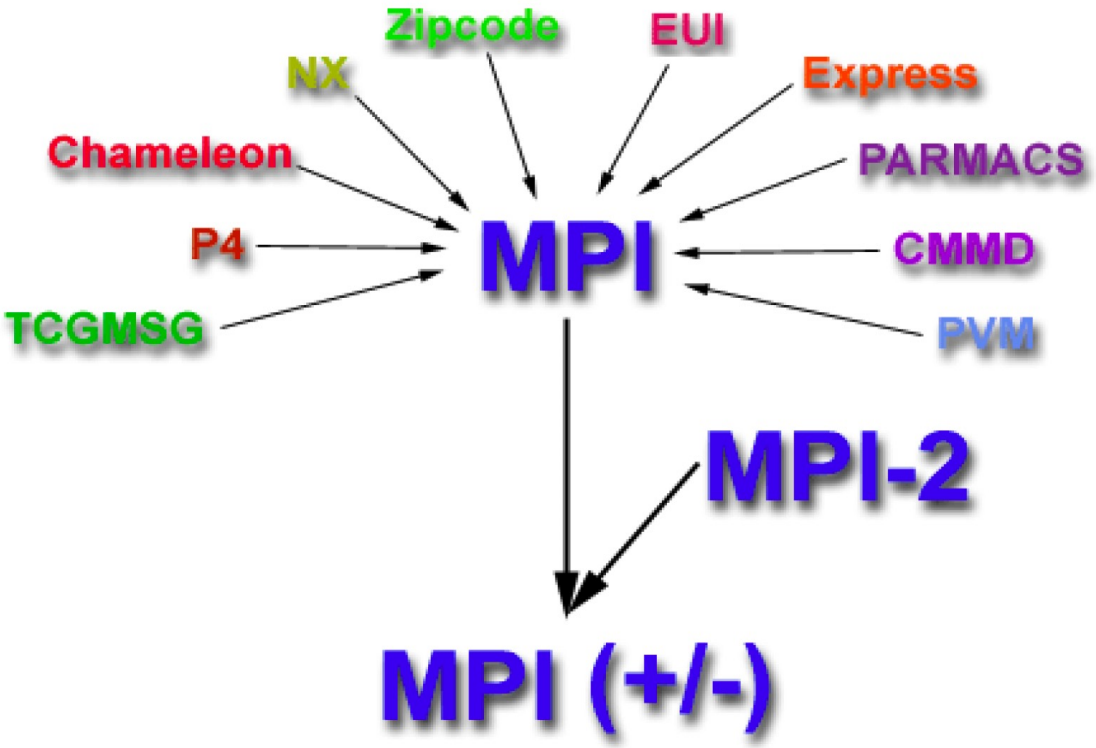
MPI介绍

消息传递模型-回顾

- 指令流：不同指令流并发执行，指令流之间同步基于通信方式
- 数据：不同指令流拥有独立的地址空间，相互不可通过地址直接访问
- 适用的硬件模型：分布式内存架构
  - 通信采用消息传递的方式实现

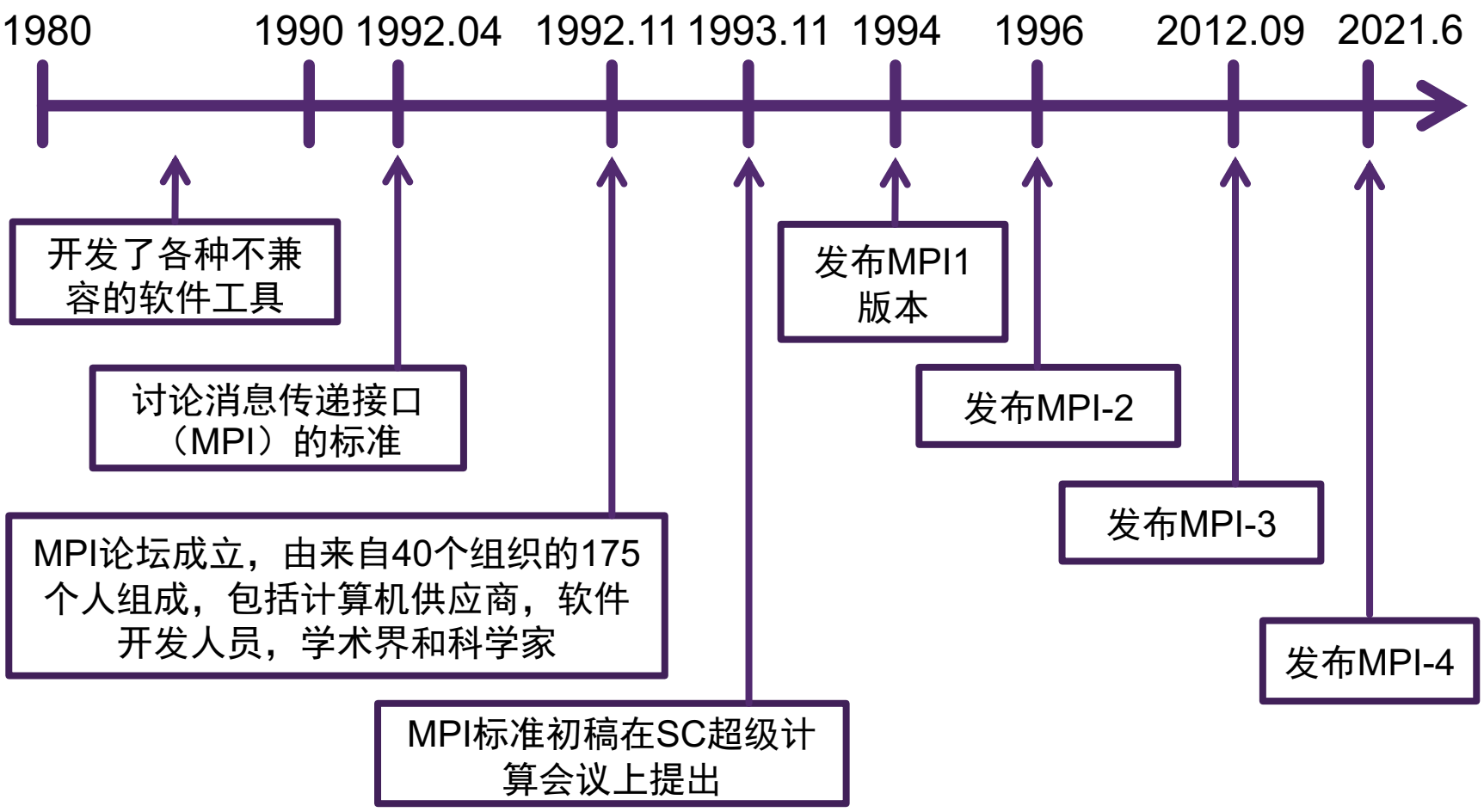


消息传递编程的历史



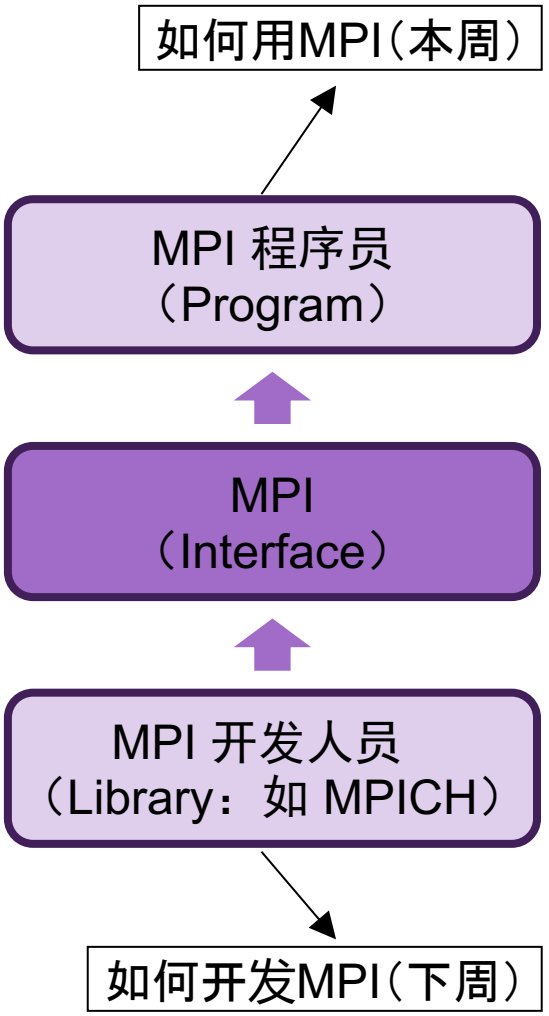
- 消息传递编程模型的发展经历很长历程
- MPI成为最终事实的标准

MPI 的历史与演变


1980
1990
1992.04
1992.11
1993.11
1994
1996
2012.09
2021.6
开发了各种不兼容的软件工具
讨论消息传递接口（MPI）的标准
MPI论坛成立，由来自40个组织的175个人组成，包括计算机供应商，软件开发人员，学术界和科学家
MPI标准初稿在SC超级计算会议上提出
发布MPI1版本
发布MPI-2
发布MPI-3
发布MPI-4

什么是MPI

- MPI = Message Passing Interface
- MPI是基于消息传递库的开发人员和用户的一套规范和标准
  - 就其本身而言，它是一个接口而不是一个库
- 目前是应用最广泛的消息传递程序标准
- 常用于分布式内存系统



MPI 设计目标

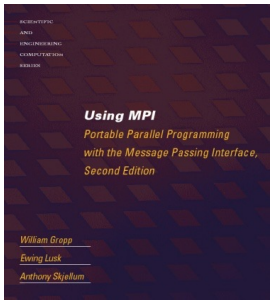
- 设计目标
  - 可移植性
    - 在不同的机器或平台上运行
    - Linux、Windows
  - 可扩展性
    - 在数百万个计算节点上运行
    - 支持当前最快的高性能计算机
  - 灵活性
    - 将 MPI 开发人员与 MPI 程序员（用户）隔离

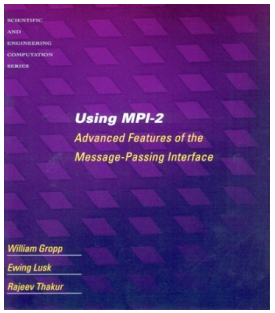
MPI 相关引用

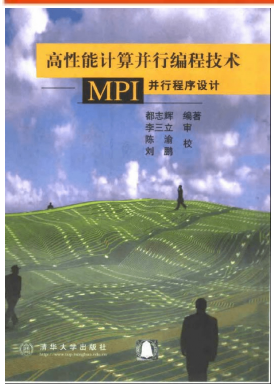
- MPI 官方标准
  - http://www.mpi-forum.org
  - 所有 MPI 官方发行版，包括 Postscript 和 HTML
  - 最新版本 MPI 4.0，2021年6月发布
- 网络相关材料
  - https://computing.llnl.gov/tutorials/mpi/
  - https://mpitutorial.com/tutorials/
  - MPI相关内容和资料

MPI 相关教材

- Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition), by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- Using MPI-2: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- MPI: The Complete Reference - Vol 1 The MPI Core, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- MPI: The Complete Reference - Vol 2 The MPI Extensions, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
- 高性能计算并行编程技术-MPI并行程序设计，都志辉 编著，2001.



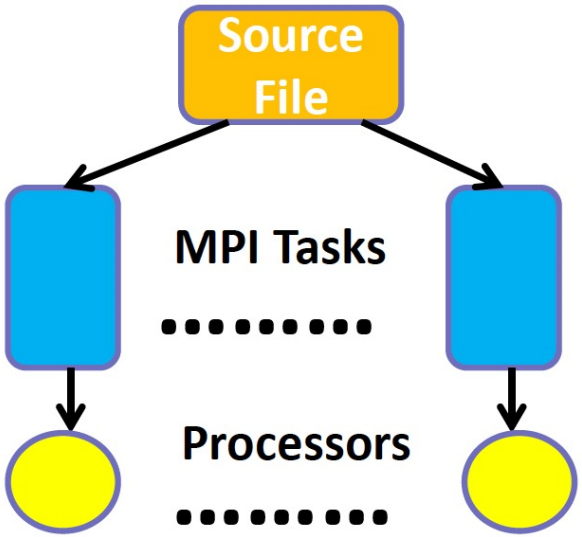




如何写一个基本MPI程序

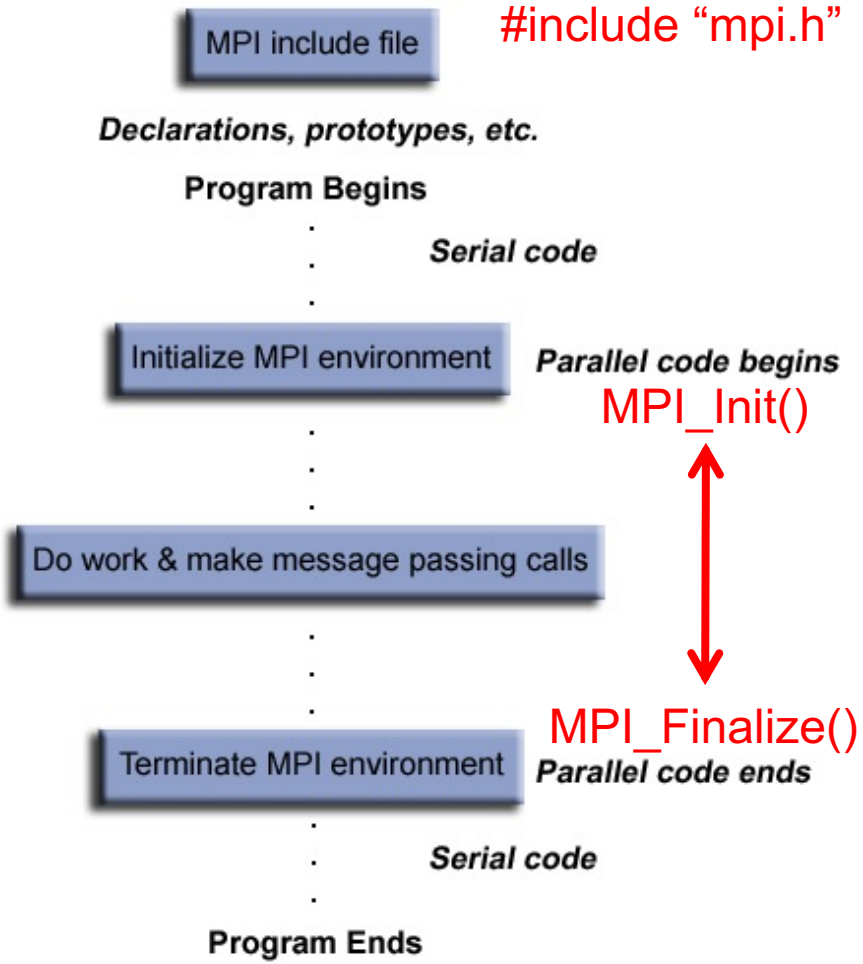
SPMD 编程风格

- 单程序多数据 (Single Program Multiple Data，SPMD)
  - 单个程序、多份数据
  - 允许任务分支执行用户指定的部分程序片段
  - SPMD：从程序级上看的，一类典型并行编程风格
  - SIMD： 从指令级上看的，一类常见硬件体系结构



MPI编程基础

- 头文件："mpi.h"
  - 所有使用MPI库的程序都必需包含
- MPI调用
  - 格式：rc = MPI_Xxx (参数, ...)
  - 示例：rc = MPI_Bcast(&buffer, count, datatype, root, comm)
  - 错误代码：返回为 rc
    如果正确执行 rc = MPI_SUCCESS
- MPI通用程序结构
  - 如右图所示



MPI编程基础

- 在并行程序开始有两个重要问题
  - **How many?**
  - **Who am I?**
- MPI提供了回答以上问题的函数：
  - MPI_Comm_size 报告进程数
    - size
  - MPI_Comm_rank 报告进程ID
    - 介于 0 和 size - 1 之间的数字
    - 以唯一识别每个进程

MPI环境配置函数

- MPI_Init ()
  - 初始化MPI运行环境
  - 必须在任何其他MPI函数之前调用
  - 在MPI程序中只能调用一次
- MPI_Finalize ()
  - 终止MPI运行环境
  - 之后不能再调用其他MPI函数
- MPI_Comm_size (comm, &size)
  - 确定与通信域关联的组中的进程数量
- MPI_Comm_rank (comm, &rank)
  - 确定通信域内的Rank
  - Rank通常称为任务ID

示例程序 Hello world - C

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d!\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

某次运行输出-5个进程

```
I am 3 of 5!
I am 2 of 5!
I am 0 of 5!
I am 1 of 5!
I am 4 of 5!
```

示例程序 Hello world - C++

```
#include "mpi.h"
#include <iostream>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " << size << "\n";
    MPI::Finalize();
    return 0;
}
```

示例程序 Hello world - Fortran

```
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

进程数如何控制？

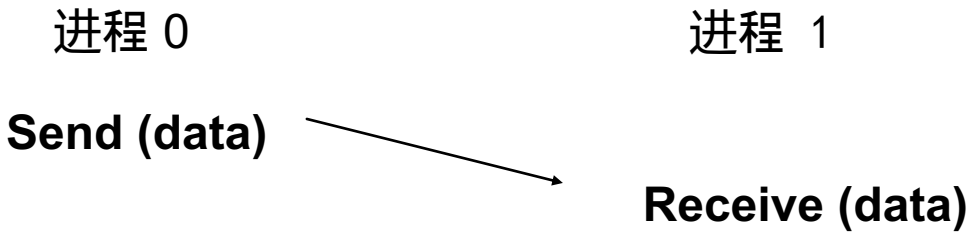
MPI通信

- 两类主要通信方式：
  - 点对点通信
  - 集合通信

点对点通信

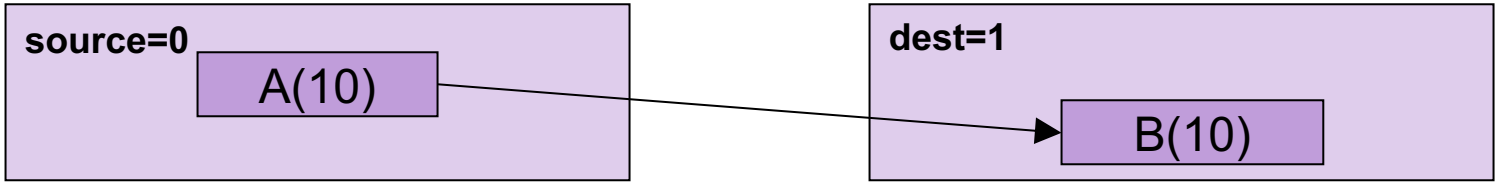
点对点通信：MPI_Send & MPI_Recv

- 需要在其中填写详细信息：



- 需要指定的信息：
  - 像我们寄送一个快递
  - 如何描述“数据”（data）？
  - 如何确定发送和接收进程？
  - 接收者如何识别/屏蔽不同消息？(0号进程给1号进程发了多个消息）
  - 这些操作完成意味着什么？

点对点通信：MPI_Send（阻塞）

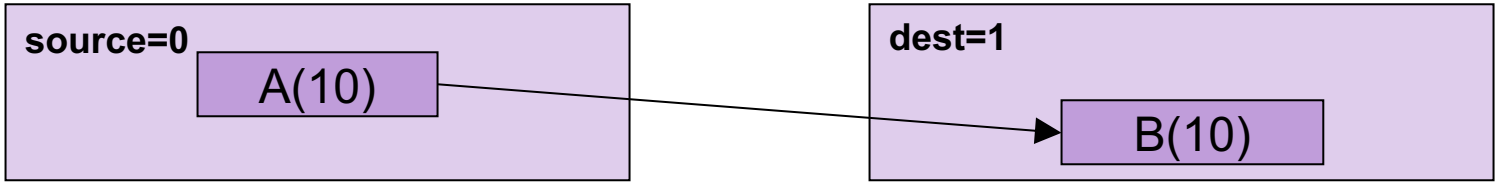


MPI_Send(A, 10, MPI_DOUBLE, 1, …)　　MPI_Recv(B, 10, MPI_DOUBLE, 0, …)

MPI_Send (buffer, count, type, dest, tag, comm)

- 发送的数据内容由 (**buffer, count, type**) 描述
- 接收进程由 **dest** 指定，**dest** 是 **comm** 通信域中目标进程的rank
- tag：消息的标签
- 当此函数返回时：消息发送缓冲区可以被复用
  - 数据拷贝到系统缓冲区
  - 但是，目标进程可能尚未收到该消息
- comm：默认值 MPI_COMM_WORLD

点对点通信：MPI_Recv（阻塞）



MPI_Send(A, 10, MPI_DOUBLE, 1, …)　　MPI_Recv(B, 10, MPI_DOUBLE, 0, …)

MPI_Recv (buffer, count, type, source, tag, comm, status)

- 接收进程等待，直到从系统收到匹配的消息（匹配 **source** 和 **tag** ），然后接收缓冲区可以被复用
- **source** 是在 **comm** 指定的通信域中的rank （或**MPI_ANY_SOURCE**）
- **tag** 是要匹配的消息标签 （或 **MPI_ANY_TAG**）
- 接收量少于 **count** 的 **type** 是可以的，但接收量超出会导致错误
- **status** 包含更多信息（例如消息大小）

通配符

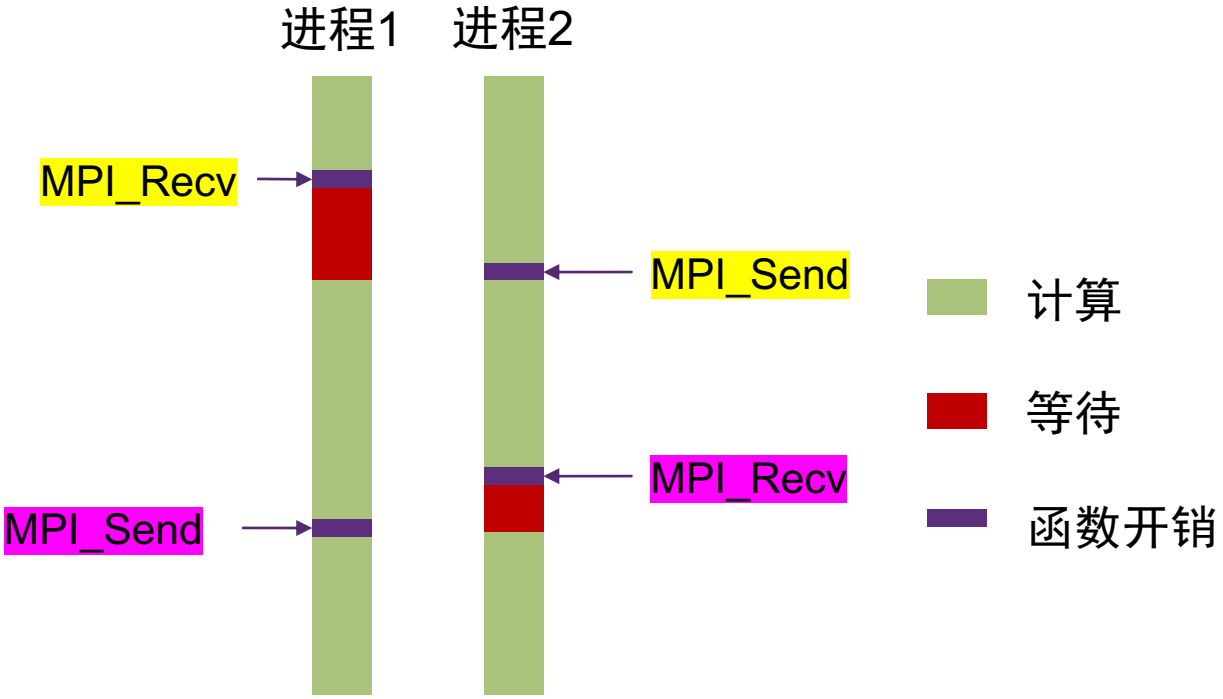
- 接收进程：
  - **source**

    MPI_ANY_SOURCE
  - **tag**

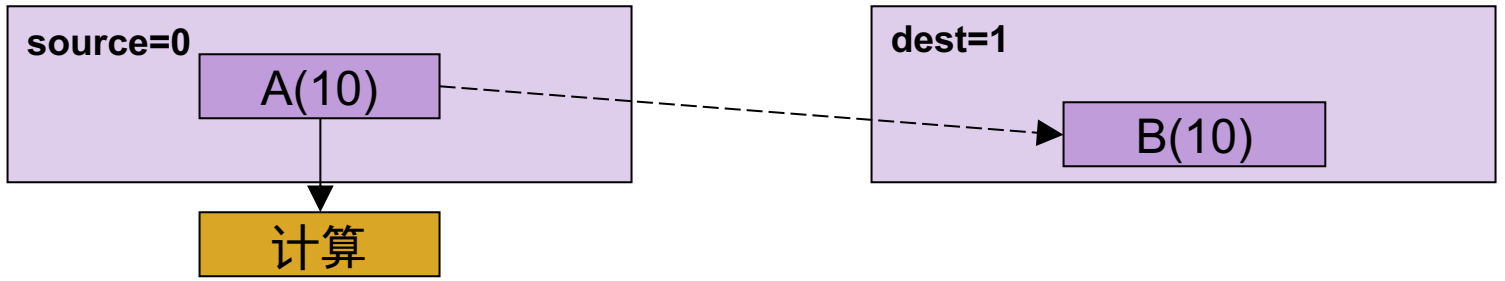
    MPI_ANY_TAG

消息传递模型中的通信类型

- 两种基本通信类型
  - 阻塞式通信
  - 非阻塞式通信
- 阻塞式通信
  - 命令发出需通信“完成”后才返回



点对点通信：MPI_Isend（非阻塞）

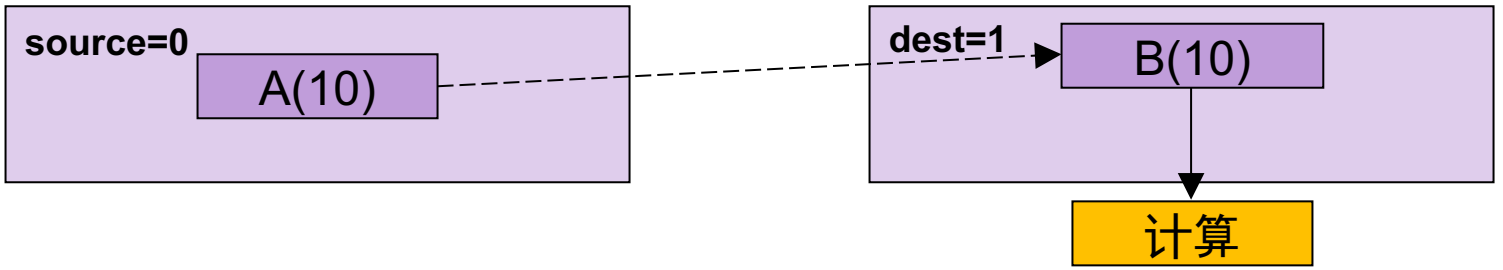


MPI_Isend(A, 10, MPI_DOUBLE, 1, …)　　MPI_Recv(B, 10, MPI_DOUBLE, 0, …)

MPI_Isend(buffer, count, type, dest, tag, comm, request)

- **request** 为返回的非阻塞通信对象（句柄），用以查询非阻塞发送是否完成
- 该函数启动一个标准的非阻塞发送操作，它调用后立即返回， 其调用返回并不意味消息已经成功发送，只表示该消息可以被发送

点对点通信：MPI_Irecv（非阻塞）

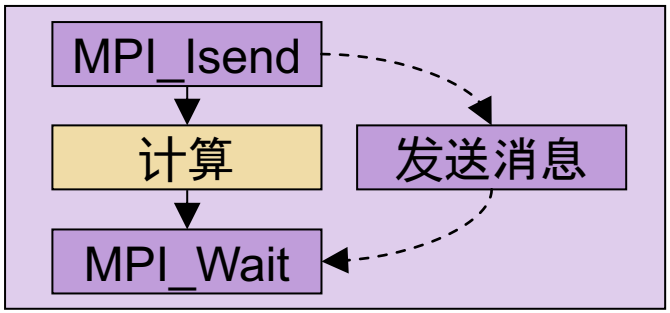


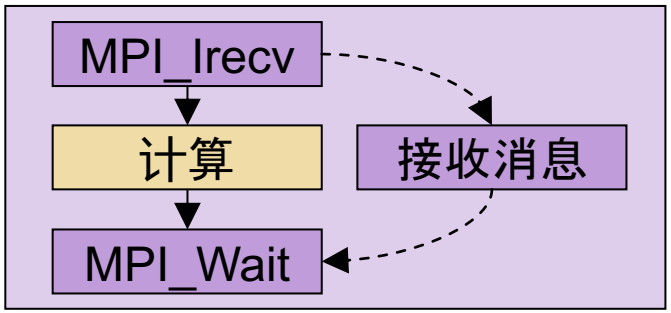
MPI_Send(A, 10, MPI_DOUBLE, 1, …)　　MPI_Irecv(B, 10, MPI_DOUBLE, 0, …)

MPI_Irecv (buffer, count, type, source, tag, comm, request)

- **request** 为返回的非阻塞通信对象（句柄），用以查询非阻塞接收是否完成
- 该函数启动一个标准的非阻塞接收操作，它调用后立即返回
- 其调用返回并不意味已经接收到相应的消息，只表示符合要求的消息可以被接收
- 目的：实现计算和通信重叠

点对点通信：MPI_Wait / MPI_Waitall





MPI_Wait (request, status)

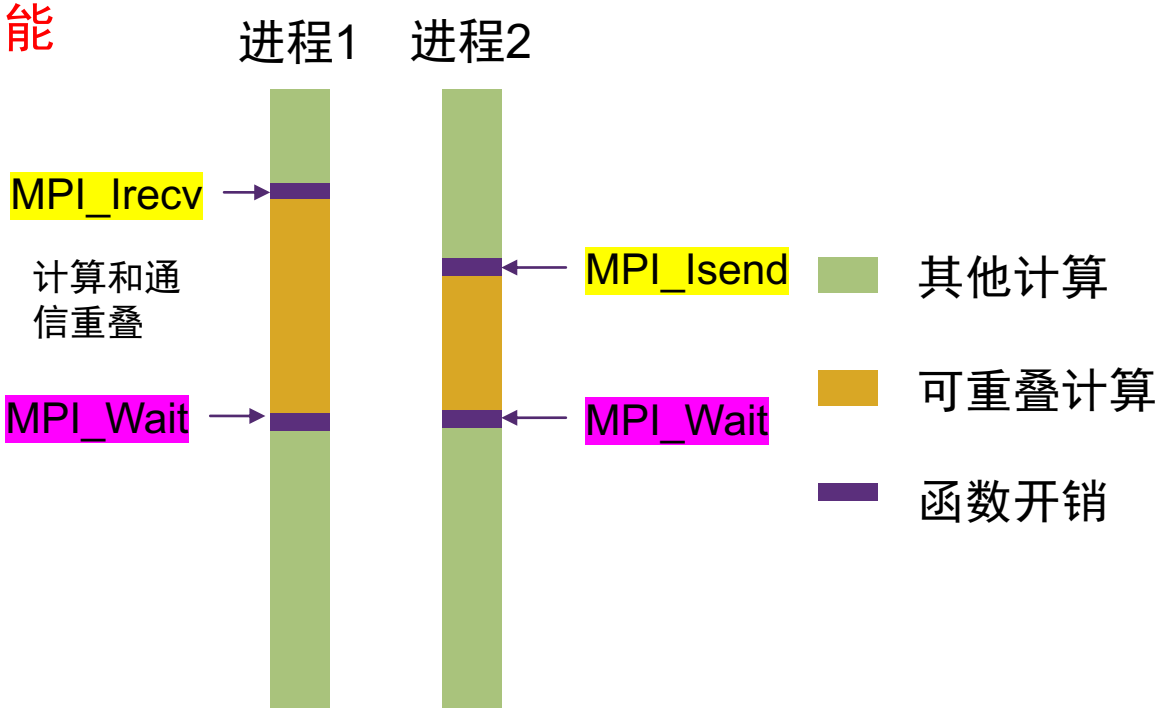
- 该函数以非阻塞通信对象（**request**）为参数，一直等到与该非阻塞通信对象相应的非阻塞通信完成后才返回，同时释放该阻塞通信对象

MPI_Waitall (count, array_of_requests, array_of_statuses)

- 该函数必须等到非阻塞通信对象表中所有的非阻塞通信对象相应的非阻塞操作都完成后才返回
- **count** 表示非阻塞通信对象的个数

消息传递模型中的通信类型

- 两种基本通信类型
  - 阻塞式通信
  - 非阻塞式通信
- 非阻塞式通信
  - 命令发出后，无需通信完成，立即返回
- 提供了计算通信重叠的可能



点对点通信参数

阻塞发送	MPI_Send(buffer, count, type, dest, tag, comm)
非阻塞发送	MPI_Isend(buffer, count, type, dest, tag, comm, request)
阻塞接收	MPI_Recv(buffer, count, type, source, tag, comm, status)
非阻塞接收	MPI_Irecv(buffer, count, type, source, tag, comm, request)

- **buffer**：表示发送/接收数据的地址空间
- **type**：表示发送数据的类型，包括 **MPI_CHAR，MPI_SHORT，MPI_INT，MPI_LONG，MPI_DOUBLE，…**
- **count**：表示要发送或接收的特定类型的数据元素的数量
- **comm**：表示通信域
- **source/dest**：表示发送者/接收者的rank（任务ID）
- **tag**：是程序员分配的用于唯一标识消息的任意非负整数。发送和接收操作必须匹配相同的消息标签。**MPI_ANY_TAG** 是通配符
- **status**：操作后状态
- **request**：用于非阻塞的发送和接收操作

MPI数据类型：MPI_datatype

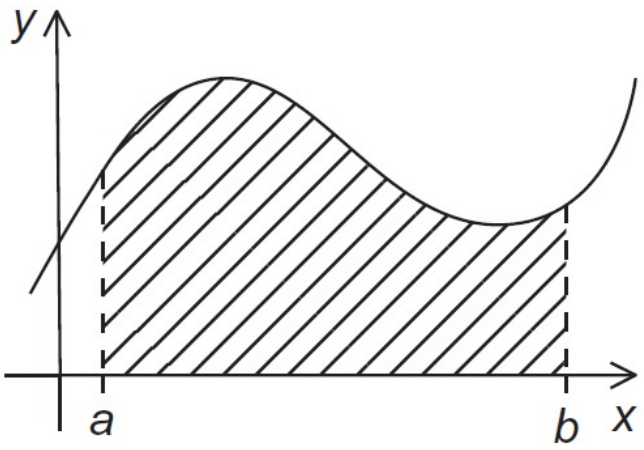
MPI中预定义的数据类型

MPI（C语言）	C语言	MPI（Fortran语言）	Fortran语言
MPI_CHAR	char	MPI_INTEGER	INTEGER
MPI_SHORT	short int	MPI_REAL	REAL
MPI_INT	int	MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_LONG	long int	MPI_COMPLEX	COMPLEX
MPI_UNSIGNED_CHAR	unsigned char	MPI_DOUBLE_COMPLEX	DOUBLE_COMPLEX
MPI_UNSIGNED_SHORT	unsigned short	MPI_LOGICAL	LOGICAL
MPI_UNSIGNED_INT	unsigned int	MPI_CHARACTER	CHARACTER(1)
MPI_UNSIGNED_LONG	unsigned long	MPI_BYTE	
MPI_FLOAT	float	MPI_PACKED	
MPI_DOUBLE	double		
MPI_LONG_DOUBLE	long double		
MPI_BYTE			
MPI_PACKED			

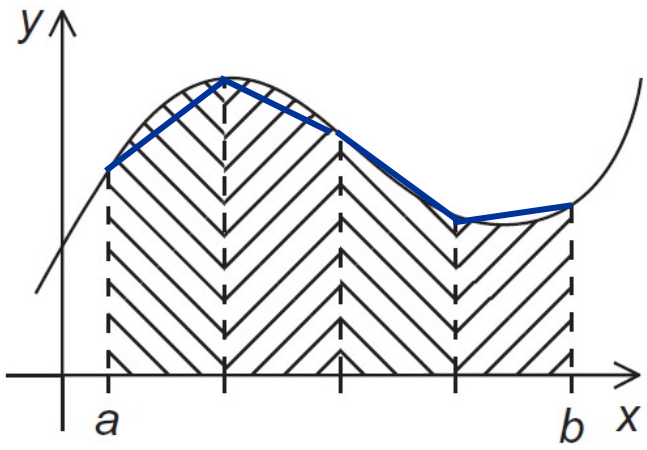
MPI并行编程示例

示例程序：The Trapezoidal Rule（梯形法则）

- 用复合梯形法则来计算积分



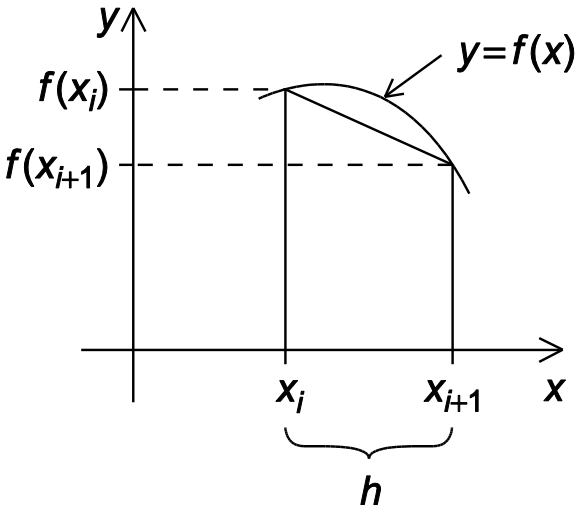
(a)



(b)

示例程序：The Trapezoidal Rule（梯形法则）

区间[a，b]分为n个子区间



一个梯形的面积 = $\frac{h}{2}[f(x_i) + f(x_{i+1})]$

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

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

所有梯形面积的总和 =

$$h\left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

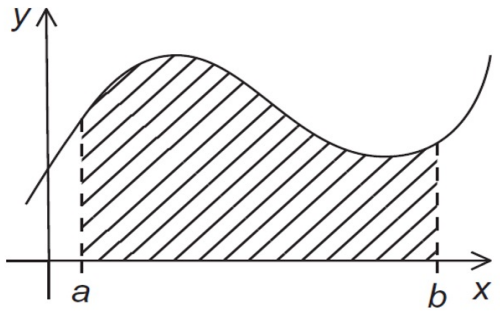
示例程序：The Trapezoidal Rule（梯形法则）

- 串行伪代码

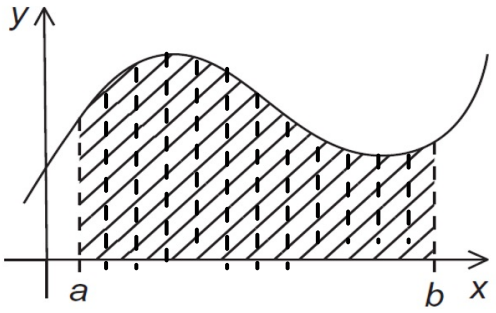
```
/** 输入 a, b, n */
Get a, b, n;
h = (b - a) / n;
approx = (f(a) + f(b)) / 2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i * h;
   approx += f(x_i);
}
approx = h * approx;
```

梯形法则的并行化策略

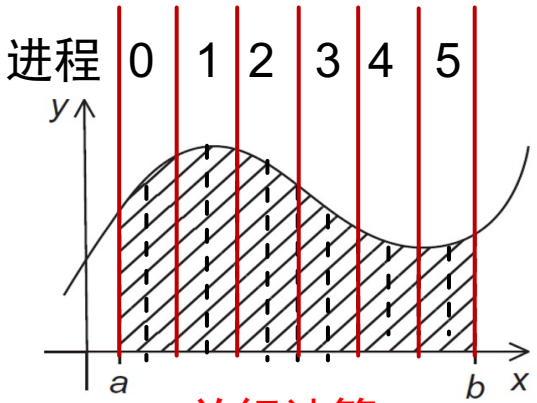
- 将问题划分为子任务并确定子任务之间的关系
- 将子任务映射到不同的计算进程
- 各进程并行计算各自任务并将结果汇总



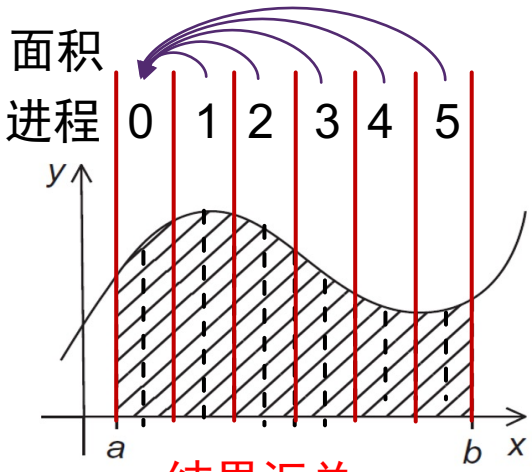
初始问题



任务划分



并行计算



结果汇总

示例程序：The Trapezoidal Rule（梯形法则）

- 并行程序伪代码

```
Get a, b, n;
/** 将计算任务划分至各个进程中 */
h = (b - a) / n;
local_n = n / comm_size; /** local_n 是每个进程计算的梯形数 */
local_a = a + my_rank * local_n * h; /** 计算起点 */
local_b = local_a + local_n * h; /** 计算终点 */
```

```
/** 每个进程分别计算对应的计算任务 */
local_integral = Trap(local_a, local_b, local_n, h);
```

```
/** 将各个进程的 local_integral 汇总至 0号进程 */
if (my_rank != 0)
    Send local_integral to process 0;
else
    total_integral = local_integral;
    for (proc = 1; proc < comm_size; proc++)
        Receive local_integral from proc;
        total_integral += local_integral;
```

```
/** 0号进程输出汇总结果 */
if (my_rank == 0)
    print result;
```

示例程序：The Trapezoidal Rule（梯形法则）

```
int main(){
    int my_rank, comm_size, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_integral, total_integral;
    int source;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
```

```
    /** 将计算任务划分至各个进程中 */
    h = (b - a) / n;
    local_n = n / comm_size; /** local_n 是每个进程计算的梯形数 */
    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
```

```
    /** 每个进程分别计算对应的计算任务 */
    local_integral = Trap(local_a, local_b, local_n, h);
```

示例程序：The Trapezoidal Rule（梯形法则）

```
    /** 将各个进程的 local_integral 汇总至 0号进程 */
    if (my_rank != 0) {
        MPI_Send(&local_integral, 1, MPI_DOUBLE, 0, 0,
                                                MPI_COMM_WORLD);
    } else {
        total_integral = local_integral;
        for (source = 1; source < comm_size; source ++) {
            MPI_Recv(&local_integral, 1, MPI_DOUBLE, source, 0,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_integral += local_integral;
        }
    }
```

从每个进程接收消息

```
    /** 0号进程输出汇总结果 */
    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_integral);
    }
    MPI_Finalize();
    return 0;
} /** main */
```

0号进程输出结果

示例程序：The Trapezoidal Rule（梯形法则）

```
/** 每个进程的本地运算 */
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 */
```

归约过程

```
/** 将各个进程的 local_integral 汇总至 0号进程 */
if (my_rank != 0) {
    MPI_Send(&local_integral, 1, MPI_DOUBLE, 0, 0,
                                                MPI_COMM_WORLD);
} else {
    total_integral = local_integral;
    for (source = 1; source < comm_size; source ++) {
        MPI_Recv(&local_integral, 1, MPI_DOUBLE, source, 0,
                                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_integral += local_integral;
    }
}
```

这个过程是否有更简单的办法？

集合通信

集合通信（Collective）

- 集合通信提供了一种更高级的方法以表示多个进程间的通信
- 集合通信由通信域中的所有进程共同调用，每个进程执行相同的通信操作
- MPI提供了一系列丰富的集合通信操作
  - 包含: **Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv**
  - **All** 前缀的操作表示将数据转发至所有进程中，而不只是单个进程
  - **v** 后缀的操作支持不同进程收发不同大小的数据
- 在许多数值算法中，可以用 **Bcast / Reduce** 代替点对点通信 Send / Recv，从而提高简便性和运行效率
  - 很多MPI版本的集合通信底层通过点对点通信实现
  - 典型集合通信实现算法-下节课详细介绍

MPI集合通信

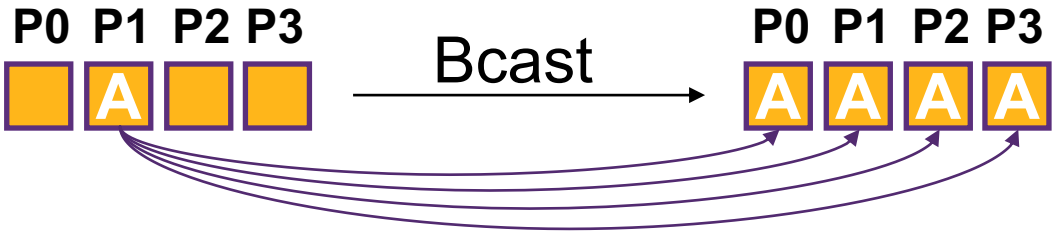
集合通信	操作内容
MPI_Bcast	将一个进程的消息广播至组内所有进程
MPI_Reduce	将组内所有进程的消息归约至一个进程
MPI_Gather	将组内每个进程的不同消息收集到一个目标进程
MPI_Scatter	将一个进程的不同消息分发到组内所有进程
MPI_Scan	对数据进行前置规约
MPI_Allgather	将消息串联到组内所有进程
MPI_Allreduce	执行归约操作并将结果存在组内所有进程中
MPI_Alltoall	将所有进程的数据转发至所有进程

不但要知道这些函数的意思，同时理解常见并发操作类型

MPI集合通信：MPI_Bcast（广播）

- MPI_Bcast (&buffer, count, datatype, root, comm)
  - 从rank为**root**的进程广播（发送）消息到组内所有其他进程

root = 1; count = 1;



MPI集合通信：MPI_Bcast（广播）

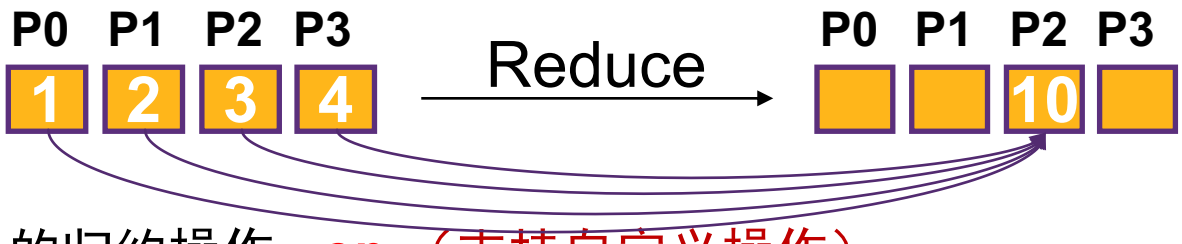
```
int main(int argc, char **argv){
    int rank, value;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    do {
        /** 进程 0读入需要广播的数据 */
        if (rank == 0)
            scanf("%d", &value);

        /** 将该数据广播出去 */
        MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);

        /** 各进程打印收到的数据 */
        printf("Process %d got %d\n", rank, value);
    }   while (value >= 0);
    MPI_Finalize();
    return 0;
}
```

理解典型的 SPMD 程序风格

MPI集合通信：MPI_Reduce（归约）

- MPI_Reduce (&sendbuf, &recvbuf, count, datatype, **op**, dest, comm)
  - 对组中的所有进程执行归约操作并将结果汇总在一个进程中

dest = 2; count = 1; op = MPI_SUM



- 预定义的归约操作 - **op** （支持自定义操作）

MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_SUM	Sum	MPI_PROD	Product
MPI_LAND	Logical AND	MPI_BAND	Bit-wise AND
MPI_LOR	Logical OR	MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR	MPI_BXOR	Bit-wise XOR

MPI集合通信：MPI_Scan

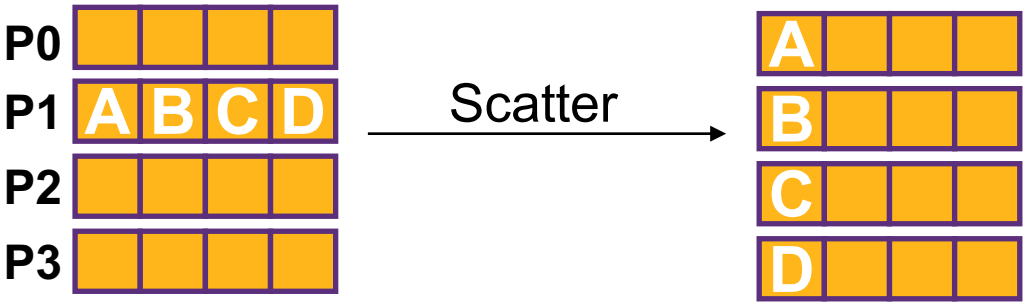
- MPI_Scan (&sendbuf, &recvbuf, count, datatype, **op**, comm)
  - 计算前缀归约

count = 1; op = MPI_SUM

MPI集合通信：MPI_Scatter（分散）

- MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)
  - 将不同的消息从root进程分发到组内所有进程

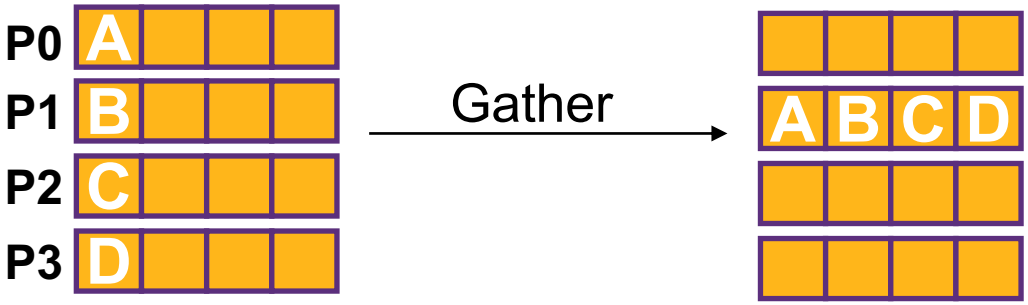
root = 1; sendcnt = recvcnt = 1;



MPI集合通信：MPI_Gather（收集）

- MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)
  - 收集组中的每个进程的不同消息到一个目标进程的操作
  - 此操作是MPI_Scatter的反向操作

root = 1; sendcnt = recvcnt = 1;

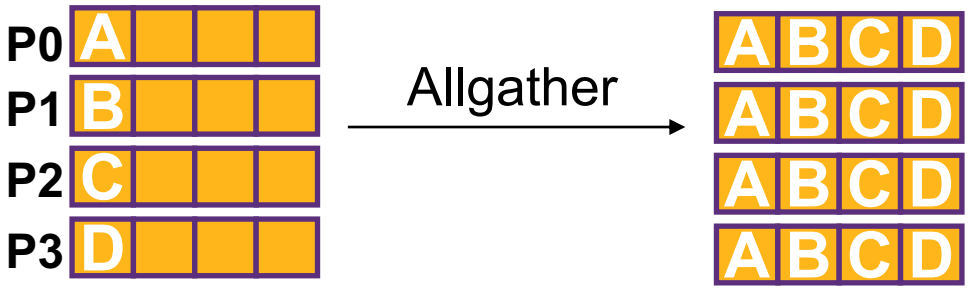


拼接操作：concatenate

MPI集合通信：MPI_Allgather

- MPI_Allgather (&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)
  - 将数据串联到所有进程
  - 等效于MPI_Gather操作后再进行MPI_Bcast操作

sendcnt = recvcnt = 1;



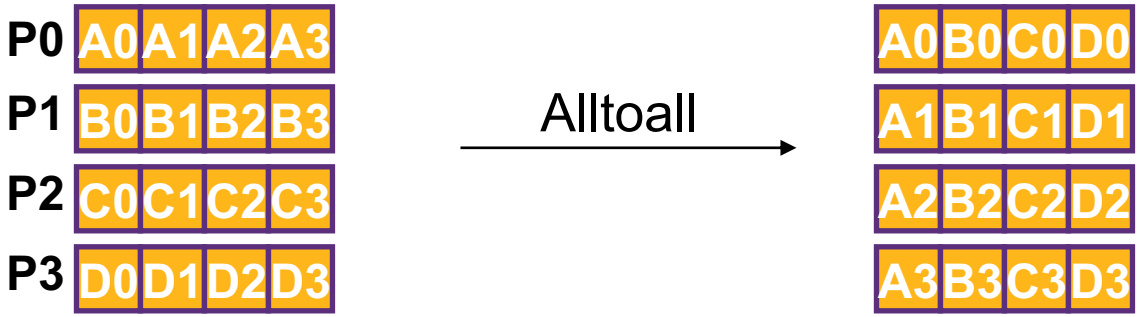
MPI集合通信：MPI_Allreduce

- MPI_Allreduce(&sendbuf, &recvbuf, count, datatype, op, comm)
  - 执行归约操作并将结果存至所有进程
  - 等效于MPI_Reduce操作后再进行MPI_Bcast操作

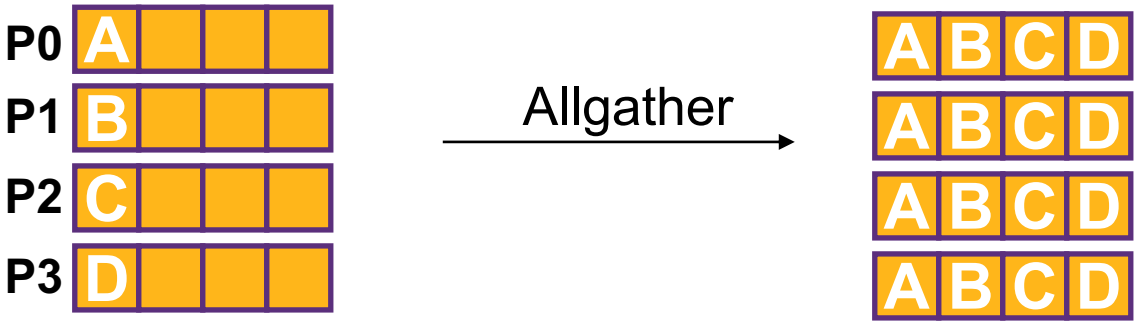
count = 1; op = MPI_SUM

MPI集合通信：MPI_Alltoall

- MPI_Alltoall((&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)
  - 将组内所有进程的数据转发至组内所有进程
  - 相当于每个进程都发送数据给其它所有进程
  - 等效于n次MPI_Gather操作（n为组内进程数）

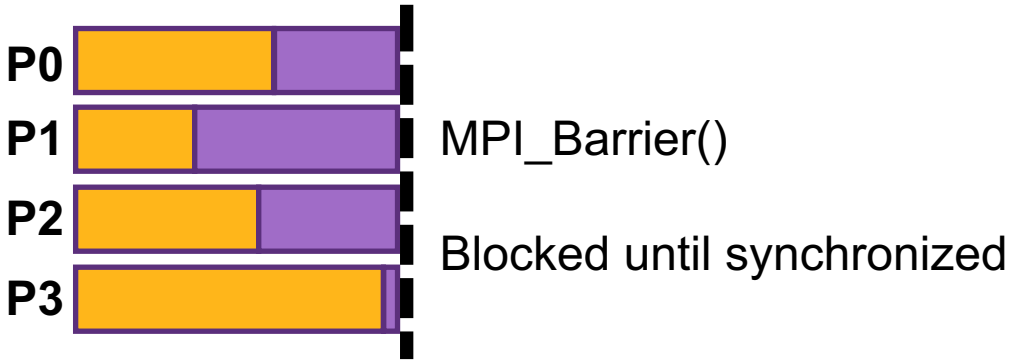


  - 与MPI_Allgather的区别如下



MPI集合通信：MPI_Barrier

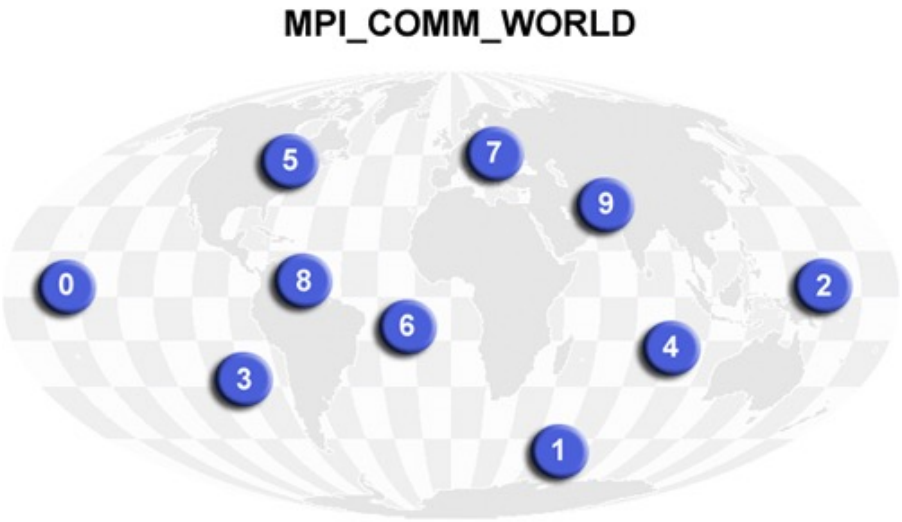
- MPI_Barrier (comm)
  - 在组内创建barrier同步
  - 阻塞直到组内所有进程到达相同的MPI_Barrier调用
  - 并行程序中执行同步操作使用



通信域

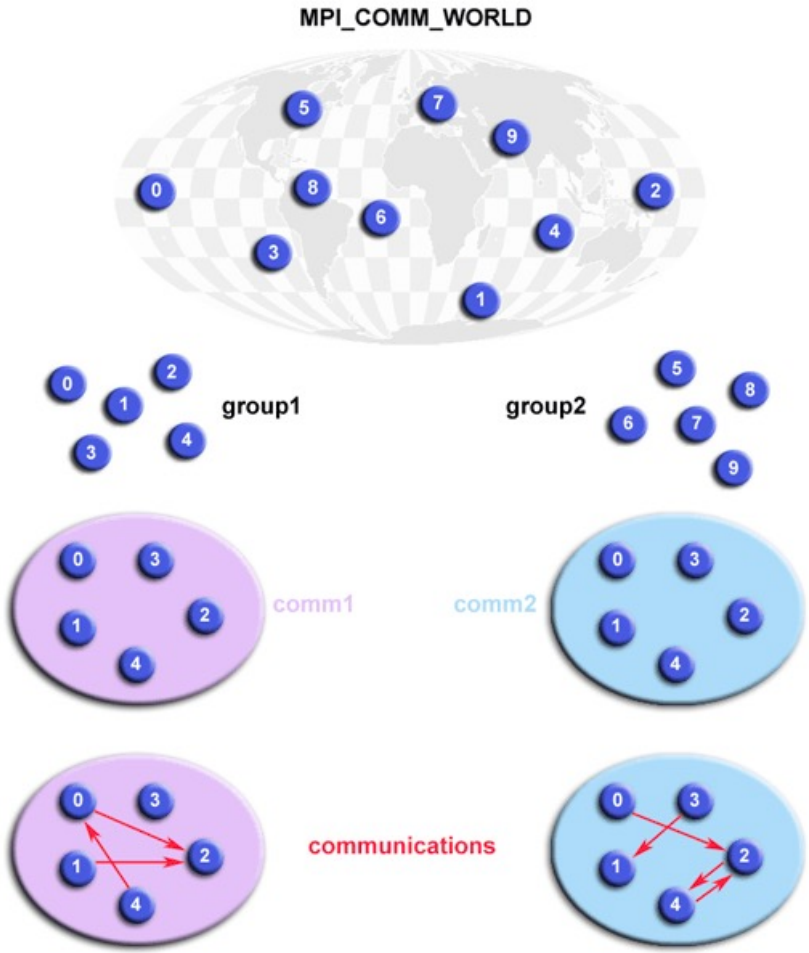
MPI通信域和组

- 通信域和组（Communicator and group）：
  - 组：定义了哪些进程之间可以相互通信
  - 通信域：每个组与一个通信域关联，以执行其通信函数调用
  - MPI_COMM_WORLD 是为所有进程预先定义的通信域
- 任务ID（Rank）
  - 一个通信域中每个进程有唯一标识符（任务ID）
  - 初始化时由系统分配
  - 连续并从零开始



MPI通信域和组的管理

- 组（group）和通信域（communicator）数据类型
  - MPI_Group
  - MPI_Comm
- MPI_Comm_group (Comm, &Group)
  - 访问通信域关联的组
- MPI_Group_incl (Group, size, ranks[], &NewGroup)
  - 通过包含现有组的成员子集来创建组
- MPI_Comm_create (Comm, NewGroup, &NewComm)
  - 创建一个新的通信域
  - 新的通信域必须是原始组的子集



可以在一个组上创建不同的通信域

示例程序：MPI进程分为两个组

```
int rank, numtasks;
MPI_Group orig_group, new_group;
MPI_Comm new_comm;

MPI_Init();
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

```
/** 提取原始组的 group handler：orig_group */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

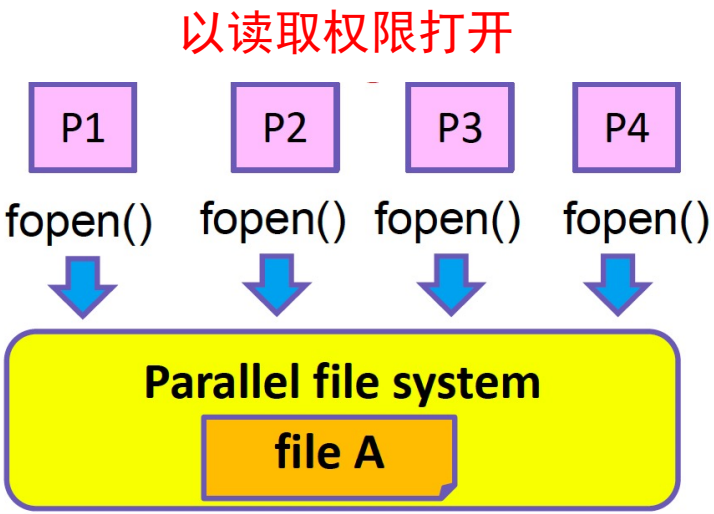
```
/** 根据任务 ID（rank）将进程分为两组 */
int rank1[4] = {0,1,2,3}, rank2[4] = {5,6,7,8};
if (rank < numtasks/2)
    MPI_Group_incl(orig_group, numtasks/2, ranks1, &new_group);
else
    MPI_Group_incl(orig_group, numtasks/2, ranks2, &new_group);
```

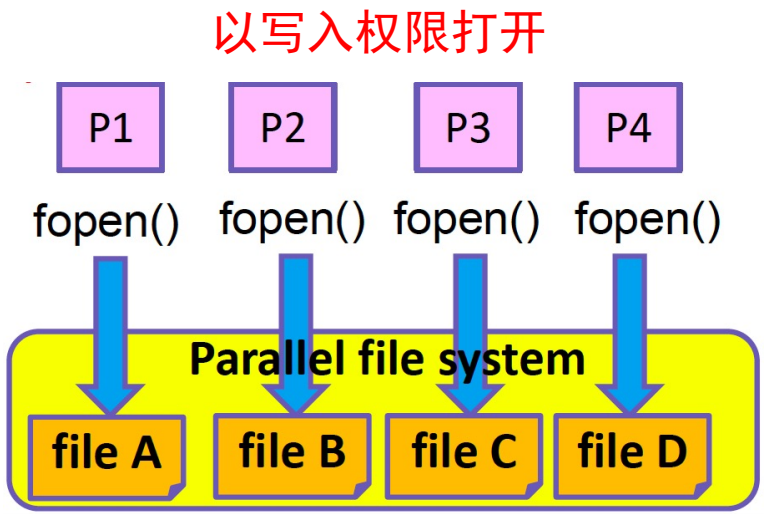
```
/** 创建新的通信域 & 在新的组内广播 */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Bcast(new_comm);
MPI_Finalize();
```

MPI-IO

POSIX 文件访问限制

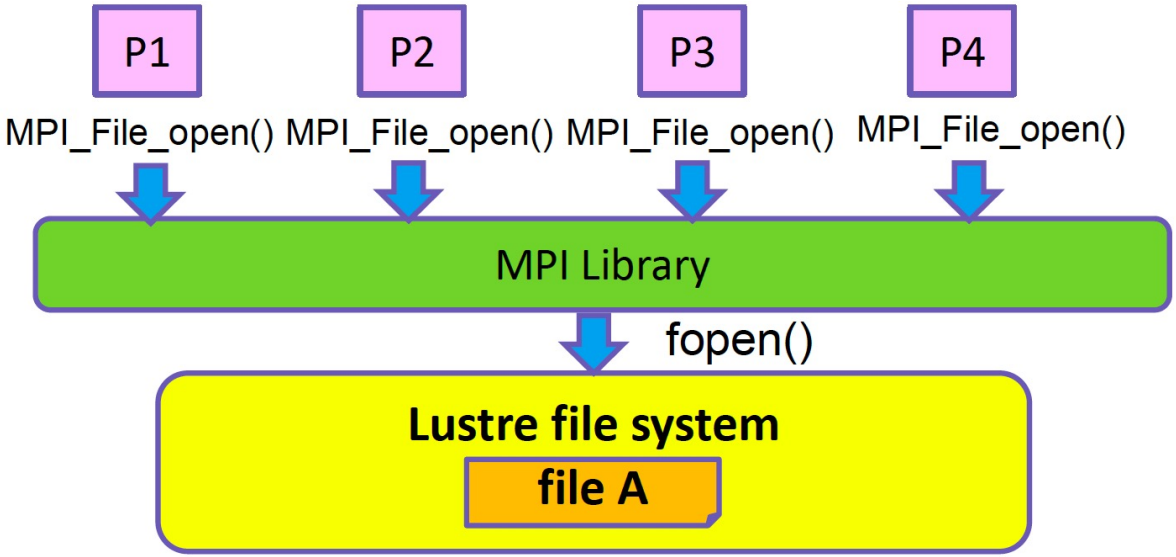
- POSIX文件系统调用 fopen():
  - 不同进程打开同一个文件 -- MPI进程中包含多个文件句柄
  - 以读取权限打开同一个文件是可行的（左下图）
  - 以写入权限打开同一个文件是不可行的；文件系统锁定机制保证数据一致性
  - 同时写入，则必须创建多个文件（无法发挥底层并行文件系统性能且难以管理）（右下图）





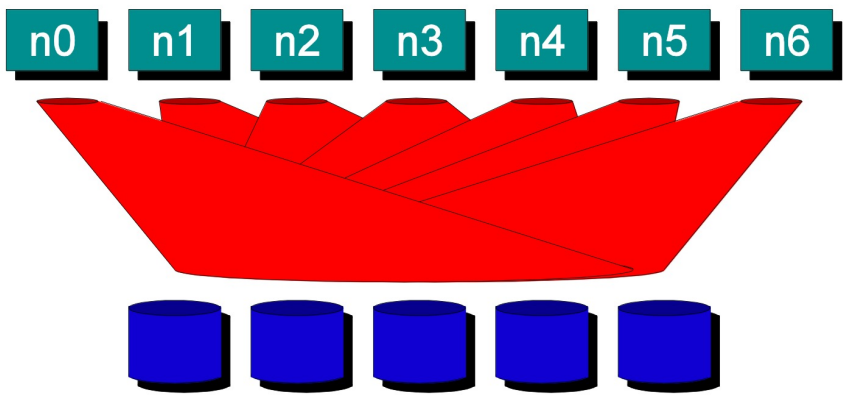
MPI-IO 文件访问操作

- MPI-IO调用 MPI_File_open()
  - 不同进程文件仅在底层打开一次
  - MPI库将共享并彼此同步以使用相同的文件句柄
  - 可以同时进行读写操作

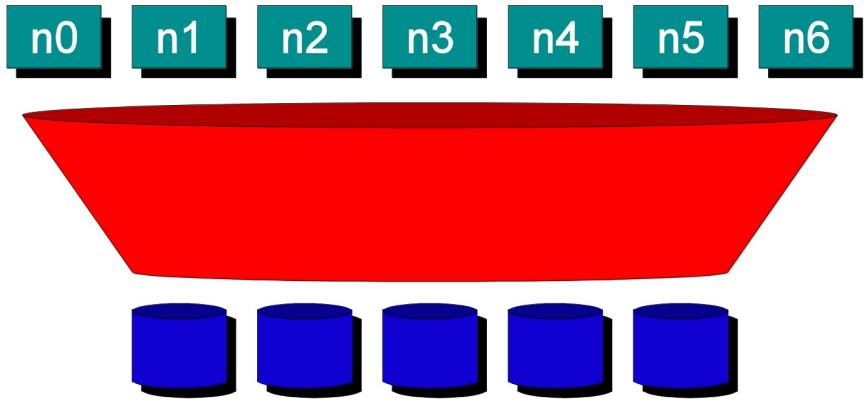


MPI-IO Independent & Collective I/O

- Independent I/O
  - 每个进程独立访问I/O
  - 没有明显的顺序或存取结构
- Collective I/O（类似集合通信）
  - 一组进程以协调方式进行I/O访问
  - 需要所有参与进程一起调用
  - 底层并行文件系统能够进行更多的性能优化



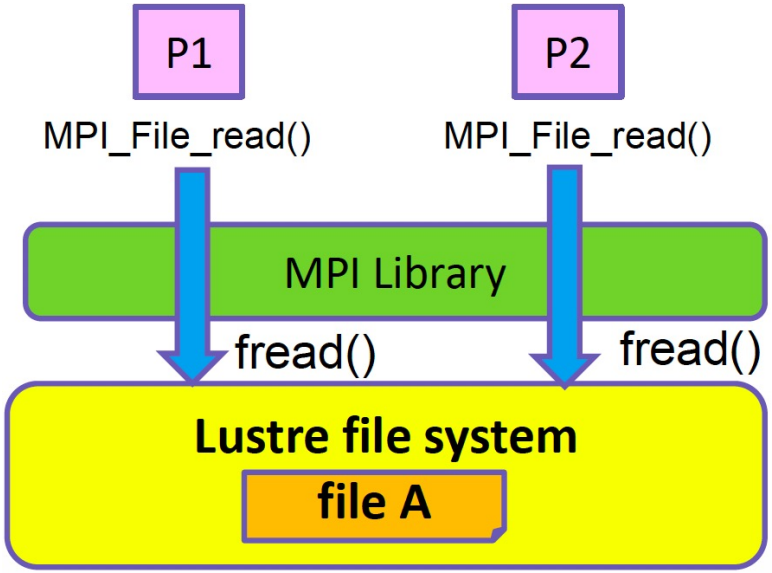
Independent I/O



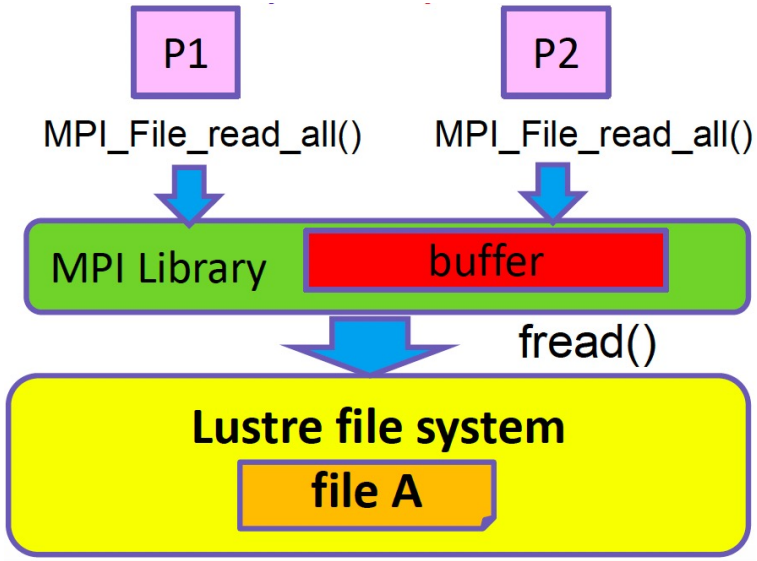
Collective I/O

MPI-IO Independent & Collective I/O

- Independent I/O
  - 单独读/写、无需同步
  - 每个进程一个文件请求
  - 如果访问相同的OST（存储服务器），则请求被串行处理
  - 适合大 I/O 请求



- Collective I/O
  - 读/写共享内存缓冲区，然后发出一个文件请求
  - 减少I/O请求个数
    - 适合小 I/O 请求
  - 不同进程需要同步



MPI-IO API

- MPI_File_open (MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)
  - 打开一个文件
- MPI_File_close (MPI_File *fh)
  - 关闭一个文件
- MPI_File_read/write (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
  - Independent 读/写操作
- MPI_File_read/write_all (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
  - Collective 读/写操作
- MPI_File_sync (MPI_File fh)
  - 将所有新的写操作刷新至存储设备中

MPI-IO API

- MPI_File_seek (MPI_File fh, MPI_Offset offset, int whence)
  - 该函数中的 whence 通过以下参数更新文件指针 fh：
    - MPI_SEEK_SET：指针被设为偏移量
    - MPI_SEEK_CUR：指针被设为当前指针位置加偏移量
    - MPI_SEEK_END：指针被设为文件末尾位置加偏移量
- MPI_File_read_at (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
  - 从偏移量 offset 开始读取文件
  - 等效于 MPI_File_seek 加上 MPI_File_read

示例程序：用 MPI-IO 读文件

```
#include <stdio.h>
#include <mpi.h>
#define FILESIZE 80
int main(int argc, char **argv){
     int rank, size, bufsize, nints;
     MPI_File fh;
     MPI_Status status;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     /** 计算每个进程读入数据的大小 */
     bufsize = FILESIZE / size;
     nints = bufszie / sizeof(int);
     int buf[nints];
     /** 打开文件 */
     MPI_File_open(MPI_COMM_WORLD, "file", MPI_MORE_RDONLY, MPI_INFO_NULL, &fh);
     /** 定位指针并读取数据 */
     MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
     MPI_File_read(fh, buf, nints, MPI_INT, &status);
     printf("rank: %d, buf[%d]: %d", rank, rank * bufsize, buf[0]);
     /** 关闭文件 */
     MPI_File_close(&fh);
     MPI_Finalize();
     return 0;
}
```

示例程序：用 MPI-IO 读文件

```c
#include <stdio.h>
#include <mpi.h>
#define FILESIZE 80
int main(int argc, char **argv){
    int rank, size, bufsize, nints;
    MPI_File fh;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /** 计算每个进程读入数据的大小 */
    bufsize = FILESIZE / size;
    nints = bufszie / sizeof(int);
    int buf[nints];

    /** 打开文件 */
    MPI_File_open(MPI_COMM_WORLD, "file", MPI_MORE_RDONLY, MPI_INFO_NULL, &fh);

    /** 定位指针并读取数据 */
    MPI_File_read_at(fh, rank * bufsize, buf, nints, MPI_INT, &status);
    printf("rank: %d, buf[%d]: %d", rank, rank * bufsize, buf[0]);

    /** 关闭文件 */
    MPI_File_close(&fh);
    MPI_Finalize();
    return 0;
}
```

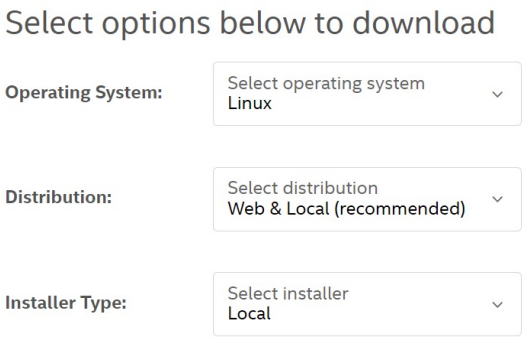
MPI程序编译与运行

基于Linux系统安装使用OpenMPI

- 以 Ubuntu / Debian 系统为例
- 安装：
  - `sudo apt install openmpi-bin`
- 无需任何配置即可使用
- 编译命令：
  - 使用 `mpicc/mpicxx/mpifort` 编译程序
  - 分别对应 C / C++ / Fortran
- 较为简单，推荐在本地尝试时使用
- 在实验集群上请严格按照手册指示操作

基于Linux系统安装Intel MPI

- 需要先下载对应的 Intel oneAPI Base Toolkit
  - https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit/download.html
  - 建议选择 local 版本
- 再下载 oneAPI HPC Toolkit
  - https://software.intel.com/content/www/us/en/develop/tools/oneapi/hpc-toolkit/download.html
- 安装
  - 先安装 Base Toolkit（文件名以实际下载为准）
    - `sudo ./l_BaseKit_p_2021.1.0.2659_offline.sh`
    - 使用默认选项，一路选择下一步即可
  - 再安装HPC Toolkit
    - `sudo ./l_HPCKit_p_2021.1.0.2684_offline.sh`



基于Linux系统使用Intel MPI

- 加载 MPI 环境
  - `source /opt/intel/oneapi/mpi/latest/env/vars.sh`
  - 可以将上述语句加入 `~/.bashrc` 以在登录时自动加载
- MPI 程序的编译
  - 使用 `mpiicpc/mpiicc/mpiifort/mpiifort` 编译程序
  - 分别对应 C++ / C / F77 / F90
- 注意：不同 MPI 实现编译的文件不能混用
  - OpenMPI 编译的文件只能用 OpenMPI 运行
  - Intel MPI 编译的文件只能用 Intel MPI 运行
  - 可以使用 `ldd ./exec_files` 的方式确认
    - 根据 `libmpi.so` 的路径确认使用的 `MPI` 实现

示例程序

```
int main() {
    char greeting[MAX_STRING];
    int comm_sz, my_rank;

    MPI_Init();
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!",
                                                my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
                                                MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!", my_rank,
                                                    comm_sz);
        for (int s = 1; s < comm_sz; s++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, s, 0,
                                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }
    MPI_Finalize();
    return 0;}
```

MPI 程序的编译与运行

- 编译命令
  - mpiicc/mpicc -g -o mpi_hello mpi_hello.c
- 运行命令
  - mpirun -n <number of processes> <executable>
  - mpirun -n 1 ./mpi_hello（运行1个进程）
  - mpirun -n 4 ./mpi_hello （运行4个进程）

命令行控制运行的进程数量

- 运行结果

```
mpirun -n 1 ./mpi_hello
```

```
Greetings from process 0 of 1 !
```

```
mpirun -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典型实现

MPI的典型实现

- 常见的 MPI 实现:
  - MPICH　　阿贡国家实验室
  - Intel MPI　　Intel (Based on MPICH)
  - MVAPICH　　Ohio州立大学 (Based on MPICH)
  - OpenMPI　　Indiana 大学

- MPICH是MPI在各种机器上的可移植实现
  - 可以安装在几乎所有平台上
  - 台式机、工作站、SMP服务器等
- OpenMPI 是目前较有影响力的开源实现
  - 版本更新快、采用技术新

总结

- MPI背景
- SPMD
- MPI编程基础
- 点对点通信
- 集合通信
- MPI-IO
- MPI程序编译与运行

课后练习

- 阅读《高性能计算之并行编程技术》 6-8章、12-13章
- 编译并运行课件示例代码：
  - 38页、46页、72页
  - 补齐变量声明、头文件等
  - 读懂代码并理解运行过程
  - 38的程序：用集合通信替代重新实现
  - 不计入成绩，但强烈建议课后练习

小作业-2

- 运行分别使用 MPI 阻塞通信和非阻塞通信的两份程序，对比性能

- 目的：

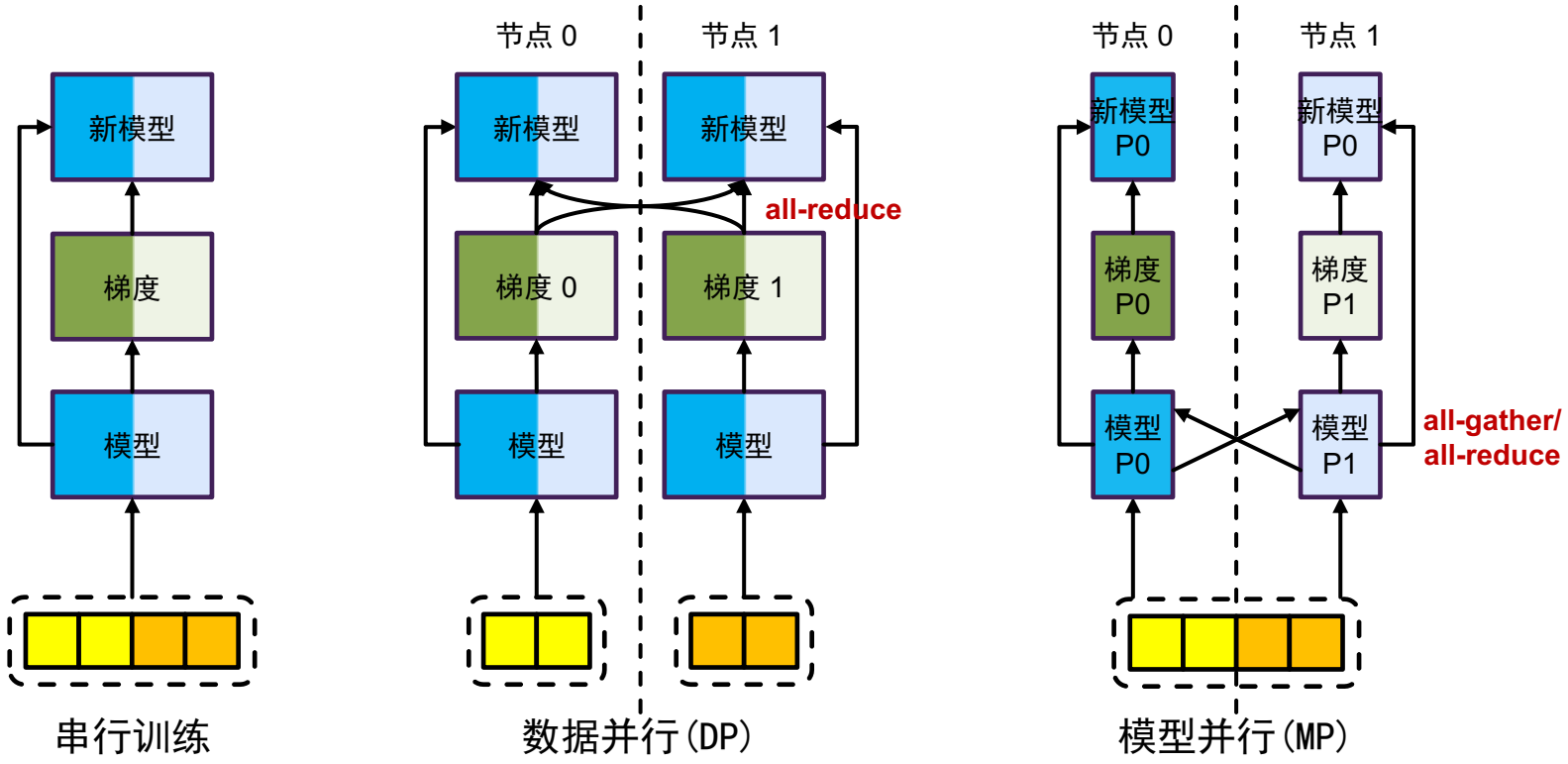
  学习 MPI 非阻塞通信编程方法

  体会计算通信重叠带来的性能提升

- 详见网络学堂发布

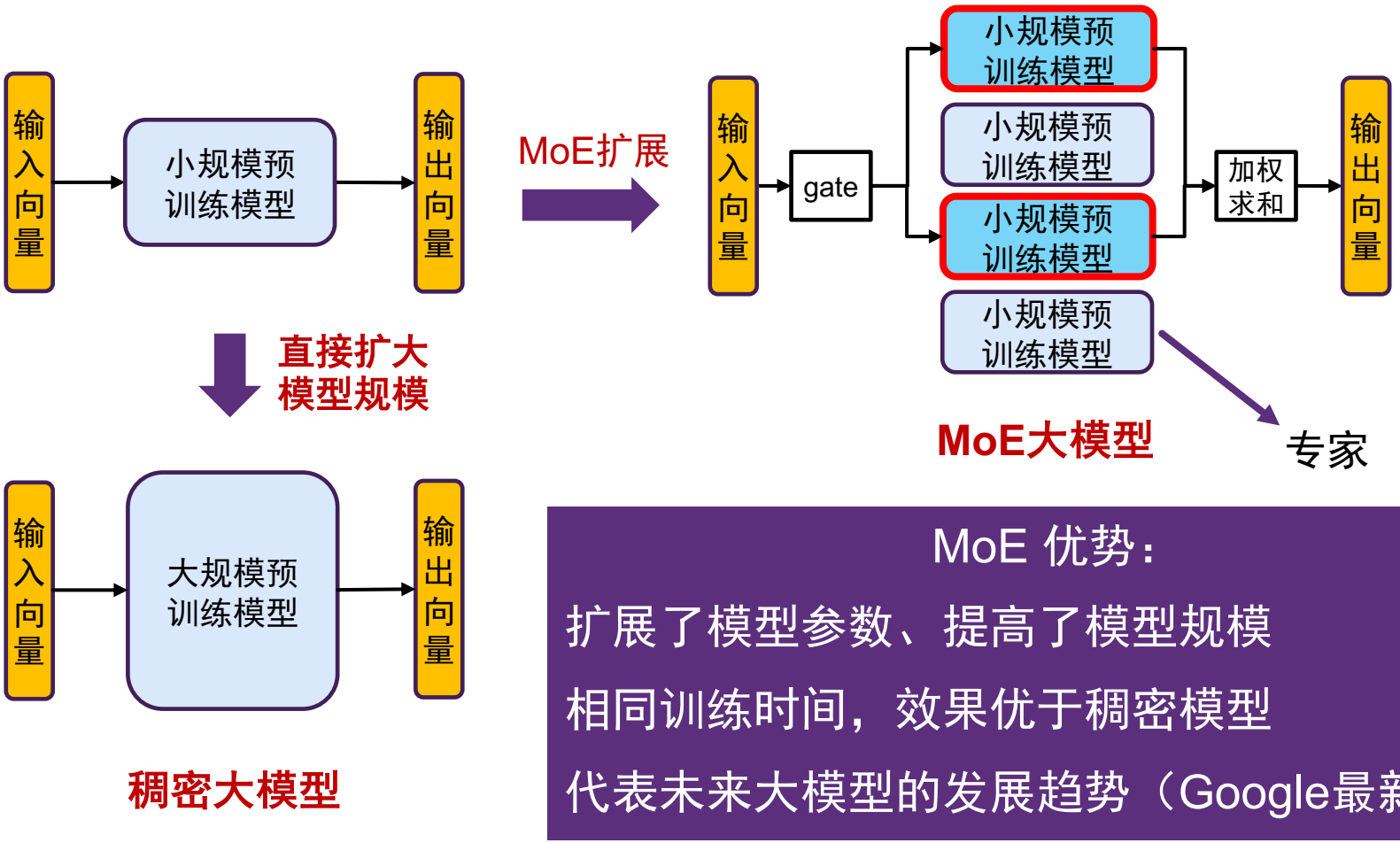
并行训练

- 随着模型规模扩大，训练数据增多
  - 单机训练无法满足参数规模和数据吞吐的需求
  - **并行训练**成为大模型的训练"标配"



混合专家系统 MoE

- 混合专家系统 MoE（Mixture of Experts）
  - 允许不增加样本计算量的同时，增大模型参数量



MoE 优势：

扩展了模型参数、提高了模型规模

相同训练时间，效果优于稠密模型

代表未来大模型的发展趋势（Google最新模型）

MoE 模型并行训练

- MoE 模型并行训练
  - 需要对输入数据划分、专家划分
  - 引入 All-to-All 通信