分布式计算综合实践

进程级并行：MPI编程

Chapter 3 – Process Level Parallel Computing: MPI

线程与进程

回顾在操作系统中的线程与进程概念：

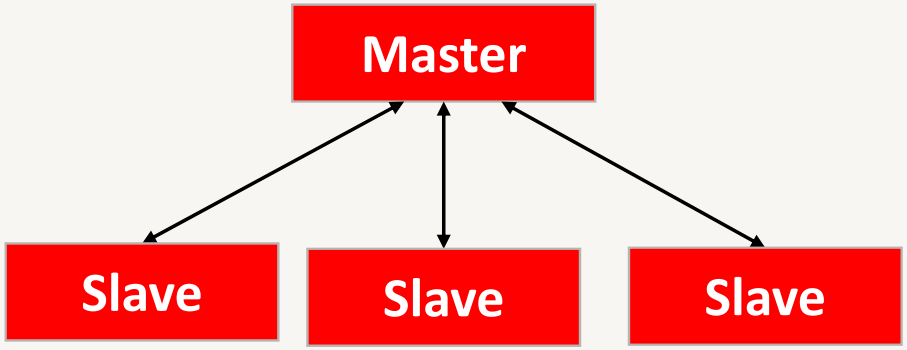
- 进程：一个正在执行程序的实例，包括程序计数器、寄存器和变量的当前值(更独立，有独自的地址空间)
- 线程：轻量级进程，共享地址空间，但各有一套堆栈

进程级并行

- 相比于上一章的线程级并行（内存共享），本章要探讨的进程级并行具有**独立的地址空间**（内存不共享）
- 进程级并行天然地适用于在多个处理器甚至多台机器场景下对任务进行并行处理

多进程工作模式

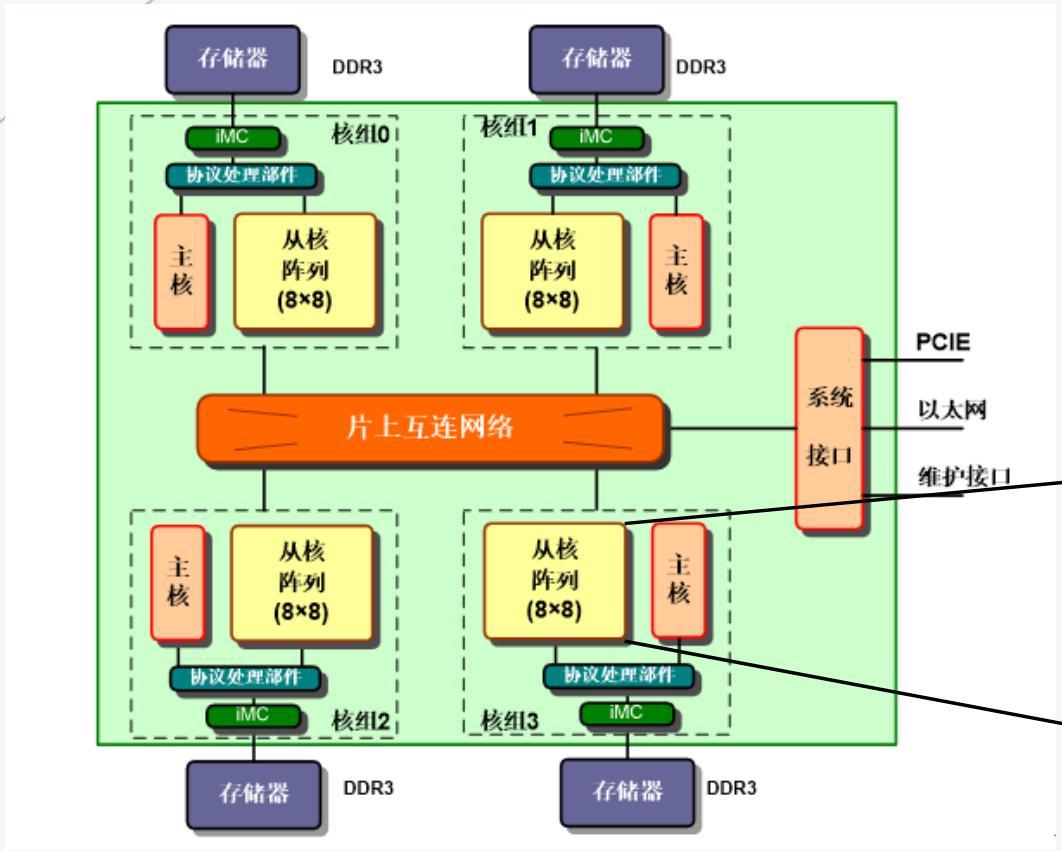
- 多进程并行工作时，有两种常见的思路：
  - 主从模式
  - 单控制流多数据流模式

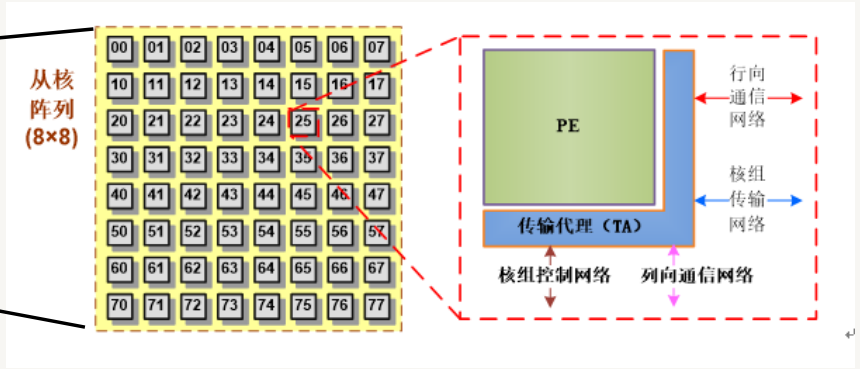


主从模式

- 将一个待求解的任务分成一个主任务（主进程）和一些从任务（子进程）。
- 主进程负责将任务分解、派发和收集各个子任务的求解结果并最后汇总得到问题的最终解。
- 各子进程接收主进程发来的消息；并行进行各自计算；然后向主进程发回各自的计算结果。

申威CPU架构


存储器
DDR3
存储器
DDR3
核组0
核组1
iMC
iMC
协议处理部件
协议处理部件
主核
从核阵列(8×8)
从核阵列(8×8)
主核
片上互连网络
系统接口
PCIE
以太网
维护接口
主核
从核阵列(8×8)
从核阵列(8×8)
主核
协议处理部件
协议处理部件
iMC
iMC
核组2
核组3
存储器
DDR3
存储器
DDR3


从核阵列(8×8)
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
60 61 62 63 64 65 66 67
70 71 72 73 74 75 76 77
PE
传输代理（TA）
行向通信网络
核组传输网络
核组控制网络
列向通信网络

单控制流多数据流模式

- 首先需要将数据预先分配给各个计算进程；
- 然后各个计算进程并行地完成各自的计算任务，包括计算过程中各进程间的数据交换（施行通信同步）；
- 最后才将各计算结果汇集起来。

进程间通信

- 进程间通信（IPC，Inter-Process Communication）让程序员能够协调不同的进程，使之能在一个操作系统里同时运行，并相互传递、交换信息。

并行计算编程模型（上一章）

根据进程交互方式，我们有以下几类并行编程模型：

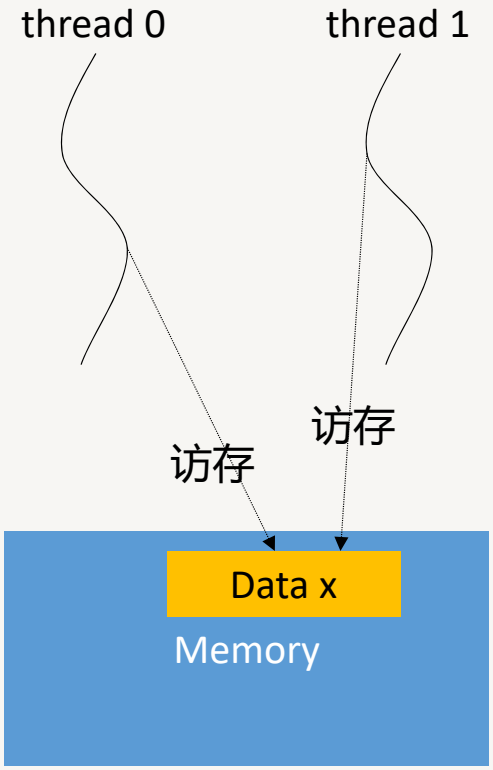
- 隐式交互（完全由编译器实现，这里不展开）
- 共享变量（英特尔Cilk、OpenMP）
- 消息传递（MPI）

共享变量编程存在的隐藏问题（上一章）

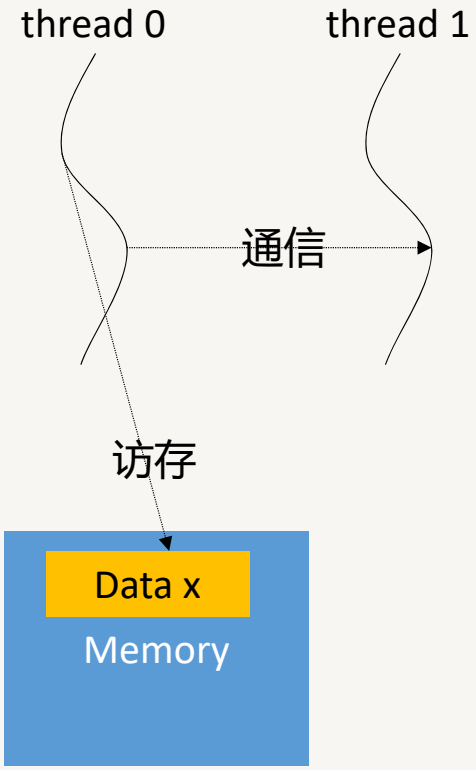
- 在多处理机上使用多线程所面临的主要问题在于线程间如何协调，从而正确操作计算的对象——数据

- 由于数据（从硬件角度看就是内存）属于一种被计算所使用的资源，因此当多个核心共用这些资源（共享变量）时，自然就会因为对数据的竞争（谁先谁后）而产生一些奇怪的问题，归结起来就是：
- *多个线程访问一个变量时，会发生什么？*

- 共享变量


thread 0
thread 1
访存
访存
Data x
Memory

- 消息传递


thread 0
thread 1
通信
访存
Data x
Memory

消息传递模型

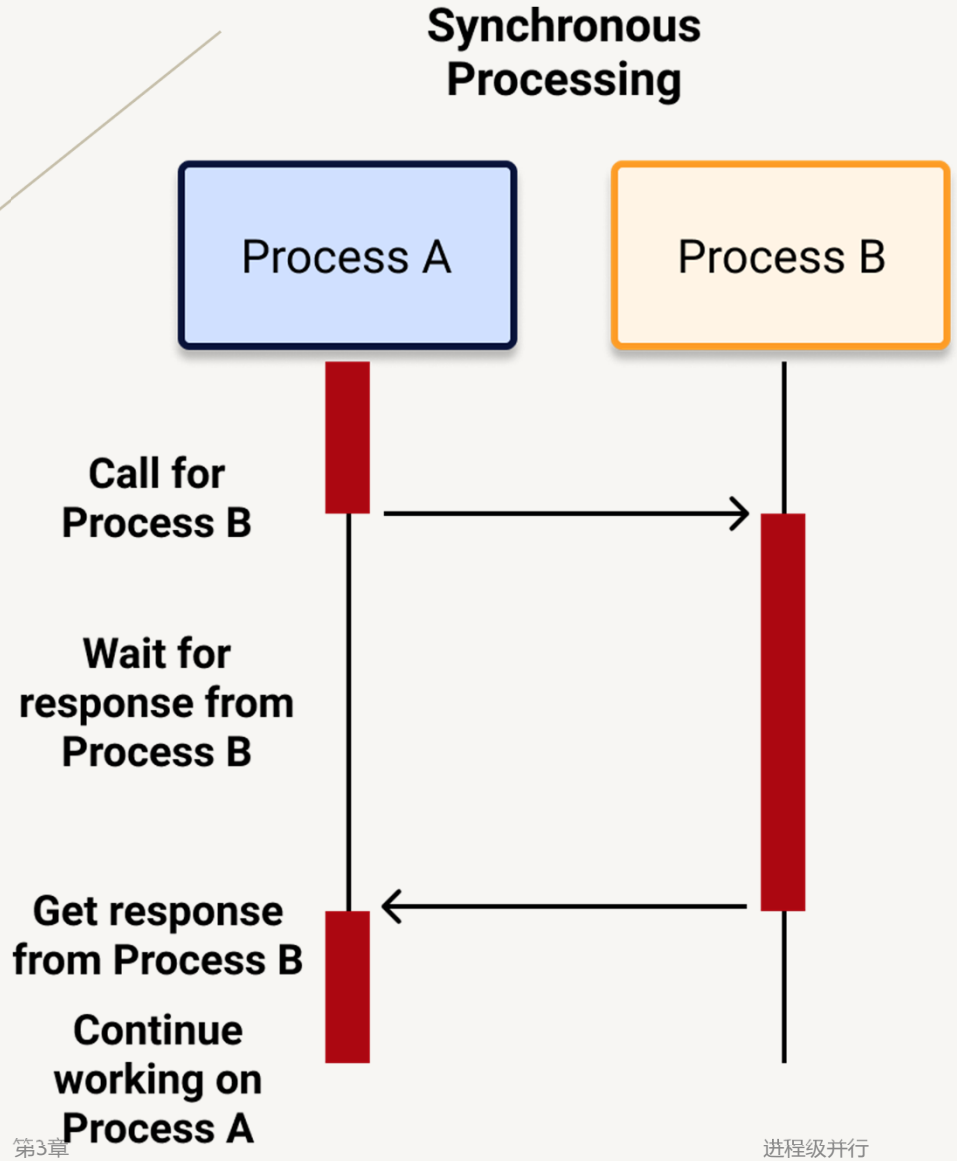
- 在消息传递模型中，并行进程通过相互传递消息来交换数据
- 管道、消息队列、套接字等IPC方式都属于消息传递
- 上一章我们主要讲述了共享变量模型（子任务间共享内存），本章我们将关注消息传递模型（子任务间不能共享内存）。这也是在进程级并行中最简单、最自然的进程交互方式

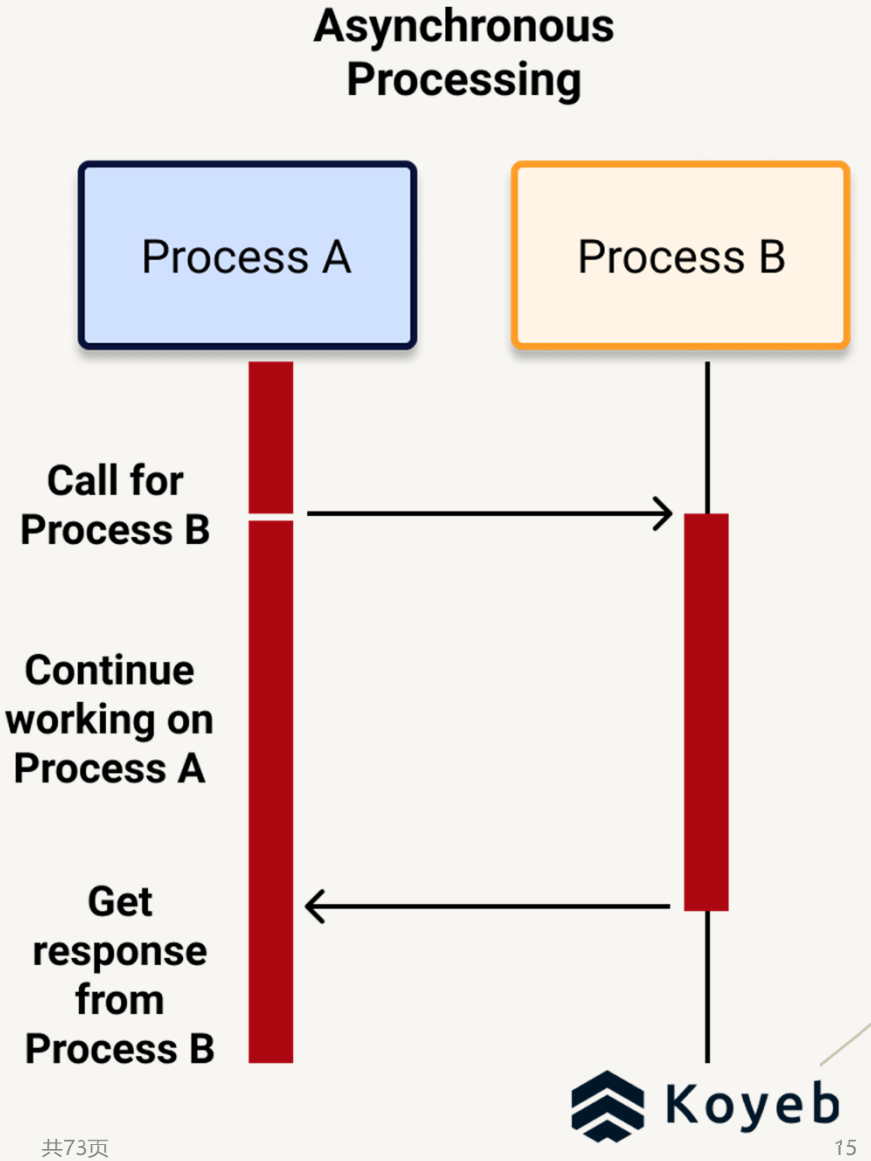
SEND & RECEIVE

- 发送/接收（send/receive）是一对最基本的消息传递原语
- 相当于：
  - Pipe 的 write/read 操作（利用管道实现父子进程间的通信）
  - Message Queue 或 Socket 上的 send/receive 操作
- 我们能接触到的绝大多数消息传递方式，都可以抽象为 send/receive 操作

通信机制：同步 & 异步

- 在消息传递中，同步和异步是发送方对通信的两种不同处理方式
- 同步：消息发送方等待消息接收方完成接收并响应，然后开始处理其他工作
- 异步：发送方在发送消息后不等待响应，直接开始处理其他工作


Synchronous Processing
Process A
Process B
Call for Process B
Wait for response from Process B
Get response from Process B
Continue working on Process A


Asynchronous Processing
Process A
Process B
Call for Process B
Continue working on Process A
Get response from Process B

Koyeb

MPI

- MPI(消息传递接口 Message Passing Interface)是一个跨语言的通讯协议，实现进程间的通讯，用于编写并行计算程序。
- MPI是一个消息传递接口标准，是目前最重要的一个基于消息传递的并行编程工具，它提供一个可移植、高效、灵活的消息传递接口库，以语言独立的形式存在，提供了与C、Fortran和Java语言的绑定，可运行在不同的操作系统和硬件平台上。
- MPI是一个标准，提供很多接口的定义，但不是编程语言

MPI主要版本

- CHIMP Edinburg 大学
- LAN(Local Area Multicomputer) Ohio超级计算中心
- MPICH Argonne国家实验室与Mississippi州立大学
- MPICH是MPI在各种机器上的可移植实现,可以安装在几乎所有的平台上：PC，工作站，SMP，MPP，COW

MPI 对比 OPENMP

MPI	OpenMP
多进程并行	多线程并行
消息传递	共享内存
数据分配方式显式	数据分配方式隐式
单机/多机	单机多核

是否具有独立的地址空间

MPI程序的特点

- 用户必须通过显式地发送和接收消息来实现处理机间的数据交换。
- 每个并行进程均有自己独立的地址空间。
- 并行计算粒度大，特别适合于大规模可扩展并行算法。

OpenMPI

- OpenMPI是一个高性能消息传递库，是MPI的一种开源实现
- OpenMPI支持多个操作系统，支持C/C++或Fortran语言

程序结构初探

```
#include“mpi.h”                          头文件
// ......
int main(int argc, char *argv[])         主程序
{
    // ......

    MPI_Init(&argc,&argv);               MPI程序开始
    // ......
    // 并行代码                          MPI程序主体
    // ......
    MPI_Finalize();                      MPI程序结束

    // 串行代码
}
```

- 常用函数

  - MPI_Init(&argc, &argv)：通知MPI系统进行所有初始化设置，在调用MPI_Init前不应该调用其他MPI函数。
  - MPI_Finalize()：用来清理MPI环境，这个调用之后就没有MPI函数可以被调用了。
  - MPI_Comm_rank(MPI_COMM_WORLD, &rank)：获取当前进程号。
  - MPI_Comm_size(MPI_COMM_WORLD, &size)：获取可用的进程数量。

MPI代码示例（HELLO WORLD）

```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char *argv){
        int myid, numproc;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        printf(“Hello world! I am %d of %d.\n”, myid, numproc);
        MPI_Finalize();
        return 0;
}
```

- 编译程序

```
mpicc -o hello hello.c
```

hello.c为你的代码文件名，hello是输出的可执行文件名

- 运行程序

```
mpirun -np 4 hello
```

运行生成的可执行文件，4指定np的实参，np表示进程数（the number of process），由用户指定。

编译：

```
mpicc -o hello hello.c
```

运行：

```
mpirun -np 4 hello
```

运行结果：

```
Hello world! I am 0 of 4.
Hello world! I am 1 of 4.
Hello world! I am 2 of 4.
Hello world! I am 3 of 4.
```

MPI进程间执行是相互独立的，以上代码中，printf()语句按照随机顺序执行，甚至无法保证一次输出一行。

MPI 通信函数 SEND & RECEIVE

- MPI_Send(
  void* data,
  int count,
  MPI_Datatype datatype,
  int destination,
  int tag,
  MPI_Comm communicator)
  通信域

- MPI_Recv(
  void* data,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm communicator,
  MPI_Status* status)

MPI消息

- 一个消息好比一封信
- 消息的内容，即信的内容，在MPI中称为消息缓冲(Message Buffer)，消息缓冲由三元组<起始地址，数据个数，数据类型>标识
- 消息的接收发送者，即信的地址，在MPI中成为消息封装(Message Envelop)，消息信封由三元组<源/目标进程，消息标签，通信域>标识

```
MPI_Send (buf, count, datatype, dest, tag, comm)
```

消息缓冲　　消息信封

TAG：MPI消息标签

- 为什么需要消息标签？
- 当发送者连续发送两个相同类型消息给同一个接收者，如果没有消息标签，接收者将无法区分这两个消息 没法区分给谁
  - 进程P：send(A,32,Q); send(B,16,Q)
  - 进程Q：recv(X,32,P); recv(Y,16,P)
- 这段代码打算传送A的前32个字节进入X，传送B的前16个字节进入Y。但是，尽管消息B后发送，但可能先到达进程Q，就会被第一个接收函数接收在X中。使用标签可以避免这个错误
  - 进程P：send(A,32,Q,tag1); send(B,16,Q,tag2)
  - 进程Q：recv(X,32,P,tag1); recv(Y,16,P,tag2)

数据类型

- MPI的特性之一是所有的通信函数都带一个数据类型参数，用于描述发送和接受数据的类型
- 你可以创建自己的数据类型
- 使用数据类型参数是为了在异构环境中，例如在具有不同字节存储顺序或者不同基本数据类型长度的异构计算及系统上，当数据类型被指定后，MPI能够在内部转换成相应的字节数，保证通信顺利进行。

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

COMMUNICATOR（通信器）

- 进程可以被划分成组，每个组用来处理某项特定任务
- 每条消息都应该在同样的上下文中发送和接收
- 这样的分组以及消息传递上下文，组合在一起被称为通信器 communicator
- MPI_COMM_WORLD 是默认的通信器，如果没有分组的需求，直接使用这个通信器就可以

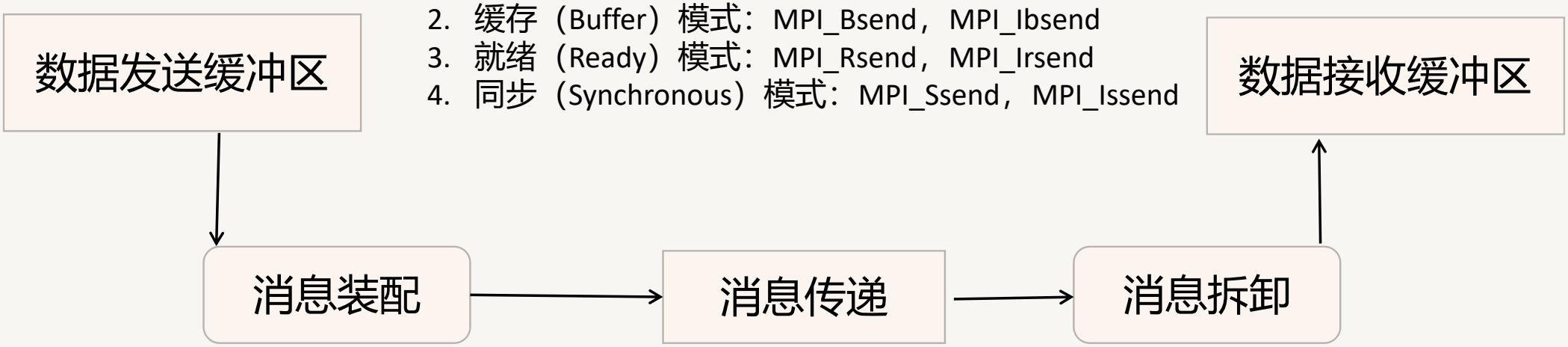
MPI_STATUS（消息的状态）

- 接收方在调用MPI_Recv时，会提供一个MPI_Status结构体作为参数，该结构体包含了一些关于消息的信息：
  - 发送端rank，在MPI_SOURCE字段中（stat.MPI_SOURCE）
  - 消息标签，在MPI_TAG中（stat.MPI_TAG）
  - 消息长度，通过 MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count)函数读出

```
typedef struct _MPI_Status {
  int count;
  int cancelled;
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
} MPI_Status, *PMPI_Status;
```

点到点通信示意图

1. 标准模式：MPI_Send，MPI_Isend
2. 缓存（Buffer）模式：MPI_Bsend，MPI_Ibsend
3. 就绪（Ready）模式：MPI_Rsend，MPI_Irsend
4. 同步（Synchronous）模式：MPI_Ssend，MPI_Issend



阻塞通信（阻塞发送）

阻塞发送意味着，从函数参数列表中指定位置复制数据之前，发送函数不会返回，因此可以在发送函数调用后更改数据而不会影响原始消息。

```
int send_data = 1;
MPI_Send(&send_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
send_data = 2;
```

阻塞发送意味着下面修改send_data的值操作是安全的，对于非阻塞发送，下面这条一句可能会导致接收方收到的值是2而不是1，这显然不符合目的

阻塞通信（阻塞接受）

阻塞接受意味着所有接收到的数据都存储在函数参数列表中指定的变量中之前，接受函数不会返回，因此可以在调用后使用数据并确保所有数据都在那里。

```
int recv_data = 1;
MPI_Recv(&recv_data, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("%d", recv_data);
```

阻塞接受意味着下面 recv_data的操作是安全的，对于非阻塞接受，下面这条语句可能会导致输出的值是1而不是实际接收到的值，这也并不符合预期。

示例： SEND RECEIVE

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

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);
}
```

```
mpirun -np 2 ./send_recv
Process 1 received number -1 from process 0
```

- 以上示例中，进程首先获取自己的进程号。然后根据进程号，0号进程的任务是将-1发送给1号进程；1号进程则是从0号进程接收一个数字并输出

示例：环

```
int token;
if (world_rank != 0) {
    MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_rank - 1);
} else {
    // Set the token's value if you are process 0
    token = -1;
}
MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size,
         0, MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if (world_rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_size - 1);
}
```

- 输出：

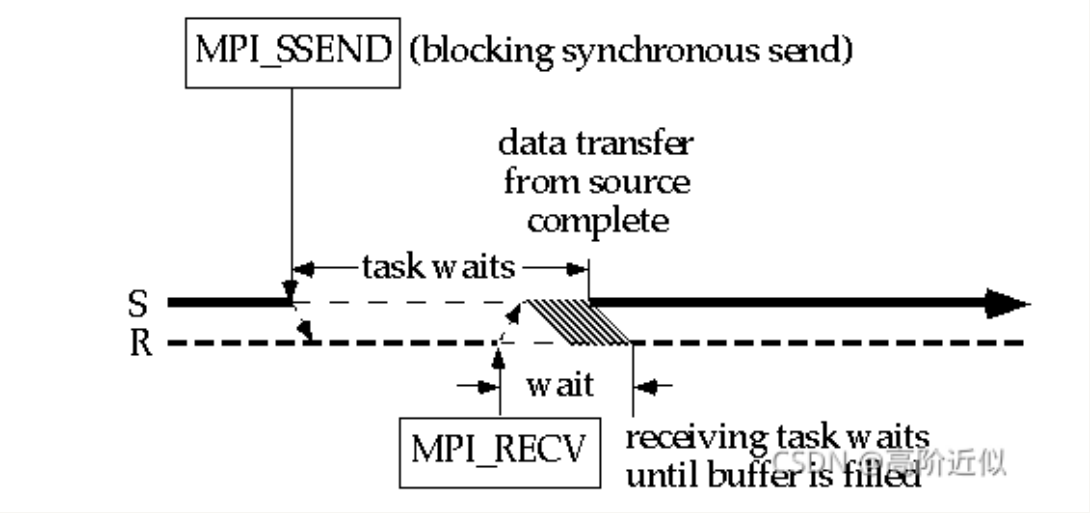
```
Process 1 received token -1 from process 0
Process 2 received token -1 from process 1
Process 3 received token -1 from process 2
Process 4 received token -1 from process 3
Process 0 received token -1 from process 4
```

MPI的四种点对点通信模式

- 通信模式(Communication Mode)指的是缓冲管理，以及发送方和接收方之间的同步方式。
- 共有下面四种通信模式
  - 同步(synchronous)通信模式
  - 缓冲(buffered)通信模式
  - 标准(standard)通信模式（刚刚讲的）
  - 就绪(ready)通信模式

同步通信

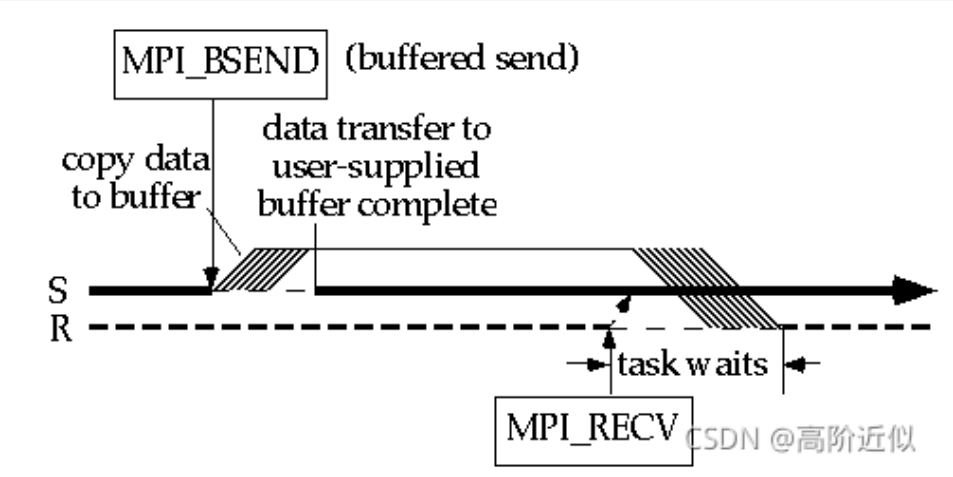
- 只有相应的接收过程已经启动，发送过程才正确返回。
- 因此，同步发送返回后，表示发送缓冲区中的数据已经全部被系统缓冲区缓存，并且已经开始发送。
- 当同步发送返回后，发送缓冲区可以被释放或者重新使用。




```
int MPI_Ssend(const void *buf,
int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

缓冲通信

- 缓冲通信模式的发送不管接收操作是否已经启动都可以执行，但是需要用户程序事先申请一块足够大的缓冲区，通过MPI_Buffer_attch实现，通过MPI_Buffer_detach来回收申请的缓冲区。

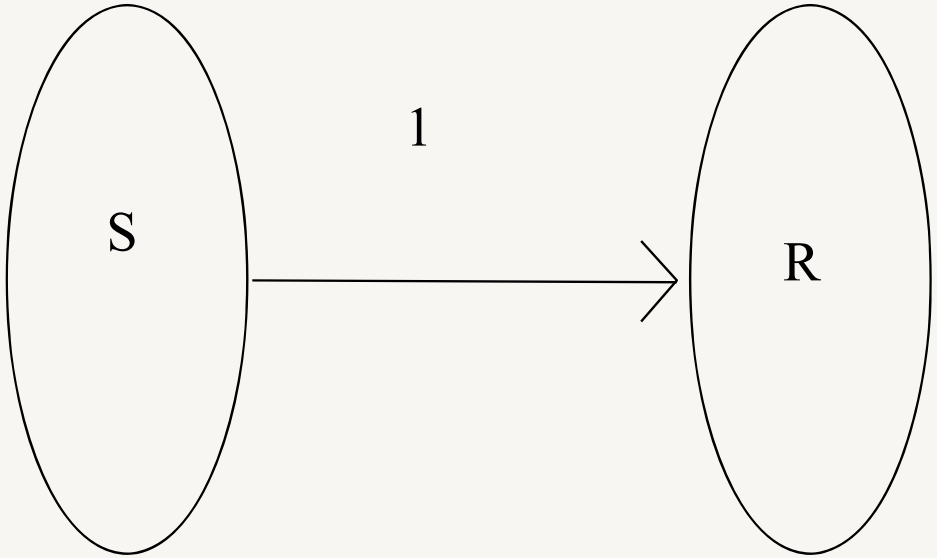



```
int MPI_Bsend(const void *buf, int
count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm)
```

标准通信

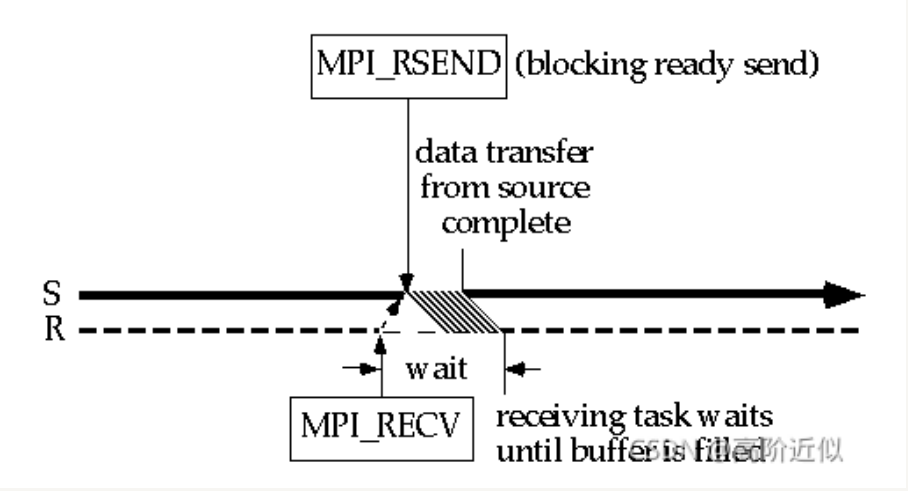
- 是否对发送的数据进行缓冲由MPI的实现来决定，而不是由用户程序来控制。发送可以是同步的或缓冲的，取决于实现。

Standard



就绪通信

- 发送操作只有在接收进程相应的接收操作已经开始才进行发送。
- 当发送操作启动而相应的接收还没有启动，发送操作将出错。
- 就绪通信模式的特殊之处就是接收操作必须先于发送操作启动。




```
int MPI_Rsend(const void *buf, int
count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm)
```

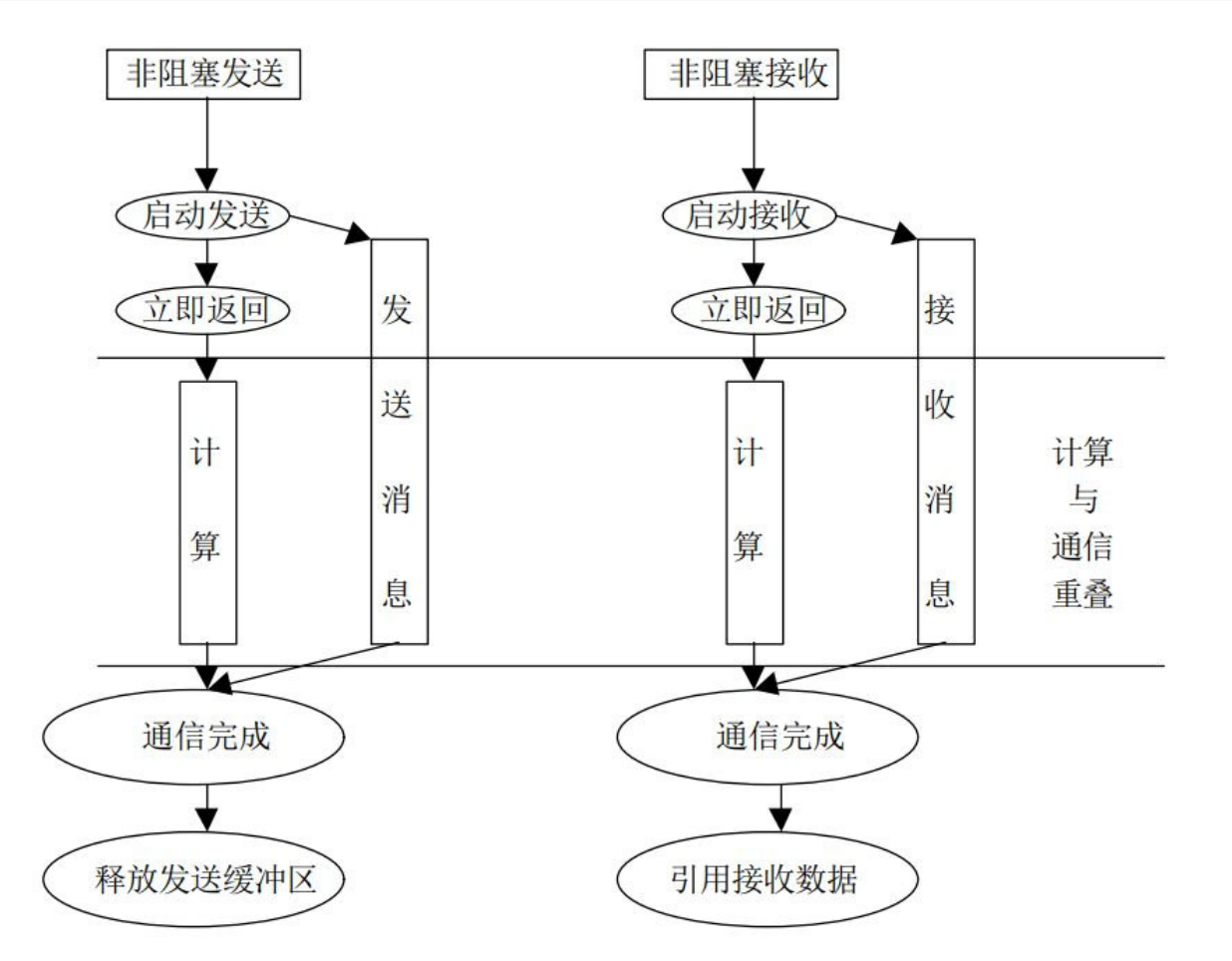
点对点通信模式小结

- 标准通信模式（MPI_Send），它如何发送数据由具体实现决定
- 就绪发送 MPI_Rsend：只有当接收方已经开始接收，发送方才可以发送，否则发送出错
- 同步发送 MPI_Ssend：无论接收方是否启动都可以开始发送，但只有当接收方开始接收，函数才会返回
- 缓冲发送 MPI_Bsend：由用户定义、使用缓冲区，无论接收方操作是否启动，只要缓冲区可用都可以开启发送

- `MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

MPI的两类通信机制

- 前文讨论的均为阻塞的通信模式，当要传输大量数据时，阻塞的通信模式可能拖累执行速度，阻塞通信返回的条件：
  - 通信操作已经完成，也就是消息已经发送或接收；
  - 调用的缓冲区可用。若是发送操作，则该缓冲区可以被其它的操作更新；若是接收操作，该缓冲区的数据已经完整，可以被正确引用。
- 使用非阻塞通信，通信函数可以立即返回。但这种方式不能保证缓冲区等资源可以立即被重新使用


非阻塞发送
启动发送
立即返回
计算
发送消息
非阻塞接收
启动接收
立即返回
计算
接收消息
计算与通信重叠
通信完成
释放发送缓冲区
通信完成
引用接收数据

- 非阻塞的通信函数，其写法是在函数名中加上I字母，即MPI_I[r/b/s]send 例如，标准发送与接收的非阻塞形式为：
  - `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
  - `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`

- 非阻塞的通信函数是立即返回的，在这种异步的情况下我们并不知道发送/接收是否成功，因此需要一个Request对象作为Handler（句柄）来检查数据发送状态，MPI还提供了对非阻塞通信完成的检测，主要的有两种：MPI_Wait函数和MPI_Test函数。

wait函数用来等待直到发送/接收结束（阻塞）：

- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

- test函数用来检测发送/接收是否结束（非阻塞）：
  - `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
  - flag为操作是否完成的标志，True为已完成

示例：非阻塞通信（伪代码）

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
    int x=10;
    MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req1);
    compute();
}
else if (myrank == 1) {
    int x;
    MPI_Irecv(&x, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &req1);
}

MPI_Wait(&req1, &status);
```

以上都是1V1通信，下面介绍 ＮＶＮ （集群通信）

- 群集通信(Collective Communications)是一个进程组中的所有进程都参加的全局通信操作。两种分类方式：功能分类，方向分类
- 功能分类
  - 通信功能：主要完成组内数据的传输
  - 聚集功能：在通信的基础上对给定的数据完成一定的操作
  - 同步功能：实现组内所有进程在执行进度上取得一致
- 方向分类
  - 一对多通信：一个进程向其它所有的进程发送消息，这个负责发送消息的进程叫做Root进程。
  - 多对一通信：一个进程负责从其它所有的进程接收消息，这个接收的进程也叫做Root进程。
  - 多对多通信：每一个进程都向其它所有的进程发送或者接收消息。

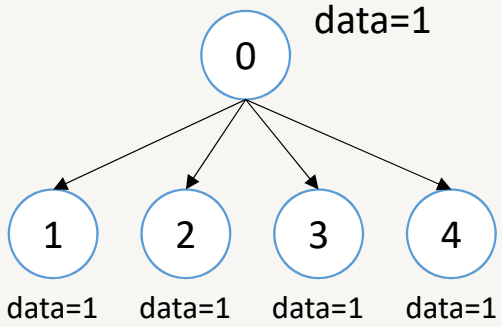
集合通信同步

- 集合通信在进程间引入了同步点（Barrier路障）的概念。这意味着所有的进程在执行代码的时候必须首先都到达一个同步点才能继续执行后面的代码。
- 在该操作调用返回后，可以保证组内所有的进程都已经执行完了调用之前的所有操作，可以开始该调用后继的操作。
- 如果你需要对程序显式地进行同步，也可以使用屏障指令：
  - `MPI_Barrier(MPI_Comm communicator)`

广播 MPI_Bcast

- 广播用来从根节点将一个数据传给communicator中的所有进程

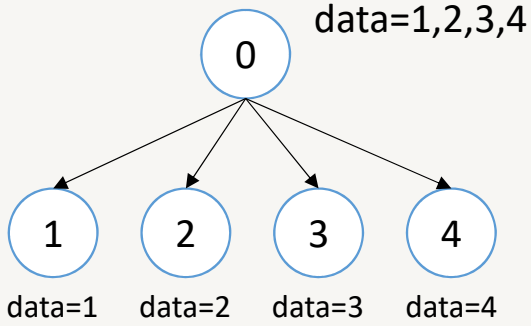
```
MPI_Bcast(
    void* data,      // 根节点：要发送的数据
                     // 其他节点：接收数据的位置
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```



分发 MPI_Scatter

- 分发是从根节点将一组数据中的每一部分分别发送给communicator中的每个进程,每个进程收到的数据可以不同

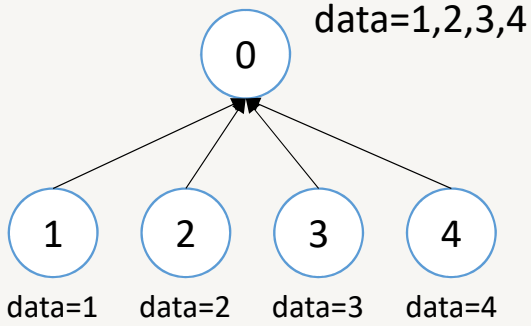
```
MPI_Scatter(
    void* send_data,  // 要分发的全部数据
    int send_count,   // 每个节点要分到几个数据
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```



收集 MPI_Gather

- 与分发相反，收集是将所有进程的数据收集到根进程上

```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```



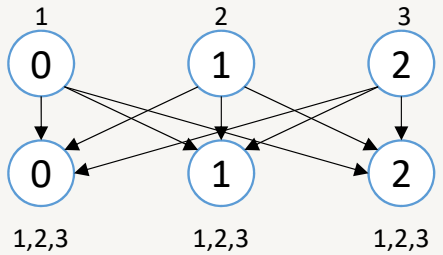
示例：使用集合通信计算平均值（伪代码）

```
if (world_rank == 0) {    生成一些随机数
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);

// more on next page...
```

```
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
}
```

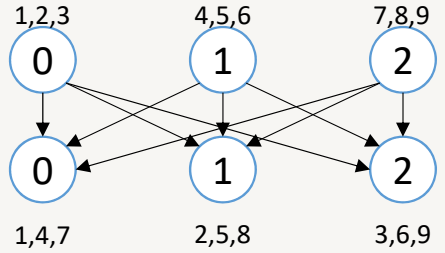
多对多　全局收集VS 全局交换

```
int MPI_Allgather(const void *sendbuf, int  sendcount,
     MPI_Datatype sendtype, void *recvbuf, int recvcount,
     MPI_Datatype recvtype, MPI_Comm comm)
```



全局交换

```
int MPI_Alltoall(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
```

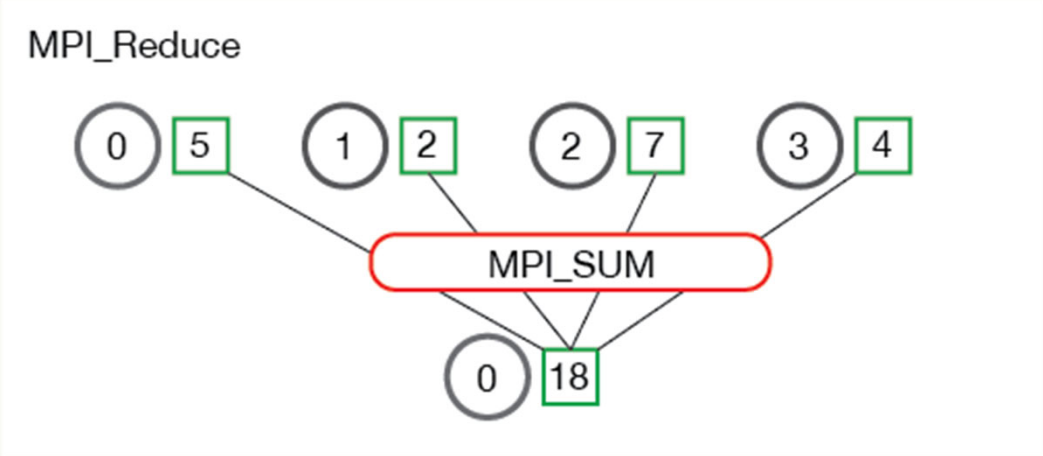


规约 REDUCE

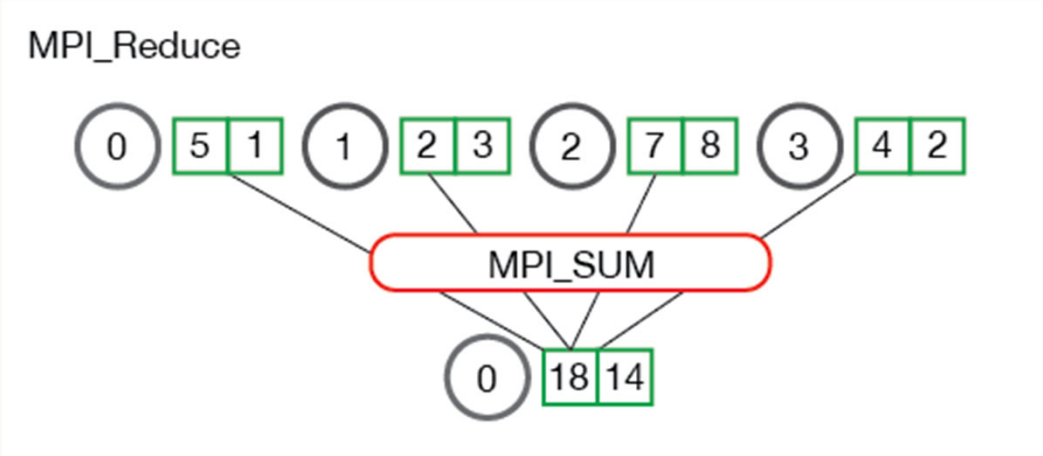
- 规约是函数式编程中的概念，它接受一组数据，并输出更小的一组数字，如 sum([1,2,3,4])=10，MPI提供函数来在各个进程之间进行规约操作

```
int MPI_Reduce(
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,                  // 规约操作
    int root,
    MPI_Comm comm)
```

count=1



count=2



预定义的REDUCE操作

- MPI_MAX - 返回最大元素。
- MPI_MIN - 返回最小元素。
- MPI_SUM - 对元素求和。
- MPI_PROD - 将所有元素相乘。
- MPI_LAND - 对元素执行逻辑与运算。
- MPI_LOR - 对元素执行逻辑或运算。
- MPI_BAND - 对元素的各个位按位与执行。
- MPI_BOR - 对元素的位执行按位或运算。
- MPI_MAXLOC - 返回最大值和所在的进程的秩。
- MPI_MINLOC - 返回最小值和所在的进程的秩。
- （也可以自定义规约操作）

示例：使用REDUCE计算平均值

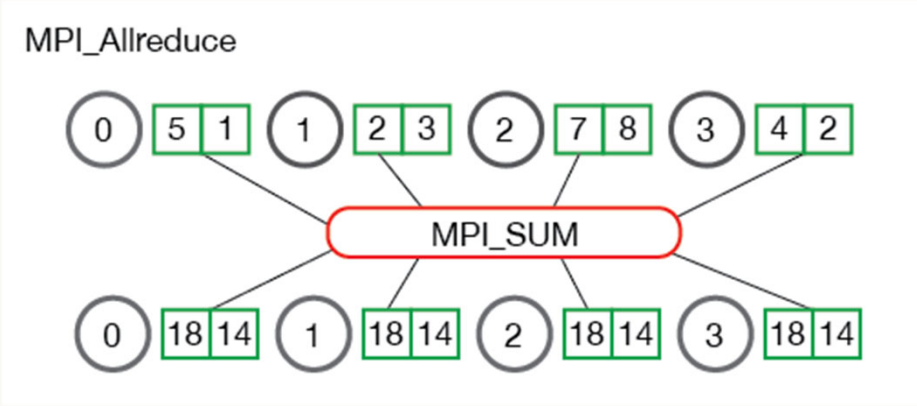
```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);
```

```
// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

ALLREDUCE

- 将Reduce的结果传递给每一个进程

```
int MPI_Allreduce(
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm comm)
```



示例：使用ALLREDUCE计算标准差

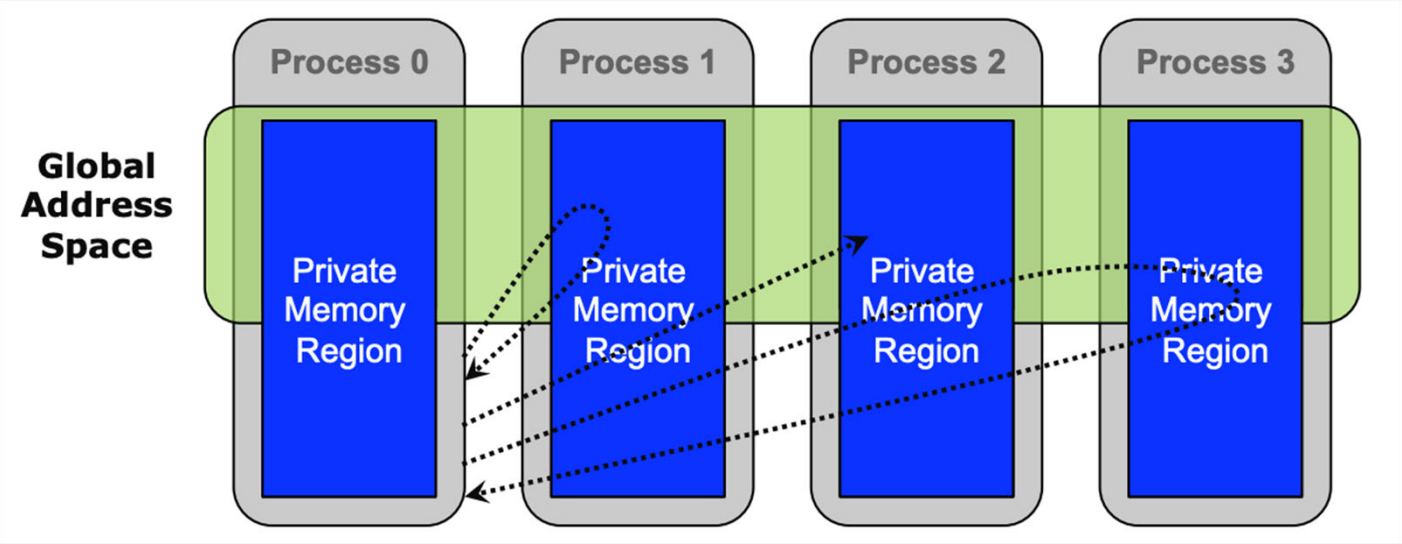
总体标准差：$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}}$

```
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Reduce all of the local sums into the global sum in order to
// calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM,
              MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);
```

```
// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean);
}

// Reduce the global sum of the squared differences to the root
// process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the
// squared differences.
if (world_rank == 0) {
  float stddev = sqrt(global_sq_diff /
                      (num_elements_per_proc * world_size));
  printf("Mean - %f, Standard deviation = %f\n", mean, stddev);
}
```

单边通信（RMA）

- 单边通信又称作远端内存访问（Remote Memory Access，RMA）
- 无论点到点通信或集合通信，都需要发送方和接收方配合，是基于同步的消息传递方式
- 单边通信将数据传递和同步两个操作解耦，每个进程将一部分内存暴露给其他进程，其他进程可以任意访问该内存区域，在不需同步的情况下传递数据


Process 0
Process 1
Process 2
Process 3
Global Address Space
Private Memory Region
Private Memory Region
Private Memory Region
Private Memory Region

- 使用单边通信时，要先创建窗口（MPI_Win），然后在窗口上进行单边通信操作

创建和销毁窗口

```
// 在每个进程上创建一个窗口用于单边通信
MPI_Win_create(void *base,          // 窗口地址
               MPI_Aint size,       // 窗口大小
               int disp_unit,       // 指定窗口中元素大小
               MPI_Info info,       // 其他信息，没有就写 MPI_INFO_NULL
               MPI_Comm comm,       // 通信器
               MPI_Win *win)        // 生成的窗口对象

// 销毁窗口
int MPI_Win_free(MPI_Win *win)
```

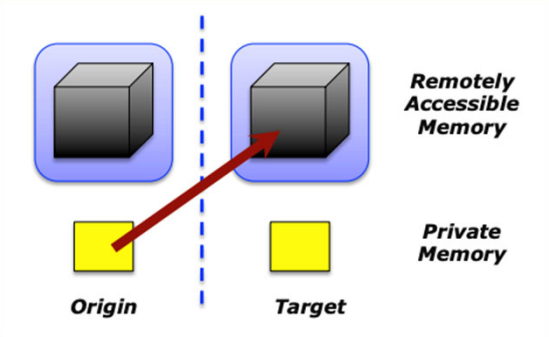
- 创建窗口的方式还包括MPI_Win_allocate、MPI_Win_create_dynamic等

```
int main(int argc, char ** argv)
{
    int *a; MPI_Win win;
    MPI_Init(&argc, &argv);

    /* create private memory */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* use private memory like you normally would */
    a[0] = 1; a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int),
                   MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* Array ‘a’ is now accessible by all processes in
    * MPI_COMM_WORLD */

    MPI_Win_free(&win);
    MPI_Free_mem(a);
    MPI_Finalize(); return 0;
}
```

单边通信操作 PUT

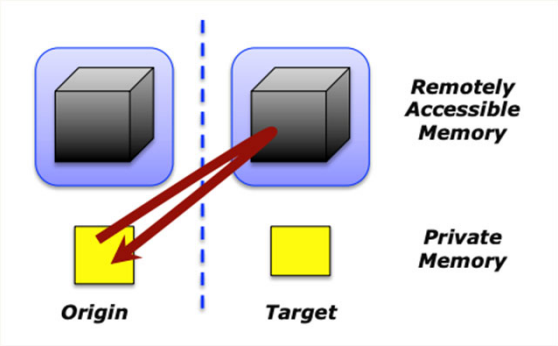
```
int MPI_Put(const void *origin_addr, int origin_count,MPI_Datatype
origin_datatype, int target_rank,MPI_Aint target_disp, int
target_count,MPI_Datatype target_datatype, MPI_Win win)
```

- origin_addr 原始进程缓冲区的初始地址
- origin_count 原始进程缓冲区中的条目数量（非负数整数）
- origin_datatype 原始进程缓冲区中每个条目的数据类型
- target_rank 目标进程的序号(非负整数)
- target_disp 从窗口开始到目标缓冲区的位移(非负整数）
- target_count 目标缓冲区中的条目数量（非负整数整数）
- target_datatype 目标缓冲区中每个条目的数据类型　　win 用于通信的窗口对象



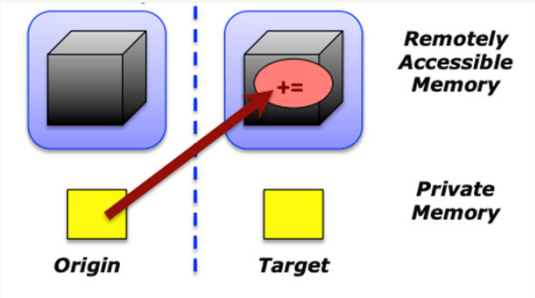
单边通信操作 GET

```
int MPI_Get(void *origin_addr, int origin_count,MPI_Datatype
      origin_datatype, int target_rank,MPI_Aint target_disp, int
      target_count, MPI_Datatype target_datatype,MPI_Win win)
```



单边通信操作 原子加法

```
int MPI_Accumulate(const void *origin_addr, int origin_count,
    MPI_Datatype origin_datatype, int target_rank,
    MPI_Aint target_disp, int target_count,
    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

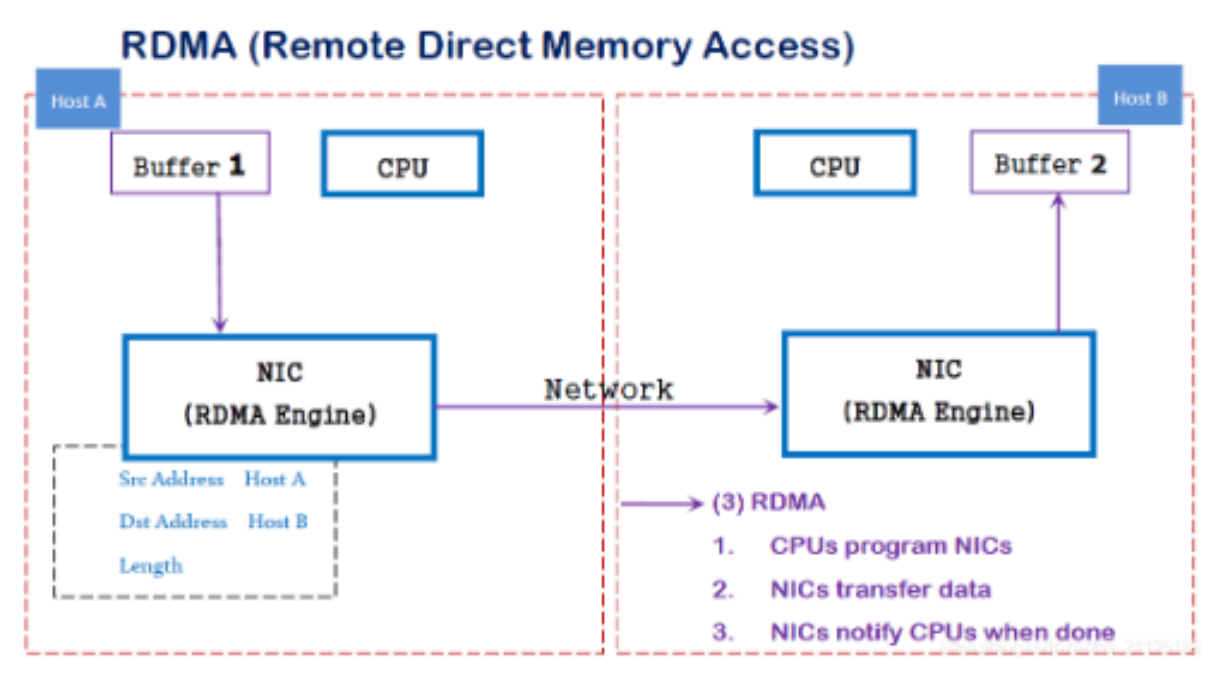


- MPI RMA 和上一章通过共享变量方式实现的进程级并行十分相似
- 必须注意共享变量编程时需要注意的问题，例如单边通信不保证操作的顺序；多个PUT，或是GET、PUT同时发生时结果未定义等等
- 单边通信的接口不提供同步**并不意味**着程序不需要同步，因此请在需要的地方自己加上相应的语句

单边通信 锁

```
// 加锁
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

// 解锁
int MPI_Win_unlock(int rank, MPI_Win win)
```

RDMA（REMOTE DIRECT MEMORY ACCESS）



Infiniband：新一代网络协议，需要支持该技术的网卡和交换机。

RoCE：即RDMA over Ethernet，它也是由IBTA组织指定的。RoCE为以太网提供了RDMA语义，并不需要复杂低效的TCP传输。（IWARP需要）

iWARP也是一个允许在TCP上执行RDMA的网络协议。IB和RoCE中存在的功能在iWARP中不受支持。它支持在标准以太网基础设施（交换机）上使用RDMA

在多个机器上运行OPEN-MPI程序

- 首先配置好ssh，使得各机器之间能无密码互相登录，open-mpi会用到ssh处理机器之间的连接
- 编写一个hosts文件，包含每个节点的host名、其上的slot数量（该节点允许运行的进程数量）

```
node1 slots=4
node2 slots=2
```

- 执行：
  - `mpiexec -hostfile hosts -np 6 ./helloworld`