8 MPI编程

- 大纲
  - 消息传递编程原理
  - 构建块：发送和接收操作
  - MPI：消息传递接口
  - 与计算重叠通信（关键）
  - 组和通讯器
- **什么是 MPI 编程模型?**
  - 每个独立的处理器（processor）都有独立私有的内存，通过互联网络连接起来的分布式内存系统，利用消息传递来编程的模型。
- **消息传递编程原理**
  - 支持消息传递范式的机器的逻辑视图由 p 个进程组成，每个进程都有自己的专有地址空间
  - 限制
    - 每个数据元素必须属于空间的分区之一； 因此，数据必须明确分区和放置
    - 所有交互（只读或读/写）都需要两个进程的合作——拥有数据的进程和想要访问数据的进程
  - 这两个限制虽然繁重，但对程序员来说是非常明确的基础成本
  - 消息传递程序通常使用异步或松散同步范例编写
  - 在异步范式中，所有并发任务都是异步执行的
  - 在松散同步模型中，任务或任务子集同步以执行交互。 在这些交互之间，任务完全异步执行
  - 大多数消息传递程序都是使用**单程序多数据 (SPMD) 模型编写**的
- 构建块：发送和接收操作
  - 原型

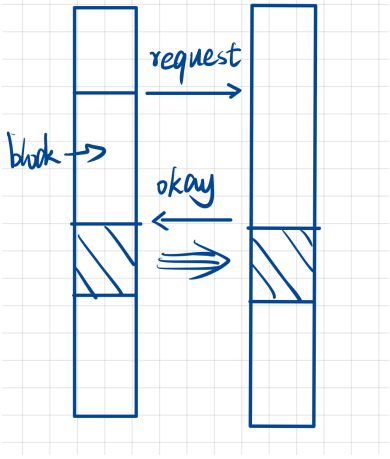
```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

```
   P0                    P1
a = 100;              receive(&a, 1, 0)
send(&a, 1, 1);       printf("%d\n", a);
a = 0;
```

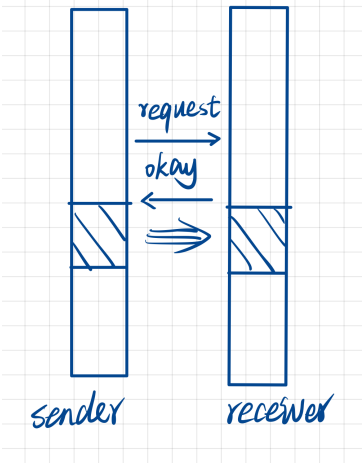
有可能发出0值

  - 这激发了发送和接收协议的设计
  - **发送和接收操作**
    - 没有哪一种？？

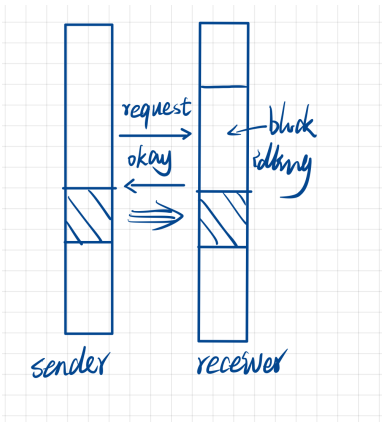
- 无缓冲阻塞
  - 强制发送/接收语义的一种简单方法是发送操作仅在安全时返回
  - 在无缓冲阻塞发送中，直到在接收过程中遇到匹配的接收，操作才会返回
  - 空闲和死锁是非缓冲阻塞发送的主要问题
  - 场景——在通信的时候不做其他事。用于阻塞非缓冲发送/接收操作的握手。很容易看出，在发送方和接收方没有同时到达通信点的情况下，可能会有相当大的空闲开销。
    - 发送方先到达，在发送方有空闲



    - 发送发接收方同时到达，空闲时间最小

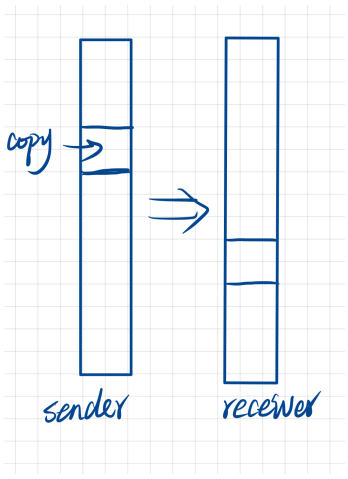


    - 接收方先到达，在接受方处有空闲

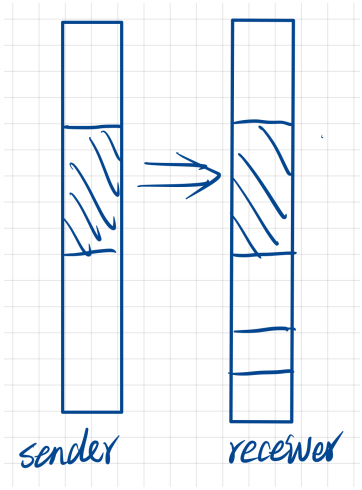


- 缓冲阻塞

- 在缓冲阻塞发送中，发送方简单地将数据复制到指定的缓冲区中，并在复制操作完成后返回。 数据也被复制到接收端的缓冲区
- 缓冲以复制开销为代价减轻空闲
- 当buffer满了，或者上一次发送还没完成，就会先阻塞后面的发送操作。
- 场景
  - 有通信硬件



  - 没有通信硬件——发送方中断接收方并将数据存放在接收方的缓冲区中。接收方使用时再从buffer中取出



- 有界缓冲区大小会对性能产生重大影响

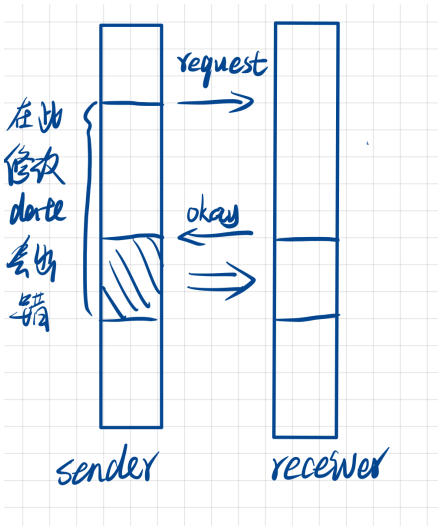
```
          P0                                  P1
for (i = 0; i < 1000; i++){    for (i = 0; i < 1000; i++){
   produce_data(&a);              receive(&a, 1, 0);
   send(&a, 1, 1);                consume_data(&a);
}                                          }
```

- 由于接收操作阻塞，缓冲仍然可能出现死锁。

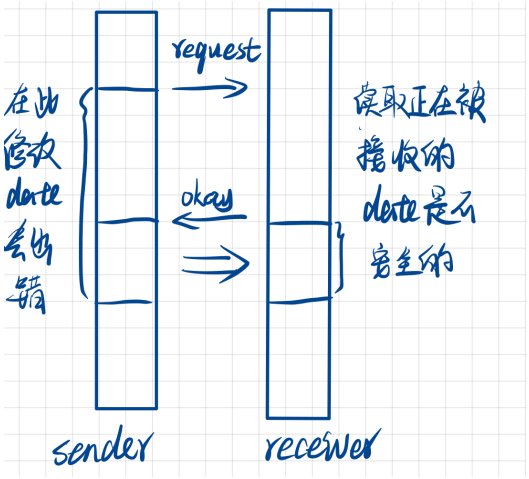
```
P0                              P1
receive(&b, 1, 1);              receive(&a, 1, 0);
send(&a, 1, 1);                 send(&b, 1, 0);
```

- 非阻塞

- 程序员必须确保发送和接收的语义。（异步）
- 此类非阻塞协议在语义安全之前从发送或接收操作返回。
- 非阻塞操作通常伴随着检查状态操作。
- 如果使用得当，这些原语能够将通信开销与有用的计算重叠。
- 消息传递库通常提供阻塞和非阻塞原语。
- 场景
  - 没有通信硬件



  - 有通信硬件



- MPI：消息传递接口
  - 历史
    - 1980 年代后期：供应商拥有独特的库
    - 1989 年：在橡树岭国家实验室开发的并行虚拟机 (PVM)
    - 1992：开始 MPI 标准的工作
    - 1994：MPI 标准 1.0 版
    - 1997：MPI 标准 2.0 版
    - 今天：MPI 是主要的消息传递库标准
  - MPI 定义了一个用于消息传递的标准库，可用于使用 C 或 Fortran 开发可移植的消息传递程序。

- MPI 标准定义了一组核心库例程的语法和语义。
- 几乎所有商业并行计算机上都可以使用 MPI 的供应商实现。
- **仅使用这六个例程就可以编写功能齐全的消息传递程序。**

The minimal set of MPI routines.

MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of calling process.
MPI_Send	Sends a message.
MPI_Recv	Receives a message.

- 启动和终止MPI库
  - 这两个函数的原型是：

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

    - 在调用其他 MPI 例程之前调用。 其目的是初始化 MPI 环境
    - 在计算结束时调用，它执行各种清理任务以终止 MPI 环境
  - 还剥离任何与 MPI 相关的命令行参数
  - 所有 MPI 例程、数据类型和常量都以“MPI_”为前缀。 成功完成的返回码是 MPI_SUCCESS
- 一些基本观点
  - 可以将进程收集到组中
  - 每条消息都在一个上下文中发送，并且必须在同一个上下文中接收（为库提供必要的支持）
  - 一个组和上下文一起形成一个通信子
  - 进程由其在与通信子关联的组中的排名来标识
  - 有一个默认通信器，其组包含所有初始进程，称为 MPI_COMM_WORLD。
- 组和通讯子
  - 相关MPI函数
    - MPI_Group_incl：形成新组作为全局组的子集
    - MPI_Comm_create：为新组创建新的通信子
    - MPI_Comm_rank：确定新通信子中的新序号
    - MPI_Comm_free and MPI_Group_free：完成后，释放新的通信子和组（可选）
  - **什么是通信子**
    - 通信子定义了一个通信域——一组允许相互通信的进程
    - 有关通信域的信息存储在 MPI_Comm 类型的变量中
    - 通信子用作所有消息传输 MPI 例程的参数

    - 一个进程可以属于许多不同（可能重叠）的通信域
    - MPI 定义了一个名为 MPI_COMM_WORLD 的默认通信器，其中包括所有进程
- 查询信息
    - MPI_Comm_size 和 MPI_Comm_rank 函数分别用于确定进程数和调用进程的标签。
    - 这些例程的调用顺序如下：

```
int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

    - 进程的rank是一个整数，范围从零到通信子的大小减一
- MPI消息
    - data : (address, count, datatype)
        - 数据类型是预定义类型或自定义类型
    - message : (data, tag)
        - tag 是一个整数，用于协助接收进程识别消息；

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- 发送和接收消息
    - MPI中发送和接收消息的基本函数分别是MPI_Send和MPI_Recv
    - 例程
        - int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
        - int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- MPI 为所有 C 数据类型提供等效的数据类型。 这样做是出于便携性的原因
- 数据类型 MPI_BYTE 对应于一个字节（8 位）， MPI_PACKED 对应于通过打包非连续数据创建的数据项的集合
- 消息标签的取值范围从零到 MPI 定义的常量 MPI_TAG_UB（Tag的最大值）
- **MPI发送模式**
  - MPI_Send（标准模式）：在您可以使用发送缓冲区之前不会返回（非本地）《有缓冲阻塞》
  - MPI_Bsend（缓冲模式）
    - 立即返回，您可以使用发送缓冲区
    - 相关：MPI_buffer_attach()、MPI_buffer_detach()
  - MPI_Ssend（同步模式）
    - 在匹配的接收发布之前不会返回
    - 发送+同步通信语义
  - MPI_Rsend（就绪模式）
    - 仅当匹配接收已经准备好时才可以使用
    - 发件人向系统提供额外的信息，可以节省一些开销
  - MPI_Isend（非阻塞标准模式）
    - 非阻塞发送，但您不能立即重用发送缓冲区
    - 相关：MPI_Wait()、MPI_Test()
  - MPI_Ibsend
  - MPI_Issend
  - MPI_Irsend
- 其他
  - MPI 允许为源和标签指定通配符参数。
  - 如果 source 设置为 MPI_ANY_SOURCE，则通信域的任何进程都可以是消息的来源。
  - 如果标记设置为 MPI_ANY_TAG，则接受带有任何标记的消息。
  - 在接收端，消息的长度必须等于或小于指定的长度字段
  - 在接收端，status 变量可用于获取有关 MPI_Recv 操作的信息
  - 对应的数据结构包含：typedef struct MPI_Status {int MPI_SOURCE; int MPI_TAG; int MPI_ERROR; };
  - MPI_Get_count 函数返回接收到的数据项的精确计数：int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
- **避免死锁**
  - 用非阻塞对应语句替换发送或接收操作可修复此死锁。

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If *MPI_Send* is blocking, there is a deadlock.

- 解决不安全问题
  - 改变发送接收顺序
  - 使用MPI_Sendrecv函数：支持发送和接收缓冲
    - int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
    - 参数包括发送和接收函数的参数。 如果我们希望对发送和接收使用相同的缓冲区，我们可以使用：
      - int **MPI_Sendrecv_replace**(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
  - 提供自己的空间作为发送缓冲区；使用非缓冲

Supply own space as buffer for send

P0	P1
Bsend(1)	Bsend(0)
Recv(1)	Recv(0)

User non-blocking operations

P0	P1
Isend(1)	Irecv(0)
Irecv(1)	Isend(0)
Waitall	Waitall

- 与计算重叠通信（关键）
  - 为了将通信与计算重叠，MPI 提供了一对用于执行非阻塞发送和接收操作的函数。
    - int MPI_Isend(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)
    - 这些操作在操作完成之前返回。 函数 MPI_Test 测试其请求标识的非阻塞发送或接收操作是否完成
        - int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
    - MPI_Wait 等待操作完成。
        - int MPI_Wait(MPI_Request *request, MPI_Status *status)
- 考虑以下代码，其中进程 i 向进程 i + 1 发送消息（以进程数为模）并从进程 i - 1（以进程数为模）接收消息。

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD);
...
```

Once again, we have a deadlock if *MPI_Send* is blocking.

- 避免死锁：奇偶进程分情况

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD);
}
else {
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
MPI_COMM_WORLD);
}
```

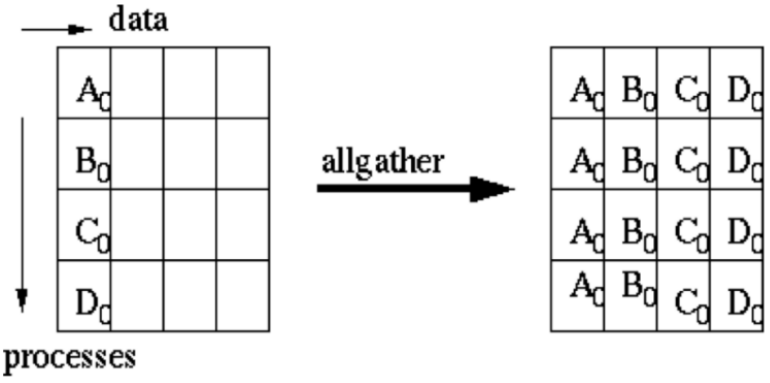
- **群组通信和计算**
    - 在 MPI 中执行屏障同步操作：int MPI_Barrier(MPI_Comm comm)
    - 一对多的广播操作是：int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)
    - 多对一规约操作：int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int target，MPI_Comm comm)

- MPI_MAXLOC 返回（v， l），v是所有进程中值最小的，l是对应进程号，如果有多个进程v一样，则返回l最小的那个
- MPI_MINLOC返回最小值以及其对应进程号

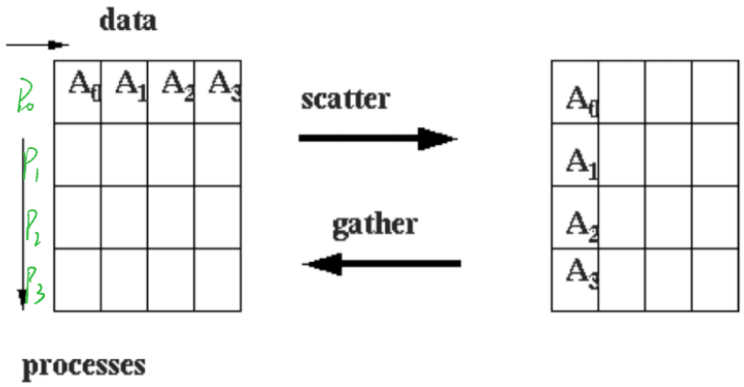
MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations.

MPI Datatype	C Datatype
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_INT	long and int
MPI_LONG_DOUBLE_INT	long double and int
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int

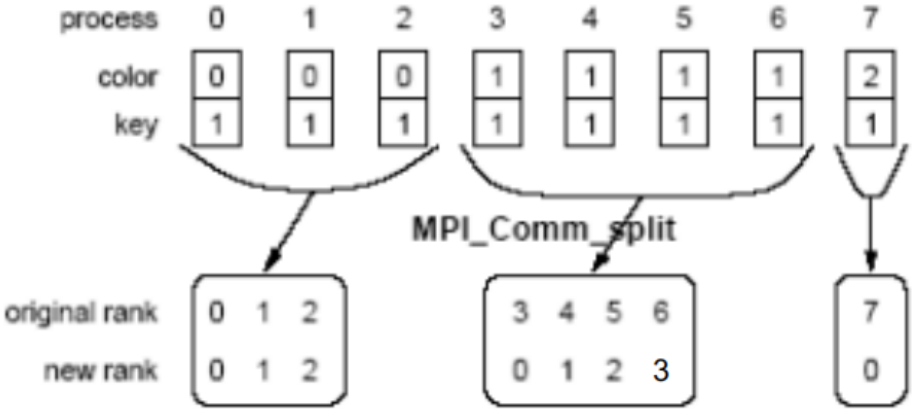
- 如果规约的值所有进程都需要：int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- 计算前缀和：int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- 聚合操作：int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int target, MPI_Comm comm)
- 聚合到所有进程：int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype MPI_Comm comm)



- 相对应的分发函数：int **MPI_Scatter**(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)



- 多对多的个性化通信操作：int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)
- 使用这组核心集合操作，可以大大简化许多程序。
- 一个组里可以由多个通信子，一个进程也可以属于多个组
- 在许多并行算法中，通信操作需要限制在某些进程子集中：See MPI_Group_*() and MPI_Comm_*() for more APIs
- MPI 提供了将属于通信子的进程组划分为子组的机制，每个子组对应于不同的通信子：int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
  - 此操作按颜色对进程进行分组，并对键上的结果组进行排序



以上内容整理于 幕布文档